Solving Difficult Game Positions

Solving Difficult Game Positions

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. mr. G.P.M.F. Mols,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op woensdag 15 december 2010 om 14.00 uur

door

Jahn-Takeshi Saito

Promotor: Prof. dr. G. Weiss
Copromotor: Dr. M.H.M. Winands
Dr. ir. J.W.H.M. Uiterwijk

Leden van de beoordelingscommissie:
Prof. dr. ir. R.L.M. Peeters (voorzitter)
Prof. dr. T. Cazenave (Université Paris-Dauphine)
Prof. dr. M. Gyssens (Universiteit Hasselt / Universiteit Maastricht)
Prof. dr. ir. J.C. Scholtes
Prof. dr. C. Witteveen (Technische Universiteit Delft)

Netherlands Organisation for Scientific Research

The research has been funded by the Netherlands Organisation for Scientific Research (NWO), in the framework of the project Go for Go, grant number 612.066.409.

Dissertation Series No. 2010-49
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN: 978-90-8559-164-1

Preface

After receiving my Master's degree in Computational Linguistics and Artificial Intelligence in Osnabrück, I faced the pleasant choice between becoming a Ph.D. student in Osnabrück or Maastricht. The first option would have led me further into the field of Computational Linguistics. However, I decided for the second option, crossed the border to the Netherlands, and continued my work on Computer Go that had been the topic of my Master's thesis. It was a choice that I have never regretted. While I had sensed the gravity of my choice, I could not possibly have foreseen all actual consequences. One consequence was that I entered the research of solving difficult game positions. Another consequence was the production of this volume. Yet another consequence was my encounter with many persons whom I would like to acknowledge here.

Of course, this thesis would not have been possible without the help of a large group of people. I would first and foremost like to thank my daily advisors, Mark Winands and Jos Uiterwijk, and my supervisor Gerhard Weiss. I am grateful to Mark for his firm guidance on scientific questions and for motivating me whenever necessary. To Jos I owe gratitude for his patient devotion to detail that so often reminded me to work with greater care and less haste. I am indebted to Gerhard for taking over the responsibility of supervising me and for giving me valuable advice from a point of view wider than that of games and search.

Furthermore, I would like to thank all people with whom I collaborated over the years. I am grateful to Jaap van den Herik of whose inspiring drive for perfection, professional attitude, and superb editing skills I was allowed to profit in particular during my first three years in Maastricht and still continue to profit today. I owe particular gratitude to two people who might be caught by surprise: Bruno Bouzy and Helmar Gust. Their support during my Master's studies led my way to the Netherlands. Similarly, I am grateful to Erik van der Werf who initiated the project that I later worked on and who additionally gave me valuable support on many occasions. I am indebted to Dr. Yngvi Björnsson for his collaboration and Prof. Hiroyuki Iida for comments on proof-number search.

I will never forget my roommates in my years in the "women's prison" of Minderbroedersberg 6a: Laurens van der Maaten, Maarten Schadd, Guillaume Chaslot, Andra Waagmeester, Pim Nijssen, and David Lupien St-Pierre. Thank you for easing the burden of labor with occasional moments of recreation, and for your advice on matters scientific and beyond. I would like to thank Sander Spek and Laurens van der Maaten who shared with me not only an appreciation for decent coffee but

also valuable experience that influenced this thesis. I am indebted for inspiring discussions to my colleagues past and present: Sander Bakkes, Niek Bergboer, Guido de Croon, Jeroen Donkers, Steven de Jong, Michael Kaisers, Joyca Lacroix, Nyree Lemmens, Georgi Nalbantov, Marc Ponsen, Evgueni Smirnov, Pieter Spronck, Ben Torben-Nielsen, Philippe Uyttendaele, Stijn Vanderlooy and Jean Derks.

The one special person I cannot possibly thank enough for enduring my stress during producing this thesis also happens to be the person I love, Silja. Danke für Deine unverzichtbare Hilfe.

This work has been supported by FHS's system administrators Peter Geurtz and Ton Derix and secretaries Joke Hellemons, Tons van den Bosch, Karin Braeken and Marijke Verheij.

Countless more people should be thanked here, like the people from BiGCaT Bioinformatics Department of Maastricht University. I would just like to thank all of my friends, family, and colleagues for the support they have given me during the work on this thesis.

Jahn-Takeshi Saito
Maastricht, August 2010

Acknowledgments

The research has been carried out under the auspices of the Dutch Research School for Information and Knowledge Systems (SIKS). I gratefully acknowledge the financial support by the Netherlands Organisation for Scientific Research (NWO).

Table of Contents

List of Abbreviations

dn	Disproof number
df-pn	Depth-first Proof-Number Search
LOA	Lines of Action
MCE	Monte-Carlo Evaluation
MC-PNS	Monte-Carlo Proof-Number Search
MCTS	Monte-Carlo Tree Search
pn	Proof number
ParaPDS	Parallel Proof-and-Disproof-Number Search
PDS	Proof-and-Disproof-Number Search
PDS–PN	Two-level Proof-and-Disproof-Number Search
PNS	Proof-Number Search
PN^*	Iterative-deepening depth-first Proof-Number Search
PN^2	Two-level Proof-Number Search
PN_1	First-level search of PN^2
PN_2	Second-level search of PN^2
RP–PNS	Randomized-Parallel Proof-Number Search
RP–PN^2	Two-level Randomized-Parallel Proof-Number Search

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is in the field of games and Artificial Intelligence (AI). It has long been claimed, that games are an indicator of or even a precondition to culture (Huizinga, 1955). Humans participate in games not only to satisfy their desire for entertainment but also because they seek an intellectual challenge. One obvious challenge in games is defeating the opponent(s). The AI equivalent to this challenge is the designing of strong game-playing programs. Another challenge in games is finding the result of a game position, for instance whether a Chess position is a win or a loss. The AI equivalent to this challenge is the designing of algorithms that solve positions. While game-playing programs have become much stronger over the years, solving games still remains a difficult task today and has therefore been receiving attention continuously. Throughout the thesis, we are interested in this difficult task of solving game positions. The following chapters present algorithms that are based on recent research in the field of search algorithms. In particular, the thesis extends recent developments in Monte-Carlo search for the task of solving. Moreover, open questions of Proof-Number Search for solving are addressed. To that end, the here described research contributes and tests new search algorithms.

This chapter introduces the basic notion of *solving* and gives the problem statement and four research questions guiding our research. Section 1.1 summarizes the relevance of games for AI. Section 1.2 discusses the concept of solving games and game positions. Section 1.3 formulates the problem statement and gives four research questions. Section 1.4 completes the introduction by presenting the structure of this thesis and the relationship between chapters and research questions.

1.1 Games and AI

Since the early days of AI, games have served as a testing ground for measuring the progress of the field. Two founding fathers of the discipline, Shannon (1950) and Turing (1953), were among the first to describe Chess-playing programs. Ever since, progress has been steady. Programs have improved and nowadays play a broad range of games stronger than humans. An event illustrating the progress of playing strength is the defeat of the human World Champion Gary Kasparov by the Chess

program DEEP BLUE. On May 11, 1997 DEEP BLUE won a six-game match by two wins to one with three draws. This outcome was recognized as a historic victory for AI (DeCoste, 1998).

The number of games in which humans can easily defeat the best game-playing program is declining. An example of this trend is computer Go. The game of Go had been one of the remaining challenges in which the human player had still been clearly superior to the programs. However, the recent past has witnessed the rise of the Monte-Carlo Tree Search framework (Coulom, 2007a; Kocsis and Szepesvári, 2006). It has helped closing the competitive gap between man and machine (Chaslot, 2008).

Allis, Van den Herik, and Herschberg (1991) suggest categorizing games based on the strength of available programs. The five suggested categories are: (1) amateur-level games, (2) grand-master-level games, (3) champion-level games, (4) over- champion games, and (5) solved games. The categories (1) to (4) relate the strength of the programs to the strength of human players. For instance, an amateur-level game is a game in which the strongest available program plays at human amateur level. Category (5), solved games, is not defined in relation to human strength. A game is solved if its game-theoretic value has been calculated (e.g., win for the first player) assuming both players play optimally.

The advance of AI research has resulted in a growing number of over-champion games including the above-mentioned Chess. Although many games have been solved, solving games has remained challenging (Van den Herik, Uiterwijk, and Van Rijswijck, 2002). For some games researchers therefore solved smaller versions, e.g., 5×5 Go (Van der Werf, Van den Herik, and Uiterwijk, 2003), 6×6 Lines of Action (LOA) (Winands, 2008), and 8×8 Hex (Henderson, Arneson, and Hayward, 2009). Even solving mate positions in games such as Chess or Shogi still poses a hard problem today. This thesis deals with search techniques for addressing this problem.

1.2 Solving Games and Game Positions

Solving is finding the game-theoretic value of a given game position. Some games, such as Hex (Nash, 1952) and Nim (Bouton, 1902), can be solved by a theoretic proof. Many other games cannot be solved in this way. Solving the latter games requires search. Search for solving is described in detail in Chapter 2. We call an algorithm describing how to find a game-theoretic value of a game position a *solving algorithm*. A *solver* is an implementation of a solving algorithm.

Different degrees of solving have been distinguished by Paul Colley and Donald Michie (Allis, 1994). The degrees form a hierarchy in which the minimal solution is called *ultra weak* and the maximal solution is called *strong*. The hierarchy of solving games is as follows (Allis, 1994).

Ultra-weakly solved. For the initial position(s), the game-theoretic value has been determined. Crucially, the proof is not necessarily constructive, i.e., it does not have to provide a strategy of optimal play. A famous example of an ultra-weakly solved game is Hex (Nash, 1952).

Weakly solved. For the initial position(s), a strategy has been determined to obtain at least the game-theoretic value of the game for both players under reasonable resources. Examples of weakly solved games include Qubic (Patashnik, 1980), Go-Moku (Allis, Huntjes, and Van den Herik, 1996), Nine-Men's Morris (Gasser, 1996), various k-in-a-row games (Uiterwijk and Van den Herik, 2000), Domineering (Breuker, Uiterwijk, and Van den Herik, 2000), Renju (Wágner and Virág, 2001), 5×5 Go (Van der Werf *et al.*, 2003), Checkers (Schaeffer *et al.*, 2007), and Fanorona (Schadd *et al.*, 2008).

Strongly solved. For all legal positions, a strategy has been determined to obtain the game-theoretic value of the position, for both players, under reasonable resources. Examples of recently strongly solved games include Kalah (Irving, Donkers, and Uiterwijk, 2000), Awari (Romein and Bal, 2003), and Connect Four (Tromp, 2008).

When solving an arbitrary game position (as opposed to only regarding initial positions), the three degrees of solving can be adopted trivially as follows. A non-initial position can be thought of as the initial position of a derived game. All rules of the derived game except those determining the start position are the same as in the original game. We call a position weakly solved if the derived game is solved weakly. Under this notion, a solving algorithm is a method for solving games or game positions weakly.

Solvers for positions of two-player games with perfect information such as Go (Wolf, 1994; Kishimoto and Müller, 2005a; Wolf and Shen, 2007), and Shogi (Takizawa and Grimbergen, 2000; Nagai, 2002) have been available for a long time. The earliest solvers were mate solvers for Chess. Two of the earliest solvers deserve mentioning here. In 1912, the Spanish engineer Torres y Quevedo presented an electromechanic automaton called El Ajedrecista ("the Chess player") which was able to solve one kind of King-Rook-King endgames (Bell, 1978) with the constraint of occasional mechanical failure. About 40 years later, in 1951, Prinz programmed the Manchester-Mark I in order to solve mate-in-two endgame problems.

Already in 1985, Grottling (1985) compared the performance of several mate solvers on Chess problems. As pointed out by Lindner (1985), the *Chess problem* is characterized by the following three attributes: (1) it has a solution (normally a checkmate), (2) the solution is reachable in a fixed number of moves, and (3) the first move is required to be unique. A *mate-in-n problem* is furthermore constraint by the fact that no refutation is possible which prolongs the game beyond n moves.

Besides two-player games with *perfect information*, several researchers provided solving algorithms for two-player games with *imperfect information*. Sakuta and Iida (2000) solved positions for the imperfect-information game of Screen Shogi. Similarly, Ferguson (1992) was able to solve the KBNK ending in the game of Kriegspiel. Bolognesi and Ciancarini (2003) successfully solved more demanding endgames of Kriegspiel by considering extensive search spaces given by meta-positions, i.e., collections of possible positions deducible from observations with limited information.

The solving algorithms we are studying in this thesis are search algorithms. They can be distinguished into *forward search* and *backward search* (Schaeffer *et al.*, 2007). Forward search expands a search tree or search graph from a given position and

gradually searches deeper until all subtrees are solved. Backward search starts by considering the game-theoretic values for all terminal positions. Then, the values for all positions directly preceding terminal positions are calculated. Next, the values for the preceding positions are calculated and so on. This information is stored as an endgame database. In this way, the search tree is built bottom up until the game-theoretic value of the initial position has been found. The most common backward-search technique for creating an endgame database is retrograde analysis (Ströhlein, 1970). While some games like Go are not suitable for endgame databases due to the overwhelming number of terminal positions, other games such as Chess (Thompson, 1986; Nalimov, Haworth, and Heinz, 2000) are more suitable because their state space is converging (Allis, 1994). For such games, including Checkers (Schaeffer *et al.*, 2007) and Fanorona (Schadd *et al.*, 2008), it is possible to combine forward and backward search by expanding a search tree up to a certain depth at which an endgame database provides the game-theoretic value.

Forward search commonly applies heuristic knowledge to solve a game, but it is also possible to combine search and analytical proofs. An analytical proof provides perfect knowledge which the search algorithm can exploit for solving. In this way, Bullock (2002) solved Domineering on large board sizes up to 10×10 by supplying perfect knowledge in an evaluation function. Similarly, Allis (1988) provided a rule-based approach to solving Connect Four. Hayward, Björnsson, and Johanson (2005) solved 7×7 Hex with perfect knowledge of virtual connections. More recently, Wu and Huang (2006) and Chiang, Wu, and Lin (2010) solved k-in-a-row games by combining analytical proofs with search algorithms.

1.3 Problem Statement and Research Questions

The previous section reflected on solving games and game positions by computer programs. This is exactly the topic of the thesis. The game positions studied in this thesis generalize from the narrow definition of mate-in-n for Chess problems. Throughout this thesis and if not stated otherwise explicitly, the games considered are deterministic turn-taking adversarial games with perfect information. We consider any position of a game for solving. In most cases, we do not know a game position's outcome (win or loss) or the exact depth of the solution. Moreover, the solution is not required to have a unique best move.

While much effort has been put into $\alpha\beta$ search (Knuth and Moore, 1975), this thesis investigates solving games with Monte-Carlo search techniques (Abramson, 1990; Brügmann, 1993; Bouzy and Helmstetter, 2003; Coulom, 2007a; Kocsis and Szepesvári, 2006) and Proof-Number Search (Allis, Van der Meulen, and Van den Herik, 1994). In particular, we test new forward-search algorithms on the games of Go, LOA, and Rolit. The following problem statement (**PS**) guides our research.

PS *How can we improve forward search for solving game positions?*

We give four research questions to address the problem statement. The context of the four research questions is the current research in game-tree search in general and two current aspects in particular, namely: (i) the recent progress in Monte-Carlo

search techniques for game-playing programs, and (ii) open questions in applying Proof-Number Search variants to solving difficult game positions. The four research questions deal with (1) Monte-Carlo evaluation, (2) Monte-Carlo Tree Search, (3) parallelized search, and (4) search for multi-player games.

RQ 1 *How can we use Monte-Carlo evaluation to improve Proof-Number Search for solving game positions?*

The recent past in the game-tree search domain has witnessed graspable advances by applying Monte-Carlo evaluation (Abramson, 1990; Brügmann, 1993; Bouzy and Helmstetter, 2003). **RQ 1** inquires in how far Proof-Number Search can exploit this development.

RQ 2 *How can the Monte-Carlo Tree Search framework contribute to solving game positions?*

Based on Monte-Carlo evaluation, Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007a) has seen many successes in the past few years. **RQ 2** asks how these successes can be exploited for solving. In particular, the question calls for investigating how game-theoretic values can be propagated in the search tree of MCTS.

RQ 3 *How can Proof-Number Search be parallelized?*

So far, only a small amount of research treated the issue of parallelizing Proof-Number Search (Kishimoto and Kotani, 1999; Kaneko, 2010). To answer **RQ 3**, we propose a way to parallelize Proof-Number Search by randomization.

RQ 4 *How can Proof-Number Search be applied to multi-player games?*

Most solving algorithms are designed to solve two-player games. **RQ 4** addresses a generalization of solving beyond the two-player limit (Luckhardt and Irani, 1986; Korf, 1991; Sturtevant and Korf, 2000). The thesis proposes to extend Proof-Number Search for multiple players. In doing so, we address the problem of defining what solving means in the context of games with more than two players.

1.4 Outline of the Thesis

This thesis is organized in seven chapters. Chapter 1 gives a general introduction to the topic of this thesis, solving games, and presents the problem statement and four research questions that guide the research. Chapter 2 establishes the terminology used throughout the remainder of the thesis and surveys existing search algorithms. In particular, it explains Monte-Carlo search techniques and variants of Proof-Number Search in detail.

Each of Chapters 3, 4, 5, and 6 answers one of the four research questions and thereby addresses the problem statement of this thesis. Chapter 3 describes how Monte-Carlo evaluation can be combined with Proof-Number Search to form an

algorithm called MC-PNS. We test MC-PNS on Go problems. The chapter provides an answer to **RQ 1**. Chapter 4 answers **RQ 2** by showing how Monte-Carlo Tree Search can be transformed into a solving algorithm called MCTS Solver. The solver is tested on LOA positions. Chapter 5 addresses **RQ 3** and describes a parallelization of PNS which is based on randomization. The resulting parallel algorithm is called RP–PNS. RP–PNS and its two-level variant RP–PN^2 are tested on LOA positions. Chapter 6 focuses on solving multi-player games and therefore supplies an answer to **RQ 4**. We propose an adaptation of PNS for multi-player games. The resulting algorithm, Paranoid Proof-Number Search, is tested on the game of Rolit, a multi-player version of Othello (Reversi).

Chapter 7 concludes the thesis by reviewing the results of the preceding chapters and relating them to the problem statement and the four research questions, and gives recommendations for future research. The appendix contains the rules for the games of Go and LOA.

Chapter 2

Basics of Game-Tree Search for Solvers

This chapter is partially based on the following two publications.[1]

1. J-T. Saito, G.M.J.B. Chaslot, J.W.H.M. Uiterwijk, and H.J. van den Herik. Monte-Carlo Proof-Number Search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Computers and Games - 5th International Conference (CG '06)*, volume 4630 of *Lecture Notes in Computer Science*, pp. 50–61. Springer, Berlin, Germany, 2007.
2. J-T. Saito, M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. Grouping Nodes for Monte-Carlo Tree Search. In M.M. Dastani and E. de Jong, editors, *Proceedings of the 19th Belgian-Dutch Conference on Artificial Intelligence (BNAIC '07)*, pp. 276–283. Utrecht University Press, Utrecht University, Utrecht, The Netherlands, 2007.

The solving algorithms presented in the later chapters require a basic understanding of standard search techniques. The aim of this chapter is to provide an overview of search techniques related and relevant to solving. To this end, we introduce basic concepts and give notational conventions applied throughout the thesis. We then devote particular detail to two topics: (1) proof-number algorithms (Allis, 1994; Van den Herik and Winands, 2008), and (2) Monte-Carlo techniques (Abramson, 1990; Kocsis and Szepesvári, 2006; Coulom, 2007a). Proof-number algorithms are stressed because they are well-studied standard techniques for solving. The reason for paying particular attention to Monte-Carlo techniques is that they can also be used for solving positions (cf. Zhang and Chen, 2008; Winands, Björnsson, and Saito, 2008).

This chapter is organized as follows. Section 2.1 introduces the basic concepts of game-tree search as well as the notational conventions required for describing

[1] The author is grateful to Springer-Verlag and Utrecht University Press for the permission to reuse relevant parts of the articles.

solving techniques. Section 2.2 introduces the family of proof-number algorithms. Section 2.3 gives an overview of Monte-Carlo techniques relevant for Monte-Carlo-based solvers which are described in detail in Chapter 4. Section 2.4 concludes the chapter by relating the search techniques introduced in this chapter to the techniques presented in the following chapters.

2.1 Game-Tree Search

This section describes game-tree search. Subsection 2.1.1 introduces basic concepts of game-tree search. Subsection 2.1.2 describes the search tree and three common search methods. Subsection 2.1.3 explains transposition tables and the Graph-History-Interaction problem; the latter is relevant to Section 2.2.

2.1.1 Game Tree

A *game tree* is an explicit representation of the state space for turn-based games. The *nodes* of the tree represent positions, the *edges* represent moves. The node representing the initial position is the *root*. A node is *terminal* if its corresponding position is an end position according to the rules of the game. Terminal nodes have a game-theoretic value according to the rules of the games, e.g., win or loss. A node's immediate successor is called a *child*. Analogously, a node's immediate predecessor is called its *parent*. Any node may have zero or more children. The root has no parent. All other nodes have exactly one parent. An *internal node* is a node that has a child. A node is *fully expanded* by generating all of its children. A node without children is called a *leaf*. A *successor* of a node P is recursively defined as either (1) a child of P, or (2) a child of a successor of P. Analogously, a node P is called predecessor of another node S if S is a successor of P. A *path* to L is the set of all predecessors of L.

A game tree is generated by first fully expanding the root and then repeatedly expanding all non-terminal nodes until all leaves are terminal. The *game-theoretic value* of the game is the value of the initial position under optimal play by both players. The game-theoretic value can be found by observing the game tree.

The depth of the root is zero. For non-root nodes the depth is the one-increment of the depth of the parent.

2.1.2 Search Tree

Often a game tree is too large to be fully expanded in practice. Instead, a *search tree* can be considered. The search tree is part of the game tree. It can be used for finding a solution for two purposes. The first purpose is finding the game-theoretic value of the initial position. The second purpose is finding the best move given constraints on time or space.

A search tree is developed starting from the root according to a search method. Search methods are often guided by an evaluation function. Evaluation functions estimate the game-theoretic value of a game position (Pearl, 1984) by assigning a heuristic value to a game position.

We distinguish three kinds of search methods: (1) breadth-first search, (2) depth-first search, and (3) best-first search. We shall briefly describe each of them.

Breadth-first search first expands the root node. Next, it expands all non-terminal leaves at depth 1. Then all non-terminal leaves at the next depth are expanded. This process is repeated incrementally until a solution is found. Thus, siblings are expanded before children. Breadth-first search always finds a solution with the shortest path, if there is any, but typically requires too much memory in practice.

Depth-first search first expands the root. Next, one of its children is selected for expansion. If the selected node is not terminal, the node is expanded. Then again one of the newborn children is selected for expansion and so forth. If a child is a terminal, one of its siblings is chosen for further investigation. If all children have been investigated, one of the parent's siblings is chosen and so forth. In this manner, children are expanded before siblings. Depth-first search requires relatively little memory. In practice it is able to find solutions quickly when an evaluation function is used. A disadvantage is that the search method often spends much time in subtrees not contributing to finding a solution. $\alpha\beta$ search (Knuth and Moore, 1975) is an instance of depth-first search that uses an evaluation function in the leaves.

Best-first search aims at combining the advantages of breadth-first and depth-first search. Best-first search iterates a loop that consists of two steps: (1) a heuristic is employed to find the most-promising node which is a leaf in the tree; (2) the most-promising leaf is expanded. The loop is repeated until a solution is found. A disadvantage of best-first search is that it stores the whole search tree in memory. Examples of best-first search are Proof-Number Search (Allis *et al.*, 1994) and MCTS (Kocsis and Szepesvári, 2006; Coulom, 2007a).

2.1.3 Transposition Tables and the GHI Problem

Search trees may grow large in practical applications. To reduce the size of a search tree, transposition tables are used to store the results of a searched position (Greenblatt, Eastlake, and Croker, 1967; Breuker, 1998). Transposition tables exploit the fact that nodes representing the same game positions may occur multiple times in the same search tree. More precisely, a transposition table is a table that maps a game position to some data relevant for that position. Transposition tables may encounter difficulties in games that allow reaching the same position by different paths.

As pointed out by Breuker (1998) two difficulties can arise because of the different paths. (1) A position may actually be illegal. For instance, in Go a move may violate a repetition rule when reached via a certain path but not when reached via another path. This is called the *move-generation problem*. (2) A position may be assigned a wrong game-theoretic value. For instance, in Chess a position may be a draw by the three-fold repetition rule. This is called the *evaluation problem*. Both problems are collectively referred to as the *Graph-History-Interaction problem* (GHI problem, Palay, 1983; Campbell, 1985).

2.2 Proof-Number Algorithms

As noted above, depth-first search and best-first search are more relevant to practical applications than breadth-first search. Arguably the strongest contribution that depth-first search made to solving is by $\alpha\beta$. Many games were (partially) solved by iterative-deepening $\alpha\beta$. Among them are Checkers (Schaeffer *et al.*, 2007) and small Go boards (Van der Werf *et al.*, 2003; Van der Werf and Winands, 2009). Two instances of best-first search that are particularly relevant for solving are Proof-Number Algorithms and Monte-Carlo Tree Search. Proof-Number Algorithms are presented in this section and Monte-Carlo Tree Search in Section 2.3.

Since the invention of Proof-Number Search (PNS) (Allis *et al.*, 1994) in the 1990s a family of PNS variants evolved. These variants have been successfully applied to a variety of domains including Othello (Nagai, 2002), Shogi (Seo, Iida, and Uiterwijk, 2001; Nagai, 2002), Tsume-Go (Kishimoto and Müller, 2003), and LOA (Winands, Uiterwijk, and Van den Herik, 2004). We call the family of algorithms that are variants of PNS, *proof-number algorithms*. All of them have two things in common: (1) they are solving algorithms for binary goals (proving, e.g., win or no win), and (2) they rely on the concept of proof number.

This section gives a detailed account of proof-number algorithms. Subsection 2.2.1 describes the basic PNS algorithm. Subsection 2.2.2 discusses five proof-number algorithms. Subsection 2.2.3 explains how the performance of different proof-number algorithms compares and Subsection 2.2.4 introduces three enhancements.

2.2.1 Proof-Number Search

Proof-Number Search (PNS) by Allis *et al.* (1994) is the most basic proof-number algorithm. All other variants of proof-number algorithms are descendants of PNS. This subsection describes the PNS algorithm. We do so in four parts: (A) Conspiracy-Numbers Search, (B) basic idea of PNS, (C) proof and disproof numbers, and (D) the PNS algorithm.

A. Forerunner: Conspiracy-Number Search

McAllester (1985; 1988) presented Conspiracy-Number Search which is a conceptual forerunner of PNS. Conspiracy numbers were introduced for minimax search (Von Neumann and Morgenstern, 1944) and indicate how likely it is that the root takes on a certain value v. This likelihood is expressed by the *conspiracy number* which is the minimum number of leaves (conspirators) that must change their value to cause the root to change its value to v. The idea was later improved by Schaeffer (1989; 1990).

Two types of numbers, $\uparrow CN$ and $\downarrow CN$, are used in order to calculate the conspiracy numbers for each non-terminal node N. The minimum number of conspirators to increase N's value to v is $\uparrow CN$. If N's value is greater than or equal to v, then $\uparrow CN$ is 0. $\downarrow CN$ is the minimum number of conspirators required to decrease the value of N to v. If N's value is smaller than or equal to v, then $\downarrow CN$ is 0. For calculating $\uparrow CN$ and $\downarrow CN$ at the root, $\uparrow CN$ and $\downarrow CN$ values are calculated recursively for all nodes in the tree starting at the leaves. Different updating rules apply for $\uparrow CN$ and

$\downarrow CN$ depending on whether N is the maximizing player's node or the minimizing player's node.

Conspiracy numbers are similar to proof numbers with respect to two aspects: (1) two numbers at each node represent information on the expected game-theoretic value of the position, and (2) the numbers are calculated bottom up by backpropagation from the leaves to the root.

B. Basic Idea of PNS

PNS is a best-first search algorithm. Its heuristic determines the most-promising leaf by selecting the *most-proving node* (cf. Subsection 2.1.2). The most-proving node is found by exploiting two characteristics of the search tree: (1) its shape (determined by the branching factors of internal nodes), and (2) the game-theoretic values of the leaves (which are known if a leaf is terminal). Since no other information is used by it, PNS is uninformed search, i.e., the search does not require to use any knowledge beyond the rules of the game.

C. Proof and Disproof Numbers

As already mentioned, PNS employs ideas similar to Conspiracy Number Search. In order to find the most-proving node, PNS maintains two numbers for each node N. (1) The proof number, pn or $pn(\mathrm{N})$, represents the number of leaf nodes that at least have to be proven in order to prove the goal. Analogously, (2) the disproof number, dn or $dn(\mathrm{N})$, represents the number of leaf nodes that at least have to be disproven to disprove the goal. The values of pn and dn can be calculated for each node in the tree as follows.

We start by describing how the values are initialized for leaf nodes. When a goal is proven, no further expansion is required for proving it and no further expansion can disprove it anymore. The corresponding holds for a disproven node. Adhering to this observation, for any terminal node T, pn and dn are set as follows. If the goal is proven at T, then $pn(\mathrm{T}) = 0$ and $dn(\mathrm{T}) = \infty$. If the goal is disproven at T, then $pn(\mathrm{T}) = \infty$ and $dn(\mathrm{T}) = 0$.

For any non-terminal leaf L, it is not yet possible to say what its game-theoretic value is. Thus the values for L are set according to an *initialization rule*. The most simple initialization rule directly follows the definition of pn and dn. It sets $pn(\mathsf{L}) = dn(\mathsf{L}) = 1$.

The values for each internal node I are calculated from the set of its children, $children(\mathrm{I})$. The backpropagation rules take into account whether I is an OR node or an AND node.

Backpropagation rule for OR nodes:

$$pn(\mathrm{I}) = \min_{\mathrm{S} \in children(\mathrm{I})} pn(\mathrm{S}) \ , \tag{2.1}$$

$$dn(\mathrm{I}) = \sum_{\mathrm{S} \in children(\mathrm{I})} dn(\mathrm{S}) \ . \tag{2.2}$$

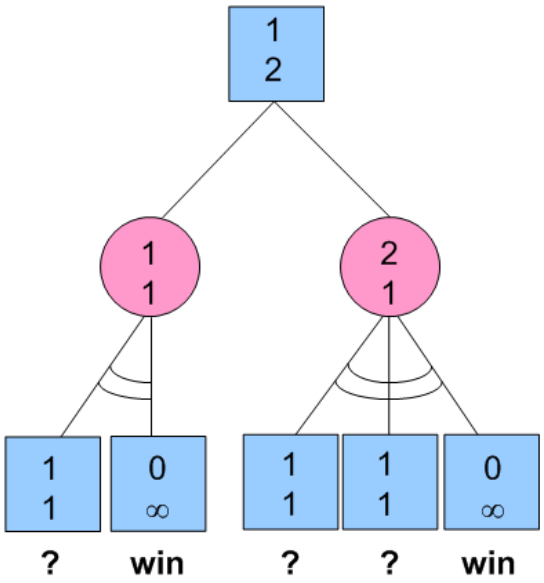


Figure 2.1: Example of a PNS tree. Square nodes are OR nodes and circular nodes are AND nodes. Each node's *pn* is given by the upper number, its *dn* by the lower number.

Backpropagation rule for AND nodes:

$$pn(\mathrm{I}) = \sum_{\mathrm{S} \in children(\mathrm{I})} pn(\mathrm{S}) \ , \tag{2.3}$$

$$dn(\mathrm{I}) = \min_{\mathrm{S} \in children(\mathrm{I})} dn(\mathrm{S}) \ . \tag{2.4}$$

Figure 2.1 gives an example of a PNS tree. OR nodes are represented by square boxes and are played by MAX, AND nodes are represented by circles and are played by MIN. We assume that player MAX tries to prove and player MIN tries to disprove the goal. Each node contains two numbers. The upper number indicates the node's proof number and the lower number its disproof number.

The values for *pn* and *dn* are calculated by applying the initialization rule and the backpropagation rules. There are two terminal leaf nodes. Their game-theoretic value is a win for MAX. Therefore, their *pn* has value 0 and their *dn* has value ∞. The non-terminal leaves have been initialized by the simple initialization rule. Their *pn* and *dn* have value 1. The values of the leaves' parents are obtained by backpropagating the leaf values according to the backpropagation rule for AND nodes. The root's *pn* and *dn* are set by applying the backpropagation rule for OR nodes on the values of the AND nodes.

The most-proving node in the example is the leftmost leaf. It is found by successively building a path starting from the root and ending at the frontier of the tree. At the root, the left AND node is chosen because it has the smallest proof

number. At the selected AND node, the child with minimal *dn* is selected. This is the leftmost leaf.

D. The PNS Algorithm

At the start of PNS, *pn* and *dn* of the root are both set to 1. As long as neither *pn* nor *dn* of the root has value 0, there are still expansions required and thus the search is continued by performing a best-first search iteration.

At each iteration, the *pn* and *dn* are kept up to date and consistent for all nodes in the tree. Each iteration consists of four phases as follows.

(1) *Selection.* The values of *pn* and *dn* guide the search towards the most-proving leaf node. This is achieved by the following rule: in OR nodes, the child with minimal *pn* is selected; in AND nodes the child with minimal *dn* is selected.

(2) *Expansion.* The most-promising leaf is expanded.

(3) *Evaluation.* The new leaves are evaluated. Their *pn* and *dn* are set using the game-theoretic values and the initialization rule as described above.

(4) *Backpropagation.* The values of *pn* and *dn* are updated such that the parents' *pn* and *dn* are consistent with their children's values. The updating is repeated successively for all predecessors of the expanded leaf.

2.2.2 Variants of PNS

In many practical applications, PNS runs out of space quickly because it stores the whole search tree in memory. We refer to this phenomenon as the *memory problem of PNS*. Several variants of PNS have been designed to address the issue. This subsection gives an overview of five of such proof-number algorithms: PN^2, PN^*, PDS, df-pn, and PDS–PN.

A. PN^2

Allis (1994) introduced PN^2, a proof-number algorithm that addresses the memory problem of PNS by discarding subtrees that have been searched. Whenever necessary, the previously discarded subtrees are re-searched. To achieve this strategy, PNS is performed at two levels, called PN_1 and PN_2.

At the first level, PN_1, a normal PNS search is conducted. However, PN^2 procedurally deviates from normal PNS when a leaf L is expanded. There, PNS is used as initialization procedure. Instead of a simple initialization for non-terminal leaves, a second-level PNS, called PN_2, is launched. The root of the PN_2 tree is the leaf L and the size of the tree is limited to a certain maximum number of nodes. The second-level search terminates if either the maximum number of nodes is exceeded or a (dis)proof has been found. When PN_2 terminates, the children of L with their *pn* and *dn* are kept as new leaves of the PN_1. All successors below the children of L are removed from memory.

As a consequence of using PN_2, leaves of the PN_1 tree contain more information than leaves of regular PNS. Memory is saved because useless PN_2 trees are removed. The memory advantage of PN^2 comes at a cost in terms of speed because the best PN_2 trees have to be re-searched.

Different approaches for setting the maximum number of nodes in the PN_2 tree exist. We note that this maximum is crucial because it determines the trade-off between the memory consumption and speed of PN^2. The main idea is to express the maximum as a function $f(x)$ of the size of the PN_1 tree given by the number x of its nodes. In his experiments on Chess positions, Breuker (1998) suggests to set $f(x)$ to be a logistic-growth function

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \ . \tag{2.5}$$

The parameters a and b are strictly positive and require tuning for optimal performance (cf. Breuker, 1998). Furthermore, the size of PN_2 is also limited by the amount of memory physically available. If N is the physical limit of nodes that memory can hold, the maximum size of PN_2 is

$$min(x \times f(x), N - x) \ . \tag{2.6}$$

B. PN*

As we have seen, PN^2 addresses the memory problem of PNS by introducing two levels of search and discarding most of the second-level results. This approach may be described as wasteful because important information has to be re-generated every time a previously discarded subtree is re-searched.

A different approach to addressing the memory problem consists of applying iterative deepening. The first algorithm to follow this approach was PN^* by Seo *et al.* (2001). PN^* uses a threshold on the proof numbers to perform iterative deepening. The search is initialized with a threshold of 2 on the root's *pn*. If no proof can be found the threshold is gradually increased until a proof can be found for the given threshold.

Essentially, PN^* performs *multiple-iterative deepening* by iteratively deepening at all AND nodes. Every node has a threshold on the proof number. The values for these thresholds are increased at the AND nodes recursively if no proof can be found within a certain threshold. We note that the disproof numbers are not used for this iterative deepening.

A transposition table is used to reduce the overhead of re-searching nodes. Seo *et al.* (2001) report that their game engine based on PN^* solved more Shogi problems from a collection of 295 problems than any other compared solver. By its iterative-deepening PN^* essentially is a depth-first search, enabling PN^* to find very deep solutions, e.g., for the notoriously hard *Microcosmos problem* with a solution sequence of 1,525 steps.

PN^* is able to cope with harsh memory constraints better than PN^2. But as a consequence of disregarding the disproof numbers for move selection, PN^* is weak at disproving subtrees.

C. PDS

To counter the deficiency PN* shows for disproving goals, Nagai (1998) presented a straightforward further-developed algorithm called *Proof-and-Disproof-Number* Search (PDS). PDS performs multiple-iterative deepening at OR nodes and AND nodes. It introduces two thresholds to limit the iterations in a node. The threshold for pn is called pt and the threshold for dn is called dt. To make explicit that the thresholds refer to a node N, we write $pt(\mathsf{N})$ and $dt(\mathsf{N})$, respectively.

The search in the subtree of a node continues until (1) a proof or disproof has been reached, or (2) simultaneously $pt < pn$ and $dt < dn$.

At the start of every iteration pt is initialized with the value of pn, and dt with the value of dn. If the tree is more proof-like than disproof-like the pt is increased. Otherwise dt is increased.

Nagai (1998) introduced a heuristic for estimating whether a subtree of a node is likely to be a proof (proof-like) or a disproof (disproof-like). An OR node N with parent P is proof-like exactly if

$$(pt(\mathsf{P}) > pn(\mathsf{P})) \wedge (pn(\mathsf{N}) \leq dn(\mathsf{N}) \vee dt(\mathsf{P}) \leq dn(\mathsf{P})) \ . \tag{2.7}$$

Similarly, an AND node N with parent P is proof-like exactly if

$$(dt(\mathsf{P}) > dn(\mathsf{P})) \wedge (dn(\mathsf{N}) \leq pn(\mathsf{N}) \vee pt(\mathsf{P}) \leq pn(\mathsf{P})) \ . \tag{2.8}$$

At the outset of each iteration at the root, one of the two thresholds of the root is increased. If $pn > dn$, then pt is increased by 1 and otherwise dt is increased by 1.

The expanded nodes of PDS may be stored in a TwoBig transposition table (cf. Breuker, Uiterwijk, and Van den Herik, 1996) to guarantee more efficient re-searching.

D. df-pn

Nagai (1999; 2002) introduced an improved variant of the PDS algorithm, called *depth-first proof-number search*, df-pn. In comparison to PDS, df-pn further reduces the memory requirements by applying depth-first search. However, df-pn has been shown to suffer strongly from the GHI problem (cf. Subsection 2.1.3). An advantage of df-pn over PDS is that it has been shown by Nagai (2002) that df-pn always selects the most-proving node (this is not the case for PDS).

We recall that PDS gradually increased the thresholds pt and dt for a node if a proof was not found within the given boundaries. df-pn follows a different approach. Assume some node P has children $\mathsf{C1}$ and $\mathsf{C2}$. df-pn sets the threshold of $\mathsf{C1}$ depending on the next best sibling $\mathsf{C2}$. More precisely, if P is an OR node

$$pt(\mathsf{C1}) = min(pt(\mathsf{P}), pn(\mathsf{C2}) + 1) \ , \tag{2.9}$$

$$dt(\mathsf{C1}) = dt(\mathsf{P}) - dn(\mathsf{P}) + dn(\mathsf{C1}) \ . \tag{2.10}$$

If P is an AND node,

$$pt(\mathsf{C1}) = pt(\mathsf{P}) - pn(\mathsf{P}) + pn(\mathsf{C1}) \ , \tag{2.11}$$
$$dt(\mathsf{C1}) = min(dt(\mathsf{P}), dn(\mathsf{C2}) + 1) \ . \tag{2.12}$$

E. PDS–PN

A fifth proof-number algorithm, PDS–PN by Winands *et al.* (2004), tries to combine the strength of PN^2 and PDS. PDS–PN can be described as a PN^2 which relies on PDS as PN_1 search and normal PNS as PN_2. At the first level, the search is a depth-first search. This assures that PDS–PN is practically not restricted by memory. At the second level, it profits from the speed of best-first search given by PNS.

2.2.3 Performance of Proof-Number Algorithms

Two main points require attention when comparing proof-number algorithms: (1) the trade-off between speed and memory, and (2) robustness with respect to the GHI problem (cf. Subsection 2.1.3). This subsection elaborates on these two points and shows how the six proof-number algorithms compare with each other.

A. Speed vs. Memory

In order to address the memory problem as experienced by PNS, the variants PN^2, PN^*, PDS, df-pn, and PDS–PN trade speed for memory. The trade-off is an important feature for comparing proof-number algorithms. In general, an algorithm that emphasizes speed more than memory may achieve good results on small problems but also may run out of memory on large problems. Conversely, an algorithm that manages memory economically may be able to solve harder problems but re-searches more often and therefore lose speed.

No uniform comparison exists juxtaposing all proof-number algorithms on the same data set, but comparisons involving several algorithms have been conducted.

Experiments by Nagai (1999; 2002) in Othello and Tsume-Shogi have produced better results for PDS than for PNS and PN^*. df-pn in turn has been shown to solve hard problems faster than PDS (Nagai, 2002).

Winands, Uiterwijk, and Van den Herik (2003b) carried out a comparison between PNS, PDS, and PN^2 on 488 LOA endgame positions. In this comparison, PDS was able to solve only three problems more than PN^2 which solved 470. At the same time PDS was more than six times slower. In this experiment, PN^2 was allowed to use transposition tables and the same rule for initializing non-terminal leaves as PDS. PNS was about 20% faster than PN^2 but solved only 356 positions.

PDS–PN was found by Winands *et al.* (2003b) to be the intended compromise between PN^2 and PNS requiring less memory than PN^2 while simultaneously solving faster than PN^2. The exact trade-off is a function of the number of nodes that PDS is allowed to store in memory with more nodes allowing for more speed.

B. Robustness to the GHI Problem

All proof-number algorithms that perform iterative deepening depend heavily on the use of transposition tables. This makes them susceptible to the GHI problem (cf. Subsection 2.1.3). Both PDS and particularly df-pn were found to be vulnerable in practice (Kishimoto and Müller, 2003).

2.2.4 Enhancements for Proof-Number Algorithms

So far, we have seen which proof-number algorithms exist and what points are important when comparing the performance of such algorithms. This subsection presents three kinds of enhancement that have been proposed for proof-number algorithms: (A) improved initialization of proof and disproof numbers, (B) the $1 + \epsilon$ trick, and (C) solutions to the GHI problem.

A. Initialization of Proof and Disproof Numbers

Subsection 2.2.1 presented a simple initialization rule that sets pn and dn to 1 for newly generated non-terminal leaves. Better results can be achieved by using more advanced initialization rules. Two different kinds of advanced initialization rules can be distinguished: (1) using the branching factor, and (2) exploiting domain-dependent knowledge.

Ad (1). The branching factor $bf(\mathsf{L})$ of a leaf L can be used as follows (cf. Allis, 1994). If L is an OR node, pn is set to 1 and dn is set to $bf(\mathsf{L})$. If L is an AND node, dn is set to 1 and pn is set to $bf(\mathsf{L})$. Effectively, this initialization rule adds an additional ply of information. We note that combining this initialization rule with an uninformed proof-number algorithm leaves the algorithm uninformed. The branching factor is used implicitly in the original description of PDS (Nagai, 1998). Winands (2004) calls this initialization rule *mobility* when applied to LOA and reports a reduction in nodes of roughly a factor of five for both PNS and PN^2.

Ad (2). The second approach to initialize non-terminal leaves consists of using heuristic knowledge. In this case, heuristics estimate the number of further expansions required to prove or disprove the goal. These estimates are then used as pn or dn, respectively. The heuristics contain domain knowledge. Domain knowledge was used for initialization successfully by Allis (1994) who used the number of captured stones in Awari to improve PNS. A second example following this approach is Nagai (1999) for df-pn on Othello. Nagai's solver uses a pattern-based evaluation function for estimating the cost of reaching a proof or disproof at a given node. The latter achieved a reduction of up to 50% in nodes. A third example is given by Kishimoto and Müller (2005a) who compute an approximation of the number of successive moves required to reach a tactical goal in the game of Go. These approximations are then used to initialize the proof and disproof numbers.

B. The $1 + \epsilon$ Trick

A second powerful enhancement for proof-number algorithms is the so-called $1 + \epsilon$ trick by Pawlewicz and Lew (2007). This 'trick' is based on a simple observation

in df-pn. We have seen that df-pn imposes thresholds on its nodes' *pn* and *dn* to determine how deeply subtrees are searched. If the thresholds are too big, the search in a subtree may occupy a large share of the transposition table.

Assume an OR node N with threshold $pt(\mathsf{N})$ is searched and has the two children C1 and C2 with thresholds $pt(\mathsf{C1})$ and $pt(\mathsf{C2})$, respectively, and both are much smaller than N's threshold $pt(\mathsf{N})$. The search switches from C1 to C2 if the threshold $pt(\mathsf{C1}) = pt(\mathsf{C2}) + 1$ is exceeded. Next, the search continues in the tree under C2 until the threshold $pt(\mathsf{C2}) = pt(\mathsf{C1}) + 1$ is exceeded. The search switches back to the tree under C1. Every time the search switches between the two subtrees, information in the transposition table is overwritten. As a consequence, the trees have to be re-constructed by an expensive effort. Pawlewicz and Lew (2007) concluded that not enough time is spent in the same subtree continuously.

The $1 + \epsilon$ trick increases the time continuously spent in the same subtree. The constraint used in df-pn for setting the thresholds to be just 1 larger than the next best child's threshold, is relaxed. The new threshold depends on a small factor $1 + \epsilon$. The next best child's threshold is multiplied with this factor to produce the new threshold. Thus, the new rule for setting the threshold of an OR node is:

$$pt(\mathsf{C1}) = min(pt(\mathsf{N}), \lceil pt(\mathsf{C2}) \times (1 + \epsilon) \rceil) \ . \tag{2.13}$$

The rules for setting the thresholds for AND nodes are set analogously.

Experiments in LOA show a speed increase of up to a factor of three for df-pn. An adaptation of the threshold rules for PDS shows a speed increase of still up to 10%. Moreover, the $1 + \epsilon$ trick was shown to give an improvement on the number of problems solved for Atari Go problems Pawlewicz and Lew (2007).

C. Solutions to the GHI Problem

At least three approaches for addressing the GHI problem for proof-number algorithms have been suggested: (1) the base-twin algorithm (BTA) by Breuker *et al.* (2001), (2) modified bounds on the root node of df-pn by Nagai (2002), and (3) verification of search results by Kishimoto and Müller (2003). We briefly outline strengths and weaknesses reported for the three approaches.

Ad (1). BTA (Breuker *et al.*, 2001) distinguishes two kinds of nodes: *base nodes* and *twin nodes*. This distinction allows to separate nodes that are reached by different paths. Kishimoto and Müller (2003) point out three problems of BTA. (i) There is no proof whether BTA works on depth-first algorithms with limited memory, (ii) it does not work in positions in which the current player loses, (iii) the algorithm can deliver incorrect results when real draws are not stored and thus requires much more memory.

Ad (2). Nagai (2002) proposes an ad-hoc solution to the GHI problem in df-pn. This solution is used for problems known to have a solution for the starting player under so the called *first-player loss scenario*. In this scenario, any repetition is a loss for the first player, i.e., the root player. Any Shogi problem in which the first player tries to checkmate is a first-player loss scenario: if a repetition is encountered, it disproves the possibility of the checkmate.

Instead of assigning ∞ to both thresholds at the root, $\infty - 1$ is assigned initially. Nagai (2002) uses integers to express ∞ and $\infty - 1$ with $\infty - 1 < \infty$. Next, the search is launched. When the termination criterion is met (e.g., $pn = 0$ and $dn = \infty$ at the root) during the search process, the proof was made without loops. If the criterion has not been met, this means that a loop was encountered. In this case, the normal initialization of the root thresholds with ∞ instead of $\infty - 1$ is used. This allows a proof that may contain loops. Thus, Nagai's solution can distinguish between proofs that are guaranteed to be loop free and such that may contain loops.

The two disadvantages of the modified df-pn are: (1) it may take a long time to find a solution, (2) for a solution found by initializing the thresholds with ∞, the proof may still suffer from the GHI in games with a *current-player-loss scenario*. In such games, the player who repeats a position, loses. An example is Go with the *situational super-ko rule* (Kishimoto and Müller, 2003).

Ad (3). Kishimoto and Müller (2005b) and Kishimoto (2005) propose a solution to the GHI that can be applied to search algorithms other than proof-number algorithms but was particularly tested for df-pn. The main idea is to verify the value for a position retrieved from the table by a re-search. The verification requires storing the path by which a position was reached in the transposition table. For this idea to work in practice an efficient algorithm is used for storing and comparing paths. Compared to the modified-bounds approach the verification algorithm solved three additional problems. It used 2.5% more nodes and about 12.4% more time on a set of 200 Checkers problems.

We may conclude by saying that currently three solutions to the GHI problem exist. In particular, the proposal of Kishimoto and Müller (2005b) is promising for practical applications.

2.3 Monte-Carlo Techniques for Search

Monte-Carlo methods have been applied for a long time in computer science for approximating function values in complex domains (Metropolis and Ulam, 1949). In computer games, however, Monte-Carlo has gained increased popularity only recently. To understand this development, we introduce a distinction between two kinds of Monte-Carlo techniques used in search and games.

The first technique is Monte-Carlo Evaluation (MCE). It has been studied since the mid 1990s and has been employed as an evaluation function (Abramson, 1990). The second technique is Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007a; Chaslot *et al.*, 2006). It is a framework for game-tree search based on MCE. Monte-Carlo Tree Search may be seen as a recent development in the history of game-tree search. In particular, it improved the strength of game engines in domains lacking satisfactory evaluation functions. The most prominent of these domains, Go, saw an increase in the playing strength of programs from amateur to professional level, at least on 9×9 boards, owing to MCTS (cf. Chaslot, 2008).

In this section we introduce both, MCE in Subsection 2.3.1 and MCTS in Subsection 2.3.2. MCTS can be transformed into a solving algorithm (Winands *et al*, 2008). We remark that this is described in detail in Chapter 4.

2.3.1 Monte-Carlo Evaluation

Evaluation functions estimate the game-theoretic value of a game position (Pearl, 1984). Features such as material or mobility of the position are commonly exploited to compute the heuristic value. A different approach to evaluation is Monte-Carlo Evaluation (MCE, Abramson, 1990): a number of *playouts* (also called *samples* or *simulations*) is started at a position. A playout is a sequence of (pseudo-)random moves. At the end of every playout, the final position is scored according to the rules of the game.

The evaluation of a position is calculated by applying a statistical aggregate function of all scores. Two simple examples of such an aggregate function are: (1) the average game score (e.g., Black wins with 5.5 points on average) and (2) the ratio of wins to losses (e.g., Black wins 49% of the games).

The remainder of this subsection shows an overview of games to which MCE has been applied, (A), and explains two common ways of improving MCE, (B) and (C).

A. Monte-Carlo Evaluation in Games

During the 1990s Monte-Carlo Evaluation (MCE) was applied to stochastic games and games with imperfect information such as Backgammon (Tesauro and Galberin, 1997), Bridge (Smith, Nau, and Throop, 1998; Ginsberg, 1999), Poker (Billings *et al.* 1999), and Scrabble (Sheppard, 2002). For instance, MCE was tested in Bridge (Smith, Nau, and Throop, 1998) because it offers a way of coping with imperfect information. Concealed cards are treated by MCE as follows. First, a pseudo-random deal is generated for the concealed cards. Given this known deal, the game is then played as a perfect-information game and scored accordingly. The perfect-information game constitutes a playout. The statistical evaluation of several playouts is a heuristic score for the initial position (Frank, Basin, and Matsubara, 1998). In a similar way, uncertainty in the game of Skat can be treated (Kupferschmid and Helmert, 2007).

The first systematic investigation of MCE in two-player perfect-information games was conducted by Abramson (1990). He tested MCE on Tic-Tac-Toe, on Chess positions, and on 6×6 Othello, concluding that MCE can outperform alternative evaluation functions. Furthermore, he suggested to include domain knowledge to improve MCE in ways described below in (B) and (C).

In the domain of perfect-information games, MCE has in particular affected Computer Go. In Go, alternative evaluation functions had been unsuccessful because of two reasons as stated by Müller (2002): (1) a large branching factor, and (2) global effects of local moves. The first to apply MCE to Go was Brügmann (1993), who evaluated the initial position of the 9×9 board.

B. Biasing Playouts and Alternative Statistical Evaluation

Standard MCE as described above can be improved in two ways: (1) by biasing playouts with domain knowledge, and (2) by alternative statistical evaluation.

Ad (1). The pseudo-randomness of moves during playouts can be altered with domain knowledge to produce better estimates in many games. For Go, Bouzy

and Helmstetter (2003) suggest setting the likelihood of a playout move proportionate to the known frequency distribution of local patterns completed by that move in a database of game records. The Monte-Carlo version of the world champion LOA engine MIA (Winands and Björnsson, 2010) can stop playouts before reaching a terminal position. It scores the last position of the playout with a classic evaluation function. Thereby, MIA can run more MCEs while simultaneously ensuring a sufficient quality of the scores for each playout. A similar approach is used in the Amazons program InvaderMC by Lorentz (2008). The difference is that InvaderMC stops the MCE at a fixed depth whereas MIA allows the playouts to terminate at any depth.

Ad (2). The simple aggregate functions described above (average game score, win-loss ratio) are used to score only one position. This may be wasteful particularly if playouts cannot be computed in sufficient quantity. Brügmann (1993) suggested an alternative statistical evaluation, the *all-moves-as-first* heuristic for evaluating the initial position of 9×9 Go. In normal MCE, the score of a single playout contributes only to the evaluation of the initial position of that playout. The all-moves-as-first heuristic proposes instead to use one score for the evaluations of all moves that occur in the playout. In this way, few playouts affect the evaluations of many moves. The all-moves-as-first heuristic is inaccurate but converges faster with respect to the number of playouts.

C. Exploration and Exploitation in the Move-Selection Problem

So far, we have described how MCE can give an evaluation of a position. In practice, the evaluation function is only a means in a game-playing engine. The actual end is solving a *move-selection* problem, i.e., given all possible moves to find the best.

When applying MCE in the evaluation function for multiple moves the time and therefore the playouts have to be distributed efficiently among all available moves. Because of MCE's inherent randomness the estimated game value has a high variance. The variance decreases inversely proportionate to the number of playouts. The demand for distributing the playouts over the available positions efficiently leads to the problem of optimizing the ratio between exploration and exploitation in MCE. Exploration performs playouts for a move that has so far been assessed as bad. Exploring this move may lead to discovering that the move is actually promising. Conversely, exploitation invests more time in comparing such moves more precisely that have already been assessed as relevant and avoids wasting time on irrelevant moves.

A naive solution to balancing exploration and exploitation for move selection consists of uniformly distributing the playouts over all available moves. All evaluations are compared and the best-scoring move is selected. However, this naive solution generally produces suboptimal results (cf. Bouzy and Helmstetter, 2003).

Bouzy and Helmstetter (2003) introduced the idea of *progressive pruning* for the game of Go. For all available moves i the mean values μ_i and standard deviations σ_i are recorded for the number of playouts made so far. A move 2 is said to be *inferior* to a move 1 if $\mu_2 + \sigma_2 \leq \mu_1 - \sigma_1$. To select the best move, first, the playouts are distributed uniformly over all moves. Next, the playouts are made and every

move is evaluated. Then, all moves inferior to some other move are eliminated. The procedure is iterated with the remaining moves. In this manner, promising moves are sampled more frequently than initially less promising candidates.

2.3.2 Monte-Carlo Tree Search

In the past, MCE has been used as an evaluation function for game-tree search (Bouzy and Helmstetter, 2003). Yet, this approach remained too slow to achieve a satisfying search depth. To overcome this problem Bouzy (2006) extended the idea of progressive pruning from one ply to the whole search tree. He suggested to grow a search tree by iterative deepening and pruning unpromising nodes while keeping only promising nodes. All leaf nodes are evaluated by MCE. A problem with this approach is that actually good branches are pruned entirely because of the variance underlying MCE.

Monte-Carlo Tree Search (MCTS) follows Bouzy's generalization of extending MCE to a search tree and constitutes an entire best-first search framework for MCE. MCTS avoids the problem of pruning too early by maintaining all nodes and controlling exploration and exploitation better than Bouzy's method.

The idea of MCTS was independently introduced by Coulom (2007a) and Kocsis and Szepesvári (2006). The MCTS algorithm by Coulom (2007a) was specifically designed and tested for the domain of computer Go. The program Crazy Stone had incorporated the algorithm, participated in the 12^{th} Computer Games Olympiad, and won the 9×9 Go competition (Coulom and Chen, 2006). The MCTS algorithm by Kocsis and Szepesvári (2006) is called Upper Confidence bounds applied to Trees (UCT). It was motivated by research on the Multi-Armed Bandit Problem (Robbins, 1952).

Since its introduction MCTS has been applied over a variety of domains including optimization (Chaslot *et al.*, 2006), one-player games (Schadd *et al.*, 2008; Cazenave, 2009), two-player perfect-information games such as Go (Coulom, 2007a; Gelly and Silver, 2008; Enzenberger and Müller, 2009), LOA (Winands *et al.*, 2008), Amazons (Lorentz, 2008), Hex (Cazenave and Saffidine, 2009), two-player imperfect-information games such as Phantom Go (Cazenave and Borsboom, 2007), multi-player games including Chinese Checkers (Sturtevant, 2008), and general game playing (Finnsson and Björnsson, 2008).

This subsection is organized as follows. Subsection A details an algorithmic description of the MCTS framework. Subsection B introduces UCT, one of the most common representatives of MCTS. Subsection C outlines three common enhancements for MCTS.

A. The MCTS Algorithm

Monte-Carlo Tree Search applies a best-first search on a global level and MCE as an evaluation function at the leaf nodes. The results of previous playouts are used for developing a best-first search tree. MCTS repeatedly iterates the following four stages (Chaslot *et al.*, 2008b), also depicted in Figure 2.2:

(1) *move selection,*

(2) *expansion,*

(3) *playout,*

(4) *backpropagation.*

Each node in the tree contains at least three different tokens of information: (i) a state representing the game position associated with this node, (ii) the number of times n the node has been visited during all previous iterations, and (iii) a value v representing the estimate of the position's game-theoretic value. The search tree is stored in memory. Before the first iteration, the tree consists only of the root node. By applying the four stages successively in each iteration, the tree grows gradually. The four stages of the iteration work as follows (cf. Figure 2.2).

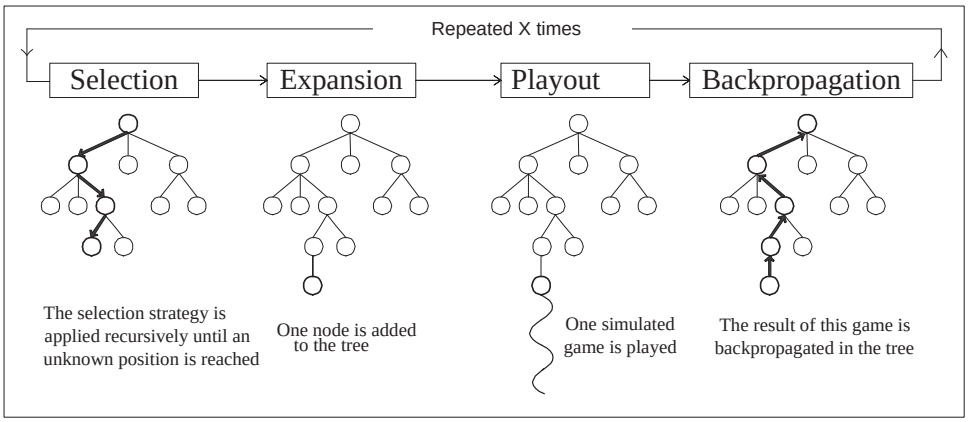


Figure 2.2: The four stages of MCTS (slightly adapted from Chaslot *et al.*, 2008b).

(1) The move selection determines a path from the root to a leaf node. This path is gradually developed. At each node, starting with the root node, the best successor node is selected by applying a function to all child nodes. Then, the same procedure is applied to the selected child node. This procedure is repeated until a leaf node L is reached.

(2) In case a leaf node has been sampled sufficiently frequently (a threshold is set manually) the node is expanded.

(3) In the playout stage, Monte-Carlo evaluation is applied to L. It consists of a randomly played sequence of moves that ends in a terminal position. A statistic aggregate of all random games sampled from L constitutes the Monte-Carlo evaluation for L.

(4) During the backpropagation stage, the result of the leaf node is propagated back along the path followed in the move selection. For each node on the path back to the root, the node's value (cf. above) is adjusted according to the

update function.[2] After the root node has been updated, this stage and the iteration are completed. As a consequence of altering the values of the nodes on the path, the move selection of the next iteration is influenced.

The various MCTS algorithms proposed in the literature differ with regard to their move selection and schemes used for expansions. Next, a specific move selection called UCT is briefly described.

B. UCT

The most influential representative of MCTS has so far been Upper Confidence bounds applied to Trees (UCT). In UCT, Kocsis and Szepesvári (2006) apply solutions for the Multi-Armed Bandit Problem (MAB) (Robbins, 1952) to tree search. This problem is structurally similar to move selection and the task of efficiently distributing actions over several slot machines to maximize the expected return. The distributions underlying the returns are unknown and have to be approximated. Auer, Cesa-Bianchi, and Fischer (2002) suggested to use a formula called Upper Confidence Bounds (UCB) to cope with the MAB. This UCB formula is the basis for UCT. The move selection of UCT is as follows. Given a node with children, the move-selection function of UCT chooses the child which maximizes the following formula:

Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the k-th child of the node p that satisfies Formula 2.14:

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} \right) \quad , \tag{2.14}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of node p. The coefficient C determines the balance between exploitation and requires tuning experimentally. The update function used in UCT sets the value n_i of a node to the average of all the children's values.

C. MCTS Enhancements

In this subsection we describe three common enhancements to MCTS: (1) RAVE, (2) integrating domain knowledge in MCTS, and (3) parallelization.

Ad (1). Gelly and Silver (2008) suggested Rapid Action Value Estimation (RAVE) which is closely related to the all-moves-as-first heuristic by Brügmann (1993) (cf. Subsection 2.3.1.B). RAVE achieves an advantage by using all-moves-as-first values to bias the UCT value of a move. Compared to UCT values (Formula 2.14), all-moves-as-first values have smaller variance and converge faster. Therefore the all-moves-as-first values are used to bias the move selection only in the beginning. When sufficiently many playouts have been sampled and the UCT values are expected to be sufficiently precise, the effect of the RAVE value diminishes. The speed of fading out is subject to parameter tuning. Gelly and Silver (2008) have shown that RAVE can improve the playing strength of a Go engine substantially.

[2] Coulom (2007a) refers to the update function as backpropagation operator.

Ad (2). We have seen that domain knowledge may improve the quality of playouts in MCE (cf. Subsection 2.3.1.B). Domain knowledge can also be applied outside of the playout stage in MCTS. There are at least three approaches of achieving this. The first approach modifies the move selection. Gelly and Silver (2008) and Chaslot *et al.* (2008b) suggest adding a summand to the right-hand side of the original UCT formula (Formula 2.14). The additional summand represents a value based on small patterns and thus representing domain knowledge. A second approach is *progressive unpruning* (Chaslot *et al.*, 2008b) which first reduces and then increases the branching factor of an MCTS node based on domain knowledge. A similar approach is referred to as *progressive widening* by Coulom (2007b). A third approach consists of using move categories based on domain knowledge in order to group moves (Saito *et al.*, 2007b).

Ad (3). MCE and MCTS are suitable for parallelization because the playouts can be computed as independent parallel tasks. A difficulty lies in parallelizing the operations on the MCTS tree. Several methods have been suggested for parallelization. Building up on Cazenave and Jouandeau (2007) who had suggested three simple ways of parallelizing UCT, (Chaslot *et al.,* 2008a) compared the following four methods for parallelization: (i) leaf parallelization executes multiple MCEs at the same leaf in parallel; (ii) root parallelization runs multiple independent instances of MCTS and merges the results of the root into one value when the search terminates; (iii) tree parallelization with global mutex maintains one lock on the whole search tree such that each of several threads locks the whole tree when updating data; (iv) tree parallelization with local mutexes allows each thread accessing different parts of the subtree. Based on experiments for 9×9 Go, it was found that the simple root parallelization produced the best results (Cazenave and Jouandeau, 2007; Chaslot, Winands, and Van den Herik, 2008a). More recently, Enzenberger and Müller (2010) documented the parallelization of FUEGO, the 2009 Computer Olympiad's gold medalist program in the 9×9 Go competition. It uses a lock-less implementation of the local mutex parallelization. This implementation produces inaccurate updates at minor rates. Although playouts are discarded because of the imprecision, the locking scales well for up to seven processors.

2.4 Chapter Summary

In this chapter we introduced basic definitions for describing game-tree search and fundamental concepts including three search methods. The elementary concepts then served to outline six proof-number algorithms and two Monte-Carlo techniques. The following chapters will introduce novel algorithms based on the algorithms that have been presented here.

In Chapter 3 we show how MCE can be combined with Proof-Number Search for solving Go problems. Chapter 4 describes an approach that utilizes MCTS for solving and also introduces a parallel version of this approach.

Chapter 3

Monte-Carlo Proof-Number Search

This chapter is based on the following publications.[1]

1. J-T. Saito, G.M.J.B. Chaslot, J.W.H.M. Uiterwijk, and H.J. van den Herik. Monte-Carlo Proof-Number Search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Computers and Games - 5th International Conference (CG '06)*, volume 4630 of *Lecture Notes in Computer Science*, pp. 50–61. Springer, Berlin, Germany, 2007.
2. J-T. Saito, G.M.J.B. Chaslot, J.W.H.M. Uiterwijk, and H.J. van den Herik. Pattern Knowledge for Proof-Number Search in Computer Go. In P.Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence (BNAIC'06)*, pp. 275–281, 2006.
3. J-T. Saito, G.M.J.B. Chaslot, J.W.H.M. Uiterwijk, H.J. van den Herik, and M.H.M. Winands. Developments in Monte-Carlo Proof-Number Search. In H. Matsubara, T. Ito, and T. Nakamura, editors, *Proceedings of the 11th Game Programming Workshop*, pp. 27–31. Information Processing Society of Japan, Tokyo, Japan, 2006.

The overview on solving algorithms given in Chapter 2 divides Monte-Carlo methods for game-tree search in two kinds: (1) Monte-Carlo evaluation, and (2) Monte-Carlo Tree Search. So far, little attention has been paid to the potentially beneficial effect of applying Monte-Carlo methods to solvers such as Proof-Number Search (PNS) (Allis *et al.*, 1994). In this chapter, we answer the first research question, **RQ 1**, by investigating how Monte-Carlo evaluation can be used for solving game positions. We propose integrating Monte-Carlo evaluation with Proof-Number Search to form a new solving algorithm. We call the new algorithm Monte-Carlo Proof-Number Search (MC-PNS).

[1] The author is grateful to Springer-Verlag and the Information Processing Society of Japan for the permission to reuse relevant parts of the articles.

This chapter presents two experiments evaluating the new algorithm on a subproblem of the game of Go, the *life-and-death* problem. Experiment 1 tunes parameters of MC-PNS. The results of Experiment 1 also enable a first comparison between PNS and MC-PNS. Experiment 2 compares PNS, MC-PNS, and pattern-based improvements of PNS and MC-PNS.

The chapter is organized as follows. Section 3.1 explains MC-PNS in detail. Section 3.2 outlines the setup of Experiment 1, Section 3.3 gives the results, and Section 3.4 discusses the findings. Section 3.5 explains patterns in Go and describes how they can be combined with PNS. Section 3.6 presents the setup of Experiment 2 and Section 3.7 the results. These are in turn discussed in Section 3.8. Section 3.9 concludes the chapter and points out three directions for future research.

3.1 Monte-Carlo Proof-Number Search

For integrating Monte-Carlo evaluation (MCE) and PNS two ways appear natural. First, a global Monte-Carlo move-selection framework might take advantage of local PNS. In this case a top-level co-ordination mechanism needs to determine when to shift from MCE to local PNS. Second, PNS forms the search framework, and MCE is used to influence the move selection. In this chapter, we address only the second approach.

This section proposes the new algorithm, Monte-Carlo Proof-Number Search (MC-PNS). It is based on PNS and MCE. MC-PNS extends PNS' best-first heuristic by adding MCE at the initialization of the leaves' values. This may lead to a more efficient selection of moves.

The section is organized in two parts as follows: Subsection 3.1.1 presents the algorithm. Subsection 3.1.2 gives three parameters that control the behavior of MC-PNS.

3.1.1 Algorithm

Like PNS, MC-PNS performs a best-first search in an AND/OR game tree. The search aims at proving or disproving a binary goal, i.e., a goal that can be reached by player OR or be refuted by player AND under optimal play by both sides. Each node N in the tree contains two real-valued numbers called the proof number ($pn(\mathsf{N})$) and the disproof number ($dn(\mathsf{N})$), respectively.

MC-PNS performs a four-step cycle fully identical to that of PNS. For reasons of readability we summarize the four-step cycle here. In the first step (selection) the best-first strategy requires the algorithm to traverse down the tree starting at the root guided by the smallest proof or disproof number until the most-proving leaf L is reached. The second step (expansion) consists of expanding L. The third step (evaluation) assigns initial values to the newborn children of L. In the fourth step (backpropagation), the values of the proof and disproof numbers are updated and backpropagated for all nodes on a path from L through the previously traversed nodes, all the way back to the root.

The cycle is complete as soon as the root has been reached and its values are updated. The cycle is repeated until the termination criterion is met. The criterion

is satisfied exactly if either the root's proof number is 0 and the disproof number is infinity, or vice versa. In the first case, the goal is proven. In the latter case it is refuted. Still, there is a difference between PNS and MC-PNS. The next paragraph outlines the details of the algorithm more formally and shows the small differences which mainly lie in the third step, the evaluation of the leaf nodes.[2]

Let L be a leaf node. If L is a node proving the goal then $pn(\mathsf{L}) = 0$ and $dn(\mathsf{L}) = \infty$ holds. If L is a node disproving the goal then $pn(\mathsf{L}) = \infty$ and $dn(\mathsf{L}) = 0$ holds. If L does not immediately prove or disprove a goal $pn(\mathsf{L}) = pmc(\mathsf{L})$ and $dn(\mathsf{L}) = dmc(\mathsf{L})$, where *pmc* and *dmc* are the Monte-Carlo-based evaluation functions mapping a node to a target range. This range is $(0, 1]$, with *pmc* reflecting a node's estimated probability to reach the goal, and *dmc* reflecting a node's expected probability not to reach a goal. The value 0 is excluded to avoid setting $pn = 0$ or $dn = 0$ inaccurately. Thereby, we prevent a false classification of a node as proven or disproven, respectively.

The *pmc* and *dmc* numbers for a position with n simulated games are gained by calculating the evaluation function $eval_n : \{0, ..., n\} \to (0, 1]$. The function $eval_n$ depends on the number n_+ of simulated games in which the goal was reached and the number n_- of simulated games in which the goal was not reached. We let $eval_n(0) = \epsilon$ for some small positive real number $\epsilon < 1$ and $eval_n(n_+) = n_+/(n+1)$ for $n_+ > 0$. We set $dmc = eval_n(n_+)$ and $pmc = 1 - dmc$.

Starting at the most-proving leaf, *pn* and *dn* values are backpropagated by PNS' backpropagation rule (cf. Section 2.2). Note that MC-PNS causes a different move selection, because the values of the proof number and disproof numbers also change for internal nodes. Figure 3.1 shows a game tree expanded by MC-PNS. The leaves of the right subtree have already been evaluated: the *pn* and *dn* values are real numbers based on the MCE. The right leaf of the left subtree has just been sampled. The MCE, represented by a set of random move sequences, has initialized the *pn* and *dn* resulting in updated values of the root.

3.1.2 Controlling Parameters

MC-PNS imposes an overhead in evaluation for each expansion of a node. The algorithm aims at achieving a better move selection as a trade-off. Two extreme approaches can be distinguished: (1) MC-PNS spends hardly any time on evaluation, and (2) MC-PNS takes plenty time on evaluation. Below, three control parameters are introduced which will enable a trade-off between these extremes.

1. *Number of MCEs per node.* The precision of the MCE can be determined by the number of simulated games at each evaluated node. Henceforth, this number will be denoted by n.

2. *Look-ahead for MCE.* The look-ahead (la) is the maximum length of a simulated game.

[2]The four-step cycle can be optimized by not fully backpropagating the values when this is not required (cf. Allis, 1994). Thereby, the cost of traversal can be reduced. The implementation used in the experiments described here does not include this optimization.

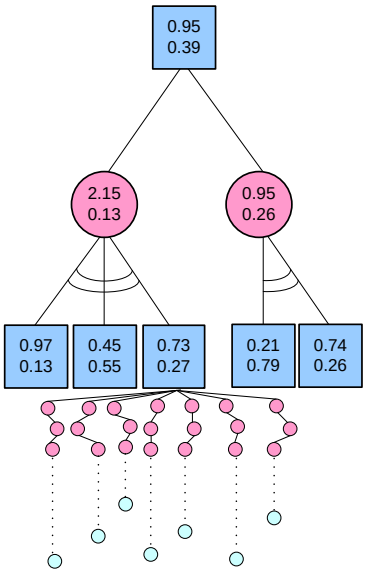


Figure 3.1: Schematic representation of a search tree generated by MC-PNS.

3. *Launching level of the MCE.* Given a limited look-ahead, it is not useful to waste time on evaluating close to the root of the search tree. The nodes close to the root must be expected to be expanded anyway. The launching level (*depth*) is the minimum required level of the tree at which nodes are evaluated.

3.2 Experiment 1: Tuning the Parameters of MC-PNS

This section describes Experiment 1. It tests different settings of the three parameters of MC-PNS introduced in Subsection 3.1.2 and compares the results obtained for the various settings with those of PNS. The experiment is conducted by applying the algorithms to a test set of life-and-death problems of the game of Go. Subsection 3.2.1 explains life-and-death problems. Subsection 3.2.2 describes the test set, Subsection 3.2.3 the implementation of the algorithm and Subsection 3.2.4 the test procedure.

3.2.1 Life-and-Death Problems

We applied MC-PNS for solving instances of the life-and-death problem which is a frequently occurring sub-problem in Go games. Life-and-death problems are also studied separately as puzzles. Applying search for solving life-and-death problems has been addressed before, e.g., by Wolf (2000), Cazenave (2003), and Kishimoto and

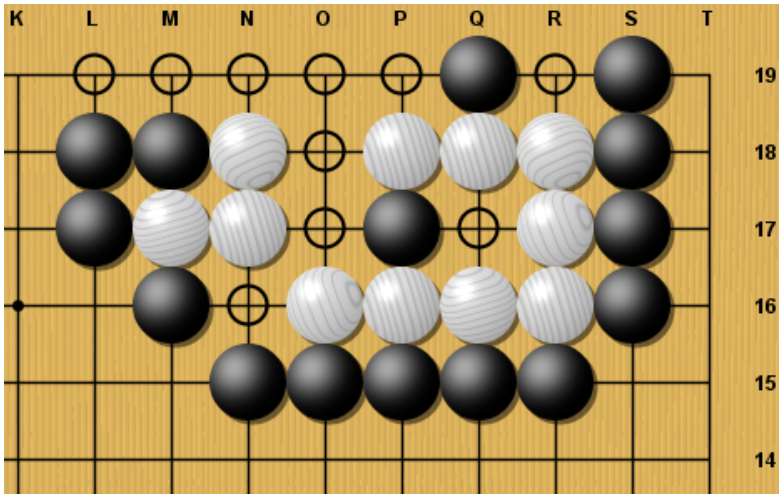


Figure 3.2: Example of a life-and-death problem.

Müller (2005a). The general life-and-death problem consists of a locally bounded game position with a target group of stones. Black moves first and has to determine the group's status as either alive, dead, ko, or seki. For the current investigation, however, the problem is reduced to a binary classification of the target group to either alive or dead.

In order to fit the algorithm to the Go domain, the goal to prove, i.e., the status of a group of stones as either alive or dead, is checked by a simple status-detection function. This function is called by each simulated move to determine whether to play further or to stop. Each simulated game halts after either the goal has been met or a certain pre-set depth has been reached.

Figure 3.2 depicts a life-and-death problem from the test set used in Experiments 1 and 2. Player Black is to move and tries to capture the white group. The intersections marked by circles are taken into consideration as starting moves.

3.2.2 Test Set

The test set consists of 30 life-and-death problems. The problems are of beginner and intermediate level (10 Kyu to 1 Dan) and taken from a set publicly available at GoBase.[3] All test problems are a win for Black. The test cases were annotated with marks for playable intersections and marks for the groups subject to the life-and-death classification. The number of intersections I becoming playable during search is determined by the initial empty intersections and by intersections which are initially occupied but become playable because the occupying stones are captured. For the test cases this indicator I of search space varied from 8 to 20. The factorial of I gives a rough lower bound for the number of nodes in the fully expanded search tree. Thereby, it provides an estimate for the size of the search space.

[3]Gobase, www.gobase.org.

3.2.3 Algorithm and Implementation

PNS and various parameter settings for MC-PNS were implemented in a C++ framework. All experiments were conducted on a Linux workstation with AMD Opteron architecture and 2.8 GHz clock rate. 16 GB of working memory were available. Mobility was used to initialize PNS. No special pattern matcher or other feature detector was implemented to enhance the MCE or detect the life-and-death status of a position. Instead, tree search and MCE are carried out in a brute-force manner in the implementation. In the experiment, Zobrist hashing (Zobrist, 1990) was implemented to store proof and disproof numbers estimated by MCE. In order to save memory, the game positions are not stored in memory for any leaf. A complete game is played for each cycle. The proof tree is stored completely in memory.

The MCE used was based on the Go program MANGO (Chaslot *et al.*, 2008b). The program's Monte-Carlo engine provides a speed of up to 5,000 games per second for 19×19 Go with an average length of 120 moves on the hardware specified above. The Monte-Carlo engine was not specially designed for the local life-and-death task. Its speed and memory consumption must therefore be expected to perform suboptimally. The MCE was introduced in Subsection 3.1.1 as $eval_n$.

3.2.4 Test Procedure

Three independent parameters and three dependent variables describe the experiment. The independent parameters control the amount and behavior of MCE during search. The dependent variables measure the time and memory resources required to solve a problem by a specified configuration of the algorithm. The configuration is synonymously called parameter setting.

The independent parameters available for testing are: (1) the number of simulated games per evaluated node ($n \in \mathbb{N}$), (2) the look ahead ($la \in \mathbb{N}$), and (3) the launching depth level for MCEs ($depth \in \mathbb{N}$).

There are three dependent test variables measured: (1) the time spent for solving a problem measured in seconds ($time \in Time \subseteq \mathbb{R}$), (2) the number of nodes expanded during search ($nodes \in Nodes \subseteq \mathbb{N}$), and (3) a move ($move \in Moves = \{0, 1, ..., 361\}$) proven to reach the goal.[4] The set $Moves$ represents the range of possible intersections of the 19×19 Go board together with an additional null move (0). The null move is returned if no answer can be found in the time provided (cf. below). Because no other dynamic-memory cost is imposed, *nodes* suffices for calculating the amount of memory occupied. The memory consumption of a single node in the search tree is 288 bytes.

For the experiment, each configuration consisted of a triple of preset parameters $(n, la, depth)$. The following parameter ranges were applied: $n \in \{3, 5, 10, 20\}$, $la \in \{3, 5, 10\}$, $depth \in \{I, \frac{1}{2}I, \frac{3}{4}I\}$. The *depth* parameter requires additional explanation. In the experiment's implementation, the number of initially empty intersections I is employed to calculate a heuristic *depth* value. Thus I, $\frac{1}{2}I$, and $\frac{3}{4}I$ represent functions dependent on the specific instance of I given by each test case. We call these choices

[4] For three test problems, some parameter settings found a different forcing move first, i.e., there were two best first moves.

Rank	n	la	$depth$	g_t	g_s
1	3	10	3	2.05	4.26
2	5	10	3	1.93	4.54
3	3	10	2	1.87	3.90
4	5	10	1	1.80	4.59
5	3	10	1	1.75	4.30
6	5	10	2	1.74	4.11
7	3	5	3	1.56	2.70
8	3	5	1	1.51	2.70
9	10	10	3	1.42	4.99
10	10	10	1	1.36	4.95

Rank	n	la	$depth$	g_s	g_t
1	20	10	3	5.31	0.95
2	20	10	1	5.23	0.96
3	10	10	3	4.99	1.42
4	10	10	1	4.95	1.36
5	5	10	1	4.59	1.80
6	20	10	2	4.56	0.87
7	5	10	3	4.54	1.93
8	3	10	1	4.30	1.75
9	10	10	2	4.28	1.26
10	3	10	3	4.26	2.05

Table 3.1: Time and node consumption of various configurations of MC-PNS relative to PNS. Each table presents the ten best ranked of the 36 parameter settings. **Left:** Ordered by a factor representing the gain of speed (g_t). **Right:** Ordered by a factor representing the reduction of nodes.

of I the starting strategies and refer to them as 1, 2, and 3 for I, $\frac{1}{2}I$, and $\frac{3}{4}I$, respectively.

The outcome of an experiment is a triple of measured variables $(t, s, m) \in Time \times Nodes \times Moves$. Each experimental record consists of a configuration, a problem it is applied to, and an outcome. An experiment delivers a set of such records. In order to account for the randomness of the MCE and potential inaccuracy for measuring the small time spans well below a 10th of a second, PNS and each configuration of MC-PNS was applied to each test case 20 times resulting in 20 records.

3.3 Results of Experiment 1

This section outlines the results of Experiment 1. First, a statistical measure for comparing the algorithms is introduced, then the experimental results for different configurations are presented and described in detail. The section concludes by summarizing the results in six propositions.

The experiment consumed about 21 hours and produced 22,200 records. All solutions by PNS as well as by MC-PNS configurations were correct. Aggregates for each combination of test cases and configurations are considered in order to enable a comparison of different parameter settings. For a parameter setting $p = (n, la, depth)$ and a test case ϑ the average outcome is the triple $(t, s, m) \in Time \times Nodes \times Moves$. For this triple t, s, and m are averaged over all 20 records with the parameter setting p applied to the test case ϑ. $t_{(p,\vartheta)}$ is the average time and $s_{(p,\vartheta)}$ the average number of nodes expanded for solving ϑ. In order to make a parameter setting p comparable, its average result is compared to the average result of PNS. We define the gain of time as $g_t(p) = \frac{1}{30} \sum_{\vartheta=1}^{30} t_{(pns,\vartheta)}/t_{(p,\vartheta)}$. (Here, $t_{(pns,\vartheta)}$ is the average time consumed by PNS to solve test case ϑ.) The gain of space, g_s, is defined analogously. The positive real numbers g_t and g_s express the average gain of a parameter setting for all thirty test cases. Each gain value is a factor relevant to the performance of the PNS benchmark.

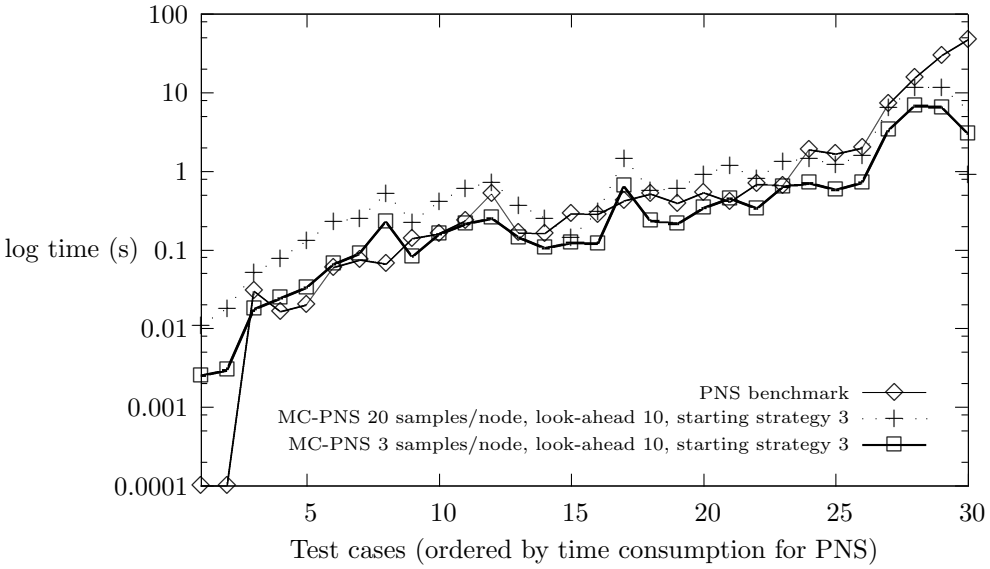


Figure 3.3: Time consumption of various configurations of MC-PNS.

In the experiment the configuration using 3 MCEs per node, a look-ahead of 10 moves, and starting strategy 3, is found to be the fastest configuration ($p_{fast} = (3, 10, 3)$). The $g_t(p_{fast}) = 2.05$ indicating that it is about twice as fast as the PNS benchmark (cf. Table 3.1, left and the definition of the gain of speed and gain of space). The gain $g_s(p_{fast}) = 4.26$. Thus p_{fast} expands fewer than a quarter of nodes which PNS expands. The parameter setting $p_{narrow} = (20, 10, 3)$ is expanding the smallest number of nodes. It requires less than a fifth of the expansions compared to the benchmark on average on the test set. It is slightly slower than PNS in spite of that (cf. Table 3.1).

Parameter settings with large n and large look-ahead require the least number of nodes to prove or disprove the goal. Parameter settings with small n but large look-ahead perform the fastest.

So far, this section focused on outlining the results relevant for characterizing the set consisting of PNS and MC-PNS variations. The remainder of this section describes the results required for comparing p_{fast}, p_{narrow}, and PNS in greater detail. For this purpose, the data hidden in the aggregates of the test cases are unfolded. This is achieved by comparing the average time and space performance for each test case. Figures 3.3 and 3.4 illustrate these comparisons, respectively.

A comparison of the time behavior (Figure 3.3) shows a pattern containing variety. Overall, p_{narrow} is characterized by the least time-efficient results: in 22 of the 30 test cases it is the slowest of the three compared solvers. PNS performs slowest on the remaining 8 test problems. But PNS also performs as the fastest solver in 6 cases and as fast as p_{fast} in 6 cases. p_{fast} shows the most efficient time consumption in 24 cases including the 6 cases in which it is as fast as PNS. We remark that the

two leftmost points which are plotted to require 0.0001 seconds actually have value 0 because they were measured beyond the precision of the time measurement function. It is reasonable to assume that the values should be ca. 0.001 seconds.

A comparison of the space behavior (Figure 3.4) shows that PNS requires the highest number of node expansions to prove or disprove a goal, irrespective of the test case. Its memory requirements are roughly the same as those of p_{fast} and p_{narrow} except for two test cases. PNS requires much more space than the two MC-PNS variations on virtually all other test cases. p_{fast} and p_{narrow} show a similar behavior in memory consumption with p_{narrow} performing slightly more efficient.

The results show that the performance depends on the complexity inherent to the tested problem. PNS finds its proofs faster than the other solvers on simple problems, i.e., problems requiring fewer than 5,000 nodes to be solved. It outperforms its competitors only once in the 20 more complex tasks while achieving this five times in the 10 least complex problems. The two MC-PNS variations perform comparably faster on the 20 most complex tasks. The experimental outcome shows that the speed advantage of p_{fast} relative to PNS grows with the complexity of the tested problem.

The main results of this section can be summarized in six propositions.

1. (3,10,3) is the fastest parameter configuration. On average it performs two times faster than PNS on the test set and expands fewer than a quarter of the nodes.

2. The configuration (20,10,3) is the MC-PNS configuration with the least node expansions on average. It expands only a fifth of the number of nodes expanded by PNS.

3. p_{fast} and p_{narrow} consistently expand considerably fewer nodes than PNS.

4. p_{fast} performs consistently faster than PNS.

5. The advantage of time performance of p_{fast} relative to PNS even grows with the complexity of the problem.

6. PNS performs better than p_{fast} on problems with small complexity.

3.4 Discussion of Experiment 1

On average p_{fast} solves problems twice as fast as PNS. This is coherent with a general tendency observed. As outlined above, settings with small n (i.e., few simulated games) and large la (i.e., far look-ahead) are generally the fastest. The speed of a MC-PNS depends mainly on two items: (1) the number of nodes it expands, and (2) the CPU time needed to evaluate a node. These two items are mutually dependent. The number of expansions decreases with the intensity of the evaluation because the heuristic is more reliable and the search investigates fewer nodes. This is reflected by the experimental finding that nodes with thorough evaluation (large n and large la) reach their goals with few expansions (cf. Table 3.1). But more intensive evaluations require more time. Therefore, an optimization problem has to

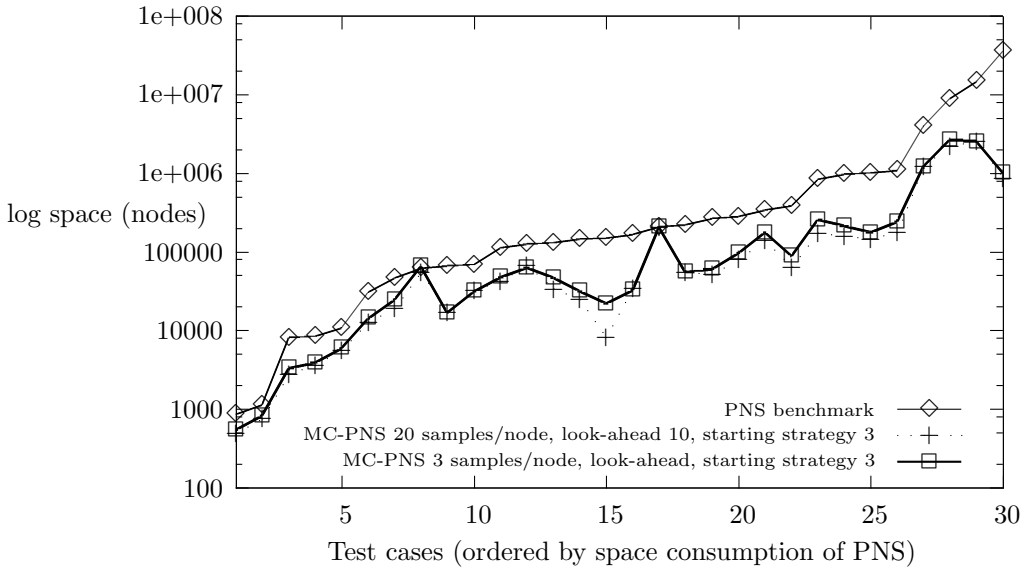


Figure 3.4: Space consumption of various configurations of MC-PNS.

be solved to compensate for the intensity of the evaluation. The optimum trades off between the number of nodes visited and the evaluation time for each single node. This optimum is found to be $p_{fast} = (3, 10, 3)$. Extensively evaluating each node, as pursued by p_{narrow}, devotes too much time on each single evaluation. The strategy of omitting a heuristic evaluation entirely, as embodied by PNS, is cheap for each node but generates larger search trees.

Thus the n and la parameters can be said to control the intensity of each evaluation successfully. The *depth* parameter has a minor influence on controlling this intensity. More importantly, only few MCEs per node can produce a reasonable heuristic for the MC-PNS framework.

One might object that the composition of the set is arbitrary. Still, we may argue it is reasonable to assume that the inferences made about the quality of p_{fast} are valid in general. The test cases chosen are rather easy. One may therefore expect that real-time problems are harder than the cases presented. Thus in practice p_{fast} should be even more relevant. The absolute time saved by p_{fast} is much larger for complex problems. For instance, PNS requires 47.7 seconds to solve the most complex problem whereas p_{fast} solves the problem in fewer than 6 seconds (eight times faster). All problems which require fewer than 10,000 nodes were solved in less than 0.1 second by PNS and p_{fast}. The absolute time saved is crucial for real-time applications, e.g., in a Go program. Thus we may conclude that it is valid to generalize our finding that p_{fast} is performing faster than PNS beyond our test set.

3.5 Patterns for PNS

As an alternative to MCE, we could use Go patterns to initialize proof and disproof numbers in the evaluation step of PNS (cf. Subsection 3.1.1). Moreover, we could use patterns to improve the MCE of MC-PNS. This section describes patterns in computer Go and in particular 3×3 patterns (Subsection 3.5.1). They are used in Experiment 2 as a heuristic for initializing the values of leaves in PNS and also for altering the probability distribution of MCE in MC-PNS (Subsection 3.5.2).

3.5.1 Patterns in Computer Go

Patterns are a standard means for representing knowledge in computer Go (e.g., Bouzy and Cazenave, 2001; Bouzy and Chaslot, 2006). A pattern in computer Go is a configuration of intersections. High-level representations contain features additional to the coloring of the intersections. Two examples are: (1) tactical information on connectivity and (2) life-and-death status. The size of a pattern varies considerably depending on the application's purpose. Large-scaled patterns are employed for openings, while tactical analysis is often guided by small patterns. Patterns can be created manually or auto-generated (Graepel *et al.*, 2001; Van der Werf, 2004; Bouzy and Chaslot, 2006).

The patterns applied in Experiment 2 are auto-generated 3×3 patterns. They are small, but offer the advantage of low pattern-matching cost. This characteristic is a precondition to any application in the Monte-Carlo framework as given by MC-PNS. The patterns should match each move in each simulated game. The patterns are generated by statistical ranking in self-play as described by Bouzy and Chaslot (2006). Each of the 6,561 patterns describes the desirability of the move in the center expressed by an integer value. The patterns do not account for information on the edge of the Go board. They were generated for a MCE Go engine for the whole game and are not specifically tailored to life-and-death problems.

3.5.2 Two Pattern-Based Heuristics

The patterns are the ingredients for two heuristic variations of PNS. We refer to them as (1) PNS$_p$ and (2) MC-PNS$_p$. As a consequence, we compare four solving algorithms in total: (i) PNS, (ii) PNS$_p$, (iii) MC-PNS, and (iv) MC-PNS$_p$.

Ad (1). PNS$_p$ applies the patterns to a node X by assigning the suggested pattern value of the last played intersection to $pn(\mathsf{X})$ and $dn(\mathsf{X})$. The desirability depends on the neighbors surrounding the intersection last played. The intersection's pattern value v ranges between 0 and 340. The values of pn and dn are set as $pn(\mathsf{X}) = \frac{1}{v+1}$ and $dn(\mathsf{X}) = \frac{v}{340}$. Local knowledge on the desirability of the move last played is thus taken into account.

Ad (2). MC-PNS$_p$ is a variation of MC-PNS. The patterns are employed to alter the probability distribution of the moves selected for the random sequence. Pure MC-PNS plays randomly distributed legal moves in each random sequence implying a uniform distribution of the probability for selecting a move. In MC-PNS$_p$ this uniform distribution is altered by the pattern values. Whenever a move is played

in a random sequence, the pattern values of the intersections next to that move are updated. These neighbors' new pattern values are set according to a matching pattern. The probability to play at a neighbor is proportional to its pattern value. Thus, moves evaluated well by the patterns are more likely to be played than moves less promisingly assessed.

The intuition underlying both heuristics is that main lines of play should be considered first. These lines are given preference by the pattern evaluation. Comparing the two heuristics, we may predict that MC-PNS$_p$ requires considerably more time for pattern matching because patterns are matched for each move in each random sequence. PNS$_p$ matches the patterns only once at each new node evaluation.

3.6 Experiment 2: Initialization by Patterns or by Monte-Carlo Evaluation

The four variations of the PNS algorithm outlined in the previous section (Section 3.5) are tested in Experiment 2: (1) PNS (without heuristics), (2) PNS$_p$ (PNS with pattern heuristic), (3) MC-PNS, and (4) MC-PNS$_p$ (MC-PNS with pattern heuristic). Each variation was tested on the test set of 30 life-and-death problems described in Subsection 3.2.2. The time spent and the number of nodes expanded for solving each test position were measured for each variation. The number of nodes is an indication of efficiency. Simultaneously, the memory consumption is measured by the same number of nodes. In order to account for the randomness of the MCEs and potential inaccuracy for measuring the small time spans well below a 10th of a second, each configuration was applied to each test case 20 times. The same test problems and machinery described for Experiment 1 are used.

3.7 Results of Experiment 2

The experiment required about six hours to complete. All variations solved all problems correctly. The variations that solved the test cases in decreasing order of average speed were as follows: first MC-PNS, second MC-PNS$_p$, third PNS, and fourth PNS$_p$ which required most time to solve the problems. PNS$_p$ was 3% slower than PNS, while MC-PNS was ca. two times faster than PNS. With respect to nodes expanded on average, MC-PNS and MC-PNS$_p$ performed equal and better than the other variants; PNS$_p$ performed third best and PNS performed least. The effect of the patterns was small: PNS$_p$ expanded about only 6% fewer nodes than PNS in total, while MC-PNS expanded about 75% fewer nodes than PNS. MC-PNS$_p$ was about as slow as PNS but used as little space as MC-PNS.

In order to avoid a strong influence of a few large test positions, we suggest an additional metric for comparing the results. For each test case, a ranking can be established for each variation. Each variation then ranks either first, second, third, or fourth in terms of speed performance. Similarly there is a rank for space consumption. The average ranking is introduced to compare the performance of the variations on all tests. The average rankings for finding a solution are shown in Table 3.2.

Algorithm	Time rank	Space rank
MC-PNS	1.6	1.6
MC-PNS$_p$	2.6	1.6
PNS	2.7	3.6
PNS$_p$	3.0	3.1

Table 3.2: Time and space ranking of the four compared PNS variations averaged over 30 test problems.

The average ranking by time consumption leads to roughly the same order as the ranking by space consumption, with one significant exception: PNS solves the problems faster than PNS$_p$ on average (in 17 out of 30 cases), but PNS consumes more space than PNS$_p$ (in two thirds of the cases). However, it should be remarked that there are two large outliers in the experimental data.

The results can be structured by noting the explicit occurrence of two features: (1) the Monte-Carlo feature and (2) the pattern feature. The Monte-Carlo feature is present in MC-PNS and MC-PNS$_p$; the pattern feature is present in PNS$_p$ and MC-PNS$_p$.

The speed of MC-PNS variations was about twice as high as for PNS on average. Patterns slowed down the speed for finding a solution in both PNS and MC-PNS, whenever they were applied. This tendency was particularly noticeable in MC-PNS$_p$. The variations relying on patterns expanded on average about 6% fewer nodes for each solution but the variation between the feature groups was larger than within each feature group. Analogously, the variation of time consumption is smaller between the feature groups than within. The internal variation of time consumption is larger within the Monte-Carlo-feature group than within the pattern-feature group. The reverse holds for space consumption.

3.8 Discussion of Experiment 2

The findings of the previous section can be summarized in three observations. (1) The Monte-Carlo evaluation leads to a better speed improvement than evaluation by patterns. (2) The patterns slow down the speed of the solver and particularly hinder it in case of MC-PNS$_p$. (3) There is a positive effect of the pattern heuristic, evident in the better space ranking of PNS$_p$ compared to that of PNS.

These three observations can be explained as follows. (1) The patterns' positive effect is outweighed by the cost of matching they impose. (2) While the advantage can be seen in reduced space consumption, this effect is too weak to result in time savings. (3) The outliers can be explained by the nature of pattern knowledge. Patterns are static and generalize from common cases. They are therefore potentially weak at handling exceptions. The effect is particularly strong because the patterns are not specialized on life-and-death problems (cf. Subsection 3.5.1). So, the combination of patterns and MCE in the manner proposed does not seem promising. However, overall the results, though not practically applicable in the current state,

can still be seen as encouraging for future work since opportunities for improvement exist (cf. Subsection 3.9.2).

3.9 Chapter Conclusion and Future Research

We conclude this chapter by summarizing the results of Experiment 1 and Experiment 2 (Subsection 3.9.1) and give an outlook on future research (Subsection 3.9.2) based on the findings presented.

3.9.1 Chapter Conclusion

This chapter introduced a new algorithm, MC-PNS, based on MCE within the PNS framework. An experimental application of the new algorithm and several variations to the life-and-death sub-problem of Go were described; moreover, its interpretation was presented. It was demonstrated experimentally that given the right setting of parameters MC-PNS outperforms PNS. For such a setting, MC-PNS is on average two times faster than PNS and expands four times fewer nodes. Experiment 1 resulted in evidence for assuming that this result will be generalized beyond the test cases observed. Experiment 2 compared MC-PNS with another pattern-based heuristic for initialization called PNS_p. We observed that MCE initializes proof and disproof numbers of leaves in PNS better than small patterns do. We speculate that the reason for the superior performance of MC-PNS is the flexibility of the MCE; the patterns are a generalization over many game positions and therefore too static to perform on the same level as MCE.

We may conclude that MC-PNS constitutes a genuine improvement of PNS that uses MCE to enhance the initialization of proof and disproof numbers. Moreover, we found that the patterns did not improve MC-PNS. Finally, MC-PNS, although it is a general search algorithm, requires a domain in which MCE can be implemented efficiently. We note that, still, MCE is less domain specific than patterns are.

3.9.2 Future Research

We propose three directions of future research related to our findings presented in this chapter.

(1) The proposed algorithm, MC-PNS, performed well in the domain of Go in which MCE proved to be useful. Future research should investigate the feasibility of the approach in other solving scenarios which have binary goals and can be searched by AND/OR trees. In particular, it would be of great interest to investigate whether MC-PNS could be successfully applied to one-player games or multi-player games with more than two players.

(2) The here proposed pattern enhancement of MC-PNS did not contribute to further improving MC-PNS. This observation does not rule out the possibility that MC-PNS can be combined with other initialization or move-selection heuristics. As a more general question, we can ask how different heuristics for PNS and even other best-first search algorithms can be combined with MCE.

(3) One reason pointed out for the relatively weak performance of the patterns lies in the kind of patterns used in Experiment 2. As remarked earlier, the patterns were originally designed for improving the playing strength of a Monte-Carlo program for the whole game of Go. More specific patterns for life-and-death problems could potentially provide better results. For instance, the patterns could be larger and thus take into account more context.

Chapter 4

Monte-Carlo Tree Search Solver

Sections 4.1 and 4.2 of this chapter are based on the following publication.[1]

- M.H.M. Winands, Y. Björnsson, and J-T. Saito. Monte-Carlo Tree Search Solver. In H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands, editors, *Proceedings of Computers and Games 2008 Conference (CG'08)*, volume 5131 of *Lecture Notes in Computer Science*, pp. 25-36. Springer, Berlin, Germany, 2008.

In Chapter 2 we distinguished between two Monte-Carlo methods: (1) Monte-Carlo Evaluation (MCE) (Abramson, 1990), and (2) Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007a). Chapter 3 proposed an approach for using MCE for solving game positions. In this chapter we investigate, in how far MCTS can be used for solving.

In the course of the last few years, MCTS has advanced the field of computer Go substantially. Moreover, it is used for other games such as Phantom Go (Cazenave and Borsboom, 2007) and Clobber (Kocsis, Szepesvári, and Willemson, 2006), and even for games for which reasonable evaluation functions have been known, e.g., Amazons (Lorentz, 2008; Kloetzer, Müller, and Bouzy, 2008), LOA (Winands and Björnsson, 2010), and Hex (Cazenave and Saffidine, 2009). Although MCTS is able to find the best move, it is not able to prove the game-theoretic value of (parts of) the search tree. A search method that is not able to prove or estimate quickly the game-theoretic value of a node may run into problems. This is especially true for sudden-death games, such as Chess, that may abruptly end by the creation of one of a prespecified set of patterns (Allis, 1994) (e.g., checkmate in Chess). In this case $\alpha\beta$ search or Proof-Number Search (PNS) (Allis *et al.*, 1994) is traditionally preferred to MCTS.

This chapter answers the second research question, **RQ 2**: *How can the Monte-Carlo Tree Search framework contribute to solving game positions?* To that end, we

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

introduce a new MCTS variant called MCTS-Solver that has been designed to prove the game-theoretic value of a node in a search tree. This is an important step towards applying MCTS-based approaches effectively in sudden-death-like games (including Chess). We use the game Lines of Action (LOA) as a testbed. It is an ideal candidate because its intricacies are less complicated than those of Chess. Therefore, we can focus on the sudden-death property. In this chapter we assess MCTS-Solver on a test set of LOA positions. First, we test the influence of the selection strategy on MCTS-Solver. Second, we compare MCTS-Solver with $\alpha\beta$ and PN^2. Third, root parallelization and tree parallelization are evaluated for MCTS-Solver. We note that we investigate the *solving strength* of MCTS-Solver; its improvement in *playing strength* was already investigated by Winands *et al.* (2008). Experimental results indicated that MCTS-Solver defeated the original MCTS program by a winning score of 65%.

The chapter is organized as follows. In Section 4.1 we introduce MCTS-Solver. In Section 4.2 we discuss its application to LOA. We empirically evaluate the method in Section 4.3. Finally, Section 4.4 gives conclusions and an outlook on future research.

4.1 Monte-Carlo Tree Search Solver

Although MCTS is unable to *prove* the game-theoretic value, in the long run MCTS equipped with the UCT formula is able to *converge* to the game-theoretic value. For example, in endgame positions of fixed-termination games like Go and Amazons, MCTS is able to find the optimal move relatively fast (Zhang and Chen, 2007; Kloetzer *et al.*, 2008). However, in a sudden-death game like LOA, where the main line towards the winning position is narrow, MCTS may often lead to an erroneous outcome because the nodes' values in the tree do not converge fast enough to their game-theoretic value. For example, if we let MCTS analyze the LOA position in Figure 4.1 for 5 seconds, it selects **c7xc4** as the best move, winning 67.2% of the simulations. However, this move is a forced 8-ply loss, while **f8-f7** (scoring 48.2%) is a 7-ply win.[2] Only when we let MCTS search for 60 seconds, it selects the optimal move. For a reference, we remark that it takes $\alpha\beta$ in this position less than a second to select the best move and prove the win.

We designed a new variant called MCTS-Solver which is able to prove the game-theoretic value of a position. Of the four stages of MCTS (described in Chapter 2) two are changed for MCTS-Solver. The simulation and expansion stages remain the same. The backpropagation and the selection are modified for MCTS-Solver. The changes are discussed in Subsections 4.1.1 and 4.1.2, respectively. The pseudocode of MCTS-Solver is given in Subsection 4.1.3.

[2]The rules of LOA can be found in Appendix A.

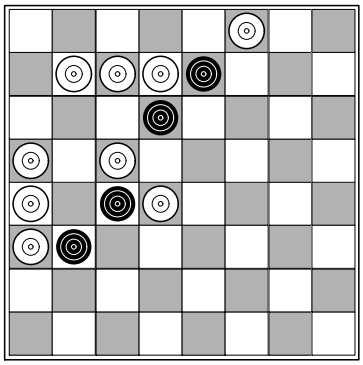


Figure 4.1: LOA position with White to move.

4.1.1 Backpropagation

In addition to backpropagating the values {1,0,−1}, the search also propagates the game-theoretic values ∞ or $-\infty$.[3] The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax (Knuth and Moore, 1975) in the context of minimax search in such a way that we do not have to distinguish between MIN and MAX nodes. If the selected move (child) of a node returns ∞, the node is a win. To prove that a node is a win, it suffices to prove that one child of that node is a win. In the case that the selected child of a node returns $-\infty$, all its siblings have to be checked. If their values are also $-\infty$, the node is a loss. To prove that a node is a loss, we must prove that all its children lead to a loss. In the case that one or more siblings of the node have a different value, we cannot prove the loss. Therefore, we will propagate -1, the result for a lost game, instead of $-\infty$, the game-theoretic value of a position. We note that the backpropagation of game-theoretic values is similar to the backpropagation of another solving algorithm, namely PNS. The value of the node will be updated according to the backpropagation strategy as described in Subsection 2.3.2.

4.1.2 Selection

As seen in the previous subsection, a node can have the game-theoretic value ∞ or $-\infty$. The question arises how these game-theoretic values affect the selection strategy. Of course, when a child is a proven win, the node itself is a proven win, and no selection has to take place. But when one or more children are proven to be a loss, it is tempting to discard them in the selection phase. However, this can lead to overestimating the value of a node, especially when moves are pseudo-randomly selected by the simulation strategy. For example, in Figure 4.2 we have

[3]Draws are in general more problematic to prove than wins and losses. Because draws only happen in exceptional cases in our test domain LOA, we took the decision not to handle proven draws separately; in this way we maintain efficiency.

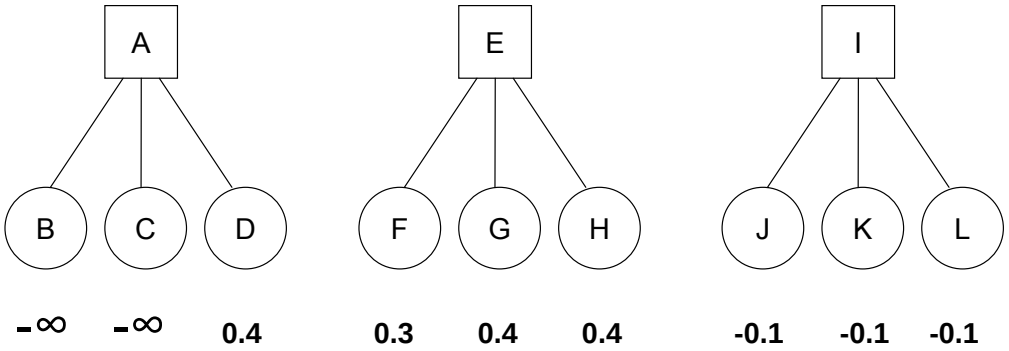


Figure 4.2: Search trees showing a weakness of Monte-Carlo evaluation.

three one-ply subtrees. Leaf nodes B and C are proven to be a loss, indicated by $-\infty$; the numbers below the other leaves are the *expected* pay-off values. Assume that we select the moves with the same likelihood (as could happen when a simulation strategy is applied). If we would prune the loss nodes, we would prefer node A to E. The average of A would be 0.4 and that of E 0.37. It is easy to see that A is overestimated because E has more good moves.

If we do not prune proven loss nodes, we run the risk of underestimation. Especially, when we have a strong preference for certain moves (because of a bias) or we would like to explore our options (because of the UCT formula), we could underestimate positions. Assume that we have a strong preference for the first move in the subtrees of Figure 4.2. We would prefer node I to A. It is easy to see that A is underestimated because I has no good moves at all.

Based on preliminary experiments (cf. Winands *et al.*, 2008), selection is performed in the following way. In case a selection strategy such as UCT is applied, child nodes with the value $-\infty$ will never be selected. For nodes of which the visit count is below a certain threshold, moves are selected according to the simulation strategy instead of selection strategy. In that case, children with the value $-\infty$ *can* be selected. However, when a child with a value $-\infty$ is selected, the search is not continued at that point. The results are propagated backwards according to the strategy described in the previous subsection.

For a node N that has not been visited yet, we test whether one of its moves leads directly to a win or all moves are losses at N. If there is such a move, we stop searching at this node and set the node's value. This check at node N must be performed because otherwise it could take many simulations before the child leading to a mate-in-one is selected and the N is proven.

4.1.3 Pseudocode for MCTS-Solver

The pseudocode for MCTS-Solver is listed in Algorithm 4.1. The algorithm is constructed similar to negamax in the context of minimax search. `select(Node N)` is the selection function as discussed in Subsection 4.1.2, which returns the best child

of the node N. The procedure `add_to_tree(Node N)` adds one more node to the tree; `play_out(Node N)` is the function which plays a simulated game from the node N, and returns the result $R \in \{1, 0, -1\}$ of this game; `compute_average(Integer R)` is the procedure that updates the value of the node depending on the result R of the last simulated game; `get_children(Node N)` generates the children of node N.

4.2 Monte-Carlo LOA

In this section we discuss how we applied MCTS-Solver to LOA. First, we propose possible selection strategies for MCTS-Solver in Subsection 4.2.1. We explain the simulation strategy in Subsection 4.2.1. Finally, we discuss how we parallelized the search in Subsection 4.2.3.

4.2.1 Selection Strategies

We use the UCT (**U**pper **C**onfidence bounds applied to **T**rees) strategy (Kocsis and Szepesvári, 2006). UCT is easy to implement and used in many Monte-Carlo Go programs. UCT can be enhanced with Progressive Bias (PB) (Chaslot *et al.*, 2008b). PB is a technique to embed domain-knowledge bias into the UCT formula. It was successfully applied in the Go program MANGO (Chaslot *et al.*, 2008b). UCT with PB works as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the child k of the node p that satisfies Formula 4.1:

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} + \frac{W \times P_{mc}}{n_i + 1} \right) \quad , \tag{4.1}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient which must be tuned experimentally. $\frac{W \times P_{mc}}{n_i + 1}$ is the PB part of the formula. W is a constant which must be set manually (here $W = 50$). P_{mc} is the *transition probability* of a move category mc (Tsuruoka, Yokoyama, and Chikayama, 2002).

For each move category (e.g., capture, blocking) the probability that a move belonging to that category will be played is determined. The probability is called the *transition probability*. This statistic is obtained off-line from game records of matches played by expert players. The transition probability for a move category mc is calculated as follows:

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \quad , \tag{4.2}$$

Algorithm 4.1: Pseudocode for MCTS-Solver.

```
1:  procedure MCTSSolver(Node N)                    ▷ search one iteration in N
2:                                                                ▷ return its value
3:      if N.player_to_move_wins() then return  ∞
4:      else
5:          if N.player_to_move_loses() then return  −∞
6:          end if
7:      end if
8:
9:      B = select(N)                                    ▷ B is best child of N
10:
11:     if B.value != ∞ AND B.value != −∞  then
12:         if B.visit_count == 0 then
13:             R = -play_out(B)                         ▷ R is result of this call
14:             add_to_tree(B)
15:             goto DONE
16:         else
17:             R = -MCTSSolver(B)
18:         end if
19:     else
20:         R = B.value
21:     end if
22:
23:     if R == ∞  then
24:         N.value = −∞
25:         return R
26:     else
27:         if R == −∞  then
28:             for C in N.children do
29:                 if C.value != R then
30:                     R = -1
31:                     goto DONE
32:                 end if
33:             end for
34:         end if
35:         N.value = ∞
36:         return R
37:     end if
38:
39:     DONE:
40:     N.compute_average(R)                 ▷ also increases N.visit_count
41:     return R
42:
43: end procedure
```

where $n_{played(mc)}$ is the number of game positions in which a move belonging to category mc was played, and $n_{available(mc)}$ is the number of positions in which moves belonging to category mc were available.

The move categories of the MC-LOA program are similar to the ones used in the Realization-Probability Search of the program MIA (Winands and Björnsson, 2008). They are applied in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim, and the central 2×2 board. Finally, moves are further classified based on the number of squares traveled away from or towards the center-of-mass.

The influence of PB has to be important when a few games have been played but has to decrease fast (when more games have been played) to ensure that the strategy converges to the UCT selection strategy. Standard, this is done through dividing the PB component by the number of games played through that node (i.e., n_i). A disadvantage is that by dividing by n_i the decrease rate of PB is independent of the actual value of a child (i.e., v_i). Nodes that do not perform well should not be biased that long, whereas nodes that continue to have a high score should continue to be biased. Instead of dividing the PB component by the number of games played, it may be divided by the number of losses l_i (Nijssen and Winands, 2010) in Formula 4.3.

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} + \frac{W \times P_{mc}}{l_i + 1} \right) \ , \tag{4.3}$$

Moreover, favoring nodes with a high value could be taken a step further by replacing every n_i with l_i which would produce Formula 4.4.

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{l_i}} + \frac{W \times P_{mc}}{l_i + 1} \right) \ , \tag{4.4}$$

All three PB variants are available in the MCTS program. In Section 4.3, we will determine which PB variant works the best for *solving positions*. Regarding *playing strength*, applying the standard PB enables that the MC-LOA program wins almost 75% of the games against the standard UCT. The two enhanced variants only gave a slight additional improvement.

Finally, we remark that the selection strategy is applied only at nodes with a visit count higher than a certain threshold T (here 5) (Coulom, 2007a). If the node has been visited fewer times than this threshold, the next move is selected according to the *simulation strategy* discussed in the next subsection.

4.2.2 Simulation Strategy

In the Monte-Carlo LOA program, the move categories together with their transition probabilities, as discussed in Subsection 4.2.1, are used to select the moves pseudo-randomly during the MCE. The simulation strategy draws the moves randomly based on their transition probabilities in the first part of a simulation, but selects them based on their evaluation score in the second part of a simulation (cf. Winands and Björnsson, 2010).

A simulation requires that the number of moves per game is limited. When considering the game of LOA, the simulated game is stopped after 200 moves and scored as a draw. The game is also stopped when heuristic knowledge indicates that the game is probably over. In general, once a LOA position gets very lopsided, an evaluation function can return a quite trustworthy score, more so than even elaborate simulation strategies. The game can thus be safely terminated both earlier and with a more accurate score than if continuing the simulation (which might fail to deliver the win). We use the MIA 4.5 evaluation function (Winands and Van den Herik, 2006) for this purpose. When the evaluation function gives a value that exceeds a certain threshold (i.e., 700 points), the game is scored as a win. If the evaluation function gives a value that is below a certain threshold (i.e., -700 points), the game is scored as a loss. For efficiency reasons the evaluation function is called only every 3 plies, starting at the second ply (thus at 2, 5, 8, 11 etc.).

4.2.3 Parallelization

The MC-LOA program employs two parallelization methods, (1) root parallelization and (2) tree parallelization. They are explained below.

Root Parallelization

"Single-run" parallelization (Cazenave and Jouandeau, 2007), also called *root parallelization* (Chaslot *et al.*, 2008a) consists of building multiple MCTS trees in parallel, with one thread per tree. These threads do not share information with each other. When the available time is up, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. Based on this grand total, the best move is selected. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy to implement even on a cluster. For a small number of threads, root parallelization performs remarkably well in comparison to other parallelization methods (Cazenave and Jouandeau, 2007; Chaslot *et al.*, 2008a). Root parallelization improved the playing strength of the MC-LOA program considerably (Winands and Björnsson, 2010).

Tree Parallelization

Tree parallelization (Cazenave and Jouandeau, 2007; Chaslot *et al.*, 2008a) uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree; therefore, care needs to be taken to

prevent that threads corrupt the values of the nodes by simultaneously operating on the same data. A lock is used when a leaf is expanded to prevent corruption of data. Inspired by the suggestion of Enzenberger and Müller (2010), no locks are used when two threads access the same internal node. At the moment that two threads try to update a node at the same time, one of the updates may be lost. In our implementation, a special check takes care that the game-theoretic value of a proven node is never overwritten.

If several threads start from the root at the same time, it is possible that they traverse the tree for a large part along the same path. Simulated games might start from leaves close to each other. It may even occur that simulated games begin from the same leaf node. Because a search tree typically has millions of nodes, it may be redundant to explore a rather small part of the tree several times. Instead of selecting the child solely on its UCT score, we added a small random factor to the UCT formulas. Shoham and Toledo (2002) call this the *randomization* of the move selection. Randomization of the move selection has been shown to be beneficial for parallelizing best-first search in the past (cf. Shoham and Toledo, 2002; Saito, Winands, and Van den Herik, 2010, and Chapter 5 of this thesis). Adding a random factor to Formula 4.4 produces Formula 4.5.

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{l_i}} + \frac{W \times P_{mc}}{l_i + 1} + k \times \epsilon \right) \quad , \tag{4.5}$$

where ϵ is a random variable, $\epsilon \in [0, 1]$ and k a constant. This constant is here set to 0.02, to ensure that playing strength is at least the same as when applying root parallelization.

4.3 Experiments

In this section we evaluate the tactical performance of MCTS-Solver by testing it on LOA endgame positions. First, we explain briefly the experimental setup in Subsection 4.3.1. Next, we test the influence of the selection strategy on MCTS-Solver in Subsection 4.3.2. Subsequently, we compare MCTS-Solver with $\alpha\beta$ and PN^2 search in Subsection 4.3.3. Finally, root parallelization and tree parallelization are evaluated for MCTS-Solver in 4.3.4.

4.3.1 Experimental Setup

We tested MCTS-Solver on 488 endgame positions of Lines of Action (LOA).[4] This test set and subsets thereof have been used frequently in the past (Pawlewicz and Lew, 2007; Sakuta *et al.*, 2003; Winands *et al.*, 2003b). We note that the rules of LOA can be found in the Appendix A. All experiments are carried out on a Linux server with eight 2.66 GHz Xeon cores and 8 GB of RAM. In this section

[4]The test set is available at www.personeel.unimaas.nl/m-winands/loa/tscg2002a.zip.

MCTS-Solver is compared with $\alpha\beta$ and PN^2 search. The details of $\alpha\beta$ and PN^2 are discussed below. All search algorithms have been implemented in Java.

$\alpha\beta$ Search

The $\alpha\beta$ search performs a depth-first iterative-deepening in the PVS / NegaScout framework (Marsland, 1983; Reinefeld, 1983). The evaluation function of MIA 4.5 is used (Winands and Van den Herik, 2006) to assess the values of leaf nodes. A TwoDeep transposition table (Breuker *et al.*, 1996) is applied to prune a subtree or to narrow the $\alpha\beta$ window. At all interior nodes that are more than 2 plies away from the leaves, it generates all moves to perform Enhanced Transposition Cutoffs (ETC) (Schaeffer and Plaat, 1996). For move ordering, the move stored in the transposition table (if applicable) is always tried first, followed by two killer moves (Akl and Newborn, 1977). These are the last two moves that were best or at least caused a cutoff at the given depth. Thereafter follow: (1) capture moves going to the inner area (the central 4×4 board) and (2) capture moves going to the middle area (the 6×6 rim). All the remaining moves are ordered decreasingly according to the relative history heuristic (Winands *et al.*, 2006). At the leaf nodes of the regular search, a quiescence search is performed to get more accurate evaluations (cf. Schrüfer, 1989).

PN^2 Search

The PN^2 search used, was previously employed to solve 6×6 LOA (cf. Winands, 2008). In this implementation, the proof number and disproof number are initialized to values 1 and n, respectively, for an OR node (and the reverse for an AND node), where n is the number of legal moves. In LOA, n is equivalent to the *mobility* of the moving player. It is also an important feature in the evaluation function (Winands, Van den Herik, and Uiterwijk, 2003a). It is known that mobility speeds up PN with a factor of 5 to 6 (Winands, 2004).

As a reminder, we note that PN^2 consists of two levels of PN search. The first level consists of a PN search (PN$_1$) which calls a PN search at the second level (PN$_2$) for an evaluation of the most-proving node of the PN$_1$ tree. This PN$_2$ search is bound by a maximum number of nodes N to be stored in memory. In this implementation, N is equal to the size of the PN$_1$ tree (Allis, 1994). The PN$_2$ search is stopped when the number of nodes stored in memory exceeds N or the subtree is (dis)proven. After completion of the PN$_2$ search, the children of the root of the PN$_2$ tree are preserved, but subtrees are removed from memory.

4.3.2 Selection Strategies

In the first experiment we test the effect of enhancing MCTS-Solver with Progressive Bias (PB) on a set of 488 forced-win LOA positions. We compare the standard UCT selection strategy with the three PB variants, here called Classic PB, PB-L1, and PB-L2 (i.e., Formulas 4.1, 4.3 and 4.4, respectively). We will look how much effort (in nodes and CPU time) it takes to solve endgame positions. For MCTS-Solver, all children at a leaf node, evaluated for the termination condition during the search

(cf. Subsection 4.1.2), are counted. The maximum number of nodes searched is 5,000,000. The limit ensures that MCTS-Solver does not use more than 30 seconds for trying to solve a position.

The results are given in Table 4.1. The first column gives the names of the algorithms, the second the number of positions solved, and the third and fourth the number of nodes searched and the time it took, respectively. In the second column we see that 174 positions are solved by UCT, 245 by Classic PB, 306 by PB-L1, and 319 by PB-L2. In the third and fourth column the number of nodes and the time consumed are given for the subset of 154 positions which all algorithms are able to solve. We see that the PB variants not only solve more problems, but also that they explore considerably smaller trees than the UCT variant. For the best PB variant, PB-L2, positions were solved tree times faster in nodes and time than UCT. We may conclude that PB not only improves the playing performance but also the solving performance.

	# positions solved	154 positions	
Algorithm	(out of 488)	Total nodes	Total time (ms.)
UCT	174	148,278,311	514,933
Classic PB	245	103,369,811	381,944
PB-L1	306	57,010,037	212,437
PB-L2	319	45,444,437	171,246

Table 4.1: Comparing different selection strategies on 488 test positions with a limit of 5,000,000 nodes.

For better insight into how much faster the enhanced PB variants, PB-L1 and PB-L2, are than the Classic PB in CPU time, a second comparison was performed. In Table 4.2 we compare Classic PB, PB-L1, and PB-L2 on the subset of 224 test positions which the three algorithms were able to solve. The table reveals that the PB-L1 and PB-L2 are 1.5 to 2 times faster than Classic-PB in CPU time. Taking into account the number of losses instead of the number of visits in the PB formula improves the solving strength of MCTS-Solver.

Algorithm	Total nodes	Total time (ms.)
Classic PB	217,975,808	799,893
PB-L1	138,742,124	497,161
PB-L2	124,635,541	448,170

Table 4.2: Comparing the different Progressive-Bias variants on 224 test positions with a limit of 5,000,000 nodes.

Finally, to assess which of the enhanced PB variants, PB-L1 and PB-L2, is the best, we performed a third comparison. In Table 4.3 we compare PB-L1 and PB-L2 on a subset of 286 test positions which both algorithms were able to solve. We can see that PB-L2 is 10 to 15% faster in nodes and CPU time than PB-L1. For the remaining experiments of this chapter we will use the PB-L2 enhancement.

Algorithm	Total nodes	Total time (ms.)
PB-L1	254,285,077	914,709
PB-L2	222,521,350	795,575

Table 4.3: Comparing PB-L1 and PB-L2 on 286 test positions with a limit of 5,000,000 nodes.

4.3.3 Comparing Different Solvers

In this section we test the tactical performance of MCTS-Solver by comparing it to $\alpha\beta$ search and PN^2. The goal is to investigate the effectiveness of the MCTS-Solver experimentally. We will measure how much effort (in nodes and CPU time) it takes to solve endgame positions. For the $\alpha\beta$ depth-first iterative-deepening search, nodes at depth i are counted only during the first iteration that the level is reached. This is in agreement with the way analogous comparisons have been carried out by Allis *et al.* (1994). For PN^2 all nodes evaluated for the termination condition during the search are counted. MCTS-Solver, PN^2, and $\alpha\beta$ are tested on the same set of 488 forced-win LOA positions. Two comparisons are made below.

Table 4.4 gives the results of MCTS-Solver, PN^2, and $\alpha\beta$ on the set of 488 LOA positions. The maximum number of nodes searched is again 5,000,000. In the second column of Table 4.4 we see that 319 positions were solved by the MCTS-Solver, 252 positions by $\alpha\beta$, and 405 positions by PN^2. In the third and fourth column the number of nodes and the time consumed are given for the subset of 218 positions which all three algorithms were able to solve. If we have a look at the fourth column, we see that PN^2 search is the fasted and that MCTS-Solver is the slowest, although its speed is not much slower that of $\alpha\beta$ search: PN^2 is (slightly more than) 5 times faster than MCTS-Solver whereas $\alpha\beta$ is only 1.3 times faster than MCTS-Solver.

Algorithm	# positions solved	218 positions	
	(out of 488)	Total nodes	Total time (ms.)
MCTS-Solver	319	153,863,729	497,859
$\alpha\beta$	252	217,800,566	355,061
PN^2	405	120,489,289	92,636

Table 4.4: Comparing MCTS-Solver, $\alpha\beta$, and PN^2 on 488 test positions with a limit of 5,000,000 nodes.

For a better insight into how much faster PN^2 is than MCTS-Solver in CPU time, we conducted a second comparison. In Table 4.5 we compare PN^2 and MCTS-Solver on the subset of 304 test positions which both algorithms were able to solve. Again, PN^2 solves positions approximately 5 times faster than MCTS-Solver.

Algorithm	Total nodes	Total time (ms.)
MCTS-Solver	274,602,402	948,124
PN^2	245,786,282	189,893

Table 4.5: Comparing MCTS-Solver and PN^2 on 304 test positions.

4.3.4 Testing Parallelized Monte-Carlo Tree Search Solver

In the next series of experiments we tested the performance of root parallelization for solving 319 positions which the regular MCTS-Solver (cf. Table 4.4) was able to solve. The results regarding time and nodes evaluated are given for 1, 2, 4, and 8 threads in Table 4.6. We remark that the root parallelization for 1 thread behaves exactly the same as the MCTS-Solver tested in the previous subsection. We observe that the scaling factors for 2, 4, and 8 threads, are only 1.1, 1.3, and 1.4, respectively. The results for root parallelization is therefore rather weak. The reason for the mediocre scaling is that this parallelization method builds substantially larger trees. For instance, in the case of 8 threads, the search explores 5.4 times more nodes than the standard MCTS-Solver.

Processors	1	2	4	8
Total time (ms.)	1,097,167	979,239	842,531	811,426
Total scaling factor	1	1.1	1.3	1.4
Total nodes evaluated	308,722,292	541,309,966	937,087,892	1,666,367,476

Table 4.6: Experimental results for MCTS-Solver with root parallelization on 319 test positions.

In the second series of experiments we tested the performance of tree parallelization by comparing it to the regular MCTS-Solver. We note that tree parallelization with one thread does not behave the same as MCTS-Solver caused by the random factor in the selection strategy. The results regarding time and nodes evaluated, on the subsets that both algorithms could solve for 1, 2, 4, and 8 threads are given in Table 4.6. For 1, 2, 4, and 8 threads these subsets consisted of 298, 314, 317, and 315 positions.

We observe that the scaling factor for 1, 2, 4, and 8 threads is 0.84, 1.4, 2.2 and 4.1, respectively. Compared to root parallelization the scaling factor of tree parallelization is better for configurations consisting of more than 1 thread. The tree parallelization with one thread spends approximately 20% more time than the standard MCTS-Solver. This slowing down is due to the randomization of the selection strategy. Moreover, we notice that the tree parallelization builds smaller trees than root parallelization. For instance, in the case of 8 threads, tree parallelization searches 1.3 times more nodes whereas root parallelization searches 5.4 times more nodes.

Processors	1	2	4	8
Solved positions	298	314	317	315
MCTS-Solver				
Total time (ms.)	860,599	1,040,594	1,074,670	1,049,238
Total nodes evaluated	242,905,355	295,828,149	303,362,399	298,530,220
Parallel MCTS-Solver				
Total time (ms.)	1,027,303	751,816	484,176	257,543
Total nodes evaluated	264,398,977	355,828,413	412,037,677	400,838,588
Total scaling factor	0.84	1.4	2.2	4.1

Table 4.7: Experimental results for MCTS-Solver with tree parallelization.

4.4 Chapter Conclusion and Future Research

We end this chapter by summarizing the results of the experiments (Subsection 4.4.1) and giving an outlook on future research (Subsection 4.4.2) based on the findings presented.

4.4.1 Chapter Conclusion

In this chapter we introduced a new MCTS variant, called MCTS-Solver. This variant differs from the traditional Monte-Carlo approaches in that it can solve positions by proving game-theoretic values. As a side effect it converges much faster to the best move in narrow tactical lines. This is especially important in tactical sudden-death-like games such as LOA. We discussed four selection strategies: UCT, "classic" Progressive Bias (PB), and two enhanced PB variants: PB-L1 and PB-L2. Experiments in LOA revealed that PB-L2 which takes the number of losses into account, solved the most positions and was the fastest (i.e., it solved positions in 3 times less time than UCT). In our comparison of MCTS-Solver and $\alpha\beta$, MCTS-Solver required about the same effort as $\alpha\beta$ to solve positions. However, PN^2 was in general 5 times faster than MCTS-Solver. Finally, we found empirically that tree parallelization for MCTS-Solver has a scaling factor of ca. 4 with 8 threads, easily outperforming root parallelization.

Two conclusions may be drawn from the work presented in this chapter. The first conclusion is that at least during game-play (online) MCTS-Solver is comparable with a standard $\alpha\beta$ search in solving positions. However, for offline solving positions, PNS is still a better choice. We remark that an $\alpha\beta$ search with variable-depth methods such as singular-extensions (Anantharaman, Campbell, and Hsu, 1988), null-move (Donninger, 1993), multi-cut (Björnsson and Marsland, 1999; Winands *et al.*, 2005) and Realization Probability Search (Tsuruoka *et al.*, 2002; Winands and Björnsson, 2008) could have given similar results. Although, there is no guarantee that they improve the $\alpha\beta$ search for a given domain.

The second conclusion we may draw is that the strength of MCTS-Solver is dependent on enhancements such as PB. Just as for $\alpha\beta$, search enhancements (cf. Marsland, 1986) are crucial for the performance.

4.4.2 Future Research

For search methods based on MCE, to be able to handle proven outcomes is an essential step. With continuing improvements it is not unlikely that in the not so distant future enhanced MCE-based approaches may even become an alternative to PNS. As future research, experiments are envisaged in other games to test the performance of MCTS-Solver. One possible next step would be to test the method in Go, a domain in which MCTS is already widely used. What makes this a somewhat more difficult task is that additional work is required in enabling perfect endgame knowledge – such as Benson's Algorithm (Benson, 1988; Van der Werf *et al.*, 2003) – in MCTS.

Chapter 5

Parallel Proof-Number Search

This chapter is based on the following publications.[1]

1. J-T. Saito, M.H.M. Winands and H.J. van den Herik. Randomized Parallel Proof-Number Search. In T. Calders, K. Tuyls, and M. Pechenizkiy, editors, *Proceedings of the 21st BeNeLux Conference on Artificial Intelligence (BNAIC'09)*, pp. 365–366, TU/e Eindhoven, Eindhoven, The Netherlands, 2009.
2. J-T. Saito, M.H.M. Winands and H.J. van den Herik. Randomized Parallel Proof-Number Search. In H.J. van den Herik and P. Spronck, editors, *Proceedings of the 13th Advances in Computer Games Conference (ACG'09)*, volume 6048 of *Lecture Notes in Computer Science*, pp. 75-87. Springer, Berlin, Germany, 2010.

A variety of parallel $\alpha\beta$ algorithms have been proposed in the past (cf. Brockington, 1996), but so far little research has been conducted on parallelizing PNS. With multi-core processors becoming established as standard equipment, parallelizing PNS has become an important topic. Pioneering research has been conducted by Kishimoto and Kotani (1999), who parallelized the depth-first PNS variant PDS. His algorithm, called ParaPDS, is designed for distributed memory systems.

This chapter addresses **RQ 3**: *How can Proof-Number Search be parallelized?* We parallelize PNS and PN^2 for shared-memory systems. The parallelization is based on randomizing the move selection of multiple threads, which operate on the same search tree. The new method is called Randomized Parallel Proof-Number Search (RP–PNS). Its PN^2 version is called RP–PN^2. We evaluate the new parallelization on a set of LOA problems.

The chapter is organized in four sections. Section 5.1 describes the parallelization of search algorithms in general in as far as it is relevant for this work. Moreover, the only other existing parallelization of PNS, ParaPDS, is explained. Section 5.2 introduces our new parallelization of PNS and PN^2, i.e., RP–PNS and RP–PN^2. In

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

Section 5.3 RP–PNS and RP–PN^2 are tested on a set of LOA endgame positions. Section 5.4 concludes this chapter and gives directions for future work.

5.1 Parallelization of PNS

This section introduces some basic concepts for describing the behavior of parallel search algorithms (Subsection 5.1.1), outlines ParaPDS, a parallelization of PDS (Subsection 5.1.2), and explains parallel randomized search (Subsection 5.1.3).

5.1.1 Terminology

Parallelization aims at reducing the time that a sequential algorithm requires for terminating successfully. The speedup is achieved by distributing computations to multiple threads executed in parallel.

Parallelization gains from dividing computation by multiple resources but simultaneously it may impose a computational cost. According to Brockington and Schaeffer (1997) three kinds of overhead may occur when parallelizing a search algorithm: (1) search overhead, resulting from extra search not performed by the sequential algorithm; (2) synchronization overhead, created at synchronization points when one thread is idle waiting for another thread; (3) communication overhead, created by exchanging information between threads.

Search overhead is straightforward and can be measured by the number of additional nodes searched. Synchronization and communication overhead depend on the kind of information sharing used. There are two kinds of information sharing: (i) message passing and (ii) shared memory. Message passing simply consists of passing information between memory units exclusively accessed by a particular thread. Under shared memory all threads can access a common part of memory. With the advent of multi-core CPUs memory sharing has become common place.

An important property governing the behavior of parallel algorithms is scaling. It describes the efficiency of parallelization with respect to the number of threads as a fractional relation t_1/t_T between the time t_1 for terminating successfully with one thread and the time t_T for terminating successfully with T threads.

5.1.2 ParaPDS and the Master-Servant Design

The only existing parallelization of PNS described in the literature has so far been ParaPDS by Kishimoto and Kotani (1999).[2] This pioneering work of parallel PNS achieved a speedup of 3.6 on 16 processors on a distributed memory machine. We note that Kishimoto and Kotani referred to processes instead and whereas we refer to threads in our shared memory setting. ParaPDS relies on a master-servant design.

[2]Conceptually related to PNS is Conspiracy-Number Search (CNS) by McAllester (1988). Lorenz (2001) proposes to parallelize a variant of CNS (PCCNS). PCCNS uses a master-servant model ("Employer-Worker Relationship").
We note that just before the printing of this thesis, a parallel version of df-pn on shared memory systems has been proposed by Kaneko (2010). It was tested on Shogi problems on which it produced a scaling factor of 3.6 with 8 threads.

One master process is coordinating the work of several servant processes. The master manages a search tree up to a fixed depth d. The master traverses through the tree in a depth-first manner typical for PDS. On reaching depth d it assigns the work of searching further to an idle servant. The search results of the servant process are backed up by the master.

The overhead created by ParaPDS is mainly a search overhead. There are two reasons for this overhead: (1) lack of a shared-memory transposition table, and (2) the particular master-servant design. Regarding reason (1), ParaPDS is asynchronous, i.e., no data is passed between the processes except at the initialization and the return of a servant process. ParaPDS thereby avoids message passing. The algorithm is designed for distributed-memory machines common at the time ParaPDS was invented (i.e., 1999). Transposition tables are important to PDS, as this variation of PNS performs iterative deepening. An implication of using distributed-memory machines is that ParaPDS cannot profit from a shared transposition table and loses time on re-searching nodes. Regarding reason (2), the master-servant design can lead to situations in which multiple servant processes are idle because the master process is too busy updating the results of another process or finding the next candidate to pass to a servant process.

One may speculate that the lack of a shared-memory transposition table in ParaPDS could nowadays be amended to a certain degree, at the expense of a synchronization overhead, by the availability of shared-memory machines. However, the second reason for ParaPDS' overhead due to the master-servant design still remains.

5.1.3 Randomized Parallelization

An alternative to the master-servant design of ParaPDS for parallelizing tree search is *randomized parallelization*. Shoham and Toledo (2002) proposed a method for parallelizing *any* kind of best-first search on AND/OR trees. The method relies on a heuristic which may seem counterintuitive at first. Instead of selecting the child with the best heuristic evaluation, a probability distribution of the children determines which node is selected. Shoham and Toledo call this a *randomization* of the move selection. Randomized Parallel Proof-Number Search (RP–PNS) as proposed in this chapter adheres to the principle of randomized parallelization. The specific probability distribution is obviously based on the selection heuristic. We note that we successfully applied this randomization technique to MCTS-Solver (cf. Chapter 4).

The master-servant design of ParaPDS and randomized parallelization may be compared as follows. ParaPDS maintains a privileged master thread: only the master thread operates on the top-level tree. The master thread selects the subtree in which the servant threads search. It also coordinates the results of the servant processes. Each servant thread maintains a separate transposition table in memory. In randomized parallelization there is no master thread. Each thread is guided by its own probabilities for selecting the branch to explore. There is no communication overhead but instead there is synchronization overhead. All threads can operate on the same tree which is held in shared memory.

5.2 RP–PNS

This section introduces RP–PNS. Subsection 5.2.1 explains the basic functioning of RP–PNS and describes how it differs from ParaPDS. Subsection 5.2.2 explains details of the implementation of RP–PNS.

5.2.1 Detailed Description of Randomized Parallelization for PNS

There are two kinds of threads in RP–PNS: (1) principal-variation (PV) threads, and (2) alternative threads. RP–PNS maintains one PV thread; all other threads operating on the search tree are alternative threads. Both kinds of threads apply a best-first selection heuristic in which the most-proving node is calculated. In the PV thread, the most-proving node is always selected. In alternative threads a probability determines if the most-proving node or one of its siblings is selected.

The PV thread always applies the same selection strategy as sequential PNS. At a node N in the PV thread the most-proving child is selected. Its value is the most-proving number which corresponds exactly to the proof or disproof number of most-proving child. More precisely, if N is an OR node, the successor with the smallest proof number is always selected; if N is an AND node, the successor with the smallest disproof number is always selected. The PV thread therefore operates on the PV, i.e., the path from root to leaf following the heuristic for finding the most-proving node. We call this selection strategy *PV selection strategy* and a child on the PV, a *PV node*.

The alternative threads select a node according to a modified selection strategy. Instead of always selecting the most-proving node, there is a chance of selecting a suboptimal sibling. A probability distribution in the heuristic creates the desired effect: the expanded nodes are always close to the PV since nodes expanded in alternative threads would likely be on the PV at a later cycle. The alternative threads anticipate a possible future PV. The probability of a suboptimal node to be selected for an alternative thread depends on the degree by which it deviates from the PV. In the selection step, alternative threads consider a subset of best children instead of always picking the single best child. All children in this subset have the same probability to be selected.

More precisely, the probability of an alternative thread to select a child of a node N can be calculated as follows. If N is an OR node, we define $\phi_D(\mathsf{N})$ to be the number of N's children C for which $pn(\mathsf{C}) \leq pn(\mathsf{N}) + D$. Analogously, if N is an AND node, we define $\delta_D(\mathsf{N})$ to be the number of N's children C for which $dn(\mathsf{C}) \leq dn(\mathsf{N}) + D$. The probability $p(\mathsf{N}, \mathsf{C})$ to select a certain child C of N is then given by Formula 5.1:

$$p(\mathsf{N},\mathsf{C}) = \begin{cases} (\phi_D(\mathsf{N}))^{-1} & : \quad \mathsf{N} \text{ is an OR node and } pn(\mathsf{C}) \leq pn(\mathsf{N}) + D \\ (\delta_D(\mathsf{N}))^{-1} & : \quad \mathsf{N} \text{ is an AND node and } dn(\mathsf{C}) \leq dn(\mathsf{N}) + D \\ 0 & : \quad \text{otherwise} \end{cases} \tag{5.1}$$

The parameter D in Formula 5.1 is a natural number and regulates the degree to which the alternative threads differ from the PV. Setting $D = 0$ will result in the

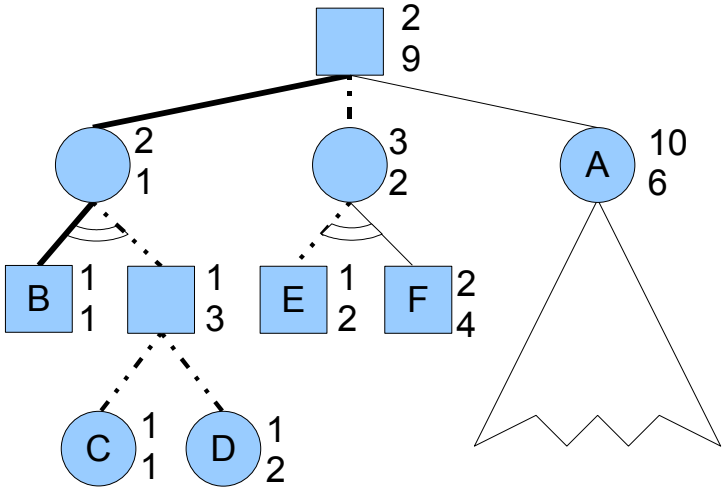


Figure 5.1: Example of a PNS tree. Squares represent OR nodes; circles represent AND nodes. Depicted next to each node are its proof number at the top and its disproof number at the bottom.

PV selection strategy.[3] Setting D too high results in threads straying too far from the PV.

Figure 5.1 illustrates the consequences of varying the parameter D. In this example, the PV is represented by the bold line and reaches leaf B. An alternative selection with $D = 1$ is represented by the bold, dotted lines. It will select one of the leaves B, C, D, or E with equal probability. Setting $D = 2$ will result in also selecting F. We note that the subtree at A is selected only for $D \geq 8$.

In addition to these probabilities, for all alternative threads we assign a second probability of deviating from the PV. This is done by choosing with a probability of $2/d$ randomly from the second and third best child (determined by trial-and-error) instead of choosing the best child if so far the thread has not deviated from the PV. This choice is determined by the depth d of the last PV. This mechanism is only applied if depth $d > 2$. The additional randomization is necessary because it enables sufficient deviation from the PV in case that D is not large enough to produce any effect; it has a strong effect on the behavior of the parallelization.

We remark that RP–PNS differs from the original randomized parallelization with respect to three points. (1) Shoham and Toledo (2002) do not distinguish between PV and alternative threads. (2) The original randomized parallelization selects children with a probability proportionate to their best-first value while RP–PNS uses an equi-distribution for the best candidates. (3) The original randomized

[3]More precisely, this is true if the most-proving child is unique. If multiple children have the same most-proving number, the alternative threads can deviate from the PV which we assume to be selected deterministically in PNS.

parallelization does not rely on a second probability. The differences in points 2 and 3 are based on the desire to produce more deviations from the PV in order to avoid that too many threads congest the same subtree. The selection in RP–PNS is similar to Buro's selection of a move from an opening book (Buro, 1999).

In RP–PNS multiple threads operate on the same tree. To facilitate the parallel access some complications in the implementation require our attention. The next subsection gives details of the actual implementation of RP–PNS.

5.2.2 Implementation

As pointed out in the previous subsection, all threads in RP–PNS operate on the same search tree held in shared memory. In order to prevent errors in the search tree, RP–PNS has to synchronize the threads. This is achieved in the implementation by a locking policy. Each tree node has a lock. It guarantees that only one thread at a time operates on the same node while avoiding deadlocks. The locking policy consists of two parts: (1) when a thread selects a node, it has to lock it; (2) when a thread updates a node N it has to lock N and its parent. The new values for N are computed. After N has been updated, it is released and the updating continues with the parent.

Each node N maintains a set of flags, one for each thread, to facilitate the deletion of subtrees. Each flag indicates whether the corresponding thread is in the subtree below N. A thread can delete the subtree of N only if no other thread has set its flag in N.

If a transposition table is used to store proof and disproof numbers, each table entry needs an additional lock. The number of locks for the transposition table could be reduced by sharing locks for multiple entries. Similar policies have been used in parallel Monte-Carlo Tree Search by (Chaslot *et al.*, 2008a) (to which the master-servant design has also been applied, cf. Cazenave and Jouandeau, 2007 and 2008). Synchronization imposes a cost on RP–PNS in terms of memory and time consumption. The memory consumption increases due to the additional locks (per node, 16 bytes for a spin lock and flags, cf. Chaslot *et al.*, 2008a) in each node.

The overhead is partially a synchronization overhead and partially a search overhead. The synchronization overhead occurs whenever a thread has to wait for another thread due to the locking policy or due to the transposition-table locking. The search overhead is created by any path that would not have been selected by the sequential PNS and that at the same time does not contribute to find the proof. The following section describes experiments that also test the overhead of RP–PNS.

5.3 Experiments

This section presents experiments and results for RP–PNS and RP–PN^2. Subsection 5.3.1 outlines the experimental setup, Subsection 5.3.2 shows the results obtained, and Subsection 5.3.3 discusses the findings.

5.3.1 Setup

We implemented RP–PNS as described in the previous section and tested it on complex endgame positions of Lines of Action (LOA).[4] We chose LOA because it is an established domain for applying PNS. The test set consisting of 286 problems has been applied before frequently (Winands *et al.*, 2003b; Pawlewicz and Lew, 2007; Van den Herik and Winands, 2008).

The experiment tests two parallelization methods: RP–PNS and RP–PN^2 for 1, 2, 4, and 8 threads. The combination of an algorithm with a specific number of threads is called a *configuration* and denoted by indexing the number of threads, e.g., RP–PNS$_8$ is RP–PNS using eight threads. We remark that PNS = RP–PNS$_1$ and PN^2 = RP–PN$_1^2$.

The implementation of RP–PN^2 uses RP–PNS for PN$_1$ and PNS for PN$_2$. The size of PN$_2$ was limited to S^ϵ/T, where S is the size of the PN$_1$ tree, T is the number of threads used, and ϵ is a parameter. We set ϵ such that it is a compromise between memory consumption and speed suitable for the test set. The compromise is faster than using the full S as suggested by Allis (1994). Using S for the limit slows down RP–PN^2 disproportionally when many threads are used because the PN$_1$ tree grows faster in RP–PN^2 than in the sequential PN^2. Moreover, the size of PN$_2$ grows rapidly resulting in slowing down RP–PN^2. An advantage of using the above limit compared to the method of Breuker (1998) is that the former is robust to varying problem sizes. The values for the parameters of RP–PNS were set to $D = 5$ and $\epsilon = 0.75$ based on trial-and-error.

The experiments were carried out on a Linux server with eight 2.66 GHz Xeon cores and 8 GB of RAM. The program was implemented in C++.

5.3.2 Results

Two series of experiments were conducted. The first series tests the efficiency of RP–PNS; the second tests the efficiency of RP–PN^2.

For comparing the efficiency of different configurations, we selected a subset consisting of the 143 problems for which PNS was able to find a solution in less than 30 seconds. This selection enabled us to acquire the experimental results for the series of experiments for RP–PN^2 in a reasonable time. We call the set of 143 problems the comparison set, S_{143}. PNS required an average of 4.28 million evaluated nodes for solving a problem of S_{143} with a standard deviation of 2.9 million nodes.

In the first series of experiments we tested the performance of RP–PNS for solving the positions of S_{143}. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the upper part of Table 5.1. We observe that the scaling factors for 2, 4, and 8 threads are 1.6, 2.5, and 3.5, respectively. Based on the results we compute that the search overhead expressed by the number of nodes evaluated is only ca. 33% for 8 threads. This means that the synchronization overhead is responsible for the largest part of the total overhead. Finally, we see that RP–PNS$_8$ uses 50% more memory than PNS.

[4] The test set is available at http://www.personeel.unimaas.nl/m-winands/loa/endpos.html, "Set of 286 hard positions".

	PNS	RP–PNS$_2$	RP–PNS$_4$	RP–PNS$_8$
Total Time (sec.)	1,679	1,072	682	478
Total scaling factor	1	1.6	2.5	3.5
Total nodes evaluated (million)	612	673	745	815
Total nodes in memory (million)	367	423	494	550

	PN^2	RP–PN$^2_2$	RP–PN$^2_4$	RP–PN$^2_8$
Total Time (sec.)	6,735	3,275	1,966	1,419
Total scaling factor vs. PN^2	1	1.9	3.4	4.7
Total scaling factor vs. PNS	0.25	0.52	0.85	1.18
Total nodes evaluated (million)	2,271	2,426	2,534	2,883
Total nodes in memory (million)	68	68	70	73

Table 5.1: Experimental results for RP–PNS and RP–PN^2 on S_{143}. The total time is the time required for solving all problems. "Nodes in memory" is the sum of all M_i, where M_i is the maximum number of nodes in memory used for test problem i. Nodes evaluated is the sum of all nodes evaluated for all problems. For RP–PN^2, this includes evaluations in the PN$_2$ tree and possible double evaluations when trees are re-searched.

In the second series of experiments we tested the performance of RP–PN^2. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the lower part of Table 5.1. We observe that the scaling factors for 2, 4, and 8 threads are 1.9, 3.4, and 4.7, respectively. Compared to RP–PNS the relative scaling factors of RP–PN^2 are better for all configurations. The search overhead of RP–PN$^2_8$ is 27% which is comparable to the search overhead of RP–PNS$_8$ (33%, cf. above). At the same time the total overhead of RP–PN$^2_8$ is smaller. This means that the synchronization overhead is smaller for RP–PN$^2_8$ than for RP–PNS$_8$. The reason is that more time is spent in the PN$_2$ trees. Therefore, the probability that two threads simultaneously try to lock the same node of the PN$_1$ tree is reduced. Finally, we remark that in absolute terms, RP–PN$^2_8$ is slightly faster than PNS.

Despite the fact that RP–PN^2 has a better scaling factor than RP–PNS, RP–PNS is still faster than RP–PN^2 when the same number of threads is used. However, RP–PN^2 consumes less memory than RP–PNS.

5.3.3 Discussion

It would be interesting to compare the results of the experiments presented in Subsection 5.3.2 to the performance of ParaPDS. However, the direct comparison between the results obtained for ParaPDS and RP–PNS is not feasible because of at least three difficulties.

First, the games tested are different (ParaPDS was tested on Othello, whereas RP–PNS is tested on LOA). Second, the type of hardware is different. As described in Section 5.1, ParaPDS is designed for distributed memory whereas and RP–PNS is

designed for shared memory. Third, ParaPDS is a depth-first search variant of PNS whereas RP–PNS is not. ParaPDS is slowed down because of the transposition tables in distributed memory.

ParaPDS and RP–PN^2 both re-search in order to save memory. When comparing the experimental results for these two algorithms they appear to scale up in the same order of magnitude at a superficial glance. On closer inspection, a direct comparison of the numbers would be unfair. ParaPDS and RP–PN^2 parallelize different sequential algorithms. Furthermore, RP–PN^2 parallelizes transposition tables while our implementation of RP–PNS does not. Moreover, it can be expected that sequential PN^2 profits more from transposition tables than RP–PN^2 because the parallel version would suffer from additional communication and synchronization overhead.

In RP–PN^2 the size of the PN_2 tree determines how much the algorithm trades speed for memory. If the PN_2 tree is too large, the penalty for searching an unimportant subtree will be too large as well. In our implementation, we chose rather small PN_2 trees because the randomization is imprecise. Moreover, the PN_2 trees are bigger when fewer threads are used. This explains why RP–PN^2 (with a scaling factor of 4.7) scales better than RP–PNS (with a scaling factor of 3.5). A second factor contributing to the better scaling is the reduced synchronization overhead compared to RP–PNS. This effect is produced by the smaller relative number of waiting threads.

We may speculate that RP–PNS and RP–PN^2 could greatly profit from a more precise criterion for deviating from the PV. To that end, it is desirable to find a quick algorithm for finding the k-most-proving nodes in a proof-number search tree. Thereby, the search could process the true best variations instead of pseudo-randomly chosen variants close to the PV.

5.4 Chapter Conclusion and Future Research

We end this chapter by summarizing the results of the experiments (Subsection 5.4.1) and giving an outlook on future research (Subsection 5.4.2) based on the findings presented.

5.4.1 Chapter Conclusion

In this chapter, we introduced a new parallel Proof-Number Search algorithm for shared memory, called RP–PNS. The parallelization is achieved by threads that select moves close to the principal variation based on a probability distribution. Furthermore, we adapted RP–PNS for PN^2, resulting in an algorithm called RP–PN^2.

The scaling factor for RP–PN^2 (4.7) is even better than that of RP–PNS (3.5) but this is mainly because the size of the PN_2 tree depends on the number of threads used. Based on these results we may conclude that RP–PNS and RP–PN^2 are viable for parallelizing PNS and PN^2, respectively. Strong comparative conclusions cannot be made for ParaPDS and RP–PNS.

5.4.2 Future Research

Future research will address the following four directions. (1) A combined parallelization at PN_1 and PN_2 trees of RP–PN^2 will be tested on a shared-memory system with more cores. (2) A better distribution for guiding the move selection, possibly by including more information in the nodes, will be tested to reduce the search overhead. For instance, the probability of selecting a child could variably depend on characteristics of the previous PV. (3) The concept of the k-most-proving nodes of a proof-number search tree and an algorithm for finding these nodes efficiently on a parallelized tree will be investigated. (4) The speedup of reducing the number of node locks by pooling will be investigated. In pooling multiple nodes share the same lock and thereby reduce the number of locks used.

Chapter 6

Paranoid Proof-Number Search

This chapter is based on the following publication.[1]

- J-T. Saito and M.H.M. Winands Paranoid Proof-Number Search. In G.N. Yannakakis and J. Togelius, editors, *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG'10)*, pp. 203–210, IEEE Press, 2010.

Solving algorithms have been specifically designed to tackle two-player games (cf. Van den Herik *et al.*, 2002; Heule and Rothkrantz, 2007). The previous chapters elaborated on algorithms for solving these games. This chapter extends the scope to multi-player games (i.e., games with more than two players). So far, the notion of solving in multi-player games has attracted little if any attention by the game-and-search community. By addressing this topic, we supply an answer to **RQ 4:** *How can solvers be applied to multi-player games?*

The standard search algorithms for multi-player games are $\max^n$ (Luckhardt and Irani, 1986) and paranoid search (Sturtevant and Korf, 2000). These search algorithms were designed to provide playing strength for programs but not to solve games. Proof-Number Search (PNS) (Allis *et al.*, 1994), however, was designed for solving as we have seen in Chapter 2. The notion of *solving* in multi-player games differs from that in two-player games. Therefore, we suggest a definition for solving a game under the *paranoid condition* (Sturtevant and Korf, 2000). Under this condition, a player assumes that all opponents form a coalition against him. Furthermore, we propose to modify PNS for solving under the paranoid condition, and call the resulting algorithm *Paranoid Proof-Number Search* (PPNS). We apply PPNS to small-board variants of the multi-player game of Rolit to find the optimal scores that a player can achieve.

This chapter is organized as follows. Section 6.1 explains two search algorithms for multi-player games, $\max^n$ and paranoid search, and introduces concepts for describing search in multi-player game trees. Based on these concepts, Section 6.2

[1] The author is grateful to IEEE for the permission to reuse relevant parts of the article in the thesis.

gives a definition for solving a position in a multi-player game and proposes the PPNS algorithm. Section 6.3 discusses how PPNS can be used to find an optimal score. The game of Rolit, its rules and properties are outlined in Section 6.4. Section 6.5 describes the experimental setup for applying PPNS to solve variants of Rolit. The results of the experiments are presented and discussed in Section 6.6. Section 6.7 concludes the chapter and gives an outlook on future research.

6.1 Search Algorithms for Multi-Player Games

This section discusses two well understood search algorithms for multi-player turn-taking games, max^n (Luckhardt and Irani, 1986) and paranoid search (Sturtevant and Korf, 2000), and provides concepts for characterizing them. The section is organized in three subsections as follows.

Subsection 6.1.1 describes max^n, a straightforward generalization of minimax search suitable for multi-player games. The limitations of max^n relate to its way of coping with so-called equilibrium points. These are the subject of Subsection 6.1.2. Subsection 6.1.3 outlines Paranoid search which forms the basis for PPNS described in Section 6.2.

6.1.1 The Max^n Algorithm

The max^n algorithm (Luckhardt and Irani, 1986) is a generalization of minimax search to games with more than two players. It assumes that players $1, ..., n$ participate in a turn-taking game. Each node N in the max^n tree has an n-tuple $\vec{\mathsf{N}} = (v_1, v_2, ..., v_n)$ called the *max^n value of* N. $\vec{\mathsf{N}}[i] = v_i$ represents the value of the game at node N for Player i. Furthermore, if Player i is to move at node N we occasionally say that $\vec{\mathsf{N}}[i]$ is *the value of* N (omitting that we mean *"for Player i"*).

If N is a leaf, the value $\vec{\mathsf{N}}[i]$ is the value of Player i computed by the evaluation function. If N is an internal node with m children C_1,...,C_m and player i to move, then N's tuple is calculated from the children's tuples recursively as $\vec{\mathsf{N}} = \vec{\mathsf{C}_j}$ where $\vec{\mathsf{C}_j}$ is the child with the largest value for Player i among all children.

Figure 6.1 depicts a three-ply max^n tree. Player 1 moves at the root and the leaves, Player 2 at the first ply, and Player 3 at the second ply. The tuple at node D is based on D's children. In this case, F offers the largest value, for Player 3, i.e., 3. (It is just the same as its sibling G's value). All values (the whole tuple) of F are taken over to D. Similarly, at node B, Player 2 chooses its children. Node D offers the largest value for Player 2, namely 6. The root A takes over the tuple from node B because Player 1 prefers value 3 above 1.

The $\alpha\beta$ algorithm (Knuth and Moore, 1975) is successful in two-player games because it allows pruning. In combination with move ordering $\alpha\beta$ has proven to be a powerful tool that leads the way to success in many games, most notably in Chess (Hsu, 2002). Unfortunately, when generalizing from minimax to max^n, in spite of the existence of certain pruning techniques such as *shallow pruning* (Luckhardt and Irani, 1986), the potential of pruning is reduced considerably (Sturtevant and Korf, 2000). Korf (1991) showed that shallow pruning guarantees the correct solution

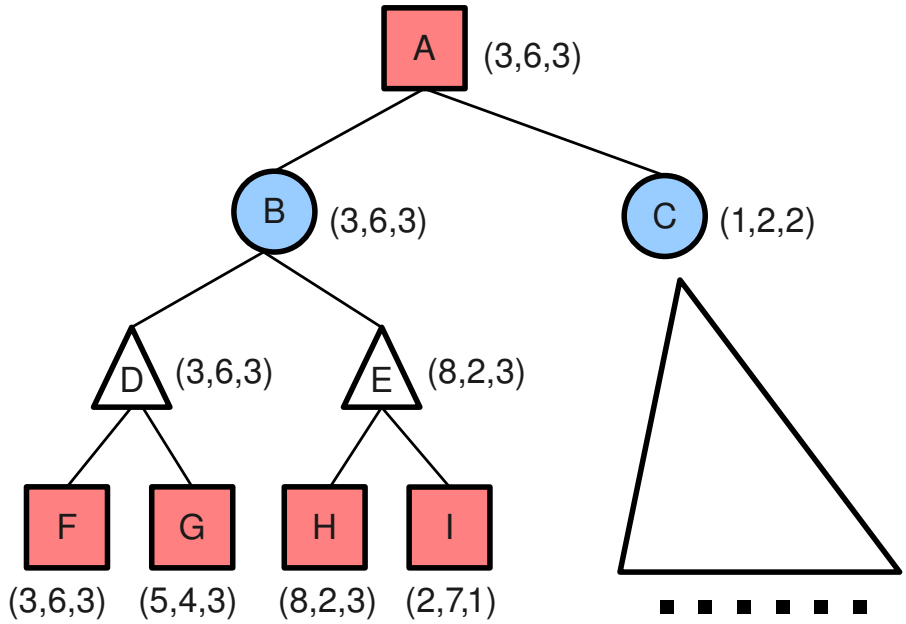


Figure 6.1: Example of a three-player three-ply max^n tree.

but in comparison to $\alpha\beta$ pruning, it is often inefficient and also imposes some mild restrictions on the game scores (cf. Sturtevant, 2003a). Other pruning techniques such as *speculative pruning* have been shown to be more efficient in game playing but cannot always guarantee the correct solution (Sturtevant, 2003a).

6.1.2 Equilibrium Points

As stated in Chapter 2, a fully expanded two-player minimax tree has exactly one game-theoretic value. This game-theoretic value can be described as an equilibrium point (cf. Luce and Raiffa, 1957), a state in which "no player finds it to his advantage to change to a different strategy so long as he believes that the other players will not change." Multi-player games may have multiple equilibrium points with different values. This means, that without loss of generality no unique game-theoretic value or strategy exists (Sturtevant, 2003a). In particular, it is possible that a multi-player game has multiple equilibrium points with different equilibrium values.

Figure 6.1 illustrates this observation. As described above, the root has the max^n value (3,6,3). We observe that Player 3 at D chose to take over the value from F. This choice is arbitrary. Player 3 has the same value at D regardless of choosing F or G. Each of the two alternatives results in a different outcome, both of which create equilibrium points. Moreover, if Player 3 decided in favor of G, this would cause Player 2 to select the max^n value of the now different D, namely (5,4,3)

for node B. As a consequence Player 1 at the root A would follow suit. Thus the max^n values at the root would become (5,4,3) instead of (3,6,3).

The rule for choosing between equivalent children is called the *tie-breaking rule*. The tie-breaking rule determines what equilibrium point max^n will find. If all players apply the same tie-breaking rule, the game play may be "reasonable" (Sturtevant and Korf, 2000). A trivial example of a tie-breaking rule is minimizing the value of the starting player.

6.1.3 Paranoid Search

The lack of pruning in max^n has encouraged research on alternative multi-player search algorithms. Paranoid search by Sturtevant and Korf (2000) is such an alternative that offers deeper search. It reduces an n-player game to a two-player game by assuming that one player (the *paranoid player*) faces a coalition of all other players (the *coalition players*). While the paranoid player tries to maximize his or her own value as in max^n, all coalition players try to minimize the paranoid player's value instead of maximizing their own.

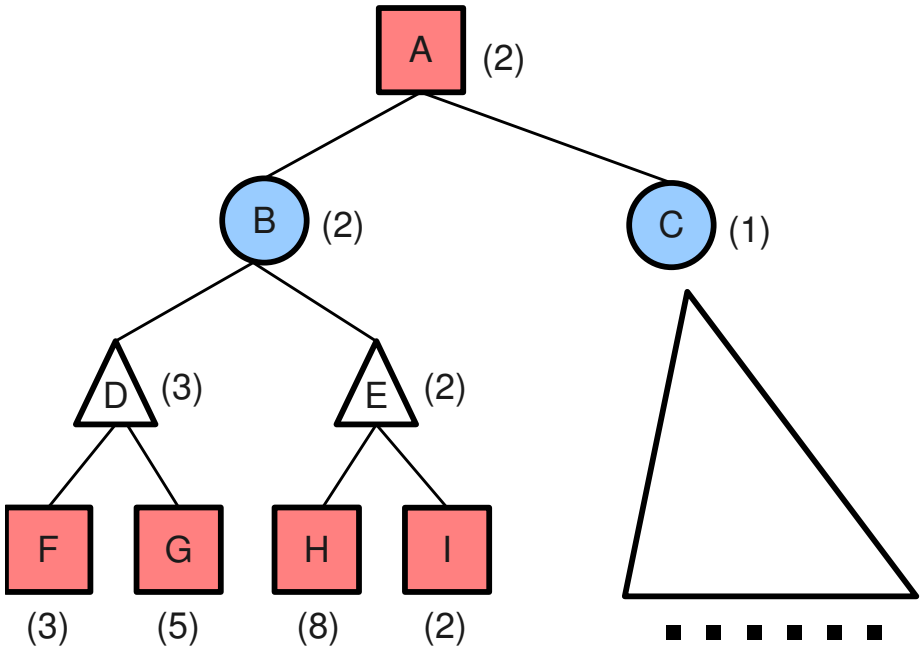


Figure 6.2: Example of a three-player three-ply paranoid tree.

This is illustrated in Figure 6.2. The values at the leaves are the same as in the previous max^n example (cf. Figure 6.1) but the values are propagated to the root differently. All players take into account only the values of the paranoid player, Player 1 (Sturtevant, 2003a). The coalition consists of Players 2 and 3.

At node D, Player 3 chooses F instead of G minimizing the value of Player 1. At node E, Player 3 chooses I with value 2 for Player 1 over H which has value 8 for Player 1. Similarly, Player 2 chooses to minimize Player 1's value at node B. Finally, Player 1 is left with maximizing his value at the root A resulting in the value 2.

Given branching factor b and depth d the best-case for the paranoid search tree is $O(b^{d(n-1)/n})$ nodes (Sturtevant and Korf, 2000). This size is easily derived from the best case of minimax, $O(b^{d/2})$, by collapsing consecutive plies of the coalition players into one ply.

Paranoid search operates under a strict opponent model assuming that one player faces a coalition of collaborating opponents. This strong constraint may influence the playing strength of the algorithm negatively. However, Sturtevant (2003b) showed that paranoid search can play on the same level as or even outperform max^n in Hearts (3 and 4 players), Spades (3 players), and Chinese Checkers (3 to 6 players) in practice. However, max^n with advanced pruning methods can outperform paranoid search in game play (Sturtevant, 2003c).

6.2 Paranoid Proof-Number Search

Besides the improvements in playing strength, paranoid search has a property that makes it attractive for solving. As stated in Subsection 6.1.2, max^n (in general) has multiple equilibrium points with different outcomes. Paranoid search avoids this problem. Since a minimax tree is produced, only one game-theoretic outcome can be produced. Moreover, this outcome is identical to the optimal score the paranoid player can achieve without help by his opponents. Thus the paranoid condition can be used to find a characteristic optimal score. The following definition applies to multi-player games in which each individual player has a score.

> **Definition.** A *solution to an n-player game under the paranoid condition for Player i* (with $i \in \{1, ..., n\}$) is the *optimal score* that Player i is guaranteed to achieve independent of the opponents' strategies.

Paranoid Proof-Number Search (PPNS) extends PNS for multi-player games under the paranoid condition. In this way PPNS can exploit all benefits of PNS for multi-player games. PPNS performs an efficient variable-depth search designed for solving. It develops the tree into the direction where the opposition is the weakest. PPNS exploits just as PNS the non-uniformity of the game tree (i.e., a variable branching factor and depth, Plaat, 1996), which paranoid search is not able to do. An example of effective exploitation of non-uniformity by PNS is the solving of 6×6 LOA (Winands, 2008).

Since the PNS framework is used in PPNS, a binary goal is set. If the goal of PPNS is to prove a win for the paranoid player, then all OR nodes represent positions in which the paranoid player moves and all AND nodes represent positions in which a coalition player moves. While in regular PNS OR and AND nodes alternate every ply, in PPNS (n-1) ply sequences of the coalition nodes alternate with single nodes of the paranoid player. We note that the same four-step cycle (cf. Chapter 2) is applied in PPNS as known for PNS.

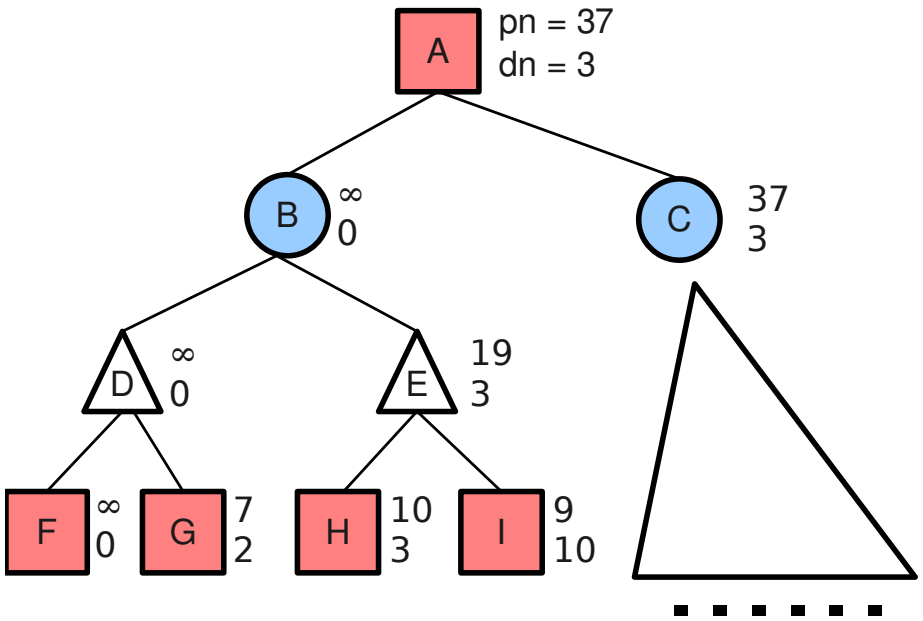


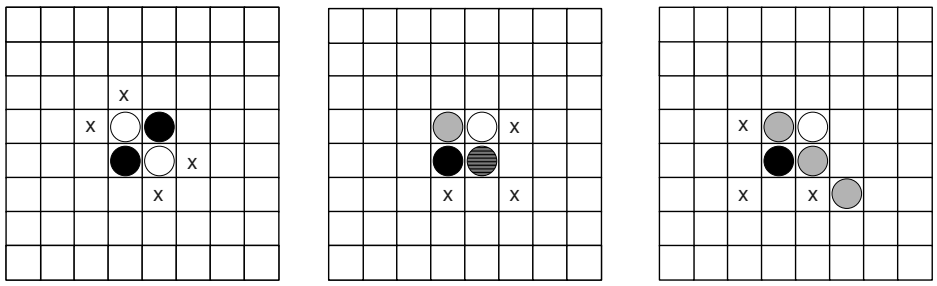
Figure 6.3: Example of a three-player three-ply PPNS tree. The paranoid player's nodes, squares, are OR nodes. The coalition players' nodes, circles and triangles, are AND nodes.

The above Figure 6.3 depicts a three-player three-ply PPNS tree (not related to the previous examples) in which the paranoid player is represented by OR nodes and two coalition players are represented by AND nodes. Thus, the *pn* and *dn* values of nodes B, D and E are calculated according to the update rule for AND nodes. The values of the root A are calculated from B and C by the update rule for OR nodes.

6.3 Finding the Optimal Score

All variants of PNS (Van den Herik and Winands, 2008) including PPNS attempt to proof a binary goal. However, we are interested in finding an optimal value among many possible values, namely the optimal score for the paranoid player. Thus, PPNS must be applied several times.

A naïve approach to finding the optimal score consists of simply running one PNS for each possible *score*. For instance, in 6×6 Rolit a Player p can end the game with a score between 0 and 36. PNS could be run 37 times to proof or disproof a win for p for each score. The highest score for which a win is found is the optimal score. In order to increase efficiency compared to the naïve approach, Allis *et al.* (1994) suggested using binary search to find the optimal score. Furthermore, one could resort to re-using information when attempting to find the optimal score. Moldenhauer (2009) therefore introduced *iterative proof number search* (IPN) which reduces search time by caching information stored in earlier runs.



(a) Initial position of Reversi. (b) Initial position of four-player 8 × 8 Rolit. (c) Four-player 8×8 Rolit after one move.

Figure 6.4: Game boards of 8 × 8 Reversi and 8 × 8 Rolit. The crosses indicate playable positions for the player to move.

6.4 The Game of Rolit

In this section we describe the game of Rolit. Subsection 6.4.1 presents a predecessor of Rolit and two other related games. Subsection 6.4.2 outlines the rules of Rolit. Subsection 6.4.3 considers the search space of the game. Subsection 6.4.4 completes the characterization by evaluating Rolit with a Monte-Carlo player.

6.4.1 Predecessors and Related Games

Rolit$^{\text{TM}}$ (Hasbro International Inc., 1999) is a multi-player variant of the well-known game Reversi. It is therefore instructive to start with a description of this game before turning to Rolit. Reversi is a deterministic turn-taking two-player perfect-information game played on an 8 × 8 board. The players, Black and White, take turns in which they either pass or place one piece in a field. Black moves first. The pieces of the game are disks with two faces. One face is black, the other white. Player Black must place pieces showing the black face and player White must place pieces showing the white face. Both players start with 30 pieces each. Black moves first from the defined starting position (cf. Figure 6.4). A player can only place a piece, if he can play a flipping move. If no flipping move is possible, the player has to pass.

A *flipping move* is a move that places a piece at the end of a flipping line. A *flipping line* is a line of directly connected fields on the board such that four conditions are met: (1) The fields on the line have to be all vertically, all horizontally, or all diagonally connected. (2) All fields on the line are occupied. (3) Both ends of the line must show the turn-taking player's face. (4) All pieces between the ends show the opponent's face. If a flipping move is played, the pieces between the ends are flipped to show the turn-taking player's face. If a move enables more than one

flipping line, all affected opponent's pieces have to be flipped.

The game ends if either (a) one player runs out of his or her 30 pieces, or (b) both players pass consecutively. When the game has ended, Black's score is the number of pieces showing the black face, and White's score is the number of pieces showing the white face. The player with the higher score wins. If both players have the same score the game is a draw. We note that pieces which have been played remain on the board until the end of the game.

In the Reversi variant called Othello, there is no limit to the players' number of pieces. Thus the game ends in either of two cases: (a) the board is filled, or (b) both players pass consecutively (i.e., no player can make a flipping move).

Fujii *et al.* (2006) proposed *4-player Reversi Yonin* (Yonin). With the exception of Rolit, it is the only multi-player variant of Reversi. This variant is played by exactly four players. Two of them, Players A and C, place pieces showing the black face and the other two, Players B and D, place pieces with the white face up. The players take turns in the order A, B, C, D and the board is separated into quarters. Each player owns exactly one quarter. Players of the same color own quarters diagonally opposing each other. The game is played and scored like Reversi, but there are two deviations. First, a player may not place a piece in the diagonally opposing quarter. Thus the branching factor for the 8×8 board is reduced from ca. 8.5 to ca. 6.3 (Fujii *et al.*, 2006). Second, in the end, a Player p's score is determined by counting the usual way but only in the quarter owned by P. (We note that 4-player Reversi Yonin can also be played in teams of two players each. In that case, the teams are A and C versus B and D). Fujii *et al.* (2006) studied the properties of 6×6 and 8×8 Yonin empirically by applying Monte-Carlo evaluation.

6.4.2 Rules of Rolit

Rolit is a straightforward multi-player generalization of Reversi. Instead of disks with two faces, balls with four different-colored quarter spheres are placed. Correspondingly, up to four players can play Rolit, each placing his characteristic color facing up. The four players are Red, Green, Yellow, and Blue.

Red always moves first. Depending on how many players participate (two, three, or four), the following orders are applied for turn-taking;

2 players: (1) Red, (2) Green;

3 players: (1) Red, (2) Yellow, (3) Green;

4 players: (1) Red, (2) Yellow, (3) Green, (4) Blue.

Irrespective of the number of players participating, Rolit is always played with the same starting position shown in the center diagram of Figure 6.4.

In Rolit, one player's color can be completely eliminated from the board by the opponent's flipping moves. In order to allow a player who has no piece left on the board to come back into the game, the rules are changed compared to Reversi. Reversi's passing rule is replaced by Rolit's *free-choice rule*: a player who cannot

	Board size	Number players	Branching factor	Estimated game-tree size	Upper bound to state space
Reversi	8×8	2	8.47	4.7×10^{55}	6.8×10^{29}
	6×6	2	5.26	1.3×10^{23}	3.0×10^{16}
Rolit	8×8	2	8.50	5.8×10^{55}	6.8×10^{29}
	8×8	3	8.65	1.7×10^{56}	1.3×10^{37}
	8×8	4	8.25	9.7×10^{54}	2.2×10^{44}
	6×6	2	5.27	1.3×10^{23}	3.0×10^{16}
	6×6	3	5.25	1.1×10^{23}	1.8×10^{19}
	6×6	4	5.17	6.8×10^{22}	6.0×10^{24}
Yonin	8×8	4	6.26	1.2×10^{49}	6.8×10^{29}
	6×6	4	4.34	2.5×10^{20}	3.0×10^{16}

Table 6.1: Search spaces of 6×6 and 8×8 Reversi variants. The figures for Reversi and Yonin are according to Fujii *et al.* (2006).

make a flipping move does not pass but is allowed to place a piece (showing the player's color) on an empty field. The empty field must be horizontally, vertically, or diagonally adjacent to a piece on the board.

Due to this change in the rules, a game of Rolit ends exactly when all fields of the board are filled. The game is scored analogously to Reversi. The player with the most points wins.

6.4.3 Search Space of Rolit

Rolit is retailed in two board sizes, 6×6 (known as *Traveler's Rolit* and *Rolit Jr.*) and 8×8. In addition to these two board sizes, we also regard 4×4 Rolit for completeness. Throughout the remainder of this chapter a particular combination of board size and number of players is called a *configuration* (of Rolit). We consider nine configurations that result from combining the three board sizes with the three possible numbers of players.

The state space for solving a game can be estimated by the number of possible board configurations. For $N \times N$ Rolit with M players an upper bound for the state space is $M^4 \times (M+1)^{N^2-4}$. We arrive at this bound as follows. The first factor (M^4) is the number of states that the four center fields can take on. None of these fields can be empty at any point during the game, and they can show at most one of four colors. The second factor represents the state space of the remaining fields. Each of them can be either empty or occupied by a piece. Each piece can have one of the M colors. The actual state space is smaller than the upper bound because not all possible board positions can be reached by a sequence of legal moves from the starting position. For instance, a game position with empty fields entirely surrounding one field containing a piece is not reachable.

The fully expanded game tree has at most $b^{(N \times N-4)}$ positions, where b is the

	Red	**Green**	**Yellow**	**Blue**
		4×4		
2 players	44.3 (7.5)	*55.7 (8.5)*	-	-
3 players	26.1 (5.0)	35.1 (5.5)	*38.8 (5.6)*	-
4 players	18.9 (3.8)	26.4 (4.0)	27.0 (4.1)	*27.8 (4.1)*
		6×6		
2 players	42.2 (16.9)	*57.8 (19.1)*	-	-
3 players	32.9 (12.0)	21.7 (10.5)	*45.4 (13.5)*	-
4 players	12.3 (7.4)	29.7 (9.6)	19.8 (8.5)	
		8×8		
2 players	41.1 (30.2)	*58.9 (33.8)*	-	-
3 players	22.8 (19.1)	*44.2 (23.5)*	33.0 (21.4)	-
4 players	14.4 (13.8)	28.4 (16.8)	20.6 (15.3)	*36.6 (18.2)*

Table 6.2: Percentages of games won for one million Monte-Carlo games in 4×4, 6×6, and 8×8 Rolit with 2, 3, and 4 players. Average scores in parenthesis, frequently winning players italicized.

branching factor. The branching factor is small for the beginning of the game. Table 6.3 lists the empirically measured branching factors for all configurations of Rolit. The measurements were obtained by playing 1,000,000 Monte-Carlo games for each configuration. We can see that for each board size the number of players affects the branching factor only slightly.

Replacing the passing rule of Reversi by the free-choice rule of Rolit has two consequences: (1) the average game length is marginally smaller in Reversi than in Rolit because a game of Reversi can end if no player can make a flipping move. Similarly, (2) Reversi has a slightly smaller branching factor than Rolit for the same reason. Table 6.1 gives an overview of the search-space size for 6×6 and 8×8 Reversi, Rolit, and Yonin. The figures for Reversi and Yonin are based on the branching factors that Fujii *et al.* (2006) computed by performing Monte-Carlo simulations.[2]

6.4.4 A Monte-Carlo Player for Rolit

Although we cannot give a general Nash strategy for Rolit, we can characterize a configuration of Rolit by using players of the same strength. To that end, we implemented a Monte-Carlo player and recorded the results of 1,000,000 random games for each of the nine configurations of Rolit. We remark that the Monte-Carlo playouts neglect coalitions and the random player does not adapt his opponent model

[2]We note that the branching factor of 8.47 empirically found for 8×8 Reversi by Fujii *et al.* (2006) is slightly lower than an earlier estimate for the similar variant of 8×8 Othello given by Allis (1994) who assumed a branching factor of 10.

	4×4	**6×6**	**8×8**
2 players	2.79	5.27	8.50
3 players	2.92	5.25	8.65
4 players	3.15	5.17	8.25

Table 6.3: Average branching factors of 4×4, 6×6, and 8×8 Rolit for 2, 3, and 4 players in 1,000,000 Monte-Carlo games per configuration.

during the game. The results for the playouts are listed in Table 6.2. The random games show that Rolit is not a fair game: a player's starting position influences his or her likelihood of winning. Moreover, all games are won most often by the player who has the advantage of taking the final turn. The only exception occurs on three-player 4×4 Rolit. Here, Green moves last because the turn-taking order is Red-Yellow-Green and there are 12 turns. However, Yellow has the highest number of wins. We may speculate that this observation can be explained by the fact that smaller boards allow more exceptions. The reason for this is that the ratio the number of strategically important moves on the rim of the board to the number of center moves is highest on the 4×4 Rolit board.

6.5 Experimental Setup

We implemented PPNS in the PN^2 framework. Subsection 6.5.1 explains how the initialization of the leaves of the second-level search, i.e., PN_2, is performed. Subsection 6.5.2 describes the knowledge representation in the PPNS solver.

6.5.1 Initialization

The initialization of the *pn* and *dn* at the leaf nodes of the second-level search (PN_2) can be done by using the following domain knowledge.

Occupying corners is a large advantage in Rolit. We take this observation into account when setting *pn* and *dn* for a non-terminal leaf L on 6×6 Rolit and 8×8 Rolit and use the difference $\delta(\mathsf{L}) = corners_{OR}(\mathsf{L}) - corners_{AND}(\mathsf{L})$ between the number of corners occupied by the OR player and the number of corners occupied by the AND player(s). Based on this, we set $pn = (1 + c \times 4) - (c \times \delta(\mathsf{L}))$ and $dn = (1 + c \times 4) + (c \times \delta(\mathsf{L}))$. The constant c is a weight that determines how important the corner advantage is.

In pre-experiments we found that for 6×6 Rolit the initialization worked better than evaluation functions which take into account more move categories such as next-to-corner moves, or rim moves. We set $c = 4$ for 6×6 Rolit and $c = 6$ for 8×8 Rolit after trial-and-error.

6.5.2 Knowledge Representation and Hardware

The search algorithm, move generator and all other parts of the software were implemented in C. Game positions were implemented as 4×64 bitboards. Nodes store

	4×4			**6×6**			**8×8**		
	2	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**
Red	7	3	2	16	0	0	?	?	0
Green	9	4	1	20	0	0	?	?	0
Yellow	-	3	2	-	1	0	-	?	0
Blue	-	-	1	-	-	1	-	-	1

Table 6.4: Proven optimal scores for all players on 4×4, 6×6, and 8×8 Rolit for 2, 3, and 4 players under the paranoid condition.

the complete bitboards in order to speed up the traversal by PN^2 search.

All experiments were carried out on a Linux mixed cluster with two kinds of nodes: 10 nodes with four 2.33 Opteron processors with 4 GB, and 2 nodes with eight 2.66 GHz Xeon processors and 8 GB of RAM.

6.6 Results

In the experiments, we applied PPNS to solve 4×4, 6×6, and 8×8 Rolit, each for two, three, and four players, resulting in a total of 9 variants. We attempted to produce the optimal score under the paranoid condition for all 4×4 Rolit and 6×6 Rolit variants. For 8×8 Rolit we attempted to find bounds on the optimal score. The results of the experiments are given in three subsections. Subsection 6.6.1 gives the game results, and Subsection 6.6.2 describes the amount of search performed by PPNS. Finally, Subsection 6.6.3 compares search trees of PPNS with the ones of paranoid search for different configurations.

6.6.1 Game Results

Because up to 60 processors were at the disposal and each configuration can be tested for one score on one processor at the time, we did not apply binary search on the score (cf. Section 6.3) but used Monte-Carlo evaluations (cf. Subsection 6.4.4) to assign the scores to be tested. Subsequently, we ran multiple proofs for a range of scores in parallel.

Because of the large search space of 6×6 Rolit and 8×8 Rolit we tried to solve as few scores as possible to find the optimal score. Two cases were distinguished for 4×4 Rolit and 6×6 Rolit after some pre-experimental runs. First, the two-player configurations, and second, the multi-player configurations (i.e., 3 or 4 players). For two-player 6×6 Rolit with Red as paranoid player, the score was estimated to be in the range of 15 to 18 based on the Monte-Carlo estimate. Moreover, the two-player results were verified by running the proofs twice: first with solving for player Red, and then for player Green.

For 3 and 4 players, an empirical maximum score of 5 was set for the multi-player configurations. Then, attempts were made to prove all scores smaller than or

	4×4			**6×6**			**8×8**
	2	**3**	**4**	**2**	**3**	**4**	**4**
R	4.1×10^4	5.6×10^4	9.1×10^4	9.5×10^{13}	4.2×10^{10}	1.5×10^6	1.8×10^8
G	4.1×10^4	9.5×10^4	5.2×10^4	5.0×10^{12}	5.7×10^9	6.1×10^6	8.1×10^8
Y	-	6.5×10^4	1.2×10^5	-	2.5×10^9	2.4×10^6	5.2×10^{10}
B	-	-	5.1×10^4	-	-	1.0×10^5	1.7×10^9

Table 6.5: Nodes evaluated in 4×4, 6×6, and 8×8 Rolit for 2, 3, and 4 players under the paranoid condition.

equal to the empirical maximum score. Thus, we attempted to prove a win for the paranoid player with scores of 5, 4, 3, 2, and 1. In addition, the 5 scores are reduced to 4 scores for multi-player 6×6 Rolit and 8×8 Rolit for the last-moving player. At the moment PPNS has disproven score 2, PPNS can omit the proof attempt for score 1 (which is always the case). This is particularly convenient because disproofs can be achieved easier than proofs.

All configurations of 4×4 Rolit and 6×6 Rolit were solved. For 8×8 Rolit only the four-player configuration was solved, which was approached in a similar way as the multi-player configurations of 6×6 Rolit. We did not attempt to solve the two and three-player configurations because of the too large search space. The evaluation function was found to reduce the total number of nodes searched by up to 10%. This is little compared to a reduction of up to 50% with the much more elaborate evaluation function by Nagai (1999) designed specifically for Othello endgames.

The optimal scores found by PPNS are given in Table 6.4. The results are, that 4×4 Rolit has solutions that are non-trivial, i.e., scores larger than the minimum theoretic scores of 0 or 1 point exist for three- and four-player configurations. This is different in the case of 6×6 Rolit. Here, the finding is that 6×6 Rolit is a rather uninteresting game under the paranoid condition: the paranoid player can always be forced to the minimum analytical score of either 0 or 1 point.

We note that, interestingly, the Monte-Carlo estimate of 19.1 points for Green used for seeding the score in two-player 6×6 Rolit differs 0.9 points from the actual score of 20 points.

6.6.2 Search Trees

Table 6.5 shows the node consumption for solving the configurations as described in the previous subsection. Solving 4×4 Rolit required only a few hundred thousand nodes for all possible configurations and scores.

6×6 Rolit was solved by searching in the order of millions of nodes in the case of four-player configurations, billions for three-player configurations, and trillions of nodes for two-player configurations. The latter are the largest proofs completed in our experiments. The CPU time required for proving the most difficult configuration solved, two-player 6×6 Rolit, is ca. 225 hours (9 days).

6.6.3 PPNS vs. Paranoid Search

To assess the performance of PPNS we compared the empirically found search trees of PPNS with the analytical best-cases of the search trees that standard paranoid search would create for proving the scores of Red (cf. Subsection 6.1.3). We assume that standard paranoid search would not take advantage of the possible non-uniform nature of the game tree as PPNS does and therefore we can apply $O(b^{d(n-1)/n})$. For d we use the maximum game length of Rolit and for b the branching factor estimated by the Monte-Carlo playouts (cf. Table 6.3). For instance, to obtain estimates of the tree size of four-player 6×6 Rolit for paranoid search we calculate $5.17^{32\times3/4}$ and for PPNS we use the number of nodes evaluated for proving the scores of Red. We remark that paranoid search could perform better than $O(b^{d(n-1)/n})$ if domain-dependent move ordering would be available that prefers slim subtrees above large subtrees, taking advantage of the non-uniform nature of the game tree as PPNS does.

	4×4		6×6		8×8	
Players	**Paranoid**	**PPNS**	**Paranoid**	**PPNS**	**Paranoid**	**PPNS**
2	4.7×10^2	4.1×10^4	3.5×10^{11}	9.5×10^{13}	3.1×10^{27}	-
3	5.3×10^3	5.6×10^4	2.3×10^{15}	4.2×10^{10}	5.5×10^{35}	-
4	3.0×10^4	9.1×10^4	1.3×10^{17}	1.5×10^6	1.7×10^{41}	1.8×10^8

Table 6.6: Comparison of search-tree sizes on 4×4, 6×6, and 8×8 Rolit for paranoid search and PPNS. Estimated best cases for paranoid search vs. experimental outcome of PPNS.

In Table 6.6 we see that for 4×4 Rolit PPNS creates larger search trees than paranoid search would do in the best-case. For 6×6 Rolit, PPNS generates larger search trees than the best-case for the two-player configuration. For the multi-player cases the outcome is different. The PPNS trees we found here are smaller than the best-case trees of paranoid search. In the three-player case, PPNS is smaller in the order of 10^5, and in the four-player case the factor is ca. 10^{11}. The outcomes differ with the board sizes, because for 4×4 Rolit all 16 possible scores are solved instead of only 5 or 4, and the 6×6 Rolit performs disproofs only. Thus, we may say that in 4×4 Rolit, PPNS performs worse than the estimated best-case of paranoid search. However, PPNS outperforms paranoid search clearly in the 6×6 Rolit multi-player and 8×8 Rolit four-player cases.

In the three-player case, the opponents get two moves, in the four-player case even three moves for every move of the paranoid player. Therefore disproving is simpler for the three-player case than for the two-player case and the four player case is even simpler than the three-player case. Coalition players have an additional advantage by a property of Rolit: they can limit the paranoid player to forced moves or free-choice moves.

The game of 6×6 Rolit is particularly suitable for PPNS because the trees are non-uniform, and the proportion of disproofs in terminal positions is quite high compared to the proportion of proofs, especially for the multi-player configurations in

which only disproofs can be made. Here, PPNS profits from disproving the paranoid player at every OR node by a single disproof for each of its three-level subtrees of AND nodes. Moreover, the prior Monte-Carlo estimates reduce the considered 37 scores for solving to only 5. In addition to that, the most difficult proof, i.e., the proof of the analytical minimum for the player moving last in multi-player configurations could be omitted (because it could be inferred) reducing the scores to be proved to 4. We may therefore state that Rolit can be regarded as a benign game for solving under the paranoid condition.

6.7 Chapter Conclusion and Future Research

We end this chapter by summarizing the results of experiments (Subsection 6.7.1) and giving an outlook on future research (Subsection 6.7.2) based on the findings presented.

6.7.1 Chapter Conclusion

In this chapter, we introduced the notion of solving multi-player games under the paranoid condition and the PPNS algorithm for solving multi-player games under the paranoid condition. We described the multi-player variant of Reversi, Rolit, and applied PPNS to solve 4×4 and 6×6 Rolit for three and for four players as well as 8×8 Rolit for four players. The outcome is that in 4×4 Rolit under the paranoid condition some players can achieve more than the minimum score while in 6×6 and 8×8 Rolit (for four players) no player can achieve more than the minimum score. Moreover, we observed that for 6×6 Rolit and four-player 8×8 Rolit PPNS performed better than we would expect from standard paranoid search. We may conclude that PPNS was able to exploit the non-uniformity of the game tree in multi-player games.

6.7.2 Future Research

Based on our results for 6×6 Rolit we may speculate that the result for solving three-player 8×8 Rolit will be analogous to the results for 6×6 Rolit, i.e., no player can achieve a score higher than 0 (or 1 for the player moving last). Moreover, the task of finding games that are more interesting than Rolit, i.e., more like in 4×4 Rolit than in 6×6 Rolit, remains. Such games may also be more interesting for human players. We might speculate that non-square versions of Rolit might exhibit such properties.

The challenge of solving three-player 8×8 Rolit remains for the future. Parallelization of PPNS and a better evaluation function may at best give bounds on the optimal scores for 8×8 Rolit under the paranoid condition.

Chapter 7

Conclusions and Future Research

In this thesis we investigated solving algorithms for games and game positions. In spite of AI programs becoming stronger at playing games, the task of solving games still remains difficult. Our research utilizes the current developments in the rapidly developing field of Monte-Carlo methods for game-tree search and the continuously evolving field of Proof-Number Search to develop new solving algorithms. The research was guided by the problem statement formulated in Section 1.3 which also provided four research questions addressing the problem statement. In this chapter we formulate conclusions and recommendations for future research based on our work.

Section 7.1 revisits the four research questions one by one and consequentially formulates an answer to the problem statement in Section 7.2. Finally, Section 7.3 suggest directions for future research.

7.1 Conclusions on the Research Questions

The four research questions stated in Chapter 1 concern four topics central to solving game positions with forward search. More precisely, the four questions deal with the following topics that are part of the recent progress in game-tree search: (1) search with Monte-Carlo evaluation, (2) Monte-Carlo Tree Search, (3) parallelized search, and (4) search for multi-player games. The four questions are revisited in the following subsections.

7.1.1 Monte-Carlo Evaluation

Our research question **RQ 1** addressed in Chapter 3 was as follows.

RQ 1 *How can we use Monte-Carlo evaluation to improve Proof-Number Search for solving game positions?*

Chapter 3 introduced a new algorithm, MC-PNS, based on Monte-Carlo evaluation (MCE) within the Proof-Number Search (PNS) framework. An application of the new algorithm and several of its variants to the life-and-death sub-problem of Go were described. It was demonstrated experimentally that given the right setting of parameters MC-PNS outperforms simple PNS. For such a setting, MC-PNS is on average two times faster than PNS and expands four times fewer nodes. We furthermore compared MC-PNS with a pattern-based heuristic for initialization, called PNS_p, and observed that MCE initializes proof and disproof numbers of leaves in PNS better than small patterns. We concluded that the reason for the superior performance of MC-PNS is the flexibility of the MCE. In conclusion, we answer **RQ 1** by stating that MC-PNS constitutes a genuine improvement of PNS that uses MCE to enhance the initialization of proof and disproof numbers.

7.1.2 Monte-Carlo Tree Search Solver

The research question addressed by Chapter 4 was as follows.

RQ 2 *How can the Monte-Carlo Tree Search framework contribute to solving game positions?*

In Chapter 4 we introduced a new Monte-Carlo Tree Search (MCTS) variant, called MCTS-Solver. It adapts the recently developed MCTS framework for solving game positions. MCTS-Solver differs from the traditional MCTS approach in that it can solve positions by proving game-theoretic values. As a result it converges much faster to the best move in narrow tactical lines. We discussed four selection strategies: UCT, "classic" Progressive Bias (PB), and two enhanced PB variants: PB-L1 and PB-L2. Experiments in LOA revealed that PB-L2, which takes the number of losses into account, solved the most positions and was the fastest: it solved positions in three times less time than UCT. When comparing MCTS-Solver to $\alpha\beta$ search, MCTS-Solver required about the same effort as $\alpha\beta$ to solve positions. However, PN^2 was in general 5 times faster than MCTS-Solver. Finally, we found empirically that tree parallelization for MCTS-Solver has a scaling factor of 4 with 8 threads, easily outperforming root parallelization. Thus we may say, that at least during game-play (online) MCTS-Solver is comparable with a standard $\alpha\beta$ search in solving positions. However, for off-line solving positions, PNS is still a better choice. Finally, we concluded that the strength of MCTS-Solver is dependent on enhancements such as PB. Just as for $\alpha\beta$, search enhancements are crucial for the performance. Chapter 4 answers **RQ 2** question by introducing MCTS-Solver and applying it to solving game positions.

7.1.3 Parallel Proof-Number Search

The research question addressed by Chapter 5 was as follows.

RQ 3 *How can Proof-Number Search be parallelized?*

Chapter 5 introduced a parallel Proof-Number Search algorithm for shared memory, called RP–PNS. The parallelization is achieved by threads that select moves

close to the principal variation based on a probability distribution. Furthermore, we adapted RP–PNS for PN^2, resulting in an algorithm called RP–PN^2. The algorithms were tested on LOA positions. For eight threads, the scaling factor found for RP–PN^2 (4.7) was even better than that of RP–PNS (3.5) but this was mainly because the size of the second-level (i.e., PN_2) tree depends on the number of threads used. Based on these results we may conclude that RP–PNS and RP–PN^2 are viable for parallelizing PNS and PN^2, respectively. The chapter answers **RQ 5** by proposing RP–PNS and RP–PN^2 which parallelize Proof-Number Search, and describes the performance of the parallel algorithms.

7.1.4 Paranoid Proof-Number Search

The research question addressed in Chapter 6 was as follows.

RQ 4 *How can Proof-Number Search be applied to multi-player games?*

The starting point of Chapter 4 was the observation that many two-player games have been solved but so far little if any research has been devoted to solving multi-player games. We identified as a reason for this observation that multi-player games generally do not have a unique game-theoretic value or best strategy because their search trees have multiple equilibrium points. Therefore, the straightforward way of solving a game by finding the outcome under optimal play (that is applied in two-player games) cannot be applied to multi-player games directly. We therefore proposed solving multi-player games under the paranoid condition. This is equivalent to finding the optimal score that a player can achieve independently of the other players' strategies. We then proposed Paranoid Proof-Number Search (PPNS) for solving multi-player games under the paranoid condition. In doing so, we made a constructive contribution to the originally still open question formulated in **RQ 4**.

Furthermore, we described the multi-player variant of Reversi, Rolit, and applied PPNS to solve various multi-player versions of 4×4 and 6×6 Rolit, and four-player 8×8 Rolit. The outcome was that in 4×4 Rolit under the paranoid condition some players can achieve more than the minimum score while in 6×6 Rolit and in four-player 8×8 Rolit no player can achieve more than the minimum score. Moreover, we observed that for 6×6 Rolit and four-player 8×8 Rolit PPNS performed better than we would expect from standard paranoid search. We may conclude that PPNS is able to exploit the non-uniformity of the game tree in multi-player games.

Our answer to **RQ 4** is that multi-player games can be solved by finding the best score under the paranoid condition and provided PPNS for finding this score.

7.2 Conclusions on the Problem Statement

Our problem statement was as follows.

PS *How can we improve forward search for solving game positions?*

Taking the answers to the research questions above into account we see that there are several new ways of utilizing the recently so successful Monte-Carlo methods for

solving game positions, and of tackling open questions related to Proof-Number Search for solving game positions. We introduced algorithms that demonstrated (1) how Monte-Carlo evaluation can be used for solving (MC-PNS), (2) how Monte-Carlo Tree Search can be adapted for solving (MCTS-Solver), (3) how Proof-Number Search and its two-level variant PN^2 can be parallelized (RP–PNS, RP–PN^2), and (4) how Proof-Number Search can be used for solving multi-player games (PPNS).

7.3 Recommendations for Future Research

We complete this chapter by listing recommendations for future research grouped by the four algorithms suggested in this thesis.

1. **MC-PNS.** The proposed algorithm, MC-PNS, performed well in the domain of Go in which MCE proved to be useful. Future research should investigate the feasibility of the approach in other solving scenarios which have binary goals and can be searched by AND/OR trees. In particular, it would be of great interest to investigate whether MC-PNS could be successfully applied to one-player games or multi-player games.

2. **MCTS-Solver.** With continuing improvements it is not unlikely that enhanced Monte-Carlo-based approaches may even become an alternative to PNS. As future research, experiments are envisaged in other games to test the performance of MCTS-Solver. One possible next step is testing the method in Go, a domain in which MCTS is already widely used. What makes this a somewhat more difficult task is that additional work is required in enabling perfect endgame knowledge – such as Benson's Algorithm (Benson, 1988; Van der Werf *et al.*, 2003) – in MCTS.

3. **Parallel Proof-Number Search.** Randomized Parallel PNS could still be improved in several ways. First, a combined parallelization at PN_1 and PN_2 trees of RP–PN^2 will be tested on a shared-memory system with more cores. Second, a better distribution for guiding the move selection, possibly by including more information in the nodes, will be tested to reduce the search overhead. Third, the speedup of reducing the number of node locks by pooling will be investigated. Fourth, the concept of the k-most-proving nodes of a proof-number search tree and an algorithm for finding these nodes efficiently for a parallelized tree could be investigated and then applied to parallelizing with a master-servant framework for PNS.

4. **Paranoid Proof-Number Search.** The challenge of solving three-player 8×8 Rolit remains for the future. Parallelization and integration of a better evaluation function will give bounds on the scores for three-player 8×8 Rolit under the paranoid condition. Moreover, Paranoid PNS could be applied for solving other more interesting games under the paranoid condition. Paranoid PNS gives an answer to how game positions can be solved in multi-player games.

This thesis has contributed to the challenge of solving game positions by improving Monte-Carlo methods for search and proof-number algorithms. While we provided PPNS in order to address the class of deterministic multi-player games with *perfect information*, solving deterministic multi-player games with *imperfect information* is still a challenge. We speculate that devising a combination of methods proposed and investigated in this thesis such as parallelization with PPNS, or Paranoid PNS for solving multi-player games under the paranoid condition, with methods used for solving two-player games with imperfect information, such as metapositions (cf. Sakuta, 2001; Favini and Ciancarini, 2007), may help to further advance algorithms for solving difficult game positions.

References

Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *IEEE transactions on PAMI*, Vol. 12, No. 2, pp. 182–193. [4, 5, 7, 19, 20, 43]

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466–473, ACM Press, New York, NY, USA. [52]

Allis, L.V. (1988). A Knowledge-Based Approach of Connect Four: The Game is Over, White to Move Wins. M.Sc. thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Report No. IR-163. [4]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, University of Limburg, The Netherlands. [2, 4, 7, 13, 17, 29, 43, 52, 65, 78]

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

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124. [4, 9, 10, 27, 43, 54, 69, 74]

Allis, L.V., Huntjes, M.P.H., and Herik, H.J. van den (1996). Go-Moku Solved by New Search Techniques. *Computational Intelligence*, Vol. 12, No. 1, pp. 7–24. [3]

Anantharaman, T.S., Campbell, M., and Hsu, F.-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135–143. Also published (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109. [56]

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [24]

Bell, A.G. (1978). *The Machine Plays Chess?* Pergamon Press, Oxford, UK. [3]

Benson, D.B. (1988). Life in the Game of Go. *Computer Games* (ed. D.N.L. Levy), Vol. 2, pp. 203–213. Springer-Verlag, New York, NY, USA. [57, 88]

Billings, D., Castillo, L.P., Schaeffer, J., and Szafron, D. (1999). Using Probabilistic Knowledge and Simulation to Play Poker. *AAAI/IAAI*, pp. 697–703, AAAI Press / MIT Press, Pasadena, CA, USA. [20]

Björnsson, Y. and Marsland, T.A. (1999). Multi-Cut Pruning in Alpha-Beta Search. *Computers and Games (CG'98)* (eds. H.J. van den Herik and H. Iida), Vol. 1558 of *Lecture Notes in Computer Science*, pp. 15–24, Springer-Verlag, Berlin, Germany. [56]

Bolognesi, A. and Ciancarini, P. (2003). Searching over Metapositions in Kriegspiel. *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 325–341, Kluwer Academic Publishers, Boston, MA, USA. [3]

Bouton, C.L. (1902). Nim, a game with a complete mathematical theory. *Annals of Mathematics*, Vol. 3, pp. 35–39. [2]

Bouzy, B. (2006). Associating Shallow and Selective Global Tree Search with Monte Carlo for 9×9 Go. *Proceedings of the 4th Computers and Games Conference (CG'04)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science*, pp. 67–80, Springer, Heidelberg, Germany. [22]

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI oriented Survey. *Artificial Intelligence*, Vol. 132, No. 1, pp. 39–103. [37]

Bouzy, B. and Chaslot, G.M.J.B. (2006). Monte-Carlo Go Reinforcement Learning Experiments. *IEEE Symposium on Computational Intelligence and Games* (eds. S.J. Louis and G. Kendall), pp. 187–194, Willey - IEEE Press, Malden, MA, USA. [37]

Bouzy, B. and Helmstetter, B. (2003). Monte Carlo Developments. *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 159–174, Kluwer Academic Publishers, Boston, MA, USA. [4, 5, 20, 21, 22]

Breuker, D.M. (1998). *Memory versus search*. Ph.D. thesis, Maastricht University, The Netherlands. [9, 14, 65]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Replacement schemes and two-level tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180. [15, 52]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2000). Solving 8×8 Domineering. *Theoretical Computer Science*, Vol. 230, Nos. 1–2, pp. 195–206. [3]

Breuker, D.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Allis, L.V. (2001). A Solution to the GHI Problem for Best-First Search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 121–149. [18]

Brockington, M.G. (1996). A taxonomy of parallel game-tree search algorithms. *ICCA Journal*, Vol. 19, No. 3, pp. 162–174. [59]

Brockington, M.G. and Schaeffer, J. (1997). APHID Game-Tree Search. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 69–92, Universiteit Maastricht. [60]

Brügmann, B. (1993). Monte Carlo Go. Technical report, Physics Department, Syracuse University, Syracuse, NY, USA. [4, 5, 20, 21, 24]

Bullock, N. (2002). Domineering: Solving Large Combinatorial Search Spaces. *ICGA Journal*, Vol. 25, No. 2, pp. 67–84. [4]

Buro, M. (1999). Toward Opening Book Learning. *ICCA Journal*, Vol. 22, No. 2, pp. 98–102. [64]

Campbell, M.S. (1985). The graph-history interaction: On ignoring position history. *Proceedings of the 1985 ACM annual conference on the range of computing: mid-80's perspective*, pp. 278–280, ACM, New York, NY, USA. [9]

Cazenave, T. (2003). Metarules to Improve Tactical Go Knowledge. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 173–188. [30]

Cazenave, T. (2009). Nested Monte-Carlo Search. *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)* (ed. C. Boutilier), pp. 456–461, AAAI Press, Pasadena, CA, USA. [22]

Cazenave, T. and Borsboom, J. (2007). Golois Wins Phantom Go Tournament. *ICGA Journal*, Vol. 30, No. 3, pp. 165–166. [22, 43]

Cazenave, T. and Jouandeau, N. (2007). On the Parallelization of UCT. *Computer Games Workshop (CGW'07)* (eds. H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd), Vol. 07-06 of *MICC Technical Report Series*, pp. 93–101, Maastricht University, Maastricht, The Netherlands. [25, 50, 64]

Cazenave, T. and Jouandeau, N. (2008). A Parallel Monte-Carlo Tree Search Algorithm. *Proceedings of the 6th Computers and Games Conference (CG'08)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 72–80, Springer, Heidelberg, Germany. [64]

Cazenave, T. and Saffidine, A. (2009). Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, Vol. 23, Nos. 2–3, pp. 183–202. (in French). [22, 43]

Chaslot, G.M.J.B. (2008). IA-Go Challenge: MoGo vs. Catalin Taranu. *ICGA Journal*, Vol. 31, No. 2, p. 126. [2, 19]

Chaslot, G.M.J.B. (2010). *Monte-Carlo Tree Search.* Ph.D. thesis, Maastricht University, The Netherlands. [105]

Chaslot, G.M.J.B., Jong, S. de, Saito, J-T., and Uiterwijk, J.W.H.M. (2006). Monte-Carlo Tree Search in Production Management Problems. *Proceedings of the 18th BeNeLux Artificial Intelligence Conference (BNAIC'06)* (eds. P.Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 91–98, Twente University Press, Enschede, The Netherlands. [19, 22]

Chaslot, G.M.J.B., Winands, M.H.M., and Herik, H.J. van den (2008a). Parallel Monte-Carlo Tree Search. *Proceedings of the 6th Conference on Computers and Games (CG'08)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 60–71, Springer, Berlin, Germany. [25, 50, 64]

Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2008b). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [22, 25, 32, 47]

Chiang, S.-H., Wu, I-C., and Lin, P.-H. (2010). On Drawn K-In-A-Row Games. *Proceedings of the 13th Advances in Computers Games Conference (ACG'09)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *Lecture in Computer Science*, pp. 158–169, Springer-Verlag, Heidelberg, Germany. [4]

Coulom, R. (2007a). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture in Computer Science*, pp. 72–83, Springer, Heidelberg, Germany. [2, 4, 5, 7, 9, 19, 22, 24, 43, 49]

Coulom, R. (2007b). Computing "Elo Ratings" of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 199–208. [25]

Coulom, R. and Chen, K.-H. (2006). Crazy Stone Wins 9×9 Go Tournament. *ICGA Journal*, Vol. 29, No. 3, p. 96. [22]

Davies, J. (1977). *The Rules and Elements of Go.* Ishi Press, Tokyo, Japan. [105]

DeCoste, D. (1998). The Significance of Kasparov versus Deep Blue and the Future of Computer Chess. *ICCA Journal*, Vol. 21, No. 1, pp. 33–43. [2]

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143. [56]

Enzenberger, M. and Müller, M. (2009). Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search. Technical Report 08, Dept. of Computing Science, University of Alberta, Edmonton, Canada. [22]

Enzenberger, M. and Müller, M. (2010). A lock-free multithreaded Monte-Carlo tree search algorithm. *Proceedings of the 13th Advances in Computers Games Conference (ACG'09)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *Lecture in Computer Science*, pp. 14–20, Springer, Berlin, Germany. [25, 51]

Favini, G.P. and Ciancarini, P. (2007). Representing Kriegspiel states with metapositions. *Proceedings of the 20th International Joint Conference on AI (IJCAI'07)*, pp. 2450–2455, Morgan Kaufmann, San Francisco, CA, USA. [89]

Ferguson, T.S. (1992). Mate with bishop and knight in Kriegspiel. *Theoretical Computer Science*, Vol. 96, No. 2, pp. 389–403. [3]

Finnsson, H. and Björnsson, Y. (2008). Simulation-based approach to general game playing. *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'08)* (eds. D. Fox and C.P. Gomes), pp. 259–264, AAAI Press, Menlo Park, CA, USA. [22]

Frank, I., Basin, D.A., and Matsubara, H. (1998). Finding Optimal Strategies for Imperfect Information Games. *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)* (eds. D. Allison and T. Balch), pp. 500–507, AAAI Press, Menlo Park, CA, USA. [20]

Fujii, M., Kita, H., Murata, T., Hashimoto, J., and Iida, H. (2006). Four-person Reversi Yonin. *Game Informatics GI-15*, Vol. 2006, No. 23, pp. 73–80. (in Japanese). [76, 78]

Gasser, R. (1996). Solving Nine Men's Morris. *Computational Intelligence*, Vol. 12, pp. 24–41. [3]

Gelly, S. and Silver, D. (2008). Achieving master level play in 9×9 computer Go. *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'08)* (eds. D. Fox and C.P. Gomes), pp. 1537–1540, AAAI Press, Menlo Park, CA, USA. [22, 24, 25]

Ginsberg, M.L. (1999). GIB: Steps Toward an Expert-Level Bridge-Playing Program. *Proceedings of the 16th Joint Conference on Artificial Intelligence (IJCAI'99)*, pp. 584–593. [20]

Graepel, T., Goutrie, M., Krüger, M., and Herbrich, R. (2001). Learning on graphs in the game of Go. *Proceedings of the 11th International Conference on Artificial Neural Networks (ICANN'01)* (eds. G. Dorffner, H. Bischof, and K. Hornik), Vol. 2130 of *Lecture Notes in Computer Science*, pp. 347–352, Springer, Heidelberg, Germany. [37]

Greenblatt, R., Eastlake, D., and Croker, S. (1967). The Greenblatt Chess program. *Proceedings of the Fall Joint Computer Conference*, pp. 801–810. Reprinted (1988) in Computer Chess Compendium (ed. D.N.L. Levy), pp. 56-66. Batsford, London, UK. [9]

Grottling, G. (1985). Problem-solving ability tested. *ICCA Journal*, Vol. 8, No. 2, pp. 107–110. [3]

Hasbro International Inc. (1999). *Rolit*. Hasbro International Inc., Pawtucket, R.I., USA. [75]

Hayward, R., Björnsson, Y., and Johanson, M. (2005). Solving 7×7 Hex with domination, fill-in, and virtual connections. *Theoretical Computer Science*, Vol. 349, No. 2, pp. 123–139. [4]

Henderson, P., Arneson, B., and Hayward, R.B. (2009). Solving 8×8 Hex. *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)* (ed. C. Boutilier), pp. 505–510, AAAI Press, Pasadena, CA, USA. [2]

Herik, H.J. van den and Winands, M.H.M. (2008). Proof-Number search and its variants. *Oppositional Concepts in Computational Intelligence* (eds. H.R. Tizhoosh and M. Ventresca), Vol. 155 of *Studies in Computational Intelligence*, pp. 91–118. Springer, Heidelberg, Germany. [7, 65, 74]

Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311. [2, 69]

Heule, M.J.H. and Rothkrantz, L.J.M. (2007). Solving Games - Dependence of Applicable Solving Procedures. *Science of Computer Programming*, Vol. 67, No. 1, pp. 105–124. [69]

Hsu, F.-h. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. [70]

Huizinga, J. (1955). *Homo ludens; A study of the play-element in culture*. Beacon Press, Boston, MA, USA. [1]

Irving, G., Donkers, H.H.L.M., and Uiterwijk, J.W.H.M. (2000). Solving Kalah. *ICGA Journal*, Vol. 23, No. 3, pp. 139–148. [3]

Kaneko, T. (2010). Parallel Depth First Proof Number Search. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, (AAAI'10)* (eds. M. Fox and D. Poole), pp. 95–100, AAAI Press, Menlo Park, CA, USA. [5, 60]

Kishimoto, A. (2005). *Correct and efficient search algorithms in the presence of repetitions*. Ph.D. thesis, University of Alberta, Canada. [19]

Kishimoto, A. and Kotani, Y. (1999). Parallel AND/OR tree search based on proof and disproof numbers. *Proceedings of the 5th Game Programming Workshop*, Vol. 99(14) of *IPSJ Symposium Series*, pp. 24–30, Hakone, Japan. [5, 59, 60]

Kishimoto, A. and Müller, M. (2003). DF-PN in Go: Application to the one-eye problem. *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 125–141, Kluwer Academic Publishers, Boston, MA, USA. [10, 17, 18, 19]

Kishimoto, A. and Müller, M. (2005a). Search versus knowledge for solving life and death problems in Go. *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)* (eds. M.M. Veloso and S. Kambhampati), pp. 1374–1379, AAAI Press / MIT Press, Menlo Park, CA, USA. [3, 17, 30]

Kishimoto, A. and Müller, M. (2005b). A Solution to the GHI problem for depth-first proof-number search. *Information Sciences*, Vol. 175, pp. 296–314. [19]

Kloetzer, J., Müller, M., and Bouzy, B. (2008). A Comparative Study of Solvers in Amazons Endgames. *2008 IEEE Symposium on Computational Intelligence and Games (CIG'08)* (eds. P. Hingston and L. Barone), pp. 378–384. [43, 44]

Knuth, D.E. and Moore, R.W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [4, 9, 45, 70]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Proceedings of the 17th European Conference on Machine Learning (ECML'06)* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, Springer, Heidelberg, Germany. [2, 4, 5, 7, 9, 19, 22, 24, 43, 47]

Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved Monte-Carlo Search. http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf. [43]

Korf, R.E. (1991). Multi-Player Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 48, No. 1, pp. 99–111. [5, 70]

Kupferschmid, S. and Helmert, M. (2007). A Skat player based on Monte-Carlo Simulation. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 135–147, Springer, Berlin, Germany. [20]

Lindner, L. (1985). A critique of Problem-Solving Ability. *ICCA Journal*, Vol. 8, No. 3, pp. 182–185. [3]

Lorentz, R.J. (2008). Amazons Discover Monte-Carlo. *Proceedings of the 6th Computers and Games Conference (CG'08)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 13–24, Springer, Berlin, Germany. [21, 22, 43]

Lorenz, U. (2001). Parallel Controlled Conspiracy Number Search. *Euro-Par* (eds. M. Monien and R. Feldmann), Vol. 2400 of *Lecture Notes in Computer Science*, pp. 420–430, Springer, Heidelberg, Germany. [60]

Luce, R. and Raiffa, H. (1957). *Games and decisions*. John Wiley & Sons, New York, NY, USA. [71]

Luckhardt, C. and Irani, K.B. (1986). An Algorithmic Solution of N-Person Games. *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI'86)*, pp. 158–162, Morgan Kaufmann, San Francisco, CA, USA. [5, 69, 70]

Marsland, T.A. (1983). Relative Efficiency of Alpha-Beta Implementations. *Proceedings of the 8^{th} International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 763–766, Karlsruhe, Germany. [52]

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. [57]

McAllester, D.A. (1985). A New Procedure for Growing Min-Max Trees. Technical report, Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA. [10]

McAllester, D.A. (1988). Conspiracy numbers for min-max search. *Artificial Intelligence*, Vol. 35, No. 3, pp. 287–310. [10, 60]

Metropolis, N. and Ulam, S. (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, Vol. 44, No. 247, pp. 335–341. [19]

Moldenhauer, C. (2009). Game tree search algorithms for the game of Cops and Robber. M.Sc. thesis, School of Computing Science, University of Alberta, Alberta, Canada. [74]

Müller, M. (2002). Computer Go. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 145–179. [20]

Nagai, A. (1998). A new depth-first search algorithm for AND/OR trees. *Proceedings of the Complex Games Lab Workshop* (eds. H. Matsubara, Y. Kotani, T. Takizawa, and A. Yoshikawa), pp. 40–45, ETL, Tsuruoka, Japan. [15, 17]

Nagai, A. (1999). Application of df-pn+ to Othello endgames. *Proceedings of the 5th Game Programming Workshop*, Vol. 99(14) of *IPSJ Symposium Series*, pp. 16–23. [15, 16, 17, 81]

Nagai, A. (2002). *Df-pn algorithm for searching AND/OR trees and its applications*. Ph.D. thesis, University of Tokyo, Japan. [3, 10, 15, 16, 18, 19]

Nalimov, E.V., Haworth, G.M^{c}C., and Heinz, E.A. (2000). Space-Efficient Indexing of Chess Endgame Tables. *ICGA Journal*, Vol. 23, No. 3, pp. 148–162. [4]

Nash, J. (1952). Some games and machines for playing them. Technical Report D-1164, Rand Corp. [2]

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, USA. [10]

Nijssen, J.A.M. and Winands, M.H.M. (2010). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG'10)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), (in print). [49]

Palay, A.J. (1983). *Searching with probabilities*. Ph.D. thesis, Carnagie Mellon University, PA, USA. [9]

Patashnik, O. (1980). Qubic: $4 \times 4 \times 4$ Tic-Tac-Toe. *Mathematics Magazine*, Vol. 53, No. 4, pp. 202–216. [3]

Pawlewicz, J. and Lew, Ł. (2007). Improving depth-first pn-search: $1+\epsilon$ trick. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 160–170, Computers and Games, Springer, Heidelberg, Germany. [17, 18, 51, 65]

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman, Boston, MA, USA. [8, 20]

Plaat, A. (1996). *Research Re: Search & Re-search*. Ph.D. thesis, Erasmus Universiteit Rotterdam, Rotterdam, The Netherlands. [73]

Reinefeld, A. (1983). An Improvement to the Scout Search Tree Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14. [52]

Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, Vol. 58, No. 5, pp. 527–535. [22, 24]

Romein, J.W. and Bal, H.E. (2003). Solving Awari with parallel retrograde analysis. *IEEE Computer*, Vol. 36, No. 10, pp. 26–33. [3]

Saito, J.-T. and Winands, M.H.M. (2010). Paranoid Proof-Number Search. *In Proceedings of the Computational Intelligence and Games Conference (CIG'10)* (eds. G.N. Yannakakis and J. Togelius), pp. 203–210, IEEE Press. [69]

Saito, J-T., Chaslot, G.M.J.B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2006a). Pattern Knowledge for Proof-Number Search in Computer Go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence (BNAIC'06)* (eds. P.Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 275–281, University of Twente Press, Enschede, The Netherlands. [27]

Saito, J-T., Chaslot, G.M.J.B., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Winands, M.H.M. (2006b). Developments in Monte-Carlo Proof-Number Search. *Proceedings of the 11th Game Programming Workshop* (eds. H. Matsubara, T. Ito, and T. Nakamura), pp. 27–31, Information Processing Society of Japan, Tokyo, Japan. [27]

Saito, J-T., Chaslot, G.M.J.B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2007a). Monte-Carlo Proof-Number Search. *Proceedings of the 5th Computers and Games Conference (CG'06)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 50–61, Springer, Berlin, Germany. [7, 27]

Saito, J-T., Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2007b). Grouping nodes for Monte-Carlo Tree Search (BNAIC'07). *Proceedings of the 19th Belgian-Dutch Conference on Artificial Intelligence* (eds. M.M. Dastani and E. de Jong), pp. 276–283, Utrecht University, Utrecht, The Netherlands. [25, 7]

Saito, J-T., Winands, M.H.M., and Herik, H.J. van den (2009). Randomized Parallel Proof-Number Search. *Proceedings of the 21st BeNeLux Conference on Artificial Intelligence (BNAIC'09)* (eds. T. Calders, K. Tuyls, and M. Pechenizkiy), pp. 365–366, TU/e Eindhoven, Eindhoven, The Netherlands. [59]

Saito, J-T., Winands, M.H.M., and Herik, H.J. van den (2010). Randomized Parallel Proof-Number Search. *Proceedings of the 13th Advances in Computers Games Conference (ACG'09)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science*, pp. 75–87, Springer-Verlag, Heidelberg, Germany. [51, 59]

Sakuta, M. (2001). *Deterministic Solving of Problems with Uncertainty*. Ph.D. thesis, Shizuoka University, Hamamatsu, Japan. [89]

Sakuta, M. and Iida, H. (2000). Solving Kriegspiel-like Problems: Exploiting a Transposition Table. *ICGA Journal*, Vol. 23, No. 4, pp. 218–229. [3]

Sakuta, M., Hashimoto, T., Nagashima, J., Uiterwijk, J.W.H.M., and Iida, H. (2003). Application of the Killer-tree Heuristic and the Lamba-Search Method to Lines of Action. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 141–155. [51]

Schadd, M.P.D., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bergsma, M.H.J. (2008). Best Play in Fanorona leads to draw. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 369–387. [3, 4, 22]

Schaeffer, J. (1989). Conspiracy Numbers. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 199–217, Elsevier, Maryland Heights, MO, USA. [10]

Schaeffer, J. (1990). Conspiracy Numbers. *Artificial Intelligence*, Vol. 43, No. 1, pp. 67–84. [10]

Schaeffer, J. and Plaat, A. (1996). New Advances in Alpha-Beta Searching. *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, pp. 124–130. ACM Press, New York, NY, USA. [52]

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [3, 4, 10]

Schrüfer, G. (1989). A Strategic Quiescence Search. *ICCA Journal*, Vol. 12, No. 1, pp. 3–9. [52]

Seo, M., Iida, H., and Uiterwijk, J.W.H.M. (2001). The PN*-Search algorithm: application to Tsume Shogi. *Artificial Intelligence*, Vol. 129, Nos. 1–2, pp. 253–277. [10, 14]

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

Sheppard, B. (2002). *Towards Perfect Play of Scrabble*. Ph.D. thesis, Maastricht University, Maastricht, The Netherlands. [20]

Shoham, Y. and Toledo, S. (2002). Parallel randomized best-first minimax search. *Artificial Intelligence*, Vol. 137, Nos. 1–2, pp. 165–196. [51, 61, 63]

Smith, S.J.J., Nau, D., and Throop, T.A. (1998). Computer Bridge: A Big Win for AI Planning. *AI Magazine*, Vol. 19, No. 2, pp. 93–106. [20]

Ströhlein, T. (1970). *Untersuchungen über kombinatorische Spiele*. Ph.D. thesis, Technische Hochschule München, München, Germany. (in German). [4]

Sturtevant, N.R. (2003a). *Multiplayer games: Algorithms and approaches*. Ph.D. thesis, University of California, Los Angeles, CA, USA. [71, 72]

Sturtevant, N.R. (2003b). A Comparison of Algorithms for Multi-player Games. *Proceedings of the 3rd Computers and Games Conference 2002 (CG'02)* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 108–122, Springer, Heidelberg, Germany. [73]

Sturtevant, N.R. (2003c). Last-Branch and Speculative Pruning Algorithms for Max^{n}. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)* (eds. G. Gottlob and T. Walsh), pp. 669–678, Morgan Kaufmann, San Francisco, CA, USA. [73]

Sturtevant, N.R. (2008). An Analysis of UCT in Multi-player Games. *Proceedings of the 6th Computers and Games Conference (CG'08)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 37–49, Springer, Heidelberg, Germany. [22]

Sturtevant, N.R. and Korf, R.E. (2000). On Pruning Techniques for Multi-Player Games. *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'00)* (eds. H.A. Kautz and B. Porter), pp. 201–207, MIT Press, Cambridge, MA, USA. [5, 69, 70, 72, 73]

Takizawa, T. and Grimbergen, R. (2000). Review: Computer Shogi through 2000. *Proceedings of the 2nd Computers and Games Conference (CG'00)* (eds. H.J. van den Herik and H. Iida), Vol. 2063 of *Lecture Notes in Computer Science*, pp. 431–442, Springer, Heidelberg, Germany. [3]

Tesauro, G. and Galperin, G.R. (1997). On-line policy improvement using Monte-Carlo search. *Advances in Neural Information Processing Systems*, Vol. 9, pp. 1068–1074. [20]

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131–139. [4]

Tromp, J. (2008). Solving Connect-4 on Medium Board Sizes. *ICGA Journal*, Vol. 31, No. 2, pp. 110–112. [3]

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-tree Search Algorithm based on Realization Probability. *ICGA Journal*, Vol. 25, No. 3, pp. 132–144. [47, 56]

Turing, A.M. (1953). Chess. *Faster than thought: A symposium of digital computing machines* (ed. B.W. Bowden). Pitman Publishing, London, England. [1]

Uiterwijk, J.W.H.M. and Herik, H.J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No. 1, pp. 43–58. [3]

Wágner, J. and Virág, I. (2001). Solving Renju. *ICGA Journal*, Vol. 24, No. 1, pp. 30–34. [3]

Werf, E.C.D. van der (2004). *AI techniques for the game of Go*. Ph.D. thesis, Maastricht University, The Netherlands. [37, 105]

Werf, E.C.D. van der and Winands, M.H.M. (2009). Solving Go for Rectangular Boards. *ICGA Journal*, Vol. 32, No. 2, pp. 77–88. [10]

Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003). Solving Go on Small Boards. *ICGA Journal*, Vol. 26, No. 2, pp. 92–107. [2, 3, 10, 57, 88]

Winands, M.H.M. (2004). *Informed search in complex games*. Ph.D. thesis, Maastricht University, The Netherlands. [17, 52, 106]

Winands, M.H.M. (2008). 6×6 LOA is Solved. *ICGA Journal*, Vol. 31, No. 3, pp. 234–238. [2, 52, 73]

Winands, M.H.M. and Björnsson, Y. (2008). Enhanced Realization Probability Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 329–342. [49, 56]

Winands, M.H.M. and Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. *Proceedings of the 13th Advances in Computers Games Conference (ACG'09)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science*, pp. 33–44, Springer, Heidelberg, Germany. [21, 43, 50]

Winands, M.H.M. and Herik, H.J. van den (2006). MIA: A World Champion LOA Program. *The 11th Game Programming Workshop in Japan (GPW 2006)*, pp. 84–91. [50, 52]

Winands, M.H.M., Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003a). An Evaluation Function for Lines of Action. *Advances in Computer Games 10 (ACG'03): Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 249–260, Kluwer Academic Publishers, Boston, MA, USA. [52]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003b). PDS-PN: A new proof-number search algorithm: Application to Lines of Action. *Proceedings of the 3rd Computers and Games Conference 2002 (CG'02)* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 170–185, Springer, Heidelberg, Germany. [16, 51, 65]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2004). An effective two-level Proof-Number Search algorithm. *Theoretical Computer Science*, Vol. 313, No. 3, pp. 511–525. [10, 16]

Winands, M.H.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Werf, E.C.D. van der (2005). Enhanced Forward Pruning. *Information Sciences*, Vol. 175, No. 4, pp. 315–329. [56]

Winands, M.H.M., Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2006). The Relative History Heuristic. *Computers and Games* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science (LNCS)*, pp. 262–272, Springer-Verlag, Berlin, Germany. [52]

Winands, M.H.M., Björnsson, Y., and Saito, J-T. (2008). Monte-Carlo Tree Search Solver. *Proceedings of the 6th Computers and Games Conference (CG'08)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 25–36, Springer, Berlin, Germany. [7, 22, 43, 44, 46]

Wolf, T. (1994). The program GoTools and its computer-generated Tsume Go database. *Proceedings of the 1st Game Programming Workshop* (ed. H. Matsubara), pp. 84–96, Information Processing Society of Japan, Tokyo, Japan. [3]

Wolf, T. (2000). Forward pruning and other heuristic search techniques in Tsume Go. *Information Sciences*, Vol. 122, No. 1, pp. 59–76. [30]

Wolf, T. and Shen, L. (2007). Checking Life-and-Death problems in Go I: the program ScanLD. *ICGA Journal*, Vol. 30, No. 2, pp. 67–74. [3]

Wu, I-C. and Huang, D.-Y. (2006). A New Family of k-in-a-Row Games. *Proceedings of the 11th Advances in Computer Games Conference* (eds. H.J. van den Herik, S. Hsu, T. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science*, pp. 180–194, Springer. [4]

Zhang, P. and Chen, K.-H. (2007). Monte-Carlo Go Tactic Search. *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)* (eds. P. Wang and others), pp. 662–670, World Scientific Publishing Co. Pte. Ltd. [44]

Zhang, P. and Chen, K.-H. (2008). Monte Carlo Go capturing tactic search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 359–367. [7]

Zobrist, A.L. (1990). A new hashing method with application for game playing. *ICCA Journal*, Vol. 13, No. 2, pp. 69–73. [32]

Appendix A

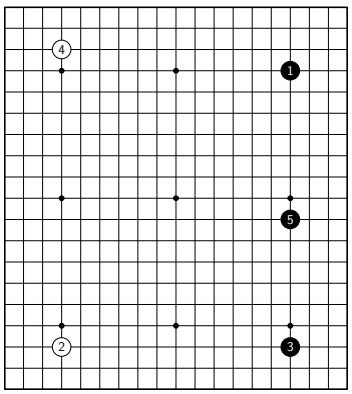
Rules of Go and LOA

This Appendix gives the rules for the games of Go and LOA on which some of the algorithms presented in this thesis have been tested.

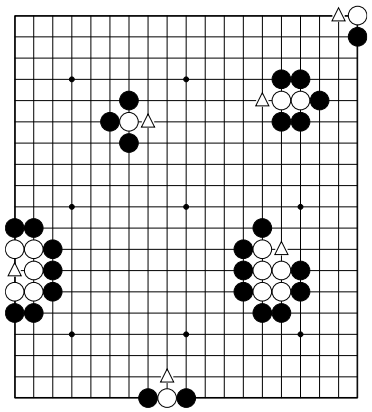
A.1 Go Rules

It is beyond the scope of the thesis to explain all rules in detail. For a more elaborate introduction we refer to Van der Werf (2004). A basic set of rules, adapted from Davies (1977), is given below. Figure A.1 is adopted from Chaslot (2010).

1. The square grid board is empty at the outset of the game. Usually the grid contains 19×19 intersections, though 9×9 is used as well.
2. There are two players, called Black and White.
3. Black makes the first move, alternating with White (cf. Figure A.1(a)).
4. A move consists of placing one stone of one's own color on an empty intersection on the board.
5. A player may pass his turn at any time.
6. A stone or a set of stones of one color that are orthogonally connected by grid lines is captured and removed from the board when all the intersections directly adjacent to it are occupied by the opponent (cf. Figure A.1(b)).
7. No stone may be played to repeat a former board position.
8. Two consecutive passes end the game.
9. A player's territory consists of all the board points he has either occupied or surrounded.
10. The player with more territory wins.



(a) From the empty board, Black and White play alternately on the intersections.



(b) Stones that are surrounded are captured. Black can capture white stones by playing on the marked intersections.

Figure A.1: Two illustrations for the rules of Go (adopted from Chaslot, 2010).

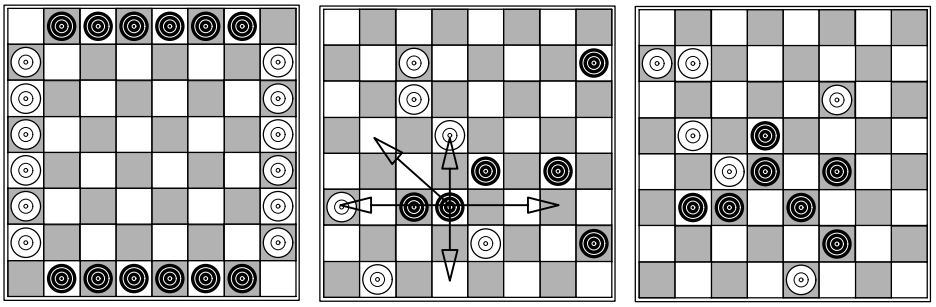
A.2 LOA Rules

The LOA rules set is based on Winands (2004). LOA is played on an 8×8 board by two sides, Black and White. Each side has twelve (checker) pieces at its disposal. Game play is specified by the following rules:[1]

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

[1]These are the rules used at the Computer Olympiads and at the MSO World Championships. In some books, magazines or tournaments, there may be a slight variation on rules 2, 7, 8, and 9.

8. A player that cannot move must pass.
9. If a position with the same player to move occurs for the third time, the game is drawn.



(a) The initial position. (b) Example of possible moves. (c) A terminal position.

Figure A.2: Three illustrations for the rules of LOA (adopted from Winands, 2004).

In Figure A.2(b) the possible moves of the black piece on **d3** (using the same coordinate system as in chess) are shown by arrows. The piece cannot move to **f1** because its path is blocked by an opposing piece. The move to **h7** is not allowed because the square is occupied by a black piece.

Index

Summary

Humans enjoy playing games not only to satisfy their desire for entertainment but also because they seek an intellectual challenge. One obvious challenge in games is defeating an opponent. The AI equivalent to this challenge is the design of strong game-playing programs. Another challenge in games is finding the result of a game position, for instance whether a Chess position is a win or a loss. The AI equivalent to this challenge is the design of algorithms that solve positions. While game-playing programs have become much stronger over the years, solving games still remains a difficult task today and has therefore been receiving attention continuously.

The topic of the thesis is the difficult task of solving game positions. Our research utilizes current developments in the rapidly developing field of Monte-Carlo methods for game-tree search and the continuously evolving field of Proof-Number Search to develop new solving algorithms. To that end, the here described research contributes and tests new *forward-search* algorithms, i.e., algorithms that explore a search space by starting from a game position and developing a search tree from top to bottom. The new algorithms are empirically evaluated on three games, (1) Go, (2) Lines of Action (LOA), and (3) Rolit.

Chapter 1 provides an introduction. It describes the place of games in the domain of AI and provides the notion of *solving games* and *solving game positions*. The chapter introduces the problem statement:

PS *How can we improve forward search for solving game positions?*

Moreover, Chapter 1 provides four research questions that address four aspects of the problem statement that are central to solving game positions with forward search. The four questions deal with the following topics, which are part of the recent progress in the research on game-tree search: (1) search with Monte-Carlo evaluation, (2) Monte-Carlo Tree Search, (3) parallelized search, and (4) search for multi-player games.

The aim of Chapter 2 is to provide an overview of search techniques related and relevant to the solving algorithms presented in later chapters. It introduces basic concepts and gives notational conventions. It devotes particular detail to two topics: proof-number algorithms and Monte-Carlo techniques. Proof-number algorithms are stressed because they are well-studied standard techniques for solving. The reason for paying particular attention to Monte-Carlo techniques is that they have recently

contributed substantially to the field of games and AI.

Chapter 3 introduces a new algorithm, MC-PNS, that combines Monte-Carlo evaluation (MCE) with Proof-Number Search (PNS). Thereby the first research question is addressed.

RQ 1 *How can we use Monte-Carlo evaluation to improve Proof-Number Search for solving game positions?*

An application of the new algorithm and several of its variants to the life-and-death sub-problem of Go is described. It is demonstrated experimentally that given the right setting of parameters MC-PNS outperforms simple PNS. Moreover, MC-PNS is compared with a pattern-based heuristic for initialization leaf nodes in the search tree of PNS. The result is that MC-PNS outperforms the purely pattern-based initialization. We conclude that the reason for the superior performance of MC-PNS is the flexibility of the MCE.

Chapter 4 introduces a novel Monte-Carlo Tree Search (MCTS) variant, called MCTS-Solver, addressing the second research question.

RQ 2 *How can the Monte-Carlo Tree Search framework contribute to solving game positions?*

The chapter adapts the recently developed MCTS framework for solving game positions. MCTS-Solver differs from traditional MCTS in that it can solve positions by propagating game-theoretic values. As a result it converges faster to the best move in narrow tactical lines. Experiments in the game of Lines of Action (LOA) show that MCTS-Solver with a particular selection strategy solves LOA positions three times faster than MCTS-Solver using the standard selection strategy UCT. When comparing MCTS-Solver to $\alpha\beta$ search, MCTS-Solver requires about the same effort as $\alpha\beta$ to solve positions. Moreover, we observe that PN^2 (a two-level Proof-Number Search algorithm) is still five times faster than MCTS-Solver. Additionally, we show that tree parallelization for MCTS-Solver has a scaling factor of 4 with 8 threads, easily outperforming root parallelization. We conclude that at least during game-play (online) MCTS-Solver is comparable with a standard $\alpha\beta$ search in solving positions. However, for off-line solving positions, PNS is still a better choice.

Chapter 5 introduces a parallel Proof-Number Search algorithm for shared memory, called RP–PNS. Thereby, we answer the third research question.

RQ 3 *How can Proof-Number Search be parallelized?*

The parallelization is achieved by threads that select moves close to the principal variation based on a probability distribution. Furthermore, we adapted RP–PNS for PN^2, resulting in an algorithm called RP–PN^2. The algorithms are evaluated on LOA positions. For eight threads, the scaling factor found for RP–PN^2 (4.7) is even better than that of RP–PNS (3.5) but mainly achieved because the size of the second-level (i.e., PN_2) tree depends on the number of threads used. Based on these

results the chapter concludes that RP–PNS and RP–PN^2 are viable for parallelizing PNS and PN^2, respectively.

Chapter 6 focuses on solving multi-player games. The chapter gives an answer to the fourth research question.

RQ 4 *How can Proof-Number Search be applied to multi-player games?*

The starting point of Chapter 6 is the observation that many two-player games have been solved but virtually no research has been devoted to solving multi-player games. We identify as a reason for this observation that multi-player games generally do not have a unique game-theoretic value or strategy because their search trees have multiple equilibrium points. Therefore, the straightforward way of solving a game by finding the outcome under optimal play (that is applied in two-player games) cannot be applied to multi-player games directly. We therefore propose solving multi-player games under the paranoid condition. This is equivalent to finding the optimal score that a player can achieve independently of the other players' strategies. We then propose Paranoid Proof-Number Search (PPNS) for solving multi-player games under the paranoid condition.

The chapter describes the multi-player variant of the game of Reversi, Rolit, and discusses how to apply PPNS to solve various multi-player variants of 4×4 and 6×6 Rolit, and four-player 8×8 Rolit. The outcome is that in 4×4 Rolit under the paranoid condition some players can achieve more than the minimum score while in 6×6 Rolit and in four-player 8×8 Rolit no player can achieve more than the minimum score. However, we observe that for 6×6 Rolit and four-player 8×8 Rolit PPNS performs better than we would expect from standard paranoid search. The chapter concludes by stating that PPNS is able to exploit the non-uniformity of the game tree in multi-player games.

Chapter 7 concludes the thesis and gives an outlook to open questions and directions for future research. While the thesis provides PPNS addressing the class of deterministic multi-player games with *perfect information*, we end by pointing out that solving deterministic multi-player games with *imperfect information* still remains a challenge and we briefly speculate how this problem can be tackled.

Samenvatting

Mensen genieten van abstracte spelen niet alleen om hun verlangen naar amusement te bevredigen maar ook omdat ze een intellectuele uitdaging zoeken. Een voor de hand liggende uitdaging in spelen is het verslaan van de tegenstander. De corresponderende AI uitdaging is het ontwerpen van sterke spelprogramma's. Een andere uitdaging in spelen is het vinden van het resultaat van een spelpositie, bijvoorbeeld of een schaakstelling een winst of verlies is. De corresponderende AI uitdaging is het ontwerpen van algoritmen die posities oplossen. Terwijl spelprogramma's door de jaren heen veel sterker geworden zijn, blijft het oplossen van spelen nog steeds een moeilijke taak en heeft daarom voortdurend aandacht gekregen.

Het onderwerp van dit proefschrift is de moeilijke taak van het oplossen van spelposities. Ons onderzoek maakt gebruik van actuele ontwikkelingen in het zich snel ontwikkelende veld van Monte-Carlo methoden voor spelboom zoekprocessen en het continu ontwikkelende veld van Proof-Number Search om nieuwe algoritmen te ontwikkelen die oplossen. Het hier beschreven onderzoek draagt bij en test nieuwe *voorwaarts zoekalgoritmen*, dat wil zeggen, algoritmen die een zoekruimte verkennen door vanuit een positie een zoekboom te ontwikkelen. De nieuwe algoritmen worden empirisch geëvalueerd in drie spelen, (1) Go, (2) Lines of Action (LOA), en (3) Rolit.

Hoofdstuk 1 geeft een inleiding. Het beschrijft de plaats van spelen in het AI domein en geeft de begrippen van *het oplossen van spelen* en *het oplossen van spelposities*. Het hoofdstuk introduceert de volgende probleemstelling:

PS *Hoe kunnen we beter voorwaarts zoeken bij het oplossen van spelposities?*

Om de probleemstelling te beantwoorden hebben we vier onderzoeksvragen geformuleerd over onderwerpen op het gebied van het oplossen van spelposities door middel van voorwaarts zoeken. Ze gaan over (1) het zoeken met Monte-Carlo evaluaties, (2) Monte-Carlo Tree Search, (3) geparallelliseerde zoekprocessen, en (4) zoekprocessen voor meerdere spelers.

Hoofdstuk 2 geeft een overzicht van zoektechnieken die gerelateerd en relevant zijn voor de zoekalgoritmen die in de latere hoofdstukken worden gegeven. Het hoofdstuk introduceert basisbegrippen en geeft conventies voor notatie. Er wordt dieper in gegaan op twee onderwerpen: Proof-Number algoritmen en Monte-Carlo technieken. Proof-Number algoritmen worden benadrukt omdat ze goed bestudeerde standaard technieken zijn voor oplossen van spelposities. De reden voor de aandacht

voor Monte-Carlo technieken is dat ze een aanzienlijk bijdrage hebben geleverd in het gebied van spelen en AI.

Hoofdstuk 3 introduceert een nieuw algoritme, MC-PNS, dat Monte-Carlo evaluaties (MCE) combineert met Proof-Number Search (PNS). Dit leidt tot de eerste onderzoeksvraag.

RQ 1 *Hoe kunnen we Monte-Carlo evaluaties gebruiken om Proof-Number Search te verbeteren voor het oplossen van spelposities?*

Een toepassing van het nieuwe algoritme op het leven-en-dood subprobleem in het spel Go wordt beschreven. Er wordt experimenteel aangetoond dat met de juiste parameterinstellingen MC-PNS de standaard PNS overtreft. Tevens wordt MC-PNS vergeleken met een op patronen gebaseerde heuristiek om de bladeren van de PNS zoekboom te initialiseren. Het resultaat is dat MC-PNS de patroongebaseerde initialisatie overtreft. We concluderen dat de reden voor de betere prestaties van MC-PNS is gelegen in de flexibiliteit van de toegepaste MCE.

Hoofdstuk 4 introduceert een nieuw Monte-Carlo Tree Search (MCTS) variant, genaamd MCTS-Solver. Dit heeft ons gebracht tot de tweede onderzoeksvraag.

RQ 2 *Hoe kan het Monte-Carlo Tree Search raamwerk bijdragen aan het oplossen van spelposities?*

Het hoofdstuk past het MCTS raamwerk aan voor het oplossen van spelposities. De variant MCTS-Solver verschilt van de standaard MCTS aanpak door het feit dat het posities kan oplossen door middel van het propageren van speltheoretische waarden. Als gevolg daarvan convergeert het sneller naar de beste zet in tactische posities. Experimenten in het spel Lines of Action (LOA) laten zien dat MCTS-solver met een specifieke selectie strategie LOA posities drie keer sneller oplost dan MCTS-Solver met behulp van de standaard selectie strategie UCT. Vergelijken we MCTS-Solver met een standaard $\alpha\beta$ zoekproces, dan lost MCTS-Solver LOA posities op met ongeveer dezelfde inspanning als $\alpha\beta$. Bovendien stellen we vast dat PN^2 (een Proof-Nunber Search algoritme bestaande uit twee niveaus) nog steeds vijf keer sneller is dan MCTS-Solver. Daarnaast tonen we aan dat boomparallellisatie voor MCTS-Solver schaalt met een factor 4 voor 8 processorkernen. Wortelparallellisatie wordt hier eenvoudig overtroffen. We concluderen dat gedurende het spel (online) MCTS-Solver vergelijkbaar is met een standaard $\alpha\beta$ zoekproces voor het oplossen van posities. Echter, voor het oplossen van posities offline is PNS nog steeds een betere keuze.

Hoofdstuk 5 introduceert een parallel Proof-Number Search algoritme voor gedeeld geheugen, genaamd RP–PNS. Daarmee beantwoorden we de derde onderzoeksvraag.

RQ 3 *Hoe kunnen we Proof-Number Search parallelliseren?*

De parallellisatie wordt bereikt door middels een kansverdeling zetten te selecteren in de buurt van de hoofdvariant. Verder hebben we RP–PNS aangepast voor PN^2, resulterend in RP–PN^2. De algoritmen worden getest op LOA posities. Voor 8 processorkernen is de gevonden schaalfactor voor RP–PN^2 (4.7) zelfs beter dan die van RP–PNS (3.5). De reden hiervoor is dat de grootte van de zoekboom op het tweede niveau (dat wil zeggen, PN_2) afhangt van het aantal gebruikte processorkernen. Op basis van deze resultaten concluderen we dat RP–PNS en RP–PN^2 geschikte technieken zijn om respectievelijk PNS en PN^2 te paralleliseren.

Hoofdstuk 6 richt zich op het oplossen van speldomeinen met meerdere spelers. Het hoofdstuk geeft een antwoord op de vierde onderzoeksvraag.

RQ 4 *Hoe kunnen we Proof-Number Search toepassen voor speldomeinen met meerdere spelers?*

Het uitgangspunt van dit hoofdstuk is de constatering dat veel tweespeler spelen zijn opgelost, maar vrijwel geen onderzoek is besteed aan het oplossen van spelen met meerdere spelers. De laatstgenoemde spelen hebben in het algemeen geen unieke speltheoretische waarde of strategie omdat hun zoekbomen meerdere evenwichtspunten kunnen hebben. Daarom kan de aanpak voor het oplossen van een tweespeler spel niet worden toegepast op een meerspeler spel. Wij stellen voor dit soort spelen op te lossen onder de paranoïde conditie. Dit is gelijk aan het vinden van de optimale score die een speler zelfstandig kan bereiken onafhankelijk van de strategieën van zijn opponenten. We introduceren Paranoid Proof-Number Search (PPNS) voor het oplossen van meerspeler spelen onder de paranoïde conditie.

Dit hoofdstuk beschrijft de meerspeler variant van het spel Reversi, Rolit, en bespreekt hoe PPNS toe te passen om diverse meerspeler varianten van 4×4 en 6×6 Rolit, en vierspeler 8×8 Rolit op te lossen. Het resultaat is dat voor 4×4 Rolit onder de paranoïde conditie sommige spelers meer dan de minimale score kunnen behalen, terwijl voor 6×6 Rolit en vierspeler 8×8 Rolit geen enkele speler meer kan bereiken dan de minimale score. Wij constateren dat voor 6×6 Rolit en vierspeler 8×8 Rolit PPNS beter presteert dan het beste geval voor Paranoid Search. Het hoofdstuk sluit af door te stellen dat PPNS in staat is om het non-uniforme karakter van de spelboom in meerspeler spelen uit te buiten.

Hoofdstuk 7 sluit het proefschrift af en geeft een vooruitblik op open vragen en richtingen van toekomstig onderzoek. Hoewel in het proefschrift het PPNS algoritme is voorgesteld om deterministische meerspeler spelen met *perfecte informatie* op te lossen, eindigen we door erop te wijzen dat het oplossen van deterministische meerspeler spelen met *imperfecte informatie* nog steeds een uitdaging is. We speculeren in het kort hoe dit probleem aangepakt kan worden.

Curriculum Vitae

Jahn-Takeshi Saito was born in Düsseldorf, Germany, on November 6, 1976. He attended school in Germany and Japan and graduated from Max-Plank-Gymnasium in Bielefeld, Germany in 1996. After 13 months of compulsory community service at Mara Epilepsy Center Bethel in Bielefeld, he went on to study Artificial Intelligence and Computational Linguistics at Osnabrück University with minors in mathematics and computer science. During his studies he joined the students' Go club. In 2001, he studied mathematics at Trinity College Dublin, Ireland, and did an internship at the Knowledge Engineering group at IBM's Tokyo Research Laboratory (which turned out to be located outside of Tokyo prefecture). In 2002, he joined the students' parliament and students' self-governing body (AStA), started working as a part-time programmer for KIWI GmbH in parallel to pursuing his studies, and decided that the game of Go would be a suitable subject for a Master thesis. In 2004, he graduated on Computer Go. In 2005, he started his Ph.D. project at the Games and AI Group, Department of Knowledge Engineering, Maastricht University. There, he acquired an inside perspective of academia. He worked on search algorithms for board games on the *Go for Go* project funded by the Netherlands Organisation for Scientific Research. Gradually, his research interest evolved towards solving game positions, Proof-Number Search and Monte-Carlo methods. Besides performing research, Jahn was involved in lecturing and helped organizing the 2006 Computers and Games conference in Turin, Italy. In early 2010, Jahn joined the team of the Bioinformatics Department, BiGCaT, Maastricht University where he has been working on systems biology databases.

SIKS Dissertation Series

1998

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

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

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

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

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

1999

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

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

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

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

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

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

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

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

2000

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

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

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

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*

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

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

2001

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

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

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

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

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

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

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

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

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

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

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

2002

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2003

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*

19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*

20 Madelon Evers (Nyenrode) *Learning from Design: facilitating multidisciplinary design teams*

2005

1 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*

2 Erik van der Werf (UM) *AI techniques for the game of Go*

3 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*

4 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*

5 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*

6 Pieter Spronck (UM) *Adaptive Game AI*

7 Flavius Frasincar (TU/e) *Hypermedia Presentation Generation for Semantic Web Information Systems*

8 Richard Vdovjak (TU/e) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

9 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*

10 Anders Bouwer (UvA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

12 Csaba Boer (EUR) *Distributed Simulation in Industry*

13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*

16 Joris Graaumans (UU) *Usability of XML Query Languages*

17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*

18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*

19 Michel van Dartel (UM) *Situated Representation*

20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*

21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006

1 Samuil Angelov (TU/e) *Foundations of B2B Electronic Contracting*

2 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*

3 Noor Christoph (UvA) *The role of metacognitive skills in learning to solve problems*

4 Marta Sabou (VU) *Building Web Service Ontologies*

5 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*

6 Ziv Baida (VU) *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*

7 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*

8 Eelco Herder (UT) *Forward, Back and Home Again - Analyzing User Behavior on the Web*

9 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*

10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*

11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*

12 Bert Bongers (VU) *Interactivation - Towards an e-cology of people, our technological environment, and the arts*

13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*

14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*

15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*

16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*

17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*

18 Valentin Zhizhkun (UvA) *Graph transformation for Natural Language Processing*

19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*

20 Marina Velikova (UvT) *Monotone models for prediction in data mining*

21 Bas van Gils (RUN) *Aptness on the Web*

22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*

23 Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*

24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*

25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*

26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*

27 Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*

28 Borkur Sigurbjornsson (UvA) *Focused Information Access using XML Element Retrieval*

2007

1 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*

2 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*

3 Peter Mika (VU) *Social Networks and the Semantic Web*

4 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach*

5 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance*

6 Gilad Mishne (UvA) *Applied Text Analytics for Blogs*

7 Natasa Jovanovic' (UT) *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings*

8 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*

9 David Mobach (VU) *Agent-Based Mediated Service Negotiation*

10 Huib Aldewereld (UU) *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols*

11 Natalia Stash (TU/e) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*

12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*

13 Rutger Rienks (UT) *Meetings in Smart Environments; Implications of Progressing Technology*

14 Niek Bergboer (UM) *Context-Based Image Analysis*

15 Joyca Lacroix (UM) *NIM: a Situated Computational Memory Model*

16 Davide Grossi (UU) *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems*

17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*

18 Bart Orriens (UvT) *On the development and management of adaptive business collaborations*

19 David Levy (UM) *Intimate relationships with artificial partners*

20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*

21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*

22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*

23 Peter Barna (TU/e) *Specification of Application Logic in Web Information Systems*

24 Georgina Ramírez Camps (CWI) *Structural Features in XML Retrieval*

25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008

1 Katalin Boer-Sorbán (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*

2 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*

3 Vera Hollink (UvA) *Optimizing hierarchical menus: a usage-based approach*

4 Ander de Keijzer (UT) *Management of Uncertain Data - towards unattended integration*

5 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*

6 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*

7 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*

8 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*

9 Christof van Nimwegen (UU) *The paradox of the guided user: assistance can be counter-effective*

10 Wauter Bosma (UT) *Discourse oriented Summarization*

11 Vera Kartseva (VU) *Designing Controls for Network Organizations: a Value-Based Approach*

12 Jozsef Farkas (RUN) *A Semiotically Oriented Cognitive Model of Knowledge Representation*

13 Caterina Carraciolo (UvA) *Topic Driven Access to Scientific Handbooks*

14 Arthur van Bunningen (UT) *Context-Aware Querying; Better Answers with Less Effort*

15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*

16 Henriette van Vugt (VU) *Embodied Agents from a User's Perspective*

17 Martin Op't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*

18 Guido de Croon (UM) *Adaptive Active Vision*

19 Henning Rode (UT) *From document to entity retrieval: improving precision and performance of focused text search*

20 Rex Arendsen (UvA) *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven*

21 Krisztian Balog (UvA) *People search in the enterprise*

22 Henk Koning (UU) *Communication of IT-architecture*

23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*

24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*

25 Geert Jonker (UU) *Efficient and Equitable exchange in air traffic management plan repair using spender-signed currency*

26 Marijn Huijbregts (UT) *Segmentation, diarization and speech transcription: surprise data unraveled*

27 Hubert Vogten (OU) *Design and implementation strategies for IMS learning design*

28 Ildiko Flesh (RUN) *On the use of independence relations in Bayesian networks*

29 Dennis Reidsma (UT) *Annotations and subjective machines- Of annotators, embodied agents, users, and other humans*

30 Wouter van Atteveldt (VU) *Semantic network analysis: techniques for extracting, representing and querying media content*

31 Loes Braun (UM) *Pro-active medical information retrieval*

32 Trung B. Hui (UT) *Toward affective dialogue management using partially observable markov decision processes*

33 Frank Terpstra (UvA) *Scientific workflow design; theoretical and practical issues*

34 Jeroen de Knijf (UU) *Studies in Frequent Tree Mining*

35 Benjamin Torben-Nielsen (UvT) *Dendritic morphology: function shapes structure*

2009

1 Rasa Jurgelenaite (RUN) *Symmetric Causal Independence Models*

2 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*

3 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*

4 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*

5 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality*

6 Muhammad Subianto (UU) *Understanding Classification*

7 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*

8 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*

9 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*

10 Jan Wielemaker (UvA) *Logic programming for knowledge-intensive interactive applications*

11 Alexander Boer (UvA) *Legal Theory, Sources of Law & the Semantic Web*

12 Peter Massuthe (TU/e, Humboldt-Universtät zu Berlin) *Operating Guidelines for Services*

13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*

14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)*

15 Rinke Hoekstra (UvA) *Ontology Representation - Design Patterns and Ontologies that Make Sense*

16 Fritz Reul (UvT) *New Architectures in Computer Chess*

17 Laurens van der Maaten (UvT) *Feature Extraction from Visual Data*

18 Fabian Groffen (CWI) *Armada, An Evolving Database System*

19 Valentin Robu (CWI) *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets*

20 Bob van der Vecht (UU) *Adjustable Autonomy: Controling Influences on Decision Making*

21 Stijn Vanderlooy (UM) *Ranking and Reliable Classification*

22 Pavel Serdyukov (UT) *Search For Expertise: Going beyond direct evidence*

23 Peter Hofgesang (VU) *Modelling Web Usage in a Changing Environment*

24 Annerieke Heuvelink (VU) *Cognitive Models for Training Simulations*

25 Alex van Ballegooij (CWI) *RAM: Array Database Management through Relational Mapping*

26 Fernando Koch (UU) *An Agent-Based Model for the Development of Intelligent Mobile Services*

27 Christian Glahn (OU) *Contextual Support of social Engagement and Reflection on the Web*

28 Sander Evers (UT) *Sensor Data Management with Probabilistic Models*

29 Stanislav Pokraev (UT) *Model-Driven Semantic Integration of Service-Oriented Applications*

30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*

31 Sofiya Katrenko (UvA) *A Closer Look at Learning Relations from Text*

32 Rik Farenhorst and Remco de Boer (VU) *Architectural Knowledge Management: Supporting Architects and Auditors*

33 Khiet Truong (UT) *How Does Real Affect Affect Affect Recognition In Speech?*

34 Inge van de Weerd (UU) *Advancing in Software Product Management: An Incremental Method Engineering Approach*

35 Wouter Koelewijn (UL) *Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling*

36 Marco Kalz (OU) *Placement Support for Learners in Learning Networks*

37 Hendrik Drachsler (OU) *Navigation Support for Learners in Informal Learning Networks*

38 Riina Vuorikari (OU) *Tags and Self-Organisation: A Metadata Ecology for Learning Resources in a Multilingual Context*

39 Christian Stahl (TU/e, Humboldt-Universtät zu Berlin) *Service Substitution – A Behavioral Approach Based on Petri Nets*

40 Stephan Raaijmakers (UvT) *Multinomial Language Learning: Investigations into the Geometry of Language*

41 Igor Berezhnyy (UvT) *Digital Analysis of Paintings*

42 Toine Bogers (UvT) *Recommender Systems for Social Bookmarking*

43 Virginia Nunes Leal Franqueira (UT) *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients*

44 Roberto Santana Tapia (UT) *Assessing Business-IT Alignment in Networked Organizations*

45 Jilles Vreeken (UU) *Making Pattern Mining Useful*

46 Loredana Afanasiev (UvA) *Querying XML: Benchmarks and Recursion*

2010

1 Matthijs van Leeuwen (UU) *Patterns that Matter*

2 Ingo Wassink (UT) *Work flows in Life Science*

3 Joost Geurts (CWI) *A Document Engineering Model and Processing Framework for Multimedia documents*

4 Olga Kulyk (UT) *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments*

5 Claudia Hauff (UT) *Predicting the Effectiveness of Queries and Retrieval Systems*

6 Sander Bakkes (UvT) *Rapid Adaptation of Video Game AI*

7 Wim Fikkert (UT) *Gesture interaction at a Distance*

8 Krzysztof Siewicz (UL) *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments*

9 Hugo Kielman (UL) *Politiële gegevensverwerking en Privacy, Naar een effectieve waarborging*

10 Rebecca Ong (UL) *Mobile Communication and Protection of Children*

11 Adriaan Ter Mors (TUD) *The world according to MARP: Multi-Agent Route Planning*

12 Susan van den Braak (UU) *Sensemaking software for crime analysis*

13 Gianluigi Folino (RUN) *High Performance Data Mining using Bio-inspired techniques*

14 Sander van Splunter (VU) *Automated Web Service Reconfiguration*

15 Lianne Bodenstaff (UT) *Managing Dependency Relations in Inter-Organizational Models*

16 Sicco Verwer (TUD) *Efficient Identification of Timed Automata, theory and practice*

17 Spyros Kotoulas (VU) *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications*

18 Charlotte Gerritsen (VU) *Caught in the Act: Investigating Crime by Agent-Based Simulation*

19 Henriette Cramer (UvA) *People's Responses to Autonomous and Adaptive Systems*

20 Ivo Swartjes (UT) *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative*

21 Harold van Heerde (UT) *Privacy-aware data management by means of data degradation*

22 Michiel Hildebrand (CWI) *End-user Support for Access to Heterogeneous Linked Data*

23 Bas Steunebrink (UU) *The Logical Structure of Emotions*

24 Dmytro Tykhonov (TUD) *Designing Generic and Efficient Negotiation Strategies*

25 Zulfiqar Ali Memon (VU) *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective*

26 Ying Zhang (CWI) *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines*

27 Marten Voulon (UL) *Automatisch contracteren*

28 Arne Koopman (UU) *Characteristic Relational Patterns*

29 Stratos Idreos (CWI) *Database Cracking: Towards Auto-tuning Database Kernels*

30 Marieke van Erp (UvT) *Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval*

31 Victor de Boer (UvA) *Ontology Enrichment from Heterogeneous Sources on the Web*

32 Marcel Hiel (UvT) *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems*

33 Robin Aly (UT) *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval*

34 Teduh Dirgahayu (UT) *Interaction Design in Service Compositions*

35 Dolf Trieschnigg (UT) *Proof of Concept: Concept-based Biomedical Information Retrieval*

36 Jose Janssen (OU) *Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification*

37 Niels Lohmann (TU/e) *Correctness of services and their composition*

38 Dirk Fahland (TU/e) *From Scenarios to components*

39 Ghazanfar Farooq Siddiqui (VU) *Integrative modeling of emotions in virtual agents*

40 Mark van Assem (VU) *Converting and Integrating Vocabularies for the Semantic Web*

41 Guillaume Chaslot (UM) *Monte-Carlo Tree Search*

42 Sybren de Kinderen (VU) *Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach*

43 Peter van Kranenburg (UU) *A Computational Approach to Content-Based Retrieval of Folk Song Melodies*

44 Pieter Bellekens (TU/e) *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain*

45 Vasilios Andrikopoulo (UvT) *A theory and model for the evolution of software services*

46 Vincent Pijpers (VU) *e3alignment: Exploring Inter-Organizational Business-ICT Alignment*

47 Chen Li (UT) *Mining Process Model Variants: Challenges, Techniques, Examples*

48 Milan Lovric (EUR) *Behavioral Finance and Agent-Based Artificial Markets*

49 Jahn-Takeshi Saito (UM) *Solving difficult game positions*