Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains

DISSERTATION

to obtain the degree of Doctor at
Maastricht University,
on the authority of the Rector Magnificus,
Prof. dr. L. L. G. Soete,
in accordance with the decision
of the Board of Deans,
to be defended in public
on Tuesday, November 24, 2015, at 16:00 hours

by

Hendrik Johannes Sebastian Baier

Supervisor:
Prof. dr. ir. R. L. M. Peeters

Co-supervisor:
Dr. M. H. M. Winands

Assessment Committee:
Prof. dr. G. Weiss (*chair*)
Prof. dr. S. M. Lucas (University of Essex)
Prof. dr. ir. J. A. La Poutré (Delft University of Technology)
Prof. dr. ir. J. C. Scholtes
Prof. dr. A. J. Vermeulen

Netherlands Organisation for Scientific Research

This research has been funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

SIKS Dissertation Series No. 2015-29
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 978-94-6159-489-1

Preface

Early on in my computer science studies at Darmstadt University of Technology, I became fascinated by the $\alpha\beta$ algorithm. I considered it impressive how quickly this elegant search algorithm, expressed in very few lines of code, allowed me to produce a program that beat me at a game seemingly requiring intelligence. As a hobby, I started reading the literature on Chess programming, and wrote a program to play Connect-4. Later, this program would evolve into the focus of my Bachelor's thesis on $\alpha\beta$ and its enhancements.

During my time in Darmstadt I also first came into contact with the game of Go. At first, I was interested in the rich culture and tradition surrounding it; then, I was surprised by the profound depth and complexity arising from its simple rules. Finally, I learned that writing a program to play Go on the level of human experts still stands as one of the grand challenges of AI. While I was studying cognitive science at Osnabrück University, I delved more deeply into the field, intrigued by recent successes of Monte-Carlo search techniques. Eventually I wrote my Master's thesis on the topic of adaptive rollouts in Monte-Carlo Tree Search—a technique that does not only exploit expert knowledge or information learned offline, but learns how to improve itself while the search process is running.

My PhD position at the Department of Knowledge Engineering of Maastricht University has allowed me to dedicate four years of full-time work to my longtime fascination with search and games AI, touching on Connect-4 and Go, Monte-Carlo Tree Search and $\alpha\beta$, and many more topics along the way. I have grown in many ways throughout these exciting years, both professionally and personally. However, this thesis is the result of an effort I could not have undertaken alone. Therefore, I would like to use this opportunity to thank the people who have made this journey possible, who have accompanied and supported it, and who have made it so much fun.

First of all, I would like to thank my daily supervisor Mark Winands. His inexhaustible knowledge on search in games and his thorough and systematic approach to scientific writing were of great help in my work. In addition, he always knew how to keep me on my PhD track, without interfering with the research directions I had chosen. I would also like to thank Ralf Peeters for agreeing to be my promotor. Furthermore, my thanks go to Johannes Fürnkranz and Kai-Uwe Kühnberger, the supervisors of my Bachelor's and Master's theses, who gave me the chance to follow my interests long before I knew how far they would lead me one day.

I would like to thank the people who have been part of the games group over

the past years: Maarten Schadd, Marc Lanctot, Pim Nijssen, Mandy Tak, and Jahn-Takeshi Saito. It was a pleasure to work with you, learn with you, discuss with you, and have the occasional after-work beer with you. The whole Department of Knowledge Engineering has felt like home to me, and I will always remember our birthday cakes, department trips, sandwiches on the city wall, football bets, and our brilliant lunch discussions on scientific and less-than-scientific level with a smile. Thanks for that to my friends and colleagues Steven de Jong, Nela Lekic, Michael Clerx, Michael Kaisers, Frederik Schadd, Daniel Hennes, Pietro Bonizzi, Nasser Davarzani, Bijan Ranjbar-Sahraei, Siqi Chen, Nyree Lemmens, Daniel Claes, Joscha Fossel, You Li, Libo He, Philippe Uyttendaele, Haitham Bou Ammar, and Ignacia Arcaya. Moreover, many thanks to Peter Geurtz for the technical support, and Marie-Lou Mestrini for the administrative support—both were invaluable.

But life in Maastricht does not consist of work alone! I would therefore like to express my gratitude to all my friends who have made these past years such an amazing and rewarding experience—from the first days of confusion to the last days of nostalgia. Thank you to everyone I have met through the parties, movie nights, dance courses, sports trips and other events of PhD Academy; thank you to everyone I have met through my new hobby of Salsa dancing, which has brought so much joy to my life; thank you to the "crazy bunch", to the jazz night crew, to the expats and socialites; thank you to my UNU-MERIT friends, my FHML friends, and even my SBE friends; thank you to the great people I have met through courses and on conferences; and thank you to many, many others that have crossed my path either in the winding streets of Maastricht or during my travels. Special thanks to my paranymphs Daan Bloembergen and Mare Oehlen for the friendship we share. Meeting all of you has delayed my PhD for at least one fantastic year, and I am sure the Maastricht family will stay in contact with each other and with this cute town that brought us together.

Last but not least, I would like to give my deepest thanks to my parents, Peter and Jutta, whose unconditional love has supported me throughout this adventure as well as countless others. Danke euch für alles.

Hendrik Baier, 2015

Acknowledgements

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems. This research has been funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The topic of this thesis is *Monte-Carlo Tree Search* (MCTS), a technique for making decisions in a given problem or domain by constructing an internal representation of possible actions, their effects, and the possible next actions that result. This representation takes the form of a tree, and it is grown and improved over time so as to find the best action to take. During the construction of the tree, MCTS focuses on the actions that currently seem most promising, where promising actions are identified through the so-called Monte-Carlo simulations. The basic idea behind these simulations is estimating the quality of an action by repeatedly sampling its possible consequences, the possible futures resulting from it.

In order to enhance MCTS, this thesis uses *games* as test domains. Games have two advantages for the study of searching, planning and decision-making. On the one hand, they provide ideal abstractions from real-world problems thanks to their clear rules—there is for example no uncertainty as to which moves are legal in any given situation, and no external factors can add noise to the results of playing those moves. Because games are closed systems, they are relatively easy to handle as testbeds for search algorithms. On the other hand, games still pose considerably complex and challenging problems both for humans and computers, which makes their study interesting and fruitful for the field of Artificial Intelligence (AI). This thesis is concerned with games of two different types: one-player games and two-player games.

This chapter is structured as follows. Section 1.1 provides an introduction to games and their connection to AI. A number of game properties is used to classify the games used as test domains in this thesis. Section 1.2 briefly outlines the search techniques relevant for this work, with emphasis on MCTS. Section 1.3 states the problem statement guiding the research, and four research questions are defined to specify the approach taken. Section 1.4 finally presents an overview of this thesis.

1.1 Games and AI

Games have been a cornerstone of AI research ever since the early pioneering days. Only four years after the construction of ENIAC, the world's first general-purpose electronic computer, Claude Shannon published a paper on computer Chess (Shannon, 1950). Alan Turing started writing a Chess-playing program around the same time and published its description three years later (Turing, 1953). An early Checkers program was written by Christopher Strachey at the University of Manchester in 1951-52. The first Checkers program able to defeat human amateur players, also one of the first AI programs capable of *learning*, was completed in 1955 and demonstrated on television in 1956 (Samuel, 1959). Classic board games like Chess and Checkers do not only provide well-defined abstractions from real-world problems. They also have an intuitive appeal to the general population, and their mastery is considered the epitome of intelligence and rational thought by many. Before the term AI was even coined in 1956, researchers had been fascinated by the idea of challenging human intellectual supremacy by teaching a computer how to play.

With the optimism of the "golden years" of AI, Herbert A. Simon predicted in 1957 that "within ten years a digital computer will be the world's Chess champion" (Simon and Newell, 1957). It turned out to take three more decades to reach that level. But throughout these years and beyond, Chess and other games have proved excellent test-beds for new ideas, architectures and algorithms, illustrating many important problems and leading to successful generalizations for work in other fields. Chess became, with the words of the Russian mathematician Alexander Kronrod in 1965, the "drosophila of artificial intelligence" (cf. McCarthy 1990). Raj Reddy called the game an "AI problem par excellence" in his presidential address to AAAI in 1988, listing computer Chess together with natural language, speech, vision, robotics and expert systems (Reddy, 1988). In 1997 finally, the world Chess champion Garry Kasparov was defeated by IBM's Chess supercomputer Deep Blue with 3.5 to 2.5 (Hsu, 2002).

However, many challenges remain: In the ancient Asian board game of Go for example, human masters are still stronger than any program written to date. In Poker, programs have to deal with hidden information as well as opponent modelling for multiple players (Billings et al., 2002; Bowling et al., 2015). Work has begun on new board games such as *Hex* (Anshelevich, 2002), *Amazons* (Lieberum, 2005), or *Havannah* (Teytaud and Teytaud, 2009)—some of them such as *Arimaa* explicitly designed to be difficult for traditional game-playing algorithms (Syed and Syed, 2003), encouraging new approaches e.g. for large action spaces (Fotland, 2004). Researchers have started to tackle General Game Playing (GGP), the quest for a program that is able to play not only one specific game, but all games whose rules can be defined in a Game Description Language (Genesereth et al., 2005; Thielscher, 2010; Schaul,

2014). Supported by a growing industry, video games call for more intelligent and convincing artificial opponents and allies (Laird and van Lent, 2001; Ontañón et al., 2013; Schrum et al., 2011). And RoboCup fosters research into AI and robotics for team games in real-time settings (Kitano et al., 1997).

Games can be categorized into several different classes according to their properties. These properties often help to determine the optimal choice of AI technique for the game at hand. In the following, six different game properties are listed that are relevant to characterize the games investigated in this thesis.

Number of players. Games can require one, two, or more players. *One-player games* resemble optimization problems in which the player tries to achieve the best possible outcome without having to take other players into account. A well-known example of a one-player game is Solitaire. In *two-player games*, two players interact, either in a cooperative or in an adversarial way. A prominent example is Chess. Games with more than two players are called *multi-player games*. Multi-player games can pose additional challenges due to the possibility of coalition formation among the players. Chinese Checkers, Monopoly and Risk are well-known games of this type. The games used as test domains in this thesis are one-player games (in Part I) and two-player games (in Part II).

Competition and Cooperation. If for a given two-player game, the outcomes for both players always add up to zero—i.e. if gains and losses of both players always balance each other out—the game is called a *zero-sum game*. This is for example the case in any game where a win of the first player (represented by the value +1) is necessarily a loss for the second player (represented by the value -1) and vice versa. These games are strictly competitive. In contrast, *non-zero-sum games* can sometimes be non-competitive, if the sum of the players' outcomes can be increased through cooperation. All two-player games in this thesis are zero-sum.

Determinism. If every legal move in every legal position in a given game leads to a uniquely defined next position, the game is called *deterministic*. The course of such games is fully specified by the moves of the players. Checkers for instance is a deterministic game. If a game features chance events such as the roll of a die or the drawing of a card however, it is *non-deterministic*. Backgammon is a game of this category. This thesis considers only deterministic games.

Observability. If all players have access to all information defining a game's current state at all times throughout the game, the game is called a *perfect information game*. An example is Connect-4, as well as most classic board games. If any information about the current state is hidden from any player at any point during a game, the game has *imperfect information*. Poker for example belongs in this category, together

with most card games. All games investigated in this thesis are perfect-information games.

Cardinality of state and action spaces. Some games, in particular video games, feature a *continuous* (uncountable) set of possible game states and/or player actions. Ignoring a necessary fine discretization of the controller input and graphical output, for example, the player of a typical first-person shooter is free to move to and shoot at any reachable point in a given level. In contrast, card and board games usually have *discrete* (countable), and in most cases finite, state and action spaces. The number of legal moves in any given Chess position for example is finite, and the number of possible Chess positions is large but finite as well. This is the case for all games studied in this thesis.

Game flow. If the players can perform actions at any point in time during a game, i.e. if the game flow is continuous, we speak of a *real-time game*. Again apart from a fine discretization due to technical constraints, many video games fall into this category. In other games time is structured into *turns* in which players can move. Such turn-based games fall into one of two categories. In the category of *simultaneous move games*, more than one player can move at the same time. Some modern board games such as 7 Wonders or Diplomacy fall into this group. In the category of *sequential move games* (also called *turn-taking* games) such as Chess and Go, players move one at a time. This thesis is concerned with sequential move games.

In Part I of this thesis, the one-player games *SameGame*, *Clickomania*, and *Bubble Breaker* are used as test domains. In Part II, the two-player games *Go*, *Connect-4*, *Breakthrough*, *Othello*, and *Catch the Lion* are used. All these games are deterministic perfect-information turn-taking games with discrete action spaces and state spaces. See Chapter 3 for their detailed descriptions.

1.2 Search Techniques

For decades, much work of AI researchers in the field of games has been done on deterministic perfect-information discrete turn-taking two-player games, with Chess as the prime example. A high level of play in the domains of Chess and many similar games was achieved by programs based on *minimax search* (von Neumann and Morgenstern, 1944) and its enhancement $\alpha\beta$ *pruning* (Knuth and Moore, 1975). Minimax search decides on the next move to play by computing all possible future states of the game that can be reached in a prespecified number of moves. It compares the desirability of those future states with the help of a *heuristic evaluation function*. Finally, it chooses the move that leads to the most desirable future state, under the

assumption that the opponent makes no mistakes. $\alpha\beta$ pruning speeds up this process by excluding future states from the search if they provably cannot influence the final result. A number of extensions have been developed based on the minimax framework over the years, such as *expectimax* for non-deterministic two-player games (Michie, 1966), or max^n for multi-player games (Luckhardt and Irani, 1986).

However, these traditional minimax-based search techniques have not been successful in all games. Computing all relevant future states of a game, even looking only a few moves ahead, poses a computational problem in games with large numbers of legal moves per position. More importantly, minimax requires a game-specific heuristic evaluation function to assign meaningful estimates of desirability to any given game state. In some games, Go being the most prominent example, the construction of such an evaluation function has turned out to be difficult.

For finding solutions to one-player games (puzzles), one of the classic approaches is the A* search algorithm (Hart et al., 1968). A* starts with a queue containing the current state of the game. It then repeatedly removes the first element of the queue and adds its possible successor states, keeping the entire list sorted with the help of a heuristic evaluation function. The search stops when a goal state appears at the start of the list. Various enhancements of A* have been proposed as well, e.g. a combination with iterative-deepening depth-first search in order to achieve a lower memory usage (IDA*, Korf 1985). A*-based techniques are mainly applied to pathfinding (Sturtevant and Buro, 2005), but have also been successful in problems such as the 20-puzzle (Sadikov and Bratko, 2007) or Sokoban (Junghanns and Schaeffer, 2001).

But similarly to minimax search in the case of two-player games, A*-based methods require domain knowledge in the form of a heuristic evaluation function. Moreover, only when this function meets the requirement of *admissibility*—in minimal-cost problems, the requirement of never overestimating the remaining cost of an optimal solution leading through the evaluated state—is A* guaranteed to find the optimal solution. For some games, for example SameGame (Schadd et al., 2008b), it is unknown how to design an effective admissible heuristic.

These problems with heuristic evaluation functions in both one- and two-player games have led researchers to consider the domain-independent method of Monte-Carlo evaluation—a method that evaluates a given state not with the help of heuristic knowledge provided by the programmer, but with the average outcome of random games starting from that state (Abramson, 1990). The combination of Monte-Carlo evaluation and best-first tree search finally resulted in the development of *Monte-Carlo Tree Search* (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007b). The algorithm consists of four phases: the selection phase, the expansion phase, the rollout phase, and the backpropagation phase. The selection phase picks an interesting region of the search tree to examine, trading off spending search effort on moves which we are

most uncertain about (exploration) versus spending effort on moves which currently look best (exploitation). The expansion phase grows the search tree in the direction of the chosen moves. The rollout phase randomly simulates the rest of the game, and the backpropagation phase uses the result of that simulation to update an estimate of the quality of the chosen moves. One of the advantages of MCTS is that it is *aheuristic*—in its basic form, it does not require any understanding of the game beyond its rules. Further advantages are its *selectivity*, making it well-suited to games with large branching factors, and its *anytime* property, allowing it to return a current best estimate of optimal play whenever it is interrupted.

The MCTS approach has quickly become the dominating paradigm in the challenging field of computer Go (Lee et al., 2009). Beyond Go, MCTS and its enhancements have considerable success in domains as diverse as deterministic, perfect-information two-player games (Lorentz, 2008; Winands et al., 2010; Arneson et al., 2010), non-deterministic two-player games (Lorentz, 2012), imperfect-information two-player games (Ciancarini and Favini, 2010; Whitehouse et al., 2011), non-deterministic imperfect-information two-player games (Cowling et al., 2012), deterministic one-player games (Schadd et al., 2008b; Cazenave, 2009), non-deterministic one-player games (Bjarnason et al., 2009), multi-player games (Sturtevant, 2008; Nijssen and Winands, 2013), simultaneous move games (Perick et al., 2012; Tak et al., 2014), real-time video games (Balla and Fern, 2009; Pepels et al., 2014), and General (Video) Game Playing (Björnsson and Finnsson, 2009; Perez et al., 2014b). Beyond the field of games, MCTS has also been successful in various planning and optimization domains such as partially observable MDPs (Silver and Veness, 2010; Müller et al., 2012), mixed integer programming (Sabharwal et al., 2012), expression simplification (Ruijl et al., 2014), boolean satisfiability (Previti et al., 2011), variants of the travelling salesman problem (Rimmel et al., 2011; Perez et al., 2014a), hospital planning (van Eyck et al., 2013), and interplanetary trajectory planning (Hennes and Izzo, 2015). See Browne et al. (2012) for a recent survey of the field.

1.3 Problem Statement and Research Questions

The previous sections reflected on games in the field of AI, as well as the technique of MCTS and its application in different areas. MCTS is now an active and promising research topic with room for improvements in various directions. For example, there are still a number of two-player games in which the traditional approach to adversarial search, minimax with $\alpha\beta$ pruning, remains superior. This has been related to certain properties of these games (Ramanujan et al., 2010a) and motivates research into improving the ability of MCTS to deal with these properties. Furthermore, while MCTS can be implemented in a way that guarantees convergence to optimal play in

one- and two-player domains (Kocsis and Szepesvári, 2006; Kocsis et al., 2006), its performance in settings with limited time can still be improved. Possible approaches for this are enhancements of the rollout phase or the selection phase of MCTS. This thesis focuses on enhancing MCTS in one-player and two-player domains. The research is guided by the following problem statement.

Problem Statement: *How can the performance of Monte-Carlo Tree Search in a given one- or two-player domain be improved?*

Four research questions have been formulated in order to approach this problem statement. They are divided into two groups. Two questions are concerned with one-player domains, while two questions are dealing with adversarial two-player domains. The four research questions address (1) the rollout phase of MCTS in one-player domains, (2) the selectivity of MCTS in one-player domains, (3) time management for MCTS in two-player tournament play, and (4) combining the strengths of minimax and MCTS in two-player domains.

Research Question 1: *How can the rollout quality of MCTS in one-player domains be improved?*

In the rollout phase of MCTS, moves are usually chosen randomly or selected by an inexpensive, domain-dependent heuristic. In some domains, this can lead to quickly diminishing returns as search times get longer, caused by the relatively low quality of the rollout policy (see e.g. Robilliard et al. 2014). In order to approach this problem, *Nested Monte-Carlo Tree Search* (NMCTS) is proposed in this thesis, replacing simple rollouts with nested MCTS searches. Without requiring any domain knowledge, the recursive use of MCTS can improve the quality of rollouts, making MCTS stronger especially when longer search times are available. NMCTS is a generalization of regular MCTS, which is equivalent to level-1 NMCTS. Additionally, NMCTS can be seen as a generalization of Nested Monte-Carlo Search (NMCS) (Cazenave, 2009), allowing for an exploration-exploitation tradeoff by nesting MCTS instead of naive Monte-Carlo search. The approach is tested in the puzzles SameGame, Clickomania, and Bubble Breaker.

Research Question 2: *How can the selectivity of MCTS in one-player domains be improved?*

In *Upper Confidence bounds applied to Trees* or UCT (Kocsis and Szepesvári, 2006), the most widely used variant of MCTS (Browne et al., 2012; Domshlak and Feldman, 2013), the selectivity of the search can be controlled with a single parameter: the exploration factor. In domains with long solution lengths or when searching with a

short time limit however, MCTS might not be able to grow a search tree deep enough even when exploration is completely turned off. The result is a search process that spends too much time on optimizing the first steps of the solution, but not enough time on optimizing the last steps. This problem is approached in this thesis by proposing *Beam Monte-Carlo Tree Search* (BMCTS), a combination of MCTS with the idea of beam search (Lowerre, 1976). BMCTS expands a tree whose size is linear in the search depth, making MCTS more effective especially in domains with long solution lengths or short time limits. Test domains are again SameGame, Clickomania, and Bubble Breaker.

Research Question 3: *How can the time management of MCTS in two-player domains be improved?*

In competitive gameplay, time is typically limited—for example by a fixed time budget per player for the entire game (*sudden-death* time control). Exceeding this time budget means an instant loss for the respective player. Since longer thinking times, especially for an anytime algorithm like MCTS, usually result in better moves, the question arises how to distribute the time budget wisely among all moves in the game. A number of *time management strategies* are investigated in this thesis, both taken from the literature (Baudiš, 2011; Huang et al., 2010b) as well as newly proposed and improved ones. These strategies are tested and analyzed in the two-player games Go, Connect-4, Breakthrough, Othello, and Catch the Lion.

Research Question 4: *How can the tactical strength of MCTS in two-player domains be improved?*

One of the characteristics of MCTS is Monte-Carlo simulation, taking distant consequences of moves into account and therefore providing a strategic advantage in many domains over traditional depth-limited minimax search. However, minimax with $\alpha\beta$ pruning considers every relevant move within the search horizon and can therefore have a tactical advantage over the highly selective MCTS approach, which might miss an important move when precise short-term play is required (Ramanujan et al., 2010a). This is especially a problem in games with a high number of terminal states throughout the search space, where weak short-term play can lead to a sudden loss. Therefore, *MCTS-minimax hybrids* are proposed in this thesis, integrating shallow minimax searches into the MCTS framework and thus taking a first step towards combining the strengths of MCTS and minimax.

These hybrids can be divided into approaches that require domain knowledge, and approaches that are knowledge-free. For the knowledge-free case, three different hybrids are studied using minimax in the selection/expansion phase, the rollout phase,

and the backpropagation phase of MCTS. Test domains are Connect-4, Breakthrough, Othello, and Catch the Lion. For the case where domain knowledge is available, three more hybrids are investigated employing minimax to choose rollout moves, to terminate rollouts early, or to bias the selection of moves in the MCTS tree. Test domains are Breakthrough, Othello, and Catch the Lion. Go is not studied in these chapters because it has a high number of terminal states only at the very end of the game, and because an effective heuristic evaluation function is unknown.

1.4 Structure of the Thesis

This thesis is divided into nine chapters. Chapters 1 to 3 introduce the necessary background. Afterwards, Chapters 4 to 8 answer the research questions posed in the previous section. These are grouped into two parts—Part I on one-player games consists of Chapters 4 and 5, and Part II on two-player games comprises Chapters 6 to 8. Finally, the conclusions of the thesis are presented in Chapter 9.

Chapter 1 provides a brief introduction to the field. It then presents the problem statement and the four research questions that have been posed to approach it.

Chapter 2 describes the basic terms and concepts of search in games. It also introduces the two classes of search methods used in Chapters 4 to 8: minimax-based search techniques for two-player games, and MCTS techniques for both one- and two-player games. Enhancements for both minimax and MCTS are explained as far as relevant for our research. Additionally, the topic of this thesis is related to the field of reinforcement learning. This learning perspective is not necessary background for the chapters that follow, but can provide a deeper understanding of MCTS for the interested reader—in particular with regard to the *multi-armed bandit* algorithms which are applied recursively within MCTS.

Chapter 3 introduces the test domains used in the following chapters. These include the one-player games SameGame, Clickomania, and Bubble Breaker, which are used in Part I; and the two-player games Go, Connect-4, Breakthrough, Othello, and Catch the Lion, which are used in Part II. For each game, its origin is described, its rules are outlined, and its complexity is analyzed.

Part I on one-player games begins with Chapter 4. This chapter answers the first research question by introducing Nested Monte-Carlo Tree Search. NMCTS is presented as a generalization of both MCTS and NMCS. After parameter tuning, it is tested against regular MCTS with randomized restarts in order to show the effect of introducing nested searches. Furthermore, NMCTS is directly compared to its special case of NMCS in order to show the effect of introducing selective tree search. The experiments are performed in SameGame with both random and informed rollout policies, and in Clickomania and Bubble Breaker with random rollouts.

Chapter 5 answers the second research question by proposing Beam Monte-Carlo Tree Search. BMCTS can be understood as a generalization of move-by-move search, allowing to keep any chosen number of alternative moves instead of committing to one of them when moving on to the next tree depth. After examining the parameter landscape of the algorithm, BMCTS is tested against regular MCTS both with a single search run per test position and with multiple search runs per test position. Results on combining NMCTS and BMCTS are presented as well. These experiments are again performed in SameGame with both random and informed rollout policies, and in Clickomania and Bubble Breaker with random rollouts.

Part II on two-player games begins with Chapter 6. This chapter answers the third research question. It outlines a number of time-management strategies for MCTS, organized along two dimensions: whether they make timing decisions before the start of a search or during the search, and whether they are domain-independent or specific to the game of Go. The strategies are then tested in Go, Connect-4, Breakthrough, Othello, and Catch the Lion. Afterwards, their performance is analyzed and compared across domains, in particular with regard to shifting available time to the opening, midgame, or endgame.

Chapter 7 answers the fourth research question for the case where domain knowledge is not available. It investigates three MCTS-minimax hybrids, using shallow minimax searches without a heuristic evaluation function in different phases of the MCTS framework. The baseline MCTS-Solver is described, and the hybrids are tested against the baseline and against each other in the domains of Connect-4, Breakthrough, Othello, and Catch the Lion. The performance of the hybrids is then analyzed across domains, relating it to the density and difficulty of search traps. In additional experiments, we study the influence of different time settings and different branching factors on hybrid performance, and test their effectiveness for solving endgame positions as well.

Chapter 8 answers the fourth research question for the case where domain knowledge is available. It studies three more MCTS-minimax hybrids, embedding shallow minimax searches with an evaluation function into MCTS. The evaluation functions are explained, and the hybrids are tested against MCTS-Solver and against each other in Breakthrough, Othello, and Catch the Lion. Additionally, their performance is compared across domains. After identifying the branching factor of a domain as a limiting factor for the hybrids' performance, move ordering and k-best pruning are introduced. The move ordering functions are explained, and the hybrids are again tuned and tested against the baseline and against each other. Further experiments compare the hybrids across domains, study the effects of different time settings and branching factors, combine the hybrids with each other, and test them against basic $\alpha\beta$ search.

Chapter 9 finally summarizes the answers to the four research questions, and addresses the problem statement. It also gives possible directions for future work.

2

Search Methods

Parts of this chapter are based on:

Baier, H. and Winands, M. H. M. (2012). Nested Monte-Carlo Tree Search for Online Planning in Large MDPs. In L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, editors, *20th European Conference on Artificial Intelligence, ECAI 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 109–114.

Baier, H. and Drake, P. (2010). The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, volume 2, number 4, pages 303–309.

This chapter describes the search methods used in this thesis for playing one-player and two-player games. After explaining their basic frameworks, it also outlines some of their commonly used enhancements as far as necessary to understand the chapters that follow. Two families of search methods are considered: Monte-Carlo Tree Search (MCTS) techniques for both one- and two-player games as the focus of this thesis, and minimax-based search techniques for two-player games as an additional prerequisite for the MCTS-minimax hybrids of Chapters 7 and 8.

This chapter is organized as follows. Section 2.1 introduces the basic terms and concepts of search in games that are used throughout this thesis. Section 2.2 describes minimax-based search techniques. Section 2.3 gives an introduction to MCTS, and Section 2.4 discusses a number of MCTS enhancements. In Section 2.5 finally, an alternative introduction to MCTS is given in order to connect this thesis to wider areas of work in AI. This introduction presents MCTS as a reinforcement learning method.

2.1 Search in Games

A *game* of the type considered in this thesis can be modeled with the help of the following components, resulting from its rules:

- A set of possible states of the game S, also called positions. A state of a board game like Chess, for example, consists of the arrangement of pieces on the board, the player to move, as well as all further relevant information for the future of the game, such as the castling and en passant options, the positions which have already occurred for following the threefold repetition rule, etc.
- An initial state $I \in S$, for example the starting arrangement of pieces with the information who is the first player to move.
- A set of players P. Part I of this thesis is concerned with one-player games, for which we set $P = \{\text{MAX}\}$. Part II is dealing with two-player games where we set $P = \{\text{MAX}, \text{MIN}\}$, with MAX representing the player moving first.
- A function *tomove*: $S \to P$ which indicates for any given state the current player to move.
- A set of actions A, also called moves. For any state $s \in S$, the subset of legal actions in that state is denoted by $A_s \subseteq A$. Selecting and carrying out an action $a \in A_s$ brings the game to a new state s' as determined by the rules of the game: $s \xrightarrow{a} s'$. The set of all states that can be reached from s with a single action is denoted by $C_s \subseteq S$. Any state s with $A_s = \{\}$, i.e. any state where the game has ended, is called a *terminal state*. We call the set of all terminal states Z.
- A function R: $S \times A \to \mathbb{R}^{|P|}$ which returns for any given state and action a vector containing the rewards received by all players. In the case of the one-player games discussed in Part I of this thesis, this function just returns a scalar reward value. In the case of the two-player games covered in Part II, their zero-sum property allows simplifying the reward vector to a scalar value as well, representing the reward received by MAX. MIN's reward is the negation of that value.

Knowing the initial state I of a game and the possible actions A_s in any given state s allows us to represent all possible ways this game can be played in the form of its *game tree*. Figure 2.1 shows a part of the game tree for the well-known game of Tic-Tac-Toe. Each *node* in this tree represents a possible state of the game. Each *edge* represents a possible action. The tree starts at the *root*, marked I in the figure, as the node representing the initial state. This root is *expanded* by adding an edge for each legal action $a \in A_I$, and a node for each next state s' with $I \xrightarrow{a} s'$. The game tree

is created by recursively expanding all added nodes in the same way. Note that this means one state can potentially be represented by more than one node, if that state can be reached from the initial state through different action sequences. If a node X represents a state x and a node Y represents a state y such that $y \in C_x$, Y is called a *child* (immediate successor) of X, and X is a called the *parent* (immediate predecessor) of Y. In Figure 2.1 for example, C is A's child, and A is C's parent. Nodes A and B are called *siblings* because they share the same parent. Nodes can have zero, one, or more children, but each node has exactly one parent—except for the root which does not have a parent. Every node X on the path from the root node to the parent of a given node Y is called an *ancestor* of Y, whereas Y is called a *descendant* of X. In Figure 2.1 for example, I and A are the ancestors of C, and C is a descendant of I and of A. A node X together with all its descendants is called a *subtree* rooted in X. For example, C is in A's subtree, while I and B are not. Nodes that have at least one child are called *internal* or *interior* nodes, whereas nodes without any children are called *leaf nodes*. Nodes representing terminal states are *terminal nodes*. In a game tree, all leaf nodes are terminal nodes. In these leaves the game is over and its *outcome*, the sum of all rewards received during the game, is known for all players.

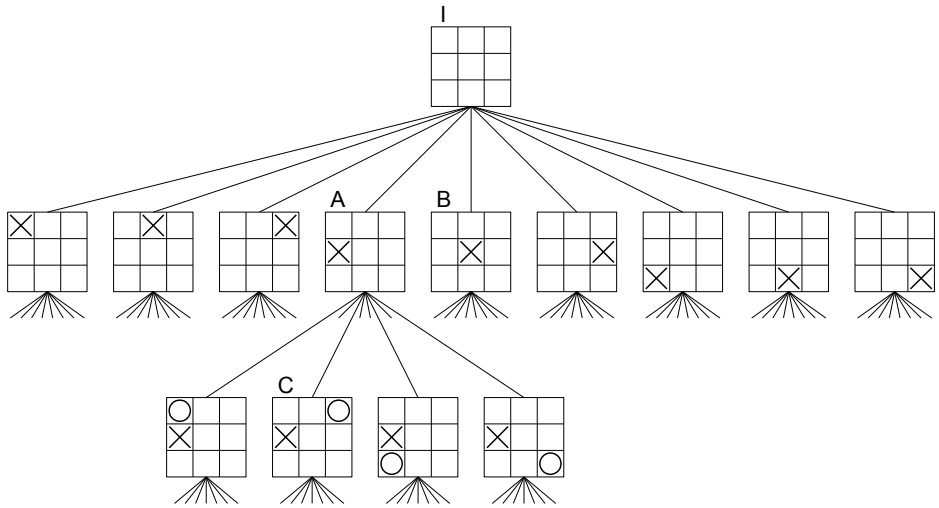


Figure 2.1: A part of a game tree. The figure shows all legal first actions and a few legal second actions from the initial state of Tic-Tac-Toe.

Representing a game as a game tree allows us to view the task of playing the game as a *search problem*. In a one-player game as discussed in Part I, the player is searching for a sequence of actions, starting from the initial state and leading to a terminal state, with optimal outcome. In a two-player game as used in Part II, the player is

searching for a sequence maximizing her[1] outcome as well. However, the opponent can interfere with such a sequence as soon as it is her turn. Each player is therefore searching for a strategy leading to the best possible outcome for herself without having certainty regarding the opponent's actions in the future. This is achieved by exploring and analyzing the game tree, for example with the search methods described in the following sections.

It is usually impossible to investigate the complete game tree for a game of non-trivial complexity. The game tree of Chess for example is estimated to have approximately 10^{123} nodes (Shannon, 1950)—for comparison, the number of atoms in the observable universe is estimated to be roughly 10^{80}. Therefore, computer programs typically analyze only a part of the game tree in practice, called the *search tree*. The nodes of a search tree are gradually generated by a *search process*, starting at the root. Figure 2.2 presents an example search tree from a game of Tic-Tac-Toe. The search tree is rooted in the current state the player is in (node A in Figure 2.2) instead of the initial state of the entire game (node I in Figure 2.1). Furthermore, not all descendants of this root are usually considered. Instead, the search tree is constructed and examined until a predefined limit has been reached—for example until a given search time has passed, or until the search tree has been analyzed to a given *depth*. The depth of a tree is measured in *plies* (Samuel, 1959). In Figure 2.2 for example, a two-ply search tree is shown, meaning that the tree looks two turns ahead from the root state. Not all nodes in a search tree have to be expanded—node B for example is expanded, whereas node C is not. Search trees, in contrast to complete game trees, can have *non-terminal leaf nodes*. These are leaf nodes that do not have children (yet), but do not represent terminal states. Nodes C, D, F, and H are non-terminal leaves, while nodes E and G are terminal leaves.

The following Sections 2.2 and 2.3 outline two frameworks for the construction and analysis of such search trees, and explain how they can lead to effective action decisions for the game-playing program.

2.2 Minimax

This section introduces the first basic method for making action decisions in games—*minimax* search. Minimax is used in Chapters 7 and 8 of this thesis. It is a search technique for finite two-player zero-sum games.

[1] Where gendered forms cannot elegantly be avoided, the generic feminine is used throughout this thesis in order to avoid cumbersome forms such as "he or she", "(s)he", or "her/his".

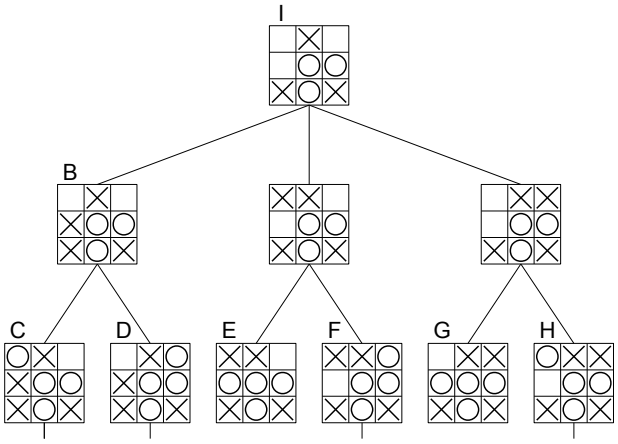


Figure 2.2: A search tree. The figure shows the first two ply of a search starting from the Tic-Tac-Toe position shown at the root.

2.2.1 The Minimax Value

Recall that we call the two players of a two-player game *MAX* and *MIN*. In the two-player games used in this thesis, the reward received when reaching a terminal state only depends on that state and is independent of the sequence of actions which was taken to reach it. The players only collect rewards when transitioning to the terminal state. All other actions in the game give a reward of 0 to both players. Therefore we can define a utility function *utility*: $Z \to \mathbb{R}$, returning for any given terminal state the outcome of the game when reaching that state, from the point of view of MAX. Due to the zero-sum property of the games studied here, this also specifies the outcomes from the point of view of MIN. The game of Chess for example has the possible outcomes win, loss, and draw, represented by a utility of 1 for a win for MAX (a loss for MIN), -1 for a loss for MAX (a win for MIN), and 0 for a draw. MAX tries to reach a terminal state with maximal utility, and MIN tries to reach one with minimal utility.

Without loss of generality, assume that MAX is the player to move. Given the current state and the rules of the game, and thus given the ability to construct and analyze a search tree, MAX is trying to find an *optimal strategy*. A *strategy* is defined by the next action to take, and an action to take in every possible future state of the game depending on the opponent's actions. Informally speaking, an *optimal* strategy guarantees the best possible outcome for MAX assuming MIN will not make any mistakes.

However, the utility function only assigns values to terminal states. If MAX's

possible next actions do not all lead to terminal states, she needs a way to choose between non-terminal states as well. This can be done by computing the *minimax value* of these states, representing the utility for MAX of being in that state assuming that both players play optimally throughout the rest of the game.

By definition, the minimax value of a terminal state is given by the utility function. The minimax value of a non-terminal state is determined by the fact that MAX will choose actions maximizing the minimax value, and MIN will choose actions minimizing the minimax value. Putting it all together, the minimax value $V_m(s)$ of a state $s \in S$ is recursively defined by

$$V_m(s) = \begin{cases} \mathit{utility}(s) & \text{if } s \in Z \\ \max\limits_{s' \in C_s} V_m(s') & \text{if } \mathit{tomove}(s) = \text{MAX} \\ \min\limits_{s' \in C_s} V_m(s') & \text{if } \mathit{tomove}(s) = \text{MIN} \end{cases} \tag{2.1}$$

Consider the search tree in Figure 2.3. Six moves have already been played in this game of Tic-Tac-Toe, leading to the current root of the search tree. MAX (the player using the $\times$ symbol) is analyzing the search tree to find an optimal action at the root node. The search tree is fully expanded from the root to the terminal nodes. In the figure, each layer of the tree is marked MAX or MIN depending on the player to move in the nodes on that layer.

The minimax value of each state is written in the lower right corner of the corresponding node. For any terminal node, the minimax value is the value returned by the utility function for that state—0 for the two states in which the game is drawn, and 1 for the two states in which MAX has won. For any internal node with MAX to move, the minimax value is the highest minimax value among its children, since this corresponds to the optimal action choice for MAX. For any internal node with MIN to move, the minimax value is the lowest minimax value among its children. We can see that the root has a minimax value of 0, and that the optimal action choice (the *minimax decision*) for MAX at the root is the action labeled "1". This action leads to the only child guaranteeing an outcome of at least 0 for MAX even if MIN plays optimally.

2.2.2 The Minimax Algorithm

The *minimax algorithm* (von Neumann and Morgenstern, 1944), or *minimax* for short, is the classic search technique for turn-taking two-player games. It computes the minimax decision at the root state by directly implementing Equation 2.1. The minimax algorithm does this by starting at the root and recursively expanding all

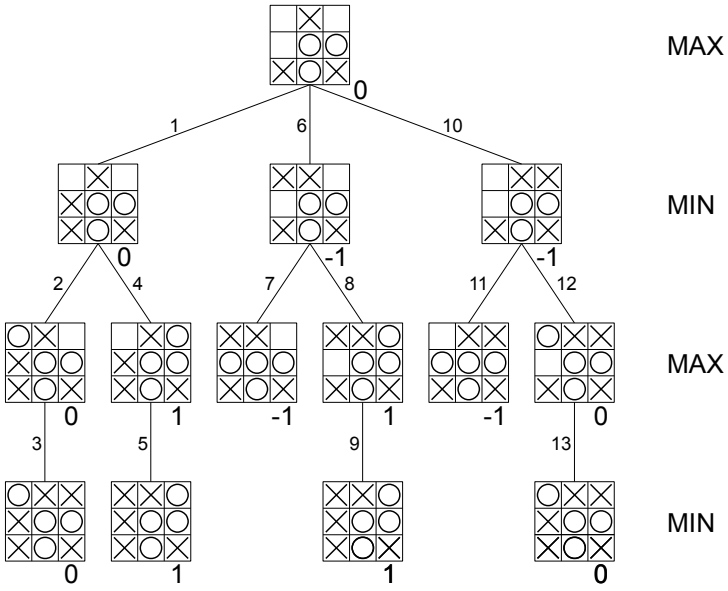


Figure 2.3: A minimax tree in Tic-Tac-Toe.

nodes of the tree in a depth-first manner. In Figure 2.3, the numbers next to the edges indicate the order in which the nodes are expanded. When a leaf of the tree is reached, the utility function is called to assign a value to that state. After values for all children of a node have been returned, the value of this node is computed as either the maximum (at MAX nodes) or minimum (at MIN nodes) of the children's values. After the entire tree has been traversed, the minimax values of all nodes are known. We can then return the minimax decision by picking an action at the root which leads to a child of maximal value (if MAX is to move) or minimal value (if MIN is to move). In Figure 2.3, this is the $\times$ player's move in the left column of the middle row. Algorithm 2.1 shows pseudocode for minimax. The initial call e.g. by MAX is `MINIMAX(rootstate,1)`.

This implementation uses separate code for MAX and MIN nodes, and needs two different recursive calls for them. In practice, it is often simplified to the *negamax* formulation (Knuth and Moore, 1975), which treats MAX and MIN nodes uniformly. The basic idea of negamax is that $\min(a, b) = -\max(-a, -b)$. It does not compute minimax values, but *negamax values* $V_n(s)$ as defined by

```
MINIMAX(state,currentPlayer) {
   if(state.isTerminal()) {
      return utility(state)
   } else if(currentPlayer=MAX) {
      return     max    MINIMAX(s,-1)
              s∈children(state)
   } else if(currentPlayer=MIN) {
      return     min    MINIMAX(s,1)
              s∈children(state)
   }
}
```

Algorithm 2.1: Minimax.

$$V_n(s) = \begin{cases} \texttt{currentPlayer} \times \mathit{utility}(s) & \text{if } s \in Z \\ \max\limits_{s' \in C_s} -V_n(s') & \text{otherwise} \end{cases} \tag{2.2}$$

where the variable `currentPlayer` takes the value of 1 in MAX nodes, and -1 in MIN nodes. Negamax values are identical to minimax values in MAX nodes, and their negation in MIN nodes. The pseudocode of negamax is shown in Algorithm 2.2. Negamax thus replaces alternating maximization and minimization when traversing the tree with maximization throughout. It does so by returning utility values of terminal nodes from the point of view of the player to move instead of MAX, and by negating backpropagated values from layer to layer. The initial call e.g. by MAX is `NEGAMAX(rootstate,1)`.

Both the minimax and the negamax algorithms explore the complete game tree

```
NEGAMAX(state,currentPlayer) {
   if(state.isTerminal()) {
      return currentPlayer × utility(state)
   } else {
      return     max    -NEGAMAX(s,-currentPlayer)
              s∈children(state)
   }
}
```

Algorithm 2.2: Negamax.

rooted in the current state. If the maximum depth of the tree is d, and if there are b legal moves in every state (the *branching factor*), then the time complexity of the minimax algorithm is $O(b^d)$. As minimax investigates the game tree in a depth-first manner, the space complexity is $O(bd)$ (assuming that all legal actions have to be generated at once in each node). Its time complexity makes naive minimax infeasible for non-trivial games, but it serves as the foundation for more practically useful algorithms such as those presented in the following.

2.2.3 $\alpha\beta$ Search

Minimax needs to visit every node in the game tree in order to find the best action at the root node. However, closer analysis of the search process shows that in most cases, there are many nodes or entire subtrees whose values are irrelevant for finding the minimax decision. These subtrees can therefore safely be ignored without affecting the result of minimax, speeding up the search significantly. The technique of excluding subtrees from the search is called *pruning*. The most-used pruning technique for minimax is $\alpha\beta$ pruning (Knuth and Moore, 1975).

Consider the search tree in Figure 2.4, starting from the same initial state as Figure 2.3. After traversing the first six nodes, the root A and the subtree of node B, we know that the minimax value of B is 0. This means that MAX is guaranteed a value of *at least* 0 in the root node. MAX is not interested in actions leading to values ≤ 0 anymore, since they are no improvement to action 1. Next, the search process generates nodes C and E, where MIN wins the game. It is now clear that if MAX chose action 6, MIN could play action 7 and get a result of -1. The value of C is thus no greater than -1, and action 6 cannot improve on action 1 anymore. The subtree below action 6 does not have to be explored any further, as $\max(0, \min(-1, x)) = 0$ independently of the value of x. With the same reasoning, action 8 is proven to be no improvement to action 1 as well. Two subtrees can thus be pruned from the search without influencing the final decision. $\alpha\beta$ pruning reduces the number of visited nodes in this example from 14 to 10.

The name $\alpha\beta$ comes from the two parameters α and β of the algorithm, representing in any node throughout the search the value that is already guaranteed for MAX, and the value that is already guaranteed for MIN. α and β form a lower bound and an upper bound on the unknown minimax value of the root, the *$\alpha\beta$ window*. If any node returns a value outside of this interval, we know that we are currently exploring a suboptimal subtree that either MAX or MIN would avoid under optimal play. We can then safely prune the remaining children of the current node (*$\alpha\beta$ cutoff*).

Algorithm 2.3 shows pseudocode for minimax search with $\alpha\beta$ pruning, or $\alpha\beta$ for short. It is written in the negamax formulation, so the $\alpha\beta$ window is inversed between

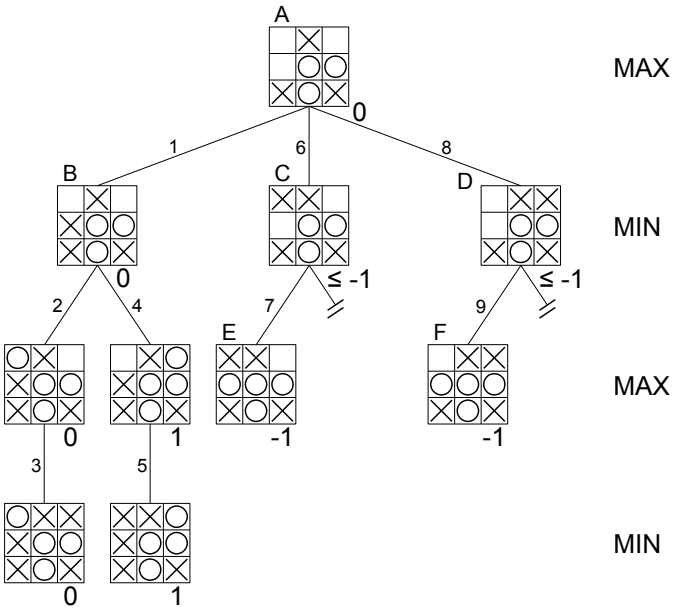


Figure 2.4: An $\alpha\beta$ tree in Tic-Tac-Toe.

recursive calls. The initial call e.g. by MAX is `ALPHABETA(rootstate,1,`$-\infty,\infty$`)`.

In practice, minimax without the $\alpha\beta$ enhancement is only used for teaching purposes. All minimax implementations in this thesis use $\alpha\beta$ pruning.

2.2.4 Move Ordering and k-best Pruning

Move Ordering

The effectiveness of $\alpha\beta$ strongly depends on the *order* in which child nodes are generated and recursively analyzed in line 6 of Algorithm 2.3. As an example, compare Figure 2.4 to Figures 2.5 and 2.6. All three figures present the search tree of an $\alpha\beta$ search from the same initial state, leading to the same result of node B being the optimal child of the root. However, the nodes are visited in different order. In Figure 2.4, all nodes are analyzed in optimal order. At the root for example, the best child node B is traversed first. This allows $\alpha\beta$ to establish the guaranteed value of 0 for MAX early in the search, leading to two cutoffs and a total number of only 10 visited nodes. In Figure 2.5, B is examined second at the root—with the result that only one $\alpha\beta$ cutoff is possible, increasing the total number of visited nodes to 12. In Figure 2.6 finally, node B is chosen last at the root. In this case, no cutoffs are possible at all,

```
1  ALPHABETA(state,currentPlayer,α,β) {
2     if(state.isTerminal()) {
3        return currentPlayer × utility(state)
4     }
5     forall(c ∈ children(state)) {
6        α = max(α,-ALPHABETA(c,-currentPlayer,-β,-α))
7        if(α ≥ β) {
8           return β
9        }
10    }
11    return α
12 }
```

Algorithm 2.3: $\alpha\beta$ search.

and $\alpha\beta$ has to traverse the same full tree of 14 nodes that minimax without $\alpha\beta$ would generate.

In the much larger trees examined in non-trivial games, these differences are much more significant. In the best case of searching all nodes in the optimal order, $\alpha\beta$ has a time complexity of only $O(b^{d/2})$ compared to the $O(b^d)$ of minimax (Knuth and Moore, 1975). This best case reduces the effective branching factor from b to $\sqrt{b}$, which in Chess for example means looking at only about 6 actions per state instead of about 35 actions on average. As a result, $\alpha\beta$ can search trees twice as deep as minimax in the same amount of time.

In practice, the optimal order of nodes is of course unknown before the search has ended. A program that could perfectly order nodes would not have to perform any search at all, but could simply execute the action leading to the best node at the root. However, the strong effect of the order in which nodes are visited—or equivalently, the order in which moves are generated and examined—has led to the development of various *move ordering* techniques. These techniques can be divided into two classes. *Static* move ordering techniques use domain knowledge independent of the current search process. In Chess for example, capturing moves tend to be more promising than other moves, so it improves move ordering to search them first. *Dynamic* move ordering techniques use information acquired from the search currently running. The *killer heuristic* (Akl and Newborn, 1977) for example is based on the idea that a move which was good (produced an $\alpha\beta$ cutoff) in one state might also be good in other, similar states. Static move ordering is used in Chapter 8 of this thesis.

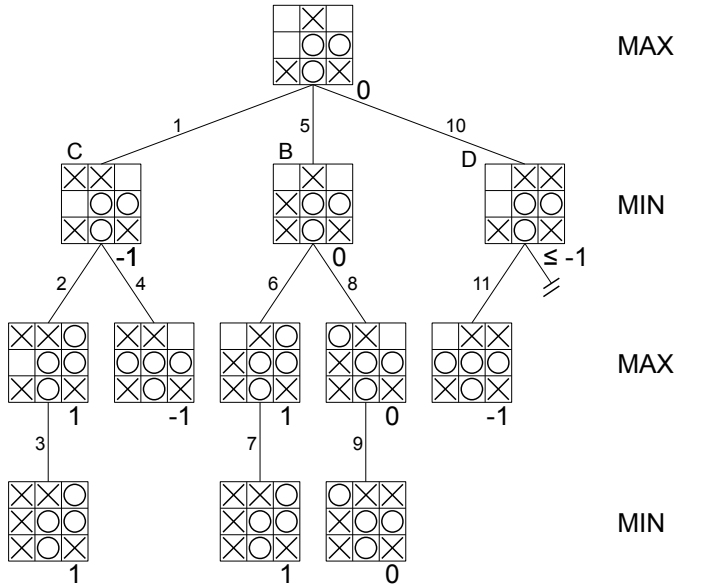


Figure 2.5: An $\alpha\beta$ tree with suboptimal move ordering in Tic-Tac-Toe.

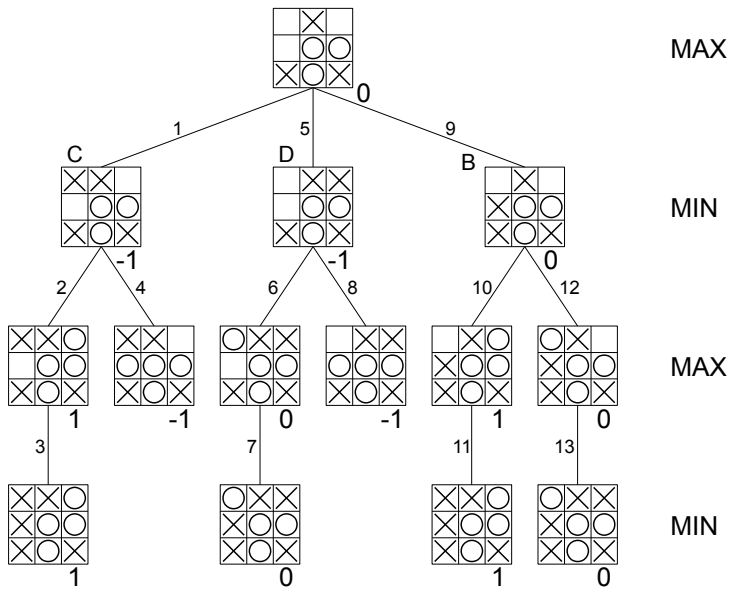


Figure 2.6: An $\alpha\beta$ tree with pessimal move ordering in Tic-Tac-Toe.

k-best Pruning

Move ordering as discussed so far affects the performance of $\alpha\beta$, but not the result. The only moves that are pruned from the search tree are the ones that provably do not influence the minimax decision at the root. In some cases, it is effective to go further than that and to use move ordering techniques not only to search moves in the order that appears most promising, but to exclude the least promising moves entirely from the search. When the search is restricted to the k actions in each state that are ranked most highly by a move ordering method, this technique is called *k-best pruning*. It is a type-B strategy according to Shannon (1950). Such selective strategies considering only a subset of moves were popular in early research on Chess, when computational power was limited. k-best pruning does not guarantee the same result as a regular minimax or $\alpha\beta$ search anymore. But while there is a risk of occasionally making mistakes and pruning good moves, the advantage of a smaller effective branching factor—and thus deeper searches in the same time—can be substantial.

Figure 2.7 shows an example of k-best pruning on the Tic-Tac-Toe search tree we have seen before. The moves are here searched according to the following static move ordering: first any move that blocks the opponent from making three-in-a-row is tried, then all other moves in random order. In this example $k = 2$, meaning that no more children than 2 are searched for any node in the tree. This is why the search stops after examining the root children B and D, instead of looking at C as well. Below node D, there is an additional cut due to $\alpha\beta$ pruning. In this case, the search determines the correct minimax value (0) and optimal decision (action 1) at the root, after visiting only 8 nodes. Note though that the k-best cut is not always correct—its success depends on the quality of the move ordering in the game at hand. Move ordering and k-best pruning are used in Chapter 8.

2.2.5 Depth-Limited Search

$\alpha\beta$ pruning provides a strong improvement to minimax search, allowing it in the best case to search twice as deep in the same time. However, even $\alpha\beta$ as presented so far still has to expand the tree all the way to some terminal states where the utility function can be applied. In non-trivial games, this is usually impossible for all but the latest moves in the game.

In order to make minimax search useful in practice, Shannon (1950) therefore proposed to expand the minimax tree only to a given depth, whether this is deep enough to reach terminal states or not. Since the recursive minimax calls need a base case, this requires the search process to assign values to non-terminal leaves as well. Unfortunately, their exact minimax values are not known. For this purpose, the utility function returning the *exact* utilities of *terminal* states is replaced with a function

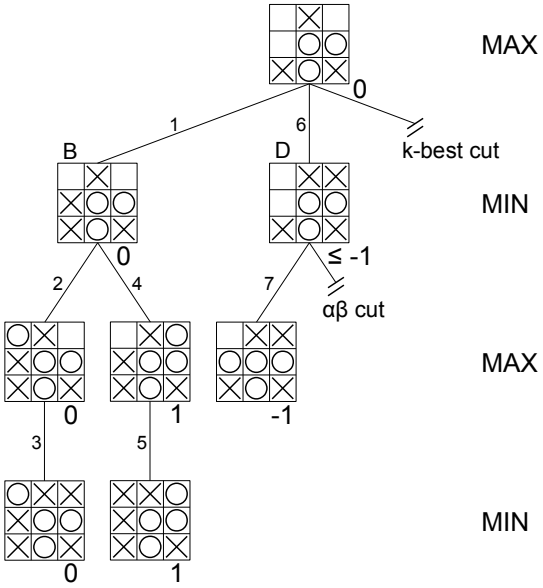


Figure 2.7: An $\alpha\beta$ tree with k-best pruning ($k = 2$) in Tic-Tac-Toe.

returning an *estimated* utility for *any* given leaf state. This function is called a static *heuristic evaluation function* and allows for depth-limited search.

Similar to a static move ordering, a heuristic evaluation function encapsulates the domain knowledge of the programmer. The quality of an evaluation function strongly influences the success of the search—since the search process does not look beyond the leaves of the tree, it depends on their accurate evaluation. Russell and Norvig (2002) list three requirements for a good evaluation function. First, it should assign the same values to terminal states as the utility function does. Second, it should be fast, which is why it is static and not conducting a search itself. Third, it should return values for non-terminal leaves that are strongly correlated with the actual minimax value, with the chances of winning the game, or the profitability of a state for the player (Donkers, 2003). In many games evaluation functions are constructed by linearly combining several domain-dependent features, such as material balance or mobility of pieces in Chess, or territorial balance in Amazons. We define and use evaluation functions for several games in in Chapter 8.

Algorithm 2.4 shows pseudocode for depth-limited $\alpha\beta$ search. The following lines have changed compared to Algorithm 2.3. In line 1, $\alpha\beta$ receives a new parameter indicating the remaining search depth. In line 2, $\alpha\beta$ stops searching not only when a state is terminal, but also when the depth parameter has reached 0. In line 3, the value returned at such a leaf state is computed by the evaluation function, not the

utility function. In line 6 finally, the recursive call decrements the depth parameter. The initial call e.g. by MAX is `ALPHABETA(rootstate,1,`$-\infty$`,`∞`,`d`)` where d is the desired search depth. All $\alpha\beta$ implementations in this thesis are of this type.

```
1  ALPHABETA(state,currentPlayer,α,β,depth) {
2     if(state.isTerminal() or depth≤0) {
3        return currentPlayer × evaluation(state)
4     }
5     forall(c ∈ children(state)) {
6        α = max(α,-ALPHABETA(c,-currentPlayer,-β,-α,depth-1))
7        if(α ≥ β) {
8           return β
9        }
10    }
11    return α
12 }
```

Algorithm 2.4: Depth-limited $\alpha\beta$.

2.3 Monte-Carlo Tree Search

This section introduces the second basic method for making action decisions in games—the *Monte-Carlo Tree Search* (MCTS) family of algorithms. MCTS is the main focus of this thesis and used in all chapters. We apply it both to one-player games (see Chapters 4 and 5) and to two-player games (see Chapters 6 to 8).

2.3.1 Monte-Carlo Evaluation

In some games it is difficult to find a suitable static heuristic evaluation function of the type discussed in Subsection 2.2.5. One alternative technique for evaluating non-terminal states is *Monte-Carlo evaluation* (Abramson, 1990). For this evaluation technique, the state at hand is not statically analyzed and evaluated according to domain-specific features. Instead, the state is evaluated by the expected value of the game's result from this state on, estimated by random sampling. In order to do this, several *rollouts* are started from the state. A rollout is a fast, (semi-)random continuation of the game from the state to be evaluated all the way to the end of the game. In the simplest case, actions are simply chosen uniformly random among all legal actions until a terminal state is reached. The utility of this terminal state is

then returned and stored as rollout result, and the evaluation of the state at hand is formed by the average of several such rollout results. In a two-player game with the only outcomes *win* and *loss* for example, this average can be interpreted as the probability of winning the game from that state *under (semi-)random play.*

A disadvantage of Monte-Carlo evaluation is that it is often time-consuming compared to static evaluation. The main advantage of Monte-Carlo evaluation is its domain independence—no evaluation function or other domain knowledge beyond the rules of the game is required. Researchers have applied it to e.g. Backgammon (Tesauro and Galperin, 1997), Poker (Billings et al., 1999), Bridge (Ginsberg, 2001), Scrabble (Sheppard, 2002), and Go (Brügmann, 1993; Bouzy and Helmstetter, 2004).

2.3.2 The MCTS Framework

Because Monte-Carlo evaluation is computationally expensive, it typically cannot be applied to every leaf node of a large tree such as those traversed by $\alpha\beta$. In order to be able to evaluate all leaf nodes, Bouzy and Helmstetter (2004) for example restricted their tree search in the game of Go to one ply. However, this means that the search process cannot find the minimax decision at the root, even when assuming infinite time. Optimal play beyond the first ply cannot be discovered. Bouzy (2006) iteratively expanded search trees to greater depths, but pruned less promising nodes after each expansion step in order to control the size of the tree. As this pruning was irreversible, convergence to optimal play was again not guaranteed.

The breakthrough for combining Monte-Carlo evaluation with tree search came with the development of *Monte-Carlo Tree Search* (MCTS) (Coulom, 2007b; Kocsis and Szepesvári, 2006). MCTS constructs a search tree for each move decision in a best-first manner. This tree starts from the current state, represented by the root node, and is selectively deepened into the direction of the most promising actions. Promising actions are chosen according to the results of Monte-Carlo rollouts starting with these actions. Unlike the $\alpha\beta$ tree which is traversed depth-first, the best-first MCTS tree is kept in memory. Each node added to the tree stores the current value estimate for the state it represents, which is continuously updated and improved throughout the search.

MCTS works by repeating the following four-phase loop until computation time runs out (Chaslot et al., 2008). Each loop represents one simulated game. The phases are visualized in Figure 2.8.

1: *Selection* phase (see Subfigure 2.8(a)). The tree is traversed, starting from the root node and using a *selection policy* at each node to choose the next action to sample. The selection policy tries to balance exploitation of nodes with high value estimates and exploration of nodes with uncertain value estimates. Exploitation means that the

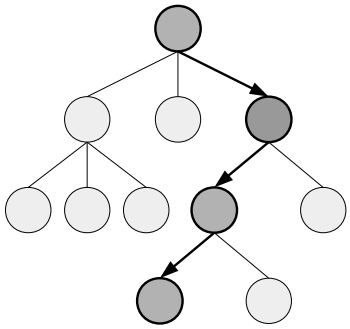
tree is deepened into the direction of the child that currently seems best. Exploration means that more samples for other children are collected in order to increase certainty on which child is the best. In this thesis, the selection policy used is UCB1-TUNED (Auer et al., 2002).

2: *Expansion* phase (see Subfigure 2.8(b)). When the traversal reaches a state that is not yet represented in the tree, a decision is made on how to grow the tree into this direction. A common expansion policy is the addition of one newly sampled node per simulation (Coulom, 2007b), representing the newly encountered state. This policy is used in this thesis. If the amount of available memory is limited, it is possible to add nodes only after the corresponding states have been visited a given number of times.

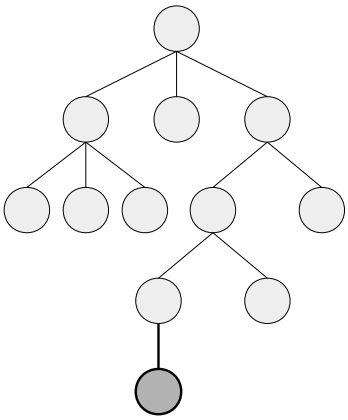
3: *Rollout* phase (see Subfigure 2.8(c)). Actions are played, starting from the state corresponding to the newly added node until the end of the game. Every action is chosen by a *rollout policy*. In the literature, rollouts are sometimes also called *playouts*, *samples* or *simulations*—in this thesis however, the term simulation refers to an entire loop through the four phases of MCTS. While uniformly random action choices are sufficient to achieve convergence of MCTS to the optimal move in the limit, more sophisticated rollout policies have been found to improve convergence speed. These can make use of domain knowledge or of domain-independent enhancements. In Part I of this thesis, one of the rollout policies used for SameGame employs domain-specific knowledge, while all other games use uniformly random rollouts. Nested MCTS searches are proposed as domain-independent improved rollout strategies for higher-level searches. In Chapter 6 of Part II, Go uses an informed rollout policy with both domain-dependent and domain-independent enhancements, whereas the rollouts in all other games are uniformly random. Chapters 7 and 8 propose and compare various improvements to MCTS rollouts.

4: *Backpropagation* phase (see Subfigure 2.8(d)). Once the end of the rollout has been reached and the winner of the simulated game has been determined, the result is used to update the value estimates stored in all nodes that were traversed during the simulation. The most popular and effective backpropagation strategy stores the average result of all rollouts through the respective node (Coulom, 2007b). It is also used in this thesis. Note that the UCB1-TUNED selection policy requires rollout results to be in the interval $[0, 1]$. In two-player games, wins are therefore represented by 1, losses by 0, and draws by 0.5 in this thesis. In one-player games, the range of possible game outcomes is mapped to $[0, 1]$ before backpropagation.

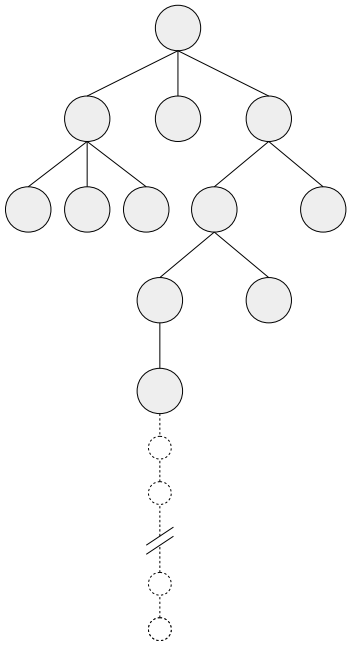
For the case of a two-player zero-sum game, Figure 2.9 shows in more detail how value estimates are stored and used in a typical MCTS implementation such as used in this thesis. Each node contains a visit counter v and a win counter w. These counters



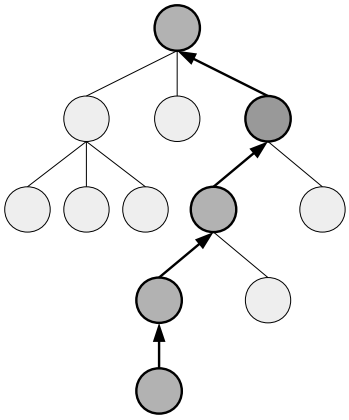
(a) The selection phase. The selection policy is applied recursively until an unsampled action is reached.



(b) The expansion phase. The newly sampled action is executed and the resulting state is added to the tree.



(c) The rollout phase. One simulated game is played by the rollout policy.



(d) The backpropagation phase. The result of the rollout is used to update value estimates in the tree.

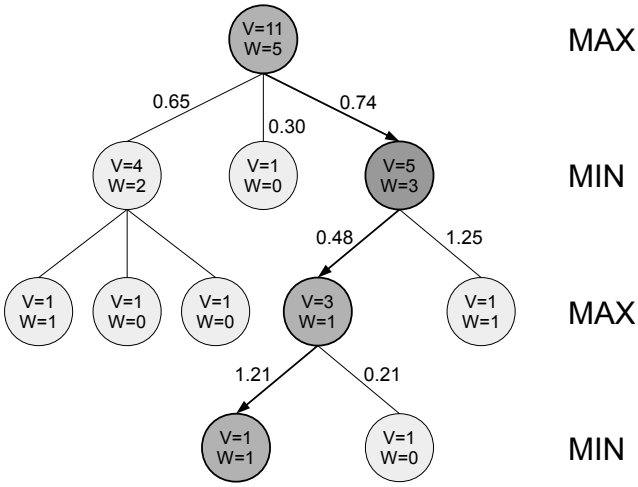
Figure 2.8: MCTS.

are used by the selection policy to compute a value for each child of the current node. The child with the maximal value is chosen for sampling. After each rollout, the visit counters of the traversed nodes are incremented, and the rollout result (1 for a win, 0 for a loss, or 0.5 for a draw) is added to the win counters from the point of view of the player to move in the corresponding state (MAX or MIN). The flipping of rollout results from layer to layer is similar to the negation between layers in the negamax algorithm (see Subsection 2.2.2). In one-player games, rollout results do not have to be flipped from level to level, as the search always takes the point of view of the MAX player.

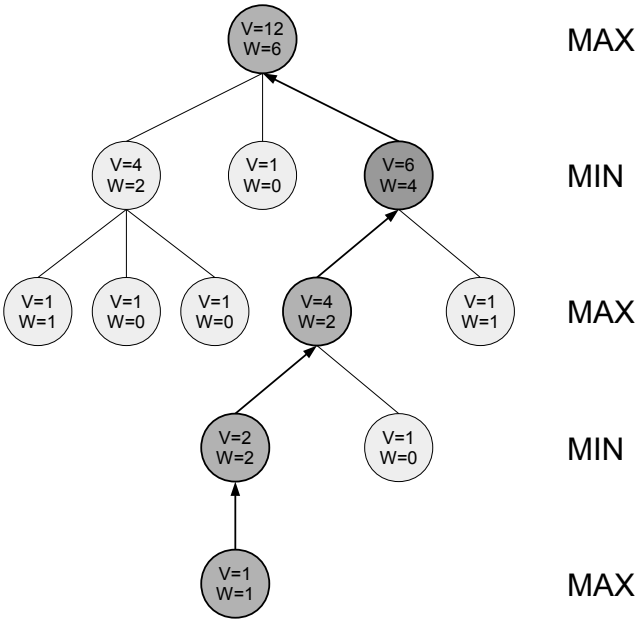
MCTS can be interrupted after any number of iterations to return the current decision on the best child of the root. Several *final move selection* strategies are possible, notably returning the child with the currently highest value estimate, or the child with the currently highest number of rollouts. At long time settings, the performance difference between the two strategies tends to be negligible, while at short time settings, the latter strategy can be more robust (Chaslot et al., 2008). Choosing the most-sampled move is popular in Go research, and was chosen for all games in this thesis.

Since MCTS is based on sampling, it does not require a heuristic evaluation function, but only the ability to generate simulated game trajectories in the domain at hand. Because it grows a highly selective, best-first search tree guided by its simulations, it can handle search spaces with large branching factors. By using Monte-Carlo rollouts, MCTS can take long-term effects of actions into account better than a depth-limited search. Combined with appropriate selection policies to trade off exploration and exploitation (for example with the UCB1 policy in Kocsis and Szepesvári 2006), the search tree spans the entire game tree in the limit, and the state value estimates converge to the game-theoretic values in one- and two-player games (Kocsis and Szepesvári, 2006; Kocsis et al., 2006)—in the case of two-player zero-sum games, to the minimax values. In addition, MCTS can be stopped after every rollout and return a move choice that makes use of the complete search time so far, while $\alpha\beta$ searchers can only make use of completely explored root moves of a deepening iteration.

Algorithm 2.5 shows pseudocode of MCTS for two-player domains, and Algorithm 2.6 represents MCTS for one-player domains. The only difference is additional bookkeeping in the one-player version for the current high score and the simulation that achieved it. The reason is that in a two-player domain, the decisions of the opponent make the future uncertain. MCTS tries to find a next action that maximizes the expected outcome of the player under the assumption that the opponent tries the same for herself. In one-player domains however, all actions are under the player's control. MCTS can therefore search not only for the next action to take, but for an entire outcome-maximizing action sequence until a terminal position is reached.



(a) The selection phase.



(b) The backpropagation phase.

Figure 2.9: An example two-player MCTS tree. Each node is labeled with its visit counter v and win counter w. The numbers on the edges represent the values assigned by the selection policy UCB1-TUNED to the respective moves.

```
MCTS(startState) {
   for(numberOfIterations) {
      currentState ← startState
      simulation ← {}
      # selection
      while(currentState ∈ Tree) {
         currentState ← takeSelectionPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      # expansion
      addToTree(currentState)
      # rollout
      while(currentState.notTerminalPosition) {
         currentState ← takeRolloutPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      # backpropagation
      score ← cumulativeReward(simulation)
      forall(state ∈ {simulation ∩ Tree}) {
         state.value ← backPropagate(state.value, score)
      }
   }
   return finalMoveChoice(Tree)
}
```

Algorithm 2.5: Two-player MCTS.

```
MCTS(startState) {
   bestScore ← -Infinity
   bestSimulation ← {}
   for(numberOfIterations) {
      currentState ← startState
      simulation ← {}
      # selection
      while(currentState ∈ Tree) {
         currentState ← takeSelectionPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      # expansion
      addToTree(currentState)
      # rollout
      while(currentState.notTerminalPosition) {
         currentState ← takeRolloutPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      # backpropagation
      score ← cumulativeReward(simulation)
      forall(state ∈ {simulation ∩ Tree}) {
         state.value ← backPropagate(state.value, score)
      }
      if(score > bestScore) {
         bestScore ← score
         bestSimulation ← simulation
      }
   }
   return (bestScore, bestSimulation)
}
```

Algorithm 2.6: One-player MCTS.

2.4 MCTS Enhancements

The name MCTS represents a family of search algorithms which modify, adapt, or improve the basic MCTS framework presented above in a variety of ways. There has been research on enhancements for every phase of MCTS. This section briefly discusses the enhancements used by the MCTS implementations in this thesis. Where more detailed explanations are necessary, these can be found in the relevant chapters.

Three enhancements of the selection phase are used in this thesis. The *MCTS-Solver* (Winands et al., 2008) improves selection by allowing MCTS to prove the minimax value of nodes. It is used in Chapters 7 and 8; see Section 7.1 for details. *Node priors* (Gelly and Silver, 2007) improve selection by taking prior domain knowledge into account. They are used in Chapter 8; see Subsection 8.2.3 for an in-depth explanation. *Rapid Action Value Estimation* (RAVE) (Gelly and Silver, 2007) improves selection by combining regular MCTS value estimates with a second type of value estimate in every node. This second estimate is based on the assumption that the order in which rollout actions were visited from the current node onwards is irrelevant—all of them are updated as if they had been the first action. This allows many updates per rollout, and in domains such as Go where the assumption is true to at least some degree it can help guide search when only few rollout returns are available. RAVE is used for Go in Chapter 6.

Three types of enhancements of the rollout phase are used in the following chapters. The first type are *informed rollout policies*, depending on knowledge acquired offline before the start of the search. This can be hand-coded domain knowledge, or knowledge gained through machine learning techniques. The rollout policy used for Go in Chapter 6 and the TabuColorRandomPolicy (Schadd et al., 2008b) used for SameGame in Chapters 4 and 5 are such informed rollout policies. In Chapter 8, informed rollout policies are constructed from heuristic evaluation functions for several two-player games. See the respective chapters for details. The second type are *adaptive rollout policies* which learn from knowledge acquired online during the search. This includes the Last-Good-Reply policy with forgetting (Baier and Drake, 2010) which stores and re-uses moves that have been successful answers to opponent moves in previous rollouts. It is used for Go in Chapter 6. The third type are *rollout cutoffs* (Lorentz, 2008; Winands et al., 2010), a technique where the rollout is stopped before reaching a terminal state, and an evaluation of the reached state is used to compute the rollout result. This is used in Chapter 8 and explained in more detail in Subsection 8.2.2.

Another enhancement used by the Go program in Chapter 6 is a *transposition table* (Childs et al., 2008; Greenblatt et al., 1967). This is not an MCTS enhancement alone, but can be applied to other search algorithms such as $\alpha\beta$ as well. It is based on the fact that in many games, identical states can be reached through different

sequences of actions. These states are called *transpositions*. In a search tree, this leads to the same state being represented by different tree nodes. If the history of the game leading to this state is irrelevant for the search process, these nodes can be replaced by one, which can reduce search effort. The state then does not have to be explored several times, but search results can be reused. The game tree is thus changed into a *game graph* in which every state is represented by exactly one node. All other MCTS implementations in this thesis do not take transpositions into account.

2.5 A Learning View on MCTS

Section 2.3 introduced MCTS from a *game-tree search* point of view. In order to form a connection to other fields of research in computer science and AI, this section gives an alternative introduction to MCTS from a *learning* point of view. Additionally, it puts some of the enhancements discussed in Section 2.4 into a common framework of learning.

2.5.1 MCTS in the Framework of Learning

Reinforcement learning (see Sutton and Barto (1998) for an introduction) is the study of learning from interaction how to achieve a goal. It deals with the problem of learning optimal behavior, without being given descriptions or examples of such, solely from acting and observing the consequences of actions. The classic reinforcement learning task consists of an interactive loop between a learning *agent* and its *environment*: The agent repeatedly observes its situation—the *state* of the environment—, chooses an *action* to perform, and receives a response in form of a numerical *reward* signal, indicating success or failure. In many cases, the agent's actions can also affect the next state. Trying to maximize its cumulative reward in the long run, the agent therefore has to learn by trial-and-error how his action choices influence not only immediate, but also delayed rewards. This subsection briefly outlines some of the basic concepts of reinforcement learning, with emphasis on the class of Monte-Carlo methods. This forms a foundation for the presentation of MCTS as a reinforcement learning algorithm.

The Multi-Armed Bandit Problem

An agent faced with the reinforcement learning problem has to learn from its own experience, without explicit guidance or supervision. One typical challenge of this task is the tradeoff between exploration and exploitation. *Exploration* means trying out new behaviors, choosing actions that have not been tried before, in order to determine their effects and returns. *Exploitation* denotes the choice of actions that are known to

be successful, the application of learned knowledge, in order to generate the maximal reward.

The simplest setting in which this task can be studied is the case of the one-state environment. In the so-called *multi-armed bandit* problem (Robbins, 1952), an analogy is drawn to slot machines, casino gambling machines also known as "one-armed bandits" because they are operated by a single lever. The multi-armed bandit problem confronts the gambler with a machine with a number of arms instead, each of which provides a reward drawn from its own probability distribution. The gambler, initially without knowledge about the expected values of the arms, tries to maximize his total reward by repeatedly trying arms, updating his estimates of the reward distributions, and gradually focusing on the most successful arms. Exploitation in this scenario corresponds to choosing the arm with currently highest estimated value. Exploration corresponds to choosing one of the seemingly suboptimal arms in order to improve its value estimate, which may lead to greater accumulated reward in the long run.

Formally, the multi-armed bandit problem is defined by a finite set of arms or actions $A = \{1, \ldots, a_{\max}\}$, each arm $a \in A$ corresponding to an independent random variable X_a with unknown distribution and unknown expectation μ_a. At each time step $t \in \{1, 2, \ldots\}$, the gambler chooses the next arm a_t to play depending on the past sequence of selected arms and obtained rewards, and the bandit returns a reward r_t as a realization of X_{a_t}. Let $n_a(t)$ be the number of times arm a has been played during the first t time steps. Then the objective of the gambler is to minimize the *cumulative regret* defined by

$$\mu^* n - \sum_{a=1}^{a_{\max}} \mu_a E\left[n_a(t)\right] \tag{2.3}$$

where $\mu^* = \max_{1 \leq i \leq A} \mu_i$ and $E\left[n_a(t)\right]$ denotes the expected value of $n_a(t)$.

Many algorithms have been developed for choosing arms in the multi-armed bandit problem. Lai and Robbins (1985) showed that the best regret obtainable grows logarithmically with the number of time steps t. Auer et al. (2002) achieved logarithmical regret not only in the limit, but uniformly over time. Their UCB1-TUNED algorithm chooses at each time step the arm that maximizes a formula combining an exploitation term and an exploration term. The exploitation term represents the current estimate for the expected reward of the arm, and the exploration term represents an upper confidence bound for the expected reward. UCB1-TUNED is used by MCTS in this thesis. It is defined in Formula 2.7 below.

Markov Decision Processes

In the full reinforcement learning problem, the agent has to learn how to act in more than one situation, and explores the consequences of its actions both with regard to immediate rewards received, and to changing states of the environment. *Markov decision processes* (MDPs) represent a classic framework for modeling this problem. A deterministic MDP is defined as a 4-tuple $(S, A, T(\cdot,\cdot), R.(\cdot,\cdot))$, where S is the set of *states* of the environment, A is the set of *actions* available to the agent (with $A(s) \subseteq A$ being the set of actions available in state $s \in S$), T is the *transition function* with $T(s,a) = s'$ iff choosing action a in state s at time t will lead to state s' at time $t+1$, and R is the *reward function* with $R(s,a)$ being the direct reward given to the agent after choosing action a in state s. Note that this is similar to the one-player case of a game as described in Section 2.1, where for all states s and s' there exists an action a with $T(s,a) = s'$ iff $s \xrightarrow{a} s'$. MDPs with finite sets S and A are called *finite MDPs*.

In this thesis, we are dealing with *episodic* tasks (Sutton and Barto, 1998). In episodic tasks, the agent's experience can naturally be divided into independent sequences of interactions (independent games) leading from a start state to one of a number of terminal states. The agent chooses an action $a_t \in A$ based on the current state $s_t \in S$ of the environment at each discrete time step $t \in \{1, 2, 3, \ldots, t_{\max}\}$, where $t_{\max}$ is the final time step of the episode. The environment then returns a new state s_{t+1} and a reward r_{t+1}. The agent chooses its actions according to a *policy*, a mapping $\pi(s,a) = Pr(a_t = a | s_t = s)$ from states of the environment to probabilities of selecting each possible action when in those states.

The goal of the agent is to find a policy that at any point in time t maximizes the *expected return*, the expected cumulative reward $R_t = \sum_{k=t+1}^{t_{\max}} r_k$. In value-based reinforcement learning, this is accomplished by learning a *value function* $V^\pi(s) = E_\pi [R_t | s_t = s]$ representing the expected return when starting in a given state s and following policy π thereafter. For every finite MDP, there is a unique *optimal value function* V^* defined by $\forall s \in S.\ V^*(s) = \max_\pi V^\pi(s)$, and at least one *optimal policy* π^* achieving V^*.

Value-based RL algorithms typically find an optimal policy via *policy iteration*. This process alternatingly computes the value function V^π of the current policy π (policy evaluation), and uses the newfound V^π to derive a better policy π' (policy improvement).

Monte-Carlo Planning and Search in MDPs

Model-based reinforcement learning methods assume that the transition and reward functions of the environment are known to the learning agent. *Model-free* methods require only experience and no prior knowledge of the environment's dynamics. *Monte-*

Carlo methods are a class of model-free policy evaluation algorithms specifically tailored to episodic tasks. Since episodic tasks provide well-defined returns for all visited states at the end of each episode, the return of a given state can be estimated by averaging the returns received after visiting that state in a number of episodes. According to the law of large numbers, such Monte-Carlo estimates converge to the true value function of the current policy as the agent collects more and more experience in its environment.

Monte-Carlo approaches have several advantages. First, they require no prior understanding of the environment's dynamics. Second, they naturally focus learning on the states and actions that are actually relevant to the agent, which is often a small subset of the entire state and action spaces. Third, given a generative model of the environment—a model that is able to draw samples from the transition function—Monte-Carlo methods can be applied to simulated experience (*rollouts*), without actually interacting with the environment. This process is called *planning*.

As opposed to *learning*, planning produces or improves an agent policy solely through internal computation. Through a sample model or forward model, an agent can select hypothetical actions, sample their hypothetical consequences, and collect hypothetical experience about their values. If the model approximates reality closely enough, the policy learnt through internal policy iteration will succeed when applied to the real environment of the agent. In game programs, the sample model is typically constructed by modelling the opponent's behavior with a policy similar to that of the agent, but with the opposite goal. This can be understood as an extension of the reinforcement learning problem to two agents, taking turns in executing their actions.

If planning is focused on improving an agent policy only for the *current* state, it is called *search* (Sutton and Barto, 1998). Other than general planning algorithms which aim at finding optimal actions for all states in the state space, or at least for states likely to be visited by the current policy as in Monte-Carlo planning, search is only concerned with finding the agent's optimal *next* action or action sequence.

Monte-Carlo Tree Search

As mentioned above, the task of playing a game can be described as a reinforcement learning problem by viewing the game's positions as states of the environment, the game's moves as actions of the agent, the game's rules as defining the transition function, and the results of the game as rewards for the agent. In this light, MCTS is a value-based reinforcement learning technique, which in combination with a generative model becomes a search technique. For each action decision of the agent, MCTS constructs a search tree $T \subseteq S$, starting from the current state as root. This tree is selectively deepened into the direction of the most promising actions, which are

determined by the success of Monte-Carlo rollouts starting with these actions. After n rollouts, the tree contains nodes for $n+1$ states, for which distinct estimates of V^π are maintained. For states outside of the tree, values are not explicitly estimated, and moves are chosen randomly or according to an informed rollout policy.

In a variety of applications, a variant of MCTS called *Upper Confidence Bounds for Trees* (UCT) (Kocsis and Szepesvári, 2006) has shown excellent performance (Browne et al., 2012). UCT uses the UCB1 formula, originally developed for the multi-armed bandit problem (Auer et al., 2002), to select states in the tree and to trade off exploration and exploitation. In this thesis, a variant of UCT with the selection policy UCB1-TUNED is used. This policy takes the empirical variance of actions into account and has been shown to be empirically superior to UCB1 in several multi-armed bandit scenarios (Auer et al., 2002) as well as within MCTS e.g. in the game of Tron (Perick et al., 2012; Lanctot et al., 2013).

Described in the framework of policy iteration, there are two interacting processes within MCTS.

Policy evaluation: In the backpropagation phase after each episode of experience, the return (cumulative reward) from that episode is used to update the value estimates of each visited state $s \in T$.

$$n_s \longleftarrow n_s + 1 \tag{2.4a}$$

$$\hat{V}^\pi(s) \longleftarrow \hat{V}^\pi(s) + \frac{r - \hat{V}^\pi(s)}{n_s} \tag{2.4b}$$

where n_s is the number of times state s has been traversed in all episodes so far, and r is the return received at the end of the current episode.

Policy improvement: During each episode, the policy adapts to the current value estimates. In case of a deterministic MDP and MCTS using UCB1-TUNED in the selection phase, and a uniformly random policy in the rollout phase, let

$$U^{\mathrm{Var}}(s,a) = \left(\frac{1}{n_{s,a}} \sum_{t=1}^{n_{s,a}} r_{s,a,t}^2\right) - \left(\frac{1}{n_{s,a}} \sum_{t=1}^{n_{s,a}} r_{s,a,t}\right)^2 + \sqrt{\frac{2 \ln n_s}{n_{s,a}}} \tag{2.5}$$

be an upper confidence bound for the variance of action a in state s, where $n_{s,a}$ is the number of times action a has been chosen in state s in all episodes so far, and $r_{s,a,t}$ is the reward received when action a was chosen in state s for the t-th time. Let

$$U^{\mathrm{Val}}(s,a) = \sqrt{\frac{\ln(n_s)}{n_{s,a}} \min\left(\frac{1}{4}, U^{\mathrm{Var}}(s,a)\right)} \tag{2.6}$$

be an upper confidence bound for the value of action a in state s. Then, the policy of the MCTS agent is

$$\pi(s) = \begin{cases} \underset{a \in A(s)}{\operatorname{argmax}} \left(\hat{V}^{\pi}\big(P_a(s)\big) + C \times U^{\mathrm{Val}}(s,a)\right) & \text{if } s \in T \\ \underset{a \in A(s)}{\operatorname{random}}(s) & \text{otherwise} \end{cases} \tag{2.7}$$

where $P_a(s)$ is the state reached from position s with action a, $\underset{a \in A(s)}{\operatorname{random}}(s)$ chooses one of the actions available in s with uniform probability, and C is an exploration factor whose optimal value is domain- and implementation-dependent.

An alternative implementation focuses on the estimation of state-action values instead of state values. The core ideas of the algorithm remain unchanged, but the nodes of the tree maintain an estimate of $Q^{\pi}(s,a)$ for every legal move $a \in A(s)$ instead of just one estimate of $V^{\pi}(s)$. In this case, the policy evaluation step for each visited state-action pair (s,a) is:

$$n_{s,a} \longleftarrow n_{s,a} + 1 \tag{2.8a}$$

$$\hat{Q}^{\pi}(s,a) \longleftarrow \hat{Q}^{\pi}(s,a) + \frac{r - \hat{Q}^{\pi}(s,a)}{n_{s,a}} \tag{2.8b}$$

where $n_{s,a}$ is the total number of times action a has been chosen from state s; and the policy improvement step is

$$\pi(s) = \begin{cases} \underset{a \in A(s)}{\operatorname{argmax}} \left(\hat{Q}^{\pi}(s,a) + C \times U^{\mathrm{Val}}(s,a)\right) & \text{if } s \in T \\ \underset{a \in A(s)}{\operatorname{random}}(s) & \text{otherwise} \end{cases} \tag{2.9}$$

if UCB1-TUNED is used for the selection phase and uniformly random actions are chosen in the rollout phase. The UCT implementation used in this thesis is of the state-action value estimation type.

When playing two-player games, two different implementations are possible, similar to the minimax and negamax algorithms described in Subsection 2.2.2. Rewards can

either be represented from the point of view of alternating players, which allows for maximization at all levels of the tree, or from the point of view of the same player throughout the tree, which has to be combined with alternating maximization and minimization. The first solution is more typical and used in this thesis as well.

2.5.2 MCTS Enhancements in the Framework of Learning

Basic MCTS as outlined above does not generalize between states—values are estimated separately for each state or state-action pair represented in the tree. In games with large numbers of states and actions, starting to learn from scratch for every single newly added state is time-consuming. *Generalization* techniques allow to kick start this learning by transferring some of the acquired knowledge from *similar* states as well. The definition of similarity used here determines whether this transfer is rather accurate, but rarely applicable (small number of similar states), or widely applicable, but rather noisy (large number of similar states).

In the following, s is the state reached by the sequence of actions $a_1^s a_2^s a_3^s \dots a_{l(s)}^s$ of length $l(s)$, and r is the state reached by the sequence of actions $a_1^r a_2^r a_3^r \dots a_{l(r)}^r$ of length $l(r)$. $N(s)$ is the *neighborhood* of s, that is the set of states considered similar enough to s to allow for generalization. Knowledge is transferred from $N(s)$ to s.

The selection phase of basic MCTS uses the narrowest definition of similarity, with exactly one state in each neighborhood. No generalization is possible, but the learnt values are always accurate for the state at hand.

$$N(s) = \{s\} \tag{2.10}$$

The other extreme is the widest possible neighborhood, containing all states. Value estimates for each legal action are then maintained regardless of the context in which they are played, an approach called *all-moves-as-first* (AMAF) that was used in some early work on Monte-Carlo Go (Brügmann, 1993).

$$N(s) = S \tag{2.11}$$

The Go program Orego used in this thesis (Drake et al., 2011) maintains two different value estimates for each state-action pair—the regular MCTS estimate and the RAVE estimate (see 2.4). Both estimates use a form of generalization. The regular MCTS estimate uses a transposition table, identifying each state s with any other state r which has the same configuration of stones on the board $c(r)$, the same simple

ko point[2] $k(r)$, and the same player to move $l(r) \bmod 2$:

$$N(s) = \left\{ r \in S : c(r) = c(s) \wedge k(r) = k(s) \wedge l(s) \overset{\text{mod } 2}{\equiv} l(r) \right\} \tag{2.12}$$

The RAVE estimate (Gelly and Silver, 2007) uses a neighborhood where all previously seen *successors* of a state are considered similar. When determining the action to sample, past actions chosen in the current state and in any subsequent state of previous rollouts are taken into account. Data from subsequent states are accumulating quickly during sampling, but are typically discounted due to the noise of such large neighborhoods. In computer Go, this neighborhood is so effective that in combination with other enhancements such as domain knowledge for biasing the search, it makes the exploration term of selection policies superfluous in some engines such as MoGo (Lee et al., 2009).

$$N(s) = \{ r \in S : l(r) \geq l(s) \wedge \forall t \leq l(s), a_t^s = a_t^r \} \tag{2.13}$$

Learning in the rollout phase of MCTS has become a promising research topic as well (Rimmel and Teytaud, 2010; Finnsson and Björnsson, 2010; Tak et al., 2012). Only one learning technique is used in the rollout phase in this thesis: the LGRF-2 rollout policy in Go (Baier and Drake, 2010). LGRF-2 considers each rollout move a reply to the context of the two immediately preceding moves, and maintains a reply table for each player. Replies are considered successful if the player making the reply eventually wins the simulated game. Successful replies are stored in the winning player's reply table after each rollout, and unsuccessful replies are deleted from the losing player's reply table again. In the rollout phase, replies to the last two moves are played whenever they can be found in the reply table of the current player and are legal in the state at hand. Otherwise, the default rollout policy is used as backup. For the LGRF-2 policy, the state neighborhood includes all states where the last two moves are the same[3].

$$N(s) = \left\{ r \in S : a_{l(s)}^s = a_{l(r)}^r \wedge a_{l(s)-1}^s = a_{l(r)-1}^r \wedge l(s) \overset{\text{mod } 2}{\equiv} l(r) \right\} \tag{2.14}$$

[2]The term *ko* refers to a Go-specific rule that forbids infinite loops of capturing and re-capturing.

[3]This simplified explanation ignores that during the rollout phase, LGRF-2 can back up to LGRF-1 when no reply is found. LGRF-1 uses one-move contexts, which are more frequent but less specific than two-move contexts. Analogously to LGRF-2, the neighborhood of LGRF-1 includes all states where the last move is identical.

3 Test Domains

This chapter describes the test domains used in this thesis: the one-player games *SameGame*, *Clickomania*, and *Bubble Breaker*, which are used in Part I; and the two-player games *Go*, *Connect-4*, *Breakthrough*, *Othello*, and *Catch the Lion*, which are used in Part II. The one-player games are discussed in Section 3.1, while the two-player games are described in Section 3.2. All games are deterministic perfect-information turn-taking games with discrete action spaces and state spaces.

For each game, we describe the origin, outline the rules, and provide references to previous scientific work. Furthermore, we indicate the size of the domains with the help of their *state-space complexity* and *game-tree complexity* (Allis, 1994). The state-space complexity is the number of legal positions reachable from the start position of the game. Upper bounds are often provided as an approximation due to the difficulty of determining the exact number. The game-tree complexity is the number of leaf nodes in the smallest full-width minimax search tree needed to solve the start position of the game. If no more accurate number is known from the literature, this is approximated using b^d, where b is the average branching factor and d the average length of a game in the domain at hand. These numbers are determined by self-play of our MCTS engine.

3.1 One-Player Domains

The one-player game *SameGame* was invented by Kuniaki Moribe and published under the name *Chain Shot!* in 1985 (Moribe, 1985). Eiji Fukomoto ported it to Unix in 1992 and gave it the name SameGame. It has since been ported to multiple platforms and enjoys popularity especially on mobile devices. *Clickomania* and *Bubble Breaker* are names of SameGame variants. As the rules of these games are not always defined consistently in the literature as well as in commercial products, this thesis assumes the rules given in this section.

At the beginning of the game, a two-dimensional board or grid is filled with $M \times N$ tiles of C different colors, usually randomly distributed (see Figure 3.1(a)). Each move consists of selecting a group of two or more vertically or horizontally connected, identically-colored tiles. When the move is executed, the tiles of this group are removed from the board. If there are tiles above the deleted group, they fall down. If an entire column of the board is emptied of tiles, the columns to the right shift to the left to close the gap (Figures 3.1(b) to 3.1(d) show the effects of three moves). The game ends when no moves are left to the player. The score the player receives depends on the specific variant of the puzzle:

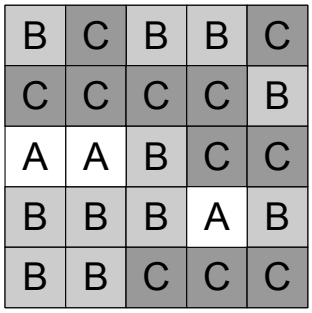
Clickomania. The goal of Clickomania is to clear the board of tiles as far as possible. At the end of each game, the player receives a score equivalent to the number of tiles removed.

Bubble Breaker. The goal of Bubble Breaker is to create and then remove the largest possible groups of tiles. After each move removing a group of size `groupSize`, the player receives a score of $\texttt{groupSize} \times (\texttt{groupSize} - 1)$ points.

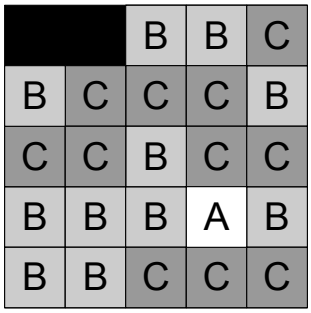
SameGame. In SameGame, both the removal of large groups and the clearing of the board are rewarded. Each move removing a group of size `groupSize` results in a score of $(\texttt{groupSize} - 2)^2$ points. Additionally, ending the game by clearing the board completely is rewarded with an extra 1000 points. If the game ends without clearing the board, the player receives a negative score. It is computed by assuming that all remaining tiles of the same color are connected into virtual groups, and subtracting points for all colors according to the formula $(\texttt{groupSize} - 2)^2$.

SameGame, Clickomania, and Bubble Breaker are popular test domains for Monte-Carlo search approaches (Cazenave, 2009; Matsumoto et al., 2010; Schadd et al., 2008a,b, 2012; Takes and Kosters, 2009). The three variants have identical move rules, but different scoring rules, resulting in different distributions of high-scoring solutions. The decision problem associated with these optimization problems is NP-complete (Biedl et al., 2002; Schadd et al., 2012).

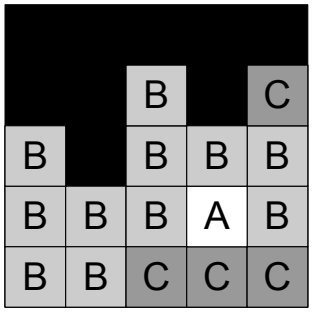
The state-space complexity for all SameGame variants can be computed as follows. The board is either empty or contains a number of non-empty columns from 1 to the width of the board w. Each non-empty column contains a number of non-empty tiles from 1 to the height of the board h. Each non-empty tile can have any of the c available colors. This leads to the following formula for computing the number of legal SameGame positions:



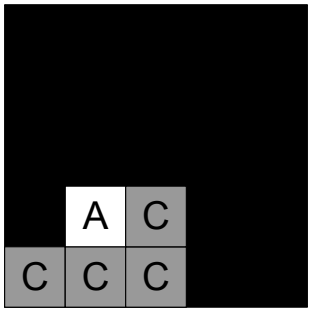
(a) A random start position on a 5×5 board with 3 colors.



(b) Result after playing A in the leftmost column as first move.



(c) Result after playing C in the leftmost column as second move.



(d) Result after playing B in the leftmost column as third move.

Figure 3.1: Moving in the SameGame family of games.

$$1 + \sum_{i=1}^{w} \left(\sum_{j=1}^{h} c^j \right)^i \tag{3.1}$$

For the parameters used in this thesis—15×15 boards with 5 colors for SameGame and Bubble Breaker, and 20×20 boards with 10 colors for Clickomania—this results in 5.27×10^{158} and 8.23×10^{400} states, respectively.

For a fixed initial position a conservative upper bound for the number of reachable states can be given as follows. We assume that for every move, the number of legal moves is equal to the maximal number of legal moves on any board with the current number of tiles. This maximal number is the number of remaining tiles, divided by two (the minimum size of a group) and rounded down. This leads to the following formula:

$$\left\lfloor \frac{(w \times h)}{2} \right\rfloor ! \tag{3.2}$$

For 20×20 boards this results in 7.89×10^{374} states. For 15×15 boards it results in 1.97×10^{182} and does therefore not improve on the bound of 5.27×10^{158} given above.

According to Schadd (2011), the game-tree complexity of SameGame is 10^{85}, based on empirically determined values of $b = 20.7$ and $d = 64.4$. Similarly, we can approximate the game-tree complexity of the other game variants using b^d, determining b and d through simulations. In 100ms MCTS runs on 1000 randomly generated start positions, the average length and branching factor of the best solutions found was determined. The results were $b = 19.4$ and $d = 50.0$ for Bubble Breaker, and $b = 24.8$ and $d = 119.1$ for Clickomania. This leads to game-tree complexities of 2.46×10^{64} for Bubble Breaker, and 1.20×10^{166} for Clickomania.

3.2 Two-Player Domains

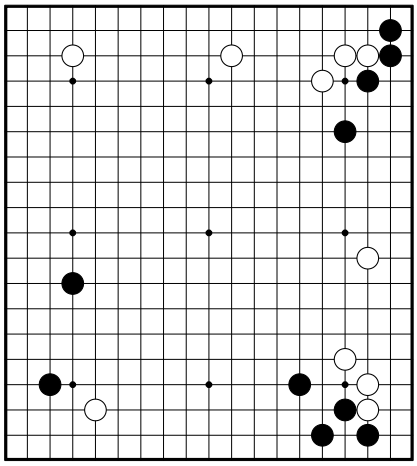
In this section, we introduce the two-player games used as test domains for this thesis. In addition to being discrete deterministic perfect-information turn-taking games, all of them are zero-sum games. They differ, however, in their *themes*, i.e. in the goals for their players. Go and Othello are territory games, Connect-4 is a connection game, Breakthrough is a race game, and Catch the Lion is a capture game.

3.2.1 Go

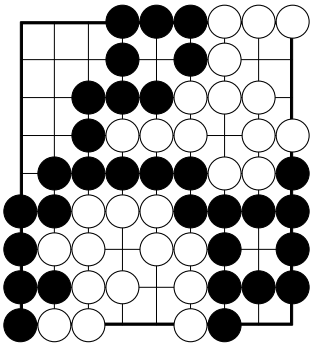
Go is an ancient board game originating from China thousands of years ago. It is believed to be played by 25–100 million people. China, Japan, and Korea have associations for professional, full-time Go players.

Go is played on a grid board of typically 19×19 intersections, although smaller board sizes of 9×9 and 13×13 intersections are popular for quicker, informal games and for educational purposes. Starting with an empty board, two players alternatingly place white and black stones on an empty intersection of their choice (Figure 3.2(a) shows a possible beginning of a game). If a player can surround any enemy stones completely with her own, the surrounded stones are removed ("captured"). Passing is allowed and typical if a player considers the outcome of the match sufficiently clear. At the end of the game—after both players have passed—the player who occupies or surrounds more intersections on the board ("territory") than her opponent wins (Figure 3.2(b) shows a possible final position).

There are several variations of the precise rules of Go in existence and in use. The differences between these do not change the outcome of a game in the vast majority of cases, but can occasionally have some influence on playing strategy. The Chinese rules are assumed in this thesis.



(a) The first moves of a 13×13 game.



(b) The end of a 9×9 game.

Figure 3.2: The game of Go.

The game of Go is the most well-known success story of MCTS. Due to the introduction of the technique in 2006, the strength of computer Go programs has been increased considerably (Lee et al., 2009). However, the world's best human Go

players are still superior to the best computer programs, and writing a master strength Go program stands as a grand challenge of AI (Cai and Wunsch, II, 2007; Müller, 2002). In order to measure the progress made regarding this challenge, competitions between high-ranking human professional players and strong Go programs are organized regularly. In the last two *Densei-sen* competitions[1], organized in 2013 and 2014 by the University of Electro-Communications Tokyo and the Nihon Ki-in (Japanese Go Association), the professionals Yoshio Ishida 9p and Norimoto Yoda 9p both achieved a result of one win and one loss in these human-computer matches[2]. Their computer opponents however still used four handicap stones, i.e. they were allowed to begin the game with four stones of their color already on the board to compensate for the difference in playing strength. This illustrates both the progress of computer Go in the past years as well as the challenge still remaining until computers are on a par with human professionals.

According to Tromp and Farnebäck (2007), the state-space complexity of 13×13 Go is 3.72×10^{79}, and the complexity of 19×19 Go is approximately 2.08×10^{170}. The game-tree complexity can be approximated by determining the average branching factor b and game length d of test games. Note that many computer programs play a game of Go until the very end, when only one-intersection territories remain and the determination of the winner is trivial. Human players tend to resign much earlier in the game, as soon as both players agree on the owner of larger territories and the overall winner of the game. In Orego (Drake et al., 2011), the Go program used in this thesis, such behavior can be simulated by resigning as soon as the win rate at the root of the search tree falls below a given threshold (set to 0.1 per default). Both for programs and human players, the average game length (and thus also the branching factor) therefore strongly depends on the player strength—the stronger the players, the sooner they will agree on the outcome of a game on average. Allis (1994) estimates $b = 250$ and $d = 150$ for 19×19 Go, indicating a game-tree complexity of approximately 10^{360}. Based on these numbers and other estimates for human play, a realistic estimate for 13×13 Go is $b = 120$ and $d = 80$, leading to $b^d \approx 2.2 \times 10^{166}$. In 1000 games of self-play at 1 second per move with Orego, we determined $b = 90$ and $d = 150$ ($b^d = 1.37 \times 10^{293}$). This shows the large differences due to the more conservative behavior of computer programs when it comes to resigning.

3.2.2 Connect-4

Connect-4 was developed by Howard Wexler and published by Milton Bradley under the name *Connect Four* in 1974, although its concept is claimed to be centuries old.

[1] See http://entcog.c.ooco.jp/entcog/densei/eng/index.html

[2] The abbreviation "9p" stands for "9 dan", the highest possible ranking for a professional Go player.

In this thesis, the standard variant played on a 7×6 board is used.

At the start of the game, the board is empty. The two players alternatingly place white and black discs in one of the seven columns, always filling the lowest available space of the chosen column. Columns with six discs are full and cannot be played anymore. The game is won by the player who succeeds first at connecting four tokens of her own color either vertically, horizontally, or diagonally. A possible winning position for White is shown in Figure 3.3. If the board is filled completely without any player reaching this goal, the game ends in a draw.

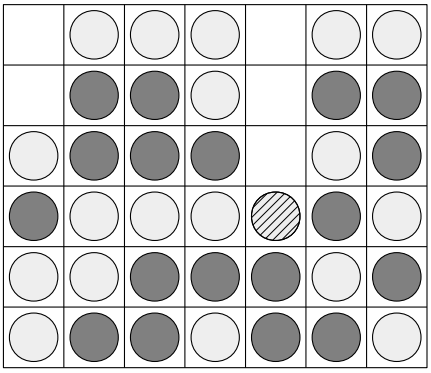


Figure 3.3: Connect-4. White won the game by playing the marked move.

The game of Connect-4 has been weakly solved by Allis (1988) and strongly solved by Tromp (2008). It is a win for the first player (White). Connect-4 has also been used in the MCTS framework in the context of General Game Playing (Finnsson and Björnsson, 2008; Kirci et al., 2009; Sharma et al., 2008), and for solving positions (Cazenave and Saffidine, 2011). Connect-4 was chosen as a test domain for this thesis due to its simple rules and bounded game length, while still providing a search space of non-trivial size and complexity.

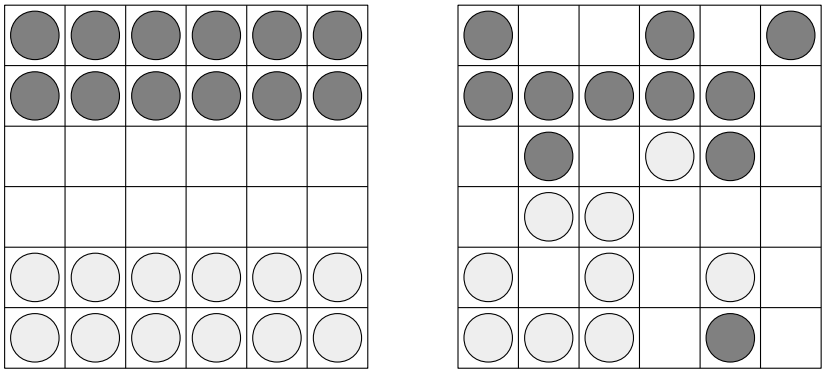
The state-space complexity of Connect-4 is 4.53×10^{12} (Edelkamp and Kissmann, 2008). Allis (1994) estimates $b = 4$ and $d = 36$, indicating a game-tree complexity of approximately 10^{21}.

3.2.3 Breakthrough

Breakthrough was invented by Dan Troyka in 2000 for a game design competition. The variant of Breakthrough used in this thesis is played on a 6×6 board. The game was originally described as being played on a 7×7 board, but other sizes such as 8×8 are popular as well, and the 6×6 board preserves an interesting search space.

At the beginning of the game, the first two rows of the board are occupied by

twelve white pieces, and the last two rows are occupied by twelve black pieces (see Figure 3.4(a)). The two players alternatingly move one of their pieces straight or diagonally forward, onto an empty square of the board. Two pieces cannot occupy the same square. However, players can capture the opponent's pieces by moving onto their square in diagonal direction only. The game is won by the player who succeeds first at reaching the home row of her opponent, i.e. reaching the first row as Black or reaching the last row as White, with one piece (see Figure 3.4(b)).



(a) The start position.

(b) A possible terminal position. Black won by advancing one piece to White's home row.

Figure 3.4: Breakthrough.

The application of MCTS to (8×8) Breakthrough has been investigated by Lorentz and Horey (2014). Breakthrough is also a popular domain in the General Game Playing community (Finnsson and Björnsson, 2008; Gudmundsson and Björnsson, 2013; Kirci et al., 2009; Sharma et al., 2008; Tak et al., 2012). The game has been solved on the smaller 6×5 board—it is a second player win (Saffidine et al., 2012).

An upper bound for the state-space complexity of Breakthrough can be estimated as follows. A simple approach is based on the fact that every square of the board can be either empty or contain either a white or black piece, which leads to an estimate of $3^{36} = 1.5 \times 10^{17}$ for the 6×6 board. A more refined estimate is based on the idea that either player can have between 0 and 12 pieces on the board in any legal position. Ignoring the fact that no more than 1 piece can ever be on the opponent's home row, the number of different distributions of x pieces on 36 squares is $\binom{36}{x}$, and the number of different distributions of y opponent pieces is $\binom{36-x}{y}$ for each of these. This leads to the following upper bound for the number of legal states:

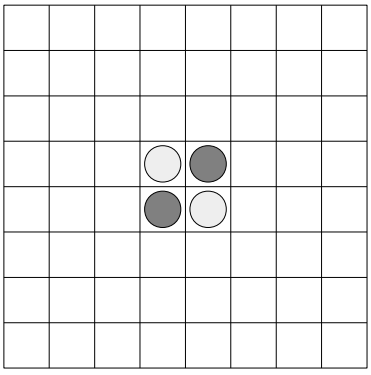
$$\sum_{x=0}^{12}\left(\binom{36}{x}\sum_{y=0,(x,y)\neq(0,0)}^{12}\binom{36-x}{y}\right)\approx 3.78\times 10^{16} \tag{3.3}$$

The game-tree complexity was estimated by sampling. In 1000 self-play games, we found $b = 15.5$ and 29, indicating a game-tree complexity of 3.31×10^{34}.

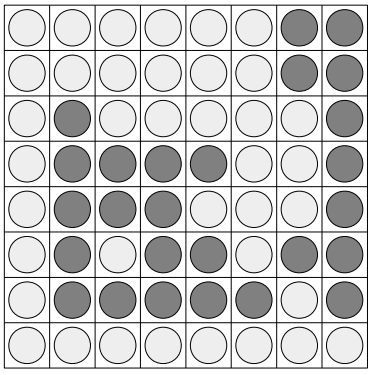
3.2.4 Othello

Othello is closely related to the game *Reversi*, invented by either Lewis Waterman or John W. Mollett in 1883. Othello's modern rules were developed in Japan in the 1970s. The game is played on an 8×8 board.

The game starts with four discs on the board, as shown in Figure 3.5(a). Each disc has a black side and a white side, with the side facing up indicating the player the disc currently belongs to. The two players alternatingly place a disc on the board, in such a way that between the newly placed disc and another disc of the moving player there is an uninterrupted horizontal, vertical or diagonal line of one or more discs of the opponent. All these discs are then turned over, changing their color to the moving player's side, and the turn goes to the other player. If there is no legal move for a player, she has to pass. If both players have to pass or if the board is filled, the game ends. The game is won by the player who owns the most discs at the end (see Figure 3.5(b)).



(a) The start position.



(b) A possible terminal position. White won by owning 38 of the 64 discs on the final board.

Figure 3.5: Othello.

The game of Othello has been the subject of research in the minimax framework (Rosenbloom, 1982; Buro, 2000), but is also used as a test domain for MCTS in General Game Playing (Finnsson and Björnsson, 2008; Tak et al., 2012). Its three- and four-player variant *Rolit* has been used for investigating multi-player search algorithms (Schadd and Winands, 2011; Nijssen and Winands, 2013).

Allis (1994) estimates the state-space complexity of Othello to be roughly 10^{28}, and the game-tree complexity to be roughly 10^{58}.

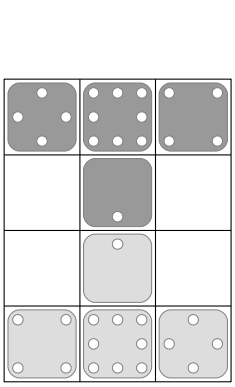
3.2.5 Catch the Lion

Catch the Lion—or *doubutsu shogi*, "animal Chess" in Japanese—was developed by professional Shogi (Japanese Chess) players Madoka Kitao and Maiko Fujita in 2008 in order to attract children to Shogi. It attempts to reflect all essential characteristics of Shogi in the simplest possible form (see Sato et al. (2010) for an MCTS approach to the full game of Shogi). *Catch the Lion* is played on a 3×4 board.

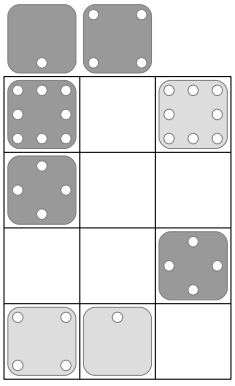
At the beginning of the game, each player has four pieces: a Lion, a Giraffe, an Elephant, and a Chick. The pieces are marked with the directions in which they can move—the Chick can move one square forward, the Giraffe can move one square in the vertical and horizontal directions, the Elephant can move one square in the diagonal directions, and the Lion can move one square in any direction (see Figure 3.6(a)). In the commercial version, the pieces are additionally marked with animal pictures. During the game, the players alternatingly move one of their pieces. Pieces of the opponent can be captured. As in Shogi, they are removed from the board, but not from the game. Instead, they switch sides, and the player who captured them can later on drop them on any square of the board instead of moving one of her pieces. If the Chick reaches the home row of the opponent, it is promoted to a Chicken, now being able to move one square in any direction except for diagonally backwards. A captured Chicken, however, is demoted to a Chick again when dropped. The game is won by either capturing the opponent's Lion, or moving your own Lion to the home row of the opponent (see Figure 3.6(b)).

The game of Catch the Lion has been strongly solved (Tanaka, 2009). It is a win for the first player (White). Catch the Lion is used in this thesis because it represents a simple instance of Chess-like games, which tend to be particularly difficult for MCTS (Ramanujan et al., 2010a).

The state-space complexity of Catch the Lion is 1.57×10^9 (Tanaka, 2009). From 1000 self-play experiments, we estimated $b = 10$ and $d = 35$, which indicates a game-tree complexity of 10^{35}. The game length depends strongly on the player.



(a) The start position.



(b) A possible terminal position. White won by advancing her Lion to Black's home row. Black has two prisoners, a Chick and an Elephant.

Figure 3.6: Catch the Lion.

Part I

MCTS in One-Player Domains

4

Nested Monte-Carlo Tree Search

This chapter is based on:

Baier, H. and Winands, M. H. M. (2012). Nested Monte-Carlo Tree Search for Online Planning in Large MDPs. In L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, editors, *20th European Conference on Artificial Intelligence, ECAI 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 109–114.

Baier, H. and Winands, M. H. M. (2012). Nested Monte-Carlo Tree Search for Online Planning in Large MDPs. In J. W. H. M. Uiterwijk, N. Roos, and M. H. M. Winands, editors, *24th Benelux Conference on Artificial Intelligence, BNAIC 2012*, pages 273–274. Extended abstract.

In MCTS, every state in the search tree is evaluated by the average outcome of Monte-Carlo rollouts from that state. For the consistency of MCTS, i.e. for the convergence to the optimal policy, uniformly random rollouts beyond the tree are sufficient. However, stronger rollout strategies typically greatly speed up convergence. The strength of Monte-Carlo rollouts can be improved for example by hand-coded heuristics (Gelly et al., 2006). A more principled approach is the automated tuning of rollout policies through supervised learning (Coulom, 2007a) or reinforcement learning (Bouzy and Chaslot, 2006; Silver and Tesauro, 2009). In recent years, the topic of online learning of rollout policies has received more and more attention, i.e. improving the rollout policy while the search is running (Finnsson and Björnsson, 2008; Baier and Drake, 2010; Rimmel and Teytaud, 2010; Tak et al., 2012).

In this chapter, we answer the first research question by proposing *Nested Monte-Carlo Tree Search* (NMCTS), using the results of lower-level searches recursively to provide rollout policies for searches on higher levels. Instead of improving a given

set of rollout policy parameters either offline or online, we replace calls to the rollout policy with calls to MCTS itself. We compare the performance of NMCTS to that of regular (multi-start) MCTS as well as Nested Monte-Carlo Search (NMCS), at equal time controls, in the deterministic one-player domains SameGame, Clickomania, and Bubble Breaker.

The structure of this chapter is as follows. Section 4.1 provides an overview of related work on nested or meta-search in a Monte-Carlo framework, and presents the competing approach of NMCS. Section 4.2 proposes NMCTS as a generalization of NMCS, and Section 4.3 shows experimental results in three test domains. Conclusions and future research follow in Section 4.4.

4.1 Background

This section describes related work on nested search or meta-search. Furthermore, Nested-Monte Carlo Search (NMCS) is introduced as the main competing approach to which we are comparing NMCTS in Section 4.3.

4.1.1 Related Work

Tesauro and Galperin (1997) were the first to use Monte-Carlo rollouts for improving an agent's policy online. For each possible move (action) m in the current position (state) of the agent, they generated several rollouts starting with m and then following the policy as given by a "base controller" (an arbitrary heuristic). After estimating the expected reward of each move by averaging rollout results, they improved the heuristic by choosing and executing the move with the best estimated value. This resembles one cycle of policy iteration, focused on the current state.

Yan et al. (2004) introduced the idea of online improvement of a base policy through *nested* search. The first level of nesting corresponds to a rollout policy as proposed in Tesauro and Galperin (1997), estimating the value of each move by starting with this move and then following the base policy. The second level estimates the value of each move by starting with this move and then executing a first-level search; higher levels are defined analogously. Bjarnason et al. (2007) improved this approach for Solitaire by using different heuristics and nesting levels for every phase of the game.

Cazenave (2007, 2009) proposed similar search methods to Yan's *iterated rollouts* under the names of *Reflexive Monte-Carlo Search* (RMCS) and *Nested Monte-Carlo Search* (NMCS). The main difference to preceding approaches is that RMCS and NMCS assume a uniformly random base policy instead of an informed search heuristic, and the best sequence found so far is kept in memory. NMCS has since been applied to a variety of problems, such as bus network regulation (Cazenave et al., 2009),

expression discovery (Cazenave, 2010), the snake-in-the-box problem (Kinny, 2012), and General Game Playing (Méhat and Cazenave, 2010). It has been improved for certain types of domains by adding the AMAF technique (Akiyama et al., 2010) and by re-introducing and optimizing base search heuristics (Rimmel et al., 2011). We describe NMCS in detail in the next subsection.

Rosin (2011) developed *Nested Rollout Policy Adaptation* (NRPA), a variant of NMCS that adapts the rollout policy during search using gradient ascent. At each level of the nested search, NRPA shifts the rollout policy towards the best solution found so far, instead of advancing towards this solution directly on the search tree. The algorithm depends on a domain-specific representation of moves that allows for the generalization of move values across different positions. Variants of NRPA have been applied to logistics problems (Edelkamp et al., 2013; Edelkamp and Gath, 2014).

Outside of the Monte-Carlo framework, the concept of nested searches has been applied to Proof-Number Search (Allis et al., 1994). The PN^2 algorithm (Allis, 1994) uses a nested, lower-level PN search at the leaves of the original, higher-level PN search, improving the performance of PNS in solving games and endgame positions of games (Breuker et al., 2001). The main improvement of PN^2 over PNS is a strong reduction of its memory requirements, an advantage that also applies to the nested MCTS approach proposed in this chapter.

In the context of MCTS, nested search has so far only been used for the preparation of opening books for the deterministic two-player game of Go (Audouard et al., 2009; Chaslot et al., 2009; Chou et al., 2012). In these applications, nested search was performed offline to provide opening databases for the underlying online game playing agent. The different levels of search therefore used different tree search algorithms adapted to their respective purpose, and nested and regular MCTS have not been compared on the same task.

So far, no nested search algorithm has made use of the selectivity and exploration-exploitation control that MCTS provides. In this chapter, we propose Nested Monte-Carlo Tree Search (NMCTS) as a general online planning algorithm. We expect it to outperform MCTS in a similar way to how NMCS outperforms naive Monte-Carlo search—through nesting. Furthermore, we expect it to outperform NMCS in a similar way to how MCTS outperforms naive Monte-Carlo search—through selective tree search.

4.1.2 Nested Monte-Carlo Search

NMCS (Cazenave, 2009) is a popular approach to nested search in a one-player Monte-Carlo framework. Algorithm 4.1 shows pseudocode for it. NMCS chooses and executes actions step by step until a terminal position is reached (lines 4–32). It then returns

the best score found together with the simulation that lead to this score (line 33). The way actions are chosen in each step depends on the level of the search. At level 1, NMCS conducts one uniformly random rollout to complete one simulation for each legal action (lines 12–16). At level n for $n \geq 2$, NMCS recursively conducts one level-$(n-1)$ NMCS run for each legal action (line 18), returning the best simulation found in that run. At all levels, NMCS keeps track of the globally highest-scoring simulation found so far (lines 26–29). This simulation can be a result of the current search step (lines 20–23), as well as any previous search step at equal or lower level. NMCS then chooses the next action of the simulation with the globally highest score so far for execution (lines 30–31).

NMCS does not make use of a tree or similar data structure as MCTS does, and can therefore not balance exploration and exploitation through continuously improved value estimates. In the next section, NMCS is characterized as a special case of the newly proposed NMCTS algorithm. In Subsection 4.3.3, the two approaches are compared experimentally.

4.2 Nested Monte-Carlo Tree Search

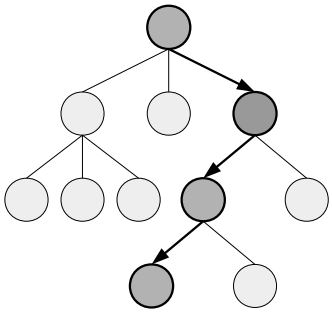
We define a level-0 *Nested Monte-Carlo Tree Search* (NMCTS) as a single rollout with the base rollout policy—either uniformly random, or guided by a simple heuristic. Level-1 NMCTS corresponds to MCTS, employing level-0 searches as position evaluations. A level-n NMCTS run for $n \geq 2$ recursively utilizes the results of level-$(n-1)$ searches as evaluation returns. Level-2 NMCTS is illustrated in Figure 4.1. The only difference to the illustration of regular MCTS in Figure 2.3 is the rollout phase, where a level-1 NMCTS run (an MCTS run) replaces the rollout.

Algorithm 4.2 shows pseudocode of NMCTS for deterministic domains. There are only three differences between the NMCTS pseudocode in Algorithm 4.2 and the pseudocode of regular MCTS in Algorithm 2.6. First, NMCTS is called on the highest nesting level with the desired number of search levels and an empty simulation as an additional argument (see code line 1). Second, a regular MCTS rollout with the base rollout policy is only performed in the rollout phase on level 1 (see lines 12-18). On higher levels, a lower-level NMCTS run is called as replacement for the rollout phase (see lines 19-20). Third, the number of iterations is defined separately for each level (see line 4). Finding the most effective trade-off between the numbers of iterations at each level is subject to empirical optimization.

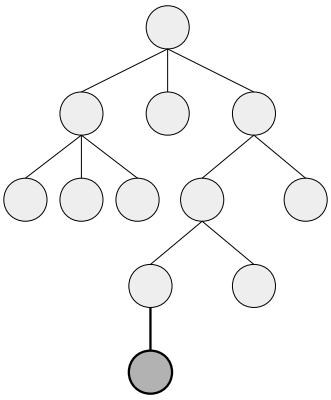
As the selection, expansion and backpropagation phases of MCTS are preserved in NMCTS, many successful techniques from MCTS research such as the UCB1-TUNED selection policy can be applied in NMCTS as well. Parameters can be tuned for each level of search independently.

```
NMCS(startState, level) {
   globalBestScore ← -Infinity
   globalBestSimulation ← {}
   while(startState.notTerminalPosition) {
      currentBestScore ← -Infinity
      currentBestSimulation ← {}
      currentState ← startState
      for(numberOfLegalActions(currentState)) {
         untriedAction ← findUntriedAction(currentState)
         currentState ← takeAction(untriedAction, currentState)
         if(level = 1) {
            while(simulationNotEnded) {
               currentState ← takeRandomAction(currentState)
               simulation ← simulation + currentState
            }
            score ← cumulativeReward(simulation)
         } else {
            (score, simulation) ← NMCS(currentState, level-1)
         }
         if(score > currentBestScore) {
            currentBestScore ← score
            currentBestSimulation ← simulation
         }
         currentState ← startState
      }
      if(currentBestScore > globalBestScore) {
         globalBestScore ← currentBestScore
         globalBestSimulation ← currentBestSimulation
      }
      startState ← takeAction(globalBestSimulation.nextAction,
                                                startState)
   }
   return (globalBestScore, globalBestSimulation)
}
```

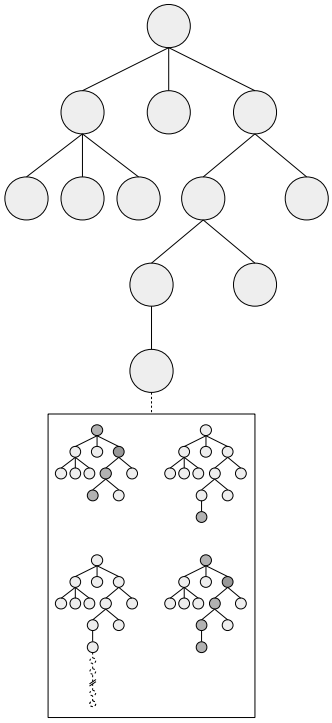
Algorithm 4.1: NMCS.



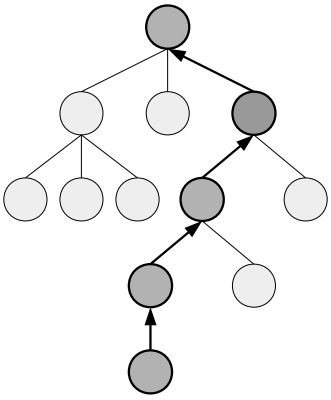
(a) The selection phase. The selection policy is applied recursively until an unsampled action is reached.



(b) The expansion phase. The newly sampled action is executed and the resulting state is added to the tree.



(c) The "rollout" phase. Instead of one rollout, an entire level-1 NMCTS run (i.e. an MCTS run) is conducted.



(d) The backpropagation phase. The best solution found by the level-1 search is returned to level 2 and backpropagated.

Figure 4.1: NMCTS.

```
1  NMCTS(startState, simulation, level) {
2     bestScore ← -Infinity
3     bestSimulation ← {}
4     for(numberOfIterationsForLevel(level)) {
5        currentState ← startState
6        while(currentState ∈ Tree) {
7           currentState ← takeSelectionPolicyAction(currentState)
8           simulation ← simulation + currentState
9        }
10       addToTree(currentState)
11       if(level = 1) {
12          while(currentState.notTerminalPosition) {
13             currentState ← takeRolloutPolicyAction(currentState)
14             simulation ← simulation + currentState
15          }
16          score ← cumulativeReward(simulation)
17       } else {
18          (score,simulation) ←
19             NMCTS(currentState, simulation, level-1)
20       }
21       forall(state ∈ {simulation ∩ Tree}) {
22          state.value ← backPropagate(state.value, result)
23       }
24       if(score > bestScore) {
25          bestScore ← score
26          bestSimulation ← simulation
27       }
28    }
29    return (bestScore, bestSimulation)
30 }
```

Algorithm 4.2: NMCTS.

Note that the tree nodes used by a lower-level search do not have to be kept in memory after that search has finished. NMCTS can free them for future lower-level searches. Moreover, a higher-level tree is typically much smaller when using the more time-consuming nested searches than when using the base rollout policy. These factors together result in NMCTS using far less memory than a global MCTS run with the same time limits, giving NMCTS an additional advantage over MCTS similar to the advantage of PN^2 over PNS (Breuker, 1998).

In Schadd et al. (2012), it was found to be effective in SameGame not to spend the entire search time on the initial position of a game, but to distribute it over all moves (or the first z moves). We call this technique *move-by-move search* as opposed to *global search*, and it is applicable at all levels of NMCTS, distributing time over z_i moves on level i. If it is used on level 1 for example, the rollouts of the level-2 search are not replaced by one MCTS search anymore, but by several MCTS searches that are performed in sequence. If it is used on level 2, the higher-level search itself is split into a sequence of several MCTS searches. In case move-by-move search is used, a decision has to be made which move to choose and execute between two such searches. Two possible options are a) choosing the most-sampled next move (as traditionally done in MCTS), or b) choosing the next move in the overall best solution found so far.

NMCTS is a generalization of MCTS, which is equivalent to level-1 NMCTS. Furthermore, NMCTS can be seen as a generalization of NMCS. NMCTS behaves like NMCS if move-by-move search is applied at all levels, only one rollout per legal move is used in each move search, and the next move of the best known solution is chosen for execution after each move search. This special case of NMCTS does not provide for an exploration-exploitation tradeoff, nor does it build a tree going deeper than the number of nesting levels used, but it can allow relatively deep nesting due to the low number of rollouts per search level.

4.3 Experimental Results

We have tested NMCTS on three different deterministic, fully observable domains: the puzzles SameGame, Clickomania and Bubble Breaker. A random rollout policy was used in all three domains. For SameGame, we additionally employed an informed rollout policy. It consists of the TabuColorRandomPolicy (Schadd et al., 2008b; Cazenave, 2009), setting a "tabu color" at the start of each rollout which is not chosen as long as groups of other colors are available. We improved this policy further by adding a multi-armed bandit (based on UCB1-TUNED) for globally learning the best-performing tabu color in each search. The speed of our engine was 9500 rollouts per second in SameGame with random rollouts, 9100 rollouts per second in SameGame with informed rollouts, 9900 rollouts per second in Bubble Breaker (the domain with

the shortest games on average, compare Section 3.1), and 8200 rollouts per second in Clickomania (the domain with the longest games on average). These numbers were averaged over 10-second runs from the start position of 10 randomly generated boards per domain.

For all domains, rollout outcomes were normalized to the interval $[0, 1]$. This was done by determining a lower bound l and upper bound u for the possible scores in each domain, and linearly mapping the resulting interval $[l, u]$ to $[0, 1]$. The upper bound for SameGame assumes that the entire board is filled with tiles of only one color in the start position. This allows to clear the board with a single move, leading to a score of

$$((15 \times 15) - 2)^2 + 1000 = 50749 \tag{4.1}$$

for a board size of 15×15. The lower bound for SameGame assumes that the starting board is filled with a checkerboard pattern of two colors such that no move is possible. This leads to a score of

$$-\left(\left\lfloor \frac{15 \times 15}{2} \right\rfloor - 2\right)^2 - \left(\left\lceil \frac{15 \times 15}{2} \right\rceil - 2\right)^2 = -24421 \tag{4.2}$$

In Bubble Breaker, the same starting boards lead to an upper bound of $(15 \times 15) \times (15 \times 15 - 1) = 50400$ for clearing the board in one move, and a lower bound of 0 for having no legal move. The upper bound for Clickomania is a score of 400 for clearing a 20×20 board, and the lower bound is a score of 0 for having no legal move.

In Subsection 4.3.1, we explain the tuning of the MCTS parameters. The optimal settings found are used in Subsection 4.3.2 to compare the performance of level-2 NMCTS and (multi-start) MCTS, and in Subsection 4.3.3 to compare the performance of level-2 NMCTS and NMCS.

4.3.1 Parameter Optimization

As mentioned in Section 4.2, the optimal numbers of samples at each level of NMCTS are subject to empirical optimization. A level-2 NMCTS run with a total time of 9120 seconds can for example consist of 152 level-1 (MCTS) searches of 60 seconds each, or 2280 level-1 (MCTS) searches of 4 seconds each, and so on. MCTS runs of different lengths can be used. In the first series of experiments, we therefore tuned the exploration factor C for regular MCTS runs of various time settings. For each time setting and domain, 10-20 values of C at different orders of magnitude between 0 and

0.1 were tested. Figures 4.2 to 4.5 present the performance of MCTS with different exploration factors at 15 s, 60 s, and 240 s (in Clickomania at 4 s, 16 s, and 64 s) in order to give an idea of the parameter landscapes in the four test domains. Tables 4.1 and 4.2 show the values of C found to perform optimally at all time settings. These values are used in Subsection 4.3.2. Additionally, we activated move-by-move search and tuned C as well as the number of moves z over which to distribute the total search time. For each time setting and domain, 6-12 values of z between 1 and 160 were tried. The optimal values found are shown in Tables 4.3 and 4.4. These values are used in Subsection 4.3.3. All tuning experiments employed independent sets of randomly generated training positions.

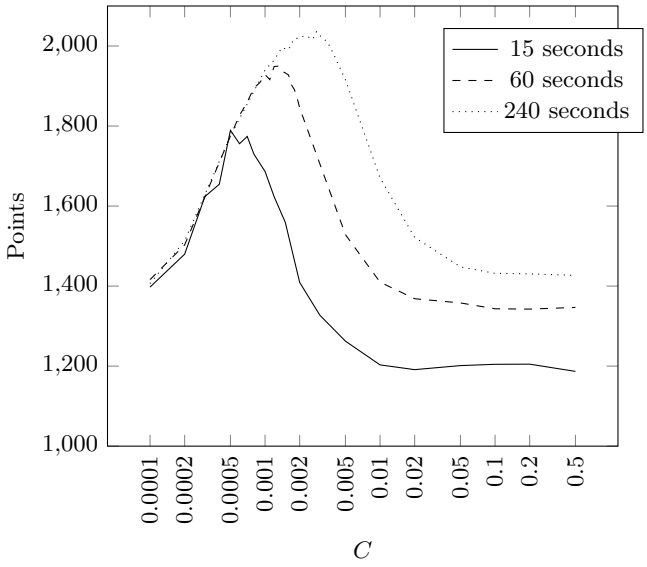


Figure 4.2: Performance of MCTS in SameGame with random rollouts.

A general observation is that the less search time MCTS has, the lower its exploration factor has to be in order to build a deep enough tree. If C is too high, the last moves of the game stay beyond the tree and may therefore remain suboptimal—chosen by the rollout policy instead of optimized by the selection policy of MCTS. A second observation is that the optimal exploration factors for the one-player domains are much smaller than for the two-player domains used in Part II of this thesis. One reason for this is that a player in a one-player game wants to find a good solution to the entire game, consisting of many moves, which potentially requires a very deep search. Missing some good moves in the next few plies is often less problematic on average than having a too shallow tree and not optimizing the end of the game at all.

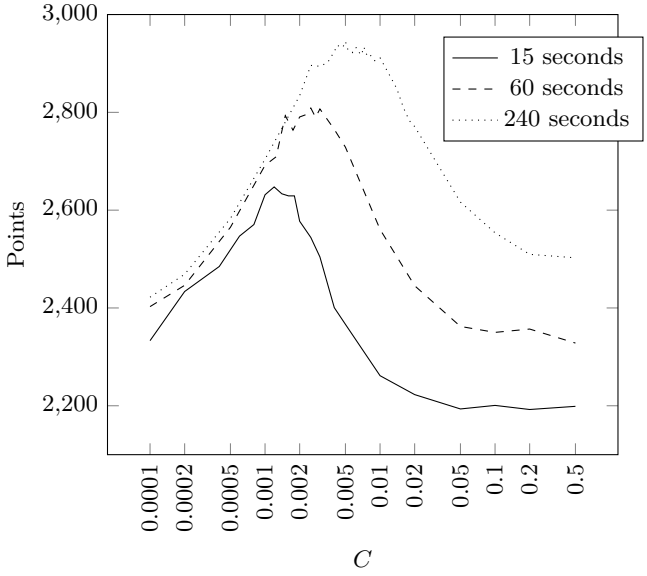


Figure 4.3: Performance of MCTS in SameGame with informed rollouts.

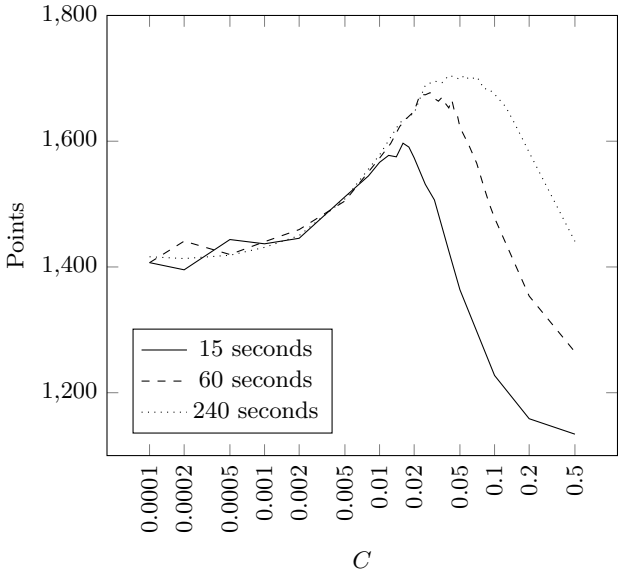


Figure 4.4: Performance of MCTS in Bubble Breaker.

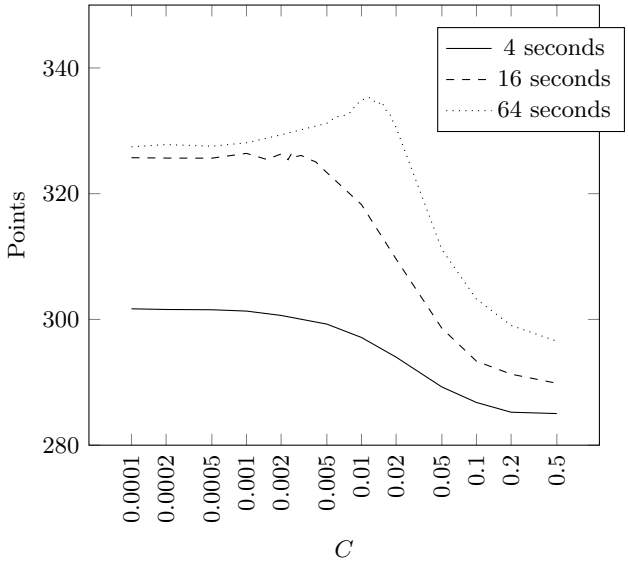


Figure 4.5: Performance of MCTS in Clickomania.

Table 4.1: Best-performing exploration factors C for MCTS in SameGame with random rollouts, SameGame with informed rollouts, and Bubble Breaker.

Time in s	Best-performing value of C in		
	SameGame random	SameGame informed	Bubble Breaker
0.25	0	0	0
1	0	0.0002	0.0004
4	0.0002	0.0005	0.008
15	0.0005	0.0012	0.016
60	0.0013	0.0025	0.0275
240	0.0028	0.005	0.055

Table 4.2: Best-performing exploration factors C for MCTS in Clickomania.

Time in s	C
0.016	0
0.05	0
0.25	0
1	0
4	0
16	0.0025
64	0.012
1280	0.032

Table 4.3: Best-performing exploration factors C and numbers of moves z for move-by-move MCTS in SameGame with random rollouts and SameGame with informed rollouts.

Time in s	SameGame random		SameGame informed	
	C	z	C	z
0.05	0.0001	50	0.0002	10
0.1	0.0001	50	0.0001	5
0.2	0.0001	50	0.00005	5
0.4	0.00005	40	0.00005	1
0.8	0.0001	40	0.00005	1
1.6	0.00003	30	0.0001	1
3.2	0.00001	30	0.0003	30
6.4	0.0001	25	0.0003	30
12.8	0.0002	25	0.0004	30

Table 4.4: Best-performing exploration factors C and numbers of moves z for move-by-move MCTS in Bubble Breaker and Clickomania.

Time in s	Bubble Breaker		Clickomania	
	C	z	C	z
0.05	0.0002	50	0	100
0.1	0.0002	50	0	100
0.2	0.0002	40	0.0001	100
0.4	0.0002	40	0.0001	100
0.8	0.0002	40	0.0001	100
1.6	0.0003	30	0.0005	100
3.2	0.0008	30	0.0005	100
6.4	0.004	30	0.001	100
12.8	0.02	30	0.003	100

A player in a two-player game however wants to find a good next move only. The future of the game depends on her adversary as well, so optimizing a possible game ending that might never be reached is often less useful on average than exploring more strongly and avoiding mistakes in the next few plies.

When move-by-move search is activated, we can observe two different trends. The first trend, observed in SameGame and Bubble Breaker, is that the optimal values for z are lower for longer total search times. The longer the total search time, the fewer moves counting from the start of the game it should be distributed over, resulting in a longer search time per move. The reason seems to be that with a relatively low z, more effort is spent on the optimization of the beginning of the game. This is crucial in the game variants where large groups have to be formed in order to achieve high scores. At the end of these games, after the high-scoring move or moves have been made, all further moves are less relevant to the total score on average. The longer time per move search resulting from a lower z is sufficient at long time settings for the last move search to grow its tree deep enough, and optimize the last moves of the game well enough. Clickomania seems to behave differently since the optimization of all moves from the beginning to the end of the game appears to be similarly important in this domain. The optimal z therefore stays constant with larger total search times.

The second trend is observed only in SameGame with informed rollouts. Here, the optimal value for z increases again with the longest total search times tested (3.2 to 12.8 seconds). This may be due to the TabuColorRandomPolicy being restarted in every move and requiring a certain time to learn the best "tabu color" effectively. On

average, this reduces the optimal values of z. Only with relatively long search times is it optimal to distribute the total time over many moves, as the TabuColorRandomPolicy then has enough time in each move search to find a suitable tabu color. It is unclear why this effect comes into play so suddenly when increasing search time from 1.6 seconds to 3.2 seconds, but the performance differences are rather small so noise could play a role.

4.3.2 Comparison to MCTS

As it has been shown for SameGame that restarting several short MCTS runs on the same problem can lead to better performance than a single, long run (Schadd et al., 2008b), we compared NMCTS against multi-start MCTS. The settings for C found to be optimal for MCTS in the previous subsection (see Tables 4.1 and 4.2) were used as level-1 exploration factor C_1 for NMCTS and as C for multi-start MCTS. NMCTS and multi-start MCTS had the same total time per position, and the number of nested level-1 NMCTS runs was equal to the number of restarts for multi-start MCTS. The exploration factor C_2 of level 2 was set to 0 in all NMCTS conditions. Since NMCTS did not use move-by-move search here, and restarting searches results in similar memory savings as nesting them, the results purely reflect the advantage of nesting the level-1 searches into a tree (NMCTS) instead of performing them sequentially (multi-start MCTS).

The experiments for Bubble Breaker and SameGame were conducted on the first 100 test positions used in Schadd et al. (2008b)[1]. These positions consist of 15×15 boards with randomly distributed tiles of 5 different colors. Algorithms were allocated 9120 seconds (about 2.5 hours) of computation time per position. The experiments on Clickomania were conducted using a test set of 100 randomly generated 20×20 boards with 10 different tile colors, to provide a greater challenge. For the same reason, each algorithm only ran for 1280 seconds per position in Clickomania.

Figures 4.6, 4.7, and 4.8 show the results for Bubble Breaker and SameGame with both random and informed rollouts. In these three domains, the effectiveness and behavior of multi-start MCTS confirms the findings of Schadd et al. (2008b). The TabuColorRandomPolicy also performed well compared to the random rollout policy in SameGame. Furthermore, we observe that level-2 NMCTS significantly outperformed multi-start MCTS in all experimental conditions ($p<0.001$ in a paired-samples, two-tailed t-test). The figures show both the performance of NMCTS and multi-start MCTS as well as the average difference in performance and the corresponding 95% confidence interval. The best results in SameGame for example were achieved building a level-2 tree out of 9120 level-1 searches of 1 second duration each, with informed

[1] Available online at http://project.dke.maastrichtuniversity/games/SameGame/TestSet.txt

base-level rollouts. In comparison to the best performance of multi-start MCTS, achieved with 2280 restarts of 4-second searches, the use of a nested tree increased the average best solution per position from 3409.1 to 3487.7. As a comparison, a doubling of the multi-start MCTS search time to 4560 restarts only resulted in an increase to 3432.5. The best results in SameGame with random rollouts were achieved with 9120 level-1 searches of 1 s each, increasing the average best solution per position from 2650.1 to 2861.7 compared to multi-start MCTS. In Bubble Breaker, 9120 level-1 searches of 1 second duration increase the average best outcome from 2335.0 to 2661.7. Also note how the advantage of NMCTS over multi-start MCTS is the largest with short level-1 searches, especially in SameGame. Longer and therefore fewer level-1 searches do not seem to allow for large enough level-2 trees, so that most level-1 searches are started relatively close to the root position where they are also started in multi-start MCTS. In SameGame and Bubble Breaker, the numbers of restarts/level-1 searches tested ranged between 38 and 36480, i.e. the time per restart/level-1 search ranged between 240 seconds and 250 milliseconds. Longer and shorter searches were far off the observed maximum performance; additionally, longer searches resulted in memory problems, and shorter searches resulted in problems with precise timing.

In Clickomania, level-2 NMCTS also achieved the highest score (see Figure 4.9). 25600 level-1 searches of 50 ms each score an average of 360.3 for NMCTS, while the best result for multi-start MCTS is 351.2. This difference is statistically significant ($p<0.0001$). Because the observed performance curve was not concave in Clickomania, we extended the tested range to 1280 seconds per restart/level-1 search on the longer end and to 16 milliseconds per restart/level-1 search on the lower end. The results for long durations of restarts/level-1 searches suggest that a single, global MCTS run could perform best in Clickomania—but memory limitations reduced the effectivity of this approach, leading to the dip in performance visible at 1 restart in Figure 4.9. NMCTS however is able to make better use of many short searches due to nesting them into a tree instead of conducting them sequentially. This is probably why the performance of NMCTS, other than that of multi-start MCTS, increases again for high numbers of restarts/level-1 searches. We observed that the best-performing NMCTS setting tested used less than 15% memory of what a single, global MCTS run would have required for optimal performance.

4.3.3 Comparison to NMCS

The last series of experiments was concerned with a comparison of level-2 NMCTS to NMCS. Here, NMCTS was tuned using move-by-move search on both level 1 and 2, and advancing from move to move by choosing the next move of the best solution found so far—with the exception of Clickomania, where move-by-move search did not

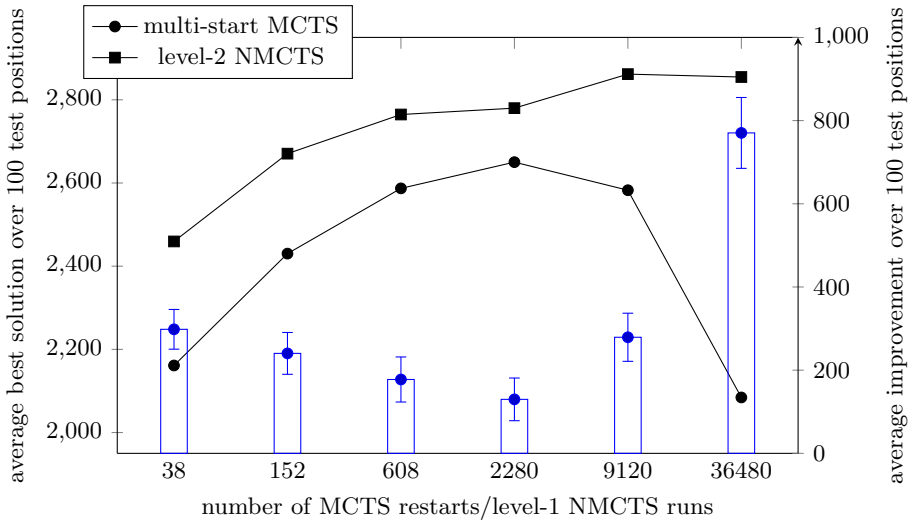


Figure 4.6: Performance of level-2 NMCTS in SameGame with random rollout policy. Bars show the average performance increase over multi-start MCTS with a 95% confidence interval. The search time was 9120 seconds per position.

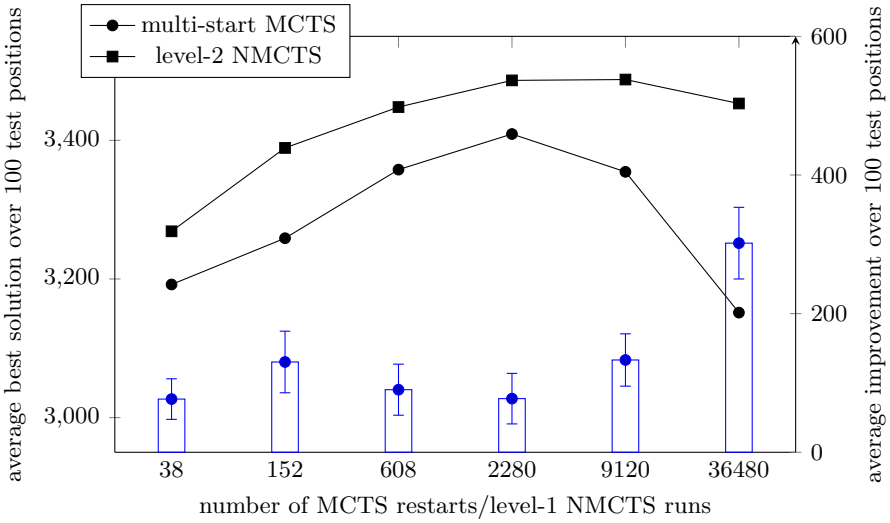


Figure 4.7: Performance of level-2 NMCTS in SameGame with informed rollout policy. Bars show the average performance increase over multi-start MCTS with a 95% confidence interval. The search time was 9120 seconds per position.

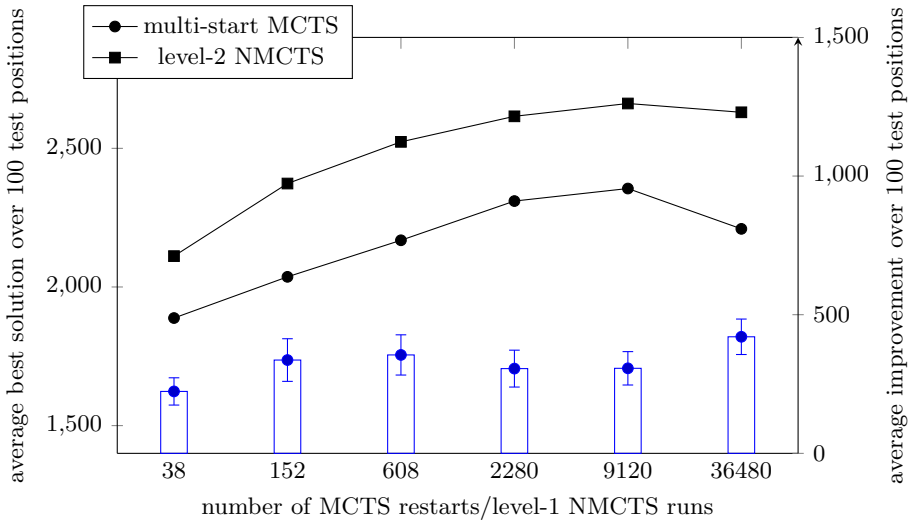


Figure 4.8: Performance of level-2 NMCTS in Bubble Breaker. Bars show the average performance increase over multi-start MCTS with a 95% confidence interval. The search time was 9120 seconds per position.

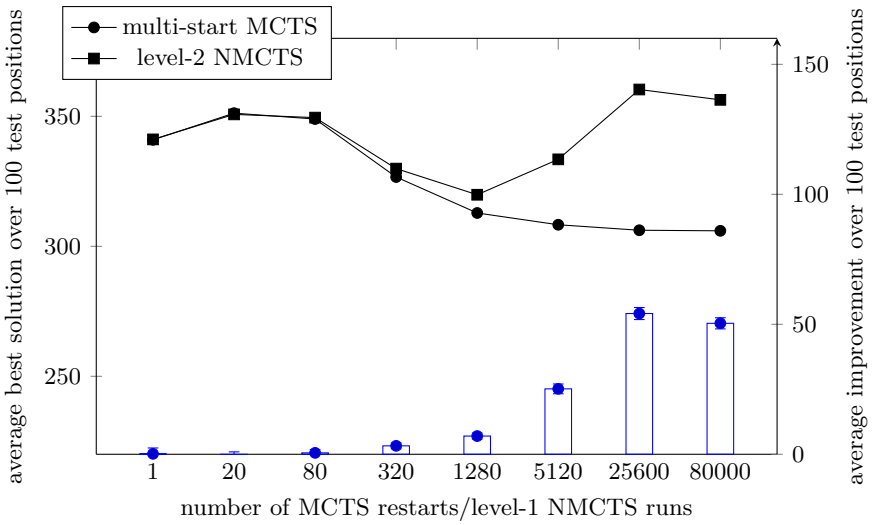


Figure 4.9: Performance of level-2 NMCTS in Clickomania. Bars show the average performance increase over multi-start MCTS with a 95% confidence interval. The search time was 1280 seconds per position.

improve performance in our experiments. The settings for C_1 and z_1 of NMCTS were taken from Tables 4.3 and 4.4. Only in Bubble Breaker a lower z_1 (20 instead of 40) was more effective, possibly because this setting wastes less time when the level-2 tree is already relatively deep and level 1 cannot add another 40 plies anymore. z_2 was individually tuned for each domain, and C_2 was 0 throughout. Games were played on the same test positions and with the same total time as in Subsection 4.3.2, i.e. 9120 seconds in SameGame and Bubble Breaker, and 1280 seconds in Clickomania. NMCS was not able to complete a level-3 search in the given time; consequently, the best solutions found when time ran out were used for the comparisons. Level-2 NMCS however was repeated several times until time ran out, with the best values of all repetitions used for the comparisons.

Figures 4.10 to 4.13 include both the average results of the three algorithms as well as the average performance increase of NMCTS over the best-performing NMCS version, and the corresponding 95% confidence interval. They also show the best performance of NMCTS without move-by-move search in order to demonstrate the effect of this option, except for Clickomania where move-by-move in ineffective. These performance data are taken from Section 4.3.2.

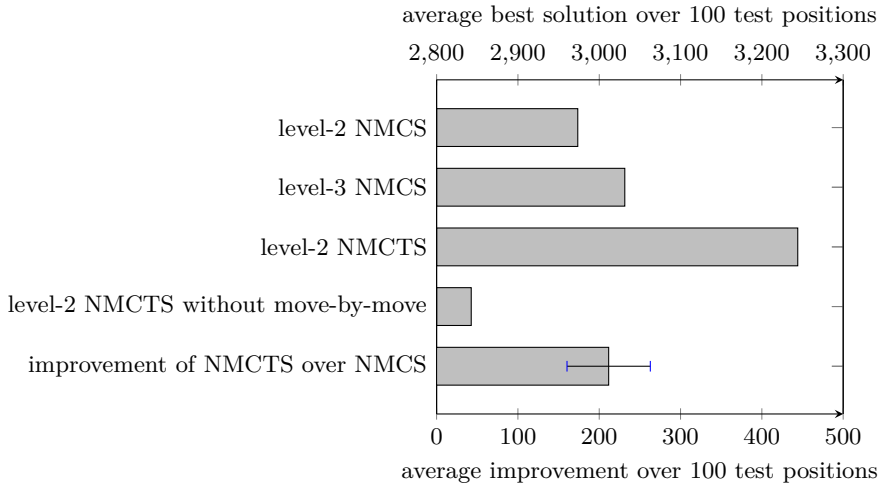


Figure 4.10: Performance of NMCS and level-2 NMCTS in SameGame with random rollout policy. NMCTS employs 142 level-1 searches, each 1600 ms long, for each of the first $z_2 = 40$ moves of a game. The 1600 ms are distributed over $z_1 = 30$ moves. $C_1 = 0.00003$.

In conclusion, NMCTS outperforms NMCS in SameGame with random rollouts ($p<0.0001$), SameGame with informed rollouts ($p<0.0001$), Bubble Breaker ($p<0.05$), and Clickomania ($p<0.05$).

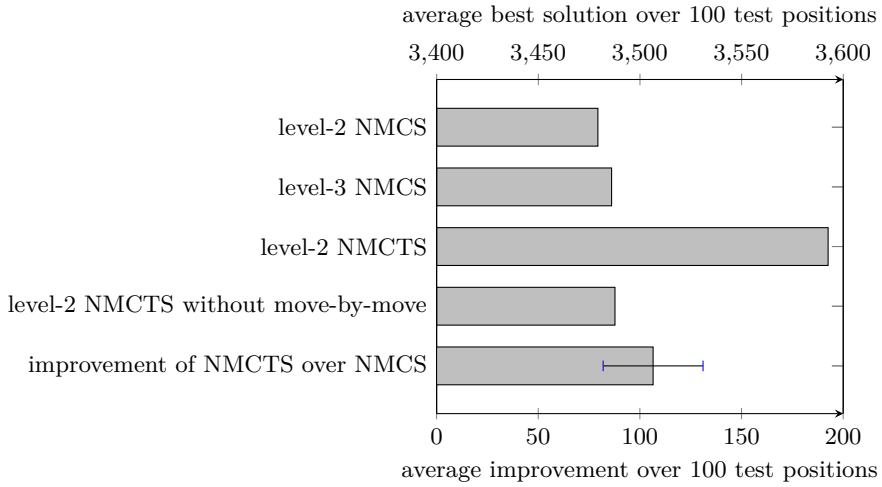


Figure 4.11: Performance of NMCS and level-2 NMCTS in SameGame with informed rollout policy. NMCTS employs 285 level-1 searches, each 3200 ms long, for each of the first $z_2 = 10$ moves of a game. The 3200 ms are distributed over $z_1 = 30$ moves. $C_1 = 0.0003$.

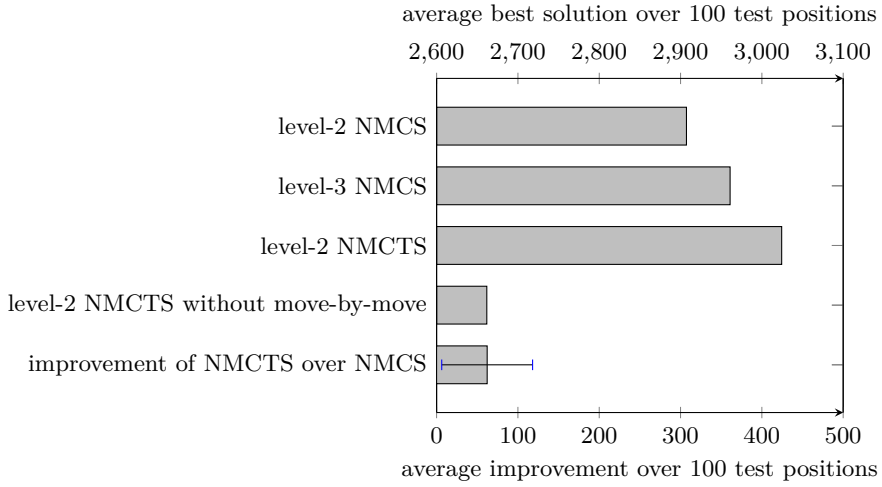


Figure 4.12: Performance of NMCS and level-2 NMCTS in Bubble Breaker. NMCTS employs 570 level-1 searches, each 400 ms long, for each of the first $z_2 = 40$ moves of a game. The 400 ms are distributed over $z_1 = 20$ moves. $C_1 = 0.0002$.

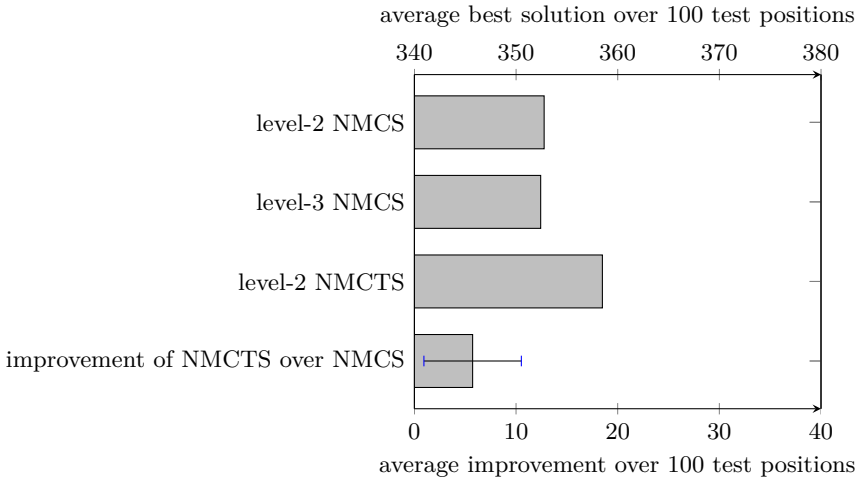


Figure 4.13: Performance of NMCS and level-2 NMCTS in Clickomania. NMCTS employs 25600 level-1 searches of 50 ms each. Move-by-move search is not used. $C_1 = 0$.

4.4 Conclusion and Future Research

In this chapter, we proposed *Nested Monte-Carlo Tree Search* (NMCTS) as an online planning algorithm for large sequential decision problems. It replaces calls to the rollout policy with calls to MCTS itself, recursively creating higher-quality rollouts for the higher levels of the search. Empirical results in the test domains of SameGame (with random and with informed rollouts), Bubble Breaker and Clickomania show that NMCTS significantly outperforms multi-start Monte-Carlo Tree Search (MCTS). Experiments also indicate performance superior to Nested Monte-Carlo Search (NMCS) in all test domains. Since both MCTS and NMCS represent specific parameter settings of NMCTS, correct tuning of NMCTS has to lead to greater or equal success in any domain. In conclusion, NMCTS is a promising approach to one-player search, especially for longer time settings.

When move-by-move search is applied at level 1 of a level-2 NMCTS search, the resulting algorithm is an MCTS variant using shallow MCTS searches to determine the rollout moves. A similar idea, using shallow *minimax* searches to determine the rollout moves of a higher-level MCTS search, is explored for two-player games in Chapters 7 and 8 of this thesis.

Three promising directions remain for future research on NMCTS. First, in the experiments so far we have only used an exploration factor of 0 for level 2. This means that the second level of tree search proceeded greedily in all experiments—it

only made use of the selectivity of MCTS, but not of the exploration-exploitation tradeoff. Careful tuning of exploration at all search levels could lead to performance improvements. Second, it appears that NMCTS is most effective in domains where multi-start MCTS outperforms a single, long MCTS run (like SameGame and Bubble Breaker), although its lower memory requirements can still represent an advantage in domains where multi-start MCTS is ineffective (like Clickomania). The differences between these classes of tasks remain to be characterized. Third, NMCTS could be extended to non-deterministic and partially observable domains, for example in the form of a nested version of POMCP (Silver and Veness, 2010).

5

Beam Monte-Carlo Tree Search

This chapter is based on:

Baier, H. and Winands, M. H. M. (2012). Beam Monte-Carlo Tree Search. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pages 227–233.

In the most widely used variants of MCTS, employing UCB1 or UCB1-TUNED (Auer et al., 2002) as selection strategies, the selectivity of the search is controlled by a single parameter: the exploration factor C. In some cases however, MCTS is not able to grow a search tree deep enough even when exploration is completely turned off—in one-player domains with long solution lengths for example, or when searching with a short time limit. Because the search effort potentially grows exponentially in the tree depth, the search process then spends too much time on optimizing the first moves of the solution, and not enough time on optimizing the last moves. One option to approach this problem is *move-by-move search* (Schadd et al., 2012) as used in the previous chapter, distributing the total search time over several or all moves in the game instead of conducting only one global search from the initial position. However, move-by-move search has to commit to a single move choice at each tree depth d before it starts a new search at $d + 1$. New results from simulations deeper in the tree cannot influence these early move decisions anymore.

Another option is *beam search*. This classic search method reduces the number of nodes at each tree level to a constant number, allowing for search effort linear in the tree depth. Since its first application in speech recognition (Lowerre, 1976), it has been used in a multitude of fields, such as machine translation (Tillmann and Ney, 2003), planning (Zhou and Hansen, 2006), and scheduling (Sabuncuoglu and Bayiz, 1999).

In this chapter, we answer the second research question by proposing *Beam Monte-Carlo Tree Search* (BMCTS), combining the MCTS framework with the idea of beam search. Like MCTS, BMCTS builds a search tree using Monte-Carlo simulations as state evaluations. When a predetermined number of simulations has traversed the nodes of a given tree depth, these nodes are sorted by a heuristic value, and only a fixed number of them is selected for further exploration. BMCTS is reduced to a variant of move-by-move MCTS if this number, the beam width, is set to one. However, it generalizes from move-by-move search as it allows to keep any chosen number of alternative moves when moving on to the next tree depth. BMCTS expands a tree whose size is linear in the search depth, improving on MCTS especially in domains with long solution lengths or short time limits. We compare the performance of BMCTS to that of regular MCTS, at a variety of time controls, in the one-player test domains SameGame, Clickomania, and Bubble Breaker.

This chapter is organized as follows. Section 5.1 provides an overview of related work on beam search. After Section 5.2 proposes BMCTS, Section 5.3 shows the behavior of the algorithm with respect to its parameters, and experimental results of testing it against MCTS. A combination of NMCTS (see the previous chapter) and BMCTS is considered as well. Conclusions and future work follow in Section 5.4.

5.1 Related Work

Beam search (Lowerre, 1976) is a technique that reduces the memory requirements of breadth-first or best-first search at the cost of completeness and optimality. Its basic idea is using heuristic value estimates to determine the most promising states at *each level* of the search tree. Only these states are then selected for further expansion, while all others are permanently pruned. Consequently, time and memory complexity of the search are linear in the beam width and the tree depth. By increasing or decreasing the beam width, memory can be traded off against solution quality, with a width of 1 resulting in a greedy search, and an infinite width resulting in a complete search.

Beam search has also been extended by combining it with depth-first search (Zhou and Hansen, 2005) as well as with limited discrepancy search (Furcy and Koenig, 2005). These variants turn beam search into a *complete* search algorithm, i.e. an algorithm guaranteed to find a solution when there is one.

In the Monte-Carlo framework, Monte-Carlo Beam Search (MCBM, Cazenave 2012) combines beam search with Nested Monte-Carlo Search (NMCS), a special case of NMCTS (see Chapter 4) that has shown good results in various one-player games (Cazenave, 2009). Beam search has so far not been applied to MCTS. A similar idea to beam search, however, has been applied *per node* instead of *per tree level*. *Progressive widening* or *unpruning* (Chaslot et al., 2008; Coulom, 2007a) reduces the number of

children of rarely visited nodes. Heuristics are used to choose the most promising children. As the number of rollouts passing through the node increases, i.e. as the node is found to be more and more important by the search, the pruned children are progressively added again. This way, search effort for most nodes can be reduced, while retaining convergence to the optimal moves in the limit. A disadvantage of this technique, especially for one-player games, is that the search effort per tree level cannot be controlled, and therefore exponential growth in the tree depth still potentially poses a problem.

In the next section, we describe an application of the beam search idea in MCTS, reducing the time and space complexity of MCTS to linear in the tree depth. Our algorithm does not require heuristic knowledge.

5.2 Beam Monte-Carlo Tree Search

In this section, we propose *Beam Monte-Carlo Tree Search* (BMCTS), our approach for combining beam search with MCTS. In addition to the MCTS tree, BMCTS maintains a counter for each tree depth, counting the number of simulated games that have passed through any tree node at this depth in the search so far. During backpropagation, these counters are compared with the first parameter of BMCTS: the *simulation limit* L. If any tree depth d reaches this limit, the tree is pruned at level d.

Pruning restricts the number of tree nodes at depth d to the maximum number given by the second parameter of BMCTS: the *beam width* W. In order to do this, all tree nodes of depth d are first sorted by their heuristic values. Due to the large variance of Monte-Carlo value estimates at low simulation counts, we use the number of visits of a tree node instead of its estimated value as our heuristic—nodes that have seen the highest numbers of simulations are judged to be most promising by the search algorithm (similar to the final move selection mechanism "robust child", Chaslot et al. 2008). Then, the best W nodes at level d together with all of their ancestor nodes are retained, while all of their descendants as well as the less promising nodes of depth d are discarded. Deleting the descendants might not be optimal in every domain, but helped avoid getting stuck in local optima in preliminary experiments.

When the search continues, no new nodes up to depth d will be created anymore. The selection policy takes only those moves into account that lead to the retained nodes. Beyond depth d, the tree grows as usual.

Note that with $W = 1$, BMCTS is reduced to a variant of *move-by-move search* as described in Schadd et al. (2012). This time-management technique distributes search time over several or all moves in the game instead of conducting only one global search from the initial position (see Chapter 4). If $W = 1$, the L parameter of

BMCTS determines the timings when move-by-move MCTS proceeds from move to move, and therefore the total number of move searches the search time is distributed over. However, BMCTS generalizes from move-by-move MCTS to larger beam widths than 1, examining a number of alternative moves instead of focusing on just one when proceeding with the next move of the game.

Algorithms 5.1 and 5.2 show pseudocode of BMCTS for deterministic one-player games, using a uniformly random rollout policy. There are only two differences between the BMCTS pseudocode in Algorithm 5.1 and the pseudocode of regular MCTS in Algorithm 2.6. In line 8, the rollout counter for the depth currently traversed in the selection phase is incremented. In lines 20 to 22, the tree pruning method `pruneTree` is called in case any traversed depth has reached the simulation limit L. This three-step tree pruning method is outlined in Algorithm 5.2 and visualized in Figure 5.1. Lines 2, 3, and 4 of Algorithm 5.2 correspond to Subfigures 5.1(b), 5.1(c), and 5.1(d), respectively.

5.3 Experimental Results

In this section, we compare regular MCTS and BMCTS in the domains SameGame, Clickomania, and Bubble Breaker, using a random rollout policy. For SameGame, we also employ the TabuColorRandomPolicy as rollout policy (see Section 4.3). In Subsection 5.3.1, we start by showing the parameter landscape of BMCTS in the test domains, explaining how optimal settings for C, L, and W were found. Afterwards, these settings are used in Subsection 5.3.2 to compare the performance of BMCTS and regular MCTS. The performance of genuine beam search with $W > 1$ is compared to that of move-by-move search with $W = 1$ as well. In Subsection 5.3.3, BMCTS and regular MCTS are compared again, this time using the maximum over multiple runs instead of the result of a single search run for comparison. These results are finally used in Subsection 5.3.4 to explore the combination of BMCTS with NMCTS as proposed in Chapter 4.

5.3.1 Parameter Optimization

In the first set of experiments, we examine the influence of the BMCTS parameters C, L, and W in each test domain. These tuning experiments are conducted on a training set of 500 randomly generated 20×20 boards with 10 different tile colors in Clickomania, and 500 randomly generated 15×15 boards with 5 different tile colors in SameGame and Bubble Breaker.

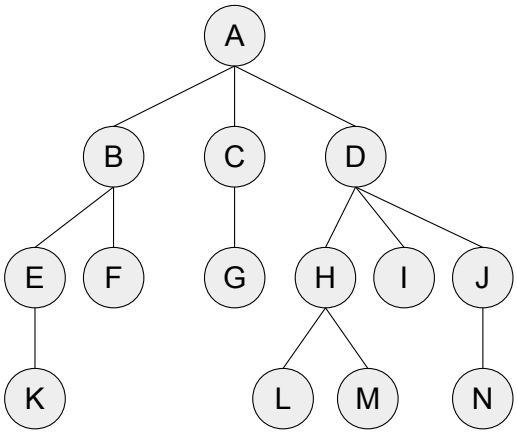
The time settings at which we compare BMCTS to regular MCTS in Subsections 5.3.2 to 5.3.4 are 0.05, 0.25, 1, 4, 15, 60, and 240 seconds in SameGame and Bubble

```
BMCTS(startState) {
   bestResult ← -Infinity
   bestSimulation ← {}
   for(numberOfIterations) {
      currentState ← startState
      simulation ← {}
      while(currentState ∈ Tree) {
         numberOfRolloutsThrough[currentState.depth]++
         currentState ← takeSelectionPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      addToTree(currentState)
      while(currentState.notTerminalPosition) {
         currentState ← takeRolloutPolicyAction(currentState)
         simulation ← simulation + currentState
      }
      result ← cumulativeReward(simulation)
      forall(state ∈ {simulation ∩ Tree}) {
         state.value ← backPropagate(state.value, result)
         if(numberOfRolloutsThrough[state.depth] = SIMLIMIT) {
            pruneTree(state.depth, Tree)
         }
      }
      if(result > bestResult) {
         bestResult ← result
         bestSimulation ← simulation
      }
   }
   return (bestResult, bestSimulation)
}
```

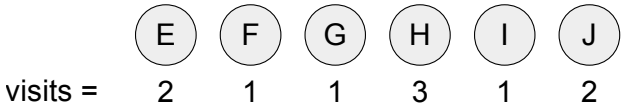
Algorithm 5.1: BMCTS.

```
pruneTree(depth, Tree) {
   nodeSet ← treeNodesAtDepth(Tree, depth)
   nodeSet ← mostVisitedTreeNodes(nodeSet, BEAMWIDTH)
   Tree ← nodeSet + ancestorNodes(nodeSet)
}
```

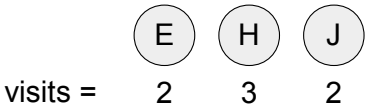
Algorithm 5.2: Tree pruning in BMCTS.



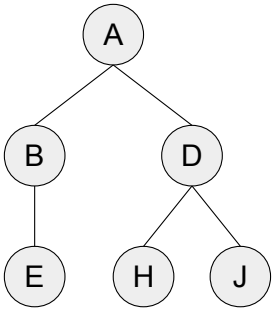
(a) The tree before pruning on depth $d = 2$.



(b) The nodes with $d = 2$ are collected. $L = 10$ rollouts have passed through them in total.



(c) The $W = 3$ nodes with the highest visit count are retained, the rest is pruned.



(d) Ancestor nodes are added. Search continues on the new tree.

Figure 5.1: Tree pruning in BMCTS. Depth $d = 2$, beam width $W = 3$, simulation limit $L = 10$.

Table 5.1: Best-performing simulation limits L and beam widths W for BMCTS in SameGame with random rollouts, SameGame with informed rollouts, and Bubble Breaker.

Time in s	SameGame random		SameGame informed		Bubble Breaker	
	L	W	L	W	L	W
0.25	100	5	75	5	20	2
1	250	5	250	10	150	15
4	2500	10	1000	100	2500	5
15	20000	2	20000	50	20000	5
60	50000	5	100000	100	100000	5
240	250000	25	250000	250	500000	25

Breaker, and 0.016, 0.05, 0.25, 1, 4, 16, and 64 seconds in Clickomania. Optimal MCTS exploration factors C for these time settings have already been determined in Subsection 4.3.1, and are also used for BMCTS in this chapter. Although slightly larger exploration factors were found to improve the performance of BMCTS in some cases, increasing its advantage over MCTS by up to 50%, a systematic optimization for all time settings and test domains was not possible due to computational limitations.

BMCTS was optimized with regard to its parameters W and L, using the exploration factor that was found to be optimal for regular MCTS at the same time setting and in the same domain. We tested about 10 different W values between 1 and 1000, and about 15 different L values from 10 to 10^6. The optimal W and L settings for all games and search times are listed in Tables 5.1 and 5.2. As illustrative examples, Figures 5.2 to 5.5 show the performance of BMCTS with different values of L and W for the case of 4-second searches in all four test domains. Each data point represents the average result over 500 test positions.

Qualitatively, all domains show the same behavior, although the effects appear to be strongest in Clickomania and weakest in SameGame with informed rollouts. If the simulation limit L is very high, the tree pruning of BMCTS is never triggered, and if the beam width W is very high, it is triggered but has no effect. BMCTS therefore reaches a plateau with very high values of L and/or W, corresponding to the performance of regular MCTS with the same time setting. If W is small enough for the tree pruning to have an effect, the performance depends strongly on L, i.e. on the frequency of the pruning. Very high values of L, as mentioned above, result in no pruning and BMCTS is equivalent to MCTS. Somewhat smaller values of L result in a drop in performance—pruning hurts performance by prematurely excluding a number of moves at the root or close to the root now, but there is not enough pruning yet to push the search deep in the tree and reap the benefits of beam search. With

Table 5.2: Best-performing simulation limits L and beam widths W for BMCTS in Clickomania.

Time in s	L	W
0.016	10	3
0.05	10	3
0.25	10	3
1	10	3
4	100	15
16	1000	25
64	10000	500

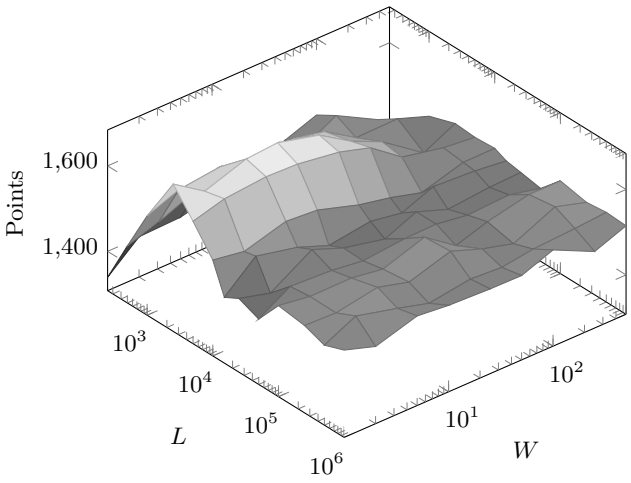


Figure 5.2: Performance of BMCTS at 4 seconds per position in SameGame with random rollouts. $C = 0.0002$.

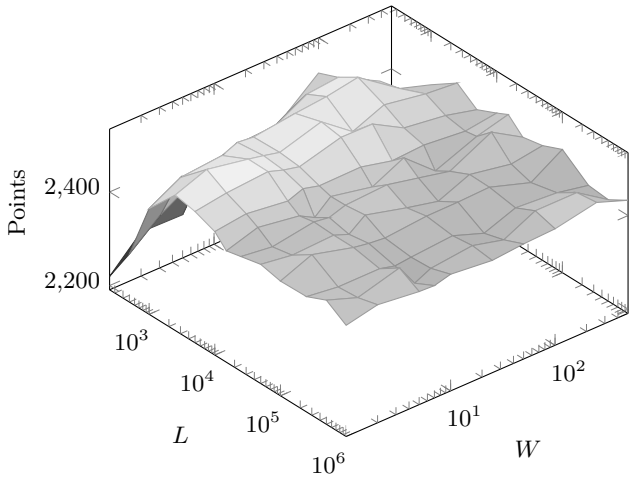


Figure 5.3: Performance of BMCTS at 4 seconds per position in SameGame with informed rollouts. $C = 0.0005$.

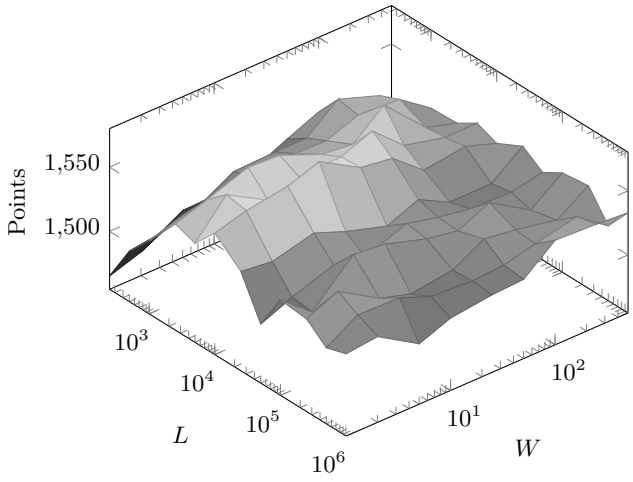


Figure 5.4: Performance of BMCTS at 4 seconds per position in Bubble Breaker. $C = 0.008$.

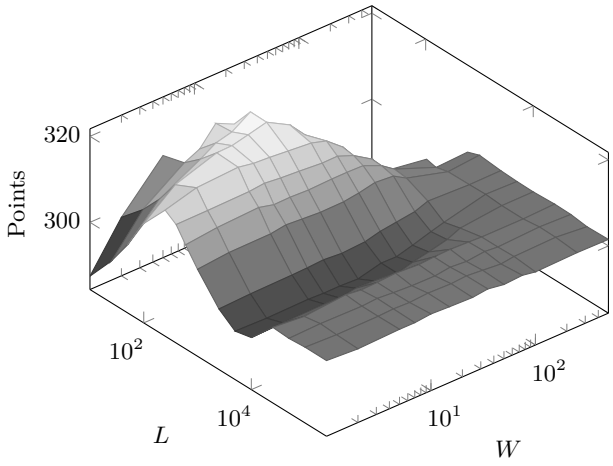


Figure 5.5: Performance of BMCTS at 4 seconds per position in Clickomania. $C = 0$.

decreasing values of L, pruning is triggered more and more often. The search explores deeper and deeper moves in the tree, and performance increases until it reaches a maximum. If even smaller values of L are chosen, performance drops again as the search reaches the leaves of the tree too quickly and cannot fully exploit the 4 seconds of search time anymore.

Comparing the domains, Clickomania is the longest game variant on average (compare Section 3.1), which is part of the reason why pruning works best at low values of L in this domain. Clickomania needs to prune more frequently to get deep enough in the tree. Moreover, it is relatively more important to optimize late moves in Clickomania. While the other domains reward large groups so strongly that the effects of moves after deleting the largest group are often negligible, Clickomania only rewards emptying the board as far as possible in the end. Therefore, it is important to optimize all moves of the game in Clickomania, while it is more useful to spend time on the first part of the game and the forming of the largest group in the other domains. This is probably another factor leading to relatively low optimal L values in Clickomania, and also to beam search overall having the strongest effect in this game variant. Finally, Clickomania is different from the other domains in that its parameter landscape in Figure 5.5 appears smoother. The reason is that Clickomania rewards are linear in the number of tiles cleared from the board, which means that good solutions have similar rewards—clearing 1 tile more only results in 1 more point. SameGame and Bubble Breaker give rewards quadratic in the group size—this can lead to two good solutions having drastically different total rewards if their largest groups are just 1 tile different in size.

Another factor influencing the performance of BMCTS is the total search time. Whereas the previous paragraphs discussed the parameter landscape of BMCTS at 4 seconds per search in all test domains, Figures 5.6 to 5.9 show the algorithm's behavior at 0.25, 1, 16, and 64 seconds in Clickomania.

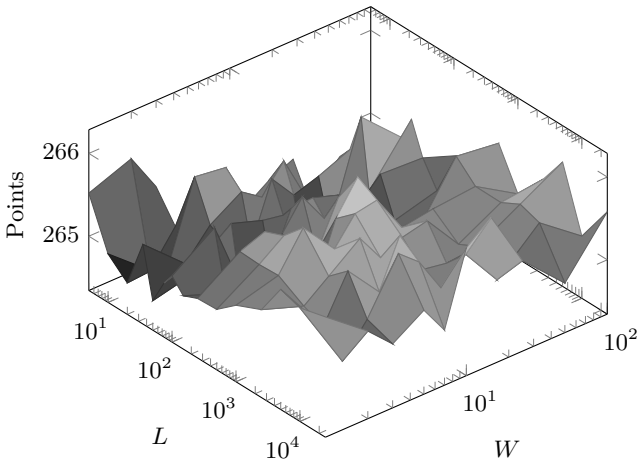


Figure 5.6: Performance of BMCTS at 0.25 seconds per position in Clickomania. $C = 0$.

The results of these experiments are again qualitatively similar to the other domains, although the strengths of the effects and the exact time settings at which they occur vary. One can observe that the shorter 1 second time setting shows similar behavior to 4 seconds, except that the difference between MCTS performance (the plateau at high L values) and the optimal BMCTS performance (the highest peak) is smaller in both absolute and relative terms. A more detailed comparison of BMCTS to MCTS follows in the next subsection. At the even shorter time setting of 0.25 seconds per position, the parameter landscape is essentially random. Beam search has no measurable effect anymore at this low number of samples. At the higher time setting of 16 seconds, we find the same landscape as at 4 seconds again—but the difference between the MCTS plateau and the maximum is smaller here as well. At the highest tested setting of 64 seconds finally, there is virtually no difference anymore and BMCTS is similarly ineffective as at extremely short time settings. The reason for the drop in BMCTS performance at long time settings is possibly the fact that with a small enough exploration factor and enough time, regular MCTS already searches sufficiently deep in the tree. Pruning is not beneficial since the MCTS selection policy is able to decide on most moves of the game. Using higher exploration factors for BMCTS than for MCTS might give BMCTS an additional edge at such long time settings, depending on pruning instead of low exploration to get deeper in the tree.

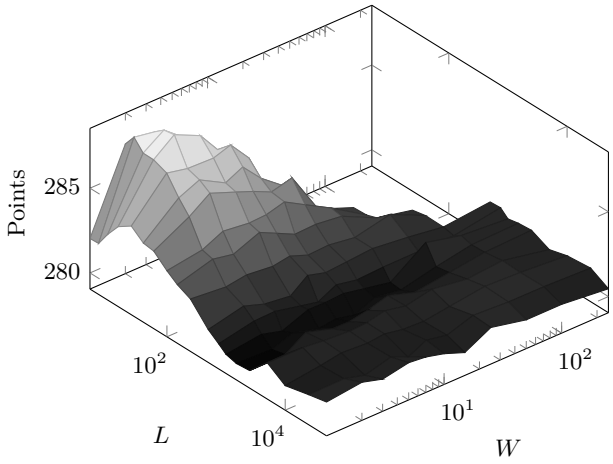


Figure 5.7: Performance of BMCTS at 1 second per position in Clickomania. $C = 0$.

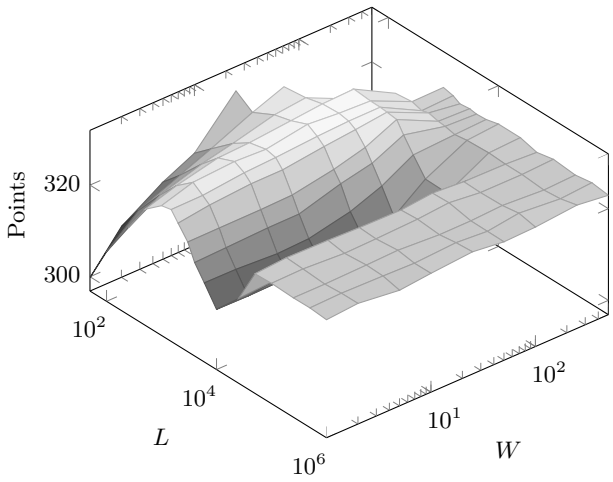


Figure 5.8: Performance of BMCTS at 16 seconds per position in Clickomania. $C = 0.0025$.

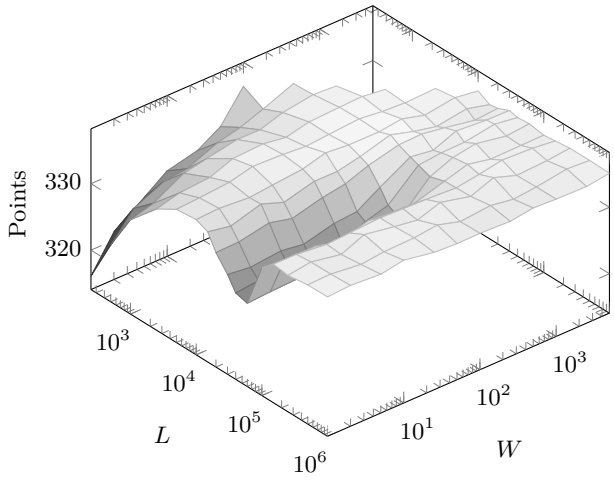


Figure 5.9: Performance of BMCTS at 64 seconds per position in Clickomania. $C = 0.012$.

5.3.2 Comparison to MCTS

After finding the optimal values for simulation limit L and beam width W for each time setting and domain, independent test sets of 1000 positions for each domain were used to compare BMCTS to regular MCTS. Clickomania again used 20×20 boards with 10 different tile colors, while for all other test domains 15×15 boards with 5 different tile colors were created. The time limits were 0.016, 0.05, 0.25, 1, 4, 16, and 64 seconds in Clickomania, and 0.05, 0.25, 1, 4, 15, 60, and 240 seconds in the other domains. At each of these settings, the average performance of regular MCTS was determined, the average performance of BMCTS with the optimal L and $W > 1$ found in the last subsection, and also the performance of move-by-move search—implemented in our framework as the special case of BMCTS search with $W = 1$—with the optimal L. $W > 1$ is required for BMCTS in this comparison in order to test the effect of genuine beam search.

Figures 5.10 to 5.13 present the results. They show that at the shortest tested time settings, up to 0.25 seconds per search, the performance of a global MCTS search cannot be improved by splitting up the already short search time into several consecutive move searches. At 1 second per search, BMCTS begins to provide an improvement to MCTS (significant at $p<0.05$ in SameGame with random rollouts, $p<0.01$ in Bubble Breaker, $p<0.0001$ in Clickomania—not significant with $p<0.08$ in SameGame with informed rollouts). Move-by-move search is not able to improve on MCTS at this time setting yet, except for the domain of Clickomania, where it is still weaker than BMCTS ($p<0.0001$). At 4 seconds per search, BMCTS significantly

improves on MCTS in all domains (at $p<0.01$ in SameGame with random rollouts, $p<0.05$ in SameGame with informed rollouts, $p<0.01$ in Bubble Breaker, $p<0.0001$ in Clickomania). Move-by-move search however has now caught up with BMCTS in all domains but Clickomania, where BMCTS is still stronger at $p<0.0001$. The same relationships hold at longer time controls (15, 60, and 240 seconds in SameGame and Bubble Breaker, 16 seconds in Clickomania)—BMCTS is significantly better than MCTS, but $W > 1$ is only significantly different from $W = 1$ in Clickomania. At the longest times tested in Clickomania, 64 seconds per search, BMCTS is not stronger than MCTS anymore. In the other domains, we can also see a trend of the BMCTS performance approaching the MCTS performance. However, MCTS runs into memory problems at longer time settings, which gives BMCTS an additional advantage.

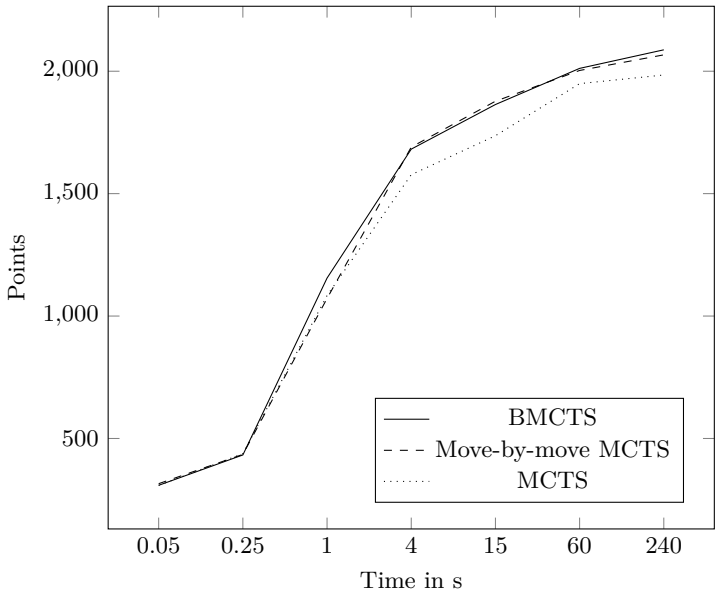


Figure 5.10: Performance of BMCTS in SameGame with random rollouts.

In conclusion, one can observe a relatively wide timing window in which BMCTS is superior to MCTS. This windows starts somewhere between 0.25 and 1 second per search in our experiments, and ends somewhere between 16 and 64 seconds in Clickomania. In the other domains, the upper limit of the timing window could not definitely be determined due to memory limitations that gave BMCTS an additional advantage over MCTS. Furthermore, there is another timing window in which BMCTS with $W > 1$ (called BMCTS in the preceding paragraph) is superior to BMCTS with $W = 1$ (called move-by-move search in the preceding paragraph). In SameGame and

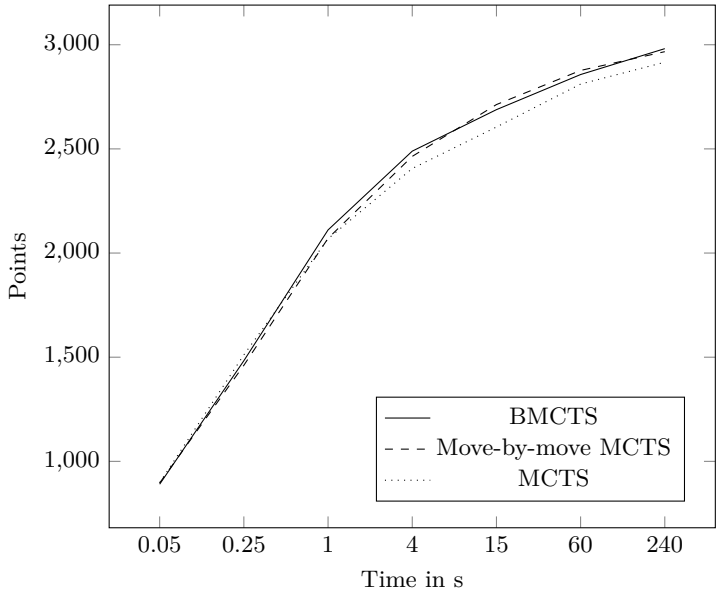


Figure 5.11: Performance of BMCTS in SameGame with informed rollouts.

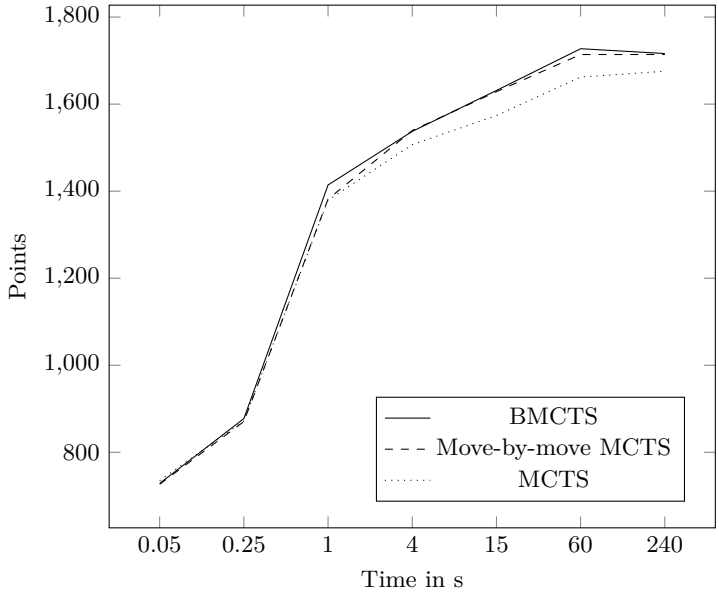


Figure 5.12: Performance of BMCTS in Bubble Breaker.

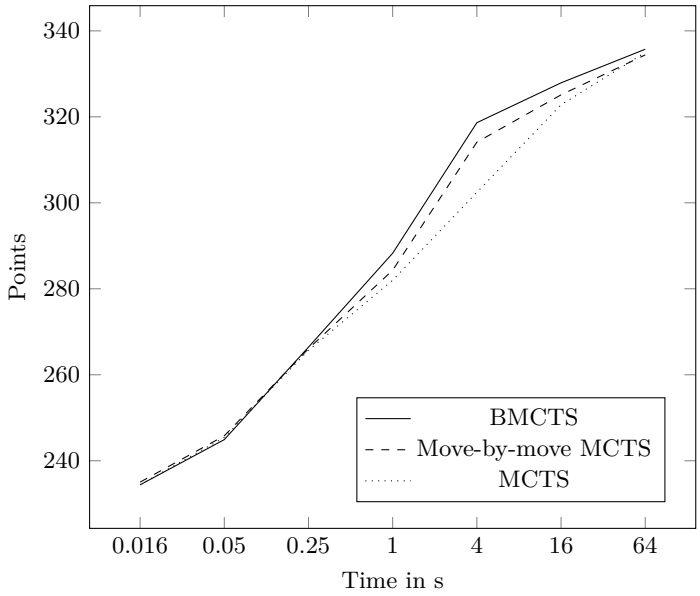


Figure 5.13: Performance of BMCTS in Clickomania.

Bubble Breaker, this window was found to lie somewhere between 0.25 and 4 seconds per search. In Clickomania, $W > 1$ is stronger than $W = 1$ whenever BMCTS is stronger than MCTS. This illustrates once more that among the tested game variants, deep search and therefore beam search is most relevant in Clickomania.

5.3.3 Multi-Start Comparison to MCTS

An interesting observation about BMCTS is that its behavior does not necessarily stay the same when one moves from single-start to multi-start experiments, i.e. from measuring the average result of a single search run on each test position to considering the maximum result of several search runs on each test position. As an example, compare Figures 5.6 and 5.14. Each data point in Figure 5.6 represents the average performance of a single BMCTS run of 0.25 seconds length in Clickomania, and each point in Figure 5.14 represents the average maximum of 50 search runs of 0.25 seconds length each per position. For computational reasons, only 100 instead of 500 training positions were used here.

While the parameter landscape of BMCTS with regard to L and W shows only noise in the single-start scenario—BMCTS cannot improve on the average result of a single MCTS run at 0.25 seconds in Clickomania—the results of the multi-start

experiment look similar to the single-start results at 1 second or more per position (see e.g. Figure 5.7). The plateau at high values of L and W represents the performance of multi-start MCTS in Figure 5.6, and the optimal multi-start BMCTS performance is significantly better. It seems that whereas the solutions found by BMCTS do not have a higher mean than the solutions found by MCTS, they have a higher variance. The very low number of rollouts on each level of the tree, and therefore the essentially random pruning decisions, seem to lead to different search runs exploring different parts of the search space. This leads to more exploration and higher maxima over multiple search runs. The other test domains showed similar results.

This effect seems to be stronger at short time settings than at long time settings. Experiments were consequently conducted in all domains in order to determine whether testing with 50 runs leads to different optimal L and W values than testing with a single run. The first 100 training positions of the sets described in Subsection 5.3.1 were used. In particular at short time settings, the multi-start effect described in the previous paragraph leads to different optimal parameter settings. The tests therefore started at the shortest time setting in each domain (e.g. 0.25 seconds) and continued tuning with longer time settings (1 second, 4 seconds etc.) until no difference to the optimal single-start L and W was found anymore. This made it unnecessary to retune at the longest time settings, which would have been computationally expensive.

But even at longer time settings and with unchanged parameters, an additional advantage of BMCTS over MCTS can be observed in the multi-start scenario. The optimal W and L settings for all games and search times are listed in Tables 5.3 and 5.4. Figures 5.15 to 5.18 show the performance of these settings analogously to Figures 5.10 to 5.13, replacing single-run results with maxima over 50 search runs. The positions are the first 100 test positions of the sets used in Subsection 5.3.2. Multi-start BMCTS is significantly stronger than multi-start MCTS at all search times from 0.05 to 60 seconds in SameGame with random rollouts, from 0.05 to 15 seconds in SameGame with informed rollouts, from 0.05 to 60 seconds in Bubble Breaker, and from 0.25 to 16 seconds in Clickomania.

5.3.4 Combination of BMCTS and NMCTS

The previous subsection demonstrated that BMCTS can have an additional advantage over MCTS when the algorithms are run multiple times on the same test position, especially at short search times. This led to the idea of examining how BMCTS would perform in a nested setting, i.e. replacing MCTS as the basic search algorithm in Nested Monte-Carlo Tree Search (see Chapter 4). We call the resulting search algorithm *NBMCTS* for *Nested Beam Monte-Carlo Tree Search*. In principle, BMCTS could be applied at all levels of NBMCTS. In this subsection however, we only report

Table 5.3: Simulation limits L and beam widths W for multi-start BMCTS in SameGame with random rollouts, SameGame with informed rollouts, and Bubble Breaker.

Time in s	SameGame random		SameGame informed		Bubble Breaker	
	L	W	L	W	L	W
0.25	75	2	25	5	75	3
1	500	2	250	7	1000	7
4	10000	2	1000	100	5000	5
15	20000	2	20000	50	20000	5
60	50000	5	100000	100	100000	5
240	250000	25	250000	250	500000	25

Table 5.4: Simulation limits L and beam widths W for multi-start BMCTS in Clickomania.

Time in s	L	W
0.016	10	3
0.05	10	3
0.25	10	3
1	25	2
4	100	15
16	1000	25
64	10000	500

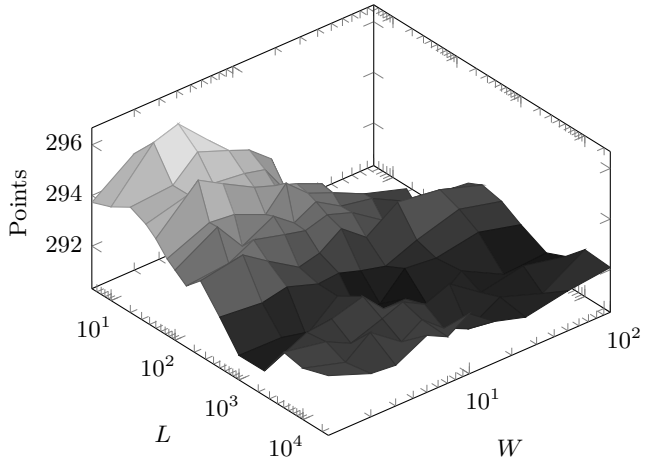


Figure 5.14: Performance of multi-start BMCTS at 0.25 seconds per run in Clickomania. $C = 0$, 100 test positions, 50 search runs per position.

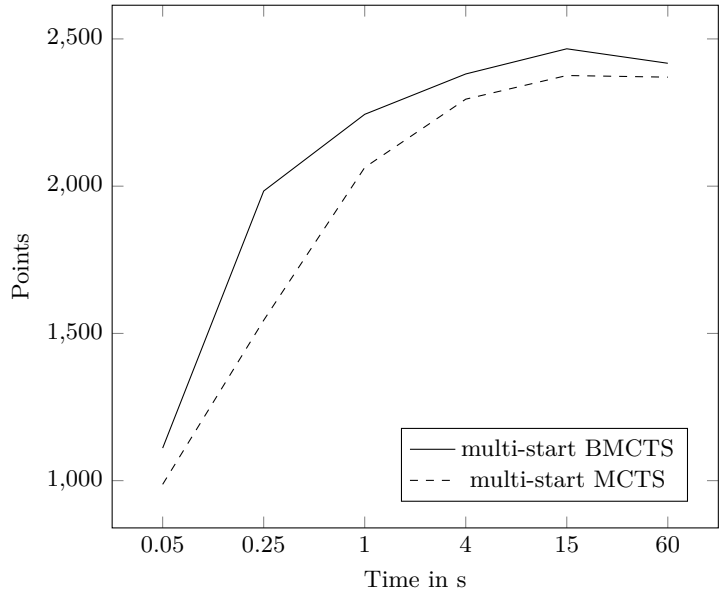


Figure 5.15: Performance of multi-start BMCTS in SameGame with random rollouts. 100 test positions, 50 search runs per position.

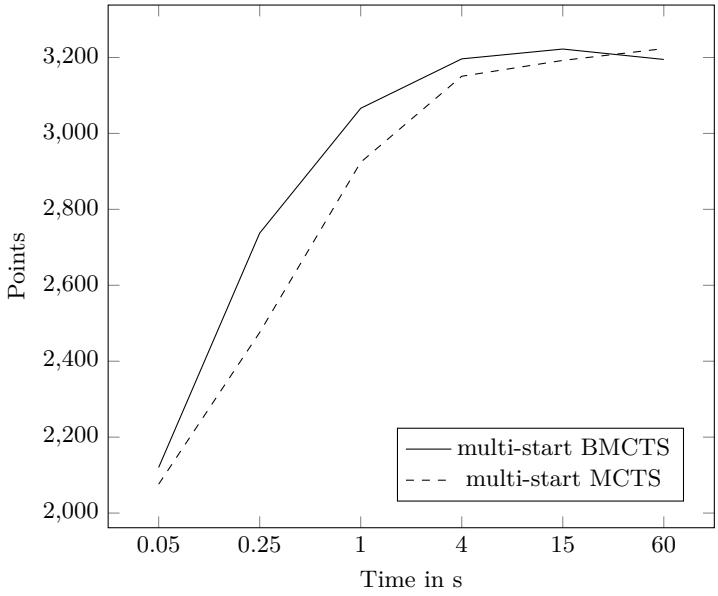


Figure 5.16: Performance of multi-start BMCTS in SameGame with informed rollouts. 100 test positions, 50 search runs per position.

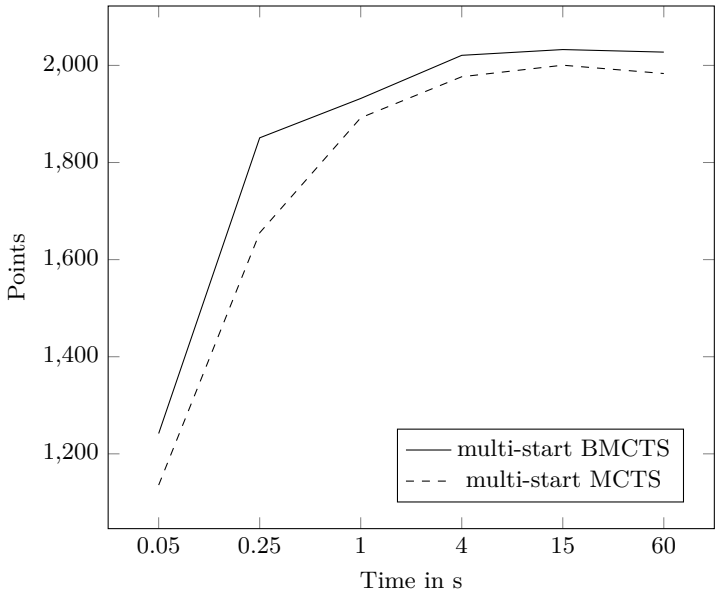


Figure 5.17: Performance of multi-start BMCTS in Bubble Breaker. 100 test positions, 50 search runs per position.

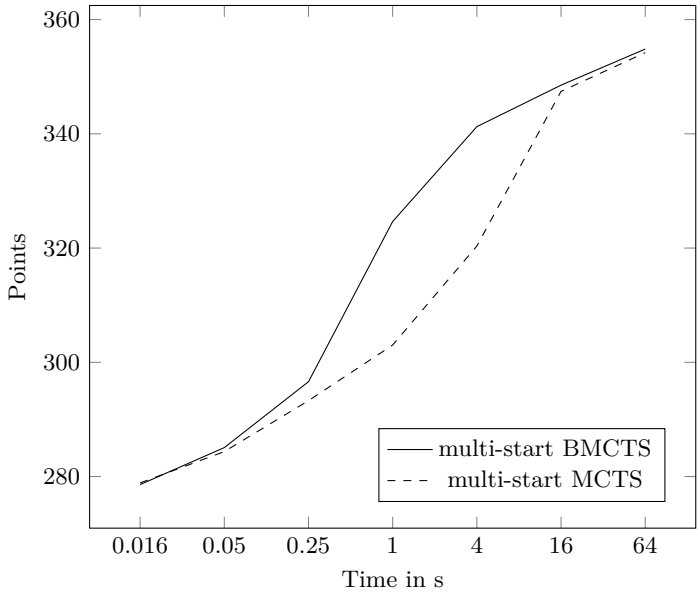


Figure 5.18: Performance of multi-start BMCTS in Clickomania. 100 test positions, 50 search runs per position.

preliminary results for applying it at level 1, while level 2 remains regular MCTS. The main reason is the computational cost of optimizing the parameters L and W for level 2.

The previous subsection showed that the optimal parameter settings for multi-start BMCTS can be different from those of single-start BMCTS. This suggests the number of restarts is likely to have an influence as well, e.g. the optima for 50 runs could be different from the optima for 500 or 5000 runs. In order to properly optimize multi-start BMCTS or NBMCTS, we would therefore have to use the same number of restarts or level-1 searches during tuning and testing. Using the same total search time as multi-start MCTS and NMCTS in Chapter 4—9120 seconds in SameGame and Bubble Breaker, and 1280 seconds in Clickomania—this would require tuning for 608 searches of 15 seconds duration for example, tuning for 2280 searches of 4 seconds duration etc. Since this would have been prohibitively computationally expensive, the settings found for 50 restarts in Subsection 5.3.3 were used as an approximation in all domains and time settings.

Figures 5.19 to 5.22 present the performance of multi-start BMCTS and NBMCTS. The multi-start MCTS and NMCTS results of Chapter 4 are added for comparison. Just as level-2 NMCTS significantly outperformed multi-start MCTS, level-2 NBMCTS

was significantly ($p<0.05$) stronger than multi-start BMCTS at all numbers of restarts in SameGame with both random and informed rollouts, and in Bubble Breaker. It was also stronger in Clickomania with 1280 and more restarts, while performing equally with fewer restarts. Moreover, multi-start BMCTS performed significantly better than multi-start MCTS at all numbers of restarts in SameGame with random rollouts, all numbers but 608 in SameGame with informed rollouts, all numbers but 2280 in Bubble Breaker, and with 1280 and 320 restarts in Clickomania, while showing no significant difference otherwise. NBMCTS finally outperformed NMCTS at all numbers of restarts but 36480 in SameGame with random rollouts and in Bubble Breaker, and again with 1280 and 320 restarts in Clickomania. In SameGame with informed rollouts, the generally higher level of performance might make more thorough parameter tuning necessary than was possible for this chapter (see the previous two paragraphs)—here NBMCTS could only be shown to be stronger than NMCTS with 38 and with 9120 restarts. At all other numbers of restarts, there was no significant difference to NMCTS.

Note that whereas NMCTS has the largest advantage over multi-start MCTS at the highest tested number of restarts in both SameGame variants and in Bubble Breaker (cf. Subsection 4.3.2), probably because only a level-2 tree with this many level-1 searches grows to a sufficient size to have a large impact on performance, this is not the case when comparing NBMCTS to multi-start BMCTS. The improvement of adding beam search to multi-start MCTS is much larger than the improvement of adding beam search to NMCTS. This could indicate that both multiple nested searches and multiple beam searches improve on MCTS partly in a similar way, namely by improving exploration. This could be investigated further in future work. However, NBMCTS either outperforms NMCTS or performs comparatively to it in all domains and at all time settings. In conclusion, NBMCTS can therefore be considered a successful combination of the NMCTS and BMCTS approaches, and is the overall strongest one-player algorithm proposed in this thesis.

5.4 Conclusion and Future Research

In this chapter, we proposed *Beam Monte-Carlo Tree Search* (BMCTS), integrating the concept of beam search into an MCTS framework. Its time and space complexity are linear in the search depth. Therefore, it can improve on the selectivity of regular MCTS especially in domains whose solutions are relatively long for the available search time. Experimental results show BMCTS to significantly outperform regular MCTS at a wide range of time settings in the test domains of Bubble Breaker, Clickomania, and SameGame. Outside of this domain-dependent range, BMCTS is equally strong as MCTS at the shortest and the longest tested search times—although the lower

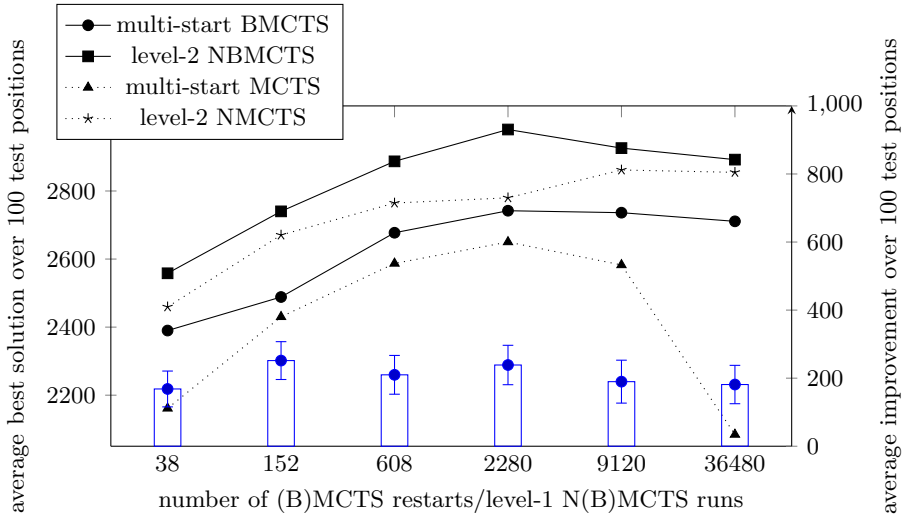


Figure 5.19: Performance of level-2 NBMCTS in SameGame with random rollout policy. Bars show the average performance increase over multi-start BMCTS with a 95% confidence interval. The search time was 9120 seconds per position.

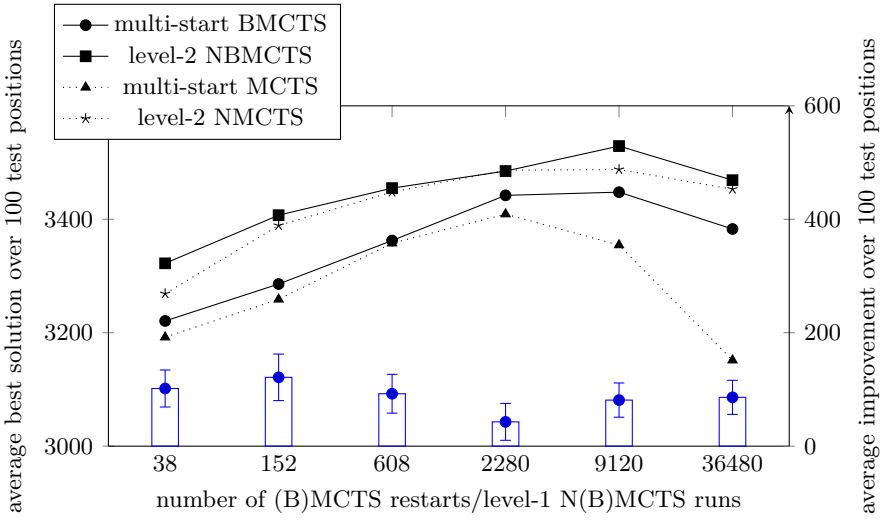


Figure 5.20: Performance of level-2 NBMCTS in SameGame with informed rollout policy. Bars show the average performance increase over multi-start BMCTS with a 95% confidence interval. The search time was 9120 seconds per position.

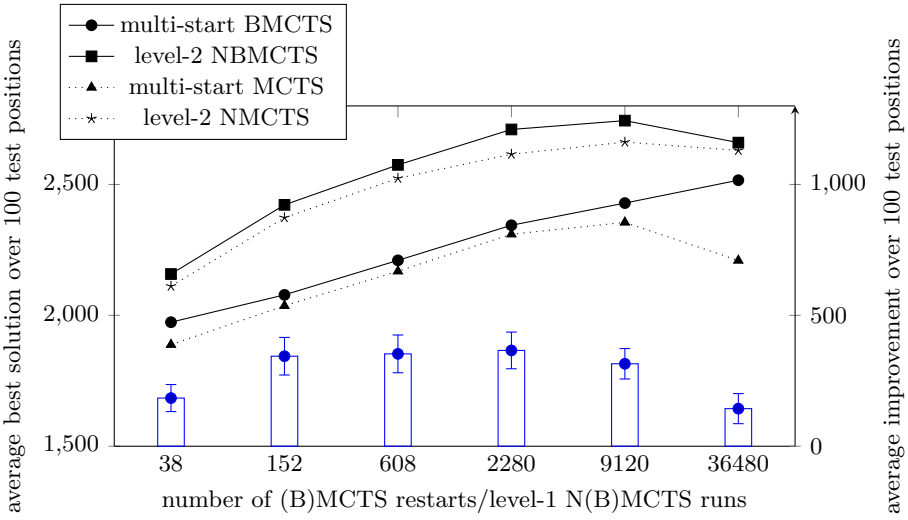


Figure 5.21: Performance of level-2 NBMCTS in Bubble Breaker. Bars show the average performance increase over multi-start BMCTS with a 95% confidence interval. The search time was 9120 seconds per position.

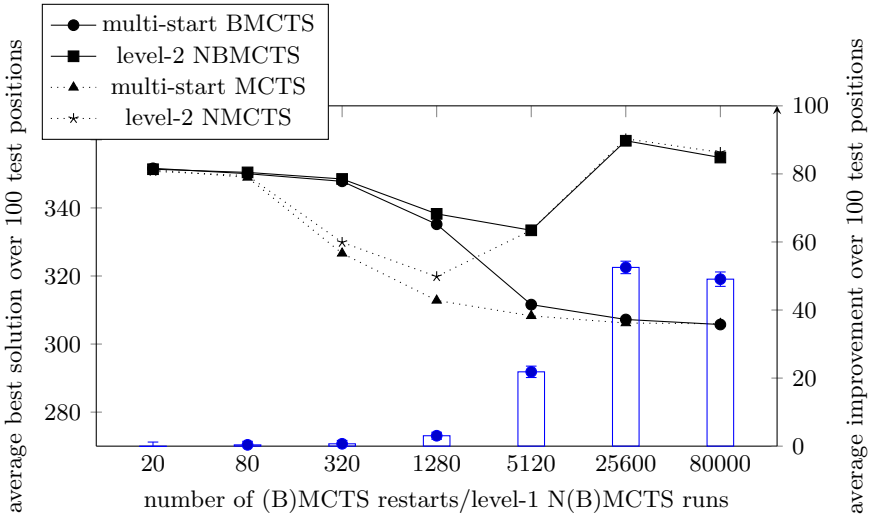


Figure 5.22: Performance of level-2 NBMCTS in Clickomania. Bars show the average performance increase over multi-start BMCTS with a 95% confidence interval. The search time was 1280 seconds per position.

memory requirements of BMCTS can give it an additional advantage over MCTS at long search times. In the experiments described here, the improved performance of beam search was achieved even without systematically optimizing the exploration factor C for BMCTS. Depending on the domain and time setting, optimal parameter settings can either result in a move-by-move time management scheme ($W = 1$), or in a genuine beam search using several states per tree level ($W > 1$). Move-by-move search is in this sense a special case of beam search. BMCTS with $W > 1$ was found to significantly outperform move-by-move search at a domain-dependent range of search times. Overall, BMCTS was most successful in Clickomania as this domain seems to profit most from deep searches.

Further experiments demonstrated BMCTS to have a larger advantage over MCTS in multi-start scenarios where maxima over several runs per position are considered instead of results of a single run per position. This suggests that the performance of BMCTS tends to have a higher variance than the performance of regular MCTS, even in some cases where the two algorithms perform equally well on average. In all test domains, multi-start BMCTS is superior to multi-start MCTS at a wider range of time settings than single-start BMCTS to single-start MCTS.

This observation led to the idea of combining the NMCTS and BMCTS approaches into the *Nested Beam Monte-Carlo Tree Search* (NBMCTS) algorithm. Experiments have shown NBMCTS to be the overall strongest one-player algorithm proposed in this thesis, performing better than or equal to NMCTS in all domains and at all search times. In conclusion, BMCTS is a promising approach to one-player search, especially for shorter time settings. At longer time settings, it combines well with NMCTS as proposed in the previous chapter.

Three directions appear promising for future work. First, BMCTS as presented in this chapter does not retain the asymptotic properties of MCTS—due to the permanent pruning of nodes, optimal behavior in the limit cannot be guaranteed. The addition of e.g. gradually increasing beam widths, similar to progressive widening (Chaslot et al., 2008; Coulom, 2007a) but on a per-depth instead of per-node basis, could restore this important completeness property. Second, the basic BMCTS algorithm could be refined in various ways, for instance by using different simulation limits and beam widths for different tree depths, or by experimenting with different heuristics for selecting the beam nodes. Techniques such as *stratified search* (Lelis et al., 2013) could potentially increase the diversity of nodes in the beam and therefore improve the results. Any such refinements should ideally go along with characterizations of the classes of tasks for which they are most effective. Third, it would be interesting to further compare the effects of multiple nested searches, and the effects of multiple beam searches on MCTS exploration, as mentioned in the previous section.

Part II

MCTS in Two-Player Domains

6

Time Management for Monte-Carlo Tree Search

This chapter is based on:

Baier, H. and Winands, M. H. M. (2015). Time Management for Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*. In press.

Baier, H. and Winands, M. H. M. (2012). Time Management for Monte-Carlo Tree Search in Go. In H. J. van den Herik and A. Plaat, editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 39–51.

In tournament gameplay, time is a limited resource. *Sudden death*, the simplest form of time control, allocates to each player a fixed time budget for the whole game. If a player exceeds this time budget, she loses the game immediately. Since longer thinking times typically result in stronger moves, the player's task is to distribute her time budget wisely among all moves in the game. This is a challenging task both for human and computer players. Previous research on this topic (Althöfer et al., 1994; Donninger, 1994; Hyatt, 1984; Markovitch and Sella, 1996; Šolak and Vučković, 2009) has mainly focused on the framework of $\alpha\beta$ search with iterative deepening. In a number of game domains however, this algorithm is more and more losing its appeal.

Compared to $\alpha\beta$ search, less has been published on *time management* for MCTS (Baudiš, 2011; Huang et al., 2010b). MCTS however allows for much more fine-grained time-management strategies due to its *anytime* property. It can be stopped after every rollout and return a move choice that makes use of the complete search time so far, while $\alpha\beta$ searchers can only make use of completely explored root moves of a deepening iteration.

This chapter answers the third research question by investigating and comparing a variety of time-management strategies for MCTS. We include newly proposed strategies as well as strategies described in Huang et al. (2010b) or independently proposed in Baudiš (2011), partly in enhanced form. These strategies are tested in the domains of 13×13 and 19×19 Go, and as far as possible in Connect-4, Breakthrough, Othello, and Catch the Lion.

The structure of this chapter is as follows. Section 6.1 gives an overview of related work on time management for game-playing programs. Section 6.2 outlines the approaches to time management studied in this work—both domain-independent techniques and techniques specific to Go. Section 6.3 presents experimental results of all strategies in Go, while Section 6.4 gives the results of testing and analyzing the domain-independent strategies in the games of Connect-4, Breakthrough, Othello, and Catch the Lion. Conclusions and future research follow in Section 6.5.

6.1 Time Management

The first publication to address the topic of time management in computer games was Hyatt (1984). He observed that human Chess grandmasters do not use an equal amount of time per move, but play standard openings quickly, think longest directly after coming out of the opening, and then play increasingly fast towards the end of the game. He also suggested a technique that lets $\alpha\beta$ search explore a position longer to find a better move if the best move of the last deepening iteration turns out to lose material.

Donninger (1994) gave four "golden rules" for the use of time during a Chess game, both for human and computer players: "a) Do not waste time in easy positions with only one obvious move. b) Use the opponent's thinking time effectively. c) Spend considerable time before playing a crucial move. d) Try to upset the opponent's timing." He considered rule c) to be the most important one by far, but also the hardest. In this chapter, we try to approach rules a) and c) simultaneously by attempting to estimate the importance or difficulty of a position and adjusting search time accordingly. Rule b) can be addressed by MCTS engines with *pondering*, thinking during the opponent's turn, which allows to transfer a part of the search tree into the next move search. If the opponent makes an expected move, the relevant part of the search tree can be large enough to make a move without much further thinking on your own time, which takes away the opponent's opportunity to ponder (an example for rule d). Pondering is not considered in this chapter.

Althöfer et al. (1994) published the first systematic evaluation of time-management algorithms for Chess. Amongst others, strategies were proposed to identify trivial moves that can be made quickly, as well as troublesome positions that require more

thinking. The time controls considered, typical for Chess, specify a given amount of time for a given number of moves. They are insofar different from sudden death as used in this chapter as it here does not refer to the number of moves by the player, but only to the total amount of time per game.

Markovitch and Sella (1996) used the domain of Checkers to automatically acquire a simple time-allocation strategy, distributing a fixed number of deep searches among the moves of a game. The authors divided time-management strategies into three categories. (1) *Static* strategies decide about time allocation to all future moves before the start of the game. (2) *Semi-dynamic* strategies determine the computation time for each move before the start of the respective move search. (3) *Dynamic* strategies make "live" timing decisions while the search process is running. This categorization is used in the remainder of this chapter.

Šolak and Vučković (2009) devised and tested a number of time-management models for modern Chess engines. Their model M2a involved the idea of estimating the remaining number of moves, given the number of moves already played, from a database of master games. We use a similar approach as the basis for our time-management strategies (called EXP). In more sophisticated models, Šolak and Vučković developed definitions for the complexity of a position—based on the number of legal moves—and allocated time accordingly.

Kocsis et al. (2001) compared temporal difference learning and genetic algorithms for training a neural network to make semi-dynamic timing decisions in the game Lines of Action. The network could set the underlying $\alpha\beta$ program to one of three predefined search depths.

For the framework of MCTS, only two publications exist so far. Huang et al. (2010b) evaluated a number of both dynamic and semi-dynamic time-management heuristics for 19×19 Go, assuming sudden-death time controls. We implemented and optimized their heuristics as a baseline for our approaches. The ideas of the "unstable evaluation" heuristic (UNST) and the "think longer when behind" heuristic (BEHIND) were first described and tested in Huang et al. (2010b). UNST continues searching if after the regular search time, the most-visited move is not the highest-valued move as well. BEHIND searches longer when the player's win rate at the root is low. Enhanced versions are described under the names UNST-L and BEHIND-L in the next Section.

During the preparation of our experiments, Baudiš published brief descriptions of the dynamic and semi-dynamic time management of the state-of-the-art MCTS Go program Pachi (Baudiš, 2011). Variants similar to our "close second" heuristic (CLOSE) and a special case of our "early stop" heuristic (STOP_A) were here formulated independently. CLOSE searches longer if the best and second best move are too close to each other. STOP_A stops searching if the currently best move cannot change anymore in the rest of the planned search time. We evaluate these strategies and

propose generalized versions with the names CLOSE-L and STOP as well.

Independent from time management considerations, Huang et al. (2010a) proposed two pruning conditions for MCTS: the *absolute pruning condition* and the *relative pruning condition*. These techniques are related to the STOP strategy and are discussed in Section 6.2.2.

6.2 Time-Management Strategies

In this section, we describe first the semi-dynamic (6.2.1), and then the dynamic time-management strategies (6.2.2) for the MCTS framework which were investigated in this chapter.

6.2.1 Semi-Dynamic Strategies

The following five strategies determine the search time for each move *before* the search for this move is started. EXP, OPEN and MID are domain-independent strategies, while KAPPA-EXP and KAPPA-LM are specific to the game of Go.

EXP. The simple EXP strategy for time allocation, used as the basis of all further enhancements in this chapter, divides the remaining thinking time for the entire game ($t_{\text{remaining}}$) by the expected number of remaining moves for the player (m_{expected}) and uses the result as the search time for the next move (t_{nextmove}). The formula is as follows:

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.1}$$

m_{expected} can be estimated in various ways. Three heuristics are investigated in this chapter, two of them game-independent and one game-specific to the game of Go. The first game-independent heuristic (EXP-MOVES) estimates the number of remaining moves given the number of moves already played. An example would be an expectation of 71 remaining moves for the player at the first turn, or an expectation of 27 remaining moves at turn 90. The second game-independent heuristic (EXP-SIM) estimates the number of remaining moves given the length of simulated games in the preceding search. As an example, 63 more moves could be expected if the average simulated game in the search for the preceding move was 200 moves long, or an average simulation length of 60 moves could lead to an expectation of 28 more moves in the actual game. The third heuristic (EXP-STONES) is specific to Go and uses the number of stones on the board as an estimator of remaining game length. 40 stones on the board could be

mapped to an expected 50 more moves, while 140 stones on the board could map to an expectation of 12 more moves for the player. Other games may or may not provide other indicators. The parameters for all three heuristics, e.g. the precise mapping from played moves to remaining moves for EXP-MOVES, are determined from a set of 1000 games played in self-play.

OPEN. The OPEN strategy puts emphasis on the opening phase of the game. Formula 6.2 modifies the search time for every move in the game by multiplying it with a constant "opening factor" $f_{\text{opening}} > 1$.

$$t_{\text{nextmove}} = f_{\text{opening}} \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.2}$$

This results in more time per move being used in the beginning of the game than at the end. As opposed to the implicit assumption of Formula 6.1 that equal time resources should be allocated to every expected move, here it is assumed that the first moves of a game have greater influence on the final outcome than the last moves and thus deserve longer search times.

MID. Instead of moves in the opening phase, the MID strategy increases search times for moves in the middle game, which can be argued to have the highest decision complexity of all game phases (Huang et al., 2010b). For this purpose, the time as given by Formula 6.1 is increased by a percentage determined by a Gaussian function over the set of move numbers, using three parameters a, b, and c for height, position and width of the "bell curve". The formula for the bell curve is

$$f_{\text{Gaussian}}(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{6.3}$$

$$t_{\text{nextmove}} = (1 + f_{\text{Gaussian}}(\text{current move number})) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.4}$$

KAPPA-EXP. In Coulom (2009), the concept of *criticality* was suggested for Go—as some intersections on the board are more important for winning the game than others, these should be recognized as "critical" or "hot", and receive special attention or search effort. To identify critical points, statistics are collected during rollouts on which player owns which intersections at the end of each simulation, and on how strongly this ownership is correlated with winning the simulated game (Coulom, 2009; Pellegrino

et al., 2009). In the KAPPA-EXP strategy, we use a related concept for identifying not only critical intersections from the set of all intersections of a board, but also critical move choices from the set of all move choices in a game. A highly critical move choice is here understood as a choice that involves highly critical intersections. The KAPPA-EXP strategy distributes time proportional to the expected maximum point criticality given the current move number, as estimated from a database of 1000 games played by the program itself. The idea is that the maximum point criticality, taken over the set of all intersections I on the board, indicates how crucial the current move choice is. We chose Formula 6.5 to represent the criticality of an intersection i in move m of game g—the *kappa statistic*, a chance-corrected measure of agreement typically used to quantify inter-rater reliability (Cohen, 1960). Here, it is employed to quantify agreement between the variables "intersection i is owned by the player at the end of a rollout during m's move search" and "the player wins a rollout during m's move search".

$$\begin{aligned} \kappa_g^m(i) &= \frac{\text{agreement}^m_{\text{observed}} - \text{agreement}^m_{\text{expected}}}{1 - \text{agreement}^m_{\text{expected}}} \\ &= \frac{\frac{o^m_{\text{winner}}(i)}{n} - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})}{1 - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})} \end{aligned} \tag{6.5}$$

where n is the total number of rollouts, $o^m_{\text{winner}}(i)$ is the number of rollouts in which point i ends up being owned by the rollout winner, $o^m_{\text{white}}(i)$ and $o^m_{\text{black}}(i)$ are the numbers of rollouts in which point i ends up being owned by White and Black, respectively, and w^m_{white} and w^m_{black} are the numbers of rollouts won by White and Black, respectively. All numbers refer to the search for move m.

For application at move number m during a given game, the average maximum point criticality $\kappa^m_{\text{avg}} = \frac{1}{y} \sum_{g=1}^{y} \max_{i \in I} \kappa_g^m(i)$ is precomputed from a database of y games, linearly transformed using parameters for slope and intercept $s_{\kappa_{\text{avg}}}$ and $i_{\kappa_{\text{avg}}}$, and finally multiplied with the search time resulting in Formula 6.6.

$$t_{\text{nextmove}} = (\kappa^m_{\text{avg}} \cdot s_{\kappa_{\text{avg}}} + i_{\kappa_{\text{avg}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.6}$$

KAPPA-LM. Instead of using the expected criticality for the current move number as defined above, the KAPPA-LM strategy uses the observed criticality as computed during the search for the player's previous move in the game. This value $\kappa_{\text{lastmove}} = \max_{i \in I} \kappa^{m-2}_{\text{current game}}(i)$ is again linearly transformed using parameters $s_{\kappa_{\text{lastmove}}}$ and

$i_{\kappa_{\text{lastmove}}}$, and multiplied with the base search time. The formula is as follows:

$$t_{\text{nextmove}} = (\kappa_{\text{lastmove}} \cdot s_{\kappa_{\text{lastmove}}} + i_{\kappa_{\text{lastmove}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.7}$$

For both KAPPA-EXP and KAPPA-LM, lower and upper bounds for the κ factor ensure lower and upper bounds for the total search time even in extreme positions. The algorithms are not very sensitive to these parameters, but without them games can be lost (in particular in the online version KAPPA-LM) due to occasional searches receiving too much or almost no time (extreme f values).

6.2.2 Dynamic Strategies

The following five strategies make time-allocation decisions for a move search *while* the respective search process is being carried out. BEHIND, UNST, CLOSE, and STOP are domain-independent strategies, while KAPPA-CM is specific to the game of Go. Note that our implementations of CLOSE and STOP are only valid if MCTS plays the most-visited move after each move search, which is the case for all MCTS players in this chapter. Other approaches to CLOSE and STOP are imaginable if MCTS chooses for example the move with the highest estimated value.

BEHIND. As suggested in Huang et al. (2010b) as the "think longer when behind" heuristic, the BEHIND strategy prolongs the search if the player is falling behind. It triggers if after the regular search time—as computed by the semi-dynamic strategies described above—the win rate of the best move at the root is lower than a threshold v_{behind}. If this is the case, the search is continued for a time interval determined by multiplying the previously used search time with a factor f_{behind}. The rationale is that by using more time resources, the player could still find a way to turn the game around, while saving time for later moves is less important in a losing position. We have also modified this heuristic to check its condition for search continuation repeatedly in a loop. The maximum number of loops until the search is terminated is bound by a parameter l_{behind}. The single-check heuristic is called BEHIND, the multiple-check heuristic BEHIND-L (for "loop") in the following.

UNST. The UNST strategy, called "unstable evaluation" heuristic in Huang et al. (2010b), prolongs the search if after the regular search time the most-visited move at the root is not the highest-valued move as well. In this case, the search is continued for the previously used search time multiplied with a factor f_{unstable}. The idea is that by searching longer, the highest-valued move could soon become the most-visited and thus change the final move choice. Analogously to the BEHIND-L technique, UNST-L was

introduced as an enhancement of UNST that repeatedly checks its trigger condition in a loop. The parameter specifying the maximum number of loops is l_{unstable}.

CLOSE. The proposed CLOSE strategy prolongs the search if after the regular search time the most-visited move and the second-most-visited move at the root are "too close", defined by having a relative visit difference lower than a threshold d_{close}. A similar strategy was developed independently in Baudiš (2011). The search is then continued for the previously used search time multiplied with a factor f_{close}. Like the UNST strategy, CLOSE aims to identify difficult decisions that can make efficient use of an increase in search time. We propose two variants of this strategy. It can either be triggered only once (CLOSE) or repeatedly (CLOSE-L) after the regular search time is over. For CLOSE-L, a parameter l_{close} defines the maximum number of loops.

KAPPA-CM. Unlike the three dynamic strategies described above, the KAPPA-CM strategy does not wait for the regular search time to end. Instead, it uses the first e.g. 100 milliseconds of the search process to collect criticality data. Then it uses the maximum point criticality of the current move $\kappa_{\text{currentmove}} = \max_{i \in I} \kappa^m_{\text{current game}}(i)$ to modify the remaining search time. The formula is as follows:

$$t_{\text{currentmove}} = (\kappa_{\text{currentmove}} \cdot s_{\kappa_{\text{currentmove}}} + i_{\kappa_{\text{currentmove}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6.8}$$

The remaining search time can be either reduced or increased by this strategy. Upper and lower limits to the total search time apply.

STOP. The proposed "early stop" (STOP) strategy is based on the idea of terminating the search process as early as possible in case the best move cannot change anymore. For STOP, the search speed in simulations per second is measured, and in regular intervals (e.g. 50 rollouts) it is checked how many rollouts are still expected in the remainder of the total planned search time. If the number of simulations required for the second-most-visited move at the root to catch up to the most-visited one exceeds this expected number of remaining simulations, the search can safely be terminated without changing the final outcome.

However, not all of the remaining simulations in a search generally start with the second-most-visited move. Therefore, we introduce a parameter $p_{\text{earlystop}} \leq 1$ representing an estimate of the proportion of remaining rollouts that actually sample the second-most-visited move. The search is terminated if the number of rollouts needed for the second-most-visited move at the root to catch up to the most-visited one exceeds the expected number of remaining rollouts multiplied with $p_{\text{earlystop}}$. When setting this parameter to a value smaller than 1, an unchanged final outcome is no longer

guaranteed. Optimal values of $p_{\text{earlystop}}$ have to be determined empirically. The termination criterion of STOP is:

$$\frac{n \cdot \text{timeleft}_n}{\text{timespent}_n} \cdot p_{\text{earlystop}} < \text{visits}_{\text{best}_n} - \text{visits}_{\text{secondbest}_n} \tag{6.9}$$

where n is the number of rollouts so far, timeleft_n is the rest of the planned search time, timespent_n is the search time already spent, $\text{visits}_{\text{best}_n}$ is the currently highest number of visits of any move at the root, and $\text{visits}_{\text{secondbest}_n}$ is the currently second-highest number of visits of any move at the root. All numbers refer to the state of the search after n rollouts.

If the expected time savings by the STOP strategy are not considered when computing planned search times, savings will accumulate throughout the game and early moves cannot benefit from them. In order to achieve a different distribution of the resulting time savings among all searches in the game, planned search times are multiplied with a parameter $f_{\text{earlystop}} \geq 1$ that is also determined empirically.

In order to test the effects of the two parameters $p_{\text{earlystop}}$ and $f_{\text{earlystop}}$ independently of each other, we introduce the name STOP_B for the special case of STOP with $p_{\text{earlystop}} = 1$ and free parameter $f_{\text{earlystop}}$. This variant can redistribute search time, but never stops a search before the final outcome is definitely known (it uses "safe" stopping). If $p_{\text{earlystop}} = 1$ and $f_{\text{earlystop}} = 1$ (stopping is "safe", and the redistribution of search time is deactivated as well), STOP is identical to a strategy mentioned independently—but not evaluated—in Baudiš (2011). In the following, we call this special case STOP_A.

The *absolute pruning condition* proposed by Huang *et al.* in Huang et al. (2010a) can be seen as a weaker form of STOP_A, only stopping if one move has more than half the simulations planned for the entire search. This does not take the visit difference between the most-visited and second-most-visited move into account. The authors also proposed an "unsafe" criterion for pruning: Their *relative pruning condition* excludes individual moves from the search as soon as they are not expected to catch up with another move anymore. This expectation is based on a formula for the upper bound of the remaining simulations for a given move. However, the authors state that their condition is strict and rarely triggers, making a relaxed condition desirable. STOP allows to relax its stopping condition through the $p_{\text{earlystop}}$ parameter.

6.3 Experimental Results in Go

All time-management strategies were implemented in Orego (Drake et al., 2011) version 7.08. Orego is a Go program using a number of MCTS enhancements like a transposition table (Childs et al., 2008; Greenblatt et al., 1967), RAVE (Gelly and Silver, 2007), a rollout policy similar to that proposed in Gelly et al. (2006), and LGRF-2 (Baier and Drake, 2010). The rollout policy takes domain-specific knowledge into account by trying to save groups under attack (*atari*), attacking opponent groups, and giving preference to moves that match a prespecified set of 3×3 intersection patterns on the board. After each search, the most-sampled move at the root is played. Orego resigns if its win rate at the root falls below 10%. The program ran on a CentOS Linux server consisting of four AMD Twelve-Core OpteronT 6174 processors (2.2 GHz). Unless specified otherwise, each experimental run involved 5000 games (2500 as Black and 2500 as White) of Orego against the classic (non-MCTS-based) program GNU Go 3.8 (Free Software Foundation, 2009), played on the 13×13 board, using Chinese rules (area scoring), positional superko, and 7.5 komi. The playing strength of a non-MCTS program like GNU Go is difficult to measure since it has weaknesses that can be relatively easily exploited by human players, but it is estimated to play at amateur level (8-12 kyu on the 19×19 board to 5-7 kyu on the 9×9 board). Orego plays at around 5-6 kyu with 30 minutes per game on the 19×19 board, and probably stronger on the 13×13 board. For testing against GNU Go, Orego's strength was reduced here by lowering the time to 30 seconds per game unless specified otherwise. GNU Go ran at its default level of 10, with the capture-all-dead option turned on. It had no time limit. Orego used a single thread and no pondering. The speed of Orego was about 1850 simulations per second when searching from the initial position. Optimal parameter settings for the time management strategies were found by manually testing a wide range of parameter values, from around 10-20 for strategies with a single parameter to hundreds of settings for strategies with three parameters, with 500 or 1000 games each against GNU Go.

The remainder of this section is structured as follows. In 6.3.1, the strategies in Huang et al. (2010b) are tested as a baseline. Next, 6.3.2 presents results of experiments with semi-dynamic strategies. Dynamic strategies are tested in 6.3.3. Finally, in 6.3.4 the best-performing strategy is compared to the baseline in self-play, as well as to Orego with fixed time per move.

6.3.1 ERICA-BASELINE

In order to compare the results to a state-of-the-art baseline, the strategies described in Huang et al. (2010b) were implemented and evaluated. The thinking time per move

Table 6.1: Performance of Erica's time management according to Huang et al. (2010b) in 13×13 Go.

Player	Win rate against GNU Go	95% conf. int.
Basic formula	28.6%	27.3%–29.9%
Enhanced formula	31.4%	30.1%–32.7%
ERICA-BASELINE	35.3%	34.0%–36.7%

was computed according to the "basic formula"

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{C} \tag{6.10}$$

where $C = 30$ was found to be optimal for Orego, as well as the "enhanced formula"

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{C + \max(\text{MaxPly} - \text{MoveNumber}, 0)} \tag{6.11}$$

with $C = 20$ and MaxPly = 40. The UNST heuristic, using a single loop as proposed in Huang et al. (2010b), worked best with $f_{\text{unstable}} = 0.5$. The BEHIND heuristic was most successful in Orego with $v_{\text{behind}} = 0.6$ and $f_{\text{behind}} = 0.75$. Note that BEHIND had not been found to be effective at first in Baier and Winands (2012). This is because only win rate thresholds up to 0.5 had been tested originally—a player with a win rate of more than 0.5 cannot be called "behind" after all. See subsection 6.4.5 for a discussion of the effect of higher thresholds.

Erica's time-management strategies were tested against GNU Go using the basic formula, using the enhanced formula, and using the enhanced formula plus UNST and BEHIND heuristic (called ERICA-BASELINE from now on). Table 6.1 presents the results—the enhanced formula is significantly stronger than the basic formula ($p<0.01$), and ERICA-BASELINE is significantly stronger than the enhanced formula ($p<0.001$).

6.3.2 Semi-Dynamic Strategies

This subsection presents the results for the EXP, OPEN, MID, KAPPA-EXP, and KAPPA-LM strategies in Go.

EXP-MOVES, EXP-SIM and EXP-STONES. As our basic time-management approach, EXP-MOVES, EXP-SIM and EXP-STONES were tested. The first three rows

Table 6.2: Performance of the investigated semi-dynamic strategies in 13×13 Go.

Player	Win rate against GNU Go	95% conf. int.
EXP-MOVES	24.0%	22.9%–25.2%
EXP-SIM	23.5%	22.3%–24.7%
EXP-STONES	25.5%	24.3%–26.7%
EXP-STONES with OPEN	32.0%	30.8%–33.4%
EXP-STONES with MID	30.6%	29.3%–31.9%
EXP-STONES with KAPPA-EXP	31.7%	30.5%–33.0%
EXP-STONES with KAPPA-LM	31.1%	29.8%–32.4%
ERICA-BASELINE	35.3%	34.0%–36.7%

of Table 6.2 show the results. As EXP-STONES appeared to perform best, it was used as the basis for all further experiments with the game of Go. Note however that the differences were not statistically significant. The average error in predicting the remaining number of moves was 14.16 for EXP-STONES, 13.95 for EXP-MOVES, and 14.23 for EXP-SIM.

OPEN. According to preliminary experiments with OPEN, the "opening factor" $f_{\text{opening}} = 2.5$ seemed most promising. It was subsequently tested with 5000 additional games against GNU Go. Table 6.2 shows the result: EXP-STONES with OPEN is significantly stronger than plain EXP-STONES ($p<0.001$).

MID. Initial experiments with MID showed Formula 6.3 to perform best with $a = 2$, $b = 40$ and $c = 20$. It was then tested with 5000 additional games. As Table 6.2 reveals, EXP-STONES with MID is significantly stronger than plain EXP-STONES ($p<0.001$).

KAPPA-EXP. The best parameter setting for KAPPA-EXP found in preliminary experiments was $s_{\kappa_{\text{avg}}} = 8.33$ and $i_{\kappa_{\text{avg}}} = -0.67$. Lower and upper bounds for the *kappa* factor were set to 0.5 and 10, respectively. Table 6.2 presents the result of testing this setting. EXP-STONES with KAPPA-EXP is significantly stronger than plain EXP-STONES ($p<0.001$).

KAPPA-LM. Here, $s_{\kappa_{\text{lastmove}}} = 8.33$ and $i_{\kappa_{\text{lastmove}}} = -0.67$ were chosen for further testing against GNU Go as well. Lower and upper bounds for the *kappa* factor were set to 0.25 and 10. The test result is shown in Table 6.2. EXP-STONES with KAPPA-LM is significantly stronger than plain EXP-STONES ($p<0.001$).

6.3.3 Dynamic Strategies

In this subsection the results for the BEHIND, UNST, CLOSE, STOP, and KAPPA-CM strategies in the game of Go are given.

BEHIND. Just like the "enhanced formula" of Huang et al. (2010b), EXP-STONES was found to be significantly improved by BEHIND in OREGO only with a threshold v_{behind} of higher than 0.5. This is also true after the introduction of BEHIND-L, and will be discussed in subsection 6.4.5. The best parameter settings in preliminary experiments were $f_{\text{behind}} = 0.5$ and $v_{\text{behind}} = 0.6$ for BEHIND, and $f_{\text{behind}} = 0.25$, $v_{\text{behind}} = 0.6$, and $l_{\text{behind}} = 2$ for BEHIND-L. Detailed results of an additional 5000 games with these settings are given in Table 6.3. EXP-STONES with BEHIND is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with BEHIND-L, however, could not be shown to be significantly stronger than EXP-STONES with BEHIND.

UNST. The best results in initial experiments with UNST were achieved with $f_{\text{unstable}} = 1.5$. For UNST-L, $f_{\text{unstable}} = 0.75$ and $l_{\text{unstable}} = 2$ turned out to be promising values. These settings were tested in 5000 further games. Table 6.3 shows the results. EXP-STONES with UNST is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with UNST-L, in turn, is significantly stronger than EXP-STONES with UNST ($p<0.05$).

CLOSE. The best-performing parameter settings in initial experiments with CLOSE were $f_{\text{close}} = 1.5$ and $d_{\text{close}} = 0.4$. When we introduced CLOSE-L, $f_{\text{close}} = 0.5$, $d_{\text{close}} = 0.5$ and $l_{\text{close}} = 4$ appeared to be most successful. Table 6.3 presents the results of testing both variants in 5000 more games. EXP-STONES with CLOSE is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with CLOSE-L, in turn, is significantly stronger than EXP-STONES with CLOSE ($p<0.001$).

KAPPA-CM. The best parameter setting for KAPPA-CM found in preliminary experiments was $s_{\kappa_{\text{currentmove}}} = 8.33$ and $i_{\kappa_{\text{currentmove}}} = -1.33$. Lower and upper bounds for the *kappa* factor were set to 0.6 and 10. Table 6.3 reveals the result of testing this setting. EXP-STONES with KAPPA-CM is significantly stronger than plain EXP-STONES ($p<0.05$). However, it is surprisingly weaker than both EXP-STONES using KAPPA-EXP and EXP-STONES with KAPPA-LM ($p<0.001$). The time of 100 msec used to collect current criticality information might be too short, such that noise is too high. Preliminary tests with longer collection times (keeping the other parameters equal) did not show significantly better results. A retuning of the other parameters with longer collection times remains as future work.

STOP. The best settings found for STOP were $f_{\text{earlystop}} = 2.5$ and $p_{\text{earlystop}} = 0.4$. This variant significantly outperformed ($p<0.001$) plain EXP-STONES as well as ERICA-

Table 6.3: Performance of the investigated dynamic strategies in 13×13 Go.

Player	Win rate against GNU Go	95% conf. int.
EXP-STONES with BEHIND	29.9%	28.7%–31.2%
EXP-STONES with BEHIND-L	30.5%	29.2%–31.8%
EXP-STONES with UNST	33.6%	32.3%–34.9%
EXP-STONES with UNST-L	35.8%	34.4%–37.1%
EXP-STONES with CLOSE	32.6%	31.3%–33.9%
EXP-STONES with CLOSE-L	36.5%	35.2%–37.9%
EXP-STONES with KAPPA-CM	27.3%	26.1%–28.6%
EXP-STONES with STOP_A	25.3%	24.1%–26.5%
EXP-STONES with STOP_B	36.7%	35.4%–38.0%
EXP-STONES with STOP	**39.1%**	**38.0%–40.8%**
EXP-STONES	25.5%	24.3%–26.7%
ERICA-BASELINE	35.3%	34.0%–36.7%

BASELINE in 5000 games each against GNU Go. It is also significantly stronger ($p<0.01$) than the best-performing setting of STOP_B with $f_{\text{earlystop}} = 2$. STOP_B in turn is significantly stronger than plain EXP-STONES as well as STOP_A ($p<0.001$). The STOP_A strategy did not show a significant improvement to the EXP-STONES baseline. Table 6.3 presents the results.

6.3.4 Strength Comparisons

This subsection focuses on the time-management strategy that proved most successful in Go, the STOP strategy. It attempts to answer the question whether STOP's effectiveness generalizes to longer search times (60 seconds, 120 seconds per game) and to the larger 19×19 board size. Furthermore, an indication is given of how strong the effect of time management is as compared to fixed time per move.

Comparison with ERICA-BASELINE on 13×13. Our strongest time-management strategy on the 13×13 board, EXP-STONES with STOP, was tested in self-play against Orego with ERICA-BASELINE. Time settings of 30, 60, and 120 seconds per game were used with 2000 games per data point. Table 6.4 presents the results. For all time settings, EXP-STONES with STOP was significantly stronger ($p<0.001$).

Comparison with ERICA-BASELINE on 19×19. In this experiment, we pitted

Table 6.4: Performance of EXP-STONES with STOP vs. ERICA-BASELINE in 13×13 Go.

Time setting	Win rate against ERICA-BASELINE	95% conf. int.
30 sec sudden death	63.7%	61.5%–65.8%
60 sec sudden death	59.4%	57.2%–61.5%
120 sec sudden death	60.7%	58.5%–62.8%

Table 6.5: Performance of EXP-STONES with STOP vs. ERICA-BASELINE in 19×19 Go.

Time setting	Win rate against ERICA-BASELINE	95% conf. int.
300 sec sudden death	62.7%	60.6%–64.9%
900 sec sudden death	60.2%	58.0%–62.4%

EXP-STONES with STOP against ERICA-BASELINE on the 19×19 board. The best parameter settings found were $C = 60$, MaxPly = 110 and $f_{\text{unstable}} = 1$ for ERICA-BASELINE, and $f_{\text{earlystop}} = 2.2$ and $p_{\text{earlystop}} = 0.45$ for STOP. Time settings of 300 and 900 seconds per game were used with 2000 games per data point. Orego played about 960 simulations per second when searching from the empty 19×19 board. The results are shown in Table 6.5—for both time settings, EXP-STONES with STOP was significantly stronger ($p<0.001$).

Comparison with fixed time per move. To illustrate the effect of successful time management, two additional experiments were conducted with Orego using fixed time per move in 13×13 Go. In the first experiment, the time per move (650 msec) was set so that approximately the same win rate against GNU Go was achieved as with EXP-STONES and STOP at 30 seconds per game. The result of 2500 games demonstrated that the average time needed per game was 49.0 seconds—63% more than needed by our time-management strategy. In the second experiment, the time per move (425 msec) was set so that the average time per game was approximately equal to 30 seconds. In 2500 games under these conditions, Orego could only achieve a 27.6% win rate, 11.5% less than with EXP-STONES and STOP.

6.4 Experimental Results in Other Domains

The goal of this section is to investigate the generality of the domain-independent strategies described above: the semi-dynamic strategies OPEN and MID, and the dynamic strategies BEHIND, UNST, CLOSE, and STOP. All time-management strategies were therefore tested in Connect-4, Breakthrough, Othello, and Catch the

Lion. Unless specified otherwise, each experimental run again involved 5000 games (2500 as Black and 2500 as White) against the baseline player.

We used our own engine with EXP-MOVES as the baseline. EXP-MOVES was chosen because EXP-STONES is domain-specific to Go, and EXP-SIM could be problematic in domains such as Catch the Lion which do not naturally progress in every move towards the end of the game (*progression property*, Finnsson and Björnsson 2011). ERICA-BASELINE was not used as a baseline in other domains than Go since it was proposed specifically for Go and not as a domain-independent technique. The engine uses MCTS with UCB1-TUNED (Auer et al., 2002) as selection policy and uniformly random rollouts in all conditions. The exploration factor C of UCB1-TUNED was optimized for each domain at time controls of 1 second per move and set to 1.3 in Connect-4, 0.8 in Breakthrough, 0.7 in Othello, and 0.7 in Catch the Lion. After each search, the most-sampled move at the root is played. Draws, which are possible in Connect-4 and Othello, were counted as half a win for both players.

6.4.1 Connect-4

In Connect-4, a time limit of 20 seconds per player and game was used. A draw was counted as half a win for both players. The baseline player's speed was about 64500 simulations per second when searching from the initial position. Table 6.6 shows the results of all investigated time-management strategies in Connect-4. Each strategy is listed together with the parameter setting that was found to perform best in initial systematic testing. The win rate given in the table was found by using this parameter setting in an additional 5000 games against the baseline.

As Table 6.6 reveals, EXP-MOVES enhanced with the OPEN, MID, BEHIND, UNST, or CLOSE strategies played significantly stronger ($p<0.001$) than plain EXP-MOVES. BEHIND-L, UNST-L, and CLOSE-L could not be shown to significantly improve on BEHIND, UNST, and CLOSE, respectively. STOP improved significantly on EXP-MOVES alone ($p<0.001$) as well as STOP_B ($p<0.05$). EXP-MOVES with STOP_B was still significantly stronger than EXP-MOVES alone ($p<0.001$). As in Go, STOP_A did not have a significant effect.

6.4.2 Breakthrough

In Breakthrough, a time limit of 20 seconds per player and game was used. Searching from the initial board position, the baseline player reached about 24800 simulations per second. Table 6.7 displays the results of all investigated time-management strategies in Breakthrough.

As Table 6.7 shows, EXP-MOVES enhanced with the UNST or CLOSE strategies played significantly stronger ($p<0.001$ and $p<0.05$, respectively) than regular EXP-

Table 6.6: Performance of the investigated time-management strategies in Connect-4.

Player	Win rate against EXP-MOVES	95% conf. int.
EXP-MOVES with OPEN $f_{\text{opening}} = 2.25$	55.8%	54.4%–57.1%
EXP-MOVES with MID $a = 2.5$, $b = 30$, $c = 20$	57.0%	55.6%–58.4%
EXP-MOVES with BEHIND $f_{\text{behind}} = 1.5$, $v_{\text{behind}} = 0.6$	55.8%	54.4%–57.1%
EXP-MOVES with BEHIND-L $f_{\text{behind}} = 0.75$, $v_{\text{behind}} = 0.6$, $l_{\text{behind}} = 3$	57.0%	55.6%–58.4%
EXP-MOVES with UNST $f_{\text{unstable}} = 0.75$	54.9%	53.6%–56.3%
EXP-MOVES with UNST-L $f_{\text{unstable}} = 1.0$, $l_{\text{unstable}} = 2$	55.8%	54.4%–57.2%
EXP-MOVES with CLOSE $f_{\text{close}} = 1.5$, $d_{\text{close}} = 0.8$	58.7%	57.3%–60.0%
EXP-MOVES with CLOSE-L $f_{\text{close}} = 0.75$, $d_{\text{close}} = 0.8$, $l_{\text{close}} = 3$	59.7%	58.3%–61.1%
EXP-MOVES with STOP_{A}	50.8%	49.4%–52.2%
EXP-MOVES with STOP_{B} $f_{\text{earlystop}} = 5$	63.0%	61.6%–64.3%
EXP-MOVES with STOP $\mathbf{f_{earlystop} = 5}$, $\mathbf{p_{earlystop} = 0.9}$	**65.0%**	**63.7%–66.3%**

MOVES. Neither OPEN nor MID or BEHIND had a significant effect. BEHIND-L, UNST-L, and CLOSE-L could also not be shown to significantly improve on BEHIND, UNST, and CLOSE. STOP played significantly stronger than the baseline ($p<0.001$). It could not be shown to improve on STOP_B however. STOP ($p<0.001$) as well as STOP_B ($p<0.05$) improved significantly on STOP_A. In contrast to Go and Connect-4, STOP_A already played significantly stronger than the baseline ($p<0.001$).

6.4.3 Othello

In Othello, a time limit of 30 seconds per player and game was used. The longer time setting partly compensates for the longer average game length of Othello compared to Connect-4, Breakthrough, and Catch the Lion. The baseline player's speed was about 4400 simulations per second during a search from the initial position. Table 6.8 shows the results of all investigated time-management strategies in Othello.

Table 6.8 reveals that EXP-MOVES enhanced with the UNST or MID strategies played significantly stronger ($p<0.05$) than the EXP-MOVES baseline. Neither OPEN nor CLOSE or BEHIND had a significant effect. BEHIND-L, UNST-L, and CLOSE-L could again not be shown to significantly improve on BEHIND, UNST, and CLOSE. STOP played significantly stronger than the baseline ($p<0.05$). Similar to Breakthrough, this was already true for STOP_A. The more general variants of STOP could not be demonstrated to significantly improve on STOP_A in Othello.

6.4.4 Catch the Lion

In Catch the Lion, a time limit of 20 seconds per player and game was used. From the initial board, the baseline had a speed of about 18500 simulations per second. Table 6.9 shows the results of all investigated time-management strategies in Catch the Lion.

As Table 6.9 shows, EXP-MOVES enhanced with the UNST or CLOSE strategies played significantly stronger ($p<0.001$) than regular EXP-MOVES. Neither OPEN, MID, nor BEHIND had a significant effect. BEHIND-L, UNST-L, and CLOSE-L could not be shown to significantly improve on their simpler versions BEHIND, UNST, and CLOSE. STOP was significantly stronger ($p<0.001$) than both the EXP-MOVES baseline and STOP_B. STOP_B improved on STOP_A ($p<0.001$). As in Connect-4, STOP_A did not have a significant advantage over the baseline.

6.4.5 Discussion of the Results

In the previous subsections, experimental data were ordered by domain. This subsection summarizes and discusses the results ordered by the type of time-management technique instead, in order to allow for comparisons across different domains. Tables 6.10 and

Table 6.7: Performance of the investigated time-management strategies in Breakthrough.

Player	Win rate against EXP-MOVES	95% conf. int.
EXP-MOVES with OPEN $f_{\text{opening}} = 1$	50.6%	49.2%–52.0%
EXP-MOVES with MID $a = 1.5$, $b = 50$, $c = 20$	49.7%	48.3%–51.1%
EXP-MOVES with BEHIND $f_{\text{behind}} = 1$, $v_{\text{behind}} = 0.5$	50.2%	48.8%–51.6%
EXP-MOVES with BEHIND-L $f_{\text{behind}} = 0.25$, $v_{\text{behind}} = 0.5$, $l_{\text{behind}} = 3$	50.9%	49.5%–52.3%
EXP-MOVES with UNST $f_{\text{unstable}} = 0.75$	54.2%	52.8%–55.6%
EXP-MOVES with UNST-L $f_{\text{unstable}} = 1.0$, $l_{\text{unstable}} = 3$	55.2%	53.8%–56.6%
EXP-MOVES with CLOSE $f_{\text{close}} = 0.5$, $d_{\text{close}} = 0.3$	53.1%	51.7%–54.5%
EXP-MOVES with CLOSE-L $f_{\text{close}} = 0.5$, $d_{\text{close}} = 0.3$, $l_{\text{close}} = 2$	53.9%	52.5%–55.3%
EXP-MOVES with STOP_{A}	54.5%	53.1%–55.9%
EXP-MOVES with STOP_{B} $f_{\text{earlystop}} = 1.25$	56.9%	55.5%–58.2%
EXP-MOVES with STOP $\mathbf{f_{earlystop} = 1.67}$**,** $\mathbf{p_{earlystop} = 0.3}$	**58.2%**	**56.8%–59.5%**

Table 6.8: Performance of the investigated time-management strategies in Othello.

Player	Win rate against EXP-MOVES	95% conf. int.
EXP-MOVES with OPEN $f_{\text{opening}} = 1$	50.1%	48.7%–51.5%
EXP-MOVES with MID $a = 2.5$, $b = 50$, $c = 10$	53.2%	51.8%–54.6%
EXP-MOVES with BEHIND $f_{\text{behind}} = 0.5$, $v_{\text{behind}} = 0.5$	49.2%	47.9%–50.6%
EXP-MOVES with BEHIND-L $f_{\text{behind}} = 0.5$, $v_{\text{behind}} = 0.4$, $l_{\text{behind}} = 4$	51.3%	49.9%–52.7%
EXP-MOVES with UNST $f_{\text{unstable}} = 0.5$	53.1%	51.7%–54.5%
EXP-MOVES with UNST-L $f_{\text{unstable}} = 0.5$, $l_{\text{unstable}} = 4$	52.2%	50.8%–53.6%
EXP-MOVES with CLOSE $f_{\text{close}} = 1.0$, $d_{\text{close}} = 0.2$	50.0%	48.6%–51.4%
EXP-MOVES with CLOSE-L $f_{\text{close}} = 0.25$, $d_{\text{close}} = 0.5$, $l_{\text{close}} = 4$	51.4%	50.0%–52.8%
EXP-MOVES with STOP_{A}	52.9%	51.5%–54.3%
EXP-MOVES with STOP_{B} $f_{\text{earlystop}} = 1.25$	54.2%	52.8%–55.6%
EXP-MOVES with STOP $f_{\textbf{earlystop}} = 1.25$, $p_{\textbf{earlystop}} = 0.9$	**54.8%**	**53.4%–56.2%**

Table 6.9: Performance of the investigated time-management strategies in Catch the Lion.

Player	Win rate against EXP-MOVES	95% conf. int.
EXP-MOVES with OPEN $f_{\text{opening}} = 1.5$	50.7%	49.3%–52.1%
EXP-MOVES with MID $a = 1.0$, $b = 20$, $c = 10$	51.2%	49.8%–52.6%
EXP-MOVES with BEHIND $f_{\text{behind}} = 0.75$, $v_{\text{behind}} = 0.4$	51.6%	50.2%–53.0%
EXP-MOVES with BEHIND-L $f_{\text{behind}} = 0.25$, $v_{\text{behind}} = 0.3$, $l_{\text{behind}} = 3$	50.5%	49.0%–51.9%
EXP-MOVES with UNST $f_{\text{unstable}} = 0.5$	56.4%	55.0%–57.8%
EXP-MOVES with UNST-L $f_{\text{unstable}} = 0.75$, $l_{\text{unstable}} = 2$	56.0%	54.6%–57.4%
EXP-MOVES with CLOSE $f_{\text{close}} = 1.0$, $d_{\text{close}} = 0.7$	57.4%	56.0%–58.8%
EXP-MOVES with CLOSE-L $f_{\text{close}} = 0.5$, $d_{\text{close}} = 0.7$, $l_{\text{close}} = 3$	55.9%	54.5%–57.2%
EXP-MOVES with STOP_{A}	50.0%	48.6%–51.4%
EXP-MOVES with STOP_{B} $f_{\text{earlystop}} = 1.67$	56.9%	55.5%–58.2%
EXP-MOVES with STOP $\boldsymbol{f_{\text{earlystop}} = 5}$**,** $\boldsymbol{p_{\text{earlystop}} = 0.2}$	**60.4%**	**59.1%–61.8%**

Table 6.10: Time management summary – simple strategies. Checkmarks identify strategies that were shown to significantly improve on their respective baseline (EXP-MOVES in the column game). Strategies that showed no significant improvement are left blank.

EXP-MOVES with	Connect-4	Breakthrough	Othello	Catch the Lion	13×13 Go
OPEN	✓				✓
MID	✓		✓		✓
BEHIND	✓				✓
UNST	✓	✓	✓	✓	✓
CLOSE	✓	✓		✓	✓
STOP_A		✓	✓		
STOP_B	✓	✓	✓	✓	✓
STOP	✓	✓	✓	✓	✓

Table 6.11: Time management summary – loop strategies. Checkmarks identify loop strategies that were shown to significantly improve on their respective simple versions (e.g. CLOSE-L on CLOSE). Strategies that showed no significant improvement are left blank.

EXP-MOVES with	Connect-4	Breakthrough	Othello	Catch the Lion	13×13 Go
BEHIND-L					
UNST-L					✓
CLOSE-L					✓

6.11 give an overview by recapitulating which strategies were shown to result in a significant increase of playing strength in which game. Table 6.10 illustrates the improvements of *simple strategies*—the ones not involving repeated checks in loops, such as UNST—compared to the baseline player. Table 6.11 shows the improvements of the *loop strategies* compared to their simple counterparts, such as UNST-L to UNST. In the following, these results are discussed ordered by time-management strategy.

OPEN and MID. Some domains, e.g. Connect-4, Othello, and Go, profit from shifting available time from the endgame to the midgame or early game. Intuitively, this is the case in games that allow players to build up a positional advantage, essentially deciding the game result many moves before an actual terminal state is reached. Less effort is therefore required in the endgame for the leading player to keep the lead and

execute the win. Search effort is also largely futile for a player who has fallen behind in the endgame, and is more effectively spent in the early or midgame to avoid falling behind in the first place.

Other games, for example Catch the Lion and Breakthrough, are more tactical in nature and require a higher search effort even in the endgame. One of the reasons is that with sufficient effort, it can be possible throughout the entire game for both players to lure their opponent into *traps* (Ramanujan et al., 2010a). The player falling behind can therefore still make effective use of additional time in the endgame, while the leading player still needs to spend time to avoid losing her positional gains in this way. See Chapter 7 for a discussion of tacticality and traps and a more in-depth comparison of the test domains in this respect.

The following experiment was conducted in order to compare the test domains with respect to the usefulness of spending search time in the endgame. Player A used 1 second thinking time per move. Player B used 1 second until 10 turns before the average game length of the respective domain, and then switched to 100ms per move. 1000 games were played in each domain, with players A and B both moving first in half of the games. In this setting, player B won 49.1% (95% confidence interval: $46.0\% - 52.3\%$) of games in Connect-4 and 49.0% ($45.9\% - 52.2\%$) in Othello—the loss of time in the endgame could not be shown to significantly weaken the player. In Breakthrough however, player B won only 36.6% ($33.6\% - 39.7\%$) of games, and in Catch the Lion only 31.4% ($28.6\% - 34.4\%$). This explains why shifting large amounts of time from the endgame to the opening or midgame is not effective in these domains.

In Go, switching from 1000ms to 100ms in the described way does result in decreased performance as well (38.4% win rate for player B). Even though endgame time does not seem to be wasted time in Go however, moving a part of it towards the opening or midgame is still effective. Note that Orego resigns whenever its win rate at the root falls below 10%, which cuts off most of the late endgame of Go as human players typically would. This feature is meant to restrict the game to the moves that actually matter. Deactivating resigning makes games much longer on average (231 moves instead of 118 in self-play at 1 second per move), and creates even more opportunity to shift time away from the endgame.

Furthermore, it is interesting to note that the optimal parameter settings of the MID strategy in Connect-4 largely turn it into a variant of OPEN by shifting time far to the beginning of the game. In for example Othello and 13×13 Go this is not the case. Figures 6.1, 6.2, and 6.3 show the average time distribution over a game of Connect-4, Othello, and Go, respectively, using the optimized settings of the MID strategy in each game. The figures demonstrate how the peak of the average time spent by MID per move appears at a later stage of the game in Othello and Go than

in Connect-4. This is probably explained by the fact that Othello and Go games take much longer than Connect-4 games on average (compare Table 6.14). In Othello and Go it is therefore prudent not to spend too much time on the first moves of a game, as potential consequences of actions are not yet visible to MCTS. Connect-4 however requires more search effort in the opening, as games are often decided early. One could say that opening and midgame fall into one phase in a short game such as Connect-4.

In conclusion, OPEN and MID can be useful not only to improve performance in a given game, but also to gain insight into the relative importance of early game, midgame and endgame decisions in the game at hand. Note that this importance is always dependent on the search engine and search timing used—an engine with opening or endgame databases for example might optimally distribute search time differently, and the use of long search times can make the consequences of moves visible somewhat earlier in the game compared to short search times.

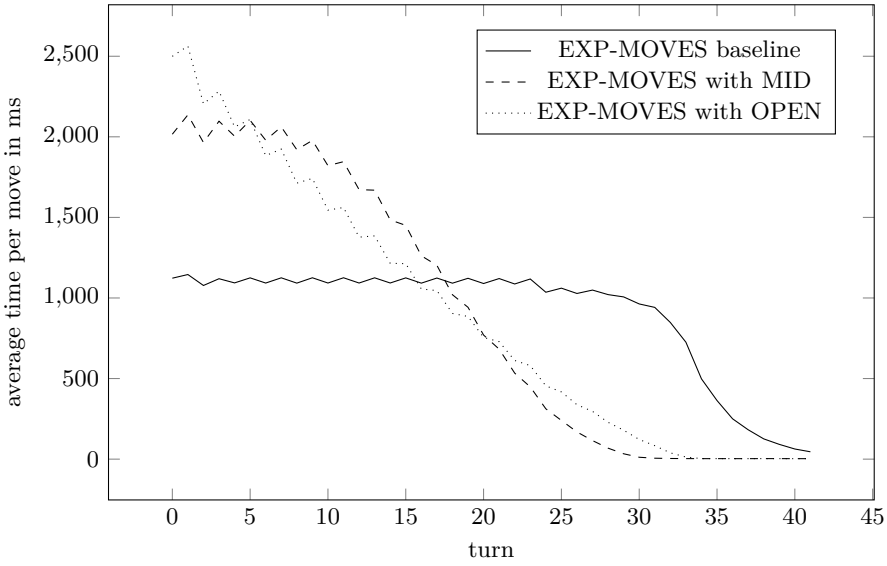


Figure 6.1: Average time distribution over a game of Connect-4 with the MID and OPEN strategies.

BEHIND. Among the tested domains, the BEHIND strategy is improving performance in Connect-4 and 13×13 Go. Looking at the optimal parameter values for these two games however, we can see that search times are prolonged whenever the player's win rate falls below 0.6. This means they are essentially always prolonged in the opening and midgame, when win rates tend to be close to 0.5. The strategy largely turns from a method to come back from a disadvantageous position into a method to shift more

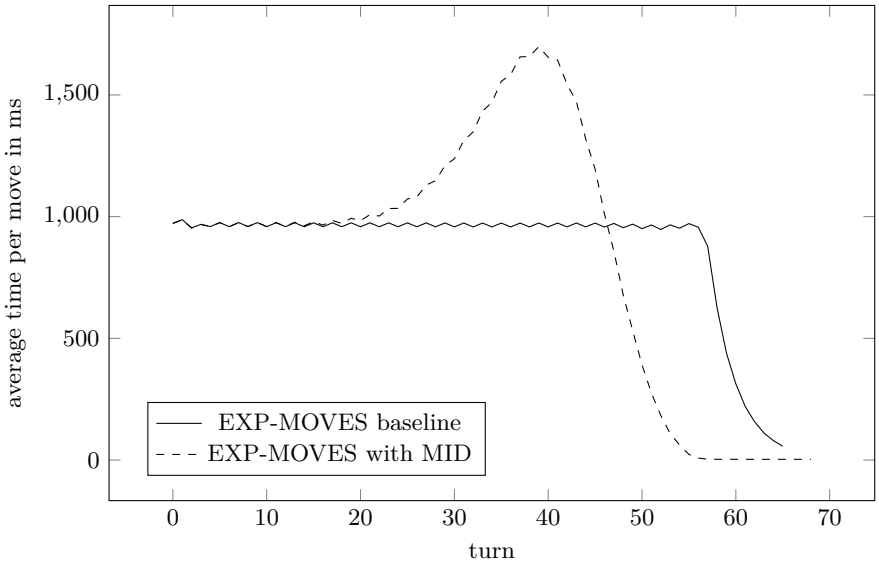


Figure 6.2: Average time distribution over a game of Othello with the MID strategy.

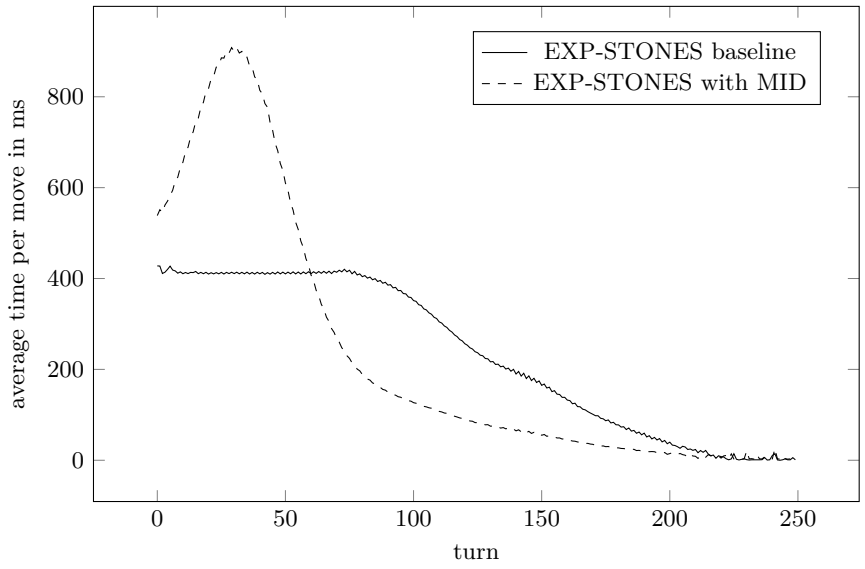


Figure 6.3: Average time distribution over a game of 13×13 Go with the MID strategy.

search time to the earlier phase of the game. For Connect-4, Figure 6.4 shows the similar effects of OPEN and BEHIND at optimal parameter settings when considering the average time distribution over the turns of the game. Since we already know the OPEN strategy to be effective in Connect-4, and OPEN and BEHIND are equally strong according to Table 6.6, it is an interesting question whether BEHIND provides any additional positive effect beyond this similar time shift.

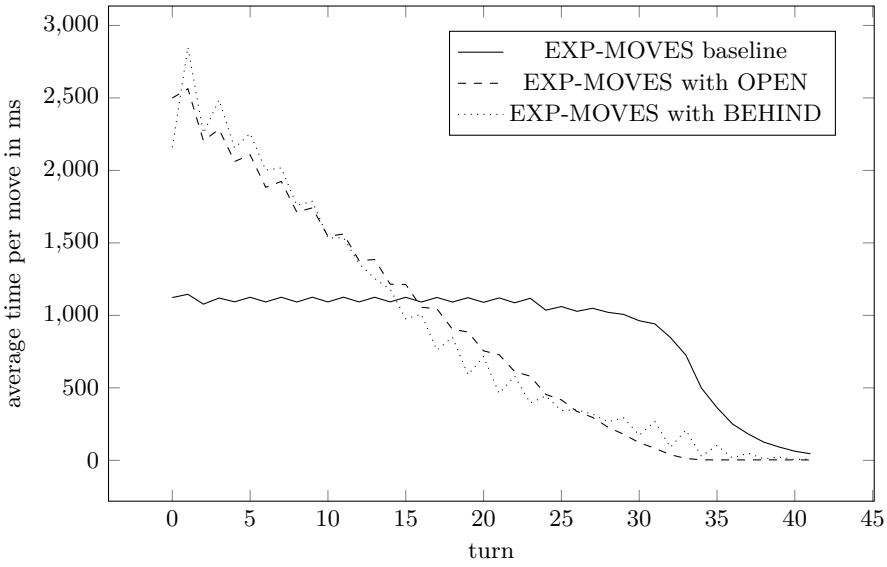


Figure 6.4: Average time distribution over a game of Connect-4 with the OPEN and BEHIND strategies.

In order to investigate this question, we determined the average time spent on each turn of the game when using the BEHIND strategy—as presented for Connect-4 in Figure 6.4—and created a regular MCTS player BEHIND-DISTRIBUTION which directly uses these search times depending on the current turn. Thus, it achieves the same time distribution over the game as BEHIND, while not using the BEHIND method itself. As an example, if BEHIND on average uses 673ms in turn 17 of a game, this is a result of win rates being compared to the threshold parameter v_{behind} and some proportion of searches being prolonged by a factor of f_{behind} in all turns up to 17. BEHIND-DISTRIBUTION just uses 673ms every time it reaches turn 17 in this example, regardless of the win rates observed. By comparing the playing strength of BEHIND-DISTRIBUTION to that of BEHIND we can determine whether the BEHIND method has a significant effect in addition to the time distribution over turns that it causes. BEHIND-DISTRIBUTION was also implemented for 13×13 Go.

Since the baseline in Go was EXP-STONES instead of EXP-MOVES, it used the same search times as BEHIND depending on the current number of stones on the board, not depending on the current turn. The results are comparable since the turn and the number of stones on the board correlate strongly.

As Table 6.12 shows, the win rate of BEHIND-DISTRIBUTION in Connect-4 is not significantly different from that of BEHIND. The same is true in Go, as indicated in Table 6.13. BEHIND only appears to improve playing strength in Connect-4 and 13×13 Go due to its time shift to the opening phase. It can be concluded that the BEHIND strategy is not effective in any of the tested domains except for Connect-4 and 13×13 Go, where it can be replaced by a strategy directly manipulating the time distribution over turns (or stones) such as OPEN.

UNST. The UNST strategy is significantly stronger than the baseline in all tested domains. Figure 6.5 demonstrates that in Go, UNST at optimal parameter settings has the effect of shifting time to the opening, similar to BEHIND. Figure 6.6 however shows that this is not the case for UNST at optimal parameter settings in Connect-4. The time distribution of UNST appears nearly identical to that of the EXP-MOVES baseline. In order to test for an effect of the UNST method beyond potential influences on the time distribution over the turns of the game, we constructed a UNST-DISTRIBUTION player for each test domain according to the procedure described for BEHIND-DISTRIBUTION above. The Go player was again based on the time distribution over the number of stones on the board.

Tables 6.12 and 6.13 show that UNST-DISTRIBUTION is not significantly better than the baseline in any domain but Go, where it is not significantly different from UNST. We can therefore conclude that UNST is useful in Connect-4, Breakthrough, Othello, and Catch the Lion, independently of the time distribution over turns that results from it. In 13×13 Go however, the success of UNST largely depends on a time shift to the opening, similar to BEHIND.

CLOSE. The CLOSE strategy is improving the performance of the baseline in all tested domains except for Othello. As for BEHIND and UNST, a CLOSE-DISTRIBUTION player was created for each domain in order to test the influence of a shifted time distribution over turns of the game that is caused by applying CLOSE. Tables 6.12 and 6.13 demonstrate that in Breakthrough and Catch the Lion, CLOSE-DISTRIBUTION does not achieve win rates significantly higher than 50%, meaning that CLOSE is playing stronger independently of the time distribution over the game that it causes. In Connect-4, CLOSE-DISTRIBUTION does perform better than EXP-MOVES—but still significantly worse than CLOSE itself, which means that the CLOSE strategy has a positive effect in addition to the time shift. In Go, this additional effect could not be shown to be statistically significant.

Table 6.12: Performance of the DISTRIBUTION players in Connect-4, Breakthrough, Othello, and Catch the Lion. 5000 games per player were played against the EXP-MOVES baseline.

Player	Win rate	95% conf. int.	Derived from	Win rate
In Connect-4:				
BEHIND-DISTR.	55.3%	53.9%-56.7%	BEHIND	55.8%
UNST-DISTR.	50.8%	49.4%-52.2%	UNST	54.9%
CLOSE-DISTR.	54.9%	53.5%-56.3%	CLOSE	58.7%
STOP_A-DISTR.	46.1%	44.7%-47.5%	STOP_A	50.8%
STOP_B-DISTR.	52.9%	51.5%-54.3%	STOP_B	63.0%
STOP-DISTR.	55.1%	53.7%-56.5%	STOP	65.0%
In Breakthrough:				
UNST-DISTR.	48.7%	47.3%-50.1%	UNST	54.2%
CLOSE-DISTR.	49.6%	48.2%-51.0%	CLOSE	53.1%
STOP_A-DISTR.	47.6%	46.2%-49.0%	STOP_A	54.5%
STOP_B-DISTR.	47.3%	45.9%-48.7%	STOP_B	56.9%
STOP-DISTR.	47.2%	45.8%-48.6%	STOP	58.2%
In Othello:				
UNST-DISTR.	48.9%	47.5%-50.3%	UNST	53.1%
CLOSE-DISTR.	49.3%	47.9%-50.7%	CLOSE	50.0%
STOP_A-DISTR.	49.1%	47.7%-50.5%	STOP_A	52.9%
STOP_B-DISTR.	49.6%	48.2%-51.0%	STOP_B	54.2%
STOP-DISTR.	50.4%	49.0%-51.8%	STOP	54.8%
In Catch the Lion:				
UNST-DISTR.	51.3%	49.9%-52.7%	UNST	56.4%
CLOSE-DISTR.	50.1%	48.7%-51.5%	CLOSE	57.4%
STOP_A-DISTR.	47.7%	46.3%-49.1%	STOP_A	50.0%
STOP_B-DISTR.	51.8%	50.4%-53.2%	STOP_B	56.9%
STOP-DISTR.	49.3%	47.9%-50.7%	STOP	60.4%

Table 6.13: Performance of the DISTRIBUTION players in 13×13 Go. 5000 games per player were played against GNU Go.

Player	Win rate	95% conf. int.	Derived from	Win rate
BEHIND-DISTR.	31.3%	30.0%-32.6%	BEHIND	29.9%
UNST-DISTR.	32.6%	31.3%-33.9%	UNST	33.6%
CLOSE-DISTR.	31.0%	29.7%-32.3%	CLOSE	32.6%
STOP_A-DISTR.	18.7%	17.6%-19.8%	STOP_A	25.3%
STOP_B-DISTR.	29.8%	28.5%-31.1%	STOP_B	36.7%
STOP-DISTR.	32.8%	31.5%-34.1%	STOP	39.1%

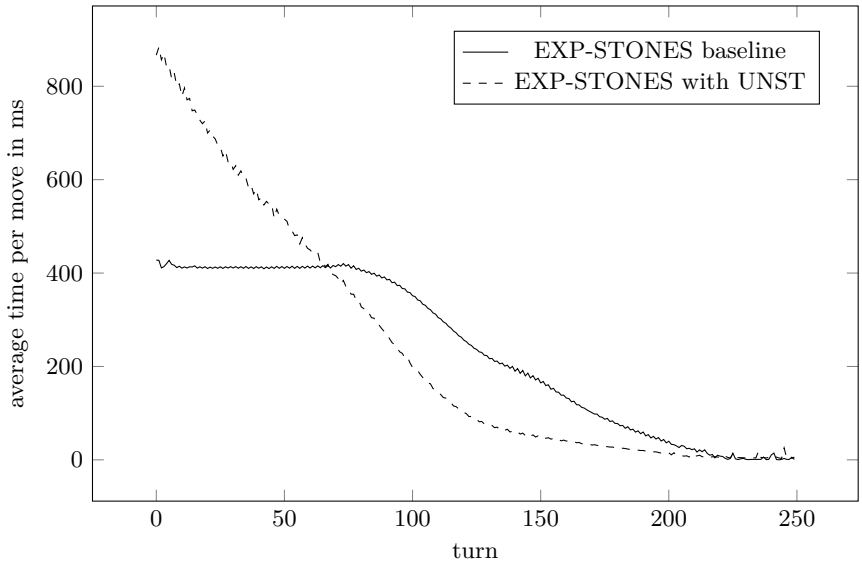


Figure 6.5: Average time distribution over a game of 13×13 Go with the UNST strategy.

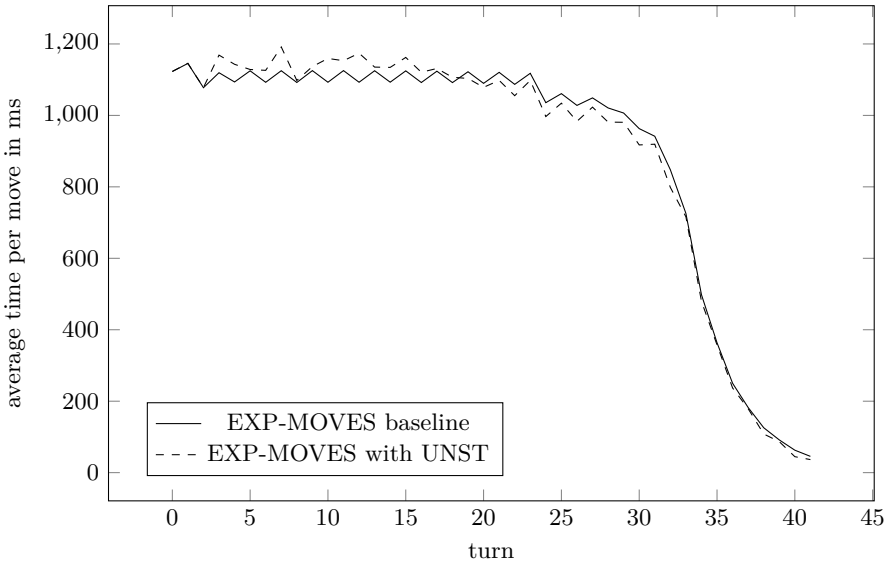


Figure 6.6: Average time distribution over a game of Connect-4 with the UNST strategy.

It remains to be shown why CLOSE does not perform well in Othello. It could be the case that there are too many situations in which two moves have relatively high and close visit counts, but investing additional search effort does not lead to a better move decision—either because the moves are equally valid, or because the consequences of the two moves cannot be reliably distinguished even with the help of a limited additional amount of time.

STOP. The STOP strategy—as well as the special case STOP_B—significantly improves on the baseline in all tested domains. Furthermore, STOP consistently achieved the highest win rate of all investigated time-management techniques across the five domains and is thus the most successful strategy tested in this chapter.

We determined the average percentage of the regular search time that was saved per move when STOP, STOP_A or STOP_B were active. For STOP_A/STOP_B, this percentage was 31.1%/32.1% in Connect-4, 24.7%/25.0% in Breakthrough, 23.1%/23.1% in Othello, and 32.4%/33.7% in Catch the Lion. Note that more than 50% cannot be saved with STOP_B, because the rest of the search time would then still be long enough to turn a completely unsampled move into the most-sampled one. In the most general STOP variant, more than 50% can theoretically be saved, as it gives up the guarantee of never stopping a search whose outcome could still change. The saved percentages of the search time for STOP were 34.5% in Connect-4, 47.7% in

Breakthrough, 25.2% in Othello, and 68.5% in Catch the Lion. The increase in saved time is related to the parameter $p_{\text{earlystop}}$—domains with high optimal $p_{\text{earlystop}}$ (0.9 in Connect-4 and Othello) only relax the guarantee to a small degree, domains with low optimal $p_{\text{earlystop}}$ (0.3 in Breakthrough and 0.2 in Catch the Lion) relax it further.

For all STOP variants, the performance of a player imitating their time distribution over the game was tested as well (named STOP_A-DISTRIBUTION etc.). According to Tables 6.12 and 6.13, the distribution over turns (or stones) of STOP and STOP_B alone has no significant positive effect on playing strength in all games except for Connect-4 and Go. In Connect-4 and Go, STOP-DISTRIBUTION does have a positive effect, but is still significantly weaker than STOP. The same holds for STOP_B-DISTRIBUTION. In conclusion, STOP and STOP_B are effective in all tested domains, independently of (or in addition to) their time distribution over turns.

The basic strategy STOP_A works in some domains (Breakthrough, Othello) but not in others (Connect-4, Catch the Lion, Go). As the success of STOP_B shows—a strategy that works just like STOP_A but simultaneously shifts time to the opening of the game—this is largely due to STOP_A's adverse effect of shifting time to the endgame. The fact that such time shifts can potentially occur with any given strategy makes studying and comparing these two STOP variants worthwhile. As illustrative examples, see Figures 6.7, 6.8, and for a visualization of the effects of STOP_A and STOP_B in Connect-4, Go, and Breakthrough, respectively. The time distribution of the most general STOP variant—not shown here for clarity—looks very similar to the STOP_B distribution in these games. An additional argument for the hypothesis that a time shift to the endgame hurts the performance of STOP_A comes from the performance of STOP_A-DISTRIBUTION (see Tables 6.12 and 6.13). In all tested domains except for Othello, also representing the only domain where STOP_B does not improve on STOP_A, the time distribution over turns (or stones) of STOP_A hurts performance significantly.

BEHIND-L, UNST-L, and CLOSE-L. As Table 6.11 shows, the repeated check for the termination conditions in UNST-L and CLOSE-L only resulted in significantly stronger play in the domain of Go. None of the other games profited from it. BEHIND-L was not significantly more effective than BEHIND in any tested domain. (Note that UNST, CLOSE, and BEHIND are special cases of UNST-L, CLOSE-L, and BEHIND-L, respectively. Therefore, the looping versions cannot perform worse than the simple versions if tuned optimally.) It is possible that in the set of domains used, only Go is sufficiently complex with regard to branching factor and game length to make such fine-grained timing decisions worthwhile. See Table 6.14 for a comparison of average game lengths and average branching factors of all five domains investigated. All values are averages from 1000 self-play games of regular MCTS, 1 second per move.

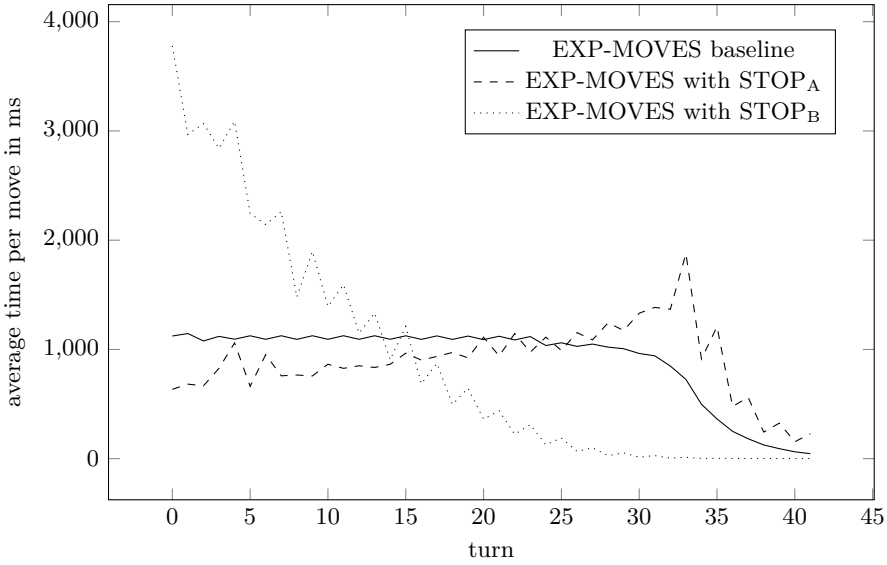


Figure 6.7: Average time distribution over a game of Connect-4 with the STOP_A and STOP_B strategies. While STOP_A shifts time to the endgame when compared to the baseline, STOP_B shifts time to the opening phase of the game.

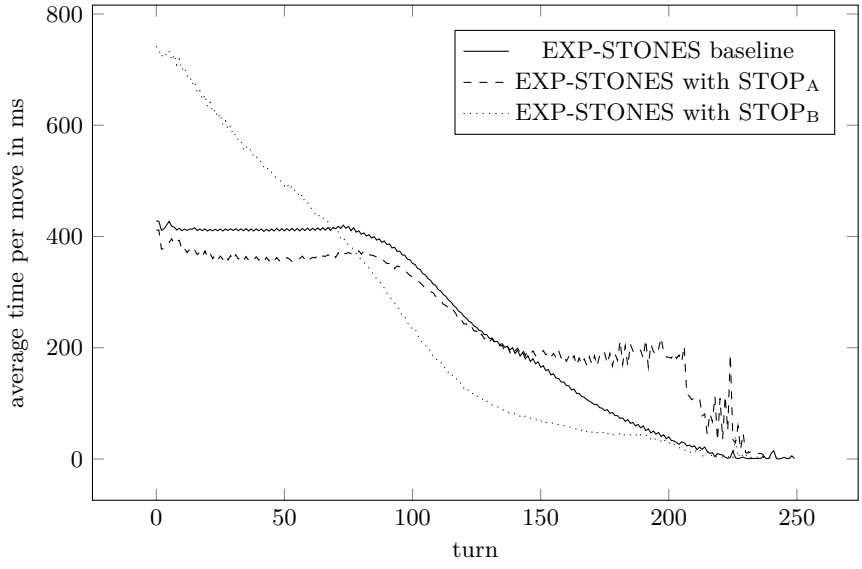


Figure 6.8: Average time distribution over a game of 13×13 Go with the STOP_A and STOP_B strategies.

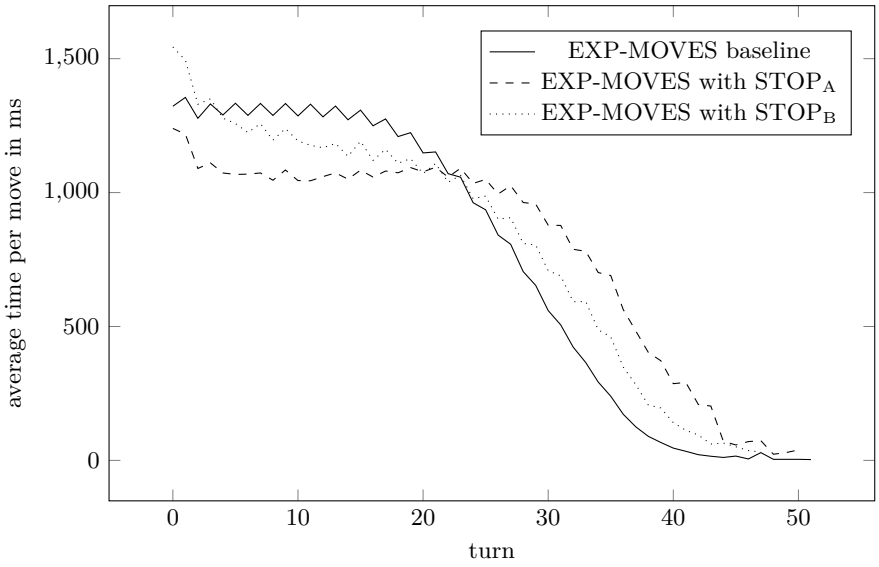


Figure 6.9: Average time distribution over a game of Breakthrough with the STOP_A and STOP_B strategies.

In Go, we took the average against GNU Go at 30 seconds per game.

All time-management strategies that *prolong* search times when certain criteria are met, such as BEHIND, UNST, and CLOSE, take available time from the later phases and shift it to the earlier phases of the game. Strategies that *shorten* search times based on certain criteria, such as STOP_A, move time from the opening towards the endgame instead. When analyzing the effect of time-management approaches, it is therefore worth testing whether these shifts have a positive or negative effect. Should the effect be negative, STOP_B provides an example of how to possibly counteract it by introducing an explicit shift in the opposite direction.

Table 6.14: Game length and branching factor in Connect-4, Breakthrough, Othello, Catch the Lion, and 13×13 Go.

	Connect-4	Breakthrough	Othello	Catch the Lion	13×13 Go
Game length	37	29	61	35	150
Branching factor	5.8	15.5	8	10.5	90

6.5 Conclusion and Future Research

In this chapter, we investigated a variety of time-management strategies for Monte-Carlo Tree Search, using the games of Go, Connect-4, Breakthrough, Othello, and Catch the Lion as a testbed. This included newly proposed strategies (called OPEN, MID, KAPPA-EXP, KAPPA-LM, and KAPPA-CM) as well as strategies described in Huang et al. (2010b) (UNST and BEHIND) or independently proposed in Baudiš (2011) (CLOSE and STOP_A), partly in enhanced form (UNST-L, BEHIND-L, CLOSE-L, STOP_B, and STOP). Empirical results show that the proposed strategy EXP-STONES with STOP provides a significant improvement over the state of the art as represented by ERICA-BASELINE in 13×13 Go. For sudden-death time controls of 30 seconds per game, EXP-STONES with STOP increased Orego's win rate against GNU Go from 25.5% (using a simple baseline) or from 35.3% (using the state-of-the-art ERICA-BASELINE) to 39.1%. In self-play, this strategy won approximately 60% of games against ERICA-BASELINE, both in 13×13 and 19×19 Go under various time controls.

Based on a comparison across different games we conclude that the domain-independent strategy STOP is the strongest of all tested time-management strategies. It won 65.0% of self-play games in Connect-4, 58.2% in Breakthrough, 54.8% in Othello, and 60.4% in Catch the Lion. With the exception of CLOSE in Othello, UNST and CLOSE also prove effective in all domains. Since many time-management strategies result in a shift of available time towards either the opening or the endgame, a methodology was developed to isolate the effect of this shift and judge the effect of a given strategy independently of it. In conclusion, STOP is a promising sudden-death time management strategy for MCTS.

The following directions appear promising for future research. First, a natural next step is the combined testing of all above strategies—in order to determine to which degree their positive effects on playing strength can complement each other, or to which degree they could be redundant (such as OPEN and BEHIND in Connect-4), or possibly interfere. ERICA-BASELINE demonstrates that some combinations can be effective at least in Go. Second, a non-linear classifier like a neural network could be trained to decide about continuing or aborting the search in short intervals, using all relevant information used by above strategies as input. A third direction is the development of improved strategies to measure the complexity and importance of a position and thus to effectively use time where it is most needed. In Go for example, counting the number of independent fights on the board could be one possible, domain-dependent approach. Furthermore, possible interactions of time management strategies with other MCTS enhancements could be studied, such as for instance the sufficiency-based selection strategy by Gudmundsson and Björnsson (2013).

7

MCTS and Minimax Hybrids

This chapter is based on:

Baier, H. and Winands, M. H. M. (2015). MCTS-Minimax Hybrids. *IEEE Transactions on Computational Intelligence and AI in Games*, volume 7, number 2, pages 167–179.

Baier, H. and Winands, M. H. M. (2013). Monte-Carlo Tree Search and Minimax Hybrids. In *2013 IEEE Conference on Computational Intelligence and Games, CIG 2013*, pages 129–136.

Although MCTS has shown considerable success in a variety of domains, there are still a number of games such as Chess and Checkers in which the traditional approach to adversarial planning, minimax search with $\alpha\beta$ pruning (Knuth and Moore, 1975), remains superior. The comparatively better performance of $\alpha\beta$ cannot always be explained by the existence of effective evaluation functions for these games, as evaluation functions have been successfully combined with MCTS to produce strong players in games such as Amazons and Lines of Action (Lorentz, 2008; Winands et al., 2010).

Since MCTS builds a highly selective search tree, focusing on the most promising lines of play, it has been conjectured that it could be less appropriate than traditional, non-selective minimax search in domains containing a large number of terminal states and *shallow traps* (Ramanujan et al., 2010a). In trap situations such as those frequent in Chess, precise tactical play is required to avoid immediate loss. MCTS, which is based on sampling, could easily miss a crucial move or underestimate the significance of an encountered terminal state due to averaging value backups. Conversely, MCTS could be more effective in domains such as Go, where terminal states and potential traps do not occur until the latest stage of the game. MCTS can here fully play out

its strategic and positional understanding resulting from Monte-Carlo simulations of entire games.

This chapter and the next one answer the fourth research question by exploring ways of *combining* the strategic strength of MCTS and the tactical strength of minimax in order to produce more universally useful hybrid search algorithms. In this chapter we do not assume the existence of heuristic evaluation functions, allowing the MCTS-minimax hybrids to be applied in any domain where MCTS is used without such heuristics (e.g. General Game Playing). The three proposed approaches use minimax search in the selection/expansion phase, the rollout phase, and the backpropagation phase of MCTS. We investigate their effectiveness in the test domains of Connect-4, Breakthrough, Othello, and Catch the Lion.

This chapter is structured as follows. Section 7.1 provides some background on MCTS-Solver as the baseline algorithm. Section 7.2 gives a brief overview of related work on the relative strengths of minimax and MCTS, as well as results with combining or nesting tree search algorithms. Section 7.3 describes different ways of incorporating shallow-depth minimax searches into the different parts of the MCTS framework, and Section 7.4 shows experimental results of these knowledge-free MCTS-minimax hybrids in the four test domains. Conclusions and future research follow in Section 7.5.

7.1 MCTS-Solver

In this chapter, we do not assume the availability of heuristic evaluation functions. Therefore, minimax search can only distinguish terminal and non-terminal game states, potentially producing search results such as *proven win* or *proven loss* through minimax backup. In order to be able to handle these proven values, we use MCTS with the *MCTS-Solver* extension (Winands et al., 2008) as the baseline algorithm.

The basic idea of MCTS-Solver is allowing for the backpropagation of not only regular simulation outcomes such as 0 (loss) or 1 (win), but also game-theoretic values such as *proven losses* and *proven wins* whenever terminal states are encountered by the search tree. First, whenever a move from a given game state s has been marked as a proven win for player A, the move leading to s can be marked as a proven loss for the opposing player B. Second, whenever *all* moves from a given state s have been marked as proven losses for A, the move leading to s can be marked as a proven win for B. If at least one move from s has not been proven to be a loss yet, the move leading to s is only updated with a regular rollout win in this backpropagation phase. We do not prove draws in this thesis. Draws are backpropagated with the value 0.5.

Whenever the selection policy encounters a node with a child marked as a proven win, a win can be returned for this simulation without performing any rollout. Similarly, proven losses can be avoided without having to re-sample them.

This solving extension to plain MCTS has been successfully used e.g. in Lines of Action (Winands et al., 2008), Hex (Arneson et al., 2010; Cazenave and Saffidine, 2009), Havannah (Lorentz, 2011), Shogi (Sato et al., 2010), Tron (Den Teuling and Winands, 2012), Focus (Nijssen and Winands, 2011), and Breakthrough (Lorentz and Horey, 2014). It has been generalized to Score-bounded MCTS, which handles more than two game outcomes in a way that allows for pruning (Cazenave and Saffidine, 2011), and to simultaneous move games in the concept of General Game Playing (Finnsson, 2012).

MCTS-Solver handles game-theoretic values better than MCTS without the extension because it avoids spending time on the re-sampling of proven game states. However, it suffers from the weakness that such game-theoretic values often propagate slowly up the tree before they can influence the move decision at the root. MCTS-Solver may have to keep sampling a state many times until it has proved *all* moves from this state to be losses, such that it can backpropagate a proven win to the next-higher level of the tree. In Subsection 7.3.3 we describe how we use shallow-depth, exhaustive minimax searches to speed up this process and guide MCTS more effectively.

7.2 Related Work

The research of Ramanujan et al. (2010a), Ramanujan and Selman (2011), and Ramanujan et al. (2011) has repeatedly dealt with characterizing search space properties that influence the performance of MCTS relative to minimax search. *Shallow traps* were identified in Ramanujan et al. (2010a) as a feature of domains that are problematic for MCTS, in particular Chess. Informally, the authors define a *level-k search trap* as the possibility of a player to choose an unfortunate move such that *after* executing the move, the opponent has a guaranteed winning strategy at most k plies deep. While such traps at shallow depths of 3 to 7 are not found in Go until the latest part of the endgame, they are relatively frequent in Chess games even at grandmaster level (Ramanujan et al., 2010a), partly explaining the problems of MCTS in this domain. A resulting hypothesis is that in regions of a search space containing no or very few terminal positions, shallow traps should be rare and MCTS variants should make comparatively better decisions, which was confirmed in Ramanujan and Selman (2011) for the game of Kalah (called Mancala by the authors). In Ramanujan et al. (2011) finally, an artificial game tree model was used to explore the dependence of MCTS performance on the density of traps in the search space. A similar problem to shallow traps was presented by Finnsson and Björnsson (2011) under the name of *optimistic moves*—seemingly strong moves that can be refuted right away by the opponent, but take MCTS prohibitively many simulations to find the refutation. One of the motivations of the work in this chapter was to employ shallow-depth minimax searches

within MCTS to increase the visibility of shallow traps and allow MCTS to avoid them more effectively.

In the context of General Game Playing, Clune (2008) compared the performance of minimax with $\alpha\beta$ pruning and MCTS. Restricted to the class of turn-taking, two-player, zero-sum games we are addressing here, the author identified a stable and accurate evaluation function as well as a relatively low branching factor as advantages for minimax over MCTS. In this chapter, we explore the use of minimax within the MCTS framework even when no evaluation function is available.

One method of combining different tree search algorithms that was proposed in the literature is the use of shallow minimax searches in every step of the MCTS *rollout phase*. This was typically restricted to checking for decisive and anti-decisive moves, as in Teytaud and Teytaud (2010) and Lorentz (2011) for the game of Havannah. 2-ply searches have been applied to the rollout phase in Lines of Action (Winands and Björnsson, 2011), Chess (Ramanujan et al., 2010b), as well as various multi-player games (Nijssen and Winands, 2012). However, the existence of a heuristic evaluation function was assumed here. For MCTS-Solver, a 1-ply lookahead for winning moves in the *selection phase* at leaf nodes has already been proposed in Winands et al. (2008), but was not independently evaluated. A different hybrid algorithm $UCTMAX_H$ was proposed in Ramanujan and Selman (2011), employing minimax backups in an MCTS framework. However, again a strong heuristic evaluator was assumed as a prerequisite. In our work, we explore the use of minimax searches of various depths without any domain knowledge beyond the recognition of terminal states. Minimax in the rollout phase is covered in Subsection 7.3.1.

In the framework of proof-number search (PNS, see Allis et al. 1994), 1- and 3-ply minimax searches have been applied in the expansion phase of PNS (Kaneko et al., 2005). In Winands et al. (2001), nodes proven by PNS in a first search phase were stored and reused by $\alpha\beta$ search in a second search phase. In Saito et al. (2007), Monte-Carlo rollouts were used to initialize the proof and disproof numbers at newly expanded nodes.

Furthermore, the idea of nesting search algorithms has been used in Cazenave (2009) and Chapter 4 of this thesis to create Nested Monte-Carlo Search and Nested Monte-Carlo Tree Search, respectively. In this chapter, we are not using search algorithms recursively, but nesting two different algorithms in order to combine their strengths: MCTS and minimax.

7.3 Hybrid Algorithms

In this section, we describe three different approaches for applying minimax with $\alpha\beta$ pruning within the MCTS framework.

7.3.1 Minimax in the Rollout Phase

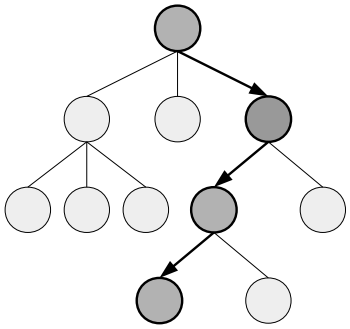
While uniformly random move choices in the rollout are sufficient to guarantee the convergence of MCTS to the optimal policy, more informed rollout strategies typically greatly improve performance (Gelly et al., 2006). For this reason, it seems natural to use fixed-depth minimax searches for choosing rollout moves. Since we do not use evaluation functions in this chapter, minimax can only find forced wins and avoid forced losses, if possible, within its search horizon. If minimax does not find a win or loss, we return a random move. The algorithm is illustrated in Figure 7.1.

This strategy thus improves the quality of play in the rollouts by avoiding certain types of blunders. It informs tree growth by providing more accurate rollout returns. We call this strategy *MCTS-MR* for *MCTS with Minimax Rollouts*.

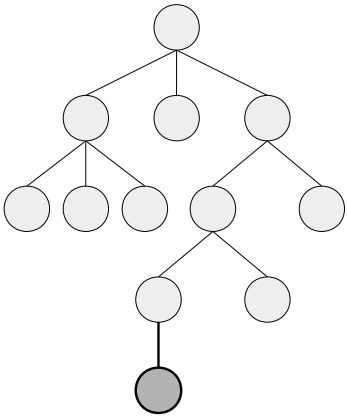
7.3.2 Minimax in the Selection and Expansion Phases

Minimax searches can also be embedded in the phases of MCTS that are concerned with traversing the tree from root to leaf: the selection and expansion phases. This strategy can use a variety of possible criteria to choose whether or not to trigger a minimax search at any state encountered during the traversal. In the work described in this chapter, we experimented with starting a minimax search as soon as a state has reached a given number of visits (for 0 visits, this would include the expansion phase). Figure 7.2 illustrates the process. Other possible criteria include e.g. starting a minimax search for a loss as soon as a given number of moves from a state have already been proven to be losses, or starting a minimax search for a loss as soon as average returns from a node fall below a given threshold (or searching for a win as soon as returns exceed a given threshold, conversely), or starting a minimax search whenever average rollout lengths from a node are short, suggesting proximity of terminal states. According to preliminary experiments, the simple criterion of visit count seemed most promising, which is why it was used in the remainder of this chapter. Furthermore, we start independent minimax searches for each legal move from the node in question, which allows to store found losses for individual moves even if the node itself cannot be proven to be a loss.

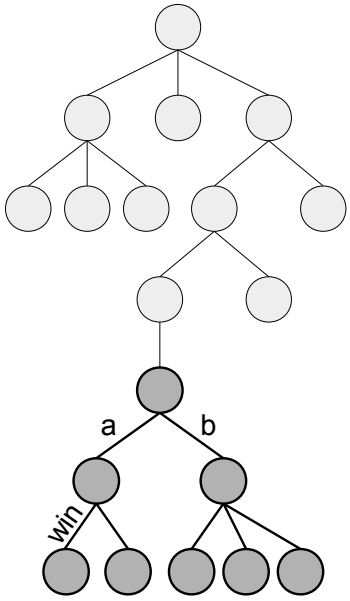
This strategy improves MCTS by performing shallow-depth, full-width checks of the immediate descendants of a subset of tree nodes. It guides tree growth by avoiding shallow losses, as well as detecting shallow wins, within or close to the MCTS tree. We call this strategy *MCTS-MS* for *MCTS with Minimax Selection*.



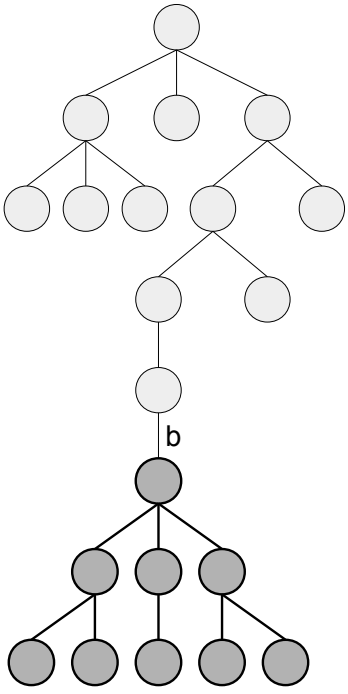
(a) The selection phase.



(b) The expansion phase.

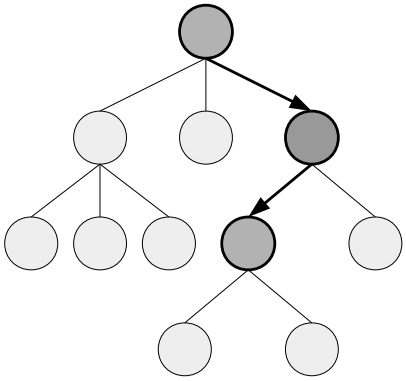


(c) A minimax search is started to find the first rollout move. Since the opponent has a winning answer to move a, move b is chosen instead in this example.

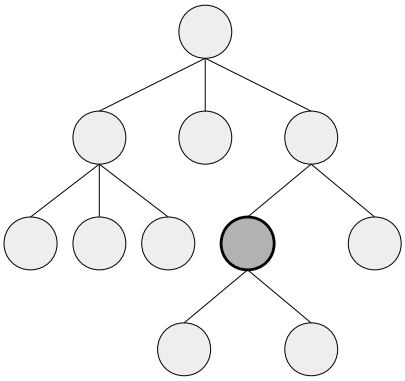


(d) Another minimax search is conducted for the second rollout move. In this case, no terminal states are found and a random move choice will be made.

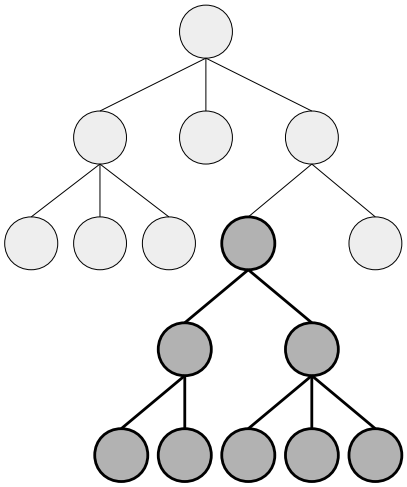
Figure 7.1: The MCTS-MR hybrid. The minimax depth is 2.



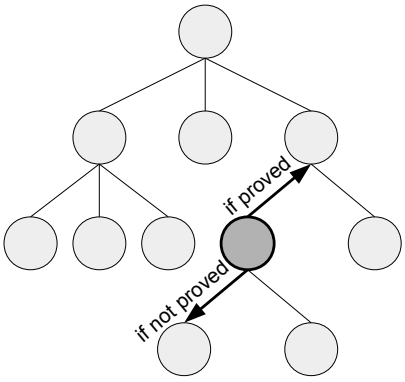
(a) Selection and expansion phases. The tree is traversed in the usual fashion until a node satisfying the minimax trigger criterion is found.



(b) In this case, the marked node has reached a prespecified number of visits.



(c) A minimax search is started from the node in question.



(d) If the minimax search has proved the node's value, this value can be backpropagated. Otherwise, the selection phase continues as normal.

Figure 7.2: The MCTS-MS hybrid. The minimax depth is 2.

7.3.3 Minimax in the Backpropagation Phase

As mentioned in Section 7.1, MCTS-Solver tries to propagate game-theoretic values (*proven win* and *proven loss*) as far up the tree as possible, starting from the terminal state visited in the current simulation. It has to switch to regular rollout returns (*win* and *loss*) as soon as at least one sibling of a proven loss move is not marked as proven loss itself. Therefore, we employ shallow minimax searches whenever this happens, actively searching for proven losses instead of hoping for MCTS-Solver to find them in future simulations. If minimax succeeds at proving all moves from a given state s to be losses, we can backpropagate a *proven loss* instead of just a *loss* to the next-highest tree level—i.e. a proven win for the opponent player's move leading to s (see a negamax formulation of this algorithm in Figure 7.3).

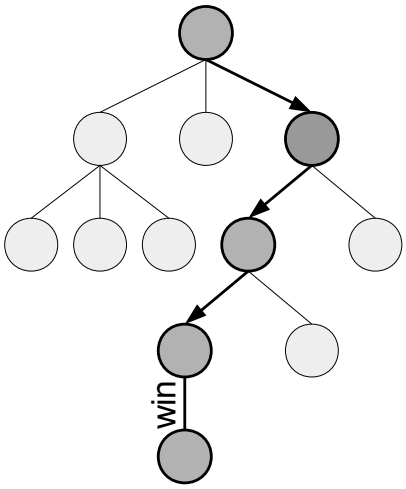
This strategy improves MCTS-Solver by providing the backpropagation step with helpful information whenever possible, which allows for quicker proving and exclusion of moves from further MCTS sampling. Other than the strategies described in 7.3.1 and 7.3.2, it only triggers when a terminal position has been found in the tree and the MCTS-Solver extension applies. For this reason, it avoids spending computation time on minimax searches in regions of the search space with no or very few terminal positions. Minimax can also search deeper each time it is triggered, because it is triggered less often. We call this strategy *MCTS-MB* for *MCTS with Minimax Backpropagation*.

7.4 Experimental Results

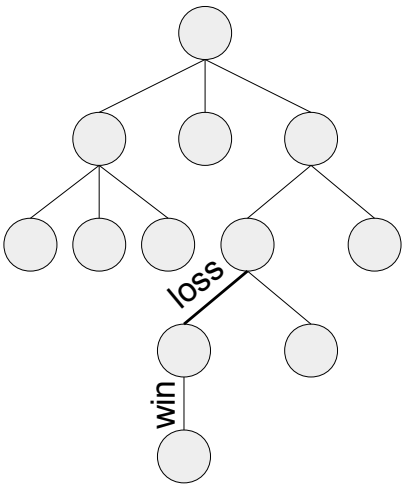
We tested the MCTS-minimax hybrids in four different domains: the two-player, zero-sum games of *Connect-4*, *6×6 Breakthrough*, *Othello*, and *Catch the Lion*. In all experimental conditions, we compared the hybrids against regular MCTS-Solver as the baseline. UCB1-TUNED (Auer et al., 2002) is used as selection policy. The exploration factor C of UCB1-TUNED was optimized once for MCTS-Solver in all games and then kept constant for both MCTS-Solver and the MCTS-minimax hybrids during testing. Optimal values were 1.3 in Connect-4, 0.8 in Breakthrough, 0.7 in Othello, and 0.7 in Catch the Lion. Draws, which are possible in Connect-4 and Othello, were counted as half a win for both players. We used minimax with $\alpha\beta$ pruning, but no other search enhancements—both in order to keep the experiments simple, and because the overhead of many search enhancements is too high for shallow searches. Unless stated otherwise, computation time was 1 second per move.

To ensure a minimal standard of play, the MCTS-Solver baseline was tested against a random player. MCTS-Solver won 100% of 1000 games in all four domains.

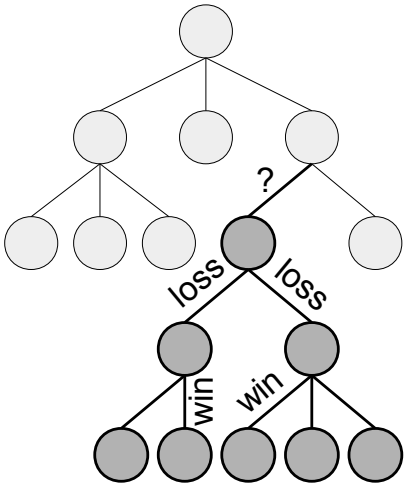
Note that Figures 7.11 to 7.22 show the results of parameter tuning experiments.



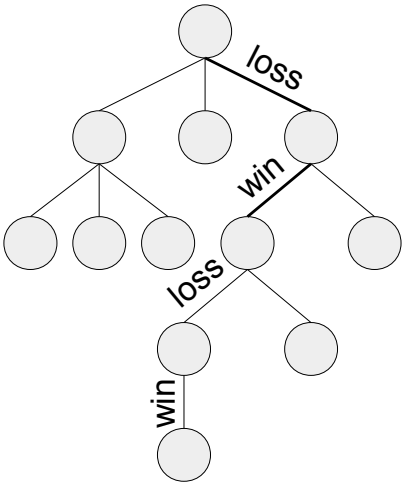
(a) Selection and expansion phases. The expanded move wins the game.



(b) This implies the opponent's previous move was proven to be a loss.



(c) A minimax search is triggered in order to check whether the move marked by "?" can be proven to be a win. In this example, all opponent answers are proven losses, so it can.



(d) This also implies the opponent's previous move was proven to be a loss. The root state's value is now proven.

Figure 7.3: The MCTS-MB hybrid. The minimax search is 2.

The best-performing parameter values found during tuning were tested with an *additional* 5000 games after each tuning experiment. The results of these replications are reported in the text.

This section is organized as follows. In 7.4.1, the density and difficulty of shallow traps in the four domains is measured. This gives an indication of how tactical each game is and how well we therefore expect the hybrids to perform in relation to the other games. Next, 7.4.2 to 7.4.5 present experimental results for all three hybrids in Connect-4, Breakthrough, Othello, and Catch the Lion. In 7.4.6, the relative strength of the hybrids is confirmed by testing them against each other instead of the baseline. The expectations of relative algorithm performance across domains from Subsection 7.4.1 are then confirmed in 7.4.7. Afterwards, the effects of different time settings and different branching factors are studied in 7.4.8 and 7.4.9, respectively. Subsection 7.4.10 finally provides results on the solving performance of the three hybrids.

7.4.1 Density and Difficulty of Shallow Traps

In order to measure an effect of employing shallow minimax searches without an evaluation function within MCTS, terminal states have to be present in sufficient density throughout the search space, in particular the part of the search space relevant at our level of play. We played 1000 self-play games of MCTS-Solver in all domains to test this property, using 1 second per move. At each turn, we determined whether there exists *at least one* trap at depth (up to) 3 for the player to move. The same methodology was used in Ramanujan et al. (2010a).

Figures 7.4, 7.5, 7.6, and 7.7 show that shallow traps are indeed found throughout most domains, which suggests improving the ability of MCTS to identify and avoid such traps is worthwhile. Traps appear most frequently in Catch the Lion—making it a highly tactical domain—followed by Breakthrough and Connect-4. A game of Othello usually only ends when the board is completely filled however, which explains why traps only appear when the game is nearly over. Furthermore, we note that in contrast to Breakthrough and Othello the density of traps for both players differs significantly in Connect-4 and in the early phase of Catch the Lion. Finally, we see that Breakthrough games longer than 40 turns, Othello games longer than 60 turns and Catch the Lion games longer than 50–60 moves are rare, which explains why the data become more noisy.

In order to provide a more condensed view of the data, Figure 7.8 compares the *average number* of level-3 to level-7 search traps over all positions encountered in the test games. These were 34,187 positions in Catch the Lion, 28,344 positions in Breakthrough, 36,633 positions in Connect-4, and 60,723 positions in Othello. Note that a level-k trap requires a winning strategy *at most* k plies deep, which means every

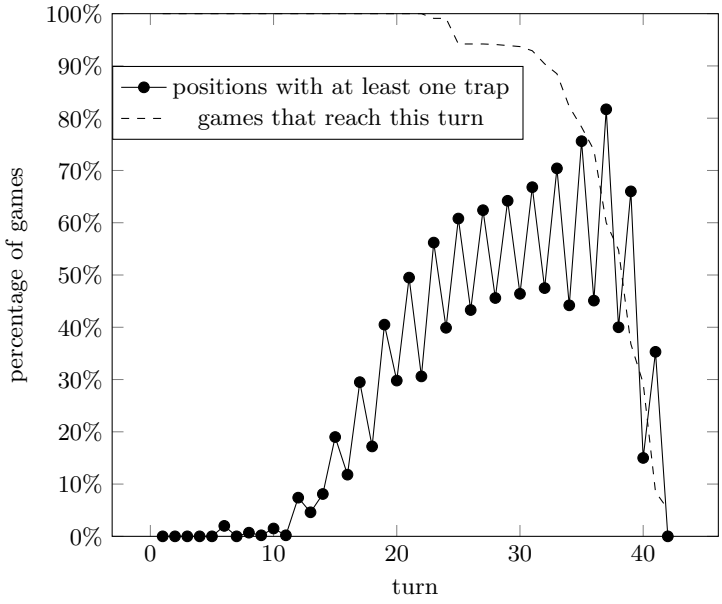


Figure 7.4: Density of level-3 search traps in Connect-4.

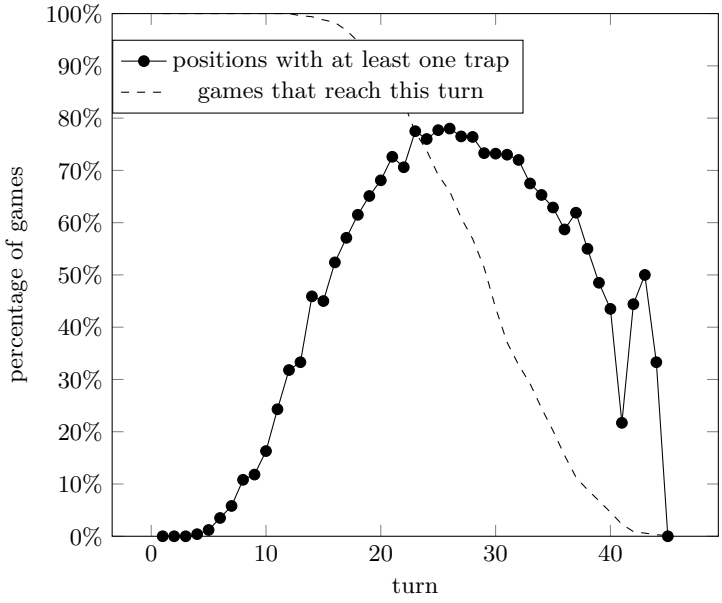


Figure 7.5: Density of level-3 search traps in Breakthrough.

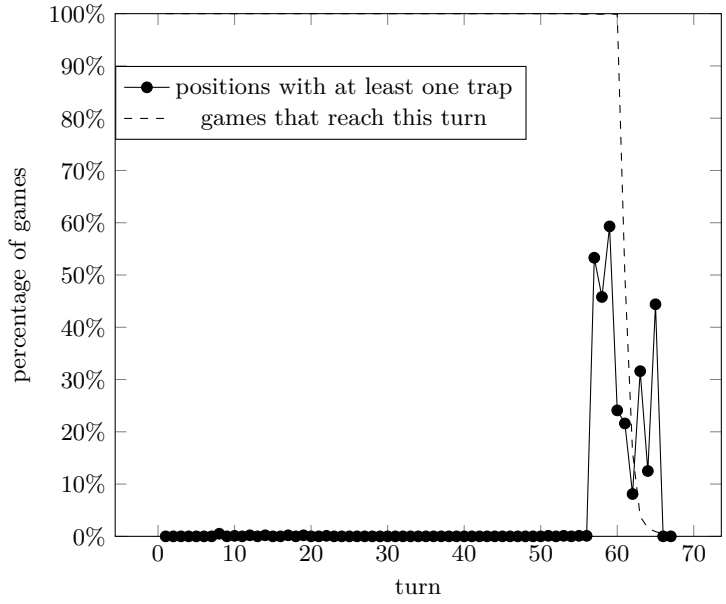


Figure 7.6: Density of level-3 search traps in Othello.

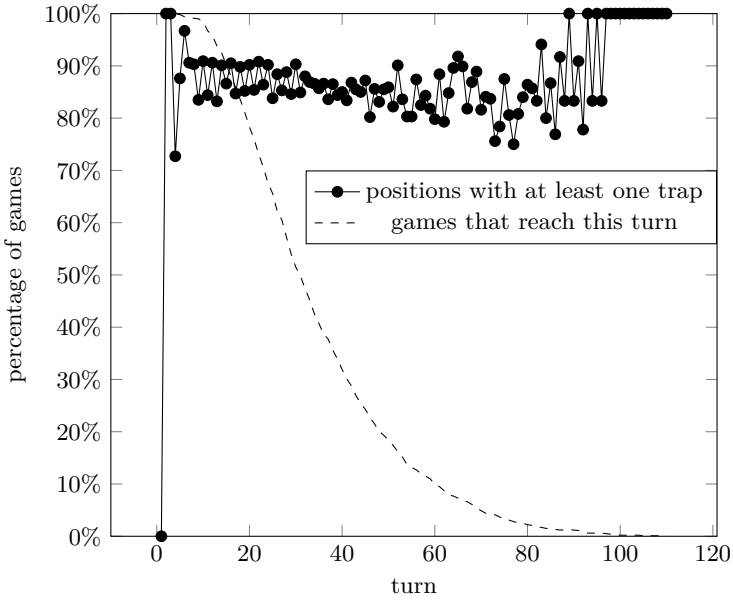


Figure 7.7: Density of level-3 search traps in Catch the Lion.

level-k trap is a level-$(k+1)$ trap as well. As Figure 7.8 shows, Catch the Lion is again leading in trap density, followed by Breakthrough, Connect-4, and finally Othello with a negligible average number of traps.

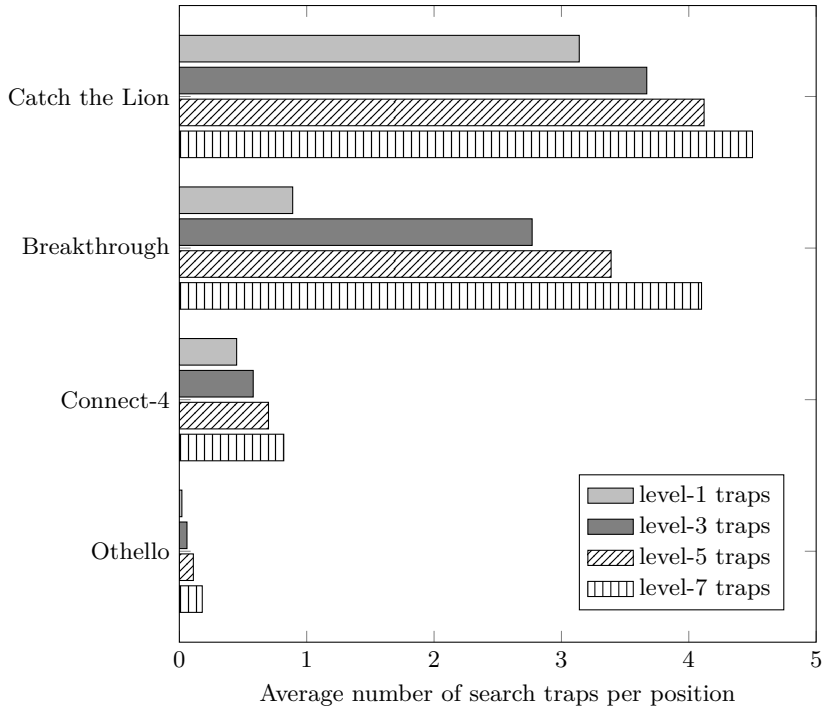


Figure 7.8: Comparison of trap density in Catch the Lion, Breakthrough, Connect-4, and Othello.

In an additional experiment, we tried to quantify the average *difficulty* of traps for MCTS. MCTS is more likely to "fall into" a trap, i.e. waste much effort on exploring a trap move, if rollouts starting with this move frequently return a misleading winning result instead of the correct losing result. Depending on the search space, trap moves might be relatively frequent but still easy to avoid for MCTS because they get resolved correctly in the rollouts—or less frequent but more problematic due to systematic errors in rollout returns. Therefore, 1000 random rollouts were played starting with every level-3 to level-7 trap move found in the test games. No tree was built during sampling. Any rollout return other than a loss was counted as incorrect. Figure 7.9 shows the proportion of rollouts that returned the incorrect result, averaged over all traps.

We observe that in the set of four test domains, the domains with the highest

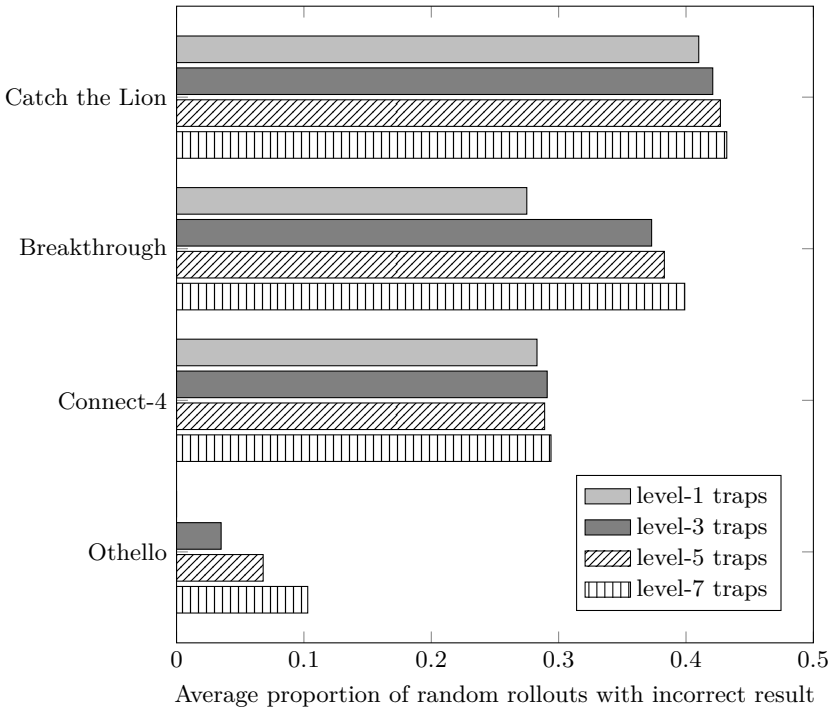


Figure 7.9: Comparison of trap difficulty in Catch the Lion, Breakthrough, Connect-4, and Othello.

average *number* of traps are also the domains with the highest expected *difficulty* of traps. Catch the Lion is again followed by Breakthrough, Connect-4, and Othello in last place with near-perfect random evaluation of traps. This property of Othello can be explained with the fact that almost all traps appear at the end of the game when the board is filled, and the last filling move has no alternatives. Thus, the opponent can make one mistake less than in other games when executing the trap, and in many situations executing the trap is the only legal path.

In conclusion, due to both the difference in trap density as well as trap difficulty, we expect MCTS-minimax hybrids to work relatively better in domains like Catch the Lion than in domains like Othello. In Subsection 7.4.7, the observations of this section are related to the performance of the MCTS-minimax hybrids presented in the following, and this expectation is confirmed.

7.4.2 Connect-4

In this subsection, we summarize the experimental results in the game of Connect-4. The baseline MCTS-Solver implementation performs about 91000 simulations per second when averaged over an entire game.

Minimax in the Rollout Phase

We tested minimax at search depths 1 ply to 4 plies in the rollout phase of a Connect-4 MCTS-Solver player. Each resulting player, abbreviated as MCTS-MR-1 to MCTS-MR-4, played 1000 games against regular MCTS-Solver with uniformly random rollouts. Figure 7.11 presents the results.

Minimax is computationally more costly than a random rollout policy. MCTS-MR-1 finishes about 69% as many simulations per second as the baseline, MCTS-MR-2 about 31% as many, MCTS-MR-3 about 11% as many, MCTS-MR-4 about 7% as many when averaged over one game of Connect-4. This typical speed-knowledge trade-off explains the decreasing performance of MCTS-MR for higher minimax search depths, although the quality of rollouts increases. Remarkably, MCTS-MR-1 performs significantly worse than the baseline. This also held when we performed the comparison using equal numbers of MCTS iterations (100000) per move instead of equal time (1 second) per move for both players. In this scenario, we found MCTS-MR-1 to achieve a win rate of 36.3% in 1000 games against the baseline. We suspect this is due to some specific imbalance in Connect-4 rollouts with depth-1 minimax—it has been repeatedly found that the strength of a rollout policy as a standalone player is not always a good predictor of its strength when used within a Monte-Carlo search algorithm (see Bouzy and Chaslot (2006) and Gelly and Silver (2007) for similar observations in the game of Go using naive Monte-Carlo and Monte-Carlo Tree Search, respectively).

In order to illustrate this phenomenon, Figure 7.10 gives an intuition for situations in which depth-1 minimax rollouts can be less effective for state evaluation than uniformly random rollouts. The figure shows the partial game tree (not the search tree) of a position that is a game-theoretic win for the root player. Only if the root player does not choose move a at the root state can her opponent prevent the win. The building of a search tree is omitted for simplification in the following. If we start a uniformly random rollout from the root state, we will get the correct result with a probability of $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{2}{3} = 0.75$. If we use a depth-1 minimax rollout however, the root player's opponent will always be able to make the correct move choice at states A, B, and C, leading to an immediate loss for the root player. As a result, the correct result will only be found with a probability of $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot 0 = 0.25$. If similar situations appear in sufficient frequency in Connect-4, they could provide an explanation for systematic evaluation errors of MCTS-MR-1.

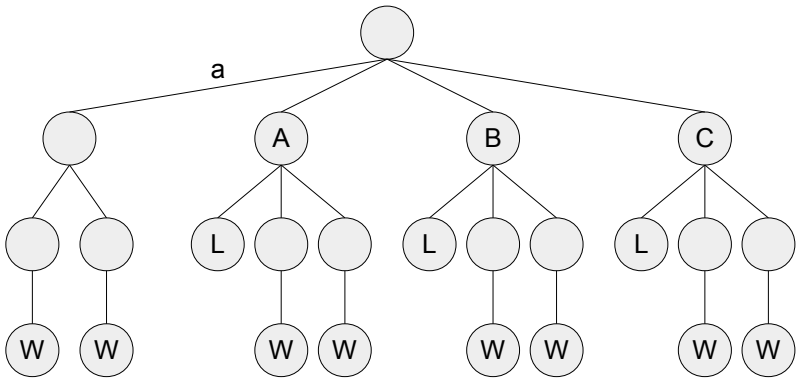


Figure 7.10: A problematic situation for MCTS-MR-1 rollouts. The figure represents a partial game tree. "L" and "W" mark losing and winning states from the point of view of the root player.

In the Connect-4 experiments, MCTS-MR-2 outperformed all other variants. Over an entire game, it completed about 28000 simulations per second on average. In an additional 5000 games against the baseline, it won 72.1% (95% confidence interval: 70.8% − 73.3%) of games, which is a significant improvement ($p<0.001$).

Minimax in the Selection and Expansion Phases

The variant of MCTS-MS we tested starts a minimax search from a state in the tree if that state has reached a fixed number of visits when encountered by the selection policy. We call this variant, using a minimax search of depth d when reaching v visits, *MCTS-MS-d-Visit-v*. If the visit limit is set to 0, this means every tree node is searched

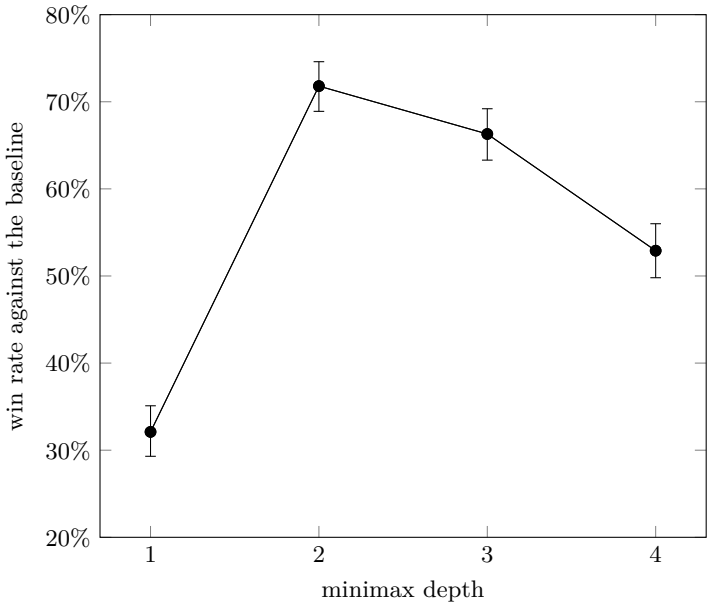


Figure 7.11: Performance of MCTS-MR in Connect-4.

immediately in the expansion phase even before it is added to the tree.

We tested MCTS-MS-d-Visit-v for $d \in \{2, 4\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100\}$. We found it to be most effective to set the $\alpha\beta$ search window such that minimax was only used to detect forced losses (traps). Since suicide is impossible in Connect-4, we only searched for even depths. Each condition consisted of 1000 games against the baseline player. The results are shown in Figure 7.12. Low values of v result in too many minimax searches being triggered, which slows down MCTS. High values of v mean that the tree below the node in question has already been expanded to a certain degree, and minimax might not be able to gain much new information. Additionally, high values of v result in too few minimax searches, such that they have little effect.

MCTS-MS-2-Visit-1 was the most successful condition. It played about 83700 simulations per second on average over an entire game. There were 5000 additional games played against the baseline and a total win rate of 53.6% (95% confidence interval: $52.2\% - 55.0\%$) was achieved, which is a significantly stronger performance ($p<0.001$).

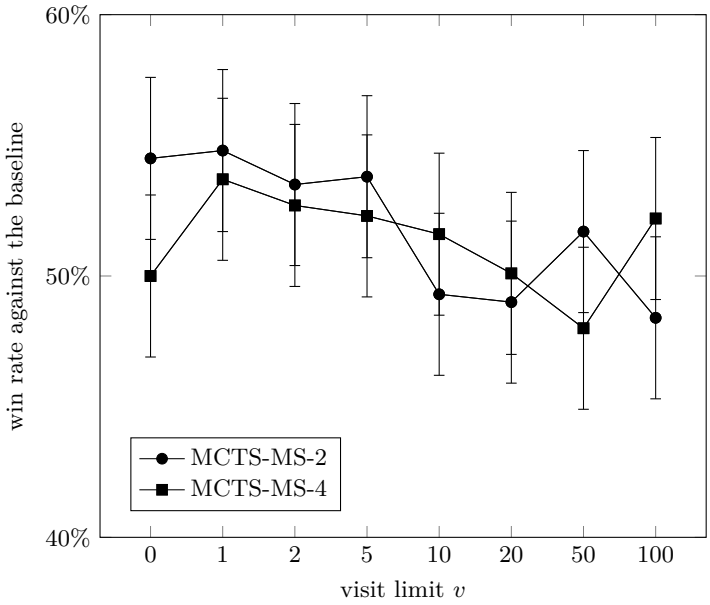


Figure 7.12: Performance of MCTS-MS in Connect-4.

Minimax in the Backpropagation Phase

MCTS-Solver with minimax in the backpropagation phase was tested with minimax search depths 1 ply to 6 plies. Contrary to MCTS-MS as described in 7.4.2, we experimentally determined it to be most effective to use MCTS-MB with a full minimax search window in order to detect both wins and losses. We therefore included odd search depths. Again, all moves from a given node were searched independently in order to be able to prove their individual game-theoretic values. The resulting players were abbreviated as MCTS-MB-1 to MCTS-MB-6 and played 1000 games each against the regular MCTS-Solver baseline. The results are shown in Figure 7.13.

MCTS-MB-1 as the best-performing variant played 5000 additional games against the baseline and won 49.9% (95% confidence interval: $48.5\% - 51.3\%$) of them, which shows no significant difference in performance. It played about 88500 simulations per second when averaged over the whole game.

7.4.3 Breakthrough

The experimental results in the 6×6 Breakthrough domain are described in this subsection. Our baseline MCTS-Solver implementation plays about 45100 simulations per second on average.

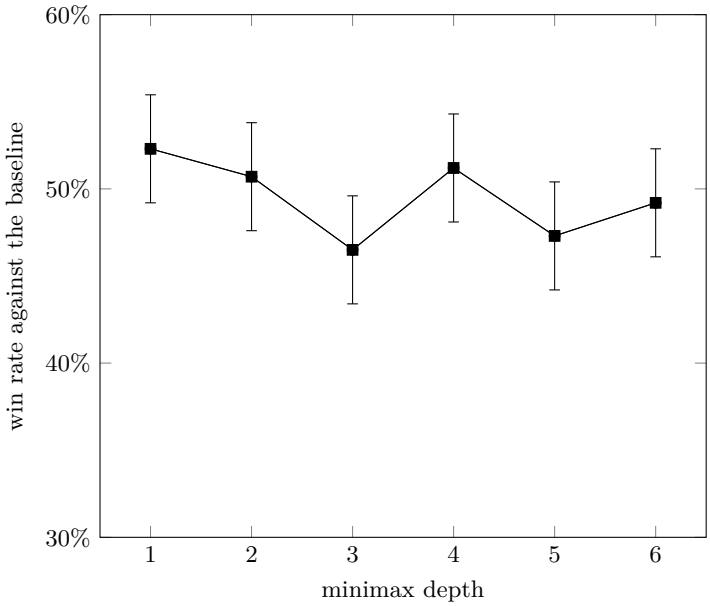


Figure 7.13: Performance of MCTS-MB in Connect-4.

Minimax in the Rollout Phase

As in Connect-4, we tested 1-ply to 4-ply minimax searches in the rollout phase of a Breakthrough MCTS-Solver player. The resulting players MCTS-MR-1 to MCTS-MR-4 played 1000 games each against regular MCTS-Solver with uniformly random rollouts. The results are presented in Figure 7.14.

Interestingly, all MCTS-MR players were significantly weaker than the baseline ($p<0.001$). The advantage of a 1- to 4-ply lookahead in rollouts does not seem to outweigh the computational cost in Breakthrough, possibly due to the larger branching factor, longer rollouts, and more time-consuming move generation than in Connect-4. MCTS-MR-1 searches only about 15.8% as fast as the baseline, MCTS-MR-2 about 2.3% as fast, MCTS-MR-3 about 0.5% as fast, MCTS-MR-4 about 0.15% as fast when measured in simulations completed in a one-second search of the empty Connect-4 board. When comparing with equal numbers of MCTS iterations (10000) per move instead of equal time (1 second) per move for both players, MCTS-MR-1 achieved a win rate of 67.6% in 1000 games against the baseline. MCTS-MR-2 won 83.2% of 1000 games under the same conditions. It may be possible to optimize our Breakthrough implementation. However, as the following subsections indicate, application of minimax in other phases of MCTS seems to be the more promising approach in this game.

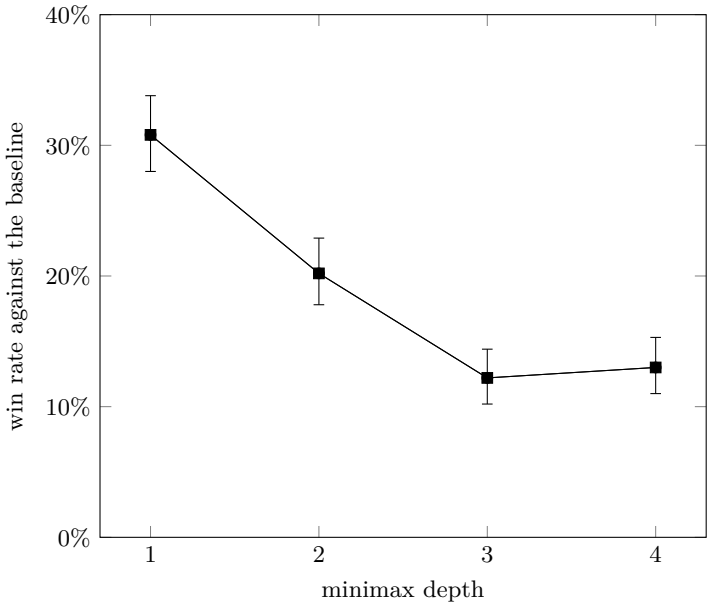


Figure 7.14: Performance of MCTS-MR in Breakthrough.

Minimax in the Selection and Expansion Phases

We tested the same variants of MCTS-MS for Breakthrough as for Connect-4: MCTS-MS-d-Visit-v for $d \in \{2, 4\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$. 1000 games against the baseline player were played for each experimental condition. Figure 7.15 shows the results.

MCTS-MS-2-Visit-2 appeared to be the most effective variant. When averaged over the whole game, it performed about 33000 simulations per second. 5000 additional games against the baseline confirmed a significant increase in strength ($p<0.001$) with a win rate of 67.3% (95% confidence interval: $66.0\% - 68.6\%$).

Minimax in the Backpropagation Phase

MCTS-MB-1 to MCTS-MB-6 were tested analogously to Connect-4, playing 1000 games each against the regular MCTS-Solver baseline. Figure 7.16 presents the results.

The best-performing setting MCTS-MB-2 played 5000 additional games against the baseline and won 60.6% (95% confidence interval: $59.2\% - 62.0\%$) of them, which shows a significant improvement ($p<0.001$). It played about 46800 simulations per second on average.

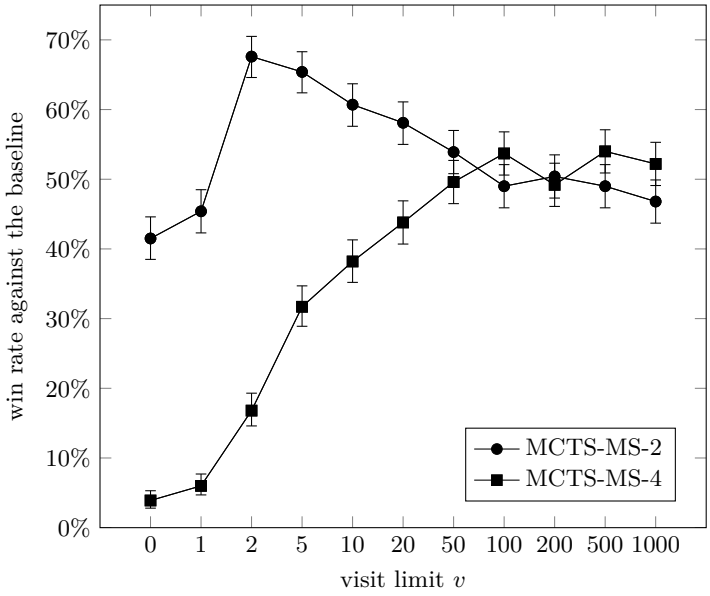


Figure 7.15: Performance of MCTS-MS in Breakthrough.

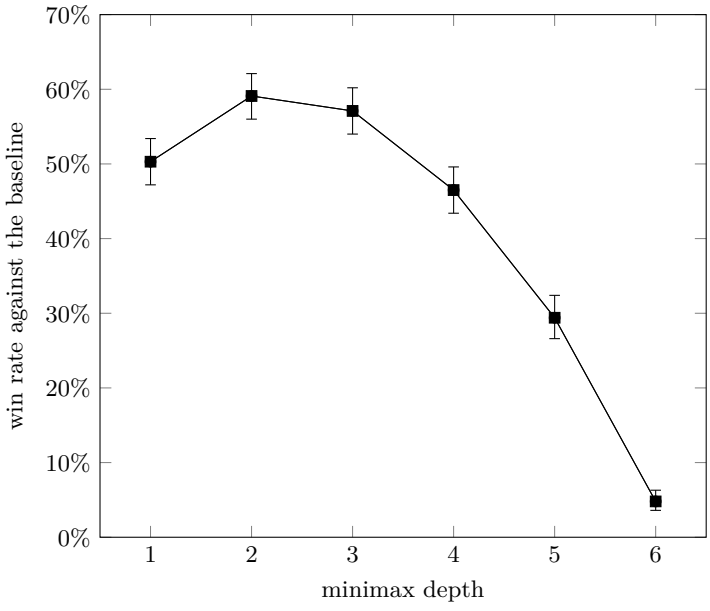


Figure 7.16: Performance of MCTS-MB in Breakthrough.

7.4.4 Othello

This subsection describes the experimental results in Othello. Our baseline MCTS-Solver implementation plays about 8700 simulations per second on average in this domain.

Minimax in the Rollout Phase

Figure 7.17 presents the results of MCTS-MR-1 to MCTS-MR-4 playing 1000 games each against the MCTS-Solver baseline. None of the MCTS-MR conditions tested had a positive effect on playing strength. The best-performing setting MCTS-MR-1 played 5000 additional games against the baseline and won 43.7% (95% confidence interval: $42.3\% - 45.1\%$) of them, which is significantly weaker than the baseline ($p<0.001$). MCTS-MR-1 simulated about 6400 games per second on average.

When playing with equal numbers of MCTS iterations per move, MCTS-MR-1 won 47.9% of 1000 games against the baseline (at 5000 rollouts per move), MCTS-MR-2 won 54.3% (at 1000 rollouts per move), MCTS-MR-3 won 58.2% (at 250 rollouts per move), and MCTS-MR-4 won 59.6% (at 100 rollouts per move). This shows that there is a positive effect—ignoring the time overhead—of minimax searches in the rollouts, even though these searches can only return useful information in the very last moves of an Othello game. It could be worthwhile in Othello and similar games to restrict MCTS-MR to minimax searches only in these last moves.

Minimax in the Selection and Expansion Phases

Again, MCTS-MS-d-Visit-v was tested for $d \in \{2, 4\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$. Each condition played 1000 games against the baseline player. Figure 7.18 presents the results.

The best-performing version, MCTS-MS-2-Visits-50, won 50.8% (95% confidence interval: $49.4\% - 52.2\%$) of 5000 additional games against the baseline. Thus, no significant difference in performance was found. The speed was about 8200 rollouts per second.

Minimax in the Backpropagation Phase

MCTS-MB-1 to MCTS-MB-6 played 1000 games each against the regular MCTS-Solver baseline. The results are shown in Figure 7.19.

MCTS-MB-2, the most promising setting, achieved a result of 49.2% (95% confidence interval: $47.8\% - 50.6\%$) in 5000 additional games against the baseline. No significant performance difference to the baseline could be shown. The hybrid played about 8600 rollouts per second on average.

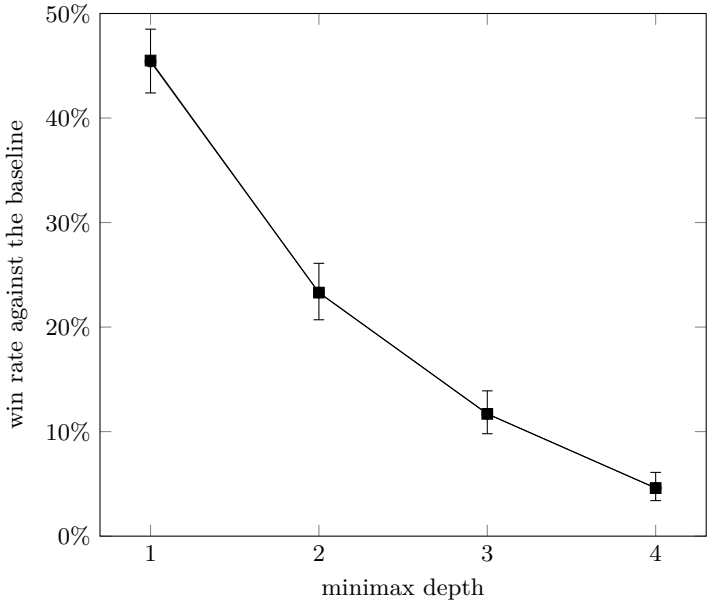


Figure 7.17: Performance of MCTS-MR in Othello.

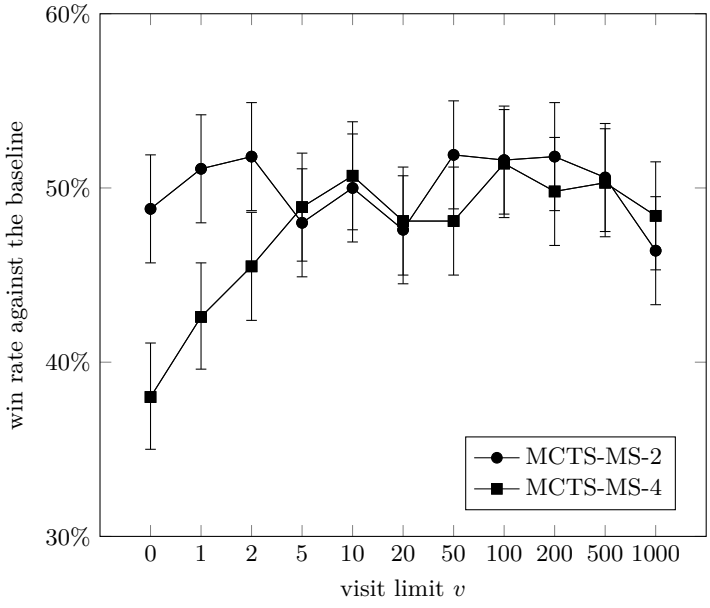


Figure 7.18: Performance of MCTS-MS in Othello.

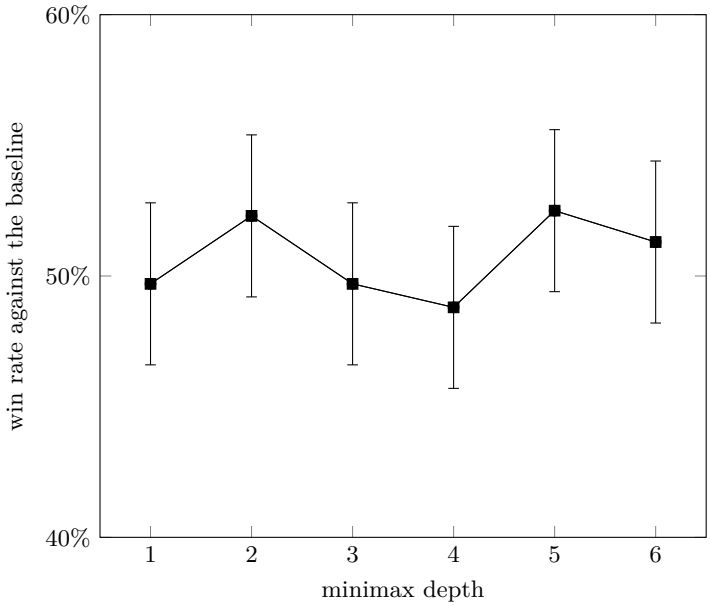


Figure 7.19: Performance of MCTS-MB in Othello.

In conclusion, no effect of the three tested MCTS-minimax hybrids can be shown in Othello, the domain with the lowest number of traps examined in this chapter.

7.4.5 Catch the Lion

In this subsection, the results of testing in the domain of Catch the Lion are presented. In this game, the baseline MCTS-Solver plays approximately 34700 simulations per second on average.

Minimax in the Rollout Phase

Figure 7.20 presents the results of MCTS-MR-1 to MCTS-MR-4 playing 1000 games each against the MCTS-Solver baseline.

There is an interesting even-odd effect, with MCTS-MR seemingly playing stronger at odd minimax search depths. Catch the Lion is the only domain of the four where MCTS-MR-3 was found to perform better than MCTS-MR-2. MCTS-MR-1 played best in these initial experiments and was tested in an additional 5000 games against the baseline. The hybrid won 94.8% (95% confidence interval: $94.1\% - 95.4\%$) of these, which is significantly stronger ($p<0.001$). It reached about 28700 rollouts per second.

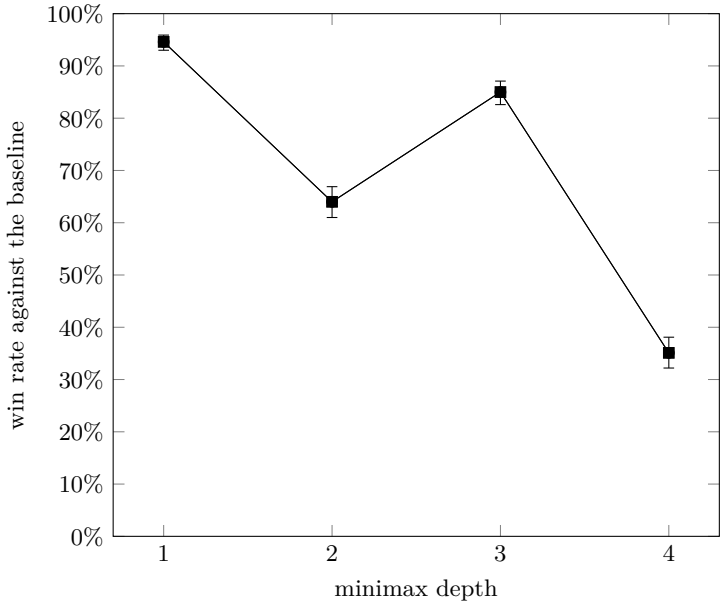


Figure 7.20: Performance of MCTS-MR in Catch the Lion.

Minimax in the Selection and Expansion Phases

MCTS-MS-d-Visit-v was tested for $d \in \{2, 4, 6\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100\}$. With each parameter setting, 1000 games were played against the baseline player. Figure 7.21 shows the results.

MCTS-MS-4-Visits-2 performed best of all tested settings. Of an additional 5000 games against the baseline, it won 76.8% (95% confidence interval: $75.6\% - 78.0\%$). This is a significant improvement ($p<0.001$). It played about 14500 games per second on average.

Minimax in the Backpropagation Phase

MCTS-MB-1 to MCTS-MB-6 were tested against the baseline in 1000 games per condition. Figure 7.22 presents the results.

MCTS-MB-4 performed best and played 5000 more games against the baseline. It won 73.1% (95% confidence interval: $71.8\% - 74.3\%$) of them, which is a significant improvement ($p<0.001$). The speed was about 20000 rollouts per second.

In conclusion, the domain with the highest number of traps examined in this chapter, Catch the Lion, also showed the strongest performance of all three MCTS-minimax hybrids.

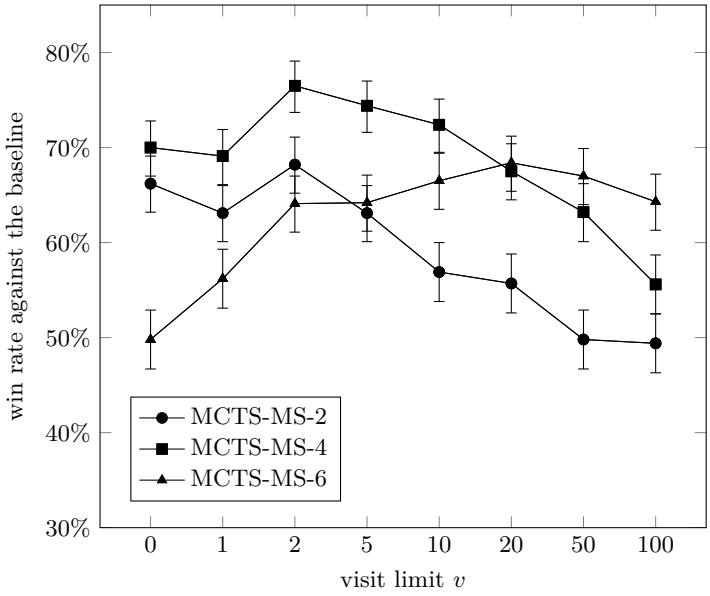


Figure 7.21: Performance of MCTS-MS in Catch the Lion.

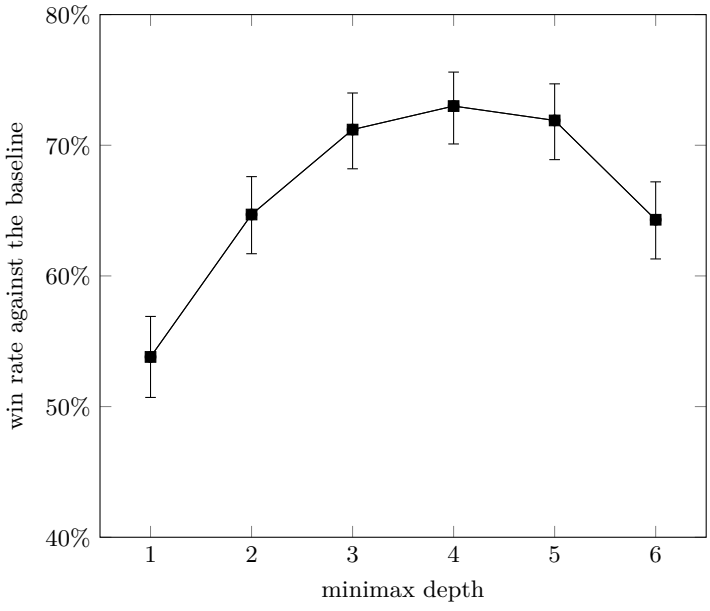


Figure 7.22: Performance of MCTS-MB in Catch the Lion.

7.4.6 Comparison of Algorithms

Sections 7.4.2 to 7.4.5 showed the performance of MCTS-MR, MCTS-MS, and MCTS-MB against the baseline player in the four test domains. In order to facilitate comparison, we also tested the best-performing variants of these MCTS-minimax hybrids against each other. In Connect-4, MCTS-MR-2, MCTS-MS-2-Visit-1, and MCTS-MB-1 played in each possible pairing; in Breakthrough, MCTS-MR-1, MCTS-MS-2-Visit-2, and MCTS-MB-2 were chosen; in Othello, MCTS-MR-1, MCTS-MS-2-Visit-50, and MCTS-MB-2; and in Catch the Lion, MCTS-MR-1, MCTS-MS-4-Visits-2, and MCTS-MB-4. 2000 games were played in each condition. Figure 7.23 presents the results.

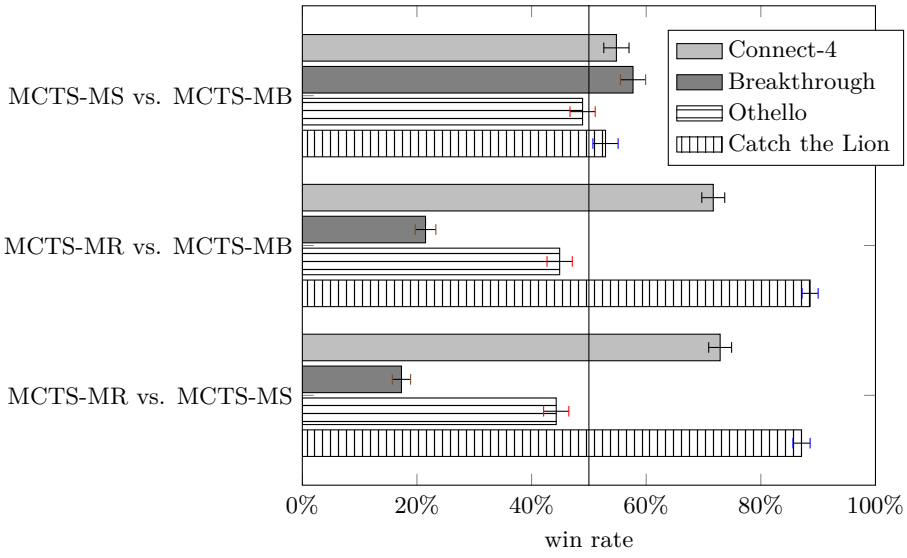


Figure 7.23: Performance of MCTS-MR, MCTS-MS, and MCTS-MB against each other in Connect-4, Breakthrough, Othello, and Catch the Lion.

Consistent with the results from the previous sections, MCTS-MS outperformed MCTS-MB in Breakthrough and Connect-4, while no significant difference could be shown in Othello and Catch the Lion. MCTS-MR was significantly stronger than the two other algorithms in Connect-4 and Catch the Lion, but weaker than both in Breakthrough and Othello.

7.4.7 Comparison of Domains

In Subsections 7.4.2 to 7.4.5, the experimental results were ordered by domain. In this subsection, the data on the best-performing hybrid variants are presented again, ordered by the type of hybrid instead. Figures 7.24, 7.25, and 7.26 show the performance of MCTS-MS, MCTS-MB, and MCTS-MR, respectively.

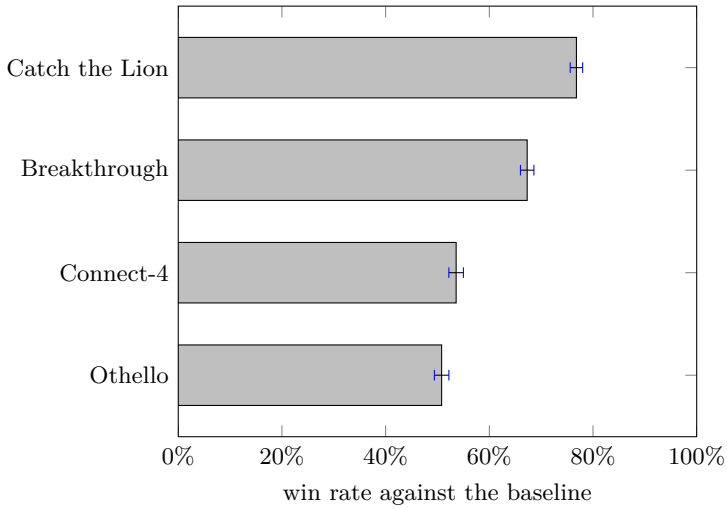


Figure 7.24: Comparison of MCTS-MS performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

Both MCTS-MS and MCTS-MB are most effective in Catch the Lion, followed by Breakthrough, Connect-4, and finally Othello, where no positive effect could be observed. The parallels to the ordering of domains with respect to tacticality, i.e. trap density (Figure 7.8) and trap difficulty (Figure 7.9), are striking. As expected, these factors seem to strongly influence the relative effectivity of MCTS-minimax hybrids in a given domain. This order is different in MCTS-MR only due to the poor performance in Breakthrough, which may be explained by this domain having a higher average branching factor than the other three (see Table 6.14).

7.4.8 Effect of Time Settings

The results presented in the previous subsections of this chapter were all based on a time setting of 1000ms per move. In this set of experiments, the best-performing MCTS-minimax hybrids played against the baseline at different time settings from

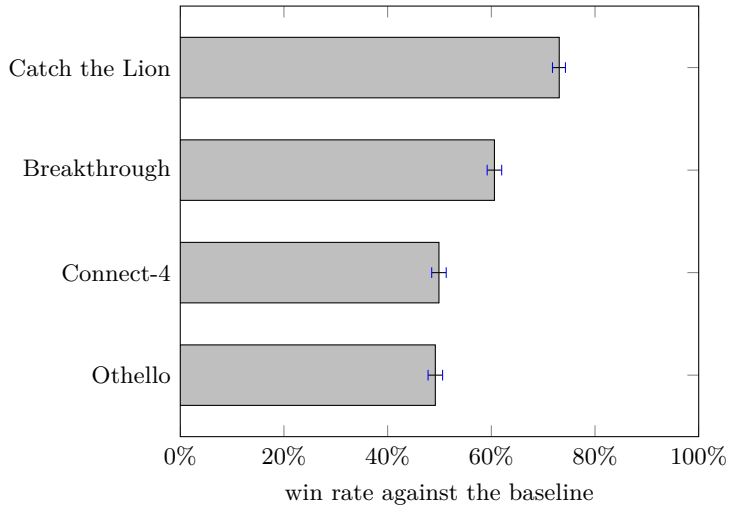


Figure 7.25: Comparison of MCTS-MB performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

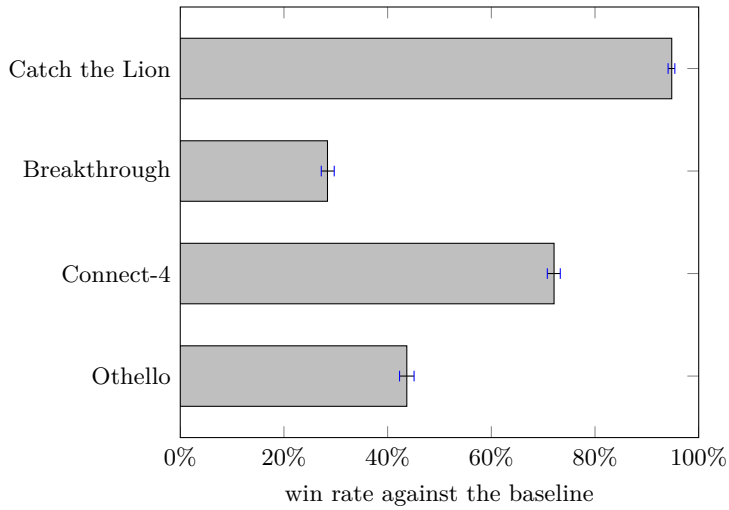


Figure 7.26: Comparison of MCTS-MR performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

250 ms per move to 5000 ms per move. 2000 games were played in each condition. The results are shown in Figure 7.27 for Connect-4, Figure 7.28 for Breakthrough, Figure 7.29 for Othello, and Figure 7.30 for Catch the Lion.

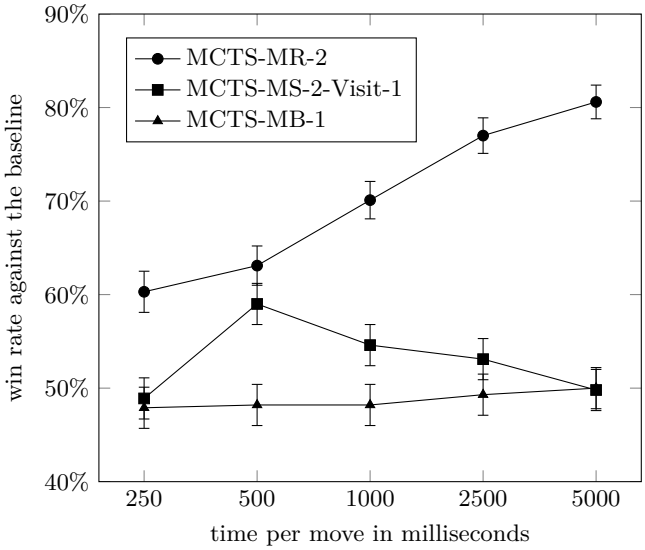


Figure 7.27: Performance of MCTS-MR-2, MCTS-MS-2-Visit-1, and MCTS-MB-1 at different time settings in Connect-4.

We can observe that at least up to 5 seconds per move, additional time makes the significant performance differences between algorithms more pronounced in most domains. While in Connect-4, it is MCTS-MR that profits most from additional time, we can see the same effect for MCTS-MS and MCTS-MB in Breakthrough. The time per move does not change the ineffectiveness of hybrid search in Othello. Interestingly however, MCTS-MR does not profit from longer search times in Catch the Lion. It is possible that in this highly tactical domain, the baseline MCTS-Solver scales better due to being faster and building larger trees. The larger trees could help avoid deeper traps than the MCTS-MR rollouts can detect.

7.4.9 Effect of Branching Factor

In order to shed more light on the influence of the average branching factor mentioned above, the hybrids were also tested on Breakthrough boards of larger widths. In addition to the 6×6 board used in the previous experiments (average branching factor 15.5), we included the sizes 9×6 (average branching factor 24.0), 12×6 (average

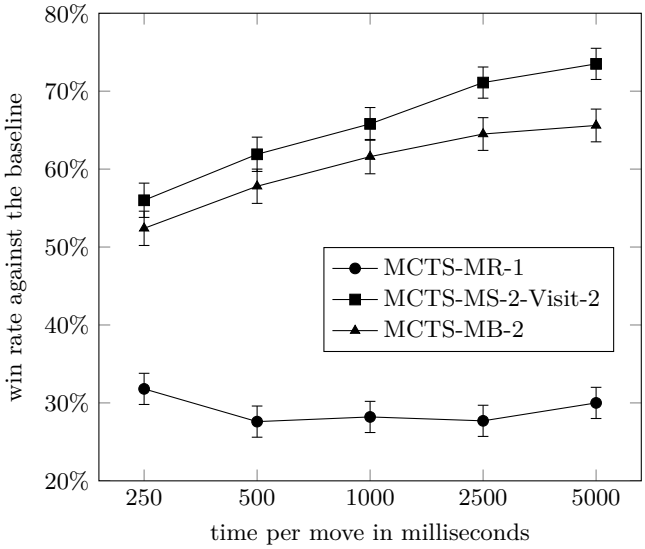


Figure 7.28: Performance of MCTS-MR-1, MCTS-MS-2-Visit-2, and MCTS-MB-2 at different time settings in Breakthrough.

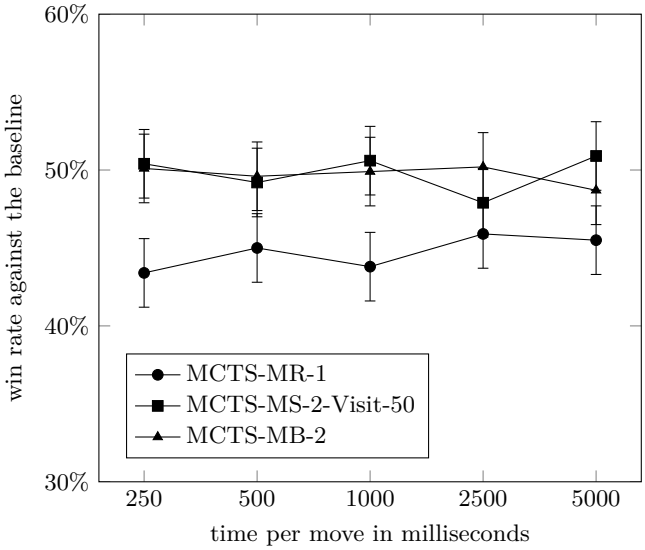


Figure 7.29: Performance of MCTS-MR-1, MCTS-MS-2-Visit-50, and MCTS-MB-2 at different time settings in Othello.

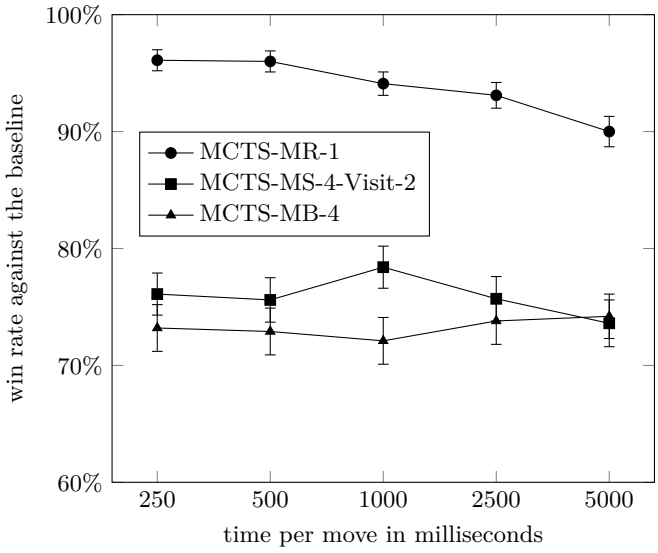


Figure 7.30: Performance of MCTS-MR-1, MCTS-MS-4-Visit-2, and MCTS-MB-4 at different time settings in Catch the Lion.

branching factor 37.5), 15×6 (average branching factor 43.7), and 18×6 (average branching factor 54.2) in this series of experiments. While the average game length also increases with the board width—from about 30 moves on the 6×6 board to about 70 moves on the 18×6 board—this setup served as an approximation to varying the branching factor while keeping other game properties as equal as possible (without using artificial game trees). Figure 7.31 presents the results, comparing the best-performing settings of MCTS-MS, MCTS-MB, and MCTS-MR across the five board sizes. The hybrids were tuned for each board size separately. Each data point represents 2000 games.

The branching factor has a strong effect on MCTS-MR, reducing the win rate of MCTS-MR-1 from 30.8% (on 6×6) to 10.2% (on 18×6). Deeper minimax searches in MCTS rollouts scale even worse: The performance of MCTS-MR-2 for example drops from 20.2% to 0.1% (not shown in the Figure). The MCTS-minimax hybrids newly proposed in this chapter, however, do not seem to be strongly affected by the range of branching factors examined. Both MCTS-MS and MCTS-MB are effective up to a branching factor of at least 50.

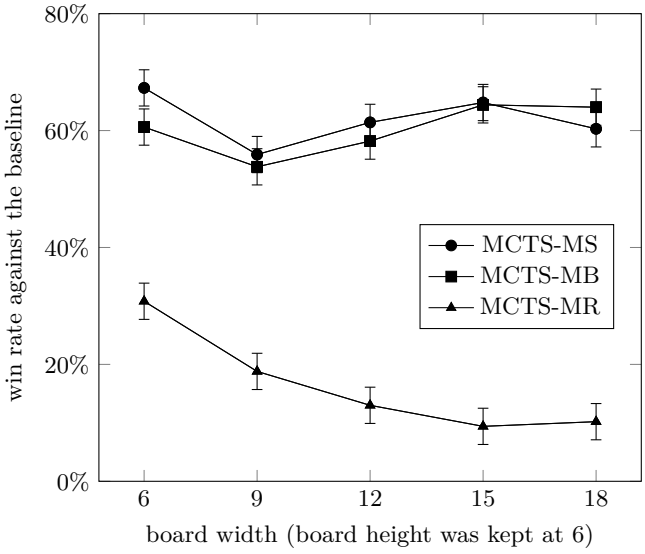


Figure 7.31: Performance of MCTS-minimax hybrids across different board widths in Breakthrough.

7.4.10 Solving Performance

The previous subsections analyzed the playing performance of the MCTS-minimax hybrids. This subsection presents results on their solving performance in order to determine whether their playing strength is related to more effective solving of game positions.

For each domain, we collected 1000 test positions suitable for solving by playing 1000 self-play games with MCTS-Solver. We stored the first endgame position of each game whose game-theoretic value could be determined by the player to move—500 times the first solved position for the first player, and 500 times the first solved position for the second player. The time setting was 5 seconds per move.

Next, the solving performance of the best-performing hybrids in each domain was tested by giving them 5 seconds to solve each of these 1000 test positions. The MCTS-Solver baseline completed the same task. In addition to the number of test positions successfully solved by each player, we determined for each domain the average number of nodes and the average time in milliseconds needed to solve the subset of positions that was solved by all players. Tables 7.1 to 7.4 show the results. In the following, we relate the solving performance to the playing performance of the MCTS-minimax hybrids.

Table 7.1: Solving performance of the MCTS-minimax hybrids in Othello.

Player	Solved positions	For the 822 positions solved by all players	
		Average nodes	Average time in ms
MCTS-Solver	900	41158	1881
MCTS-MR	883	41404	2155
MCTS-MS	890	41426	1928
MCTS-MB	921	38202	1879

In Othello, only MCTS-MB appears to be more effective in solving than the baseline. MCTS-MB does not play better than MCTS-Solver, but its improved backpropagation of game-theoretic values is helpful in endgame positions such as those tested here. The other hybrids are not improving on the baseline in either playing or solving.

In Connect-4, all of the hybrids are inferior to the MCTS-Solver baseline in solving. On the one hand, MCTS-MS and MCTS-MR, the two hybrids found to outperform the baseline in playing, need fewer nodes to solve the positions solved by all players. On the other hand, they cannot solve as many positions as the baseline. In the case of MCTS-MR this can be explained by slower tree growth. MCTS-MS may grow the tree in a way that avoids terminal positions as long as possible, making it more difficult to backpropagate them in the tree.

In Breakthrough, all hybrids are much stronger than the baseline in solving endgame positions. They solve more positions and do so with fewer nodes and in less time. Whereas the expensive rollout policy of MCTS-MR makes it an inferior player, it does not pose as much of a problem in solving since endgame positions require only relatively short rollouts.

In Catch the Lion, all hybrids are much more effective than the baseline not only in playing, but also in solving. As in Breakthrough, they solve more positions than the baseline, and need fewer nodes and less time to do so.

In conclusion, it seems that the MCTS-minimax hybrids are most likely to improve both solving strength and playing strength in the more tactical domains such as Catch the Lion and Breakthrough. Solving and playing do not necessarily favor the same type of hybrid however, due to differences such as much shorter rollouts in the case of endgame positions.

Note that for all solving experiments presented so far, the hybrids were not tuned for optimal solving performance, but the settings previously found to be optimal for playing were used instead. This helped us understand whether performance improvements in playing can be explained by or related to performance improvements

Table 7.2: Solving performance of the MCTS-minimax hybrids in Connect-4.

Player	Solved positions	For the 433 positions solved by all players	
		Average nodes	Average time in ms
MCTS-Solver	897	65675	2130
MCTS-MR	677	46464	2189
MCTS-MS	597	43890	3586
MCTS-MB	888	65999	2270

Table 7.3: Solving performance of the MCTS-minimax hybrids in Breakthrough.

Player	Solved positions	For the 717 positions solved by all players	
		Average nodes	Average time in ms
MCTS-Solver	812	47400	1645
MCTS-MR	913	7426	1210
MCTS-MS	922	7418	842
MCTS-MB	921	12890	962

Table 7.4: Solving performance of the MCTS-minimax hybrids in Catch the Lion.

Player	Solved positions	For the 599 positions solved by all players	
		Average nodes	Average time in ms
MCTS-Solver	678	10227	1445
MCTS-MR	912	1086	537
MCTS-MS	932	410	500
MCTS-MB	891	1049	654

Table 7.5: Solving performance of MCTS-MB-2 in Connect-4.

Player	Solved positions	For the 865 positions solved by both players	
		Average nodes	Average time in ms
MCTS-Solver	889	84939	2705
MCTS-MB-2	919	72832	2536

in proving. However, the optimal parameter settings for playing are not always identical to the optimal parameter settings for solving. The reason that MCTS-MB improves solving in Othello, but not in Connect-4, for instance, might simply be due to the fact that the best-playing Othello setting MCTS-MB-2 solves better than the best-playing Connect-4 setting MCTS-MB-1 in all domains. A preliminary experiment was conducted with MCTS-MB in Connect-4 to demonstrate this difference between correct tuning for playing and for solving. Comparing Tables 7.2 and 7.5 shows how much more effective MCTS-MB-2 is than MCTS-MB-1 for the solving of Connect-4 positions, while it is no stronger at playing (see Figure 7.13).

7.5 Conclusion and Future Research

The strategic strength of MCTS lies to a great extent in the Monte-Carlo simulations, allowing the search to observe even distant consequences of actions, if only through the observation of probabilities. The tactical strength of minimax lies largely in its exhaustive approach, guaranteeing to never miss a consequence of an action that lies within the search horizon, and backing up game-theoretic values from leaves with certainty and efficiency.

In this chapter, we examined three knowledge-free approaches of integrating minimax into MCTS: the application of minimax in the rollout phase with *MCTS-MR*, the selection and expansion phases with *MCTS-MS*, and the backpropagation phase with *MCTS-MB*. The newly proposed variant MCTS-MS significantly outperformed regular MCTS with the MCTS-Solver extension in Catch the Lion, Breakthrough, and Connect-4. The same holds for the proposed MCTS-MB variant in Catch the Lion and Breakthrough, while the effect in Connect-4 is neither significantly positive nor negative. The only way of integrating minimax search into MCTS known from the literature (although typically used with an evaluation function), MCTS-MR, was quite strong in Catch the Lion and Connect-4 but significantly weaker than the baseline in Breakthrough, suggesting it might be less robust with regard to differences between domains such as the average branching factor. As expected, none of the MCTS-

minimax hybrids had a positive effect in Othello. The game of Go would be another domain where we do not expect any success with knowledge-free MCTS-minimax hybrids, because it has no trap states until the latest game phase.

With the exception of the weak performance of MCTS-MR in Breakthrough, probably mainly caused by its larger branching factor, we observed that all MCTS-minimax hybrids tend to be most effective in Catch the Lion, followed by Breakthrough, Connect-4, and finally Othello. The density and difficulty of traps, as discussed in Subsection 7.4.1, thus seem to predict the relative performance of MCTS-minimax hybrids across domains well. Additional experiments showed that the hybrids are also overall most effective in Catch the Lion and Breakthrough when considering the task of solving endgame positions—a task that e.g. alleviates the problems of MCTS-MR in Breakthrough. In conclusion, MCTS-minimax hybrids can strongly improve the performance of MCTS in tactical domains, with MCTS-MR working best in domains with low branching factors (up to roughly 10 moves on average), and MCTS-MS and MCTS-MB being more robust against higher branching factors. This was tested for branching factors of up to roughly 50 in Breakthrough on different boards.

According to our observations, problematic domains for MCTS-minimax hybrids seem to feature a low density of traps in the search space, as in Othello, or in the case of MCTS-MR a relatively high branching factor, as in Breakthrough. In the next chapter, these problems will be addressed with the help of domain knowledge. On the one hand, domain knowledge can be incorporated into the hybrid algorithms in the form of *evaluation functions*. This can make minimax potentially much more useful in search spaces with few terminal nodes before the latest game phase, such as that of Othello. On the other hand, domain knowledge can be incorporated in the form of a *move ordering function*. This can be effective in games such as Breakthrough, where traps are relatively frequent, but the branching factor seems to be too high for some hybrids such as MCTS-MR. Here, the overhead of embedded minimax searches can be reduced by only taking the highest-ranked moves into account (Nijssen and Winands, 2012; Winands and Björnsson, 2011).

Preliminary experiments with combinations of the three knowledge-free hybrids seem to indicate that their effects are overlapping to a large degree. Combinations do not seem to perform significantly better than the best-performing individual hybrids in the domain at hand. This could still be examined in more detail.

Note that in all experiments except those concerning MCTS-MR, we used fast, uniformly random rollout policies. On the one hand, the overhead of our techniques would be proportionally lower for any slower, informed rollout policies such as typically used in state-of-the-art programs. On the other hand, improvement on already strong policies might prove to be more difficult. Examining the influence of such MCTS enhancements is a possible second direction of future research.

Finally, while we have focused primarily on the game tree properties of trap density and difficulty as well as the average branching factor in this chapter, the impact of other properties such as the game length or the distribution of terminal values also deserve further study. Artificial game trees could be used to study these properties in isolation—the large number of differences between "real" games potentially confounds many effects, such as Breakthrough featuring more traps throughout the search space than Othello, but also having a larger branching factor. Eventually, it might be possible to learn from the success of MCTS-minimax hybrids in Catch the Lion, and transfer some ideas to larger games of similar type such as Shogi and Chess.

8

MCTS and Minimax Hybrids with Heuristic Evaluation Functions

This chapter is based on:

Baier, H. and Winands, M. H. M. (2014). Monte-Carlo Tree Search and Minimax Hybrids with Heuristic Evaluation Functions. In T. Cazenave, M. H. M. Winands, and Y. Björnsson, editors, *Computer Games Workshop at 21st European Conference on Artificial Intelligence, ECAI 2014*, volume 504 of *Communications in Computer and Information Science*, pages 45–63.

In the previous chapter, *MCTS-minimax hybrids* have been introduced, embedding shallow minimax searches into the MCTS framework. This was a first step towards combining the strategic strength of MCTS with the tactical strength of minimax, especially for highly tactical domains where the selective search of MCTS can lead to missing important moves and falling into traps. The results of the hybrid algorithms MCTS-MR, MCTS-MS, and MCTS-MB have been promising even without making use of domain knowledge such as heuristic evaluation functions. However, their inability to evaluate non-terminal states makes them ineffective in games with very few or no terminal states throughout the search space, such as the game of Othello. Furthermore, some form of domain knowledge is often available in practice, and it is an interesting question how to use it to maximal effect. This chapter continues to answer the fourth research question by addressing the case where domain knowledge is available.

The enhancements discussed in this chapter make use of *state evaluations*. These state evaluations can either be the result of simple evaluation function calls, or the result of minimax searches using the same evaluation function at the leaves. Three different approaches for integrating state evaluations into MCTS are considered. The

first approach uses state evaluations to choose rollout moves (MCTS-*IR* for *informed rollouts*). The second approach uses state evaluations to terminate rollouts early (MCTS-*IC* for *informed cutoffs*). The third approach uses state evaluations to bias the selection of moves in the MCTS tree (MCTS-*IP* for *informed priors*). Using minimax with $\alpha\beta$ to compute state evaluations means accepting longer computation times in favor of typically more accurate evaluations as compared to simple evaluation function calls. Only in the case of MCTS-IR, minimax has been applied before; the use of minimax for the other two approaches is newly proposed in the form described here. The MCTS-minimax hybrids are tested and compared to their counterparts using evaluation functions without minimax in the domains of Othello, Breakthrough, and Catch the Lion.

Since the branching factor of a domain is identified as a limiting factor of the hybrids' performance, further experiments are conducted using domain knowledge not only for state evaluation, but also for *move ordering*. Move ordering reduces the average size of $\alpha\beta$ trees, and furthermore allows to restrict the effective branching factor of $\alpha\beta$ to only the k most promising moves in any given state (see Chapter 2). Again, this has only been done for MCTS with minimax rollouts before. The enhanced MCTS-minimax hybrids with move ordering and k-best pruning are tested and compared to the unenhanced hybrids as well as the equivalent algorithms using static evaluations in all three domains.

This chapter is structured as follows. Section 8.1 provides a brief overview of related work on algorithms combining features of MCTS and minimax, and on using MCTS with heuristics. Section 8.2 outlines three different methods for incorporating heuristic evaluations into the MCTS framework, and presents variants using shallow-depth minimax searches for each of these. Two of these MCTS-minimax hybrids are newly proposed in this chapter. Section 8.3 shows experimental results of the MCTS-minimax hybrids without move ordering and k-best pruning in the test domains of Othello, Breakthrough, and Catch the Lion. Section 8.4 adds results of introducing move ordering and k-best pruning to each hybrid in each domain. Section 8.5 concludes and suggests future research.

8.1 Related Work

Previous work on developing algorithms influenced by both MCTS and minimax has taken two principal approaches. On the one hand, one can extract individual features of minimax such as minimax-style backups and integrate them into MCTS. This approach was chosen e.g. in Ramanujan and Selman (2011), where the algorithm $UCTMAX_H$ replaces MCTS rollouts with heuristic evaluations and classic averaging MCTS backups with minimaxing backups. In *implicit minimax backups* (Lanctot et al.,

2014), both minimaxing backups of heuristic evaluations and averaging backups of rollout returns are managed simultaneously. On the other hand, one can nest minimax searches into MCTS searches. This is the approach taken in this and the previous chapter.

The idea of improving Monte-Carlo rollouts with the help of heuristic domain knowledge has first been applied to games by Bouzy (2005). It is now used by state-of-the-art programs in virtually all domains. Shallow minimax in every step of the rollout phase has been proposed as well, e.g. a 1-ply search in Lorentz (2011) for the game of Havannah, or a 2-ply search for Lines of Action (Winands and Björnsson, 2011), Chess (Ramanujan et al., 2010b), and multi-player games (Nijssen and Winands, 2012). Similar techniques are considered in Subsection 8.2.1.

The idea of stopping rollouts before the end of the game and backpropagating results on the basis of heuristic knowledge has been explored in Amazons (Lorentz, 2008), Lines of Action (Winands et al., 2010), and Breakthrough (Lorentz and Horey, 2014). To the best of our knowledge, it was first described in a naive Monte Carlo context (without tree search) by Sheppard (2002). A similar method is considered in Subsection 8.2.2, where we also introduce a hybrid algorithm replacing the evaluation function with a minimax call. Our methods are different from Lorentz (2008) and Winands et al. (2010) as we backpropagate the actual heuristic values instead of rounding them to losses or wins. They are also different from Winands et al. (2010) as we backpropagate heuristic values after a fixed number of rollout moves, regardless of whether they reach a threshold of certainty.

The idea of biasing the selection policy with heuristic knowledge has been introduced in Gelly and Silver (2007) and Chaslot et al. (2008) for the game of Go. Our implementation is similar to Gelly and Silver (2007) as we initialize tree nodes with knowledge in the form of virtual wins and losses. We also propose a hybrid using minimax returns instead of simple evaluation returns in Subsection 8.2.3.

8.2 Hybrid Algorithms

As in the previous chapter, MCTS-Solver is used as the baseline. This section describes the three different approaches for employing heuristic knowledge within MCTS that we explore in this chapter. For each approach, a variant using simple evaluation function calls and a hybrid variant using shallow minimax searches is considered. Two of the three hybrids are newly proposed in the form described here.

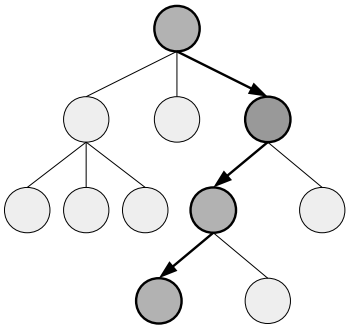
8.2.1 MCTS with Informed Rollouts (MCTS-IR)

The convergence of MCTS to the optimal policy is guaranteed even with uniformly random move choices in the rollouts. However, more informed rollout policies can greatly improve performance (Gelly et al., 2006). When a heuristic evaluation function is available, it can be used in every rollout step to compare the states each legal move would lead to, and choose the most promising one. Instead of choosing this *greedy* move, it is effective in some domains to choose a uniformly random move with a low probability ϵ, so as to avoid determinism and preserve diversity in the rollouts. Our implementation additionally ensures non-deterministic behavior even for $\epsilon = 0$ by picking moves with equal values at random both in the selection and in the rollout phase of MCTS. The resulting rollout policy is typically called ϵ-*greedy* (Sturtevant, 2008). In the context of this work, we call this approach *MCTS-IR-E* (MCTS with informed rollouts using an evaluation function).

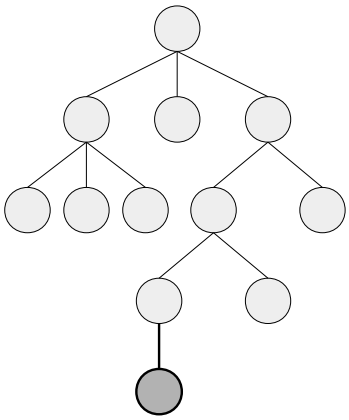
The depth-one lookahead of an ϵ-greedy policy can be extended in a natural way to a depth-d minimax search for every rollout move (Winands and Björnsson, 2011; Nijssen and Winands, 2012). We use a random move ordering in minimax as well in order to preserve non-determinism. In contrast to Winands and Björnsson (2011) and Nijssen and Winands (2012) where several enhancements such as move ordering, k-best pruning, and killer moves were added to $\alpha\beta$, we first test unenhanced $\alpha\beta$ search in Subsection 8.3.2. We are interested in its performance before introducing additional improvements, especially since our test domains have smaller branching factors than e.g. the games Lines of Action (around 30) or Chinese Checkers (around 25-30) used in Winands and Björnsson (2011) and Nijssen and Winands (2012), respectively. Move ordering and k-best pruning are then added in Subsection 8.4.2. Using a depth-d minimax search for every rollout move aims at stronger move choices in the rollouts, which make rollout returns more accurate and can therefore help to guide the growth of the MCTS tree. We call this approach *MCTS-IR-M* (MCTS with informed rollouts using minimax). An example is visualized in Figure 8.1.

8.2.2 MCTS with Informed Cutoffs (MCTS-IC)

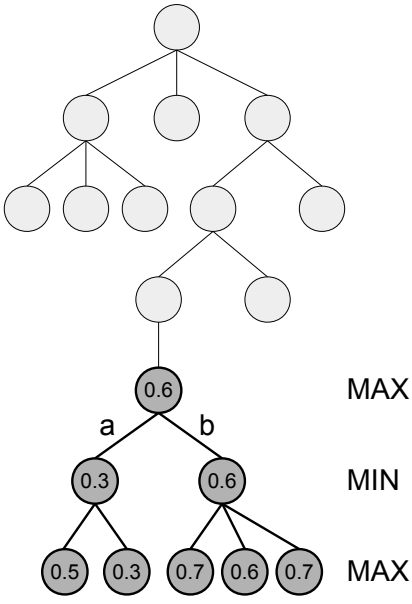
The idea of rollout cutoffs is an early termination of the rollout in case the rollout winner, or the player who is at an advantage, can be reasonably well predicted with the help of an evaluation function. The statistical noise introduced by further rollout moves can then be avoided by stopping the rollout, evaluating the current state of the simulation, and backpropagating the evaluation result instead of the result of a full rollout to the end of the game (Lorentz, 2008; Winands et al., 2010). If on average, the evaluation function is computationally cheaper than playing out the rest of the rollout, this method can also result in an increased sampling speed as measured in rollouts



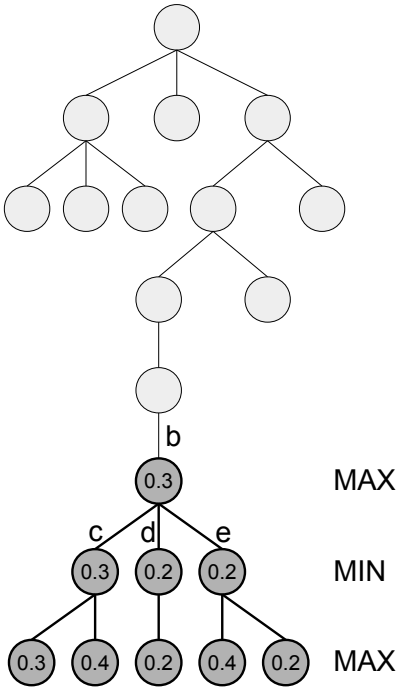
(a) The selection phase.



(b) The expansion phase.



(c) A $d = 2$ minimax search is started to find the first rollout move. The maximizing player chooses move b with a heuristic value of 0.6.



(d) Another $d = 2$ minimax search is conducted for the second rollout move. In this case, the maximizing player chooses move c with a heuristic value of 0.3.

Figure 8.1: The MCTS-IR-M hybrid. $\epsilon = 0$ and $d = 2$.

per second. A fixed number m of rollout moves can be played before evaluating in order to introduce more non-determinism and get more diverse rollout returns. If $m = 0$, the evaluation function is called directly at the newly expanded node of the tree. As in MCTS-IR, our MCTS-IC implementation avoids deterministic gameplay through randomly choosing among equally valued moves in the selection policy. We scale all evaluation values to $[0, 1]$. We do not round the evaluation function values to wins or losses as proposed in Lorentz (2008), nor do we consider the variant with dynamic m and evaluation function thresholds proposed in Winands et al. (2010). In the following, we call this approach *MCTS-IC-E* (MCTS with informed cutoffs using an evaluation function).

We propose an extension of this method using a depth-d minimax search at cutoff time in order to determine the value to be backpropagated. In contrast to the integrated approach taken in Winands and Björnsson (2011), we do not assume MCTS-IR-M as rollout policy and backpropagate a win or a loss whenever the searches of this policy return a value above or below two given thresholds. Instead, we play rollout moves with an arbitrary policy (uniformly random unless specified otherwise), call minimax when a fixed number of rollout moves has been reached, and backpropagate the heuristic value returned by this search. Like MCTS-IR-M, this strategy tries to backpropagate more accurate rollout returns, but by computing them directly instead of playing out the rollout. We call this approach *MCTS-IC-M* (MCTS with informed cutoffs using minimax). An example is shown in Figure 8.2. Subsections 8.3.3 and 8.4.3 present results on the performance of this strategy using $\alpha\beta$ in unenhanced form and with $\alpha\beta$ using move ordering and k-best pruning, respectively.

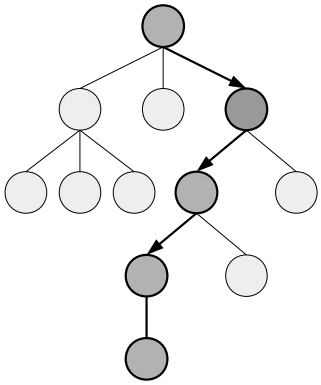
8.2.3 MCTS with Informed Priors (MCTS-IP)

Node priors (Gelly and Silver, 2007) represent one method for supporting the selection policy of MCTS with heuristic information. When a new node is added to the tree, or after it has been visited n times, the heuristic evaluation h of the corresponding state is stored in this node. This is done in the form of virtual wins and virtual losses, weighted by a prior weight parameter γ, according to the following formulas.

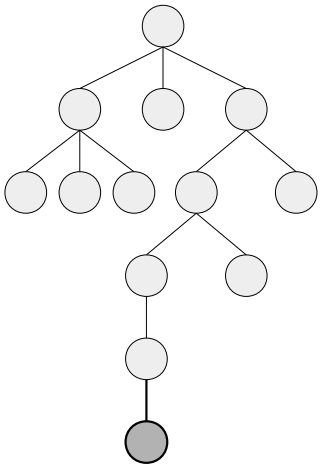
$$v \longleftarrow v + \gamma \tag{8.1a}$$

$$w \longleftarrow w + \gamma h \tag{8.1b}$$

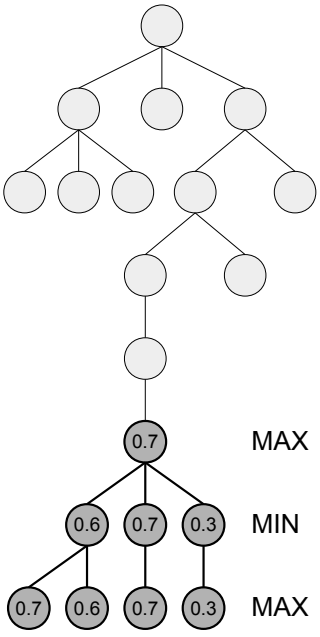
We assume $h \in [0, 1]$. If the evaluation value h is 0.6 and the weight γ is 100, for example, 60 wins and 100 visits are added to the w and v counters of the node at hand. This is equivalent to 60 virtual wins and $100 - 60 = 40$ virtual losses. Since heuristic



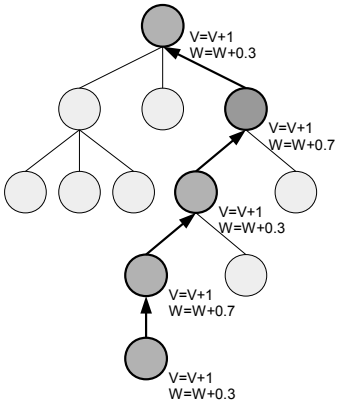
(a) The selection and expansion phases.



(b) $m = 1$ move is played by the rollout policy.



(c) The resulting position is evaluated with a $d = 2$ minimax search. The heuristic evaluation is 0.7 for the maximizing player.



(d) This value is backpropagated as rollout return. Each traversed node increments its visit count by 1, and its win count by 0.7 or $1 - 0.7 = 0.3$ depending on the player to move.

Figure 8.2: The MCTS-IC-M hybrid. $m = 1$ and $d = 2$.

evaluations are typically more reliable than the MCTS value estimates resulting from only a few samples, this prior helps to guide tree growth into a promising direction. If the node is visited frequently however, the influence of the prior progressively decreases over time, as the virtual rollout returns represent a smaller and smaller percentage of the total rollout returns stored in the node. Thus, MCTS rollouts progressively override the heuristic evaluation. We call this approach *MCTS-IP-E* (MCTS with informed priors using an evaluation function) in this chapter.

We propose to extend MCTS-IP with a depth-d minimax search in order to compute the prior value to be stored. This approach aims at guiding the selection policy through more accurate prior information in the nodes of the MCTS tree. We call this approach *MCTS-IP-M* (MCTS with informed priors using minimax). See Figure 8.3 for an illustration. Just like the other hybrids, we first test MCTS-IP-M with $\alpha\beta$ in unenhanced form (Subsection 8.3.4), and then introduce move ordering and k-best pruning (Subsection 8.4.4).

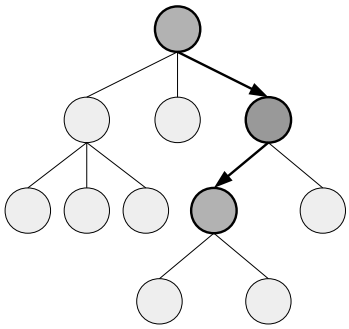
8.3 Experimental Results with Unenhanced $\alpha\beta$

We tested the MCTS-minimax hybrids with heuristic evaluation functions in three different domains: *Othello*, *Catch the Lion*, and *6×6 Breakthrough*. In all experimental conditions, we compared the hybrids as well as their counterparts using heuristics without minimax against regular MCTS-Solver as the baseline. Rollouts were uniformly random unless specified otherwise. Optimal MCTS parameters such as the exploration factor C were determined once for MCTS-Solver in each game and then kept constant for both MCTS-Solver and the MCTS-minimax hybrids during testing. C was 0.7 in Othello and Catch the Lion, and 0.8 in Breakthrough. Draws, which are possible in Othello, were counted as half a win for both players. We used minimax with $\alpha\beta$ pruning, but no other search enhancements. Computation time was 1 second per move.

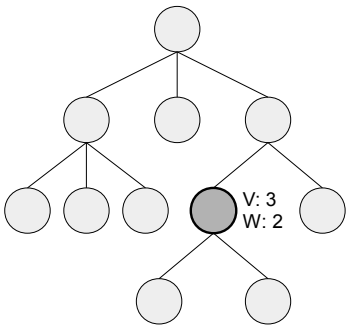
This section is organized as follows. Subsection 8.3.1 outlines the evaluation functions used for each domain. Next, 8.3.2 to 8.3.4 present experimental results for MCTS-IR, MCTS-IC, and MCTS-IP in all games. In 8.3.5, the relative strength of the best-performing hybrids is confirmed by testing them against each other instead of the baseline. In parallel to 7.4.7, Subsection 8.3.6 compares the performance of the hybrids across domains. Finally, combinations of two hybrids are studied in 8.3.7.

8.3.1 Evaluation Functions

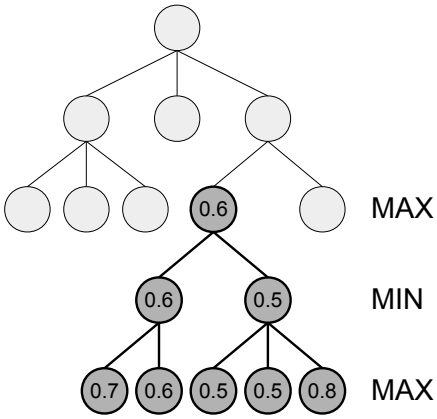
This subsection outlines the heuristic board evaluation functions used for each of the three test domains. The evaluation function from the point of view of the current



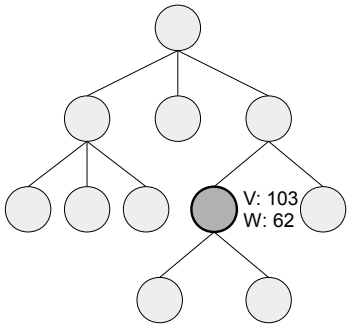
(a) The selection phase.



(b) A tree node with $v = n = 3$ visits is encountered.



(c) This triggers a $d = 2$ minimax search. The heuristic evaluation is $h = 0.6$ for the maximizing player.



(d) This value is stored in the node in the form of $\gamma = 100$ virtual visits and $\gamma * 0.6 = 60$ virtual wins.

Figure 8.3: The MCTS-IP-M hybrid. $n = 3$, $d = 2$, and $\gamma = 100$.

player is always her total score minus her opponent's total score, normalized to $[0, 1]$ as a final step.

Othello

The evaluation function we use for Othello is adapted from the Rolit evaluation function described in Nijssen (2013). It first determines the number of *stable* discs for the player, i.e. discs that cannot change color anymore for the rest of the game. For each stable disc of her color, the player receives 10 points. Afterwards, the number of legal moves for the player is added to the score in order to reward mobility.

Catch the Lion

The evaluation function we use for Catch the Lion represents a weighted material sum for each player, where a Chick counts as 3 points, a Giraffe or Elephant as 5 points, and a Chicken as 6 points, regardless of whether they are on the board or captured by the player.

Breakthrough

The evaluation score we use for 6×6 Breakthrough gives the player 3 points for each piece of her color. Additionally, each piece receives a location value depending on its row on the board. From the player's home row to the opponent's home row, these values are 10, 3, 6, 10, 15, and 21 points, respectively. This evaluation function is a simplified version of the one used in Lorentz and Horey (2014).

8.3.2 Experiments with MCTS-IR

MCTS-IR-E was tested for $\epsilon \in \{0, 0.05, 0.1, 0.2, 0.5\}$. Each parameter setting played 1000 games in each domain against the baseline MCTS-Solver with uniformly random rollouts. Figures 8.4 to 8.6 show the results. The best-performing conditions used $\epsilon = 0.05$ in Othello and Catch the Lion, and $\epsilon = 0$ in Breakthrough. They were each tested in 2000 additional games against the baseline. The results were win rates of 79.9% in Othello, 75.4% in Breakthrough, and 96.8% in Catch the Lion. All of these are significantly stronger than the baseline ($p<0.001$).

MCTS-IR-M was tested for $d \in \{1, \ldots, 4\}$ with the optimal value of ϵ found for each domain in the MCTS-IR-E experiments. Each condition played 1000 games per domain against the baseline player. The results are presented in Figures 8.7 to 8.9. The most promising setting in all domains was $d = 1$. In an additional 2000 games against the baseline per domain, this setting achieved win rates of 73.9% in Othello,

65.7% in Breakthrough, and 96.5% in Catch the Lion. The difference to the baseline is significant in all domains ($p<0.001$).

In each domain, the best settings for MCTS-IR-E and MCTS-IR-M were then tested against each other in 2000 further games. The results for MCTS-IR-M were win rates of 37.1% in Othello, 35.3% in Breakthrough, and 47.9% in Catch the Lion. MCTS-IR-M is weaker than MCTS-IR-E in Othello and Breakthrough ($p<0.001$), while no significant difference could be shown in Catch the Lion. This shows that the incorporation of shallow $\alpha\beta$ searches into rollouts did not improve MCTS-IR in any of the domains at hand. Depth-1 minimax searches in MCTS-IR-M are functionally equivalent to MCTS-IR-E, but have some overhead in our implementation due to the recursive calls to a separate $\alpha\beta$ search algorithm. This results in the inferior performance.

Higher settings of d were not successful because deeper minimax searches in every rollout step require too much computational effort. In an additional set of 1000 games per domain, we compared MCTS-IR-E to MCTS-IR-M at 1000 rollouts per move, ignoring the time overhead of minimax. Here, MCTS-IR-M won 78.6% of games with $d = 2$ in Othello, 63.4% of games with $d = 2$ in Breakthrough, and 89.3% of games with $d = 3$ in Catch the Lion. All of these conditions are significantly stronger than MCTS-IR-E ($p<0.001$). This confirms MCTS-IR-M is suffering from its time overhead.

Interestingly, deeper minimax searches do not always guarantee better performance in MCTS-IR-M, even when ignoring time. While MCTS-IR-M with $d = 1$ won 50.4% (47.3%-53.5%) of 1000 games against MCTS-IR-E in Catch the Lion, $d = 2$ won only 38.0%—both at 1000 rollouts per move. In direct play against each other, MCTS-IR-M with $d = 2$ won 38.8% of 1000 games against MCTS-IR-M with $d = 1$. As standalone players however, a depth-2 minimax beat a depth-1 minimax in 95.8% of 1000 games. Such cases where policies that are stronger as standalone players do not result in stronger play when integrated in MCTS rollouts have been observed before (compare Chapter 7; Gelly and Silver 2007; Silver and Tesauro 2009).

8.3.3 Experiments with MCTS-IC

MCTS-IC-E was tested for $m \in \{0, \ldots, 5\}$. 1000 games were played against the baseline MCTS-Solver per parameter setting in each domain. Figures 8.10 to 8.12 present the results. The most promising condition was $m = 0$ in all three domains. It was tested in 2000 additional games against the baseline. The results were win rates of 61.1% in Othello, 41.9% in Breakthrough, and 98.1% in Catch the Lion. This is significantly stronger than the baseline in Othello and Catch the Lion ($p<0.001$), but weaker in Breakthrough ($p<0.001$). The evaluation function in Breakthrough may not be accurate enough for MCTS to fully rely on it instead of rollouts. Testing higher

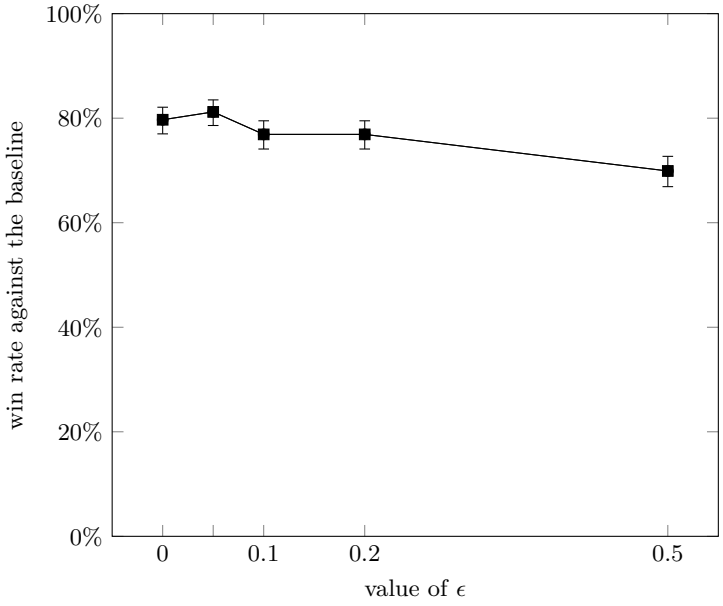


Figure 8.4: Performance of MCTS-IR-E in Othello.

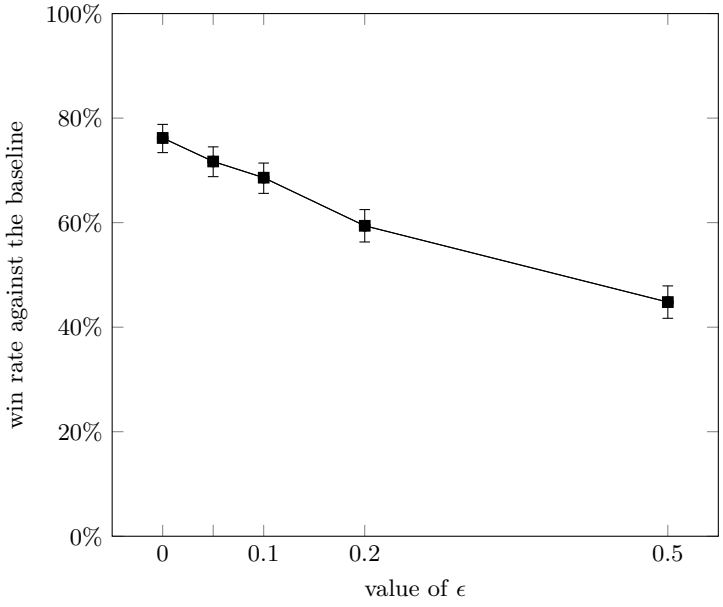


Figure 8.5: Performance of MCTS-IR-E in Breakthrough.

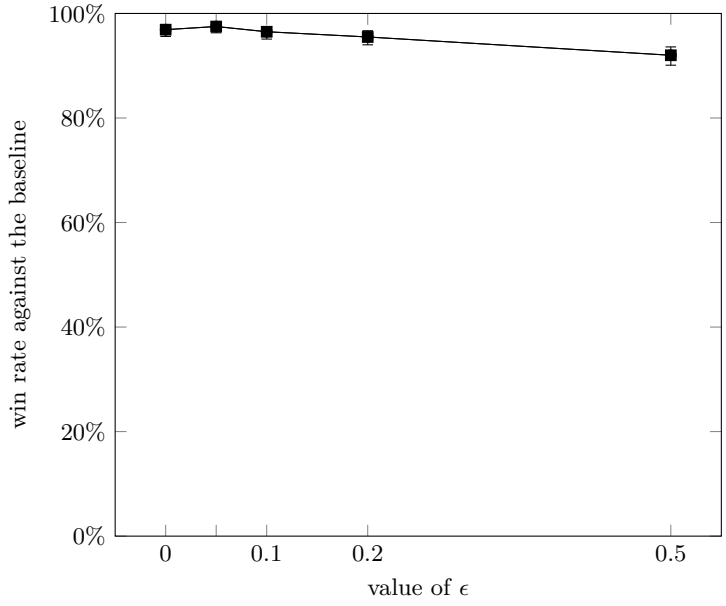


Figure 8.6: Performance of MCTS-IR-E in Catch the Lion.

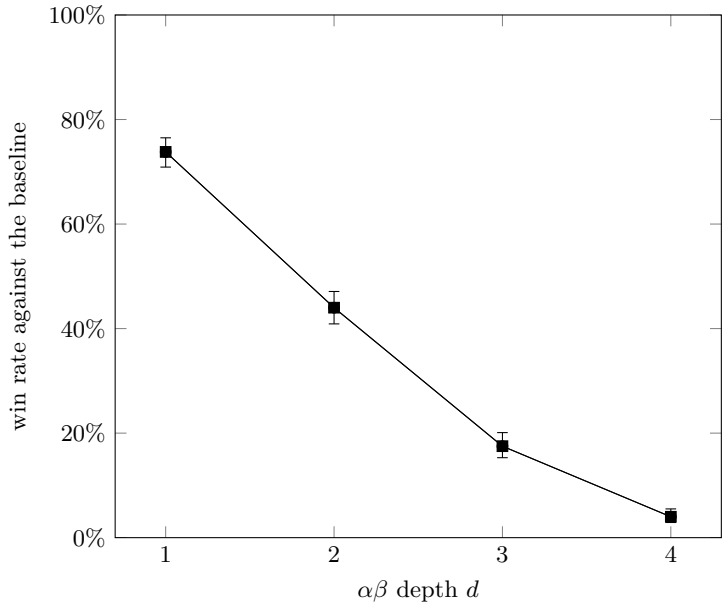


Figure 8.7: Performance of MCTS-IR-M in Othello. For all conditions, $\epsilon = 0.05$.

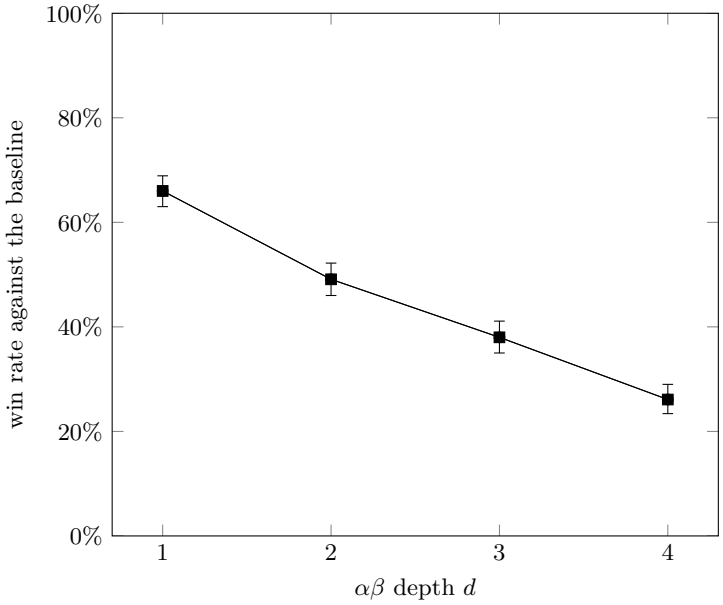


Figure 8.8: Performance of MCTS-IR-M in Breakthrough. For all conditions, $\epsilon = 0$.

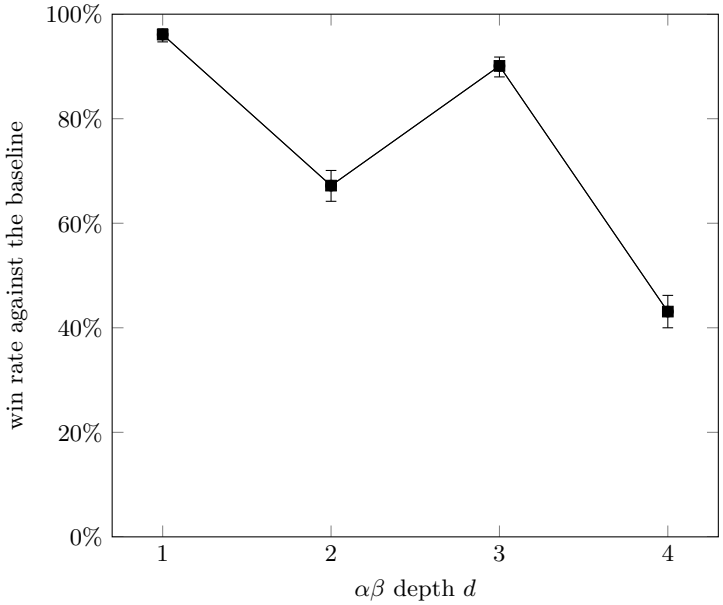


Figure 8.9: Performance of MCTS-IR-M in Catch the Lion. For all conditions, $\epsilon = 0.05$.

values of m showed that as fewer and fewer rollouts are long enough to be cut off, MCTS-IC-E effectively turns into the baseline MCTS-Solver and also shows identical performance. Note that the parameter m can sometimes be sensitive to the opponents it is tuned against. In this subsection, we tuned against regular MCTS-Solver only, and both MCTS-Solver and MCTS-IC used uniformly random rollouts.

MCTS-IC-M was tested for all combinations of $m \in \{0, \ldots, 5\}$ and $d \in \{1, 2, 3\}$, with 1000 games each per domain. The results are shown in Figures 8.13 to 8.15. The best performance was achieved with $m = 0$ and $d = 2$ in Othello, $m = 4$ and $d = 1$ in Breakthrough, and $m = 1$ and $d = 2$ in Catch the Lion. Of an additional 2000 games against the baseline per domain, these settings won 62.4% in Othello, 32.4% in Breakthrough, and 99.6% in Catch the Lion. This is again significantly stronger than the baseline in Othello and Catch the Lion ($p<0.001$), but weaker in Breakthrough ($p<0.001$).

The best settings for MCTS-IC-E and MCTS-IC-M were also tested against each other in 2000 games per domain. Despite MCTS-IC-E and MCTS-IC-M not showing significantly different performance against the regular MCTS-Solver baseline in Othello and Catch the Lion, MCTS-IC-E won 73.1% of these games against MCTS-IC-M in Othello, 58.3% in Breakthrough, and 66.1% in Catch the Lion. All conditions are significantly superior to MCTS-IC-M ($p<0.001$). Thus, the integration of shallow $\alpha\beta$ searches into rollout cutoffs did not improve MCTS-IC in any of the tested domains either.

Just as for MCTS-IR, this is a problem of computational cost for the $\alpha\beta$ searches. We compared MCTS-IC-E with optimal parameter settings to MCTS-IC-M at equal rollouts per move instead of equal time in an additional set of experiments. Here, MCTS-IC-M won 65.7% of games in Othello at 10000 rollouts per move, 69.8% of games in Breakthrough at 6000 rollouts per move, and 86.8% of games in Catch the Lion at 2000 rollouts per move (the rollout numbers were chosen so as to achieve comparable times per move). The parameter settings were $m = 0$ and $d = 1$ in Othello, $m = 0$ and $d = 2$ in Breakthrough, and $m = 0$ and $d = 4$ in Catch the Lion. All conditions here are stronger than MCTS-IC-E ($p<0.001$). This confirms that MCTS-IC-M is weaker than MCTS-IC-E due to its time overhead.

A seemingly paradoxical observation was made with MCTS-IC as well. In Catch the Lion, the values returned by minimax searches are not always more effective for MCTS-IC than the values of simple static heuristics, even when time is ignored. In Catch the Lion for example, MCTS-IC-M with $m = 0$ and $d = 1$ won only 14.8% of 1000 test games against MCTS-IC-E with $m = 0$, at 10000 rollouts per move. With $d = 2$, it won 38.2%. Even with $d = 3$, it won only 35.9% (all at 10000 rollouts per move). Once more these results demonstrate that a stronger policy can lead to a weaker search when embedded in MCTS.

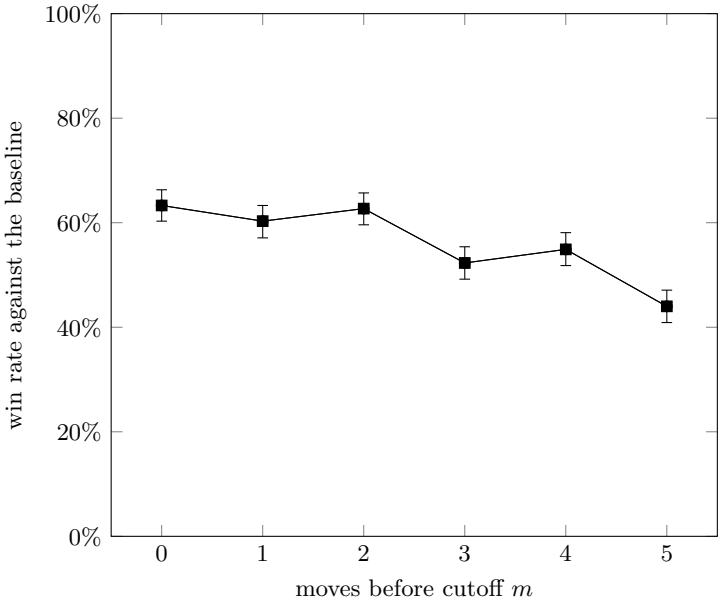


Figure 8.10: Performance of MCTS-IC-E in Othello.

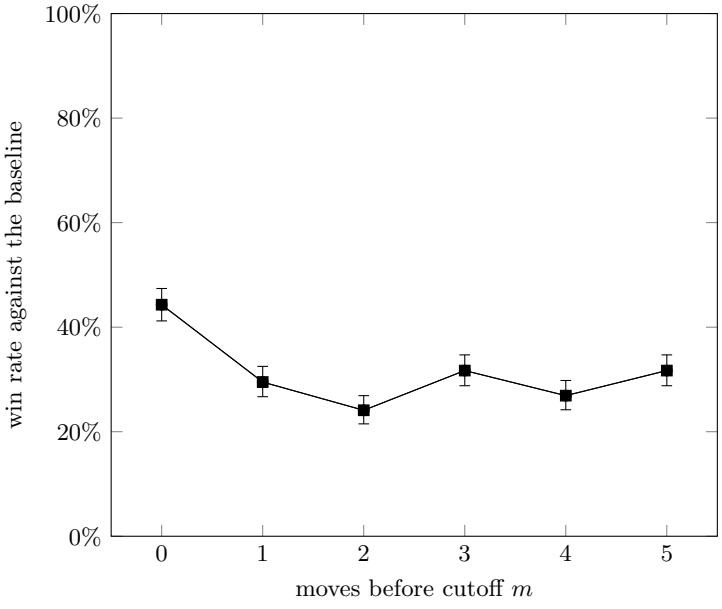


Figure 8.11: Performance of MCTS-IC-E in Breakthrough.

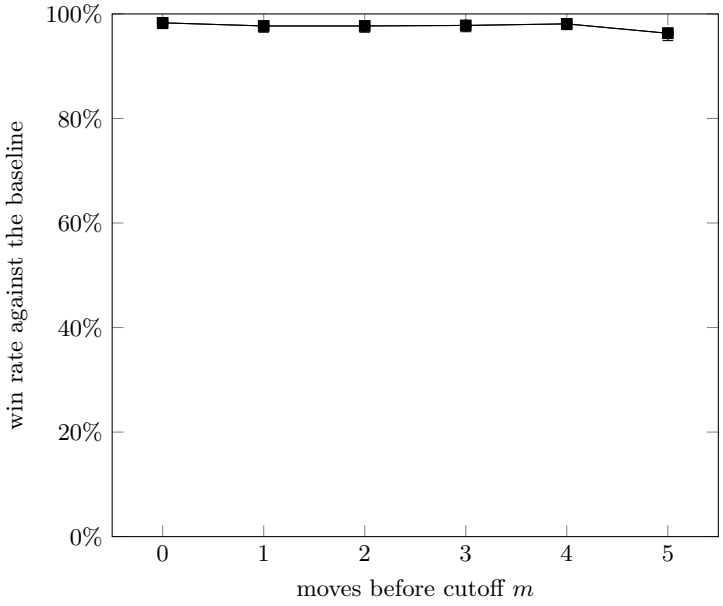


Figure 8.12: Performance of MCTS-IC-E in Catch the Lion.

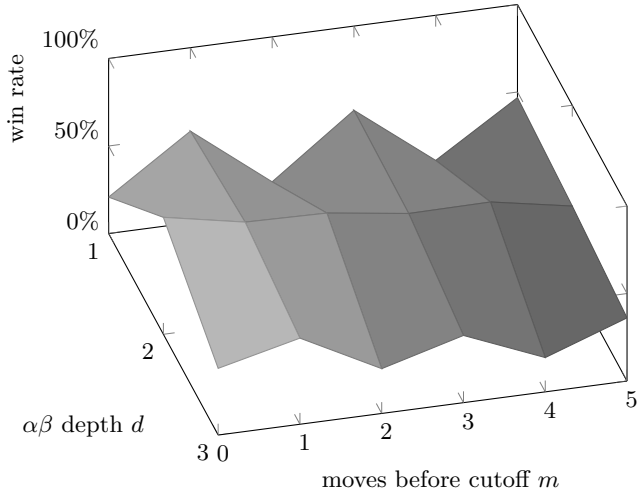


Figure 8.13: Performance of MCTS-IC-M in Othello.

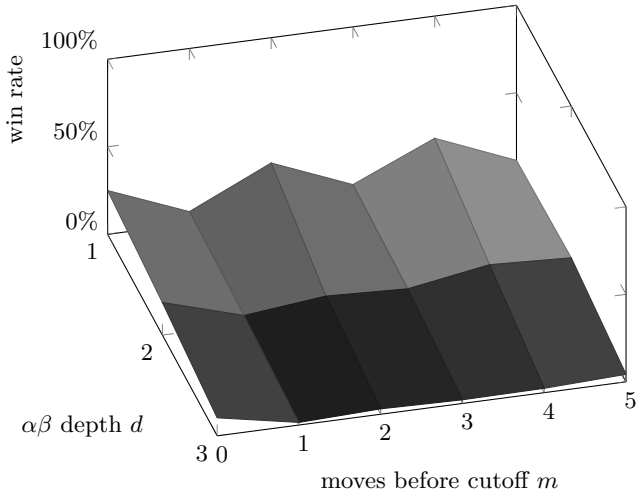


Figure 8.14: Performance of MCTS-IC-M in Breakthrough.

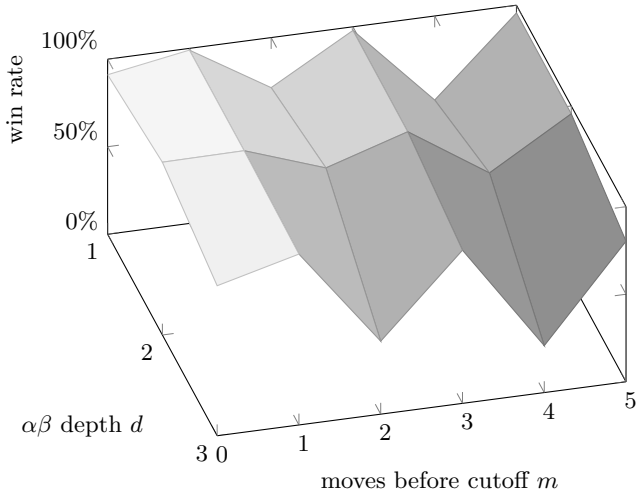


Figure 8.15: Performance of MCTS-IC-M in Catch the Lion.

8.3.4 Experiments with MCTS-IP

MCTS-IP-E was tested for all combinations of $n \in \{0, 1, 2\}$ and $\gamma \in \{50, 100, 250, 500, 1000, 2500, 5000\}$. Each condition played 1000 games per domain against the baseline player. The results are shown in Figures 8.16 to 8.18. The best-performing conditions were $n = 1$ and $\gamma = 1000$ in Othello, $n = 1$ and $\gamma = 2500$ in Breakthrough, and $n = 0$ and $\gamma = 100$ in Catch the Lion. In 2000 additional games against the baseline, these conditions achieved win rates of 56.8% in Othello, 86.6% in Breakthrough, and 71.6% in Catch the Lion (all significantly stronger than the baseline with $p<0.001$).

MCTS-IP-M was tested for all combinations of $n \in \{0, 1, 2, 5, 10, 25\}$, $\gamma \in \{50, 100, 250, 500, 1000, 2500, 5000\}$, and $d \in \{1, \ldots, 5\}$ with 1000 games per condition in each domain. Figures 8.19 to 8.21 present the results, using the optimal setting of d for all domains. The most promising parameter values found in Othello were $n = 2$, $\gamma = 5000$, and $d = 3$. In Breakthrough they were $n = 1$, $\gamma = 1000$, and $d = 1$, and in Catch the Lion they were $n = 1$, $\gamma = 2500$, and $d = 5$. Each of them played 2000 additional games against the baseline, winning 81.7% in Othello, 87.8% in Breakthrough, and 98.0% in Catch the Lion (all significantly stronger than the baseline with $p<0.001$).

The best settings for MCTS-IP-E and MCTS-IP-M subsequently played 2000 games against each other in all domains. MCTS-IP-M won 76.2% of these games in Othello, 97.6% in Catch the Lion, but only 36.4% in Breakthrough (all of the differences are significant with $p<0.001$). We can conclude that using shallow $\alpha\beta$ searches to compute node priors strongly improves MCTS-IP in Othello and Catch the Lion, but not in Breakthrough. This is once more a problem of time overhead due to the larger branching factor of Breakthrough (see Table 6.14). At 1000 rollouts per move, MCTS-IP-M with $n = 1$, $\gamma = 1000$, and $d = 1$ won 91.1% of 1000 games against the best MCTS-IP-E setting in this domain.

An interesting observation is the high weight assigned to the node priors in all domains. It seems that at least for uniformly random rollouts, best performance is achieved when rollout returns never override priors for the vast majority of nodes. They only differentiate between states that look equally promising for the evaluation functions used. The exception is MCTS-IP-E in Catch the Lion, where the static evaluations might be too unreliable to give them large weights due to the tactical nature of the game. Exchanges of pieces can often lead to quick and drastic changes of the evaluation values, or even to the end of the game by capturing a Lion (see 7.4.1 for a comparison of the domains' tacticality in terms of trap density and trap difficulty). The quality of the priors in Catch the Lion improves drastically when minimax searches are introduced, justifying deeper searches ($d = 5$) than in the other tested domains despite the high computational cost. However, MCTS-IC still works better in this

case, possibly because inaccurate evaluation results are only backpropagated once and are not stored to influence the selection policy for a longer time as in MCTS-IP. In Othello, minimax searches in combination with a seemingly less volatile evaluation function lead to MCTS-IP-M being the strongest hybrid tested in this section.

The effect of stronger policies resulting in weaker performance when integrated into MCTS can be found in MCTS-IP just as in MCTS-IR and MCTS-IC. In Breakthrough for example, MCTS-IP-M with $n = 1$, $\gamma = 1000$, and $d = 2$ won only 83.4% of 1000 games against the strongest MCTS-IP-E setting, compared to 91.1% with $n = 1$, $\gamma = 1000$, and $d = 1$—both at 1000 rollouts per move. The difference is significant ($p<0.001$). As standalone players however, depth-2 minimax won 80.2% of 1000 games against depth-1 minimax in our Breakthrough experiments.

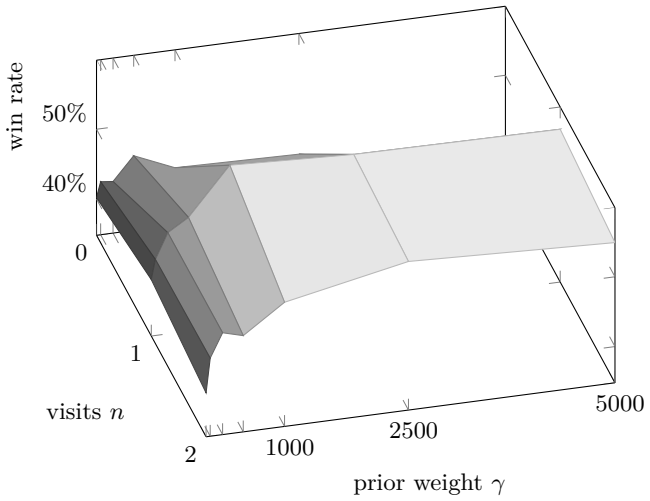


Figure 8.16: Performance of MCTS-IP-E in Othello.

8.3.5 Comparison of Algorithms

Subsections 8.3.2 to 8.3.4 showed the performance of MCTS-IR, MCTS-IC and MCTS-IP against the baseline MCTS-Solver player. We also tested the best-performing variants of these algorithms (MCTS-IP-M in Othello, MCTS-IP-E in Breakthrough, and MCTS-IC-E in Catch the Lion) against all other tested algorithms. In each condition, 2000 games were played. Figures 8.22 to 8.24 present the results, which confirm that MCTS-IP-M is strongest in Othello, MCTS-IP-E is strongest in Breakthrough, and MCTS-IC-E is strongest in Catch the Lion.

The best individual players tested in Othello and Breakthrough thus make use

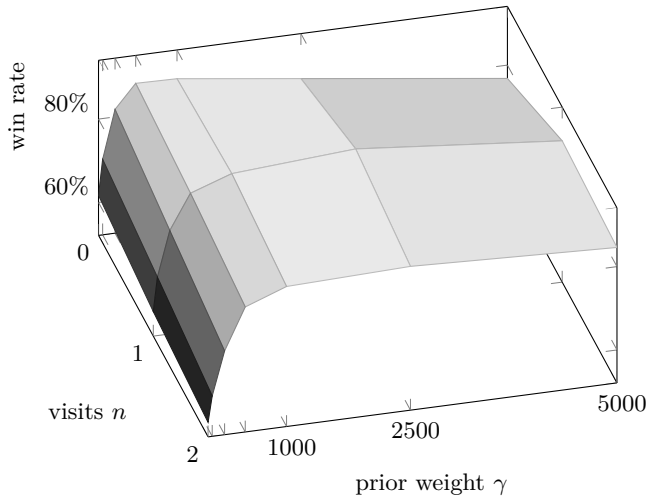


Figure 8.17: Performance of MCTS-IP-E in Breakthrough.

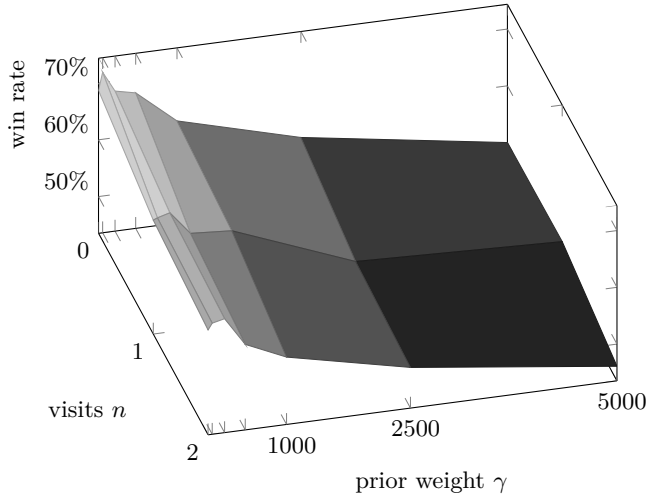


Figure 8.18: Performance of MCTS-IP-E in Catch the Lion.

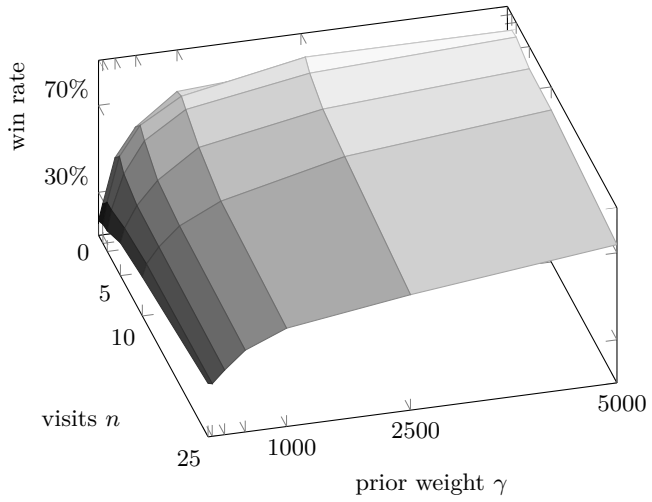


Figure 8.19: Performance of MCTS-IP-M in Othello. For all conditions, $d = 3$.

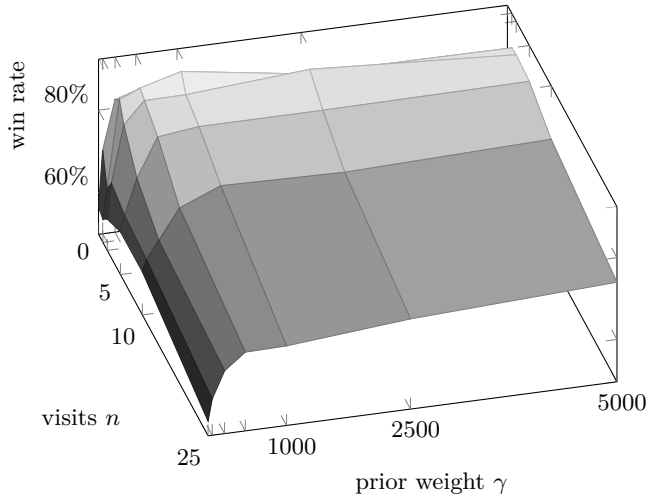


Figure 8.20: Performance of MCTS-IP-M in Breakthrough. For all conditions, $d = 1$.

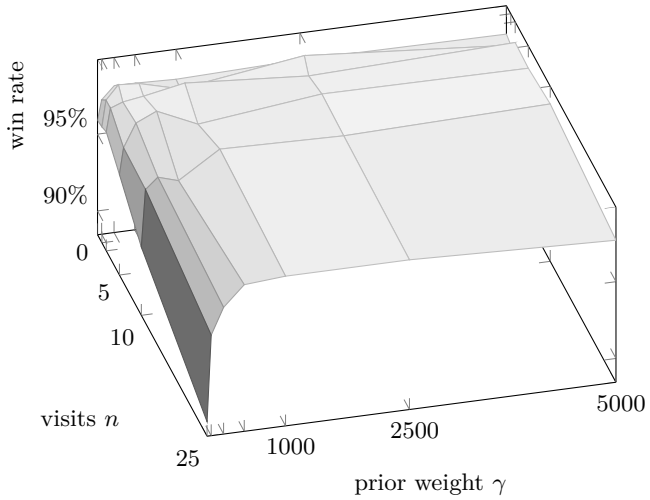


Figure 8.21: Performance of MCTS-IP-M in Catch the Lion. For all conditions, $d = 5$.

of priors in order to combine heuristic information with rollout returns. Because of the different branching factors, computing these priors works best by embedding shallow minimax searches in Othello, and by a simple evaluation function call in Breakthrough. In Catch the Lion, random rollouts may too often return inaccurate results due to the tacticality and possibly also due to the non-converging nature of the domain. Replacing these rollouts with the evaluation function turned out to be the most successful of the individually tested approaches.

8.3.6 Comparison of Domains

In Subsection 7.4.7, the performance of the knowledge-free MCTS-minimax hybrids MCTS-MR, MCTS-MB, and MCTS-MS was related to the tacticality of the test domains. It turned out that integrating minimax into the MCTS framework without using an evaluation function is most effective in the most tactical domains, as defined by their density of shallow traps (Ramanujan et al., 2010a).

In this subsection, we compare the best-performing variants of the MCTS-minimax hybrids MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M in the same manner across domains. Figures 8.25 to 8.27 summarize the relevant results from the previous subsections.

As in the case of knowledge-free hybrids presented in the last chapter, all hybrids are most effective in Catch the Lion. This is expected due to this domain having the highest density and difficulty of shallow traps (compare Figures 7.8 and 7.9).

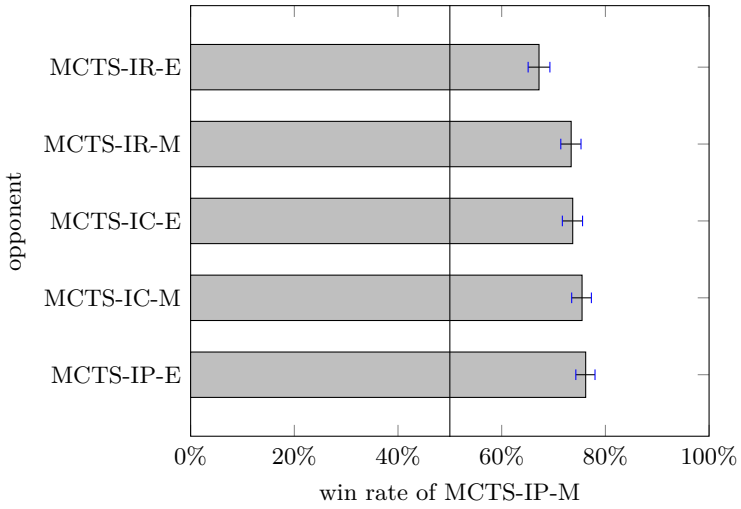


Figure 8.22: Performance of MCTS-IP-M against the other algorithms in Othello.

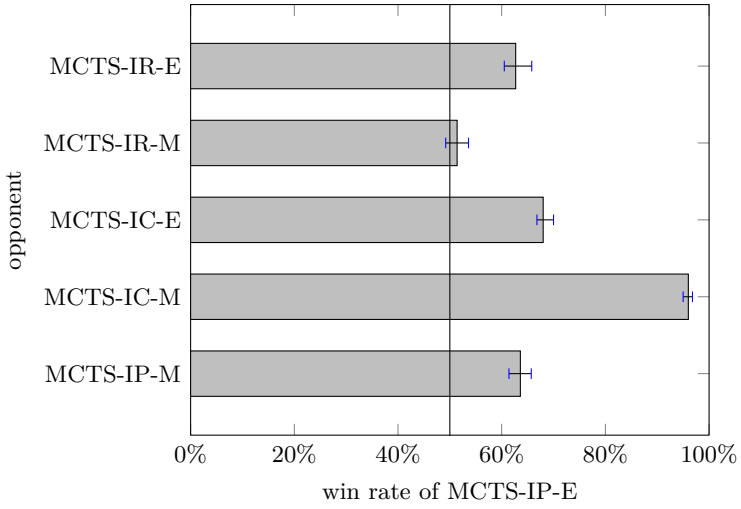


Figure 8.23: Performance of MCTS-IP-E against the other algorithms in Breakthrough.

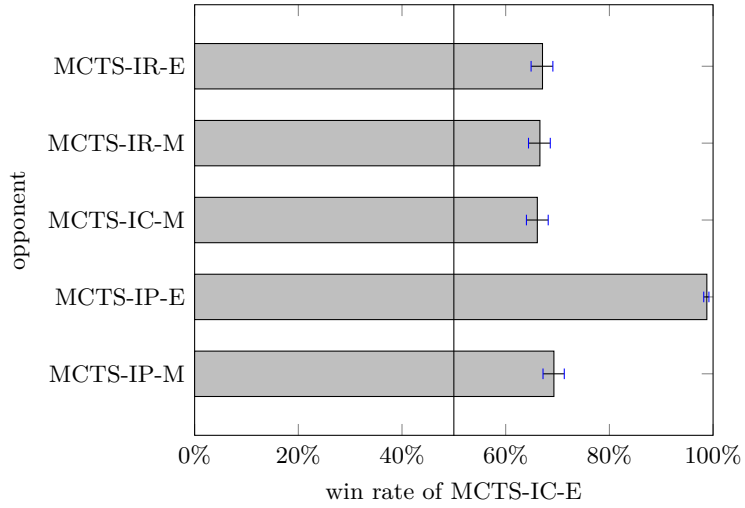


Figure 8.24: Performance of MCTS-IC-E against the other algorithms in Catch the Lion.

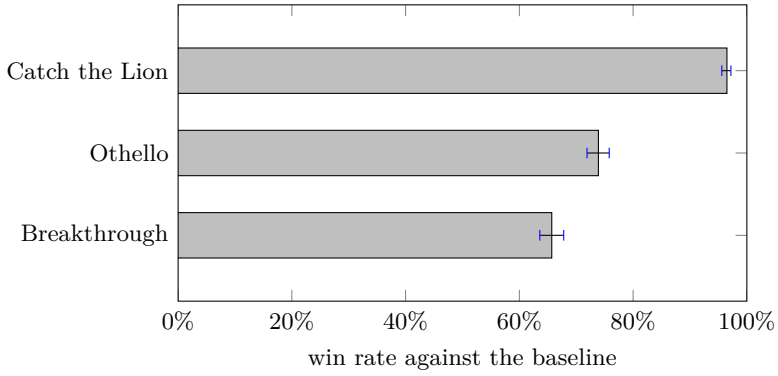


Figure 8.25: Comparison of MCTS-IR-M performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain.

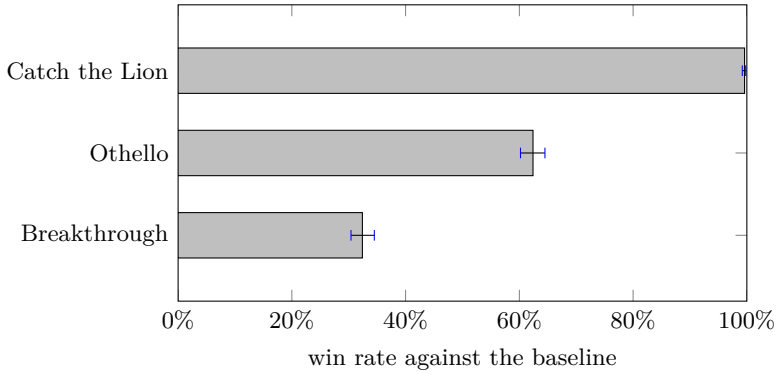


Figure 8.26: Comparison of MCTS-IC-M performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain.

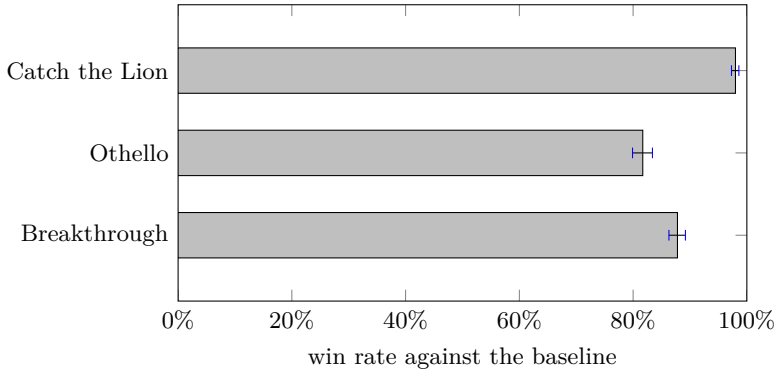


Figure 8.27: Comparison of MCTS-IP-M performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain.

However, MCTS-IR-M and MCTS-IC-M perform worse in Breakthrough than in Othello, although the former has a higher trap density than the latter. As in the case of MCTS-MR in the last chapter, this can probably be explained by the comparatively larger branching factor of 6×6 Breakthrough, which makes embedding minimax more expensive. MCTS-IR-M and MCTS-IC-M typically require more minimax calls than MCTS-IP-M. Further research is required to investigate the influence of the used evaluation function as well. The use of heuristic knowledge could make it possible for MCTS-minimax hybrids to not only detect the type of traps studied in the previous chapter—traps that lead to a lost game—but also *soft traps* that only lead to a disadvantageous position (Ramanujan et al., 2010b).

8.3.7 Combination of Algorithms

Subsections 8.3.2 to 8.3.5 showed the performance of MCTS-IR, MCTS-IC and MCTS-IP in isolation. In order to get an indication whether the different methods of applying heuristic knowledge can successfully be combined, we conducted the following experiments. In Othello, the best-performing algorithm MCTS-IP-M was combined with MCTS-IR-E. In Breakthrough, the best-performing algorithm MCTS-IP-E was combined with MCTS-IR-E. In Catch the Lion, it is not possible to combine the best-performing algorithm MCTS-IC-E with MCTS-IR-E, because with the optimal setting $m = 0$ MCTS-IC-E leaves no rollout moves to be chosen by an informed rollout policy. Therefore, MCTS-IP-M was combined with MCTS-IR-E instead. 2000 games were played in each condition. The results are shown in Figures 8.28 to 8.30. The performance against the MCTS-Solver baseline is given as reference as well.

Applying the same domain knowledge both in the form of node priors and in the form of ϵ-greedy rollouts leads to stronger play in all three domains than using priors alone. In fact, such combinations are the overall strongest players tested in this section even without being systematically optimized. In Othello, the combination MCTS-IP-M-IR-E won 55.2% of 2000 games against the strongest individual algorithm MCTS-IP-M (stronger with p=0.001). In Breakthrough, the combination MCTS-IP-E-IR-E won 53.9% against the best-performing algorithm MCTS-IP-E (stronger with p<0.05). In Catch the Lion, the combination MCTS-IP-M-IR-E with $n = 1$, $w = 2500$, and $d = 4$ won 61.1% of 2000 games against the strongest algorithm MCTS-IC-E (stronger with p<0.001). This means that in Catch the Lion, two algorithm variants that are weaker than MCTS-IC-E can be combined into a hybrid that is stronger.

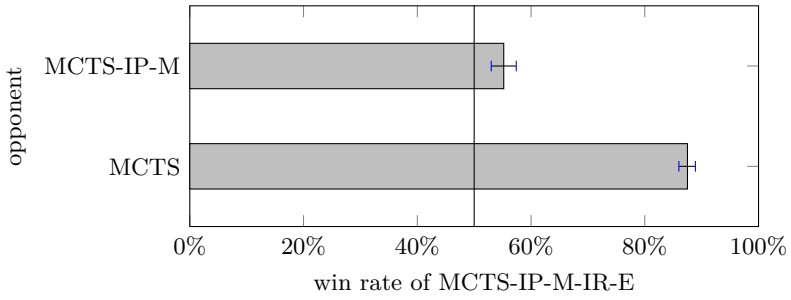


Figure 8.28: Performance of MCTS-IP-M combined with MCTS-IR-E in Othello.

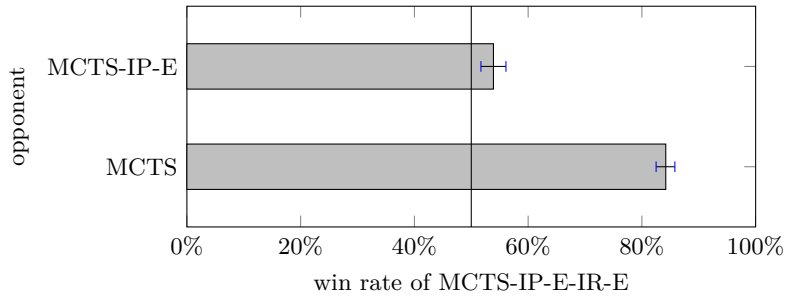


Figure 8.29: Performance of MCTS-IP-E combined with MCTS-IR-E in Breakthrough.

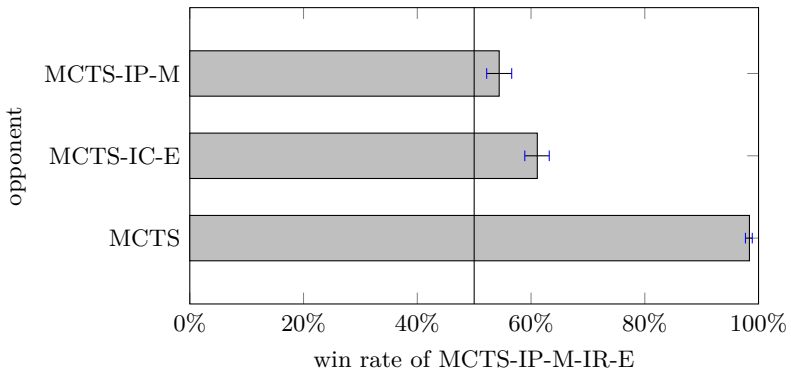


Figure 8.30: Performance of MCTS-IP-M combined with MCTS-IR-E in Catch the Lion.

8.4 Experimental Results with Move Ordering and k-best Pruning

In the previous section, $\alpha\beta$ search was used in its basic, unenhanced form. This was sufficient to improve MCTS-IP in Othello and Catch the Lion, but too computationally expensive for MCTS-IR and MCTS-IC in these two domains, as well as for all hybrids in Breakthrough. The performance difference between the hybrids on the one hand can be explained by the fact that MCTS-IP allows to control the frequency of minimax calls with the parameter n, while MCTS-IC needs to call minimax once in every rollout, and MCTS-IR even for every single rollout move. This makes it easier for MCTS-IP to trade off the computational cost of embedded minimax searches against their advantages over static evaluation function calls. In previous work on a combination of MCTS-IR and MCTS-IC (Winands and Björnsson, 2011), the computational cost seemed to be have less impact due to the longer time settings of up to 30 seconds per move. A reduction of the number of rollouts is less problematic at long time settings because rollouts bring diminishing returns. The performance difference between the domains on the other hand can be explained by the larger branching factor of 6×6 Breakthrough compared to Othello and Catch the Lion, which affects full-width minimax more strongly than for example the sampling-based MCTS. The main problem of MCTS-minimax hybrids seems to be their sensitivity to the branching factor of the domain.

In this section, we therefore conduct further experiments applying limited domain knowledge not only for state evaluation, but also for move ordering. The application of move ordering is known to strongly improve the performance of $\alpha\beta$ through a reduction of the average size of search trees (see Subsection 2.2.4). Additionally, with a good move ordering heuristic one can restrict $\alpha\beta$ to only searching the k moves in each state that seem most promising to the heuristic (*k-best pruning*). The number of promising moves k is subject to empirical optimization. This technique, together with other enhancements such as *killer moves*, has been successfully used for MCTS-IR-M before even in Lines of Action, a domain with an average branching factor twice as high as 6×6 Breakthrough (Winands et al., 2010). Move ordering and k-best pruning could make all MCTS-minimax hybrids viable in domains with much higher branching factors, including the newly proposed MCTS-IC-M and MCTS-IP-M. We call the hybrids with activated move ordering and k-best pruning *enhanced hybrids* or *MCTS-IR-M-k*, *MCTS-IC-M-k*, and *MCTS-IP-M-k*, respectively.

In contrast to the previous section, we only report on the performance of the strongest parameter settings per hybrid and domain in this section. Parameter landscapes are not provided. The reason is that all tuning for this section was conducted with the help of a bandit-based optimizer, distributing test games to

parameter settings via the UCB1-TUNED formula. This method speeds up the tuning process as it does not waste samples on clearly inferior settings, but does not result in comparable win rates for different settings due to the different number of samples for each setting.

Another difference to Section 8.3 is that we observed overfitting in some cases. Especially MCTS-IP-M-k turned out to be sensitive to the opponent it is tuned against—possibly because it has four parameters as opposed to the three of the other hybrids. Instead of tuning against the comparatively weak baseline MCTS-Solver, all hybrids in this section were therefore tuned against a mix of opponents. The mix consists of the best-performing MCTS-IR, MCTS-IC, and MCTS-IP variants (with or without embedded minimax searches) found in Section 8.3 for the domain at hand. The reported win rates are results of testing the tuned hybrid in 2000 additional games against a single opponent. The chosen opponent is the overall best-performing player found in Section 8.3 for the respective domain, namely MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough, each with their best-performing settings.

Results are provided for MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. The techniques MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E are unaffected by the introduction of move ordering and k-best pruning, so their performance remains unchanged from Section 8.3.

This section is organized as follows. Subsection 8.4.1 explains the move ordering functions used for each game and tests their effectiveness. Next, 8.4.2 to 8.4.4 present experimental results for MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k in all domains. Subsection 8.4.5 tests the hybrids against each other instead of the baseline, confirming the relative strength of MCTS-IP-M-k. Similarly to 7.4.7 and 8.3.6, the hybrids are then compared across domains in 8.4.6. Afterwards, the effects of different time settings are studied in 8.4.7 and the effects of different branching factors in 8.4.8. In parallel to 8.3.7, combinations are two hybrids are tested in 8.4.9. Subsection 8.4.10 finally compares the best-performing hybrids to $\alpha\beta$ instead of our MCTS baseline.

8.4.1 Move Ordering Functions

This subsection outlines the heuristic move ordering functions used for each of the three test domains. All move orderings are applied lazily by $\alpha\beta$, meaning that first the highest-ranked move is found and searched, then the second-highest-ranked move is found and searched, etc.—instead of first ranking all available moves and then searching them in that order. When two moves are equally promising to the move ordering function, they are searched in random order.

Othello

The move ordering we use for Othello is adapted from the Rolit move ordering described in Nijssen (2013). Moves are ordered according to their location on the board. The heuristic value of a move on each square of the board is shown in Figure 8.31. Corners, for instance, have the highest heuristic values because discs in the corner are always stable and can provide an anchor for more discs to become stable as well.

5	2	4	3	3	4	2	5
2	1	3	3	3	3	1	2
4	3	4	4	4	4	3	4
3	3	4	4	4	4	3	3
3	3	4	4	4	4	3	3
4	3	4	4	4	4	3	4
2	1	3	3	3	3	1	2
5	2	4	3	3	4	2	5

Figure 8.31: The move ordering for Othello.

Catch the Lion

The move ordering we use for Catch the Lion ranks winning moves first. After winning moves it ranks capturing moves, ordered by the value difference between the capturing and the captured piece. Capturing a more valuable piece with a less valuable piece is preferred (see 8.3.1 for the piece values). This is an idea related to MVV-LVA (*Most Valuable Victim—Least Valuable Aggressor*) capture sorting in computer Chess. After capturing moves, it ranks promotions, and after that all other moves in random order.

Breakthrough

For 6×6 Breakthrough, we consider two different move orderings: a weaker one and a stronger one. This allows us to demonstrate the effect that the quality of the move ordering has on the performance of the MCTS-minimax hybrids. The weaker move ordering ranks moves according to how close they are to the opponent's home row—the closer, the better. The stronger move ordering ranks winning moves first. Second, it ranks saving moves (captures of an opponent piece that is only one move away from winning). Third, it ranks captures, and fourth, all other moves. Within all four groups of moves, moves that are closer to the opponent's home row are preferred as well.

Table 8.1: Effectiveness of the move orderings (m.o.) in Breakthrough, Othello, and Catch the Lion. The table shows the average size of $\alpha\beta$ trees of depth $d \in \{1 \dots 5\}$.

domain	$\alpha\beta$ depth d				
	1	2	3	4	5
Breakthrough					
without m.o.	17.0	90.9	657.4	2847.9	28835.5
weaker m.o.	16.5	60.4	365.2	1025.0	4522.6
stronger m.o.	16.6	46.9	258.6	751.3	3278.6
Othello					
without m.o.	8.9	41.3	180.8	555.6	1605.3
with m.o.	9.1	35.6	147.9	466.1	1139.0
Catch the Lion					
without m.o.	10.9	48.5	279.9	1183.8	5912.9
with m.o.	10.9	30.0	128.8	427.0	1663.6

Effectiveness of the move ordering functions

The following experiment was conducted in order to test whether the proposed move ordering functions are valid, i.e. whether they rank moves more effectively than a random ordering. Effective move orderings on average lead to quicker cutoffs in $\alpha\beta$ and thus to smaller $\alpha\beta$ trees for a given search depth. For each domain, move ordering, and $\alpha\beta$ search depth from 1 to 5, we therefore played 50 fast games (250ms per move) between two MCTS-IC-M players, logging the average sizes of $\alpha\beta$ trees for each player. Since MCTS-IC-M uses $\alpha\beta$ in each rollout, these 50 games provided us with a large number of $\alpha\beta$ searches close to typical game trajectories. Table 8.1 presents the results. All move orderings are successful at reducing the average size of $\alpha\beta$ trees in their respective domains. As expected, the stronger move ordering for Breakthrough results in smaller trees than the weaker move ordering. Depth-1 trees are unaffected because no $\alpha\beta$ cutoffs are possible. The deeper the searches, the greater the potential benefit from move ordering. When comparing the move orderings across domains, it seems that the stronger move ordering in Breakthrough is most effective (88.6% tree size reduction at depth 5), while the Othello move ordering is least effective (29.0% tree size reduction at depth 5). Note that these data do not show whether the gains from move ordering outweigh the overhead, nor whether they allow effective k-best pruning—this is tested in the following subsections. We return to time settings of 1 second per move.

8.4.2 Experiments with MCTS-IR-M-k

The best-performing settings of MCTS-IR-M-k in the tuning experiments were $d = 1$, $\epsilon = 0$, and $k = 2$ in Othello and Catch the Lion, $d = 1$, $\epsilon = 0$, and $k = 20$ in Breakthrough with the weaker move ordering, and $d = 1$, $\epsilon = 0.1$, and $k = 1$ in Breakthrough with the stronger move ordering. In all domains, no search deeper than one ply proved to be worthwhile ($d = 1$). In Breakthrough with the stronger move ordering, the optimal strategy is playing the highest-ordered move in each state without using the evaluation function to differentiate between moves ($k = 1$). Such *move ordering rollouts* (compare Nijssen 2013) are faster than rollouts that rely on the state evaluation function in our Breakthrough implementation. Each of the best-performing settings played 2000 games against the best player with unenhanced $\alpha\beta$ in the respective domain: MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough, using the settings found to be optimal in Section 8.3. Figures 8.32 to 8.35 show the results.

The introduction of move ordering and k-best pruning significantly improved the performance of MCTS-IR-M in all domains ($p<0.0002$) except for Breakthrough with the weaker move ordering. This move ordering turned out not to be effective enough to allow for much pruning, so the problem of the branching factor remained. With the stronger move ordering however, the best-performing setting of MCTS-IR-M-k only considers the highest-ranking move in each rollout step, increasing the win rate against MCTS-IP-E from 49.6% to 77.8%. The stronger move ordering is therefore performing significantly better than the weaker move ordering in MCTS-IR-M-k ($p<0.0002$).

As in Subsection 8.3.2, MCTS-IR-M-k was also tested in 2000 games per domain against the non-hybrid, static evaluation version MCTS-IR-E. In this comparison, the enhancements improved the win rate of MCTS-IR-M from 37.1% to 62.8% in Othello, from 35.3% to 80.3% in Breakthrough (strong ordering), and from 47.9% to 65.3% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in all three domains. However, the best-performing hybrids do not perform deeper search than MCTS-IR-E—their strength comes from move ordering and k-best pruning alone, not from embedded minimax searches. Similar rollout strategies have been successful in Lines of Action before (Winands and Björnsson, 2010).

8.4.3 Experiments with MCTS-IC-M-k

The most promising settings of MCTS-IC-M-k in the tuning experiments were $d = 100$, $m = 1$, and $k = 1$ in Othello, $d = 2$, $m = 1$, and $k = 3$ in Catch the Lion, $d = 20$, $m = 5$, and $k = 1$ in Breakthrough with the weaker move ordering, and $d = 15$, $m = 0$, and $k = 1$ in Breakthrough with the stronger move ordering. Note that the settings in Othello and Breakthrough correspond to playing out a large number of

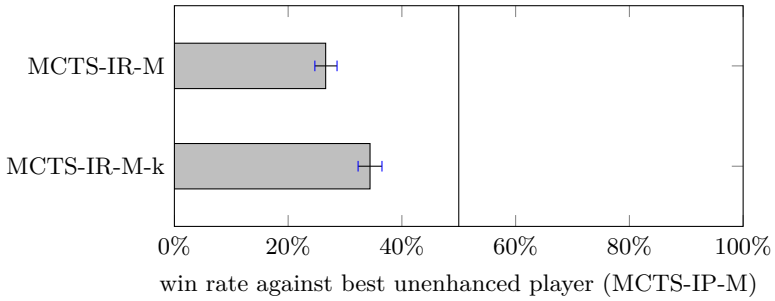


Figure 8.32: Performance of MCTS-IR-M-k in Othello.

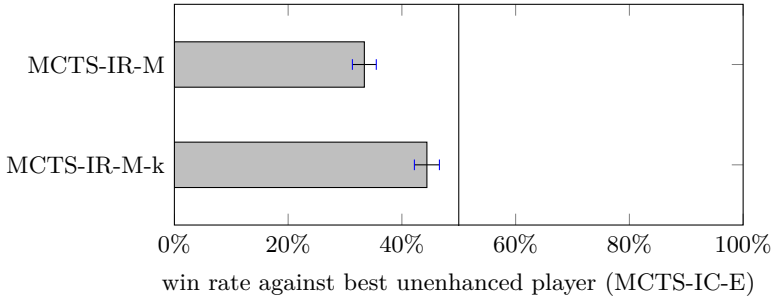


Figure 8.33: Performance of MCTS-IR-M-k in Catch the Lion.

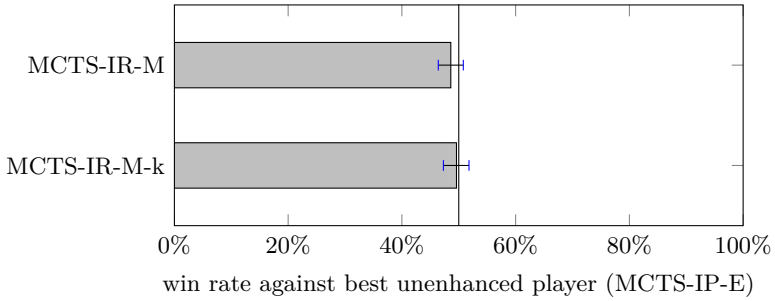


Figure 8.34: Performance of MCTS-IR-M-k with the weaker move ordering in Breakthrough.

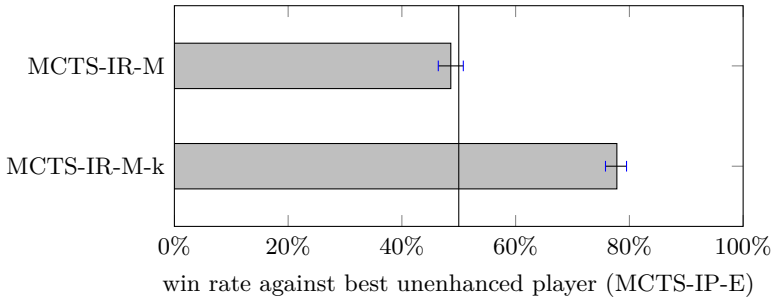


Figure 8.35: Performance of MCTS-IR-M-k with the stronger move ordering in Breakthrough.

moves from the current position, using the best move ordering move in each step, and evaluating the final position. In Othello with $d = 100$, games are played out all the way to terminal positions in this fashion. This makes the best-performing MCTS-IC-M-k players in Othello and Breakthrough similar to the move ordering rollouts discussed in the previous subsection. Only in Catch the Lion, we achieve better play by choosing a small value for d and $k > 1$, leading to an actual embedded minimax search. As speculated in Subsection 8.3.5 for the case of random rollouts, move ordering rollouts might still return too unreliable rollout results in this highly tactical and non-converging domain. Replacing the largest part of rollouts with an evaluation, here through pruned minimax, could be more successful for this reason.

Each of the best-performing settings played 2000 games against MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough. The results are presented by Figures 8.36 to 8.39. Move ordering and k-best pruning significantly improved the performance of MCTS-IC-M in all domains ($p<0.0002$ in Othello and Breakthrough, $p=0.001$ in Catch the Lion). The strength difference between the weaker and the stronger move ordering in Breakthrough is again significant ($p<0.0002$).

For comparison to Subsection 8.3.3, MCTS-IC-M-k played 2000 games in each domain against MCTS-IC-E. The enhancements improved the win rate of MCTS-IC-M from 26.9% to 66.3% in Othello, from 41.7% to 89.8% in Breakthrough (strong ordering), and from 33.9% to 38.9% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in Othello and Breakthrough. However, similar to the case of MCTS-IR-M-k in the previous subsection, the best-performing MCTS-IC-M-k players do not perform true embedded minimax searches with a branching factor of $k > 1$ in these domains. The strength of MCTS-IC-M-k comes from move ordering and k-best pruning alone, not from minimax. Only in Catch the Lion, minimax with $k > 1$ is used—but here the simple evaluation function call of MCTS-IC-E still works better.

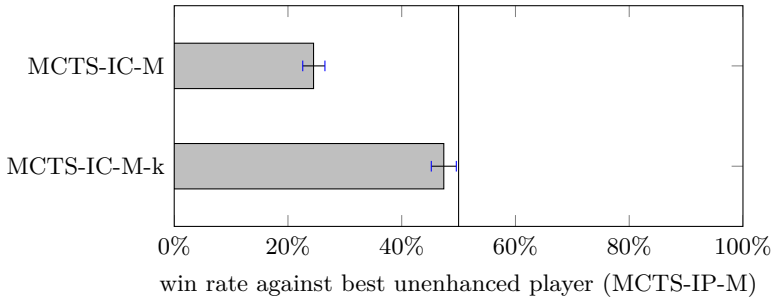


Figure 8.36: Performance of MCTS-IC-M-k in Othello.

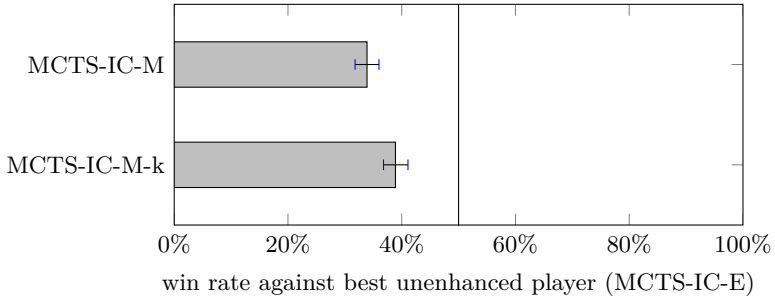


Figure 8.37: Performance of MCTS-IC-M-k in Catch the Lion.

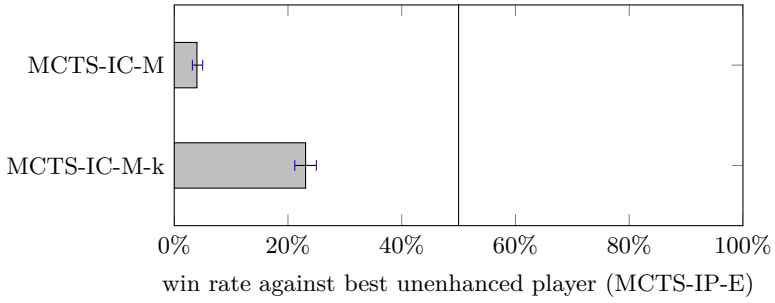


Figure 8.38: Performance of MCTS-IC-M-k with the weaker move ordering in Breakthrough.

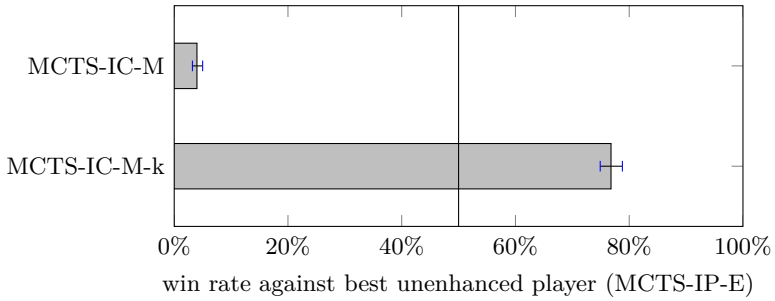


Figure 8.39: Performance of MCTS-IC-M-k with the stronger move ordering in Breakthrough.

8.4.4 Experiments with MCTS-IP-M-k

The tuning results for MCTS-IP-M-k were $n = 6$, $\gamma = 15000$, $d = 5$, and $k = 10$ in Othello, $n = 4$, $\gamma = 1000$, $d = 7$, and $k = 50$ in Catch the Lion, $n = 3$, $\gamma = 30000$, $d = 5$, and $k = 50$ in Breakthrough with the weaker move ordering, and $n = 2$, $\gamma = 20000$, $d = 8$, and $k = 7$ in Breakthrough with the stronger move ordering. We can observe that move ordering and k-best pruning make it possible to search deeper than in plain MCTS-IP-M. The depth parameter d increases from 3 to 5 in Othello, from 5 to 7 in Catch the Lion, and from 1 to 5 or even 8 in Breakthrough, depending on the strength of the move ordering used. The cost of these deeper searches is further reduced by calling them less often: The visits parameter n increases from 2 to 6 in Othello, from 1 to 4 in Catch the Lion, and from 1 to 2 or 3 in Breakthrough. Thanks to their greater depth, these improved $\alpha\beta$ searches get a stronger weight in all domains except for Catch the Lion. The weight parameter γ increases from 5000 to 15000 in Othello, and from 1000 to 20000 or 30000 in Breakthrough. In Catch the Lion, it decreases from 2500 to 1000—a significant strength difference could however not be observed. Each of the settings listed above played 2000 additional games against the best unenhanced player in the respective domain. The results are shown in Figures 8.40 to 8.43. Move ordering and k-best pruning significantly improved the performance of MCTS-IP-M in all domains ($p<0.0002$). Just like with MCTS-IR-M-k and MCTS-IC-M-k, the performance difference between the two Breakthrough move orderings is significant ($p<0.0002$). For comparison to Subsection 8.3.4, MCTS-IP-M-k was tested in 2000 games per domain against MCTS-IP-E. Move ordering and k-best pruning improved the win rate of MCTS-IP-M from 76.2% to 92.4% in Othello, from 36.4% to 80.3% in Breakthrough (strong ordering), and from 97.6% to 98.9% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in all three domains.

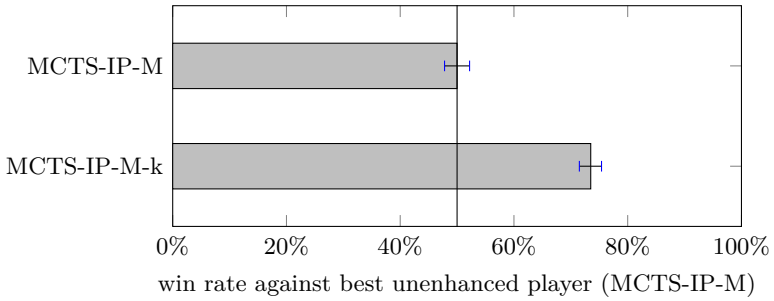


Figure 8.40: Performance of MCTS-IP-M-k in Othello.

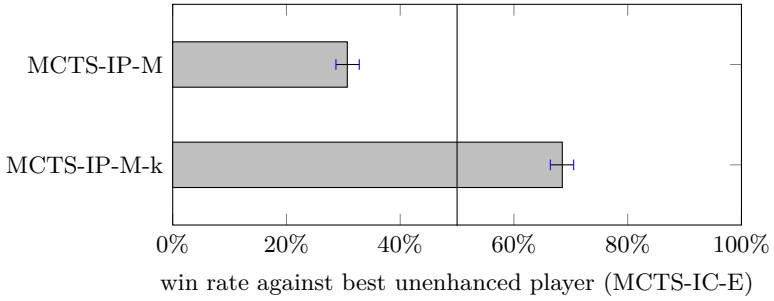


Figure 8.41: Performance of MCTS-IP-M-k in Catch the Lion.

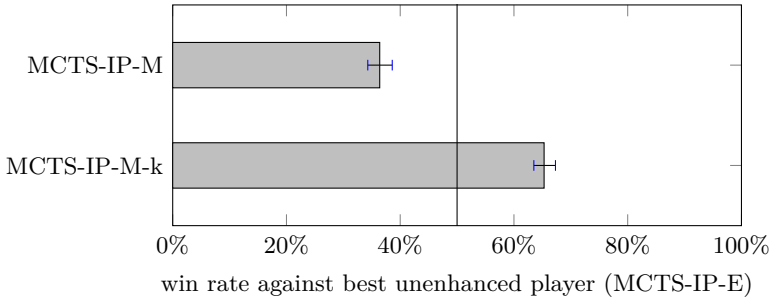


Figure 8.42: Performance of MCTS-IP-M-k with the weaker move ordering in Breakthrough.

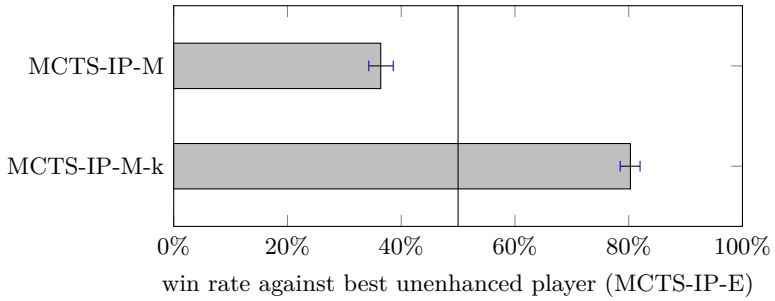


Figure 8.43: Performance of MCTS-IP-M-k with the stronger move ordering in Breakthrough.

In conclusion, we can say that introducing move ordering and k-best pruning has a strong positive effect on all hybrids in all domains. As demonstrated with the example of Breakthrough, the quality of the move ordering used is crucial; but even relatively weak move orderings such as that used in Othello (compare Table 8.1) are quite effective in improving the MCTS-minimax hybrids. In particular, MCTS-IP-M-k is significantly stronger than the best player found with unenhanced $\alpha\beta$ in all domains ($p<0.0002$). MCTS-IP-M-k is also the only hybrid that truly profits from embedded minimax searches in all domains, whereas the best-performing variants of MCTS-IR-M-k do not search deeper than one ply, and the strongest versions of MCTS-IC-M-k do not perform real searches in two out of three domains (they reduce the branching factor to 1 and effectively perform move ordering rollouts instead).

8.4.5 Comparison of Algorithms

The previous subsections show the performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k against a single opponent: the best-performing player without using move ordering and k-best pruning. Analogously to Subsection 8.3.5 for the case of unenhanced $\alpha\beta$, we also tested the best-performing variant of the enhanced hybrids (MCTS-IP-M-k in all domains) against all other discussed algorithms. Each condition consisted of 2000 games. Figures 8.44 to 8.47 show the results.

These results confirm that MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid tested in this chapter. It is significantly stronger than MCTS-IR-E, MCTS-IC-E, MCTS-IP-E, MCTS-IR-M-k, and MCTS-IC-M-k in all domains ($p<0.0002$), with the exception of Breakthrough with the stronger move ordering. In this domain it is stronger than MCTS-IC-M-k only with $p<0.05$, and could not be shown to be significantly different in performance from MCTS-IR-M-k. No tested algorithm played significantly better than MCTS-IP-M-k in any domain.

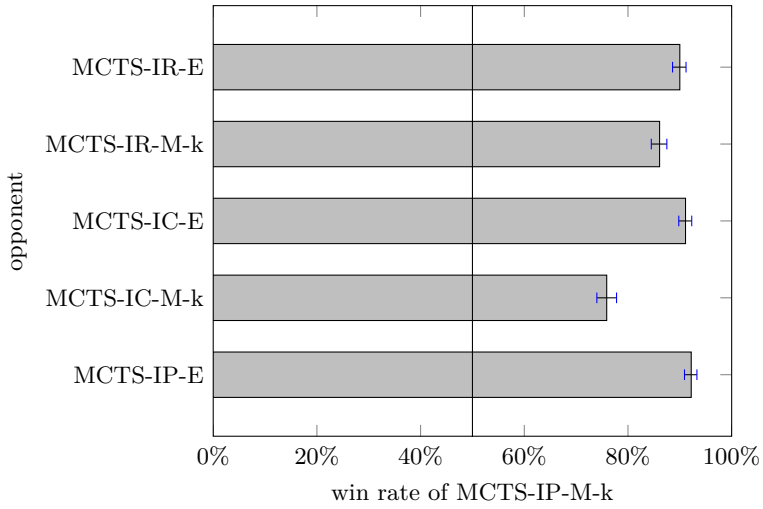


Figure 8.44: Performance of MCTS-IP-M-k against the other algorithms in Othello.

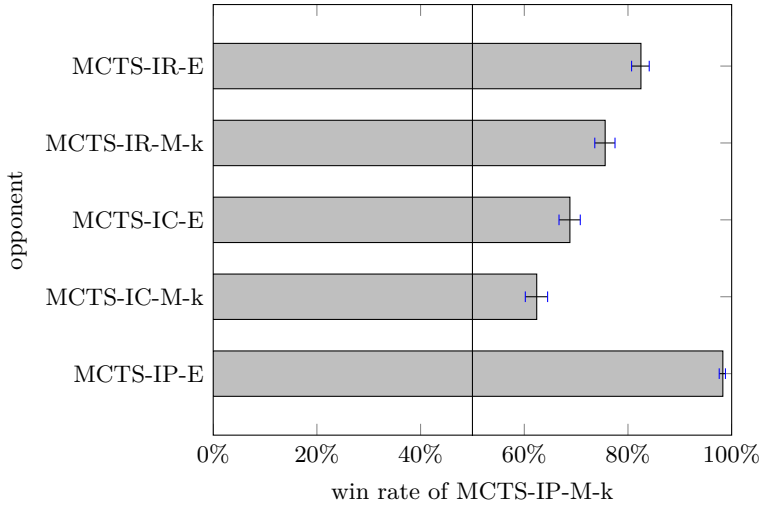


Figure 8.45: Performance of MCTS-IP-M-k against the other algorithms in Catch the Lion.

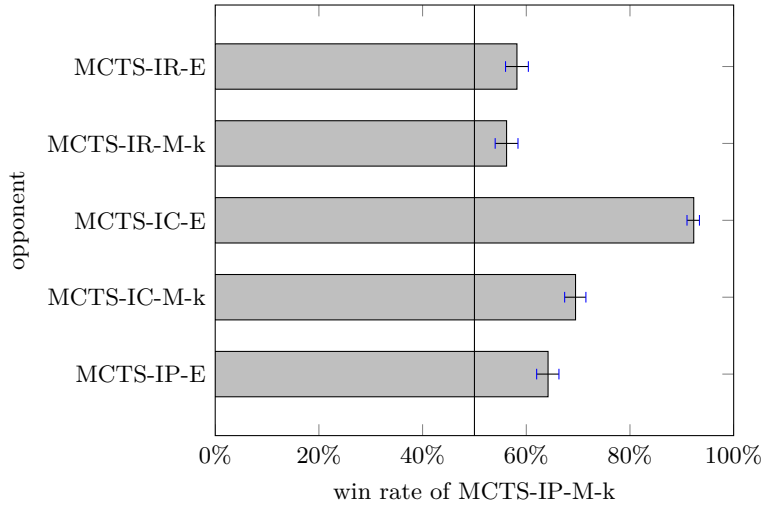


Figure 8.46: Performance of MCTS-IP-M-k against the other algorithms with the weaker move ordering in Breakthrough.

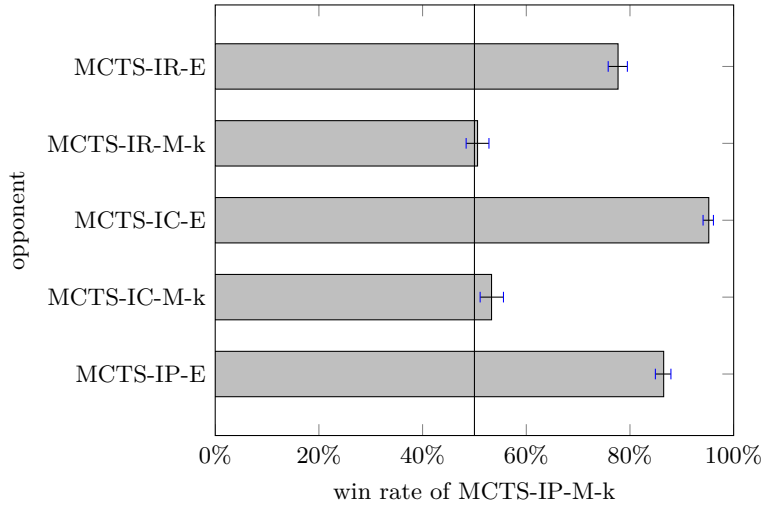


Figure 8.47: Performance of MCTS-IP-M-k against the other algorithms with the stronger move ordering in Breakthrough.

In the rest of this chapter, we are always using Breakthrough with the stronger move ordering.

8.4.6 Comparison of Domains

In analogy to Subsection 8.3.6, we also compare the best-performing variants of the enhanced MCTS-minimax hybrids MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k across domains. This could help us understand in which games each hybrid might be expected to be most successful. Since Subsections 8.4.2 to 8.4.4 have tested the performance of these hybrids against different opponents—the strongest unenhanced players in each domain—we are testing them against the regular MCTS-Solver baseline here to allow for this cross-domain comparison. Because of the large disparity in playing strength, each condition gave 1000 ms per move to the tested hybrid, and 3000 ms per move to MCTS-Solver. The results are presented in Figures 8.48 to 8.50.

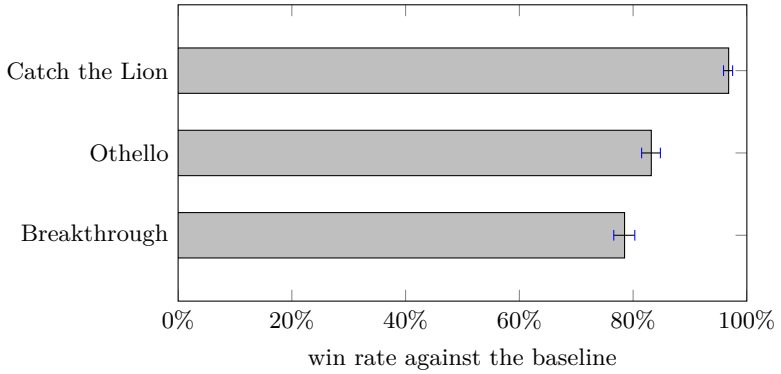


Figure 8.48: Comparison of MCTS-IR-M-k performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain. Breakthrough uses the stronger move ordering. The baseline uses 300% search time.

Just like their counterparts without move ordering and k-best pruning, as well as the knowledge-free hybrids presented in the previous chapter, all hybrids are most successful in Catch the Lion. It seems that the high number of shallow hard traps in this Chess-like domain makes any kind of embedded minimax searches an effective tool for MCTS, with or without evaluation functions. The hybrids are comparatively least successful in Breakthrough. It remains to be determined by future research whether this is caused by the quality of the evaluation function or move ordering used, or whether the higher branching factor of Breakthrough compared to Othello remains an influential factor even when pruning is activated.

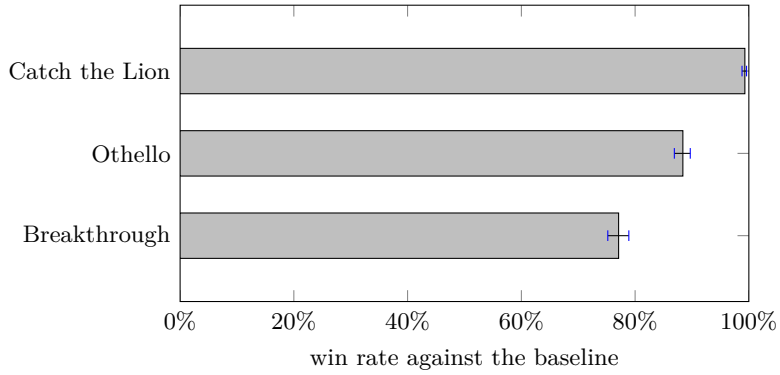


Figure 8.49: Comparison of MCTS-IC-M-k performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain. Breakthrough uses the stronger move ordering. The baseline uses 300% search time.

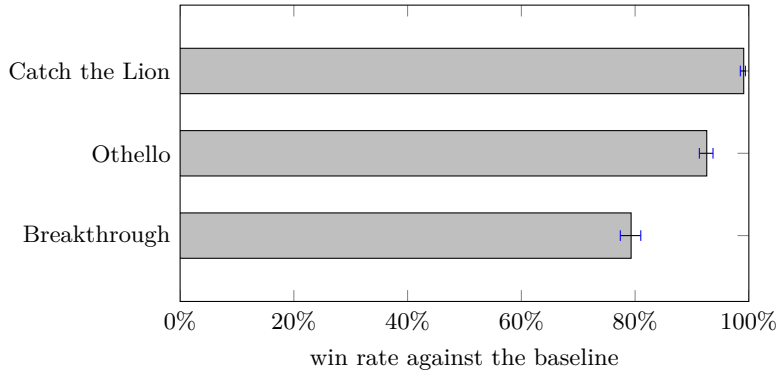


Figure 8.50: Comparison of MCTS-IP-M-k performance in Catch the Lion, Othello, and Breakthrough. The best-performing parameter settings are compared for each domain. Breakthrough uses the stronger move ordering. The baseline uses 300% search time.

8.4.7 Effect of Time Settings

The results presented in the previous subsections of this chapter were all based on a time setting of 1000ms per move. In this set of experiments, the best-performing variants of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k played at different time settings from 250 ms per move to 5000 ms per move. The opponent in all conditions was the regular MCTS-Solver baseline at equal time settings, in order to determine how well the hybrids scale with search time. Each condition consisted of 2000 games. Figure 8.51 shows the results for Breakthrough, Figure 8.52 for Othello, and Figure 8.53 for Catch the Lion.

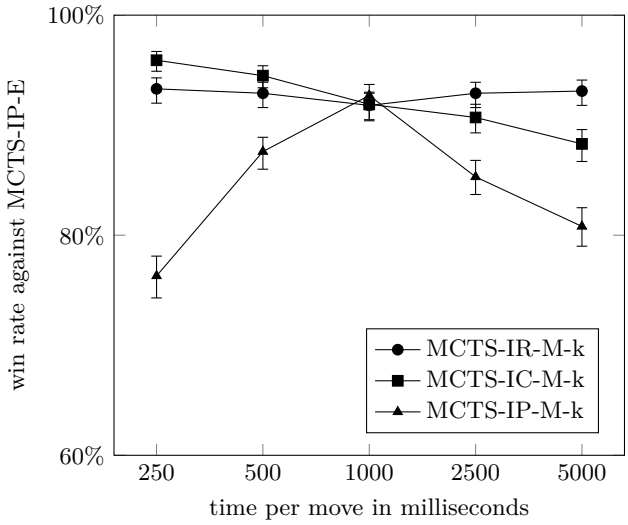


Figure 8.51: Performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k at different time settings in Breakthrough. The stronger move ordering and the best-performing parameter settings at 1000 ms are used.

The results indicate that in the range of time settings investigated, it is easier to transfer parameter settings to longer search times than to shorter search times (see in particular Figure 8.53). MCTS-IP-M-k seems the most likely algorithm to overfit to the specific time setting it was optimized for (see in particular Figure 8.51). As mentioned before, MCTS-IP-M-k is also most likely to overfit to the algorithm it is tuned against—the flexibility of its four parameters might be the reason. In case playing strength at different time settings is of importance, these time settings could be included in the tuning, just as different opponents were included in the tuning for this chapter.

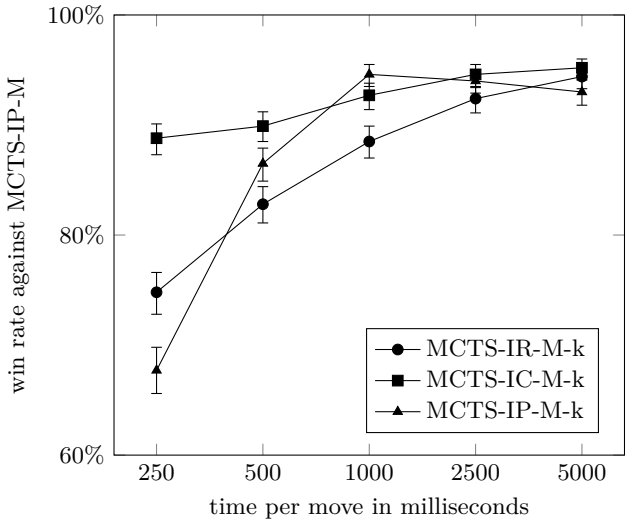


Figure 8.52: Performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k at different time settings in Othello. The best-performing parameter settings at 1000 ms are used.

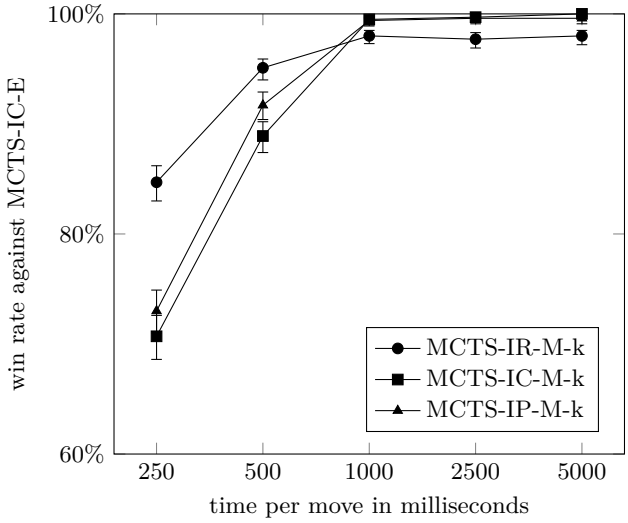


Figure 8.53: Performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k at different time settings in Catch the Lion. The best-performing parameter settings at 1000 ms are used.

Note that these results do not mean the hybrids are ineffective at low time settings. We retuned MCTS-IP-M-k for 250 ms in Breakthrough and Catch the Lion, finding the parameter settings $n = 8$, $\gamma = 30000$, $d = 8$, and $k = 5$ for Breakthrough, and $n = 10$, $\gamma = 10000$, $d = 5$, and $k = 50$ for Catch the Lion. Switching to these variants increased the win rate against the baseline from 76.3% to 96.7% in Breakthrough, and from 73.0% to 90.0% in Catch the Lion. The parameter settings for low time limits are characterized by shortening the embedded minimax searches, either by decreasing k as in Breakthrough or by decreasing d as in Catch the Lion. Increased values of n also mean that the embedded searches are called less often, although from our tuning experience, it seems that the parameter landscape of n and γ is fairly flat compared to that of k and d for MCTS-IP-M-k.

8.4.8 Effect of Branching Factor

In order to give an indication how well move ordering and k-best pruning can help deal with larger branching factors, the enhanced hybrids were also tuned for Breakthrough on an 18×6 board. Increasing the board width from 6 to 18 increases the average branching factor of Breakthrough from 15.5 to 54.2. As mentioned in Subsection 7.4.9, this setup served as an approximation to varying the branching factor while keeping other game properties as equal as possible (without using artificial game trees). The best parameter settings found for 18×6 Breakthrough were $d = 1$, $\epsilon = 0.05$, and $k = 1$ for MCTS-IR-M-k, $d = 4$, $m = 0$, and $k = 25$ for MCTS-IC-M-k, and $n = 2$, $\gamma = 20000$, $d = 3$, and $k = 30$ for MCTS-IP-M-k. Figures 8.54 to 8.56 compare the best-performing settings of the three hybrids on the two board sizes. Each data point represents 2000 games.

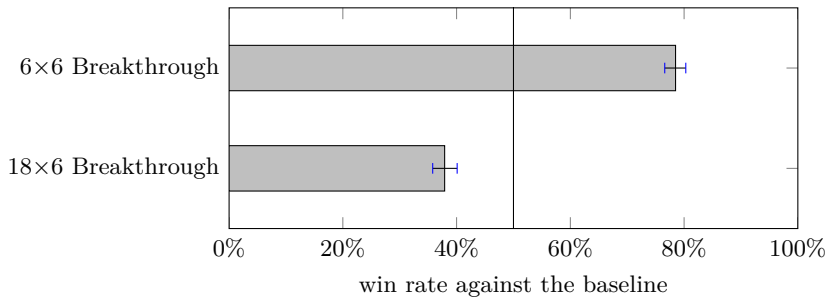


Figure 8.54: Performance of MCTS-IR-M-k in 18×6 Breakthrough. The MCTS-Solver baseline uses 300% search time.

As observed in Subsection 7.4.9, the branching factor has a strong effect on hybrids that use minimax in each rollout step. The win rate of MCTS-IR-M-k is reduced from

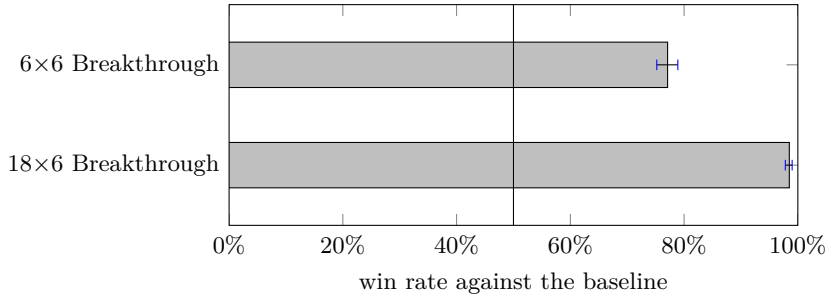


Figure 8.55: Performance of MCTS-IC-M-k in 18×6 Breakthrough. The MCTS-Solver baseline uses 300% search time.

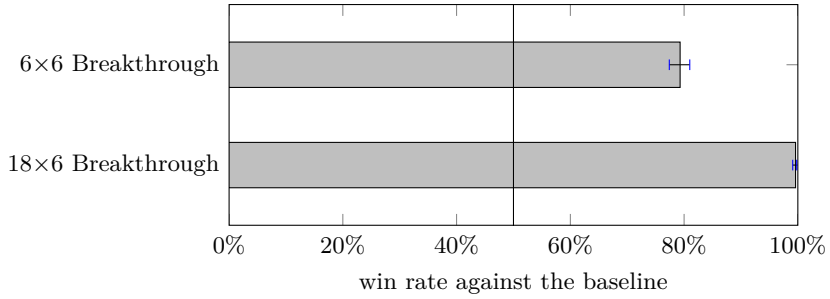


Figure 8.56: Performance of MCTS-IP-M-k in 18×6 Breakthrough. The MCTS-Solver baseline uses 300% search time.

Table 8.2: Best-performing parameter settings for MCTS-IP-M-k-IR-M-k.

Domain	MCTS-IP-M-k parameters				MCTS-IR-M-k parameters		
	n	γ	d	k	d	ϵ	k
Othello	6	5000	5	12	6	0	1
Breakthrough	1	500	4	2	1	0.05	1
Catch the Lion	10	20000	7	100	3	0	8

Table 8.3: Best-performing parameter settings for MCTS-IC-M-k-IR-M-k.

Domain	MCTS-IC-M-k parameters			MCTS-IR-M-k parameters		
	d	m	k	d	ϵ	k
Othello	100	0	1	4	0	1
Breakthrough	20	1	1	3	0	2
Catch the Lion	7	0	100	2	0	2

78.5% (on 6×6) to 37.9% (on 18×6). The MCTS-minimax hybrids newly proposed in this chapter, however, show the opposite effect. Both MCTS-IC-M-k and MCTS-IP-M-k become considerably more effective as we increase the branching factor—probably because more legal moves mean fewer simulations per move, and domain knowledge becomes increasingly more important to obtain useful value estimates.

8.4.9 Combination of Hybrids

Subsections 8.4.2 to 8.4.5 show the performance of MCTS-IR-M-k, MCTS-IC-M-k and MCTS-IP-M-k in isolation. Analogously to Subsection 8.3.7 for the case of unenhanced $\alpha\beta$, this subsection provides some initial results on combinations of different enhanced hybrids. In all three domains, the best-performing hybrid MCTS-IP-M-k as well as MCTS-IC-M-k were combined with MCTS-IR-M-k. The resulting hybrids were named MCTS-IP-M-k-IR-M-k and MCTS-IC-M-k-IR-M-k, respectively. They were tuned against a mix of opponents consisting of the best-performing MCTS-IR, MCTS-IC, and MCTS-IP variants (with or without embedded minimax searches) found in Section 8.3, as well as the best-performing MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k variants found in this section, for the domain at hand.

The parameter values found to work best for MCTS-IP-M-k-IR-M-k are presented in Table 8.2. Tuning of MCTS-IC-M-k-IR-M-k resulted in values of $m = 0$ for Othello and Catch the Lion, and $m = 1$ for Breakthrough—meaning that the informed rollout

policies are used for at most one move per simulation and have little to no effect on MCTS-IC-M-k. MCTS-IC-M-k-IR-M-k does therefore not seem to be a promising combination of hybrids.

The tuned MCTS-IP-M-k-IR-M-k then played an additional 2000 test games against three opponents: the best unenhanced player in the respective domain in order to have results comparable to Subsections 8.4.2 to 8.4.4, and the two hybrids that the combination consists of (MCTS-IR-M-k and MCTS-IP-M-k). The results are shown in Figures 8.57 to 8.59.

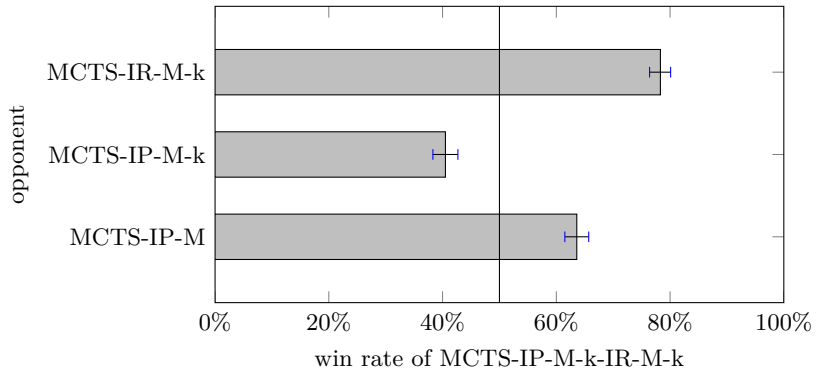


Figure 8.57: Performance of MCTS-IP-M-k combined with MCTS-IR-M-k in Othello.

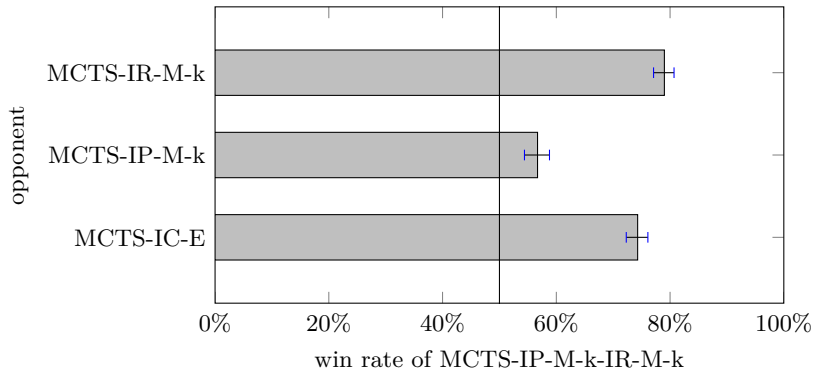


Figure 8.58: Performance of MCTS-IP-M-k combined with MCTS-IR-M-k in Catch the Lion.

MCTS-IP-M-k-IR-M-k leads to stronger play than the individual hybrids in all three domains, with one exception. In Othello, the combination MCTS-IP-M-k-IR-M-k does perform better than MCTS-IR-M-k, but worse than the stronger component

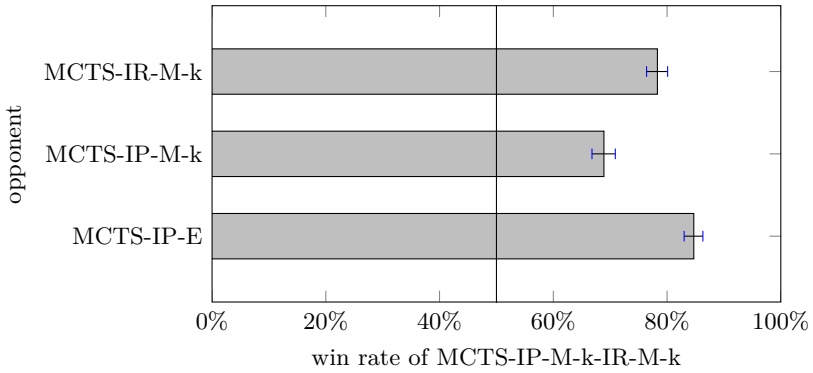


Figure 8.59: Performance of MCTS-IP-M-k combined with MCTS-IR-M-k in Breakthrough with the stronger move ordering.

MCTS-IP-M-k. It seems that the slowness of the minimax rollouts outweighs their added strength in this domain.

We may conclude it is potentially useful to combine different ways of integrating identical domain knowledge into MCTS-minimax hybrids. Other than MCTS-IC-M-k, MCTS-IP-M-k profits from the combination with MCTS-IR-M-k. When testing MCTS-IP-M-k-IR-M-k and MCTS-IC-M-k-IR-M-k in 2000 games against each other, MCTS-IP-M-k-IR-M-k achieves a win rate of 63.6% in Othello, 74.7% in Breakthrough, and 67.0% in Catch the Lion. This means MCTS-IP-M-k-IR-M-k is significantly stronger than MCTS-IC-M-k-IR-M-k in all domains ($p<0.0002$).

8.4.10 Comparison to $\alpha\beta$

Subsections 8.4.2 to 8.4.9 showed the performance of the MCTS-minimax hybrids against various MCTS-based baseline opponents. MCTS-IP-M-k-IR-M-k turned out to be the strongest player tested in Breakthrough and Catch the Lion, and MCTS-IP-M-k with a simple uniformly random rollout policy was the best player tested in Othello. This proves that combining MCTS with minimax can improve on regular MCTS. However, it does not show whether such combinations can also outperform minimax. If the MCTS-minimax hybrids improve on their MCTS component, but not on their minimax component, one could conclude that the test domains are more well-suited to minimax than to MCTS, and that the integration of minimax into MCTS only worked because it resulted in algorithms more similar to minimax. If the MCTS-minimax hybrids can improve on both their MCTS and their minimax component however, they represent a successful combination of the strengths of these two different search methods.

In the last set of experiments in this chapter, we therefore preliminarily examined the performance of the strongest hybrid players against minimax. The strongest hybrid players were MCTS-IP-M-k-IR-M-k in Breakthrough and Catch the Lion, and MCTS-IP-M-k in Othello, with the optimal parameter settings found above. As the minimax baseline we used the same basic $\alpha\beta$ implementation that is also used within the hybrids. It used move ordering and k-best pruning as well, with parameter settings tuned against the hybrids. The optimal k for this standalone $\alpha\beta$ was 10 in Breakthrough, 50 in Catch the Lion, and 14 in Othello. In addition, the standalone $\alpha\beta$ used iterative deepening.

1000 games were played in each domain. The results were win rates for the MCTS-minimax hybrids of 66.9% in Breakthrough, 17.4% in Catch the Lion, and 26.9% in Othello. This means that the MCTS-minimax hybrids presented in this chapter outperform both their MCTS part and their $\alpha\beta$ part in Breakthrough, demonstrating a successful combination of the advantages of the two search approaches. In Catch the Lion and Othello however, regular $\alpha\beta$ is still stronger. This result was expected in the extremely tactical domain of Catch the Lion, where the selectivity of MCTS often leads to missing important moves, and the information returned from rollouts seems unreliable (compare with Subsections 8.3.5 and 8.4.3). The relative strength of $\alpha\beta$ in Othello is more surprising and warrants future investigation. It is possible that the search space of Othello is full of soft traps (Ramanujan et al., 2010b), giving minimax a similar advantage as the hard traps in Catch the Lion.

8.5 Conclusion and Future Research

In this chapter, we continued the research on MCTS-minimax hybrids for the case where domain knowledge in the form of heuristic evaluation functions is available. Three approaches for integrating such knowledge into MCTS were considered. MCTS-IR uses heuristic knowledge to improve the rollout policy. MCTS-IC uses heuristic knowledge to terminate rollouts early. MCTS-IP uses heuristic knowledge as prior for tree nodes. For all three approaches, we compared the computation of state evaluations through simple evaluation function calls (MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E) to the computation of state evaluations through shallow-depth minimax searches using the same heuristic knowledge (MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M). This hybrid MCTS-minimax technique has only been applied to MCTS-IR before in this form.

Experiments with unenhanced $\alpha\beta$ in the domains of Othello, Breakthrough and Catch the Lion showed that MCTS-IP-M is strongest in Othello, MCTS-IP-E is strongest in Breakthrough, and MCTS-IC-E is strongest in Catch the Lion. The embedded minimax searches improve MCTS-IP in Othello and Catch the Lion, but

are too computationally expensive for MCTS-IR and MCTS-IC in these two domains, as well as for all hybrids in Breakthrough. The main problem of MCTS-minimax hybrids with unenhanced $\alpha\beta$ seems to be the sensitivity to the branching factor of the domain at hand.

Further experiments introduced move ordering and k-best pruning to the hybrids in order to cope with this problem, resulting in the enhanced hybrid players called MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. Results showed that with these relatively simple enhancements, MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid investigated in this chapter in all three tested domains. Because MCTS-IP-M-k does not have to call minimax in every rollout or even in every rollout move, it performs better than the other hybrids at low time settings when performance is most sensitive to a reduction in rollouts. MCTS-IP-M-k was also shown to work better for Breakthrough on an enlarged 18×6 board than on the 6×6 board, demonstrating the suitability of the technique for domains with higher branching factors. Additionally, it was shown that the combination of MCTS-IP-M-k with minimax rollouts can lead to further improvements in Breakthrough and Catch the Lion. Moreover, the best-performing hybrid outperformed a simple $\alpha\beta$ implementation in Breakthrough, demonstrating that at least in this domain, MCTS and minimax can be combined to an algorithm stronger than its parts. MCTS-IP-M-k, the use of enhanced minimax for computing node priors, is therefore a promising new technique for integrating domain knowledge into an MCTS framework.

A first direction for future research is the application of additional $\alpha\beta$ enhancements. As a simple static move ordering has proven quite effective in all domains, one could for example experiment with dynamic move ordering techniques such as killer moves or the history heuristic.

Second, some combinations of the hybrids play at a higher level than the hybrids in isolation, despite using the same heuristic knowledge. This may mean we have not yet found a way to fully and optimally exploit this knowledge, which should be investigated further. Combinations of MCTS-IR, MCTS-IC and MCTS-IP could be examined in more detail, as well as new ways of integrating heuristics into MCTS.

Third, all three approaches for using heuristic knowledge have shown cases where embedded minimax searches did not lead to stronger MCTS play than shallower minimax searches or even simple evaluation function calls at equal numbers of rollouts (compare Subsections 8.3.2 to 8.3.4). This phenomenon has only been observed in MCTS-IR before and deserves further study.

Fourth, differences between test domains such as their density of terminal states, their density of hard and soft traps, or their progression property (Finnsson and Björnsson, 2011) could be studied in order to better understand the behavior of MCTS-minimax hybrids with heuristic evaluation functions, and how they compare

to standalone MCTS and minimax. The influence of the quality of the evaluation functions and the move ordering functions themselves could also be investigated in greater depth. Comparing Breakthrough with two different move orderings was only a first step in this direction. As mentioned before, artificial game trees could be a valuable tool to separate the effects of individual domain properties.

9 Conclusions and Future Work

This thesis investigated several approaches to improving Monte-Carlo Tree Search (MCTS) in one-player and two-player domains. The first part on one-player games covered enhancements of the rollout quality and the selectivity of the search, and the second part on two-player games dealt with enhancements of the time management and the tactical strength of MCTS. The research was guided by the problem statement and the research questions formulated in Section 1.3.

This chapter presents the conclusions of this thesis. Section 9.1 answers the four research questions, Section 9.2 provides an answer to the problem statement, and Section 9.3 gives recommendations for future work.

9.1 Answers to the Research Questions

The following subsections address the four research questions posed in Section 1.3 one by one.

9.1.1 Rollouts in One-Player Domains

In MCTS, every state in the search tree is evaluated by the average outcome of Monte-Carlo rollouts from that state. For the consistency of MCTS, i.e. for the convergence to the optimal policy, uniformly random rollouts beyond the tree are sufficient. However, stronger rollout strategies typically greatly speed up convergence. The strength of Monte-Carlo rollouts can be improved for example by hand-coded heuristics, with the help of supervised learning or reinforcement learning conducted offline, or even through online learning while the search is running. But most current techniques show scaling problems in the form of quickly diminishing returns for MCTS as search times get longer. This led to the first research question.

Research Question 1: *How can the rollout quality of MCTS in one-player domains be improved?*

This research question was answered by introducing Nested Monte-Carlo Tree Search (NMCTS), replacing simple rollouts with nested MCTS searches. Instead of improving a given set of rollout policy parameters either offline or online, calls to the rollout policy are replaced with calls to MCTS itself. Independent of the quality of the base-level rollouts, this recursive use of MCTS makes higher-quality rollouts available at higher levels, improving MCTS especially when longer search times are available. NMCTS is a generalization of regular MCTS, which is equivalent to level-1 NMCTS. Additionally, NMCTS can be seen as a generalization of Nested Monte-Carlo Search (NMCS) (Cazenave, 2009), allowing for an exploration-exploitation tradeoff by nesting MCTS instead of naive Monte-Carlo search. The approach was tested in the puzzles SameGame (with both random and informed base-level rollouts), Clickomania, and Bubble Breaker. The time settings were relatively long—either ~21 minutes or ~2.5 hours per position.

As it is known for SameGame that restarting several short MCTS runs on the same problem can lead to better performance than a single, long run (Schadd et al., 2008b), NMCTS was compared to multi-start MCTS. Level-2 NMCTS was found to significantly outperform multi-start MCTS in all test domains. In Clickomania, memory limitations limited the performance of very long searches, which gave an additional advantage to NMCTS.

Furthermore, level-2 NMCTS was experimentally compared to its special case of NMCS. For this comparison, NMCTS (just like NMCS) used the additional technique of move-by-move search, distributing the total search time over several or all moves in the game instead of conducting only one global search from the initial position. NMCTS significantly outperformed both level-2 and level-3 NMCS in all test domains. Since both MCTS and NMCS represent specific parameter settings of NMCTS, correct tuning of NMCTS has to lead to greater or equal success in any domain.

In conclusion, NMCTS is a promising new approach to one-player search, especially for longer time settings.

9.1.2 Selectivity in One-Player Domains

In the most widely used variants of MCTS, employing UCB1 or UCB1-TUNED (Auer et al., 2002) as selection strategies, the selectivity of the search is controlled by a single parameter: the exploration factor. In domains with long solution lengths or when searching with a short time limit however, MCTS might not be able to grow a search tree deep enough even when exploration is completely turned off. Because the search effort can grow exponentially in the tree depth, the search process then spends

too much time on optimizing the first moves of the solution, and not enough time on optimizing the last moves. One option to approach this problem is move-by-move search—however, move-by-move search has to commit to a single move choice at each tree depth d before it starts a new search at $d + 1$. New results from simulations deeper in the tree cannot influence these early move decisions anymore. This led to the second research question.

Research Question 2: *How can the selectivity of MCTS in one-player domains be improved?*

This research question was answered by proposing Beam Monte-Carlo Tree Search (BMCTS), a combination of MCTS with the idea of beam search. BMCTS expands a tree whose size is linear in the search depth, making MCTS more effective especially in domains with long solution lengths or short time limits. Like MCTS, BMCTS builds a search tree using Monte-Carlo simulations as state evaluations. When a predetermined number of simulations has traversed the nodes of a given tree depth, these nodes are sorted by a heuristic value, and only a fixed number of them is selected for further exploration. BMCTS is reduced to a variant of move-by-move MCTS if this number, the beam width, is set to one. However, it generalizes from move-by-move search as it allows to keep any chosen number of alternative moves when moving on to the next tree depth. The test domains for the approach were again SameGame (with both random and informed rollouts), Clickomania, and Bubble Breaker. The time settings were relatively short—they ranged between 0.016 seconds and 4 minutes per position.

First, BMCTS was compared to MCTS using one search run per position. The experimental results show BMCTS to significantly outperform regular MCTS at a wide range of time settings in all tested games. Outside of this domain-dependent range, BMCTS was equally strong as MCTS at the shortest and the longest tested search times—although the lower memory requirements of BMCTS can give it an additional advantage over MCTS at long search times. Depending on the domain and time setting, optimal parameter settings can either result in a move-by-move time management scheme (beam width 1), or in a genuine beam search with a larger beam width than 1. BMCTS with a larger width than 1 was found to significantly outperform move-by-move search at a domain-dependent range of search times. Overall, BMCTS was most successful in Clickomania as this domain seems to profit most from deep searches.

Next, BMCTS was compared to MCTS using the maximum over multiple search runs per position. The experiments demonstrated BMCTS to have a larger advantage over MCTS in this multi-start scenario. This suggests that the performance of BMCTS tends to have a higher variance than the performance of regular MCTS, even in some cases where the two algorithms perform equally well on average. In all test domains,

multi-start BMCTS was better than multi-start MCTS at a wider range of time settings than single-start BMCTS to single-start MCTS.

This observation led to the idea of examining how BMCTS would perform in a nested setting, i.e. replacing MCTS as the basic search algorithm in Nested Monte-Carlo Tree Search. The resulting search algorithm was called Nested Beam Monte-Carlo Tree Search (NBMCTS), and experiments showed NBMCTS to be the overall strongest one-player search technique proposed in this thesis. It performed better than or equal to NMCTS in all domains and at all search times.

In conclusion, BMCTS is a promising new approach to one-player search, especially for shorter time settings. At longer time settings, it combines well with NMCTS as proposed in the previous chapter.

9.1.3 Time Management in Two-Player Domains

In competitive gameplay, time is typically a limited resource. Sudden death, the simplest form of time control, allocates to each player a fixed time budget for the whole game. If a player exceeds this time budget, she loses the game immediately. Since longer thinking times typically result in stronger moves, the player's task is to distribute her time budget wisely among all moves in the game. This is a challenging task both for human and computer players. Previous research on this topic has mainly focused on the framework of $\alpha\beta$ search with iterative deepening. MCTS however allows for much more fine-grained time-management strategies due to its anytime property. This led to the third research question.

Research Question 3: *How can the time management of MCTS in two-player domains be improved?*

This research question was answered by investigating and comparing a variety of time-management strategies for MCTS. The strategies included newly proposed ones as well as strategies described in Huang et al. (2010b) or independently proposed in Baudiš (2011), partly in enhanced form. The strategies can be divided into semi-dynamic strategies that decide about time allocation for each search before it is started, and dynamic strategies that influence the duration of each move search while it is already running. All strategies were tested in the domains of 13×13 and 19×19 Go, and the domain-independent ones in Connect-4, Breakthrough, Othello, and Catch the Lion.

Experimental results showed the proposed strategy called STOP to be most successful. This strategy is based on the idea of estimating the remaining number of moves in the game and using a corresponding fraction of the available search time. However, MCTS is stopped as soon as it appears unlikely that the final move selection

will change by continuing the search. The time saved by these early stops throughout the game is anticipated, and earlier searches in the game are prolonged in order not to have only later searches profit from accumulated time savings. For sudden-death time controls of 30 seconds per game, EXP-STONES with STOP increased Orego's win rate against GNU Go from 25.5% (using a simple baseline) or from 35.3% (using the state-of-the-art ERICA-BASELINE) to 39.1%. In self-play, this strategy won approximately 60% of games against ERICA-BASELINE, both in 13×13 and 19×19 Go under various time controls. Furthermore, comparison across different games showed that the domain-independent strategy STOP is the strongest of all tested time-management strategies. It won 65.0% of self-play games in Connect-4, 58.2% in Breakthrough, 54.8% in Othello, and 60.4% in Catch the Lion. The strategies UNST and CLOSE also proved effective in all domains, with the exception of CLOSE in Othello.

The performance of the time-management strategies was compared across domains, and a shift of available time towards either the opening or the endgame was identified as an effect influencing the performance of many strategies. All time-management strategies that *prolong* search times when certain criteria are met take available time from the later phases and shift it to the earlier phases of the game. All strategies that *shorten* search times based on certain criteria move time from the opening towards the endgame instead. When testing the effect of time-management approaches, it is therefore worth investigating whether these shifts have a positive or negative effect. Consequently, a methodology was developed to isolate the effect of these shifts and judge the effect of a given strategy independently of it. This allows to partly explain the performance of a given strategy. Should the shifting effect be negative, the strategy STOP-B provides an example of how to possibly counteract it by introducing an explicit shift in the opposite direction.

In conclusion, STOP is a promising new sudden-death time management strategy for MCTS.

9.1.4 Tactical Strength in Two-Player Domains

MCTS builds a highly selective search tree, guided by gradually changing rollout statistics. This seems to make it less successful than the traditional approach to adversarial planning, minimax search with $\alpha\beta$ pruning, in domains containing a large number of terminal states and shallow traps (Ramanujan et al., 2010a). In trap situations, precise tactical play is required to avoid immediate loss. The strength of minimax in these situations lies largely in its exhaustive approach, guaranteeing to never miss a consequence of an action that lies within the search horizon, and quickly backing up game-theoretic values from the leaves. MCTS methods based on

sampling however could easily miss a crucial move or underestimate the significance of an encountered terminal state due to averaging value backups. This led to the fourth research question.

Research Question 4: *How can the tactical strength of MCTS in two-player domains be improved?*

This research question was answered by proposing and testing *MCTS-minimax hybrids*, integrating shallow minimax searches into the MCTS framework and thus taking a first step towards combining the strengths of MCTS and minimax. These hybrids can be divided into approaches that require domain knowledge, and approaches that are knowledge-free.

For the knowledge-free case, three different hybrids were studied using minimax in the selection/expansion phase (MCTS-MS), the rollout phase (MCTS-MR), and the backpropagation phase of MCTS (MCTS-MB). Test domains were Connect-4, Breakthrough, Othello, and Catch the Lion. The newly proposed variant MCTS-MS significantly outperformed regular MCTS with the MCTS-Solver extension in Catch the Lion, Breakthrough, and Connect-4. The same holds for the proposed MCTS-MB variant in Catch the Lion and Breakthrough, while the effect in Connect-4 was neither significantly positive nor negative. The only way of integrating minimax search into MCTS known from the literature, MCTS-MR, was quite strong in Catch the Lion and Connect-4 but significantly weaker than the baseline in Breakthrough, suggesting it might be less robust with regard to differences between domains such as the average branching factor. As expected, none of the MCTS-minimax hybrids had a positive effect in Othello due to the low number of terminal states and shallow traps throughout its search space. The density and difficulty of traps predicted the relative performance of MCTS-minimax hybrids across domains well.

Problematic domains for the knowledge-free MCTS-minimax hybrids seemed to feature a low density of traps in the search space, as in Othello, or in the case of MCTS-MR a relatively high branching factor, as in Breakthrough. These problems were consequently addressed with the help of domain knowledge. On the one hand, domain knowledge can be incorporated into the hybrid algorithms in the form of evaluation functions. This can make minimax potentially much more useful in search spaces with few terminal nodes before the latest game phase, such as that of Othello. On the other hand, domain knowledge can be incorporated in the form of a move ordering function. This can be effective in games such as Breakthrough, where traps are relatively frequent, but the branching factor seems to be too high for some hybrids such as MCTS-MR.

For the case where domain knowledge is available, three more algorithms were therefore investigated employing heuristic state evaluations to improve the rollout

policy (MCTS-IR), to terminate rollouts early (MCTS-IC), or to bias the selection of moves in the MCTS tree (MCTS-IP). For all three approaches, the computation of state evaluations through simple evaluation function calls (MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E, where -E stands for "evaluation function") was compared to the computation of state evaluations through shallow-depth minimax searches using the same heuristic knowledge (MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M, where -M stands for "minimax"). MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M are MCTS-minimax hybrids. The integration of minimax has only been applied to MCTS-IR before in this form. Test domains were Breakthrough, Othello, and Catch the Lion.

The hybrids were combined with move ordering and k-best pruning in order to cope with the problem of higher branching factors, resulting in the enhanced hybrid players MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. Results showed that with these two relatively simple $\alpha\beta$ enhancements, MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid investigated in this thesis in all three tested domains. Because MCTS-IP-M-k does not have to call minimax in every rollout or even in every rollout move, it performs better than the other hybrids at low time settings when performance is most sensitive to a reduction in rollouts. MCTS-IP-M-k was also shown to work well on an enlarged 18×6 Breakthrough board, demonstrating the suitability of the technique for domains with higher branching factors. Moreover, the best-performing hybrid outperformed a simple $\alpha\beta$ implementation in Breakthrough, demonstrating that at least in this domain, MCTS and minimax can be combined to an algorithm stronger than its parts.

In conclusion, MCTS-minimax hybrids can improve the performance of MCTS in tactical domains both with and without an evaluation function. MCTS-IP-M-k—using minimax for computing node priors—is a promising new technique for integrating domain knowledge into an MCTS framework.

9.2 Answer to the Problem Statement

After addressing the four research questions, we can now provide an answer to the problem statement.

Problem Statement: *How can the performance of Monte-Carlo Tree Search in a given one- or two-player domain be improved?*

The answer to the problem statement is based on the answers to the research questions above. First, the performance of MCTS in one-player domains can be improved in two ways—by nesting MCTS searches (NMCTS), and by combining MCTS with beam search (BMCTS). The first method is especially interesting for

longer search times, and the second method for shorter search times. A combination of NMCTS and BMCTS can also be effective. Second, the performance of MCTS in two-player domains can be improved in two ways as well—by using the available time for the entire game more intelligently (time management), and by integrating shallow minimax searches into MCTS (MCTS-minimax hybrids). The first approach is relevant to scenarios such as tournaments where the time per game is limited. The second approach improves the performance of MCTS specifically in tactical domains.

9.3 Directions for Future Work

To conclude the last chapter of this thesis, the following paragraphs summarize some promising ideas for future research. They are organized by the chapters from which they result.

Chapter 4: *Nested Monte-Carlo Tree Search.*

Chapter 4 introduced NMCTS and demonstrated its performance in SameGame, Bubble Breaker and Clickomania. The experiments presented so far have only used an exploration factor of 0 for level 2 however. This means that the second level of NMCTS proceeded greedily—it only made use of the selectivity of MCTS, but not of the exploration-exploitation tradeoff. The first step for future research could therefore be the careful tuning of exploration at all search levels. Furthermore, it appears that NMCTS is most effective in domains where multi-start MCTS outperforms a single, long MCTS run (like SameGame and Bubble Breaker), although its lower memory requirements can still represent an advantage in domains where multi-start MCTS is ineffective (like Clickomania). The differences between these classes of tasks remain to be characterized. Finally, NMCTS could be extended to non-deterministic and partially observable domains, for example in the form of a nested version of POMCP (Silver and Veness, 2010).

Chapter 5: *Beam Monte-Carlo Tree Search.*

Chapter 5 presented BMCTS and tested it in SameGame, Bubble Breaker and Clickomania. However, the chosen implementation of BMCTS does not retain the asymptotic properties of MCTS—due to permanent pruning of nodes, optimal behavior in the limit cannot be guaranteed. The addition of e.g. gradually increasing beam widths, similar to progressive widening (Chaslot et al., 2008; Coulom, 2007a) but based on tree levels instead of individual nodes, could restore this important completeness property. Moreover, the basic BMCTS algorithm could be refined in various ways, for instance by using different simulation limits and beam widths for different tree

depths, or by experimenting with different heuristics for selecting the beam nodes. Techniques such as *stratified search* (Lelis et al., 2013) could potentially increase the diversity of nodes in the beam and therefore improve the results. Additionally, it would be interesting to compare the effects of multiple nested searches, and the effects of multiple beam searches on MCTS exploration.

Chapter 6: *Time Management for Monte-Carlo Tree Search.*

Chapter 6 discussed various time management strategies for MCTS in adversarial games with sudden-death time controls. A natural next research step is the combined testing and optimization of these strategies—in order to determine to which degree their positive effects on playing strength are complementary, redundant, or possibly even interfering. The strategy ERICA-BASELINE, consisting of a move-dependent time formula in combination with the UNST and BEHIND heuristics, demonstrates that some combinations can be effective at least in Go. A possible combination of all strategies could take the form of a classifier, trained to decide about continuing or aborting the search in short intervals while using all information relevant to the individual strategies as input. Another worthwhile research direction is the development of improved strategies to measure the complexity and importance of a position and thus to effectively use time where it is most needed.

Chapter 7: *MCTS and Minimax Hybrids.*

Chapter 7 investigated three MCTS-minimax hybrids which are independent of domain knowledge in the form of heuristic evaluation functions. They were tested in Connect-4, Breakthrough, Othello, and Catch the Lion. In all experiments except those concerning MCTS-MR, we used fast, uniformly random rollout policies. On the one hand, the overhead of our techniques would be proportionally lower for any slower, informed rollout policies such as typically used in state-of-the-art programs. On the other hand, improvement on already strong policies might prove to be more difficult. Examining the influence of such MCTS enhancements is a possible direction of future work. Furthermore, while we have focused primarily on the game tree properties of trap density and difficulty as well as the average branching factor in this chapter, the impact of other properties such as the game length or the distribution of terminal values also deserve further study. Eventually, it might be possible to learn from the success of MCTS-minimax hybrids in Catch the Lion, and transfer some ideas to larger games of similar type such as Shogi and Chess.

Chapter 8: *MCTS and Minimax Hybrids with Heuristic Evaluation Functions.*

Chapter 8 studied three MCTS-minimax hybrids for the case where heuristic evaluation functions are available. A first direction for future research is the application

of additional $\alpha\beta$ enhancements. As a simple static move ordering has proven quite effective in all domains, one could for example experiment with dynamic move ordering techniques such as killer moves or the history heuristic. Moreover, some combinations of the hybrids play at a higher level than the hybrids in isolation, despite using the same heuristic knowledge. This may mean we have not yet found a way to fully and optimally exploit this knowledge, which should be investigated further. Combinations of the three hybrids could be examined in more detail, as well as new ways of integrating heuristics into MCTS. In addition, all three approaches for using heuristic knowledge have shown cases where embedded minimax searches did not lead to stronger MCTS play than shallower minimax searches or even simple evaluation function calls at equal numbers of rollouts. This phenomenon has only been observed in MCTS-IR before and deserves further study. Finally, differences between test domains such as their density of terminal states, their density of hard and soft traps, or their progression property (Finnsson and Björnsson, 2011) could be studied in order to better understand the behavior of MCTS-minimax hybrids with heuristic evaluation functions, and how they compare to standalone MCTS and minimax. The influence of the quality of the evaluation functions and the move ordering functions themselves could also be investigated in greater depth.

General remarks. Research in the field of AI in games can be torn between the requirement to improve algorithm *performance* (e.g. in the sense of playing or solving performance) in concrete games—"real" games whose rules are known and whose challenge is intuitively understood—and the aspiration to provide scientific *insight* that generalizes well beyond one or two such games. The approach to this problem chosen in this thesis is the use of a small set of different test domains, as far as possible taken from the class of games known to the literature, and the study of algorithm performance across this set. Unfortunately, different algorithm behavior in different games in such a small set cannot always be explained easily. The large number of differences between any two "real" games potentially confounds many effects, such as for example Breakthrough featuring more traps throughout the search space than Othello, but also having a larger branching factor. As a result, some conclusions regarding the reasons of observed phenomena remain somewhat restricted. In future research on topics similar to those covered in this thesis, different approaches could therefore be considered.

A first option is the use of large sets of test domains, allowing e.g. for the application of regression analysis to quantify the relationships between game properties (independent variables) and algorithm performance (dependent variable). This option however requires considerable programming effort. A possible way to reduce the effort has been found in the area of General Game Playing, where game domains are described

in a common game description language and can thus be exchanged with ease among all GGP researchers and their engines.

A second option is the use of artificial game trees—parameterized artificial domains whose search spaces are constructed with the desired properties, for example while being searched by a game playing algorithm. Artificial game trees can be a valuable tool to separate the effects of domain properties and study them in isolation. In this way, they can help researchers not to be distracted by the potentially idiosyncratic behavior of a number of "real" games chosen for investigation. However, great care must be taken to construct sufficiently realistic game models, in order to be able to generalize findings from artificial back to "real" domains.

References

Abramson, B. (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193. (See pages 5 and 25.)

Akiyama, H., Komiya, K., and Kotani, Y. (2010). Nested Monte-Carlo Search with AMAF Heuristic. In *15th International Conference on Technologies and Applications of Artificial Intelligence, TAAI 2010*, pages 172–176. (See page 59.)

Akl, S. G. and Newborn, M. M. (1977). The Principle Continuation and the Killer Heuristic. In *1977 ACM Annual Conference Proceedings*, pages 466–473. (See page 21.)

Allis, L. V. (1988). A Knowledge-Based Approach of Connect-Four. Master's thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. (See page 49.)

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands. (See pages 43, 48, 49, 52, and 59.)

Allis, L. V., van der Meulen, M., and van den Herik, H. J. (1994). Proof-Number Search. *Artificial Intelligence*, 66(1):91–124. (See pages 59 and 144.)

Althöfer, I., Donninger, C., Lorenz, U., and Rottmann, V. (1994). On Timing, Permanent Brain and Human Intervention. In van den Herik, H. J., Herschberg, I. S., and Uiterwijk, J. W. H. M., editors, *Advances in Computer Chess*, volume 7, pages 285–297. University of Limburg, Maastricht. (See pages 107 and 108.)

Anshelevich, V. V. (2002). A Hierarchical Approach to Computer Hex. *Artificial Intelligence*, 134(1-2):101–120. (See page 2.)

Arneson, B., Hayward, R. B., and Henderson, P. (2010). Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258. (See pages 6 and 143.)

Audouard, P., Chaslot, G.-B., Hoock, J.-B., Perez, J., Rimmel, A., and Teytaud, O. (2009). Grid Coevolution for Adaptive Simulations: Application to the Building of Opening Books in the Game of Go. In Giacobini, M., Brabazon, A., Cagnoni, S., Caro, G. A. D., Ekárt, A., Esparcia-Alcázar, A., Farooq, M., Fink, A., Machado, P.,

McCormack, J., O'Neill, M., Neri, F., Preuss, M., Rothlauf, F., Tarantino, E., and Yang, S., editors, *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science*, pages 323–332. (See page 59.)

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256. (See pages 27, 35, 38, 79, 122, 148, and 234.)

Baier, H. and Drake, P. (2010). The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309. (See pages 33, 41, 57, and 116.)

Baier, H. and Winands, M. H. M. (2012). Time Management for Monte-Carlo Tree Search in Go. In van den Herik, H. J. and Plaat, A., editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 39–51. (See page 117.)

Balla, R.-K. and Fern, A. (2009). UCT for Tactical Assault Planning in Real-Time Strategy Games. In Boutilier, C., editor, *21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 40–45. (See page 6.)

Baudiš, P. (2011). MCTS with Information Sharing. Master's thesis, Charles University, Prague, Czech Republic. (See pages 8, 107, 108, 109, 114, 115, 140, and 236.)

Biedl, T. C., Demaine, E. D., Demaine, M. L., Fleischer, R., Jacobsen, L., and Munro, J. I. (2002). The Complexity of Clickomania. In Nowakowski, R. J., editor, *More Games of No Chance, Proceedings of the MSRI Workshop on Combinatorial Games*, pages 389–404. (See page 44.)

Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). The Challenge of Poker. *Artificial Intelligence*, 134(1-2):201–240. (See page 2.)

Billings, D., Peña, L., Schaeffer, J., and Szafron, D. (1999). Using Probabilistic Knowledge and Simulation to Play Poker. In Hendler, J. and Subramanian, D., editors, *Sixteenth National Conference on Artificial Intelligence, AAAI 99*, pages 697–703. (See page 26.)

Bjarnason, R., Fern, A., and Tadepalli, P. (2009). Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In Gerevini, A., Howe, A. E., Cesta, A., and Refanidis, I., editors, *19th International Conference on Automated Planning and Scheduling, ICAPS 2009*. (See page 6.)

Bjarnason, R., Tadepalli, P., and Fern, A. (2007). Searching Solitaire in Real Time. *ICGA Journal*, 30(3):131–142. (See page 58.)

Björnsson, Y. and Finnsson, H. (2009). CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15. (See page 6.)

Bouzy, B. (2005). Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. *Information Sciences*, 175(4):247–257. (See page 181.)

Bouzy, B. (2006). Associating Shallow and Selective Global Tree Search with Monte Carlo for 9×9 Go. In van den Herik, H. J., Björnsson, Y., and Netanyahu, N. S., editors, *4th International Conference on Computers and Games, CG 2004. Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 67–80. (See page 26.)

Bouzy, B. and Chaslot, G. M. J.-B. (2006). Monte-Carlo Go Reinforcement Learning Experiments. In Kendall, G. and Louis, S., editors, *2006 IEEE Symposium on Computational Intelligence and Games, CIG 2006*, pages 187–194. (See pages 57 and 155.)

Bouzy, B. and Helmstetter, B. (2004). Monte-Carlo Go Developments. In van den Herik, H. J., Iida, H., and Heinz, E. A., editors, *10th International Conference on Advances in Computer Games, ACG 2003*, volume 135 of *IFIP Advances in Information and Communication Technology*, pages 159–174. (See page 26.)

Bowling, M., Burch, N., Johanson, M., and Tammelin, O. (2015). Heads-up Limit Hold'em Poker is Solved. *Science*, 347(6218):145–149. (See page 2.)

Breuker, D. M. (1998). *Memory versus Search in Games*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands. (See page 64.)

Breuker, D. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2001). The PN^2-search Algorithm. In van den Herik, H. J. and Monien, B., editors, *9th Conference on Advances in Computer Games, ACG 2001*, pages 115–132. (See page 59.)

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43. (See pages 6, 7, and 38.)

Brügmann, B. (1993). Monte Carlo Go. Technical report, Max-Planck Institute of Physics, München, Germany. (See pages 26 and 40.)

Buro, M. (2000). Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. In van den Herik, H. J. and Iida, H., editors, *Games in AI Research*, pages 77–96. Universiteit Maastricht. (See page 52.)

Cai, X. and Wunsch, II, D. C. (2007). Computer Go: A Grand Challenge to AI. In Duch, W. and Mandziuk, J., editors, *Challenges for Computational Intelligence*, volume 63 of *Studies in Computational Intelligence*, pages 443–465. (See page 48.)

Cazenave, T. (2007). Reflexive Monte-Carlo Search. In van den Herik, H. J., Uiterwijk, J. W. H. M., Winands, M. H. M., and Schadd, M. P. D., editors, *Proceedings of Computer Games Workshop 2007*, pages 165–173. (See page 58.)

Cazenave, T. (2009). Nested Monte-Carlo Search. In Boutilier, C., editor, *21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 456–461. (See pages 6, 7, 44, 58, 59, 64, 80, 144, and 234.)

Cazenave, T. (2010). Nested Monte-Carlo Expression Discovery. In Coelho, H., Studer, R., and Wooldridge, M., editors, *19th European Conference on Artificial Intelligence, ECAI 2010*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 1057–1058. (See page 59.)

Cazenave, T. (2012). Monte-Carlo Beam Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):68–72. (See page 80.)

Cazenave, T., Balbo, F., and Pinson, S. (2009). Using a Monte-Carlo Approach for Bus Regulation. In *12th International IEEE Conference on Intelligent Transportation Systems, ITSC 2009*, pages 1–6. (See page 58.)

Cazenave, T. and Saffidine, A. (2009). Utilisation de la Recherche Arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, 23(2-3):183–202. In French. (See page 143.)

Cazenave, T. and Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. In van den Herik, H. J., Iida, H., and Plaat, A., editors, *7th International Conference on Computers and Games, CG 2010. Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 93–104. (See pages 49 and 143.)

Chaslot, G. M. J.-B., Hoock, J.-B., Perez, J., Rimmel, A., Teytaud, O., and Winands, M. H. M. (2009). Meta Monte-Carlo Tree Search for Automatic Opening Book Generation. In *IJCAI 2009 Workshop on General Game Playing, GIGA'09*, pages 7–12. (See page 59.)

Chaslot, G. M. J.-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M., and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357. (See pages 26, 29, 80, 81, 103, 181, and 240.)

Childs, B. E., Brodeur, J. H., and Kocsis, L. (2008). Transpositions and Move Groups in Monte Carlo Tree Search. In Hingston, P. and Barone, L., editors, *2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008*, pages 389–395. (See pages 33 and 116.)

Chou, C.-W., Chou, P.-C., Doghmen, H., Lee, C.-S., Su, T.-C., Teytaud, F., Teytaud, O., Wang, H.-M., Wang, M.-H., Wu, L.-W., and Yen, S.-J. (2012). Towards a Solution of 7×7 Go with Meta-MCTS. In van den Herik, H. J. and Plaat, A., editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 84–95. (See page 59.)

Ciancarini, P. and Favini, G. P. (2010). Monte Carlo Tree Search in Kriegspiel. *Artificial Intelligence*, 174(11):670–684. (See page 6.)

Clune, J. E. (2008). *Heuristic Evaluation Functions for General Game Playing*. PhD thesis, University of California, Los Angeles, USA. (See page 144.)

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46. (See page 112.)

Coulom, R. (2007a). Computing Elo Ratings of Move Patterns in the Game of Go. *ICGA Journal*, 30(4):198–208. (See pages 57, 80, 103, and 240.)

Coulom, R. (2007b). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M., editors, *5th International Conference on Computers and Games, CG 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. (See pages 5, 26, and 27.)

Coulom, R. (2009). Criticality: a Monte-Carlo Heuristic for Go Programs. Invited talk, University of Electro-Communications, Tokyo, Japan. (See page 111.)

Cowling, P. I., Ward, E. D., and Powley, E. J. (2012). Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):241–257. (See page 6.)

Den Teuling, N. G. P. and Winands, M. H. M. (2012). Monte-Carlo Tree Search for the Simultaneous Move Game Tron. In *Computer Games Workshop at ECAI 2012*, pages 126–141. (See page 143.)

Domshlak, C. and Feldman, Z. (2013). To UCT, or not to UCT? (Position Paper). In Helmert, M. and Röger, G., editors, *Sixth Annual Symposium on Combinatorial Search, SOCS 2013*, pages 63–70. (See page 7.)

Donkers, H. H. L. M. (2003). *Nosce Hostem: Searching with Opponent Models*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands. (See page 24.)

Donninger, C. (1994). A la Recherche du Temps Perdu: 'That was Easy'. *ICCA Journal*, 17(1):31–35. (See pages 107 and 108.)

Drake, P. et al. (2011). Orego Go Program. Available online: https://sites.google.com/a/lclark.edu/drake/research/orego. (See pages 40, 48, and 116.)

Edelkamp, S. and Gath, M. (2014). Solving Single Vehicle Pickup and Delivery Problems with Time Windows and Capacity Constraints using Nested Monte-Carlo Search. In Duval, B., van den Herik, H. J., Loiseau, S., and Filipe, J., editors, *6th International Conference on Agents and Artificial Intelligence, ICAART 2014*, pages 22–33. (See page 59.)

Edelkamp, S., Gath, M., Cazenave, T., and Teytaud, F. (2013). Algorithm and Knowledge Engineering for the TSPTW Problem. In *2013 IEEE Symposium on Computational Intelligence in Scheduling, CISched 2013*, pages 44–51. (See page 59.)

Edelkamp, S. and Kissmann, P. (2008). Symbolic Classification of General Two-Player Games. In Dengel, A., Berns, K., Breuel, T. M., Bomarius, F., and Roth-Berghofer, T., editors, *31st Annual German Conference on AI, KI 2008*, volume 5243 of *Lecture Notes in Computer Science*, pages 185–192. (See page 49.)

Finnsson, H. (2012). *Simulation-Based General Game Playing*. PhD thesis, Reykjavík University, Reykjavík, Iceland. (See page 143.)

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. In Fox, D. and Gomes, C. P., editors, *23rd AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 259–264. (See pages 49, 50, 52, and 57.)

Finnsson, H. and Björnsson, Y. (2010). Learning Simulation Control in General Game-Playing Agents. In Fox, M. and Poole, D., editors, *24th AAAI Conference on Artificial Intelligence, AAAI 2010*, pages 954–959. (See page 41.)

Finnsson, H. and Björnsson, Y. (2011). Game-Tree Properties and MCTS Performance. In *IJCAI 2011 Workshop on General Intelligence in Game Playing Agents, GIGA'11*, pages 23–30. (See pages 122, 143, 230, and 242.)

Fotland, D. (2004). Building a World-Champion Arimaa Program. In van den Herik, H. J., Björnsson, Y., and Netanyahu, N. S., editors, *4th International Conference on Computers and Games, CG 2004. Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 175–186. (See page 2.)

Free Software Foundation (2009). GNU Go 3.8, 2009. Available online: http://www.gnu.org/software/gnugo/. (See page 116.)

Furcy, D. and Koenig, S. (2005). Limited Discrepancy Beam Search. In Kaelbling, L. P. and Saffiotti, A., editors, *19th International Joint Conference on Artificial Intelligence, IJCAI 2005*, pages 125–131. (See page 80.)

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. In Ghahramani, Z., editor, *24th International Conference on Machine Learning, ICML 2007*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. (See pages 33, 41, 116, 155, 181, 184, and 189.)

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical report, HAL - CCSd - CNRS, France. (See pages 57, 116, 145, and 182.)

Genesereth, M. R., Love, N., and Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2):62–72. (See page 2.)

Ginsberg, M. L. (2001). GIB: Imperfect Information in a Computationally Challenging Game. *Journal of Artificial Intelligence Research (JAIR)*, 14:303–358. (See page 26.)

Greenblatt, R., Eastlake III, D., and Crocker, S. D. (1967). The Greenblatt Chess Program. In *Proceedings of the Fall Joint Computer Conference*, pages 801–810. (See pages 33 and 116.)

Gudmundsson, S. F. and Björnsson, Y. (2013). Sufficiency-Based Selection Strategy for MCTS. In *23rd International Joint Conference on Artificial Intelligence, IJCAI 2013*, pages 559–565. (See pages 50 and 140.)

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107. (See page 5.)

Hennes, D. and Izzo, D. (2015). Interplanetary Trajectory Planning with Monte Carlo Tree Search. In Yang, Q. and Wooldridge, M., editors, *24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 769–775. (See page 6.)

Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press. (See page 2.)

Huang, J., Liu, Z., Benjiea, L., and Feng, X. (2010a). Pruning in UCT Algorithm. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 177–181. (See pages 110 and 115.)

Huang, S.-C., Coulom, R., and Lin, S.-S. (2010b). Time Management for Monte-Carlo Tree Search Applied to the Game of Go. In *International Conference on Technologies and Applications of Artificial Intelligence, TAAI 2010*, pages 462–466. (See pages 8, 107, 108, 109, 111, 113, 116, 117, 119, 140, and 236.)

Hyatt, R. M. (1984). Using Time Wisely. *ICCA Journal*, 7(1):4–9. (See pages 107 and 108.)

Junghanns, A. and Schaeffer, J. (2001). Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge. *Artificial Intelligence*, 129(1-2):219–251. (See page 5.)

Kaneko, T., Kanaka, T., Yamaguchi, K., and Kawai, S. (2005). Df-pn with Fixed-Depth Search at Frontier Nodes. In *10th Game Programming Workshop, GPW 2005*, pages 1–8. In Japanese. (See page 144.)

Kinny, D. (2012). A New Approach to the Snake-In-The-Box Problem. In De Raedt, L., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., and Lucas, P. J. F., editors, *20th European Conference on Artificial Intelligence, ECAI 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 462–467. (See page 59.)

Kirci, M., Schaeffer, J., and Sturtevant, N. (2009). Feature Learning Using State Differences. In *IJCAI 2009 Workshop on General Game Playing, GIGA'09*, pages 35–42. (See pages 49 and 50.)

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., and Matsubara, H. (1997). RoboCup: A Challenge Problem for AI. *AI Magazine*, 18(1):73–85. (See page 3.)

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326. (See pages 4, 17, 19, 21, and 141.)

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *17th European Conference on Machine Learning, ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. (See pages 5, 7, 26, 29, and 38.)

Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved Monte-Carlo Search. Technical report, MTA SZTAKI, Budapest, Hungary - University of Tartu, Tartu, Estonia. (See pages 7 and 29.)

Kocsis, L., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2001). Learning Time Allocation Using Neural Networks. In Marsland, T. A. and Frank, I., editors, *2nd International Conference on Computers and Games, CG 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 170–185. (See page 109.)

Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109. (See page 5.)

Lai, T. L. and Robbins, H. (1985). Asymptotically Efficient Adaptive Allocation Rules. *Advances in Applied Mathematics*, 6(1):4–22. (See page 35.)

Laird, J. E. and van Lent, M. (2001). Human-Level AI's Killer Application: Interactive Computer Games. *AI Magazine*, 22(2):15–25. (See page 3.)

Lanctot, M., Winands, M. H. M., Pepels, T., and Sturtevant, N. R. (2014). Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014*, pages 341–348. (See page 180.)

Lanctot, M., Wittlinger, C., Winands, M. H. M., and Den Teuling, N. G. P. (2013). Monte Carlo Tree Search for Simultaneous Move Games: A Case Study in the Game of Tron. In Hindriks, K., de Weerdt, M., van Riemsdijk, B., and Warnier, M., editors, *25th Benelux Conference on Artificial Intelligence, BNAIC 2013*, pages 104–111. (See page 38.)

Lee, C.-S., Wang, M.-H., Chaslot, G. M. J.-B., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73–89. (See pages 6, 41, and 47.)

Lelis, L. H. S., Zilles, S., and Holte, R. C. (2013). Stratified Tree Search: A Novel Suboptimal Heuristic Search Algorithm. In *12th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013)*, pages 555–562. (See pages 103 and 241.)

Lieberum, J. (2005). An Evaluation Function for the Game of Amazons. *Theoretical Computer Science*, 349(2):230–244. (See page 2.)

Lorentz, R. J. (2008). Amazons Discover Monte-Carlo. In van den Herik, H. J., Xu, X., Ma, Z., and Winands, M. H. M., editors, *6th International Conference on Computers and Games, CG 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 13–24. (See pages 6, 33, 141, 181, 182, and 184.)

Lorentz, R. J. (2011). Experiments with Monte-Carlo Tree Search in the Game of Havannah. *ICGA Journal*, 34(3):140–149. (See pages 143, 144, and 181.)

Lorentz, R. J. (2012). An MCTS Program to Play EinStein Würfelt Nicht! In van den Herik, H. J. and Plaat, A., editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 52–59. (See page 6.)

Lorentz, R. J. and Horey, T. (2014). Programming Breakthrough. In van den Herik, H. J., Iida, H., and Plaat, A., editors, *8th International Conference on Computers and Games, CG 2013*, volume 8427 of *Lecture Notes in Computer Science*, pages 49–59. (See pages 50, 143, 181, and 188.)

Lowerre, B. T. (1976). *The Harpy Speech Recognition System*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA. (See pages 8, 79, and 80.)

Luckhardt, C. A. and Irani, K. B. (1986). An Algorithmic Solution of N-Person Games. In Kehler, T., editor, *5th National Conference on Artificial Intelligence, AAAI 1986*, volume 1, pages 158–162. (See page 5.)

Markovitch, S. and Sella, Y. (1996). Learning of Resource Allocation Strategies for Game Playing. *Computational Intelligence*, 12(1):88–105. (See pages 107 and 109.)

Matsumoto, S., Hirosue, N., Itonaga, K., Yokoo, K., and Futahashi, H. (2010). Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem. In *International MultiConference of Engineers and Computer Scientists 2010*, volume 3, pages 2086–2091. (See page 44.)

McCarthy, J. (1990). Chess as the Drosophila of AI. In Marsland, T. A. and Schaeffer, J., editors, *Computers, Chess, and Cognition*, pages 227–238. Springer. (See page 2.)

Méhat, J. and Cazenave, T. (2010). Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277. (See page 59.)

Michie, D. (1966). Game-Playing and Game-Learning Automata. In Fox, L., editor, *Advances in Programming and Non-Numerical Computation*, pages 183–200. Pergamon Press. (See page 5.)

Moribe, K. (1985). Chain Shot! *Gekkan ASCII*, November issue. In Japanese. (See page 43.)

Müller, F., Späth, C., Geier, T., and Biundo, S. (2012). Exploiting Expert Knowledge in Factored POMPDs. In De Raedt, L., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., and Lucas, P. J. F., editors, *20th European Conference on Artificial Intelligence, ECAI 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 606–611. (See page 6.)

Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134(1-2):145–179. (See page 48.)

Nijssen, J. A. M. (2013). *Monte-Carlo Tree Search for Multi-Player Games*. PhD thesis, Maastricht University, Maastricht, The Netherlands. (See pages 188, 209, and 211.)

Nijssen, J. A. M. and Winands, M. H. M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. In van den Herik, H. J., Iida, H., and Plaat, A., editors, *7th International Conference on Computers and Games, CG 2010. Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 238–249. (See page 143.)

Nijssen, J. A. M. and Winands, M. H. M. (2012). Playout Search for Monte-Carlo Tree Search in Multi-player Games. In van den Herik, H. J. and Plaat, A., editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 72–83. (See pages 144, 177, 181, and 182.)

Nijssen, J. A. M. and Winands, M. H. M. (2013). Search Policies in Multi-Player Games. *ICGA Journal*, 36(1):3–21. (See pages 6 and 52.)

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311. (See page 3.)

Pellegrino, S., Hubbard, A., Galbraith, J., Drake, P., and Chen, Y.-P. (2009). Localizing Search in Monte-Carlo Go Using Statistical Covariance. *ICGA Journal*, 32(3):154–160. (See page 111.)

Pepels, T., Winands, M. H. M., and Lanctot, M. (2014). Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257. (See page 6.)

Perez, D., Powley, E. J., Whitehouse, D., Rohlfshagen, P., Samothrakis, S., Cowling, P. I., and Lucas, S. M. (2014a). Solving the Physical Traveling Salesman Problem: Tree Search and Macro Actions. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):31–45. (See page 6.)

Perez, D., Samothrakis, S., and Lucas, S. M. (2014b). Knowledge-Based Fast Evolutionary MCTS for General Video Game Playing. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014*, pages 68–75. (See page 6.)

Perick, P., St-Pierre, D. L., Maes, F., and Ernst, D. (2012). Comparison of Different Selection Strategies in Monte-Carlo Tree Search for the Game of Tron. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pages 242–249. (See pages 6 and 38.)

Previti, A., Ramanujan, R., Schaerf, M., and Selman, B. (2011). Monte-Carlo Style UCT Search for Boolean Satisfiability. In Pirrone, R. and Sorbello, F., editors, *12th International Conference of the Italian Association for Artificial Intelligence, AI*IA 2011*, volume 6934 of *Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence)*, pages 177–188. (See page 6.)

Ramanujan, R., Sabharwal, A., and Selman, B. (2010a). On Adversarial Search Spaces and Sampling-Based Planning. In Brafman, R. I., Geffner, H., Hoffmann, J., and Kautz, H. A., editors, *20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, pages 242–245. (See pages 6, 8, 52, 129, 141, 143, 150, 201, and 237.)

Ramanujan, R., Sabharwal, A., and Selman, B. (2010b). Understanding Sampling Style Adversarial Search Methods. In Grünwald, P. and Spirtes, P., editors, *26th Conference on Uncertainty in Artificial Intelligence, UAI 2010*, pages 474–483. (See pages 144, 181, 205, and 229.)

Ramanujan, R., Sabharwal, A., and Selman, B. (2011). On the Behavior of UCT in Synthetic Search Spaces. In *ICAPS 2011 Workshop on Monte-Carlo Tree Search: Theory and Applications*. (See page 143.)

Ramanujan, R. and Selman, B. (2011). Trade-Offs in Sampling-Based Adversarial Planning. In Bacchus, F., Domshlak, C., Edelkamp, S., and Helmert, M., editors, *21st International Conference on Automated Planning and Scheduling, ICAPS 2011*, pages 202–209. (See pages 143, 144, and 180.)

Reddy, R. (1988). Foundations and Grand Challenges of Artificial Intelligence: AAAI Presidential Address. *AI Magazine*, 9(4):9–21. (See page 2.)

Rimmel, A. and Teytaud, F. (2010). Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In Chio, C. D., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Goh, C. K., Guervós, J. J. M., Neri, F., Preuss, M., Togelius, J., and Yannakakis, G. N., editors, *Applications of Evolutionary Computation, EvoApplicatons 2010*, volume 6024 of *Lecture Notes in Computer Science*, pages 201–210. (See pages 41 and 57.)

Rimmel, A., Teytaud, F., and Cazenave, T. (2011). Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows. In Chio, C. D., Brabazon, A., Caro, G. A. D., Drechsler, R., Farooq, M., Grahl, J., Greenfield, G., Prins, C., Romero, J., Squillero, G., Tarantino, E., Tettamanzi, A., Urquhart, N., and Etaner-Uyar, A. S., editors, *Applications of Evolutionary Computation - EvoApplications 2011*, volume 6625 of *Lecture Notes in Computer Science*, pages 501–510. (See pages 6 and 59.)

Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematics Society*, 58(5):527–535. (See page 35.)

Robilliard, D., Fonlupt, C., and Teytaud, F. (2014). Monte-Carlo Tree Search for the Game of "7 Wonders". In Cazenave, T., Winands, M. H. M., and Björnsson, Y., editors, *Computer Games Workshop at 21st European Conference on Artificial Intelligence, ECAI 2014*, volume 504 of *Communications in Computer and Information Science*, pages 64–77. (See page 7.)

Rosenbloom, P. S. (1982). A World-Championship-Level Othello Program. *Artificial Intelligence*, 19(3):279–320. (See page 52.)

Rosin, C. D. (2011). Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In Walsh, T., editor, *22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, pages 649–654. (See page 59.)

Ruijl, B., Vermaseren, J., Plaat, A., and van den Herik, H. J. (2014). Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. In *6th International Conference on Agents and Artificial Intelligence, ICAART 2014*, pages 724–731. (See page 6.)

Russell, S. J. and Norvig, P. (2002). *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, USA. (See page 24.)

Sabharwal, A., Samulowitz, H., and Reddy, C. (2012). Guiding Combinatorial Optimization with UCT. In Beldiceanu, N., Jussien, N., and Pinson, É., editors, *9th International Conference on Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimization Problems, CPAIOR 2012*, volume 7298 of *Lecture Notes in Computer Science*, pages 356–361. (See page 6.)

Sabuncuoglu, I. and Bayiz, M. (1999). Job Shop Scheduling with Beam Search. *European Journal of Operational Research*, 118(2):390–412. (See page 79.)

Sadikov, A. and Bratko, I. (2007). Solving 20×20 Puzzles. In van den Herik, H. J., Uiterwijk, J. W. H. M., Winands, M. H. M., and Schadd, M. P. D., editors, *Proceedings of Computer Games Workshop 2007*, pages 157–164. (See page 5.)

Saffidine, A., Jouandeau, N., and Cazenave, T. (2012). Solving Breakthrough with Race Patterns and Job-Level Proof Number Search. In van den Herik, H. J. and Plaat, A., editors, *13th International Conference on Advances in Computer Games, ACG 2011*, volume 7168 of *Lecture Notes in Computer Science*, pages 196–207. (See page 50.)

Saito, J.-T., Chaslot, G., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2007). Monte-Carlo Proof-Number Search for Computer Go. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M., editors, *5th International Conference on Computers and Games, CG 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 50–61. (See page 144.)

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):211–229. (See pages 2 and 14.)

Sato, Y., Takahashi, D., and Grimbergen, R. (2010). A Shogi Program Based on Monte-Carlo Tree Search. *ICGA Journal*, 33(2):80–92. (See pages 52 and 143.)

Schadd, M. (2011). *Selective Search in Games of Different Complexity*. PhD thesis, Maastricht University, Maastricht, The Netherlands. (See page 46.)

Schadd, M. P. D. and Winands, M. H. M. (2011). Best Reply Search for Multiplayer Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):57–66. (See page 52.)

Schadd, M. P. D., Winands, M. H. M., Tak, M. J. W., and Uiterwijk, J. W. H. M. (2012). Single-Player Monte-Carlo Tree Search for SameGame. *Knowledge-Based Systems*, 34:3–11. (See pages 44, 64, 79, and 81.)

Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., and Aldewereld, H. (2008a). Addressing NP-Complete Puzzles with Monte-Carlo Methods. In *AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning*, volume 9, pages 55–61. (See page 44.)

Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J.-B., and Uiterwijk, J. W. H. M. (2008b). Single-Player Monte-Carlo Tree Search. In van den Herik, H. J., Xu, X., Ma, Z., and Winands, M. H. M., editors, *6th International Conference on Computers and Games, CG 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 1–12. (See pages 5, 6, 33, 44, 64, 71, and 234.)

Schaul, T. (2014). An Extensible Description Language for Video Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331. (See page 2.)

Schrum, J., Karpov, I., and Miikkulainen, R. (2011). UT^2: Human-like Behavior via Neuroevolution of Combat Behavior and Replay of Human Traces. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, pages 329–336. (See page 3.)

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(314):256–275. (See pages 2, 14, and 23.)

Sharma, S., Kobti, Z., and Goodwin, S. D. (2008). Knowledge Generation for Improving Simulations in UCT for General Game Playing. In Wobcke, W. and Zhang, M., editors, *21st Australasian Conference on Artificial Intelligence, AI 2008*, volume 5360 of *Lecture Notes in Computer Science*, pages 49–55. (See pages 49 and 50.)

Sheppard, B. (2002). World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275. (See pages 26 and 181.)

Silver, D. and Tesauro, G. (2009). Monte-Carlo Simulation Balancing. In Danyluk, A. P., Bottou, L., and Littman, M. L., editors, *26th Annual International Conference on Machine Learning, ICML 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 945–952. (See pages 57 and 189.)

Silver, D. and Veness, J. (2010). Monte-Carlo Planning in Large POMDPs. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., editors, *23th Conference on Advances in Neural Information Processing Systems, NIPS 2010*, pages 2164–2172. (See pages 6, 78, and 240.)

Simon, H. A. and Newell, A. (1957). Heuristic Problem Solving: The Next Advance in Operations Research. *Operations Research*, 6(1):1–10. (See page 2.)

Šolak, R. and Vučković, V. (2009). Time Management during a Chess Game. *ICGA Journal*, 32(4):206–220. (See pages 107 and 109.)

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-Player Games. *ICGA Journal*, 31(4):195–208. (See pages 6 and 182.)

Sturtevant, N. R. and Buro, M. (2005). Partial Pathfinding Using Map Abstraction and Refinement. In Veloso, M. M. and Kambhampati, S., editors, *Twentieth National Conference on Artificial Intelligence, AAAI 2005*, pages 1392–1397. (See page 5.)

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA. (See pages 34, 36, and 37.)

Syed, O. and Syed, A. (2003). Arimaa—A New Game Designed to be Difficult for Computers. *ICGA Journal*, 26(2):138–139. (See page 2.)

Tak, M. J. W., Lanctot, M., and Winands, M. H. M. (2014). Monte Carlo Tree Search Variants for Simultaneous Move Games. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014*, pages 232–239. (See page 6.)

Tak, M. J. W., Winands, M. H. M., and Björnsson, Y. (2012). N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83. (See pages 41, 50, 52, and 57.)

Takes, F. W. and Kosters, W. A. (2009). Solving SameGame and its Chessboard Variant. In Calders, T., Tuyls, K., and Pechenizkiy, M., editors, *21st Benelux Conference on Artificial Intelligence, BNAIC 2009*, pages 249–256. (See page 44.)

Tanaka, T. (2009). An Analysis of a Board Game "Doubutsu Shogi". *IPSJ SIG Notes*, 2009-GI-22(3):1–8. In Japanese. (See page 52.)

Tesauro, G. and Galperin, G. R. (1997). On-line Policy Improvement using Monte-Carlo Search. In Mozer, M., Jordan, M. I., and Petsche, T., editors, *9th Conference on Advances in Neural Information Processing Systems, NIPS 1996*, pages 1068–1074. (See pages 26 and 58.)

Teytaud, F. and Teytaud, O. (2009). Creating an Upper-Confidence-Tree Program for Havannah. In van den Herik, H. J. and Spronck, P., editors, *12th International Conference on Advances in Computer Games, ACG 2009*, volume 6048 of *Lecture Notes in Computer Science*, pages 65–74. (See page 2.)

Teytaud, F. and Teytaud, O. (2010). On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. In Yannakakis, G. N. and Togelius, J., editors,

2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, pages 359–364. (See page 144.)

Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. In Fox, M. and Poole, D., editors, *24th AAAI Conference on Artificial Intelligence, AAAI 2010*, pages 994–999. (See page 2.)

Tillmann, C. and Ney, H. (2003). Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation. *Computational Linguistics*, 29(1):97–133. (See page 79.)

Tromp, J. (2008). Solving Connect-4 on Medium Board Sizes. *ICGA Journal*, 31(2):110–112. (See page 49.)

Tromp, J. and Farnebäck, G. (2007). Combinatorics of Go. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M., editors, *5th International Conference on Computers and Games, CG 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 84–99. (See page 48.)

Turing, A. (1953). Chess. In Bowden, B. V., editor, *Faster than Thought*, pages 286–295. Pitman, London. (See page 2.)

van Eyck, J., Ramon, J., Güiza, F., Meyfroidt, G., Bruynooghe, M., and den Berghe, G. V. (2013). Guided Monte Carlo Tree Search for Planning in Learned Environments. In *5th Asian Conference on Machine Learning, ACML 2013*, pages 33–47. (See page 6.)

von Neumann, J. and Morgenstern, O. (1944). *The Theory of Games and Economic Behavior, Second Edition*. Princeton University Press, Princeton, NJ, USA. (See pages 4 and 16.)

Whitehouse, D., Powley, E. J., and Cowling, P. I. (2011). Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, pages 87–94. (See page 6.)

Winands, M. H. M. and Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. In van den Herik, H. J. and Spronck, P., editors, *12th International Conference on Advances in Computer Games, ACG 2009*, volume 6048 of *Lecture Notes in Computer Science*, pages 33–44. (See page 211.)

Winands, M. H. M. and Björnsson, Y. (2011). Alpha-Beta-based Play-outs in Monte-Carlo Tree Search. In Cho, S.-B., Lucas, S. M., and Hingston, P., editors, *2011 IEEE*

Conference on Computational Intelligence and Games, CIG 2011, pages 110–117. (See pages 144, 177, 181, 182, 184, and 207.)

Winands, M. H. M., Björnsson, Y., and Saito, J.-T. (2008). Monte-Carlo Tree Search Solver. In van den Herik, H. J., Xu, X., Ma, Z., and Winands, M. H. M., editors, *6th International Conference on Computers and Games, CG 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 25–36. (See pages 33, 142, 143, and 144.)

Winands, M. H. M., Björnsson, Y., and Saito, J.-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250. (See pages 6, 33, 141, 181, 182, 184, and 207.)

Winands, M. H. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2001). Combining Proof-Number Search with Alpha-Beta Search. In Kröse, B., de Rijke, M., Schreiber, G., and van Someren, M., editors, *13th Belgium-Netherlands Conference on Artificial Intelligence, BNAIC 2001*, pages 299–306. (See page 144.)

Yan, X., Diaconis, P., Rusmevichientong, P., and Roy, B. V. (2004). Solitaire: Man Versus Machine. In Saul, L. K., Weiss, Y., and Bottou, L., editors, *17th Conference on Advances in Neural Information Processing Systems, NIPS 2004*, pages 1553–1560. (See page 58.)

Zhou, R. and Hansen, E. A. (2005). Beam-Stack Search: Integrating Backtracking with Beam Search. In Biundo, S., Myers, K. L., and Rajan, K., editors, *15th International Conference on Automated Planning and Scheduling, ICAPS 2005*, pages 90–98. (See page 80.)

Zhou, R. and Hansen, E. A. (2006). Breadth-First Heuristic Search. *Artificial Intelligence*, 170(4-5):385–408. (See page 79.)

Index

Summary

This thesis is concerned with enhancing the technique of *Monte-Carlo Tree Search* (MCTS), applied to making move decisions in games. MCTS has become the dominating paradigm in the challenging field of computer Go, and has been successfully applied to many other games and non-game domains as well. It is now an active and promising research topic with room for improvements in various directions. This thesis focuses on enhancing MCTS in one-player and two-player domains.

Chapter 1 provides a brief introduction to the field of games and AI, and presents the following problem statement guiding the research.

Problem Statement: *How can the performance of Monte-Carlo Tree Search in a given one- or two-player domain be improved?*

Four research questions have been formulated to approach this problem statement. Two questions are concerned with one-player domains, while two questions are dealing with adversarial two-player domains. The four research questions address (1) the rollout phase of MCTS in one-player domains, (2) the selection phase of MCTS in one-player domains, (3) time management for MCTS in two-player tournament play, and (4) combining the strengths of minimax and MCTS in two-player domains.

Chapter 2 describes the basic terms and concepts of search in games. It also introduces the two classes of search methods used in the thesis: minimax-based search techniques for two-player games, and MCTS techniques for both one- and two-player games. Enhancements for both minimax and MCTS are explained as far as relevant for this thesis.

Chapter 3 introduces the test domains used in the following chapters. These include the one-player games SameGame, Clickomania, and Bubble Breaker, and the two-player games Go, Connect-4, Breakthrough, Othello, and Catch the Lion. For each game, its origin is described, its rules are outlined, and its complexity is analyzed.

In MCTS, every state in the search tree is evaluated by the average outcome of Monte-Carlo rollouts from that state. For the consistency of MCTS, i.e. for the convergence to the optimal policy, uniformly random rollout moves are sufficient.

However, stronger rollout strategies typically greatly speed up convergence. The strength of Monte-Carlo rollouts can be improved for example through hand-coded knowledge, or by automated offline tuning of rollout policies. In recent years, the topic of improving rollout policies online has received more and more attention, i.e. while the search is running. This leads to the first research question.

Research Question 1: *How can the rollout quality of MCTS in one-player domains be improved?*

Chapter 4 answers this research question by introducing Nested Monte-Carlo Tree Search (NMCTS), replacing simple rollouts with nested MCTS searches. Instead of improving a given set of rollout policy parameters either offline or online, calls to the rollout policy are replaced with calls to MCTS itself. Independent of the quality of the base-level rollouts, this recursive use of MCTS makes higher-quality rollouts available at higher levels, improving MCTS especially when longer search times are available. NMCTS is a generalization of regular MCTS, which is equivalent to level-1 NMCTS. Additionally, NMCTS can be seen as a generalization of Nested Monte-Carlo Search (NMCS), allowing for an exploration-exploitation tradeoff by nesting MCTS instead of naive Monte-Carlo search. The approach was tested in the puzzles SameGame, Clickomania, and Bubble Breaker, with relatively long time settings.

As it is known for SameGame that restarting several short MCTS runs on the same problem can lead to better performance than a single, long run, NMCTS was compared to multi-start MCTS. Level-2 NMCTS was found to significantly outperform multi-start MCTS in all test domains. In Clickomania, memory limitations limited the performance of very long searches, which gave an additional advantage to NMCTS.

Furthermore, level-2 NMCTS was experimentally compared to its special case of NMCS. For this comparison, NMCTS (just like NMCS) used the additional technique of move-by-move search, distributing the total search time over several or all moves in the game instead of conducting only one global search from the initial position. NMCTS significantly outperformed both level-2 and level-3 NMCS in all test domains. Since both MCTS and NMCS represent specific parameter settings of NMCTS, correct tuning of NMCTS has to lead to greater or equal success in any domain.

In *Upper Confidence bounds applied to Trees* or UCT, the most widely used variant of MCTS, the selectivity of the search can be controlled with a single parameter: the exploration factor. In domains with long solution lengths or when searching with a short time limit however, MCTS might not be able to grow a search tree deep enough even when exploration is completely turned off. The result is a search process that spends too much time on optimizing the first steps of the solution, but not enough time on optimizing the last steps. This leads to the second research question.

Research Question 2: *How can the selectivity of MCTS in one-player domains be improved?*

Chapter 5 answers this research question by proposing Beam Monte-Carlo Tree Search (BMCTS), a combination of MCTS with the idea of beam search. BMCTS expands a tree whose size is linear in the search depth, making MCTS more effective especially in domains with long solution lengths or short time limits. Like MCTS, BMCTS builds a search tree using Monte-Carlo simulations as state evaluations. When a predetermined number of simulations has traversed the nodes of a given tree depth, these nodes are sorted by a heuristic value, and only a fixed number of them is selected for further exploration. BMCTS is reduced to a variant of move-by-move MCTS if this number, the beam width, is set to one. However, it generalizes from move-by-move search as it allows to keep any chosen number of alternative moves when moving on to the next tree depth. The test domains for the approach were again SameGame, Clickomania, and Bubble Breaker, with relatively short time settings.

BMCTS was compared to MCTS both using one search run per position (single-start), and using the maximum over multiple search runs per position (multi-start). The experimental results show BMCTS to significantly outperform regular MCTS at a wide range of time settings in all tested games. BMCTS was also shown to have a larger advantage over MCTS in the multi-start scenario. In all test domains, multi-start BMCTS was stronger than multi-start MCTS at a wider range of time settings than single-start BMCTS to single-start MCTS. This suggests that the performance of BMCTS tends to have a higher variance than the performance of regular MCTS, even in some cases where the two algorithms perform equally well on average.

This observation led to the idea of examining how BMCTS would perform in a nested setting, i.e. replacing MCTS as the basic search algorithm in Nested Monte-Carlo Tree Search. The resulting search algorithm was called Nested Beam Monte-Carlo Tree Search (NBMCTS), and experiments showed NBMCTS to be the overall strongest one-player search technique proposed in this thesis. It performed better than or equal to NMCTS in all domains and at all search times.

In competitive gameplay, time is typically limited—in the basic case by a fixed time budget per player for the entire game (*sudden-death* time control). Exceeding this time budget means an instant loss for the respective player. However, longer thinking times usually result in better moves, especially for an *anytime* algorithm like MCTS. The question arises how to distribute the time budget wisely among all moves in the game. This leads to the third research question.

Research Question 3: *How can the time management of MCTS in two-player domains be improved?*

Chapter 6 answers this research question by investigating and comparing a variety of time-management strategies for MCTS. The strategies included newly proposed ones as well as strategies described in the literature, partly in enhanced form. The strategies can be divided into semi-dynamic strategies that decide about time allocation for each search before it is started, and dynamic strategies that influence the duration of each move search while it is already running. All strategies were tested in the domains of 13×13 and 19×19 Go, and the domain-independent ones in Connect-4, Breakthrough, Othello, and Catch the Lion.

Experimental results showed the proposed strategy called STOP to be most successful. STOP is based on the idea of estimating the remaining number of moves in the game and using a corresponding fraction of the available search time. However, MCTS is stopped as soon as it appears unlikely that the final move selection will change by continuing the search. The time saved by these early stops throughout the game is anticipated, and earlier searches in the game are prolonged in order not to have only later searches profit from accumulated time savings. In self-play, this strategy won approximately 60% of games against state-of-the-art time management, both in 13×13 and 19×19 Go and under various time controls. Furthermore, comparison across different games showed that the domain-independent strategy STOP is the strongest of all tested time-management strategies.

All time-management strategies that *prolong* search times when certain criteria are met take available time from the endgame and shift it to the opening of the game. All strategies that *shorten* search times based on certain criteria move time from the opening towards the endgame instead. Analysis showed that such a shift can have a positive or negative effect. Consequently, a methodology was developed to isolate the effect of a shift and judge the effect of a given strategy independently of it. Should the shifting effect be negative, it can be counteracted by introducing an explicit shift in the opposite direction.

One of the characteristics of MCTS is Monte-Carlo simulation, taking distant consequences of moves into account and therefore providing a strategic advantage in many domains over traditional depth-limited minimax search. However, minimax with $\alpha\beta$ pruning considers every relevant move within the search horizon and can therefore have a tactical advantage over the highly selective MCTS approach, which might miss an important move when precise short-term play is required. This is especially a problem in games with a high number of terminal states throughout the search space, where weak short-term play can lead to a sudden loss. This leads to the fourth research question.

Research Question 4: *How can the tactical strength of MCTS in two-player domains be improved?*

This research question is answered by proposing and testing *MCTS-minimax hybrids*, integrating shallow minimax searches into the MCTS framework and thus taking a first step towards combining the strengths of MCTS and minimax. These hybrids can be divided into approaches that require domain knowledge, and approaches that are knowledge-free.

Chapter 7 studies three different hybrids for the knowledge-free case, using minimax in the selection/expansion phase (MCTS-MS), the rollout phase (MCTS-MR), and the backpropagation phase of MCTS (MCTS-MB). Test domains were Connect-4, Breakthrough, Othello, and Catch the Lion. The newly proposed variant MCTS-MS significantly outperformed regular MCTS with the MCTS-Solver extension in Catch the Lion, Breakthrough, and Connect-4. The same held for the proposed MCTS-MB variant in Catch the Lion and Breakthrough, while the effect in Connect-4 was neither significantly positive nor negative. The only way of integrating minimax search into MCTS known from the literature, MCTS-MR, was quite strong in Catch the Lion and Connect-4 but significantly weaker than the baseline in Breakthrough, suggesting it might be less robust with regard to differences between domains such as the average branching factor. As expected, none of the MCTS-minimax hybrids had a positive effect in Othello due to the low number of terminal states and shallow traps throughout its search space. The density and difficulty of traps predicted the relative performance of MCTS-minimax hybrids across domains well.

Problematic domains for the knowledge-free MCTS-minimax hybrids can be addressed with the help of domain knowledge. On the one hand, domain knowledge can be incorporated into the hybrid algorithms in the form of evaluation functions. This can make minimax potentially much more useful in search spaces with few terminal nodes before the latest game phase, such as that of Othello. On the other hand, domain knowledge can be incorporated in the form of a move ordering function. This can be effective in games such as Breakthrough, where traps are relatively frequent, but the branching factor seems to be too high for some hybrids such as MCTS-MR.

Chapter 8 therefore investigates three more algorithms for the case where domain knowledge is available, employing heuristic state evaluations to improve the rollout policy (MCTS-IR), to terminate rollouts early (MCTS-IC), or to bias the selection of moves in the MCTS tree (MCTS-IP). For all three approaches, the computation of state evaluations through simple evaluation function calls (MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E, where -E stands for "evaluation function") was compared to the computation of state evaluations through shallow-depth minimax searches using the same heuristic knowledge (MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M, where -M

stands for "minimax"). MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M are MCTS-minimax hybrids. The integration of minimax has only been applied to MCTS-IR before in this form. Test domains were Breakthrough, Othello, and Catch the Lion.

The hybrids were combined with move ordering and k-best pruning to cope with the problem of higher branching factors, resulting in the enhanced hybrid players MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. Results showed that with these two relatively simple $\alpha\beta$ enhancements, MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid investigated in this thesis in all three tested domains. Because MCTS-IP-M-k calls minimax less frequently, it performs better than the other hybrids at low time settings when performance is most sensitive to a reduction in speed. MCTS-IP-M-k also worked well on an enlarged Breakthrough board, demonstrating the suitability of the technique for domains with higher branching factors. Moreover, the best-performing hybrid outperformed a simple $\alpha\beta$ implementation in Breakthrough, demonstrating that at least in this domain, MCTS and minimax can be combined to an algorithm stronger than its parts.

Chapter 9 finally concludes the thesis, and gives possible directions for future work. The answer to the problem statement is twofold. First, the performance of MCTS in one-player domains can be improved in two ways—by nesting MCTS searches (NMCTS), and by combining MCTS with beam search (BMCTS). The first method is especially interesting for longer search times, and the second method for shorter search times. A combination of NMCTS and BMCTS can also be effective. Second, the performance of MCTS in two-player domains can be improved in two ways as well—by using the available time for the entire game more smartly (time management), and by integrating shallow minimax searches into MCTS (MCTS-minimax hybrids). The first approach is relevant to scenarios such as tournaments where the time per game is limited. The second approach improves the performance of MCTS specifically in tactical domains.

Several areas of future research are indicated. These include (1) applying NMCTS to non-deterministic and partially observable domains, (2) enhancing BMCTS to guarantee optimal behavior in the limit, (3) testing various combinations of time management strategies, and (4) examining the paradoxical observations of weaker performance with deeper embedded searches in MCTS-minimax hybrids, as well as studying which properties of test domains influence the performance of these hybrids.

Samenvatting

Dit proefschrift houdt zich bezig met het verbeteren van de *Monte-Carlo Tree Search* (MCTS) techniek om zodoende beslissingen te nemen in abstracte spelen. De laatste jaren is MCTS het dominante paradigma geworden in allerlei speldomeinen zoals Go. Tevens is het succesvol toegepast in verschillende optimaliseringsproblemen. MCTS is tegenwoordig een actief en veelbelovend onderzoeksdomein met ruimte voor verbeteringen in verscheidene richtingen. Dit proefschrift richt zich op het verder ontwikkelen van MCTS voor één- en tweespeler domeinen.

Hoofdstuk 1 geeft een kort introductie over de rol die spelen vervullen in de kunstmatige intelligentie. De volgende probleemstelling is geformuleerd om het onderzoek te sturen.

Probleemstelling: *Hoe kunnen we de prestatie van Monte-Carlo Tree Search verbeteren voor een gegeven één- of tweespeler domein?*

Vier onderzoeksvragen zijn geformuleerd voor het aanpakken van deze probleemstelling. Twee vragen houden zich bezig met éénspeler domeinen, terwijl de andere twee zich richten op tweespeler domeinen. De vier onderzoeksvragen adresseren (1) de Monte-Carlo simulatiefase van MCTS in éénspeler domeinen, (2) de selectiefase van MCTS in éénspeler domeinen, (3) tijdmanagement voor MCTS in tweespeler toernooien, en (4) het combineren van de kracht van minimax en MCTS in tweespeler domeinen.

Hoofdstuk 2 beschrijft de basisbegrippen en concepten voor het zoeken in spelen. Het introduceert ook twee klassen van zoekmethoden die worden gebruikt in het proefschrift: minimax technieken voor tweespeler domeinen, en MCTS technieken voor één- en tweespeler domeinen. Uitbreidingen voor zowel minimax als MCTS worden uitgelegd zover ze relevant zijn voor het proefschrift.

Hoofstuk 3 introduceert de gebruikte testdomeinen in het proefschrift. Deze bevatten de éénspeler domeinen SameGame, Clickomania en Bubble Breaker, en de tweespeler domeinen Go, Vier op 'n rij, Breakthrough, Othello en Catch the Lion. Voor elk spel wordt de achtergrond beschreven, de regels geschetst en de complexiteit geanalyseerd.

De sterkte van MCTS wordt mede bepaald door Monte-Carlo simulaties. Tijdens deze fase wordt zetten geselecteerd op basis van een simulatiestrategie. Een dergelijke strategie kan gebaseerd zijn op gecodeerde expertkennis of automatisch zijn verkregen tijdens het zoeken. Dit heeft geleid tot de eerste onderzoeksvraag.

Onderzoeksvraag 1: *Hoe kan de kwaliteit van de Monte-Carlo simulaties in MCTS worden verbeterd voor éénspeler domeinen?*

Hoofdstuk 4 beantwoordt deze onderzoeksvraag door Nested Monte-Carlo Tree Search (NMCTS) te introduceren. Het vervangt de Monte-Carlo simulaties door geneste aanroepen van MCTS. Onafhankelijk van de kwaliteit van de Monte-Carlo simulaties op het basis niveau, zorgt dit recursief gebruik van MCTS voor een betere kwaliteit van de simulaties op de hogere niveaus. Dit geldt vooral als er veel zoektijd beschikbaar is. NMCTS kan gezien worden als een generalisatie van de reguliere MCTS en de Nested Monte-Carlo Search (NMCS). Deze aanpak is getest met relatief hoge tijdsinstellingen in SameGame, Clickomania en Bubble Breaker.

Er wordt aangetoond dat NMCTS beter presteert dan multi-start MCTS in alle testdomeinen. Verder wordt NMCTS vergeleken met NMCS. Het blijkt dat NMCTS significant beter presteert dan NMCS in alle testdomeinen. Aangezien MCTS en NMCS specifieke instanties zijn van NMCTS, leidt het correct afstellen van NMCTS altijd tot betere dan wel gelijke prestaties in elk domein.

Upper Confidence bounds applied to Trees ofwel UCT is de meest gebruikte variant van MCTS. In deze variant wordt de selectiviteit van het zoekproces gecontroleerd door één parameter, de zogenaamde exploratiefactor. In domeinen met een lange oplossingslengte of een korte zoektijd kan het zo zijn dat MCTS niet diep genoeg komt. Het resultaat is dat het zoekproces te veel tijd spendeert aan de eerste stappen van de oplossing, maar niet meer genoeg tijd heeft voor de laatste stappen. Dit heeft geleid tot de tweede onderzoeksvraag.

Onderzoeksvraag 2: *Hoe kan de selectiviteit van MCTS in éénspeler domeinen worden verbeterd?*

Hoofstuk 5 beantwoordt deze onderzoeksvraag door Beam Monte-Carlo Tree Search (BMCTS) voor te stellen. Het is een combinatie van MCTS met *beam search*. BMCTS laat een boom groeien waarvan de grootte lineair is aan de zoekdiepte. Dit maakt MCTS meer effectief in domeinen met lange oplossingen of korte zoektijden. Wanneer de knopen op een bepaalde zoekdiepte zijn bezocht door een vooraf ingesteld aantal simulaties, worden ze gesorteerd op basis van hun waarde en worden de beste geselecteerd voor verdere exploratie. De testdomeinen zijn wederom SameGame, Clickomania en Bubble Breaker. Echter de zoektijden zijn deze keer relatief kort.

BMCTS is vergeleken met MCTS waar beiden ofwel één zoekproces gebruiken per positie (single-start), ofwel meerdere zoekprocessen gebruiken per positie (multi-start). Voor het eerste scenario laten de experimenten zien dat BMCTS significant beter presteert dan MCTS in alle domeinen voor een bepaalde reeks van zoektijden. Voor het tweede scenario is multi-start BMCTS sterker dan multi-start MCTS voor een grotere reeks van zoektijden dan in de voorgaande vergelijking.

Deze positieve resultaten hebben aanleiding gegeven tot het idee om BMCTS te combineren met NMCTS. Het heeft geresulteerd tot het zoekalgoritme Nested Beam Monte-Carlo Tree Search (NBMCTS). Experimenten tonen aan dat NBMCTS de beste éénspeler zoektechniek voorgesteld in dit proefschrift is. NBMCTS presteert beter dan wel gelijk aan NMCTS in alle domeinen voor alle zoektijden.

Tijd is typisch gelimiteerd in tweespeler toernooien. In de basis variant is er een vaste hoeveelheid tijd voor de gehele partij. Overschrijding van de tijd betekent een onmiddellijke nederlaag voor de betreffende speler. Echter meer bedenktijd voor een zoektechniek zoals MCTS resulteert in het algemeen in betere zetten. De vraag rijst dan hoe de tijd wijselijk te verdelen over alle zetten in de partij. Dit heeft geleid tot de derde onderzoeksvraag.

Onderzoeksvraag 3: *Hoe kan het tijdmanagement van MCTS in tweespeler domeinen worden verbeterd?*

Hoofstuk 6 beantwoordt deze onderzoeksvraag door het onderzoeken en vergelijken van een verscheidenheid van tijdmanagementstrategieën voor MCTS. De strategieën zijn ofwel nieuw, ofwel beschreven in de literatuur, ofwel verbeterd. De strategieën kunnen worden onderverdeeld in semi-dynamische strategieën, die de hoeveelheid zoektijd bepalen voor elke zoekproces voordat het wordt gestart, en dynamische strategieën die de duur van elke zoekproces beïnvloeden, terwijl die al wordt uitgevoerd. Alle strategieën zijn getest in de domeinen van 13×13 en 19×19 Go. De domein onafhankelijke strategieën zijn tevens onderzocht in Vier op 'n rij, Breakthrough, Othello en Catch the Lion.

Experimentele resultaten tonen aan dat de voorgestelde strategie STOP het meest succesvol is. De strategie is gebaseerd op het idee van het schatten van het resterende aantal zetten in het spel en gebruik te maken van een overeenkomstige fractie van de beschikbare zoektijd. Echter, het MCTS zoekproces wordt gestopt zodra het onwaarschijnlijk is dat de huidige beste zet zal veranderen als men langer zou blijven denken. Er wordt tevens rekening gehouden met deze tijdsbesparingen gedurende de partij. Zoekprocessen vroeg in het spel krijgen relatief meer tijd, zodat niet alleen de latere zoekprocessen profiteren van de opgebouwde tijdwinst. Deze strategie wint ongeveer 60% van de partijen tegen de beste tijdmanagementstrategie tot nu toe,

zowel in 13×13 en 19×19 Go voor verschillende tijdsinstellingen. Bovendien laten experimenten in verschillende spelen zien dat de strategie STOP de sterkste is van alle geteste tijdmanagementstrategieën.

Alle tijdmanagementstrategieën die onder bepaalde voorwaarden de zoektijd verlengen, verschuiven tijd van de latere fasen naar de vroegere fasen van het spel. Alle strategieën die onder bepaalde voorwaarden de zoektijd verkorten verschuiven tijd van de opening naar het eindspel. Verdere analyse toont aan dat deze verschuiving een positief of negatief effect kan hebben. Bijgevolg is een methode ontwikkeld om het effect van deze verschuiving te isoleren en het effect onafhankelijk van de strategie te beoordelen. Indien de verschuiving een negatief effect heeft kan dit worden tegengegaan met een expliciete verschuiving in de tegengestelde richting.

Eén van de kenmerken van MCTS is de Monte-Carlo simulatie, die rekening houdt met de lange termijn. Het geeft daarom in veel domeinen een strategisch voordeel ten opzichte van traditionele minimax zoekmethode. Echter, minimax met $\alpha\beta$ snoeiing beschouwt alle relevante zetten binnen de zoekhorizon. Het kan daardoor een tactisch voordeel hebben ten opzichte van de zeer selectieve MCTS aanpak, die een belangrijke zet kan missen wanneer tactisch spel is vereist. Het is vooral een probleem bij spellen met een groot aantal eindtoestanden verspreid over de gehele zoekruimte, waarbij zwak tactisch spel kan leiden tot een plotseling verlies. Dit heeft geleid tot de vierde onderzoeksvraag.

Onderzoeksvraag 4: *Hoe kan de tactische sterkte van MCTS in tweespeler domeinen worden verbeterd?*

Deze onderzoeksvraag wordt beantwoord door het introduceren en testen van *MCTS-minimax hybriden*, die kleine minimax zoekprocessen in het MCTS raamwerk integreren. Het is daarmee een eerste stap in de richting van het combineren van de sterke punten van MCTS en minimax. Deze hybriden kunnen worden onderverdeeld in aanpakken die domeinkennis vereisen en aanpakken die vrij zijn van domeinkennis.

Hoofdstuk 7 bestudeert drie verschillende hybriden voor de kennisvrije situatie. Er wordt gebruik gemaakt van minimax bij de selectie- / expansiefase (MCTS-MS), de simulatiefase (MCTS-MR), en de terugpropagatiefase van MCTS (MCTS-MB). Testdomeinen zijn Vier op 'n rij, Breakthrough, Othello en Catch the Lion. De voorgestelde variant MCTS-MS presteert aanzienlijk beter dan MCTS in Catch the Lion, Breakthrough en Vier op 'n rij. Hetzelfde geldt voor de voorgestelde MCTS-MB variant in Catch the Lion en Breakthrough, terwijl er geen meetbaar effect is in Vier op 'n rij. De enige uit de literatuur bekende manier om minimax te integreren in MCTS, MCTS-MR, is vrij sterk in Catch the Lion en Vier op 'n rij, maar aanzienlijk zwakker dan de reguliere MCTS in Breakthrough. Dit laatste suggereert dat MCTS-MR minder

robuust is met betrekking tot domeinverschillen zoals de gemiddelde vertakkingsgraad. Ten slotte heeft geen van de MCTS-minimax hybriden een positief effect op Othello door het lage aantal eindtoestanden en ondiepe matsituaties over het gehele zoekgebied. De dichtheid en de moeilijkheidsgraad van de matsituaties zijn goede voorspellers voor de relatieve prestaties van de MCTS-minimax hybriden.

Problematische domeinen voor deze MCTS-minimax hybriden kunnen worden aangepakt met behulp van domeinkennis. Enerzijds, kan domeinkennis in de hybride algoritmen worden geïncorporeerd in de vorm van heuristische evaluatiefuncties. Dit kan minimax veel effectiever maken in zoekruimten met relatief weinig eindtoestanden vóór de eindfase, zoals die van Othello. Anderzijds kan domeinkennis in de vorm van een zettenordeningsfunctie worden geïncorporeerd. Dit kan effectief zijn voor spellen zoals Breakthrough, waar matsituaties relatief vaak voorkomen, maar de vertakkingsfactor te hoog lijkt te zijn voor sommige hybriden zoals MCTS-MR.

Hoofdstuk 8 onderzoekt daarom drie algoritmen voor het geval wanneer domeinkennis aanwezig is. Er wordt gebruik gemaakt van heuristische evaluatiefuncties om de simulatiestrategie te verbeteren (MCTS-IR), om Monte-Carlo simulaties voortijdig te stoppen (MCTS-IC), of om de selectie van zetten in de zoekboom te sturen (MCTS-IP). Voor alle drie de aanpakken wordt de waardering van toestanden middels directe aanroepen van de evaluatiefunctie (MCTS-IR-E, MCTS-IC-E en MCTS-IP-E) vergeleken met de waardering middels kleine minimax zoekprocessen gebruikmakend van dezelfde evaluatiefunctie (MCTS-IR-M, MCTS-IC-M, en MCTS-IP-M). Deze laatste drie zijn MCTS-minimax hybriden. De integratie van minimax is alleen door MCTS-IR gebruikt in deze vorm. Testdomeinen zijn Breakthrough, Othello en Catch the Lion.

De hybriden worden gecombineerd met zettenordening en k-best snoeiing om zodoende om te gaan met hogere vertakkingsgraden resulterend in de verbeterde hybride spelers MCTS-IR-Mk, MCTS-IC-Mk en MCTS-IP-Mk. Resultaten tonen aan dat met deze twee relatief eenvoudige $\alpha\beta$ verbeteringen, MCTS-IP-Mk de sterkste MCTS minimax-hybride in alle drie de geteste domeinen is. Omdat MCTS-IP-Mk minimax minder vaak aanroept, presteert het beter dan de andere hybriden bij lagere tijdsinstellingen waar de prestatie van MCTS het meest gevoelig is voor een reductie van het aantal simulaties per seconde. MCTS-IP-Mk werkt ook goed op een groter Breakthrough bord, waaruit de geschiktheid van de techniek voor domeinen met hogere vertakkingsgraden blijkt. Bovendien overtreft de best presterende hybride een $\alpha\beta$ implementatie in Breakthrough, waaruit blijkt dat althans voor dit domein MCTS en minimax kunnen worden gecombineerd in een algoritme dat sterker is dan ieder afzonderlijk.

Hoofstuk 9 sluit het proefschrift af. Het antwoord op de probleemstelling is

tweeledig. Ten eerste kan de prestatie van MCTS in éénspeler domeinen op twee manieren worden verbeterd — door het nesten van MCTS zoekprocessen (NMCTS), en het combineren van MCTS met beam search (BMCTS). De eerste methode is vooral interessant voor langere zoektijden, en de tweede methode meer geschikt voor kortere zoektijden. Een combinatie van NMCTS en BMCTS kan ook effectief zijn.

Ten tweede kan de prestatie van MCTS tevens op twee manieren worden verbeterd — door de beschikbare tijd voor de hele partij op een slimmere manier te gebruiken (tijdmanagement), en door het integreren van kleine minimax zoekprocessen in MCTS (MCTS-minimax hybriden). De eerste aanpak is relevant voor scenario's zoals toernooispel waar de hoeveelheid tijd per partij gelimiteerd is. De tweede aanpak verbetert de prestatie van MCTS specifiek voor tactische domeinen.

Er zijn vier richtingen voor vervolgonderzoek. Deze zijn (1) het toepassen van NMCTS in non-deterministische en gedeeltelijk waarneembare domeinen, (2) het verbeteren van BMCTS om zo in de limiet optimaal gedrag te garanderen, (3) het uittesten van verscheidene tijdmanagementstrategieën, en (4) het onderzoeken van zwakkere prestaties met diepere ingebedde zoekprocessen in MCTS-minimax hybriden, alsmede het bestuderen van welke eigenschappen van de testdomeinen invloed hebben op de prestaties van deze hybriden.

Curriculum Vitae

Hendrik Baier was born on January 20, 1982 in Frankfurt am Main, Germany. He graduated with a B.Sc. in Computer Science from Darmstadt Technical University in 2006, and with a M.Sc. in Cognitive Science from Osnabrück University in 2010 (with distinction).

From October 2010 to September 2014, Hendrik was employed as a Ph.D. researcher in the Games and AI group at the Department of Knowledge Engineering, Maastricht University. His research performed there resulted in several journal and conference publications, and finally this thesis. In addition to his scientific work, he guided and supervised students with their thesis work and in practical sessions of courses such as Computer Science II and Data Structures and Algorithms. Furthermore, he was elected to the University Council of Maastricht University as a representative of Ph.D. students in 2013, and functioned as the Ph.D. Coordinator of the Department of Knowledge Engineering as well. In his spare time, Hendrik was actively involved in Ph.D. Academy Maastricht, an organization aiming to stimulate interaction between Ph.D. students at Maastricht University. He chaired the board of Ph.D. Academy in 2014.

Hendrik continues his work as a postdoctoral researcher at the Advanced Concepts Team of the European Space Agency in Noordwijk, The Netherlands.

SIKS Dissertation Series

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 (TUE / Humboldt-Universitaet 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*

Abbreviations: SIKS – Dutch Research School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; OU – Open Universiteit; RUN – Radboud Universiteit Nijmegen; TUD – Technische Universiteit Delft; TUE – 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.

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 (TUE, Humboldt-Universitaet 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) *A Politiele 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 (TUE) *Correctness of services and their composition*

38 Dirk Fahland (TUE) *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 (TUE) *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain*

45 Vasilios Andrikopoulos (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 Jahn-Takeshi Saito (UM) *Solving difficult game positions*

49 Bouke Huurnink (UVA) *Search in Audiovisual Broadcast Archives*

50 Alia Khairia Amin (CWI) *Understanding and supporting information seeking tasks in multiple sources*

51 Peter-Paul van Maanen (VU) *Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention*

52 Edgar Meij (UVA) *Combining Concepts and Language Models for Information Access*

2011

1 Botond Cseke (RUN) *Variational Algorithms for Bayesian Inference in Latent Gaussian Models*

2 Nick Tinnemeier (UU) *Work flows in Life Science*

3 Jan Martijn van der Werf (TUE) *Compositional Design and Verification of Component-Based Information Systems*

4 Hado van Hasselt (UU) *Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms*

5 Base van der Raadt (VU) *Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.*

6 Yiwen Wang (TUE) *Semantically-Enhanced Recommendations in Cultural Heritage*

7 Yujia Cao (UT) *Multimodal Information Presentation for High Load Human Computer Interaction*

8 Nieske Vergunst (UU) *BDI-based Generation of Robust Task-Oriented Dialogues*

9 Tim de Jong (OU) *Contextualised Mobile Media for Learning*

10 Bart Bogaert (UVT) *Cloud Content Contention*

11 Dhaval Vyas (UT) *Designing for Awareness: An Experience-focused HCI Perspective*

12 Carmen Bratosin (TUE) *Grid Architecture for Distributed Process Mining*

13 Xiaoyu Mao (UVT) *Airport under Control. Multiagent Scheduling for Airport Ground Handling*

14 Milan Lovric (EUR) *Behavioral Finance and Agent-Based Artificial Markets*

15 Marijn Koolen (UVA) *The Meaning of Structure: the Value of Link Evidence for Information Retrieval*

16 Maarten Schadd (UM) *Selective Search in Games of Different Complexity*

17 Jiyin He (UVA) *Exploring Topic Structure: Coherence, Diversity and Relatedness*

18 Mark Ponsen (UM) *Strategic Decision-Making in complex games*

19 Ellen Rusman (OU) *The Mind's Eye on Personal Profiles*

20 Qing Gu (VU) *Guiding service-oriented software engineering - A view-based approach*

21 Linda Terlouw (TUD) *Modularization and Specification of Service-Oriented Systems*

22 Junte Zhang (UVA) *System Evaluation of Archival Description and Access*

23 Wouter Weerkamp (UVA) *Finding People and their Utterances in Social Media*

24 Herwin van Welbergen (UT) *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior*

25 Syed Waqar ul Qounain Jaffry (VU) *Analysis and Validation of Models for Trust Dynamics*

26 Matthijs Aart Pontier (VU) *Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots*

27 Aniel Bhulai (VU) *Dynamic website optimization through autonomous management of design patterns*

28 Rianne Kaptein (UVA) *Effective Focused Retrieval by Exploiting Query Context and Document Structure*

29 Faisal Kamiran (TUE) *Discrimination-aware Classification*

30 Egon van den Broek (UT) *Affective Signal Processing (ASP): Unraveling the mystery of emotions*

31 Ludo Waltman (EUR) *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality*

32 Nees-Jan van Eck (EUR) *Methodological Advances in Bibliometric Mapping of Science*

33 Tom van der Weide (UU) *Arguing to Motivate Decisions*

34 Paolo Turrini (UU) *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations*

35 Maaike Harbers (UU) *Explaining Agent Behavior in Virtual Training*

36 Erik van der Spek (UU) *Experiments in serious game design: a cognitive approach*

37 Adriana Birlutiu (RUN) *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference*

38 Nyree Lemmens (UM) *Bee-inspired Distributed Optimization*

39 Joost Westra (UU) *Organizing Adaptation using Agents in Serious Games*

40 Viktor Clerc (VU) *Architectural Knowledge Management in Global Software Development*

41 Luan Ibraimi (UT) *Cryptographically Enforced Distributed Data Access Control*

42 Michal Sindlar (UU) *Explaining Behavior through Mental State Attribution*

43 Henk van der Schuur (UU) *Process Improvement through Software Operation Knowledge*

44 Boris Reuderink (UT) *Robust Brain-Computer Interfaces*

45 Herman Stehouwer (UVT) *Statistical Language Models for Alternative Sequence Selection*

46 Beibei Hu (TUD) *Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work*

47 Azizi Bin Ab Aziz (VU) *Exploring Computational Models for Intelligent Support of Persons with Depression*

48 Mark Ter Maat (UT) *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent*

49 Andreea Niculescu (UT) *Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality*

2012

1 Terry Kakeeto (UVT) *Relationship Marketing for SMEs in Uganda*

2 Muhammad Umair (VU) *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models*

3 Adam Vanya (VU) *Supporting Architecture Evolution by Mining Software Repositories*

4 Jurriaan Souer (UU) *Development of Content Management System-based Web Applications*

5 Marijn Plomp (UU) *Maturing Interorganisational Information Systems*

6 Wolfgang Reinhardt (OU) *Awareness Support for Knowledge Workers in Research Networks*

7 Rianne van Lambalgen (VU) *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions*

8 Gerben de Vries (UVA) *Kernel Methods for Vessel Trajectories*

9 Ricardo Neisse (UT) *Trust and Privacy Management Support for Context-Aware Service Platforms*

10 David Smits (TUE) *Towards a Generic Distributed Adaptive Hypermedia Environment*

11 J.C.B. Rantham Prabhakara (TUE) *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*

12 Kees van der Sluijs (TUE) *Model Driven Design and Data Integration in Semantic Web Information Systems*

13 Suleman Shahid (UVT) *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions*

14 Evgeny Knutov (TUE) *Generic Adaptation Framework for Unifying Adaptive Web-based Systems*

15 Natalie van der Wal (VU) *Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.*

16 Fiemke Both (VU) *Helping people by understanding them - Ambient Agents supporting task execution and depression treatment*

17 Amal Elgammal (UVT) *Towards a Comprehensive Framework for Business Process Compliance*

18 Eltjo Poort (VU) *Improving Solution Architecting Practices*

19 Helen Schonenberg (TUE) *What's Next? Operational Support for Business Process Execution*

20 Ali Bahramisharif (RUN) *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing*

21 Roberto Cornacchia (TUD) *Querying Sparse Matrices for Information Retrieval*

22 Thijs Vis (UVT) *Intelligence, politie en veiligheidsdienst: verenigbare grootheden?*

23 Christian Muehl (UT) *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction*

24 Laurens van der Werff (UT) *Evaluation of Noisy Transcripts for Spoken Document Retrieval*

25 Silja Eckartz (UT) *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application*

26 Emile de Maat (UVA) *Making Sense of Legal Text*

27 Hayrettin Gurkok (UT) *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games*

28 Nancy Pascall (UVT) *Engendering Technology Empowering Women*

29 Almer Tigelaar (UT) *Peer-to-Peer Information Retrieval*

30 Alina Pommeranz (TUD) *Designing Human-Centered Systems for Reflective Decision Making*

31 Emily Bagarukayo (RUN) *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure*

32 Wietske Visser (TUD) *Qualitative multi-criteria preference representation and reasoning*

33 Rory Sie (OU) *Coalitions in Cooperation Networks (COCOON)*

34 Pavol Jancura (RUN) *Evolutionary analysis in PPI networks and applications*

35 Evert Haasdijk (VU) *Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics*

36 Denis Ssebugwawo (RUN) *Analysis and Evaluation of Collaborative Modeling Processes*

37 Agnes Nakakawa (RUN) *A Collaboration Process for Enterprise Architecture Creation*

38 Selmar Smit (VU) *Parameter Tuning and Scientific Testing in Evolutionary Algorithms*

39 Hassan Fatemi (UT) *Risk-aware design of value and coordination networks*

40 Agus Gunawan (UVT) *Information Access for SMEs in Indonesia*

41 Sebastian Kelle (OU) *Game Design Patterns for Learning*

42 Dominique Verpoorten (OU) *Reflection Amplifiers in self-regulated Learning*

43 (Withdrawn)

44 Anna Tordai (VU) *On Combining Alignment Techniques*

45 Benedikt Kratz (UVT) *A Model and Language for Business-aware Transactions*

46 Simon Carter (UVA) *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation*

47 Manos Tsagkias (UVA) *A Model and Language for Business-aware Transactions*

48 Jorn Bakker (TUE) *Handling Abrupt Changes in Evolving Time-series Data*

49 Michael Kaisers (UM) *Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions*

50 Steven van Kervel (TUD) *Ontologogy driven Enterprise Information Systems Engineering*

51 Jeroen de Jong (TUD) *Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching*

2013

1 Viorel Milea (EUR) *News Analytics for Financial Decision Support*

2 Erietta Liarou (CWI) *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing*

3 Szymon Klarman (VU) *Reasoning with Contexts in Description Logics*

4 Chetan Yadati (TUD) *Coordinating autonomous planning and scheduling*

5 Dulce Pumareja (UT) *Groupware Requirements Evolutions Patterns*

6 Romulo Gonzalves (CWI) *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience*

7 Giel van Lankveld (UVT) *Quantifying Individual Player Differences*

8 Robbert-Jan Merk (VU) *Making Enemies: Cognitive Modeling for Opponent Agents in Fighter Pilot Simulators*

9 Fabio Gori (RUN) *Metagenomic Data Analysis: Computational Methods and Applications*

10 Jeewanie Jayasinghe Arachchige (UVT) *A Unified Modeling Framework for Service Design*

11 Evangelos Pournaras (TUD) *Multi-level Reconfigurable Self-organization in Overlay Services*

12 Maryam Razavian (VU) *Knowledge-driven Migration to Services*

13 Mohammad Zafiri (UT) *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly*

14 Jafar Tanha (UVA) *Ensemble Approaches to Semi-Supervised Learning Learning*

15 Daniel Hennes (UM) *Multiagent Learning - Dynamic Games and Applications*

16 Eric Kok (UU) *Exploring the practical benefits of argumentation in multi-agent deliberation*

17 Koen Kok (VU) *The PowerMatcher: Smart Coordination for the Smart Electricity Grid*

18 Jeroen Janssens (UVT) *Outlier Selection and One-Class Classification*

19 Renze Steenhuisen (TUD) *Coordinated Multi-Agent Planning and Scheduling*

20 Katja Hofmann (UVA) *Fast and Reliable Online Learning to Rank for Information Retrieval*

21 Sander Wubben (UVT) *Text-to-text generation by monolingual machine translation*

22 Tom Claassen (RUN) *Causal Discovery and Logic*

23 Patricio de Alencar Silva (UVT) *Value Activity Monitoring*

24 Haitham Bou Ammar (UM) *Automated Transfer in Reinforcement Learning*

25 Agnieszka Anna Latoszek-Berendsen (UM) *Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System*

26 Alireza Zarghami (UT) *Architectural Support for Dynamic Homecare Service Provisioning*

27 Mohammad Huq (UT) *Inference-based Framework Managing Data Provenance*

28 Frans van der Sluis (UT) *When Complexity becomes Interesting: An Inquiry into the Information eXperience*

29 Iwan de Kok (UT) *Listening Heads*

30 Joyce Nakatumba (TUE) *Resource-Aware Business Process Management: Analysis and Support*

31 Dinh Khoa Nguyen (UVT) *Blueprint Model and Language for Engineering Cloud Applications*

32 Kamakshi Rajagopal (OU) *Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development*

33 Qi Gao (TUD) *User Modeling and Personalization in the Microblogging Sphere*

34 Kien Tjin-Kam-Jet (UT) *Distributed Deep Web Search*

35 Abdallah El Ali (UVA) *Minimal Mobile Human Computer Interaction*

36 Than Lam Hoang (TUe) *Pattern Mining in Data Streams*

37 Dirk Börner (OU) *Ambient Learning Displays*

38 Eelco den Heijer (VU) *Autonomous Evolutionary Art*

39 Joop de Jong (TUD) *A Method for Enterprise Ontology based Design of Enterprise Information Systems*

40 Pim Nijssen (UM) *Monte-Carlo Tree Search for Multi-Player Games*

41 Jochem Liem (UVA) *Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning*

42 Léon Planken (TUD) *Algorithms for Simple Temporal Reasoning*

43 Marc Bron (UVA) *Exploration and Contextualization through Interaction and Concepts*

2014

1 Nicola Barile (UU) *Studies in Learning Monotone Models from Data*

2 Fiona Tuliyano (RUN) *Combining System Dynamics with a Domain Modeling Method*

3 Sergio Raul Duarte Torres (UT) *Information Retrieval for Children: Search Behavior and Solutions*

4 Hanna Jochmann-Mannak (UT) *Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation*

5 Jurriaan van Reijsen (UU) *Knowledge Perspectives on Advancing Dynamic Capability*

6 Damian Tamburri (VU) *Supporting Networked Software Development*

7 Arya Adriansyah (TUE) *Aligning Observed and Modeled Behavior*

8 Samur Araujo (TUD) *Data Integration over Distributed and Heterogeneous Data Endpoints*

9 Philip Jackson (UVT) *Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language*

10 Ivan Salvador Razo Zapata (VU) *Service Value Networks*

11 Janneke van der Zwaan (TUD) *An Empathic Virtual Buddy for Social Support*

12 Willem van Willigen (VU) *Look Ma, No Hands: Aspects of Autonomous Vehicle Control*

13 Arlette van Wissen (VU) *Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains*

14 Yangyang Shi (TUD) *Language Models With Meta-information*

15 Natalya Mogles (VU) *Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare*

16 Krystyna Milian (VU) *Supporting trial recruitment and design by automatically interpreting eligibility criteria*

17 Kathrin Dentler (VU) *Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability*

18 Mattijs Ghijsen (VU) *Methods and Models for the Design and Study of Dynamic Agent Organizations*

19 Vincius Ramos (TUE) *Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support*

20 Mena Habib (UT) *Named Entity Extraction and Disambiguation for Informal Text: The Missing Link*

21 Kassidy Clark (TUD) *Negotiation and Monitoring in Open Environments*

22 Marieke Peeters (UU) *Personalized Educational Games - Developing agent-supported scenario-based training*

23 Eleftherios Sidirourgos (UVA/CWI) *Space Efficient Indexes for the Big Data Era*

24 Davide Ceolin (VU) *Trusting Semi-structured Web Data*

25 Martijn Lappenschaar (RUN) *New network models for the analysis of disease interaction*

26 Tim Baarslag (TUD) *What to Bid and When to Stop*

27 Rui Jorge Almeida (EUR) *Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty*

28 Anna Chmielowiec (VU) *Decentralized k-Clique Matching*

29 Jaap Kabbedijk (UU) *Variability in Multi-Tenant Enterprise Software*

30 Peter de Cock (UVT) *Anticipating Criminal Behaviour*

31 Leo van Moergestel (UU) *Agent Technology in Agile Multiparallel Manufacturing and Product Support*

32 Naser Ayat (UVA) *On Entity Resolution in Probabilistic Data*

33 Tesfa Tegegne (RUN) *Service Discovery in eHealth*

34 Christina Manteli (VU) *The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.*

35 Joost van Ooijen (UU) *Cognitive Agents in Virtual Worlds: A Middleware Design Approach*

36 Joos Buijs (TUE) *Flexible Evolutionary Algorithms for Mining Structured Process Models*

37 Maral Dadvar (UT) *Experts and Machines United Against Cyberbullying*

38 Danny Plass-Oude Bos (UT) *Making brain-computer interfaces better: improving usability through post-processing.*

39 Jasmina Maric (UVT) *Web Communities, Immigration, and Social Capital*

40 Walter Omona (RUN) *A Framework for Knowledge Management Using ICT in Higher Education*

41 Frederik Hogenboom (EUR) *Automated Detection of Financial Events in News Text*

42 Carsten Eijckhof (CWI/TUD) *Contextual Multidimensional Relevance Models*

43 Kevin Vlaanderen (UU) *Supporting Process Improvement using Method Increments*

44 Paulien Meesters (UVT) *Intelligent Blauw. Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.*

45 Birgit Schmitz (OU) *Mobile Games for Learning: A Pattern-Based Approach*

46 Ke Tao (TUD) *Social Web Data Analytics: Relevance, Redundancy, Diversity*

47 Shangsong Liang (UVA) *Fusion and Diversification in Information Retrieval*

2015

1 Niels Netten (UVA) *Machine Learning for Relevance of Information in Crisis Response*

2 Faiza Bukhsh (UVT) *Smart auditing: Innovative Compliance Checking in Customs Controls*

3 Twan van Laarhoven (RUN) *Machine learning for network data*

4 Howard Spoelstra (OU) *Collaborations in Open Learning Environments*

5 Christoph Bösch (UT) *Cryptographically Enforced Search Pattern Hiding*

6 Farideh Heidari (TUD) *Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes*

7 Maria-Hendrike Peetz (UVA) *Time-Aware Online Reputation Analysis*

8 Jie Jiang (TUD) *Organizational Compliance: An agent-based model for designing and evaluating organizational interactions*

9 Randy Klaassen (UT) *HCI Perspectives on Behavior Change Support Systems*

10 Henry Hermans (OU) *OpenU: design of an integrated system to support lifelong learning*

11 Yongming Luo (TUE) *Designing algorithms for big graph datasets: A study of computing bisimulation and joins*

12 Julie M. Birkholz (VU) *Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks*

13 Giuseppe Procaccianti (VU) *Energy-Efficient Software*

14 Bart van Straalen (UT) *A cognitive approach to modeling bad news conversations*

15 Klaas Andries de Graaf (VU) *Ontology-based Software Architecture Documentation*

16 Changyun Wei (UT) *Cognitive Coordination for Cooperative Multi-Robot Teamwork*

17 André van Cleeff (UT) *Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs*

18 Holger Pirk (CWI) *Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories*

19 Bernardo Tabuenca (OU) *Ubiquitous Technology for Lifelong Learners*

20 Lois Vanhée (UU) *Using Culture and Values to Support Flexible Coordination*

21 Sibren Fetter (OU) *Using Peer-Support to Expand and Stabilize Online Learning*

22 Zhemin Zhu (UT) *Co-occurrence Rate Networks*

23 Luit Gazendam (VU) *Cataloguer Support in Cultural Heritage*

24 Richard Berendsen (UVA) *Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation*

25 Steven Woudenberg (UU) *Bayesian Tools for Early Disease Detection*

26 Alexander Hogenboom (EUR) *Sentiment Analysis of Text Guided by Semantics and Structure*

27 Sándor Héman (CWI) *Updating compressed column-stores*

28 Janet Bagorogoza (UVT) *Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO*

29 Hendrik Baier (UM) *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*

ERRATA

belonging to the dissertation
Monte-Carlo Tree Search Enhancements
for One-Player and Two-Player Domains
by Hendrik Baier

page	reads	should read
15	shows the first two **ply**	shows the first two **plies**
27	**winner** of the simulated game	**outcome** of the simulated game
35	$\mu^*\mathbf{n} - \sum_{a=1}^{a_{\max}} \mu_a E[n_a(t)]$	$\mu^*\mathbf{t} - \sum_{a=1}^{a_{\max}} \mu_a E[n_a(t)]$
35	$\mu^* = \max_{1 \le i \le \mathbf{A}} \mu_i$	$\mu^* = \max_{1 \le i \le \mathbf{a_{max}}} \mu_i$
37	able to draw samples from the **transition function**	able to draw samples from the **transition and reward functions**
65–66	between 0 and **0.1**	between 0 and **0.5**
112	$\frac{\frac{o^m_{\text{winner}}(i)}{n} - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})}{1 - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})}$	$\frac{\frac{o^m_{\text{winner}}(i)}{n} - (\frac{o^m_{\text{white}}(i) w^m_{\text{white}} + o^m_{\text{black}}(i) w^m_{\text{black}}}{n^2})}{1 - (\frac{o^m_{\text{white}}(i) w^m_{\text{white}} + o^m_{\text{black}}(i) w^m_{\text{black}}}{n^2})}$
137	see Figures 6.7, 6.8, and for a visualization	see Figures 6.7, 6.8, and **6.9** for a visualization
159	the empty Connect-4 position	the initial Breakthrough board
184	at the newly expanded node of the tree	where the search leaves the tree
236	multi-start BMCTS **was better than** multi-start MCTS at a wider range of time settings than single-start BMCTS **to** single-start MCTS	multi-start BMCTS **improved on** multi-start MCTS at a wider range of time settings than single-start BMCTS **on** single-start MCTS