Monte-Carlo Tree Search for Multi-Player Games

Monte-Carlo Tree Search for Multi-Player Games

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. dr. L.L.G. Soete,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op maandag 2 december 2013 om 14.00 uur

door

Joseph Antonius Maria Nijssen

This research has been funded by the transnational University Limburg (tUL).

Dissertation Series No. 2013-40
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Optima Grafische Communicatie, Rotterdam, The Netherlands
ISBN 978-94-6169-452-2

Preface

Two topics that interested me the most since I was a child were computers and board games. When I found out, during my bachelor's study Knowledge Engineering at the Maastricht University, that there exists a research area that combines these two topics, I immediately knew in which direction I would want to do my bachelor's thesis. Letting computers play board games as strongly as possible, developing or investigating new techniques or heuristics along the way, certainly caught my attention. After achieving my bachelor's degree, I chose to continue with the master Artificial Intelligence. For my master's thesis, I decided to stay in the same research direction. After applying Monte-Carlo based techniques to the game of Othello, I decided to investigate search techniques for the two-player chess-like board game Khet. While working on my master's thesis, I got the idea to do Ph.D. research in the area of games and AI. Luckily, a new Ph.D. position in the Games and AI group opened a few months after I finished my master's thesis. This Ph.D. thesis presents the research that I have performed in the past four years at the Department of Knowledge Engineering at the Maastricht University. Performing research and writing a thesis, however, is not a solitary effort. Therefore, there are various people I would like to thank.

First of all I would like to thank my daily supervisor, Mark Winands. He taught me a lot about doing research and writing articles. Thanks to his support, this thesis was made possible. The second person I would like to thank is Jos Uiterwijk. He supervised my bachelor's and my master's theses, and it was thanks to his efforts that I was able to get my position as a Ph.D. student. Next, I would like to thank Ralf Peeters, for agreeing to be my promotor, and for looking at my thesis from a different point of view.

Of course, my thanks also go to the people with whom I have shared an office during my time as a Ph.D. student. Maarten Schadd, Jahn Saito, David Lupien St-Pierre, Hendrik Baier, and Mandy Tak, thank you for the interesting discussions we had and for making the 'games office' a great place to work. In particular, I want to thank Marc Lanctot for proofreading my thesis, and raising some interesting points that certainly improved its quality. I would also like to thank all my other colleagues at the Department of Knowledge Engineering. I want to thank Guillaume Chaslot, Steven de Jong, Nyree Lemmens, Marc Ponsen, Philippe Uyttendaele, Frederik Schadd, Daan Bloembergen, Michael Kaisers, Pietro Bonizzi, Nela Lekic, and many others, with whom I had the pleasure to have lunch breaks, including both the interesting and slightly less interesting discussions.

And last but not least, my thanks go to my parents Harry and Josine, and my sisters Iris and Yonne. Without their love and support, all of this would never have been possible. Bedaank veur alles.

Pim Nijssen, 2013

Acknowledgments

The research 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 transnational University Limburg (tUL).

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

The topic of this thesis lies in the area of adversarial search in multi-player zero-sum domains, i.e., search in domains having players with conflicting goals. In order to focus on the issues of searching in this type of domains, we shift our attention to abstract games. These games provide a good test domain for Artificial Intelligence (AI). They offer a pure abstract competition (i.e., comparison), with an exact closed domain (i.e., well-defined rules). The games under investigation have the following two properties. (1) They are too complex to be solved with current means, and (2) the games have characteristics that can be formalized in computer programs. AI research has been quite successful in the field of two-player zero-sum games, such as chess, checkers, and Go. This has been achieved by developing two-player search techniques. However, many games do not belong to the area where these search techniques are unconditionally applicable. Multi-player games are an example of such domains. This thesis focuses on two different categories of multi-player games: (1) deterministic multi-player games with perfect information and (2) multi-player hide-and-seek games. In particular, it investigates how Monte-Carlo Tree Search can be improved for games in these two categories. This technique has achieved impressive results in computer Go, but has also shown to be beneficial in a range of other domains.

This chapter is structured as follows. First, an introduction to games and the role they play in the field of AI is provided in Section 1.1. An overview of different game properties is given in Section 1.2. Next, Section 1.3 defines the notion of multi-player games and discusses the two different categories of multi-player games that are investigated in this thesis. A brief introduction to search techniques for two-player and multi-player games is provided in Section 1.4. Subsequently, Section 1.5 defines the problem statement and four research questions. Finally, an overview of this thesis is provided in Section 1.6.

1.1 Games and AI

Games have been played by humans since the dawn of civilization. The first board games date back more than 5000 years. One of the oldest discovered board games is *Senet*, which was played by the ancient Egyptians around 3500 BC. It was played on a board consisting of 30 squares arranged into three rows of ten. Two contesting players strategically moved their team of draughtsmen through these squares. The exact rules of this game have long been forgotten, but it is known that this game was played for approximately 3000 years (Piccione, 1980).

Some of these ancient games are still played today. The well-known board game *Go* originated in China, which is played there for around 4000 years. The game used to be only popular in East Asia, but in the recent years, it has also become more popular in the rest of the world. Over the centuries, the number of abstract games has grown tremendously. Nowadays, thousands of different games are played every day, varying from classic board games such as chess or checkers to modern board games such as Settlers of Catan and Carcassonne.

Since the invention of computers, abstract games have become an important research area in AI. The reason for this is that games are well-defined and structured, but often still considerably complex. There exist three different directions in AI research in games. The first direction is solving games. In general, a game is solved if the game-theoretic value (e.g., win, loss, or draw for the first player) is known, assuming that all players play optimally. Some games, such as checkers (Schaeffer *et al.*, 2007), Connect Four (Allis, 1988; Tromp, 2008), and Go-Moku (Allis, Huntjes, and Van den Herik, 1996) have been solved. For a substantial number of games, such as chess or Go, this is currently infeasible. For instance, a game of chess can be played in around 10^{123} ways and there are approximately 10^{47} different positions a game of chess can be in (Shannon, 1950). Intelligent search techniques have to be developed for playing a game as strong as possible. This is the second, and quite popular, research direction in games. While humans rely much on experience and intuition, computers generally use brute-force computing power to find the best move. The research described in this thesis belongs to this direction. The third research direction concerns developing programs for providing entertainment for human players, rather than trying to defeat them. This direction mostly investigates the domain of video games. Two major challenges are simulating human-like behavior and difficulty scaling. Simulating human-like behavior concerns the development of an agent that is as indistinguishable as possible from a human player (Togelius *et al.*, 2011). Difficulty scaling is the automatic adaptation of an agent or the game itself to the level of a human player (Spronck, Sprinkhuizen-Kuyper, and Postma, 2004; Shaker *et al.*, 2011).

Regarding the second research direction, abstract two-player games have been studied in AI since the 1950s. Shannon (1950) and Turing (1953) described how computers can be used for playing chess. Over the decades that followed, game-playing programs became progressively stronger. This was not only due to better hardware, but also because of the development of new search techniques. A major milestone in the field of AI research in games was in 1997, when chess grandmaster Garry Kasparov was beaten by IBM's DEEP BLUE (Hsu, 2002). This machine managed to defeat Kasparov by 3½–2½. This was the first time a human world champion was beaten by a computer in a game of chess. Since then, research shifted to other games, most notably Go. Until recently, the best Go programs could only play at a weak amateur level (Müller, 2002). Due to the introduction of a new search technique, Monte-Carlo Tree Search, much progress has been made in this game over the past decade (Lee *et al.*, 2009).

Besides research in two-player domains, multi-player games have gained some popularity as well since the introduction of the first search technique for multi-player games by Luckhardt and Irani (1986). By modifying existing search techniques to accommodate for domains with more than two players, computer programs were developed to play multi-player games such as Sergeant Major (Sturtevant and Korf, 2000), Chinese Checkers (Sturtevant, 2003a; Sturtevant, 2008a; Schadd and Winands, 2011), four-player chess (Lorenz and Tscheuschner, 2006), and multi-player Go (Cazenave, 2008).

1.2 Game Properties

Games can be classified using several properties. This classification of games is important, because games with different properties may require different AI techniques. Below, seven important properties are given.

(1) Number of players. The first distinguishing property is the number of players. Games can be played by one, two, or more players. Examples of one-player games, also called *optimization problems* or *puzzles*, are Solitaire and the 15-puzzle. Well-known examples of two-player games are chess and Go. Games that can be played by more than two players are called *multi-player games*. In multi-player games, we can distinguish two subcategories: cooperative and non-cooperative. In a *cooperative* game, two or more players cooperate in order to achieve a common goal. If this goal is fulfilled, all cooperating players win the game. An example is the hide-and-seek game Scotland Yard. In a *non-cooperative* game, all players play for themselves and they all have conflicting goals. Examples of this category of games are Chinese Checkers and Hearts. This thesis investigates both cooperative and non-cooperative multi-player games.

(2) Information. In many games, all information about a position is available throughout the whole game for all players. These games are called *perfect-information games*. If at least one of the players does not have all information about a position at any point in the game, it is a game with *imperfect information*. This is, for instance, the case in most card games, such as poker. This thesis investigates four games with perfect information and one game with imperfect information.

(3) Chance. If in a game events occur on which no player has any direct influence on the outcome, such as the roll of a die, the game is called a *non-deterministic game*. If chance events do not occur and each action leads to a predictable new position, the game is called *deterministic*. The games in this thesis generally are deterministic, though one game (see Chapter 7) has a minor element of chance.

(4) Decision space. In most abstract games, players have a *discrete* (i.e., countable) set of *moves* they can make. This is also the case with all games investigated in this thesis. Video games, however, usually simulate a *continuous* decision space. The number of possible moves is arbitrarily large. Often, the decision space is discretized to allow an agent to easier make decisions.

(5) Game flow. Most abstract games are *turn-based*. In these games, players can only move at a certain point in time. There exist two different types of turn-based games. In *sequential move games*[1] the players take turns, one at a time. This is the most common game flow in abstract games. In *simultaneous move games*, more than one player takes a turn at a time. Examples of games with simultaneous moves are modern board games such as 7 Wonders, El Grande, and Diplomacy. In AI research, a discretized version of the game Tron is sometimes used as a test domain for abstract games with simultaneous moves. A different game flow system is often used in video games, namely *real-time*. All players can move at any time and there is no such notion as turns. This thesis only considers turn-based games with sequential moves.

(6) Theme. Abstract games have different types of goals for the players. Depending on the goal, games can be classified into different themes. Examples of themes include *connection* games such as Hex and Havannah, *territory* games such as Go and Othello, *race* games such as Chinese Checkers, *capture* games such as chess and Checkers, and *tile-based* games such as dominoes. This theses investigates games that belong to a disparate range of themes.

(7) Symmetry. In the majority of the abstract games, the players have a similar goal. For instance, in chess, the goal for both players is to checkmate the opponent's king. These games are called *symmetric*. If the players have different goals, the game is called *asymmetric*. Examples are the ancient family of Tafl games and the similarly themed Breakthru. In the latter game, one player has to escape from the middle of the board to the edge with a flag ship, while the other player should try to prevent this by capturing it. In this thesis, both symmetric and asymmetric games are investigated.

1.3 Multi-Player Games

Much research in the field of games has been performed in the domain of deterministic two-player games with perfect information. For a long time, chess has been the prime domain for researchers. After the defeat of Kasparov in 1997, the focus shifted to Go, which was at that time a domain in which computer programs could only play on a weak amateur level. Still today, Go remains one of the most popular domains for computer games research.

In this thesis, we shift focus from the well-trodden domain of two-player games to the relatively unknown domain of multi-player games (Sturtevant, 2003b). In this thesis we focus on two different types of multi-player games. First, Subsection 1.3.1 discusses deterministic multi-player games with perfect information. Hide-and-seek games are described in Subsection 1.3.2.

1.3.1 Perfect-Information Multi-Player Games

The first type of multi-player games investigated in this thesis are deterministic, non-cooperative multi-player games with perfect information. These games do not involve

[1]Alternatively called sequential games or turn-taking games.

an element of chance, and each player knows the exact game state at all times.

In AI, most research has been performed in the area of deterministic perfect information games. Two of the most investigated games in AI, chess and Go, both have these properties. In multi-player games, initial research was performed in games without chance or hidden information as well (cf. Luckhardt and Irani, 1986; Korf, 1991; Sturtevant and Korf, 2000).

One of the difficulties of this type of multi-player games, compared to two-player games, is that there are more complex dynamics between players. In two-player zero-sum games, it is usually the case that if a player improves his own position, he diminishes the opponent's position. Therefore, the players always play against each other. In multi-player games, this is not necessarily the case. In non-cooperative games, temporary *coalitions* may occur (Schadd, 2011). For instance, if one player is far ahead of the other players, they may form a coalition to catch up with the leading player or even diminish the leading player's position. This coalition often only lasts as long as this player is ahead. Coalitions may be formed and broken as the game progresses. Because of these complex dynamics, opponents are less predictable and assumptions have to be made about the opponents, for instance whether they will try to increase their own position or decrease one of the opponents' positions.

Another difficulty in non-cooperative multi-player games is the *kingmaker* phenomenon. In some games it is possible that a player, who cannot win anymore, can determine the outcome of the game. Kingmakers are highly unpredictable, because it is usually unknown which player they will let win.

In this thesis, we analyze the performance of various search techniques and enhancements in four different deterministic multi-player games with perfect information: Chinese Checkers, Focus, Rolit, and Blokus. More information about these games is provided in Chapter 3.

1.3.2 Multi-Player Hide-and-Seek Games

The second type of multi-player games investigated in this thesis are hide-and-seek games. They are played on a graph consisting of vertices, also called locations, that are connected to each other via edges, also called paths. The players can travel between locations using these paths. The goal for the seekers is to locate and capture one or more hiders. These hiders can be mobile or immobile. Players can act as the hider and the seeker at the same time, such as in two-player games like Battleship or Stratego, or players take on either of these roles. The seekers do not know the location of the hider(s). This means that hide-and-seek games are games with imperfect information. In general, hide-and-seek games are deterministic games, though the starting locations of the players may be random.

Most research in the area of hide-and-seek games concerns finding optimal strategies for the seekers, such that they are guaranteed to find the hider(s) in a limited amount of time. A research domain is for instance pursuit-evasion games (Parsons, 1978; Megiddo *et al.*, 1988; Adler *et al.*, 2003), in which the hiders are mobile, and may be visible, partially visible, or invisible to the seekers. In this thesis, we focus on search techniques that make the hider(s) and the seekers play as strongly as possible in a game that is too large to solve.

We are interested in hide-and-seek games with the following three aspects. (1) They have partially imperfect information. The hider knows the exact game state at all times, but the seekers do not. Based on a limited amount of information, the seekers have to deduce where the hider can be located at any point in the game. (2) Contrary to many games, the hide-and-seek games are asymmetric. The hider has a different goal than the seekers. (3) The hide-and-seek games feature a fixed coalition between the seekers. If one of the seekers captures the hider, then all seekers have won the game. This encourages collaboration between the seekers. The challenge is to make the seekers cooperate in an effective way.

As a test domain, we consider the multi-player pursuit-evasion hide-and-seek game Scotland Yard. More information about this game is given in Chapter 7.

1.4 Search Techniques

The most successful AI method in abstract games is search. Over the past decades, many search techniques have been developed for playing various kinds of abstract games using a computer. For two-player deterministic sequential games, *minimax* (Von Neumann and Morgenstern, 1944) is the foundation for most search techniques. The most popular two-player search technique is $\alpha\beta$ search (Knuth and Moore, 1975). For two-player games with chance, a generalization of minimax, called expectimax (Michie, 1966), may be used.

The first minimax-based technique developed for multi-player games is *max*^{n} (Luckhardt and Irani, 1986). Over the years, different minimax-based search techniques have been developed for multi-player games, such as *paranoid* search (Sturtevant and Korf, 2000) and *Best-Reply Search* (BRS) (Schadd and Winands, 2011). These search techniques all have a different assumption about the opponents. Max^n assumes that each player only tries to increase his own position, without looking at the opponents. With paranoid, an assumption is made that all opponents have formed a coalition. BRS assumes that only one of the opponents plays a counter move, while all other opponents pass, even if this is not allowed in the game.

All aforementioned search techniques require a heuristic evaluation function. These evaluation functions require game-specific knowledge to assign heuristic values to the leaf nodes in the search tree. However, in some games, such as (multi-player) Go, developing such an evaluation function is quite difficult. This is one of the reasons why minimax-based search techniques do not perform well in this game. An alternative way to evaluate leaf nodes is by applying Monte-Carlo evaluations (Abramson, 1990; Bouzy and Helmstetter, 2004). This provided the basis of Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007a). It is a best-first search technique that is guided by Monte-Carlo simulations. The MCTS algorithm consists of four phases: selection, expansion, playout, and backpropagation. These four phases are performed iteratively.

MCTS gained much popularity after it significantly increased the playing strength of the state-of-the-art programs in computer Go. It has become a successful technique in disparate domains, such as General Game Playing (Björnsson and Finnsson, 2009), Hex (Arneson, Hayward, and Henderson, 2010; Cazenave and Saffidine, 2010), and Ms. PacMan (Pepels and Winands, 2012). Also in the multi-player game Chinese Checkers, MCTS has outperformed minimax-based approaches (Sturtevant, 2008b). Furthermore, MCTS has been successfully applied in optimization problems, such as Production Management Problems (Chaslot *et al.*, 2006a), Library Performance Tuning (De Mesmay *et al.*, 2009), navigation problems (Müller *et al.*, 2012), and the Physical Traveling Salesman Problem (Perez, Rohlfshagen, and Lucas, 2012; Powley, Whitehouse, and Cowling, 2012). An in-depth discussion of existing minimax-based and Monte-Carlo based search techniques and enhancements is provided in Chapter 2.

1.5 Problem Statement and Research Questions

In the previous sections, we discussed the relevance of games in the field of AI and the properties and difficulties of multi-player games. This thesis focuses on the application and improvement of MCTS in multi-player games. The following problem statement will guide the research.

> **Problem statement**: *How can Monte-Carlo Tree Search be improved to increase the performance in multi-player games?*

In order to answer the problem statement, four research questions have been formulated. They deal with (1) incorporating different search policies in MCTS, (2) improving the selection phase of MCTS, (3) improving the playout phase of MCTS, and (4) adapting MCTS to a hide-and-seek game.

> **Research question 1**: *How can multi-player search policies be incorporated in MCTS?*

The advantage of MCTS is that it can be extended to multi-player games. In the standard multi-player variant of MCTS, each player is concerned with maximizing his own win rate. In this sense, this variant is comparable to the minimax-based multi-player search technique max^n (Luckhardt and Irani, 1986), where each player tries to maximize his own score, regardless of the scores of the other players. Besides max^n, there also exist other minimax-based multi-player search techniques, such as paranoid (Sturtevant and Korf, 2000) and Best-Reply Search (BRS) (Schadd and Winands, 2011).

To answer the first research question, we investigate how three minimax-based multi-player search techniques, namely max^n, paranoid, and BRS, can be embedded in the MCTS framework. These search techniques are integrated as search policies. A search policy determines how nodes are chosen during the selection phase of MCTS and how the results of the playouts are backpropagated in the tree. The performance of these search policies is tested in four different deterministic multi-player games with perfect information.

The aforementioned search policies are not able to prove game-theoretic values in the search tree. Therefore, a multi-player variant of the MCTS-Solver is introduced.

This technique is combined with the max^n search policy. Three different update rules are proposed for solving nodes in a multi-player MCTS tree. The performance of multi-player MCTS-Solver with these update rules is investigated in the multi-player game Focus.

> **Research question 2**: *How can the selection phase of MCTS be enhanced in perfect-information multi-player games?*

An important phase in the MCTS algorithm is the selection phase. During the selection phase, the search tree is traversed until a leaf node is reached. A selection strategy determines how the tree is traversed. Over the past years, several selection strategies and enhancements have been developed for different types of games. The most popular selection strategy is Upper Confidence bounds applied to Trees (UCT) (Kocsis and Szepesvári, 2006). This is an extension of the Upper Confidence Bounds (UCB1) (Auer, Cesa-Bianchi, and Fischer, 2002) algorithm that is used in bandit problems. There exist various enhancements for the UCT selection strategy (Browne *et al.*, 2012). Most of them are domain dependent, which means that they cannot unconditionally be applied in every domain. Examples of domain-dependent enhancements include prior knowledge (Gelly and Silver, 2007), Progressive Widening, and Progressive Bias (Chaslot *et al.*, 2008b). A domain-independent enhancement is Rapid Action Value Estimation (RAVE) (Gelly and Silver, 2007), which is in particular successful in the field of computer Go, but less successful in others, such as the multi-player game Chinese Checkers (Sturtevant, 2008a; Finnsson, 2012).

To answer the second research question, a new domain-independent selection strategy based on UCT is proposed, namely Progressive History. This is a combination of the relative history heuristic (Winands *et al.*, 2006) and Progressive Bias (Chaslot *et al.*, 2008b). Several variations of this technique are tested in four different deterministic perfect-information games.

> **Research question 3**: *How can the playouts of MCTS be enhanced in perfect-information multi-player games?*

Similar to the selection phase, the playout phase is an important phase in the MCTS algorithm. During the playout phase, the game is finished by playing moves that are selected using a playout strategy. More realistic playouts usually provide more reliable results, thus increasing the playing strength of an MCTS-based player. Playouts can be made more realistic by adding domain knowledge (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008). The disadvantage is that this may reduce the number of playouts per second, decreasing the playing strength (Chaslot, 2010). The challenge is to find a good balance between speed and quality of the playouts.

Winands and Björnsson (2011) proposed to apply relatively time-expensive two-ply $\alpha\beta$ searches in the playout phase of MCTS. While this significantly reduced the number of playouts per second, it increased the overall playing strength by improving the quality of the playouts.

To answer the third research question, two-ply max^n, two-ply paranoid, and two-ply BRS searches in the playouts are investigated and their performance is analyzed in two deterministic, perfect-information multi-player games.

Research question 4: *How can MCTS be adapted for hide-and-seek games?*

As described in Subsection 1.3.2, the hide-and-seek games of interest have three properties that make them a challenging domain for MCTS. These properties are (1) imperfect information for the seekers, (2) asymmetry in the goals of the players, and (3) cooperation between the seekers. To answer the fourth research question, we investigate techniques for tackling these intricacies.

As the test domain we choose the game Scotland Yard. This hide-and-seek game features the three aforementioned properties and is currently too complex for computers to solve. To handle the imperfect information in Scotland Yard, two different determinization techniques are investigated, namely single-tree determinization and separate-tree determinization. Also, a new technique to bias the determinization towards more likely positions, called Location Categorization, is introduced. The asymmetric nature of Scotland Yard requires implementing different domain knowledge for the hider and the seekers. This domain knowledge is required for ϵ-greedy playouts. Furthermore, a technique called Coalition Reduction is introduced to handle the cooperation between the seekers. This technique balances each seeker's participation in the coalition.

1.6 Thesis Overview

This thesis is organized into eight chapters. Chapter 1 provides a general introduction to games in AI and the different types of multi-player games investigated in this thesis. Furthermore, the problem statement and four research questions that guide our research are formulated.

Chapter 2 introduces the search techniques and standard enhancements. It discusses minimax-based techniques such as $\alpha\beta$ search and the multi-player techniques max^n, paranoid, and Best-Reply Search (BRS). Furthermore, an introduction to Monte-Carlo techniques, in particular Monte-Carlo Tree Search (MCTS) is given.

Chapter 3 explains the four test domains used in Chapters 4, 5, and 6: Chinese Checkers, Focus, Rolit, and Blokus. We provide for each of these games the rules, a complexity analysis, and an overview of the heuristic domain knowledge that is applied in the different search techniques. Furthermore, this chapter introduces the program MAGE, which is used as the test engine for the experiments in Chapters 4, 5, and 6. MAGE was developed during this Ph.D. research.

Chapter 4 answers the first research question. We investigate how the max^n, paranoid, and BRS search policies perform in the MCTS framework in Chinese Checkers, Focus, Rolit, and Blokus with different numbers of players and different time settings. This is done by matching the search policies against each other and against minimax-based search techniques. Furthermore, we introduce a multi-player variant of MCTS-Solver for proving game-theoretic values. This policy is tested in the sudden-death game Focus.

The second research question is answered in Chapter 5. In this chapter, we propose a domain-independent enhancement to improve the selection strategy of MCTS, namely Progressive History. This enhancement is compared against the standard

UCT selection strategy to investigate the performance increase in MCTS. Furthermore, three variations on Progressive History are tested. The experiments are performed in the two-player and multi-player versions of Chinese Checkers, Focus, Rolit, and Blokus.

Chapter 6 answers the third research question. In this chapter we compare different playout strategies to determine the tradeoff between search and speed in the playout phase of MCTS. We introduce two-ply max^n, paranoid, and BRS playouts, and compare them to random, greedy, and one-ply playouts in the three-player and four-player variants of Chinese Checkers and Focus. These experiments are performed with different time settings to investigate the influence of the amount of thinking time on the performance of the different playout strategies.

The fourth and final research question is answered in Chapter 7. In this chapter we investigate the application and enhancement of MCTS in the hide-and-seek game Scotland Yard. The chapter starts with an introduction to the game of Scotland Yard. Next, we explain how MCTS can be applied to play Scotland Yard. Two different determinization techniques are defined: single tree and separate tree. Determinization is improved by using Location Categorization. Furthermore, we introduce Coalition Reduction, which is a technique to balance the fixed coalition of the seekers. Subsequently, we explain how paranoid search and expectimax can be used for the hider and the seekers, respectively. Also, Location Categorization is introduced as an enhancement of expectimax. All enhancements are systematically tested to investigate their influence on the performance of MCTS-based and minimax-based players.

Finally, Chapter 8 answers the four research questions and addresses the problem statement. Also, we provide an outlook on possible future research topics.

Additionally, Appendix A summarizes different variants of the RAVE formula. Appendix B provides detailed results of the experiments performed in Chapter 4.

CHAPTER 2

Search Techniques

This chapter contains excerpts from the following publications:

1. Nijssen, J.A.M. and Winands, M.H.M. (2012a). An Overview of Search Techniques in Multi-Player Games. *Computer Games Workshop at ECAI 2012*, pp. 50–61, Montpellier, France.
2. Nijssen, J.A.M. and Winands, M.H.M. (2012b). Monte-Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 4, pp. 282–294.

This chapter describes search techniques and commonly used enhancements for two-player and multi-player sequential games. In this thesis we focus on two different families of search methods: minimax-based search techniques and Monte-Carlo based search techniques. These techniques form the basis for the chapters that follow.

This chapter is organized as follows. First, Section 2.1 explains the basics of search in games and defines terms and concepts that are used throughout this thesis. Section 2.2 describes the minimax techniques for searching in trees for two-player games. Next, Section 2.3 discusses three different minimax-based techniques for searching in trees for multi-player games. Subsequently, Section 2.4 gives an overview of common enhancements for minimax-based search techniques, namely dynamic move ordering, transposition tables and iterative deepening. Section 2.5 provides an introduction to Monte-Carlo search. Section 2.6 describes Monte-Carlo Tree Search, which is a search technique that can be applied to both two-player games and multi-player games. Section 2.7 discusses related work in Monte-Carlo Tree Search.

2.1 Tree Search

In order to play any kind of abstract game, computer programs need a representation of the current game position and the outcomes for all possible sequences of moves of the players the game can have. The standard representation is a *tree*. In Subsection 2.1.1 we define the principle of game trees and the terms belonging to this concept. In Subsection 2.1.2 we describe the concept of search trees.

2.1.1 The Game Tree

A *game tree* is a representation of the state space of a sequential game. The *nodes* in the tree represent the positions in the game and the *edges* represent the possible moves from a position. The *root* is the node representing the initial position. The immediate successors of a node are called the node's *children*. They represent the positions that can be reached after performing one legal move. A node's predecessor is called its *parent*. This represents the position before the last move. Nodes sharing the same parent are called *siblings*. A node can have zero, one, or multiple children. Each node has only one parent, except for the root node, which does not have a parent. If a node m is on the path between the root and a node n, then m is an *ancestor* of n. Analogously, if m is an ancestor of n, then n is a *descendant* of m. A node with all its descendants and corresponding edges is called a *subtree*. Nodes that have at least one child are called *internal nodes*. Nodes without any children are called *leaf nodes*. *Terminal nodes* are leaf nodes that represent *terminal positions*, where the game is finished and the game-theoretic value, e.g., win, loss, or draw for the root player, can be determined.

2.1.2 The Search Tree

For non-trivial games, exploring the complete game tree is an impossible task. For example, the size of the game tree of chess from the initial position is estimated to be around 10^{123} nodes (Shannon, 1950). Instead of searching the complete game tree, a smaller portion of the tree is analyzed. This portion is called the *search tree*. The search tree has the same root node as the corresponding game tree, but it usually contains fewer nodes. Often, a search tree is constructed up to a certain *depth*. The depth of a tree is counted in *plies*. For example, a 4-ply search tree is a tree that looks four turns ahead. Search trees are gradually created during a *search process*, starting at the root node. A search tree introduces a new type of node, namely the *non-terminal leaf node*. A non-terminal leaf node does not have any children (yet), but it does not correspond to a terminal position. An internal node of which all children are added to the search tree is *expanded*.

The size of a search tree depends on the branching factor b, which is determined by the average number of children for each internal node, and the search depth d. Assuming b and d are uniform, the search tree consists of $O(b^d)$ nodes, which means that the number of nodes increases exponentially with the depth of the tree.

2.2 Searching in Two-Player Games

There exist various search techniques for analyzing two-player games. The basic technique for this purpose is minimax, which is explained in Subsection 2.2.1. $\alpha\beta$ search is an improvement of minimax search, which is described in Subsection 2.2.2. The basic $\alpha\beta$ search technique can be applied in deterministic games with perfect information. For handling chance events or hidden information, expectimax can be applied, which is discussed in Subsection 2.2.3.

2.2.1 Minimax Search

Minimax (Von Neumann and Morgenstern, 1944) is the classic depth-first search technique for sequential two-player games. These two players are called MAX and MIN . The minimax algorithm is designed for finding the optimal move for MAX, who is the player to move at the root node. The search tree is created by recursively expanding all nodes from the root in a depth-first manner until either the end of the game or the maximum search depth is reached. An *evaluation function* is used to assign a value to terminal nodes. For example, a win for the MAX player could be rewarded 1, a draw 0, and a loss -1. Assigning a value to non-terminal leaf nodes is more difficult, because the game-theoretic value is unknown. A static *heuristic evaluation function* is applied to evaluate these nodes. Donkers (2003) defined three ways to interpret the heuristic values: (1) as a prediction of the true game-theoretic value, (2) as an estimate of the probability to win the game, and (3) as a measure of the profitability of the position for the MAX player. Usually the third interpretation is used. Such a heuristic value is often a linear combination of domain-dependent features. Examples of such features include material balance in games like chess or checkers, or territorial balance in games like Amazons and Go. The values at the leaf nodes are backed up to the root, where MAX always chooses the child with the highest value and MIN chooses the child with the lowest value. When all moves have been investigated, the move corresponding to the child of the root with the highest value, is chosen as the best move. In short, the value V_n of a node n with children C in a minimax search tree can be calculated as follows.

$$V_n = \begin{cases} evaluate_n & \text{if } n \text{ is a terminal node} \\ heuristic_n & \text{if } n \text{ is a non-terminal leaf node} \\ \max_{c \in C} V_c & \text{if } n \text{ is an internal MAX node} \\ \min_{c \in C} V_c & \text{if } n \text{ is an internal MIN node} \end{cases} \tag{2.1}$$

where MIN and MAX nodes indicate the player to move. An example of a minimax search tree is given in Figure 2.1. Here, a Tic-tac-toe position is investigated. Six moves have been played already and it is currently ×'s turn, which means that × is the MAX player and ◦ is the MIN player. Each terminal node is assigned a value based on the outcome of the game. Because the search tree spans the complete game tree, all leaf nodes are terminal. Wins, draws, and losses of × are awarded 1, 0, and –1, respectively. The numbers next to the edges indicate the order in which the nodes are visited. The numbers under the Tic-tac-toe boards indicate the value of the corresponding node in the tree. After investigating the search tree, it turns out that × should play at the middle row of the right column.

When implementing minimax in a computer program, often the *negamax* (Knuth and Moore, 1975) algorithm is applied. Instead of having two separate methods for MAX and MIN nodes, there is only one method, which saves a significant number of lines of code and makes the code easier to maintain. The principle behind negamax is that $\max(a, b) = -\min(-b, -a)$. A negamax tree consists of only MAX nodes. Between two layers of the tree, the backpropagated values are inversed.

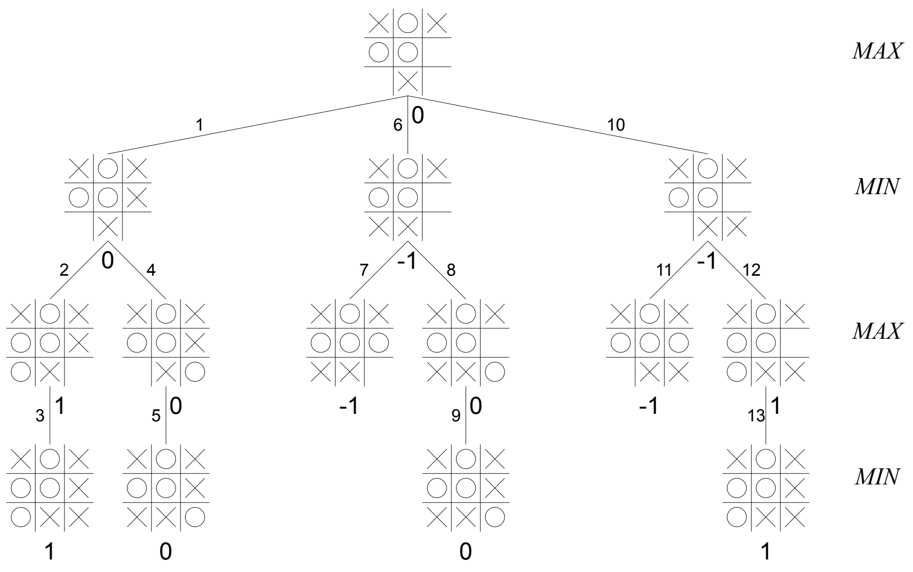


Figure 2.1: An example of a minimax tree in Tic-tac-toe.

Algorithm 2.1 Pseudo code of the negamax algorithm.

```
1: function NEGAMAX(node, depth, currentPlayer)
2:     if node.isTerminal() or depth ≤ 0 then
3:         return currentPlayer × evaluation(node);
4:     else
5:         return $\max_{c \in \text{node.children}}$ –NEGAMAX($c$, depth–1, –currentPlayer);
6:     end if
7: end function
```

The pseudo code for negamax can be found in Algorithm 2.1. In this algorithm, `node` refers to the current node, `depth` indicates the depth of the subtree of the current node, and `currentPlayer` is used for distinguishing between MAX and MIN nodes. In a MAX node, `currentPlayer` has value 1, and in MIN nodes it has value –1. The function `evaluation` assigns a value in perspective of the root player to the provided node. The initial call for the negamax algorithm is NEGAMAX(root, maximumDepth, 1).

2.2.2 $\alpha\beta$ Search

When analyzing minimax search trees, it is not necessary to investigate every node to determine the value of the root node and to find the best move. The technique of eliminating branches of the search tree from being explored is called *pruning*. The most-used pruning technique in minimax trees is *$\alpha\beta$ pruning* (Knuth and Moore, 1975).

When applying $\alpha\beta$ pruning, an $\alpha\beta$ window is maintained, which keeps track of the lower bound, α, and the upper bound, β, on the expected minimax value. $\alpha\beta$ search produces a cutoff if the returned value lies outside the $\alpha\beta$ window, i.e., the value is smaller than α or greater than β. This indicates that currently a suboptimal subtree is being investigated and that the remaining children of the currently investigated node cannot change the value of the root node.

In the best case, $O(b^{d/2})$ nodes are investigated (Knuth and Moore, 1975). With minimax, $O(b^d)$ nodes are investigated. It means that, in the best case, it is possible with $\alpha\beta$ pruning to search twice as deep in the same amount of time, compared to minimax search.

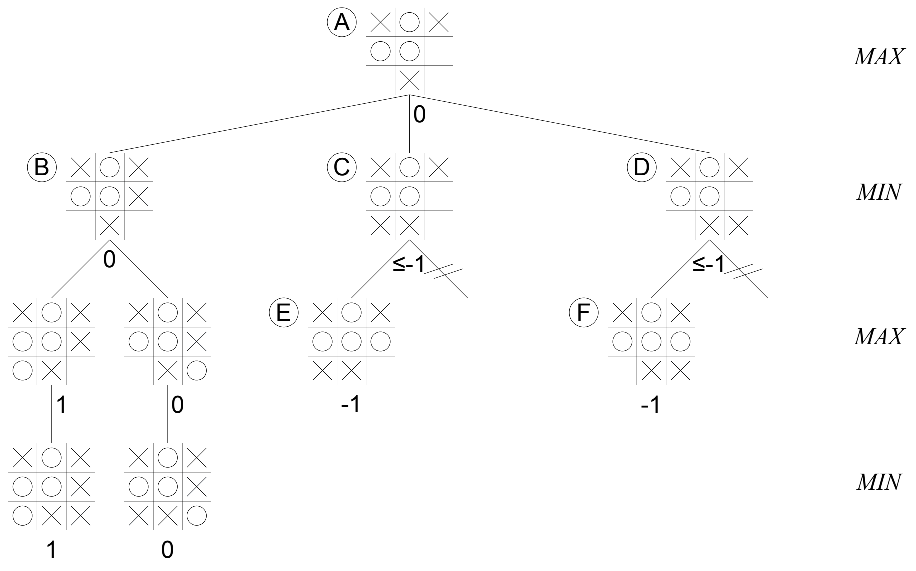


Figure 2.2: An example of $\alpha\beta$ pruning in Tic-tac-toe.

Figure 2.2 shows an example of $\alpha\beta$ pruning using the same initial position as in Figure 2.1. After investigating the subtree of node B, we can conclude that the value of node B is 0. This means that × is guaranteed a value of at least 0 in node A. The $\alpha\beta$ window in node A is $(0, \infty)$, i.e., $\alpha = 0$ and $\beta = \infty$. Because the value of node E is -1, we can conclude that the value of node C is ≤ -1. This value lies outside the $\alpha\beta$ window, because the value of node C is lower than α. × will always prefer node B over node C, because 0 is larger than ≤ -1. This means that × will not choose node C, regardless of the values of the siblings of node E, so all remaining siblings of node E, including their descendants, can be pruned. A similar situation occurs after investigating node F. Because node F has a value of -1, the value of node D is ≤ -1. × will prefer node B over node D, so the remaining siblings of node F can be pruned.

In Algorithm 2.2, the pseudo code for $\alpha\beta$ search in the negamax framework is

Algorithm 2.2 Pseudo code of the $\alpha\beta$ algorithm.

```
function ALPHABETA(node, depth, currentPlayer, $\alpha$, $\beta$);
    if node.isTerminal() or depth $\leq 0$ then
        return currentPlayer $\times$ evaluation(node);
    end if
    for all $c \in$ node.children do
        $\alpha = \max(\alpha$, –ALPHABETA($c$, depth–1, –currentPlayer, $-\beta$, $-\alpha$));
        if $\alpha \geq \beta$ then
            return $\beta$;
        end if
    end for
    return $\alpha$;
end function
```

given[1]. The major difference with Algorithm 2.1 is the introduction of the α and β parameters. At the root node, α is initialized to $-\infty$ and β to ∞. Because the negamax framework is applied, the $\alpha\beta$ window is flipped between two layers of the search tree. The initial call for the $\alpha\beta$ algorithm is ALPHABETA(root, maximumDepth, 1, $-\infty$, ∞).

2.2.3 Expectimax Search

$\alpha\beta$ search can be applied in two-player deterministic games with perfect information. If a game is non-deterministic, i.e., has chance events, then the basic minimax algorithm cannot be applied. *Expectimax* search (Michie, 1966) is a generalization of minimax search, introducing a new type of node, namely the CHANCE node. A CHANCE node represents a chance event, for instance a die roll. Expectimax can also be applied in games with imperfect information. CHANCE nodes are then used to model the revealing of hidden information. More information about this is provided in Chapter 7.

The value of a chance node is the weighted average over all child nodes, i.e.,

$$expectimax_n = \sum_{c \in C} P_c \times V_c \tag{2.2}$$

Here, n is the current chance node and C represents the children of the chance node, i.e., the set of possible outcomes of the chance event. P_c is the probability that child c will be chosen, i.e., the probability that the chance event corresponding to node c occurs, and V_c is the value of child c.

An example of an expectimax search tree is provided in Figure 2.3. MAX nodes are denoted with $\triangle$, MIN nodes with $\bigtriangledown$, and CHANCE nodes with $\diamond$. The numbers next to the edges indicate the probability for each chance event to occur. The value of node A can be computed as $expectimax_A = 0.5 \times 6 + 0.2 \times 4 + 0.3 \times 5 = 5.3$. The value of node B equals $expectimax_B = 0.5 \times 0 + 0.5 \times 9 = 4.5$. This means that MAX will choose node A.

[1]In all algorithms, we assume that the parameters are passed by value.

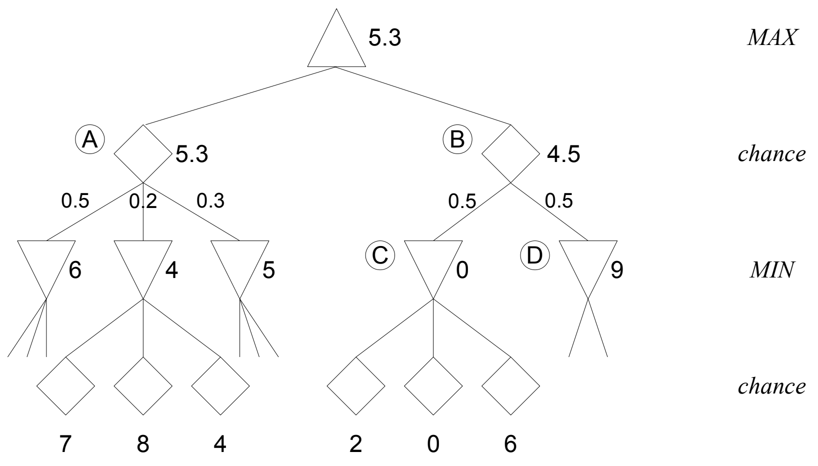


Figure 2.3: An example expectimax search tree.

Algorithm 2.3 Pseudo code of the expectimax algorithm.

```
function EXPECTIMAX(node, depth, currentPlayer, $\alpha$, $\beta$);
    if node.isTerminal() or depth $\leq 0$ then
        return currentPlayer × evaluation(node);
    end if
    if node is CHANCE node then
        $v = 0$;
        for all $e \in$ node.children do
            $v = v + P_e \times$ EXPECTIMAX($e$, depth–1, currentPlayer, $-\infty$, $\infty$);
        end for
    else
        for all $c \in$ node.children do
            $\alpha$ = max($\alpha$, –EXPECTIMAX($c$, depth–1, –currentPlayer, $-\beta$, $-\alpha$));
            if $\alpha \geq \beta$ then
                return $\beta$;
            end if
        end for
    end if
    return $\alpha$;
end function
```

The pseudo code for expectimax in the negamax framework is displayed in Algorithm 2.3. Note that in a chance node, the $\alpha\beta$ window cannot be passed.

Pruning at chance nodes can be applied by using Star1 or Star2 pruning (Ballard, 1983; Hauk, Buro, and Schaeffer, 2006a). With Star1, pruning occurs if it can be proven that the weighted sum of the values of the children falls outside the search window. Star1 pruning is only possible if the lower bound L and the upper bound U

of the possible values for the players are known. Pruning occurs if, after investigating the i^{th} child,

$$P_1V_1 + P_2V_2 + ... + P_iV_i + L(P_{i+1} + ... + P_n) \geq \beta \tag{2.3}$$

or

$$P_1V_1 + P_2V_2 + ... + P_iV_i + U(P_{i+1} + ... + P_n) \leq \alpha \tag{2.4}$$

where n indicates the number of children. In Figure 2.3, node D can be pruned using Star1 after investigating nodes A and C. Assuming $L = 0$ and $U = 10$, the value of node B can be at most $0.5 \times 0 + 10 \times 0.5 = 5$. This value is smaller than the value MAX can obtain by playing node A.

The drawback of Star1 is that generally the amount of pruning is rather small (Hauk, Buro, and Schaeffer, 2006b). This is because the worst case is assumed for each of the remaining children of the chance node. To obtain tighter bounds for a node p, Star2 may be employed. Star2 investigates one of the opponent's nodes for each child i of the chance node to gain a bound for node p. This is called probing. If the chance node is followed by MAX nodes, the obtained values act as a lower bound. Pruning occurs after investigating node i if

$$P_1V_1 + P_2V_2 + ... + P_iV_i + P_{i+1}W_{i+1} + ... + P_nW_n \geq \beta \tag{2.5}$$

In this formula, W_i is the probed value for child i. If the chance node is followed by MIN nodes, the values obtained by probing act as an upper bound. Pruning then occurs if

$$P_1V_1 + P_2V_2 + ... + P_iV_i + P_{i+1}W_{i+1} + ... + P_nW_n \leq \alpha \tag{2.6}$$

A disadvantage of Star2 pruning is that additional search effort is caused if no pruning is possible. This effect can be reduced if a transposition table (see Subsection 2.4.2) is applied. If no pruning occurs, the obtained bounds can be stored in the transposition table, which can later be used for tightening the search window during the regular search.

2.3 Searching in Multi-Player Games

The aforementioned techniques can be applied to two-player sequential games. However, many popular (board) games nowadays can be played by more than two players. These games are called *multi-player games*. When analyzing a search tree of a multi-player game, the basic $\alpha\beta$-pruning techniques cannot be applied. In this section, we describe three different search techniques that can be used for playing multi-player games. The oldest technique, called max$^{\text{n}}$, is described in Subsection 2.3.1. Next, Subsection 2.3.2 explains paranoid search, a search technique that allows $\alpha\beta$ pruning. Finally, Best-Reply Search is described in Subsection 2.3.3. This new technique is a variation on paranoid search.

2.3.1 Max^n Search

The traditional search technique for multi-player games is *max^n* (Luckhardt and Irani, 1986). This technique is a modification of minimax search to multi-player games. In the leaf nodes of the search tree, each player is awarded a value, based on a heuristic evaluation function. These values are stored in a tuple of size N, where N is the number of players. The sum of the values for all players can be constant. When backing up the values in the tree, each player always chooses the move that maximizes his score. An example of a max^n tree is displayed in Figure 2.4. In this example, $N = 3$ and the sum of the values for all players is 10. The numbers in the nodes denote which player is to move.

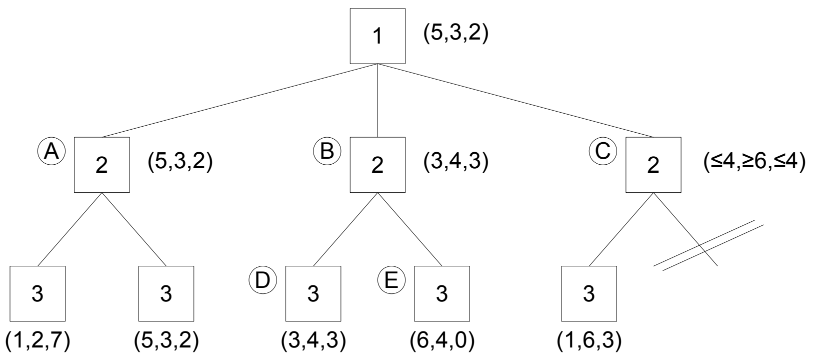


Figure 2.4: A max^n search tree with shallow pruning.

In multi-player games, multiple equilibrium points may exist (Nash, 1951; Jones, 1980). This may occur at any point if a player is indifferent between two or more children. Luckhardt and Irani (1986) showed that max^n is capable of finding one equilibrium point in any N-player zero-sum game with perfect information. Sturtevant (2003a) showed that, by changing the tie-breaking rule for choosing between children with the same value, the max^n value of a tree may change arbitrarily. An example of this phenomenon is provided in Figure 2.4. At node B, Player 2 is indifferent between nodes D and E. As shown in this figure, if Player 2 chooses node D, the value of the root node is $(5,3,2)$. However, if Player 2 would choose node E, the value of the root would be $(6,4,0)$. This shows that this tree has two equilibrium points where, depending on the tie-breaker rule, one is found. In Figure 2.4, the paranoid tie-breaker rule is applied. In case of ties, this rule chooses the node with the lowest value for the root player. This is a common tie-breaker rule. Another way to handle ties is *soft-max^n*, which was proposed by Sturtevant and Bowling (2006). Instead of choosing a single tuple to return in case of a tie, a set of tuples is returned. This set represents the possible outcomes if the corresponding node would be chosen. They proved that soft-max^n calculates a superset of all pure-strategy equilibria values at every node in a game tree. Another variant, called *prob-max^n* (Sturtevant, Zinkevich, and Bowling, 2006), uses opponent models to evaluate leaf nodes and backpropagate values in the tree. In the three-player perfect-information variant of Spades, these two variants outperform regular max^n significantly.

A disadvantage of max^n is that $\alpha\beta$ pruning is not possible. Luckhardt and Irani (1986) proposed *shallow pruning*, which is an easy and safe way to achieve some cutoffs. It is only possible if the sum of the values for all players has a fixed upper bound. Korf (1991) proved that shallow pruning finds the same results as basic max^n search, provided that they share the same tie-breaker rule. With shallow pruning, the asymptotic branching factor is $\frac{1+\sqrt{4b-3}}{2}$. However, in the average case the asymptotic branching factor is b, which means that $O(b^d)$ nodes are investigated (Korf, 1991). In the example in Figure 2.4, Player 2 is guaranteed a value of at least 6 at node C. Because the sum of the rewards for all players is 10, Player 1 cannot receive a value of more than 4 at this node. Because this is less than his reward when playing node A, which is 5, Player 1 will always prefer node A over node C, which means that the remaining children of node C can be pruned.

The max^n algorithm with shallow pruning is provided in Algorithm 2.4. In this algorithm, U indicates the upper bound on the sum of rewards for all players. The max^n algorithm is initialized using MAXN(root, maximumDepth, rootPlayer, $-\infty$).

Algorithm 2.4 Pseudo code of the max^n algorithm with shallow pruning.

```
1:  function MAXN(node, depth, currentPlayer, $\alpha$);
2:      if node.isTerminal() or depth $\leq 0$ then
3:          return evaluation(node);
4:      end if
5:      best = $(-\infty_1, -\infty_2, ..., -\infty_N)$;
6:      for all $c \in$ node.children do
7:          result = MAXN($c$, depth–1, nextPlayer, best[currentPlayer]);
8:          if result[currentPlayer] > best[currentPlayer] then
9:              best = result;
10:         end if
11:         if result[currentPlayer] $\geq U - \alpha$ then
12:             return result;
13:         end if
14:     end for
15:     return best;
16: end function
```

Sturtevant (2003c) introduced last-branch pruning and speculative pruning for max^n. These pruning techniques use the assumption that if the sum of lower bounds for a consecutive sequence of different players meets or exceeds U, the max^n value of any child of the last player in the sequence cannot be the max^n value of the game tree. Using this assumption, last-branch pruning guarantees that the cutoff will be correct by only pruning when the intermediate players between the first (root) player and the last player are searching their last branch. Speculative pruning is identical to last-branch pruning, except that it does not wait until intermediate players are on their last branch. Instead, it prunes speculatively, re-searching if necessary. Though the paranoid algorithm (Sturtevant and Korf, 2000) can search deeper in the same amount of time, speculative max^n is able to perform slightly better than paranoid in the perfect-information variants of the card games Hearts and Spades.

Another disadvantage of max^n lies in the fact that the assumption is made that opponents do not form coalitions to reduce the player's score. This can lead to too optimistic play. The optimism can be diminished by making the heuristic evaluation function more paranoid or by using the aforementioned paranoid tie-breaker rule (Sturtevant, 2003a).

2.3.2 Paranoid Search

Paranoid search was first mentioned by Von Neumann and Morgenstern (1944) and was later investigated by Sturtevant and Korf (2000). It assumes that all opponents have formed a coalition against the root player. Using this assumption, the game can be reduced to a two-player game where the root player is represented in the tree by MAX nodes and the opponents by MIN nodes. The advantage of this assumption is that $\alpha\beta$-like deep pruning is possible in the search tree, allowing deeper searches in the same amount of time. An example of a paranoid search tree is provided in Figure 2.5.

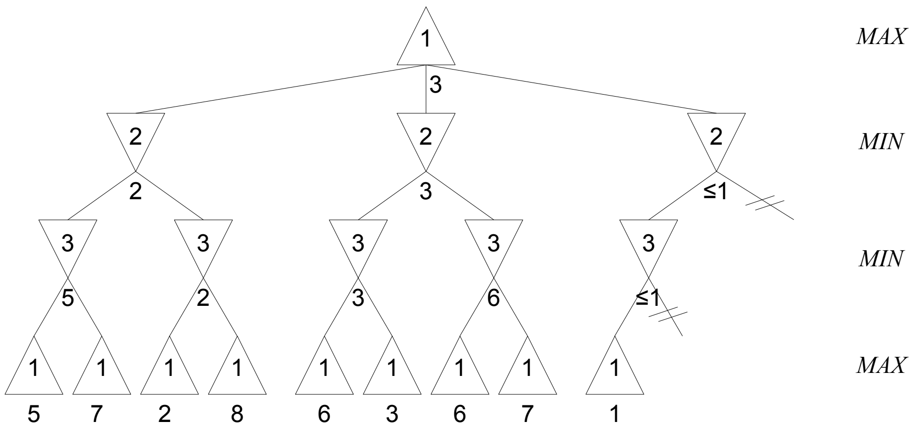


Figure 2.5: A paranoid search tree with $\alpha\beta$ pruning.

In the best case, $O(b^{\frac{N-1}{N}d})$ nodes are investigated in a paranoid search tree. This is a generalization of the best case for two-player games. Because more pruning is possible than in max^n, paranoid has a larger lookahead (Sturtevant, 2003a). Because of this larger lookahead, paranoid is able to outperform max^n in various multi-player games, such as Sergeant Major (Sturtevant and Korf, 2000), Chinese Checkers, Hearts (Sturtevant, 2003a), Focus, and Rolit (Schadd and Winands, 2011).

The disadvantage of paranoid search is that, because of the often incorrect paranoid assumption, the player may become too defensive. Furthermore, if the complete game tree is evaluated, a paranoid player may find that all moves are losing, because winning is often not possible if all opponents have formed a coalition. For instance, using Paranoid Proof-Number Search in the game of three-player 6×6 Rolit, Saito and Winands (2010) found that the first and the second player cannot get any points under the paranoid assumption. The third player can get 1 point, because he is the last one to play a piece on the board, which cannot be captured anymore by one of the opponents. In general, the deeper a paranoid player searches, the more pessimistic he becomes.

According to Sturtevant (2003a), the equilibrium points that are calculated by the paranoid algorithm are theoretically equivalent to those calculated by minimax. This means that a paranoid search tree has a single paranoid value that is the lower bound on the root player's score. We remark that the value found by paranoid search is only an equilibrium under the assumption that the opponents are in a fixed coalition. Because this is not the case in the actual game, paranoid search usually does not find an equilibrium point for the actual game.

The pseudo code of the paranoid algorithm in the negamax framework is provided in Algorithm 2.5.

Algorithm 2.5 Pseudo code of the paranoid algorithm.

```
function ALPHABETA(node, depth, currentPlayer, $\alpha$, $\beta$);
    if node.isTerminal() or depth ≤ 0 then
        if currentPlayer == rootPlayer then
            return evaluation(node);
        else
            return –evaluation(node);
        end if
    end if
    for all $c \in$ node.children do
        if currentPlayer == rootPlayer or nextPlayer == rootPlayer then
            $\alpha$ = max($\alpha$, –ALPHABETA($c$, depth–1, nextPlayer, $-\beta$, $-\alpha$));
        else
            $\alpha$ = max($\alpha$, ALPHABETA($c$, depth–1, nextPlayer, $\alpha$, $\beta$));
        end if
        if $\alpha \geq \beta$ then
            return $\beta$;
        end if
    end for
    return $\alpha$;
end function
```

2.3.3 Best-Reply Search

Recently, Schadd and Winands (2011) proposed a new search technique for playing multi-player games, namely *Best-Reply Search* (BRS). This technique is similar to paranoid search, but instead of allowing all opponents to make a move, only one opponent is allowed to do so. This is the player with the best counter move against the root player. Similar to paranoid, $\alpha\beta$ pruning is possible in BRS. In the best case, BRS investigates $O\left((b(N-1))^{\left\lceil \frac{2d}{N} \right\rceil /2}\right)$ nodes.

Schadd and Winands (2011) showed that BRS performs better than max^n and paranoid in the multi-player games Chinese Checkers and Focus. Furthermore, Esser (2012) and Gras (2012) found that BRS performs better than paranoid and max^n in four-player chess and the multi-player game Billabong, respectively.

The advantage of BRS is that more layers of MAX nodes are investigated, which leads to more long-term planning. Furthermore, this technique is less pessimistic than paranoid search, and less optimistic than max^n search. Compared to paranoid, only one opponent performs a counter move against the root player, instead of all opponents, and compared to max^n, BRS does not use the optimistic assumption that the opponents are only concerned with their own value.

The disadvantage is that, if passing is not allowed, invalid positions, i.e., illegal positions or positions that are unreachable in the actual game, are taken into account. This is the reason why in some games, such as trick-based card games like Bridge or Hearts, BRS cannot be applied. To overcome this disadvantage, Esser *et al.* (2013) proposed a modification of BRS, namely BRS$^+$, where the other opponents, rather than skip their turn, play a move based on domain knowledge. They showed that, using this enhancement, the modified version of BRS was able to outperform standard BRS in four-player chess. However, the performance of this variant strongly depends on the type of move ordering that is used. Furthermore, Gras (2012) found that in the race game Billabong, standard BRS performs better than BRS$^+$. In this thesis, we only investigate the performance of the standard BRS technique.

An example of a BRS tree is provided in Figure 2.6. Note that the moves of all opponents are compressed into one layer. The numbers next to the edges indicate to which players the corresponding moves belong. The pseudo code for BRS is similar to the pseudo code for $\alpha\beta$ search in Algorithm 2.2.

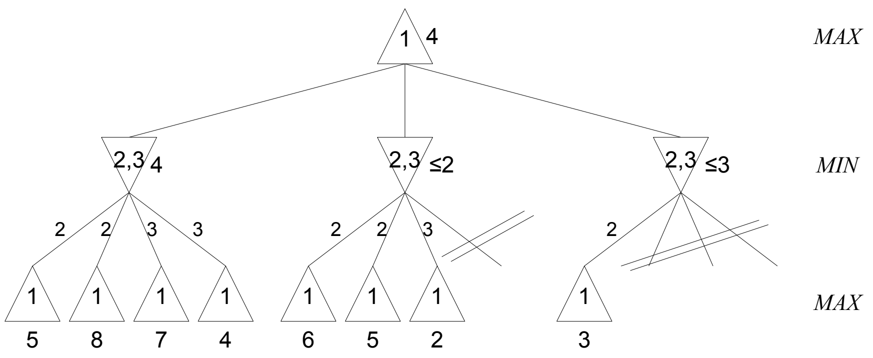


Figure 2.6: A BRS search tree with $\alpha\beta$ pruning.

We remark that BRS does not aim to find an equilibrium point or a theoretical bound on one's score. Its purpose is improving the playing strength of a minimax-based player in a multi-player game. Saffidine (2013) proved that if, in a game without zugzwang, a search for a best-reply win fails, then a search for a paranoid win fails as well and there is no need to perform such a search . The advantage of a BRS search over a paranoid search is that it is more shallow.

2.4 Enhancements for Minimax-Based Techniques

In this section we discuss three common enhancements used in minimax-based search techniques. In Subsection 2.4.1 we explain two commonly used dynamic move ordering techniques: killer moves and the history heuristic. Transposition tables are described in Subsection 2.4.2. Finally, iterative deepening is discussed in Subsection 2.4.3.

2.4.1 Dynamic Move Ordering

Move ordering is an important aspect in search techniques applying $\alpha\beta$-like pruning. The principle of move ordering is that the best moves are investigated first, so that the other moves can be pruned. There exist several move-ordering techniques that can be divided into two categories (Kocsis, 2003). (1) *Static move ordering* is independent of the search. It achieves a move ordering by using domain knowledge (see Chapters 3 and 7) or offline machine-learning techniques, such as the neural movemap heuristic (Kocsis, Uiterwijk, and Van den Herik, 2001). (2) *Dynamic move ordering* relies on information that was gained during the search. In the remainder of this section, we describe two dynamic domain-independent move-ordering techniques, namely killer moves and the history heuristic.

Killer Moves

One way to order moves is by applying *killer moves* (Akl and Newborn, 1977). The killer-move heuristic assumes that when a move produces a cutoff in a certain position, it will also do so in similar positions. In the implementation used in this thesis, for each ply in the search tree two killer moves are stored that produced a cutoff in the corresponding ply. At each newly searched node, the killer moves of the corresponding ply are checked first, provided that they are legal. If one of the killer moves produces a cutoff, all remaining legal moves do not have to be generated and investigated, saving a significant amount of time. When a new killer move is found, i.e., a move produces a cutoff that is not in the list of killer moves, it replaces the move that did not cause a recent cutoff.

History Heuristic

The *history heuristic* (Schaeffer, 1983; Schaeffer, 1989) is a generalization of the killer-move heuristic. This technique allows move ordering based on the number of prunings caused by the moves, irrespective of the position where these moves have been made.

For every move performed in the search tree, a history is maintained in a separate table. Because the total number of moves in a game is often relatively small, it is feasible to keep track of the history of all moves performed by all players. Moves are typically indexed based on the coordinates on the board, for instance the ‘from’ location and the ‘to’ location. When a move causes an $\alpha\beta$ pruning, its value in the table is increased by an amount, e.g. d^2 (Hyatt, Gower, and Nelson, 1990), where d is the depth of the subtree of the corresponding node.

Winands *et al.* (2006) described a new method for move ordering, called the *relative history heuristic*. It is a combination of the history heuristic and the butterfly heuristic (Hartmann, 1988). Instead of only recording moves that are the best move at a node, also the moves that are applied in the search tree are recorded. Both scores are taken into account in the relative history heuristic. In this way, moves that are strong on average, but occur less frequently, are favored over moves that occur more often, but are only sometimes the best.

2.4.2 Transposition Tables

During an $\alpha\beta$ search, it often happens that identical positions occur on different locations in the search tree. These are called *transpositions*. When the search algorithm finds a transposition, it may not be necessary to explore the node again. Instead, the results of the evaluation of the previous node are used. To store and retrieve this information, a *transposition table* (Greenblatt, Eastlake, and Crocker, 1967) is used.

Ideally, every position that has been evaluated in the past should be stored, but due to memory constraints this is not possible. Therefore, a *hash table* is used. A transposition table consists of 2^n entries, where each entry can hold information about a certain position. In order to determine where a position is stored, Zobrist hashing (Zobrist, 1970) is used to assign a 64-bit hash value to each game position. From this hash value, the first n bits are used to determine the location of the current board position in the transposition table. This is called the *hash index*. The remaining $64 - n$ bits are used as the *hash key*. The hash key is used to distinguish between positions with the same hash index.

An entry in the transposition table usually contains the following elements (Marsland, 1986; Hyatt *et al.*, 1990): (1) the hash key, (2) the value of the node, (3) a flag indicating whether the stored value is a bound (in case of a cutoff) or the exact value, (4) the best move, and (5) the search depth.

If a transposition is found during the search, the information stored in the transposition table can be used. Let d_t be the search depth stored in the table and d_c the current remaining search depth, If $d_t \geq d_c$ and the stored value is an exact value, then this value can immediately be returned as the value of the current node. If the stored value is a bound, then this value can be used to update the $\alpha\beta$ window. If $d_t < d_c$ the search still has to be executed. The stored move is investigated first, because it is probable that this move will produce a cutoff (again).

When using transposition tables, there are two types of errors that can occur: *type-1 errors* and *type-2 errors*. Type-1 errors occur when two different positions have the exact same hash value. It is difficult to recognize this type of error, as the hash key for both positions is the same. One possible way to detect this type of error is checking whether the stored best move is legal. If it is not, we can be assured that a type-1 error is found. If the move is legal, the error will go unnoticed and may lead to a wrong evaluation of the tree. Fortunately, this is quite rare (Breuker, 1998; Hyatt and Cozzie, 2005). A type-2 error, also called a *collision* (Knuth, 1973), occurs when two different positions with two different hash values are assigned the same table entry, i.e., the hash indices are the same. This type of error is easily recognized, because the hash keys of the two positions are different.

If a collision is detected, a replacement scheme is used to determine which node is stored. One of the most commonly used replacement schemes is called DEEP (Marsland, 1986; Hyatt *et al.*, 1990). When using this replacement scheme, the node with the largest search depth is stored in the table. If the depth of the new node is equal to the depth of the stored node, the new node overwrites the old one.

2.4.3 Iterative Deepening

As explained in Subsection 2.1.2, search trees usually have a predefined depth. However, it is difficult to predict how long an $\alpha\beta$ search up to a certain depth will take. To overcome this problem, *iterative deepening* (Korf, 1985) can be employed. Instead of performing one search up to a fixed depth, a sequence of tree searches is performed, where for each following iteration the maximum search depth is increased. By applying this technique, one has more control over how much time is spent for selecting a move. This is useful if only a limited amount of time is available.

Intuitively, it may seem that iterative deepening is an inefficient technique, as nodes will be visited and expanded multiple times. However, it turns out that the overhead is relatively small. This can be illustrated by the following example. If we traverse a search tree of depth $d = 4$ and branching factor $b = 20$, then the number of investigated nodes is (without pruning) $\sum_{i=0}^{d} b^i = 168{,}421$. With iterative deepening, the number of investigated nodes is $\sum_{i=0}^{d}(d-i+1)b^i = 177{,}285$. This calculation shows that the total number of investigated nodes is only 5.26% higher than without iterative deepening. This percentage is inversely proportional to the branching factor of the tree. The higher the branching factor, the lower the overhead. In practice, iterative deepening may even reduce the number of nodes searched. The reason for this is that the killer-move heuristic, the history heuristic and the transposition table are able to reuse information from previous searches to improve the move ordering at the next iteration, which may increase the amount of pruning in subsequent iterations.

2.5 Monte-Carlo Methods

Monte-Carlo methods (Metropolis and Ulam, 1949) are applied in a wide array of domains varying from field theory to cosmology (Metropolis, 1985). One application domain is computer game playing. Monte-Carlo simulations can be applied for evaluating leaf nodes. This is described in Subsection 2.5.1. In Subsection 2.5.2, we discuss the multi-armed bandit problem, which turned out to be an inspiration for the most popular Monte-Carlo based technique nowadays, namely Monte-Carlo Tree Search.

2.5.1 Monte-Carlo Evaluation

If constructing a strong static evaluation function for a game is too difficult, a different approach may be used. One such technique is called Monte-Carlo Evaluation (MCE) (Abramson, 1990). Instead of using domain-dependent features such as mobility or material advantage, a number of *playouts* (also sometimes called *samples*, *rollouts* or *simulations*) is started from the position to be evaluated. A playout is a sequence of (semi-)random moves. A playout ends when the game is finished and a winner can be determined according to the rules of the game. The evaluation of a position is determined by combining the results of all playouts using a statistical aggregate function. This can be for example the average game score, e.g., Black scored 35.5 points on average. Another example is the win rate, e.g., Black wins 53% of the games. The latter statistic is used often, including in this thesis.

In the domain of perfect-information games, MCE was particularly popular in Go (Bouzy and Helmstetter, 2004). This is mainly because constructing a strong static heuristic evaluation function for Go is difficult. The first application of MCE to Go was done by Brügmann (1993), who evaluated the initial position of the 9×9 Go board.

MCE is not only applicable to deterministic games with perfect information. During the 1990s, this technique was applied to non-deterministic games such as Backgammon (Tesauro and Galperin, 1997), and games with imperfect information such as *Bridge* (Ginsberg, 1999), *poker* (Billings *et al.*, 1999), and *Scrabble* (Sheppard, 2002).

The major disadvantage of Monte-Carlo Evaluation is that it is time-expensive. Because of the relatively long time to evaluate a leaf node, only a limited search depth can be reached. A solution was provided by the multi-armed bandit problem, which was an inspiration for the development of the Upper Confidence Bounds algorithm (Auer *et al.*, 2002).

2.5.2 Multi-Armed Bandit Problem

Monte-Carlo methods are suitable for tackling the *multi-armed bandit* problem (Auer *et al.*, 2002; Kocsis and Szepesvári, 2006). Suppose we have a set of k slot machines. When played, each machine i will gave a random reward according to an unknown distribution R_i with average reward μ_i. The player may iteratively select one of the machines and observe the reward. The goal is to maximize the total reward by finding the slot machine with the highest average reward as quickly as possible. To achieve this, the player should play (exploit) more promising machines more often, while still exploring other machines enough to ensure that he was not just unlucky with one of the machines. This is called the exploration-exploitation dilemma.

One way of solving this problem, is by computing an *upper confidence bound* for each machine. The easiest way to compute such an upper bound is by applying the UCB1 algorithm (Auer *et al.*, 2002). Each iteration, the machine i with the highest value v_i in Formula 2.7 is played.

$$v_i = \bar{x}_i + \sqrt{\frac{2\ln n}{n_i}} \tag{2.7}$$

Here, $\bar{x}_i$ is the sampled average reward obtained from playing machine i, n_i is the number of times machine i has been played, and n is the total number of times any machine has been played so far. The term $\bar{x}_i$ encourages exploitation of machines with higher average rewards, while the term $\sqrt{\frac{2\ln n}{n_i}}$ encourages exploration of machines that have not been played often yet.

A game can basically also be modeled as a multi-armed bandit problem. The player has a number of possible moves and should find the move that provides the highest chance to win. The reward is computed by randomly finishing the game. The player should find the move with the highest average reward, i.e., the move with the highest win rate.

The UCB1 algorithm can also be applied for selecting moves for the opponents. A tree is built to store the statistical data obtained from the previous iterations. This forms the basis of UCT (Kocsis and Szepesvári, 2006), which is the most widely-used selection mechanism in Monte-Carlo Tree Search.

2.6 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007a) is a best-first search technique that gradually builds up a search tree, guided by Monte-Carlo simulations. It extends the principle of Monte-Carlo evaluations to trees. In contrast to classic search techniques such as $\alpha\beta$-search, it does not require a static heuristic evaluation function. This makes MCTS an interesting technique for domains where constructing such an evaluation function is a difficult task.

In Subsection 2.6.1, we provide an overview of the basic MCTS algorithm. The application of MCTS to multi-player games in discussed in Subsection 2.6.2.

2.6.1 Basic MCTS Algorithm

The MCTS algorithm consists of four phases (Chaslot *et al.*, 2008b): selection, expansion, playout, and backpropagation (see Figure 2.7). By repeating these four phases iteratively, the search tree is constructed gradually. We explain these four phases in more detail below.

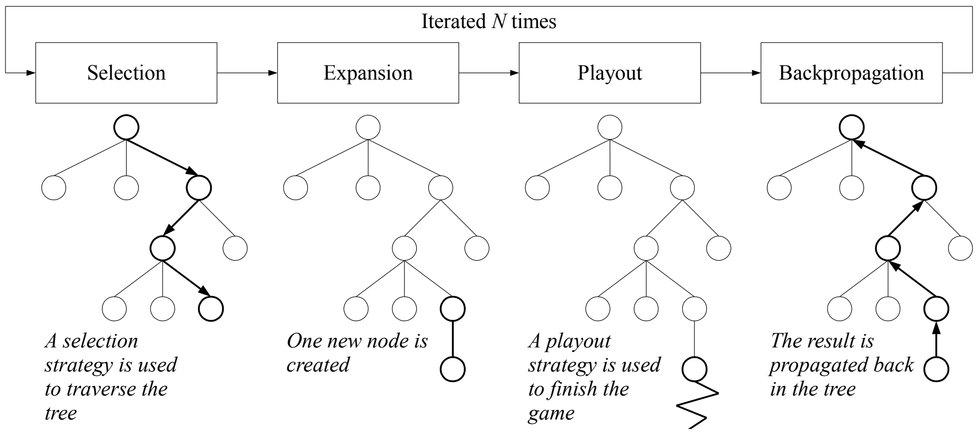

Iterated *N* times
Selection
Expansion
Playout
Backpropagation
A selection strategy is used to traverse the tree
One new node is created
A playout strategy is used to finish the game
The result is propagated back in the tree

Figure 2.7: MCTS scheme.

Selection. In the *selection* phase, the search tree is traversed, starting from the root, using a *selection strategy*. The most popular selection strategy is *UCT* (Kocsis and Szepesvári, 2006), which has been applied in several MCTS engines (cf.

Browne *et al.*, 2012). This selection strategy is based on UCB1. From a node p, the child i with the highest score v_i in Formula 2.8 is selected.

$$v_i = \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} \tag{2.8}$$

In this formula, $\bar{x}_i$ denotes the average score, i.e., the win rate, of node i. n_i and n_p denote the total number of times child i and its parent p have been visited, respectively. C is a constant, which balances exploration and exploitation. This selection strategy is applied until a node is reached that is not fully expanded, i.e., not all of its children have been added to the tree yet.

Expansion. In the *expansion* phase, one or more nodes are added to the tree. A popular expansion strategy is adding one node to the tree (Coulom, 2007a). This node corresponds to the first encountered position that was not added to the tree yet. If a large amount of memory is available, it is also possible to fully expand at once. In contrast, if only a limited amount of memory is available, nodes are only expanded if they have been visited at least a certain number of times.

Playout. During the *playout* phase, moves are played, starting from the position represented by the newly added node, until the game is finished. These may be random moves. However, game knowledge can be incorporated to make the playouts more realistic. This knowledge is incorporated in a *playout strategy* (Bouzy, 2005; Gelly *et al.*, 2006). One approach to improve the quality of the playouts is by applying *ϵ-greedy* playouts (Sutton and Barto, 1998; Sturtevant, 2008a). For each move played in the playouts, there is a probability of ϵ that a random move is played. Otherwise, domain knowledge can be used to assign a value to each valid move for the current player. The move with the highest heuristic value is played.

On the one hand, assigning a heuristic value to each possible move costs time, which may reduce the number of playouts per second. On the other hand, informed playouts improve the quality of the playouts to provide more reliable results (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008). Chapter 6 will go into further detail about applying domain knowledge in playouts.

Backpropagation. In the *backpropagation* phase, the result of the playout is propagated back along the previously traversed path up to the root node. Wins, draws, and losses are rewarded with 1, $\frac{1}{2}$, and 0 points, respectively. There exist various backpropagation strategies. The most popular and also the most effective strategy is *Average*, which keeps track of the average of the results of all playouts through each node (Coulom, 2007a). Other backpropagation strategies include *Max* (Coulom, 2007a), *Informed Average* (Chaslot *et al.*, 2006b), and *Mix* (Coulom, 2007a).

These four phases are repeated either a fixed number of times or until the time runs out. After the search is finished, one of the children of the root is selected as the best move. *Final move selection* techniques include choosing the max child (the child with the highest win rate), the robust child (the child with the highest visit count), the robust-max child (the child with both the highest win rate and visit count, where the search is continued until such a child exists), or the secure child (the child that maximizes a lower confidence bound) (Chaslot *et al.*, 2008b).

2.6.2 MCTS for Multi-Player Games

MCTS was first applied to deterministic perfect-information multi-player games by Sturtevant (2008a) and Cazenave (2008). Sturtevant applied MCTS for playing Chinese Checkers, and perfect-information variants of Spades and Hearts. He proved that UCT is able to compute a mixed equilibrium in a multi-player game tree and showed that, given sufficient thinking time, MCTS is able to outperform $\max^n$ and paranoid in Chinese Checkers. Cazenave applied MCTS to multi-player Go. He proposed several modifications, such as Paranoid UCT, UCT with alliances, and Confident UCT. Paranoid UCT modifies the playouts by introducing paranoid moves against the root player for the opponents. UCT with alliances models an explicit coalition of players in the playout phase. Confident UCT dynamically forms coalitions in the tree, based on the current board position.

Applying MCTS to multi-player games is quite straightforward. The difference with the application to two-player games is that, after each playout is finished, instead of a single value, a tuple of N values, is backpropagated in the tree. Each value in the tuple corresponds to the score achieved by one player. Each value is in $[0,1]$, where 1 corresponds to a win and 0 to a loss. In the case of a draw between several players, 1 point is divided evenly among these players. The selection strategy as described in Subsection 2.6.1 remains the same. Each player tries to maximize his own win rate using the standard UCT formula.

The search policy in this case is similar to $\max^n$, where each player tries to maximize this own score. In Chapter 4 we show that it is also possible to apply different search policies, such as paranoid or BRS, in the multi-player MCTS framework.

An example of a three-player MCTS tree is given in Figure 2.8. Note that at each node only the scores of the player who has to move at the parent's node are stored. For instance, at nodes B, C, and D only the score of Player 1 is stored. The win rate $\bar{x}_i$ is calculated as $\frac{s_q}{n}$, where s_q is the score of the current player q. Note that the sum of the visits of the children $\sum_i n_i$ is 1 smaller than the number of visits of the parent n_p. This is because the parent is visited once when it is created, and that only from the second visit of the parent, the children are visited. This is true for all nodes, except for the root. This is because, in the program MAGE (see Chapter 3), the root is created before the search starts.

The pseudo code of MCTS is provided in Algorithm 2.6. We remark that this pseudo code is generalized so that it can be applied to both the two-player and the multi-player variants of MCTS. The `chooseFinalMove(root)` function checks the children of the root node and determines which move to play in the actual game. The `uct(c)` function calculates the UCT value of child c. Instead of UCT, a different selection strategy may be used. The `expand(bestChild)` function adds a child, if it has not been visited yet, to the tree. Finally, `getPlayoutMove(currentPosition)` uses a playout strategy to select a move for the current player in the playout.

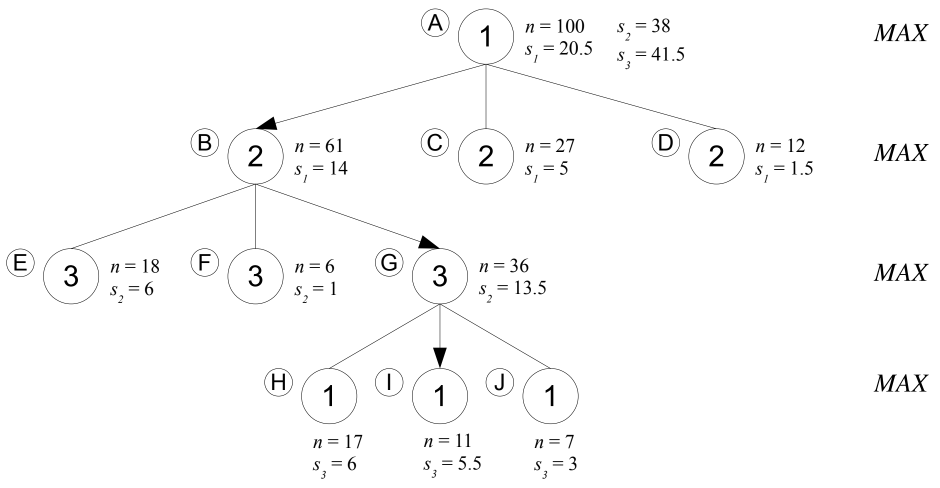


Figure 2.8: Example of a three-player MCTS tree.

Algorithm 2.6 Pseudo code of the MCTS algorithm.

```
1:  while time left do
2:      result = MCTS(root, rootPlayer);
3:      root.update(result);
4:  end while
5:  return chooseFinalMove(root);

6:  function MCTS(node, currentPlayer)
7:      bestChild = arg max_{c ∈ node.children} uct(c);
8:      if bestChild.visits == 0 then
9:          expand(bestChild);
10:         result = PLAYOUT(currentPosition);
11:     else
12:         result = MCTS(bestChild, nextPlayer);
13:     end if
14:     bestChild.update(result);
15:     return result;
16: end function

17: function PLAYOUT(currentPosition)
18:     while not currentPosition.isFinished() do
19:         playoutMove = getPlayoutMove(currentPosition);
20:         currentPosition.play(playoutMove);
21:     end while
22:     return currentPosition.evaluate();
23: end function
```

2.7 Related Work in MCTS

Over the past years, several enhancements have been developed to improve the performance of MCTS (cf. Browne *et al.*, 2012). In this section, we briefly discuss some of these enhancements.

There exist many ways to improve the selection phase of MCTS. The major challenge in the selection phase is selecting moves when the number of playouts in a move is still low. Two techniques to tackle this problem are discussed in Section 5.1, namely Rapid Action Value Estimation (Gelly and Silver, 2007) and Progressive Bias (Chaslot *et al.*, 2008b). Other selection strategies include First Play Urgency (Gelly and Wang, 2006), transposition tables (Childs, Brodeur, and Kocsis, 2008), move groups (Saito *et al.*, 2007; Childs *et al.*, 2008), Progressive Widening (Chaslot *et al.*, 2008b), and prior knowledge (Gelly and Silver, 2007). A new selection strategy is introduced in Chapter 5, namely Progressive History (Nijssen and Winands, 2011a).

Yajima *et al.* (2011) introduced four new expansion strategies, namely (1) siblings2, which expands a node when its visit count is more than double of its siblings, (2) transition probability, which expands a node if it has a high heuristic evaluation value compared to its siblings, (3) salient winning rate, which expands a node once it has a win rate that exceeds the win rate of its siblings by a certain margin, and (4) visit count estimate, which delays expansion of a node until the estimated visit count of a node can be shown to exceed the number of siblings of that node before the end of the search.

The ϵ-greedy playouts, as described in Subsection 2.6.1, require (light) domain knowledge in order to be functional. There exist various domain-independent techniques to enhance the quality of the playouts, including Last Good Reply (Drake, 2009), Last Good Reply with Forgetting (Baier and Drake, 2010), N-grams (Tak, Winands, and Björnsson, 2012), Move-Average Sampling Technique (Finnsson and Björnsson, 2008), Predicate-Average Sampling Technique (Finnsson and Björnsson, 2010), and Feature-Average Sampling Technique (Finnsson and Björnsson, 2010).

Nowadays, many computers have more than one processor core. To utilize the full potential of a multi-core machine, *parallelization* can be applied to an MCTS program. There exist three different parallelization techniques for MCTS: (1) root parallelization, (2) leaf parallelization, and (3) tree parallelization (Chaslot, Winands, and Van den Herik, 2008a). With *root parallelization*, each thread builds its own tree on which a regular MCTS search is performed. At the end of the simulations, the results of the different trees are combined. With *leaf parallelization*, one tree is traversed using a single thread. For the playouts, each thread performs one playout from this leaf node. Once all threads have finished, the results are backpropagated. When using *tree parallelization*, one tree is built, in which all threads operate independently. Mutexes may be used to lock (parts of) the tree, so that multiple threads cannot edit the same information in the tree at the same time. Enzenberger and Müller (2010) found, however, that tree parallelization without mutexes is more efficient. In a survey of different parallelization techniques, Bourki *et al.* (2011) showed that tree parallelization is the most effective.

CHAPTER 3

Test Environment

This chapter contains excerpts from the following publication:

1. Nijssen, J.A.M. and Winands, M.H.M. (2012a). An Overview of Search Techniques in Multi-Player Games. *Computer Games Workshop at ECAI 2012*, pp. 50–61, Montpellier, France.

This chapter describes the test environment used to answer the first three research questions. A test environment consists of games and a game program. First, we describe the rules and the employed domain knowledge for the four deterministic perfect-information games used in this thesis: Chinese Checkers, Focus, Rolit, and Blokus. Domain knowledge is incorporated in two different ways. (1) The *heuristic board evaluator*, i.e., the static heuristic evaluation function, assigns a value to each of the players, based on the current board position. This is used for evaluating leaf nodes in the minimax-based search techniques. (2) *Static move ordering* assigns a value to a move by a given player. This evaluation function is applied for move ordering in the minimax-based search techniques and in the playouts of MCTS. Furthermore, for each of the games we give the state-space complexity and the game-tree complexity (Allis, 1994). They provide an indication on how difficult the games are for computers to play them optimally. The former indicates how many different positions a game can be in. Because it is often difficult to give an exact number of legal positions, an upper bound is provided. The latter indicates the size of the full game tree, i.e., in how many ways the game can be played. The game-tree complexity is calculated using b^d, where b is the average branching factor of the tree and d represents the average depth. Finally, this chapter introduces the program MAGE, which is used to run the experiments in Chapters 4, 5, and 6 of this thesis.

In the next sections, we provide the rules, a complexity analysis, and the applied domain knowledge for Chinese Checkers (Section 3.1), Focus (Section 3.2), Rolit (Section 3.3) and Blokus (Section 3.4). Section 3.5 compares the complexity of these four games to the complexity of various other games that are popular in games research. An introduction to the computer program MAGE is given in Section 3.6.

3.1 Chinese Checkers

Chinese Checkers is a race game that can be played by two to six players. The game was invented in 1893 and since then has been released by various publishers under different names. Chinese Checkers is a popular test domain for multi-player search techniques. For instance, Sturtevant (2008b) used the game for investigating MCTS in multi-player games. Schadd and Winands (2011) tested the performance of BRS against paranoid and max^n in this game. Chinese Checkers is also frequently used as one of the test domains in General Game Playing (Clune, 2007; Finnsson, 2012; Tak *et al.*, 2012).

3.1.1 Rules

Chinese Checkers is played on a star-shaped board. The most commonly used board contains 121 fields, where each player starts with ten pieces. In this thesis a slightly smaller board is used (Sturtevant, 2008a) (see Figure 3.1). In this version, the board consists of 73 fields and each player has six pieces. The advantage of a smaller board is that games take a shorter amount of time to complete, which means that more Monte-Carlo simulations can be performed and more experiments can be run. Also, it allows the application of a stronger static evaluation function.

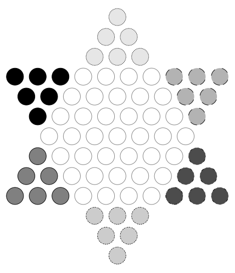


Figure 3.1: A Chinese Checkers board.

The goal for each player is to move all his pieces to his home base at the other side of the board. Pieces may move to one of the adjacent fields or they may jump over another piece to an empty field. Players are also allowed to make multiple jumps with one piece in one turn, making it possible to create a setup that allows pieces to jump over a large distance. The goal for each player is to move all his pieces to his home base at the other side of the board. Pieces may move to one of the adjacent fields or they may jump over another piece to an empty field. Players are also allowed to make multiple jumps with one piece in one turn, making it possible to create a setup that allows pieces to jump over a large distance. The first player who manages to fill his home base wins the game. The rules dictate that a player should move all his pieces to his home base. This rule can be exploited by keeping one piece in an opponent's home base to prevent this player from winning. To avoid this behavior, in this thesis a player has won if his home base is filled, regardless of the owner of the pieces, as long as the player has at least one of his own pieces in his home base. Draws do not occur in this game.

3.1.2 Complexity

A small Chinese Checkers board consists of 73 fields and each player has six checker pieces. For one player, there are $\binom{73}{6} = 170,230,452$ possible ways to assign the six pieces on the board. If there are two players, then there are $\binom{67}{6} = 99,795,696$ possible ways to assign the pieces for the second player for each of the assignments of the first player, leading to a total of 1.70×10^{16} possible positions in the two-player variant of Chinese Checkers.

For any number of players N, the number of possible ways to assign the pieces on the board can be computed using Formula 3.1.

$$\prod_{p=0}^{N-1} \binom{73-6p}{6} = \frac{73!}{(73-6N)!(6!)^N} \tag{3.1}$$

Using this formula, we can compute that in the three-player variant of Chinese Checkers the state-space complexity is 9.43×10^{23}. In the four-player variant the complexity is 2.73×10^{31}. Finally, in the six-player variant there are 2.33×10^{45} ways to assign the pieces of the players.

The game-tree complexity strongly depends on the number of players as well. The branching factor (b), average game length (d), and game-tree complexity (b^d) for the different variants are summarized in Table 3.1. These values were obtained by running 1500 selfplay games using the best MCTS-based players in the program MAGE (see Section 3.6).

Table 3.1 Average number of moves and game length for each variant of Chinese Checkers.

Variant	Average number of moves (b)	Average game length (d)	Game-tree complexity (b^d)
2 players	29.3	57.1	5.73×10^{83}
3 players	30.3	81.6	7.69×10^{120}
4 players	27.7	134.1	2.73×10^{193}
6 players	23.5	228.5	1.95×10^{313}

The numbers show that the average game length increases drastically if there are more players. This has two reasons. First, the players progress slower if there are more players, because it is each player's turn less often. Second, with more players the board becomes more cluttered in the midgame, causing the progress of the players to slow down even further. The average number of moves decreases with more players. Because the board is quite full, especially in the midgame, there are fewer moves available because the players block each other.

3.1.3 Domain Knowledge

For Chinese Checkers, the heuristic board evaluator uses a lookup table (Sturtevant, 2003a) that stores, for each possible configuration of pieces, the minimum number of moves a player should perform to get all pieces in the home base, assuming that there are no opponents' pieces on the board. For any player, the value of the board equals $28 - m$, where m is the value stored in the table that corresponds to the piece configuration of the player. We remark that 28 is the highest value stored in the table.

The static move ordering of Chinese Checkers is based on how much closer the moved piece gets to the home base (Sturtevant, 2008a). It uses the function $d_s - d_t$, where d_s is the distance to the home base of the piece that is moved, and d_t represents the distance of the target location to the home base. Note that the value of a move is negative if the piece moves away from the home base.

3.2 Focus

Focus is a capture game, which was described by Sid Sackson in 1969. This game has also been released under the name *Domination*. Focus has been used as a test bed for comparing the performance of BRS to paranoid and max^n by Schadd and Winands (2011).

3.2.1 Rules

Focus is played on an 8×8 board where in each corner three fields are removed. It can be played by two, three or four players. In Figure 3.2, the initial board positions for the two-, three- and four-player variants are given. In the two-player variant, each player has eighteen pieces. With three players, each player has thirteen pieces. Twelve of them are already on the board and one is played on one of the empty fields during the first round. In the four-player version, each player has thirteen pieces as well.

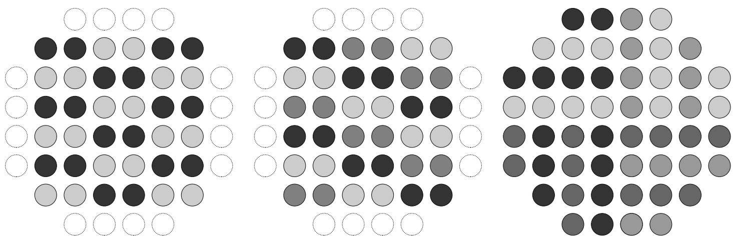

(a) 2 players (b) 3 players (c) 4 players

Figure 3.2: Set-ups for Focus.

In Focus, pieces can be stacked on top of each other. A stack may contain up to five pieces. Each turn a player moves a stack orthogonally as many fields as the stack is tall. A player may only move a stack of pieces if a piece of his color is on top. It is also allowed to split stacks into two smaller stacks. If a player decides to do so, then he only moves the upper stack as many fields as the number of pieces that is being moved. If a stack lands on top of another stack, the stacks are merged. If the merged stack has a size of $n > 5$, then the bottom $n - 5$ pieces are captured by the player, such that there are five pieces left. If a player captures one of his own pieces, he may later place one piece back on the board, instead of moving a stack. This piece may be placed either on an empty field or on top of an existing stack.

An example is provided in Figure 3.3. Player 4 controls the leftmost stack and can perform four different moves. The bottom two moves in this figure are examples of capture moves.

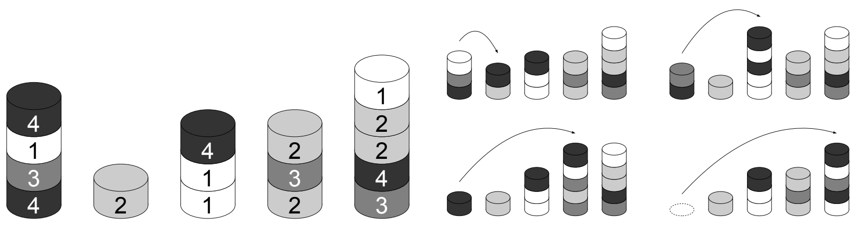


Figure 3.3: Example moves in Focus.

There exist two variations of the game, each with a different winning condition. In the standard version of the game, a player wins if all other players cannot make a legal move. However, such games can take a considerable amount of time to finish. Therefore, we chose to use the shortened version of the game. In this version, a player wins if he has either captured a certain number of pieces in total, or a number of pieces from each player. In the two-player variant, a player wins if he has captured at least six pieces from the opponent. In the three-player variant, a player has won if he has captured at least three pieces from both opponents or at least ten pieces in total. In the four-player variant, the goal is to capture at least two pieces from each opponent or to capture at least ten pieces in total. In both variants of the game, draws do not occur.

3.2.2 Complexity

To calculate the state-space complexity of Focus, we first compute in how many ways the pieces can be stacked on the board and distributed as captured pieces among the players, regardless of their color. Then, the number of ways to color the pieces is computed. To compute the number of ways the pieces can be distributed, a brute-force algorithm was used. This algorithm first divides the pieces in two groups, namely (1) pieces that have been captured by the players and (2) pieces that are still on the board. Next it computes, for each possible division of pieces, in how many ways the captured pieces can be distributed among the players and in how many ways the remaining pieces can be placed on the 52 squares on the board. Using this algorithm we found that, in the two-player variant, there are 8.64×10^{24} ways to assign the 36 pieces and 9.08×10^{9} ways to color them. This leads to a state-space complexity of 7.84×10^{34}.

In the three-player variant there are 1.94×10^{26} ways to distribute the 39 pieces and 8.45×10^{16} ways to color them. This means that the state-space complexity of the three-player variant is 1.64×10^{43}. Finally, in the four-player variant there are 52 pieces and there are 6.18×10^{30} ways to stack them and distribute them as captured pieces among the players. There are 5.36×10^{28} ways to color the pieces. This leads to a total of 3.31×10^{59} ways to assign the pieces.

The game-tree complexity of Focus is determined by running 1500 selfplay games using MCTS-based players in MAGE. The average number of moves, average game length, and game-tree complexity are summarized in Table 3.2. A four-player game of Focus takes on average about 25% longer than a three-player game, while a three-player game takes about 50% longer that a two-player game. The average branching factor of the three-player and four-player variants are quite similar. The branching factor in the two-player variant is significantly larger. This is because in the two-player variant both players start with eighteen pieces, instead of thirteen in the three-player and four-player variants. If a player has more pieces, he generally has more possible moves.

Table 3.2 Average number of moves and game length for each variant of Focus.

Variant	Average number of moves (b)	Average game length (d)	Game-tree complexity (b^d)
2 players	66.4	38.5	1.88×10^{101}
3 players	43.0	57.6	1.22×10^{94}
4 players	39.8	72.1	2.25×10^{115}

3.2.3 Domain Knowledge

For Focus, the heuristic board evaluator is based on the minimum number of pieces each player still needs to capture to win the game, r, and the number of stacks each player controls, c. Capturing the necessary pieces to win the game is the first goal for each player, so this feature is the most important in the evaluation function. Controlling many stacks is a secondary goal, because this increases the player's mobility. For each player, the score is calculated using the formula $600 - 100r + c$. Because the maximum value for r is 6, we ensure that the heuristic value will never be below 0.

The static move ordering gives a preference to moves that involve many pieces. The reasoning behind this is that creating large stacks gives the player more mobility, as larger stacks provide more possible moves. Furthermore, if a stack larger than five pieces is created, pieces are captured and the larger the stack, the more pieces are captured. The static move ordering applies the function $10(n + t) + s$, where n is the number of pieces moved, t is the number of pieces on the target location, and s is the number of stacks the player gained, i.e., the number of stacks the player controls after the move minus the number of stacks the player controls before the move. The value of s can be 1, 0, or -1. This feature adds a small preference for moves that increase the number of stacks that the player controls.

3.3 Rolit

Rolit, published in 1997 by Goliath, is a multi-player variant of the two-player territory-based game Othello. This game was introduced in 1975. It is similar to a game invented around 1880, called Reversi. This game was invented by either Lewis Waterman or John W. Mollett. At the end of the 19th century it gained much popularity in England, and in 1893 games publisher Ravensburger started producing the game as one of its first titles.

Othello is a popular two-player test domain. Currently, the best Othello programs outperform the best human players (Allis, 1994). Buro used Othello as a test domain for experiments in ProbCut (Buro, 1995) and Multi-ProbCut (Buro, 2000).

Saito and Winands (2010) applied Paranoid Proof-Number Search to solve the two-, three-, and four-player variants of Rolit on 4×4 and 6×6 boards, and the four-player variant on the 8×8 board. They used this technique to calculate the minimum score each player is guaranteed, assuming optimal play and that all opponents have formed a coalition. It was also one of the domains for testing the performance of BRS, paranoid, and max^n by Schadd and Winands (2011).

3.3.1 Rules

Before describing the rules of Rolit, we first explain the rules of Othello. Othello is played by two players, Black and White, on an 8×8 board. On this board, so-called discs are placed. Discs have two different sides: a black one and a white one. A disc belongs to the player whose color is faced up. The game starts with four discs on the board, as shown in Figure 3.4(a). Black always starts the game, and the players take turns alternately. When it is a player's turn he places a disc on the board in such a way that he captures at least one of the opponent's discs. A disc is captured when it lies on a straight line between the placed disc and another disc of the player making the move. Such a straight line may not be interrupted by an empty square or a disc of the player making the move. All captured discs are flipped and the turn goes to the other player. If a player cannot make a legal move, he has to pass. If both players have to pass, the game is over. The player who owns the most discs, wins the game.

For Rolit, the rules are slightly different. Rolit can be played by up to four players, called Red, Yellow, Green, and Blue. The pieces in Rolit are four-sided; each side

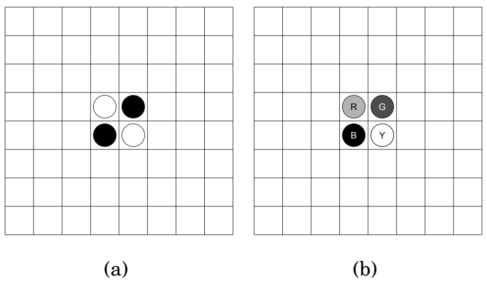


Figure 3.4: Set-ups for Othello (a) and Rolit (b).

represents one color. The initial board position is shown in Figure 3.4(b). This setup is used for the two-, three-, and four-player variants. With three and two players, there are one and two neutral pieces, respectively. The largest difference is that if a player cannot capture any pieces, which will occur regularly during the first few rounds of a four-player game, he may put a piece orthogonally or diagonally adjacent to any of the pieces already on the board. Using this rule, passing does not occur and the game is finished when the entire board is filled. The scoring is similar to Othello. The player owning the most pieces wins. We remark that, contrary to Focus and Chinese Checkers, Rolit can end in a draw between two or more players.

3.3.2 Complexity

An upper bound of the state-space complexity of Rolit with N players can be computed using the formula $N^4 \times (N+1)^{60}$. The center four squares can each have N different states. They cannot be empty because they are already filled at the start of the game. The remaining 60 squares can each have $N+1$ different states. They can be occupied by any of the N players or they can be empty. For the two-player variant of Rolit, the state-space complexity is 6.78×10^{29}. For the three-player variant the complexity is 1.08×10^{38}. For the four-player variant the complexity is 2.22×10^{44}. These numbers are upper bounds for the state-space complexity. The actual number of legal positions in Rolit is smaller than these numbers. For example, any position where not all pieces are connected to each other are illegal, because a piece may, by definition, only be placed adjacent to a piece already on the board.

A game of Rolit always takes 60 turns. Saito and Winands (2010) found that the average branching factor of Rolit for any number of players is approximately 8.5. This means that the game-tree complexity is approximately $8.5^{60} = 5.82 \times 10^{55}$. For Othello, the average game length is 59.85 and the average branching factor is 8.25. These numbers are gained by analyzing the games from the WTHOR database, which stores nearly 110,000 games by professional players, played between 1977 and 2012.[1] This leads to a game-tree complexity of $8.25^{59.85} = 7.08 \times 10^{54}$. Compared to Othello, the game-tree complexity of Rolit is slightly larger. There are two reasons for this difference. (1) A game of Rolit always takes 60 turns, while a game of Othello can be finished earlier if none of the players can move anymore. (2) If a player cannot perform a capture move, he may place a piece anywhere adjacent to a piece on the board. Usually, the branching factor at these positions is relatively high.

3.3.3 Domain Knowledge

The heuristic board evaluation of Rolit depends on two features: the mobility of the player and the number of stable pieces (Rosenbloom, 1982; Nijssen, 2007). The mobility is measured by the number of moves a player has. The higher the number of possible moves, the better the position generally is for the player. Stable pieces are pieces that cannot change color anymore during the rest of the game. The score of a player is calculated using $m + 10s$, where m is the number of moves and s is the number of stable pieces.

[1]http://www.ffothello.org/info/base.php

The static move ordering depends on the location of the square where the piece is placed. The values of the squares are displayed in Figure 3.5. Corners are most valuable, as pieces on these squares are always stable and provide an anchor for adjacent pieces to be stable as well. The squares adjacent to the corners are tricky, especially if the corner has not been captured yet. Pieces on these squares may provide the opponents an opportunity to capture the corner.

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 3.5: The values of the squares in Rolit.

3.4 Blokus

Blokus is a four-player tile-based game developed by Bernard Tavitian in 2000. Because Blokus is a relatively new game, not much research has been performed in this domain. Shibahara and Kotani (2008) used the two-player variant of Blokus, Blokus Duo, as a test domain for combining final score and win rate in Monte-Carlo evaluations. Blokus Duo is played on smaller board than Blokus and the players start on a different position.

3.4.1 Rules

A Blokus board consists of 20×20 squares. Each player receives 21 pieces varying in size from one to five squares in all possible shapes (see Table 3.3). Alternately, the players place one of their pieces on the board. The pieces may be rotated in any way. The difficulty in this game is that any square may only be occupied by one piece and two pieces of the same player may not be orthogonally adjacent. However, if a new piece is placed, it has to be diagonally adjacent to one or more of the player's pieces on the board. The first pieces of the players should all be placed in one of the corners.

The game finishes when none of the players can place a piece on the board anymore. The player who has the largest number of squares on the board occupied is the winner. Note that, similar to Rolit, draws can occur. However, there is one tie breaker in this game. If more than one player manages to place all pieces on the board, the winner is the player who placed the monomino, the piece of size 1, on the board during the last round.

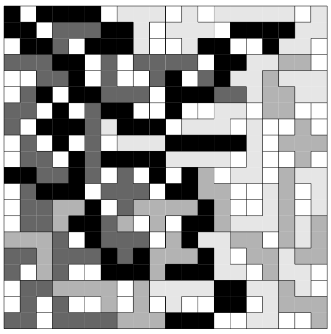
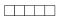
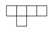
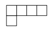
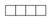
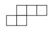


Figure 3.6: A terminal position in Blokus.

Table 3.3 Overview of the 21 different pieces in Blokus.

Piece	Rotations	Places	Total	Piece	Rotations	Places	Total
	1	20 × 20	401		4	18 × 18	1297
	2	19 × 20	761		8	18 × 19	2337
	4	19 × 19	1445		1	18 × 18	325
	2	18 × 20	721		8	18 × 18	2593
	1	19 × 19	362		2	16 × 20	641
	4	18 × 19	1369		8	17 × 19	2585
	4	18 × 19	1369		8	17 × 19	2585
	2	17 × 20	681		8	17 × 19	2585
	8	18 × 19	2737		4	18 × 18	1297
	4	18 × 19	1369		4	18 × 18	1297
	4	18 × 18	1297				

3.4.2 Complexity

In Blokus, there are 400 squares and each can be in one of five different states: it can be empty or occupied by one of the four players. Therefore, the theoretical upper bound of the state-space complexity for Blokus is $5^{400} = 3.87 \times 10^{279}$. A smaller upper bound can be calculated using the values from Table 3.3. This table shows in how many ways each piece can be placed on the board, including still being in the player's hand. Using these values we can compute that the 21 pieces of each player can be placed in 3.40×10^{64} on the board. This does not consider the rules that pieces may not overlap or be orthogonally adjacent or that all pieces have to be diagonally adjacent. With four players, the 84 pieces can be arranged in at most 1.33×10^{258} ways. The actual state space of Blokus is much smaller than this number, because many illegal positions where pieces overlap, pieces are not connected or pieces of the same player are orthogonally adjacent are counted.

A game of Blokus can take at most 84 turns. In practice, most games are finished earlier because often the players cannot place all pieces on the board. A game of Blokus takes on average 73.1 turns. The average number of moves of the players during the game is 139.1. These numbers were obtained by running 1500 selfplay games with four MCTS-based players in MAGE. This leads to a game-tree complexity of $139.1^{73.1} = 4.76 \times 10^{156}$.

3.4.3 Domain Knowledge

The goal in Blokus is to have as many squares on the board occupied as possible. For Blokus, the heuristic board evaluator counts the number of squares that each player has occupied. Because Blokus has a relatively large board, using a more complex evaluation function is quite time-consuming, so only this feature is used in the evaluation function.

The static move ordering takes the size of the piece that is played on the board into account. Large pieces are preferred over small ones. Playing large pieces earlier in the game is generally advantageous, because there is still plenty of space to place them. As the board becomes more filled, placing large pieces will quickly become impossible, while placing smaller pieces in the remaining gaps may still be possible. Furthermore, keeping the monomino until the end of the game may be useful to gain the bonus if this piece is played last.

3.5 Comparison of Complexities to Other Games

A graphical representation of the state-space and game-tree complexity of Chinese Checkers, Focus, Rolit, and Blokus, along with various other games, can be found in Figure 3.7 (Shannon, 1950; Allis, 1994; Tesauro and Galperin, 1997; Van Rijswijck, 2000; Winands, Uiterwijk, and Van den Herik, 2001; Van den Herik, Uiterwijk, and Van Rijswijck, 2002; Iida, Sakuta, and Rollason, 2002; Joosten, 2009). The numbers in bold indicate the games that are investigated in this thesis. The numbers in italic represent games that have been solved.

The state-space complexity of the Chinese Checkers variants increases with more players. While the state-space complexity of the two-player variant is close to the solved game Awari, the state-space complexity of the six-player variant is comparable to chess. The game-tree complexity increases faster than the state-space complexity. The game-tree complexity of two-player Chinese Checkers is between 15×15 Go-Moku and 11×11 Hex, while the game-tree complexity of the six-player variant is higher than most games, approaching 19×19 Go.

The complexity of the different variants of Focus is relatively close to chess and 11×11 Hex. With more players, the state-space complexity slightly increases, while the game-tree complexity fluctuates slightly.

The complexity of two-player Rolit is close to Lines of Action and, as explained in Subsection 3.3.2, Othello. While the game-tree complexity remains the same for the three-player and four-player variants, the state-space complexity slightly increases with more players. The state-space complexity of four-player Rolit is similar to chess.

The game-tree complexity of Blokus is similar to 19 × 19 Connect6 and Havannah, though its state-space complexity is considerably larger. However, with a more sophisticated algorithm to approximate the state-space complexity of Blokus, the upper bound may be lowered.

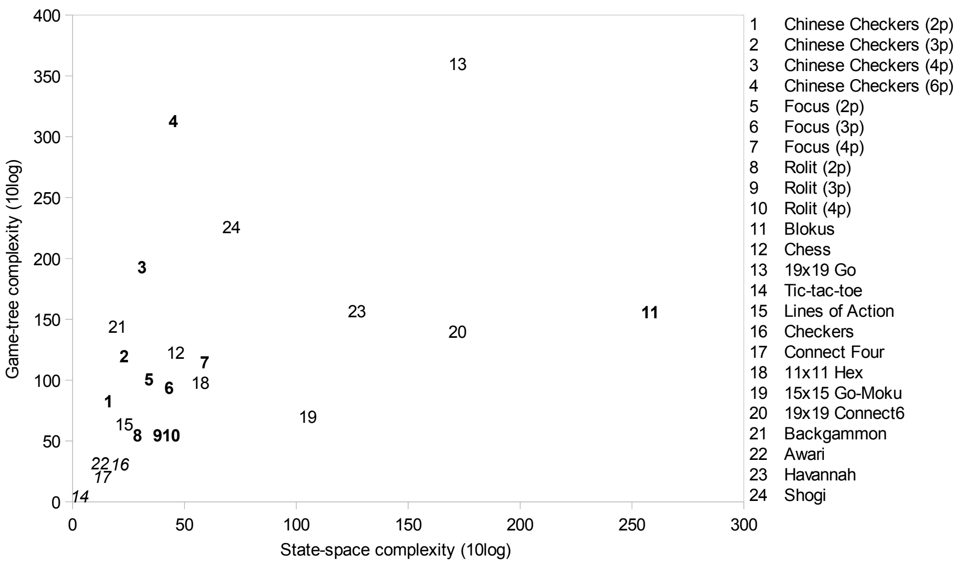


Figure 3.7: Graphical representation of the state-space and game-tree complexity of 24 different game variants.

3.6 Game Engine: MAGE

The experiments in Chapters 4, 5 and 6 are performed in the program called MAGE, which is an acronym for *Modular Advanced Game Engine*. It was designed during this Ph.D. research. As the name suggests, MAGE is a modular engine. It contains interfaces for the games and the players. Using this system, adding new search techniques or new games is quite easy. Furthermore, comparisons of search techniques between games are more fair, because in each different game the same engine is used. A disadvantage is that optimization is more difficult, because game-specific optimizations cannot be implemented in the engine of the MCTS-based and minimax-based players. MAGE is used for playing deterministic games with perfect information.

MAGE is written in Java. For the MCTS-based players, the UCT exploration constant C is set to 0.2. They also use ϵ-greedy playouts with $\epsilon = 0.05$ and Tree-Only Progressive History (see Chapter 5) with $W = 5$. These values were achieved by systematic testing. Because sufficient memory is available, in the expansion phase, all children are added to a node at once. The advantage of this is that possible moves only have to be generated once. After the last MCTS iteration, the robust child, i.e., the child of the root node with the highest visit count, is chosen as the best move.

All minimax-based players use a transposition table with the DEEP replacement scheme (Breuker, Uiterwijk, and Van den Herik, 1996), and static move ordering. Furthermore, the paranoid and BRS players use killer moves (Akl and Newborn, 1977) and the history heuristic (Schaeffer, 1983). The killer moves are always tried first. The values in the history table are used to break ties between moves with the same value according to the static move ordering. Finally, for the max^n player, shallow pruning is applied (Sturtevant and Korf, 2000). To allow shallow pruning, the scores retrieved from the heuristic evaluation function are normalized, such that the sum of the scores for all players is 1. To make the minimax-based players less deterministic, a small random factor is added to the board evaluators. This random factor differentiates board positions with the same heuristic value. This idea was investigated by Beal and Smith (1994), who found that a chess program performed considerably better if a random factor was added to a positional evaluation function.

CHAPTER 4

Search Policies for Multi-Player MCTS

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

1. Nijssen, J.A.M. and Winands, M.H.M. (2011a). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 238–249, Springer-Verlag, Berlin, Germany.
2. Nijssen, J.A.M. and Winands, M.H.M. (2012a). An Overview of Search Techniques in Multi-Player Games. *Computer Games Workshop at the ECAI 2012*, pp. 50–61, Montpellier, France.
3. Nijssen, J.A.M. and Winands, M.H.M. (2013). Search Policies in Multi-Player Games. *ICGA Journal*, Vol. 36, No. 1, pp. 3–21.

Monte-Carlo Tree Search (MCTS) is a best-first search technique that can easily be extended from two-player to multi-player games (Sturtevant, 2008b). Different search policies can be applied that indicate how the children are selected and how the results are backpropagated in the tree. The basic multi-player MCTS algorithm applies a search policy that is analogous to the max^n search tree. In a max^n tree (see Subsection 2.3.1), each player maximizes his own score. In standard MCTS, a similar principle is applied. Each player tries to maximize his own win rate, while not considering the win rates of the opponents. Similar to the classic minimax framework, it is possible to apply the paranoid and BRS search policies to MCTS as well.

This chapter answers the first research question by investigating the max^n, paranoid, and BRS search policies in the MCTS framework. They are called MCTS-max^n, MCTS-paranoid, and MCTS-BRS, respectively. Their performance is tested in four different multi-player games, namely Chinese Checkers, Focus, Rolit, and Blokus. Furthermore, these MCTS variants are compared to the minimax-based search techniques.

Next, MCTS-max^n is modified so that it is able to prove positions and therefore play tactical lines better. The Monte-Carlo Tree Search Solver (MCTS-Solver) (Winands, Björnsson, and Saito, 2008) concept is applied in MCTS-max^n. Experiments are performed in the sudden-death game Focus.

The chapter is structured as follows. First, in Section 4.1 related work on search techniques in multi-player games is discussed. Section 4.2 explains how the paranoid and BRS search policies can be applied in the MCTS framework. Next, Section 4.3 provides the experimental results of the different search policies. A background of the MCTS-Solver is given in Section 4.4. MCTS-Solver for multi-player games is introduced in Section 4.5. Subsequently, in Section 4.6 we provide the experimental results for the multi-player MCTS-Solver. Finally, the chapter conclusions and an overview of possible future research directions are given in Section 4.7.

4.1 Search Policies: Related Work

Sturtevant (2008b) provided an analysis of MCTS in Chinese Checkers and perfect-information variants of Spades and Hearts. He showed that, in Chinese Checkers, MCTS wins over 90% of the games against max^n and paranoid, and that paranoid significantly outperforms max^n. In these experiments, all players were allowed 250,000 node expansions per move. He found that with fewer expansions there is not enough data for UCT to form accurate estimates. For instance, MCTS only wins 16.7% of games against paranoid when both search techniques are only allowed 1600 node expansions per move. In other domains, MCTS plays on a par with existing programs in the game of Spades, and better than existing programs in Hearts. Furthermore, Sturtevant (2008b) proved that UCT is able to compute a mixed equilibrium in a multi-player game tree.

Cazenave (2008) applied MCTS to multi-player Go. He introduced a technique called Paranoid UCT. In his design of Paranoid UCT, the paranoia is modeled in the playouts, while the MCTS tree is traversed in the usual way. He tested the performance of Paranoid UCT against a RAVE player in three-player Go, and found that Paranoid UCT performed better. A disadvantage of this technique is that it requires a definition of paranoid moves for each game in which it is applied. Cazenave also introduced Confident UCT. This variant uses the current board position to model coalitions in the tree. Three different variants were proposed. The first, called the Confident algorithm, develops multiple MCTS trees, where in each tree the root player has a coalition with a different opponent. For the final move selection, it assumes a coalition which is most beneficial for the root player. The second, called the Symmetric Confident algorithm, is similar to the confident algorithm, but assumes a coalition only if it is beneficial for the opponent as well. The third, called the Same algorithm, assumes a coalition with other Same players. Experimental results showed, however, that Confident UCT is outperformed by RAVE and paranoid search.

Schadd and Winands (2011) introduced Best-Reply Search (BRS), for deterministic multi-player games with perfect information. They tested BRS against max^n and paranoid in the games of Chinese Checkers, Focus, and Rolit. In all games, BRS was superior to max^n. They showed that BRS also outperforms paranoid in Chinese Checkers and Focus. In Rolit, BRS was on equal footing with paranoid. The application of BRS to the MCTS framework was proposed, but it was neither tested nor implemented.

4.2 Alternative Search Policies

In this section, we propose the application of the paranoid and BRS search policies in MCTS. First, MCTS-paranoid is explained in more detail in Subsection 4.2.1. MCTS-BRS is discussed in Subsection 4.2.2. For a description of the max^n policy in MCTS, called MCTS-max^n in this chapter, we refer to Subsection 2.6.2.

4.2.1 MCTS-Paranoid

The idea of using a paranoid search policy in MCTS was suggested by Cazenave (2008), however he did not implement or test it. When applying the paranoid search policy in MCTS, the structure of the tree remains intact, however all nodes in the opponents' layers are changed into MIN nodes. All nodes in the root player's layers remain MAX nodes and the standard UCT formula is applied. When incorporating the paranoid search policy in MCTS, the opponents use a different UCT formula. Instead of maximizing their own win rate, they try to minimize the win rate of the root player. In the MIN nodes of the tree, the following modified version of the UCT formula is used. The child i with the highest value v_i is selected as follows (Formula 4.1).

$$v_i = (1 - \bar{x}_i) + C\sqrt{\frac{\ln(n_p)}{n_i}} \tag{4.1}$$

Similar to the UCT formula (Formula 2.8), $\bar{x}_i$ denotes the win rate of node i. n_i and n_p denote the total number of times child i and its parent p have been visited, respectively. C is a constant, which balances exploration and exploitation.

The major difference with the standard UCT formula is that, at the MIN nodes, $\bar{x}_i$ does not represent the win rate at child i of the current player, but of the root player. Essentially, $(1-\bar{x}_i)$ indicates the win rate of the coalition of the opponents. Analogous to paranoid in the minimax framework, the opponents do not consider their own win rate.

An example of a paranoid tree in the MCTS framework for a three-player game is provided in Figure 4.1. In this example, 100 playouts have been simulated so far. For each node, n indicates how often the node has been visited so far and s indicates the cumulative score of the root player. The value $\bar{x}_i$ of node i is computed using $\bar{x}_i = \frac{s_i}{n_i}$. The root player applies the standard UCT formula (Formula 2.8) to select a child. Assuming $C = 0.2$, the UCT values for nodes B, C and D are:

$$v_B = \frac{14}{61} + 0.2\sqrt{\frac{\ln 100}{61}} \approx 0.284$$

$$v_C = \frac{5}{27} + 0.2\sqrt{\frac{\ln 100}{27}} \approx 0.268$$

$$v_D = \frac{1.5}{12} + 0.2\sqrt{\frac{\ln 100}{12}} \approx 0.249$$

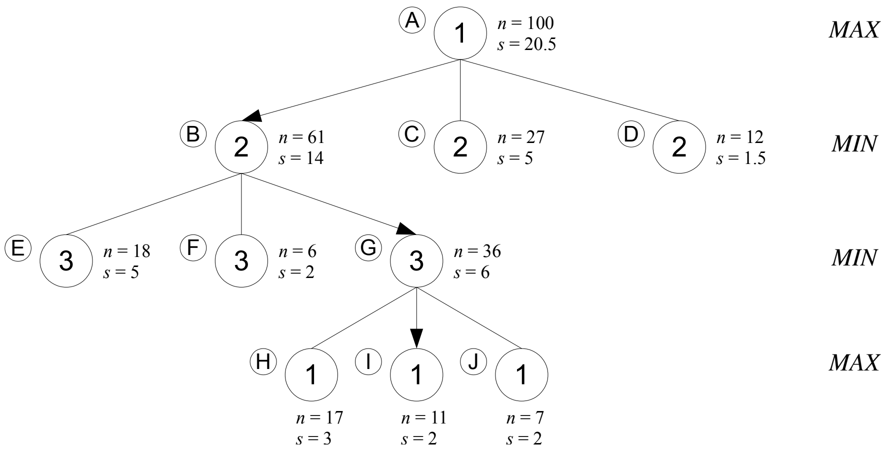


Figure 4.1: Example of an MCTS-paranoid tree.

In this iteration, the root player chooses node B. From this node, the first opponent can choose between nodes E, F and G. Because the paranoid search policy is applied, the modified UCT formula is applied to select the next child. The UCT values of the children of node B are calculated as follows.

$$v_E = (1 - \frac{5}{18}) + 0.2\sqrt{\frac{\ln 61}{18}} \approx 0.818$$

$$v_F = (1 - \frac{2}{6}) + 0.2\sqrt{\frac{\ln 61}{6}} \approx 0.832$$

$$v_G = (1 - \frac{6}{36}) + 0.2\sqrt{\frac{\ln 61}{36}} \approx 0.901$$

Even though the first opponent may have a higher win rate at node E or F, he chooses node G to try to minimize the win rate of the root player. After selecting node G, the UCT values for the second opponent are calculated in a similar way.

$$v_H = (1 - \frac{3}{17}) + 0.2\sqrt{\frac{\ln 36}{17}} \approx 0.915$$

$$v_I = (1 - \frac{2}{11}) + 0.2\sqrt{\frac{\ln 36}{11}} \approx 0.932$$

$$v_J = (1 - \frac{2}{7}) + 0.2\sqrt{\frac{\ln 36}{7}} \approx 0.857$$

For the second opponent, node I is chosen. If node I is fully expanded, then the next child is chosen using the standard UCT formula. Once a leaf node is reached, the playout is performed in the standard way. When the playout is finished, the value s of nodes A, B, G, and I is increased with the score of the root player and n is, similar to standard MCTS, increased by 1.

Equilibrium points

The essence of the paranoid policy is that a multi-player game is reduced to a two-player game (Sturtevant and Korf, 2000; Sturtevant, 2003a). All opponents are considered as one player, playing against the root player. Similarly, we may say that MCTS-paranoid is a reduction of multi-player MCTS to two players.

Kocsis, Szepesvári, and Willemson (2006) proved that, in a two-player game, MCTS with UCT is able to converge to the minimax tree, given sufficient time and memory. As such, the MCTS and the minimax tree have the same equilibrium. Because MCTS-paranoid is essentially a two-player search policy, it will converge to the corresponding minimax tree, namely the paranoid tree, as well.

In general, a paranoid tree is not able to find an equilibrium point of the game. It is, however, able to find a guaranteed lower bound on the root player's value. Similarly, MCTS-paranoid converges asymptotically to the guaranteed lower bound of the root player's value as well.

4.2.2 MCTS-BRS

With the BRS policy in MCTS, all opponents' layers are compressed into one layer, similar to BRS in the minimax framework. The standard UCT formula is applied in the root player's layers and the paranoid UCT formula is used in the opponents' layers.

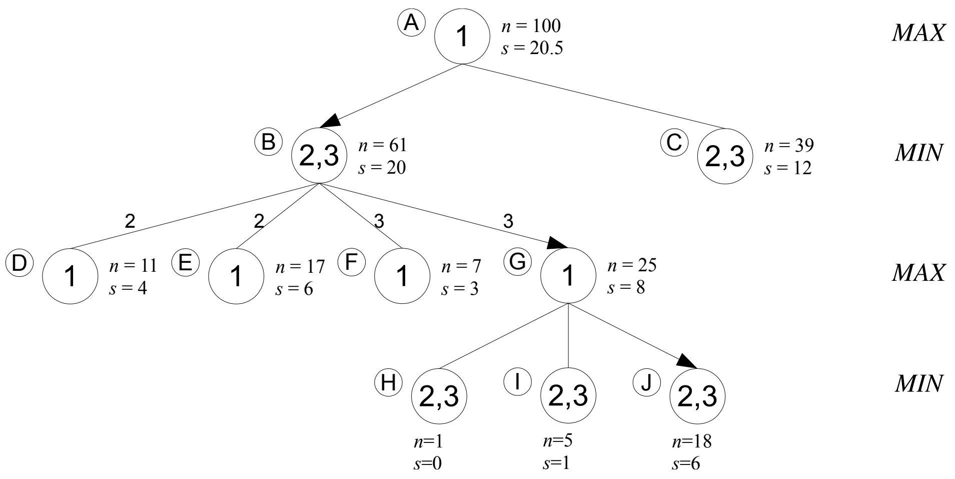


Figure 4.2: Example of an MCTS-BRS tree.

An example of an MCTS-BRS tree is given in Figure 4.2. Again, assume that $C = 0.2$. Similar to MCTS-paranoid and MCTS-max^n, the root player selects a move using the standard UCT formula as follows.

$$v_B = \frac{20}{61} + 0.2\sqrt{\frac{\ln 100}{61}} \approx 0.383$$

$$v_C = \frac{12}{39} + 0.2\sqrt{\frac{\ln 100}{39}} \approx 0.376$$

The root player selects node B. Next, the moves of the two opponents are compressed into one layer. Nodes D and E represent positions that are reached after moves by Player 2, i.e., the first opponent, and nodes E and F represent positions that are reached after moves by Player 3, i.e., the second opponent. The paranoid UCT formula is applied to select a move for one of the opponents as follows.

$$v_D = (1 - \frac{4}{11}) + 0.2\sqrt{\frac{\ln 61}{11}} \approx 0.759$$

$$v_E = (1 - \frac{6}{17}) + 0.2\sqrt{\frac{\ln 61}{17}} \approx 0.745$$

$$v_F = (1 - \frac{3}{7}) + 0.2\sqrt{\frac{\ln 61}{7}} \approx 0.725$$

$$v_G = (1 - \frac{8}{25}) + 0.2\sqrt{\frac{\ln 61}{25}} \approx 0.761$$

Node G is selected, so the first opponent skips a move and only the second opponent plays a move at this point. Next, the root player selects a move again using the standard UCT formula.

$$v_H = \frac{0}{1} + 0.2\sqrt{\frac{\ln 25}{1}} \approx 0.358$$

$$v_I = \frac{1}{5} + 0.2\sqrt{\frac{\ln 25}{5}} \approx 0.360$$

$$v_J = \frac{6}{18} + 0.2\sqrt{\frac{\ln 25}{18}} \approx 0.418$$

After selecting node J, this procedure continues until a leaf node is reached. Once a leaf node is reached, the tree is expanded and the playout is performed in the standard way. The result of the playout is backpropagated in a similar way as in MCTS-paranoid.

Equilibrium points

Similar to MCTS-paranoid, MCTS-BRS will, given sufficient time and memory, converge to the same value as the corresponding BRS tree. This means that, similar to BRS, MCTS-BRS does not find an equilibrium point or a guaranteed bound on one's score.

4.3 Experimental Results for Search Policies

In this section, we describe the experiments for the search policies and their results. First, Subsection 4.3.1 gives an overview of the experimental setup. Subsection 4.3.2 provides a comparison between the three minimax-based search techniques for multi-player games: max^n, paranoid, and BRS. Next, Subsection 4.3.3 investigates how the paranoid and BRS search policies in MCTS compare against the max^n policy in the MCTS framework. A comparison between the strongest minimax-based technique and the different MCTS-based techniques is given in Subsection 4.3.4. In the final set of experiments, the strongest MCTS-based technique is compared to the three minimax-based techniques in Subsection 4.3.5.

4.3.1 Experimental Setup

The experiments were run on a server consisting of AMD OpteronT 2.2 GHz processors. There are various ways to assign two or three types of players to the different seats in multi-player games (Sturtevant, 2003a). Table 4.1 shows in how many ways the player types can be assigned. Only the configurations where at least one instance of each player type is present are considered. There may be an advantage regarding the order of play and the number of instances of each player type. Therefore, each assignment is played multiple times until at least 1000 games are played and each assignment was played equally often. All experiments are performed with 250 ms, 1000 ms, and 5000 ms thinking time per move, unless stated otherwise. The results are given with a 95% confidence interval (Heinz, 2001).

Table 4.1 The number of ways two or three different player types can be assigned. The number between brackets is the number of games that are played per match.

Number of players	2 player types	3 player types
3	6 (1050)	6 (1050)
4	14 (1050)	36 (1044)
6	62 (1054)	540 (1080)

4.3.2 Comparison of Minimax-Based Techniques

Before testing the max^n, paranoid, and BRS search policies in MCTS, we first investigate how they perform when applied in the minimax framework. In the first set of experiments, we therefore match the three basic minimax-based players against each other. The win rates and the average search depths of the players in the different games are displayed in Table 4.2. In this series of experiments, we validate the results found by Schadd and Winands (2011), and extend the experiments with one more game (Blokus) and more variation in the number of players. An in-depth overview of the results is provided in Appendix B.1.

In Chinese Checkers, BRS is the best search technique by a considerable margin. In the variants with three, four, and six players, BRS wins between 67.7% and 86.6%

Table 4.2 Results of max^{n} versus paranoid versus BRS.

			Max^{n}		Paranoid		BRS	
Game	Players	Time (ms)	Win rate (%)	Depth (ply)	Win rate (%)	Depth (ply)	Win rate (%)	Depth (ply)
Chinese Checkers	3	250	1.1 ± 0.6	3.04	24.8 ± 2.6	4.44	74.1 ± 2.6	4.75
		1000	1.0 ± 0.6	3.41	20.5 ± 2.4	5.11	78.6 ± 2.5	5.44
		5000	1.4 ± 0.7	4.15	21.6 ± 2.5	5.75	76.9 ± 2.5	6.72
Chinese Checkers	4	250	5.3 ± 1.4	2.95	11.7 ± 1.9	3.52	83.0 ± 2.3	4.09
		1000	4.0 ± 1.2	3.57	23.0 ± 2.5	4.83	72.9 ± 2.7	5.04
		5000	5.7 ± 1.4	4.07	19.4 ± 2.4	5.43	74.8 ± 2.6	5.86
Chinese Checkers	6	250	15.0 ± 2.1	2.88	13.9 ± 2.1	3.34	71.1 ± 2.7	3.53
		1000	12.2 ± 2.0	3.85	13.1 ± 2.0	4.10	74.1 ± 2.6	4.74
		5000	16.9 ± 2.2	4.13	12.8 ± 2.0	4.59	69.5 ± 2.7	5.12
Focus	3	250	4.4 ± 1.2	3.58	35.7 ± 2.9	4.27	59.9 ± 3.0	4.34
		1000	3.8 ± 1.2	4.06	28.6 ± 2.7	4.88	67.6 ± 2.8	5.03
		5000	3.8 ± 1.2	4.63	27.3 ± 2.7	5.26	69.0 ± 2.8	5.93
Focus	4	250	9.4 ± 1.8	3.34	17.5 ± 2.3	3.55	73.1 ± 2.7	4.15
		1000	7.0 ± 1.5	3.81	24.0 ± 2.6	4.66	69.0 ± 2.8	4.86
		5000	6.7 ± 1.5	4.39	27.9 ± 2.7	5.23	65.4 ± 2.9	5.36
Rolit	3	250	8.1 ± 1.7	5.29	38.6 ± 2.9	5.72	53.3 ± 3.0	5.56
		1000	8.9 ± 1.7	6.12	39.7 ± 3.0	6.74	51.3 ± 3.0	6.65
		5000	6.3 ± 1.5	6.86	45.4 ± 3.0	7.88	48.4 ± 3.0	7.73
Rolit	4	250	15.9 ± 2.2	4.81	41.5 ± 3.0	5.48	42.5 ± 3.0	5.01
		1000	14.7 ± 2.1	5.48	42.7 ± 3.0	6.38	42.6 ± 3.0	5.90
		5000	14.9 ± 1.2	6.39	42.2 ± 3.0	7.28	42.9 ± 3.0	7.08
Blokus	4	250	17.8 ± 2.3	2.21	30.4 ± 2.8	3.11	51.8 ± 3.0	2.80
		1000	15.4 ± 2.2	2.66	29.6 ± 2.8	3.70	55.1 ± 3.0	3.65
		5000	8.6 ± 1.7	3.28	23.5 ± 2.6	4.32	68.0 ± 2.8	4.43

of the games with any time setting. In three-player and four-player Chinese Checkers, paranoid is significantly stronger than max^{n}. This is because paranoid can, on average, search more than 1 ply deeper and can therefore search a second layer of MAX nodes more often. In six-player Chinese Checkers, max^{n} is at least as strong as paranoid. In this variant, both max^{n} and paranoid usually do not reach a second layer of MAX nodes, as this requires a 7-ply search. BRS has a considerable advantage here, because this technique only requires a 3-ply search to reach a second layer of MAX nodes, which happens quite often. We note that for Chinese Checkers, max^{n} does not normalize the heuristic evaluation function and, as such, does not use shallow pruning. Empirical testing showed that this variant performs better in this game than the default approach, where the evaluation scores are normalized and shallow pruning is applied.

In Focus, again BRS is the best search technique and, similar to Chinese Checkers, reaches on average the highest search depth. Max^{n} performs quite poorly in Focus, where it never reaches a win rate of more than 10%.

In Rolit, the difference between BRS and paranoid is much smaller. In three-player Rolit, BRS is still significantly better than paranoid for short time settings, but with 5000 ms thinking time BRS and paranoid are equally strong. In the four-player variant, BRS and paranoid are on equal footing with any time setting. One of the possible reasons is that, contrary to Chinese Checkers and Focus, BRS does not reach a higher search depth than paranoid. This is true for all time settings.

Finally, in Blokus BRS achieves the highest win rate again. In this game the average search depth is lower than in the other games. This is because Blokus has, especially in the midgame, a high branching factor that can go up to more than 500 legal moves. Furthermore, because the board is large, computing the legal moves for a player is quite time-consuming, which reduces the number of nodes that are investigated per second.

General remarks

The results show that among the three tested search techniques, max^n performs the least. In every game with any number of players and time setting, max^n has a significantly lower win rate that both paranoid and BRS. The exception is six-player Chinese Checkers. Because there is not much pruning possible when using paranoid search, max^n plays at least as strong as paranoid. We remark that if better pruning techniques are applied for max^n, this search technique may perform better in other game variants as well. Max^n also plays relatively well in Blokus, where all players have difficulty reaching a decent search depth. Only the BRS player can regularly reach a second level of MAX nodes. In most games, BRS is the best search technique. Overall, the BRS players can search slightly deeper than the paranoid players. The exception is Rolit. In this game, the paranoid players can generally search slightly deeper and perform on a similar level as BRS. Overall, the experimental results are comparable with the results found by Sturtevant (2008b) and Schadd and Winands (2011).

4.3.3 Comparison of MCTS-Based Techniques

In the second set of experiments, the performance of the three different search policies in MCTS is tested. Each player uses a different policy: max^n (MCTS-max^n), paranoid (MCTS-paranoid), or BRS (MCTS-BRS). They are enhanced with ϵ-greedy playouts and Tree-Only Progressive History (see Section 5.2). The win rates and the median number of playouts per move are summarized in Table 4.3. An extensive overview of the results is given in Appendix B.2.

In Chinese Checkers, MCTS-max^n is, with any number of players and with any time setting, the strongest search technique. Overall, MCTS-BRS performs better than MCTS-paranoid. If more time is provided, MCTS-max^n performs relatively better than with lower time settings. With 250 ms thinking time, MCTS-max^n wins between 40.3% and 47.8% of the games, depending on the number of players. With 5000 ms of thinking time, the win rate increases to between 61.2% and 64.4%. MCTS-paranoid performs relatively worse with higher time settings, while MCTS-BRS remains stable. Furthermore, we note that there is overall no large difference between the median number of playouts per move between the different search policies. This

Table 4.3 Results of MCTS-max^n versus MCTS-paranoid versus MCTS-BRS.

			MCTS-max^n		MCTS-paranoid		MCTS-BRS	
Game	Players	Time (ms)	Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)
Chinese Checkers	3	250	40.2 ± 3.0	1,007	28.5 ± 2.7	1,003	31.3 ± 2.8	994
		1000	51.7 ± 3.0	4,318	19.9 ± 2.4	4,368	28.4 ± 2.7	4,257
		5000	62.1 ± 2.9	22,693	10.8 ± 1.9	22,765	27.0 ± 2.7	22,163
Chinese Checkers	4	250	47.8 ± 3.0	791	28.9 ± 2.7	786	23.3 ± 2.6	767
		1000	52.8 ± 3.0	3,520	19.0 ± 2.4	3,546	28.3 ± 2.7	3,396
		5000	64.4 ± 2.9	18,513	12.2 ± 2.0	18,921	23.5 ± 2.6	17,698
Chinese Checkers	6	250	46.9 ± 3.0	623	28.2 ± 2.7	635	24.8 ± 2.6	595
		1000	54.4 ± 3.0	2,792	20.7 ± 2.4	3,033	24.9 ± 2.6	2,725
		5000	61.2 ± 2.9	14,948	14.1 ± 2.1	18,787	24.7 ± 2.6	14,151
Focus	3	250	40.8 ± 3.0	1,629	29.1 ± 2.7	1,642	30.2 ± 2.8	1,609
		1000	42.9 ± 3.0	6,474	26.1 ± 2.7	6,668	31.0 ± 2.8	6,382
		5000	48.7 ± 3.0	32,987	19.6 ± 2.4	34,446	31.7 ± 2.8	31,990
Focus	4	250	37.3 ± 2.9	1,416	33.3 ± 2.9	1,405	29.4 ± 2.8	1,350
		1000	41.2 ± 3.0	6,310	26.1 ± 2.7	6,619	32.8 ± 2.8	5,945
		5000	52.3 ± 3.0	33,618	18.8 ± 2.4	37,693	28.9 ± 2.7	31,299
Rolit	3	250	50.6 ± 3.0	1,460	28.9 ± 2.7	1,465	20.5 ± 2.4	1,428
		1000	57.3 ± 3.0	5,933	24.6 ± 2.6	5,905	18.1 ± 2.3	5,787
		5000	63.2 ± 2.9	30,832	20.4 ± 2.4	30,019	16.4 ± 2.2	29,673
Rolit	4	250	43.6 ± 3.0	1,496	31.4 ± 2.8	1,497	25.0 ± 2.6	1,409
		1000	50.0 ± 3.0	6,064	27.5 ± 2.7	6,034	22.5 ± 2.5	5,651
		5000	56.5 ± 3.0	31,689	20.8 ± 2.5	30,977	22.7 ± 2.5	28,818
Blokus	4	250	36.7 ± 2.9	325	34.5 ± 2.9	320	28.8 ± 2.7	295
		1000	36.0 ± 2.9	1,406	35.3 ± 2.9	1,344	28.8 ± 2.7	1,231
		5000	33.6 ± 2.9	6,932	34.3 ± 2.9	6,824	32.0 ± 2.8	6,210

is not only true in Chinese Checkers, but also in the three other games. Although, in Chinese Checkers, if the number of players increases, the median number of playouts drops.

In Focus, MCTS-max^n is the best technique as well, though its win rate is generally lower than in Chinese Checkers. With 250 ms thinking time, it performs only slightly better than MCTS-BRS and MCTS-paranoid in the three- and four-player variants. Similar to Chinese Checkers, however, MCTS-max^n performs relatively better with higher time settings. Its win rate increases to around 50% with 5000 ms thinking time, while especially MCTS-paranoid performs worse with this time setting.

In Rolit, MCTS-max^n is again the strongest of the three variants. In the three-player variant of Rolit, MCTS-max^n appears to play relatively better than in the four-player variant, while MCTS-BRS appears to play relatively better in four-player Rolit. Similar to Chinese Checkers and Focus, the performance of MCTS-max^n increases with more thinking time, while the performance of MCTS-paranoid decreases.

Finally, in Blokus there is no clear winner. With 250 ms and 1000 ms of thinking time, MCTS-max^n and MCTS-paranoid are equally strong, with MCTS-BRS slightly behind. With 5000 ms thinking time, the three players are all on the same footing and there is no significant difference between the players. Similar to the results in the previous set of experiments, in Blokus the smallest number of positions is explored. Again, this is caused by the time-consuming generation of moves.

Experiments with Vanilla MCTS

Because ϵ-greedy playouts and Progressive History alter the selection and the playout phase of MCTS, we validate the previous experiments by rerunning them with Vanilla MCTS, i.e., with ϵ-greedy playouts and Progressive History disabled for all players. Only the experiments with 1000 ms of thinking time are repeated. The results are given in Table 4.4.

Table 4.4 Results of MCTS-max^n versus MCTS-paranoid versus MCTS-BRS without ϵ-greedy playouts and Progressive History.

		MCTS-max^n		MCTS-paranoid		MCTS-BRS	
Game	Players	Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)
Chinese Checkers	3	36.7 ± 2.9	1,709	30.7 ± 2.8	1,714	32.7 ± 2.8	1,702
	4	48.7 ± 3.0	2,412	23.7 ± 2.6	2,396	27.7 ± 2.7	2,369
	6	71.0 ± 2.7	6,470	8.1 ± 1.6	6,918	20.9 ± 2.5	6,182
Focus	3	24.5 ± 2.6	242	38.5 ± 2.9	242	37.0 ± 2.9	242
	4	29.0 ± 2.7	427	35.5 ± 2.9	426	35.4 ± 2.9	422
Rolit	3	48.5 ± 3.0	5,983	27.1 ± 2.7	6,022	24.4 ± 2.6	5,827
	4	49.1 ± 3.0	6,443	26.5 ± 2.7	6,473	24.3 ± 2.6	5,970
Blokus	4	36.0 ± 2.9	1,217	34.5 ± 2.9	1,114	29.5 ± 2.8	1,048

There are two striking results. First, the median number of playouts per move increases with the number of players in Chinese Checkers. This is in contrast with the results found in Table 4.3. This phenomenon is caused by the fact that the pieces move randomly on the board and that the game is finished when one of the home bases is filled. With more players, there are more home bases and more pieces on the board. As a result it takes, on average, fewer moves before one of the home bases is filled. Second, MCTS-max^n is outperformed by both MCTS-paranoid and MCTS-BRS in Focus. This may be caused by the low number of playouts per move. Because the moves in the playouts are chosen randomly, games can take a long time to finish. This result is in accordance with the results in Subsection 4.3.3, where we found that MCTS-paranoid performs relatively better and MCTS-max^n performs relatively worse if the number of playouts is lower. Also in Chinese Checkers, playouts take much longer than with ϵ-greedy playouts, except in the six-player variant.

In Rolit and Blokus, the average length of the playouts is similar to the previous set of experiments. This is because the length of these games is not dependent on the strategy of the players. A game of Rolit always takes 60 turns, and a game of Blokus never takes more than 84 turns. This may explain why the results in this set of experiments are comparable to those in Table 4.3.

General remarks

Overall, the results reveal that MCTS clearly performs best using the standard max^n search policy. Only in Blokus, MCTS-max^n is not significantly stronger than MCTS-paranoid and MCTS-BRS. Without ϵ-greedy playouts and Progressive History, MCTS-max^n is outperformed by MCTS-paranoid and MCTS-BRS in Focus.

This is different to the minimax framework, where paranoid and BRS significantly outperform max^n. There are two main reasons for this difference. First, paranoid and BRS perform well in the minimax framework because they increase the amount of pruning. Because $\alpha\beta$ pruning does not occur in MCTS, this advantage is non-existent in the MCTS framework. Second, in the minimax framework, BRS reduces the *horizon effect*. It allows more planning ahead because more layers of MAX nodes are investigated. In MCTS, the advantage of having more layers of MAX nodes in the search tree is considerably smaller. The horizon effect in MCTS is already diminished due to the playouts. An additional problem with MCTS-BRS is that, in the tree, invalid positions are investigated, which may reduce the reliability of the playouts.

The results also show that MCTS-max^n performs relatively better than the other two techniques if more time is provided. Especially MCTS-paranoid performs relatively worse with more thinking time. The reason for this may be that the paranoid assumption causes the player to become too paranoid with larger search depths, similar to paranoid in the minimax framework. In Blokus, the performance of the three different players is stable with different time settings. Finally, the results reveal that there is overall no large difference in the median number of playouts between the different players. This indicates that the different search policies do not produce a significantly different amount of overhead.

4.3.4 MCTS-Based Techniques versus BRS

The experiments in Subsection 4.3.3 revealed that MCTS-max^n is the best among the different MCTS variants. In the next series of experiments, this result is validated by comparing the three different search policies in MCTS against the best minimax-based search technique, BRS (cf. Subsection 4.3.2). The results are displayed in Table 4.5. The percentages indicate the win rate of each of the players against BRS. An in-depth overview of the results of the matches between MCTS-max^n and BRS is provided in Appendix B.3.

In Chinese Checkers, the win rate of the MCTS players strongly depends on the thinking time. If 250 ms per move are provided, MCTS-max^n wins between 18.4% and 33.3% of the games against BRS, dependent on the number of players. With 5000 ms thinking time, the win rate lies between 68.2% and 88.1% against BRS. MCTS-paranoid and MCTS-BRS win significantly fewer games against BRS, which indicates that MCTS-max^n is a stronger player than MCTS-paranoid and MCTS-BRS. This is in accordance with the results found in Subsection 4.3.3.

In Focus, similar results are observed. With a lower time setting, all MCTS-based opponents are significantly outperformed by BRS, while with 5000 ms of thinking time per move, the win rate increases to between 55% and 70% for MCTS-max^n and MCTS-BRS. MCTS-paranoid wins around or less than 40% of the games against BRS with most time settings.

Table 4.5 Win rates of the different MCTS-based techniques against BRS.

Game	Players	Time (ms)	MCTS-max^n win rate (%)	MCTS-paranoid win rate (%)	MCTS-BRS win rate (%)
		250	18.4 ± 2.3	15.0 ± 2.2	14.6 ± 2.1
Chinese Checkers	3	1000	42.4 ± 3.0	29.4 ± 2.6	35.5 ± 2.9
		5000	68.2 ± 2.8	29.2 ± 2.8	50.0 ± 3.0
		250	24.5 ± 2.6	16.7 ± 2.3	18.1 ± 2.3
Chinese Checkers	4	1000	57.7 ± 3.0	45.5 ± 3.0	48.0 ± 3.0
		5000	77.6 ± 2.5	47.1 ± 3.0	65.8 ± 2.9
		250	33.3 ± 2.8	25.5 ± 2.6	24.1 ± 2.6
Chinese Checkers	6	1000	72.1 ± 2.7	56.4 ± 3.0	64.5 ± 2.9
		5000	88.1 ± 2.0	73.3 ± 2.7	83.8 ± 2.2
		250	37.1 ± 2.9	32.2 ± 2.8	34.2 ± 2.9
Focus	3	1000	53.8 ± 3.0	37.7 ± 2.9	48.1 ± 3.0
		5000	62.9 ± 2.9	34.5 ± 2.9	54.7 ± 3.0
		250	42.3 ± 3.0	37.0 ± 2.9	39.6 ± 3.0
Focus	4	1000	54.3 ± 3.0	39.7 ± 3.0	50.5 ± 3.0
		5000	68.6 ± 2.8	42.8 ± 3.0	61.3 ± 2.9
		250	74.1 ± 2.6	65.3 ± 2.9	58.6 ± 3.0
Rolit	3	1000	84.6 ± 2.2	69.8 ± 2.8	68.0 ± 2.8
		5000	87.0 ± 2.0	68.7 ± 2.8	69.0 ± 2.8
		250	71.2 ± 2.7	66.6 ± 2.9	60.9 ± 3.0
Rolit	4	1000	80.2 ± 2.4	66.0 ± 2.9	64.5 ± 2.9
		5000	82.0 ± 2.3	64.0 ± 2.9	67.2 ± 2.8
		250	57.8 ± 3.0	56.2 ± 3.0	57.5 ± 3.0
Blokus	4	1000	77.4 ± 2.5	80.9 ± 2.4	79.9 ± 2.4
		5000	90.5 ± 1.8	89.1 ± 1.9	88.0 ± 2.0

In Rolit, the MCTS-based players perform well compared to BRS. In both the three- and four-player variant, MCTS-max^n wins more than 70% of the games with any time setting against BRS. Also MCTS-paranoid and MCTS-BRS win significantly more than 60% of the games against BRS. This again shows that Rolit is a difficult domain for BRS.

Finally, in Blokus, BRS is outperformed by the MCTS-based players as well. This is likely because BRS can only reach a limited search depth because of the high branching factor. The win rate of the three different MCTS players is similar, which again shows that the three different MCTS-based players are on equal footing in Blokus.

General remarks

The results in Table 4.5 show that MCTS-max^n is the strongest player against BRS. This result is in accordance with the results in Subsection 4.3.3. MCTS-paranoid and MCTS-BRS achieve a significantly lower win rate against BRS, except in Blokus. When comparing BRS to MCTS-max^n, for the low time settings BRS significantly outperforms MCTS-max^n in Focus and Chinese Checkers, while MCTS-max^n is stronger in Blokus and Rolit. With a higher time setting, MCTS-max^n becomes stronger than BRS in all games. This is not true for MCTS-paranoid, which performs worse than BRS in the three-player and four-player variants of Chinese Checkers and Focus, even with high time settings. Similar to the results found in Subsection 4.3.3, MCTS-paranoid does not benefit much from reaching larger search depths. MCTS-BRS does, however, benefit from higher time settings. With 5000 ms of thinking time per move, it outperforms BRS in all game variants, except in three-player Chinese Checkers, where the two players are equally strong.

4.3.5 Minimax-Based Techniques versus MCTS-max^n

In the next set of experiments we test the performance of the three minimax-based techniques against the strongest MCTS-based technique, MCTS-max^n. The win rates of max^n, paranoid, and BRS against MCTS-max^n are given in Table 4.6. We note that the win rates in the column 'BRS' are the inverse of the win rates under 'MCTS-max^n' in Table 4.5, as these two columns both show the results of the matches between MCTS-max^n and BRS.

Table 4.6 Win rates of the minimax-based techniques against MCTS-max^n.

Game	Players	Time (ms)	Max^n win rate (%)	Paranoid win rate (%)	BRS win rate (%)
Chinese Checkers	3	250	20.8±2.5	57.7±3.0	81.6±2.3
	3	1000	4.0±1.2	22.6±2.5	57.6±3.0
	3	5000	1.5±0.7	9.8±1.8	31.8±2.8
Chinese Checkers	4	250	33.2±2.8	21.3±2.5	75.5±2.6
	4	1000	6.7±1.5	12.6±2.0	42.3±3.0
	4	5000	3.0±1.0	3.9±1.2	22.4±2.5
Chinese Checkers	6	250	36.2±2.9	24.6±2.6	66.7±2.8
	6	1000	9.3±1.8	4.5±1.3	29.9±2.7
	6	5000	4.6±1.3	4.4±1.2	11.9±2.0
Focus	3	250	16.7±2.3	50.3±3.0	62.9±2.9
	3	1000	8.9±1.7	31.0±2.8	46.2±3.0
	3	5000	5.7±1.4	24.5±2.6	35.0±2.9
Focus	4	250	23.9±2.6	30.8±2.8	57.7±3.0
	4	1000	15.6±2.2	27.4±2.7	45.7±3.0
	4	5000	9.0±1.7	18.4±2.3	31.4±2.8
Rolit	3	250	9.2±1.7	31.4±2.8	25.9±2.6
	3	1000	5.4±1.4	20.7±2.5	15.4±2.2
	3	5000	4.4±1.2	16.7±2.3	13.0±2.0
Rolit	4	250	20.1±2.4	29.3±2.8	28.8±2.7
	4	1000	13.0±2.0	26.1±2.7	19.8±2.4
	4	5000	11.1±1.9	21.0±2.5	18.0±2.3
Blokus	4	250	23.5±2.6	32.4±2.8	42.2±3.0
	4	1000	5.9±1.4	10.6±1.9	22.6±2.5
	4	5000	1.2±0.7	2.1±0.9	9.5±1.8

In Chinese Checkers, max^n and paranoid are much weaker than MCTS-max^n. This result was also found by Sturtevant (2008b). BRS wins more games against MCTS-max^n than max^n and paranoid. This validates the results presented in Table 4.2. Similar to the results found in Subsection 4.3.4, MCTS-max^n performs relatively better with higher time settings. The win rate of the minimax-based players drops as the players receive more thinking time. This is not only true for BRS, but also for max^n and paranoid. Similar to the experiments in Subsection 4.3.2, max^n does not apply normalization of the heuristic evaluation function and shallow pruning in Chinese Checkers in this set of experiments.

In Focus, the performance of the minimax-based techniques against MCTS-max^n decreases if more time is provided, as well. BRS wins approximately 60% of the games with a low time setting, but its win rate drops to between 30% and 35% with 5000 ms of thinking time. Paranoid is on equal footing with MCTS-max^n in three-player Focus with a low time setting, but if more time is provided, the MCTS-based player performs significantly better. Max^n wins less than 25% of the games against MCTS-max^n with any time setting and any number of players.

In Rolit, the three different minimax-based players win around or less than 30% of the games against MCTS-max^n. Paranoid wins slightly more games than BRS against MCTS-max^n, which validates that paranoid is at least as strong as BRS in Rolit. In Subsections 4.3.2 and 4.3.4 we found that both paranoid and MCTS-max^n perform at least as well as, or better than, BRS in Rolit. When comparing paranoid to MCTS-max^n, we find that the MCTS-based player performs best. Paranoid wins around or less than 30% of the games against MCTS-max^n with any number of players or time setting.

Finally, in Blokus, all minimax-based players are outperformed by MCTS-max^n for each time setting. In Subsection 4.3.2 we found that BRS is the strongest and max^n is the weakest minimax technique in Blokus. The results in Table 4.6 reveal a similar result.

General remarks

These experiments confirm the results found in Subsection 4.3.2. Max^n achieves the lowest win rate against MCTS-max^n, showing that max^n is the weakest minimax-based search technique. The highest win rate is achieved by BRS, except in Rolit. In Rolit, paranoid has a slightly higher win percentage than BRS against the MCTS-max^n player, which is comparable to the results in Subsection 4.3.2, where we observed that paranoid and BRS perform on a similar level. Furthermore, the results reveal that all three players perform worse against MCTS-max^n if more time is provided. A similar result was found in Subsection 4.3.4, where the performance of the MCTS-based search techniques increases against BRS if the amount of thinking time is increased.

4.4 Background of MCTS-Solver

The MCTS variants described in the previous sections are not able to solve positions. Winands *et al.* (2008) proposed a new MCTS variant for two-player games, called *MCTS-Solver*, which has been designed to play narrow tactical lines better in sudden-death games. A sudden-death game is a game that may end abruptly by the creation of one of a prespecified set of patterns (Allis, 1994). The variant differs from the traditional MCTS in respect to backpropagation and selection strategy. It is able to prove the game-theoretic value of a position given sufficient time.

In addition to backpropagating the values $\{0, \frac{1}{2}, 1\}$, representing a loss, a draw, and a win respectively, the search also backpropagates the values ∞ and $-\infty$, which are assigned to a proven won or lost position, respectively. To prove that a node is a win, it is sufficient to prove that at least one of the children is a win. In order to prove that a node is a loss, it is necessary to prove that all children lead to a loss. If at least one of the children is not a proven loss, then the current node cannot be proven.

Experiments showed that for the sudden-death game Lines of Action (LOA), an MCTS program using MCTS-Solver defeats a program using MCTS by a winning percentage of 65% (Winands *et al.*, 2008). Moreover, MCTS-Solver performs much better than a program using MCTS against the world-class $\alpha\beta$-based program MIA. They concluded that MCTS-Solver constitutes genuine progress in solving and playing strength in sudden-death games, significantly improving upon MCTS-based programs. The MCTS-Solver has also been successfully applied in games such as Hex (Arneson *et al.*, 2010; Cazenave and Saffidine, 2010), Shogi (Sato, Takahashi, and Grimbergen, 2010), Twixt (Steinhauer, 2010), Tron (Den Teuling and Winands, 2012), and Breakthrough (Lorentz and Horey, 2013).

Cazenave and Saffidine (2011) proposed to improve the MCTS-Solver using *Score Bounded Monte-Carlo Tree Search* when a game has more than two outcomes, for example in games that can end in draw positions. It significantly improved the MCTS-Solver by taking into account bounds on the possible scores of a node in order to select the nodes to explore. They applied this variant for solving Seki in the game of Go and for solving small boards in Connect Four. Score Bounded Monte-Carlo Tree Search has also been applied in simultaneous move games (Finnsson, 2012).

4.5 Multi-Player MCTS-Solver

The previous experiments revealed that MCTS-max^n is the strongest multi-player MCTS variant. Therefore, for MCTS-max^n, we propose a multi-player variant of MCTS-Solver, called *Multi-Player MCTS-Solver* (MP-MCTS-Solver). For the multi-player variant, MCTS-Solver has to be modified, in order to accommodate for games with more than two players. This is discussed below.

Proving a win works similarly as in the two-player version of MCTS-Solver: if at one of the children a win is found for the player who has to move in the current node, then this node is a win for this player. If all children lead to a win for the same opponent, then the current node is also labeled as a win for this opponent. However, if the children lead to wins for different opponents, then updating the game-theoretic values becomes a non-trivial task.

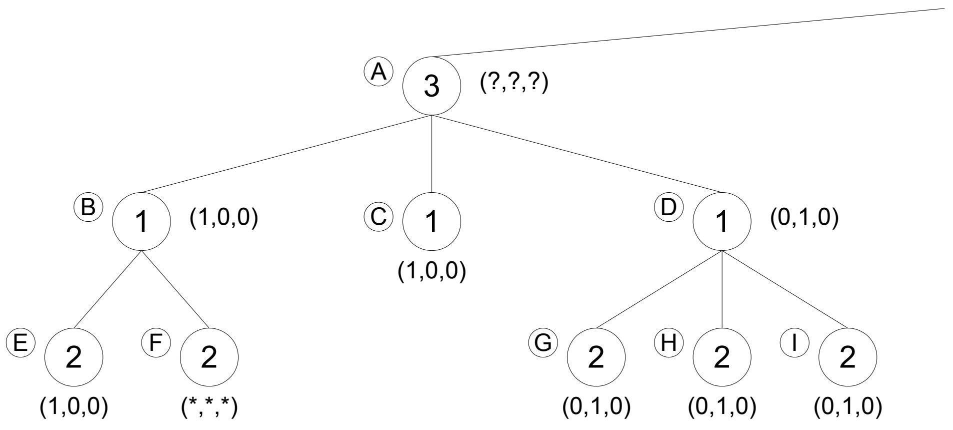


Figure 4.3: Example of backpropagating game-theoretic values in a multi-player search tree.

An example is given in Figure 4.3. Here, node E is a terminal node where Player 1 has won. This means that node B is a mate-in-one for Player 1, regardless of the value of node F. Node E is marked as solved and receives a game-theoretic value of $(1,0,0)$. Nodes G, H, and I all result in wins for Player 2. Parent node D receives a game-theoretic value of $(0,1,0)$, because this node always leads to a win for the same opponent of Player 1. The game-theoretic value of node A cannot be determined in this case, because both Player 1 and Player 2 are able to win and there is no win for Player 3. In this case, Player 3 is a *kingmaker*, i.e., a player who determines who will win, without being able to win himself. This makes him unpredictable.

Update rules have to be developed to take care of such situations. We propose three different update rules that are briefly explained below.

(1) The *standard* update rule only updates proven wins for the same opponent. This means that only if all children lead to a win for the same opponent, then the current node is also set to a win for this opponent. Otherwise, the node is not marked as solved and the UCT value is used. A disadvantage of the standard update rule is that it is quite conservative. We define two update rules that allow solving nodes that lead to different winners.

(2) The *paranoid* update rule uses the assumption that the opponents of the root player will never let him win. This rule is inspired by the paranoid tie-breaker rule for max^n (Sturtevant, 2003a). Again consider Figure 4.3. Assuming that the root player is Player 1, using the paranoid update rule, we can determine the game-theoretic value of node A. Because we assume that Player 3 will not let Player 1 win, the game-theoretic value of node A becomes $(0,1,0)$. If there are still multiple winners after removing the root player from the list of possible winners, then no game-theoretic value is assigned to the node.

The paranoid update rule may not always give the desired result. With the paranoid assumption, the game-theoretic value of node A is $(0,1,0)$ (i.e., a win for Player 2). This is actually not certain, because it is also possible that Player 3 will let Player 1 win. However, because the game-theoretic value of node A denotes a win for Player

2, and at the parent of node A Player 2 is to move, the parent of node A will also receive a game-theoretic value of $(0,1,0)$. This is actually incorrect, since choosing node A does not give Player 2 a guaranteed win.

Problems may thus arise when a player in a given node gives the win to the player directly preceding him. In such a case, the parent node will receive a game-theoretic value, which is technically not correct. This problem can be diminished by using (3) the *first-winner* update rule. When using this update rule, the player will give the win to the player who is the first winner after him. In this way the player before him does not get the win and, as a result, does not overestimate the position. When using the first-winner update rule, in Figure 4.3, node A will receive the game-theoretic value $(1,0,0)$.

Overestimation

Overestimation of a node is a phenomenon that occurs if one or more children of a node are proven to be a loss, but the node itself is not solved (yet). Winands, Björnsson, and Saito (2010) provided a case where overestimation may lead to wrong evaluations and showed how to tackle this problem by applying a threshold. If the number of visits at a node is less than the threshold, the playout strategy is used to select a node. In this way, children that are proven to be a loss can be selected, as long as the number of visits is below the threshold. In MAGE, the UCT formula is applied to value solved children if a node is not proven. For the win rate $\bar{x}_i$, the game-theoretic value of the child is used, which is usually 0.[1] Overestimation is abated by occasionally selecting nodes that are a proven loss, but because the win rate is 0, non-proven nodes are favored.

4.6 Experimental Results for Multi-Player MCTS-Solver

In this section, we give the results of MP-MCTS-Solver with the three different update rules playing against an MCTS player without MP-MCTS-Solver. These experiments are only performed in Focus, because MCTS-Solver is only successful in sudden-death games (Winands *et al.*, 2008). Chinese Checkers, Rolit, and Blokus do not belong to this category of games, and therefore MP-MCTS-Solver will not work well in these games. Focus, however, is a sudden-death game and is therefore an appropriate test domain for MP-MCTS-Solver.

The results in this section are performed with different hardware and time settings than in the previous sections. The experiments are run on a cluster consisting of AMD64 Opteron 2.4 GHz processors. Each player receives 2500 ms of thinking time to determine the move to play. The win rates are based on 3360 games, where each configuration of player types is played equally often.

In Table 4.7, we see that the standard update rule works well in Focus. The win rates for the different number of players vary between 53% and 55%. The other

[1] If draws are allowed, the game-theoretic value may be non-zero. We remind the reader that, in Focus, draws do not occur.

Table 4.7 Win rates of MP-MCTS-Solver with different update rules against default MCTS in Focus.

Type	2 players win rate (%)	3 players win rate (%)	4 players win rate (%)
Standard	53.0 ± 1.7	54.9 ± 1.7	53.3 ± 1.7
Paranoid	51.9 ± 1.7	50.4 ± 1.7	44.9 ± 1.7
First-winner	52.8 ± 1.7	51.5 ± 1.7	43.4 ± 1.7

update rules do not perform as well. For the two-player variant, they behave and perform similar to the standard update rule. The win rates are slightly lower, which may be caused by statistical noise and a small amount of overhead. In the three-player variant, MP-MCTS-Solver neither increases nor decreases the performance significantly. In the four-player variant, the win rate of the player using MP-MCTS-Solver is well below 50% for the paranoid and the first-winner update rules. Based on these results we may conclude that only the standard update rule works well.

4.7 Chapter Conclusions and Future Research

Among the three minimax-based search techniques we tested, BRS turns out to be the strongest one. Overall, it reaches the highest search depth and, because of its tree structure, more MAX nodes are investigated than in paranoid and max^n. BRS significantly outperforms max^n and paranoid in Chinese Checkers, Focus, and Blokus. Only in Rolit, paranoid performs at least as strong as BRS.

In the MCTS framework, the max^n search policy appears to perform the best. The advantages of paranoid and BRS in the minimax framework do not apply in MCTS, because $\alpha\beta$ pruning is not applicable in MCTS. An additional problem with MCTS-BRS may be that, in the tree, invalid positions are investigated, which may reduce the reliability of the playouts as well. Still, MCTS-paranoid and MCTS-BRS overall achieve decent win rates against MCTS-max^n, especially with lower time settings. Furthermore, MCTS-paranoid is on equal footing with MCTS-max^n in Blokus and, in the vanilla version of MCTS, MCTS-paranoid and MCTS-BRS significantly outperform MCTS-max^n in Focus. Based on the results, we may conclude that the max^n search policy in MCTS is the most robust, although the BRS and paranoid search policies can still be competitive.

In a comparison between MCTS-max^n and BRS, MCTS-max^n overall wins more games than BRS. In Chinese Checkers and Focus, BRS is considerably stronger with lower time settings, while in Rolit and Blokus MCTS-max^n significantly outperforms BRS. With higher time settings, MCTS-max^n outperforms BRS in all games with any number of players. From this we may conclude that with higher time settings, the MCTS-based player performs relatively better.

Finally, we proposed MP-MCTS-Solver in MCTS-max^n with three different update rules, namely (1) standard, (2) paranoid, and (3) first-winner. This variant is able to prove the game-theoretic value of a position. We tested this variant only in Focus, because MP-MCTS-Solver can only work well in sudden-death games. A win rate between 53% and 55% was achieved in Focus with the standard update rule. The other two update rules achieved a win rate up to 53% in the two-player variant, but were around or below 50% for the three- and four-player variants. We may conclude that MP-MCTS-Solver performs well with the standard update rule. The other two update rules, paranoid and first-winner, were not successful in Focus.

In this chapter we investigated three search policies for multi-player games, i.e., max^n, paranoid, and BRS, in the MCTS framework. We did not consider policies derived from these techniques, such as the Coalition-Mixer (Lorenz and Tscheuschner, 2006) or MP-Mix (Zuckerman, Felner, and Kraus, 2009). They use a combination of max^n and (variations of) paranoid search. They also have numerous parameters that have to be optimized. Tuning and testing such policies for multi-player MCTS is a first direction of future research.

A second possible future research direction is the application of BRS variants as proposed by Esser *et al.* (2013) in MCTS. The basic idea is that, besides the opponent with the best counter move, the other opponents are allowed to perform a move as well. These moves are selected using static move ordering. The advantage of these variants is that no invalid positions are searched, while maintaining the advantages of the original BRS technique.

A third future research topic is the application of paranoid and BRS policies in the playouts of MCTS. Cazenave (2008) applied paranoid playouts to multi-player Go and found promising results. BRS may be able to shorten the playouts, because all but one opponents skip their turn. This may increase the number of playouts per second, and thus increase the playing strength. Applying paranoid and BRS playouts requires developing and implementing paranoid moves for the opponents.

MP-MCTS-Solver has proven to be a genuine improvement for the sudden-death game Focus, though more research is necessary to improve its performance. As a fourth direction of future research, it may be interesting to investigate the performance of MP-MCTS-Solver in different sudden-death multi-player games, such as a multi-player variant of Tron. Furthermore, MP-MCTS-Solver is currently only applied in MCTS-max^n. It may be interesting to apply it to MCTS-paranoid and MCTS-BRS as well.

CHAPTER 5

Progressive History for MCTS

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

1. Nijssen, J.A.M. and Winands, M.H.M. (2011a). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 238–249, Springer, Berlin, Germany.
2. Nijssen, J.A.M. and Winands, M.H.M. (2010). Enhancements for Multi-Player Monte-Carlo Tree Search. *Proceedings of the 22nd Benelux Conference on Artificial Intelligence*, Luxembourg City, Luxembourg. Extended abstract.

In the selection phase of MCTS, nodes are chosen using a *selection strategy* until a leaf node is reached. A commonly used selection strategy is *Upper Confidence bounds applied to Trees* (UCT) (Kocsis and Szepesvári, 2006). This technique is based on UCB1 (Auer *et al.*, 2002). When nodes are visited for the first time or if the number of visits is still small, UCT gives unreliable results. Several techniques have been developed to solve this problem. One of such techniques is applying a threshold T (Coulom, 2007a). If the number of visits at a node is fewer than T, instead of UCT, the playout strategy is used to select a child. Another direction is the application of Rapid Action-Value Estimation (RAVE). The first RAVE techniques were introduced by Gelly and Silver (2007). The goal of RAVE is to increase the amount of usable information for the selection strategy if the number of visits at a node is small. RAVE uses AMAF (all-moves-as-first) values to increase the amount of usable data.

This chapter answers the second research question by introducing a new domain-independent selection strategy for MCTS, namely *Progressive History*. It is a combination of Progressive Bias (Chaslot *et al.*, 2008b) and the relative history heuristic (Schaeffer, 1983; Winands *et al.*, 2006). This enhancement is tested in two-player and multi-player variants of Chinese Checkers, Focus, Rolit, and Blokus.

The remainder of this chapter is structured as follows. First, Subsection 5.1 presents previous research related to this chapter. Section 5.2 introduces the new selection strategy called Progressive History. The experiments and the results are given in Section 5.3. Section 5.4 provides a brief overview of the research performed by Fossel (2010), who compared Progressive History to RAVE in the two-player game Havannah. Finally, Section 5.5 gives the conclusions that can be drawn from the research presented in this chapter and provides an outlook on future research.

5.1 Related Work

This section gives a brief overview of the work related to this chapter. First, Subsection 5.1.1 explains the domain-independent selection strategy called Rapid Action-Value Estimation (RAVE). Next, Subsection 5.1.2 discusses Progressive Bias.

5.1.1 Rapid Action-Value Estimation

Rapid Action-Value Estimation (RAVE) (Gelly and Silver, 2007) is a domain-independent technique used to improve the selection strategy when the number of playouts is small. RAVE uses the idea known as the *all-moves-as-first* (AMAF) heuristic (Brügmann, 1993). Suppose that, from a node p, we use the selection strategy to select a child i by playing a move m. With UCT, when backpropagating the result of the playout, only the value of node i is updated. With AMAF, for each move m' performed after move m by the same player, the sibling of i corresponding to move m' is updated with the result of the playout as well (Helmbold and Parker-Wood, 2009). Each node keeps track of both the UCT and the AMAF values.

An example is provided in Figure 5.1. In this example, the selection strategy chooses nodes A, C, and D, corresponding to moves q, r, and s. In the backpropagation phase, the UCT and AMAF values of these nodes are updated. Assuming that moves q and s are performed by the same player, the AMAF value of node B is updated as well.

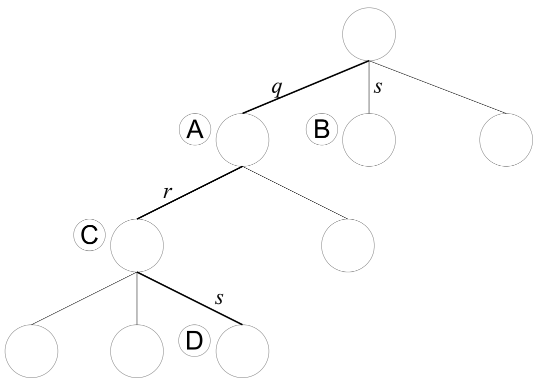


Figure 5.1: Example of an MCTS tree with UCT and AMAF updates.

Gelly and Silver (2007; 2011) introduced RAVE in the computer Go program MoGo. They incorporated the AMAF heuristic in MCTS by replacing the UCT formula (Formula 2.8) with Formula 5.1. Similar to standard UCT, the child i with the highest value v_i of parent p is chosen.

$$\begin{aligned} v_i &= \beta \times v_{RAVE} + (1-\beta) \times v_{UCT} \\ v_{RAVE} &= AMAF_a + C \times \sqrt{\frac{\ln(m_{children})}{m_a}} \\ v_{UCT} &= \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} \end{aligned} \tag{5.1}$$

In this formula, m_a indicates the number of playouts in which move a was played at any point after node p. $AMAF_a$ is the AMAF value of move a, i.e., the average score of all playouts in which move a was played after node p. $m_{children}$ is the sum of m_a of all children of p. β is a factor which decays over time. Gelly and Silver applied $\beta = \sqrt{\frac{k}{3n_p+k}}$, where k is a constant which indicates the number of playouts where the influence of the AMAF and UCT factors becomes equal.

There exist various implementations of RAVE in different games. Teytaud and Teytaud (2010a) and Rimmel, Teytaud, and Teytaud (2011a) applied RAVE in the game Havannah. Teytaud and Teytaud (2010a) showed that the efficiency of this technique increases with a larger board size, but decreases when the number of playouts per move is increased. Overall, they found that RAVE is a significant improvement in Havannah. Tom and Müller (2010) applied RAVE in the artificial game called Sum of Switches. This game acts as a best case for this selection strategy, because the relative value between moves is consistent at all stages of the game. They showed that, with RAVE, the optimal move was found much faster than with UCT. Cazenave and Saffidine (2010) applied RAVE in the Hex program YOPT, for which it was a significant improvement over regular UCT.

With some games or game problems, however, RAVE does not seem to perform well. For instance, Zhao and Müller (2008) observed that RAVE does not work for solving local problems in Go. Sturtevant (2008b) applied it in Chinese Checkers, but reported that it did not perform well in this game. Finnsson (2012) tested RAVE in General Game Playing and found that, overall, it performs better than regular UCT in two-player domains. However, in the three-player variants of Chinese Checkers and TTCC4, RAVE did not perform better than UCT.

There also exist various variations on RAVE. Lorentz (2011) proposed a variation called *killer RAVE*, in which only the most important moves are used for updating the RAVE values. In the game of Havannah, killer RAVE turned out to be a significant improvement. While trying to develop a more robust version of RAVE, Tom and Müller (2011) developed *RAVE-max* and a stochastic variant *δ-RAVE-max*. These variants allowed for correcting underestimation of RAVE in the artificial game Sum of Switches, but were not successful in Go. Hoock *et al.* (2010) introduced a RAVE variant called *poolRave*. This variant first builds a pool of k best moves according to the RAVE values. Subsequently it chooses one move from the pool, which is played by a certain probability. Otherwise, the default selection strategy is used. PoolRave was tested in Havannah and the Go program MOGO. This enhancement provided a significant improvement over the standard selection strategy without the poolRave modification in both games, especially if domain knowledge was small or absent.

5.1.2 Progressive Bias

The goal of *Progressive Bias* (Chaslot *et al.*, 2008b) is to direct the search using, possibly time-expensive, heuristic domain knowledge. The disadvantage of Progressive Bias is that it requires domain knowledge, while RAVE is domain independent. The advantage of Progressive Bias is that, even without any playouts at a particular node, the search can already be guided in a promising direction.

The influence of Progressive Bias is high when the number of simulations is small, but decreases when more simulations are played to ensure that the strategy converges to a pure selection strategy like UCT. To achieve this, Formula 2.8 is modified to incorporate the domain knowledge as shown in Formula 5.2.

$$v_i = \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} + \frac{H_i}{n_i + 1} \tag{5.2}$$

Similar to the UCT formula, $\bar{x}_i$ denotes the win rate of node i. n_i and n_p indicate the total number of times child i and its parent p have been visited, respectively. C is a constant, which balances exploration and exploitation. H_i is a heuristic value based on domain knowledge. The board position corresponding to node i is evaluated and a value is assigned to the current player.

Progressive Bias was originally used in the Go program MANGO, where this technique increased the playing strength of this program significantly (Chaslot *et al.*, 2008b). It was also, in an adaptive form, applied in MOGO (Chaslot *et al.*, 2010), which was the first program to defeat a professional 9-dan Go player with a 7-stone handicap on the 19×19 board. Progressive Bias has been used in ERICA as well (Huang, Coulom, and Lin, 2011), which won the Go tournament of the 15th Computer Olympiad in 2010. Besides being used in Go, Progressive Bias was also applied in the Lines of Action program MC-LOA by Winands *et al.* (2010) and in the Hex program MoHex 2.0 by Huang *et al.* (2013), which both won their respective tournament at the 17th Computer Olympiad.

5.2 Progressive History

A disadvantage of Progressive Bias is that heuristic knowledge is required. A solution is offered by using the (relative) history heuristic (Schaeffer, 1983; Winands *et al.*, 2006), which is used in MCTS enhancements such as the playout strategy MAST (Finnsson and Björnsson, 2008). The history heuristic does not require any domain knowledge. The idea behind the history heuristic is to exploit the fact that moves that are good in a certain position are also good in other (similar) positions. For each move the number of simulations in which it was played and the total score are stored. This information is used to compute the relative history score. This score can subsequently combined with the UCT selection strategy.[1]

[1]We remark that, around the same time our research was performed, Kozelek (2009) proposed a similar idea in the game of Arimaa.

The combination of Progressive Bias and the relative history heuristic is called *Progressive History*. The heuristic knowledge H_i of Progressive Bias is replaced with the relative history score. The child i with the highest score v_i is now selected as follows (Formula 5.3).

$$v_i = \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} + \bar{x}_a \times \frac{W}{(1-\bar{x}_i)n_i + 1} \tag{5.3}$$

The parameters $\bar{x}_i$, n_i, and n_p and the constant C are the same as in Formula 2.8. $\bar{x}_a$ represents the average score of all games in which move a was played. W is a constant that determines the influence of Progressive History. The higher the value of W, the stronger the history value affects the selection of a node.

In Formula 5.3, $\frac{W}{(1-\bar{x}_i)n_i+1}$ represents the Progressive Bias part and $\bar{x}_a$ the history heuristic part. We remark that, in the Progressive Bias part, we do not divide by the number of visits as is done in the standard definition of Progressive Bias, but by the number of visits multiplied by the inverse win rate, i.e., the number of losses[2]. In this way, nodes that do not perform well are not biased for too long, whereas nodes that do have a high score continue to be biased. To ensure that we do not divide by 0 if the win rate $\bar{x}_i$ is 1, a 1 is added in the denominator.

The move data for Progressive History is stored, for each player separately, in a global table, while RAVE keeps track of the AMAF values in every node (or edge). An advantage of Progressive History is that keeping track of the values globally instead of locally at every node saves memory space. Because the size of the table does not grow as the MCTS tree becomes larger, its memory complexity is $O(1)$. The memory complexity of RAVE is $O(n)$, because the amount of memory necessary to store the AMAF values increases linearly as the tree grows larger. Another advantage of using a global table is that, contrary to RAVE, information may be immediately available if a node is visited for the first time. A possible disadvantage of using a single history table is that the available information may be less accurate for that particular part of the tree. In order to prevent outdated information from influencing the performance of Progressive History, the history table is emptied after a move is played in the actual game.

We note that this implementation of Progressive History has similarities with the *Move-Average Sampling Technique* (MAST) by Finnsson and Björnsson (2008). The difference is that, instead of modifying the selection strategy, MAST is used to bias the playouts. There exist two variations of MAST. Regular MAST updates all moves that have been played in the tree and in the playouts. Tree-Only MAST (TO-MAST) only updates the moves that have been played in the tree (Finnsson, 2012). When using TO-MAST, less information is added to the table, but the information that is added is generally more reliable. Similar to MAST, in Progressive History it is also possible to distinguish between these two strategies. With *Full-Sample Progressive History*, the history table is updated with all moves played during the selection phase and the playout phase. With *Tree-Only Progressive History*, the table is only updated with the moves that were played during the selection phase. Similar to MAST, the

[2]In case of a draw, if a player is awarded a score s, it is counted as $1-s$ loss.

advantage of the former is that more information is available, while the advantage of the latter is that the information is more reliable and is computationally slightly simpler.

Instead of using the history values in Progressive History, it is also possible to use AMAF values instead. AMAF values are typically used in RAVE. If Progressive History is used with AMAF values, it is called *Progressive AMAF*. This selection strategy applies Formula 5.4 for selecting the next node in the tree.

$$v_i = \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} + AMAF_i \times \frac{W}{(1-\bar{x}_i)n_i + 1} \tag{5.4}$$

In this formula, the history value $\bar{x}_a$ is replaced by $AMAF_i$. This is the AMAF value of the move corresponding to node i.

5.3 Experiments and Results

This section provides the experiments and their results. First, Subsection 5.3.1 describes the experimental setup. In Subsection 5.3.2 the parameter W is tuned against MCTS with the standard UCT selection strategy. Next, Subsection 5.3.3 validates that dividing by the number of losses results in a stronger performance than dividing by the number of visits. In Subsection 5.3.4 a comparison between Full-Sample Progressive History and Tree-Only Progressive History is performed. Finally, Subsection 5.3.5 compares Progressive History to Progressive AMAF.

5.3.1 Experimental Setup

The experiments in this section are performed using MAGE (see Section 3.6). In all experiments, each player receives 2500 ms thinking time to determine the move to play. All experiments were performed on a server consisting of AMD OpteronT 2.2 GHz processors. For the four games, i.e., Chinese Checkers, Focus, Rolit, and Blokus, there may be an advantage regarding the order of play and the number of instances of each player type. To give reliable results, each possible player setup, with the exception of setups where each player is of the same type, is played equally often, until approximately 1500 games have been played (cf. Subsection 4.3.1). All results are given with a 95% confidence interval.

5.3.2 Parameter Tuning and General Strength

In the first series of experiments Progressive History is tested with different values of W against UCT in Chinese Checkers, Focus, Rolit, and Blokus. In this set of experiments, the Tree-Only variant of Progressive History is used.

Table 5.1 reveals that Progressive History is a significant improvement to MCTS in all games with any number of players. The highest win rates are achieved in two-player Chinese Checkers. Progressive History wins over 80% of the games, with the best result achieved with $W = 5$, winning 83.5% of the games. In the variants with

Table 5.1 Win rates of Progressive History with different values of W against UCT.

Game	W	2 players win rate (%)	3 players win rate (%)	4 players win rate (%)	6 players win rate (%)
Chinese Checkers	0.1	55.0 ± 2.5	52.7 ± 2.5	57.5 ± 2.5	52.3 ± 2.5
	0.25	63.3 ± 2.4	54.7 ± 2.5	54.3 ± 2.5	53.1 ± 2.5
	0.5	67.2 ± 2.4	57.2 ± 2.5	54.8 ± 2.5	53.2 ± 2.5
	1	74.2 ± 2.2	58.7 ± 2.5	56.9 ± 2.5	53.0 ± 2.5
	3	82.6 ± 1.9	61.1 ± 2.5	60.1 ± 2.4	55.1 ± 2.5
	5	83.5 ± 1.9	61.9 ± 2.5	61.1 ± 2.4	56.1 ± 2.5
	7.5	80.6 ± 2.0	60.9 ± 2.5	59.7 ± 2.4	55.9 ± 2.5
	10	77.3 ± 2.1	59.3 ± 2.5	62.3 ± 2.4	55.8 ± 2.5
	20	70.0 ± 2.3	56.9 ± 2.5	60.0 ± 2.4	54.1 ± 2.5
	50	57.1 ± 2.5	53.9 ± 2.5	57.7 ± 2.5	55.2 ± 2.5
Focus	0.1	72.0 ± 2.3	64.9 ± 2.4	58.2 ± 2.5	-
	0.25	72.2 ± 2.3	67.3 ± 2.4	56.8 ± 2.5	-
	0.5	74.9 ± 2.2	66.6 ± 2.4	58.4 ± 2.5	-
	1	78.3 ± 2.1	67.9 ± 2.4	59.9 ± 2.4	-
	3	73.0 ± 2.2	70.0 ± 2.3	60.5 ± 2.4	-
	5	72.1 ± 2.3	69.0 ± 2.3	62.0 ± 2.4	-
	7.5	69.6 ± 2.3	69.8 ± 2.3	59.9 ± 2.4	-
	10	67.1 ± 2.4	68.3 ± 2.4	60.3 ± 2.4	-
	20	60.3 ± 2.5	65.1 ± 2.4	58.7 ± 2.5	-
	50	49.5 ± 2.5	61.4 ± 2.5	58.7 ± 2.5	-
Rolit	0.1	55.9 ± 2.5	51.0 ± 2.5	52.6 ± 2.5	-
	0.25	61.1 ± 2.5	52.7 ± 2.5	51.7 ± 2.5	-
	0.5	65.8 ± 2.4	52.4 ± 2.5	54.2 ± 2.5	-
	1	72.7 ± 2.3	59.4 ± 2.5	54.0 ± 2.5	-
	3	77.7 ± 2.1	62.0 ± 2.5	54.9 ± 2.5	-
	5	79.7 ± 2.0	62.2 ± 2.5	56.8 ± 2.5	-
	7.5	79.1 ± 2.1	62.0 ± 2.5	57.5 ± 2.5	-
	10	77.8 ± 2.1	59.6 ± 2.5	57.1 ± 2.5	-
	20	74.5 ± 2.2	59.7 ± 2.5	53.4 ± 2.5	-
	50	61.3 ± 2.5	56.7 ± 2.5	51.0 ± 2.5	-
Blokus	0.1	-	-	55.4 ± 2.5	-
	1	-	-	57.5 ± 2.5	-
	5	-	-	57.2 ± 2.5	-
	10	-	-	58.0 ± 2.5	-
	20	-	-	58.0 ± 2.5	-
	50	-	-	57.9 ± 2.5	-
	100	-	-	56.8 ± 2.5	-
	250	-	-	54.6 ± 2.5	-
	500	-	-	53.5 ± 2.5	-
	1000	-	-	52.3 ± 2.5	-

three, four, and six players, the win rate drops significantly. The reason why Progressive History works well in Chinese Checkers is that for this game strong moves are not dependent on the current board position. Strong moves are often moves that move a checker far forward. These are good moves in different positions as well.

Progressive History is a considerable improvement for MCTS in Focus as well. The best result for the two-player variant is achieved with $W = 1$, reaching a win rate of 78.3%. For the three-player variant, the best results are achieved with $W = 3$, winning 70% versus UCT. In the four-player variant, Progressive History still performs well. With $W = 5$ the win rate is slightly above 60%.

For Rolit, the results are comparable to Chinese Checkers. Progressive History wins almost 80% of the games with $W = 5$ in the two-player game. With three players, the highest win rate is achieved around $W = 5$ as well, winning about 62% of the games. In the four-player variant, Progressive History is still a significant improvement, but its highest win rate is 57.5% with $W = 7.5$, which is slightly lower than in most other games.

Finally, in Blokus, Progressive History is a significant improvement as well. Compared to the other games, the performance of Progressive History seems to be less sensitive to the value of W, as similar win rates are achieved with values of W as low as 1 or as high as 50. If the value of W increases to more than 100, the win rate against UCT slowly starts to decrease.

Overall, Progressive History is a significant improvement in all games with any number of players. In most games, the highest win rates are achieved with a value of W around 5. If the value of W is smaller, the influence of Progressive History is smaller and the win rate against UCT is closer to 50%. If the value of W is too large, the influence of the relatively imprecise history values is too strong.

In Chinese Checkers, Focus, and Rolit, the maximum win rate of Progressive History decreases as the number of players increases. This is not caused by the uneven distribution of player types in some configurations (e.g., 5 versus 1). Table 5.2 reveals that the performance drops in a similar way when only the configurations are considered where the player types are evenly distributed. There are, however, two reasons that do explain this phenomenon. The first is that, if there are fewer players, each move is played more often in the same amount of time, because each player has more turns per second. This results in the history table filling faster if there are fewer players, leading to more reliable history values. The second explanation is that, with more players, the number of losses increases faster than with fewer players. This causes the influence of Progressive History to diminish faster.

5.3.3 Dividing by the Number of Losses

In the next series of experiments we validate whether dividing by the number of losses, as is done in Formula 5.3, is an improvement over dividing by the number of visits. In Table 5.3 the results are given for the former when matched against the latter in Chinese Checkers, Focus, Rolit, and Blokus. Both players used $W = 5$, which is in all games one of the best values.

The results show that, in all games with any number of players, dividing by the number of losses performs at least as well as dividing by the number of visits. In

Table 5.2 Win rates of Progressive History against UCT with evenly distributed player types.

Game	2 players (1 vs. 1) win rate (%)	4 players (2 vs. 2) win rate (%)	6 players (3 vs. 3) win rate (%)
Chinese Checkers	83.5 ± 1.9	63.0 ± 3.7	59.6 ± 4.3
Focus	72.1 ± 2.3	61.1 ± 3.7	-
Rolit	79.7 ± 2.0	56.7 ± 3.8	-
Blokus	-	57.4 ± 3.8	-

the two-player variants of Chinese Checkers, Focus, and Rolit, this modification is a significant improvement. In the three-player variants of Chinese Checkers and Focus, dividing by the number of losses is an enhancement as well. In three-player Rolit, however, there is no significant difference. In the four-player variants of all games, both players are equally strong. Only in Chinese Checkers, the difference is statistically significant. In six-player Chinese Checkers, the players are on equal footing as well. A possible explanation for this phenomenon may be that with more players, the number of losses is relatively higher and thus closer to the number of visits n_i. This may diminish the effect of using a different denominator.

Table 5.3 Win rates of Progressive History with $\frac{W}{(1-\bar{x}_i)n_i+1}$ against Progressive History with $\frac{W}{n_i+1}$.

Game	2 players win rate (%)	3 players win rate (%)	4 players win rate (%)	6 players win rate (%)
Chinese Checkers	56.1 ± 2.5	54.2 ± 2.5	52.6 ± 2.5	49.9 ± 2.5
Focus	74.5 ± 2.2	61.3 ± 2.5	51.7 ± 2.5	-
Rolit	61.0 ± 2.5	50.4 ± 2.5	50.3 ± 2.5	-
Blokus	-	-	52.0 ± 2.5	-

5.3.4 Tree-Only Progressive History

The next set of experiments tests whether the computationally less intensive Tree-Only Progressive History performs on at least the same level as Full-Sample Progressive History. For both variants, $W = 5$ is used.

The results are given in Table 5.4. These results show that Tree-Only Progressive History performs worse than Full-Sample Progressive History in two-player and six-player Chinese Checkers. However, in two-player Focus, Tree-Only Progressive History performs significantly better. In all other experiments, there is no significant difference in playing strength between Full-Sample Progressive History and Tree-Only Progressive History. On average, Tree-Only Progressive History wins slightly more games than Full-Sample Progressive History, though this difference is not significant. This is comparable to the results found by Finnsson (2012). He compared the playout strategies MAST and Tree-Only MAST in General Game Playing in twelve different games and observed that there is, on average, no large difference in the performance of the two different techniques. Because Tree-Only Progressive History is computationally lighter, this variant is used throughout the remainder of the experiments.

Table 5.4 Win rates of Tree-Only Progressive History against Full-Sample Progressive History.

Game	2 players win rate (%)	3 players win rate (%)	4 players win rate (%)	6 players win rate (%)
Chinese Checkers	44.5 ± 2.5	52.6 ± 2.5	48.5 ± 2.5	45.8 ± 2.5
Focus	55.7 ± 2.5	50.7 ± 2.5	49.2 ± 2.5	-
Rolit	51.6 ± 2.5	49.8 ± 2.5	51.0 ± 2.5	-
Blokus	-	-	52.0 ± 2.5	-

5.3.5 Application of AMAF Values

In the final set of experiments we test whether using a global history table in Progressive History performs better than using AMAF values, which are typically used in RAVE. During preliminary experiments with the various RAVE formulas (Gelly and Silver, 2007; Teytaud and Teytaud, 2010a; Tom and Müller, 2010) (see Appendix A), these variants did not appear to work in multi-player games. Progressive AMAF, which can be regarded as a RAVE variant as well, appeared to work best. Therefore, this selection strategy is used to test the performance of Progressive History against. The win rates of Progressive History against Progressive AMAF in Chinese Checkers, Focus, Rolit, and Blokus are given in Table 5.5. Again, both player types use $W = 5$.

In Chinese Checkers with two, three, and four players, Progressive History with history values performs significantly better than Progressive AMAF values. As the number of players increases, the relative performance of Progressive History against Progressive AMAF decreases. In the six-player variant, Progressive AMAF performs on a similar level as Progressive History. In Rolit, Progressive History outperforms Progressive AMAF in the two-player variant, but in the three-player and four-player variants, they perform on an equal level.

In Focus and Blokus, the win rates of Progressive History against Progressive AMAF are similar to the win rates of Progressive History against standard UCT. This shows that the application of AMAF values does not improve the performance over standard UCT in these games.

Overall, we may conclude that applying global data (i.e., a history table) provides a better performance than applying local data (i.e., AMAF values).

5.4 Progressive History in Havannah

Fossel (2010) applied Progressive History to the two-player connection game *Havannah*. This is a popular domain for the application of RAVE (Teytaud and Teytaud, 2010a; Hoock *et al.*, 2010; Lorentz, 2011; Rimmel *et al.*, 2011a). First, he compared

Table 5.5 Win rates of Progressive History against Progressive AMAF.

Game	2 players win rate (%)	3 players win rate (%)	4 players win rate (%)	6 players win rate (%)
Chinese Checkers	74.2 ± 2.2	59.0 ± 2.5	55.6 ± 2.5	49.2 ± 2.5
Focus	83.2 ± 1.9	70.3 ± 2.3	59.1 ± 2.5	-
Rolit	66.2 ± 2.4	50.2 ± 2.5	52.0 ± 2.5	-
Blokus	-	-	57.8 ± 2.5	-

RAVE to standard UCT. The following RAVE formula was applied to select the child i with the highest value v_i.

$$v_i = (1-\beta)\bar{x}_i + \beta\bar{x}_a + C \times \sqrt{\frac{\ln(n_p)}{n_i}}, \text{ with } \beta = \frac{k}{n_i + k} \tag{5.5}$$

Depending on the parameter settings, RAVE won up to 67% out of 500 games against UCT with $C = 1.8$. This result is comparable to the results achieved by Teytaud and Teytaud (2010a).

In the next set of experiments, Progressive History, with different values of C and W, was matched against RAVE. Both players received 5 seconds of thinking time per move. 500 games were played to determine the win rates. The results revealed that Progressive History won up to approximately 60% of the games against RAVE. The highest win rates were achieved with $W = 10$, winning approximately 60% of the games against RAVE. Against UCT with $C = 1.8$, Progressive History won 73% of the games. From these results it may be concluded that Progressive History is an important enhancement for the MCTS selection strategy in the game of Havannah.

Furthermore, Fossel (2010) introduced a combination of Progressive History and RAVE, namely Extended RAVE. This selection strategy applied the following formula.

$$v_i = \bar{x}_i + C \times \sqrt{\frac{\ln(n_p)}{n_i}} + (\beta_i r'_i + (1-\beta_i) r_i) \times \frac{W}{(1-\bar{x}_i)n_i + 1} \tag{5.6}$$

In this formula, the history heuristic part $\bar{x}_a$ is replaced by $(\beta_i r'_i + (1-\beta_i)r_i)$. In this formula, r_i is the RAVE value for the parent p of node i and r'_i is the RAVE value for the parent of p for the same move as i. To test the performance of Extended RAVE, it was matched against RAVE. The same settings were used as in the previous experiments. The results revealed that, with $C = 0.4$, and $W = 10$ or $W = 15$, Extended RAVE won 60% of the games against RAVE. This indicates that Progressive History and Extended RAVE perform on the same level. However, the results showed that Extended RAVE is more affected by suboptimal parameter settings and therefore Progressive History is a more robust enhancement.

5.5 Chapter Conclusions and Future Research

This chapter introduced a domain-independent enhancement for the selection phase of MCTS in two-player and multi-player games, namely Progressive History. This is a combination of Progressive Bias and the relative history heuristic. We determined the performance of this enhancement against the standard UCT selection strategy in four different games: Chinese Checkers, Focus, Rolit, and Blokus.

The strength of Progressive History was determined by letting an MCTS player with this enhancement play with different values of the constant W against an MCTS player without Progressive History. Depending on the game and the number of players, Progressive History wins approximately 60% to 80% of the games against MCTS without Progressive History with optimal values of W. With an increasing number of players, the performance of Progressive History drops, though it still remains a significant improvement over the standard UCT selection strategy.

Furthermore, dividing by the number of losses in the Progressive Bias part performs at least as well as dividing by the number of visits. In a comparison between Tree-Only Progressive History and Full-Sample Progressive History, there is overall no clear difference in playing strength between the two variants. While Tree-Only Progressive History has less information available than Full-Sample Progressive History, its information is more reliable. In general, these two factors cancel each other out. Finally, we observed that Progressive History overall achieves stronger play with global data (i.e., a history table) than with local data (i.e., AMAF values). Only in six-player Chinese Checkers and three-player and four-player Rolit, Progressive History does not play significantly stronger. In these variants, both selection strategies perform equally strong.

Moreover, for the two-player game Havannah, Fossel (2010) found that Progressive History outperforms UCT and RAVE by significant margins. Based on these results we may conclude that Progressive History significantly improves MCTS in both two-player and multi-player domains.

In multi-player games, there is still much room for improvement. Progressive History works well in five different games and may also work well in other games. This is a possible subject for future research. Moreover, comparisons with other variants to bias the selection strategy besides RAVE, such as prior knowledge (Gelly and Silver, 2007), Progressive Widening (Coulom, 2007b; Chaslot *et al.*, 2008b), and Progressive Bias (Chaslot *et al.*, 2008b) could be performed. It may also be interesting to combine Progressive History with any of the aforementioned techniques. Progressive History may also be combined with N-grams (cf. Stankiewicz, Winands, and Uiterwijk, 2012; Tak *et al.*, 2012; Powley, Whitehouse, and Cowling, 2013), which keeps track of move sequences instead of single moves. By using N-grams, more context in which the moves are played is offered.

CHAPTER 6

Search-based Playouts for Multi-Player MCTS

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

1. Nijssen, J.A.M. and Winands, M.H.M. (2012b). Playout Search for Monte-Carlo Tree Search in Multi-Player Games. *Advances in Computer Games (ACG 13)* (eds. H.J. van den Herik and A. Plaat), Vol. 7168 of *LNCS*, pp. 72–83, Springer, Berlin, Germany.
2. Nijssen, J.A.M. and Winands, M.H.M. (2012c). Playout Search for Monte-Carlo Tree Search in Multi-Player Games. *Proceedings of the 24th Benelux Conference on Artificial Intelligence* (eds. J.W.H.M. Uiterwijk, N. Roos, and M.H.M. Winands), pp. 309–310, Maastricht, The Netherlands. Extended abstract.

For the playouts in MCTS, a tradeoff between search and knowledge has to be made. The more knowledge is added, the slower each playout gets, decreasing the number of playouts per second. The trend seems to favor fast simulations with computationally light knowledge, although recently, adding more heuristic knowledge at the cost of slowing down the playouts has proven to be beneficial in some games (cf. Winands and Björnsson, 2011). Game-independent enhancements in the playout phase of MCTS such as Move-Average Sampling Technique (Finnsson and Björnsson, 2008), Predicate-Average Sampling Technique (Finnsson and Björnsson, 2010), Feature-Average Sampling Technique (Finnsson and Björnsson, 2010), Last Good Reply (Drake, 2009), Last Good Reply with Forgetting (Baier and Drake, 2010), RAVE (Rimmel, Teytaud, and Teytaud, 2011b), and N-grams (Tak *et al.*, 2012) have shown to increase the playing strength of MCTS programs significantly.

The quality of the playouts can also be enhanced by applying search techniques. Winands and Björnsson (2011) proposed $\alpha\beta$-based playouts for the two-player game Lines of Action. Although computationally intensive, it significantly improved the playing strength of the MCTS-based program MC-LOA.

This chapter answers the third research question by introducing two-ply search-based playouts for MCTS in multi-player games. Instead of using computationally light knowledge in the playout phase, two-ply minimax-based searches, equipped with a heuristic evaluation function, are used to determine the moves to play. We test three different search techniques that may be used for search-based playouts. These search techniques are max^n, paranoid, and Best-Reply Search (BRS). Search-based playouts are tested in the three-player and four-player variants of Chinese Checkers and Focus. The search-based playouts are compared to random, greedy, and one-ply playouts.

The remainder of this chapter is structured as follows. First, Section 6.1 provides a background on applying knowledge and search techniques in the playout phase of MCTS. Next, Section 6.2 gives an overview of three different two-ply search-based playout strategies in multi-player MCTS. The experimental results are discussed in Section 6.3. Finally, Section 6.4 provides the chapter conclusions and an outlook on future research.

6.1 Related Work

There exist various playout strategies that incorporate small searches in the playouts to increase their quality.

Teytaud and Teytaud (2010b) introduced decisive and anti-decisive moves for the selection and the playout phase of MCTS. Decisive moves are moves that immediately lead to a win for the current player. Anti-decisive moves are moves that prevent the opponent from making a decisive move on their next turn. At each position during the playouts, a small search is performed to determine whether the player has decisive or anti-decisive moves. If he does, then one such move is played. Otherwise, the default strategy is used. They showed that, despite the computational overhead caused by the searches, the performance of MCTS in the two-player game Havannah can be significantly improved by searching for decisive and anti-decisive moves. A similar playout strategy was used by Lorentz (2011) in the MCTS-based Havannah program WANDERER. Checking for forced wins in the playouts increases the playing strength of this program significantly.

Winands and Björnsson (2011) described the application of $\alpha\beta$ search in the MCTS-based Lines of Action (LOA) program MC-LOA. The new version of this program, MC-LOA$_{\alpha\beta}$, applied two-ply $\alpha\beta$ searches to choose moves during the playouts. A heuristic board evaluation function was used to assign values to the leaf nodes of the search trees. The $\alpha\beta$ search was enhanced with killer moves and aspiration search to reduce the overhead. Round-robin tournaments against a different LOA program, the $\alpha\beta$-based program MIA, and the standard version of MC-LOA revealed that MC-LOA$_{\alpha\beta}$ performed better with increasing thinking time for the players. With 30 seconds of thinking time per move, MC-LOA$_{\alpha\beta}$ was able to defeat both opponents in approximately 60% of the games, while with only 1 second of computing time, MC-LOA$_{\alpha\beta}$ was outperformed by both MIA and the standard version of MC-LOA. The main conclusion that could be drawn was that the small $\alpha\beta$ searches in the playout can improve the performance of an MCTS program significantly, provided that the players receive sufficient thinking time per move.

Baier and Winands (2013) applied one-ply to four-ply searches in the playout phase. Because no game-specific knowledge was applied, these searches were only used to find forced wins or losses in the playouts. They tested this enhancement in the two-player games Connect-Four and Breakthrough. The results showed that in

Connect-Four the application of two-ply and three-ply searches increased the performance of an MCTS-based player significantly. With two-ply searches, the win rate was 72.1% ± 1.9 against MCTS with random playouts. In Breakthrough, however, these searches caused MCTS to perform considerably worse. This was caused by the large amount of overhead.

6.2 Search-Based Playouts in Multi-Player MCTS

For multi-player games, the basic $\alpha\beta$ search technique cannot be applied to determine which moves to choose during the playout phase of the MCTS algorithm. Instead, one of the minimax-based multi-player techniques, i.e., max^n, paranoid, or BRS, is used for selecting moves in the playouts. These searches are performed up to two plies. A static board evaluator is used to assign a value to the leaf nodes of these small minimax-based search trees. The three different playout strategies are discussed in Subsections 6.2.1, 6.2.2, and 6.2.3, respectively.

6.2.1 Two-Ply Max^n

For each move in the playouts, a two-ply max^n search tree is built where the current player is the root player and the first opponent plays at the second ply. Both the root player and the first opponent try to maximize their own score. All other opponents are not taken into consideration. In max^n, shallow pruning may be applied. However, because in practice there is barely any pruning, especially in small two-ply searches, shallow pruning is not applied. An example of a playout with two-ply max^n searches in a three-player game is provided in Figure 6.1.

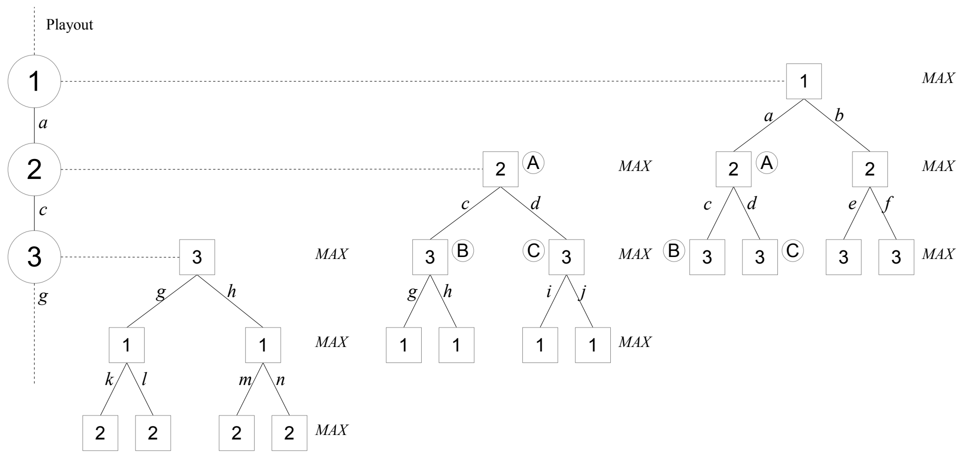


Figure 6.1: Two-ply max^n searches in a three-player game.

6.2.2 Two-Ply Paranoid

Similar to max^n, a two-ply search tree is built for each move in the playout where the current player is the root player and the first opponent plays at the second ply. Again, all other opponents are not taken into account. The root player tries to maximize its own score, while the first opponent tries to minimize the root player's score. In a two-ply paranoid search tree, $\alpha\beta$-pruning is possible. In Figure 6.2, an example of a playout with two-ply paranoid searches is provided.

This playout strategy is different to Paranoid UCT proposed by Cazenave (2008). With two-ply paranoid playouts, each player plays in a paranoid way against one of his opponents during the playouts. With Paranoid UCT the root player (of the MCTS tree) greedily selects moves to improve his own position, while the opponents play moves to decrease the root player's position.

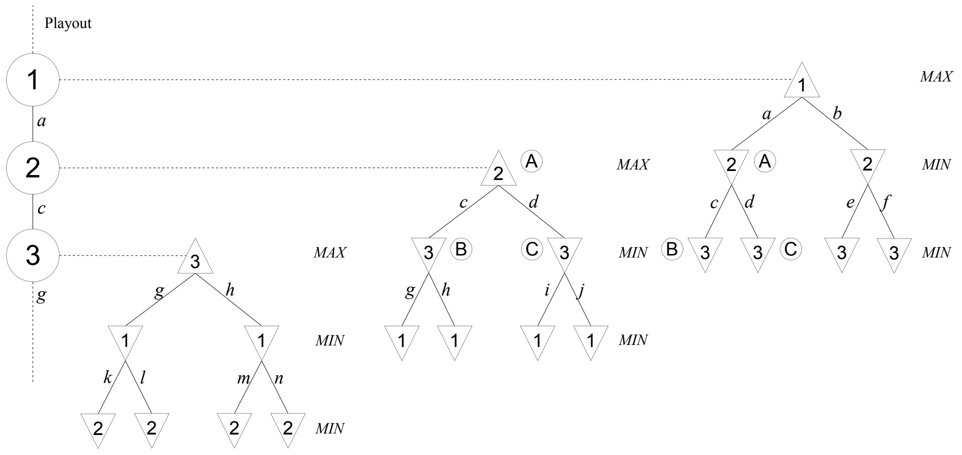


Figure 6.2: Two-ply paranoid searches in a three-player game.

6.2.3 Two-Ply BRS

The two-ply BRS search trees are similar to the aforementioned paranoid trees. The difference is that at the second ply, not only the moves of the first opponent are considered, but the moves of all opponents are investigated. Similar to paranoid search, $\alpha\beta$-pruning is possible. An advantage of this technique is that all opponents are taken into consideration. A disadvantage is that BRS searches in the playouts are more expensive. The branching factor at the second ply is higher than in max^n or paranoid, because it contains moves by all opponents, instead of only one. This may reduce the number of playouts per second significantly. An example of two-ply BRS searches in a playout of a three-player game is given in Figure 6.3.

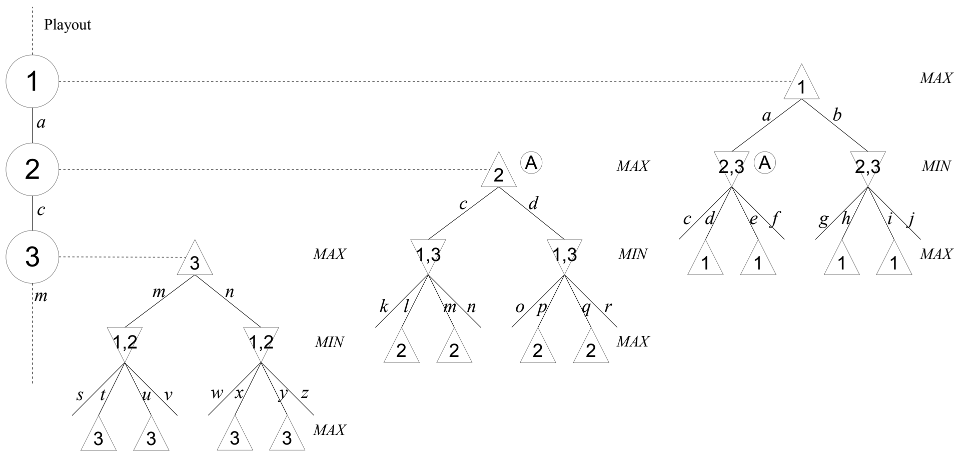


Figure 6.3: Two-ply BRS searches in a three-player game.

6.2.4 Search Enhancements

The major disadvantage of incorporating two-ply search in the playout phase of MCTS is the reduction of the number of playouts per second (Winands and Björnsson, 2011). In order to prevent this reduction from outweighing the benefit of the increase in quality of the playouts, enhancements may be implemented to speed up the search and keep the reduction of the number of playouts to a minimum. The following enhancements are employed to speed up the search-based playouts.

The number of expensive two-ply searches can be reduced by using *ϵ-greedy playouts* (Sutton and Barto, 1998; Sturtevant, 2008a). With a probability of ϵ, a move is chosen uniform randomly. Otherwise, the two-ply search-based playout strategy is used to compute the next move. An additional advantage of ϵ-greedy playouts is that the presence of this random factor gives more varied playouts and may prevent the playouts from being stuck in 'local optima', where all players keep moving back and forth. Furthermore, ϵ-greedy playouts provide more balance between exploration and exploitation.

The amount of $\alpha\beta$-pruning in a tree can be increased by applying *move ordering*. The ordering of the moves is based on a static move evaluator. In the best case, the number of evaluated positions in a two-ply search is reduced from $O(b^2)$ to $O(2b-1)$ (Knuth and Moore, 1975). The size of the tree can be further reduced by using *k-best pruning*. Only the k best moves, according to the static move ordering, are investigated (Winands and Björnsson, 2011). This reduces the branching factor of the tree from b to k. The parameter k should be chosen such that it is significantly smaller than b, while avoiding the best move being pruned. Move ordering and k-best pruning are used to limit the branching factor and enhance the amount of pruning in max^n, paranoid, and BRS. We remark that for BRS, at the second ply, k moves are investigated in total, instead of k moves per opponent.

Another move ordering technique is applying *killer moves* (Akl and Newborn, 1977). In each search, two killer moves are always tried first. These are the two last moves that were best or caused a cutoff, at the current depth. If the search is completed, the killer moves for that specific level in the playout are stored, such that they can be used during the next MCTS iterations. Killer moves are only used with search techniques where $\alpha\beta$-pruning is possible, i.e., paranoid and BRS search.

The amount of overhead may also be reduced by reusing trees between searches. For instance, in Figure 6.1, the subtree of node A, including the generated moves c and d, may be reused as this node becomes the root node in the next search. Because these nodes and the corresponding moves do not need to be created or generated again, the amount of overhead may be reduced. For max^n and paranoid, the subtrees may directly be reused. For BRS, this is slightly less trivial, because the moves of the other opponents need to be removed first. For instance, in Figure 6.3 the moves of Player 3, e and f, have to be removed from node A before it can be reused.

Other enhancements were tested, but they did not improve the performance of the MCTS program. The application of *transposition tables* (Greenblatt *et al.*, 1967) was tested, but the information gain did not compensate for the overhead. Also, *aspiration search* (Marsland, 1986) did not speed up the search significantly. This can be attributed to the limited amount of pruning possible in a two-ply search tree.

6.3 Experiments and Results

This section describes the experiments and results to determine the playing strength of the different search-based playout strategies. First, the experimental setup is discussed in Subsection 6.3.1. Next, Subsection 6.3.2 investigates how the different playout strategies affect the quality of the playouts. Subsection 6.3.3 provides an overview on how the different playout strategies affect the number of playouts per second. Finally, in Subsection 6.3.4, the speed factor is taken into account by limiting the experiments by computation time per move. These experiments are run with 5000 ms and 30,000 ms per move.

6.3.1 Experimental Setup

The experiments in this section are performed with the computer program MAGE. It uses the enhancements, parameter settings and evaluation functions described in Section 3.6, including Tree-Only Progressive History with $W = 5$ and the max^n search policy. All players, except the one with random playouts, use ϵ-greedy playouts with $\epsilon = 0.05$ and k-best pruning with $k = 5$. These values were achieved by systematic testing with values $\epsilon \in [0,1]$ and $k \in [2,10]$. The experiments are run on a cluster consisting of AMD64 Opteron 2.4 GHz processors. In order to test the performance of the two-ply search-based playouts, several round-robin tournaments are performed where each participating player uses a different playout strategy. Due to the time-consuming nature of these experiments, they are only performed in the three-player and four-player variants of Chinese Checkers and Focus. The following playout strategies are used as a comparison for the search-based playouts.

Random move selection. Moves are selected in a uniform random way. Each move has an equal probability of being selected.

Static move ordering. The static move ordering (SMO) is applied to evaluate all possible moves. The move with the highest heuristic value is played. Because the move evaluator generally uses light heuristic knowledge (see Chapter 3), this playout strategy is relatively fast. This is the default playout strategy that is applied in Chapters 4, 5, and 7.

One-ply search. The *k* best moves according to the static move ordering are investigated and the resulting board positions are evaluated using the static board evaluator (see Chapter 3). This evaluation is more time-expensive than greedy move selection, because the moves have to be played (and undone afterwards). Furthermore, the board evaluator, which is more complex, is used to evaluate (at most) *k* different board positions.

In each match of the round-robin tournament, two different player types participate. In order to fill all seats in the three-player and four-player games, multiple instances of each player type may participate. For both games, there may be an advantage regarding the order of play and the number of different players. In a three-player game there are $2^3 = 8$ different player-type assignments. Games where only one player type is playing are not interesting, leaving 6 ways to assign player types. For four players, there are $2^4 - 2 = 14$ assignments. Each assignment is played multiple times until approximately 1500 games are played and each assignment is played equally often. In Table 6.1, 95% confidence intervals of some win rates for 1500 games are given.

Table 6.1 95% confidence intervals of some win rates for 1500 games.

Win percentage	Confidence interval
50%	± 2.5%
40% / 60%	± 2.5%
30% / 70%	± 2.3%
20% / 80%	± 2.0%

6.3.2 Fixed Number of Playouts per Move

In the first set of experiments, all players are allowed to perform 5000 playouts per move. The results are given in Table 6.2. The numbers are the win percentages of the players denoted on the left against the players denoted at the top.

The results show that for three-player Chinese Checkers, BRS and paranoid are the best playout strategies, closely followed by max^n. BRS wins 53.4% of the games against max^n, which is a statistically significant difference, and 50.9% against paranoid. These three techniques perform significantly better than one-ply and SMO. The

Table 6.2 Round-robin tournament of the different playout strategies in Chinese Checkers and Focus with 5000 playouts per move (win%).

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	0.0	0.1	0.1	0.0	0.0	0.0
SMO	100.0	-	25.2	20.9	21.2	18.3	37.1
One-ply	99.9	74.8	-	44.5	40.5	38.9	59.7
Max^n	99.9	79.1	55.5	-	48.1	46.6	65.8
Paranoid	100.0	78.8	59.5	51.9	-	49.1	67.9
BRS	100.0	81.7	61.1	53.4	50.9	-	69.4

Three-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	1.1	0.5	0.7	1.0	0.9	0.8
SMO	98.9	-	30.3	27.6	26.9	22.9	41.3
One-ply	99.5	69.7	-	47.4	45.1	39.7	60.3
Max^n	99.3	72.4	52.6	-	49.1	48.1	64.3
Paranoid	99.0	73.1	54.9	50.9	-	46.2	64.8
BRS	99.1	77.1	60.3	51.9	53.8	-	68.4

Four-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	6.1	3.8	2.7	3.4	2.1	3.6
SMO	93.9	-	44.5	38.4	38.5	33.3	49.7
One-ply	96.2	55.5	-	44.3	44.1	40.5	56.1
Max^n	97.3	61.6	55.7	-	52.0	45.2	62.4
Paranoid	96.6	61.5	55.9	48.0	-	44.5	61.3
BRS	97.9	66.7	59.5	54.8	55.5	-	66.9

Three-player Focus

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	11.3	10.3	6.7	5.1	4.5	7.6
SMO	88.7	-	42.0	35.0	35.2	33.4	46.9
One-ply	89.7	58.0	-	43.3	42.6	40.1	54.7
Max^n	93.3	65.0	56.7	-	50.5	48.5	62.8
Paranoid	94.9	64.8	57.4	49.5	-	48.2	63.0
BRS	95.5	66.6	59.9	51.5	51.8	-	65.1

Four-player Focus

win rates against one-ply are approximately 55% to 60% and against SMO around 80%. The MCTS player with random playouts barely wins any games.

In the four-player variant, max^n, paranoid, and BRS remain the best techniques, where BRS performs slightly better than the other two. Max^n and paranoid are on equal footing. BRS wins 53.8% of the games against paranoid and 51.9% against max^n. The win rates of max^n, paranoid, and BRS are around 75% against SMO and from 52.6% to 60.3% against one-ply. Random wins slightly more games than in the three-player variant, but its performance is still poor.

For three-player Focus, the best technique is BRS, winning 54.8% against max^n and 55.5% against paranoid. Max^n and paranoid are equally strong. The win rates of max^n, paranoid, and BRS are more than 60% against SMO and between 55% and 60% against one-ply. Compared to Chinese Checkers, random wins relatively more games in Focus. This is because of the progression property (Winands *et al.*, 2010; Finnsson, 2012) in these two games. Some games progress towards a natural termination with every move made while other allow moves that only maintain a status quo. In Focus there is some, though little, natural progression, because captured pieces of the opponents cannot be lost. In Chinese Checkers, however, there is no natural progression at all, because pieces may move away from the home base. The less natural progression there is in games, the less effective random playouts become.

BRS is also the best technique in four-player Focus, though it is closely followed by max^n and paranoid. The latter two search strategies are equally strong. BRS wins 51.5% of the games against max^n and 51.8% against paranoid. Random performs relatively better in the four-player variant than in the three-player variant, though it is still significantly weaker than the search strategies that are enhanced with domain knowledge.

Overall, the highest win rates are achieved by applying two-ply BRS searches in the playouts. In all games, BRS performs significantly better than all opponents, though the difference with paranoid in three-player Chinese Checkers, and with max^n and paranoid in four-player Focus, is relatively small. Max^n and paranoid perform, in all variants, on an equal level. Because these playout strategies do not investigate the moves of all opponents during the two-ply searches, they do not increase the reliability of the playouts as much as BRS. All two-ply search-based playouts outperform one-ply and SMO in all game variants. This shows that applying two-ply searches in the playouts does significantly improve the reliability of the playouts. One-ply performs better than SMO in all games. The reason for this is that the board evaluator computes the effect of a move on the global board position, while the computationally lighter move evaluator only values local effects. Random playouts are by far the weakest playout strategy in all games.

6.3.3 Overhead

For reference, Table 6.3 shows the median number of playouts per second for each type of player in each game variant. These numbers were obtained from the experiments in the previous section. Note that at the start of the game, the number of playouts is smaller. As the game progresses, the playouts become shorter and the number of playouts per second increases.

Max^n and BRS are the slowest playout strategies. This is caused by the lack of pruning and a higher branching factor at the opponent's ply, respectively. Among the two-ply search-based playout strategies, the speed reduction for paranoid is the smallest. The results in this table furthermore underline that one-ply is overall slower than SMO. This is because the board evaluator is more time-expensive than the move evaluator.

Table 6.3 Median number of playouts per second for each type of player in each game variant.

Game	Random	SMO	One-ply	Max^n	Paranoid	BRS
Chinese Checkers (3p)	2419	3489	2421	517	1008	649
Chinese Checkers (4p)	1786	2841	1862	379	744	360
Focus (3p)	800	5476	4686	982	1678	756
Focus (4p)	724	5118	4413	1063	1847	561

6.3.4 Fixed Amount of Time per Move

To test the influence of the reduced number of playouts per second in the two-ply search-based playouts, in the next series of experiments the players receive a fixed amount of time to compute the best move. These experiments are performed with short (5000 ms per move) and long (30,000 ms per move) time settings.

5000 ms per move

In the second set of experiments, each player receives 5000 ms thinking time per move. The results of the round-robin tournament are given in Table 6.4.

In three-player Chinese Checkers, one-ply and paranoid are the best playout strategies. Paranoid wins 49.2% of the games against one-ply and 68.5% against SMO. BRS ranks third, followed by max^n and SMO. Similar to the previous set of experiments, random barely wins any games.

In four-player Chinese Checkers, one-ply is the best strategy, closely followed by paranoid. One-ply wins 53.7% of the games against paranoid. Paranoid is still stronger than SMO, winning 64.6% of the games. BRS comes in third place, outperforming max^n and SMO. Again, random is by far the weakest playout strategy.

One-ply also performs best in three-player Focus. Paranoid plays as strong as SMO, with paranoid winning 51.9% of the games against SMO. One-ply wins 56.8% of the games against SMO and 53.9% against paranoid. SMO and paranoid perform better than BRS and max^n. These two techniques perform equally strong.

In four-player Focus, paranoid overall performs slightly better than in the three-player variant and plays as strong as one-ply. Paranoid wins 51.7% of the games against one-ply and 59.9% against SMO. Max^n also performs significantly better than in the three-player version. It is as strong as one-ply and better than SMO, winning 58.9% of the games against the latter. The performance of BRS in the four-player variant is significantly lower than paranoid, max^n, and one-ply.

Table 6.4 Round-robin tournament of the different playout strategies in Chinese Checkers and Focus for time settings of 5000 ms per move (win%).

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	0.1	0.0	0.0	0.0	0.3	0.1
SMO	99.9	-	28.7	42.7	31.5	36.1	47.8
One-ply	100.0	71.3	-	62.5	50.8	58.2	68.6
Max^n	100.0	57.3	37.5	-	36.1	43.5	54.9
Paranoid	100.0	68.5	49.2	63.9	-	55.7	67.5
BRS	99.7	63.9	41.8	56.5	44.3	-	61.2

Three-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	0.6	1.1	1.1	0.7	1.7	1.0
SMO	99.4	-	33.7	45.9	35.4	42.9	51.5
One-ply	98.9	66.3	-	60.5	53.7	56.2	67.1
Max^n	98.9	54.1	39.5	-	40.3	46.6	55.9
Paranoid	99.3	64.6	46.3	59.7	-	56.2	65.2
BRS	98.3	57.1	43.8	53.4	43.8	-	59.3

Four-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	1.5	1.6	2.9	2.7	2.9	2.3
SMO	98.5	-	43.2	54.2	48.1	51.8	59.2
One-ply	98.4	56.8	-	58.9	53.9	57.9	65.2
Max^n	97.1	45.8	41.1	-	43.5	50.7	55.6
Paranoid	97.3	51.9	46.1	56.5	-	52.7	60.9
BRS	97.1	48.2	42.1	49.3	47.3	-	56.8

Three-player Focus

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	7.1	5.7	7.0	6.5	7.1	6.7
SMO	92.9	-	42.3	41.1	40.1	43.9	52.1
One-ply	94.3	57.7	-	51.3	48.3	54.5	61.2
Max^n	93.0	58.9	48.7	-	47.9	55.9	60.9
Paranoid	93.5	59.9	51.7	52.1	-	54.3	62.3
BRS	92.9	56.1	45.5	44.1	45.7	-	56.9

Four-player Focus

Overall, BRS and max^n perform relatively worse, compared to the previous set of experiments, while the performance of one-ply and SMO is relatively better. The win rates of paranoid with 5000 ms per move are similar to the win rates with 5000 samples per move. Paranoid and one-ply are the best playout strategies if the players receive 5000 ms of thinking time per move. BRS, which was the best playout strategy in the first series of experiments, performs worse than one-ply and paranoid in all game variants. Max^n suffers from the lower number of playouts as well. It is outperformed by one-ply and paranoid in all games, and by BRS in Chinese Checkers. SMO performs relatively better with 5000 ms per move than with 5000 samples per move, but it still performs worse than all playout strategies, except random, in Chinese Checkers and four-player Focus. Random is again the weakest playout strategy in all game variants.

30,000 ms per move

In the final set of experiments, all players receive 30,000 ms of computation time per move. Because these games take much time to finish, only around 500 games are played per match. In Table 6.5, 95% confidence intervals of some win rates for 500 games are given. The results are given in Table 6.6.

Table 6.5 95% confidence intervals of some win rates for 500 games.

Win percentage	Confidence interval
50%	± 4.4%
40% / 60%	± 4.3%
30% / 70%	± 4.0%
20% / 80%	± 3.5%

In the three-player variant of Chinese Checkers, paranoid is the best playout strategy. While its win rate against one-ply is not significantly above 50%, its overall win rate against all opponents, which is 69.6%, is significantly higher than the overall win rate of one-ply, which is 65.6%. BRS ranks third, outperforming max^n and SMO. While it wins 49.8% of the games against one-ply, its overall win rate is significantly lower. MCTS with random playouts did not win a single game against any of the opponents.

In Chinese Checkers with four players, one-ply and paranoid are the strongest playout strategies, closely followed by BRS. Paranoid wins 50.4% of the games against one-ply and 52.8%, against BRS. The two-ply search-based techniques all significantly outperform SMO, winning between 55% and 65%.

In three-player Focus, BRS performs especially well with 30,000 ms of thinking time per move. In the previous set of experiments, BRS was outperformed by both one-ply and paranoid, but in this set of experiments it performs slightly better than paranoid, winning significantly more games overall, and is on equal footing with one-ply. Max^n ranks fourth. It wins less than 50% of the games against one-ply, paranoid, and BRS, but it outperforms SMO with a significant margin. While random wins more games in Focus than in Chinese Checkers, it still performs the worst.

Table 6.6 Round-robin tournament of the different playout strategies in Chinese Checkers and Focus for time settings of 30,000 ms per move (win%).

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	0.0	0.0	0.0	0.0	0.0	0.0
SMO	100.0	-	31.9	32.1	27.0	35.7	45.3
One-ply	100.0	68.1	-	61.9	47.8	50.2	65.6
Max^n	100.0	67.9	38.1	-	34.1	43.7	56.8
Paranoid	100.0	73.0	52.2	65.9	-	57.1	69.6
BRS	100.0	64.3	49.8	56.3	42.9	-	62.3

Three-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	1.2	2.8	1.0	1.0	0.8	1.4
SMO	98.8	-	34.3	42.1	35.3	38.5	49.8
One-ply	97.2	65.7	-	58.1	49.6	53.2	64.8
Max^n	99.0	57.9	41.9	-	41.9	43.8	56.9
Paranoid	99.0	64.7	50.4	58.1	-	52.8	65.0
BRS	99.2	61.5	46.8	56.2	47.2	-	62.2

Four-player Chinese Checkers

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	4.0	2.8	4.8	3.4	5.6	4.1
SMO	96.0	-	38.7	42.7	40.9	37.9	51.2
One-ply	97.2	61.3	-	56.7	52.2	49.4	63.4
Max^n	95.2	57.3	43.3	-	46.0	44.2	57.2
Paranoid	96.6	59.1	47.8	54.0	-	48.0	61.1
BRS	94.4	62.1	50.6	55.8	52.0	-	63.0

Three-player Focus

	Random	SMO	One-ply	Max^n	Paranoid	BRS	Average
Random	-	12.5	9.3	6.3	7.9	8.7	8.9
SMO	87.5	-	40.7	36.5	37.5	42.1	48.9
One-ply	90.7	59.3	-	46.8	45.4	49.6	58.4
Max^n	93.7	63.5	53.2	-	50.2	48.2	61.8
Paranoid	92.1	62.5	54.6	49.8	-	50.8	62.0
BRS	91.3	57.9	50.4	51.8	49.2	-	60.1

Four-player Focus

Finally, in four-player Focus, the two-ply search-based playouts overall perform best. Max^n and paranoid are equally strong, closely followed by BRS. Max^n, paranoid, and BRS all win approximately 50% of the game against each other. Against SMO, Max^n and paranoid win around 63% of the games, while they win around 54% of the games against one-ply. BRS wins approximately 58% and 50% against SMO and one-ply, respectively.

Overall, the two-ply search-based playout strategies perform relatively better with 30,000 ms of thinking time per move, compared to 5000 ms of thinking time. On average, the two-ply paranoid playout strategy achieves the highest win rates in the four different game variants. In three-player Chinese Checkers, it outperforms all other playout strategies. In four-player Chinese Checkers and four-player Focus, it performs on a similar level as one-ply and max^n, respectively. Only in three-player Focus, one-ply performs slightly better than paranoid. Compared to the previous set of experiments, BRS seems to benefit most from the additional thinking time. Especially in Focus, its performance increases by a significant margin. SMO and one-ply perform relatively worse with more computation time. For these playout strategies, the advantage of having more playouts per second is diminished.

The results of the three sets of experiments in the four different domains are summarized in Figure 6.4.

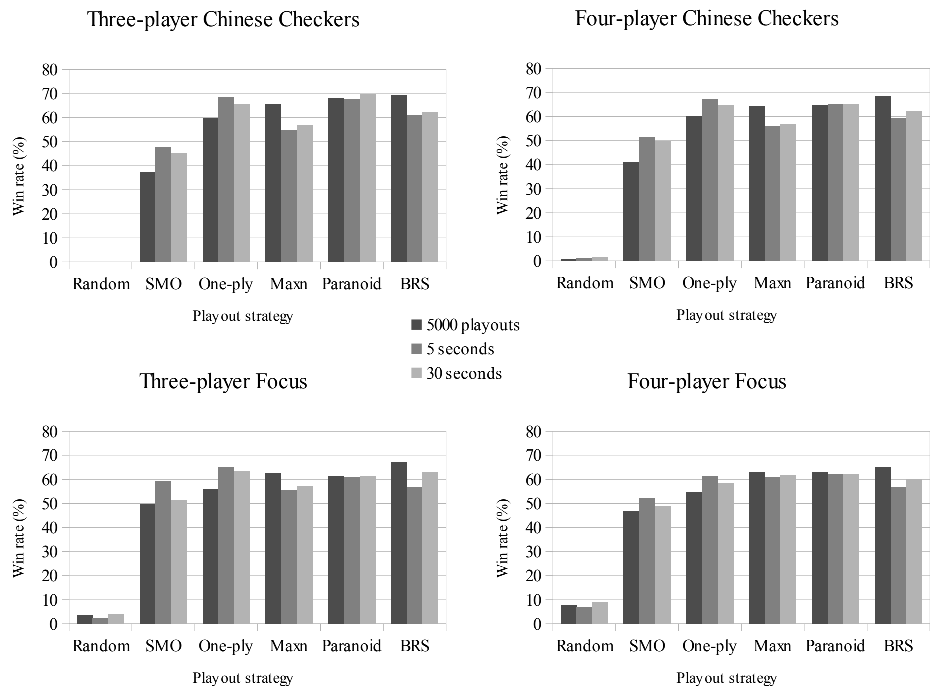


Figure 6.4: Summary of the average win rates of the different playout strategies in the four game variants.

6.4 Chapter Conclusions and Future Research

In this chapter we proposed search-based playouts for improving the playout phase of MCTS in multi-player games. Two-ply max^n, paranoid, and BRS searches were applied to compute the moves to play in the playout phase. Enhancements such as ϵ-greedy playouts, move ordering, killer moves, k-best pruning and tree reusing were implemented to speed up the search.

The results show that search-based playouts significantly improve the quality of the playouts in MCTS. Among the different playout strategies, BRS performs best, followed by paranoid and max^n. This benefit is countered by a reduction of the number of playouts per second. Especially BRS and max^n suffer from this effect. Among the tested two-ply search-based playouts, paranoid overall performs best with both short and long time settings. With more thinking time, the two-ply search-based playout strategies performed relatively better than the one-ply and SMO strategies. This indicates that with longer time settings, more computationally expensive playouts may be used to increase the playing strength of MCTS-based players. Under these conditions, search-based playouts outperforms playouts using light heuristic knowledge in the four-player variant of Focus and the three-player variant of Chinese Checkers. Based on the experimental results we may conclude that search-based playouts for multi-player games may be beneficial if the players receive sufficient thinking time. Though the results in Chapter 4 have shown that MCTS-based techniques perform better in multi-player games than minimax-based techniques, the latter search techniques still play a significant role as they can be applied in the playouts.

There are four directions for future research. First, it may be interesting to test search-based playouts in other games as well. Because of the time-consuming nature of the experiments, they were only performed in three-player and four-player Chinese Checkers and Focus. Second, the two-ply searches may be further optimized. Though a two-ply search will always be slower than a one-ply search, the current speed difference could be reduced further. This can be achieved for instance by improved move ordering or lazy evaluation functions. The third research direction is the application of three-ply searches in the playout. While these playouts are more time consuming than two-ply search-based playouts, they may increase the reliability of the playouts even more. While Winands and Björnsson (2011) noted that three-ply $\alpha\beta$ searches in the MCTS playouts for the two-player game Lines of Action are too expensive, we anticipate that three-ply search-based playouts may be beneficial with increased computing power and enhancements such as parallelizing MCTS (Enzenberger and Müller, 2010). Fourth, it may be interesting to investigate how two-ply search-based playouts perform against game-independent playouts strategies such as Last-Good Reply (Drake, 2009) or Move-Average Sampling Technique (Finnsson and Björnsson, 2008).

CHAPTER 7

MCTS for a Hide-and-Seek Game

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

1. Nijssen, J.A.M. and Winands, M.H.M. (2011b). Monte-Carlo Tree Search for the Game of Scotland Yard. *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games* (eds. S.-B. Cho, S.M. Lucas, and P. Hingston), pp. 158–165, IEEE.
2. Nijssen, J.A.M. and Winands, M.H.M. (2011c). Monte-Carlo Tree Search for the Game of Scotland Yard. *Proceedings of the 23rd Benelux Conference on Artificial Intelligence* (eds. P. De Causmaecker, J. Maervoet, T. Messelis, K. Verbeeck, and T. Vermeulen), pp. 417–418, Ghent, Belgium. Extended abstract.
3. Nijssen, J.A.M. and Winands M.H.M. (2012d). Monte Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 4, pp. 282–294.

The previous chapters discussed the application and enhancement of MCTS to deterministic multi-player games with perfect information. In this chapter, we shift our focus to hide-and-seek games. In this thesis we are interested in hide-and-seek games that have the following three properties. First, they feature imperfect information for some players. Second, some of the players have to cooperate in a fixed coalition. Though these players have a common goal, each player behaves autonomously and explicit communication between the players is not applied. Third, they are asymmetric. The different players have different types of goals. A game that features these properties is *Scotland Yard*. In this multi-player game, five seekers cooperate to try to capture a hider, which only shows its location on regular intervals.

This chapter answers the fourth research question by describing how MCTS can be adapted to tackle these challenges. We first show how to handle imperfect information using different determinization techniques. Also, a new technique, called Location Categorization, is proposed. This technique allows the seekers to make a more reliable prediction for the current location of the hider. Next, to handle the cooperation of the seekers, Coalition Reduction is introduced, which can be used to let the seekers participate in the coalition more effectively. Furthermore, domain knowledge is incorporated in MCTS, by using ϵ-greedy playouts for the seekers and the hider.

This chapter is organized as follows. First, Section 7.1 introduces the game Scotland Yard. Section 7.2 explains how MCTS is adapted for Scotland Yard and which enhancements are proposed. Next, Section 7.3 describes how paranoid search and expectimax are applied in Scotland Yard and how these techniques can be enhanced to improve the playing strength. The experiments and results are given in Section 7.4. Finally, Section 7.5 presents the conclusions based on the results, and possible future research topics.

7.1 Scotland Yard

This section provides an introduction to the game of Scotland Yard. Subsection 7.1.1 gives a brief background on Scotland Yard and in Subsection 7.1.2 the rules are described.

7.1.1 Background

The game of Scotland Yard was introduced in 1983. It was developed by Manfred Burggraf, Dorothy Garrels, Wolf Hörmann, Fritz Ifland, Werner Scheerer and Werner Schlegel. The original version was published by Ravensburger, but the game was also distributed for the English-language market by Milton Bradley.

In 1998, Ravensburger Interactive Media GmbH developed a Scotland Yard program for Microsoft Windows, which was published by Cryo Interactive Entertainment. It did not only feature the original game, but also a computer enhanced version which introduced role-playing game elements. Another Scotland Yard program was developed in 2008. It was developed by Sproing Interactive Media GmbH and published by DTP Young Entertainment GmbH & Co. KG. This version of the game was released for the Nintendo DS. The AI of this program is regarded as quite strong (cf. Frackowski, 2011).

Only a limited amount of research has been performed in Scotland Yard so far. Doberkat, Hasselbring, and Pahl (1996) applied prototype evaluation for cooperative planning and conflict resolution. One of their proposed strategies is applied in the static move evaluator (see Subsection 7.2.2). Furthermore, Sevenster (2008) performed a complexity analysis on Scotland Yard and proved that the generalized version of the game is PSPACE-complete.

7.1.2 Rules

Scotland Yard is played by six players: five seekers, called detectives, and one hider, called Mister X. The game is played on a graph consisting of numbered vertices from 1 to 199. The vertices are connected by four different types of edges representing different transportation types: taxi, bus, underground, and boat. A subgraph of the Scotland Yard map is displayed in Figure 7.1. Each player occupies one vertex and a vertex can hold at most one player. The vertex currently occupied by a player is called the *location* of that player.

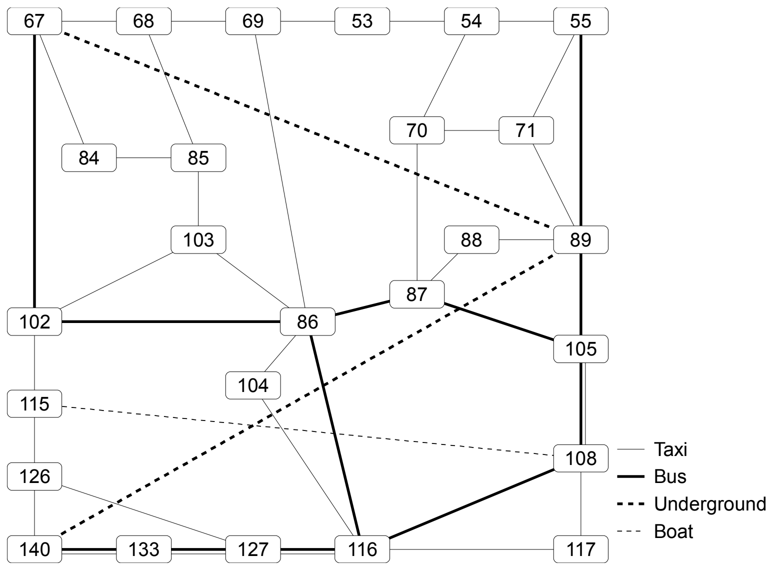


Figure 7.1: A subgraph of the Scotland Yard map.

At the start of the game, all players are placed at one of the eighteen possible pre-defined starting vertices. Each player starts at a different vertex, which is chosen randomly. In total, there are over 13 million different starting positions. Each detective receives ten taxi, eight bus, and four underground tickets. Mister X receives four taxi, three bus, and three underground tickets.

The players move alternately, starting with Mister X. A sequence of six moves, by Mister X and the five detectives, is called one *round*. When performing a move, a player moves along an edge to an unoccupied adjacent vertex, and plays the ticket corresponding to the chosen edge. Mister X receives the tickets played by the detectives. When Mister X plays a ticket, it is removed from the game.

Additionally, Mister X receives five black-fare tickets and two double-move tickets. A black-fare ticket allows him to use any transportation type, including the boat. Along with a regular ticket, Mister X may also play one of his double-move tickets. All detectives then skip their turn for that round.

During the game, Mister X keeps his location secret. Only after moving on rounds 3, 8, 13, 18, and 24 he has to announce his location. When Mister X moves, the detectives always get informed which ticket he used.

The goal for the detectives is to capture Mister X by moving to the vertex occupied by him. The goal for Mister X is to avoid being captured until no detective can perform a move anymore. A detective cannot move if he does not own a ticket which allows him to leave his current location. The maximum number of rounds in Scotland Yard is 24. Draws do not occur in this game.

7.2 MCTS for Scotland Yard

This section discusses how MCTS can incorporate the domain knowledge, imperfect information and cooperating players for Scotland Yard. This section is structured as follows. First, Subsection 7.2.1 explains how the selection and the backpropagation phase are applied in Scotland Yard. Subsection 7.2.2 describes how ϵ-greedy playouts are used to incorporate knowledge in the playout phase. Subsection 7.2.3 explains how determinization can be applied to handle imperfect information. Next, Subsection 7.2.4 discusses how to keep track of the possible locations of the hider. Subsequently, Subsection 7.2.5 proposes Location Categorization to bias the remaining possible locations. Subsection 7.2.6 shows how to handle the fixed coalition of the seekers by using the backpropagation strategy called Coalition Reduction. Finally, Subsection 7.2.7 describes how move filtering allows the hider to use his tickets more efficiently.

7.2.1 Basic MCTS for the Hider and the Seekers

Though Scotland Yard is a game with six players that all behave autonomously, during the MCTS search this game may be considered as a two-player game. This is due to the fact that the seekers have a common goal, and that if one seeker captures the hider the game is considered a win for all seekers. Because of this property, in the selection phase of MCTS the win rate $\bar{x}_i$ at the seekers' plies represents the combined win rate for all seekers. At the hider's plies it represents the win rate of the hider. This means that, for the hider, the MCTS tree is analogous to MCTS with the paranoid search policy (see Chapter 4), because all opponents essentially try to minimize the root player's win rate. In this game, this paranoid assumption is actually correct. However, the hider also assumes that the seekers know where he is located. This is due to the fact that the hider himself knows where he is located. This assumption is incorrect, and may cause the hider to play defensively.

If a seeker has captured the hider during the playout, in the backpropagation phase a score of 1 is backpropagated for all seekers, regardless of which seeker won the game. With Coalition Reduction, which is introduced in Subsection 7.2.6, different values may be backpropagated for the different seekers, dependent on which seeker captured the hider. If the hider manages to escape during the playout, a score of 1 is backpropagated for the hider and a score of 0 for all seekers.

7.2.2 ϵ-greedy Playouts

In the MCTS algorithm, ϵ-greedy playouts are applied to incorporate domain knowledge (Sutton and Barto, 1998; Sturtevant, 2008a). When selecting a move in the playouts, the move is chosen randomly with a probability of ϵ. Otherwise, a heuristic is used to determine the best move. Because Scotland Yard is an asymmetric game, different heuristics have to be defined for the hider and the seekers. Furthermore, separate values for ϵ may be determined for the hider and the seekers in the playout.

For the hider, we define one heuristic to determine the best move. This heuristic, Maximize Closest Distance (MCD), maximizes the number of moves the closest seeker should make to arrive at the target vertex of the move.

For the seekers, we define two different heuristics. The first, Minimize Total Distance (MTD), minimizes the sum of the number of moves the seeker should make to arrive at each possible location of the hider (Doberkat *et al.*, 1996). The second, Chase Actual Location or Chase Assumed Location (CAL), minimizes the number of moves the seeker should make from the target vertex of the move to the (assumed) location of the hider. If this strategy is used by an MCTS seeker, the assumed location of the hider is used, because he does not know the actual location of the hider. This assumed location corresponds to the determinization selected at the start of the playout (see Subsection 7.2.3). If this strategy is employed by the MCTS hider, the actual location of the hider is used. With this strategy, it is implicitly assumed that the seekers know the actual location of the hider.

Each MCTS player should use one heuristic for the hider and one for the seekers. This means that the hider's heuristic has to be combined with one of the two seekers' heuristics. These combinations are named MM (MCD and MTD) and MC (MCD and CAL).

7.2.3 Determinization

In order to deal with imperfect information for the MCTS-based seekers, determinization can be used. The principle behind determinization is that, at the start of each iteration, the hidden information is filled in, while being consistent with the history of the game. Determinization has several theoretical shortcomings. Frank and Basin (1998) defined two problems with determinization. The first is *strategy fusion*, where a suitable strategy for each separate determinization is found, instead of a single strategy that overall works well for all determinizations. The second is *non-locality*, where determinizations are investigated that are unlikely, because opponents may direct play in another direction if they possess information that the player does not have. Russell and Norvig (2002) call determinization 'averaging over clairvoyancy', where players will never try to hide or gain information, because in each determinization, all information is already available. Despite its theoretical shortcomings, it has produced strong results in the past, for instance in the trick-based card game Bridge (Ginsberg, 1999). Ginsberg's Bridge program, called GIB, uses determinization by dealing the cards that have not been played yet among the players while being consistent with the cards played in the previous tricks. This program can play Bridge at an expert level. Determinization has also lead to expert-level game play in the card game Skat (Buro *et al.*, 2009; Long *et al.*, 2010). Furthermore, determinization techniques were successfully applied in various other card games with imperfect information, such as Dou Di Zhu (Powley, Whitehouse, and Cowling, 2011; Whitehouse, Powley, and Cowling, 2011), Magic: The Gathering (Cowling, Ward, and Powley, 2012b), and Spades (Whitehouse *et al.*, 2013).

Other examples of board games where determinization is applied to handle imperfect information include Phantom Go (Cazenave, 2006) and Kriegspiel, the imperfect-information variant of chess (Ciancarini and Favini, 2009). Cazenave (2006) applied determinization in Phantom Go, creating a Monte-Carlo program that was able to defeat strong human Go players. In Kriegspiel, the application of determinization

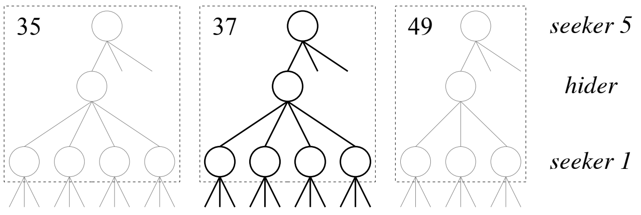


Figure 7.2: Example of a determinization with separate trees. In this example, $L = \{35, 37, 49\}$ and the selected determinization is 37.

did not work well. The MCTS-based player with determinization only played slightly better than a random player. Ciancarini and Favini (2009) provided three reasons why this technique did not work, namely (1) the assumed positions of the opponent's pieces were unrealistic, (2) the underestimation of the opponent's ability to coordinate an attack, and (3) in Kriegspiel there is no built-in notion of progression, contrary to games such as Go, Scrabble and Poker.

In Scotland Yard, the hidden information consists of the location of the hider. Based on the last surface location and the tickets the hider played since then, the seekers can deduce a list of possible locations of the hider, called L (see Subsection 7.2.4).

At the start of each MCTS iteration, an assumption is made about the location of the hider. By default, this location is chosen from L in a uniform random manner. This assumption is used throughout the whole iteration. There are two approaches to build and traverse the search tree. The first approach is by generating a separate tree for each determinization. This technique is similar to *determinized UCT* described by Cowling, Powley, and Whitehouse (2012a). In each tree, only the hider's moves that are consistent with the corresponding determinization are generated. An example is given in Figure 7.2. After selecting a determinization at the root node, the corresponding tree is traversed. In the end, there are two approaches to select the best move. The first is majority voting (Soejima, Kishimoto, and Watanabe, 2010). Each candidate move receives one vote from each tree where it is the move that was played most often. The candidate move with the highest number of votes is selected as the best move. If more moves are tied, the move with the highest number of visits over all trees is selected. The second is averaging over all search trees (Cazenave and Jouandeau, 2007). The move with the highest average number of visits over all trees is selected as the best move.

The second approach is using single-tree determinization (Nijssen and Winands, 2012d). When generating the tree, at the hider's ply, all possible moves from all possible locations are generated. When traversing the tree, only the moves consistent with the current determinization are considered. The advantage of this technique is that information is shared between different determinizations, increasing the amount of usable information. This type of determinization is similar to Single-Observer Information Set Monte-Carlo Tree Search, which was proposed by Cowling *et al.* (2012a) around the same time as single-tree determinization. An example is given in Figure 7.3.

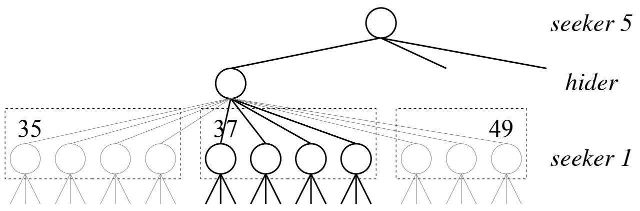


Figure 7.3: Example of a determinization with a single tree. In this example, $L = \{35, 37, 49\}$ and the selected determinization is 37.

7.2.4 Limiting the Possible Locations

It is possible for the seekers to limit the list of possible locations by removing the vertices where the hider cannot be located. The list of possible locations, L, is updated every move. When the hider plays a ticket, the new list of possible locations L_{new} is calculated, based on the old list of possible locations L_{old}, the current locations of the seekers Δ, and the ticket t played by the hider, using Algorithm 7.1. S is the set of rounds when the hider surfaces. At the start of the game, L is initialized with the 18 possible starting locations, excluding the five starting locations of the seekers. In this algorithm, the method `targets`(p, t) returns the list of locations reachable from location p using ticket t. When the hider surfaces, `location(hider)` is the vertex he surfaced at. When a seeker makes a move, the target vertex of this move is excluded from L, provided this vertex was a possible location and the hider was not captured.

Algorithm 7.1 Pseudo code for computing the list of possible locations of the hider.

```
$L_{new} \leftarrow \emptyset$
**if** currentRound $\in S$ **then**
    $L_{new} \leftarrow$ location(hider)
**else**
    **for all** $p \in L_{old}$ **do**
        $T \leftarrow$ targets$(p, t)$
        $L_{new} \leftarrow L_{new} \cup (T \setminus \Delta)$
    **end for**
**end if**
**return** $L_{new}$
```

An example of this computation is given in Figure 7.4 using the subgraph of the map in Figure 7.1. Assume the hider surfaces at location 86 in round 8 and the seekers move to locations 85, 115, 89, 116, and 127, respectively. When the hider plays a black-fare ticket in round 9, the new list of possible locations becomes $L = \{69, 87, 102, 103, 104\}$. Location 116 is also reachable from 86 with a black-fare ticket, but because this location belongs to D, i.e., there is a seeker on that location, it is not added to L. After seeker 1 moves to 103 in round 9, this location is removed from the list of possible locations. After seeker 2 moves to 102, this location is removed as well. In round 10, the hider plays a taxi ticket. The locations reachable from location

69 are 53, 68, and 86 and are all added to L. The locations reachable by taxi from 87 are 70 and 88. Because 88 belongs to D, only 70 is added to N. The two locations reachable from 104 are 86 and 116. Because 86 already belongs to L and 116 belongs to D, neither location is added to L.

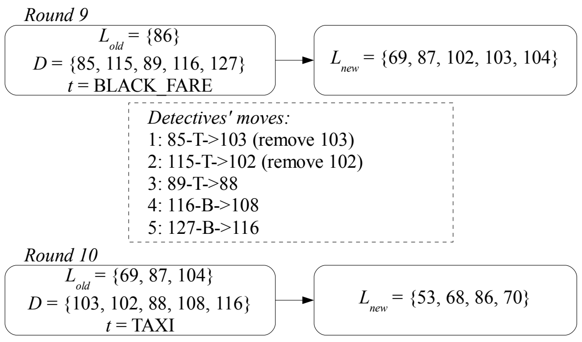


Figure 7.4: Example of a computation of possible locations.

Figure 7.5 gives an indication of the number of possible locations of the hider for each round. These numbers were obtained from the actual game and the playouts in 50 matches between MCTS-based players. The peak at round 2 is the highest. At this point, the hider has performed two moves without surfacing yet. The peaks at rounds 12 and 17 are smaller than the peak at round 7. This is because at round 7, the seekers have not closed in yet on the hider. Once the seekers close in on the hider, the number of possible locations becomes smaller. Note that if the possible locations are not limited, this number is always 194.

7.2.5 Location Categorization

Some of the possible locations computed in Algorithm 7.1 are more probable than others. The performance of the seekers can be improved by biasing the possible locations of the hider. This technique is called Location Categorization. The possible locations in L are divided into categories that are numbered from 1 to c, where c is the number of categories. Each possible location is assigned to exactly one category, so that the categorization is a partition of L. The type of categorization is domain dependent. For Scotland Yard, three different types of categorization are investigated:

(1) Minimum-distance. A categorization is made based on the distance of the possible location to the nearest seeker. The category number equals the number of moves this seeker has to perform to reach the possible location. For this categorization, we set c to 5. To accommodate for locations with a minimum distance larger than 5, all locations with a minimum distance of 5 or more are grouped into the same category. The idea behind this categorization is that the possible locations near the seekers are investigated less often, because it is unlikely that the hider is close to one of the seekers. The hider could try to exploit this behavior, though it is risky, offsetting a possible benefit.

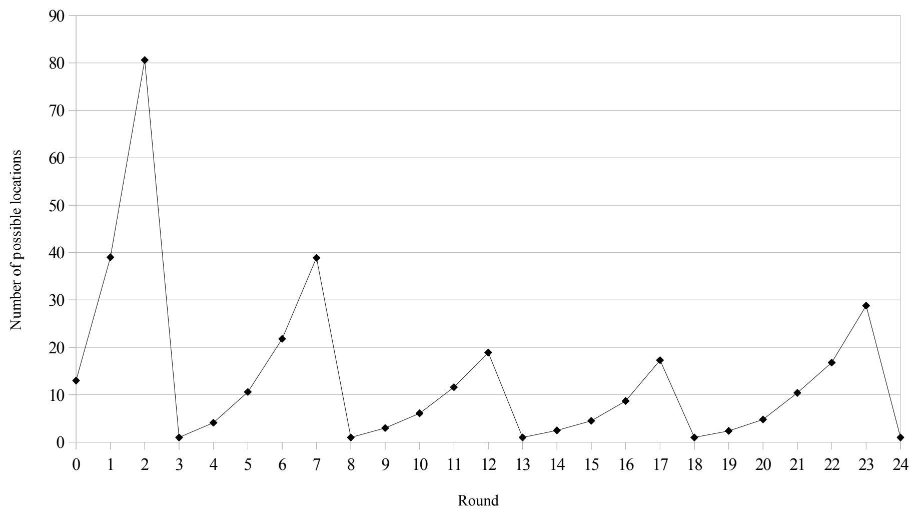


Figure 7.5: Average number of possible locations of the hider for each round.

(2) Average-distance. A categorization is made based on the average distance of all seekers to the possible location. This number is rounded down. The category number equals the average number of moves the seekers have to travel to reach the possible location. Similar to the minimum-distance categorization, the parameter c is set to 5. This means that all locations with an average distance of 5 or more are grouped into category 5.

(3) Station. A categorization is made based on the transportation types connected to the possible location. We distinguish 4 different station types, which means that $c = 4$. Locations with only taxi edges belong to category 1, locations with taxi and bus edges belong to category 2, locations with taxi, bus, and underground edges belong to category 3, and all locations with at least one boat edge belong to category 4.

After the hider performs a move the possible locations are divided into the different categories, based on the preselected categorization.

For each category, a weight has to be determined to indicate the probability that a location of a certain category is chosen, so that a biased, non-uniform preference distribution can be imposed over L. If available, this statistic may be obtained from game records of matches played by expert players. In this thesis, the statistics are gathered by a large number of selfplay games. These statistics can later be used by the seekers to determine the weights of the categories. This approach is useful when the opponent is unknown and there are not enough games to gather a sufficient amount of information.

Table 7.1 Example of a general table with the minimum-distance categorization after playing 1000 games.

Category	1	2	3	4	5
a	2454	9735	4047	1109	344
n	12523	14502	7491	2890	1756

Table 7.2 Example of a detailed table with the minimum-distance categorization after playing 1000 games.

Category	1	2	3	4	5
Combination					
1	1542	-	-	-	-
2	-	2801	-	-	-
1,2	666	4776	-	-	-
3	-	-	977	-	-
1,3	14	-	252	-	-
2,3	-	67	208	-	-
1,2,3	210	1558	1642	-	-
4	-	-	-	262	-
2,3,4	-	23	39	90	-
1,2,3,4	18	224	263	179	-
2,3,4,5	-	57	191	183	88
1,2,3,4,5	2	210	448	307	164

There are two different ways to store the statistics about the possible categories. In the *general* table, for each category both the number of times one or more possible locations belonged to the category, n, and the number of times the actual location of the hider belonged to the category, a, are stored. This way of storing and using statistics is similar to the transition probabilities used in Realization Probability Search, which was successful in Shogi (Tsuruoka, Yokoyama, and Chikayama, 2002), Amazons (Higashiuchi and Grimbergen, 2006), and Lines of Action (Winands and Björnsson, 2008). An example of the general table is given in Table 7.1. In the *detailed* table, for each possible combination of categories, i.e., the union of all categories over L, we store how many times the actual location of the hider belonged to each category. An example is given in Table 7.2. This table only shows the category combinations (i.e., rows) that occurred at least 100 times. For instance, category combination $(2,3,4)$, where L contains locations in categories 2, 3, and 4, but not in 1 and 5, occurred 152 times, where the hider was 23 times on a location of category 2, 39 times on a location of category 3, and 90 times on a location of category 4.

The seekers use a vector of length c to select a location for the hider at the start of each MCTS iteration. The values in this vector represent the weights of the categories. When using the general table, this vector consists of the values $[w_1, w_2, \cdots, w_c] = [\frac{a_1}{n_1}, \frac{a_2}{n_2}, \cdots, \frac{a_c}{n_c}]$. When using the detailed table, the vector corresponding to the combination of categories is directly extracted from the table. For instance, using Table 7.2, if L contains locations belonging to categories 2, 3, and 4, the vector would be [0,23,39,90,0]. If the total number of occurrences of this combination of categories is below a certain threshold, for instance 100, the table is not used and the possible locations are randomly chosen. This only occurs on rare occasions.

There are two different ways the vector can be used to select a possible location. When using *one-step* selection, each possible location gets a probability to be selected. Roulette-wheel selection (i.e., sampling from a non-uniform distribution) is used to select a possible location. The size of each possible location on the wheel is corresponding to the value of its category in the vector. The probability to choose location l is calculated by using Formula 7.1.

$$P(l) = \frac{w_{c_l}}{\sum\limits_{m \in L} w_{c_m}} \tag{7.1}$$

In this formula, w_{c_l} and w_{c_m} represent the weights of the category to which locations l and m belong, respectively.

When using *two-step* selection, each location category gets a probability to be selected. Roulette-wheel selection is used to select a category. The size of each category on the wheel is corresponding to its value in the vector. After selecting a category, one of the possible locations from this category is randomly chosen. The probability of choosing location l using two-step selection is calculated by using Formula 7.2.

$$P(l) = \frac{w_{c_l}}{|c_l| \sum\limits_{j=1}^{c} w_j} \tag{7.2}$$

In this formula, $|c_l|$ represents the number of possible locations that belong to the category of location l and w_j represents weight of category j.

We remark that Location Categorization uses a 'big-data' approach to set the weights. Such an approach, which obtains its statistics from a large set of games, played by humans or computers, has been successful in Othello (Buro, 2000), Shogi (Tsuruoka *et al.*, 2002), Amazons (Higashiuchi and Grimbergen, 2006), and Lines of Action (Winands and Björnsson, 2008). Machine-learning techniques, though less trivial, may also be used to further tune them.

7.2.6 Coalition Reduction

Scotland Yard is a cooperative multi-player game. Therefore, the seekers can be considered as one player, reducing the game to a two-player game. If in a playout one seeker captures the hider, the playout is considered a win for all seekers and the result is backpropagated accordingly. However, when using this backpropagation rule we observed that seekers during game play sometimes relied too much on the other seekers and did not make any efforts to capture the hider. For solving this problem, we propose Coalition Reduction. If the seeker who is the root player captures the hider, a score of 1 is returned. If another seeker captures the hider, a smaller score, $1-r$, is returned, where $r \in [0,1]$. For this enhancement to work efficiently, it is important to tune the value of r. If the value of r is too small, seekers have the tendency to become less involved. If their own position is not good, i.e., they are far away from the possible locations of the hider, they tend to rely on the other seekers too much. If the value of r is too large, the seekers become too selfish and do not cooperate anymore. In Subsection 7.4.6, this parameter is experimentally fine-tuned.

7.2.7 Move Filtering

The hider only has a limited number of black-fare and double-move tickets, so he should use them wisely. Black-fare tickets should only be used by the hider to increase the uncertainty about his location or to travel by boat, and double-move tickets are mostly used for escaping from dire positions.

Three straightforward game-specific knowledge rules regarding the use of black-fare tickets to prevent the hider from squandering them have been implemented. The hider is not allowed to use black-fare tickets in the following three situations: (1) during the first two rounds, (2) during a round when he has to surface, or (3) when all possible locations only have taxi edges. In the first situation, there is already a large uncertainty about the hider's location. In the second and third situation, using a black-fare ticket does not increase the uncertainty about the hider's location compared to using a 'normal' ticket. An exception is when the hider is located on a vertex with a boat connection. In this case, the hider may always use a black-fare ticket.

Double-move tickets are only used when the hider can be captured by one of the seekers. If all possible moves in the root node lead to a vertex that can be reached by one of the seekers in the next round, a double move ticket is added to the moves. If there is at least one 'safe' move, then double-move tickets are not added. If the search still selects a move that allows one of the seekers to capture the hider in the next round, a double-move ticket is added to the selected move. Of course, a double-move ticket can only be added if the hider has at least one of these tickets remaining.

Move filtering is a considerable improvement for the hider. Experiments in a previous version of the program revealed that this enhancement increases the win rate of an MCTS hider from 19.4% ± 1.6 to 34.0% ± 1.9 against MCTS seekers. These results show that the performance of the hider improves considerably when he is prevented from squandering black-fare tickets. It is important for the hider to effectively use his limited stock of black-fare tickets (Nijssen and Winands, 2011b).

7.3 Minimax-Based Techniques for Scotland Yard

This section gives an overview on how minimax-based search techniques are applied for playing Scotland Yard as the hider and the seekers. Subsections 7.3.1 and 7.3.2 explain how paranoid search and expectimax are implemented for the hider and the seekers, respectively. In Subsection 7.3.3 a description of the evaluation functions is given. Finally, Subsection 7.3.4 explains how Location Categorization can be used with expectimax.

7.3.1 Paranoid Search for the Hider

Figure 7.6 shows the general structure of the search tree built by the hider. The hider is the MAX player, while the seekers are the MIN players. The structure of the tree is similar to a paranoid search tree used in multi-player games (Sturtevant and Korf, 2000). However, in this case the paranoid assumption is correct because the opponents do have a coalition against the current player. On the other hand, the paranoid hider also assumes that the seekers know where he is located. Similar to the MCTS hider, this assumption is incorrect.

As discussed in Chapter 2, in a paranoid search tree, $\alpha\beta$ pruning is possible. However, in the best case the tree can only be reduced to $O(b^{\frac{N-1}{N}d})$ (Sturtevant and Korf, 2000). This means that for Scotland Yard, the tree can be reduced to $O(b^{\frac{5}{6}d})$ in the best case. To increase the amount of pruning in the search tree, killer moves and the history heuristic are applied. Furthermore, move filtering is used to allow the hider to make use of its black fare and double move tickets more efficiently.

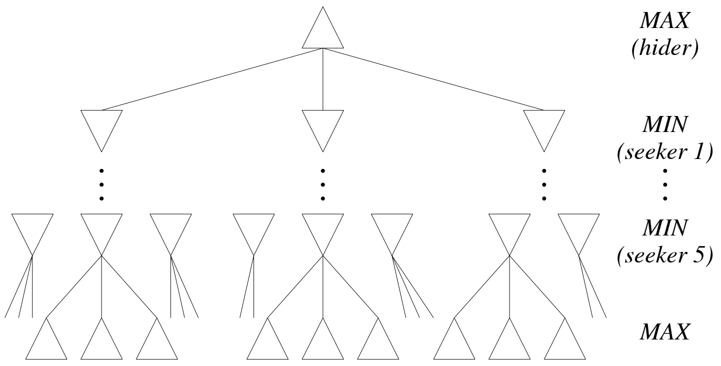


Figure 7.6: Structure of the paranoid tree for the hider.

7.3.2 Expectimax for the Seekers

To handle imperfect information in the minimax framework, expectimax (Michie, 1966) may be used. Billings *et al.* (2006) used variations of expectimax, *Miximax* and its generalization *Miximix*, to play poker. Schadd, Winands, and Uiterwijk (2009) applied expectimax for playing the game Stratego. They enhanced the search with *ChanceProbCut*, which allows forward pruning in chance nodes. Expectimax may also be employed to handle imperfect information for the seekers in Scotland Yard.

Figure 7.7 shows the structure of the expectimax search tree built by the seekers. At the second layer, the chance nodes are located. The edges leaving these nodes represent the possible locations of the hider (i.e., possible determinizations). Each possible location is weighted equally, though it is possible to bias the weights to give more reliable results (see Subsection 7.3.4).

Another notable feature is that the seeker does not incorporate the other seekers in the search tree, i.e., the other seekers do not move. This has three advantages.

(1) More pruning is possible. Because the game is reduced to two players, the size of the tree can be reduced to $O(|L| \times b^{\frac{d}{2}})$, where $|L|$ is the number of possible locations. The amount of pruning is further increased by applying killer moves and the history heuristic. (2) The seeker keeps actively participating in the game, instead of relying on other seekers. (3) The seeker achieves more long-term planning by investigating more MAX nodes. This is analogous to Best-Reply Search (Schadd and Winands, 2011; see Chapter 4) for multi-player games, where the moves of all subsequent opponents are reduced to one ply. A disadvantage of this reduction is that the seekers do not consider the other seekers and thus cooperation is not contemplated. Experiments with 1000 ms of thinking time per move revealed that reducing the tree produced a win rate of 40.6% ± 3.0 (see Table 7.7) against an MCTS hider, while without this reduction the win rate is 5.2% ± 1.4.

This tree reduction technique can also be applied to MCTS. However, experiments with 10,000 playouts per move showed that it decreases the win rate considerably, from 63.6% ± 3.0 (see Table 7.5) to 34.3% ± 2.9 against an MCTS hider. This is because the advantages for expectimax do not apply in MCTS. There are three reasons for this. (1) $\alpha\beta$-like pruning is not applicable for MCTS. (2) Reliance on other seekers is already smaller due to the random moves in the ϵ-greedy playouts. Coalition Reduction reduces this reliance even more. (3) Due to the playouts the seekers can already look further ahead.

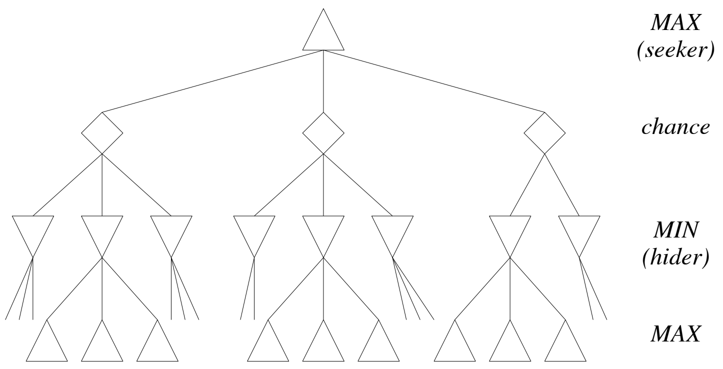


Figure 7.7: Structure of the expectimax tree for the seekers.

7.3.3 Evaluation Functions

For both the hider and the seekers, an evaluation function is necessary to evaluate the leaf nodes of the search tree. For the paranoid hider, an evaluation function is used that is based on the MCD playout strategy. First, the hider should stay as far away from the nearest seeker as possible. Second, the hider should save black-fare tickets, unless he can increase the uncertainty about his location. The leaf nodes of the paranoid search tree for the hider are valued using Formula 7.3.

$$s_{hider} = 100 \times \min_{i \in D}(d_{hider,i}) + 10 \times t_{hider,BF} + |L| + \rho \tag{7.3}$$

Here, $d_{hider,i}$ is the distance from the location of the hider to the location of seeker i. $t_{hider,BF}$ represents the number of black fare tickets the hider has left. $|L|$ indicates the number of possible locations of the hider. ρ is a small random value between 0 and 1. Similar to MCD, the hider assumes that the seekers know his actual location. This may result in the hider playing too cautiously.

For the expectimax seekers, an evaluation function is used that is similar to the MTD (Doberkat *et al.*, 1996) playout strategy used in MCTS. The seekers try to minimize the sum of the distances to all possible locations. The leaf nodes of the expectimax search tree of seeker i are evaluated by using Formula 7.4.

$$s_i = -\sum_{l \in L} d_{i,l} + \rho \tag{7.4}$$

Here, $d_{i,l}$ is the distance from the location of seeker i to possible location l. Because we want to minimize this distance, we take the negative of the sum. Again, ρ is a random value between 0 and 1.

7.3.4 Location Categorization for Expectimax

Similar to MCTS, Location Categorization can be used in the expectimax framework to bias the search towards more likely locations. Usually, in the chance level of the search tree for Scotland Yard, each location has an equal weight. By applying Location Categorization, more likely locations receive a larger weight than unlikely ones. The weight $P(i)$ of the node representing location i is calculated by using Formula 7.5.

$$P(i) = \frac{w_{c_i}}{\sum_{l \in L} w_{c_l}} \tag{7.5}$$

In this formula, w_{c_i} and w_{c_l} represent the weights of the category to which locations i and l belong, respectively. This formula is similar to Formula 7.1, which is used for one-step selection.

7.4 Experiments and Results

This section first provides an overview of the experimental setup in Subsection 7.4.1. Subsection 7.4.2 presents the results of the experiments with ϵ-greedy playouts for the MCTS players. The determinization techniques for MCTS are compared in Subsection 7.4.3. Subsection 7.4.4 gives an overview of the performance of the MCTS seekers with Location Categorization. Next, Subsection 7.4.5 shows how Location Categorization influences the performance of the expectimax seekers. Subsection 7.4.6 presents how the MCTS seekers with Coalition Reduction perform. In Subsection 7.4.7, a comparison between the MCTS and minimax-based players is provided. Finally, Subsection 7.4.8 gives an overview of how MCTS performs against the Scotland Yard program on the Nintendo DS.

7.4.1 Setup

The engines for Scotland Yard and the AI players are written in Java. For the MCTS-based hider and seekers, C is set to 0.5. Progressive History (see Chapter 5) is used for both the hider and the seekers, with $W = 5$ for both player types. These values were achieved by systematic testing with $C \in [0.1, 2]$ and $W \in [0, 25]$. Progressive History does not significantly increase the performance of the hider (from 52.2% ± 3.1 with $W = 0$ to 53.4% ± 3.1 with $W = 5$ against MCTS-based seekers), while it does improve the playing strength of the seekers considerably (from 47.8% ± 3.1 with $W = 0$ to 63.4% ± 3.0 with $W = 5$ against an MCTS-based hider). The hider uses move filtering and the seekers use single-tree determinization. All MCTS players use 10,000 playouts for selecting the best move, except when stated otherwise. The expectimax and paranoid players receive 1000 ms of thinking time for each move. In all experiments, 1000 games are played to determine the win rate. The win rates are given with a 95% confidence interval. The experiments are run on a cluster consisting of AMD64 Opteron 2.4 GHz processors. Depending on the settings, one game takes approximately 2 to 4 minutes to finish.

7.4.2 ϵ-greedy Playouts

In the first set of experiments we determine the influence of ϵ-greedy playouts (MM and MC) on the playing strength of the MCTS hider and the seekers. Because the different playout strategies have different influences on the number of playouts per second (cf. Chapter 6), the thinking time of the players is limited on time instead of samples. Due to the asymmetric nature of Scotland Yard, different values of ϵ for the hider and the seekers in the playouts may also be used. Systematic testing showed that the best results are achieved with $\epsilon = 0.1$ for the hider and $\epsilon = 0.2$ for the seekers.

Table 7.3 presents the win rates for the seekers with the two different ϵ-greedy playout strategies and with random playouts (R) for different time settings (1000 ms, 2500 ms, and 5000 ms per move). The results show that for the MCTS hider, MM is the best playout strategy, while for the MCTS seekers, MC works best. This shows that for the MCTS hider, it is best to assume that the seekers do not know his actual location during the playouts. The MTD strategy of the seekers prevents the hider from becoming too paranoid. For the MCTS seekers, however, it is best to use the assumed location of the hider. Apparently, it is best for the MCTS seekers to have a clear goal during the playouts.

Furthermore, the results show that ϵ-greedy playouts are a major improvement for both the MCTS hider and the MCTS seekers over random playouts. For example, with a thinking time of 5000 ms, the win rate of the seekers with the MC playout strategy increases from 59.8% ± 3.0 to 79.5% ± 2.5 against the hider with random playouts. For the MCTS hider, the win rate increases from 40.2% ± 3.0 to 58.2% ± 3.1 with the MM playout strategy against MCTS seekers with random playouts. Similar results are achieved with 1000 and 2500 ms thinking time. The results also suggest that the MCTS seekers have a considerable advantage over the MCTS hider, which may be explained by the asymmetric nature of the game.

For the remainder of the experiments, the MC strategy for the MCTS seekers and the MM strategy for the MCTS hider is used.

Table 7.3 Win rates of the MCTS seekers against the MCTS hider with and without ϵ-greedy playouts for different time settings.

Thinking time: 1000 ms

		hider		
		MM	MC	R
	MM	60.3% ± 3.0	65.1% ± 3.0	72.9% ± 2.8
seekers	MC	73.5% ± 2.7	74.6% ± 2.7	78.1% ± 2.6
	R	39.7% ± 3.0	42.4% ± 3.1	49.0% ± 3.1

Thinking time: 2500 ms

		hider		
		MM	MC	R
	MM	64.4% ± 3.0	72.1% ± 2.8	72.2% ± 2.8
seekers	MC	74.9% ± 2.7	79.2% ± 2.5	79.9% ± 2.5
	R	55.8% ± 3.1	52.4% ± 3.1	59.2% ± 3.0

Thinking time: 5000 ms

		hider		
		MM	MC	R
	MM	68.0% ± 2.9	74.3% ± 2.7	77.5% ± 2.6
seekers	MC	74.0% ± 2.7	82.4% ± 2.4	79.5% ± 2.5
	R	41.8% ± 3.1	56.6% ± 3.1	59.8% ± 3.0

7.4.3 Determinization

In the previous experiments, determinization was applied with a single tree. This set of experiments verifies whether this technique works better than using a separate tree for each determinization. These experiments are performed with two different playout strategies. In the first, both players apply ϵ-greedy playouts, while in the second they both use random playouts. Systematic testing showed that for separate trees the same values for C and ϵ are also optimal for the single tree. The results are summarized in Table 7.4. The upper part of the table shows that single-tree determinization gives the highest win rate with a fixed number of playouts. Especially when using ϵ-greedy playouts, this technique performs considerably better than separate trees. When using separate trees, majority voting performs significantly better than using the average score, i.e., selecting the move with the highest average number of visits. We remark that using a single tree generates more overhead, because at the hider's ply, the moves have to be checked whether they are consistent with the selected determinization. This overhead, however, is relatively small. When taking this overhead into account, the difference between single-tree determinization and separate trees hardly changes. This may be concluded from the results presented in the lower part of the table, where the thinking time is limited to 1000 ms per move, instead of providing a fixed number of playouts.

These results are similar to the results found by Cowling *et al.* (2012a). They showed that Information Set MCTS, which is similar to single-tree determinization, performs better than determinized UCT, which is similar to separate-tree determinization, by a significant margin in the modern board game Lord of the Rings: The Confrontation.

Table 7.4 Win rates of the MCTS seekers with different determinization approaches against the MCTS hider. Both player types use either ϵ-greedy or random playouts.

10,000 playouts per move

	Playouts	
Determinization	ϵ-greedy	Random
Single tree	63.6% ± 3.0	51.8% ± 3.1
Separate trees	31.3% ± 2.9	31.2% ± 2.9
Separate trees + majority voting	35.1% ± 3.0	37.5% ± 3.0

1000 ms per move

	Playouts	
Determinization	ϵ-greedy	Random
Single tree	73.5% ± 2.7	54.7% ± 3.1
Separate trees	37.1% ± 3.0	38.5% ± 3.0
Separate trees + majority voting	39.9% ± 3.0	40.1% ± 3.0

7.4.4 Location Categorization for MCTS

The next set of experiments checks which combination of categorization, table type and number of selection steps works best when using Location Categorization. The statistics for the general and detailed table are gathered by letting MCTS seekers play 1000 games against an MCTS hider. The results are summarized in Table 7.5. We let MCTS seekers with Location Categorization play against an MCTS hider. For reference, the seekers without Location Categorization win 63.6% ± 3.0 of the games against the hider. The win rate of the seekers without Location Categorization is denoted as the default win rate. The results in Table 7.5 show that the minimum-distance categorization works best. For this categorization, there is no large difference between the table types and the number of selection steps.

To test the robustness of this technique, the MCTS seekers with Location Categorization play against a different type of hider, namely the paranoid hider. The same weights that were used against the MCTS hider are used. Because in the previous set of experiments, it turned out that the minimum-distance categorization works best, we only use this categorization in this set of experiments. The results are given in Table 7.6. The results show that Location Categorization also significantly improves the performance of the seekers against a different type of opponent. For all settings a significantly better performance is achieved.

Table 7.5 Win rates of the MCTS seekers with Location Categorization against the MCTS hider.

Categorization	Table	Steps	Win rate
Minimum-distance	General	1	67.7% ± 2.9
Minimum-distance	General	2	66.3% ± 2.9
Minimum-distance	Detail	1	63.5% ± 3.0
Minimum-distance	Detail	2	65.6% ± 2.9
Average-distance	General	1	61.7% ± 3.0
Average-distance	General	2	59.6% ± 3.0
Average-distance	Detail	1	63.9% ± 3.0
Average-distance	Detail	2	63.6% ± 3.0
Station	General	1	58.6% ± 3.1
Station	General	2	58.0% ± 3.1
Station	Detail	1	57.9% ± 3.1
Station	Detail	2	58.5% ± 3.1
Default win rate: 63.6% ± 3.0			

Table 7.6 Win rates of the MCTS seekers with Location Categorization against the paranoid hider.

Categorization	Table	Steps	Win rate
Minimum-distance	General	1	86.4% ± 2.1
Minimum-distance	General	2	86.3% ± 2.1
Minimum-distance	Detail	1	87.5% ± 2.0
Minimum-distance	Detail	2	86.6% ± 2.1
Default win rate: 83.4% ± 2.3			

7.4.5 Location Categorization for Expectimax

We also test how Location Categorization increases the performance of the expectimax seekers. This enhancement is tested with the same three categorizations as in the MCTS experiments, including the same weights for the categories. First, the expectimax seekers play against an MCTS hider. The results are given in Table 7.7. Both player types receive 1000 ms of thinking time. The results show that Location Categorization also works in the expectimax framework. Similar to MCTS, the minimum-distance categorization performs best, increasing the win rate against the MCTS hider from 40.6% ± 3.0 to 50.2% ± 3.1 when using the general table.

Location Categorization in the expectimax framework is also tested against a paranoid hider. We remark that the weights of the categories are gained with a different type of seekers against a different type of hider. The results of this set of experiments are displayed in Table 7.8. It appears that the detailed table gives better results than the general table and that the minimum-distance categorization is still the best categorization. With these configurations, expectimax with Location Categorization wins 79.1% ± 2.5, compared to 74.1% ± 2.7 for expectimax without this enhancement. This confirms that Location Categorization is a robust technique.

Table 7.7 Win rates of the expectimax seekers with Location Categorization against the MCTS hider.

Categorization	Table	Win rate
Minimum-distance	General	50.2% ± 3.1
Minimum-distance	Detail	44.2% ± 3.1
Average-distance	General	41.4% ± 3.1
Average-distance	Detail	40.3% ± 3.0
Station	General	38.2% ± 3.0
Station	Detail	39.9% ± 3.0
Default win rate: 40.6% ± 3.0		

Table 7.8 Win rates of the expectimax seekers with Location Categorization against the paranoid hider.

Categorization	Table	Win rate
Minimum-distance	General	76.3% ± 2.6
Minimum-distance	Detail	79.1% ± 2.5
Average-distance	General	71.5% ± 2.8
Average-distance	Detail	77.3% ± 2.6
Station	General	69.3% ± 2.9
Station	Detail	65.1% ± 3.0
Default win rate: 74.1% ± 2.7		

7.4.6 Coalition Reduction

To test the performance of the MCTS seekers with Coalition Reduction, we let them play against different hiders with different values of r. For the seekers, the performance of Coalition Reduction is tested with and without Location Categorization enabled. To verify that this enhancement also works against another type of hider, matches against a paranoid hider are played as well. Finally, the performance of the seekers with Coalition Reduction with different time settings is tested. We remark that for $r = 0$, Coalition Reduction is disabled. For $r = 1$, there is no coalition, and all seekers only work for themselves. The results are presented in Table 7.9 and Figure 7.8. The seekers achieve the highest win rate with $r = 0.1$. The win rate increases from 63.6% ± 3.0 to 70.1% ± 2.8. With Location Categorization, the win rate even increases further, from 67.7% ± 2.9 to 76.2% ± 2.6. Also with $r = 0.2$ and $r = 0.3$ the seekers with Coalition Reduction play at least as strong than without this enhancement. If r is increased further, the performance of the seekers drops significantly. With these settings, the seekers no longer cooperate well to strategically close in on the hider, allowing the hider to escape rather easily. If there is no cooperation at all, the win rate of the seekers drops to 22.6% ± 2.6. To validate these numbers, Coalition Reduction is also tested against the paranoid hider. Again, the best results are achieved with $r = 0.1$, so it turns out that this is a good value that works well against different hiders. The results also show that MCTS with Location Categorization constantly performs better than regular MCTS. Again, this shows that Location

Table 7.9 Win rates of MCTS seekers with Coalition Reduction for different values of r against different hiders.

	Seekers:	MCTS	MCTS + LC	MCTS	MCTS + LC
	Hider:	MCTS	MCTS	Paranoid	Paranoid
	0	63.6% ± 3.0	67.7% ± 2.9	83.4% ± 2.3	87.5% ± 2.1
	0.1	70.1% ± 2.8	76.2% ± 2.6	90.2% ± 1.8	92.9% ± 1.6
	0.2	65.3% ± 3.0	74.2% ± 2.7	88.0% ± 2.0	92.5% ± 1.6
	0.3	64.3% ± 3.0	72.1% ± 2.8	84.3% ± 2.3	91.0% ± 1.8
	0.4	54.9% ± 3.1	64.9% ± 3.0	82.1% ± 2.4	88.9% ± 1.9
r	0.5	51.0% ± 3.1	65.0% ± 3.0	79.1% ± 2.5	86.0% ± 2.2
	0.6	43.5% ± 3.1	52.9% ± 3.1	72.9% ± 2.8	83.0% ± 2.3
	0.7	42.6% ± 3.1	47.5% ± 3.1	70.4% ± 2.8	74.9% ± 2.7
	0.8	34.4% ± 2.9	39.7% ± 3.0	63.9% ± 3.0	69.4% ± 2.9
	0.9	30.7% ± 2.9	32.3% ± 2.9	57.2% ± 3.1	63.2% ± 3.0
	1	22.6% ± 2.6	23.1% ± 2.6	49.7% ± 3.1	50.7% ± 3.1

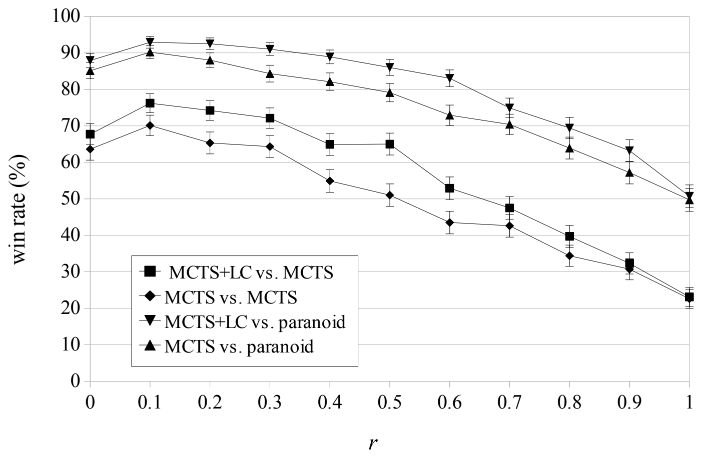


Figure 7.8: Graphical representation of Table 7.9.

Categorization is a robust improvement for MCTS.

Finally, Coalition Reduction is tested with different time settings. In addition to 10,000 samples per move, the MCTS seekers with Location Categorization and the MCTS hider are provided with 1000, 2500 and 100,000 samples per move. The results are given in Table 7.10 and Figure 7.9. With 1000 samples per move for the hider and the seekers, better results are achieved with a higher value of r. With r = 0.5, a win rate of 55.6% ± 3.1 is achieved. When providing 2500 samples, the seekers achieve the highest win rate with r = 0.2 and r = 0.3. With these time settings they win 64.5% ± 3.0 and 64.1% ± 3.0 of the games, respectively. If 100,000 samples per move are provided for both player types, Coalition Reduction is still a significant improvement with r = 0.1. The results are similar to 10,000 samples per

Table 7.10 Win rates of MCTS seekers with Coalition Reduction for different values of r with different time settings against an MCTS hider.

	Playouts:	1000	2500	10,000	100,000
	0	34.5% ± 3.0	48.3% ± 3.1	67.7% ± 2.9	77.4% ± 2.6
	0.1	43.0% ± 3.1	59.7% ± 3.0	76.2% ± 2.6	83.4% ± 2.3
	0.2	49.6% ± 3.1	64.5% ± 3.0	74.2% ± 2.7	75.3% ± 2.7
	0.3	49.9% ± 3.1	64.1% ± 3.0	72.1% ± 2.8	72.8% ± 2.8
	0.4	55.6% ± 3.1	58.7% ± 3.1	64.9% ± 3.0	68.6% ± 2.9
r	0.5	51.6% ± 3.1	58.5% ± 3.1	65.0% ± 3.0	60.1% ± 3.0
	0.6	48.7% ± 3.1	52.4% ± 3.1	52.9% ± 3.1	56.4% ± 3.1
	0.7	44.1% ± 3.1	50.1% ± 3.1	47.5% ± 3.1	45.8% ± 3.1
	0.8	40.2% ± 3.0	39.7% ± 3.0	39.7% ± 3.0	41.0% ± 3.1
	0.9	35.0% ± 3.0	37.0% ± 3.0	32.3% ± 2.9	32.7% ± 2.9
	1	26.0% ± 2.7	24.3% ± 2.7	23.1% ± 2.6	27.6% ± 2.8

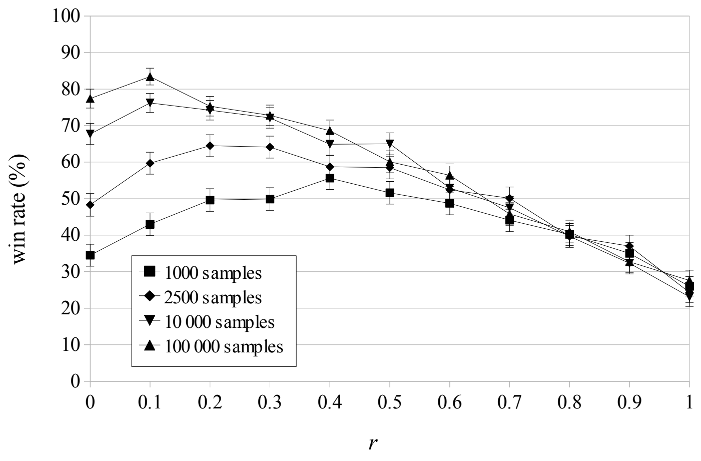


Figure 7.9: Graphical representation of Table 7.10.

move. In conclusion, these numbers show that Coalition Reduction increases the performance of the seekers significantly. However, cooperation is still important as the performance decreases if r becomes larger than 0.3. If less time is provided, the value of r should be increased. This may be because, if the seekers have less computation time, there is not sufficient time for planning.

7.4.7 MCTS versus Minimax-Based Techniques

In this set of experiments, the MCTS players are compared to the minimax-based players. For the MCTS hider, UCT with Progressive History, ϵ-greedy playouts, and move filtering is applied. For the MCTS seekers, UCT with Progressive History, single-tree determinization, ϵ-greedy playouts, Coalition Reduction with $r = 0.1$, and Location Categorization with the minimum-distance general table and 1-step selection is used. For the paranoid hider, killer moves, the history heuristic, and move filtering are applied. For the expectimax seekers, killer moves, the history heuristic, and Location Categorization with the minimum-distance general table are used. All players receive 1000 ms of thinking time per move.

Against the paranoid hider, the expectimax seekers win 76.2% ± 2.6 of the games and the MCTS seekers win 94.9% ± 1.4. Against the MCTS hider, the expectimax seekers manage to win 45.0% ± 3.1 and the MCTS seekers 81.2% ± 2.4 of the games. Consequently, the paranoid hider wins 23.8% ± 2.6 of the games against the expectimax seekers, while the MCTS hider wins 55.0% ± 3.1 of the games. Against the MCTS seekers, the paranoid hider wins only 5.1% ± 1.4 of the games, while the MCTS hider wins 18.8% ± 2.4. The results are summarized in Table 7.11. These results show that for both the hider and the seekers, MCTS appears to work far better than the minimax-based techniques. Furthermore, it is interesting to note that the win rate of the MCTS seekers against the MCTS hider is similar to the win rate of the expectimax seekers against the paranoid hider. This shows that, with both types of search techniques, the seekers have a similar advantage against the hider.

Table 7.11 Win rates of the different seekers against different hiders.

	hider	
seekers	MCTS	Paranoid
MCTS	81.2% ± 2.4	94.9% ± 1.4
Expectimax	45.0% ± 3.1	76.2% ± 2.6

7.4.8 Performance against the Nintendo DS Program

To test the strength of the MCTS-based program, it is matched against the Scotland Yard program on the Nintendo DS. The AI of this program is considered to be rather strong (cf. Frackowski, 2011).

For the hider and the seekers, the same settings and enhancements are used as described in Subsection 7.4.7. It is not possible to set the thinking time of the Nintendo DS player. It often plays immediately, but it sometimes takes 5 to 10 seconds to find a move. To have a fair comparison, we set the thinking time of the MCTS program to 2000 ms.

Because these games have to be played manually, only 50 games are played, where each program plays 25 times as the seekers and 25 times as the hider. Out of these 50 games, 34 games are won by the MCTS-based player. 23 of these games are won as the seekers and 11 as the hider. The Nintendo DS program wins 16 games, of which 14 as the seekers and 2 as the hider. These results show that the MCTS program plays stronger than the Nintendo DS program. Furthermore, they underline that the seekers have an advantage over the hider, because they win 37 out of the 50 games.

7.5 Chapter Conclusions and Future Research

This chapter investigated how MCTS can be applied to play the hide-and-seek game Scotland Yard, how it can be enhanced to improve its performance, and how it compares to minimax-based search techniques.

Using ϵ-greedy playouts to incorporate basic knowledge into MCTS considerably improves the performance of both the seekers and the hider. We observed that using different ϵ values and different playout strategies for the different players in the playouts performs significantly better than random playouts. This difference is caused by the asymmetric nature of the hide-and-seek game.

For handling the imperfect information, two different determinization techniques were investigated, namely separate-tree determinization and single-tree determinization. When using separate trees, majority voting for selecting the best move produces a higher win rate than calculating the average over all trees. Single-tree determinization has a slight overhead, but even when taking this into account, it performs significantly better than using separate trees.

Furthermore, Location Categorization was proposed, which is a technique that can be used by both the MCTS and the expectimax seekers in Scotland Yard to give a better prediction for the location of the hider. Three types of categorization were introduced for Scotland Yard: minimum-distance, average-distance and station type. The experiments revealed that the minimum-distance categorization performs best. It significantly increases the playing strength of both the MCTS and the expectimax seekers. The results gave empirical evidence that Location Categorization is a robust technique, as the weights work for both seeker types against two different hider types.

We also observed that the performance of the MCTS seekers can be improved by applying Coalition Reduction. This technique allows the seekers to cooperate more effectively in the coalition, by preventing them from becoming too passive or too selfish. It also became clear that cooperation is important, because the performance of the seekers drops significantly when the reduction becomes too large. Furthermore, if fewer playouts per move are provided, better results are achieved with a higher value of r. This may be because planning is more difficult if the seekers have less computation time.

In a direct comparison, MCTS performs considerably better than paranoid search for the hider and expectimax for the seeker. A comparison between MCTS and minimax-based techniques is not easy because each technique can be enhanced in different ways and the efficiency of the implementations may differ. However, the results do give an idea of the playing strength of the different search techniques. Finally, the experimental results showed that MCTS, a technique that uses only basic domain knowledge, was able to play Scotland Yard on a higher level than the Nintendo DS program, which is generally considered to be a strong player.

We define three possible directions for future research. The first is to improve Location Categorization. New types of categorization may be tested or different categorizations may be combined. This can be done by introducing three-step selection. The first two steps are used to select two categories using two different categorizations. In the third step, a possible location is selected that belongs to both selected categories. Other ways of combining two categorizations include taking the Cartesian product of the categories of both categorizations or using a weighted combination of category weights. It may also be interesting to test Location Categorization in other hide-and-seek games, for instance the two-player game Stratego to guess the ranks of the opponent's unknown pieces.

The second future research direction is to continue the recent work of Silver and Veness (2010), who extended MCTS to Partially Observable Markov Decision Processes (POMDPs). Their technique, Partially Observable Monte-Carlo Planning (POMCP), was successfully applied to Battleship and a partially observable variant of PacMan. Their technique could be applied to the seekers in Scotland Yard as well. With POMDPs, the theoretical shortcomings of determinization can be avoided.

The third possible future research topic is modeling Scotland Yard as a Bounded Horizon Hidden Information Game (BHHIG) (Teytaud and Flory, 2011). This technique does not have the theoretical shortcomings of determinization as well, but it is also slower. A BHHIG can be used for modeling partially observable games in which information is regularly revealed. Teytaud and Flory showed that each BHHIG can be represented as a Game with Simultaneous Actions (GSA) and that UCT can be adapted to such games.

CHAPTER 8

Conclusions and Future Research

This thesis investigated how Monte-Carlo Tree Search (MCTS) can be improved in multi-player games. It dealt with the enhancement of search policies, selection strategies, and playout strategies in MCTS for deterministic multi-player games with perfect information. Furthermore, it examined MCTS in the hide-and-seek game Scotland Yard. The research was guided by the problem statement formulated in Section 1.5. This section also provided four research questions that should be answered before addressing the problem statement. In this chapter we formulate the conclusions of this thesis and recommendations for future research.

First, Section 8.1 answers the four research questions. Based on these answers, we address the problem statement in Section 8.2. Finally, Section 8.3 provides five directions for future research.

8.1 Conclusions on the Research Questions

The four research questions formulated in Chapter 1 concern the enhancement of MCTS in deterministic multi-player games with perfect information and hide-and-seek games. These research questions are answered one by one in the following subsections.

8.1.1 Search Policies for Multi-Player Games

The standard MCTS variant for multi-player games has a search policy that is similar to max^n. However, the paranoid and BRS search technique can also be implemented in multi-player MCTS as search policies. This led us to the first research question.

> **Research question 1**: *How can multi-player search policies be incorporated in MCTS?*

To answer the first research question, we incorporated the paranoid and BRS search policies, along with the default max^n policy, in MCTS. With these search policies the selection and the backpropagation phases of MCTS are altered. For testing the performance of these search policies, four different deterministic multi-player games with perfect information were chosen as a test bed, namely Chinese Checkers, Focus, Rolit, and Blokus.

In the MCTS framework, the max^n search policy appeared to perform best. The advantages of paranoid and BRS in the minimax framework do not apply in MCTS, because $\alpha\beta$ pruning is not applicable in MCTS. An additional problem with MCTS-BRS may be that, in the tree, invalid positions are investigated, which may reduce the reliability of the playouts as well. Still, MCTS-paranoid and MCTS-BRS overall achieved decent win rates against MCTS-max^n, especially with lower time settings. Furthermore, MCTS-paranoid is on equal footing with MCTS-max^n in Blokus and, in the vanilla version of MCTS, MCTS-paranoid and MCTS-BRS significantly outperformed MCTS-max^n in Focus. Based on the results we may conclude that the max^n search policy is the most robust, though the BRS and paranoid search policies can still be competitive.

Finally, we enhanced the max^n search policy by proposing a multi-player variant of MCTS-Solver, called MP-MCTS-Solver. This variant is able to prove the game-theoretic value of a position. A win rate between 53% and 55% was achieved in the sudden-death game Focus with the standard update rule. We may conclude that proving game-theoretic values improves the playing strength of MCTS in a multi-player sudden-death domain.

8.1.2 Selection Strategies for Multi-Player MCTS

The selection phase is an important phase in the MCTS procedure. Nodes are chosen using a selection strategy until a leaf node is reached. Selecting promising moves while the number of playouts is still low is difficult, because there is only limited information available. RAVE (Gelly and Silver, 2007), which uses AMAF values to guide the selection strategy when the number of visits is low, works well in games such as Go, Havannah, and Hex. For the multi-player game Chinese Checkers, RAVE does not appear to work well. This led us to the second research question.

> **Research question 2**: *How can the selection phase of MCTS be enhanced in perfect-information multi-player games?*

To answer this question a new domain-independent selection strategy, called Progressive History, was proposed. This technique is a combination of the relative history heuristic (Schaeffer, 1983; Winands *et al.*, 2006) and Progressive Bias (Chaslot *et al.*, 2008b). Contrary to RAVE, Progressive History maintains its gathered data in a global table, in a similar way as the playout strategy MAST (Finnsson and Björnsson, 2008). The performance of this technique was tested in Chinese Checkers, Focus, Rolit, and Blokus.

Progressive History was a significant improvement in all games with different numbers of players. In a comparison with UCT, Progressive History gained the highest win rates in the two-player variants, winning around 80% of the games. Moreover, Progressive History performed better than standard UCT in the multi-player variants as well. Progressive AMAF, which applies AMAF values instead of history values, overall performed significantly worse than Progressive History. Additionally, experiments in the two-player game Havannah showed that Progressive History performed better in this game than RAVE. Furthermore, experiments revealed that Progressive History also significantly increases the playing strength of the seekers in the hide-and-seek game Scotland Yard. Based on the results we may conclude that Progressive History considerably enhances MCTS in both two-player and multi-player games.

8.1.3 Playout Strategies for Multi-Player MCTS

During the playouts, moves are selected using a playout strategy. The results of the playouts are backpropagated in the tree, which are then used in the selection phase. In order to support the selection phase, the playouts have two conflicting goals: they should be quick, so that more playouts are performed and more information is available, and they should resemble decent or strong play, which can be quite time-costly. This led us to the third research question.

> **Research question 3**: *How can the playouts of MCTS be enhanced in perfect-information multi-player games?*

To answer this research question, two-ply searches for selecting moves in the playout phase in MCTS were introduced for multi-player games. Three different search techniques were investigated for multi-player games, namely max^n, paranoid, and BRS. These playout strategies were compared against random, greedy, and one-ply playouts to determine how to balance search and speed in the playouts of multi-player MCTS.

The results showed that search-based playouts significantly improved the quality of the playouts in MCTS. Among the different playout strategies, BRS performed best, followed by paranoid and max^n. This benefit was countered by a reduction of the number of playouts per second. Especially BRS and max^n suffered from this effect. Among the tested two-ply search-based playouts, paranoid overall performed best with both short and long time settings. With more thinking time, the two-ply search-based playout strategies performed relatively better than the one-ply and greedy strategies. This indicates that with longer time settings, more computationally expensive playouts may be used to increase the playing strength of MCTS-based players. Under these conditions, search-based playouts outperforms one-ply searches in the four-player variant of Focus and the three-player variant of Chinese Checkers. Based on the experimental results we may conclude that search-based playouts for multi-player games may be beneficial if the players receive sufficient thinking time.

8.1.4 MCTS for a Hide-and-Seek Game

Hide-and-seek games feature properties that make them a challenging test domain for MCTS-based programs. These properties are the following. (1) They feature hidden information for the seekers, (2) they are asymmetric, and (3) the seekers cooperate in a fixed coalition. In order to tackle these issues, we formulated the following research question.

Research question 4: *How can MCTS be adapted for hide-and-seek games?*

To answer the fourth research question, we chose the game Scotland Yard as the test domain. This hide-and-seek game features the aforementioned properties and is currently too complex for computers to solve.

For handling the imperfect information, two different determinization techniques were investigated, namely single-tree determinization and separate-tree determinization. Single-tree determinization had a slight overhead, but even when taking this into account, it performed significantly better than using separate trees.

Furthermore, Location Categorization was proposed, which is a technique that can be used by both the MCTS and the expectimax seekers to give a better prediction for the location of the hider. The experiments revealed that for Scotland Yard the minimum-distance categorization performs best. It significantly increased the playing strength of both the MCTS and the expectimax seekers. The results gave empirical evidence that Location Categorization is a robust technique, as the weights worked for both seeker types against two different types of hider.

Because of the asymmetric nature of the hide-and-seek game Scotland Yard, during the playouts, different playout strategies may be used by the different types of players. We found that, for the MCTS hider, it is best to assume during the playouts that the seekers do not know where the hider is, while the MCTS seekers perform best if they do assume where the hider is located.

For dealing with the cooperation of the seekers, Coalition Reduction was proposed. This technique reduces the rewarded value for the root player if another player in the coalition wins the game, allowing the seekers to cooperate more effectively in the coalition. We observed that the performance of the MCTS seekers increased by applying Coalition Reduction. Cooperation still appeared to be important, because the performance of the seekers dropped significantly when the reduction became too large.

In a direct comparison, MCTS performed considerably better than paranoid search for the hider and expectimax for the seekers. Finally, the experimental results showed that MCTS was able to play Scotland Yard on a higher level than a commercial Nintendo DS program, which is generally considered to be a strong player.

In conclusion, with the incorporation of enhancements such as single-tree determinization, Location Categorization, and Coalition Reduction, we were able to let an MCTS-based player play the hide-and-seek game Scotland Yard on a strong level.

8.2 Conclusions on the Problem Statement

After answering the four research questions, we can now address the problem statement.

Problem statement: *How can Monte-Carlo Tree Search be improved to increase the performance in multi-player games?*

The answer to the problem statement may be summarized in four points, based on the research questions. First, the max^n search policy performs the best in multi-player MCTS, while the BRS and paranoid policies are still competitive. The max^n search policy can be enhanced with a multi-player variant of the MCTS-Solver. Second, the Progressive History selection strategy significantly increases the performance of two-player and multi-player MCTS. Third, two-ply search-based playouts significantly improve the quality of the playouts and, assuming a sufficient amount of thinking time is provided, increases the performance of MCTS in multi-player domains. Fourth, incorporating single-tree determinization, Location Categorization, and Coalition Reduction into MCTS significantly improves its performance in the multi-player hide-and-seek game Scotland Yard.

8.3 Recommendations for Future Research

The research presented in this thesis indicates the following five areas of future research.

1. **Application of other search policies.** Chapter 4 investigated three common search policies for multi-player games, namely max^n, paranoid, and BRS, in the MCTS framework. We did not consider policies derived from these techniques, such as the Coalition-Mixer (Lorenz and Tscheuschner, 2006) or MP-Mix (Zuckerman *et al.*, 2009). They use a combination of max^n and (variations of) paranoid search. Tuning and testing such policies for multi-player MCTS is a direction of future research. Another future research direction is the application of BRS$^+$ as proposed by Esser *et al.* (2013) in MCTS. The basic idea is that, besides the opponent with the best counter move, the other opponents are allowed to perform a move as well. These moves are selected using a static move ordering. The advantage of these variants is that no invalid positions are searched, while maintaining the advantages of the original BRS algorithm.

2. **Combination of Progressive History with other selection strategies.** In Chapter 5, the Progressive History selection strategy was introduced. It may be interesting to combine Progressive History with domain-dependent selection strategies such as prior knowledge (Gelly and Silver, 2007), Progressive Widening (Coulom, 2007b; Chaslot *et al.*, 2008b), or Progressive Bias (Chaslot *et al.*, 2008b). Progressive History may also be combined with N-grams (Stankiewicz *et al.*, 2012; Tak *et al.*, 2012), which keeps track of move sequences instead of single moves. By using N-grams, more context is offered.

3. **Enhancement of search-based playouts.** In Chapter 6, we performed a comparison between several playout strategies that require domain knowledge to function. The proposed two-ply searches may be further optimized. Though a two-ply search will always be slower than a one-ply search, the current speed difference could be reduced further. This can be achieved for instance by improved move ordering or lazy evaluation functions.

   Another future research direction is the application of three-ply searches in the playout. While these playouts are more time-intensive than two-ply search-based playouts, they may increase the reliability of the playouts even more.

While Winands and Björnsson (2011) mentioned that three-ply $\alpha\beta$ searches in the MCTS playouts for the two-player game Lines of Action are too expensive, we anticipate that three-ply search-based playouts may be beneficial with increased computing power and enhancements such as parallelization.

4. **Further investigation of MCTS in Scotland Yard.** Chapter 7 introduced Location Categorization for biasing the possible locations of the hider. For this enhancement, new types of categorization may be tested or different categorizations may be combined. This can be done by introducing three-step selection or by taking the Cartesian product of the categories of both categorizations.

   To handle the imperfect information for the hiders, determinization was applied. Two different determinization techniques were tested, namely single-tree determinization and separate-tree determinization. Different determinization techniques, such as Multiple-Observer Information Set MCTS (Cowling *et al.*, 2012a) may also be investigated. Determinization, however, has several theoretical shortcomings. Besides determinization, there exist various alternatives to handle imperfect information.

   An approach is to continue the work of Silver and Veness (2010), who extended MCTS to Partially Observable Markov Decision Processes (POMDPs). Their technique, Partially Observable Monte-Carlo Planning (POMCP), was successfully applied to Battleship and a partially observable variant of PacMan. Their technique could be applied to the seekers in Scotland Yard as well. With POMDPs, the theoretical shortcomings of determinization can be avoided.

   Scotland Yard may also be modeled as a Bounded Horizon Hidden Information Game (BHHIG) (Teytaud and Flory, 2011). This technique does not have the theoretical shortcomings of determinization, but it is also slower. A BHHIG can be used for modeling partially observable games in which information is regularly revealed. Teytaud and Flory showed that each BHHIG can be represented as a Game with Simultaneous Actions (GSA) and that the UCT algorithm can be adapted to such games.

5. **Application to other domains.** The search enhancements in this thesis have been tested in various domains. However, it may be interesting to test the performance of these enhancements in other domains as well. For instance, Progressive History was evaluated in the two-player and multi-player variants of Chinese Checkers, Focus, Rolit, and Blokus and in the two-player game Havannah. The performance of this enhancement may also be tested in popular domains such as Go and Hex. The multi-player variant of MCTS-Solver is only tested in Focus, because this is the only multi-player games investigated that has the sudden-death property. While this enhancement was tested in the two-player, three-player and the four-player variant, it may be interesting to test the performance of the multi-player variant of the MCTS-Solver in other multi-player sudden-death games, such as the multi-player variant of Tron.

   These enhancements may also be applied in modern board games, such as Settlers of Catan (cf. Szita, Chaslot, and Spronck, 2010) and Carcassonne (cf. Heyden, 2009).

Furthermore, only one hide-and-seek game was used as a test domain for enhancements such as Location Categorization and Coalition Reduction, namely Scotland Yard. For the enhancements proposed in Chapter 7, it may be interesting to test these in other games as well. For instance, Location Categorization may also work in other games with imperfect information, such as Stratego, and Coalition Reduction may also perform well in different games that feature a coalition between players, such as the ghosts in Ms. PacMan or in the pursuit-evasion game Cops and Robbers.

References

Abramson, B. (1990). Expected-outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 2, pp. 182–193. [6, 26]

Adler, M., Räcke, H., Sivadasan, N., Sohler, C., and Vöcking, B. (2003). Randomized Pursuit-Evasion in Graphs. *Combinatorics, Probability & Computing*, Vol. 12, No. 3, pp. 225–244. [5]

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *Proceedings of the ACM Annual Conference*, pp. 466–473, ACM, New York, NY, USA. [24, 45, 84]

Allis, L.V. (1988). A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins. M.Sc. thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands. [2]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, Department of Computer Science, Rijksuniversiteit Limburg, Maastricht, The Netherlands. [33, 39, 43, 62]

Allis, L.V., Huntjes, M.P.H., and Herik, H.J. van den (1996). Go-Moku Solved by New Search Techniques. *Computational Intelligence*, Vol. 12, No. 1, pp. 7–23. [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*, Vol. 2, No. 4, pp. 251–258. [7, 62]

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multi-armed Bandit Problem. *Machine Learning*, Vol. 47, No. 2, pp. 235–256. [8, 27, 67]

Baier, H. and Drake, P.D. (2010). The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 303–309. [32, 79]

Baier, H. and Winands, M.H.M. (2013). Monte-Carlo Tree Search and Minimax Hybrids. *Proceedings of the 2013 IEEE Conference on Computational Intelligence and Games*, pp. 129–136, IEEE. [80]

Ballard, B.W. (1983). The *-Minimax Search Procedure for Trees Containing Chance Nodes. *Artificial Intelligence*, Vol. 21, No. 3, pp. 327–350. [17]

Beal, D. and Smith, M.C. (1994). Random Evaluations in Chess. *ICCA Journal*, Vol. 17, No. 1, pp. 3–9. [45]

Billings, D., Peña, L., Schaeffer, J., and Szafron, D. (1999). Using Probabilistic Knowledge and Simulation to Play Poker. *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (eds. J. Hendler and D. Subramanian), pp. 697–703, AAAI Press. [27]

Billings, D., Davidson, A., Schauenberg, T., Burch, N., Bowling, M., Holte, R., Schaeffer, J., and Szafron, D. (2006). Game-Tree Search with Adaptation in Stochastic Imperfect-Information Games. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *LNCS*, pp. 21–34, Springer-Verlag, Berlin, Germany. [107]

Björnsson, Y. and Finnsson, H. (2009). CADIAPLAYER: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 4–15. [6]

Bourki, A., Chaslot, G.M.J.B., Coulm, M., Danjean, V., Doghmen, H., Hoock, J-B., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P., and Yu, Z. (2011). Scalability and Parallelization of Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 48–58, Springer-Verlag, Berlin, Germany. [32]

Bouzy, B. (2005). Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. *Information Sciences*, Vol. 175, No. 4, pp. 247–257. [8, 29]

Bouzy, B. and Helmstetter, B. (2004). Monte-Carlo Go Developments. *Advances in Computer Games (ACG 10)* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), Vol. 135 of *IFIP Advances in Information and Communication Technology*, pp. 159–174, Kluwer Academic Publishers. [6, 27]

Breuker, D.M. (1998). *Memory versus Search in Games.* Ph.D. thesis, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands. [25]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180. [45]

Browne, C.B., Powley, E., 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*, Vol. 4, No. 1, pp. 1–43. [8, 29, 32]

Brügmann, B. (1993). Monte Carlo Go. Technical report, Max-Planck-Institute of Physics, München, Germany. [27, 68]

Buro, M. (1995). ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71–76. [39]

Buro, M. (2000). Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. *Games in AI Research* (eds. H.J. van den Herik and H. Iida), pp. 77–96, Universiteit Maastricht, Maastricht, The Netherlands. [39, 105]

Buro, M., Long, J.R., Furtak, T., and Sturtevant, N.R. (2009). Improving State Evaluation, Inference, and Search in Trick-based Card Games. *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)* (ed. C. Boutilier), pp. 1407–1413, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. [99]

Cazenave, T. (2006). A Phantom Go Program. *Advances in Computer Games (ACG 11)* (eds. H.J. van den Herik, S-C. Hsu, T-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *LNCS*, pp. 120–125, Springer-Verlag, Berlin, Germany. [99]

Cazenave, T. (2008). Multi-player Go. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 50–59, Springer-Verlag, Berlin, Germany. [3, 30, 48, 49, 66, 82]

Cazenave, T. and Jouandeau, N. (2007). On the Parallelization of UCT. *Proceedings of the Computer Games Workshop* (eds. H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd), pp. 93–101. [100]

Cazenave, T. and Saffidine, A. (2010). Monte-Carlo Hex. *Board Game Studies XIIIth Colloquium*, Paris, France. [7, 62, 69]

Cazenave, T. and Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 93–104, Springer-Verlag, Berlin, Germany. [62]

Chaslot, G.M.J-B. (2010). *Monte-Carlo Tree Search*. Ph.D. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [8]

Chaslot, G.M.J-B., Jong, S. de, Saito, J-T., and Uiterwijk, J.W.H.M. (2006a). Monte-Carlo Tree Search in Production Management Problems. *Proceedings of the 18th Benelux Conference on Artificial Intelligence* (eds. P-Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 91–98, Namur, Belgium. [7]

Chaslot, G.M.J-B., Saito, J-T., Bouzy, B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2006b). Monte-Carlo Strategies for Computer Go. *Proceedings of the 18th Benelux Conference on Artificial Intelligence* (eds. P-Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 83–90, Namur, Belgium. [29]

Chaslot, G.M.J-B., Winands, M.H.M., and Herik, H.J van den (2008a). Parallel Monte-Carlo Tree Search. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 60–71, Springer-Verlag, Berlin, Germany. [32]

Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2008b). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [8, 28, 30, 32, 67, 70, 78, 122, 125]

Chaslot, G.M.J-B., Fiter, C., Hoock, J-B., Rimmel, A., and Teytaud, O. (2010). Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. *Advances in Computer Games (ACG 12)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 1–13, Springer-Verlag, Berlin, Germany. [70]

Chen, K-H. and Zhang, P. (2008). Monte-Carlo Go with Knowledge-Guided Simulations. *ICGA Journal*, Vol. 31, No. 2, pp. 67–76. [8, 29]

Childs, B.E., Brodeur, J.H., and Kocsis, L. (2008). Transpositions and Move Groups in Monte Carlo Tree Search. *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games* (eds. P. Hingston and L. Barone), pp. 389–395, IEEE. [32]

Ciancarini, P. and Favini, G.P. (2009). Monte Carlo Tree Search Techniques in the Game of Kriegspiel. *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)* (ed. C. Boutilier), pp. 474–479, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. [99, 100]

Clune, J.E. (2007). Heuristic Evaluation Functions for General Game Playing. *Proceedings of the Twenty-Second AAAI on Artificial Intelligence*, Vol. 22, pp. 1134–1139, AAAI Press. [34]

Coulom, R. (2007a). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83, Springer-Verlag, Berlin, Germany. [6, 28, 29, 67]

Coulom, R. (2007b). Computing "Elo Ratings" of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 199–208. [78, 125]

Cowling, P.I., Powley, E.J., and Whitehouse, D. (2012a). Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 120–143. [100, 112, 126]

Cowling, P.I., Ward, C.D., and Powley, E.J. (2012b). 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*, Vol. 4, No. 4, pp. 241–257. [99]

Den Teuling, N.G.P. and Winands, M.H.M. (2012). Monte-Carlo Tree Search for the Simultaneous Move Game Tron. *Computer Games Workshop at the ECAI 2012*, pp. 126–141, Montpellier, France. [62]

Doberkat, E-E., Hasselbring, W., and Pahl, C. (1996). Investigating Strategies for Cooperative Planning of Independent Agents through Prototype Evaluation. *Coordination Models and Languages (COORDINATION '96)* (eds. P. Ciancarini and C. Hankin), Vol. 1061 of *LNCS*, pp. 416–419, Springer-Verlag, Berlin, Germany. [96, 99, 109]

Donkers, H.H.L.M. (2003). *Nosce Hostem: Searching with Opponent Models.* Ph.D. thesis, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands. [13]

Drake, P.D. (2009). The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA Journal*, Vol. 32, No. 4, pp. 221–227. [32, 79, 93]

Enzenberger, M. and Müller, M. (2010). A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. *Advances in Computer Games (ACG 12)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 14–20, Springer-Verlag, Berlin, Germany. [32, 93]

Esser, M. (2012). Best-Reply Search in Multi-Player Chess. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [23]

Esser, M., Gras, M., Winands, M.H.M., Schadd, M.P.D., and Lanctot, M. (2013). Improving Best-Reply Search. *Computers and Games (CG 2013).* Accepted. [23, 66, 125]

Finnsson, H. (2012). *Simulation-Based General Game Playing.* Ph.D. thesis, School of Computer Science, Reykjavik University, Reykjavik, Iceland. [8, 34, 62, 69, 71, 75, 87]

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence* (eds. D. Fox and C.P. Gomes), pp. 259–264, AAAI Press. [32, 70, 71, 79, 93, 122]

Finnsson, H. and Björnsson, Y. (2010). Learning Simulation Control in General Game-Playing Agents. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (eds. M. Fox and D. Poole), pp. 954–959, AAAI Press. [32, 79]

Fossel, J.D. (2010). Monte-Carlo Tree Search Applied to the Game of Havannah. B.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [67, 76, 77, 78]

Frackowski, A. (2011). [NDS Review] Scotland Yard DS - Hold.Start.Select. http://holdstartselect.com/nds-review-scotland-yard-ds/. [96, 117]

Frank, I. and Basin, D. (1998). Search in Games with Incomplete Information: A Case Study using Bridge Card Play. *Artificial Intelligence*, Vol. 100, Nos. 1–2, pp. 87–123. [99]

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. *ICML '07: Proceedings of the 24th International Conference on Machine Learning* (ed. Z. Ghahramani), pp. 273–280, ACM, New York, NY, USA. [8, 32, 67, 68, 76, 78, 122, 125]

Gelly, S. and Silver, D. (2011). Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856–1875. [68]

Gelly, S. and Wang, Y. (2006). Exploration Exploitation in Go: UCT for Monte-Carlo Go. *Neural Information Processing Systems Conference On-line Trading of Exploration and Exploitation Workshop.* [32]

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, Paris, France. [8, 29]

Ginsberg, M.L. (1999). GIB: Steps Toward an Expert-Level Bridge-Playing Program. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)* (ed. T. Dean), Vol. 1, pp. 584–589, Morgan Kaufmann. [27, 99]

Gras, M. (2012). Multi-Player Search in the Game of Billabong. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [23]

Greenblatt, R.D., Eastlake, D.E., and Crocker, S.D. (1967). The Greenblatt Chess Program. *Proceedings of the AFIPS '67 Fall Joint Computer Conference*, Vol. 31, pp. 801–810. [25, 84]

Hartmann, D. (1988). Butterfly Boards. *ICCA Journal*, Vol. 11, Nos. 2–3, pp. 64–71. [25]

Hauk, T., Buro, M., and Schaeffer, J. (2006a). Rediscovering *-Minimax Search. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *LNCS*, pp. 35–50. Springer-Verlag, Berlin, Germany. [17]

Hauk, T., Buro, M., and Schaeffer, J. (2006b). *-Minimax Performance in Backgammon. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *LNCS*, pp. 51–66. Springer-Verlag, Berlin, Germany. [18]

Heinz, E.A. (2001). Self-play Experiments in Computer Chess Revisited. *Advances in Computer Games (ACG 9)* (eds. H.J. van den Herik and B. Monien), pp. 73–91. [53]

Helmbold, D.P. and Parker-Wood, A. (2009). All-Moves-As-First Heuristics in Monte-Carlo Go. *Proceedings of the 2009 International Conference on Artificial Intelligence* (eds. H.R. Arabnia, D. de la Fuente, and J.A. Olivas), pp. 605–610, CSREA Press. [68]

Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311. [43]

Heyden, C. (2009). Implementing a Computer Player for Carcassonne. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [126]

Higashiuchi, Y. and Grimbergen, R. (2006). Enhancing Search Efficiency by Using Move Categorization Based on Game Progress in Amazons. *Advances in Computer Games (ACG 11)* (eds. H.J. van den Herik, S-C. Hsu, T-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *LNCS*, pp. 73–87, Springer-Verlag, Berlin, Germany. [104, 105]

Hoock, J-B., Lee, C-S., Rimmel, A., Teytaud, F., Teytaud, O., and Wang, M-H. (2010). Intelligent Agents for the Game of Go. *IEEE Computational Intelligence Magazine*, Vol. 5, No. 4, pp. 28–42. [69, 76]

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

Huang, S-C., Coulom, R., and Lin, S-S. (2011). Monte-Carlo Simulation Balancing in Practice. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 81–92, Springer-Verlag, Berlin, Germany. [70]

Huang, S.-C., Arneson, B., Hayward, R.B., Müller, M., and Pawlewicz, J. (2013). MoHex 2.0: a Pattern-based MCTS Hex Player. *Computers and Games (CG 2013).* [70]

Hyatt, R. and Cozzie, A. (2005). The Effect of Hash Signature Collisions in a Chess Program. *ICGA Journal*, Vol. 28, No. 3, pp. 131–139. [25]

Hyatt, R.M., Gower, A.E., and Nelson, H.L. (1990). Cray Blitz. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 111–130, Springer-Verlag, New York, NY, USA. [24, 25, 26]

Iida, H., Sakuta, M., and Rollason, J. (2002). Computer Shogi. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 121–144. [43]

Jones, A.J. (1980). *Game Theory: Mathematical Models of Conflict.* Ellis Horwood, West Sussex, England. [19]

Joosten, B. (2009). Creating a Havannah Playing Agent. B.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [43]

Knuth, D.E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, USA. [25]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [6, 13, 14, 15, 83]

Kocsis, L. (2003). *Learning Search Decisions.* Ph.D. thesis, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands. [24]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *LNCS*, pp. 282–293, Springer-Verlag, Berlin, Germany. [6, 8, 27, 28, 67]

Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). Move Ordering Using Neural Networks. *Engineering of Intelligent Systems* (eds. L. Montosori, J Váncza, and M. Ali), Vol. 2070 of *LNAI*, pp. 45–50, Springer-Verlag, Berlin, Germany. [24]

Kocsis, L., Szepesvári, C., and Willemson, J. (2006). Improved Monte-Carlo Search. Technical report, MTA SZTAKI, Budapest, Hungary; University of Tartu, Institute of Computer Science, Tartu, Estonia. [51]

Korf, R.E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109. [26]

Korf, R.E. (1991). Multi-Player Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 48, No. 1, pp. 99–111. [5, 20]

Kozelek, T. (2009). Methods of MCTS and the Game Arimaa. M.Sc. thesis, Department of Theoretical Computer Science and Mathematical Logic, Charles University, Prague, Czech Republic. [70]

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*, Vol. 1, No. 1, pp. 73–89. [2]

Long, J.R., Sturtevant, N.R., Buro, M., and Furtak, T. (2010). Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search. *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence* (eds. M. Fox and D. Poole), pp. 134–140, AAAI Press. [99]

Lorentz, R.J. (2011). Improving Monte-Carlo Tree Search in Havannah. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 105–115, Springer-Verlag, Berlin, Germany. [69, 76, 80]

Lorentz, R.J. and Horey, T. (2013). Programming Breakthrough. *Computers and Games (CG 2013)*, LNCS, Springer-Verlag, Berlin, Germany. [62]

Lorenz, U. and Tscheuschner, T. (2006). Player Modeling, Search Algorithms and Strategies in Multi-player Games. *Advances in Computer Games (ACG 11)* (eds. H.J. van den Herik, S-C. Hsu, T-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *LNCS*, pp. 210–224, Springer-Verlag, Berlin, Germany. [3, 66, 125]

Luckhardt, C.A. and Irani, K.B. (1986). An Algorithmic Solution of N-Person Games. *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI)* (ed. T. Kehler), Vol. 1, pp. 158–162, Morgan Kaufmann. [3, 5, 6, 7, 19, 20]

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. [25, 26, 84]

Megiddo, N., Hakimi, S.L., Garey, M.R., Johnson, D.S., and Papadimitriou, C.H. (1988). The Complexity of Searching a Graph. *Journal of the Association for Computing Machinery*, Vol. 35, No. 1, pp. 18–44. [5]

Mesmay, F. de, Rimmel, A., Voronenko, Y., and Püschel, M. (2009). Bandit-based Optimization on Graphs with Application to Library Performance Tuning. *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)* (eds. A.P. Danyluk, L. Bottou, and M.L. Littman), Vol. 382 of *ACM International Conference Proceeding Series*, pp. 729–736, ACM. [7]

Metropolis, N. (1985). Monte Carlo: In the Beginning and Some Great Expectations. *Proceedings of the Joint Los Alamos National Laboratory* (eds. R. Alcouffe, R. Dautray, A. Forster, G. Ledanois, and B. Mercier), Vol. 240 of *Lecture Notes in Physics*, pp. 62–70, Springer. [26]

Metropolis, N. and Ulam, S. (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, Vol. 44, No. 247, pp. 335–341. [26]

Michie, D. (1966). Game-Playing and Game-Learning Automata. *Advances in Programming and Non-Numerical Computation* (ed. L. Fox), pp. 183–200. Pergamon Press. [6, 16, 107]

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

Müller, F., Späth, C., Geier, T., and Biundo, S. (2012). Exploiting Expert Knowledge in Factored POMDPs. *20th, European Conference on Artificial Intelligence* (eds. L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. Lucas), pp. 606–611. [7]

Nash, J.F. (1951). Non-Cooperative Games. *The Annals of Mathematics*, Vol. 54, No. 2, pp. 286–295. [19]

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavior.* Princeton University Press, Princeton, NJ, USA, second edition. [6, 13, 21]

Nijssen, J.A.M. (2007). Playing Othello Using Monte Carlo. B.Sc. thesis, Maastricht ICT Competence Centre, Maastricht University, Maastricht, The Netherlands. [40]

Nijssen, J.A.M. and Winands, M.H.M. (2010). Enhancements for Multi-Player Monte-Carlo Tree Search. *Proceedings of the 22nd Benelux Conference on Artificial Intelligence*, Luxembourg City, Luxembourg. Extended abstract. [67]

Nijssen, J.A.M. and Winands, M.H.M. (2011a). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 238–249, Springer-Verlag, Berlin, Germany. [32, 47, 67]

Nijssen, J.A.M. and Winands, M.H.M. (2011b). Monte-Carlo Tree Search for the Game of Scotland Yard. *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games* (eds. S-B. Cho, S.M. Lucas, and P. Hingston), pp. 158–165, IEEE. [95, 106]

Nijssen, J.A.M. and Winands, M.H.M. (2011c). Monte-Carlo Tree Search for the Game of Scotland Yard. *Proceedings of the 23rd Benelux Conference on Artificial Intelligence* (eds. P. De Causmaecker, J. Maervoet, T. Messelis, K. Verbeeck, and T. Vermeulen), pp. 417–418, Ghent, Belgium. Extended abstract. [95]

Nijssen, J.A.M. and Winands, M.H.M. (2012a). An Overview of Search Techniques in Multi-Player Games. *Computer Games Workshop at ECAI 2012*, pp. 50–61, Montpellier, France. [33, 47]

Nijssen, J.A.M. and Winands, M.H.M. (2012b). Playout Search for Monte-Carlo Tree Search in Multi-Player Games. *Advances in Computer Games (ACG 13)* (eds. H.J. van den Herik and A. Plaat), Vol. 7168 of *LNCS*, pp. 72–83, Springer-Verlag, Berlin, Germany. [79]

Nijssen, J.A.M. and Winands, M.H.M. (2012c). Playout Search for Monte-Carlo Tree Search in Multi-Player Games. *Proceedings of the 24th Benelux Conference on Artificial Intelligence* (eds. J.W.H.M. Uiterwijk, N. Roos, and M.H.M. Winands), pp. 309–310, Maastricht, The Netherlands. Extended abstract. [79]

Nijssen, J.A.M. and Winands, M.H.M. (2012d). Monte-Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 4, pp. 282–294. [95, 100]

Nijssen, J.A.M. and Winands, M.H.M. (2013). Search Policies in Multi-Player Games. *ICGA Journal*, Vol. 36, No. 1, pp. 3–21. [47]

Parsons, T.D. (1978). Pursuit-Evasion in a Graph. *Theory and Applications of Graphs* (eds. Y. Alavi and D.R. Lick), Vol. 642 of *Lecture Notes in Mathematics*, pp. 426–441, Springer-Verlag, Berlin, Germany. [5]

Pepels, T. and Winands, M.H.M. (2012). Enhancements for Monte-Carlo Tree Search in Ms Pac-Man. *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, pp. 265–272, IEEE. [7]

Perez, D., Rohlfshagen, P., and Lucas, S. (2012). Monte Carlo Tree Search: Long-term versus Short-term Planning. *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, pp. 219–226, IEEE. [7]

Piccione, P.A. (1980). In Search of the Meaning of Senet. *Archaeology*, pp. 55–58. [2]

Powley, E.J., Whitehouse, D., and Cowling, P.I. (2011). Determinization in Monte-Carlo Tree Search for the Card Game Dou Di Zhu. *Proceedings of Artificial Intelligence and Simulation of Behaviour (AISB) Convention*, pp. 17–24. [99]

Powley, E.J., Whitehouse, D., and Cowling, P.I. (2012). Monte Carlo Tree Search with Macro-Actions and Heuristic Route Planning for the Physical Travelling Salesman Problem. *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games*, pp. 234–241, IEEE. [7]

Powley, E.J., Whitehouse, D., and Cowling, P.I. (2013). Bandits All the Way Down: UCB1 as a Simulation Policy in Monte Carlo Tree Search. *Proceedings of the 2013 IEEE Conference on Computational Intelligence and Games*, pp. 81–88, IEEE. [78]

Rijswijck, J. van (2000). Computer Hex: Are Bees Better than Fruitflies? M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Canada. [43]

Rimmel, A., Teytaud, F., and Teytaud, O. (2011a). Biasing Monte-Carlo Simulations through RAVE Values. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 59–68, Springer-Verlag, Berlin, Germany. [69, 76, 146]

Rimmel, A., Teytaud, F., and Teytaud, O. (2011b). Biasing Monte-Carlo Simulations through RAVE Values. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 59–68, Springer, Berlin, Germany. [79]

Rosenbloom, P.S. (1982). A World-Championship-Level Othello Program. *Artificial Intelligence*, Vol. 19, No. 3, pp. 279–320. [40]

Russell, S.J. and Norvig, P. (2002). *Artificial Intelligence, A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, USA, second edition. [99]

Sackson, S. (1969). *A Gamut of Games.* Random House, New York, NY, USA. [36]

Saffidine, A. (2013). *Solving Games and All That.* Ph.D. thesis, Université Paris Dauphine, LAMSADE, Paris, France. [23]

Saito, J-T. and Winands, M.H.M. (2010). Paranoid Proof-Number Search. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (eds. G.N. Yannakakis and J. Togelius), pp. 203–210, IEEE. [21, 39, 40]

Saito, J-T., Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2007). Grouping Nodes for Monte-Carlo Tree Search. *Proceedings of the 19th Benelux Conference on Artificial Intelligence* (eds. M.M. Dastani and E. de Jong), pp. 276–284, Utrecht, The Netherlands. [32]

Sato, Y., Takahashi, D., and Grimbergen, R. (2010). A Shogi Program Based on Monte-Carlo Tree Search. *ICGA Journal*, Vol. 33, No. 2, pp. 80–92. [62]

Schadd, M.P.D. (2011). *Selective Search in Games with Different Complexity.* Ph.D. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [5]

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*, Vol. 3, No. 1, pp. 57–66. [3, 6, 7, 21, 22, 34, 36, 39, 48, 53, 55, 108]

Schadd, M.P.D., Winands, M.H.M., and Uiterwijk, J.W.H.M. (2009). CHANCEPROBCUT: Forward Pruning in Chance Nodes. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games* (ed. P.L. Lanzi), pp. 178–185, IEEE. [107]

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. [24, 45, 67, 70, 122]

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. [24]

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [2]

Sevenster, M. (2008). The Complexity of Scotland Yard. *Interactive Logic* (eds. J. van Benthem, B. Löwe, and D. Gabbay), pp. 209–246. Amsterdam University Press, Amsterdam, The Netherlands. [96]

Shaker, N., Togelius, J., Yannakakis, G.N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., Smith, G., and Baumgarten, R. (2011). The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 3, No. 4, pp. 332–347. [2]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 314, pp. 256–275. [2, 12, 43]

Sheppard, B. (2002). World-championship-caliber Scrabble. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 241–275. [27]

Shibahara, K. and Kotani, Y. (2008). Combining Final Score with Winning Percentage by Sigmoid Function in Monte-Carlo Simulations. *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games* (eds. P. Hingston and L. Barone), pp. 183–190, IEEE. [41]

Silver, D. and Veness, J. (2010). Monte-Carlo Planning in Large POMDPs. *Advances in Neural Information Processing Systems 23* (eds. J.D. Lafferty, C.K.I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta), pp. 2164–2172. Curran Associates, Inc. [119, 126]

Soejima, Y., Kishimoto, A., and Watanabe, O. (2010). Evaluating Root Parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 278–287. [100]

Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E. (2004). Difficulty Scaling of Game AI. *5th International Conference on Intelligent Games and Simulation (GAMEON 2004)* (eds. A. El Rhalibi and D. Van Welden), pp. 33–37, EUROSIS, Belgium. [2]

Stankiewicz, J.A., Winands, M.H.M., and Uiterwijk, J.W.H.M. (2012). Monte-Carlo tree Search Enhancements for Havannah. *Advances in Computer Games (ACG 13)* (eds. H.J. van den Herik and A. Plaat), Vol. 7168 of *LNCS*, pp. 60–71, Springer-Verlag, Berlin, Germany. [78, 125]

Steinhauer, J. (2010). Monte-Carlo Twixt. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. [62]

Sturtevant, N.R. (2003a). A Comparison of Algorithms for Multi-Player Games. *Computers and Games (CG 2002)* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *LNCS*, pp. 108–122, Springer-Verlag, Berlin, Germany. [3, 19, 21, 22, 36, 51, 53, 63]

Sturtevant, N.R. (2003b). *Multi-Player Games: Algorithms and Approaches.* Ph.D. thesis, Computer Science Department, University of California, Los Angeles, USA. [4]

Sturtevant, N.R. (2003c). Last-Branch and Speculative Pruning Algorithms for Max^n. *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (eds. G. Gottlob and T. Walsh), pp. 669–675, Morgan Kaufmann. [20]

Sturtevant, N.R. (2008a). An Analysis of UCT in Multi-player Games. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 37–49, Springer-Verlag, Berlin, Germany. [3, 8, 29, 30, 34, 36, 83, 98]

Sturtevant, N.R. (2008b). An Analysis of UCT in Multi-player Games. *ICGA Journal*, Vol. 31, No. 4, pp. 195–208. [7, 34, 47, 48, 55, 61, 69]

Sturtevant, N.R. and Bowling, M.H. (2006). Robust Game Play Against Unknown Opponents. *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)* (eds. H. Nakashima, M.P. Wellman, G. Weiss, and P. Stone), pp. 713–719, ACM. [19]

Sturtevant, N.R. and Korf, R.E. (2000). On Pruning Techniques for Multi-Player Games. *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 201–207, AAAI Press / The MIT Press. [3, 5, 6, 7, 20, 21, 45, 51, 107]

Sturtevant, N.R., Zinkevich, R., and Bowling, M.H. (2006). Prob-Max^n: Playing N-player Games with Opponent Models. *The Twenty-First National Conference on Artificial Intelligence*, pp. 1057–1063, AAAI Press. [19]

Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA. [29, 83, 98]

Szita, I., Chaslot, G.M.J-B., and Spronck, P. (2010). Monte-Carlo Tree Search in Settlers of Catan. *Advances in Computer Games (ACG 12)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 21–32. Springer-Verlag, Berlin, Germany. [126]

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*, Vol. 4, No. 2, pp. 73–83. [32, 34, 78, 79, 125]

Tesauro, G. and Galperin, G.R. (1997). On-line Policy Improvement using Monte-Carlo Search. *Advances in Neural Information Processing Systems 9* (eds. M. Mozer, M.I. Jordan, and T. Petsche), pp. 1068–1074, MIT Press. [27, 43]

Teytaud, O. and Flory, S. (2011). Upper Confidence Trees with Short Term Partial Information. *Applications of Evolutionary Computation* (eds. C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A.I. Esparcia-Alcázar, J.J. Merelo, F. Neri, M. Preuss, H. Richter, J. Togelius, and G.N. Yannakakis), Vol. 6624 of *LNCS*, pp. 153–162. Springer-Verlag, Berlin, Germany. [119, 126]

Teytaud, F. and Teytaud, O. (2010a). Creating an Upper-Confidence-Tree program for Havannah. *Advances in Computer Games (ACG 12)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 65–74, Springer-Verlag, Berlin, Germany. [69, 76, 77, 145]

Teytaud, F. and Teytaud, O. (2010b). On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (eds. G.N. Yannakakis and J. Togelius), pp. 359–364, IEEE. [80]

Togelius, J., Yannakakis, G.N., Karakovskiy, S., and Shaker, N. (2011). Assessing Believability. *Believable Bots: Can Computers Play Like People?* (ed. P. Hingston), pp. 215–230. Springer-Verlag. [2]

Tom, D. and Müller, M. (2010). A Study of UCT and Its Enhancements in an Artificial Game. *Advances in Computer Games (ACG 12)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 55–64. Springer-Verlag, Berlin, Germany. [69, 76, 145]

Tom, D. and Müller, M. (2011). Computational Experiments with the RAVE Heuristic. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 69–80, Springer-Verlag, Berlin, Germany. [69]

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

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-Tree Search Algorithm Based on Realization Probability. *ICGA Journal*, Vol. 25, No. 3, pp. 132–144. [104, 105]

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B.V. Bowden), pp. 286–297, Pitman, London, United Kingdom. [2]

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. *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games* (eds. S-B. Cho, S.M. Lucas, and P. Hingston), pp. 87–94, IEEE. [99]

Whitehouse, D., Cowling, P.I., Powley, E.J., and Rollason, J. (2013). Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. *The Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2013).* To appear. [99]

Winands, M.H.M. and Björnsson, Y. (2008). Enhanced Realization Probability Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 329–342. [104, 105]

Winands, M.H.M. and Björnsson, Y. (2011). $\alpha\beta$-based Play-outs in Monte-Carlo Tree Search. *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games* (eds. S-B. Cho, S.M. Lucas, and P. Hingston), pp. 110–117, IEEE. [8, 79, 80, 83, 93, 126]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). The Quad Heuristic in Lines of Action. *ICGA Journal*, Vol. 24, No. 1, pp. 3–15. [43]

Winands, M.H.M., Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2006). The Relative History Heuristic. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu), Vol. 3846 of *LNCS*, pp. 262–272, Springer-Verlag, Berlin, Germany. [8, 24, 67, 70, 122]

Winands, M.H.M., Björnsson, Y., and Saito, J-T. (2008). Monte-Carlo Tree Search Solver. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 25–36, Springer-Verlag, Berlin, Germany. [47, 62, 64]

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*, Vol. 2, No. 4, pp. 239–250. [64, 70, 87]

Yajima, T., Tsuyoshi, H., Matsui, T., Hashimoto, J., and Spoerer, K. (2011). Node-Expansion Operators for the UCT Algorithm. *Computers and Games (CG 2010)* (eds. H.J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 116–123, Springer-Verlag, Berlin, Germany. [32]

Zhao, L. and Müller, M. (2008). Using Artificial Boundaries in the Game of Go. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *LNCS*, pp. 81–91, Springer-Verlag, Berlin, Germany. [69]

Zobrist, A.L. (1970). A New Hashing Method with Applications for Game Playing. Technical report, Computer Sciences Department, The University of Wisconsin. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp. 69–73. [25]

Zuckerman, I., Felner, A., and Kraus, S. (2009). Mixing Search Strategies for Multi-Player Games. *Proceedings of the Twenty-first International Joint Conferences on Artificial Intelligence (IJCAI-09)* (ed. C. Boutilier), pp. 646–651, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. [66, 125]

APPENDIX A

RAVE Formulas

This appendix provides a brief overview of various different RAVE variants that have been used over the past years. While they apply different formulas, they all have in common that AMAF values are used to guide the selection strategy while the number of visits in a node is still low.

In all formulas, n_i and n_p represent the number of times child i and its parent p have been visited, respectively. $\bar{x}_i$ is the win rate of child i and $\bar{x}_a$ is the AMAF value of child i.

Teytaud and Teytaud (2010a) used a variant of RAVE in their Havannah playing program. They applied the win rate and the AMAF value of the moves, combined with Hoeffding's bound, as shown in Formula A.1.

$$\begin{aligned} v_i &= (1-\beta)\bar{x}_i + \beta\bar{x}_a + e \\ \beta &= \frac{R}{R+n_i} \\ e &= \sqrt{K\ln(2+n_p)/n_i} \end{aligned} \tag{A.1}$$

In this formula, R represents the number of visits of parent p from which the weight of the RAVE component equals the weight of the 'greedy' component. e represents Hoeffding's bound, which is determined by the exploration constant K. This bound is used instead of UCT, which is applied in most other RAVE formulas.

Tom and Müller (2010) applied Formula A.2 to compute the value of the children in the MCTS tree in the artificial game Sum of Switches.

$$\begin{aligned} v_i &= \frac{n_i}{n_i+W_i}\bar{x}_i + \frac{W_i}{n_i+W_i}\bar{x}_a + C\sqrt{\frac{\ln(n_p)}{n_i+1}} \\ W_i &= \frac{n_a w_f w_i}{w_f + w_i n_a} \end{aligned} \tag{A.2}$$

In this formula, n_i and n_p represent the number of times child i and its parent p have been visited, respectively. $\bar{x}_i$ is the win rate of child i and $\bar{x}_a$ is the AMAF value of child i. $W_{i,a}$ is the unnormalized weighting of the RAVE estimator, where m_a indicates the number of times move a has been played after parent p, and w_i and

w_f are two parameters that determine the initial and the final weight of the RAVE component, respectively.

Another implementation of RAVE was provided by Rimmel *et al.* (2011a). This RAVE variant, which is given in Formula A.3, was used in both Havannah and Go.

$$v_i = \bar{x}_i + \beta \bar{x}_a + \sqrt{\frac{2 \ln n_p}{n_i}} \tag{A.3}$$

This selection strategy is quite similar to UCB1, with only the addition of the term $\beta \bar{x}_a$. Because β is a function that tends to 0 as the number of visits increases, the influence of the AMAF value decreases accordingly.

APPENDIX B

Detailed Results for Chapter 4

This appendix provides detailed results of the experiments performed in Chapter 4. First, Section B.1 discusses the detailed results of the experiments between the minimax-based multi-player search techniques max^n, paranoid and BRS. Next, Section B.2 gives an overview of the detailed results of the experiments between the MCTS-based search techniques MCTS-max^n, MCTS-paranoid and MCTS-BRS. Finally, Section B.3 provides a brief overview of the detailed results of the experiments between MCTS-max^n and BRS.

B.1 Comparison of Minimax-Based Techniques

This section contains a detailed analysis of the results in Table 4.2. The detailed results of max^n, paranoid, and BRS for the three-player variants of Chinese Checkers, Focus, and Rolit are given in Tables B.1, B.2, and B.3, respectively. Note that, in all tables, n is the number of games on which the win rates are based. The first notable observation is that players overall perform better if they play after a weaker player. For instance, in all games and with all time settings, paranoid performs significantly better after max^n, which is the weaker opponent, than after BRS. Similarly BRS achieves considerably higher win rates after max^n than after paranoid. A possible explanation for this phenomenon is that weaker players make more mistakes, and that players moving directly after weaker players can exploit these mistakes. For max^n, there appears to be no different between playing after paranoid or after BRS. This may be because both players play relatively strong and do not make many mistakes that max^n can exploit. Also, max^n may not be strong enough to exploit mistakes made anyway by paranoid or BRS. The second notable observation is that there appears to be no clear advantage in playing first in Chinese Checkers or Focus. Apparently, the difference in playing strength between the different minimax-based players is large enough so that the advantage of playing first does not have an influence on the outcome of the game. The same seems to be true for Focus. For Rolit, however, paranoid and BRS seem to have an advantage when playing last, if at least 1000 ms of thinking time is provided. In Rolit, the last player has the advantage that any captured stones cannot be lost anymore. Apparently, only players with a sufficient level of strength can benefit from this, because max^n, and paranoid and BRS with 250 ms of thinking time, do not have an advantage when moving last.

The results in Tables B.4, B.5, B.6, and B.7 show the win rates with a different number of instances of each player for four-player Chinese Checkers, Focus, Rolit, and Blokus, respectively. These results show how the player which has two instances of itself has a significant advantage. This is an expected result, because if there are more instances of a player type, the probability that one of these players wins the game is higher. Because each player type has this advantage in an equal number of games, the final results are fair and give a good indication of the playing strength of the different players.

Finally, Table B.8 gives the results with different numbers of instances for each player type in six-player Chinese Checkers. These results show that, similar to the previous experiments, the more instances there are of a certain player type, the higher its win rate. The only fair configurations are the ones with two instances of each player type. Interestingly, the win rates of these configurations are comparable to the win rates over all configurations.

B.2 Comparison of MCTS-Based Techniques

This section contains a detailed analysis of the results in Table 4.3. The detailed results of MCTS-max^n, MCTS-paranoid, and MCTS-BRS for the three-player variants of Chinese Checkers, Focus, and Rolit are given in Tables B.9, B.10, and B.11, respectively. Note that, due to space constraints, in the tables in this section the number of games n is omitted. This number, however, is the same as in the corresponding tables in the previous section. The observations presented in these tables are similar to those found in the experiments with the minimax-based techniques. Again, the players overall perform better if they play after a weaker opponent. For instance, MCTS-BRS overall performs better after MCTS-paranoid than after MCTS-max^n, which is a stronger opponent. Contrary to the experiments with the minimax-based players, with the MCTS-based players there appears to be a small advantage for the players when playing first in Chinese Checkers, while playing last gives a slight disadvantage. The reason may be that, because the difference in playing strength is smaller between the MCTS-based players, the turn order has a relatively larger influence on the win rates. In Focus, the turn order seems to have no influence on the playing strength. In Rolit, MCTS-max^n and MCTS-paranoid both have an advantage in moving last. MCTS-BRS does not have this advantage, similar to max^n, because it does not have the strength to benefit from this.

Tables B.12, B.13, B.14, and B.15 show the detailed results with a different number of instances of each MCTS-based player type for four-player Chinese Checkers, Focus, Rolit, and Blokus, respectively. The results in these tables show a similar pattern as the results of the experiments with the minimax-based players in the four-player games. All player types have a significant advantage if there are two instances of them. When all results are combined, these advantages cancel each other out.

Finally, Table B.16 provides the results with different numbers of instances for each MCTS-based player type in six-player Chinese Checkers. Again, the more instances there are of a certain player type, the higher its win rate. The win rates of the configurations with two instances of each player type are slightly different to the win rates over all configurations. However, the results are still comparable, as MCTS-max^n is the strongest technique both with only the fair configurations and with all configurations.

B.3 BRS versus MCTS-max^n

This section contains a detailed analysis of the matches between BRS and MCTS-max^n in Tables 4.5 and 4.6.

The detailed results of MCTS-max^n versus BRS for the three-player variants of Chinese Checkers, Focus, and Rolit are given in Tables B.17, B.18, and B.19, respectively. If there are two player types and three players, there are two instances of one player type, and one instance of the other. The results in the tables show that, for all games, the player type with two instances has a considerable advantage over the other. If the two player types would be equally strong, the type with two instances would win $66\frac{2}{3}\%$ of the games. The results in Table B.19 also show that the player who moves last in Rolit has an advantage. For example, in the games with one instance of MCTS-max^n, it wins the most games if it plays last, i.e., in the B–B–M configuration, compared to the B–M–B and M–B–B configurations.

Subsequently, Tables B.20, B.21, B.22, and B.23 show the detailed results with a different number of instances of MCTS-max^n and BRS for four-player Chinese Checkers, Focus, Rolit, and Blokus, respectively. The results in these tables show that the unfair configurations, with three instances of one player type and only one of the other, do not have a significant influence on the overall win rate. Similar to the other experiments, the win rates of the fair configurations are comparable to the overall win rates.

Finally, Table B.24 provides the results with different numbers of instances for MCTS-max^n and BRS in six-player Chinese Checkers. The win rates of the fair configurations, with three instances of each player type, are relatively close to the win rates over all configurations. While there are configurations with five instances of one player type and only one of the other, they do not have a large influence on the overall win rate.

Table B.1 Detailed results of the minimax-based techniques in three-player Chinese Checkers.

250 ms		*n*	Max^n	Paranoid	BRS
Order	M–P–B	175	2.3%	33.1%	64.6%
	M–B–P	175	1.1%	16.0%	82.9%
	P–M–B	175	0.6%	29.7%	69.7%
	P–B–M	175	1.1%	29.1%	69.7%
	B–M–P	175	0.6%	25.7%	73.7%
	B–P–M	175	0.6%	15.4%	84.0%
Position	After max^n	350	-	29.3%	78.9%
	After paranoid	350	0.8%	-	69.3%
	After BRS	350	1.3%	20.4%	-
	First	350	1.7%	29.4%	78.9%
	Second	350	0.6%	24.3%	76.3%
	Third	350	0.9%	20.9%	67.2%

1000 ms		*n*	Max^n	Paranoid	BRS
Order	M–P–B	175	0.6%	30.9%	68.6%
	M–B–P	175	0.6%	17.7%	81.7%
	P–M–B	175	1.1%	11.4%	87.4%
	P–B–M	175	1.7%	20.6%	77.7%
	B–M–P	175	1.1%	18.3%	80.6%
	B–P–M	175	0.6%	24.0%	75.4%
Position	After max^n	350	-	23.3%	81.5%
	After paranoid	350	0.8%	-	75.6%
	After BRS	350	1.1%	17.7%	-
	First	350	0.6%	16.0%	78.0%
	Second	350	1.1%	27.5%	79.7%
	Third	350	1.2%	18.0%	78.0%

5000 ms		*n*	Max^n	Paranoid	BRS
Order	M–P–B	175	1.7%	29.1%	69.1%
	M–B–P	175	1.7%	16.6%	81.7%
	P–M–B	175	1.7%	17.1%	81.1%
	P–B–M	175	1.1%	22.9%	76.0%
	B–M–P	175	1.7%	24.0%	74.3%
	B–P–M	175	0.6%	20.0%	79.4%
Position	After max^n	350	-	25.3%	80.7%
	After paranoid	350	1.3%	-	73.1%
	After BRS	350	1.5%	17.9%	-
	First	350	1.7%	20.0%	76.9%
	Second	350	1.7%	24.6%	78.9%
	Third	350	0.9%	20.3%	75.1%

Table B.2 Detailed results of the minimax-based techniques in three-player Focus.

250 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	5.7%	44.0%	50.3%
	M–B–P	175	3.4%	33.1%	63.4%
	P–M–B	175	4.6%	30.9%	64.6%%
	P–B–M	175	3.4%	41.1%	55.4%
	B–M–P	175	9.1%	36.6%	54.3%
	B–P–M	175	0.0%	28.6%	71.4%
Position	After max^n	350	-	40.6%	66.5%
	After paranoid	350	2.7%	-	53.3%
	After BRS	350	6.1%	30.9%	-
	First	350	4.6%	36.0%	62.9%
	Second	350	6.9%	36.3%	59.4%
	Third	350	1.7%	34.9%	57.5%

1000 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	6.3%	41.7%	52.0%
	M–B–P	175	0.6%	12.0%	87.4%
	P–M–B	175	2.9%	19.4%	77.7%
	P–B–M	175	6.3%	29.7%	64.0%
	B–M–P	175	6.9%	41.7%	51.4%
	B–P–M	175	0.0%	26.9%	73.1%
Position	After max^n	350	-	37.7%	79.4%
	After paranoid	350	1.2%	-	55.8%
	After BRS	350	6.5%	19.4%	-
	First	350	3.5%	24.6%	62.3%
	Second	350	4.9%	34.3%	75.7%
	Third	350	3.2%	26.9%	64.9%

5000 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	6.3%	34.9%	58.9%
	M–B–P	175	1.7%	21.7%	76.6%
	P–M–B	175	1.7%	30.9%	67.4%
	P–B–M	175	6.3%	25.1%	68.6%
	B–M–P	175	6.9%	28.6%	64.6%
	B–P–M	175	0.0%	22.3%	77.7%
Position	After max^n	350	-	29.5%	73.9%
	After paranoid	350	1.1%	-	64.0%
	After BRS	350	6.5%	25.0%	-
	First	350	4.0%	28.0%	71.2%
	Second	350	4.3%	28.6%	72.6%
	Third	350	3.2%	25.2%	63.2%

Table B.3 Detailed results of the minimax-based techniques in three-player Rolit.

250 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	7.1%	52.3%	40.6%
	M–B–P	175	7.4%	24.9%	67.7%
	P–M–B	175	8.0%	22.0%	70.0%
	P–B–M	175	10.9%	45.7%	43.4%
	B–M–P	175	8.3%	55.7%	36.0%
	B–P–M	175	6.9%	30.9%	62.3%
Position	After max^n	350	-	51.2%	66.7%
	After paranoid	350	7.4%	-	40.0%
	After BRS	350	8.8%	25.9%	-
	First	350	7.3%	33.9%	49.2%
	Second	350	8.2%	41.6%	55.6%
	Third	350	8.9%	40.3%	55.3%

1000 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	7.4%	41.1%	51.4%
	M–B–P	175	5.4%	41.1%	53.4%
	P–M–B	175	13.1%	25.4%	61.4%
	P–B–M	175	9.7%	46.3%	44.0%
	B–M–P	175	7.1%	50.0%	42.9%
	B–P–M	175	10.9%	34.3%	54.9%
Position	After max^n	350	-	45.8%	56.6%
	After paranoid	350	9.8%	-	46.1%
	After BRS	350	8.1%	33.6%	-
	First	350	6.4%	35.9%	49.8%
	Second	350	10.1%	37.7%	48.7%
	Third	350	10.3%	45.6%	56.4%

5000 ms		n	Max^n	Paranoid	BRS
Order	M–P–B	175	2.3%	46.9%	50.9%
	M–B–P	175	5.4%	45.4%	49.1%
	P–M–B	175	9.4%	26.6%	64.0%
	P–B–M	175	5.4%	48.0%	46.6%
	B–M–P	175	7.7%	60.9%	31.4%
	B–P–M	175	7.4%	44.3%	48.3%
Position	After max^n	350	-	45.8%	54.4%
	After paranoid	350	8.6%	-	46.1%
	After BRS	350	8.1%	36.9%	-
	First	350	6.4%	35.9%	45.6%
	Second	350	10.1%	42.7%	48.7%
	Third	350	8.6%	45.6%	56.4%

Table B.4 Detailed results of the minimax-based techniques in four-player Chinese Checkers.

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	11.2%	10.3%	78.4%
1	2	1	350	2.0%	19.3%	78.7%
1	1	2	350	2.6%	5.5%	92.0%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	7.5%	21.8%	70.7%
1	2	1	350	2.9%	34.8%	62.1%
1	1	2	350	1.7%	12.4%	85.9%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	11.5%	19.3%	69.3%
1	2	1	350	3.4%	29.3%	67.0%
1	1	2	350	2.0%	9.8%	88.2%

Table B.5 Detailed results of the minimax-based techniques in four-player Focus.

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	17.0%	16.1%	67.0%
1	2	1	350	7.5%	25.9%	66.7%
1	1	2	350	3.7%	10.6%	85.6%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	13.2%	20.7%	66.1%
1	2	1	350	4.0%	39.9%	56.0%
1	1	2	350	3.7%	11.5%	84.8%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	12.4%	29.6%	59.2%
1	2	1	350	4.3%	42.0%	54.3%
1	1	2	350	3.4%	12.6%	83.9%

Table B.6 Detailed results of the minimax-based techniques in four-player Rolit.

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	26.1%	41.1%	32.8%
1	2	1	350	10.9%	56.7%	32.4%
1	1	2	350	10.7%	26.8%	62.5%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	24.4%	39.4%	36.2%
1	2	1	350	10.3%	59.6%	30.0%
1	1	2	350	9.3%	29.0%	61.6%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	27.2%	36.2%	37.8%
1	2	1	350	7.5%	61.8%	31.3%
1	1	2	350	10.3%	29.4%	60.2%

Table B.7 Detailed results of the minimax-based techniques in four-player Blokus.

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	29.1%	28.0%	43.0%
1	2	1	350	13.8%	43.2%	43.0%
1	1	2	350	10.5%	20.0%	69.5%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	25.5%	21.8%	52.7%
1	2	1	350	12.7%	46.6%	40.7%
1	1	2	350	7.9%	20.4%	71.8%

Instances						
M	P	B	*n*	Max^n	Paranoid	BRS
2	1	1	350	14.2%	20.0%	67.0%
1	2	1	350	6.2%	38.5%	55.9%
1	1	2	350	5.5%	12.4%	82.2%

Table B.8 Detailed results of the minimax-based techniques in six-player Chinese Checkers.

250 ms

Instances				Win rate		
M	P	B	n	Max^n	Paranoid	BRS
4	1	1	60	55.0%	8.3%	36.7%
3	2	1	120	30.8%	15.8%	53.3%
3	1	2	120	20.0%	5.0%	75.0%
2	3	1	120	19.2%	27.5%	53.3%
2	2	2	180	11.1%	13.9%	75.0%
2	1	3	120	6.7%	2.5%	90.8%
1	4	1	60	10.0%	41.7%	48.3%
1	3	2	120	5.8%	16.7%	77.5%
1	2	3	120	2.5%	9.2%	88.3%
1	1	4	60	1.7%	5.0%	93.3%

1000 ms

Instances				Win rate		
M	P	B	n	Max^n	Paranoid	BRS
4	1	1	60	42.8%	6.1%	51.1%
3	2	1	120	20.0%	20.0%	60.0%
3	1	2	120	16.9%	4.4%	78.6%
2	3	1	120	11.9%	25.3%	62.8%
2	2	2	180	15.2%	10.7%	74.1%
2	1	3	120	6.7%	5.0%	88.3%
1	4	1	60	7.2%	40.6%	52.2%
1	3	2	120	5.8%	13.3%	80.8%
1	2	3	120	1.9%	10.3%	87.8%
1	1	4	60	1.7%	3.3%	95.0%

5000 ms

Instances				Win rate		
M	P	B	n	Max^n	Paranoid	BRS
4	1	1	60	45.0%	6.7%	48.3%
3	2	1	120	36.9%	18.6%	44.4%
3	1	2	120	17.5%	7.5%	75.0%
2	3	1	120	16.4%	26.4%	57.2%
2	2	2	180	15.7%	11.3%	73.0%
2	1	3	120	11.7%	5.0%	83.3%
1	4	1	60	12.2%	32.2%	55.6%
1	3	2	120	9.4%	13.6%	76.9%
1	2	3	120	5.6%	9.7%	84.7%
1	1	4	60	10.0%	0.0%	90.0%

Table B.9 Detailed results of the MCTS-based techniques in three-player Chinese Checkers.

250 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	38.3%	30.9%	30.9%
	M–B–P	44.0%	26.3%	29.7%
	P–M–B	45.7%	33.1%	21.1%
	P–B–M	33.7%	30.9%	35.4%
	B–M–P	38.3%	20.6%	41.1%
	B–P–M	41.1%	29.1%	29.7%
Position	After MCTS-max^n	-	27.5%	26.8%
	After MCTS-paranoid	43.6%	-	35.8%
	After MCTS-BRS	36.8%	29.5%	-
	First	41.2%	32.0%	35.4%
	Second	42.0%	30.0%	32.6%
	Third	37.4%	23.5%	26.0%

1000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	50.3%	18.3%	31.4%
	M–B–P	56.0%	22.9%	21.1%
	P–M–B	49.1%	24.6%	26.3%
	P–B–M	49.1%	18.9%	32.0%
	B–M–P	53.1%	14.3%	32.6%
	B–P–M	52.6%	20.6%	26.9%
Position	After MCTS-max^n	-	17.2%	24.8%
	After MCTS-paranoid	52.6%	-	32.0%
	After MCTS-BRS	50.8%	22.7%	-
	First	53.2%	21.8%	29.8%
	Second	51.1%	19.5%	26.6%
	Third	50.9%	18.6%	28.9%

5000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	62.9%	5.7%	31.4%
	M–B–P	63.4%	13.7%	22.9%
	P–M–B	66.9%	17.1%	16.0%
	P–B–M	54.3%	10.3%	35.4%
	B–M–P	57.7%	5.1%	37.1%
	B–P–M	67.4%	13.1%	19.4%
Position	After MCTS-max^n	-	7.0%	19.4%
	After MCTS-paranoid	65.9%	-	34.6%
	After MCTS-BRS	58.3%	14.6%	-
	First	63.2%	13.7%	28.3%
	Second	62.3%	9.4%	29.2%
	Third	60.9%	9.4%	23.7%

Table B.10 Detailed results of the MCTS-based techniques in three-player Focus.

250 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	31.4%	29.1%	39.4%
	M–B–P	50.9%	22.9%	26.3%
	P–M–B	41.7%	32.6%	25.7%
	P–B–M	32.6%	29.7%	37.7%
	B–M–P	36.0%	35.4%	28.6%
	B–P–M	52.0%	24.6%	23.4%
Position	After MCTS-max^n	-	31.4%	25.1%
	After MCTS-paranoid	48.2%	-	35.2%
	After MCTS-BRS	33.3%	26.7%	-
	First	41.2%	31.2%	26.0%
	Second	38.9%	26.9%	32.0%
	Third	42.3%	29.6%	32.6%

1000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	35.4%	22.9%	41.7%
	M–B–P	43.4%	30.3%	26.3%
	P–M–B	49.7%	23.4%	26.9%
	P–B–M	43.4%	28.6%	28.0%
	B–M–P	37.1%	24.6%	38.3%
	B–P–M	48.6%	26.9%	24.6%
Position	After MCTS-max^n	-	25.4%	25.9%
	After MCTS-paranoid	47.2%	-	36.0%
	After MCTS-BRS	38.6%	26.9%	-
	First	39.4%	26.0%	31.5%
	Second	43.4%	24.9%	27.2%
	Third	46.0%	27.5%	34.3%

5000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	42.3%	17.1%	40.6%
	M–B–P	47.4%	25.1%	27.4%
	P–M–B	55.4%	19.4%	25.1%
	P–B–M	42.3%	12.6%	45.1%
	B–M–P	50.9%	16.6%	32.6%
	B–P–M	53.7%	26.9%	19.4%
Position	After MCTS-max^n	-	15.4%	24.0%
	After MCTS-paranoid	52.2%	-	39.4%
	After MCTS-BRS	45.2%	23.8%	-
	First	44.9%	16.0%	26.0%
	Second	53.2%	22.0%	36.3%
	Third	48.0%	20.9%	32.9%

Table B.11 Detailed results of the MCTS-based techniques in three-player Rolit.

250 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	48.3%	29.4%	22.3%
	M–B–P	41.7%	38.9%	19.4%
	P–M–B	47.1%	32.3%	20.6%
	P–B–M	63.4%	17.4%	19.1%
	B–M–P	51.1%	27.1%	21.7%
	B–P–M	51.7%	28.3%	20.5%
Position	After MCTS-max^n	-	24.6%	20.0%
	After MCTS-paranoid	46.8%	-	21.0%
	After MCTS-BRS	54.3%	33.2%	-
	First	45.0%	24.9%	20.9%
	Second	49.1%	28.9%	19.3%
	Third	57.6%	33.0%	21.5%

1000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	60.3%	20.3%	19.4%
	M–B–P	53.4%	35.4%	11.1%
	P–M–B	58.9%	28.0%	13.1%
	P–B–M	68.9%	14.9%	16.3%
	B–M–P	50.0%	23.4%	26.6%
	B–P–M	52.3%	25.7%	22.0%
Position	After MCTS-max^n	-	19.5%	15.4%
	After MCTS-paranoid	54.9%	-	20.8%
	After MCTS-BRS	59.7%	29.7%	-
	First	56.9%	21.5%	24.3%
	Second	54.5%	23.0%	13.7%
	Third	60.0%	29.4%	16.3%

5000 ms		MCTS-max^n	MCTS-paranoid	MCTS-BRS
Order	M–P–B	60.3%	21.7%	18.0%
	M–B–P	58.3%	31.7%	10.0%
	P–M–B	65.7%	18.6%	15.7%
	P–B–M	71.1%	12.3%	16.6%
	B–M–P	54.0%	21.1%	24.9%
	B–P–M	69.7%	16.9%	13.4%
Position	After MCTS-max^n	-	18.4%	13.0%
	After MCTS-paranoid	64.6%	-	19.8%
	After MCTS-BRS	61.8%	22.4%	-
	First	59.3%	15.5%	19.2%
	Second	59.9%	19.3%	13.3%
	Third	70.4%	26.4%	16.9%

Table B.12 Detailed results of the MCTS-based techniques in four-player Chinese Checkers.

250 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	66.1%	17.8%	16.1%
1	2	1	40.2%	42.5%	17.2%
1	1	2	37.1%	26.4%	36.5%

1000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	71.3%	9.8%	19.0%
1	2	1	43.7%	33.6%	22.7%
1	1	2	43.4%	13.5%	43.1%

5000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	78.2%	7.5%	14.4%
1	2	1	60.3%	19.0%	20.7%
1	1	2	54.6%	10.1%	35.3%

Table B.13 Detailed results of the MCTS-based techniques in four-player Focus.

250 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	54.6%	23.0%	22.4%
1	2	1	29.3%	50.0%	20.7%
1	1	2	27.9%	27.0%	45.1%

1000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	60.6%	17.5%	21.8%
1	2	1	33.0%	42.5%	24.4%
1	1	2	29.9%	18.1%	52.0%

5000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	72.7%	8.6%	18.7%
1	2	1	45.7%	31.0%	23.3%
1	1	2	38.5%	16.7%	44.8%

Table B.14 Detailed results of the MCTS-based techniques in four-player Rolit.

250 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	63.8%	18.5%	17.7%
1	2	1	33.0%	47.6%	19.4%
1	1	2	34.1%	28.1%	37.8%

1000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	68.2%	17.2%	14.6%
1	2	1	40.0%	42.5%	17.5%
1	1	2	41.8%	22.7%	35.5%

5000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	77.5%	10.2%	12.2%
1	2	1	45.8%	34.1%	20.1%
1	1	2	46.1%	18.1%	35.8%

Table B.15 Detailed results of the MCTS-based techniques in four-player Blokus.

250 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	54.7%	23.2%	22.1%
1	2	1	26.5%	54.0%	19.5%
1	1	2	29.0%	26.2%	44.8%

1000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	52.7%	27.0%	20.4%
1	2	1	27.2%	51.5%	21.3%
1	1	2	28.0%	27.3%	44.7%

5000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
2	1	1	51.5%	26.8%	21.7%
1	2	1	21.8%	50.9%	27.3%
1	1	2	27.6%	25.3%	47.1%

Table B.16 Detailed results of the MCTS-based techniques in six-player Chinese Checkers.

250 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
4	1	1	71.7%	10.0%	18.3%
3	2	1	64.2%	28.3%	7.5%
3	1	2	63.3%	15.8%	20.8%
2	3	1	43.3%	41.7%	15.0%
2	2	2	53.9%	25.6%	20.6%
2	1	3	51.7%	15.0%	33.3%
1	4	1	26.7%	63.3%	10.0%
1	3	2	34.2%	37.5%	28.3%
1	2	3	24.2%	31.7%	44.2%
1	1	4	23.3%	18.3%	58.3%

1000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
4	1	1	86.7%	3.3%	10.0%
3	2	1	76.7%	17.5%	5.8%
3	1	2	70.0%	8.3%	21.7%
2	3	1	58.3%	22.5%	19.2%
2	2	2	55.0%	20.6%	24.4%
2	1	3	60.0%	8.3%	31.7%
1	4	1	33.3%	55.0%	11.7%
1	3	2	30.0%	42.5%	27.5%
1	2	3	36.7%	21.7%	41.7%
1	1	4	30.0%	11.7%	58.3%

5000 ms

Instances			Win rate		
M	P	B	MCTS-Max^n	MCTS-Paranoid	MCTS-BRS
4	1	1	85.0%	6.7%	8.3%
3	2	1	70.0%	15.8%	14.2%
3	1	2	77.5%	4.2%	18.3%
2	3	1	71.7%	16.7%	11.7%
2	2	2	67.2%	12.2%	20.6%
2	1	3	57.5%	6.7%	35.8%
1	4	1	50.0%	31.7%	18.3%
1	3	2	41.7%	25.8%	32.5%
1	2	3	54.2%	15.8%	30.0%
1	1	4	20.0%	8.3%	71.7%

Table B.17 Detailed results of BRS versus MCTS-max^n in three-player Chinese Checkers.

250 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	66.9%	33.1%
	M–B–M	175	59.4%	40.6%
	B–M–M	175	66.9%	33.1%
	B–B–M	175	100.0%	0.0%
	B–M–B	175	98.9%	1.1%
	M–B–B	175	97.7%	2.3%
Instances	1	525	64.4%	1.1%
	2	525	98.9%	35.6%

1000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	34.9%	65.1%
	M–B–M	175	35.4%	64.6%
	B–M–M	175	37.7%	62.3%
	B–B–M	175	83.4%	16.6%
	B–M–B	175	77.1%	22.9%
	M–B–B	175	77.1%	22.9%
Instances	1	525	36.0%	20.8%
	2	525	79.2%	64.0%

5000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	10.9%	89.1%
	M–B–M	175	13.7%	86.3%
	B–M–M	175	13.1%	86.9%
	B–B–M	175	62.3%	37.7%
	B–M–B	175	52.0%	48.0%
	M–B–B	175	38.9%	61.1%
Instances	1	525	12.6%	49.0%
	2	525	51.0%	87.4%

Table B.18 Detailed results of BRS versus MCTS-max^n in three-player Focus.

250 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	42.3%	57.7%
	M–B–M	175	42.3%	57.7%
	B–M–M	175	48.0%	52.0%
	B–B–M	175	80.0%	20.0%
	B–M–B	175	85.7%	14.3%
	M–B–B	175	78.9%	21.1%
Instances	1	525	44.2%	18.5%
	2	525	81.5%	55.8%

1000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	22.9%	77.1%
	M–B–M	175	22.9%	77.1%
	B–M–M	175	21.1%	78.9%
	B–B–M	175	70.9%	29.1%
	B–M–B	175	69.1%	30.9%
	M–B–B	175	70.3%	29.7%
Instances	1	525	22.3%	29.9%
	2	525	70.1%	77.7%

5000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	9.1%	90.9%
	M–B–M	175	13.1%	86.9%
	B–M–M	175	8.6%	91.4%
	B–B–M	175	58.9%	41.1%
	B–M–B	175	67.4%	32.6%
	M–B–B	175	65.7%	34.3%
Instances	1	525	10.3%	36.0%
	2	525	64.0%	89.7%

Table B.19 Detailed results of BRS versus MCTS-max^n in three-player Rolit.

250 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	7.7%	92.3%
	M–B–M	175	12.9%	87.1%
	B–M–M	175	13.1%	86.9%
	B–B–M	175	27.7%	72.3%
	B–M–B	175	44.9%	55.1%
	M–B–B	175	49.1%	50.9%
Instances	1	525	11.2%	59.4%
	2	525	40.6%	88.8%

1000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	3.4%	96.6%
	M–B–M	175	4.0%	96.0%
	B–M–M	175	2.9%	97.1%
	B–B–M	175	20.9%	79.1%
	B–M–B	175	23.1%	76.9%
	M–B–B	175	38.0%	62.0%
Instances	1	525	3.4%	72.7%
	2	525	27.3%	96.6%

5000 ms		n	BRS	MCTS-max^n
Order	M–M–B	175	4.0%	96.0%
	M–B–M	175	4.0%	96.0%
	B–M–M	175	2.9%	97.1%
	B–B–M	175	17.1%	82.9%
	B–M–B	175	21.7%	78.3%
	M–B–B	175	28.6%	71.4%
Instances	1	525	3.6%	77.5%
	2	525	22.5%	96.4%

Table B.20 Detailed results of BRS versus MCTS-max^n in four-player Chinese Checkers.

250 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	97.7%	2.3%
2	2	450	83.1%	16.9%
1	3	300	42.0%	58.0%

1000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	73.7%	26.3%
2	2	450	42.7%	57.3%
1	3	300	13.0%	87.0%

5000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	49.7%	50.3%
2	2	450	16.7%	83.3%
1	3	300	3.7%	96.3%

Table B.21 Detailed results of BRS versus MCTS-max^n in four-player Focus.

250 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	86.3%	13.7%
2	2	450	58.4%	41.6%
1	3	300	28.0%	72.0%

1000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	76.0%	24.0%
2	2	450	45.1%	54.9%
1	3	300	16.3%	83.7%

5000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	68.0%	32.0%
2	2	450	23.1%	76.9%
1	3	300	7.3%	92.7%

Table B.22 Detailed results of BRS versus MCTS-max^n in four-player Rolit.

250 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	54.1%	45.9%
2	2	450	24.8%	75.2%
1	3	300	9.5%	90.5%

1000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	40.2%	59.8%
2	2	450	15.3%	84.7%
1	3	300	6.2%	93.8%

5000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	40.7%	59.3%
2	2	450	13.3%	86.7%
1	3	300	2.5%	97.5%

Table B.23 Detailed results of BRS versus MCTS-max^n in four-player Blokus.

250 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	67.2%	32.8%
2	2	450	40.6%	59.4%
1	3	300	19.6%	80.4%

1000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	41.6%	58.4%
2	2	450	20.4%	79.6%
1	3	300	7.0%	93.0%

5000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
3	1	300	20.8%	79.2%
2	2	450	6.6%	93.4%
1	3	300	2.7%	97.3%

Table B.24 Detailed results of BRS versus MCTS-max^n in six-player Chinese Checkers.

250 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
5	1	102	100.0%	0.0%
4	2	255	92.2%	7.8%
3	3	340	73.2%	26.8%
2	4	255	42.0%	58.0%
1	5	102	9.8%	90.2%

1000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
5	1	102	70.6%	29.4%
4	2	255	45.9%	54.1%
3	3	340	24.7%	75.3%
2	4	255	6.3%	93.7%
1	5	102	4.9%	95.1%

5000 ms

Instances			Win rate	
B	M	n	BRS	MCTS-max^n
5	1	102	42.2%	57.8%
4	2	255	19.2%	80.8%
3	3	340	7.6%	92.4%
2	4	255	2.7%	97.3%
1	5	102	0.0%	100.0%

Index

Summary

AI research has been quite successful in the field of two-player zero-sum games, such as chess, checkers, and Go. This has been achieved by developing two-player search techniques. However, a large number of games does not belong to the area where these search techniques are unconditionally applicable. Multi-player games are an example of such domains. This thesis focuses on two different categories of multi-player games: (1) deterministic multi-player games with perfect information and (2) multi-player hide-and-seek games. In particular, it investigates how Monte-Carlo Tree Search (MCTS) can be improved for games in these two categories. This technique has achieved impressive results in computer Go, but has also shown to be beneficial in a range of other domains.

Chapter 1 provides an introduction to games and the role they play in the field of AI and gives an overview of different game properties. It also defines the notion of multi-player games and discusses the two different categories of multi-player games that are investigated in this thesis. The following problem statement guides the research.

Problem statement: *How can Monte-Carlo Tree Search be improved to increase the performance in multi-player games?*

In order to answer the problem statement, four different research questions have been formulated. They deal with (1) incorporating different search policies in MCTS, (2) improving the selection phase of MCTS, (3) improving the playout phase of MCTS, and (4) adapting MCTS to a hide-and-seek game.

Chapter 2 introduces the basic definitions for game-tree search. It discusses the basic two-player minimax-based search techniques and some of their enhancements. Furthermore, this chapter describes three different minimax-based search techniques for multi-player games, namely max^n, paranoid, and Best-Reply Search (BRS). Finally, it introduces MCTS, and how it can be applied to two-player and multi-player games.

Chapter 3 describes the test environment used to answer the first three research questions. We describe the rules and the employed domain knowledge for the four deterministic perfect-information games used in this thesis: Chinese Checkers, Focus, Rolit, and Blokus. Furthermore, for each of the games the state-space complexity and the game-tree complexity is given. They provide an indication on how difficult the games are for computers to play them optimally. Finally, this chapter introduces the

program MAGE, which is used to run the experiments in Chapters 4, 5, and 6 of this thesis.

The advantage of MCTS is that it can be extended to multi-player games. In the standard multi-player variant of MCTS, each player is concerned with maximizing his own win rate. This variant is therefore comparable to the minimax-based multi-player search technique max^n, where each player tries to maximize his own score, regardless of the scores of the other players. Other multi-player search policies, such as the ones of paranoid and BRS, could also be considered. This led us to the first research question.

> **Research question 1**: *How can multi-player search policies be incorporated in MCTS?*

Chapter 4 answers the first research question by incorporating the paranoid and BRS search policies, along with the default max^n policy, in MCTS. With these search policies the selection and the backpropagation phases of MCTS are altered. In the MCTS framework, the max^n search policy appeared to perform best. The advantages of paranoid and BRS in the minimax framework do not apply in MCTS, because $\alpha\beta$ pruning is not applicable in MCTS. An additional problem with MCTS-BRS may be that, in the tree, invalid positions are investigated, which may reduce the reliability of the playouts as well. Still, MCTS-paranoid and MCTS-BRS overall achieved decent win rates against MCTS-max^n, especially with lower time settings. Based on the results we may conclude that the max^n search policy is the most robust, though the BRS and paranoid search policies can still be competitive. Finally, we enhanced the max^n search policy by proposing a multi-player variant of MCTS-Solver, called MP-MCTS-Solver. This variant is able to prove the game-theoretic value of a position. A win rate between 53% and 55% was achieved in the sudden-death game Focus. We may conclude that proving game-theoretic values improves the playing strength of MCTS in a multi-player sudden-death domain.

An important phase in the MCTS algorithm is the selection phase. During the selection phase, the search tree is traversed until a leaf node is reached. A selection strategy determines how the tree is traversed. Over the past years, several selection strategies and enhancements have been developed for different types of games. The most popular selection strategy is Upper Confidence bounds applied to Trees (UCT). There exist various enhancements for the UCT selection strategy. Most of them are domain dependent, which means that they cannot unconditionally be applied in every domain. A domain-independent enhancement is Rapid Action Value Estimation (RAVE). This enhancement is based on the all-moves-as-first (AMAF) heuristic. RAVE is in particular successful in the field of computer Go, but less successful in others, such as the multi-player game Chinese Checkers. This led us to the second research question.

> **Research question 2**: *How can the selection phase of MCTS be enhanced in perfect-information multi-player games?*

Chapter 5 answers this question by proposing a new domain-independent selection strategy called Progressive History. This technique is a combination of the relative history heuristic and Progressive Bias. Contrary to RAVE, Progressive History maintains its gathered data in a global table, in a similar way as the playout strategy Move-Average Sampling Technique. Progressive History was a significant improvement in all games with different numbers of players. In a comparison with UCT, Progressive History gained the highest win rates in the two-player variants, winning around 80% of the games. Moreover, Progressive History performed better than standard UCT in the multi-player variants as well. Progressive AMAF, which applies AMAF values instead of history values, overall performed significantly worse than Progressive History. Additionally, experiments in the two-player game Havannah showed that Progressive History performed better in this game than RAVE. Furthermore, experiments revealed that Progressive History also significantly increases the playing strength of the seekers in the hide-and-seek game Scotland Yard. Based on the results we may conclude that Progressive History considerably enhances MCTS in both two-player and multi-player games.

Similar to the selection phase, the playout phase is an important phase in the MCTS algorithm. During the playout phase, the game is finished by playing moves that are selected using a playout strategy. More realistic playouts usually provide more reliable results, thus increasing the playing strength of an MCTS-based player. Playouts can be made more realistic by adding domain knowledge. The disadvantage is that this may reduce the number of playouts per second, decreasing the playing strength. The challenge is to find a good balance between speed and quality of the playouts. For the two-player game Lines of Action (LOA), relatively time-expensive two-ply $\alpha\beta$ searches in the playout phase of MCTS have been introduced. While this significantly reduced the number of playouts per second, it increased the overall playing strength by improving the quality of the playouts. This led us to the third research question.

> **Research question 3**: *How can the playouts of MCTS be enhanced in perfect-information multi-player games?*

Chapter 6 answers this research question by introducing two-ply searches, which are equipped with a heuristic evaluation function, for selecting moves in the playout phase in MCTS for multi-player games. Three different search techniques were investigated for multi-player games, namely max$^{\mathrm{n}}$, paranoid and BRS. These playout strategies were compared against random, greedy and one-ply playouts to determine how to balance search and speed in the playouts of multi-player MCTS. The results showed that search-based playouts significantly improved the quality of the playouts in MCTS. Among the different playout strategies, BRS performed best, followed by paranoid and max$^{\mathrm{n}}$. This benefit was countered by a reduction of the number of playouts per second. Especially BRS and max$^{\mathrm{n}}$ suffered from this effect. Among the tested two-ply search-based playouts, paranoid overall performed best with both short and long time settings. With more thinking time, the two-ply search-based playout strategies performed relatively better than the one-ply and greedy strategies. This indicates that with longer time settings, more computationally expensive playouts may be used to increase the playing strength of MCTS-based players. Based on the experimental results we may conclude that search-based playouts for multi-player games may be beneficial if the players receive sufficient thinking time.

The previous chapters discussed the application and enhancement of MCTS to deterministic multi-player games with perfect information. In Chapter 7, we shift our focus to hide-and-seek games. In this thesis we are interested in hide-and-seek games that have the following three properties. First, they feature imperfect information for some players. Second, some of the players have to cooperate in a fixed coalition. Though these players have a common goal, each player behaves autonomously and explicit communication between the players is not applied. Third, they are asymmetric. The different players have different types of goals. A game that features these properties is the pursuit-evasion game Scotland Yard. In this multi-player game, five seekers cooperate to try to capture a hider, which only shows its location on regular intervals. This led us to the fourth research question.

> **Research question 4**: *How can MCTS be adapted for hide-and-seek games?*

Chapter 7 answers the fourth research question. For handling the imperfect information, two different determinization techniques were investigated, namely single-tree determinization and separate-tree determinization. Single-tree determinization had a slight overhead, but even when taking this into account, it performed significantly better than using separate trees. Furthermore, Location Categorization was proposed, which is a technique that can be used by both the MCTS and the expectimax seekers to give a better prediction for the location of the hider. It significantly increased the playing strength of both the MCTS and the expectimax seekers. The results gave empirical evidence that Location Categorization is a robust technique, as the weights worked for both seeker types against two different types of hider. Because of the asymmetric nature of the hide-and-seek game Scotland Yard, during the playouts, different playout strategies may be used by the different types of players. We found that, for the MCTS hider, it is best to assume during the playouts that the seekers do not know where the hider is, while the MCTS seekers perform best if they do assume where the hider is located. For dealing with the cooperation of the seekers, Coalition Reduction was proposed. This technique reduces the rewarded value for the root player if another player in the coalition wins the game, allowing the seekers to cooperate more effectively in the coalition. We observed that the performance of the MCTS seekers increased by applying Coalition Reduction. Cooperation still appeared to be important, because the performance of the seekers dropped significantly when the reduction became too large. In a direct comparison, MCTS performed considerably better than paranoid search for the hider and expectimax for the seekers. Finally, the experimental results showed that MCTS was able to play Scotland Yard on a higher level than a commercial Nintendo DS program, which is generally considered to be a strong player. In conclusion, with the incorporation of enhancements such as single-tree determinization, Location Categorization, and Coalition Reduction, we were able to let an MCTS-based player play the hide-and-seek game Scotland Yard on a strong level.

Chapter 8 concludes the thesis and provides an outlook on five directions for future research. The answer to the problem statement may be summarized in four points, based on the research questions. First, the max^n search policy performs the best in multi-player MCTS, while the BRS and paranoid policies are still competitive. The max^n search policy can be enhanced with a multi-player variant of the MCTS-Solver. Second, the Progressive History selection strategy significantly increases the performance of two-player and multi-player MCTS. Third, two-ply search-based playouts significantly improve the quality of the playouts and, assuming a sufficient amount of thinking time is provided, increases the performance of MCTS in multi-player domains. Fourth, incorporating single-tree determinization, Location Categorization, and Coalition Reduction into MCTS significantly improves its performance in the multi-player hide-and-seek game Scotland Yard.

The research presented in this thesis indicates five areas of future research. These areas include (1) the application of other search policies, (2) the combination of Progressive History with other selection strategies, (3) further optimization of search-based playouts or the implementation of three-ply search-based playouts, (4) further investigation of Scotland Yard, and (5) the application if the proposed enhancements to other domains.

Samenvatting

Onderzoek in kunstmatige intelligentie is succesvol geweest op het gebied van tweespeler nulsomspelen, zoals schaak, dammen en Go. Dit is bereikt door het ontwikkelen van tweespeler zoektechnieken. Echter, een groot aantal spelen behoort tot de groep waar deze technieken niet zonder meer kunnen worden toegepast. Meerspeler spelen zijn een voorbeeld van zo'n domein. Dit proefschrift richt zich op twee verschillende categorieën meerspeler spelen: (1) deterministische meerspeler spelen met perfecte informatie en (2) meerspeler zoek-en-verstopspelen. In het bijzonder wordt onderzocht hoe Monte-Carlo Tree Search (MCTS) voor spelen in deze twee categorieën kan worden verbeterd. Deze techniek heeft indrukwekkende resultaten behaald in computer Go, maar is ook waardevol gebleken in een reeks andere domeinen. Hoofdstuk 1 biedt een introductie in spelen en de rol die zij vervullen op het gebied van kunstmatige intelligentie, en daarnaast geeft het een overzicht van verschillende speleigenschappen. Ook worden meerspeler spelen gedefinieerd en worden de twee verschillende categorieën meerspeler spelen die onderzocht worden in dit proefschrift besproken. De volgende probleemstelling stuurt het onderzoek.

> **Probleemstelling**: *Hoe kan Monte-Carlo Tree Search verbeterd worden om de prestaties in meerspeler spelen te verhogen?*

Om een antwoord te geven op de probleemstelling, worden vier onderzoeksvragen gedefinieerd. Deze behandelen (1) de integratie van verschillende zoekprincipes in MCTS, (2) het verbeteren van de selectiefase van MCTS, (3) het verbeteren van de simulatiefase van MCTS, en (4) het aanpassen van MCTS voor zoek-en-verstopspelen.

Hoofdstuk 2 introduceert de basisdefinities voor het zoeken in spelbomen. De fundamentele tweespeler minimax-gebaseerde zoektechnieken en enkele verbeteringen hierop worden besproken. Verder beschrijft dit hoofdstuk drie minimax-gebaseerde zoektechnieken voor meerspeler spelen, namelijk max^n, paranoid en Best-Reply Search (BRS). Tenslotte wordt MCTS geïntroduceerd, en wordt er beschreven hoe het kan worden toegepast in meerspeler spelen.

Hoofdstuk 3 beschrijft de testomgeving die gebruikt wordt om de eerste drie onderzoeksvragen te beantwoorden. We beschrijven de regels en de gebruikte domeinkennis voor de vier deterministische perfecte-informatie spelen die ingezet worden in dit proefschrift: Chinese Checkers, Focus, Rolit en Blokus. Vervolgens wordt voor ieder van deze spelen de complexiteit van de toestandsruimte en van de spelboom gegeven. Deze geven een indicatie van hoe moeilijk de spelen zijn voor computers om ze optimaal te spelen. Tenslotte introduceert dit hoofdstuk het programma MAGE, dat gebruikt wordt om de experimenten in hoofdstukken 4, 5 en 6 uit te voeren.

Het voordeel van MCTS is dat het kan worden uitgebreid naar meerspeler spelen. In de standaard variant van MCTS is iedere speler enkel bezig met het maximaliseren van zijn eigen winstratio. Deze variant is daardoor vergelijkbaar met de minimax-gebaseerde meerspeler zoektechniek max^n, waar iedere speler probeert zijn eigen score te maximaliseren, ongeacht de scores van de andere spelers. Andere meerspeler zoekprincipes, zoals die van paranoid en BRS, kunnen ook worden bekeken. Dit heeft geleid tot de eerste onderzoeksvraag.

> **Onderzoeksvraag 1**: *Hoe kunnen meerspeler zoekprincipes worden gebruikt in MCTS?*

Hoofdstuk 4 beantwoordt de eerste onderzoekvraag door de paranoid en BRS zoekprincipes, samen met het standaard max^n principe, te gebruiken in MCTS. Middels deze drie principes worden de selectie- en de terugpropagatiefase van MCTS aangepast. In het MCTS raamwerk lijkt het max^n zoekprincipe het beste te werken. De voordelen van paranoid en BRS in het minimax raamwerk zijn niet van toepassing in MCTS, omdat $\alpha\beta$ snoeiing niet toepasbaar in MCTS is. Een bijkomend probleem van MCTS-BRS is dat in de boom onjuiste posities worden onderzocht, wat de betrouwbaarheid van de simulaties kan verlagen. Echter, MCTS-paranoid en MCTS-BRS behalen over het algemeen redelijke winstpercentages tegen MCTS-max^n, zeker bij lagere tijdsinstellingen. Op basis van de resultaten mogen we concluderen dat het max^n zoekprincipe het meest robuust is, hoewel de BRS en paranoid zoekprincipes nog steeds competitief zijn. Tenslotte verbeterden we het max^n zoekprincipe met de meerspeler variant van de MCTS-Solver, genaamd MP-MCTS-Solver. Deze variant is in staat om speltheoretische waarden in een positie te bewijzen. Een winstpercentage tussen 53% en 55% werd bereikt in het sudden-death spel Focus. We mogen concluderen dat het bewijzen van de speltheoretische waarden de speelsterkte van MCTS in een meerspeler sudden-death domein verbetert.

Een belangrijke fase in het MCTS algoritme is de selectiefase. Tijdens deze fase wordt de boom doorlopen totdat een blad is bereikt. Een selectiestrategie bepaalt hoe de boom wordt doorlopen. Gedurende de afgelopen jaren zijn verschillende selectiestrategieën en verbeteringen ontwikkeld voor verschillende soorten spelen. De populairste selectiestrategie is Upper Confidence bounds applied to Trees (UCT). Er bestaan verscheidene verbeteringen voor de UCT selectiestrategie. De meesten zijn domeinafhankelijk, wat betekent dat ze niet zonder meer in ieder domein toegepast kunnen worden. Een domeinonafhankelijke verbetering is Rapid Action Value Estimation (RAVE). Deze verbetering is gebaseerd op de all-moves-as-first (AMAF) heuristiek. RAVE is behoorlijk succesvol in het gebied van computer Go, maar minder in andere gebieden, zoals het meerspeler spel Chinese Checkers. Dit heeft geleid tot de tweede onderzoeksvraag.

> **Onderzoeksvraag 2**: *Hoe kan de selectiefase van MCTS worden verbeterd in perfecte-informatie meerspeler spelen?*

Hoofdstuk 5 beantwoordt deze vraag door het voorstellen van een nieuwe domeinonafhankelijke selectiestrategie genaamd Progressive History. Deze techniek is een combinatie van de relatieve historie-heuristiek en Progressive Bias. In tegenstelling tot RAVE slaat Progressive History de verzamelde data op in een globale tabel, op een manier die vergelijkbaar is met de simulatiestrategie Move-Average Sampling Technique. Progressive History bleek een significante verbetering in alle spelen met verschillende aantallen spelers. In een vergelijking met UCT behaalde Progressive History de hoogste winstpercentages in de tweespeler varianten, waar het ongeveer 80% van de partijen wint. Bovendien presteert Progressive History ook in de meerspeler varianten beter dan UCT. Progressive AMAF, dat AMAF waarden gebruikt in plaats van history waarden, presteert over het algemeen slechter dan Progressive History. Experimenten in het tweespeler spel Havannah lieten zien dat Progressive History in dit spel beter presteert dan RAVE. Daarnaast bleek ook dat Progressive History de prestatie van de zoekers in het zoek-en-verstopspel Scotland Yard significant verbetert. Op basis van de resultaten mogen we concluderen dat Progressive History MCTS aanzienlijk verbetert in zowel tweespeler als in meerspeler spelen.

Net als de selectiefase is de simulatiefase een belangrijk onderdeel in het MCTS algoritme. Tijdens deze fase wordt de partij voltooid met zetten die worden geselecteerd middels een simulatiestrategie. Realistischere simulaties bieden betrouwbaardere resultaten, wat de speelsterkte van een MCTS-gebaseerde speler verbetert. Simulaties kunnen realistischer gemaakt worden door het toevoegen van domeinkennis. Het nadeel is dat dit het aantal simulaties per seconde kan verkleinen, hetgeen de speelsterkte verlaagt. De uitdaging is om een goede balans te vinden tussen snelheid en kwaliteit van de simulaties. Voor het tweespeler spel Lines of Action (LOA) zijn relatief dure 2-ply $\alpha\beta$ zoekprocessen geïntroduceerd. Hoewel deze het aantal simulaties per seconde significant verkleinden, nam de speelsterkte door het verbeteren van de kwaliteit van de simulaties toe. Dit heeft geleid tot de derde onderzoeksvraag.

Onderzoeksvraag 3: *Hoe kunnen de simulaties van MCTS verbeterd worden in perfecte-informatie meerspeler spelen?*

Hoofdstuk 6 beantwoordt deze vraag door het introduceren van 2-ply zoekprocessen, uitgerust met een heuristieke evaluatiefunctie, die gebruikt worden voor het selecteren van zetten in de simulatiefase van MCTS voor meerspeler spelen. Drie verschillende technieken voor meerspeler spelen zijn onderzocht, namelijk max^n, paranoid, en BRS. Deze simulatiestrategieën werden vergeleken met willekeurig, greedy, en 1-ply simulaties om te bepalen hoe exploratie en snelheid gebalanceerd dienen te worden in de simulaties van meerspeler MCTS. De resultaten lieten zien dat zoekgebaseerde simulaties de kwaliteit van de simulaties significant verhoogden. Onder de verschillende simulatiestrategieën presteerde BRS het beste, gevolgd door paranoid en max^n. Dit voordeel werd afgezwakt door het lagere aantal simulaties per seconde. Vooral BRS en max^n leden hieronder. Onder de geteste 2-ply zoekgebaseerde simulaties presteerde paranoid over het algemeen het beste met zowel korte als lange tijdsinstellingen. Met meer denktijd presteerden de 2-ply zoekgebaseerde simulatiestrategieën relatief beter dan de 1-ply en greedy strategieën. Dit geeft aan dat met langere tijdsinstellingen de computationeel duurdere simulaties gebruikt kunnen worden om de speelsterkte van MCTS-gebaseerde spelers te verhogen. Op basis van de experimentele resultaten mogen we concluderen dat zoekgebaseerde simulaties voor meerspeler spelen voordelig kunnen zijn als de spelers voldoende denktijd krijgen.

De voorgaande hoofdstukken bespraken de toepassing en verbetering van MCTS in deterministische meerspeler spelen met perfecte informatie. In hoofdstuk 7 verschuiven we onze aandacht naar zoek-en-verstopspelen. In dit proefschrift zijn we geïnteresseerd in zoek-en-verstopspelen met de volgende drie eigenschappen. Ten eerste bevatten ze imperfecte informatie voor sommige spelers. Ten tweede spelen sommige spelers samen in een vaste coalitie. Ten derde zijn ze asymmetrisch. De verschillende spelers hebben verschillende soorten doelen. Een spel dat deze eigenschappen heeft is het spel Scotland Yard. In dit meerspeler spel werken vijf zoekers samen om te proberen een verstopper te vangen, die zijn locatie slechts op reguliere tussenpozen bekendmaakt. Dit heeft geleid tot de vierde onderzoeksvraag.

> **Onderzoeksvraag 4**: *Hoe kan MCTS aangepast worden voor een zoek-en-verstopspel?*

Hoofdstuk 7 beantwoordt de vierde onderzoeksvraag. Voor het omgaan met de imperfecte informatie werden twee verschillende determinisatietechnieken onderzocht, namelijk enkele-boom determinisatie en aparte-boom determinisatie. Enkele-boom determinisatie heeft een kleine overhead, maar zelfs als we hier rekening mee houden presteert het significant beter dan het gebruik van aparte bomen. Vervolgens werd Location Categorization voorgesteld, wat een techniek is dat gebruikt kan worden door zowel de MCTS als de expectimax zoekers om een betere voorspelling te maken van de locatie van de verstopper. Het verhoogde de speelsterkte van zowel de MCTS als de expectimax zoekers significant. De resultaten gaven empirisch bewijs dat Location Categorization een robuuste techniek is, omdat de gewichten werkten voor twee verschillende soorten zoekers tegen twee verschillende soorten verstoppers. Vanwege de asymmetrische natuur van Scotland Yard kunnen verschillende simulatiestrategieën voor de verschillende soorten spelers gebruikt worden in de simulaties. We zagen dat het voor de MCTS verstopper beter was als hij tijdens de simulaties aanneemt dat de zoekers niet weten waar hij zit, terwijl de MCTS zoekers het beste presteren als ze aannemen dat ze weten waar de verstopper zit. Om de coalitie tussen de zoekers te behandelen werd Coalition Reduction voorgesteld. Deze techniek verlaagt de toegekende waarde voor de wortelspeler als een andere speler in de coalitie de partij wint, wat de zoekers in staat stelt effeciënter aan de coalitie deel te nemen. We zagen dat de prestatie van de MCTS zoekers verbeterde door het toepassen van Coalition Reduction. Samenwerking bleek nog steeds belangrijk, want de prestatie van de zoekers daalde significant als de reductie te groot werd. In een directe vergelijking bleek MCTS aanzienlijk beter te presteren dan het paranoid zoekproces voor de verstopper en expectimax voor de zoekers. Tenslotte lieten de experimentele resultaten zien dat MCTS in staat was om op een hoger niveau te spelen dan het commercieel Nintendo DS programma, dat over het algemeen wordt gezien als een sterke speler. Dus we mogen concluderen dat we met de toevoeging van verbeteringen als enkele-boom determinisatie, Location Categorization, en Coalition Reduction in staat zijn om een MCTS-gebaseerde speler het zoek-en-verstopspel Scotland Yard op een sterk niveau te laten spelen.

Hoofdstuk 8 sluit het proefschrift af en geeft een vooruitblik op vijf richtingen voor toekomstig onderzoek. Het antwoord op de probleemstelling kan in vier punten worden samengevat. Ten eerste presteert het max^n zoekprincipe het beste in meerspeler MCTS, terwijl BRS en paranoid principes nog steeds competitief kunnen zijn. Het max^n zoekprincipe kan worden verbeterd met een meerspeler variant van MCTS-Solver. Ten tweede verbetert de Progressive History selectiestrategie de prestatie van tweespeler en meerspeler MCTS significant. Ten derde, 2-ply zoekgebaseerde simulaties verbeteren de kwaliteit van de simulaties significant en, aangenomen dat voldoende denktijd beschikbaar is, verbetert de prestatie van MCTS in meerspeler domeinen. Ten vierde, de toevoeging van enkele-boom determinisatie, Location Categorization, en Coalition Reduction aan MCTS verbeteren de prestaties ervan significant in het meerspeler zoek-en-verstopspel Scotland Yard.

Het onderzoek dat gepresenteerd is in dit proefschrift duidt vijf gebieden voor toekomstig onderzoek aan. Deze gebieden bevatten (1) de toepassing van andere zoekprincipes, (2) de combinatie van Progressive History met andere selectiestrategieën, (3) het verder optimaliseren van de implementatie van 3-ply zoekgebaseerde simulaties, (4) het verder onderzoeken van Scotland Yard, en (5) de toepassing van de voorgestelde verbeteringen in andere domeinen.

Curriculum Vitae

Pim Nijssen was born on October 1, 1986 in Maastricht, The Netherlands. From 1998 until 2004, he attended the Euro College in Maastricht, and received the Atheneum diploma. Immediately thereafter, he started a study Knowledge Engineering at the Maastricht University. In 2007, he received his B.Sc. degree. After receiving his bachelor's degree, he started a master in Artificial Intelligence at the same university. In January 2009, he completed this study cum laude. From May 2009 on, Pim was employed as a Ph.D. Student in the Games and AI group at the Department of Knowledge Engineering, Maastricht University. This resulted in several journal and conference publications, and finally this thesis. Besides performing scientific tasks, he was engaged in guiding students during practical sessions of bachelor and master courses such as Introduction to Computer Science, Games & AI, and Knowledge Management. Furthermore, he participated in the organization of the 24th Benelux Conference on Artificial Intelligence (BNAIC) in Maastricht in October 2012.

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*

30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*

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

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 (UT) *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*

Statements

belonging to the thesis

Monte-Carlo Tree Search for Multi-Player Games

by Pim Nijssen

1. Though max^n is generally outperformed by paranoid and BRS in the minimax framework, it is a better choice for a search policy in multi-player MCTS (this thesis, Chapter 4).
2. Progressive History should be considered as an alternative to the all-moves-as-first heuristic (this thesis, Chapter 5).
3. Two-ply searches in the playout phase of multi-player MCTS may increase the playing strength significantly, even at the cost of less playouts (this thesis, Chapter 6).
4. Considering yourself to be more important than your collaborators in a fixed coalition can be quite beneficial for everyone (this thesis, Chapter 7).
5. Coalitions in multi-player games have more in common with psychology and sociology than with Artificial Intelligence.
6. Robots will never be indistinguishable from humans.
7. The pinnacle of Artificial Intelligence is to make intelligent behavior emerge from a simple algorithm.
8. Often, the only reason why computers are better than humans at games with imperfect information is because they have a better memory, not because they have a better strategy.
9. When studying how computers play games, playing computer games may be a welcome diversion.
10. Ph.D. research is like a sudden-death game; it is over before you know it.