Monte-Carlo Tree Search

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. mr. G.P.M.F. Mols,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op donderdag 30 september 2010 om 14:00 uur

door

Guillaume Maurice Jean-Bernard Chaslot

Promotor: Prof. dr. G. Weiss
Copromotor: Dr. M.H.M. Winands
Dr. B. Bouzy (Université Paris Descartes)
Dr. ir. J.W.H.M. Uiterwijk

Leden van de beoordelingscommissie:
Prof. dr. ir. R.L.M. Peeters (voorzitter)
Prof. dr. M. Müller (University of Alberta)
Prof. dr. H.J.M. Peters
Prof. dr. ir. J.A. La Poutré (Universiteit Utrecht)
Prof. dr. ir. J.C. Scholtes

Netherlands Organisation for Scientific Research

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

Dissertation Series No. 2010-41
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-90-8559-099-6

Printed by Optima Grafische Communicatie, Rotterdam.
Cover design and layout finalization by Steven de Jong, Maastricht.

Preface

When I learnt the game of Go, I was intrigued by the contrast between the simplicity of its rules and the difficulty to build a strong Go program. By contrast, most problems with a simple mathematic definition are easier to solve for computers than they are for humans. Fascinated by this characteristic of Go, I was interested in developing a method that would improve the level of Go programs. My first thoughts about using Monte-Carlo were not the most optimistic. However, after experimenting with it, advised by Bruno Bouzy, I quickly changed my mind. I was impressed by the results obtained by Monte-Carlo despite its simplicity. This motivated me to spend four years at Maastricht University to develop Monte-Carlo methods for games and optimization problems.

Jos Uiterwijk, as the leader of the research project “Go for Go”, funded by the Netherlands Organisation for Scientific Research (NWO) is the first one to thank. Later, Gerhard Weiss became supervisor. I thank them for their supervision and help with scientific writing.

My daily advisors, Mark Winands and Bruno Bouzy, deserve my sincerest gratitude, for the many fruitful meetings we had over the years, and for their renewed encouragements and enthusiasm to tackle the difficult scientific problems.

I would like to thank Olivier Teytaud for welcoming me warmly in the MoGo team. His creativity and dedication makes him one of the researchers with whom I enjoyed working the most. Later, I also had the pleasure to work on MoGo together with Arpad Rimmel and Jean-Baptiste Hook. I would also like to thank computer Go competitors with whom I had interesting research discussions, and in particular Erik van der Werf, Rémi Coulom, Sylvain Gelly, and David Fotland.

In Maastricht there were many opportunities for joint work as well. I greatly enjoyed working with my colleagues Jahn-Takeshi Saito, Steven de Jong, Maarten Schadd, Istvan Szita, Marc Ponsen, Sander Bakkes, and Pieter Spronck on research that is reported in the thesis, and elsewhere in joint articles.

Research sometimes needs excessively optimistic people who believe in revolutions to appear in the near future, even when there is no evidence yet that these developments would actually be possible. An example of such a visionary manager is Jaap van den Herik, who promoted research in computer Go, and was one of the rare persons to predict that computers could defeat professionals before the year 2010. I am grateful to him, as well as Peter Michielse in particular and NCF (grant number SH-105-08) in general, for providing use of supercomputer time.

A pleasant research environment is not only facilitated by numerous opportunities for cooperation, but also by a friendly, constructive and comforting atmosphere. The supportive

staff of DKE helped me in many respects; an explicit thank-you is given to Peter Geurtz, Joke Hellemons, and Marijke Verheij. My roommates over the years, Andra Waagmeester, Nyree Lemmens, and Philippe Uyttendaele, made our room a place that I enjoyed being in. My family deserves many thanks for their constant support and communicating their passion for research. Mais surtout, mille tendresses a mon tendre amour.

Guillaume Chaslot, 2010

Table of Contents

Appendices

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we study the use of Monte-Carlo simulations for tree-search problems. Monte-Carlo simulations consist of sequences of randomized actions. They were first applied in the computers of Los Alamos (Metropolis, 1985). Nowadays, Monte-Carlo simulations have applications in numerous fields, as for instance chemistry, biology, economics, and finance (cf. Liu, 2002). Tree-search problems are problems that can be handled by progressively expanding a tree structure, for instance in pathfinding and chess.

The Monte-Carlo technique we investigate is Monte-Carlo Tree Search (MCTS). In 2006, it appeared in three different variants (Coulom, 2006; Kocsis and Szepesvári, 2006; Chaslot *et al.*, 2006a). Coulom used his variant to create the first competitive MCTS program, CRAZY STONE. This program immediately won the 9×9 Go tournament at the 2006 Computer Olympiad. The variant introduced by Kocsis and Szepesvári (2006), called UCT, was based on the Upper Confidence Bounds (UCB) algorithm (Auer, Cesa-Bianchi, and Fischer, 2002). Chaslot *et al.* (2006a) proposed the variant Objective Monte-Carlo (OMC), which was evaluated for Go positions.

In contrast to classic algorithms for tree search, such as $A^\star$ and $\alpha\beta$ search, MCTS does not rely on a *positional evaluation function*, but on Monte-Carlo simulations. MCTS is a general algorithm and can be applied to many problems. The most promising results so far have been obtained in the game of Go, in which it outperformed all classic techniques. Hence, we use Go as our main test bed.

In this chapter we provide a description of the search problems that we aim to address (Section 1.1) and the classic search techniques which are used so far to solve them (Section 1.2). Subsequently, we formulate the problem statement together with five research questions (Section 1.3). Next, we justify the choice of Go as test domain in Section 1.4. Finally, Section 1.5 provides a thesis overview.

1.1 Tree-Search Problems

A tree-search problem is a problem in which the states can be represented as nodes of a tree, and the actions can be represented by edges between the nodes. A tree is defined as an acyclic connected graph where each node has a set of zero or more child nodes, and at most one parent node (cf. Russell and Norvig, 1995).

Table 1.1: Different types of problems.

	One player	Two players	Multi players
Deterministic	TSP, PMP	Go, chess	Chinese Checkers
Stochastic	Sailing Problem	Backgammon	Simplified Catan

We distinguish three categories of problems: (1) problems without opponents (called optimization problems, one-player games, or puzzles), (2) problems with one opponent (two-player games), and (3) problems with multiple opponents (multi-player games). In two-player games, players may oppose or cooperate with each other. In multi-player games, players may create coalitions. Moreover, we distinguish between deterministic and stochastic problems. A schematic overview with examples of these problems is given in Table 1.1. Most of the experiments of this thesis are performed in Go, which is a non-cooperative two-player deterministic game.

We chose to use the terminology that is commonly used for this domain in the remainder of the thesis. Hence, we use *move* as a synonym for action, *position* as a synonym for state, and *game* as a synonym for problem.

1.2 Tree-Search Algorithms

A *search algorithm* takes a problem (i.e., a game) as input and returns a solution in the form of an action sequence (i.e., a move sequence) (cf. Russell and Norvig, 1995). Many tree-search algorithms were developed in the last century. For tree-search based optimization problems, the $A^\star$ algorithm (Hart, Nielson, and Raphael, 1968) is one of the standard algorithms. For two-player games, the foundation of most algorithms is minimax (von Neumann 1928). Minimax has been improved into a more powerful algorithm: $\alpha\beta$ (Knuth and Moore, 1975). There are many variants of the $\alpha\beta$ algorithm. Amongst them, one of the most successful is the iterative deepening principal variation search (PVS) (Marsland, 1983), which is nearly identical to nega-scout (Reinefeld, 1983). They form the basis of the best programs in many two-player games, such as chess (Marsland and Björnsson, 2001). Other $\alpha\beta$ variants are MTD(f) (Plaat, 1996) and Realization-Probability Search (Tsuruoka, Yokoyama, and Chikayama, 2002).

Several other algorithms exist for tree-search, as for instance Proof-Number (PN) search and its variants (Allis, Van der Meulen, and Van den Herik, 1994; Van den Herik and Winands, 2008), $B^\star$ (Berliner, 1979), $SSS^\star$ (Stockman, 1979), and λ-search (Thomsen, 2000).

Most of these algorithms rely on a *positional evaluation function*. This function computes a heuristic value for a board position at leaf nodes. The resulting value can be interpreted in three different ways (Donkers, 2003): (1) as a *prediction* of the game-theoretic value, (2) as an estimate of the *probability* to win, or (3) as a measure of the *profitability* of the position.

One family of algorithms presented above does not require a positional evaluation function: PN search and its variants. This family of algorithms has been used to solve games, for instance Qubic (Allis *et al.*, 1994), Checkers (Schaeffer *et al.*, 2007), and Fanorona

(Schadd *et al.*, 2008b). However, PN is not useful for real-time game play because most of the time it is not able to prove the game-theoretic value of the position.

1.3 Problem Statement and Research Questions

In the previous sections, we discussed the scientific context of our investigation. This section introduces the problem statement which guides our research.

> **Problem statement**: *How can we enhance Monte-Carlo Tree Search in such a way that programs improve their performance in a given domain?*

For decades, $\alpha\beta$ has been the standard for tree search in two-player games. The $\alpha\beta$ algorithm requires a quite good evaluation function in order to give satisfactory results. However, no such evaluation function is available for Go. As a consequence, the best Go programs in 2005 were a heterogeneous combination of $\alpha\beta$ search, expert systems, heuristics, and patterns. The methodology used was quite domain-dependent. An alternative, which emerged around that time, was to use Monte-Carlo simulations as an alternative for a positional evaluation function (Brügmann, 1993; Bouzy and Helmstetter, 2003). Soon it was developed into a complete Monte-Carlo technique, called Monte-Carlo Tree Search (MCTS). It is a best-first search method guided by the results of Monte-Carlo simulations. MCTS appeared in three different variants in 2006 (Coulom, 2006; Kocsis and Szepesvári, 2006; Chaslot *et al.*, 2006a). Coulom used his variant to create the first competitive MCTS program, CRAZY STONE, which immediately won the 9×9 Go tournament at the 11^{th} Computer Olympiad. The variant introduced by Kocsis and Szepesvári (2006), called UCT, was based on the Upper Confidence Bounds (UCB) algorithm (Auer *et al.*, 2002). The variant of Chaslot *et al.* (2006a) was based on their Objective Monte-Carlo (OMC) algorithm. In this thesis we are going to investigate how to enhance MCTS.

Our problem statement raises five research questions. They deal with Monte-Carlo simulations, the balance between exploration and exploitation, parameter optimization, parallelization, and opening-book generation.

> **Research question 1**: *How can we use knowledge to improve the Monte-Carlo simulations in MCTS?*

The most basic Monte-Carlo simulations consist of playing random moves. *Knowledge* transforms the plain random simulations into more sophisticated *pseudo-random* simulations (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008). The knowledge can be designed by a human expert or created (semi)-automatically by machine learning. We consider two different simulation strategies that apply knowledge: *urgency-based* and *sequence-like* simulation. Methods are investigated to learn automatically the knowledge of the simulation strategy.

> **Research question 2**: *How can we use knowledge to arrive at a proper balance between exploration and exploitation in the selection step of MCTS?*

In each node of the MCTS tree, a balance between exploration and exploitation has to be found. On the one hand, the search should favour the most promising move (exploitation).

On the other hand, less promising moves should still be investigated sufficiently (exploration), because their low scores might be due to unlucky simulations. This move-selection task can be facilitated by applying knowledge. We introduce the concept of "progressive strategy" that causes the knowledge to be dominant when the number of simulations is small in a node, and causes the knowledge to lose progressively influence when the number of simulations increases. The two progressive strategies we propose are *progressive bias* and *progressive widening*. Progressive bias directs the search according to knowledge. Progressive widening first reduces the branching factor, and then increases it gradually. This scheme is also dependent on knowledge.

> **Research question 3**: *How can we optimize the parameters of an MCTS program?*

MCTS is controlled by several parameters, which define the behaviour of the search. These parameters have to be adjusted in order to get the best performance out of an MCTS program. We propose to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM) (Rubinstein, 1999). The method is related to Estimation-of-Distribution Algorithms (EDAs) (Muehlenbein, 1997), a new area of evolutionary computation.

> **Research question 4**: *How can we parallelize MCTS?*

In the past, research in parallelizing search has been mainly performed in the area of $\alpha\beta$-based programs running on super-computers. DEEP BLUE (Campbell, Hoane, and Hsu, 2002) and BRUTUS/HYDRA (Donninger, Kure, and Lorenz, 2004) are famous examples of highly parallelized chess programs. The recent evolution of hardware has gone into the direction that nowadays personal computers contain several cores. To get the most out of the available hardware one has to parallelize MCTS as well. In order to formulate an answer to the fourth research question, we will investigate three parallelization methods for MCTS: leaf parallelization, root parallelization, and tree parallelization.

> **Research question 5**: *How can we automatically generate opening books by using MCTS?*

Modern game-playing programs use opening books in the beginning of the game to save time and to play stronger. Generating opening books in combination with an $\alpha\beta$ program has been well studied in the past (Buro, 1999; Lincke, 2001; Karapetyan and Lorentz, 2006). A challenge is to generate automatically an opening book for MCTS programs. We propose a method, called Meta Monte-Carlo Tree Search (Meta-MCTS), that combines two levels of MCTS. Instead of using a relatively simple simulation strategy, it uses an entire MCTS program to play a simulated game.

1.4 Choice of the Test Domain

As test domain to answer the research questions, we have chosen the game of Go. This game has been recognized as a challenge for Artificial Intelligence (Bouzy and Cazenave, 2001; Müller, 2002). We give below four reasons for choosing Go as a test domain.

- **Simple implementation.** The rules of Go are simple (see Chapter 2).
- **Well-known.** Go is played by 25 to 50 millions of people over the world, and counts around one thousand professional players.
- **Difficult to master.** No Go program has reached a level close to the best humans on the 19×19 board.
- **Well-established research field.** Several Ph.D. theses and hundreds of peer-reviewed publications are dedicated to computer Go.

1.5 Thesis Overview

The contents of this thesis are as follows. Chapter 1 contains an introduction, a classification of tree-search problems, a brief overview of tree-search algorithms, the problem statement and five research questions, the choice of the test domain, and an overview of the thesis.

Chapter 2 introduces the test environment. It explains the game of Go, which will be used as the test domain in this thesis. We provide the history of Go, the rules of the game, a variety of game characteristics, basic concepts used by humans to understand the game of Go, and a review of the role of Go in the AI domain. The Go programs MANGO and MOGO, used as test vehicles for the experiments in the thesis, are briefly described.

In Chapter 3, we start with discussing earlier research about using Monte-Carlo evaluations as an alternative for a positional evaluation function. This approach is hardly used anymore, but it established an important step towards Monte-Carlo Tree Search (MCTS). Next, we describe the structure of MCTS. MCTS consists of four main steps: selection, expansion, simulation, and backpropagation. The chapter presents different strategies for each MCTS step. Finally, we give the application of MCTS for other domains than Go.

Chapter 4 answers the first research question. We explain two different simulation strategies that apply knowledge: *urgency-based* and *sequence-like* simulations and give experimental results. Moreover, methods are investigated for learning automatically the simulation strategy and the associated experiments are presented. Related research in the domain of general game playing is discussed in the remainder of the chapter.

Chapter 5 answers the second research question by proposing two methods that integrate knowledge into the selection part of MCTS: *progressive bias* and *progressive widening*. Progressive bias uses knowledge to direct the search. Progressive widening first reduces the branching factor, and then increases it gradually. We refer to them as "progressive strategies" because the knowledge is dominant when the number of simulations is small in a node, but loses influence progressively when the number of simulations increases. We give details on the implementation and the experimental results of these progressive strategies in MANGO. Subsequently, the performance of the progressive strategies for the Go programs CRAZY STONE and MOGO, and in the LOA program MC-LOA is presented. Finally, more recent related research on enhancing the selection strategy is described.

Chapter 6 answers the third research question by proposing to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM). CEM is related to Estimation-of-Distribution Algorithms (EDAs), which constitute a new area of evolutionary computation. The fitness function for CEM measures the winning rate

for a batch of games. The performance of CEM with a fixed and variable batch size is tested by optimizing the search parameters in the MCTS program MANGO.

Chapter 7 answers the fourth research question by investigating three methods for parallelizing MCTS: leaf parallelization, root parallelization and tree parallelization. We compare them by using the *Games-Per-Second (GPS)-speedup measure* and *strength-speedup measure*. The first measure corresponds to the improvement in speed, and the second measure corresponds to the improvement in playing strength. The three parallelization methods are implemented and tested in MANGO, running on a 16-core processor node.

Chapter 8 answers the fifth research question by combining two levels of MCTS. The method is called Meta Monte-Carlo Tree Search (Meta-MCTS). Instead of using a relatively simple simulation strategy, it uses an entire MCTS program (MOGO) to play a simulated game. We present two Meta-MCTS algorithms: the first one, Quasi Best-First (QBF), favours exploitation; the second one, Beta-Distribution Sampling (BDS), favours exploration. In order to evaluate the performance of both algorithms, we test the generated 9×9 Go opening books against computer programs and humans.

The research conclusions and recommendations for future investigations are given in Chapter 9.

Appendix A discusses Production Management Problems as an auxiliary test domain. Appendix B presents the historic results of the best programs against professional Go players.

Chapter 2

Test Environment: The Game of Go

The chapter describes the test environment used to answer the problem statement and the five research questions formulated in Chapter 1. A test environment consists of a problem (also called a *game*) and one or more programs. The game under consideration is Go. We use the following two Go programs: MANGO and MOGO.

The chapter is structured in the following way. Section 2.1 provides the history of Go. Next, Section 2.2 presents the rules. Subsequently, Section 2.3 presents the characteristics of the game of Go. Section 2.4 discusses basic concepts used by humans to understand the game of Go. Then, in Section 2.5, a short review of the role of Go in the AI domain is given. Finally, Section 2.6 introduces our two Go programs, MANGO and MOGO, used as test vehicles for the experiments in the thesis.

2.1 History of Go

Go is one of the oldest games in the world, originating from ancient China. According to the legend, the Chinese emperor Yao (2337-2258 BCE) asked his counsellor to design a game to teach his son discipline, concentration, and balance (Lasker, 1934; Masayoshi, 2005). The earliest written reference of the game is given in the historical annals of Tso Chuan (4^{th} century BCE), referring to a historical event in 548 BCE (Watson, 1989). Go was originally played on a 17×17 line grid, but a 19×19 grid became standard by the Tang Dynasty (618-907 CE). It was introduced in Japan and Korea between the 5^{th} and 7^{th} century. In Japan, the game became popular at the imperial court in the 8^{th} century. Among the common people it gained popularity in the 13^{th} century. In the 17^{th} century, several Go schools were founded with the support of the Japanese government. This official recognition led to a major improvement of the playing level.

Despite its widespread popularity in East Asia, Go did only spread slowly to the rest of the world, unlike other games of Asian origin, such as chess. Although the game is mentioned in Western literature from the 16^{th} century onward, Go did not become popular in the West until the end of the 19^{th} century, when the German scientist Korschelt (1880)

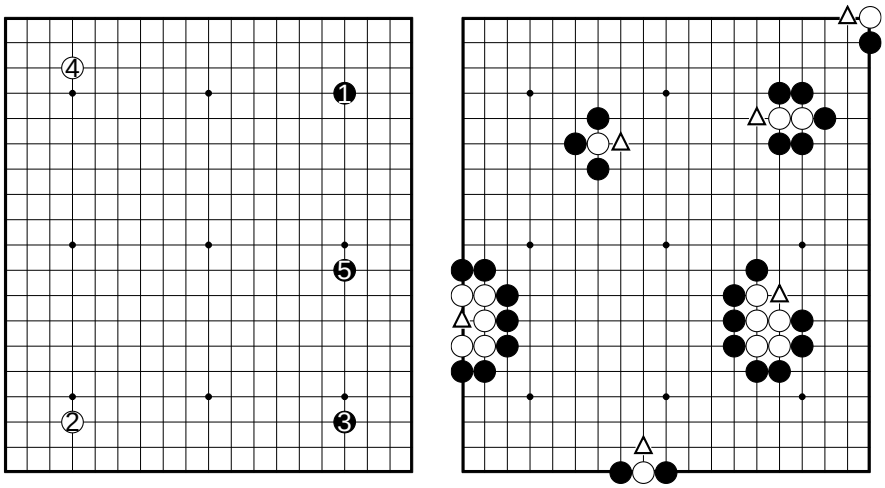
wrote a treatise on the game. By the early 20^{th} century, Go had spread throughout the German and Austro-Hungarian empires. It was only in the second part of the 20^{th} century that Go became recognized in the rest of the Western World.

Nowadays, the International Go Federation has at least 71 member countries. It has been claimed that across the world 1 person out of every 222 plays Go (Fairbairn, 2000). It is estimated that there are around 1,000 Go professionals in the world, mostly from Japan, China, and Korea.

2.2 Rules of Go

Several rule sets exist for the game of Go (e.g., Chinese, Japanese, American, or New Zealand). Although all major rule sets agree on the same general idea of how the game is to be played, there exist several subtle differences. The primary difference between rule sets is the scoring method, although it is rare to observe more than a one-point difference between scoring methods. The two-most popular rule sets are the Japanese and the Chinese sets. Due to the scoring method, the Japanese rules are regarded by some to be slightly more interesting than Chinese rules. However, it is well known that Japanese rules are quite difficult (and by some even considered impossible) to implement in a program due to ambiguities and inconsistencies in the official texts. Chinese rules also suffer from some ambiguity, but to a much lesser extent. Therefore, it is the natural choice for Go programmers to prefer Chinese rules (Van der Werf and Winands, 2009) and use them at computer Go tournaments (e.g., Computer Olympiad). It is beyond the scope of the thesis to explain all rules in detail. For a more elaborate introduction we refer to Van der Werf (2004). A basic set of rules, adapted from Davies (1977), is given below.

1. The square grid board is empty at the outset of the game. Usually the grid contains 19×19 intersections, though 9×9 is used as well.
2. There are two players, called Black and White.
3. Black makes the first move, alternating with White (see Figure 2.1(a)).
4. A move consists of placing one stone of one's own colour on an empty intersection on the board.
5. A player may pass his turn at any time.
6. A stone or through grid lines orthogonally connected set of stones of one colour is captured and removed from the board when all the intersections directly adjacent to it are occupied by the opponent (see Figure 2.1(b)).
7. No stone may be played to repeat a former board position.
8. Two consecutive passes end the game.
9. A player's territory consists of all the board points he has either occupied or surrounded.
10. The player with more territory wins.



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

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

Figure 2.1: Rules of Go.

2.3 Characteristics of Go

Formally, Go can be defined as a *turn-based*, *two-person*, *zero-sum*, *deterministic*, *partisan* game with *perfect information*.[1] Lichtenstein and Sipser (1980) proved that Go is PSPACE-hard. The state-space complexity is estimated to be 10^{171} (Tromp and Farnebäck, 2007) and the game-tree complexity 10^{360} (Allis, 1994).[2] The average branching factor of Go is much higher than in chess: 250 against 35 (Allis, 1994). Due to these characteristics, Go is considered by many experts as one of the most complex board games (Müller, 2002). Considering the current state-of-the-art computer techniques, it appears that Go 19×19 or even 9×9 is not solvable in reasonable time by brute-force methods. Up to now, the largest *square* board for which a computer proof has been published is 5×5 by Van der Werf, Van den Herik, and Uiterwijk (2003). It is a full-board win for the first player (Black). Recently, Van der Werf and Winands (2009) solved *rectangular* Go boards up to 30 intersections. An overview of the game-theoretic results is given in Table 2.1. For Go, the game-theoretic results provide the number of points by which a game is won (or lost).

[1] A *turn-based* or *sequential* game is a game where players move following a predefined order.
A game is *zero-sum* when the gain or loss of one player is exactly balanced by the gains or losses of the other player(s). For two-player games, it implies that if one player wins, the other loses.
A game is *deterministic* if the outcome of every move is deterministic, i.e., it does not involve chance.
A game is *partisan* when the possible moves are not the same for all players. We remark that in Go, possible moves are often the same for both players, but may differ in rare situations.
A game has perfect information when all the players share the same piece of information.

[2] The state-space complexity of a game is defined as the number of legal game positions reachable from the initial position of the game (Allis, 1994).
The game-tree complexity of a game is the number of leaf nodes in the solution search tree of the initial position(s)

Table 2.1: First player scores on $m \times n$ boards (Van der Werf and Winands, 2009).

$m\backslash n$	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	3	4	0	1	2	3	0	1	2	2
2		0	0	8	10	12	14	16	18	4	6	
3			9	4	15	18	21	24	5			
4				2	20	8	28					
5					25	4						

2.4 Go Concepts

Humans are using several concepts to understand the game of Go. It is interesting to mention these concepts to understand how humans deal with the complexity of the game. We explain below the concepts of *life and death* (Subsection 2.4.1), *territory* (Subsection 2.4.2), *influence* (Subsection 2.4.3), and *patterns* (Subsection 2.4.4). We also discuss how computer programs dealt with these concepts in the past.

2.4.1 Life and Death

In the thesis, an orthogonally connected set of stones is called a *block*. A *group* is defined as a (loosely) connected set of blocks of one colour that usually controls[3] one connected area at the end of the game (Van der Werf, 2004). By definition, every block is also a group. A group is said to be *alive* if it cannot be captured by the opponent. A group is said to be *unconditionally alive* if it cannot be captured by the opponent even if the defending player always passes. A group that cannot escape of being captured is considered *dead*. An example of life and death is given in Figure 2.2. It is relatively straightforward to make a procedure that tests if a group is unconditionally alive or not. However, during actual games, groups are rarely unconditionally alive, but only alive if defended. Classic search algorithms such as $\alpha\beta$ (Knuth and Moore, 1975) or proof-number search (Allis *et al.*, 1994) may help, in some cases, to find out whether a given group is alive. Predicting life and death in Go has been the aim of specific research (see Wolf, 1994; Kishimoto and Müller, 2003; Van der Werf *et al.*, 2005). The resulting programs are able to outperform strong amateurs for specific life-and-death problems.

2.4.2 Territory

Territory is defined as the intersections surrounded and controlled by one player at the end of the game. An example is given in Figure 2.3. The notion of territory is dependent on the life and death of groups forming that territory. Therefore, computing territory is more difficult than computing life and death. Several approaches are reasonably effective at evaluating territory. For example, Müller (1997) used a combination of search and static rules, whereas Bouzy (2003) chose to apply mathematical morphology. Van der Werf, Van den Herik, and Uiterwijk (2006) trained a multi-layered perceptron to predict potential territory.

of the game (Allis, 1994).

[3] To control is defined as to get all points for that area at the end of the game.

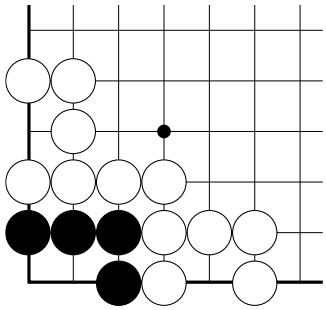


Figure 2.2: Life and death: The white group can never be captured; therefore it is said to be (unconditionally) alive. Black cannot prevent that his group is being captured; therefore Black's group is said to be dead.

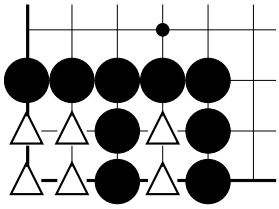


Figure 2.3: Territory: Black has territory that is marked with triangles.

2.4.3 Influence

A player has *influence* in a region if it is most likely that he is able to create territory in this region. This notion is more difficult to evaluate accurately with a computer than territory, because the amount of influence depends on the specific board situation. In Go, there is often a trade-off between influence and territory. When a player tries to create a territory, the opponent may force him to close it by playing on the outside of that territory. This often creates influence for that opponent. An example of such a trade-off can be seen in Figure 2.4. Several researchers have proposed a model for influence (e.g., Zobrist, 1969; Chen, 2002).

2.4.4 Patterns

A *pattern* is a (local) configuration of stones, which may or may not be dependent of its location on the board. Patterns are important both for humans and computers. Humans typically try to maximize the number of efficient patterns, and minimize the number of inefficient patterns. Examples of efficient and inefficient patterns are depicted in Figure 2.5. There has been quite an effort to include patterns in a Go engine, as for instance by Zobrist (1970), Cazenave (2001), and Ralaivola, Wu, and Baldi (2005).

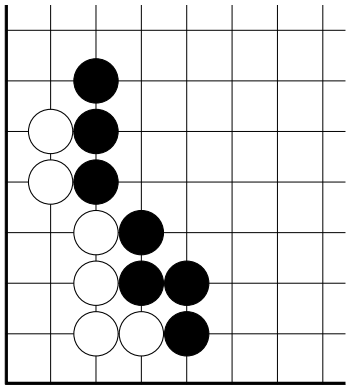


Figure 2.4: Influence and territory: White made a territory in the corner, whereas Black has *influence* on the outside.

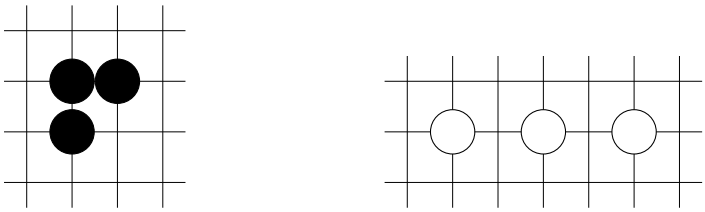


Figure 2.5: Two Go patterns. Left: a pattern of three stones that is considered as inefficient by human experts. Right: A pattern of three stones considered as efficient by human experts.

2.5 Go and Artificial Intelligence

Since the beginning of AI, mind games have been studied as relevant application fields. For instance, some of the first algorithms able to search and learn have been developed for chess (Shannon, 1950; Turing, 1953) and Checkers (Samuel, 1959), respectively.

The first scientific article describing a computer Go program was published by Remus (1962), who developed an automatic algorithm for learning Go. The program was implemented on an IBM 704, but was not able to defeat any human player. The first program to be able to defeat an absolute beginner was designed by Zobrist, who also wrote the first Ph.D. thesis on computer Go (Zobrist, 1970). Just one year later, the second Ph.D. on Go was defended by Ryder (1971). These researchers created the first Go programs, which were based on heuristic evaluation functions. The main part was an influence feature: each stone was considered to radiate influence on its surrounding intersections. These programs were not able to perform deep searches, due to the lack of computer power.

During the seventies, the best program, INTERIM.2, was built by Reitman and Wilcox (cf. Wilcox, 1988). This program used an expert system which took decisions based on an

abstract representation of the game. It was programmed in LISP. Further use of abstraction was also studied by Friedenbach (1980). The combination of search, heuristics, and expert systems led to the best programs in the eighties.

At the end of the eighties a new type of Go programs emerged. These programs made an intensive use of pattern recognition. This approach was discussed in detail by Boon (1990).

In the following years, different AI techniques, such as Reinforcement Learning (Schraudolph, Dayan, and Sejnowski, 1993), Monte Carlo (Brügmann, 1993), and Neural Networks (Richards, Moriarty, and Miikkulainen, 1998), were tested in Go. However, programs applying these techniques were not able to surpass the level of the best programs. The combination of search, heuristics, expert systems, and pattern recognition remained the winning methodology.

Brügmann (1993) proposed to use Monte-Carlo evaluations as an alternative technique for Computer Go. His idea did not got many followers in the 1990s. In the following decade, Bouzy and Helmstetter (2003) and Bouzy (2006) combined Monte-Carlo evaluations and search in Indigo. The program won three bronze medals at the Olympiads of 2004, 2005, and 2006. Their pioneering research inspired the development of Monte-Carlo Tree Search (MCTS) (Coulom, 2006; Kocsis and Szepesvári, 2006; Chaslot *et al.*, 2006a). Since 2007, MCTS programs are dominating the Computer Go field. MCTS will be explained in the next chapter.

2.6 Go Programs MANGO and MOGO

In this subsection, we briefly describe the Go programs MANGO and MOGO that we use for the experiments in the thesis. Their performance in various tournaments is discussed as well.[4]

MANGO

We developed MANGO at Maastricht University since 2006. It is programmed in C++, and was most of the time running on a 2.6 GHz quad-core computer. MANGO participated in 10 international computer tournaments in 2006 and 2007, including the Computer Olympiad 2007. MANGO finished in the top half of all tournaments that it participated in.

MOGO

MOGO was developed by the University Paris-Orsay as the master project of Yzao Wang, supervised by Rémi Munos with advice from Rémi Coulom. A few months later, Sylvain Gelly and Olivier Teytaud joined the project. We joined the MOGO team in beginning of 2008. MOGO is also programmed in C++.

MOGO participated in more than 30 internet tournaments (of which more than half of them were won), and three Computer Olympiads: Amsterdam 2007, Beijing 2008, and Pamplona 2009 (in which it finished first, second and third, respectively). The program's

[4]The overview of the online computer tournaments in which MANGO and MOGO participated can be found at: http://www.weddslist.com/kgs/past/index.html. The Go results of the Computer Olympiads in which MANGO and MOGO participated can be found at: http://www.grappa.univ-lille3.fr/icga/game.php?id=12.

most famous results were in achieving several milestones in defeating human Go professionals in official matches, in particular with 9, 7, and 6 stones handicap. The history of the best performance of programs against professional players can be found in Appendix B. More information on the tournament results of MOGO can be found on its webpage.[5]

[5] See http://www.lri.fr/ teytaud/mogo.html.

Chapter 3

Monte-Carlo Tree Search

This chapter is based on the following publications:

> G.M.J-B. Chaslot, J-T., Saito, B. Bouzy, J.W.H.M. Uiterwijk and H.J. van den Herik (2006a). 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.
>
> G.M.J-B. Chaslot, S. de Jong, J-T. Saito and J.W.H.M. Uiterwijk (2006b). 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.
>
> G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy (2008c). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357.
>
> G.M.J-B. Chaslot, S. Bakkes, I. Szita and P.H.M. Spronck (2008d). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference* (eds. M. Mateas and C. Darken), pp. 216–217.
>
> I. Szita, G.M.J-B. Chaslot and P.H.M. Spronck (2010). Monte-Carlo Tree Search in Settlers of Catan. *Advances in Computer Games Conference (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *Lecture Notes in Computer Science (LNCS)*, pp. 21–32, Springer-Verlag, Heidelberg, Germany.

In this chapter we discuss Monte-Carlo Tree Search (MCTS), which is used to answer our problem statement and research questions. MCTS appeared in three different variants in 2006 (Coulom, 2006; Kocsis and Szepesvári, 2006; Chaslot *et al.*, 2006a). Coulom used his variant to create the first competitive MCTS program, CRAZY STONE, which immediately won the 9×9 Go tournament at the 11^{th} Computer Olympiad. The variant introduced by Kocsis and Szepesvári (2006), called UCT, was based on the Upper Confidence Bounds

(UCB) algorithm (Auer *et al.*, 2002). The variant we proposed was called Objective Monte-Carlo (OMC) (Chaslot *et al.*, 2006a). This chapter introduces a general framework for MCTS that is able to incorporate these variants.

MCTS is a best-first search method guided by Monte-Carlo simulations. In contrast to classic tree-search algorithms such as $\alpha\beta$ (Knuth and Moore, 1975) and $A^\star$ (Hart *et al.*, 1968), MCTS does not require any heuristic positional evaluation function. MCTS is particularly interesting for domains where building a positional evaluation function is a difficult time-consuming issue, such as the game of Go.

MCTS consists of two strongly coupled parts: a relatively shallow *tree structure* and deep *simulated games*. The tree structure determines the first moves of the simulated games. The results of these simulated games shape the tree. MCTS uses four main steps. (1) In the *selection step* the tree is traversed from the root node until the end of the tree. (2) Next, in the *expansion step* a node is added to the tree. (3) Subsequently, during the *simulation step* moves are played in self-play until the end of the game is reached. (4) Finally, in the *backpropagation step*, the result of a simulated game is propagated backwards, through the previously traversed nodes.

This chapter is organized as follows. In Section 3.1, we discuss earlier research about using Monte-Carlo evaluations as an alternative for a heuristic positional evaluation function. This approach is hardly used anymore, but it established an important step towards MCTS. Section 3.2 presents the structure of MCTS. In Section 3.3 we present different strategies proposed for each MCTS step. Subsequently, we discuss how to select the move to be played in the actual game in Section 3.4. Then, we give applications of MCTS to different domains in Section 3.5. Finally, we present the chapter conclusions in Section 3.6.

3.1 Related Work: Monte-Carlo Evaluations

Monte-Carlo Evaluations (MCEs) were originally introduced for Othello, Tic-Tac-Toe, and Chess by Abramson (1990). Brügmann (1993) was the first to use them for Go as well, but his results were not convincing. During the nineties, MCEs were used in several stochastic games such as Backgammon (Tesauro and Galperin, 1997), Bridge (Smith, Nau, and Throop, 1998; Ginsberg, 1999), Poker (Billings *et al.*, 1999), and Scrabble (Sheppard, 2002). After 2000, a further development by Bouzy and Helmstetter (2003) led to the first competitive Monte-Carlo Go program, called INDIGO, at the Olympiads of 2003 and 2004 (cf. Chen, 2003; Fotland, 2004). Moreover, Helmstetter (2007) successfully applied MCEs in the one-player game Morpion Solitaire.

The most basic version of MCEs works in the following way: they evaluate a game position P by performing *simulations* from P. In a simulation (also called a *playout* or a *rollout*) moves are (pseudo-)randomly selected in self-play until the end of the game is reached.[1] Each simulation i gives as output a payoff vector R_i containing the payoffs for each player. The evaluation $E_n(P)$ of the position P after n simulations is the average of the results, i.e., $E_n(P) = \frac{1}{n}\sum R_i$.

A property of MCEs is that, if the values of the payoffs are bounded, $E_n(P)$ converges to a fixed value when n goes to ∞. We denote the limit of $E_n(P)$ when n approaches

[1] This requires the condition that the number of moves per game is limited. For instance, in the game of Go, this is done by introducing an extra rule which forbids to play in its own eyes.

infinity by $E_\infty(P)$, and σ to be the standard deviation of the payoff. Moreover, the Central Limit Theorem states that the random variable $E_n(P)$ converges to a normal distribution with mean value $E_\infty(P)$ and standard deviation $\frac{\sigma}{\sqrt{n}}$. For sufficiently large n, we have the following approximation:

$$E_n(P) = E_\infty(P) + X \tag{3.1}$$

where X is a normally distributed random variable with mean value 0 and standard deviation $\frac{\sigma}{\sqrt{n}}$.

The use of MCEs raises two questions. The first question concerns the quality of MCEs. Is the evaluation $E_\infty(P)$ qualitatively comparable to a positional evaluation? (Given sufficient resources, would an $\alpha\beta$ program based on MCEs perform as well as a program based on a positional evaluation function?) The second question concerns the practical and quantitative feasibility of MCEs. If $E_\infty(P)$ would theoretically provide a satisfactory evaluation, then, how many games would be required for a useful evaluation? (Would the required number of simulations be reachable in practice?)

To answer the first question, we observe that MCEs have been shown to be quite accurate for several games such as Backgammon (Tesauro and Galperin, 1997), Go (Bouzy and Helmstetter, 2003), and Othello (Abramson, 1990). To answer the second question, we observe that the number of simulations that have to be performed in order to have a meaningful evaluation appears to be in the order of a few thousands (cf. Bouzy and Helmstetter, 2003). The requirement of such a relatively large number of simulated games makes MCEs impractical in a classic $\alpha\beta$ tree search for most domains. For instance, to evaluate 1,000,000 nodes in chess, which takes approximately one second in a typical chess program, a runtime of approximately 28 hours would be required with MCE, assuming 100,000 simulations per second and 10,000 simulations per evaluation. Under tournament conditions, an MCE-based program would evaluate even a fewer number of nodes, and would have therefore a shallower search depth.

Despite the time-consuming property, there are programs that combine MCEs and $\alpha\beta$ search.[2] To reach a reasonable search depth, the programs use *lazy evaluations* (Persson, 2006): instead of evaluating the position completely, the MCE is stopped prematurely when there is a certain probability (say 95%) that the value is always lower than the α value, or always higher than the β value, respectively (Persson, 2006).

Another way of combining search and MCEs was proposed by Bouzy (2006). He suggested to grow a search tree by iterative deepening and pruning unpromising nodes while keeping only promising nodes. All leaf nodes are evaluated by MCEs. A problem with this approach is that actually good branches are pruned entirely because of the variance underlying MCEs.

By contrast, the Monte-Carlo Tree Search approach does not use MCEs as described above, but rather improves the quality of MCEs by focusing the search in the most promising regions of the search space. This mechanism is explained in detail in the next section.

[2] VIKING5, the Go program developed by Magnus Persson, was most probably the strongest in using this method.

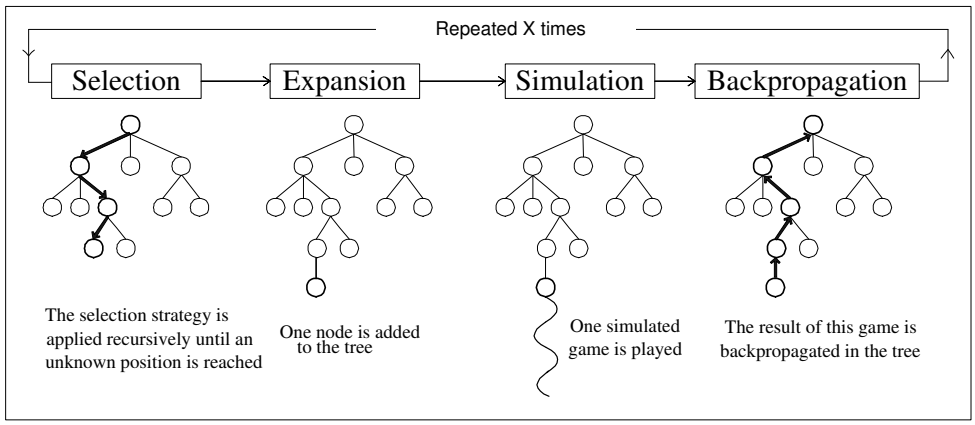


Figure 3.1: Outline of Monte-Carlo Tree Search.

3.2 Structure of Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) (Coulom, 2006; Kocsis and Szepesvári, 2006; Chaslot *et al.*, 2006a) is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space. Using the results of previous explorations, the algorithm gradually builds up a game tree in memory, and successively becomes better at accurately estimating the values of the most-promising moves. MCTS is applicable if at least the following three conditions are satisfied: (1) the payoffs are bounded (the game scores are bounded), (2) the rules are known (complete information), and (3) simulations terminate relatively fast (game length is limited).

The basic structure of MCTS is as follows. In MCTS, each node i represents a given position (also called a state) of the game. A node contains at least the following two pieces of information: (1) the current value v_i of the position (usually the average of the results of the simulated games that visited this node), and (2) the visit count n_i of this position. MCTS usually starts with a tree containing only the root node.

MCTS consists of four main steps, repeated as long as there is time left. The steps are as follows. (1) In the *selection step* the tree is traversed from the root node until we reach a node E, where we select a child that is not part of the tree yet. (2) Next, in the *expansion step* this child of E is added to the tree.[3] (3) Subsequently, during the *simulation step* moves are played in self-play until the end of the game is reached. The result R of this "simulated" game is $+1$ in case of a win for Black (the first player in Go), 0 in case of a draw, and -1 in case of a win for White. (4) In the *backpropagation step*, R is propagated backwards, through the previously traversed nodes. Finally, the move played by the program is the child of the root with the highest visit count. An outline of the four step of MCTS are depicted in Figure 3.1.

The pseudo-code for MCTS is given in Algorithm 3.1. In the pseudo-code, $\mathcal{T}$ is the set of all nodes (the search tree), *Select(Node N)* is the procedure, which returns one child of the node N. *Expand(Node N)* is the procedure that adds one node to the tree, and returns

[3] Of course more children can be stored at once. For a discussion of this topic see Subsection 3.3.2.

this node. *Play_simulated_game(Node N)* is the procedure that plays a simulated game from the newly added node, and returns the result $R \in \{-1, 0, 1\}$ of this game. *Backpropagate(Integer R)* is the procedure that updates the value of the node depending on the result R of the last simulated game. $\mathcal{N}_c$*(node N)* is the set of the children of node N.

Data: root_node
Result: best_move
while *(has_time)* **do**
  $current\_node \leftarrow root\_node$

  `/* The tree is traversed */`
  while $(current\_node \in \mathcal{T})$ **do**
    $last\_node \leftarrow current\_node$
    $current\_node \leftarrow Select(current\_node)$
  end
  `/* A node is added */`
  $last\_node \leftarrow Expand(last\_node)$

  `/* A simulated game is played */`
  $R \leftarrow Play\_simulated\_game(last\_node)$

  `/* The result is backpropagated */`
  $current\_node \leftarrow last\_node$
  while $(current\_node \in \mathcal{T})$ **do**
    $Backpropagation(current\_node, R)$
    $current\_node \leftarrow current\_node.parent$
  end
end
return $best\_move = argmax_{N \in \mathcal{N}_c}(root\_node)$

Algorithm 3.1: Pseudo-code for Monte-Carlo Tree Search.

3.3 MCTS

In this section, we discuss strategies to realize the four basic steps of MCTS: selection (Subsection 3.3.1), expansion (Subsection 3.3.2), simulation (Subsection 3.3.3), and backpropagation (Subsection 3.3.4).

3.3.1 Selection

The selection step works in the following way. From the root node, a *selection strategy* is applied recursively until a position is reached that is not a part of the tree yet. The selection strategy controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation).

On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration). A similar balancing of exploitation and exploration has been studied in the literature, in particular with respect to the Multi-Armed Bandit (MAB) problem (Robbins, 1952). The MAB problem considers a gambling device and a player, whose objective is to maximize the reward from the device. At each time step, the player can select one of N arms of the device, which gives a reward. In most settings, the reward obeys a stochastic distribution. The selection problem of MCTS can be viewed as a MAB problem for a given node: the problem is to select the next move to play, which will give an unpredictable reward (the outcome of a single random game). Knowing the past results, the problem is to find the optimal move. However, the main difference with the MAB problem is that MCTS works by using recursively several selections: the selection at the root node, the selection at depth one, the selection at depth two, etc. Several selection strategies have been designed for MCTS such as OMC (Chaslot *et al.*, 2006a) and PBBM (Coulom, 2006). Some MCTS selection strategies have been derived from MAB algorithms such as UCT (Kocsis and Szepesvári, 2006) and UCB1-TUNED (Gelly and Wang, 2006). Below, we will discuss the following four selection strategies: OMC, PPBM, UCT, and UCB1-TUNED.

- **OMC.** In Chaslot *et al.* (2006a), we proposed OMC (Objective Monte-Carlo). It consists of two parts.[4] First, an *urgency function* determines the urgency $U(i)$ for each possible move i. Second, a *fairness function* decides which move to play, with the aim to play each move proportionally to its urgency. The urgency function in OMC is as follows:

$$U(i) = \mathit{erfc}(\frac{v_0 - v_i}{\sqrt{2}\sigma_i}) \tag{3.2}$$

  where $erfc(\cdot)$ is the complementary error function, v_0 is the value of the best move, and v_i and σ_i are respectively the value and the standard deviation of the move under consideration. The idea behind this formula is to have $U(i)$ proportional to the probability of a move to be better than the current best move. Next, the child is chosen according to the following rule: select the node i that maximizes the fairness function f_i:

$$f_i = \frac{n_p \times U(i)}{n_i \times \sum_{j \in S_i} U(j)} \tag{3.3}$$

  where n_i is the visit count of i, and n_p is the visit count of p, and S_i is the set containing the sibling nodes of i.

- **PBBM.** In 2006, Coulom (2006) proposed PBBM (Probability to be Better than Best Move). Just like OMC, it consists of an urgency function and a fairness function. Its urgency function $U(i)$ is again proportional to the probability of the move to be better than the current best move. The difference with OMC is that the standard deviation of the best move is taken into account. The urgency function in PBBM is as follows:

[4] We thank Rémi Coulom for the idea of splitting this algorithm into two parts.

$$U(i) = exp(-2.4 \times \frac{v_0 - v_i}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}}) \tag{3.4}$$

where v_0 and σ_0 are the value and standard deviation of the best move, respectively. Similarly, v_i and σ_i are the value and the standard deviation of the move under consideration.

- **UCT.** Kocsis and Szepesvári (2006) proposed the UCT (Upper Confidence bounds applied to Trees) strategy. This strategy is easy to implement, and used in many programs (see Section 3.5). It is nowadays used in MANGO as well. UCT adapts the UCB (Upper Confidence Bounds) method initially developed for MAB (Auer *et al.*, 2002). UCT works as follows. Let I be the set of nodes reachable from the current node p. UCT selects a child k of the node p that satisfies Formula 3.5:

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{3.5}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which has to be tuned experimentally.

- **UCB1-TUNED.** Gelly and Wang (2006) proposed to use the UCT variant UCB1-TUNED, originally described by Auer *et al.* (2002). UCB1-TUNED selects a child k of the node p that satisfies Formula 3.6:

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i} \times min\{\frac{1}{4}, V_i(n_i)\}} \right) \tag{3.6}$$

where

$$V_i(n_i) = (\frac{1}{n_i} \sum_{t=1}^{n_i} R_{i,t,j}^2 - v_i^2 + \sqrt{\frac{2 \ln n_p}{n_i}}) \tag{3.7}$$

is an estimate upper bound for the variance of v_i. $R_{i,t,j}$ is the t^{th} payoff, obtained in node i for player j.

Recently, Audibert and Bubeck (2009) proposed MOSS (Minimax Optimal Strategy in the Stochastic case) that minimizes the regret depending on the number of moves. This selection strategy is a refinement of UCB/UCT. Other selection strategies have been designed that assume smoothness between the different moves, such as BAST (Bandit Algorithm for Smooth Trees) by Coquelin and Munos (2007) and HOO (Hierarchical Optimistic Optimization) by Bubeck *et al.* (2008). Additionally, for generating an opening book with

MCTS, we introduce QBF (Quasi Best-First) and BDS (Beta-Distribution Sampling) in Chapter 8.

We note that all these selection strategies presented here are game-independent and do not use any domain knowledge. Selection strategies that use knowledge will be discussed in Chapter 5. Finally, we remark that in some programs (e.g., CRAZY STONE and MANGO) a selection strategy is only applied in nodes with a visit count higher than a certain threshold T (Coulom, 2006). If the node has been visited fewer times than this threshold, the next move is selected according to the *simulation strategy* (see Subsection 3.3.3).

3.3.2 Expansion

The expansion step adds nodes to the MCTS tree. Because for most domains the whole game tree cannot be stored in memory, an expansion strategy decides that, for a given node L, whether this node will be expanded by storing one or more of its children in memory. A popular expansion strategy is the following:

- One node is added per simulated game. This node corresponds to the first position encountered during the traversal that was not already stored (Coulom, 2006).

This strategy is simple, efficient, easy to implement, and does not consume too much memory in general. Other strategies are possible as well. For instance, it is possible to expand the tree to a certain depth (e.g., 2 or 3 ply) before starting the search. It is also possible to add all the children of a node to the tree as soon as a certain number T of simulations have been made through this node. This strategy is only possible when a large amount of memory is available. In contrast to this strategy, one may forbid any node expansion before a certain number T of simulations have been made through this node. This allows to save memory, and reduces only slightly the level of play (Dailey, 2006). In general, the effect of these strategies on the playing strength is small. The strategy of creating one node per simulation is therefore sufficient in most cases.

3.3.3 Simulation

Simulation (also called *playout*) is the step that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *simulation strategy*. The use of an adequate simulation strategy has been shown to improve the level of play significantly (Bouzy, 2005; Gelly *et al.*, 2006). The main idea is to play interesting moves according to heuristic knowledge (e.g., for Go: patterns, capture considerations, and proximity to the last move).

A simulation strategy is subject to two trade-offs. The first one is the trade-off between search and knowledge. Adding knowledge to the simulation strategy increases the playing strength. Simulated games become more accurate and their results more reliable. However, if the heuristic knowledge is too computationally expensive, the number of simulations per second will decrease too much. The MCTS search tree is shallow and the playing strength will drop. The second trade-off deals with exploration vs. exploitation. If the strategy is too stochastic (e.g., too much randomness), too much exploration takes place. The moves played in the simulated games are often weak, causing the simulations to be unrealistic, and the level of the Monte-Carlo program decreases. In contrast, if the strategy is too

deterministic (e.g., if the selected move for a given position is almost always the same), too much exploitation takes place. The exploration of the search space becomes too selective, causing the simulations to be biased, and the level of the Monte-Carlo program decreases as well.

Due to these two trade-offs, elaborating an efficient simulation strategy is a difficult issue. There are two ways to assess the quality of a simulation strategy. We define a simulation strategy A to be *better* than a simulation strategy B, if A wins more games against B when playing against each other (i.e., without using MCTS). We define that a simulation strategy A is *MCTS-better* than a simulation strategy B, if an MCTS program using A wins more games against the same MCTS program using B. The goal of designing a simulation strategy is to make it MCTS-better. Bouzy and Chaslot (2006) showed that it is possible to have a simulation strategy which is better than another one, without being MCTS-better. This phenomena has been a reported by Gelly and Silver (2007) as well.

In Chapter 4 we will discuss the simulation strategies used in Go programs, especially for our programs MANGO and MOGO. Moreover, in Section 3.5 the simulation strategies for a few other domains are briefly described.

3.3.4 Backpropagation

Backpropagation is the step that propagates the *result* of a simulated game k backwards from leaf node L to the nodes it had to traverse to reach this leaf node. For instance, for a two-player zero-sum game (e.g., Go), this result is counted positively ($R_k = +1$) if the game is won, and negatively ($R_k = -1$) if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to compute the *value* v_L of a node. The most popular and most effective strategy is Average, which takes the plain average of the results of all simulated games made through this node (Coulom, 2006), i.e., $v_L = (\sum_k R_k)/n_L$. It was first used by Kocsis and Szepesvári (2006), and since 2007 by the vast majority of the programs. Finding a better backpropagation than Average is a challenge for MCTS. Below, we present several other backpropagation strategies proposed in the literature. They are called Max, Informed Average, Mix, and MCTS-Solver.

- **Max.** The Max strategy backpropagates the value in a negamax way (Knuth and Moore, 1975). The value of a node *p* is the maximum value of its children. Using Max backpropagation in MCTS does not give good results (Coulom, 2006). When the number of children of a node is high, and the number of simulations is low, the values of the children are noisy. So, instead of being really the best child, it is likely that the child with the best value is simply the most lucky one. Backing up the maximum value overestimates the best child and generates a great amount of search instability (Coulom, 2006; Chaslot *et al.*, 2006a).

  The noise of the value of a node can be modelled with the help of the Central Limit Theorem. Let $R_i \in \{-1, 0, +1\}$ be the result of the simulated game i, let σ be the standard deviation of the random variable R_i and let n be the number of simulated games played. The Central Limit Theorem states that the standard deviation of the mean $m_n = \frac{\sum_{i=1}^{n} R_i}{n}$ approaches $\frac{\sigma}{\sqrt{n}}$ when n approaches ∞.

  Therefore, we can deduce a statistical relation between the real value of this node,

denoted by v_∞, and the observed value v_n of the node:

$$v_n = v_\infty + x,$$

where x is a random variable with mean value 0 and standard deviation $\frac{\sigma}{\sqrt{n}}$. Based on this statistical relation, we distinguish three situations:

- If σ is large (in practice $\sigma > 0.3$), v_n is likely to be different from the real value. Therefore, the Max value is most likely to be an overestimated value, because it is the Max of (more or less) random values. Thus, if σ is large, it is not possible to use Max. In this case, the average of the children gives better results.
- If σ is close to 0 (in practice $\sigma < 0.01$), then Max can be applied.
- For moderate values of σ, a compromise between Average and Max would give a better evaluation than Average or Max, such as Mixed or Informed Average (see below).

- **Informed Average.** As an alternative for Max, we proposed Informed Average (Chaslot *et al.*, 2006a). The strategy aims to converge faster to the value of the best move than Average by assigning a larger weight to the best moves. It is computed with the following formula:

$$v_p = \frac{\sum_i (v_i \cdot n_i \cdot U_i)}{\sum_i (n_i \cdot U_i)} \tag{3.8}$$

where U_i represents the urgency of the move, calculated by Formula 3.2. We showed that MCTS using Informed Average estimates a position better than MCTS using Max (Chaslot *et al.*, 2006a). However, an MCTS program using Informed Average played not as well as an MCTS program using Average.

- **Mix.** As an alternative for Max, Coulom (2006) proposed Mix. It computes the value of the parent node p with the following formula:

$$v_p = \frac{v_{mean} \cdot w_{mean} + v_r \cdot n_r}{w_{mean} + n_r} \tag{3.9}$$

where v_r is the value of the move which has the highest number of simulations, and n_r the number of times it has been played. v_{mean} is the average of the values of the child nodes and w_{mean} the weight of this average.[5] This backpropagation was used in CRAZY STONE when it won the 9×9 Go tournament of the Computer Olympiad 2006. However, the new version of CRAZY STONE uses Average, which gives better results.

[5] In CRAZY STONE, this weight has a constant value (i.e., 162, see Coulom, 2006) until a certain number of simulated games (i.e., 1296, see Coulom, 2006) has been played. From that point on, w_{mean} is the number of simulated games played divided by 8.

- **MCTS-Solver.** To play narrow tactical lines better in sudden-death games such as LOA, Winands, Björnsson, and Saito (2008) proposed MCTS-solver. In addition to backpropagating the values by using Average, it propagates the game-theoretical values ∞ or $-\infty$. The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax. If the selected move (child) of a node returns ∞, the node is a win. To prove that a node is a win, it suffices to prove that one child of that node is a win. Because of negamax, the value of the node will be set to $-\infty$. In the case that the selected child of a node returns $-\infty$, all its siblings have to be checked. If their values are also $-\infty$, the node is a loss. To prove that a node is a loss, we must prove that all its children lead to a loss. Because of negamax, the node's value will be set to ∞. In the case that one or more siblings of the node have a different value, we cannot prove the loss. Therefore, -1 is propagated, the result for a lost game, instead of $-\infty$, the game-theoretical value of a lost position. The node will be updated then according to the backpropagation strategy Average. Winands *et al.* (2008) showed that a LOA program using MCTS-Solver defeats a program using Average backpropogation by a winning score of 65%.

3.4 Final Move Selection

After the simulations, the move finally played by the program in the actual game is the one corresponding to the "best child" of the root. There are different ways to define which child is the best.

1. *Max child.* The max child is the child that has the highest value.
2. *Robust child.* The robust child is the child with the highest visit count.
3. *Robust-max child.* The robust-max child is the child with both the highest visit count and the highest value. If there is no robust-max child at the moment, more simulations are played until a robust-max child is obtained (Coulom, 2006).
4. *Secure child.* The secure child is the child that maximizes a lower confidence bound, i.e., which maximizes the quantity $v + \frac{A}{\sqrt{n}}$, where A is a parameter (set to 4 in our experiments), v is the node's value, and n is the node's visit count.

In the experiments performed in MANGO we did not measure a significant difference between the methods discussed above, when a sufficient number of simulations per move was played. However, when only a short thinking time per move was used (e.g., below 1 second), choosing the max child turned out to be significantly weaker than other methods.

3.5 MCTS Applications

In this section, we discuss the implementations of MCTS for different application domains. We only consider domains with perfect information. We make a distinction between games with one, two or multiple (more than two) players. Another distinction is made between deterministic and stochastic games.

3.5.1 Deterministic One-Player Games

The first application of MCTS to a deterministic one-player game – also known as puzzles or scheduling problems– was for Production Management Problems (PMP) (Chaslot *et al.*, 2006b), which are found in manufacturing and airline scheduling. For PMP, we used OMC as a selection strategy and a plain random simulation strategy. Because OMC was originally designed for deterministic two-player games, it had to be adapted. An essential difference between deterministic one-player and two-player games is that there is no *uncertainty on the opponent's play*. This has two consequences. First, the best line and its score found by MCTS should be kept in memory. Second, it is possible to be more greedy than in other domains, since we do not have to consider any unknown decision made by an opponent. For the latter, we modified the OMC formula (3.2) to increase the greediness as follows:

$$U(i) = \mathit{erfc}(G \times \frac{v_0 - v_i}{\sqrt{2}\sigma_i})$$

where G is the greediness coefficient. We obtained the best result with a coefficient with a value between 50 to 100. The disadvantage of this setting is that MCTS does not give a second chance to moves that had a bad evaluation in the beginning. The advantage of this setting is that the MCTS tree manages to reach the end of the game. For the best line of play, decisions based on the selection strategy OMC are made at any depth.

Another deterministic one-player game where MCTS was applied was SameGame. Schadd *et al.* (2008a) obtained with their MCTS variant SP-MCTS the world record in this game. Cazenave (2009) broke this record with his own MCTS variant, called Nested Monte-Carlo search. It uses nested levels of simulations in order to guide the search. His method surpassed the best human score in Morpion Solitaire as well.

Finally, we note that some optimization problems can be regarded as one-player games. For instance, Mesmay *et al.* (2009) proposed the MCTS variant TAG (Threshold Ascent applied to Graph) for optimizing libraries for different platforms.

3.5.2 Deterministic Two-Player Games

For deterministic two-player games such as chess and checkers using $\alpha\beta$ with a strong evaluation function was the framework for building a strong AI player. However, this framework achieved hardly any success in Go. In 2006 MCTS started a revolution in the field of Computer Go. MCTS-based programs have won every 9×9 and 19×19 Go tournament at the Computer Olympiads since 2006 and 2007, respectively. It was the MCTS program MoGo Titan[6] that achieved the milestone of defeating a human Go professional in an official match with a 9-stones handicap. To underline the significance of this result, only a decade ago a computer Go expert even managed to defeat a top program with a handicap of 29 stones (Müller, 2002). Besides MoGo (Gelly *et al.*, 2006; Gelly, 2007; Lee *et al.*, 2009), top MCTS Go program are Crazy Stone (Coulom, 2006), Steenvreter (Van der Werf, 2007), Many Faces of Go (Fotland, 2009), Fuego (Enzenberger and Müller, 2009), and Zen written by Sai Fujiwara.

Besides Go, MCTS is also used in other deterministic two-player games, such as Amazons (Lorentz, 2008; Kloetzer, Iida, and Bouzy, 2009; Kloetzer, 2010) and LOA (Winands *et al.*,

[6] MoGo Titan is a version of MoGo running on the Dutch national supercomputer Huygens.

2008; Winands and Björnsson, 2010). Lorentz (2008) was able to create a hybrid MCTS program using UCT and a strong positional evaluation function. After a fixed length, the simulation is terminated and subsequently scored based on the value of the evaluation function. The MCTS program INVADERMC beat INVADER, the original $\alpha\beta$ program, over 80% of the time under tournament conditions. Moreover, INVADERMC won the Computer Olympiads of 2008 and 2009. Winands and Björnsson (2010) investigated several simulation strategies for using a positional evaluation function in a MCTS program for the game of LOA. Experimental results reveal that the Mixed strategy is the best among them. This strategy draws the moves randomly based on their weights in the first part of a simulation, but selects them based on their evaluation scores in the second part of a simulation. The simulation can be stopped any time when heuristic knowledge indicates that the game is probably over. Using this simulation strategy the MCTS program plays at the same level as the $\alpha\beta$ program MIA, the best LOA playing entity in the world.

3.5.3 Deterministic Multi-Player Games

Sturtevant (2008) applied MCTS in the multi-player games Chinese Checkers, Spades, and Hearts. For the game of Chinese Checkers, he showed that MCTS was able to outperform the standard multi-player search methods max^n (Luckhardt and Irani, 1986) and paranoid (Sturtevant and Korf, 2000) equipped with a strong evaluation function. For the perfect-information versions of Spades and Hearts, MCTS was on par with the state-of-the-art. Moreover, Sturtevant (2008) showed that MCTS using UCT computes a mixed-strategy equilibrium as opposed to max^n, which computes a pure-strategy equilibrium. Cazenave (2008) addressed the application of MCTS to multi-player Go. He proposed three UCT variants for multi-player games: Paranoid UCT, UCT with Alliances, and Confident UCT.

3.5.4 Stochastic One-Player Games

The first application of MCTS in a stochastic one-player game was in the Sailing Domain. The Sailing Domain is a stochastic shortest-path problem (SSP), where a sail boat has to find the shortest path between two points of a grid under variable wind conditions. Kocsis and Szepesvári (2006) tackled this domain by using UCT. The sail boat position was represented as a pair of coordinates on a grid of finite size. The controller had 7 actions available in each state, giving the directions to the neighbouring grid positions. The action of which the direction is just the opposite of the direction of the wind was forbidden. Each action had a cost in the range between [1, 8.6], depending on the direction of the action and the wind. UCT was compared to two planning algorithms: ARTDP (Barto, Bradtke, and Singh, 1995) and PG-ID (Peret and Garcia, 2004). The conclusion was that UCT requires significantly less simulations to achieve the same performance as ARTDP and PG-ID.

3.5.5 Stochastic Two-Player Games

Van Lishout, Chaslot, and Uiterwijk (2007) were the first to apply MCTS to the stochastic two-player game Backgammon. UCT was used as a selection strategy, with a plain random simulation strategy. The MCTS program was able to find an expert opening in one third of

the dice rolls, but the program was significantly weaker than the state-of-the-art programs based on Expectimax (cf. Berger, 2007).

3.5.6 Stochastic Multi-Player Games

Modern strategic board games are increasing in popularity since their (re)birth in the 1990's. Strategic board games are of particular interest to AI researchers because they provide a bridge between classic (two-player, deterministic) board games and video games. Modern strategic board games have the property that they are stochastic multi-player games. We applied MCTS – augmented with a limited amount of domain knowledge – to the stochastic multi-player game Settlers of Catan (Szita, Chaslot, and Spronck, 2010). For the experiments, the rules were changed in such a way that it became a stochastic *perfect-information* game. The MCTS program SMARTSETTLERS defeated convincingly the strongest open source AI available, JSETTLERS, and is a reasonably strong opponent for humans.

3.5.7 General Game Playing

The aim of General Game Playing is to create intelligent agents that automatically learn how to play many different games at an expert level without any human intervention. The most successful GGP agents in the past have used traditional game-tree search combined with automatically learnt heuristic functions for evaluating game states. However, since 2007, the MCTS program CADIAPLAYER, written by Finnsson and Björnsson (2008), has won the GGP tournament two times in a row. This program uses UCT, with an online learning algorithm for the simulations.

3.6 Chapter Conclusions

In this chapter we have presented a general framework for Monte-Carlo Tree Search (MCTS). It is a best-first search method that does not require a positional evaluation function in contrast to $\alpha\beta$ search. It is based on a randomized exploration of the search space. Using the results of previous explorations, MCTS gradually builds a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves. MCTS has led to the best programs in several domains. We mention Production Management Problems, Library Performance Tuning, SameGame, Morpion Solitaire, Sailing Domain, Amazons, LOA, Chinese Checkers, Settlers of Catan, General Game Playing, and in particular Go. In the larger picture, MCTS is also attractive for many other application domains, because MCTS promises to overcome the "knowledge-acquisition bottleneck" and to make intelligent applications easier to engineer in many fields.

We discussed the four main steps of MCTS: *selection*, *expansion*, *simulation*, and *backpropagation*. Each step has a strategy associated that implements a specific policy. Regarding selection, the UCT strategy is used in many programs as a specific selection strategy because it is simple to implement and effective. A standard selection strategy such as UCT does not take domain knowledge into account, which could improve an MCTS program even further. Next, a simple and efficient strategy to expand the tree is creating one node per simulation. Subsequently, we pointed out that building a simulation strategy is probably

the most difficult part of MCTS. For a simulation strategy, two balances have to be found: (i) between search and knowledge, and (ii) between exploration and exploitation. Furthermore, evaluating the quality of a simulation strategy requires it to be assessed together with the MCTS program using it. The best simulation strategy without MCTS is not always the best one when using MCTS. Finally, the backpropagation strategy that is the most successful is taking the average of the results of all simulated games made through a node.

The two strategies that, if enhanced, we expect to give the biggest performance gain in a MCTS program are the simulation strategy and the selection strategy. In Chapters 4 and 5, we investigate how to enhance the simulation strategy and selective strategy of an MCTS program, respectively.

Chapter 4

Enhancing the Simulation Strategy with Knowledge

This chapter is partially based on the publications:

B. Bouzy and G.M.J-B. Chaslot (2006). Monte-Carlo Go Reinforcement Learning Experiments. *IEEE 2006 Symposium on Computational Intelligence in Games* (eds. G. Kendall and S. Louis), pp. 187-194.

G.M.J-B. Chaslot, C. Fiter, J-B. Hoock, A. Rimmel, and O. Teytaud (2010). Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. *Advances in Computer Games Conference (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–13, Springer-Verlag, Heidelberg, Germany.

In Chapter 3, we introduced the fundamental MCTS strategies. The two strategies that if enhanced lead to the biggest performance gain in an MCTS program are the simulation strategy and the selection strategy. In this chapter, we answer the first research question by investigating how to enhance the simulation strategy of an MCTS program with knowledge.

The *knowledge* transforms the plain random simulations into more sophisticated *pseudo-random* simulations (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Zhang, 2008). The knowledge for the simulation strategy can be designed by a human expert (cf. Rimmel, 2009) or created (semi)-automatically by machine learning. We discuss two different simulation strategies that apply knowledge: *urgency-based* and *sequence-like* simulation. Moreover, methods are investigated to learn the knowledge of the simulation strategy.

The chapter is organized as follows. In Section 4.1, we explain the two simulation strategies, urgency-based and sequence-like simulations, and give main experimental results. In Section 4.2, we discuss how to learn the knowledge of the simulation strategy and give some experimental results. In Section 4.3, we present related research in the domain of General Game Playing. Section 4.4 gives the chapter conclusions.

4.1 Simulation Strategies

In this section, we discuss two successful simulation strategies. They were first proposed in Go, and are also applicable in other domains. In Subsection 4.1.1, we present urgency-based simulation that was proposed for INDIGO and used in several other programs such as MANGO and CRAZY STONE. In Subsection 4.1.2, we discuss sequence-like simulation that has been proposed for MOGO.

4.1.1 Urgency-Based Simulation

Bouzy (2005) proposed *urgency-based simulation*. It is basically a 2-step method. In the first step, an urgency value, U_j, is computed for each move j. In the second step, taking the urgencies into account a move is randomly drawn. The probability of each move to be selected is calculated by Formula 4.1.

$$p_j = \frac{U_j}{\sum_{k \in M} U_k} \tag{4.1}$$

where M is the set of all possible moves for a given position.

If the urgency-based simulation strategy is too random (e.g., all urgencies have similar values), the level of play of the MCTS program will be close to the level of a program that draws plain randomly (i.e., without using any knowledge). If the urgency-based simulation strategy is too deterministic, the simulations will be too similar, which will lead to a lack of exploration and hence to meaningless Monte-Carlo simulations.

Urgency-based simulation is used in several Go programs, and in particular in the top Go programs INDIGO (Bouzy, 2005), MANGO (Chaslot *et al.*, 2007), and CRAZY STONE (Coulom, 2007).

Computing Urgencies in the Game of Go

In order to compute the urgency value of each move, Bouzy (2005) computed for his program INDIGO the urgency as the sum of two values: (1) the capture-escape value V_{ce} and (2) the pattern value V_p.

1. The capture-escape value V_{ce} depends on (1) the number of stones that could be captured, and on (2) the number of stones that could escape a capture by playing the move. This feature is important for Go and easy to implement. In general, it is a good thing to capture opponent's pieces and not a good thing to have your own pieces captured. Furthermore, it is also easy to create a procedure checking whether stones might be immediately captured. As a consequence, moves that capture opponent's pieces are given a high weight, and moves that prevent pieces to be captured are also given a quite high weight. These weights are set in such a way to avoid as many mistakes as possible; not capturing a large group has a high probability of being a mistake, and letting a large group be captured has also a high probability of being a mistake.

2. The pattern value V_p uses 3×3 patterns. Each pattern is empty in its centre, where the move to be considered has to be played. 3×3 patterns have been chosen by Bouzy (2005) for two reasons: (1) 3×3 patterns are fast to match and (2) they represent relatively well the important concepts of connection and cut. Each pattern was given a weight depending on how good the move seemed to be for its surrounding 3×3 pattern. So the pattern value of a move i is computed as follows: $V_p(i) = \sum_j w_j \times m_{i,j}$ where w_j is the weight of pattern j, and $m_{i,j}$ is 1 if move i matches pattern j and is 0 otherwise.

Because of the 3×3 patterns, programs using this specific urgency-based simulation are significantly slower and consequently the number of simulations is decreased. Despite this reduction in the number of simulations, INDIGO with urgency-based simulations wins 68% of the game against INDIGO with plain random simulations on the 9×9 board, and 97% on the 19×19 board (Bouzy, 2005).

4.1.2 Sequence-Like Simulation

In this section, we discuss the so-called "sequence-like simulation" introduced by Gelly *et al.* (2006) in the Go program MOGO. This simulation strategy consists of selecting each move in the proximity of the last move played. This leads to moves being played next to each other, creating a sequence of adjacent moves. To select which move to play in the neighbourhood of the last move, 3×3 patterns similar to the ones proposed by Bouzy (2005) were used. After each move, the program scans for 3×3 patterns at a Manhattan distance of 1 from the last move. If several patterns are found, one is chosen randomly. The move is then played in the centre of the chosen pattern. If no pattern is found, a move is chosen randomly on the board. These patterns were hand-made by Yzao Wang. It is also possible to learn these patterns automatically, for instance by using a genetic algorithm. Gelly implemented such an algorithm, but the results were not significantly better than the expert sequence-like simulation (cf. Subsection 4.2.4).

Sequence-like strategy increased the level of play against GNU GO[1] from winning $8.88 \pm 0.42\%$ of the games to winning $48.62 \pm 1.1\%$ of the games (with 5,000 simulations per move, GNU GO version 3.7.10 level 0, cf. Gelly and Silver, 2007). Based on our experience with the sequence-like-simulation strategy in MOGO, we believe that it is particularly efficient because it simplifies the position. In particular, sequence-like play creates a border which divides the territory between the two players.

Adding diversity in the Sequence-Like Simulation

A possible way of adding diversity in sequence-like simulation is to play in the middle of an empty area. Playing in an empty area, the simulation may go in any direction. The following procedure is used. If the sequence-like simulation cannot find a pattern, we look randomly for an intersection with 8 empty neighbouring intersections. If such an intersection cannot be found after n trials, a random move is played. This enhancement was proposed by Rimmel (2009). It led to a wining rate of $78.4 \pm 2.9\%$ against the version of MOGO without this enhancement (Chaslot *et al.*, 2010).

[1] GNU GO is the Go program developed by the Free Software Foundation. This program is not MCTS-based. See http://www.gnu.org/software/gnugo/gnugo_toc.html for more details.

4.2 Learning Automatically the Simulation Strategy

In this subsection we discuss how to learn automatically the parameters (weights) of a simulation strategy. We discuss five alternative fitness functions, in order to be able to use faster learning algorithms than evolutionary algorithms. These five functions use (1) the matches between programs (Subsection 4.2.1) (2) the results of simulated games (Subsection 4.2.2), (3) move evaluations (Subsection 4.2.3), (4) mean-squared errors on a set of positions (Subsection 4.2.4), or (5) the difference between the errors made by the first player and the errors made by the second player (Subsection 4.2.5).

4.2.1 Learning from Matches between Programs

As discussed in Chapter 3, the simulation strategy should improve the level of the MCTS program using it. So the obvious fitness function is the number of victories of an MCTS program against a state-of-the-art program. However, this fitness function has two problems. First, the number of parameters can be quite huge. For instance, in the case of urgency-based simulation Bouzy (2005), quite a number of patterns have to be matched in the simulations during game-play. Second, and most important, it is difficult to evaluate how each pattern contributed to the victory: is it possible to create a feature that reveals which pattern contributed the most amongst the patterns played by the winning player? For such difficult problems, evolutionary algorithms are usually employed. However, using them to tune the patterns of Bouzy's urgency-based simulation would take too much time. In Chapter 6, we propose an algorithm, called the Cross-Entropy Method (CEM), to learn from matches between programs. CEM is related to Estimation-of-Distribution Algorithms (EDAs) (see Muehlenbein, 1997) and is able to tune parameters in every part of the MCTS tree.

4.2.2 Learning from the Results of Simulated Games

Learning from the results of simulated games consists of playing games between two simulation strategies (let S_1 and S_2 be these strategies), and observe the results $r_1, r_2, ..., r_n$ of these games. The learning algorithm is then applied after each game, based on the decisions that have been made by S_1 and S_2 for the game i, and the result r_i.

The first experiments for this method were done in the Go program INDIGO by Bouzy and Chaslot (2006). The learning algorithm, inspired by reinforcement learning, was learning either starting from a plain random simulation strategy (S_r) or from expert patterns (S_{ep}) (see Section 4.1.1). However, the program using the learnt patterns performed worse than the one using the original simulation strategy. In addition, we observed that making the strategy more *deterministic* (e.g., by taking the square of the weight values) does generally increase the strength of a simulation strategy against another simulation strategy. However, it diminished the strength of the program employing the more deterministic strategy. When learning the resulting pattern values, we observed that using the result of simulated games increased continuously the urgencies of the moves which are the best on average. This caused a deterministic strategy, which harmed the performance of the program INDIGO.

Learning from the results of simulated games was also tested by Gelly and Silver (2007). They used a linear-value-function approximation and the TD(λ) algorithm (Sutton, 1988). The results were similar to the ones obtained by us (cf. Subsection 5.A of Bouzy and

Chaslot, 2006): the new (learnt) simulation strategy outperformed the old simulation strategy, but not the performance of the MCTS program using it.

4.2.3 Learning from Move Evaluations

We propose that learning the weights of the urgency-based simulation should not be based on the results of simulated games in which the patterns matched, but rather on the differences of move evaluations (Bouzy and Chaslot, 2006). There are three issues: (1) how each move should be evaluated, (2) how the weights should be related to the move evaluations, and (3) how this relation should be learnt.

To tackle issue (1), we chose as a move evaluation, denoted v_i for a move i, to use fast Monte-Carlo Evaluations (MCEs, see Section 3.1). We found that the learning algorithm gave good results with as little as 20 simulations per move. To tackle issue (2), we first defined a matching between moves and patterns. For each possible move i, there is exactly one 3×3 pattern matching for this move. We denote w_i as the weight of the pattern that matches for the move i. Second, we chose to associate the move evaluation v with the weight w such that for every pair of legal moves (a, b) in a board position, we have the following target function:

$$e^{C \times (v_a - v_b)} = \frac{w_a}{w_b}$$

where C is the exploration-exploitation constant. This formula cannot be satisfied for every pair of moves in a board position, so the weights are learnt to minimize the quantity $e^{C \times (v_a - v_b)} - \frac{w_a}{w_b}$ on average.

To tackle issue (3), we chose a learning algorithm adapted from the *tracking algorithm* proposed by Sutton and Barto (1998). The details of this algorithm are as follows. For each move i, we define $Q_i = log(w_i)$.

The difference δ between the target function and the data is defined by:

$$\delta = Q_a - Q_b - C \times (v_a - v_b)$$

for each pair of moves (a, b) The updates are done by:

$$Q_a \leftarrow Q_a - \alpha \times \delta$$

$$Q_b \leftarrow Q_b + \alpha \times \delta$$

The learning coefficient α is proportional to $\frac{1}{\sqrt{n_k}}$, where n_k is the number of times that the pattern k was matched.

The weights were initialized to 10 (i.e., this corresponds to a random simulation strategy). The learning algorithm stopped when the weights of the patterns stabilized. These learnt weights were first used in the Monte-Carlo Go program INDIGO. On the 9×9 board, INDIGO using the learnt patterns scored on average 3 points more for a Go game than the program using expert patterns. However, on the 19×19 board, INDIGO using the learnt patterns scored on average 30 points less than the program using expert patterns. We may explain this finding by the fact that the patterns were learnt on 9×9 boards. To give an indication of the values of the patterns, we have depicted in Figure 4.1 a few patterns with their values.

Coulom (2007) improved our method further with two ideas:

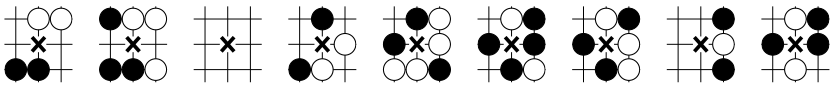


Figure 4.1: Examples of learnt pattern weights. For each move marked with a cross, the weight is given below.

- Instead of using patterns and capture value as the only features for the urgency of a move, Coulom (2007) combined additional features, such as distance to previous moves, distance to the border, statistics from the simulated games, etc.

- Evaluating the moves according to professional games. The training positions are extracted from professional games, and the evaluation of a move is 0 if the professional did not play this move, and 1 if the professional did play this move. The binary values of moves enable to compute the weight using algorithms from the literature developed to compute ELO ratings, such as the minorization-maximization method and the general Bradley-Terry model (Hunter, 2004).

The application of these patterns increased the level of Coulom's program CRAZY STONE on the 9×9 board from a winning rate of 38% against GNU GO 3.6 up to a winning rate of 68%. The thinking time was set to 1 minute and 30 seconds for a complete game.

We may conclude that learning from move evaluations gives results that are at least comparable with expert values. Furthermore, it gives significantly better results than learning from the results of simulated games.

4.2.4 Learning from the Mean-Squared Errors on a Set of Positions

Gelly[2] developed a genetic algorithm that was learning to minimize the mean-squared errors of simulations on a set of 200 Go positions. Each Go position was labelled to have a value of 1 if it was a win for Black, and 0 if it was a win for White. In order to assign automatically the label for a position, an extensive MCTS search was performed. In case MCTS indicated that one of the players was clearly winning, the label was assigned to that player. Otherwise, the game was continued by playing the move advised by MCTS. Several thousands of simulations were run on each of the 200 positions, and the fitness of the genetic algorithm was the mean-squared error between the results of the simulations and the labels. The genotype of the algorithm was a set of heuristic rules depending on the surrounding stones. The set of heuristic rules were similar to the patterns described in Subsection 4.1.2. The learnt simulation strategy was as good as using expert patterns, but decreased the number of simulations. Hence, MOGO still plays with expert patterns.

[2] Private communication.

4.2.5 Learning from Imbalance

Silver and Tesauro (2009) proposed a fitness function called *imbalance*. The imbalance is the difference between the errors made by the first player and the errors made by the second player. The underlining idea is that it is fine to make mistakes in the simulation if the other player makes mistakes as well. The simulation strategy is trained in such a way that the mistake of one player balances out the mistake of the other player. The simulation strategy, called *simulation balancing*, plays some apparently bad moves, but improves the level of the MCTS program. Huang, Coulom, and Lin (2010) showed that learning from imbalance gave better results in the 9×9 Go program ERICA than learning from move evaluations. However, their article concludes that learning the 19×19 board seems out of the reach for simulation balancing. Learning from move evaluations using the method of Coulom (2007) is still better in 19×19 Go.

4.3 Related Research: General Game Playing

In this section we describe two learning algorithms aimed at learning to improve the level of the simulations without making assumptions about the game. The simulation strategy is improved during game play (online learning).

The first learning algorithm, Gibbs sampling, was proposed by Finnsson and Björnsson (2008). It selects the moves to be played in the simulated games according to a Gibbs distribution of their history. Let Qh_j be the percentage of victories for the move j independent of the position. In the simulated games, a move j is selected with probability:

$$p_j = \frac{e^{\frac{Qh_j}{\tau}}}{\sum_{k \in M} e^{\frac{Qh_k}{\tau}}} \tag{4.2}$$

where M is the set of all possible moves for a given position. One can stretch or flatten Formula 4.2 by using the τ parameter ($\tau \to 0$ stretches the distribution, whereas higher values make it more uniform). We remark that the probabilities are adapted during the game, which is a kind of online learning. Moreover, the Qh value is set to the maximum winning score (i.e., 100%) to bias towards similar exploration as is default in the UCT algorithm or in learning the weights of the urgency-based simulation (cf. Subsection 4.2.3). The winning rate against the old version of their General-Game Playing program CADIAPLAYER varied between 54.2% in Connect-4, 54.4% in Checkers, 65% in Othello, and 90% in Breakthrough. The learning algorithm was not only applied to improve the simulation strategy but also the selection strategy. Hence, it is difficult to assess which part of the improvement is due to the enhanced simulation strategy and which part is due to the selection strategy.

The second learning algorithm, state-space knowledge, was proposed by Sharma, Kobti, and Goodwin (2008) to learn a simulation strategy in the domain of General Game Playing. It consists of computing a value for each possible feature of the game. For instance, in Tic-Tac-Toe, the position is represented with features such as “mark(1,1,X)”. Thereafter, the value of a state is computed as a combination of the value of all its features, which is subsequently squashed by using a sigmoid function. This knowledge was used in the selection and simulation step. The results of this approach against the standard MCTS

are the following: 58 wins out of 100 games in Connect-4, 72 wins out of 100 games in Breakthrough, and 56 wins out of 100 games in Checkers. These results are interesting, even if it might be argued that more games are necessary to be statistically significant, and that the method might work better for games with a small state-space complexity.

4.4 Chapter Conclusions and Future Research

In this chapter, we focussed on enhancing the simulation strategy by introducing *knowledge* in the Monte-Carlo simulations. The knowledge transforms the plain random simulations into more sophisticated *pseudo-random* simulations.

We discussed two different simulation strategies that apply knowledge: urgency-based and sequence-like simulation. Strong-expert knowledge chunks for these two strategies have been designed by Bouzy (2005) for his Go program INDIGO, and by Gelly, Wang, Teytaud, Rimmel, and Hoock for their Go program MOGO (Gelly *et al.*, 2006; Chaslot *et al.*, 2010). From their results, we may recommend that several issues are important.

1. **Avoiding big mistakes is more important than playing good moves.** If a move has a high probability to be a bad move, it should be avoided with a high probability. We achieved more improvement by trying to avoid the moves which were bad most of the time, than playing the best move in a specific situation. For instance, in Go not capturing a large group has a high probability of being a big mistake, and letting a large group be captured has also a high probability of being a big mistake.

2. **Simplifying the position.** Some simulation strategies are efficient because they simplify the situation, such as patterns developed by Bouzy (2005) or the sequence-like simulations developed by Gelly *et al.* (2006).

3. **Balancing exploration and exploitation.** The simulation strategy should not become too stochastic, nor too deterministic.

When learning the knowledge of the simulation strategy, we showed that choosing a fitness function is a major issue. The five fitness functions (together with their learning algorithms) that were proposed are the following:

1. **Learning from matches between programs.** The drawback of this method is that it is relatively slow, since learning can only be done in low dimensions. The advantage of this method is that it is able to learn simultaneously the simulation strategy together with other parts of MCTS.

2. **Learning from the results of simulated games.** This method did not lead to improvements with the learning algorithms proposed by Bouzy and Chaslot (2006) and Gelly and Silver (2007).

3. **Learning from move evaluations.** This method, which we proposed in Bouzy and Chaslot (2006), performed better than learning from the results of simulated games. Coulom (2007) enhanced our method in different ways, such that his program CRAZY STONE was improved considerably.

4. **Learning from the mean-squared errors on a set of positions.** This method, proposed by Gelly, was able to achieve the same level as a program using patterns made by a human expert.

5. **Learning from imbalance.** This method, originally proposed by Silver and Tesauro (2009), improved the program ERICA in 9×9 Go.

The main contribution of this chapter is therefore that we developed the first efficient method for learning automatically the simulation strategy by taking the move evaluations into account. Recently, it was shown that learning from imbalance was better for 9×9 Go, but that learning from move evaluations is still better for 19×19 Go (Huang *et al.*, 2010).

Finally, we remark that Finnsson and Björnsson (2008) proposed a simple and efficient algorithm, using Gibbs sampling, to learn the simulations during game play (online). It seems that the improvements are less than when using expert knowledge or offline learning. However, the simplicity of Gibbs sampling and its general applicability make it, in our opinion, an interesting algorithm for future research.

Chapter 5

Enhancing the Selection Strategy with Knowledge

This chapter is based on the following publications:

> G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy (2007). Progressive Strategies for Monte-Carlo Tree Search. *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)* (eds. P. Wang et al.), pp. 655–661.
>
> G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy (2008c). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357.
>
> G.M.J-B. Chaslot, C. Fiter, J-B. Hoock, A. Rimmel, and O. Teytaud (2010). Adding expert knowledge and exploration in Monte-Carlo Tree Search. *Advances in Computer Games Conference (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–13, Springer-Verlag, Heidelberg, Germany.

In Chapter 3, we introduced the basic MCTS strategies. The two strategies which can be improved the most are the simulation strategy and the selection strategy. In Chapter 4, we discussed methods to enhance the simulation strategy. In this chapter we focus on the selection strategy. Chapter 3 discussed that a selection strategy controls the balance between exploitation and exploration. To arrive at a proper balance, this chapter answers the second research question by introducing knowledge in the selection strategy.

Whereas the selection strategies presented in Chapter 3 solely used the winning percentage of the nodes, the algorithms presented in this chapter also use a different kind of information that is introduced by an expert or learnt automatically. We refer to this information as *knowledge*. We propose two methods to integrate (possibly time-consuming) knowledge into the selection strategy: *progressive bias* and *progressive widening*. Progressive bias directs the search according to knowledge. Progressive widening first reduces the branching

factor, and then increases it gradually. We refer to them as "progressive strategies" because the knowledge is dominant when the number of simulations is small in a node, but loses influence progressively when the number of simulations increases.

The structure of this chapter is as follows. Section 5.1 introduces progressive bias and progressive widening. In Section 5.2 we give details on the implementation and the experimental results of these progressive strategies in our Go-playing program MANGO. Subsequently, Section 5.3 presents the performance of the progressive strategies for the Go programs CRAZY STONE and MOGO, and in the LOA program MC-LOA. Section 5.4 discusses more recent related research on enhancing the selection strategy. Finally, Section 5.5 summarizes our contributions, formulates the chapter conclusions, and gives an outlook on future research.

5.1 Progressive Strategies

When the number of simulations is high, the selection strategies presented in Chapter 3 are quite accurate (Chaslot *et al.*, 2006a; Coquelin and Munos, 2007; Coulom, 2006; Kocsis and Szepesvári, 2006). However, they are inaccurate when the number of simulations is low and when the branching factor is high.

We propose "progressive strategies" that perform a soft transition between applying knowledge and using the selection strategy. Such strategies use (1) knowledge and (2) the information available for the selection strategy. A progressive strategy chooses moves according to knowledge when a few simulations have been played, and converges to a standard selection strategy with more simulations.

In the following two subsections we describe the two progressive strategies developed: *progressive bias* (Subsection 5.1.1) and *progressive widening* (Subsection 5.1.2).

5.1.1 Progressive Bias

The aim of the *progressive bias* strategy is to direct the search according to – possibly time-expensive – heuristic knowledge. For that purpose, the selection strategy is modified according to that knowledge. The influence of this modification is important when a few games have been played, but decreases fast (when more games have been played) to ensure that the strategy converges to a pure selection strategy. We modified the UCT selection in the following way. Instead of selecting the move which satisfies Formula 3.5 (see Chapter 3), we propose to select a node k according to Formula 5.1.

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + f(n_i) \right) \tag{5.1}$$

We chose $f(n_i) = \frac{H_i}{n_i+1}$, where H_i represents heuristic knowledge, which depends only on the board configuration represented by the node i. The variables v_i, n_i, n_p, and coefficient C are the same as in Subsection 3.3.1. More details on the construction of H_i are given in Subsection 5.2.1. The selection strategy has the following five properties.

1. When the number of games n_p made through a node p is lower than the threshold T, the simulation strategy is applied instead of Formula 5.1.

2. When $n_p = T$, the selection strategy starts selecting every unexplored child once. The order in which these children are selected is given by $f(n_i)$, i.e., the children with the highest heuristic scores, H_i, are selected first. If every child has been selected at least once, Formula 5.1 is applied.

3. If only a few simulations have been made through the node (e.g., from around 30 to 100 in MANGO), and if the heuristic score H_i is sufficiently high, the term $\frac{H_i}{n_i+1}$ is dominant. Hence, the number of simulations made depends more on the domain knowledge H_i than on the results of the simulated games. It is an advantage to use mainly the domain knowledge at this stage, because the results of only a few simulated games are affected by a large uncertainty. The behaviour of the algorithm is therefore close to the behaviour of a simulation strategy.

4. When the number of simulations increases (e.g., from around 100 to 500 in MANGO), both the results of the simulated games and the domain knowledge have a balanced effect on the selection.

5. When the number of simulations is high (e.g., > 500 in MANGO), the influence of the domain knowledge is low compared to the influence of the previous simulations, because the domain knowledge decreases by $O(1/n_i)$, and the term corresponding to the simulation decreases by $O(\sqrt{\ln n_p/n_i})$. The behaviour of the algorithm is, at this point, close to the behaviour of the standard selection strategy (i.e., UCT). The only difference with plain UCT occurs if two positions i and j have the same value $v_i = v_j$, but different heuristic scores H_i and H_j. Then, the position with the highest heuristic score will be selected more often.

5.1.2 Progressive Widening

We have seen in MANGO that when there is not much time available and simultaneously the branching factor is high, MCTS performs poorly. Our solution, *progressive widening*,[1] consists of (1) reducing the branching factor artificially when the selection strategy is applied, and (2) increasing it progressively as more time becomes available. When the number of games n_p in a node p equals the threshold T, progressive widening "prunes"[2] most of the children. The children, which are not pruned from the beginning, are the k_{init} children with the highest heuristic scores H_i. Next, the children of the node i are progressively "unpruned" according to their heuristic score H_i. An outline of progressive widening is given in Figure 5.1.

5.2 Experiments in MANGO

In this section we present the experimental details and results of the progressive strategies in the Go-playing program MANGO. In Section 5.2.1 we discuss the heuristic knowledge, the time efficiency and the progressive widening parameters. Next, three different series of

[1] We proposed "progressive widening" under the name "progressive unpruning" (Chaslot *et al.*, 2007). This method was proposed simultaneously by Coulom (2007) under the name "progressive widening". It was agreed later that this name was more suitable.

[2] A node is pruned if it cannot be accessed in the simulated games.

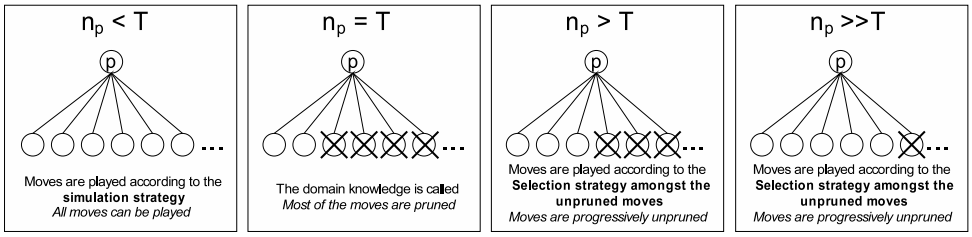


Figure 5.1: Progressive widening.

experiments were conducted in MANGO. Subsection 5.2.2 gives the effect of each progressive strategy against GNU GO. Subsection 5.2.3 shows that these methods also improve the level of the program in self-play. Subsection 5.2.4 assesses the strength of MANGO in computer tournaments.

5.2.1 Experimental Details

In this subsection we give details of the implementation of the progressive strategies in MANGO. First, we describe the heuristic knowledge used. Next, we discuss the time efficiency of using this knowledge. Finally, we explain the application of progressive widening in MANGO.

Knowledge used in MANGO

The two previous progressive strategies require computing a heuristic score H_i for a given board configuration representing the node i. In this subsection we describe the heuristic, which is based on the same idea of urgency-based simulations (see Subsection 4.1.1). However, the heuristic knowledge for H_i is much more elaborate than the one used for the urgency value U_i. In the Go program MANGO, H_i is composed of three elements: (i) a capture-escape value, (ii) a pattern value, and (iii) the proximity to the last moves.

The *capture-escape value* of each move depends on (1) the number of stones that could be captured by playing the move, and on (2) the number of stones that could escape a capture by playing the move. It is calculated the same way as for the simulation strategy.

The *pattern value* is learnt offline by using the pattern matching described by Bouzy and Chaslot (2005). This pattern matching was also implemented in the Go program INDIGO, and improved its level significantly.[3] In this research, each pattern assigns a value to the move that is in its centre. The value of each pattern is the probability that the move is played in a professional game. The learning phase has been performed on 2,000 professional games; 89,119 patterns were learnt. Each pattern contained between 0 stones (e.g., corner pattern) and 15 stones (e.g., joseki pattern). In contrast to the 3×3 patterns used in the simulation strategy (see Subsection 4.1.1), the size of the patterns for the progressive strategies is not bounded. Some patterns cover nearly the whole board, and some cover only

[3] INDIGO was third out of 17 participants in the World Computer Go Championship 2006, see http://computer-go.softopia.or.jp/gifu2006/English/index.html.

a few intersections. Computing these unbounded patterns takes on average 100 times more CPU time than for the 3×3 patterns.

The *proximity coefficients* are computed as the Euclidean distances to all moves previously played, as shown in Formula 5.2.

These elements are combined in the following formula to compute H_i:

$$H_i = (V_{ce}(i) + V_p(i)) \times \sum_k \frac{1}{(2d_{k,i})^{\alpha_k}} \tag{5.2}$$

where $V_{ce}(i)$ is the capture-escape value, $V_p(i)$ is the pattern value, $d_{k,i}$ is the (Euclidean) distance to the k^{th} last move, and $\alpha_k = 1.25 + \frac{k}{2}$. This formula has been tuned experimentally.

The knowledge utilized in MANGO has a prediction rate of 23%, i.e., 23% of the time the move with the highest H_i score is also the move played by a professional player. This result is of the same order as the prediction rate (i.e., 25%) obtained by Van der Werf *et al.* (2006), but lower than the one later obtained (i.e., 35%) by Coulom (2007).

Time Available for Knowledge

The time consumed to compute H_i is in the order of one millisecond, which is around 1,000 times slower than playing a move in a simulated game. To avoid a speed reduction in the program's performance, we compute H_i only once per node, when a certain threshold of games has been played through this node. The threshold was set to $T = 30$ in MANGO. With this setting, the speed of the program was only reduced by 4%. The speed reduction is low because the number of nodes that have been visited more than 30 times is low compared to the number of moves played in the simulated games. It can be seen in Figure 5.2 that the number of calls to the domain knowledge is reduced quickly as T increases. Even for $T = 9$, the number of calls to the domain knowledge is quite low compared to the number of simulated moves. The number of nodes having a certain visit count is plotted in Figure 5.3. The data has been obtained from a 19×19 initial position by performing a 30-seconds MCTS. We have also plotted a trend line that shows that this data can be approximated by a power law.

Progressive Widening in Mango

In MANGO, the number of children that were not pruned in the beginning, k_{init}, was 5. Next, k nodes were unpruned when the number of simulations in the parent surpassed $A \times B^{k-k_{init}}$ simulated games. A was set experimentally to 50 and B to 1.3.

5.2.2 MANGO vs. GNU GO

In the first series of experiments we tested the two progressive strategies in games against GNU GO version 3.6. The experiments were performed on the 9×9, 13×13, and 19×19 boards. Our program used $20,000$ simulations per move. It takes on average less than one second on a 9×9 board, two seconds on a 13×13 board, and five seconds on a 19×19 board. The level of GNU GO has been set to 10 on the 9×9 and 13×13 boards, and to 0

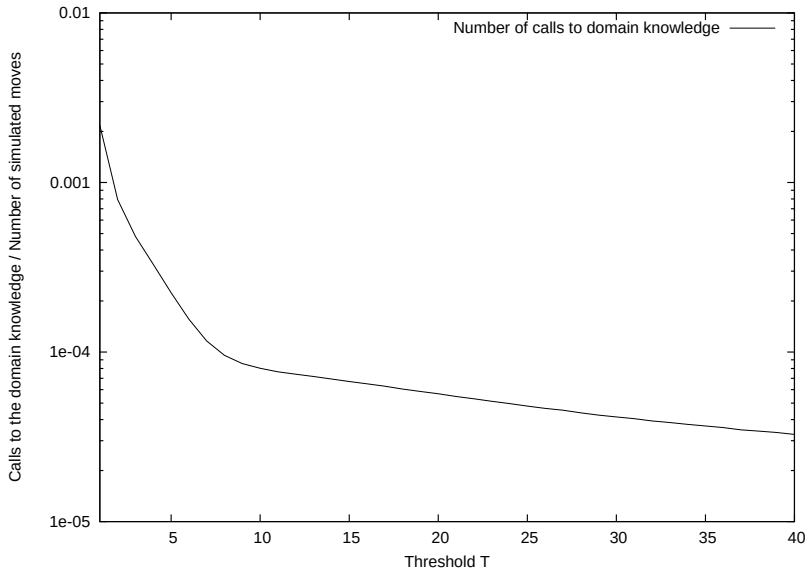


Figure 5.2: Number of calls to the domain knowledge relative to the number of simulated moves, as a function of the threshold T.

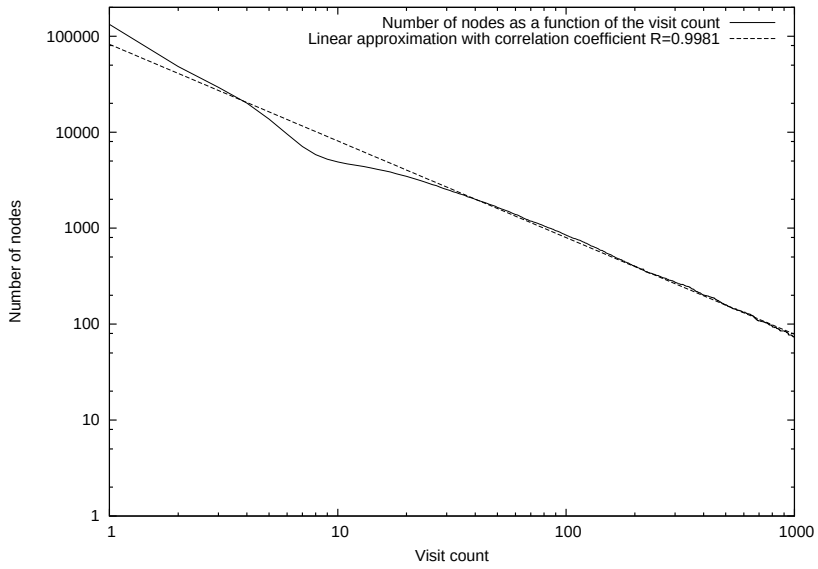


Figure 5.3: Number of nodes with a given visit count.

on the 19×19 board. The results are reported in Table 5.1, where PB stands for progressive bias and PW for progressive widening.

From these experiments, the results, and our observation, we may arrive at three conclusions. First, the plain MCTS framework does not scale well to the 13×13 board and the 19×19 board, even by using GNU GO at level 0. Second, the progressive strategies increase MANGO's level of play on every board size. Third, on the 19×19 board size the combination of both strategies is much stronger than each strategy applied separately.

Table 5.1: Results of MANGO against GNU GO.

Board size	Simulations per move	GNU GO's level	PB	PW	Wins	Games	95 percent conf. int.
9	20,000	10			33.2%	1000	3.0%
9	20,000	10		X	37.2%	1000	3.0%
9	20,000	10	X		58.3%	1000	3.1%
9	20,000	10	X	X	61.7%	2000	2.2%
13	20,000	10			8.5%	500	2.5%
13	20,000	10		X	15.6%	500	3.2%
13	20,000	10	X		30.0%	500	4.1%
13	20,000	10	X	X	35.1%	500	4.2%
19	20,000	0			0%	200	1.0%
19	20,000	0		X	3.1%	200	2.5%
19	20,000	0	X		4.8%	200	3.0%
19	20,000	0	X	X	48.2%	500	4.4%

5.2.3 Self-Play Experiment

We also performed self-play experiments on the different board sizes. The time setting of these experiments was 10 seconds per move. On the 9×9 board, MANGO using both progressive strategies won 88% of 200 games played against MANGO without progressive strategies. Next, on the 13×13 board, MANGO using both progressive strategies won 81% of 500 games played against MANGO without progressive strategies. Finally, on the 19×19 board, MANGO using both progressive strategies won all the 300 games played against MANGO without progressive strategies. These self-play experiments show that the effect of progressive strategies is larger on the 19×19 board than on the 13×13 and 9×9 boards. This conclusion is consistent with the results of the experiments of the previous subsection.

5.2.4 Tournaments

In the last series of experiments we tested MANGO's strength by competing in computer tournaments. Table 5.2 presents the results by MANGO in the tournaments entered in 2007. In all these tournaments, MANGO used both progressive strategies. In this table, KGS stands for "KGS Go Server". This server is the most popular one for computer programmers, and

most of the well-known programs have participated in one or more editions (e.g., MoGo, Crazy Stone, Go++, The Many Faces Of Go, GNU Go, Indigo, Aya, Dariush, etc...).

As shown in the previous experiments, the progressive strategies are the main strength of Mango. We remark that Mango was always in the best half of the participants.

Table 5.2: Results by Mango in 2007 tournaments.

Tournament	Board Size	Participants	Mango's rank
KGS January 2007	13×13	10	2^{nd}
KGS March 2007	19×19	12	4^{th}
KGS April 2007	13×13	10	3^{rd}
KGS May 2007	13×13	7	2^{nd}
12^{th} Computer Olympiad	9×9	10	5^{th}
12^{th} Computer Olympiad	19×19	8	4^{th}
KGS July 2007	13×13	10	4^{th}

5.3 Progressive Strategies in Other Game Programs

In the previous section we tested the progressive strategies in the Go program Mango. In this section we show that they are program- and game-independent. First, we discuss the application of progressive strategies in Crazy Stone and MoGo in Subsections 5.3.1 and 5.3.2. Next, in Subsection 5.3.3 we test progressive bias in the game of Lines of Action.

5.3.1 Progressive Strategies in Crazy Stone

Progressive widening was independently invented by Coulom (2007) and applied in his program Crazy Stone. The heuristic knowledge used for progressive widening consists mainly of pattern features, similar to those developed for Indigo and Mango (Bouzy and Chaslot, 2005). In Coulom's implementation the n^{th} move is unpruned when t_{n-1} simulations have been run, with $t_0 = 0$ and $t_{n+1} = t_n + 40 \times 1.4^n$. Results against GNU Go indicate that progressive widening brings a significant improvement to the playing strength on the 9×9 board (i.e., winning 90.6% of the games instead of 68.2%). On the 19×19 board the contribution of progressive widening to the playing strength is impressive (i.e., winning 37.5% of the games against GNU Go instead of 0%) .

The main difference with our implementation is that the speed of unpruning as implemented in his program is slower than the one used in Mango. This implies that the quality of the heuristic domain knowledge in his program is higher. Therefore it can afford to be more selective without pruning important moves.

5.3.2 Progressive Strategies in MoGo

In MoGo, progressive widening was not implemented because another enhancement already played a similar role: Rapid Action-Value Estimation (RAVE) (cf. Subsection 5.4.2).

Progressive bias has been implemented in MoGo, but was adapted. Instead of having a bias of $\frac{H_i}{n_i+1}$, the progressive bias that gave the best results was $\frac{H_i}{ln(n_i+2)}$ (Chaslot *et al.*, 2010).

Self-play experiments were performed by using MoGo on the 19×19 Go board with a thinking time of 1 second per move. The version of MoGo using progressive bias won $61.7\% \pm 3.1\%$ of the games against MoGo without progressive bias.

5.3.3 Progressive Strategies in MC-LOA

In the previous subsections, the progressive strategies were implemented and tested in Go programs. However, progressive strategies can be used in other game domains as well. The MCTS Lines-of-Action (LOA) program MC-LOA uses progressive bias to embed domain-knowledge bias into the UCT formula (Winands *et al.*, 2008; Winands and Björnsson, 2010). The UCT formula is modified as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects a child k of the node p that satisfies Formula 5.3:

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} + \frac{W \times P_{mc}}{n_i + 1} \right) \quad , \tag{5.3}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which must be tuned experimentally (here C = 0.6). $\frac{W \times P_{mc}}{n_i+1}$ is the progressive-bias part of the formula. *W* is a constant, which must be set manually (here $W = 50$). P_{mc} is the *transition probability* of a move category *mc* (Tsuruoka *et al.*, 2002).

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

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \quad , \tag{5.4}$$

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

The move categories of MC-LOA (Winands *et al.*, 2008; Winands and Björnsson, 2010) are similar to the ones used in the Realization-Probability Search of the program MIA (Winands and Björnsson, 2008). They are applied in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim, and the central 2×2 board. Finally, moves are further classified based on the number of squares travelled away from or towards the centre-of-mass.

In the following series of experiments we quantified the performance of progressive bias in a round-robin tournament. The progressive-bias variant of the MCTS program (MC-LOA + PB) played against the default MCTS program (MC-LOA). Moreover, MC-LOA and

MC-LOA + PB were matched against MIA. The latter performs an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced-Realization-Probability-Search framework (Winands and Björnsson, 2008). The program uses state-of-the-art $\alpha\beta$ enhancements (Winands, 2004). The thinking time was limited to 5 seconds per move. Each match data point represents the result of 20,000 games, with both colours played equally. A standardized set of 100 three-ply starting positions (Billings and Björnsson, 2003) was used, with a small random factor in the evaluation function preventing games from being repeated. All experiments were performed on an AMD Opteron 2.2 GHz computer. The results are given in Table 5.3.

Table 5.3: Tournament results.

Strategy	MC-LOA	MC-LOA+PB	MIA
MC-LOA	-	25.8%	28.8%
MC-LOA+PB	74.2%	-	46.6%
MIA	71.2%	53.4 %	-

Table 5.3 reveals that MC-LOA using progressive bias outplayed the default MC-LOA with a winning score of almost 75% of the available points. Moreover, MC-LOA with progressive bias played much better against MIA than MC-LOA did (i.e., winning 46.6% of the games instead of 28.8%). This result shows that the progressive bias improves the playing strength of the Monte-Carlo LOA program. Finally, we remark that adding progressive widening did not improve the program. This is not surprising. The results of Table 5.1 indicate that payoff of adding progressive widening on top of progressive bias increases when the board size / branching factor increases. For 9×9 Go the additional payoff of progressive widening was small. The fact that that LOA has a smaller branching factor than 9×9 Go (i.e., 30 vs. 40), explains why progressive widening did not work.

5.4 Related Research

In this section we present two methods, prior knowledge and Rapid Action-Value Estimation (RAVE). They were proposed simultaneously with the progressive strategies discussed above, and have a similar purpose. In Subsection 5.4.1 we explain prior knowledge. Next, we discuss RAVE in Subsection 5.4.2.

5.4.1 Prior Knowledge

An alternative enhancement to progressive bias has been proposed by Gelly and Silver (2007). It consists of introducing prior knowledge. The selected node k is the one, which satisfies Formula 5.5:

$$k \in argmax_{i \in I} \left(\frac{v_i \cdot n_i + n_{prior} \cdot Q_i}{n_i + n_{prior}} + C \times \sqrt{\frac{\ln n_p}{n_i + n_{prior}}} \right) \tag{5.5}$$

where Q_i is the prior estimation of the position. Gelly and Silver (2007) use a reinforcement-learning algorithm (Silver, Sutton, and Müller, 2007), which learnt the values from self-play

on the 9×9 board. n_{prior} is a coefficient that has to be tuned experimentally.

On the 9×9 board, this technique successfully increased MOGO's winning percentage against GNU GO from 60% to 69%. However, learning the prior value Q_i was only done for the 9×9 board. So, the scalability of this approach to larger board sizes is an open question.

5.4.2 Rapid Action-Value Estimation

Brügmann (1993) proposed to acquire results from simulations quicker by the "all-move-as-first heuristic" (AMAF). AMAF, for a given position p, assigns each move m with a value $\text{AMAF}_{p,m}$. This value is computed in the following way, which considers each move of the simulated game as important as the first move. For every simulated game S_i played starting from p, in which the move m has been played, we note $S_i(p,m) = 1$ if the player who played m won the simulated game, and $S_i(p,m) = 0$ if the player who played m lost the simulated game. The AMAF value is then the average over i of the $S_i(p,m)$ values. The AMAF values can be computed incrementally.

Gelly and Silver (2007) proposed to use the AMAF value in combination with MCTS. It replaces UCT (see Subsection 3.3.1) by Rapid Action-Value Estimator (RAVE). The selected node k has to satisfy Formula 5.6:

$$k \in argmax_{i \in I} \left((1 - \beta(n_p)) \times (v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}) + \beta(n_p) \times AMAF_{p,i} \right) \quad (5.6)$$

In Formula 5.6, p is the position associated with the current node, and β is a coefficient. Gelly and Silver (2007) proposed $\beta(N) = \sqrt{\frac{k}{3N+k}}$. This formula led to good results for all values of k from 3 to 10,000, with the best results obtained with $k = 1000$.

Silver (2008) proposed an enhancement of the RAVE formula in order to estimate bias and variance to calculate the best combination of UCT and RAVE values. This version uses $\beta(m_i, n_i) = \frac{m_i}{m_i + n_i + D \times m_i \times n_i}$, where m_i is the number of simulations on which the AMAF value of the move i is based, and D is a constant to be tuned.

A disadvantage of RAVE is that in principle it has to keep track of the AMAF values in a separate table for every node. Keeping track of the AMAF values globally instead of locally at every node could solve the memory problem, but has the risk that it diminishes the benefit of RAVE.

5.5 Chapter Conclusions and Future Research

In this chapter we introduced two progressive strategies. These strategies use (1) the information available for the selection strategy, and (2) some (possibly time-expensive) domain knowledge that is introduced by an expert, or learnt automatically. The two progressive strategies we developed are *progressive bias* and *progressive widening*. Progressive bias uses knowledge to direct the search. Progressive widening first reduces the branching factor, and then increases it gradually. This scheme is also dependent on knowledge.

Based on the results of the experiments performed with our program MANGO, we may offer four conclusions. (1) The plain MCTS method does not scale well to 13×13 Go, and

performs even worse in 19×19 Go. MCTS is weak at handling large branching factors. (2) Progressive strategies, which focus initially on a small number of moves, correct this problem. They increased the level of play of the program MANGO significantly, for every board size. (3) On the 19×19 board, the combination of both strategies is much stronger than each strategy applied separately. The fact that progressive bias and progressive widening work better in combination with each other shows that they have complementary roles in MCTS. This is especially the case when the board size and therefore branching factor grows. (4) Progressive strategies can use relatively expensive domain knowledge with hardly any speed reduction.

The progressive strategies were successfully implemented in other game programs or domains. Progressive bias increased the playing strength of MOGO and MC-LOA, while progressive widening did the same for CRAZY STONE. These results give rise to a fifth conclusion that the proposed progressive strategies are essential enhancements for an MCTS program.

A direction for future research is to modify during game play (online) the progressive strategy according to the results of the Monte-Carlo simulations. For instance, in a situation where the initial *k* moves are losing, increasing the speed of widening to find the best move seems promising. As another example, if a move that receives always a high heuristic score H_i is rarely the best move in numerous nodes, then the heuristic score of this move could be adjusted online.

Chapter 6

Optimizing Search Parameters using the Cross-Entropy Method

This chapter is based on the publication:

> G.M.J-B. Chaslot, M.H.M. Winands, I. Szita, and H.J. van den Herik (2008b). Cross-Entropy for Monte-Carlo Tree Search. *ICGA Journal*, Vol. 31, No. 3., pp. 145-156.

We have seen in Chapters 3, 4, and 5 that MCTS is controlled by several parameters, which define the behaviour of the search. Especially the selection and simulation strategies contain several important parameters. These parameters have to be optimized in order to get the best performance out of an MCTS program. In this chapter we answer the third research question how to optimize the parameters of an MCTS program. We propose to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM) (Rubinstein, 1999). The method is related to Estimation-of-Distribution Algorithms (EDAs) (Muehlenbein, 1997), a new area of evolutionary computation. We test CEM by tuning the main parameters in the Go program MANGO.

The chapter is organized as follows. We briefly discuss parameter optimization in Section 6.1. Next, in Section 6.2 we present the MCTS parameters used in MANGO. In Section 6.3 we explain CEM. We empirically evaluate CEM in combination with MCTS in Section 6.4. Finally, Section 6.5 provides the chapter conclusions and describes future research.

6.1 Parameter Optimization

Most game engines have a large number of parameters that are crucial for their performance. Optimizing these parameters by hand may be a hard and time-consuming task. Although it is possible to make educated guesses for some parameters, for other parameters it is beyond imagination. Here, a learning method can be used to find the best values for these parameters (Sutton and Barto, 1998; Beal and Smith, 2000).

Using learning methods for optimizing the parameters in order to increase the playing strength of a game program is difficult. The problem is that the fitness function[1] is rarely available analytically. Therefore, learning methods that rely on the availability of an analytic expression for the gradient cannot be used. However, there are several ways to optimize parameters despite the lack of an analytic gradient. An important class of such algorithms is represented by temporal-difference (TD) methods that have been used successfully in tuning evaluation-function parameters in Backgammon, Chess, Checkers, and LOA (Tesauro, 1995; Baxter, Tridgell, and Weaver, 1998; Schaeffer, Hlynka, and Jussila, 2001; Winands *et al.*, 2002). Obviously, any general-purpose gradient-free learning method can be used for parameter tuning in games. Just to mention two other examples, Björnsson and Marsland (2003) successfully applied an algorithm similar to Finite-Difference Stochastic Approximations (FDSA) to tune the search-extension parameters of CRAFTY. Kocsis, Szepesvári, and Winands (2006) investigated the use of RSPSA (Resilient Simultaneous Perturbation Stochastic Approximation), a stochastic hill-climbing algorithm, for the games of Poker and LOA.

In this chapter we investigate the use of the *Cross-Entropy Method* (CEM) (Rubinstein, 1999) for optimizing the parameters in MCTS. CEM is related to the Estimation-of-Distribution Algorithms (EDAs) (see Muehlenbein, 1997), which constitute a new area of evolutionary computation. Similar to EDA, CEM maintains a *probability distribution* over possible solutions. From this distribution, solution candidates are drawn. By using the idea of *Distribution Focusing*, CEM is turned into a highly effective learning method (Rubinstein, 1999).

MCTS is a relatively new method and when compared to the traditional $\alpha\beta$ (Knuth and Moore, 1975), it is less understood. Parameter optimization for MCTS is therefore a challenging task, making it an appropriate test domain for CEM. In the next section we discuss the search parameters used in the MCTS program MANGO.

6.2 MCTS Parameters in Mango

We noticed that the parameters that have the most effect on the level of play of the program are the ones of the selection and the simulation strategies. These parameters were explained in detail in Chapters 3, 4, and 5. In order to make this chapter readable independently, we describe the parameters of MANGO for selection (Subsection 6.2.1), simulation (Subsection 6.2.2), and progressive strategies (Subsection 6.2.3).

6.2.1 Selection Parameters

MANGO uses the selection strategy UCT (Kocsis and Szepesvári, 2006), as described in Chapter 3. UCT works as follows. Let I be the set of nodes reachable from the current node p. UCT selects a child k of node p that satisfies Formula 6.1:

$$k \in \arg\max_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{6.1}$$

[1]The fitness function is associated to a learning task and determines how good a solution is; for instance, in games it may be the percentage of won games.

where v_i is the value of the node i, n_i is the visit count of node i, and n_p is the visit count of node p. C is the exploration coefficient, which will be tuned using the Cross-Entropy Method (CEM). Finally, we note that in MANGO the selection strategy is only applied when a certain number T of simulations have been performed. This coefficient T will also be optimized using CEM.

6.2.2 Simulation Parameters

The simulation strategy of MANGO uses (1) a capture-escape value, (2) a pattern value, and (3) a proximity factor. We discuss them below. Let $\mathcal{M}$ be the set of all possible moves for a given position. Each move $j \in \mathcal{M}$ is given an urgency U_j. The simulation strategy selects one move from $\mathcal{M}$. The probability of each move to be selected is $p_j = \frac{U_j}{\sum_{k \in \mathcal{M}} U_k}$. The urgency U is the sum of two values: the capture-escape value and the pattern value, which is multiplied by the proximity factor P_{md} based on the Manhattan distance. So, $U_j = (V_{ce} + V_p) \times P_{md}$. We explain the three concepts below.

1. *Capture-escape value.* The value V_{ce} is given to moves capturing stones and/or escaping captures. It equals a coefficient $P_c \times$ the number of captured stones plus a coefficient $P_e \times$ the number of stones escaping to be captured. Using a capture-escape value was first proposed by Bouzy (2005), and later improved successively by Coulom (2006) and Cazenave (2007a).

2. *Pattern value.* For each possible 3×3 pattern, the value of the central move has been learnt by a dedicated algorithm described in Bouzy and Chaslot (2006). The weight of a pattern is raised to the power of a certain exponent P_p. When P_p is set to a small value, all patterns will be selected with a nearly-uniform probability. When P_p is set to a large value, only the best patterns will be played in the simulation phase.

3. *Proximity.* Moves within a Manhattan distance of 1 from the previous move have their urgency multiplied by a proximity factor P_{md}. This idea is adapted from the sequence-like simulation strategy developed by Gelly *et al.* (2006).

Summarizing, CEM will optimize the parameters P_c, P_e, P_p, and P_{md}.

6.2.3 Progressive Parameters

In MANGO we use two enhancements, *progressive bias* and *progressive widening* (Chaslot *et al.*, 2008c; Coulom, 2007), which were discussed in Chapter 5. Both enhancements significantly increase MANGO's playing strength. This happened after tuning the parameters by trial and error.

Progressive Bias

Progressive bias modifies the UCT formula in such a way that it favours moves that are regarded as "good" by some heuristic knowledge. In MANGO, the heuristic knowledge takes into account capturing, escaping captures, and large patterns. More details can be found in Chapter 5. Instead of selecting the node that satisfies Formula 6.1, we select the node k that satisfies Formula 6.2.

$$k \in \arg\max_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \frac{PB_f \times H_i^{PB_e}}{n_i + 1} \right) \tag{6.2}$$

Here H_i represents the heuristic knowledge. The coefficients PB_f and the PB_e will again be optimized using CEM. PB_f stands for the *progressive-bias factor* and PB_e for the *progressive-bias exponent* (see Chapter 5).

Progressive Widening

Progressive widening consists of (1) reducing the branching factor artificially when the selection function is applied, and (2) increasing it progressively as more time becomes available. When the number of games that visit a node p (n_p) equals a threshold T, progressive widening "prunes" most of the children. Initially, only the k_{init} children with the highest heuristic values in the sequence are not pruned. Next, the children of a node i are progressively added. In MANGO, it happens as follows. The k^{th} child node is unpruned when the number of simulations in i surpasses $A \times B^{k-k_{init}}$ simulations. A, B, and k_{init} will be optimized by using CEM.

6.3 The Cross-Entropy Method

In this section we explain the Cross-Entropy Method. First, we give an informal description (Subsection 6.3.1). Subsequently, we clarify the method in detail (Subsection 6.3.2). Finally, we discuss the normalization of parameters in Subsection 6.3.3.

6.3.1 Informal Description of the Cross-Entropy Method

The *Cross-Entropy Method* (CEM) (Rubinstein, 1999) is a population-based learning algorithm, where members of the population are sampled from a parameterized probability distribution. In each generation, the parameters of the distribution are updated so that its *cross-entropy distance* from a desired distribution is minimized.

CEM aims to find the (approximate) optimal solution $\mathbf{x}^*$ for a learning task described in the following form

$$\mathbf{x}^* \leftarrow \arg\max_{\mathbf{x}} f(\mathbf{x}). \tag{6.3}$$

We remark that $\mathbf{x}^*$ is a vector containing all parameters to be optimized. Moreover, f is a fitness function (which determines how good a solution is; for instance in games, it is the percentage of won games), and $\mathbf{x} \in X$ where X is some (possibly high-dimensional) parameter space. Most traditional learning algorithms maintain a single candidate solution $\mathbf{x}(t)$ in each time step. In contrast, CEM maintains a *distribution* over possible solutions (similar to evolutionary methods). From that distribution, solution candidates are drawn at random. This is essentially *random guessing*, but by the idea of *Distribution Focusing* it is turned into a highly effective learning method. We explain both concepts below.

Random Guessing

Random guessing is a quite simple 'learning' method: we draw many samples from a distribution g belonging to a family of parameterized distributions $\mathcal{G}$ (e.g., Gaussian, Binomial, Bernoulli, etc.), then select the best sample as an estimation of the optimum. In the extreme case of drawing infinitely many samples, random guessing finds the global optimum.

The efficiency of random guessing depends largely on the distribution g from which the samples are drawn. For example, if g is sharply peaked in the neighbourhood of the optimal solution $\mathbf{x}^*$, then a few samples may be sufficient to obtain a good estimate. In contrast, if the distribution is sharply peaked around a vector $\mathbf{x}$, which is far away from the optimal solution $\mathbf{x}^*$, a large number of samples is needed to obtain a good estimate of the global optimum.

Distribution Focusing

We can improve the efficiency of random guessing by the idea of *Distribution Focusing*. After drawing a moderate number of samples from distribution g, we may not be able to give an acceptable approximation of $\mathbf{x}^*$, but we may still obtain a *better sampling distribution*. The basic idea of CEM is that it selects the best samples, and modifies g so that it becomes more peaked around the best samples. Distribution Focusing is the central idea of CEM (Rubinstein, 1999).

Let us consider an example, where $\mathbf{x}$ is an m-dimensional vector and g is a Gaussian distribution for each coordinate. Assume that we have drawn 1000 samples and selected the 10 best. If the i^{th} coordinate of the best-scoring samples has an average of μ_i, then we may hope that the i^{th} coordinate of $\mathbf{x}^*$ is also close to μ_i, so we may shift the distribution's centre towards μ_i. In the next subsection, we describe the update rule of CEM in a more formal way.

6.3.2 Formal Description of the Cross-Entropy Method

In this subsection we will choose g from a family of parameterized distributions (e.g., Gaussian, Binomial, Bernoulli, etc.), denoted by $\mathcal{G}$, and describe an algorithm that iteratively improves the parameters of this distribution g.

Let N be the number of samples to be drawn, and let the samples $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ be drawn independently from distribution g. For each $\gamma \in \mathbb{R}$, the set of high-valued samples,

$$\hat{L}_\gamma \leftarrow \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma, 1 \leq i \leq N\}, \tag{6.4}$$

provides an approximation to the level set

$$L_\gamma \leftarrow \{\mathbf{x} \mid f(\mathbf{x}) \geq \gamma\}. \tag{6.5}$$

Let U_γ be the uniform distribution over the level set L_γ. For large values of γ, this distribution will peak around $\mathbf{x}^*$, so it would be suitable for random sampling. This approximation procedure raises two potential problems, which are discussed below. The first problem is solved by *elite samples* and the second problem by the *cross-entropy distance*.

Elite Samples

The first problem is that for (too) large γ values $\hat{L}_\gamma$ will only contain a few samples (possibly none), making learning impossible. This problem could be easily solved by choosing lower values for γ. However, setting γ too low causes a slow convergence to a (sub)optimal solution. Therefore, the following alternative is used: CEM chooses a ratio $\rho \in [0, 1]$ and adjusts $\hat{L}_\gamma$ to be the set of the best $\rho \cdot N$ samples. This corresponds to setting $\gamma \leftarrow f(\mathbf{x}^{(\rho \cdot N)})$, provided that the samples are arranged in decreasing order of their values. The best $\rho \cdot N$ samples are called the *elite samples*. In practice, ρ is typically chosen from the range $[0.01, 0.1]$.

Cross-Entropy Distance

The second problem is that the distribution of the level set L_γ is not a member of any kind of parameterized distribution family and therefore it cannot be modelled accordingly. This problem is solved by changing the approximation goal: CEM chooses the distribution g *from the distribution family* $\mathcal{G}$ that approximates the empirical distribution over $\hat{L}_\gamma$ best. The best g is found by minimizing the distance between $\mathcal{G}$ and the uniform distribution over the elite samples. The distance measure is the *cross-entropy distance* (also called the Kullback-Leibler divergence (Kullback, 1959)). The cross-entropy distance of two distributions g and h is defined as

$$D_{CE}(g||h) = \int g(\mathbf{x}) \ln \frac{g(\mathbf{x})}{h(\mathbf{x})} \, \mathrm{d}\mathbf{x}. \tag{6.6}$$

It is known that under mild regularity conditions, CEM converges with probability 1 (Costa, Jones, and Kroese, 2007). Furthermore, for a sufficiently large population, the global optimum is found with a high probability.

For many parameterized distribution families, the parameters of the minimum cross-entropy member can be computed easily from simple statistics of the elite samples. Below we sketch the special case when x is sampled from a Gaussian distribution. This distribution is used in the remainder of this chapter.

Let the domain of learning be $D = \mathbb{R}^m$, and each component be drawn from independent Gaussian distributions with parameters μ_j, σ_j^2, $1 \leq j \leq m$, that is, a distribution $g \in \mathcal{G}$ is parameterized with $2m$ parameters.

After drawing N samples $\mathbf{x}^{(1)}, \ldots \mathbf{x}^{(N)}$ and having a threshold value γ, let E denote the set of elite samples, i.e.,

$$E \leftarrow \{\mathbf{x}^{(i)} \mid f(\mathbf{x}^{(i)}) \geq \gamma\}. \tag{6.7}$$

With this notation, the distribution g' with minimum CE-distance from the uniform distribution over the elite set has parameters

$$\begin{aligned} \boldsymbol{\mu}' &\leftarrow (\mu'_1, \ldots, \mu'_m), \quad \text{where} \\ \mu'_j &\leftarrow \frac{\sum_{\mathbf{x}^{(i)} \in E} x_j^{(i)}}{\sum_{\mathbf{x}^{(i)} \in E} 1} = \frac{\sum_{\mathbf{x}^{(i)} \in E} x_j^{(i)}}{\rho \cdot N} \end{aligned} \tag{6.8}$$

and

$$\begin{aligned} \boldsymbol{\sigma^{2\prime}} &\leftarrow (\sigma^{2\prime}_1, \ldots, \sigma^{2\prime}_m), \quad \text{where} \\ \sigma^{2\prime}_j &\leftarrow \frac{\sum_{\mathbf{x}^{(i)} \in E} (x_j^{(i)} - \mu'_j)^T (x_j^{(i)} - \mu'_j)}{\sum_{\mathbf{x}^{(i)} \in E} 1} \\ &= \frac{\sum_{\mathbf{x}^{(i)} \in E} (x_j^{(i)} - \mu'_j)^T (x_j^{(i)} - \mu'_j)}{\rho \cdot N}. \end{aligned} \tag{6.9}$$

In other words, the parameters of g' are simply the componentwise empirical means and variances of the elite set. For the derivation of this rule, we refer to De Boer *et al.* (2005). Changing the distribution parameters from $(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ to $(\boldsymbol{\mu}', \boldsymbol{\sigma^{2\prime}})$ may be a too large step, so moving only a smaller step towards the new values (using step-size parameter α) is preferable. The resulting algorithm is summarized in Algorithm 6.1.

```
input: μ_0 = (μ_{0,1}, ..., μ_{0,m}) and σ_0^2 = (σ_{0,1}^2, ..., σ_{0,m}^2)
                                        % initial distribution parameters
input: N                                % population size
input: ρ                                % selection ratio
input: T                                % number of iterations
for t from 0 to T − 1,                  % CE iteration main loop
    for i from 1 to N,
        draw x^(i) from Gauss^m(μ_t, σ_t^2)   % draw N samples
        compute f_i := f(x^(i))         % evaluate them
    sort f_i-values in descending order
    γ_{t+1} := f_{ρ·N}                  % level set threshold
    E_{t+1} := {x^(i) | f(x^(i)) ≥ γ_{t+1}}   % get elite samples
    μ'_j := (Σ_{x^(i)∈E} x_j^(i))/(ρ·N)       % get parameters of nearest distribution
    σ^2'_j := (Σ_{x^(i)∈E} (x_j^(i) − μ'_j)^T (x_j^(i) − μ'_j))/(ρ·N)
    μ_{t+1,j} := α·μ'_j + (1 − α)·μ_{t,j}     % update with step size α
    σ^2_{t+1,j} := α·σ^2'_j + (1 − α)·σ_{t,j}
end loop
```

Algorithm 6.1: Pseudo-code of the Cross-Entropy Method for Gaussian distributions.

6.3.3 Normalizing Parameters

The value of each parameter x_i has to be selected from a range $[a_i; b_i]$. Due to the fact that the domain of a Gaussian distribution is unbounded, we sometimes have to throw away samples, which have one or more out-of-bound values. Theoretically, this does not cause any complications: we may assume that samples having out-of-bound values are not discarded, they are only given a large negative score. With this assumption, we are able to apply the above algorithm without changes.

Furthermore, we apply two transformations to the parameters. First, the parameters are transformed to a logarithmic scale. We illustrate the reason by mentioning the progressive

bias coefficient as an example. The progressive bias coefficient in MANGO has the following range $[0.1; 100]$. Without using a logarithmic scale, half of the values would be chosen in $[0.1; 50]$ and the other half in $[50; 100]$. Small values (say between 0.1 and 1), which could belong to the optimal solution, would be hardly drawn. Using a logarithmic scale, half of the values are picked in $[0.1; 3.16]$ and the other half in $[3.16; 100]$. Second, parameters that are only allowed to have integer values, are rounded off to the closest integer. Both transformations are part of the fitness function f.

6.4 Experiments

In this section we are going to apply CEM to optimize the MCTS parameters of MANGO. This is done by playing against GNU GO 3.7.10, level 0, on a 9×9 Go board. In each generation CEM draws 100 samples selecting the best 10 (*the elite*) samples.[2] A sample consists of playing a certain number of games for a CEM-generated parameter setting. So, the fitness function straightforwardly measures the winning rate for a batch of games. To obtain results rather quickly, MANGO only performs 3,000 simulations per move.

The section is organized as follows. An overview of the parameters together with their range is given in Subsection 6.4.1. Subsection 6.4.2 and 6.4.3 test the performance of a fixed and variable batch size, respectively. The best parameter setting against GNU GO is discussed in Subsection 6.4.4. The setting of MANGO (plus CEM) is compared against the old MANGO in four self-play experiments in Subsection 6.4.5.

6.4.1 Overview of MCTS Parameters

In total 11 parameters are optimized by CEM, 2 parameters for the selection, 4 parameters for the simulation, 2 parameters for progressive bias, and 3 parameters for progressive widening. The parameters under consideration together with their ranges are given in Table 6.1. The table shows that for most parameters the value range is quite wide. This is done to assure that the optimal parameter value can be found. Regarding the initial distribution for each parameter, the mean is computed as the average of the lower and upper bound. The standard deviation is computed as half of the difference between the lower and upper bound. We remark that they are computed in a logarithmic way, since the logarithmic values of the parameters are used by CEM.

6.4.2 Fixed Batch Size

In the following series of experiments we tested the learning performance of three batch sizes: 10, 50, and 500. The learning curves for the different batch sizes are depicted in Figure 6.1. The x-axis represents the *total* number of games played by the samples. The y-axis represents the average winning percentage against GNU GO of all the (100) samples in a generation.

A batch size of 10 games leads to an increase from a winning score of 30% to a winning score of more than 50% against GNU GO, after playing a total of 10,000 games. However, the performance increase stops after 20,000 games, due to uncertainty caused by the small

[2] The values have been chosen based on previous results (De Boer *et al.*, 2005).

Table 6.1: Parameters with their ranges.

Parameter type	Parameter name	Range
Selection	UCT exploration coefficient C	$[0.2; 2]$
	Threshold T	$[2; 20]$
Simulation	Capture value P_c	[100;20,000]
	Escape value P_e	[50;2,000]
	Pattern exponent P_p	$[0.25; 4]$
	Proximity factor P_{md}	$[10; 500]$
Progressive Bias	PB factor PB_f	$[0.1; 100]$
	PB exponent PB_e	$[0.25; 4]$
Progressive Widening	PW initial # nodes k_{init}	$[2; 10]$
	PW factor A	$[20; 100]$
	PW base B	$[1.01; 1.5]$

batch size. Subsequently, a batch size of 50 takes more than three times longer to achieve a score of 50%, but converges to a score of a little more than 60%. Finally, a batch size of 500 games is even slower, but the performance increase is steadier and the final score is therefore even better: 63.9%. With these settings, the results were obtained by using a cluster of 10 quad-core computers running for 3 days.

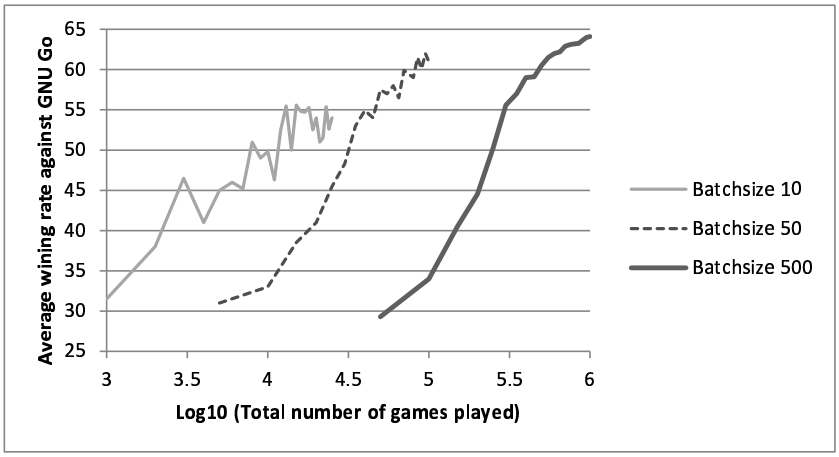


Figure 6.1: Learning curves for different fixed batch sizes.

6.4.3 Variable Batch Size

In the previous subsection we saw that learning with a small batch size quickly leads to a performance increase, but convergence is to a suboptimal score only. In contrast, learning with a large batch size is slower but it converges to a higher score. In order to benefit from

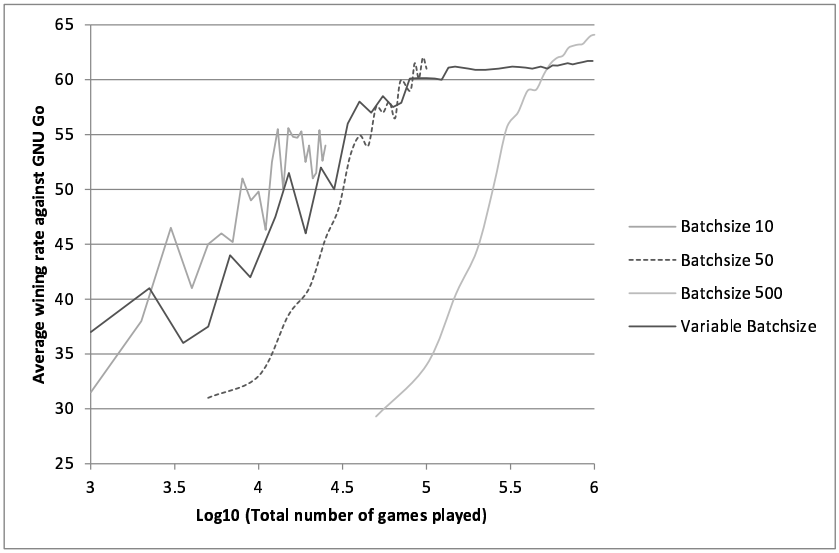


Figure 6.2: Learning curves for different fixed and variable batch sizes.

both approaches (quick increase, higher convergence), we propose to increase progressively the batch size after each generation. The scheme that we use is the following. At the first generation, the algorithm uses a batch size of 10. Next, at generation n, the algorithm uses a batch size of $10 \times 1.15^{n-1}$. The value of 1.15 has been chosen to ensure that, after 20 generations, the total number of games performed is the same as the number of games performed when using a batch size of 50. In the next series of experiments we compared the variable batch-size scheme with the three fixed batch sizes of the previous subsection. The results are depicted in Figure 6.2. The figure shows that a variable batch size performs a little bit worse than a fixed batch size of 50 or 500. However, the variable batch size converges faster than a fixed batch size of 50 or 500. These results suggest that a (sufficiently) large fixed batch size may be better than a variable batch size.

6.4.4 Comparison of Default and CEM Parameters

As we have seen in Subsection 6.4.2, after 20 generations the best parameter score was obtained by using a fixed batch size of 500. The achieved score of 63.9% is an underestimation because it represents the *average* score of *all* the samples at the 20^{th} generation. When updating the parameters by using the parameter means of the elite samples, the score may improve. This was tested in the following series of experiments. We compared the manual (default) parameter setting to the CEM parameter setting by playing twice (Default and CEM) 10,000 games against GNU GO. The parameter settings are reported in Table 6.2 together with their score against GNU GO.

We see that MANGO using the CEM parameters plays better against GNU GO than the default one (65.0% against 61.8%). We would like to remark that the parameters for

Table 6.2: Comparison of Default and CEM parameters.

Parameter	Default	CEM
UCT exploration coefficient C	0.7	0.43
Expansion threshold T	10	3
Capture value P_c	5,000	7,509
Escape value P_e	500	911
Pattern exponent P_p	1	0.7
Proximity factor P_{md}	150	85
PB factor PB_f	8	5.3
PB exponent PB_e	1	1.1
PW initial # nodes k_{init}	5	3
PW factor A	40	58
PW base B	1.2	1.12
Winning rate against GNU GO	61.8%	65.0%
Number of games	10,000	10,000
Standard deviation	0.5%	0.5%

the default version were already intensively optimized (but independently of each other). In Table 6.2, it can be seen that the values obtained by CEM are quite different from the default ones. However, most parameter modifications do not affect the level of the program much. We observed that the fitness landscape is quite flat around the optimum. For instance, modifying the capture value from 5,000 to 7,509 has almost no influence on the playing strength of the program.

6.4.5 Self-Play Experiment

As we saw in the previous subsection, the parameters were optimized by playing with a short time setting (3,000 simulations per move) against GNU GO. To check whether the parameters were not overfitted for a specific opponent or a specific time setting, four self-play experiments between MANGO with the CEM parameters and MANGO without the CEM parameters were executed. In the first experiment a short time setting of 3,000 simulations per move was used. MANGO with the CEM parameters won 55.4% of the 10,000 games played against the default version. In the second experiment a longer time setting of 200,000 simulations per move was used. The CEM version won 57.4% of the 1,000 games played. The third and fourth experiment were performed on two different board sizes, a 13×13 and a 19×19 Go board, respectively. The short time setting of 3,000 simulations per move was used. The CEM version won 61.3% of the games for 13×13 and 66.3% of the games for 19×19. The results suggest that the fine-tuning of parameters by CEM genuinely increased the playing strength of MANGO (see Table 6.3).

Table 6.3: Self-play experiments: CEM vs. Default.

Board size	Simulations per move	Winning rate of CEM	Number of games	Standard deviation
9×9	200,000	57.4%	1,000	1.6%
9×9	3,000	55.4%	10,000	0.5%
13×13	3,000	61.3%	10,000	0.5%
19×19	3,000	66.3%	10,000	0.5%

6.5 Chapter Conclusions and Future Research

In this chapter we proposed to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM). We tested CEM by optimizing 11 parameters of the MCTS program MANGO. Experiments revealed that using a batch size of 500 games gave the best results, although the convergence was slow. To be more precise, these results were obtained by using a cluster of 10 quad-core computers running for 3 days. Interestingly, a small (and fast) batch size of 10 still gave reasonable results when compared to the best one. A variable batch size performed a little bit worse than a fixed batch size of 50 or 500. However, the variable batch size converged faster than a fixed batch size of 50 or 500. Subsequently, we showed that MANGO with the CEM parameters performed better against GNU GO than the MANGO version without. Moreover, in four self-play experiments with different time settings and board sizes, the CEM version of MANGO defeated each time the default version convincingly. Based on these results, we may conclude that parameter optimization by CEM genuinely improved the playing strength of MANGO, for various time settings and board sizes. The nature of our research allows the following generalization: a hand-tuned MCTS-using game engine may improve its playing strength when re-tuning the parameters with CEM.

The idea of applying a gradient interpretation of the cross entropy in CEM (Hu and Hu, 2009) and the more general applicable idea of *adaptive noisy optimization* (Rolet and Teytaud, 2010) may improve the convergence speed for optimizing the parameters of an MCTS program. A direction of future research would be to test them in MANGO.

Chapter 7

Parallelizing Monte-Carlo Tree Search

This chapter is based on the publication:

> G.M.J-B. Chaslot, M.H.M. Winands, and H.J. van den Herik (2008a). Parallel Monte-Carlo Tree Search. *Proceedings of the Conference on Computers and Games 2008 (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands.), Vol. 5131 of *Lecture Notes in Computer Science (LNCS)*, pp. 60–71, Springer-Verlag, Heidelberg, Germany.

In the previous chapters we discussed Monte-Carlo Tree Search (MCTS) in detail. This chapter answers the fourth research question by investigating how we can parallelize MCTS. In the past, research in parallelizing search has been mainly performed in the area of $\alpha\beta$-based programs running on super-computers. DEEP BLUE (Campbell *et al.*, 2002) and BRUTUS/HYDRA (Donninger *et al.*, 2004) are famous examples of highly parallelized chess programs. The recent evolution of hardware has gone into the direction that nowadays even personal computers contain several cores. To get the most out of the available hardware one has to parallelize AI techniques. Parallelization has therefore become an important topic not only for MCTS, but for any kind of search technique.

Just as for $\alpha\beta$ search, it holds for MCTS that the more time is spent for selecting a move, the better the game play is. Moreover, the law of diminishing returns,[1] which nowadays has come into effect for many $\alpha\beta$ chess programs, appears to be less of an issue for MCTS Go programs. Hence, parallelizing MCTS seems to be a promising way to increase the strength of a Go program. Pioneering work has been done by Cazenave and Jouandeau (2007) who introduced two parallelization methods: leaf parallelization and root parallelization (originally called single-run parallelization).

In this chapter we compare them to a third method, that we call tree parallelization. We compare the three parallelization methods (leaf, root, and tree) by using the *Games-Per-*

[1] Experimental studies have demonstrated diminishing return with additional search depth (Junghanns and Schaeffer, 1997; Heinz, 2001).

Second (GPS)-speedup measure and *strength-speedup measure*. The first measure corresponds to the improvement in speed, and the second measure corresponds to the improvement in playing strength. The three parallelization methods are implemented and tested in the Go program MANGO, running on a multi-core machine containing 16 cores.

The chapter is organized as follows. In Section 7.1 we explain leaf parallelization and root parallelization. Next, we introduce tree parallelization in Section 7.2. We empirically evaluate the three parallelization algorithms in Section 7.3. Finally, Section 7.4 provides the chapter conclusions and describes future research.

7.1 Parallelization of Monte-Carlo Tree Search

In Chapter 3 we explained that MCTS consists of four main steps: (1) selection, (2) expansion, (3) simulation, and (4) backpropagation. The different parallelization methods can be distinguished for the MCTS step being parallelized.

In this chapter, we consider the parallelization of MCTS for a symmetric multiprocessor (SMP) computer. We always use one processor thread for each processor core. One of the properties of an SMP computer is that any thread can access the central (shared) memory with the same (generally low) latency. As a consequence, parallel threads should use a mutual exclusion (mutex) mechanism in order to prevent any data corruption, due to simultaneous memory access. This could happen when several threads are accessing the MCTS tree (i.e., in step 1, 2, or 4). However, the simulation step (i.e., step 3) does not require any information from the tree. There, simulated games can be played completely independently from each other. This specific property of MCTS is particularly interesting for the parallelization process. For instance, it implies that long simulated games make the parallelization easier. We distinguish three main types of parallelization, depending on which step of the MCTS is parallelized: *leaf parallelization* (Subsection 7.1.1), *root parallelization* (Subsection 7.1.2), and *tree parallelization* (elaborated in Section 7.2).

7.1.1 Leaf Parallelization

Leaf parallelization introduced by Cazenave and Jouandeau (2007) is one of the easiest ways to parallelize MCTS. To formulate it in machine-dependent terms, only one thread traverses the tree and adds one or more nodes to the tree when a leaf node is reached (step 1 and 2). Next, starting from the leaf node, independent simulated games are played for each available thread (step 3). When all games are finished, the result of all these simulated games is propagated backwards through the tree by one single thread (step 4). Leaf parallelization is depicted in Figure 7.1a.

Leaf parallelization seems interesting because its implementation is easy and does not require any mutexes. However, two problems appear. First, playing n games using n different threads takes more time on average than playing one single game using one thread, since the program needs to wait for the longest simulated game. Second, information is not shared. For instance, if 16 threads are available, and 8 (faster) finished games are all losses; it will be highly probable that most games will lead to a loss. Therefore, playing 8 more games is a waste of computational power. To decrease the waiting time, the program might stop the simulations that are still running when the results of the finished simula-

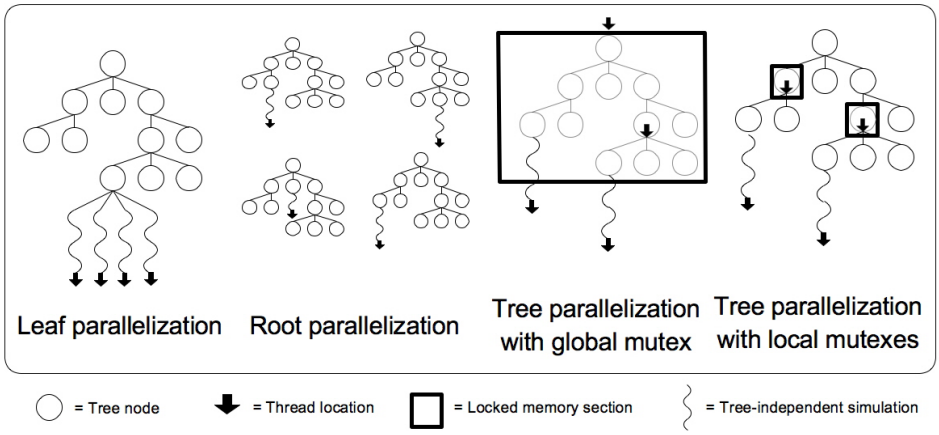


Figure 7.1: (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex and (d) with local mutexes.

tions become available. This strategy would enable the program to traverse the tree more often, but some threads would be idle. Leaf parallelization can be performed inside an SMP environment, or even on a cluster using MPI (Message Passing Interface) communication.

7.1.2 Root Parallelization

Cazenave and Jouandeau (2007) proposed a second parallelization under the name "single-run" parallelization. In this chapter we call it *root parallelization* to stress the part of the tree for which it applies. The method works as follows. It consists of building multiple MCTS trees in parallel, with one thread per tree. Similar to leaf parallelization, the threads do not share information with each other. When the available time is spent, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. The best move is selected based on this grand total. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. Root parallelization is depicted in Figure 7.1b.

7.2 Tree Parallelization

In this section we introduce a new parallelization method called *tree parallelization*. This method uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree; therefore mutexes are used to lock from time to time certain parts of the tree to prevent data corruption. There are two methods to improve the performance of tree parallelization: (1) mutex location (Subsection 7.2.1) and (2) "virtual loss" (Subsection 7.2.1).

7.2.1 Mutex Location

Based on the location of the mutexes in the tree, we distinguish two mutex location methods: (1) using a *global mutex* and (2) using several *local mutexes*.

The global-mutex method locks the whole tree in such a way that only *one* thread can access the search tree at a time (step 1, 2, and 4). In the meantime several other processes can play simulated games (step 3) starting from *different* leaf nodes. This is a major difference with leaf parallelization where all simulated games start from the *same* leaf node. The global-mutex method is depicted in Figure 7.1c. The potential speedup given by the parallelization is bounded by the time that has to be spent in the tree. Let x be the average percentage of time spent in the tree by one single thread. The maximum speedup in terms of games per second is $100/x$. In most MCTS programs x is relatively high (say between 25 to 50%), limiting the maximum speedup substantially. This is the main disadvantage of this method.

The local-mutexes method makes it possible that *several* threads can access the search tree simultaneously. To prevent data corruption because two (or more) threads access the same node, we lock a node by using a local mutex when it is visited by a thread. At the moment a thread departs the node, it is unlocked. Thus, this solution requires to frequently lock and unlock parts of the tree. Hence, fast-access mutexes such as spinlocks have to be used to increase the maximum speedup. The local-mutexes method is depicted in Figure 7.1d.

7.2.2 Virtual Loss

If several threads start from the root at the same time, it is possible that they traverse the tree for a large part in the same way. Simulated games might start from leaf nodes, which are in the neighbourhood of each other. It can even happen that simulated games begin from the same leaf node. Because a search tree typically has millions of nodes, it may be redundant to explore a rather small part of the tree several times. Coulom[2] suggests to assign a "virtual loss" when a node is visited by a thread (i.e., in step 1). Hence, the value of this node will be decreased. The next thread will only select the same node if its value remains better than its siblings' values. The virtual loss is removed when the thread that gave the virtual loss starts propagating the result of the finished simulated game (i.e., in step 4). Owing to this mechanism, nodes that are clearly better than others will still be explored by all threads, while nodes for which the value is uncertain will not be explored by more than one thread. Hence, this method keeps a certain balance between exploration and exploitation in a parallelized MCTS program.

7.3 Experiments

In this section we compare the different parallelization algorithms with each other. Subsection 7.3.1 discusses the experimental set-up. We show the performance of leaf parallelization, root parallelization, and tree parallelization in Subsection 7.3.2, 7.3.3, and 7.3.4,

[2] Personal Communication.

respectively. An overview of the results is given in Subsection 7.3.5. Root parallelization and tree parallelization are compared under different conditions in Subsection 7.3.6.

7.3.1 Experimental Set-up

The aim of the experiments is to measure the quality of the parallelization process. We use two measures to evaluate the speedup given by the different parallelization methods. The first measure is called the *Games-Per-Second (GPS) speedup*. It is computed by dividing the number of simulated games per second performed by the multithreaded program, by the number of games per second played by a single-threaded program. However, the GPS-speedup measure might be misleading, since it is not always the case that a faster program is stronger. Therefore, we propose a second measure: called *strength speedup*. It corresponds to the *increase of time needed to achieve the same strength*. For instance, a multithreaded program with a strength speedup of 8.5 has the same strength as a single-threaded program, which consumes 8.5 times more time.

In order to design the strength-speedup measurement, we proceed in three steps. First, we measure the strength of the Go program MANGO on the 13×13 board against GNU GO 3.7.10, level 0, for 1 second, 2 seconds, 4 seconds, 8 seconds, and 16 seconds. For each time setting, 2,000 games are played. Figure 7.2 reports the strength of MANGO in terms of percentage of victory. In Figure 7.3, the increase in strength in term of rating points as a function of the logarithmic time is shown. This function can be approximated accurately by linear regression, using a correlation coefficient $R^2 = 0.9922$. Second, the linear approximation is used to give a theoretical Go rating for any amount of time. Let us assume that E_t is the level of the program in rating points, T is the time in seconds per move. Linear regression gives us $E_t(T) = A \cdot log_2 T + B$ with $A = 56.7$ and $B = -175.2$. Third, the level of play of the multithreaded program is measured against the same version of GNU GO, with one second per move. Let E_m be the rating of this program against GNU GO. The strength speedup S is defined by: $S \in \mathbb{R} | E_t(S) = E_m$.

The experiments were performed on the supercomputer Huygens, which has 120 nodes, each with 16 cores POWER5 running at 1.9 GHz and having 64 Gigabytes of memory per node. Using this hardware the single-threaded version of MANGO was able to perform 3,400 games per second in the initial board position of 13×13 Go. The time setting used for the multithreaded program was 1 second per move.

7.3.2 Leaf Parallelization

In the first series of experiments we tested the performance of plain leaf parallelization. We did not use any kind of enhancement to improve this parallelization method as discussed in Subsection 7.1.1. The results regarding winning percentage, GPS speedup, and strength speedup for 1, 2, 4, and 16 threads are given in Table 7.1. We observed that the GPS speedup is quite low. For instance, when running 4 simulated games in parallel, finishing all of them took 1.15 times longer than finishing just 1 simulated game. For 16 threads, it took two times longer to finish all games compared to finishing just one. The results show that the strength speedup obtained is rather low as well (2.4 for 16 processors). So, we may conclude that plain leaf parallelization is not a good way for parallelizing MCTS.

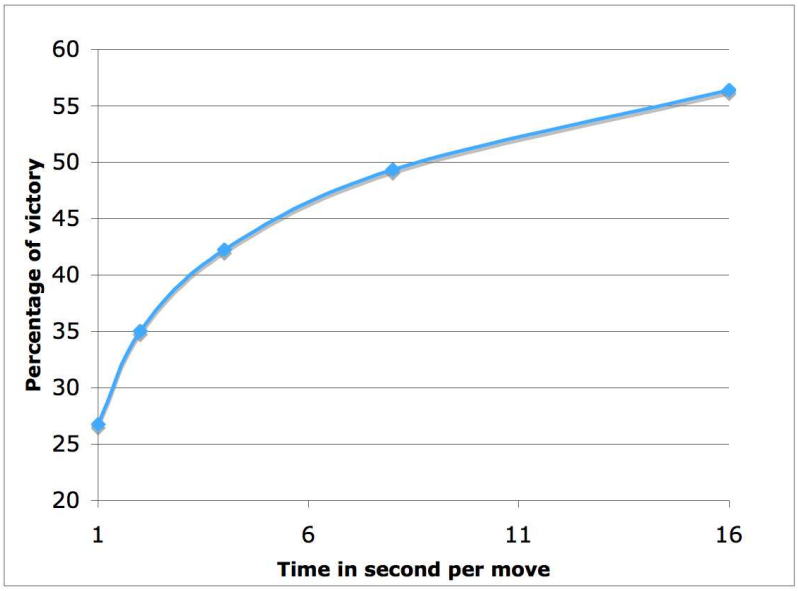


Figure 7.2: Scalability of the strength of MANGO with time.

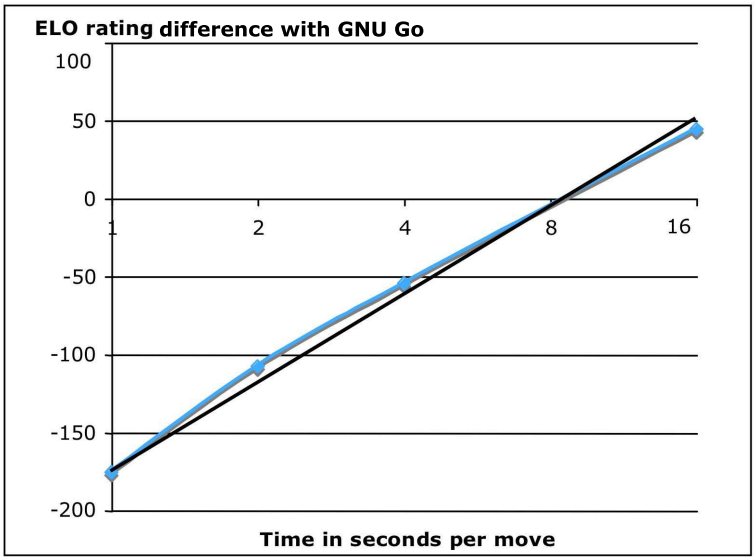


Figure 7.3: Scalability of the rating of MANGO vs. GNU GO with time. The curve represents the data points, and the line is a trend-line for this data.

Table 7.1: Leaf parallelization.

Number of threads	Winning percentage	Number of games	Confidence interval	GPS Speedup	Strength speedup
1	26.7%	2000	2.0%	1.0	1.0
2	26.8%	2000	2.0%	1.8	1.2
4	32.0%	1000	2.9%	3.3	1.7
16	36.5%	500	4.3%	7.6	2.4

7.3.3 Root Parallelization

In the second series of experiments we tested the performance of root parallelization. The results regarding winning percentage, GPS speedup, and strength speedup for 1, 2, 4, and 16 threads are given in Table 7.2.

Table 7.2: Root parallelization.

Number of threads	Winning Percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7%	2000	2.0%	1	1.0
2	38.0%	2000	2.2%	2	3.0
4	46.8%	2000	2.2%	4	6.5
16	56.5%	2000	2.2%	16	14.9

Table 7.2 indicates that root parallelization is a quite effective way of parallelizing MCTS. One particularly interesting result is that, for 2 an 4 processor threads, the strength speedup is significantly higher than the number of threads used (i.e., 3.0 and 6.5, respectively). This result implies that, in the program MANGO, it is more efficient to run four independent MCTS searches of one second than to run one large MCTS search of four seconds. It might be that the algorithm stays for quite a long time in local optima. This effect is caused by the UCT coefficient setting. For small UCT coefficients, the UCT algorithm is able to search more deeply in the tree, but also stays a longer time in local optima. For high coefficients, the algorithm escapes more easily from the local optima, but the resulting search is shallower. The optimal coefficient for a specific position can only be determined experimentally. The time setting also influences the scalability of the results. For a short time setting, the algorithm is more likely to spend too much time in local optima. Hence, we believe that with higher time settings, root parallelization will be less efficient. In any case, we may conclude that root parallelization is a simple and effective way to parallelize MCTS. Experiments executed by Cazenave and Jouandeau (2007), and Winands and Björnsson (2010) confirm that root parallelization performs remarkably well for a small number of threads.

7.3.4 Tree Parallelization

In the third series of experiments we tested the performance of tree parallelization. Below, we have a closer look at the *mutexes location* and *virtual loss*.

Mutexes Location

First, the global-mutex method was tested. The results are given in Table 7.3. These results show that the strength speedup obtained up to 4 threads is satisfactory (i.e., strength speedup is 3). However, for 16 threads, this method is clearly insufficient. The strength speedup drops from 3 for 4 threads to 2.6 for 16 threads. So, we may conclude that the global-mutex method should not be used in tree parallelization.

Table 7.3: Tree parallelization with global mutex.

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	strength speedup
1	26.7%	2000	2.0%	1.0	1.0
2	31.3%	2000	2.1%	1.8	1.6
4	37.9%	2000	2.2%	3.2	3.0
16	36.5%	500	4.3%	4.0	2.6

Next, we tested the performance for the local-mutexes method. The results are given in Table 7.4. Table 7.4 shows that for each number of threads the local-mutexes method has a better strength speedup than the global-mutex method. Moreover, by using local mutexes instead of a global mutex the number of games played per second is doubled when using 16 processor threads. However, the strength speedup for 16 processors threads is just 3.3. Compared to the result of root parallelization (14.9 for 16 threads), this result is quite disappointing.

Table 7.4: Tree parallelization with local mutexes.

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7%	2000	2.0%	1.0	1.0
2	32.9%	2000	2.1%	1.9	1.9
4	38.4%	2000	2.2%	3.6	3.0
16	39.9%	500	4.3%	8.0	3.3

Using Virtual Loss

Based on the previous results we extended the local-mutexes tree parallelization with the virtual-loss enhancement. The results of using virtual loss are given in Table 7.5.

Table 7.5 shows that the effect of the virtual loss when using 4 processor threads is moderate. If we compare the strength speedup of Table 7.4 we see an increase from 3.0 to

Table 7.5: Using virtual loss for tree parallelization with local mutexes.

Number of threads	Winning percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7%	2000	2.0%	1.0	1.0
2	33.8%	2000	2.1%	1.9	2.0
4	40.2%	2000	2.2%	3.6	3.6
16	49.9%	2000	2.2%	9.1	8.5

3.6. But when using 16 processor threads, the result is more impressive. Tree parallelization with virtual loss is able to win 49.9% of the games instead of 39.9% when it is not used. The strength speedup of tree parallelization increases from 3.3 (see Table 7.4) to 8.5. Thus, we may conclude that virtual loss is important for the performance of tree parallelization when the number of processor threads is high.

7.3.5 Overview

Figure 7.4 depicts the performance of leaf parallelization, root parallelization, and tree parallelization with global mutex or with local mutexes. The x-axis represents the logarithmic number of threads used. The y-axis represents the winning percentage against GNU GO. For comparison reasons, we have plotted the performance of the default (sequential) program when given more time instead of more processing power. We see that root parallelization is superior to all other parallelization methods, performing even better than the sequential program.

7.3.6 Root Parallelization vs. Tree Parallelization Revisited

In the previous subsection we saw that on the 13×13 board root parallelization outperformed all other parallelization algorithms, including tree parallelization. It appears that the strength of root parallelization lies not only in a more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. The results may be different for other board sizes, time settings, and parameter settings. Therefore, we switched to a different board size (9×9) and three different time settings (0.25, 2.5, and 10 seconds per move). Using 4 processor threads, root and tree parallelization played both 250 games against the same version of GNU GO for each time setting. The results are given in Table 7.6. For 4 threads, we see that root parallelization and tree parallelization perform equally well now. Nevertheless, the number of games played and the number of threads used is not sufficient to give a definite answer which method is better.

7.4 Chapter Conclusions and Future Research

In this chapter we discussed the use of leaf parallelization and root parallelization for parallelizing MCTS. We introduced a new parallelization method, called tree parallelization.

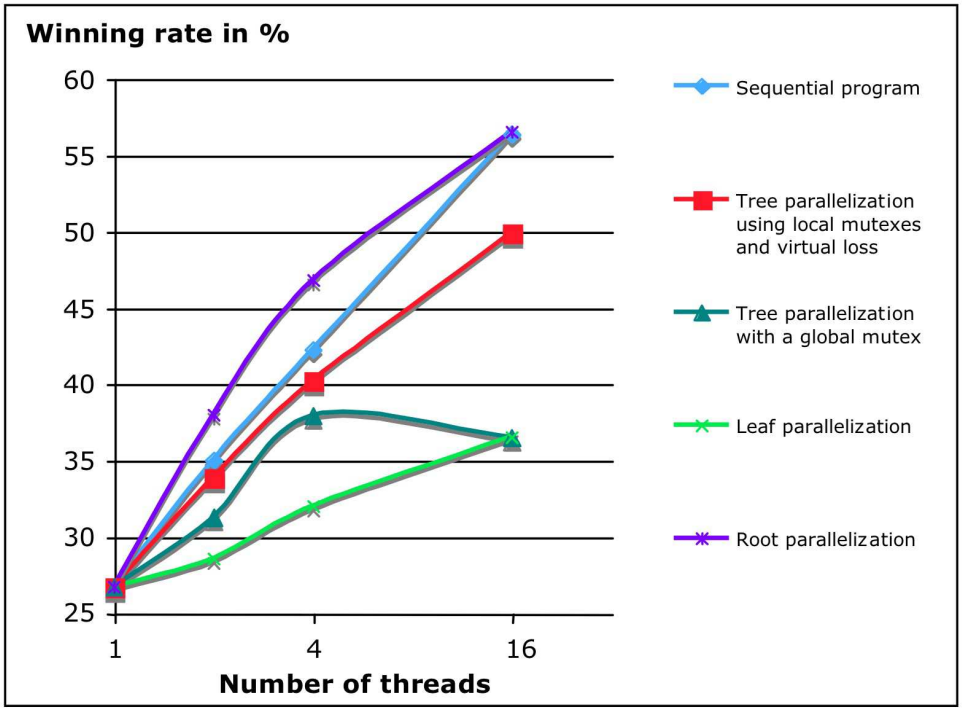


Figure 7.4: Performance of the different parallelization algorithms.

Table 7.6: 9×9 results for root and tree parallelization using 4 threads.

Time (s)	Winning percentage	
	Root parallelization	Tree parallelization
0.25	60.2%	63.9%
2.50	78.7%	79.3%
10.0	87.2%	89.2%

This method uses one shared tree from which games simultaneously are played. Experiments were performed to assess the performance of the parallelization methods in the Go program MANGO on the 13×13 board. In order to evaluate the experiments, we proposed the strength-speedup measure, which corresponds to the time needed to achieve the same strength. Experimental results indicated that leaf parallelization was the weakest parallelization method. The method led to a strength speedup of 2.4 for 16 processor threads. The simple root parallelization turned out to be the best way for parallelizing MCTS. The method led to a strength speedup of 14.9 for 16 processor threads. We saw that tree parallelization requires two techniques to be effective. First, using local mutexes instead of a global mutex doubles the number of games played per second. Second, the virtual-loss enhancement in-

creases both the games-per-second and the strength of the program significantly. By using these two techniques, we obtained a strength speedup of 8.5 for 16 processor threads.

Despite the fact that tree parallelization is still behind root parallelization, it is too early to conclude that root parallelization is the best way of parallelization. It transpires that the strength of root parallelization lies not only in a more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. Root parallelization repairs (partially) a problem in the UCT formula used by the selection mechanism, namely handling the issue of balancing exploitation and exploration. For now, we may conclude that root parallelization leads to excellent results for a specific time setting and specific program parameters. However, as soon as the selection mechanism is able to handle more adequately the balance of exploitation and exploration, we believe that tree parallelization could become the best choice for parallelizing MCTS.

There are three directions for future research. (1) In this chapter, we limited the tree parallelization to one SMP-node. For subsequent research, we will focus on tree parallelization and determine under which circumstances tree parallelization outperforms root parallelization. We believe that the selection strategy, the time setting, and the board size are important factors. Subsequently, we will test tree parallelization for a cluster with several SMP-nodes. Pioneering work on this topic has been performed by Gelly *et al.* (2008). (2) The question remains whether it is necessary to use mutexes at all for tree parallelization. Enzenberger and Müller (2010) showed that a mutex-free tree parallelization outperformed a global-mutex tree parallelization for the Go program FUEGO. It would be interesting to compare mutex-free tree parallelization with local-mutexes tree parallelization. (3) Most of the experiments in MANGO were performed in 13×13 Go. More experiments should be conducted for 9×9 and 19×19 Go.

Chapter 8

Generating Opening Books using Meta Monte-Carlo Tree Search

This chapter is based on the publications:

> P. Audouard, G.M.J-B. Chaslot, J-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud (2009). Grid Coevolution for Adaptive Simulations; Application to the Building of Opening Books in the Game of Go. *Applications of Evolutionary Computing*, Vol. 5484 of *Lecture Notes in Computer Science (LNCS)*, pp. 323–332, Springer-Verlag, Heidelberg, Germany.
>
> G.M.J-B. Chaslot, J-B. Hoock, J. Perez, A. Rimmel, O. Teytaud, and M.H.M. Winands (2009). Meta Monte-Carlo Tree Search for Automatic Opening Book Generation. *Proceedings of the IJCAI'09 Workshop on General Intelligence in Game Playing Agents*, pp. 7–12, Pasadena, CA, USA.

This chapter answers the fifth research question by investigating how we can generate automatically an opening book for an MCTS program. MCTS programs need, just like $\alpha\beta$ programs, an opening book to perform better. An opening book is a precalculated database of positions with their values that are likely to occur at the beginning of a tournament game (Lincke, 2001). Instead of performing a search, the opening book decides which move to be played. Besides saving time, an opening book may select stronger moves, assuming the time for precalculation is greater than the one used during play.

There have been a number of successful attempts to create opening books for $\alpha\beta$-based programs (Buro, 1999; Lincke, 2001; Karapetyan and Lorentz, 2006). Because the proposed methods so far are designed for programs based on a positional evaluation function, it is a challenge to generate an opening book for an MCTS program. In this chapter we propose to tackle this issue by combining two levels of MCTS. The method is called Meta Monte-Carlo Tree Search (Meta-MCTS). Instead of using a weak simulation strategy, it uses an entire MCTS program (MoGo) to play a simulated game. Cazenave (2007b) applied an online Meta-MCTS composed of two UCT algorithms (Kocsis and Szepesvári, 2006) to get the world record in the one-player game "Morpion Solitaire". However, his approach has

only been designed for one-player games. In this chapter we show that Meta-MCTS can be used for generating an opening book for Go. For this task, we present two Meta-MCTS algorithms: the first one, Quasi Best-First (QBF), favours exploitation; the second one, Beta-Distribution Sampling (BDS), favours exploration. QBF is an adaptation of greedy selection algorithms that are used for the regular MCTS. The idea of BDS is that the probability that a move is selected is proportional to the likelihood that it is the best move (according to its number of wins and losses). In contrast to UCT, selecting the moves is not deterministic in BDS. The selection strategy of BDS did not perform well in the MCTS program MANGO because it was too explorative. In order to evaluate the performance of QBF and BDS, we test the generated 9×9 Go opening books against computer programs and humans.

Section 8.1 presents previous research on creating opening books. Next, in Section 8.2 we discuss Meta-MCTS and propose two algorithms, Quasi Best-First and Beta-Distribution Sampling. Subsequently, Section 8.3 presents the experimental results. Finally, Section 8.4 concludes this chapter and gives suggestions for future research.

8.1 Automatic Opening Book Generation

An opening book can conceptually be viewed as a tree (Karapetyan and Lorentz, 2006). The root is the initial game position. Nodes in general correspond to positions that occur later in the game and record the heuristic evaluation of the position (e.g., win ratio or negamax score). Edges represent legal moves from one position to the next. During game play, if the position is in the book, the selected move is the one with the highest heuristic score.

Lincke (2001) distinguishes between *passive* and *active book construction*. Passive book construction involves adding moves to the opening book based on information gathered from experts, either from their games or from their knowledge of the game. Active construction means constructing the book automatically. For programs using $\alpha\beta$ search, there are quite a few methods for generating opening books automatically. The most popular one is based on the concept of the drop-out mechanism (Lincke, 2001; Karapetyan and Lorentz, 2006). It is a best-first search method that applies for a fixed amount of time an $\alpha\beta$ search at a leaf node. Next, it backpropagates the (heuristic) score found at the leaf node in a negamax way. For selecting the most-proving node to expand next, the procedure is as follows. At each internal node, the move is chosen that maximizes the negamax score minus a certain depth penalty. This depth penalty is proportional to the distance to the leaf node that backpropagated the negamax score. It enables that a player drops out the book quickly only when the position is quite advantageous for him.

The application of this mechanism to Go raises a problem: there is no fast and efficient evaluation function available in Go in order to use an $\alpha\beta$ search at the leaf node. It would be possible to replace the $\alpha\beta$ search by MCTS. However, the score of an MCTS search is quite instable, in contrast to the stable minimax score of an $\alpha\beta$ search equipped with a sophisticated evaluation function. The unstable nature of the MCTS score could have a negative effect when constructing the opening book. Using an MCTS variant to generate an opening book appears to be more natural and elegant. We will discuss this further in the next section. Finally, we remark that in some games, programs evolve so fast that a good opening book may become out-dated quickly. Some programs have therefore shifted to online verification of the book moves (Donninger and Lorenz, 2006).

8.2 Meta Monte-Carlo Tree Search

In this section, we first give the general structure of Meta Monte-Carlo Tree Search (Meta-MCTS) in Subsection 8.2.1. Next, in Subsection 8.2.2, we describe the Quasi Best-First algorithm. Finally, we introduce the Beta-Distribution Sampling algorithm in Subsection 8.2.3.

8.2.1 General Idea

An MCTS program uses a weak simulation strategy in order to find the best move. The idea of Meta-MCTS consists of replacing the weak simulation strategy at the lower part of the search by an entire MCTS program (e.g., the Go program MoGo). This program is the *lower level* of the Meta-MCTS. As MCTS programs are computationally expensive, applying a second level of MCTS is quite time consuming, and cannot be performed in real time. However, using this strategy off-line for generating an opening book is possible.

We call the part of the search where the selection strategy decides which move will be explored further, the *upper level*. This selection strategy has to be adapted as well. The standard UCT formula (Kocsis and Szepesvári, 2006) requires an exploration constant C to be tuned. Tuning this constant C for a two-level MCTS would take quite an amount of time. Therefore, we propose two alternatives: Quasi Best-First (QBF) and Beta-Distribution Sampling (BDS). QBF and BDS are described in Subsections 8.2.2 and 8.2.3.

8.2.2 Quasi Best-First

MCTS is often emphasized as a compromise between exploration and exploitation. Nevertheless, many programmers have seen that in the case of deterministic games, the exploration constant C of the UCT formula, when optimized, has to be set close to zero. A small exploration value is given to every move when using a specific strategy such as RAVE (Gelly and Silver, 2007) or Progressive Bias (Chaslot *et al.*, 2008c). In both cases the exploration term will converge fast to zero. The consequence of using such a small exploration value is that, after a few games, a move is further analyzed as long as it is the move with the highest winning rate. Therefore, most MCTS programs can be qualified as being greedy. This subsection introduces the Quasi Best-First (QBF) algorithm,[1] which was originally proposed by Olivier Teytaud and Arpad Rimmel. QBF usually selects the child with the highest winning rate. However, if a move's winning rate drops below a certain threshold K, QBF will ask the MCTS program (here MoGo) to choose a move. The pseudo code is given in Algorithm 8.1. Because of executing an entire MCTS program, the (opening-book) tree grows quite slowly. Instead of adding only the first position encountered that was not already stored (see Subsection 3.3.2), all the positions are added that are visited visit when playing a simulated game. Backpropagating, though, stays the same by taking the plain average of the results (see Subsection 3.3.4).

[1] QBF was previously called MVBM (see Audouard *et al.*, 2009).

```
QBF(K, λ)
while True do
  for l = 1..λ, do
    p = initial position; g = {p}.
    while p is not a terminal position do
      bestScore = K
      bestMove = Null
      for m in the set of possible moves in p do
        score = winRatio(p, m)
        if score > bestScore then
          bestScore = score
          bestMove = m
        end if
      end for
      if bestMove = Null then
        bestMove = MoGoChoice(p) // lower level MCTS
      end if
      p = playMove(p, bestMove)
      g = concat(g, p)
    end while
    addToBook(g, g.result)
  end for
end while
```

Algorithm 8.1: The "Quasi Best-First" (QBF) algorithm. λ is the number of machines available. K is a constant. g is a game, defined as a sequence of game positions. The function "MoGoChoice" asks MoGo to choose a move.

8.2.3 Beta-Distribution Sampling

Each node in a game tree has a game-theoretic value. In Go this value is either 0 in case it corresponds to a won position for White, or 1 in case it corresponds to a won position for Black. For MCTS, the convergence to the game-theoretic value is in practice slow. We observed that the value of a node may get stuck in a local optimum for a long time. From this observation, we propose a hypothesis of stability H_s: each position P has a stationary average value $\mu_{s,P}$ that only depends on P and on the simulation strategy s that is used. For instance, the standard version of MoGo uses $\mu_{fastPattern,P}$, where $fastPattern$ is a fast simulation strategy that uses 3×3 patterns to make its decision. The upper level of the Meta-MCTS uses $\mu_{MoGoGames,P}$, where $MoGoGames$ is a simulation strategy that uses MoGo to make its decision.

Let $w_{s,P}$ be the number of wins of the games made by the simulation strategy s, which went through the position P. Let $l_{s,P}$ be the number of losses of the games made by the simulation strategy s, which went through the position P. Under the hypothesis H_s, the probability that the game is a win for the player to move in position P, is $\mu_{s,P}$. The number of wins and losses obeys a Bernoulli distribution. The probability distribution of $\mu_{s,P}$ knowing $w_{s,P}$ and $l_{s,P}$ is given by the conjugate prior of the Bernoulli distribution which is a beta distribution. The formula of this distribution is given below.

```
BDS(λ)
while True do
  for l = 1..λ, do
    p = initial position; g = {p}.
    while p is not a terminal position do
      bestScore = −∞
      bestMove = Null
      for m in the set of possible moves in p do
        score =draw from distribution:
        x → x^{w_{MoGoGames,m}} · (1 − x)^{l_{MoGoGames,m}}
        if score > bestScore then
          bestScore = score
          bestMove = m
        end if
      end for
      if bestMove = Null then
        bestMove = MoGoChoice(p) // lower level MCTS
      end if
      if random_int modulo p.visit_count = 0 then
        bestMove = MoGoChoice(p) // lower level MCTS
      end if
      p = playMove(p, bestMove)
      g = concat(g, p)
    end while
    addToBook(g, g.result)
  end for
end while
```

Algorithm 8.2: The "Beta-Distribution Sampling" (BDS) algorithm. λ is the number of machines available. g is a game, defined as a sequence of game positions. The function "MoGoChoice" asks MoGo to choose a move.

$$p(\mu_{s,P} = x | w_{s,P}, l_{s,P}) = x^{w_{s,P}} \cdot (1-x)^{l_{s,P}} \tag{8.1}$$

We propose the following selection strategy, called Beta-Distribution Sampling (BDS), which consists of sampling a random number r_i from each beta distribution for each child i.[2] The child selected is the one with the best r_i. The pseudo code is provided in Algorithm 8.2. According to this selection strategy, each node is selected with the probability that it is the best node, assuming the hypothesis H_s. This concept is similar to the idea of the selection strategy developed by Chaslot *et al.* (2006a) and Coulom (2006). The benefit of BDS is that there are fewer approximations.

[2]We used the scientific library Blitz++ to draw random numbers according to a beta distribution. Webpage: http://www.oonumerics.org/blitz/

8.3 Experiments

In this section, we generate several 9×9 Go opening books using QBF and BDS. We evaluate their performances and provide statistics that help understanding the structure of these books. All these opening books were generated on a grid.[3] For all experiments the symmetry of the board positions was taken into account.

Subsection 8.3.1 tests QBF, and Subsection 8.3.2 reports on experiments comparing QBF and BDS.

8.3.1 QBF Experiments

In this subsection we show the performance of QBF for 9×9 Go. First, we perform experiments with $K = 0.5$ in QBF. Next, we present tests in self-play and with an expert opening book.

Experiments with $K = 0.5$

In the first series of experiments we tested the quality of the QBF generated opening book with a constant *K* of 0.5. When generating the book the program MoGo used 10 seconds for choosing a move at the lower level. The generated QBF book contained 6,000 games. For evaluating the quality of the QBF book we matched two versions of MoGo against each other. One was using the QBF book and the other one did not use a book at all. Both programs received 10 seconds thinking time per move and played on an 8-core machine. Moreover, we also matched the program using the QBF book against one using an "expert book". This expert opening book has been specially designed for MoGo by Pierre Audouard.[4] The results are given in Table 8.1.

Table 8.1: Performance of the QBF algorithm with 10 seconds per move and $K = 0.5$. The confidence interval is $\pm$ 1.9%.

	No book vs. no book	QBF book vs. no book	QBF vs. expert book
White	51.5%	64.3%	64.1%
Black	48.5%	48.0%	46.1%
Average	50.0%	56.2%	55.1%

The first column gives an average success rate of 50%, since it is self-play. The second column shows the results for White (respectively Black) with the QBF book against no book. We see that the one using an opening book performs significantly better. In the third column we see that the QBF book also outperforms the expert book. However, in both cases we observe that Black does not improve when using the QBF book. This can be explained as follows: as long as Black has not found a move with success rate $> 50\%$, it always asks MoGo for a move to play. Therefore, White improves its results by choosing moves with a high success rate, but not Black. This is why in the remainder of the chapter, we

[3] The grid was Grid5000, well-suited for large-scale scientific experiments.

[4] Pierre Audouard was the French Champion in 19×19 Go and is the current World Champion in 9×9 Go for people with a physical handicap.

Table 8.2: Success rate of the QBF book and expert book against the default MOGO using 6 hours for each side.

	QBF opening book	Expert opening book
White	74.5% ± 2.3%	62.9% ± 3.8%
Black	64.6% ± 2.4%	49.7% ± 3.8%
Average	69.6% ± 2.4%	56.3% ± 3.8%

use $K = 0.1$ for Black. (Table 8.2 QBF shows that this setting also improves the level as Black).

QBF in Self-Play and against Expert Opening Book

In the following series of experiments we generate a QBF book by using more time at the lower level. Instead of using 10 seconds a move we used 12 hours for the complete game (six hours for each side) on a quad-core machine. The final book contained 3,000 games.

We tested the quality of the QBF book by matching two versions of MOGO against each other. One version was using the book, the other was not. The time setting was six hours for each side. The results are presented in Table 8.2. For comparison reasons we also tested in a similar way the quality of the expert book. We observe that MOGO performs better when using the QBF book than when using the expert book. Finally, we see that the QBF book improves the performance of both colours.

8.3.2 Experiments Comparing QBF and BDS

One could consider to compare QBF and BDS by self-play experiments. However, it should be remarked that self-play experiments favour the greedy strategies (Lincke, 2001). Hence, a comparison of QBF and BDS on this basis would be biased. We propose a different way of comparison. First, we measure the length of staying in the book against other opponents. Next, we compare QBF and BDS on the computer Go server CGOS.

Comparing the Length of Staying in the Book against Other Opponents

In the following series of experiments, we compare the QBF book to the BDS book. Figure 8.1 shows the distribution of the length of staying in the opening book when playing 500 games as Black and 500 games as White against the MCTS program FUEGO (Enzenberger and Müller, 2009). This program is quite similar to MOGO. In the figure we see that as Black QBF has an average length of 9.6 and BDS has an average length of 9.8. As White, QBF has an average length of 14.6 whereas BDS has only a length of 9.0. As FUEGO is quite close to MOGO, the opening book generated by QBF is a good predictor; yet it missed some moves for Black. We may conclude that QBF builds a book that is asymmetrical in considering Black and White. Because Black has the disadvantage, its best move value will go below *K* more often than it would be for White.

In the following series of experiments, we played against GNU GO. This program is not MCTS-based and therefore much more different from MOGO than FUEGO. Figure 8.2

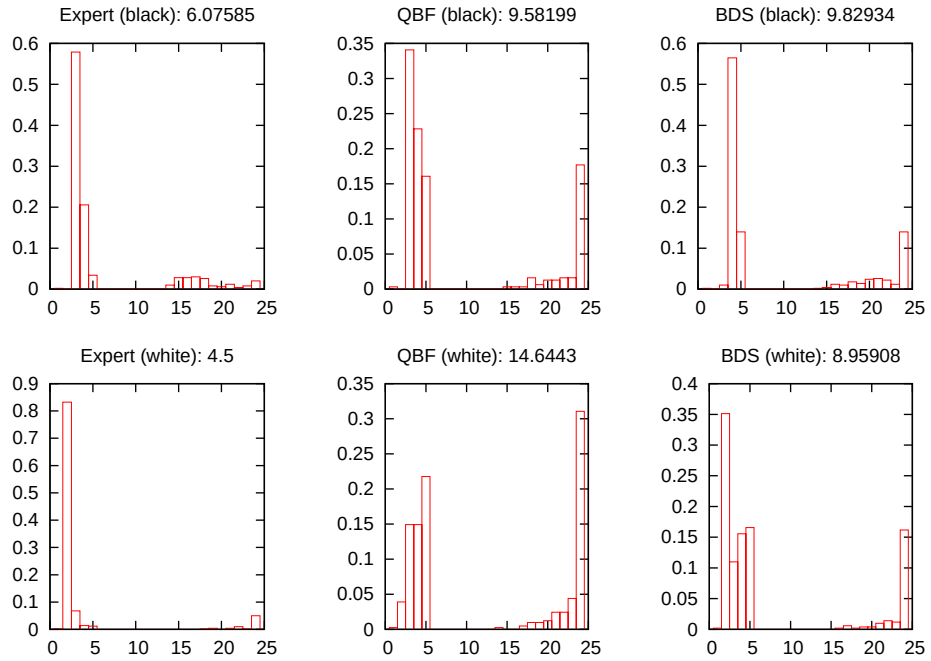


Figure 8.1: Number of book moves when playing against FUEGO with the opening book. The x-axis is the depth and y-axis is the density of probability. First row: playing as Black. Second row: playing as White. First column: expert opening book. Second column: QBF opening book of 12,000 games. Third column: BDS opening book of 12,000 games. Each label contains the average length of the book. All histograms are estimated on 500 games.

shows the distribution of the length of staying in the opening book when playing 500 games as Black and 500 games as White. It is clear that as White BDS stayed longer in the opening book than QBF, 4.7 and 3.7 moves, respectively. However, as Black BDS stayed shorter in the opening book than QBF, 5.3 and 7.8, respectively.

Next, we compared the number of moves staying in the opening book against human experts. All the responses that are found in the classic 9×9 Go book[5] are also found in the QBF book.

An in-depth analysis showed unfortunately that when QBF was generating the book it soon selected always **e5** as the first move. All other opening moves in the initial position were only explored a small number of times. This was a clear drawback of the QBF approach when playing against human players. This happened in the games played against Prof. Tsai (6 Dan).

Another example of the performance of QBF can be found in the official match against Motoki Noguchi (7 Dan) (Teytaud, 2008). The result of the match was 2–2. This was the first time that a computer program was able to draw against a player of that calibre. These games were played with the QBF and expert book, containing 6,000 games. In the games

[5] See http://senseis.xmp.net/?9x9Openings

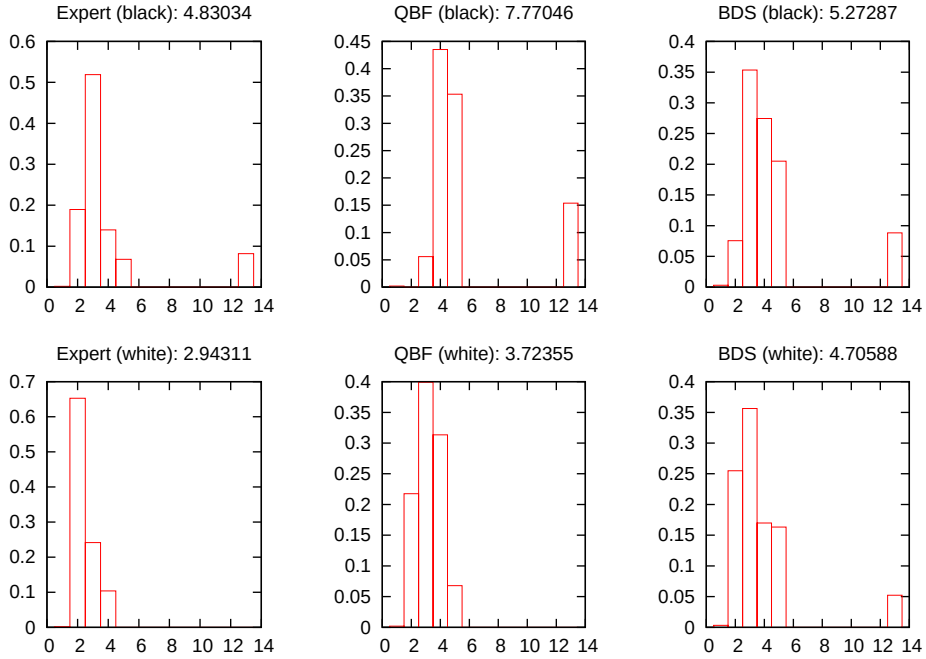


Figure 8.2: Number of book moves when playing against GNU GO with the opening book. The x-axis is the depth and y-axis is the density of probability. First row: playing as Black. Second row: playing as White. First column: expert opening book. Second column: QBF opening book of 12,000 games. Third column: BDS opening book of 12,000 games. Each label contains the average length of the book. All histograms are estimated on 500 games.

won by MOGO, the opening book gave an advantage to MOGO and continuously increased its advantage. In both lost games, Motoki Noguchi went out of the opening book quite fast. However, the book later developed by BDS would have contained more moves. The sequence **e5-c4-g3-e3-e4-d4** was explored only 22 times by the QBF opening book, but 424 times by the BDS. In the other game lost by MOGO, the sequence **e5-e7-g6-f3** has been explored 13 times by QBF and 18 times by BDS.

This shows that, despite the quite long sequence in the opening book against computers, QBF does not explore enough promising moves for playing against humans. BDS may appear to be a better alternative against human play.

Comparison on the Go Server CGOS

In the final series of experiments, we used the Go server CGOS in order to assess QBF and BDS. In order to perform a fair comparison between the algorithms, we created two dedicated opening books. The first was created by QBF and the second by BDS. Each opening book was created by using 32 cores in parallel, 1 second per move, with a total of 5,120 games. We launched four versions of MOGO on CGOS: (1) without a book, (2) with a QBF book, (3) with a BDS book, and (4) with a combined book (QBF+BDS). Subsequently, we

compared the ELO rating that they obtained by playing against a pool of different opponents. In order to make the comparison as fair as possible, we launched the four versions simultaneously on the server. Moreover, to avoid that the different MoGo versions played too many games against each other, we only launched them when there were enough other programs available. The different versions of MoGo played around 70% of their games against non-MoGo versions. The results can be found in Table 8.3. This experiment shows that QBF and BDS give a significant improvement on the version without opening book, and that merging directly the two opening books is counter-productive. The two books were not built to be combined with each other, so negative side effects may appear.

Table 8.3: Results on the computer Go server CGOS.

QBF	BDS	CGOS rating	Games
No	No	2216	371
Yes	No	2256	374
No	Yes	2268	375
Yes	Yes	2237	373

8.4 Chapter Conclusions and Future Research

In this chapter we proposed Meta Monte-Carlo Tree Search (Meta-MCTS) for generating an opening book. Meta-MCTS is similar to MCTS, but the weak simulation strategy is replaced by a standard MCTS program. We described two algorithms for Meta-MCTS: Quasi Best-First (QBF) and Beta-Distribution Sampling (BDS). The first algorithm, called QBF, is an adaptation of greedy algorithms that are used for the regular MCTS. QBF favours therefore exploitation. During actual game play we noticed that despite the good performance of the opening book, some branches were not explored sufficiently. The second algorithm, called BDS, favours exploration. In contrast to UCT, BDS does not need an exploration coefficient to be tuned. This approach created an opening book which is shallower and wider. The BDS book had the drawback to be less deep against computers, but the advantage was that it stayed longer in the book in official games against humans. Experiments on the Go server CGOS revealed that both QBF and BDS were able to improve MoGo. In both cases the improvement was more or less similar. Based on the results, we may conclude that QBF and BDS are able to generate an opening book which improves the performance of an MCTS program.

As future research, we want to test other ways for generating an opening book. In particular, transferring classic techniques derived from $\alpha\beta$ search to MCTS constitutes an interesting challenge.

Chapter 9

Conclusions and Future Research

In this chapter, we present the conclusions of the thesis. In Section 9.1 we answer the five research questions and provide an answer to the problem statement. In Section 9.2 we provide promising directions for future research.

9.1 Answers to the Problem Statement and Research Questions

In this thesis we investigated a Monte-Carlo technique called Monte-Carlo Tree Search (MCTS). It is a best-first search method guided by the results of Monte-Carlo simulations. MCTS, described in Chapter 3, can be divided in four major steps: *selection*, *expansion*, *simulation*, and *backpropagation*. The following problem statement guided our research.

> **Problem statement**: *How can we enhance Monte-Carlo Tree Search in such a way that programs improve their performance in a given domain?*

Enhancing the strategies for each MCTS step improves the playing strength of the program. We discussed that the two most crucial steps were simulation and selection (Chapter 3). This led to the first and second research question, which deal with improving the simulation strategy by using knowledge (Chapter 4), and improving the selection strategy with knowledge (Chapter 5), respectively. For the third research question we investigated how we can optimize the parameters in MCTS (Chapter 6). The fourth research question aimed at investigating how well MCTS can be parallelized on modern multi-core computers (Chapter 7). Finally, the fifth research question addressed the application of MCTS to build an opening book automatically (Chapter 8). The answers to the five research questions are given below.

> **Research question 1**: *How can we use knowledge to improve the Monte-Carlo simulations in MCTS?*

We focused on enhancing the simulation strategy by introducing *knowledge* in the Monte-Carlo simulations. The knowledge transforms the plain random simulations into more sophisticated *pseudo-random* simulations.

We discussed two different simulation strategies that apply knowledge: urgency-based and sequence-like simulation. Based on the experience gathered from implementing them in INDIGO and MOGO, respectively, we make the following three recommendations. (1) Avoiding big mistakes is more important than playing good moves. (2) Simulation strategies using sequence-like simulations or patterns in urgency-based simulations are efficient because they simplify the situation. (3) The simulation strategy should not become too stochastic, nor too deterministic, thus balancing exploration and exploitation.

Moreover, we developed the first efficient method for learning automatically the knowledge of the simulation strategy. We proposed to use *move evaluations* as a fitness function instead of learning from the results of simulated games. A coefficient was introduced that enables to balance the amount of exploration and exploitation. The algorithm was adapted from the tracking algorithm of Sutton and Barto (1998). Learning was performed for 9×9 Go, where we showed that the Go program INDIGO with the learnt patterns performed better than the program with expert patterns.

> **Research question 2**: *How can we use knowledge to arrive at a proper balance between exploration and exploitation in the selection step of MCTS?*

A selection strategy such as UCT controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation). On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration). We saw that the MCTS program MANGO equipped with UCT increasingly performed worse for larger board sizes when playing against GNU GO.

We introduced therefore two progressive strategies. These strategies use (1) the information available for the selection strategy, and (2) some (possibly time-expensive) domain knowledge that is introduced by an expert, or learnt automatically. The two progressive strategies we developed are *progressive bias* and *progressive widening*. Progressive bias uses knowledge to direct the search. Progressive widening first reduces the branching factor, and then increases it gradually. This scheme is also dependent on knowledge.

The progressive strategies were first tested in MANGO. The incorporated knowledge was based on urgency-based simulation. From the experiments with MANGO, we observed the following. (1) Progressive strategies, which focus initially on a small number of moves, are better in handling large branching factors. They increased the level of play of the program MANGO significantly, for every board size. (2) On the 19×19 board, the combination of both strategies is much stronger than each strategy applied separately. The fact that progressive bias and progressive widening work better in combination with each other shows that they have complementary roles in MCTS. This is especially the case when the board size and therefore branching factor grows. (3) Progressive strategies can use relatively expensive domain knowledge with hardly any speed reduction.

The progressive strategies were successfully implemented in other game programs and domains. Progressive bias increased the playing strength of MOGO and MC-LOA, while progressive widening did the same for CRAZY STONE. Moreover, in the case of MOGO, progressive bias was successfully combined with RAVE (Gelly and Silver, 2007), a similar technique for improving the balance between exploitation and exploration. These results give rise to the main conclusion that the proposed progressive strategies are essential enhancements for an MCTS program.

Research question 3: *How can we optimize the parameters of an MCTS program?*

In our attempt to answer this research question, we proposed to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM). The fitness function for CEM measures the winning rate for a batch of games. The performance of CEM with a fixed and variable batch size was tested by tuning 11 parameters in MANGO. Experiments revealed that using a batch size of 500 games gave the best results, although the convergence was slow. To be more precise, these results were obtained by using a cluster of 10 quad-core computers running for 3 days. Interestingly, a small (and fast) batch size of 10 still gave a reasonable result when compared to the best one. A variable batch size performed a little bit worse than a fixed batch size of 50 or 500. However, the variable batch size converged faster than a fixed batch size of 50 or 500. Subsequently, we showed that MANGO with the CEM parameters performed better against GNU GO than the MANGO version without. Moreover, in four self-play experiments with different time settings and board sizes, the CEM version of MANGO defeated the default version convincingly each time. Based on these results, we may conclude that a hand-tuned MCTS-using game engine may improve its playing strength when re-tuning the parameters with CEM.

Research question 4: *How can we parallelize MCTS?*

We first showed that MANGO's playing strength (measured in ELO points) increased nearly linearly as a function of the logarithmic time. Each doubling of time increased the strength by 50 ELO points against GNU GO. We aimed at obtaining similar results by increasing the number of cores, for a fixed time setting. We tested three different parallelization methods: (1) leaf parallelization, (2) root parallelization, and (3) tree parallelization.

Experimental results indicated that leaf parallelization was the weakest parallelization method. Root parallelization led to surprisingly good results, with a nearly linear speed-up for 16 cores. These unexpected results have been confirmed in other programs.We saw that tree parallelization requires two techniques to be effective. First, using local mutexes instead of a global mutex doubled the number of games played per second. Second, the virtual-loss enhancement increased both the speed and the strength of the program significantly. The two conclusions are as follows: (1) Root parallelization and tree parallelization perform significantly better than leaf parallelization. (2) For a multi-core machine, parallelized MCTS has almost a linear speed-up up to 16 cores and scales therefore quite well.

Research question 5: *How can we automatically generate opening books by using MCTS?*

We proposed to use Meta Monte-Carlo Tree Search (Meta-MCTS) for generating an opening book. Meta-MCTS is similar to MCTS, but the simulation strategy is replaced by a standard MCTS program. We described two algorithms for Meta-MCTS: Quasi Best-First (QBF) and Beta-Distribution Sampling (BDS). The first algorithm, QBF (proposed by Teytaud and Rimmel), is an adaptation of greedy algorithms that are used for the regular MCTS. During actual game play we noticed that despite the good performance of the opening book, some branches were not explored sufficiently. QBF therefore favours exploitation. We developed the second algorithm, that we called Beta-Distribution Sampling

(BDS), which favours exploration. The algorithm draws a move according to its likelihood of being the best move (considering the number of wins and losses). This approach created an opening book which is shallower but wider. Experiments on the 9×9 Go server CGOS revealed that both QBF and BDS were able to improve the Go program MoGo. In both cases the improvement in playing strength was approximately 50 ELO points. Based on the results, we may conclude that QBF and BDS are able to generate an opening book which improves the performance of an MCTS program.

After answering all five research questions, we are now able to provide an answer to the problem statement, which is repeated here for convenience.

> **Problem statement**: *How can we enhance Monte-Carlo Tree Search in such a way that programs improve their performance in a given domain?*

The thesis proposed an answer to the problem statement, which may essentially be summarized in five points. First, we improved the knowledge in the simulation strategy by learning from move evaluations. Second, we enhanced the selection strategy by proposing progressive strategies for incorporating knowledge. Third, we applied CEM to optimize the search parameters of an MCTS program in such a way that its playing strength was increased. Fourth, we showed that MCTS benefits substantially from parallelization. Fifth, we designed Meta-MCTS to generate an opening book that improves the performance of an MCTS program.

9.2 Future Research

From our observations, we believe that there are several research directions for the following topics.

Simulation

In the thesis we developed offline simulation strategies. Promising work for constructing online a simulation strategy include the ideas of Finnsson and Björnsson (2008) and Sharma *et al.* (2008). An interesting direction for future research is to combine offline and online strategies with each other.

Selection

An idea is to modify during game play (online) the progressive strategy according to the results of the Monte-Carlo simulations. For instance, in a situation where the initial *k* moves are losing, increasing the speed of widening to find the best move seems promising. As another example, if a move that receives always a high heuristic score H_i is rarely the best move in numerous nodes, then the heuristic score of this move could be adjusted online.

Parallelization

A research direction is to combine root parallelization with tree parallelization. Several $n \times m$ configurations exist, where *n* is the number of independent trees (root parallelization) and *m* the number of threads running on each tree (tree parallelization). Furthermore, the independent MCTS trees can be synchronized regularly. It is an open question to find the optimal setup for a certain number of processor threads.

Opening-Book Construction

We observed that the automatically generated opening book sometimes plays weak moves, due to the fact that for specific situations, MCTS evaluates a position incorrectly (e.g., scoring a position too high). Even if these positions are rare, the generated opening book often tries to reach these positions. A solution to this problem is to let human experts detect the errors and correct the value of a position. Hence, we believe that the best opening book will not be only computer-generated, neither human-generated, but will come from the interaction of humans and computers.

Other Domains

In this thesis, the analysis of MCTS was restricted to games with perfect information. Adapting MCTS to games with imperfect information is an interesting challenge. Only a small amount of research has been performed so far (e.g., the game of Kriegspiel; Ciancarini and Favini, 2009). Different choices have to be made to model imperfect information.

Understanding the Nature of MCTS

Noteworthy results were achieved of enhancing MCTS in Go and other games. Yet, the underlying principles of MCTS are not fully understood. In his seminal work Beal (1999) investigated the nature of minimax. Minimax algorithms and enhancements have benefitted greatly from this fundamental analysis. We expect that a similar phenomenon will hold for MCTS algorithms as well. Therefore, we propose to investigate the nature of MCTS in order to understand better its fundamentals.

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. [16, 17]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands. [9, 10]

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–123. [2, 10]

Audibert, J.Y. and Bubeck, S. (2009). Minimax Policies for Adversarial and Stochastic Bandits. *Proceedings of the 22nd Annual Conference on Learning Theory (COLT 2009)*, Omnipress. [21]

Audouard, P., Chaslot, G.M.J-B., Hoock, J-B., Perez, J., Rimmel, A., and Teytaud, O. (2009). Grid Coevolution for Adaptive Simulations: Application to the Building of Opening Books in the Game of Go. *Applications of Evolutionary Computing*, Vol. 5484 of *LNCS*, pp. 323–332, Springer-Verlag, Berlin Heidelberg, Germany. [77, 79]

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-Time Analysis of the Multi-Armed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [1, 3, 16, 21]

Barto, A.G., Bradtke, S.J., and Singh, S.P. (1995). Learning to Act using Real-Time Dynamic Programming. *Artificial Intelligence*, Vol. 72, Nos. 1–2, pp. 81–138. [27]

Baxter, J., Tridgell, A., and Weaver, L. (1998). Experiments in Parameter Learning Using Temporal Differences. *ICCA Journal*, Vol. 21, No. 2, pp. 84–99. [54]

Beal, D.F. (1999). *The Nature of Minimax Search.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [91]

Beal, D.F. and Smith, M.C. (2000). Temporal Difference Learning for Heuristic Search and Game Playing. *Information Sciences*, Vol. 122, No. 1, pp. 3–21. [53]

Berger, F. (2007). BGBlitz Wins Backgammon Tournament. *ICGA Journal*, Vol. 30, No. 2, p. 114. [28]

Berliner, H.J. (1979). The B*-Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, Vol. 12, No. 1, pp. 23–40. [2]

Billings, D. and Björnsson, Y. (2003). Search and Knowledge in Lines of Action. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 231–248. Kluwer Academic Publishers, Boston, MA, USA. [50]

Billings, D., Peña, L., Schaeffer, J., and Szafron, D. (1999). Using Probabilistic Knowledge and Simulation to Play Poker. *AAAI/IAAI*, pp. 697–703. [16]

Björnsson, Y. and Marsland, T.A. (2003). Learning Extension Parameters in Game-Tree Search. *Information Sciences*, Vol. 154, No. 3, pp. 95–118. [54]

Boer, P-T. de, Kroese, D.P., Mannor, S., and Rubinstein, R.Y. (2005). A Tutorial on the Cross-Entropy Method. *Annals of Operations Research*, Vol. 134, No. 1, pp. 19–67. [59, 60]

Boon, M. (1990). A Pattern Matcher for Goliath. *Computer Go*, Vol. 13, pp. 13–23. [13]

Bouzy, B. (2003). Mathematical Morphology Applied to Computer Go. *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 17, No. 2, pp. 257–268. [10]

Bouzy, B. (2005). Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, Vol. 175, No. 4, pp. 247–257. [3, 22, 31, 32, 33, 34, 38, 55]

Bouzy, B. (2006). Associating Shallow and Selective Global Tree Search with Monte Carlo for 9×9 Go. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N. Netanyahu), Vol. 3846 of *LNCS*, pp. 67–80, Springer-Verlag, Berlin Heidelberg, Germany. [13, 17]

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI Oriented Survey. *Artificial Intelligence*, Vol. 132, No. 1, pp. 39–103. [4]

Bouzy, B. and Chaslot, G.M.J-B. (2005). Bayesian Generation and Integration of K-Nearest-Neighbor Patterns for 19×19 Go. *IEEE 2005 Symposium on Computational Intelligence in Games* (eds. G. Kendall and S. Lucas), pp. 176–181, Essex, UK. [44, 48]

Bouzy, B. and Chaslot, G. (2006). Monte-Carlo Go Reinforcement Learning Experiments. *IEEE 2006 Symposium on Computational Intelligence in Games*, pp. 187–194, Reno, USA. [23, 31, 34, 35, 38, 55]

Bouzy, B. and Helmstetter, B. (2003). Monte-Carlo Go Developments. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 159–174, Kluwer Academic Publishers, Boston, MA, USA. [3, 13, 16, 17]

Brügmann, B. (1993). Monte Carlo Go. Technical report, Physics Department, Syracuse University, Syracuse, NY, USA. [3, 13, 16, 51]

Bubeck, S., Munos, R., Stoltz, G., and Szepesvári, C. (2008). Online Optimization in X-Armed Bandits. *Advances in Neural Information Processing Systems 21 (NIPS 2008)* (eds. D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou), pp. 201–208. [21]

Buro, M. (1999). Toward Opening Book Learning. *ICCA Journal*, Vol. 22, No. 2, pp. 98–102. [4, 77]

Campbell, M., Hoane, A.J., and Hsu, F-H. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 57–83. [4, 65]

Cazenave, T. (2001). Generation of Patterns with External Conditions for the Game of Go. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 275–293, Universiteit Maastricht, Maastricht, The Netherlands. [11]

Cazenave, T. (2007a). Playing the Right Atari. *ICGA Journal*, Vol. 30, No. 1, pp. 35–42. [55]

Cazenave, T. (2007b). Reflexive Monte-Carlo Search. *Proceedings of the Computer Games Workshop 2007 (CGW 2007)* (eds. H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd), pp. 165–173, Universiteit Maastricht, Maastricht, The Netherlands. [77]

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. 50–51, Springer-Verlag, Berlin Heidelberg, Germany. [27]

Cazenave, T. (2009). Nested Monte-Carlo Search. *International Joint Conference On Artificial Intelligence (IJCAI 2009)*, pp. 456–461. [26]

Cazenave, T. and Jouandeau, N. (2007). On the Parallelization of UCT. *Proceedings of the Computer Games Workshop 2007 (CGW 2007)* (eds. H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd), pp. 93–101, Universiteit Maastricht, Maastricht, The Netherlands. [65, 66, 67, 71]

Chaslot, G.M.J-B., Saito, J-T., Bouzy, B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2006a). 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. [1, 3, 13, 15, 16, 18, 20, 23, 24, 42, 81]

Chaslot, G., Jong, S. de, Saito, J-T., and Uiterwijk, J.W.H.M. (2006b). 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. [15, 26, 105]

Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2007). Progressive Strategies for Monte-Carlo Tree Search. *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)* (ed. P. Wang et al.), pp. 655–661, World Scientific Publishing Co. Pte. Ltd. [32, 41, 43]

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 Heidelberg, Germany. [65]

Chaslot, G.M.J-B., Winands, M.H.M, Szita, I., and Herik, H.J. van den (2008b). Cross-Entropy for Monte-Carlo Tree Search. *ICGA Journal*, Vol. 31, No. 3, pp. 145–156. [53]

Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2008c). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [15, 41, 55, 79]

Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008d). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference* (eds. M. Mateas and C. Darken), pp. 216–217, AAAI Press, Menlo Park, CA, USA. [15]

Chaslot, G.M.J-B., Hoock, J-B., Perez, J., Rimmel, A., Teytaud, O., and Winands, M.H.M. (2009). Meta Monte-Carlo Tree Search for Automatic Opening Book Generation. *Proceedings of the IJCAI'09 Workshop on General Intelligence in Game Playing Agents*, pp. 7–12, Pasadena, CA, USA. [77]

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 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *LNCS*, pp. 1–13, Springer-Verlag, Berlin Heidelberg, Germany. [31, 33, 38, 41, 49]

Chen, Z. (2002). Semi-Empirical Quantitative Theory of Go Part I: Estimation of the Influence of a Wall. *ICGA Journal*, Vol. 25, No. 4, pp. 211–218. [11]

Chen, K-H. (2003). GNU Go Wins 19×19 Go Tournament. *ICGA Journal*, Vol. 26, No. 4, pp. 261–262. [16]

Chen, K-H. and Zhang, P. (2008). Monte-Carlo Go with Knowledge-Guided Simulations. *ICGA Journal*, Vol. 31, No. 2, pp. 67–76. [3, 31]

Ciancarini, P. and Favini, G.P. (2009). Monte Carlo Tree Search Techniques in the Game of Kriegspiel. *International Joint Conference On Artificial Intelligence (IJCAI 2009)*, pp. 474–479. [91]

Coquelin, P-A. and Munos, R. (2007). Bandit Algorithms for Tree Search. *23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007)*, Vancouver, Canada. [21, 42]

Costa, A., Jones, O.D., and Kroese, D.P. (2007). Convergence Properties of the Cross-Entropy Method for Discrete Optimization. *Operations Research Letters*, Vol. 35, No. 5, pp. 573–580. [58]

Coulom, R. (2006). 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, Heidelberg, Germany. [1, 3, 13, 15, 18, 20, 22, 23, 24, 25, 26, 42, 55, 81]

Coulom, R. (2007). Computing "Elo Ratings" of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 199–208. [32, 35, 36, 37, 38, 43, 45, 48, 55]

Dailey, Don (2006). Computer Go Mailing List. http://computer-go.org/pipermail/computer-go/. [22]

Davies, J. (1977). *The Rules and Elements of Go.* Ishi Press, Tokyo, Japan. [8]

Donkers, H.H.L.M. (2003). *Nosce Hostem: Searching with Opponent Models.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [2]

Donninger, C. and Lorenz, U. (2006). Innovative Opening-Book Handling. *Advances in Computer Games Conference (ACG 2005)* (eds. H.J. van den Herik, S-C. Hsu, T-S. Hsu, and H.H.M.L. Donkers), Vol. 4250 of *LNCS*, pp. 1–10, Springer-Verlag, Berlin Heidelberg, Germany. [78]

Donninger, C., Kure, A., and Lorenz, U. (2004). Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program. *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, IEEE Computer Society. [4, 65]

Enzenberger, M. and Müller, M. (2009). Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search. Technical Report TR 09, No. 08, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. [26, 83]

Enzenberger, M. and Müller, M. (2010). A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm. *Advances in Computer Games (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *LNCS*, pp. 14–20, Springer-Verlag, Berlin Heidelberg, Germany. [75]

Fairbairn, J. (2000). Go Census. *Mind Sports Zine.* http://www.msoworld.com/mindzine/news/orient/go/special/census.html. [8]

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008* (eds. D. Fox and C.P. Gomes), pp. 259–264, AAAI Press. [28, 37, 39, 90]

Fotland, D. (2004). Go Intellect Wins 19×19 Go Tournament. *ICGA Journal*, Vol. 27, No. 3, pp. 169–170. [16]

Fotland, D. (2009). The Many Faces of Go, Version 12. http://www.smart-games.com/manyfaces.html. [26]

Friedenbach, K.J. (1980). *Abstraction Hierarchies: A Model of Perception and Cognition in the Game of Go.* Ph.D. thesis, University of California, Santa Cruz, CA, USA. [13]

Gelly, S. (2007). *Une Contribution à l'Apprentissage par Renforcement; Application au Computer-Go.* Ph.D. thesis, Université Paris-Sud, Paris, France. [26]

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 Press, New York, NY, USA. [23, 33, 34, 38, 50, 51, 79, 88]

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.* [20, 21]

Gelly, S. and Wang, Y. (2007). MoGo Wins 19×19 Go Tournament. *ICGA Journal*, Vol. 30, No. 2, pp. 111–112. [109]

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA. [3, 22, 26, 31, 33, 38, 55]

Gelly, S., Hoock, J-B., Rimmel, A., Teytaud, O., and Kalemkarian, Y. (2008). The Parallelization of Monte-Carlo Planning - Parallelization of MC-Planning. *Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization (ICINCO 2008)* (eds. J. Filipe, J. Andrade-Cetto, and J-L. Ferrier), pp. 244–249, INSTICC Press. [75]

Ginsberg, M.L. (1999). GIB: Steps Toward an Expert-Level Bridge-Playing Program. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99* (ed. T. Dean), pp. 584–593, Morgan Kaufmann. [16]

Hart, P.E., Nielson, N.J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernatics*, Vol. SSC-4, No. 2, pp. 100–107. [2, 16]

Heinz, E.A. (2001). Self-play, Deep Search and Diminishing Returns. *ICGA Journal*, Vol. 24, No. 2, pp. 75–79. [65]

Helmstetter, B. (2007). *Analyses de Dépendances et Méthodes de Monte-Carlo dans les Jeux de Réflexion.* Ph.D. thesis, Université de Paris 8, Paris, France. In French. [16]

Herik, H.J. van den and Winands, M.H.M. (2008). Proof-Number Search and Its Variants. *Oppositional Concepts in Computational Intelligence* (eds. H.R. Tizhoosh and M. Ventresca), Vol. 155 of *Studies in Computational Intelligence*, pp. 91–118. Springer. [2]

Huang, S-C., Coulom, R., and Lin, S-S. (2010). Monte-Carlo Simulation Balancing in Practice. *Computers and Games (CG 2010).* To appear. [37, 39]

Hunter, D.R. (2004). MM Algorithms for Generalized Bradley-Terry Models. *The Annals of Statistics*, Vol. 32, No. 1, pp. 384–406. [36]

Hu, J. and Hu, P. (2009). On the Performance of the Cross-Entropy Method. *Proceedings of the 2009 Winter Simulation Conference*, pp. 459–468. [64]

Jong, S. de, Roos, N., and Sprinkhuizen-Kuyper, I. (2005). Evolutionary Planning Heuristics in Production Management. *Proceedings of the Seventeenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2005)* (eds. K. Verbeeck, K. Tuyls, A. Nowé, B. Manderick, and B. Kuijpers), pp. 96–103. [105, 106]

Junghanns, A. and Schaeffer, J. (1997). Search versus Knowledge in Game-Playing Programs Revisited. *IJCAI-97*, pp. 692–697. [65]

Karapetyan, A. and Lorentz, R.J. (2006). Generating an Opening Book for Amazons. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N. Netanyahu), Vol. 3846 of *LNCS*, pp. 13–24, Springer-Verlag, Berlin Heidelberg, Germany. [4, 77, 78]

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

Kloetzer, J. (2010). *Monte-Carlo Techniques: Applications to the Game of the Amazons.* Ph.D. thesis, Japan Advanced Institute of Science and Technology, Kanazawa, Japan. [26]

Kloetzer, J., Iida, H., and Bouzy, B. (2009). Playing Amazons Endgames. *ICGA Journal*, Vol. 32, No. 3, pp. 140–148. [26]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [2, 10, 16, 23, 54]

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 *Lecture Notes in Artificial Intelligence*, pp. 282–293. [1, 3, 13, 15, 18, 20, 21, 23, 27, 42, 54, 77, 79]

Kocsis, L., Szepesvári, C., and Winands, M.H.M. (2006). RSPSA: Enhanced Parameter Optimisation in Games. *Advances in Computer Games Conference (ACG 2005)* (eds. H.J. van den Herik, S-C. Hsu, T-S. Hsu, and H.H.M.L. Donkers), Vol. 4250 of *LNCS*, pp. 39–56, Springer-Verlag, Berlin Heidelberg, Germany. [54]

Korschelt, O. (1880). Das Japanisch-Chinesische Spiel Go, ein Konkurrent des Schach. *Mitteilungen der Deutschen Gesellschaft für Natur und Völkerkunde*, Vol. 3. In German. [7]

Kullback, S. (1959). *Information Theory and Statistics.* John Wiley and Sons, NY, USA. [58]

Lasker, E. (1934). *Go and Go-Moku: Oriental Board Games.* Dover Publications, Inc., New York, NY, USA. [7]

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. [26]

Lichtenstein, D. and Sipser, M. (1980). Go is Polynomial-Space Hard. *Journal of the ACM*, Vol. 27, No. 2, pp. 393–401. [9]

Lincke, T. (2001). Strategies for the Automatic Construction of Opening Books. *Computers and Games (CG 2000)* (eds. T.A. Marsland and I. Frank), Vol. 2063 of *LNCS*, pp. 74–86, Springer Verlag, Berlin Heidelberg, Germany. [4, 77, 78, 83]

Lishout, F. van, Chaslot, G.M.J-B., and Uiterwijk, J.W.H.M. (2007). Monte-Carlo Tree Search in Backgammon. *Proceedings of the Computer Games Workshop 2007 (CGW 2007)* (eds. H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd), pp. 175–184, Universiteit Maastricht, Maastricht, The Netherlands. [27]

Liu, J.S. (2002). *Monte Carlo Strategies in Scientific Computing.* Springer, New York, NY, USA. [1]

Lorentz, R.J. (2008). Amazons Discover Monte-Carlo. *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. 13–24, Springer-Verlag, Berlin Heidelberg, Germany. [26, 27]

Luckhardt, C. and Irani, K. (1986). An Algorithmic Solution of N-Person Games. *Proceedings AAAI-86*, pp. 158–162. [27]

Marsland, T.A. (1983). Relative Efficiency of Alpha-Beta Implementations. *Proceedings of the 8^{th} International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 763–766, Karlsruhe, Germany. [2]

Marsland, T.A. and Björnsson, Y. (2001). Variable-Depth Search. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 9–24. Universiteit Maastricht, Maastricht, The Netherlands. [2]

Masayoshi, S. (2005). *A Journey in Search of the Origins of Go.* Yutopian Enterprises. [7]

Mesmay, F., 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*, pp. 729–736, ACM, New York, NY, USA. [26]

Metropolis, N. (1985). Monte Carlo: In the Beginning and Some Great Expectations. *Proceedings of Joint Los Alamos National Laboratory-Commissariat à l'Energie Atomique Meeting*, Vol. 240 of *Lecture Notes in Physics*, pp. 62–70. Springer, New York, NY, USA. [1]

Muehlenbein, H. (1997). The Equation for Response to Selection and its Use for Prediction. *Evolutionary Computation*, Vol. 5, No. 3, pp. 303–346. [4, 34, 53, 54]

Müller, M. (1997). Playing it Safe: Recognizing Secure Territories in Computer Go by Using Static Rules and Search. *Game Programming Workshop in Japan*, Vol. 97, pp. 80–86. [10]

Müller, M. (2002). Computer Go. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 145–179. [4, 9, 26]

Neumann, J. von (1928). Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, Vol. 100, No. 1, pp. 295–320. [2]

Peret, L. and Garcia, F. (2004). On-line Search for Solving Markov Decision Processes via Heuristic Sampling. *ECAI 2004* (eds. R.L. de Mantaras and L. Saitta), Vol. 16, pp. 530–534, IOS Press. [27]

Persson, M. (2006). Lazy Evaluation in Monte Carlo/Alpha Beta Search for Viking4. Computer Go mailing list. http://computer-go.org/pipermail/computer-go/2006-July/005709.html. [17]

Plaat, A. (1996). *Research Re: Search & Re-search.* Ph.D. thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Rotterdam, The Netherlands. [2]

Ralaivola, L., Wu, L., and Baldi, P. (2005). SVM and Pattern-Enriched Common Fate Graphs for the Game of Go. *ESANN 2005*, pp. 485–490. [11]

Reinefeld, A. (1983). An Improvement to the Scout Search Tree Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14. [2]

Remus, H. (1962). Simulation of a Learning Machine for Playing Go. *Proceedings of IFIP Congress 1962*, pp. 428–432, North-Holland Publishing Company. [12]

Richards, N., Moriarty, D.E., and Miikkulainen, R. (1998). Evolving Neural Networks to Play Go. *Applied Intelligence*, Vol. 8, No. 1, pp. 85–96. [13]

Rimmel, A. (2009). *Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm.* Ph.D. thesis, Université Paris-Sud, Paris, France. [31, 33]

Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, Vol. 58, No. 5, pp. 527–535. [20]

Rolet, P. and Teytaud, O. (2010). Adaptive Noisy Optimization. *Applications of Evolutionary Computation*, Vol. 6024 of *LNCS*, pp. 592–601, Springer-Verlag, Berlin Heidelberg, Germany. [64]

Rubinstein, R.Y. (1999). The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, Vol. 1, No. 2, pp. 127–190. [4, 53, 54, 56, 57]

Russell, S.J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [1, 2]

Ryder, J. (1971). *Heuristic Analysis of Large Trees as Generated in the Game of Go.* Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, USA. [12]

Samuel, A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 211–229. [12]

Schadd, M.P.D., Winands, M.H.M., Herik, H.J. van den, Chaslot, G.M.J-B., and Uiterwijk, J.W.H.M. (2008a). Single-Player 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. 1–12, Springer-Verlag, Berlin Heidelberg, Germany. [26]

Schadd, M.P.D., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bergsma, M.H.J. (2008b). Best Play in Fanorona Leads to Draw. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 369–384. [3]

Schaeffer, J., Hlynka, M., and Jussila, V. (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. *International Joint Conference on Artifical Intelligence (IJCAI)*, pp. 529–534. [54]

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]

Schraudolph, N.N., Dayan, P., and Sejnowski, T.J. (1993). Temporal Difference Learning of Position Evaluation in the Game of Go. *Advances in Neural Information Processing Systems*, Vol. 6, pp. 8–17. [13]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275. [12]

Sharma, S., Kobti, Z., and Goodwin, S. (2008). Knowledge Generation for Improving Simulations in UCT for General Game Playing. *AI 2008: Advances in Artificial Intelligence*, Vol. 5360 of *LNCS*, pp. 49–55, Springer-Verlag, Berlin Heidelberg, Germany. [37, 90]

Sheppard, B. (2002). *Towards Perfect Play of Scrabble*. Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [16]

Silver, D. (2008). Re: Computer-Go Digest, Vol 43, Issue 8. http://computer-go.org/pipermail/computer-go/2008-February/014093.html. [51]

Silver, D. and Tesauro, G. (2009). Monte-Carlo Simulation Balancing. *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 945–952, ACM. [37, 39]

Silver, D., Sutton, R.S., and Müller, M. (2007). Reinforcement Learning of Local Shape in the Game of Go. *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pp. 1053–1058. [50]

Smith, S.J.J., Nau, D.S., and Throop, T.A. (1998). Computer Bridge - A Big Win for AI Planning. *AI Magazine*, Vol. 19, No. 2, pp. 93–106. [16]

Stockman, G.C. (1979). A Minimax Algorithm better than Alpha-Beta? *Artificial Intelligence*, Vol. 12, No. 2, pp. 179–196. [2]

Sturtevant, N.R. (2008). An Analysis of UCT in Multi-Player Games. *ICGA Journal*, Vol. 31, No. 4, pp. 195–208. [27]

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. 201–208, AAAI Press / The MIT Press. [27]

Sutton, R.S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine learning*, Vol. 3, No. 1, pp. 9–44. [34]

Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. MIT Press Cambridge, MA, USA. [35, 53, 88]

Szita, I., Chaslot, G.M.J-B., and Spronck, P. (2010). Monte-Carlo Tree Search in Settlers of Catan. *Advances in Computer Games (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *LNCS*, pp. 21–32, Springer-Verlag, Berlin Heidelberg, Germany. [15, 28]

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. [54]

Tesauro, G. and Galperin, G.R. (1997). On-line Policy Improvement using Monte-Carlo Search. *Advances in Neural Information Processing Systems*, Vol. 9, pp. 1068–1074. [16, 17]

Teytaud, O. (2008). Computers vs Humans in Games: MoGo vs. Motoki Noguchi in 9×9 Go. http://www.lri.fr/∼teytaud/crClermont/cr.pdf. [84]

Thomsen, T. (2000). Lambda-Search in Game Trees with Application to Go. *ICGA Journal*, Vol. 23, No. 4, pp. 203–217. [2]

Tromp, J. and Farnebäck, G. (2007). Combinatorics of Go. *Proceedings of the 5th International Conference on Computer and Games* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83, Springer-Verlag, Heidelberg, Germany. [9]

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. [2, 49]

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B.V. Bowden), pp. 286–297, Pitman Publishing, London, England. [12]

Watson, B. (1989). *The Tso Chuan: Selections from China's Oldest Narrative History*. Columbia University Press, New York, NY, USA. [7]

Wedd, N. (2010). Human-Computer Go Challenges. http://www.computer-go.info/h-c/index.html. [109]

Werf, E.C.D. van der (2004). *AI Techniques for the Game of Go*. Ph.D. thesis, Maastricht University, Maastricht, The Netherlands. [8, 10]

Werf, E. van der (2007). Steenvreter Wins 9×9 Go Tournament. *ICGA Journal*, Vol. 30, No. 2, pp. 109–110. [26]

Werf, E.C.D. van der and Winands, M.H.M. (2009). Solving Go for Rectangular Boards. *ICGA Journal*, Vol. 32, No. 2, pp. 77–88. [8, 9]

Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003). Solving Go on Small Boards. *ICGA Journal*, Vol. 26, No. 2, pp. 92–107. [9]

Werf, E.C.D. van der, Winands, M.H.M., Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2005). Learning to Predict Life and Death from Go Game Records. *Information Sciences*, Vol. 175, No. 4, pp. 258–272. [10]

Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2006). Learning to Estimate Potential Territory in the Game of Go. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N. Netanyahu), Vol. 3846 of *LNCS*, pp. 81–96. Springer-Verlag, Berlin Heidelberg, Germany. [10, 45]

Wilcox, B. (1988). Computer Go. *Computer Games* (ed. D.N.L. Levy), Vol. 2, pp. 94–135. Springer, New York, NY, USA. [12]

Winands, M.H.M. (2004). *Informed Search in Complex Games.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [50]

Winands, M.H.M. and Björnsson, Y. (2008). Enhanced Realization Probability Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 329–342. [49, 50]

Winands, M.H.M. and Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. *Advances in Computer Games (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *LNCS*, pp. 33–44, Springer-Verlag, Berlin Heidelberg, Germany. [27, 49, 71]

Winands, M.H.M., Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2002). Temporal Difference Learning and the Neural MoveMap Heuristic in the Game of Lines of Action. *GAME-ON 2002*, pp. 99–103, SCS Europe Bvba, Ghent, Belgium. [54]

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 Heidelberg, Germany. [25, 26, 49]

Wolf, T. (1994). The Program GoTools and its Computer-Generated Tsume Go Database. *Proceedings of the Game Programming Workshop in Japan'94*, pp. 84–96, Hakone, Japan. [10]

Zobrist, A.L. (1969). A Model of Visual Organization for the Game of Go. *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, pp. 103–112, ACM, New York, NY, USA. [11]

Zobrist, A.L. (1970). *Feature Extraction and Representation for Pattern Recognition and the Game of Go.* Ph.D. thesis, University of Wisconsin, Madison, WI, USA. [11, 12]

Appendix A

Production Management Problems

This appendix is based on the following publication:

> G.M.J-B. Chaslot, S. de Jong, J-T. Saito, and J.W.H.M. Uiterwijk (2006b). 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.

In this appendix, we present Production Management Problems (PMPs) as an auxiliary test environment for MCTS. We give a brief overview of PMPs in Section A.1. Next, a formal description is given in Section A.2. We focus on a simplified version of PMPs as proposed by De Jong, Roos, and Sprinkhuizen-Kuyper (2005). It ignores non-determinism and earning additional money that can be utilized to cover the costs of actions.

A.1 Overview

Production Management Problems (PMPs) can be defined as planning problems which require parameter optimization over time and can be addressed by the selection of actions with side effects. They contain the following four elements. First, there is a fixed set of *products* – the *size* of the problem is equivalent to the size of this set. Second, there is a fixed set of *production actions*, which are used to convert certain products into other products or to obtain one or more products. These actions may require time and money. Some actions may also produce money (selling actions). Third, there are *constraints* such as a limited amount of time or money available. Fourth, the goal for a problem solver is to produce certain products, denoted as the *goal products*, as much as possible. This can be achieved by developing an optimal sequence (or plan) of actions, given the available products, actions, constraints and money.

A.2 Formalization

PMPs can be described in a formal way as follows:

- The set of products $P = \{p_1, \ldots, p_n\}$. The *size* of the problem, $|P|$, is denoted by n.
- The possibly infinite set of possible problem states is denoted by S. The problem state $s_t \in S$ at a certain moment in time (starting with s_0) is a tuple $(m_{s_t}, q_{s_t}(p))$. Here, m_{s_t} is the money available in this state, and the function $q_{s_t} : P \to \mathbb{N}$ defines the quantity available for each product in the state s_t.
- $G \subset P$ is the set of goal products. The reward per goal product, $r : P \to \mathbb{R}^+$, is specified by $p \notin G \to r(p) = 0$ and $p \in G \to r(p) > 0$. The total reward in a state s_t can be calculated using the function $R(s_t) = \sum_{p \in G} r(p) \cdot q_{s_t}(p)$. The goal of the problem is to reach a state s_t in which $R(s_t)$ is optimal.
- A denotes the set of actions that enable the transition from one state to another. Here, $c : A \to \mathbb{N}^+$ denotes the cost of action a. Due to this cost, we will have less money to spend in every subsequent state, and thus we ensure that any problem will have a finite number of possible actions. $t : A \to \mathbb{N}$ denotes the time action a takes to complete, assuming discrete time steps. The function $in : A \to \mathcal{P}(P \times \mathbb{N}^+)$ denotes the number of products required to execute a; $out : A \to \mathcal{P}(P \times \mathbb{N}^+)$ denotes the number of products produced by a if it is executed.

Additional constraints. De Jong *et al.* (2005) introduced five constraints in addition to this formalism. These PMPs possess some convenient properties such as a strict ordering of actions and guaranteed solvability. However, PMPs that adhere to these additional constraints are not essentially easier than PMPs that do not adhere to them. The five additional constraints are given below.

1. Every action requires at most two products and produces one or two products. Every product is produced by exactly two actions.
2. Costs and durations of actions can be selected from a limited domain for all $a \in A$: $c(a) \in \{1, 2, 3, 4\}$ and $t(a) = 1$. We limit the domain for the coefficients x for required and produced products p by implying the condition $(p, x) \in in(a) \vee (p, x) \in out(a) \to x \in \{1, 2, 3, 4\}$.
3. Cycles in the product chain are prevented: An action cannot produce products with a lower index than the highest index of its required products, plus one.
4. We define $G = \{p_n\}$ and $r(p_n) = 1$.
5. The problem solver starts with an initial amount of money equal to $20n$ and none of the products present. The number $20n$ is intentionally rather small to keep the problem complexity manageable. Thus, $s_0 = (m_{s_0}, q_0)$ with $m_{s_0} = 20 \cdot n$ and $p \in P \to q_0(p) = 0$.

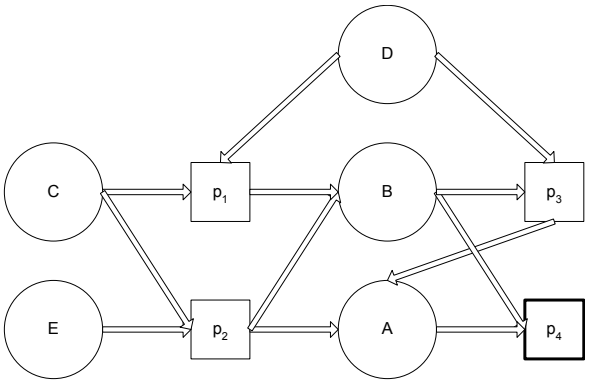


Figure A.1: Example of a PMP that adheres to the formalism and constraints presented in this chapter.

A small example PMP that adheres to the formalism and constraints presented here, is shown in Figure A.1. For legibility, all coefficients have been omitted. Circles represent actions and squares represent products. For instance, action A requires products p_2 and p_3 and produces the product p_4. The goal product is p_4, which can be produced by using various product chains, e.g., by using action C to produce products p_1 and p_2, and then using action B to produce p_3 and p_4. Some other product chains are also possible.

Complexity. Using a mathematical formalization of PMPs, it is possible to reduce the NP-Hard 3-SAT problem to a PMP, proving that PMPs are also NP-Hard.

Appendix B

Milestones of Go Programs against Professionals

This appendix briefly presents the milestones achieved by Go programs against professionals on the 19×19 board (cf. Wedd, 2010). Table B.1 shows for a certain handicap the first victory by a Go program. The level of the Go player is indicated as well. We notice that in the 1990s matches were played against *inseis*, i.e., players that are candidate to become professional. The games were played with handicaps of 11 stones and more. Between humans, games are usually played with handicaps up to 9 stones, which is considered as a strong handicap. However, traditional Go programs did not manage to reach this level, even 11 years after defeating a strong Go player with an 11-stones handicap. Moreover, the results of these non-MCTS programs against human players were quite variable. Human Go players experienced in exploiting the weaknesses of computers programs performed much better. For instance, in 1998 Jean-Loup Gailly, an amateur 5-kyu player, was able to give HANDTALK 20 stones and defeat it.

In the table we see that MOGO was the first program to defeat a Go professional with a handicap of 9 and 6 stones in 2008 and 2009, respectively. CRAZY STONE was the first to defeat a professional 4-dan player with an 8-stones handicap and then a 7-stones handicap in 2008. MOGO defeated a professional 9-dan player with a 7-stones handicap.

Finally, we would like to remark that MOGO defeated the professional 5-dan player Guo Juan on the 9×9 board in the 2007 Computer Olympiad (Gelly and Wang, 2007). It was the first 9×9 game won by a program against a professional without any handicap.

Table B.1: Milestones against human players for different handicaps on the 19×19 board.

Year	Handicap	Human Level	Program	MCTS
1991	17	Insei	GOLIATH	No
1993	15	Insei	HANDTALK	No
1995	13	Insei	HANDTALK	No
1997	11	Insei	HANDTALK	No
2008	9	Professional 8 dan	MOGO	Yes
2008	8	Professional 4 dan	CRAZY STONE	Yes
2008	7	Professional 4 dan	CRAZY STONE	Yes
2009	7	Professional 9 dan	MOGO	Yes
2009	6	Professional 1 dan	MOGO	Yes

Index

Summary

This thesis studies the use of Monte-Carlo simulations for tree-search problems. The Monte-Carlo technique we investigate is Monte-Carlo Tree Search (MCTS). It is a best-first search method that does not require a positional evaluation function in contrast to $\alpha\beta$ search. MCTS is based on a randomized exploration of the search space. Using the results of previous explorations, MCTS gradually builds a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves. MCTS is a general algorithm and can be applied to many problems. The most promising results so far have been obtained in the game of Go, in which it outperformed all classic techniques. Therefore Go is used as the main test domain.

Chapter 1 provides a description of the search problems that we aim to address and the classic search techniques which are used so far to solve them. The following problem statement guides our research.

> **Problem statement**: *How can we enhance Monte-Carlo Tree Search in such a way that programs improve their performance in a given domain?*

To answer the problem statement we have formulated five research questions. They deal with (1) Monte-Carlo simulations, (2) the balance between exploration and exploitation, (3) parameter optimization, (4) parallelization, and (5) opening-book generation.

Chapter 2 describes the test environment to answer the problem statement and the five research questions. It explains the game of Go, which is used as the test domain in this thesis. The chapter provides the history of Go, the rules of the game, a variety of game characteristics, basic concepts used by humans to understand the game of Go, and a review of the role of Go in the AI domain. The Go programs MANGO and MOGO, used for the experiments in the thesis, are briefly described.

Chapter 3 starts with discussing earlier research about using Monte-Carlo evaluations as an alternative for a positional evaluation function. This approach is hardly used anymore, but it established an important step towards MCTS. Subsequently, a general framework for MCTS is presented in the chapter. MCTS consists of four main steps: (1) In the *selection step* the tree is traversed from the root node until we reach a node, where we select a child that is not part of the tree yet. (2) Next, in the *expansion step* a node is added to the tree. (3) Subsequently, during the *simulation step* moves are played in self-play until the end of the

game is reached. (4) Finally, in the *backpropagation step*, the result of a simulated game is propagated backwards, through the previously traversed nodes.

Each step has a strategy associated that implements a specific policy. Regarding selection, the UCT strategy is used in many programs as a specific selection strategy because it is simple to implement and effective. A standard selection strategy such as UCT does not take domain knowledge into account, which could improve an MCTS program even further. Next, a simple and efficient strategy to expand the tree is creating one node per simulation. Subsequently, we point out that building a simulation strategy is probably the most difficult part of MCTS. For a simulation strategy, two balances have to be found: (1) between search and knowledge, and (2) between exploration and exploitation. Furthermore, evaluating the quality of a simulation strategy has to be assessed together with the MCTS program using it. The best simulation strategy without MCTS is not always the best one when using MCTS. The backpropagation strategy that is the most successful is taking the average of the results of all simulated games made through a node.

Finally, we give applications of MCTS to different domains such as Production Management Problems, Library Performance Tuning, SameGame, Morpion Solitaire, Sailing Domain, Amazons, Lines of Action, Chinese Checkers, Settlers of Catan, General Game Playing, and in particular Go.

The most basic Monte-Carlo simulations consist of playing random moves. *Knowledge* transforms the plain random simulations into more sophisticated *pseudo-random* simulations. This has led us to the first research question.

> **Research question 1**: *How can we use knowledge to improve the Monte-Carlo simulations in MCTS?*

Chapter 4 answers the first research question. We explain two different simulation strategies that apply knowledge: urgency-based and sequence-like simulation. Based on the experience gathered from implementing them in INDIGO and MOGO, respectively, we make the following three recommendations. (1) Avoiding big mistakes is more important than playing good moves. (2) Simulation strategies using sequence-like simulations or patterns in urgency-based simulations are efficient because they simplify the situation. (3) The simulation strategy should not become too stochastic, nor too deterministic, thus balancing exploration and exploitation.

Moreover, we develop the first efficient method for learning automatically the knowledge of the simulation strategy. We proposed to use *move evaluations* as a fitness function instead of learning from the results of simulated games. A coefficient is introduced that enables to balance the amount of exploration and exploitation. The algorithm is adapted from the tracking algorithm of Sutton and Barto. Learning is performed for 9×9 Go, where we showed that the Go program INDIGO with the learnt patterns performed better than the program with expert patterns.

In MCTS, the selection strategy controls the balance between exploration and exploitation. The selection strategy should favour the most promising moves (exploitation). However, less promising moves should still be investigated sufficiently (exploration), because their low scores might be due to unlucky simulations. This move-selection task can be facilitated by applying knowledge. This idea has guided us to the second research question.

> **Research question 2**: *How can we use knowledge to arrive at a proper balance between exploration and exploitation in the selection step of MCTS?*

Chapter 5 answers the second research question by proposing two methods that integrate knowledge into the selection step of MCTS: progressive bias and progressive widening. Progressive bias uses knowledge to direct the search. Progressive widening first reduces the branching factor, and then increases it gradually. We refer to them as "progressive strategies" because the knowledge is dominant when the number of simulations is small in a node, but loses influence progressively when the number of simulations increases.

First, the progressive strategies are tested in MANGO. The incorporated knowledge is based on urgency-based simulation. From the experiments with MANGO, we observe the following. (1) Progressive strategies, which focus initially on a small number of moves, are better in handling large branching factors. They increase the level of play of the program MANGO significantly, for every board size. (2) On the 19×19 board, the combination of both strategies is much stronger than each strategy applied separately. The fact that progressive bias and progressive widening work better in combination with each other shows that they have complementary roles in MCTS. This is especially the case when the board size and therefore branching factor grows. (3) Progressive strategies can use relatively expensive domain knowledge with hardly any speed reduction.

Next, the performance of the progressive strategies in other game programs and domains is presented. Progressive bias increases the playing strength of MOGO and of the Lines-of-Action program MC-LOA, while progressive widening did the same for the Go program CRAZY STONE. In the case of MOGO, progressive bias is successfully combined with RAVE, a similar technique for improving the balance between exploitation and exploration. These results give rise to the main conclusion that the proposed progressive strategies are essential enhancements for an MCTS program.

MCTS is controlled by several parameters, which define the behaviour of the search. Especially the selection and simulation strategies contain several important parameters. These parameters have to be optimized in order to get the best performance out of an MCTS program. This challenge has led us to the third research question.

> **Research question 3**: *How can we optimize the parameters of an MCTS program?*

Chapter 6 answers the third research question by proposing to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM). CEM is related to Estimation-of-Distribution Algorithms (EDAs), a new area of evolutionary computation. The fitness function for CEM measures the winning rate for a batch of games. The performance of CEM with a fixed and variable batch size is tested by tuning 11 parameters in MANGO. Experiments reveal that using a batch size of 500 games gives the best results, although the convergence is slow. A small (and fast) batch size of 10 still gives a reasonable result when compared to the best one. A variable batch size performs a little bit worse than a fixed batch size of 50 or 500. However, the variable batch size converges faster than a fixed batch size of 50 or 500.

Subsequently, we show that MANGO with the CEM parameters performs better against GNU GO than the MANGO version without. In four self-play experiments with different time settings and board sizes, the CEM version of MANGO defeats the default version

convincingly each time. Based on these results, we may conclude that a hand-tuned MCTS-using game engine may improve its playing strength when re-tuning the parameters with CEM.

The recent evolution of hardware has gone into the direction that nowadays personal computers contain several cores. To get the most out of the available hardware one has to parallelize MCTS as well. This has led us to the fourth research question.

Research question 4: *How can we parallelize MCTS?*

Chapter 7 answers the fourth research question by investigating three methods for parallelizing MCTS: leaf parallelization, root parallelization and tree parallelization. Leaf parallelization plays for each available thread a simulated game starting from the leaf node. Root parallelization consists of building multiple MCTS trees in parallel, with one thread per tree. Tree parallelization uses one shared tree from which games simultaneously are played.

Experiments are performed to assess the performance of the parallelization methods in the Go program MANGO on the 13×13 board. In order to evaluate the experiments, we propose the strength-speedup measure, which corresponds to the time needed to achieve the same strength. Experimental results indicate that leaf parallelization is the weakest parallelization method. The method leads to a strength speedup of 2.4 for 16 processor threads. The simple root parallelization turns out to be the best way for parallelizing MCTS. The method leads to a strength speedup of 14.9 for 16 processor threads. Tree parallelization requires two techniques to be effective. First, using local mutexes instead of a global mutex doubles the number of games played per second. Second, the virtual-loss enhancement increases both the games-per-second and the strength of the program significantly. By using these two techniques, we obtain a strength speedup of 8.5 for 16 processor threads.

Modern game-playing programs use opening books in the beginning of the game to save time and to play stronger. Generating opening books in combination with an $\alpha\beta$ program has been well studied in the past. The challenge of generating automatically an opening book for MCTS programs has led to the fifth research question.

Research question 5: *How can we automatically generate opening books by using MCTS?*

Chapter 8 answers the fifth research question by combining two levels of MCTS. The method is called Meta Monte-Carlo Tree Search (Meta-MCTS). Instead of using a relatively simple simulation strategy, it uses an entire MCTS program (MOGO) to play a simulated game. We describe two algorithms for Meta-MCTS: Quasi Best-First (QBF) and Beta-Distribution Sampling (BDS). The first algorithm, QBF, is an adaptation of greedy algorithms that are used for the regular MCTS. QBF favours therefore exploitation. During actual game play we observe that despite the good performance of the opening book, some branches are not explored sufficiently. The second algorithm, BDS, favours exploration. In contrast to UCT, BDS does not need an exploration coefficient to be tuned. The algorithm draws a move according to its likelihood of being the best move (considering the number of wins and losses). This approach created an opening book which is shallower and wider. The BDS book has the drawback to be less deep against computers, but the advantage is

that it stayed longer in the book in official games against humans. Experiments on the Go server CGOS reveal that both QBF and BDS were able to improve MOGO. In both cases the improvement is more or less similar. Based on the results, we may conclude that QBF and BDS are able to generate an opening book which improves the performance of an MCTS program.

The last chapter of the thesis returns to the five research questions and the problem statement as formulated in Chapter 1. Taking the answers to the research questions above into account we see that there are five successful ways to improve MCTS. First, learning from move evaluations improves the knowledge of the simulation strategy. Second, progressive strategies enhance the selection strategy by incorporating knowledge. Third, CEM optimizes the search parameters of an MCTS program in such a way that its playing strength is increased. Fourth, MCTS benefits substantially from parallelization. Fifth, Meta-MCTS generates an opening book that improves the performance of an MCTS program. Yet, we are able to provide additional promising directions for future research. Finally, the question of understanding the nature of MCTS is still open.

Samenvatting

Dit proefschrift bestudeert het gebruik van Monte-Carlo simulaties voor zoekproblemen. De Monte-Carlo techniek die wij onderzoeken is Monte-Carlo Tree Search (MCTS). Het is een *best-first* zoekmethode die in tegenstelling tot het $\alpha\beta$ zoekalgoritme geen positionele evaluatiefunctie vereist. MCTS is gebaseerd op een willekeurige verkenning van de zoekruimte. Met behulp van de resultaten van eerdere verkenningen, bouwt MCTS geleidelijk een spelboom op in het computergeheugen en gaat dan de waarden van de veelbelovende zetten steeds beter schatten. MCTS is een generiek algoritme en kan worden toegepast op veel problemen. De meest veelbelovende resultaten zijn tot dusver verkregen in het spel Go, waarin MCTS beter presteert dan de klassieke technieken. Go wordt daarom gebruikt als testdomein in dit proefschrift.

Hoofdstuk 1 geeft een beschrijving van de zoekproblemen die we beogen aan te pakken en de klassieke zoektechnieken die tot dusver zijn gebruikt om ze op te lossen. De volgende probleemstelling is geformuleerd.

> **Probleemstelling** : *Hoe kunnen we Monte-Carlo Tree Search op zo'n manier verder ontwikkelen dat programma's hun prestaties in een gegeven domein verbeteren?*

Voor de beantwoording van de probleemstelling hebben we vijf onderzoeksvragen geformuleerd. Ze gaan over (1) Monte-Carlo simulaties, (2) de balans tussen exploratie en exploitatie, (3) parameter optimalisatie, (4) parallellisatie, en (5) openingsboek generatie.

Hoofdstuk 2 beschrijft de testomgeving die gebruikt wordt om de probleemstelling en de vijf onderzoeksvragen te beantwoorden. Het geeft een uitleg van het spel Go, dat als testdomein in dit proefschrift wordt gebruikt. Het hoofdstuk geeft de geschiedenis van Go, de regels, verscheidene spelkarakteristieken, enkele basisprincipes, en een beschouwing over de rol van Go in het AI-domein. De Go programma's MANGO en MOGO, die worden gebruikt voor de experimenten, worden kort beschreven.

Hoofdstuk 3 begint met een bespreking van eerder onderzoek over het gebruik van Monte-Carlo evaluaties als een alternatief voor een positionele evaluatiefunctie. Deze aanpak wordt nauwelijks meer gebruikt, maar is een belangrijke stap geweest op weg naar MCTS. Hierna wordt in het hoofdstuk een algemeen raamwerk voor MCTS gepresenteerd. MCTS bestaat uit vier hoofdstappen: (1) In de *selectie stap* wordt de boom vanaf de wortel doorkruist totdat we arriveren in een knoop waar een kind geselecteerd wordt dat nog geen

onderdeel is van de zoekboom. (2) Daarna wordt er in de *expansie stap* een knoop toegevoegd aan de boom. (3) Vervolgens, wordt er gedurende de *simulatie stap* een gesimuleerde partij gespeeld. (4) In de *terugpropagatie stap* wordt dan het resultaat van die gesimuleerde partij verwerkt in de knopen langs het afgelegde pad.

Aan elke MCTS stap is een strategie verbonden dat een specifiek beleid uitvoert. Voor selectie wordt in veel programma's de UCT-strategie gebruikt omdat ze eenvoudig uit te voeren en effectief is. Een standaard selectie strategie, zoals UCT, gebruikt geen domeinkennis. Een eenvoudige en efficiënte strategie voor het expanderen van de boom is om de eerste positie die we tegenkomen in de gesimuleerde partij toe te voegen. Vervolgens wijzen wij erop dat het creeëren van een simulatie strategie waarschijnlijk het moeilijkste onderdeel is in MCTS. Voor een simulatie strategie moeten er twee balansen worden gevonden: (1) tussen zoeken en kennis, en (2) tussen exploratie en exploitatie. Bovendien moet de kwaliteit van een simulatie strategie altijd samen worden geëvalueerd met het MCTS programma waarin het wordt gebruikt. De beste simulatie strategie zonder MCTS is niet altijd de beste met MCTS. De terugpropagatie strategie, die het meest succesvol is, neemt gewoon het gemiddelde over de resultaten van alle gesimuleerde partijen in de desbetreffende knoop.

Tenslotte geven we enige toepassingen van MCTS in verschillende domeinen zoals Productie Management Problemen, Libary Performance Tuning, SameGame, Morpion Solitaire, Sailing Domain, Amazons, Lines of Action, Chinese Checkers, Kolonisten van Catan, General Game Playing, en in het bijzonder Go.

De meest basale Monte-Carlo simulaties bestaan uit het willekeurig spelen van zetten. Het gebruik van kennis kan deze simulaties in meer verfijnd spel transformeren. Dit heeft ons tot de eerste onderzoeksvraag gebracht.

> **Onderzoeksvraag 1**: *Hoe kunnen we kennis gebruiken om Monte-Carlo simulaties in MCTS te verbeteren?*

Hoofdstuk 4 geeft antwoord op de eerste onderzoeksvraag. We leggen twee verschillende simulatie strategieën uit die kennis toepassen: urgentie-gebaseerde en sequentie-gebaseerde simulaties. Op basis van de opgedane ervaringen in INDIGO en MOGO, doen we de volgende drie aanbevelingen. (1) Het vermijden van grote fouten is belangrijker dan het doen van goede zetten. (2) Simulatie strategieën zijn efficiënt als ze de situatie vereenvoudigen zoals in urgentie-gebaseerde of sequentie-gebaseerde simulaties. (3) De simulatie strategie moet niet te stochastisch noch te deterministisch zijn; dus de strategie moet balanceren tussen exploratie en exploitatie.

Verder ontwikkelen we de eerste efficiënte methode voor het automatisch leren van de kennis gebruikt in de simulatie strategie. Wij hebben voorgesteld om zetevaluaties te gebruiken als een fitheidsfunctie in plaats van leren op basis van de resultaten van gesimuleerde spelen. Een coëfficiënt wordt geïntroduceerd die het mogelijk maakt exploratie en exploitatie te balanceren. Het leeralgoritme is een aanpassing van het *tracking* algoritme van Sutton en Barto. De experimenten zijn uitgevoerd voor 9×9 Go, waar we laten zien dat het Go programma INDIGO met de geleerde patronen beter presteert dan het programma gebaseerd op expert patronen.

In MCTS regelt de selectie strategie de balans tussen exploratie en exploitatie. Aan de ene kant moet de selectie strategie zich richten op de veelbelovende zetten (exploitatie).

Aan de andere kant moeten de minder rooskleurige zetten nog steeds voldoende worden onderzocht (exploratie). Deze taak kan worden gefaciliteerd door kennis. Dit idee heeft ons tot de tweede onderzoeksvraag gebracht.

> **Onderzoeksvraag 2**: *Hoe kunnen we gebruik maken van kennis om te komen tot een goede balans tussen exploratie en exploitatie in de selectie stap van MCTS?*

Hoofdstuk 5 geeft antwoord op de tweede onderzoeksvraag door twee technieken voor te stellen die kennis integreren in de selectie stap: *progressive bias* en *progressive widening*. Progressive bias gebruikt kennis om het zoekproces bij te sturen. Op basis van kennis reduceert progressive widening eerst de vertakkingsgraad, om deze vervolgens geleidelijk weer te vergroten. We verwijzen naar deze twee technieken als "progressieve strategieën", omdat de kennis dominant is als het aantal simulaties klein is in een knoop, maar geleidelijk aan invloed verliest als het aantal simulaties toeneemt.

Eerst worden de progressieve strategieën getest in MANGO. De ingebouwde kennis is gebaseerd op de urgentie-gebaseerde simulatie. Op grond van de experimenten met MANGO observeren we het volgende. (1) Progressieve strategieën, die in zich in eerste instantie richten op een klein aantal zetten, zijn beter in het verwerken van een grote vertakkingsgraad. Ze vergroten het spelniveau van het programma MANGO aanzienlijk, voor elke bord grootte. (2) Op het 19×19 bord is de combinatie van beide strategieën veel sterker dan elke strategie afzonderlijk. Het feit dat progressive bias en progressive widening beter werken in combinatie met elkaar laat zien dat ze elkaar aanvullen in MCTS. Dit is vooral het geval wanneer de bord grootte en derhalve de vertakkingsgraad groeit. (3) Progressieve strategieën kunnen gebruik maken van relatief dure domeinkennis bijna zonder de snelheid te verlagen.

Vervolgens worden de prestaties van de progressieve strategieën in andere spelprogramma's en domeinen gepresenteerd. Progressive bias verhoogt de speelsterkte van MOGO en van het Lines-of-Action programma MC-LOA, terwijl progressive widening het Go programma CRAZY STONE verbetert. In het geval van MOGO is progressive bias succesvol gecombineerd met RAVE, een vergelijkbare techniek voor de verbetering van de balans tussen exploitatie en exploratie. Deze resultaten geven aanleiding tot de belangrijkste conclusie dat de voorgestelde progressieve strategieën essentiële verbeteringen zijn voor een MCTS programma.

MCTS wordt gecontroleerd door een aantal parameters, die het zoekgedrag bepalen. Vooral de selectie en simulatie strategieën bevatten een aantal belangrijke parameters. Deze parameters moeten worden geoptimaliseerd om de beste prestaties te krijgen voor een MCTS programma. Deze uitdaging heeft ons gebracht tot de derde onderzoeksvraag.

> **Onderzoeksvraag 3**: *Hoe kunnen we de parameters van een MCTS programma optimaliseren?*

Hoofdstuk 6 geeft antwoord op de derde onderzoeksvraag door voor te stellen om de MCTS zoekparameters te optimaliseren met behulp van een evolutionaire strategie: de *Cross-Entropy Method* (CEM). CEM is gerelateerd aan *Estimation-of-Distribution Algorithms* (EDAs). De fitheidsfunctie voor CEM meet het winspercentage voor een bepaald

aantal (*batch*) partijen. De prestaties van CEM met een vaste batch en een variabele batch worden getest door 11 parameters af te stellen in MANGO. Experimenten tonen aan dat het gebruik van een batch grootte van 500 partijen de beste resultaten geeft, hoewel de convergentie traag is. Een kleine (en snelle) batch grootte van 10 partijen geeft nog steeds een redelijk resultaat in vergelijking met de beste batch grootte. Een variabele grootte presteert iets minder dan een vaste grootte van 50 of 500 partijen. Echter, de variabele batch convergeert sneller dan een vaste batch grootte van 50 of 500 partijen.

Vervolgens laten we zien dat MANGO met CEM parameters beter presteert tegen GNU GO dan de MANGO versie met de oude parameters. In vier zelfspel experimenten met verschillende tijdsinstellingen en bord groottes verslaat de CEM versie van MANGO de standaardversie elke keer overtuigend. Gebaseerd op deze resultaten kunnen we concluderen dat een met de hand afgesteld MCTS programma zijn spelsterkte kan verbeteren door CEM toe te passen.

De recente evolutie van hardware is gegaan in de richting dat tegenwoordig PC's meerdere processorkernen bevatten. Om het maximale uit de beschikbare hardware te halen moet men MCTS parallelliseren. Dit heeft geleid tot de vierde onderzoeksvraag.

Onderzoeksvraag 4: *Hoe kunnen we MCTS parallelliseren?*

Hoofdstuk 7 geeft antwoord op de vierde onderzoeksvraag door het onderzoeken van drie methoden voor de parallellisatie van MCTS: blad-, wortel-, en boomparallellisatie. Bladparallellisatie simuleert voor elke beschikbare processorkern een partij, startend in hetzelfde blad. Wortelparallellisatie bestaat uit het construeren van meerdere MCTS bomen, waarvoor geldt dat elke zoekboom zijn eigen processorkern heeft. Boomparallellisatie maakt gebruik van een gedeelde zoekboom waarin gelijktijdig meerdere simulaties worden gespeeld.

Experimenten worden uitgevoerd om de prestaties van de parallellisatie methoden te beoordelen voor het Go programma MANGO op het 13×13 bord. Om de experimenten te evalueren, introduceren wij de *sterkte-versnelling* maat, die overeenkomt met de hoeveelheid denktijd die nodig is om dezelfde sterkte te bereiken. De experimentele resultaten wijzen erop dat bladparallellisatie de zwakste parallellisatie methode is. De methode leidt tot een sterkte-versnelling van 2,4 voor 16 processorkernen. De eenvoudige wortelparallellisatie blijkt de beste manier om MCTS te parallelliseren. De methode leidt tot een sterkte-versnelling van 14,9 voor 16 processorkernen. Boomparallellisatie vereist twee technieken om effectief te zijn. Ten eerste, het gebruik van lokale *mutexen* in plaats van één globale mutex verdubbelt het aantal gespeelde simulaties per seconde. Ten tweede, de *virtueel-verlies* techniek verhoogt zowel het aantal simulaties als de kracht van het programma aanzienlijk. Door het gebruik van deze twee technieken krijgen we een sterkte-versnelling van 8,5 voor 16 processorkernen.

Moderne spelprogramma's gebruiken een openingsboek om in het begin van het spel tijd uit te sparen en sterker te spelen. Het genereren van een openingsboek voor een $\alpha\beta$ programma is goed bestudeerd. De uitdaging om een openingsboek automatisch te genereren voor een MCTS programma heeft geleid tot de vijfde onderzoeksvraag.

Onderzoeksvraag 5: *Hoe kunnen we automatisch een openingsboek genereren door gebruik te maken van MCTS?*

Hoofdstuk 8 beantwoordt de vijfde onderzoeksvraag door twee niveaus van MCTS te combineren. De methode heet Meta Monte-Carlo Tree Search (Meta-MCTS). In plaats van een relatief eenvoudige simulatie strategie te gebruiken, wordt een volledig MCTS programma (MoGo) gebruikt om een simulatie uit te voeren. We beschrijven twee algoritmen voor Meta-MCTS: *Quasi Best-First* (QBF) en *Beta-Distribution Sampling* (BDS). Het eerste algoritme, QBF, is een aanpassing van *greedy* algoritmen die worden gebruikt voor de reguliere MCTS. QBF bevordert exploitatie. Voor toernooipartijen constateren we dat ondanks de goede prestaties van het openingsboek, sommige openingen niet voldoende zijn onderzocht. Het tweede algoritme, BDS, bevordert exploratie. In tegenstelling tot de selectie strategie UCT, heeft BDS geen exploratie constante die moet worden afgesteld. Het algoritme trekt een zet op basis van de waarschijnlijkheid dat het de beste zet is (rekening houdend met het aantal overwinningen en nederlagen). Deze benadering maakt het openingsboek minder diep maar breder. Het BDS boek heeft als nadeel dat men niet zo lang in het boek blijft tegen computer programma's, maar het voordeel dat men langer in het boek blijk in officiële wedstrijden tegen mensen. Experimenten op de Go server CGOS onthullen dat zowel QBF als BDS het programma MoGo verbeteren. In beide gevallen is de verbetering min of meer vergelijkbaar. Gebaseerd op de resultaten kunnen we concluderen dat QBF en BDS in staat zijn om een openingsboek te genereren dat de prestaties van een MCTS programma verbetert.

In het laatste hoofdstuk keren we terug naar de vijf onderzoeksvragen en de probleemstelling zoals die in hoofdstuk 1 zijn geformuleerd. Rekening houdend met de hierboven gegeven antwoorden op de onderzoeksvragen zien we dat er vijf succesvolle manieren zijn om de prestaties van een MCTS programma te verbeteren. (1) Het leren door middel van zetevaluaties verbetert de kennis van de simulatie strategie. (2) Progressieve strategieën versterken de selectie strategie door kennis te integreren. (3) CEM optimaliseert de zoekparameters van een MCTS programma op een zodanige wijze dat de speelsterkte toeneemt. (4) MCTS profiteert aanzienlijk van parallellisatie. (5) Meta-MCTS genereert een openingsboek dat de prestaties van een MCTS programma verbetert. Hierna geven we veelbelovende richtingen van vervolgonderzoek aan. Tenslotte, de vraag van het begrijpen van de aard van MCTS is nog open.

Curriculum Vitae

Guillaume Chaslot was born in Saint-Etienne, France, on February 7^{th}, 1981. He attended high school, Lycée du Parc, in Lyon and received the diploma Baccalauréat in 1999. Then, he started Classes Préparatoires in the same place, from 1999 to 2002. Thereafter, he was accepted at the graduate engineering school École Centrale de Lille for obtaining a Master in Engineering. In his second year, he did an internship at Paris Descartes University on Computer Go. In the third and last year of the Master Engineering, he studied simultaneously at the Lille University of Science and Technology, for a Master in Artificial Intelligence and Data Mining. After receiving both Master's degrees, he worked as a Ph.D. student (AIO) at the Department of Knowledge Engineering (DKE) at Maastricht University, The Netherlands, since August 2005. In September 2008 he worked as a visiting researcher at the Laboratoire de Recherche en Informatique (LRI) of the Paris-Sud 11 University. The Ph.D. research resulted in several publications and this thesis.

SIKS Dissertation Series

1998

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

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

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

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

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

1999

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

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

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

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

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

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

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

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

2000

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

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

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

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

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

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

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

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

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

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

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

2001

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

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

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

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

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

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

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

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

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

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

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

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

2002

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2003

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*

19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*

20 Madelon Evers (Nyenrode) *Learning from Design: facilitating multidisciplinary design teams*

2005

1 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*

2 Erik van der Werf (UM) *AI techniques for the game of Go*

3 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*

4 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*

5 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*

6 Pieter Spronck (UM) *Adaptive Game AI*

7 Flavius Frasincar (TU/e) *Hypermedia Presentation Generation for Semantic Web Information Systems*

8 Richard Vdovjak (TU/e) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

9 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*

10 Anders Bouwer (UvA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

12 Csaba Boer (EUR) *Distributed Simulation in Industry*

13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*

16 Joris Graaumans (UU) *Usability of XML Query Languages*

17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*

18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*

19 Michel van Dartel (UM) *Situated Representation*

20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*

21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006

1 Samuil Angelov (TU/e) *Foundations of B2B Electronic Contracting*

2 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*

3 Noor Christoph (UvA) *The role of metacognitive skills in learning to solve problems*

4 Marta Sabou (VU) *Building Web Service Ontologies*

5 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*

6 Ziv Baida (VU) *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*

7 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*

8 Eelco Herder (UT) *Forward, Back and Home Again - Analyzing User Behavior on the Web*

9 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*

10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*

11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*

12 Bert Bongers (VU) *Interactivation - Towards an ecology of people, our technological environment, and the arts*

13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*

14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*

15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*

16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*

17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*

18 Valentin Zhizhkun (UvA) *Graph transformation for Natural Language Processing*

19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*

20 Marina Velikova (UvT) *Monotone models for prediction in data mining*

21 Bas van Gils (RUN) *Aptness on the Web*

22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*

23 Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*

24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*

25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*

26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*

27 Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*

28 Borkur Sigurbjornsson (UvA) *Focused Information Access using XML Element Retrieval*

2007

1 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*

2 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*

3 Peter Mika (VU) *Social Networks and the Semantic Web*

4 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach*

5 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance*

6 Gilad Mishne (UvA) *Applied Text Analytics for Blogs*

7 Natasa Jovanovic' (UT) *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings*

8 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*

9 David Mobach (VU) *Agent-Based Mediated Service Negotiation*

10 Huib Aldewereld (UU) *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols*

11 Natalia Stash (TU/e) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*

12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*

13 Rutger Rienks (UT) *Meetings in Smart Environments; Implications of Progressing Technology*

14 Niek Bergboer (UM) *Context-Based Image Analysis*

15 Joyca Lacroix (UM) *NIM: a Situated Computational Memory Model*

16 Davide Grossi (UU) *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems*

17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*

18 Bart Orriens (UvT) *On the development and management of adaptive business collaborations*

19 David Levy (UM) *Intimate relationships with artificial partners*

20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*

21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*

22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*

23 Peter Barna (TU/e) *Specification of Application Logic in Web Information Systems*

24 Georgina Ramírez Camps (CWI) *Structural Features in XML Retrieval*

25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008

1 Katalin Boer-Sorbán (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*

2 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*

3 Vera Hollink (UvA) *Optimizing hierarchical menus: a usage-based approach*

4 Ander de Keijzer (UT) *Management of Uncertain Data - towards unattended integration*

5 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*

6 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*

7 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*

8 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*

9 Christof van Nimwegen (UU) *The paradox of the guided user: assistance can be counter-effective*

10 Wauter Bosma (UT) *Discourse oriented Summarization*

11 Vera Kartseva (VU) *Designing Controls for Network Organizations: a Value-Based Approach*

12 Jozsef Farkas (RUN) *A Semiotically Oriented Cognitive Model of Knowledge Representation*

13 Caterina Carraciolo (UvA) *Topic Driven Access to Scientific Handbooks*

14 Arthur van Bunningen (UT) *Context-Aware Querying; Better Answers with Less Effort*

15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*

16 Henriette van Vugt (VU) *Embodied Agents from a User's Perspective*

17 Martin Op't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*

18 Guido de Croon (UM) *Adaptive Active Vision*

19 Henning Rode (UT) *From document to entity retrieval: improving precision and performance of focused text search*

20 Rex Arendsen (UvA) *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven*

21 Krisztian Balog (UvA) *People search in the enterprise*

22 Henk Koning (UU) *Communication of IT-architecture*

23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*

24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*

25 Geert Jonker (UU) *Efficient and Equitable exchange in air traffic management plan repair using spender-signed currency*

26 Marijn Huijbregts (UT) *Segmentation, diarization and speech transcription: surprise data unraveled*

27 Hubert Vogten (OU) *Design and implementation strategies for IMS learning design*

28 Ildiko Flesh (RUN) *On the use of independence relations in Bayesian networks*

29 Dennis Reidsma (UT) *Annotations and subjective machines- Of annotators, embodied agents, users, and other humans*

30 Wouter van Atteveldt (VU) *Semantic network analysis: techniques for extracting, representing and querying media content*

31 Loes Braun (UM) *Pro-active medical information retrieval*

32 Trung B. Hui (UT) *Toward affective dialogue management using partially observable markov decision processes*

33 Frank Terpstra (UvA) *Scientific workflow design; theoretical and practical issues*

34 Jeroen de Knijf (UU) *Studies in Frequent Tree Mining*

35 Benjamin Torben-Nielsen (UvT) *Dendritic morphology: function shapes structure*

2009

1 Rasa Jurgelenaite (RUN) *Symmetric Causal Independence Models*

2 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*

3 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*

4 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*

5 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality*

6 Muhammad Subianto (UU) *Understanding Classification*

7 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*

8 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*

9 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*

10 Jan Wielemaker (UvA) *Logic programming for knowledge-intensive interactive applications*

11 Alexander Boer (UvA) *Legal Theory, Sources of Law & the Semantic Web*

12 Peter Massuthe (TU/e, Humboldt-Universtät zu Berlin) *Operating Guidelines for Services*

13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*

14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)*

15 Rinke Hoekstra (UvA) *Ontology Representation - Design Patterns and Ontologies that Make Sense*

16 Fritz Reul (UvT) *New Architectures in Computer Chess*

17 Laurens van der Maaten (UvT) *Feature Extraction from Visual Data*

18 Fabian Groffen (CWI) *Armada, An Evolving Database System*

19 Valentin Robu (CWI) *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets*

20 Bob van der Vecht (UU) *Adjustable Autonomy: Controling Influences on Decision Making*

21 Stijn Vanderlooy (UM) *Ranking and Reliable Classification*

22 Pavel Serdyukov (UT) *Search For Expertise: Going beyond direct evidence*

23 Peter Hofgesang (VU) *Modelling Web Usage in a Changing Environment*

24 Annerieke Heuvelink (VU) *Cognitive Models for Training Simulations*

25 Alex van Ballegooij (CWI) *RAM: Array Database Management through Relational Mapping*

26 Fernando Koch (UU) *An Agent-Based Model for the Development of Intelligent Mobile Services*

27 Christian Glahn (OU) *Contextual Support of social Engagement and Reflection on the Web*

28 Sander Evers (UT) *Sensor Data Management with Probabilistic Models*

29 Stanislav Pokraev (UT) *Model-Driven Semantic Integration of Service-Oriented Applications*

30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*

31 Sofiya Katrenko (UvA) *A Closer Look at Learning Relations from Text*

32 Rik Farenhorst and Remco de Boer (VU) *Architectural Knowledge Management: Supporting Architects and Auditors*

33 Khiet Truong (UT) *How Does Real Affect Affect Affect Recognition In Speech?*

34 Inge van de Weerd (UU) *Advancing in Software Product Management: An Incremental Method Engineering Approach*

35 Wouter Koelewijn (UL) *Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling*

36 Marco Kalz (OU) *Placement Support for Learners in Learning Networks*

37 Hendrik Drachsler (OU) *Navigation Support for Learners in Informal Learning Networks*

38 Riina Vuorikari (OU) *Tags and Self-Organisation: A Metadata Ecology for Learning Resources in a Multilingual Context*

39 Christian Stahl (TU/e, Humboldt-Universtät zu Berlin) *Service Substitution – A Behavioral Approach Based on Petri Nets*

40 Stephan Raaijmakers (UvT) *Multinomial Language Learning: Investigations into the Geometry of Language*

41 Igor Berezhnyy (UvT) *Digital Analysis of Paintings*

42 Toine Bogers (UvT) *Recommender Systems for Social Bookmarking*

43 Virginia Nunes Leal Franqueira (UT) *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients*

44 Roberto Santana Tapia (UT) *Assessing Business-IT Alignment in Networked Organizations*

45 Jilles Vreeken (UU) *Making Pattern Mining Useful*

46 Loredana Afanasiev (UvA) *Querying XML: Benchmarks and Recursion*

2010

1 Matthijs van Leeuwen (UU) *Patterns that Matter*

2 Ingo Wassink (UT) *Work flows in Life Science*

3 Joost Geurts (CWI) *A Document Engineering Model and Processing Framework for Multimedia documents*

4 Olga Kulyk (UT) *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments*

5 Claudia Hauff (UT) *Predicting the Effectiveness of Queries and Retrieval Systems*

6 Sander Bakkes (UvT) *Rapid Adaptation of Video Game AI*

7 Wim Fikkert (UT) *Gesture interaction at a Distance*

8 Krzysztof Siewicz (UL) *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments*

9 Hugo Kielman (UL) *Politiële gegevensverwerking en Privacy, Naar een effectieve waarborging*

10 Rebecca Ong (UL) *Mobile Communication and Protection of Children*

11 Adriaan Ter Mors (TUD) *The world according to MARP: Multi-Agent Route Planning*

12 Susan van den Braak (UU) *Sensemaking software for crime analysis*

13 Gianluigi Folino (RUN) *High Performance Data Mining using Bio-inspired techniques*

14 Sander van Splunter (VU) *Automated Web Service Reconfiguration*

15 Lianne Bodenstaff (UT) *Managing Dependency Relations in Inter-Organizational Models*

16 Sicco Verwer (TUD) *Efficient Identification of Timed Automata, theory and practice*

17 Spyros Kotoulas (VU) *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications*

18 Charlotte Gerritsen (VU) *Caught in the Act: Investigating Crime by Agent-Based Simulation*

19 Henriette Cramer (UvA) *People's Responses to Autonomous and Adaptive Systems*

20 Ivo Swartjes (UT) *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative*

21 Harold van Heerde (UT) *Privacy-aware data management by means of data degradation*

22 Michiel Hildebrand (CWI) *End-user Support for Access to Heterogeneous Linked Data*

23 Bas Steunebrink (UU) *The Logical Structure of Emotions*

24 Dmytro Tykhonov (TUD) *Designing Generic and Efficient Negotiation Strategies*

25 Zulfiqar Ali Memon (VU) *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective*

26 Ying Zhang (CWI) *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines*

27 Marten Voulon (UL) *Automatisch contracteren*

28 Arne Koopman (UU) *Characteristic Relational Patterns*

29 Stratos Idreos (CWI) *Database Cracking: Towards Auto-tuning Database Kernels*

30 Marieke van Erp (UvT) *Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval*

31 Victor de Boer (UvA) *Ontology Enrichment from Heterogeneous Sources on the Web*

32 Marcel Hiel (UvT) *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems*

33 Robin Aly (UT) *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval*

34 Teduh Dirgahayu (UT) *Interaction Design in Service Compositions*

35 Dolf Trieschnigg (UT) *Proof of Concept: Concept-based Biomedical Information Retrieval*

36 Jose Janssen (OU) *Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification*

37 Niels Lohmann (TU/e) *Correctness of services and their composition*

38 Dirk Fahland (TU/e) *From Scenarios to components*

39 Ghazanfar Farooq Siddiqui (VU) *Integrative modeling of emotions in virtual agents*

40 Mark van Assem (VU) *Converting and Integrating Vocabularies for the Semantic Web*

41 Guillaume Chaslot (UM) *Monte-Carlo Tree Search*