Appendix II

2011 Super Computer Go: Shih-Chieh Huang's Erica

Introduction by Peter Shotwell

Shih-Chieh Huang ('Aja' to his colleagues in the computer go e-group) received his PhD degree in Computer Science at the National Taiwan Normal University. His go-playing program Erica won the Gold Medal in the 19x19 Go tournament at the 2010 Computer Olympiad, beating such tough opponents as Ojima Yoji and Hideki Kato's Zen, and Dave Fotland's Many Faces of Go.

Aja is a Taiwanese 6-dan Go player who is now a PostDoc Fellow at the University of Alberta in Canada working on computer go MCTS (Monte Carlo Tree Search) with Martin Mueller on Erica and Fuego, Hex (see the Wikipedia article for details—
http://en.wikipedia.org/wiki/Hex_%28board_game%29)
and MoHex with Ryan Hayward (see his impressive litst of publications at
http://webdocs.cs.ualberta.ca/~hayward/publications.html).

Erica's innovations included major improvements in simulation balancing and time management and his PhD Thesis also provides an excellent background and updates events in computer go since the 2010 interviews which were recently published in a revised edition of my first book, Go! More Than a Game*. They are now posted as Appendix I of this new Computer Go article.*

He has no plans at the moment to follow Many Faces of Go *and* Zen *in commercializing Erica (which is the English name of his wife).*

New Heuristics for Monte Carlo Tree Search Applied to the Game of Go

A dissertation proposed

by

Shih-Chieh Huang

to

the Department of Computer Science

and Information Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

National Taiwan Normal University

Taipei, Taiwan, R.O.C.

2011

誌謝

感謝我正式的指導教授林順喜老師。林老師在我念研究所時就開始栽培我，還多次補助我參加電腦奧林匹亞，使我在比賽中累積了許多寶貴的經驗。

這個研究是由 Rémi Coulom 教授所指導的，所以他應該得到我最真誠的感謝。在 2009 年 6 月時，我陷入了博士班生涯的低潮，迷茫於沒有研究方向，於是寫信問他一些關於他論文上的問題。他非常有耐心的回答並鼓勵我向前。從那時候開始我們逐漸形成了一個極有生產力的合作。我們透過 email 與視訊會議的方式討論，Rémi 勤勉的態度以及許多創新的想法，實在給我極大的幫助。

關於我們的圍棋程式 ERICA 的發展，除了 Rémi 之外，還要特別感謝王一早提供了許多有趣的想法，Łukasz Lew 在速度最佳化上的實質幫助，還有加藤英樹慷慨的經驗傳授。

感謝中央研究院的研究員徐讚昇老師，在 2010 年的 UEC Cup 提供我們硬體設備，幫助我們在這個艱難的比賽中贏得了第 3 名。

本研究的成果以及論文的寫作，乃是得益於以下諸多人士的幫助。關於 Simulation Balancing 的研究，感謝 David Silver 給我們的指正與鼓勵，也感謝林中雄先生願意提供我們棋譜士網站中大量的棋譜。感謝 David Fotland 夫婦幫忙逐章修正了許多英文的錯誤。感謝加拿大 Alberta 大學的 Martin Müller 教授與德國 Friedrich-Schiller 大學的 Ingo Althöfer 教授在論文內容上提出許多精闢的見解。感謝我的論文口試委員林順喜教授、許舜欽教授、吳毅成教授、徐讚昇教授與顏士淨教授，他們的批評與指導(尤其是吳毅成教授)幫助這本論文更加完善。

感謝我的家人，特別是我的媽媽以及太太，他們的支持推動我沒有後顧之憂的完成博士學位。作為一個基督徒，我也要感謝神在暗中永不停止的引導與幫助，正如聖經所說『信靠祂的，必不至於羞愧』。

Acknowledgement

Thanks to my official adviser Professor Shun-Shii Lin, whose cultivation was from the start of my master's project. For many times, he funded my participation in the Computer Olympiads, which gave me a great deal of valuable experiences.

This research was supervised by Professor Rémi Coulom, so he deserves the earnest gratitude from my heart of hearts. On June 2009, I was wandering in my Ph.D. career, without any research direction, and turned to ask him some questions about his paper. He answered very patiently and encouraged me to proceed. Since then we gradually formed an extremely productive cooperation. We discussed through emails and video conference. Rémi's diligence and innovative ideas have always been my enormous help.

Toward the development of our Go-playing program ERICA, besides Rémi, thanks to Yizao Wang for providing many interesting ideas, to Łukasz Lew for the speed optimization and to Hideki Kato for generous sharing of his experiences.

Thanks to Professor Tsan-Sheng Hsu, Research Fellow of Academia Sinica in Taiwan, who kindly provided us the hardware resources for the 2010 UEC Cup so that we could win 3rd place in this tough competition.

The result of this research and the writing of this dissertation benefitted from the people listed in the following. About the research of Simulation Balancing, thanks to David Silver for his comments and encouragements. Thanks to Lin Chung-Hsiung for kindly providing access to the game database of web2go web site. Thanks to David and Wendy Fotland for correcting the linguistic errors chapter by chapter. Thanks to Professor Martin Müller from the Alberta University in Canada and Professor Ingo Althöfer from the Friedrich-Schiller University in German for proposing plenty of penetrating ideas about the content. Thanks to the committee of my dissertation

defense, including Professor Shun-Shii Lin, Professor Shun-Chin-Hsu, Professor I-Chen Wu, Professor Tsan-Sheng Hsu and Professor Shi-Jim Yen. Their criticism and instructions, particularly the ones from Professor Wu, helped to improve this dissertation.

Thanks to my family, especially my mother and my wife. Their support drove me to complete my Ph.D. career without any burden. As a Christian, thanks to God for his secret and unstoppable guidance and arrangements, just as what we read in the Bible “he that believes on him shall not be ashamed”.

摘要

電腦圍棋的研究開始於 1970 年，但圍棋程式卻從未曾被人們認為是強大的，直到 2006 年，當「蒙地卡羅樹搜尋」(Monte Carlo Tree Search)與「樹狀結構信賴上界法」(Upper Confidence bounds applied to Trees)出現之後，情況才開始完全不同。「蒙地卡羅樹搜尋」與「樹狀結構信賴上界法」所帶進的革命強而有力到一個地步，人們甚至開始相信，圍棋程式在 10 年或者 20 年之後，將能夠擊敗頂尖的人類棋手。

在本研究中，我們針對「蒙地卡羅樹搜尋」提出一些新的啟發式演算法，主要有兩方面的貢獻。第一個貢獻，是成功的將「模擬平衡化」(Simulation Balancing)應用到 9 路圍棋。「模擬平衡化」是一種用來訓練模擬的參數的演算法。Silver 與 Tesauro 在 2009 年提出這個方法時，只實驗在比較小的盤面上，而我們的實驗結果首先證明了「模擬平衡化」在 9 路圍棋的有效性，具體方法是證明「模擬平衡化」超越了知名的監督式演算法 Minorization-Maximization (MM)大約有 90 Elo 之多。第二個貢獻是針對 19 路圍棋，系統式的實驗了各種不同之時間控制的方法。實驗結果清楚的指明，聰明的時間控制方案可以大大的提高棋力。所有的實驗都是執行在我們的圍棋程式 ERICA，而 ERICA 正是得益於這些啟發式演算法與實驗結果，成功取得了 2010 年電腦奧林匹亞的 19 路圍棋金牌。

關鍵字：人工智慧，圍棋，電腦圍棋，蒙地卡羅樹搜尋，樹狀結構信賴上界法，模擬平衡化，時間控制，Erica。

Abstract

Research into computer Go started around 1970, but the Go-playing programs were never, in a real sense, considered to be strong until the year 2006, when the brand new search scheme Monte Carlo Tree Search (MCTS) and Upper Confidence bounds applied to Trees (UCT) appeared on the scene. The revolution of MCTS and UCT promoted progress of computer Go to such a degree that people began to believe that after ten or twenty years, Go-playing programs will be able to defeat the top human players.

In this research, we propose some new heuristics of MCTS focused on two contributions. The first contribution is the successful application of Simulation Balancing (SB), an algorithm for training the parameters of the simulation, to 9×9 Go. SB was proposed by Silver and Tesauro in 2009, but it was only practiced on small board sizes. Our experiments are the first to demonstrate its effectiveness in 9×9 Go by showing that SB surpasses the well-known supervised learning algorithm Minorization-Maximization (MM) by about 90 Elo. The second contribution is systematic experiments of various time management schemes for 19×19 Go. The results indicate that clever time management algorithms can considerably improve playing strength. All the experiments were performed on our Go-playing program ERICA, which benefitted from these heuristics and the experimental results to win the gold medal in the 19×19 Go tournament at the 2010 Computer Olympiad.

Keywords: Artificial Intelligence, Go, computer Go, Monte Carlo Tree Search (MCTS), Upper Confidence bounds applied to Trees (UCT), Simulation Balancing, Time Management, Erica.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The game of Go is a grand challenge of artificial intelligence. In this dissertation, we investigate some new heuristics of Monte Carlo Tree Search (MCTS) applied to the game of Go.

1.1 Computer Games

Artificial intelligence in games (Herik *et al.*, 2002) has made tremendous progress in the past decades. The theoretical foundation of computer games was laid in 1950, when Claude Shannon published his groundbreaking paper "*Programming a Computer for Playing Chess*" (Shannon, 1950). It was this paper that proposed the well-known search scheme *minimax procedure*, in collaboration with an *evaluation function* for evaluating the terminal positions. Minimax procedure as well as its enhancements (Schaeffer, 1989) such as alpha-beta pruning (Knuth and Moore, 1975), transposition table (Slate and Atkin, 1977), etc, constitute the framework that is still dominant in the area of computer games, particularly computer chess. The rapid and constant development of computer games, from 1950, reached a peak in the year 1997, when the chess-playing super-computer DEEP BLUE (Campbell *et al.*, 2002) built by IBM defeated the world champion Garry Kasparov in a six-game match. This achievement has been regarded as a significant milestone of artificial intelligence. In

spite of the chess-playing programs which have grown to the super-human level, the game of Go is a major challenge that remains open (Burmeister and Wiles, 1995; Bouzy and Cazenave, 2001).

1.2 The Game of Go

1.2.1 History

Go (Chinese:圍棋, Japanese:囲碁, Korean: 바둑) is an ancient board game that originated in China in the far past. According to the generally accepted legends, Go was invented by the Chinese emperor Yao (2337-2258 B.C.) in order to instruct his son Danzhu. In the long history of China, interest and research toward Go were never scarce. The game of Go was developed to an art, deeply mingled with Chinese culture as one of the "Four Arts of the Chinese Scholar", namely Qín (Guqin), Qí (Go), Shū (Chinese calligraphy), Huà (Chinese painting). Moreover, the terminology of Go was largely adopted from Chinese idioms to represent specific conceptions. Two evident examples are *Ladder* (Chinese: 征子, suggesting "a long chase") and *Ko* (Chinese: 劫, implying "infinite misfortune"). Several classical writings investigating the playing skills of Go are still circulated nowadays. For instance, "Mystery of MySter -ies " (Chinese: 玄玄棋經), a well-known Tsumego compilation documented in 1349 A.D., is still popular and studied by Go players.

Go was extensively studied and widely played by the general public after spreading to Japan in the 7th century. On account of the diligent research and continual practice of numerous Japanese top Go players, such as the celebrated Honinbo Shusaku (1829-1862 A.D.), the playing level was raised immensely. Go became "the national game of Japan" (Smith, 1908). It was in Japan that the first professional Go institution was built and a number of formal tournaments were held

annually. Such development in Japan not only popularized Go in Japan itself, but also to other countries and even the western world.

Now, the four leading countries where Go prevails are Korea, China, Japan and Taiwan. However, people that play Go are increasing in other lands such as America and Europe.

1.2.2 Rules

The description in this section is partly extracted from (Jasiek, 1997) and Sensei's Library[1]. Briefly speaking, Go is played by two players, Black (makes the first move) and White, by placing one stone of one's own color in turn on an empty intersection on the board, called *goban*, of 19×19 grids of lines (Figure 1.1). A *move* consists of placing one stone of one's own color on an empty intersection on the board. A player may pass his turn at any time and two consecutive passes end the game.

Beginners usually play on a 9×9 board for the purpose of training. Another board size of public interest is 13×13. 13×13 Go is more interesting than 9×9 Go to most Go players, because the concept of corner and edge is meaningful in 13×13 Go but not in 9×9 Go (Huang and Yen, 2010).

[1] Sensei's Library, http://senseis.xmp.net/.

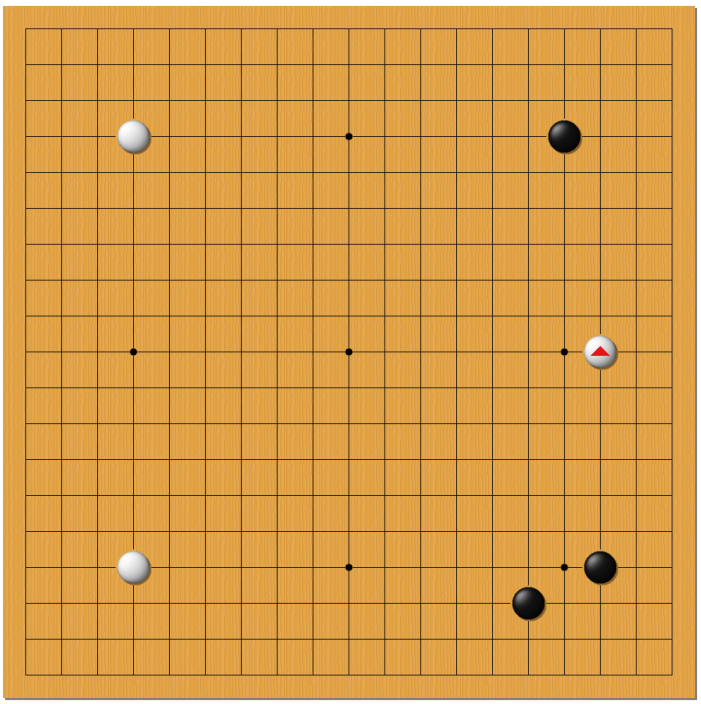

Figure 1.1: A Go board of 19×19 grid of lines, with some played stones.

The complete rules of Go can be summarized to the following four principles.

1. Removing a string without liberty.

A *liberty* is an empty intersection directly adjacent to a stone. A *string* (or *chain*) is a single or upwards of two directly adjacent stones of the same color. Any string without a liberty is *captured* by the opponent. Figure 1.2 gives an example of this principle.

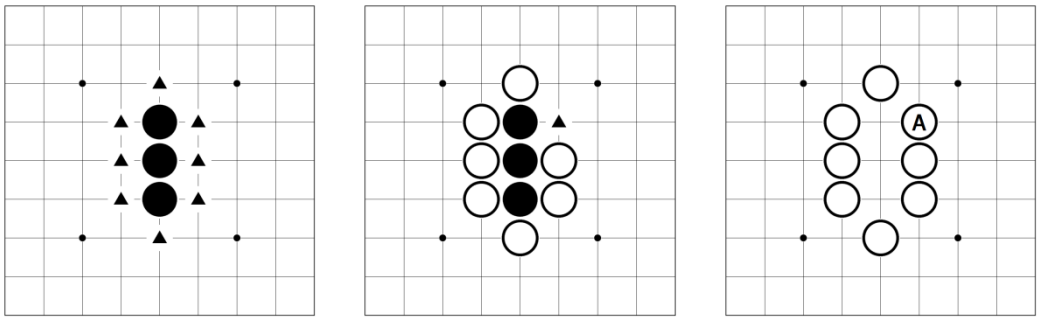

Figure 1.2: An example of "Removing a string without liberty". Left: The string of three stones has 8 liberties (marked by ▲). Middle: The string has only 1 liberty. Right: White captures the string, without a liberty, by playing at point A.

2. Prohibiting suicide.

A move is illegal if this move has no capture and the string will have no liberty after it is played. Figure 1.3 gives an example of this principle.

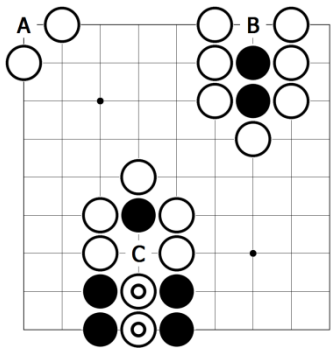

Figure 1.3: An example of "Prohibiting suicide". Point A and B are both Black's illegal moves. Point C is Black's legal move because it can capture White's string marked by ○.

3. Prohibiting repeating positions.

This principle deals with the repetition of a board position in the game of Go. The simplest case is *Ko*. The more general case, of a longer *cycle* (the number of moves) between the repeats, is called *Superko*, which is further defined as *Situational Superko* and *Positional Superko*. Figure 1.4 gives an example of this principle.

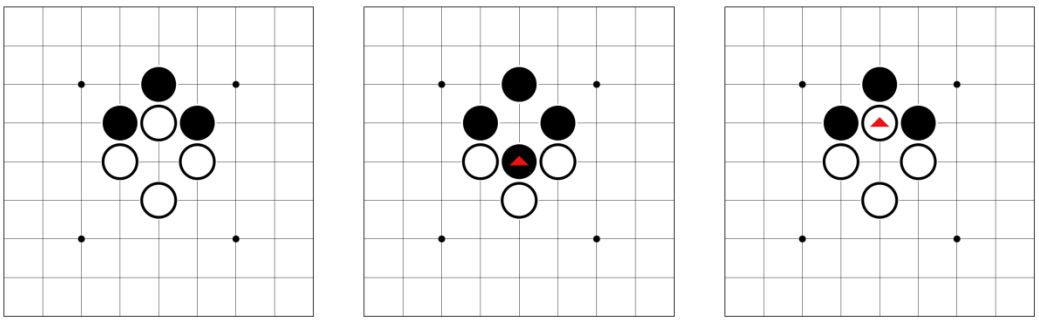

Figure 1.4: An example of "Prohibiting repeating positions". Left: The original position. Middle: Black captures a White's stone. Right: This White's move is illegal, because it recreates a formal position (the one in the left).

4. Winning by more territory.

A player's territory consists of all the board points he has either occupied or

surrounded by his own color. There are mainly two types of scoring: *Territory Scoring* and *Area Scoring*. In Territory Scoring, used in Japanese and Korean rules, each player's score is the sum of her territory plus prisoners (all of the opponent's captured stones during the game). In Area Scoring, used in Chinese rules, each player's score is the sum of her territory plus the number of her stones on the board. In order to compensate for Black's advantage of the first move, White is given a certain points, called *komi*, in scoring. Figure 1.5 gives an example of this principle. Suppose komi=7.5. By Territory Scoring, since (B,W)=(25,25+7.5)=(25,32.5), White wins by 7.5 points. By Area Scoring, since (B,W)=(41,40+7.5)=(41,47.5), White wins by 6.5 points.

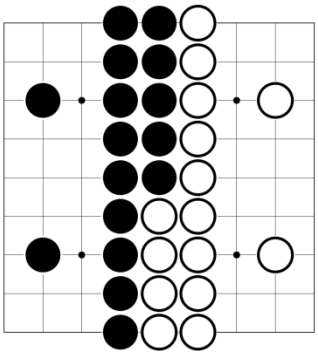

Figure 1.5: An example of "Winning by more territory".

1.3 Computer Go

The rules of Go are rather simple, but its variations are almost numberless. The state-space complexity of Go is about 10^{171} (Tromp and Farnebäck, 2006) and the game-tree complexity of Go is about 10^{360} (Allis, 1994), as shown in Table 1.1. Also, Go was proved to be PSPACE-hard (Lichtenstein and Sipser, 1978). Such high complexity makes Go a grand challenge for artificial intelligence.

Game	State-space complexity	Game-tree complexity	Status
Tic-tac-toe	10^3	10^5	Solved manually
Checkers	10^{20}	10^{31}	Solved in 2007
Chess	10^{47}	10^{123}	Programs > best humans
Chinese chess	10^{48}	10^{150}	Programs ≈ best humans
Shogi	10^{71}	10^{226}	Programs < best humans
Go	10^{71}	10^{360}	Programs << best humans

Table 1.1: Complexities of some well-known games

Research into computer Go started around 1970 (Zobrist, 1970). Although the whole search scheme of minimax procedure in most of the computer games has been a success story, it does not work well for the game of Go. The main problems are the difficulty of designing a feasible evaluation function and the huge search space that is considerably larger than other games (Bouzy and Cazenave, 2001; Müller, 2002). Consequently, before the year 2006, the traditional approach to writing a Go-playing program was to build many handcrafted and independent modules, each realizing a concept of certain Go knowledge, such as group (Chen, 1989) and life-and-death (Chen and Chen, 1999), then integrate them into a single knowledge-based expert system. Developing a competitive Go-playing program in those days, as a result, took a great deal of time and required a great deal of Go knowledge. But thanks to large money prize offered by tournaments such as the Ing Cup and FOST Cup (Fotland, 1996), computer Go became popular since the 1980s. World championship competitions drove steady increasing in playing strength among the top programs, including THE MANY FACES OF GO by David Fotland, GO INTELLECT by Ken Chen, GO++ by Michael Reiss, GOLIATH by Mark Boon, JIMMY by Shi-Jim Yen (Yen, 1999), etc. Among these traditional Go programmers, the most well-known is Chen Zhixing, renowned for developing HANDTALK (afterwards known as GOEMATE), the generally

accepted strongest Go-playing program in the 1990s. In 1997, HANDTALK won a match of 11 handicap stones against a 9-year-old, amateur 6-dan Go player. 11 handicap stones away from amateur 6-dan is approximately amateur 5-kyu level, which is far weaker than the top human level. So, that's why Go-playing programs were never, in a real sense, considered to be strong until the very year 2006, when the brand new search scheme *Monte Carlo Tree Search* (MCTS) (Coulom, 2006) and *Upper Confidence bounds applied to Trees* (UCT) (Kocsis and Szepesv´ari, 2006) appeared on the scene. The revolution of MCTS and UCT promoted progress of computer Go to such a degree that people began to believe that after ten[2] or twenty[3] years, Go-playing programs will be able to defeat the top human players.

1.4 Summary of the Contributions

In this dissertation, we study and investigate several new heuristics of Monte Carlo Tree Search (MCTS) which had been tested in our Go-playing program ERICA. Excluding from the technical and engineering details, our work can be summarized to two contributions.

The first contribution is Monte Carlo Simulation Balancing (SB) applied to 9×9 Go. SB is an algorithm to train the parameters of the simulation. It was proposed in 2009, but only practiced on small board sizes. Our experiments are the first to demonstrate its effectiveness in 9×9 Go by showing that SB surpasses the well-known supervised learning algorithm Minorization-Maximization (MM) by about 90 Elo.

The second contribution is systematic experiments of various time management

[2] The 10 years prediction is maintained by Professor Jaap van den Herik in Tilburg Centre for Creative Computing (TiCC) of the Tilburg University, Netherlands.

[3] One of the proponents of the 20 years prediction is David Fotland, the author of THE MANY FACES OF GO.

schemes for 19×19 Go. The results indicate that clever time management algorithms can considerably improve playing strength.

All the experiments were performed on our Go-playing program ERICA, the winner in the 19×19 Go tournament at the 2010 Computer Olympiad (Fotland, 2010), which is a strong confirmation of the effectiveness of these new heuristics.

1.5 Organization of the Dissertation

The organization of this dissertation is as follows. Chapter 1 gives an introduction of computer games, the game of Go, computer Go, a summary of the contributions in this research and the organization of the dissertation. Chapter 2 presents the background and related work of this research. It introduces Monte Carlo Go, explains Monte Carlo Tree Search (MCTS) and Upper Confidence bounds applied to Trees (UCT), and surveys some of the start-of-the-art Go-playing programs as well as their contributions. Chapter 3 introduces our Go-playing ERICA. We narrate its development history and standings in the tournaments that we have participated, and introduce the framework of the program. Chapter 4 presents our first contribution: applying SB to 9×9 Go. Chapter 5 shows the second contribution: time management schemes utilized in 19×19 Go. Finally, conclusions and proposals for future work are given in Chapter 6.

Chapter 2

Background and Related Work

In this Chapter, we introduce the background and related work of this research. Section 2.1 introduces the progress of Monte Carlo Go until the development of Monte Carlo Tree Search (MCTS) and Upper Confidence bounds applied to Trees (UCT). Section 2.2 explains MCTS and its four stages along with the related work. Section 2.3 explains UCT, which was mainly proposed for the first stage (selection) of MCTS. Finally, Section 2.4 surveys a number of state-of-the-art Go-playing programs as well as their contributions.

2.1 Monte Carlo Go

The idea of Monte Carlo Go was at the very beginning introduced by Brügmann (Brügmann, 1993). In his paper "Monte Carlo Go", Brügmann proposed an algorithm which attempts to find the best move by simulated annealing, without including any Go knowledge, except the rule "do-not-fill-eye" in the simulation. Based on Abramson's expected-outcome model (Abramson, 1990), a position is evaluated by the average score of a certain number of simulations (random games) played from that position on. Remarkably, by this approach, Brügmann's program GOBBLE achieved a playing strength of about 25-kyu on a 9×9 board. In 2003, on the basis of Brügmann's work, Bouzy started to make some experiments on Monte Carlo Go (Bouzy, 2003;

Bouzy and Helmstetter, 2003) and accordingly built a new version of his program INDIGO. In the next few years, Bouzy and Chaslot proceeded to bring forward not a few groundbreaking ideas, such as Bayesian generation of patterns for 19×19 Go (Bouzy and Chaslot, 2005), Progressive Pruning and its variants Miai Pruning (MP) and Set Pruning (SP) (Bouzy, 2005a), History Heuristic and Territory Heuristic (Bouzy, 2005b) and Enhanced 3×3 patterns by reinforcement learning (Bouzy and Chaslot, 2006).

It was based on these preliminary works on Monte Carlo Go that the significant breakthrough of Monte Carlo Tree Search (MCTS) (Coulom, 2006) and Upper Confidence bounds applied to Trees (UCT) (Kocsis and Szepesv´ari, 2006) independently came to realize in 2006.

2.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) (Coulom, 2006) is a kind of best-first search that tries to find the best move and to keep the balance between exploration and exploitation of all moves. MCTS was firstly implemented in CRAZY STONE, the winner in the 9×9 Go tournament at the 2006 Computer Olympiad. Together with the emergence of UCT (Kocsis and Szepesv´ari, 2006), the huge success of MCTS stimulated profound interest among Go programmers. So far, many enhancements of MCTS have been proposed and developed, such as Rapid Action Value Estimation (RAVE), proposed by (Gelly and Silver, 2007; Gelly and Silver, 2011), and progressive bias, proposed by (Chaslot *et al.*, 2007), to strengthen its effect. Plenty of comprehensive studies were also focused on the policy and better quality of the playout (Coulom, 2007; Chaslot *et al.*, 2009; Hendrik, 2010).

MCTS is commonly classified into four stages (Chaslot *et al.*, 2007): selection,

expansion, simulation and backpropagation, as shown in Figure 2.1. The operation of MCTS consists in performing these four stages ever and again as long as there is time left. The repeated four stages of MCTS and the related work are described in the following subsections.

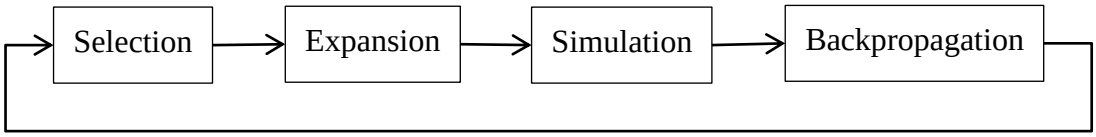

Figure 2.1: The scheme of MCTS.

2.2.1 Selection

The first stage *selection* is intent on selecting one of the children, according to a *selection function* (or *selection formula*), of a given node and repeats from the root node until the end of the tree. Figure 2.2 gives an example. The selection strategy UCT and the various selection functions adopted by different Go-playing programs will be independently investigated in Section 2.3.

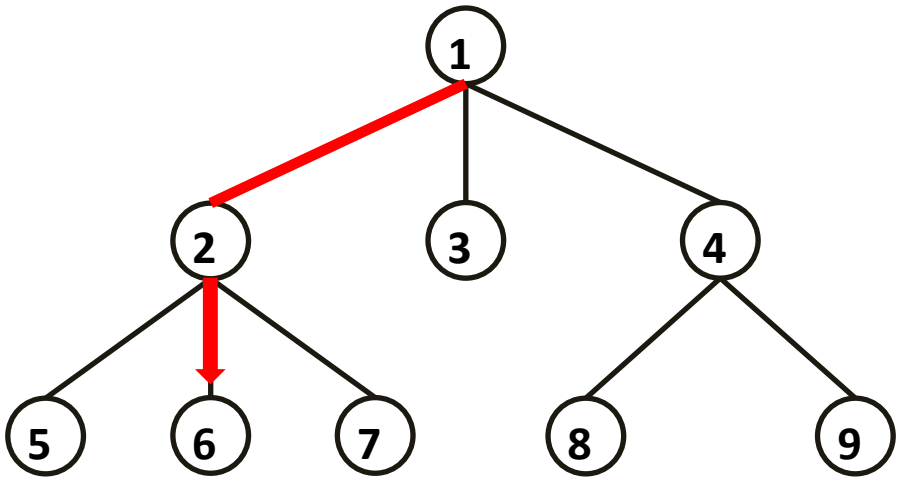

Figure 2.2: The first stage of MCTS: selection. Node 1 (Root) selects Node 2 then Node 2 selects Node 6, which reaches the end of the tree.

2.2.2 Expansion

The second stage *expansion* is to *create* a new child node, corresponding to one of the legal moves of the parent node, and store this new node to the memory to “expand”

the tree. Figure 2.3 gives an example.

The simplest scheme of expansion is to create a new node in the first visit of a leaf node (Coulom, 2006). However, for RAVE, it is necessary to create all the child nodes in preparation for updating the RAVE statistics in the fourth stage backpropagation. To reduce this memory overhead, a popular solution is *delayed node creation*, namely to expand a node in the *n*th (n>1) visit. The NOMITAN team has reported some effective variants of delayed node creation (Yajima *et al.*, 2010).

To raise the performance of RAVE, it is suggested to assign a prior value to each created node (Gelly and Silver, 2007). If many features are taken into account for the computation of a prior value, node creation can be costly and slow. To speed up node creation in multithreaded environment, FUEGO uses an independent, thread-specific, memory array for node creation (Enzenberger and Müller, 2009).

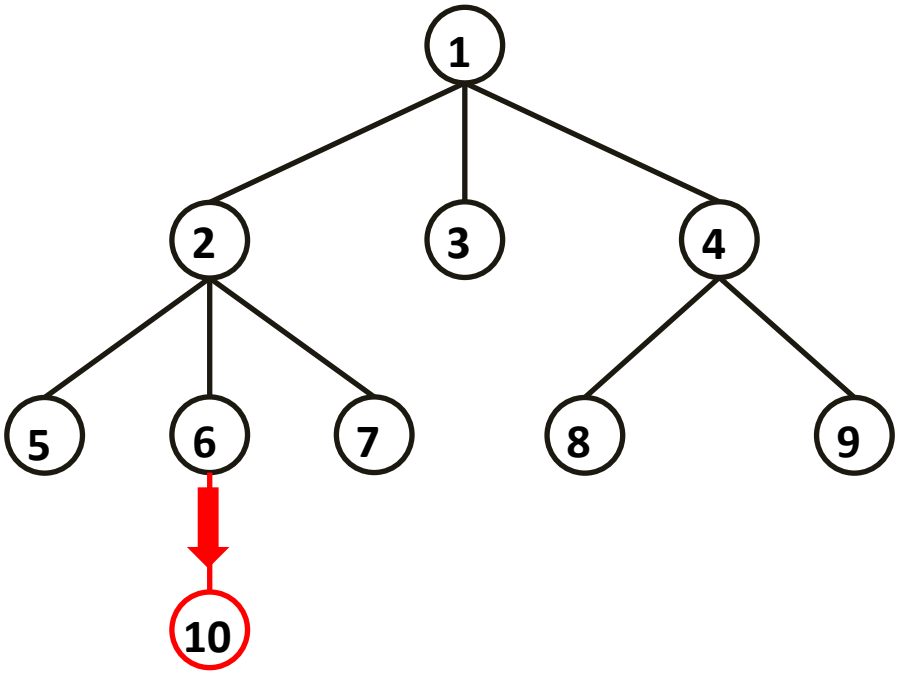

Figure 2.3: The second stage of MCTS: expansion. Node 10, the child node of the leaf node 6, is created and stored to the memory to expand the tree.

2.2.3 Simulation

The third stage *simulation* is to perform a simulation (also called *playout*) from the

position represented by the new created node. For delayed node creation, a simulation is simply performed from the leaf node. In MCTS, a simulation is carried out by Monte Carlo simulation composed of *random* or *pseudo-random* moves. This is the reason for the name "Monte Carlo Go" and "Monte Carlo Tree Search". When the *random game* is completed, the final position is scored[4] to decide the winner. Then the associated outcome 0/1 is passed to the tree to indicate loss/win of this simulation. Figure 2.4 gives an example.

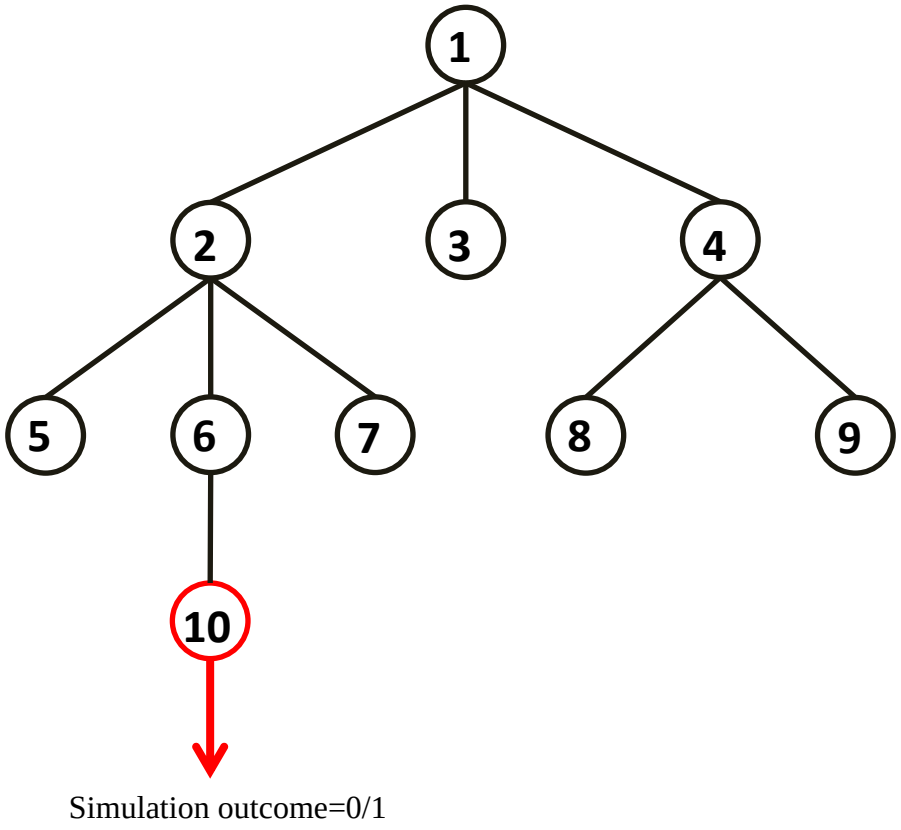

Figure 2.4: The third stage of MCTS: simulation. After Node 10 was created, a Monte Carlo simulation is performed from the position represented by this node. Finally, the outcome 0/1 is returned to indicate loss/win of this simulation.

Simulation is the most crucial step of MCTS. In general, there are mainly two

[4] For the game of Go, a simulation is usually scored by the Chinese rules.

types of Monte Carlo simulation among the current strong Go-playing programs.

The first type, called *Mogo-type*, sequence-like or fixed-sequence simulation (Gelly *et al.*, 2006), also being called *Mogo's magic formula*, is used by MOGO, PACHI, FUEGO, and many strong Go-playing programs. Mogo-type, sequence-like simulation will be further investigated in section 2.4.2.

The second type, called *CRAZY STONE-like*, probabilistic simulation, being called *CRAZY STONE's update formula* (Teytaud, 2011) that allows more flexibility, was proposed by Rémi Coulom (Coulom, 2007) and is being used by CRAZY STONE, AYA and our Go-playing program ERICA. ZEN was reported to use a mixed type of simulation between Mogo-type and CRAZY STONE-like (Yamato, 2011). CRAZY STONE-like simulation will be further investigated in the next chapter.

Recent research on simulation centers on two directions. The first direction is to balance the simulations in the framework of Boltzmann softmax playout policy with the trained feature weights which will be discussed in Chapter 4. The second direction is to improve the playout policy by letting the simulations learn from itself, according to the results of the previous simulations (Drake, 2009; Hendrik, 2010; Baier and Drake, 2010) or the statistical data accumulated in the tree (Rimmel *et al.*, 2010). Such dynamic or adaptive scheme for the simulation is being called *adaptive playout*.

2.2.4 Backpropagation

The fourth stage *backpropagation* is to propagate the simulation outcome 0/1 from the new created node, along with the path decided in the selection stage, to the root node. Each node in this path updates its own statistical data by the simulation outcome. Figure 2.5 gives an example.

In backpropagation, it is possible to update other statistical data by the information collected from the simulation, to obtain a faster estimation of the child

nodes. For instance, with RAVE (Gelly and Silver, 2007) or other kinds of AMAF (All-Moves-As-First) (Brügmann, 1993; Helmbold and Wood, 2009) a node updates all the moves that were played in the tree and the simulation after the position represented by this node.

Some researchers also tried to assign heavier weights to the later simulation outcomes when the tree grows larger (Xie and Liu, 2009), under the assumption that the larger the sub-tree the more promising the simulation outcome.

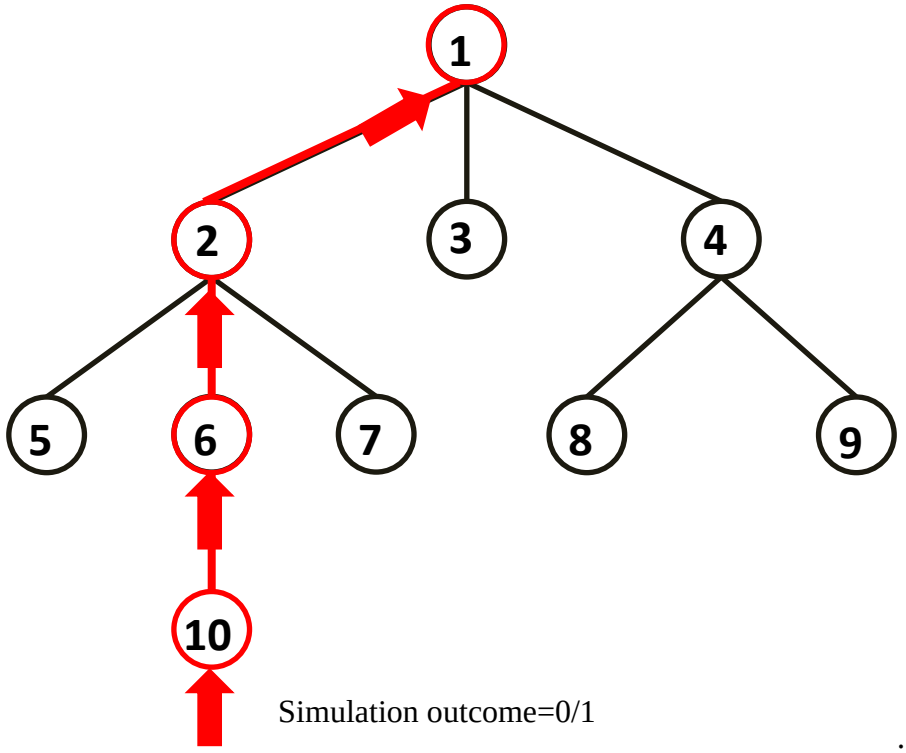

Figure 2.5: The fourth stage of MCTS: backpropagation. The simulation outcome 0/1 is propagated from Node 10 along with the path in the selection stage (Node 6 and Node 2) to the root node (Node 1). Each node updates its own statistical data by the simulation outcome.

A recent topic in backpropagation which calls for much attention is *dynamic komi*. Dynamic komi was proposed to cure the awful performance of MCTS in handicap games on the 19×19 board. The objective under current structure of MCTS is to maximize the winning rate rather than score. So, MCTS works best if the winning rate of the root node is close to 50%, because it is the very occasion that the

simulation outcomes can reflect good and bad moves to the maximum degree. In the case that the winning rate is close to 100% (the case of 0% can be deduced in the same way), MCTS becomes reluctant to explore (since a 0.5 point win or a 20.5 points win are of the same outcome) and incapable to discriminate between good and bad moves. After all, Monte Carlo simulation is more or less biased and far from perfection. This problem becomes particularly apparent in the handicap games against strong human players, as a result of the huge and early advantage offered by the handicap stones.

Figure 2.6 gives a practical example. This position is selected from the exhibition game at the 2010 Computer Olympiad, Rina Fujisawa (White) vs. ERICA (Black), with 6 handicap stones. The stone marked by Δ is the last move. Point A (extending) is a mandatory move for Black in this case but ERICA played at B, a clearly bad move, and showed over 80% winning rate.

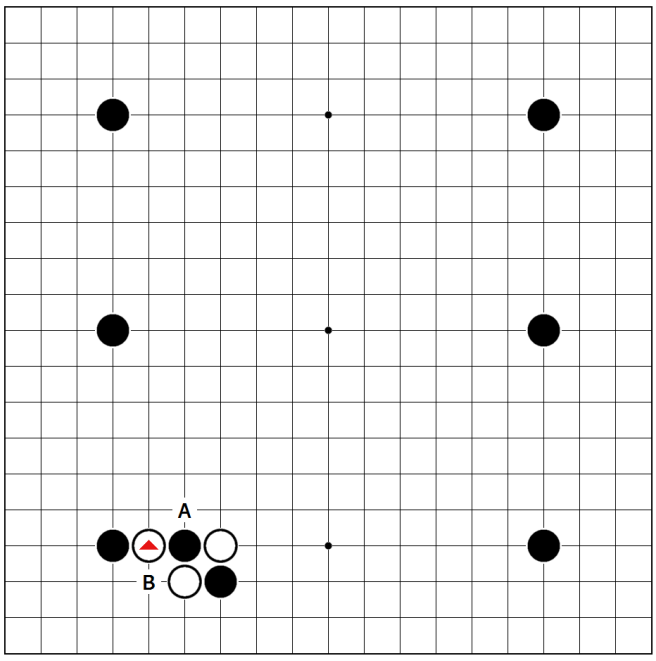

Figure 2.6: The exhibition game at the 2010 Computer Olympiad: Rina Fujisawa (White) vs. ERICA (Black), with 6 handicap stones. White won by resignation.

The main idea of dynamic komi is to adjust the komi value, by the averaged score derived from the last search, in order to shift the winning rate of the root node closer to 50%. ZEN, THE MANY FACES OF GO and PACHI[5] have been reported to benefit from dynamic komi, although each has a different approach.

2.3 Upper Confidence Bound Applied to Trees (UCT)

Upper Confidence bound applied to Trees (UCT) (Kocsis and Szepesv´ari, 2006) is the extension of the UCB1 strategy (Auer *et at.*, 2002) to minimax tree search. The deterministic UCB1 algorithm or policy was designed to solve the Multi-Armed Bandit problem (Auer *et al.*, 1995) and ensures that the optimal machine is played exponentially more than any other machine uniformly when the rewards are in [0,1]. In MCTS, UCT is mainly served as a selection function in the first stage of MCTS and, in general, can be viewed as a special case of MCTS. Under the formulation of UCT, the selection in each node is similar to the Multi-Armed Bandit problem (Coquelin and Munos, 2007). It aims to find the best move and in the meantime keep the balance between the exploration and exploitation of all moves. MOGO was the first Go-playing program that successfully applied UCT (Gelly *et al.*, 2006).

The strategy of UCT is to choose a child node which maximizes the selection formula (2.1):

$$\mathrm{UCT} = \mathrm{v}_{\mathrm{uct}} + \mathrm{C}_{\mathrm{uct}} \times \sqrt{\frac{\log \mathrm{N}_{\mathrm{uct}}}{\mathrm{n}_{\mathrm{uct}}}} \tag{2.1}$$

where v_{uct} is the value of this node, n_{uct} is the visit count of this node and N_{uct} is the visit count of the parent node. C_{uct} is a constant, which has to be tuned empirically.

[5] The author of PACHI, Petr Baudiš, described his successful implementation of dynamic komi in the draft of his paper "Balancing MCTS by Dynamically Adjusting Komi Value".

The latter part of the formula (2.1) is usually called “exploration term” for the purpose of balancing the exploration and exploitation.

The strategy of UCT was very quickly found not feasible to the game of Go, because it requires that each child node must be visited at least once. Even on a 9×9 board, the branching factor, 81 for the node representing the empty position, is still too large to do such complete search. To remedy this flaw, Rapid Action Value Estimation (RAVE) was proposed (Gelly and Silver, 2007; Gelly and Silver, 2011). RAVE is a kind of the heuristic AMAF (All-Moves-As-First) (Brügmann, 1993; Helmbold and Wood, 2009) that updates all the moves which were played in the tree and the simulation after the position represented by this node. The strategy RAVE is to choose a child node which maximizes the selection formula (2.2):

$$\mathrm{RAVE} = \mathrm{v}_{\mathrm{rave}} + \mathrm{C}_{\mathrm{rave}} \times \sqrt{\frac{\log \mathrm{N}_{\mathrm{rave}}}{\mathrm{n}_{\mathrm{rave}}}} \tag{2.2}$$

where v_{rave} is the RAVE value of this node, n_{rave} is the RAVE visit count of this node and N_{rave} is the RAVE visit count of the parent node. C_{rave} is a constant, which has to be tuned empirically.

Blending UCT with RAVE, the strategy UCT-RAVE is to choose a node which maximizes the selection formula (2.3):

$$\mathrm{UCT\text{-}RAVE} = \mathrm{Coefficient} \times \mathrm{RAVE} + (1 - \mathrm{Coefficient}) \times \mathrm{UCT} \tag{2.3}$$

where *Coefficient* is the weight of RAVE (Gelly and Silver, 2007; Silver, 2009).

In the past few years, many efforts have been paid to improve the selection function based on the strategy UCT-RAVE. Some new ideas and the various selection functions adopted by different Go-playing programs are listed as follows.

1. Chaslot *et al.* proposed two progressive strategies for the selection stage and

measured a significant improvement from 25% to 58% (200 games) on their program MANGO against GNU GO 3.7.10 on 13×13 board (Chaslot *et al.*, 2007). The first strategy is *progressive unpruning*, also called progressive widening (Coulom, 2007), which gradually unprunes the child nodes according to their scores computed by the selection function. The other substantial strategy is *progressive bias*, realized as an independent term added behind the selection formula aiming to direct the search according to time-expensive heuristic knowledge.

2. Chaslot *et al.* presented a selection formula combing online learning (bandit module), transient learning (RAVE values), expert knowledge and offline patter-information (Chaslot *et al.*, 2009), which is being used in their program MOGO.
3. Silver, in his Ph.D. dissertation, based on the experiments on MOGO (Silver, 2009), suggested to take off the exploration terms of both UCT and RAVE, namely set C_{uct} and C_{rave} to 0.
4. Rosin proposed a new algorithm PUCB under the assumption that contextual side information is available at the start of the episode (Rosin, 2010).
5. Tesauro *et al.* proposed a Bayesian framework for MCTS that allows potentially much more accurate (Bayes-optimal) estimation of node values and node uncertainties from a limited number of simulation trials (Tesauro *et al.*, 2010).
6. THE MANY FACES OF GO is using the formula (2.4) in collaboration with progressive widening (Fotland, 2011):

$$(1-beta)\times(v_{uct}+\mathrm{C}_{\mathrm{uct}}\times\sqrt{\frac{\log \mathrm{N}_{\mathrm{uct}}}{\mathrm{n}_{\mathrm{uct}}}})+beta\times\mathrm{v}_{\mathrm{rave}}+mfgo\_bias \qquad (2.4)$$

$$beta=\sqrt{\frac{500}{500+3\times\mathrm{N}_{\mathrm{uct}}}}$$

where *mfgo_bias* is unchanging, per move, within a range of about ±2%, based on the quality of the move estimated by the move generator of THE MANY FACES OF GO.

7. AYA is using the formula (2.5) in collaboration with progressive widening (Yamashita, 2011):

$$(1-beta)\times(v_{uct}+\mathrm{C_{uct}}\times\sqrt{\frac{\log \mathrm{N_{uct}}}{\mathrm{n_{uct}}}})+beta\times(\mathrm{v_{rave}}+\mathrm{C_{rave}}\times\sqrt{\frac{\log \mathrm{N_{rave}}}{\mathrm{n_{rave}}}}) \qquad (2.5)$$

$$beta=\sqrt{\frac{100}{100+3\times \mathrm{N_{uct}}}}$$

8. PEBBLES is using the formula (2.6) (Sheppard, 2011):

$$(1-beta)\times qUCT+beta\times qRAVE \qquad (2.6)$$

where *beta* is set according to Silver's dissertation (Silver, 2009). Both *qUCT* and *qRAVE* incorporate exploration terms from the Beta Distribution (Stogin *et al.*, 2010).

9. PACHI is using a formula similar to that of AYA, except that C_{uct} and C_{RAVE} are set to 0 (Baudis, 2011). The "Even game prior" is used to set v_{uct} with 0.5 at *n* playouts, where *n* can be between 7 and 40. Another important prior is "playout policy hinter", which uses the same heuristics (and code) as the playout policy to pick good tree moves.

2.4 State-of-the-Art Go-Playing Programs

In this section, we survey some start-of-the-art Go-playing programs as well as their contributions.

2.4.1 Crazy Stone

CRAZY STONE was created by Rémi Coulom, the inventor of MCTS (Coulom, 2006),

which has been regarded as the most significant contribution to Computer Go in recent years. At the 2006 Computer Olympiad, CRAZY STONE demonstrated the usefulness and effectiveness of MCTS by the overwhelming victory in the 9×9 Go tournament. In this tournament, CRAZY STONE defeated many senior Go-playing programs such as GNU GO, GOKING, JIMMY, etc, and tied with AYA and Go INTELLECT. In the first UEC Cup in 2007, CRAZY STONE won the exhibition match against Kaori Aoba 4p with 7 handicap stones. This game was described as "very beautiful". Figure 2.7 shows the final position of this game. CRAZY STONE finally killed the whole White's big group[6] in the center (marked by ×) and secured a solid win.

The second great contribution of Coulom is the supervised learning algorithm named Minorization-Maximization (MM) for computing the Elo ratings of move patterns (Coulom, 2007), which will be further investigated in Chapter 4. This learning algorithm is still used by some of the top-level Go-playing programs, such as ZEN and AYA.

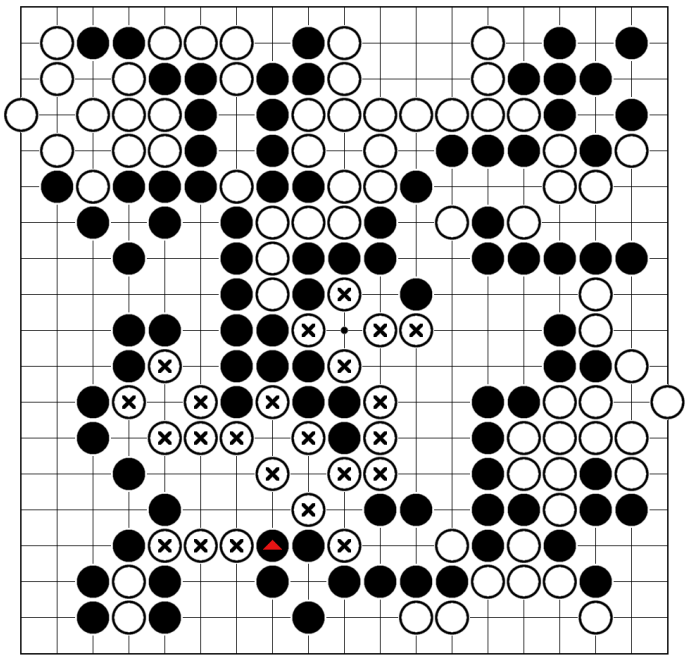

Figure 2.7: The final position of the exhibition match: Kaori Aoba 4p (White) vs. CRAZY

[6] A group consists of one or more loosely connected strings.

STONE (Black), with 7 handicap stones, in the first UEC Cup, 2007. Black won by resignation.

Currently, Coulom is again working on CRAZY STONE after a suspension of about 2 years. Right now, CRAZY STONE is rated 4-dan on the KGS Go Server (KGS) on a 24-core machine (account *CrazyStone*, retrieved at 2011-07-14 T12:12:42+08:00[7]) on the 19×19 board and reached a Bayes-Elo rating (Coulom,2010) of 2914 in Computer Go Server (CGOS) on the 9×9 board (account *bonobot*, retrieved at 2011-07-14 T12:15:34+08:00).

2.4.2 MOGO

MOGO was created in the beginning by Yizao Wang and Gelly Sylvain, supervised by Rémi Munos. Olivier Teytaud took the lead of the "MOGO team" after Yizao Wang and Gelly Sylvain left. There are several important contributions from the MOGO team.

The first and the greatest contribution is applying UCT (Kocsis and Szepesv´ari, 2006), which was invented by Kocsis *et al.* independently at the same time as Coulom's MCTS, to computer Go (Gelly *et al.*, 2006). It is widely maintained that the contributions of CRAZY STONE and MOGO collaboratively enable the Monte Carlo Go programs to be competitive with, and stronger than, the strongest traditional Go-playing programs, such as HANDTALK, THE MANY FACES OF GO and GO INTELLECT.

The second contribution of MOGO team lies in the Monte Carlo part. The earliest creators of MOGO, mainly Sylvain Gelly and Yizao Wang, designed a sequence-like simulation (Gelly *et al.*, 2006) that still has dominant influence on almost all the current strong Go-playing programs. This sequence-like simulation was further

[7] Presented in ISO 8061 date format.

improved by expert knowledge, such as nakade, and heuristics such as “Fill the board” (Chaslot *et al.*, 2009). Figure 2.8 gives the example of sequence-like simulation.

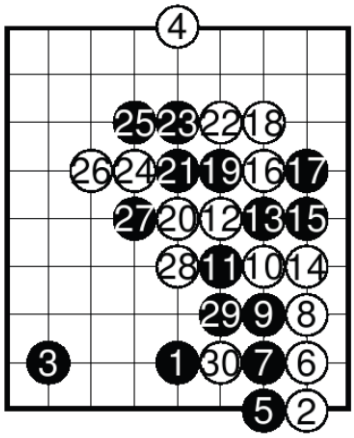

Figure 2.8: An example of sequence-like simulation proposed by MOGO team, cited from the paper “Modification of UCT with Patterns in Monte Carlo Go”.

The main principle of such sequence-like simulation consists in considering the present move by responding to the previous move played by the opponent. Two important responses to the previous move are “save a string by capturing” and “save a string by extending”. For “save a string by capturing”, it means to save the string and put in atari by the previous move, by capturing its directly neighboring opponent string. “Save a string by extending” means to save the string and put in atari by the previous move by extending its liberty. Figure 2.9 gives an example of these responses.

The most powerful part of the sequence-like simulation is considering the 3×3 patterns around the previous move. It is generally stated that the 3×3 patterns designed by Yizao Wang and RAVE are the major factors that enabled MOGO to be the solid strongest Go-playing program in the period of the first half of 2007.

This sequence-like simulation, handcrafted policy was improved by the offline reinforcement learning from games of self-play (Gelly and Silver, 2007). Gelly and Silver reported that this generated policy outperformed both the random policy and

the handcrafted policy by a margin of over 90%.

The third big idea of MOGO team is RAVE (Gelly and Silver, 2007), which is a kind of the heuristic AMAF (All-Moves-As-First) (Brügmann, 1993; Helmbold and Wood, 2009). Presently, RAVE is reported to be utilized in almost every strong Go-playing program. Some authors even reported that RAVE boosts the playing strength of their programs over 200 Elo (Kato, 2008).

Other contributions of MOGO include the parallelization of MCTS (Chaslot *et al.*, 2008; Gelly *et al.*, 2009; Bourki *et al.*, 2010), the never-ending learning algorithms for designing automatically an opening book by MCTS (Chaslot *et al.*, 2009; Audouard *et al.*, 2009; Gaudel *et al.*, 2010) and so on.

MOGOTW, a joint project between the MOGO team and a Taiwanese team, led by Chang-Shin Lee, composed of several Taiwanese universities and organizations, won the first 9×9 game as Black against the top professional Go player Chun-Hsun Chou 9p, the winner of the international professional Go tournament LG Cup 2007, in the Human vs. Computer Go Competition at WCCI 2010. Figure 2.10 shows the final position of this game.

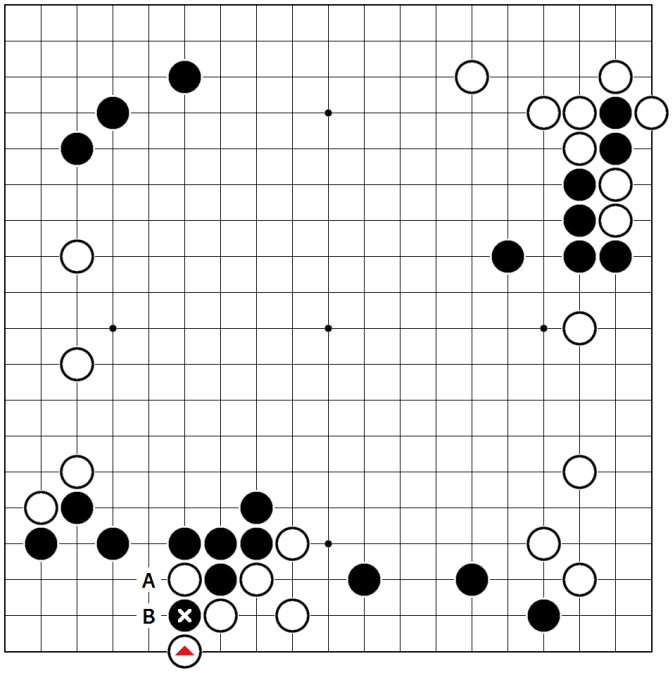

Figure 2.9: An example of "save a string by capturing" and "save a string by extending". The previous move is marked by Δ. The string marked by × was put in atari by the previous move. Point A is "save a string by capturing" and point B is "save a string by extending".

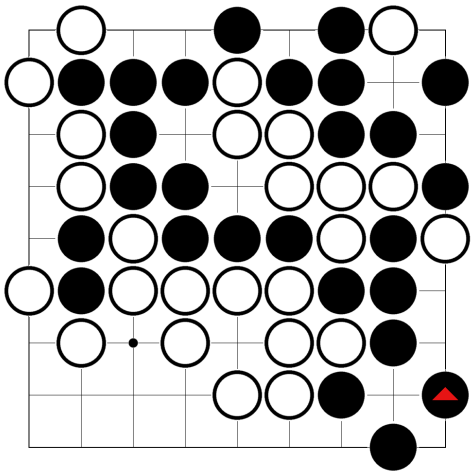

Figure 2.10: The final position of the match: Chun-Hsun Chou 9p (White) vs. MOGOTW (Black), with komi 7.5. The stone marked by Δ is the last move. Black won by resignation.

2.4.3 GNU GO

GNU GO is an open-source Go-playing program authored by many people. The first version of GNU GO was released at March 13th, 1989. It is still the most popular Go-playing program among many internet Go servers and the most common

experimental test bed in the field of computer Go.

Before the rising of MCTS and UCT, GNU GO was among the strongest Go-playing programs. It won the gold medal in the 19×19 Go tournament at the 2003 Computer Olympiad and silver medal in the 9×9 Go tournament at the 2004 Computer Olympiad. On the Bayes-Elo rating system of the Computer Go Server (CGOS), the rating of GNUGO-3.7.10-A0 on the 19×19 board is 1800 and the top ranked program is ZENGG-4X4C-TST, rated 2839 (retrieved at 2011-07-17 T17:22+08:00). This fact shows that in the past seven years, from 2004 to 2011, the improvement of the strongest Go-playing program is at least 1000 Elo.

Figure 2.11 gives the position in the opening stage of the match between JIMMY (White) and GNU GO (Black) in round 1 in the 19×19 tournament at the 2003 Computer Olympiad. This example shows that both GNU GO and JIMMY can play very good pattern shapes in the opening stage.

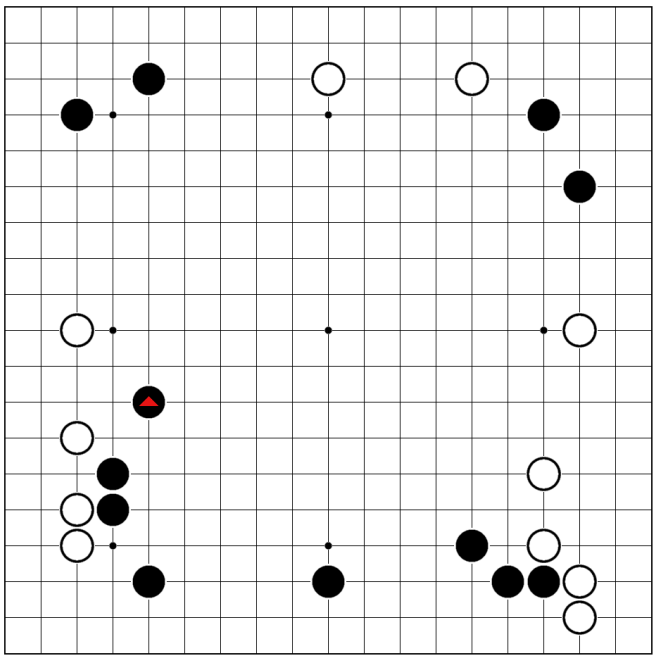

Figure 2.11: Round 1 at the 2003 Computer Olympiad: JIMMY (White) vs. GNU GO (Black), with komi 6.5. The stone marked by Δ is the previous move. Black won by resignation.

2.4.4 FUEGO

FUEGO (Enzenberger *et al.*, 2010) was created by Markus Enzenberger, Martin Müller and Broderick Arneson of the computer Go research group led by Martin Müller at the University of Alberta, Canada.

FUEGO won the first 9×9 game as White against the top professional player Chun-Hsun Chou 9p in the Human vs. Computer Go Competition of 2009 IEEE International Conference on Fuzzy Systems. In 2010, FUEGO, running on a big shared memory machine at IBM with 112 threads (Segal, 2010), won the 4th UEC Cup. Figure 2.12 shows the position in the middle game of the final match between ZEN (White) and FUEGO (Black). FUEGO's previous move, marked by Δ, was a severe and strong cut aiming to kill the White's group marked by ×. After winning this big semeai, Fuego secured the leading to the end of this game and won the 4th UEC Cup.

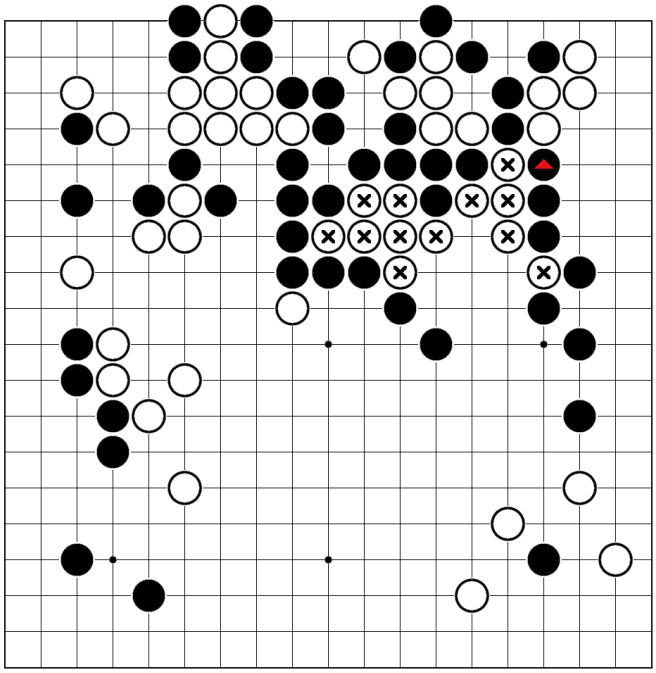

Figure 2.12: A position of the final match in 4th UEC Cup: ZEN (White) vs. FUEGO (Black). Black won by resignation.

Another contribution of FUEGO team is the release of the tool GoGui for

Go-playing program developers. GoGui allows direct communication to a Go engine by a command shell. It also provides a mechanism of automatic playing between any two Go-playing programs through *Go Text protocol* (GTP). Another convenience supplied by GoGui is the visualization of numerous features for the user-specific GTP commands.

2.4.5 The Many Faces of Go

THE MANY FACES OF GO (Fotland, 1993; Fotland, 2002) was developed by David Fotland, an American professional Go programmer who has worked on computer Go for over 25 years. THE MANY FACES OF GO is a commercial Go-playing program. It was among the strongest traditional Go-playing programs, then transformed to a Monte Carlo Go program mixed with the old engine.

THE MANY FACES OF GO has outstanding achievement in tournaments and has been a competitive program since the 1980s. The older version that did not use MCTS won the 21st Century Cup in 2003 and the 1998 Ing Cup World Championship. At the 2008 Computer Olympiad, THE MANY FACES OF GO won both the 9×9 and 19×19 Go tournaments. It also won the gold and bronze medals in the 13×13 Go and 19×19 Go tournaments at the 2010 Computer Olympiad. In the tournaments of KGS Go Server (KGS), THE MANY FACES OF GO has always been a participant at the top of the list.

Figure 2.13 shows the game between ZEN (White) and THE MANY FACES OF GO (Black) in round 7 in the 19×19 Go tournament at the 2010 Computer Olympiad. The stone marked by Δ is the last move when ZEN resigned. In this position, Black can either play at A or B to secure the center group marked by ○. It clearly shows the strong life-and-death and defense capabilities of THE MANY FACES OF GO (Fotland, 2002) under ZEN‘s continuously large-scale, fierce attack toward the center group.

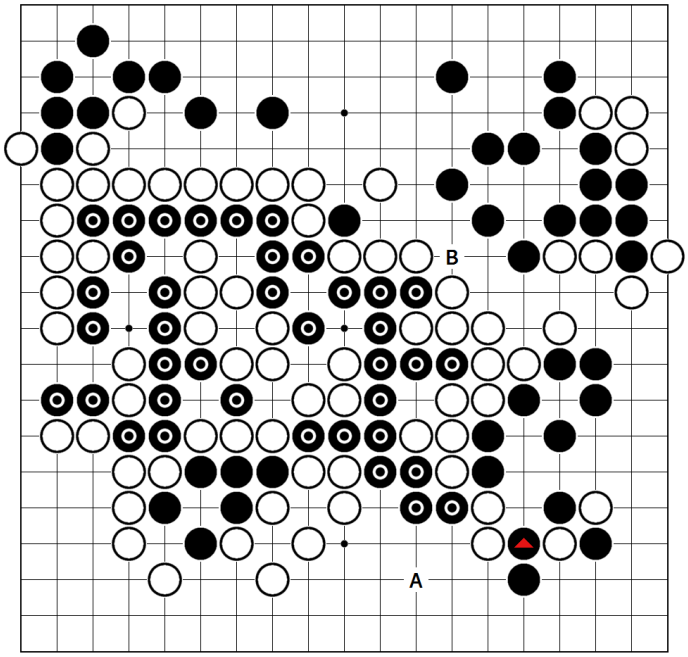

Figure 2.13: The final position of the match in round 7 in the 19×19 Go tournament at the 2010 Computer Olympiad: ZEN (White) vs. THE MANY FACES OF GO (Black). Black won by resignation.

2.4.6 ZEN

ZEN is a Japanese commercial Go-playing program created by Ojima Yoji (nickname *Yamato*), collaborating with Hideki Kato on cluster parallelization. ZEN was the winner in the 19×19 Go tournament at the 2009 Computer Olympiad. It is now doubtlessly the strongest Go-playing program (up to July 13th, 2011). On the KGS Go Server (KGS), ZEN is the only program that stands firm in 5-dan (account *Zen19D*, as shown in Figure 2.14, retrieved at 2011-07-14 T12:53:01+08:00) in blitz games and 4-dan (account *Zen19S*, retrieved at 2011-07-14 T12:53:28+08:00) in longer games, running on a 26-core cluster. It also won all the tournaments of KGS Go Server (KGS) that it had participated up to July, 2011.

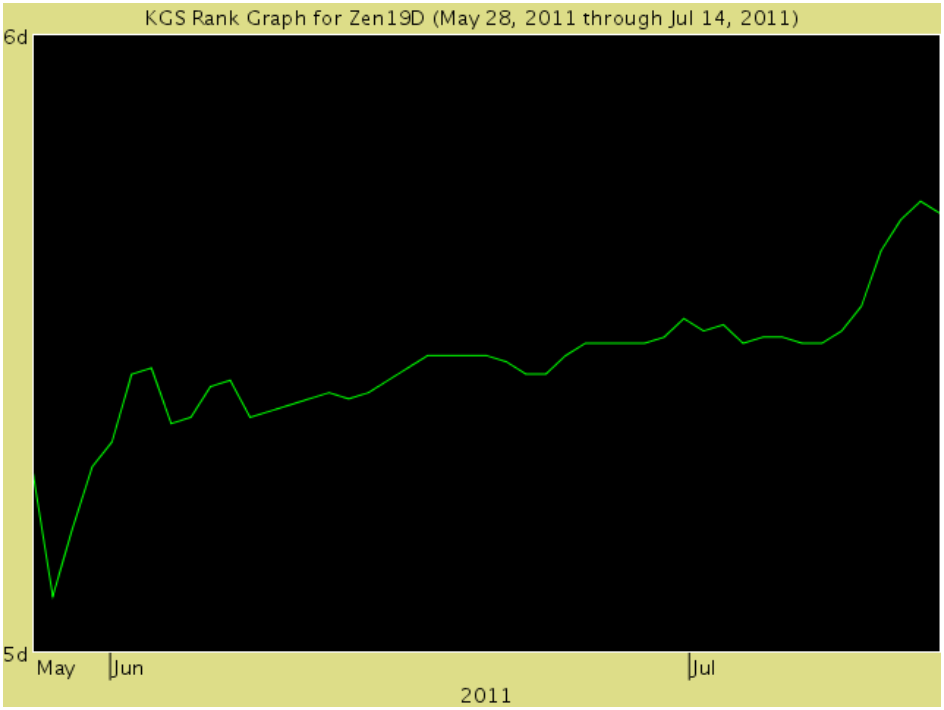

Figure 2.14: KGS Rank Graph for *Zen19D*

Good at fighting has long been the main feature of ZEN. The success story of ZEN clearly demonstrates the efficacy of heavy and informative playouts, RAVE and larger patterns in the tree (Coulom, 2007), which were reported to contribute to ZEN's playing strength.

In the 4th UEC Cup in 2010, ZEN won the exhibition match against Kaori Aoba 4p with 6 handicap stones. Figure 2.15 shows the final position of this game. In this game, ZEN thoroughly showed its strong capabilities of attack through the whole game. It finally killed the White's big group marked × and came off with a great victory.

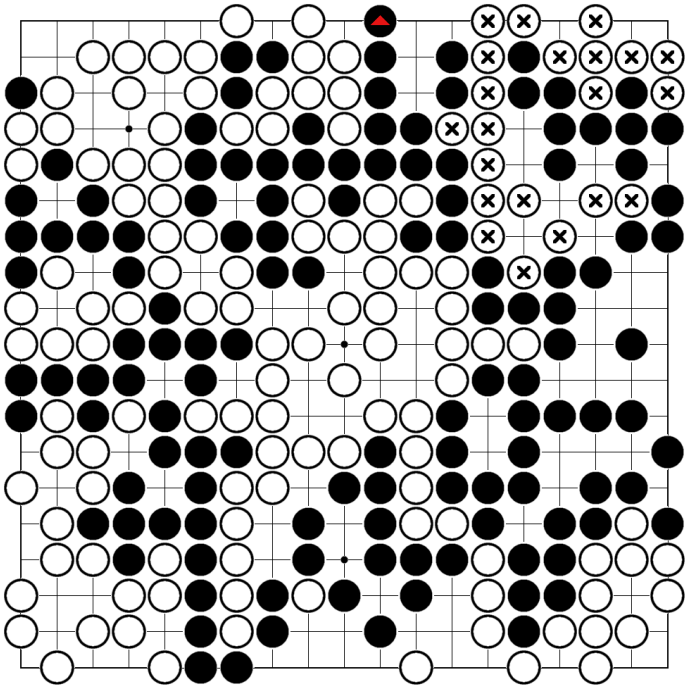

Figure 2.15: The final position of the exhibition match in the 4th UEC Cup: Kaori Aoba 4p (White) vs. ZEN (Black). Black won by resignation.

In the Computer Go Competition at the 2011 IEEE International Conference on Fuzzy Systems held on June 27-30, 2011, in Taiwan, ZEN defeated the top professional Go player Chun-Hsun Chou 9p with 6 stones. Figure 2.16 shows the final position of this game. The stone marked by Δ is the last move. In this game, Chun-Hsun Chou 9p consumed only 10 minutes totally, and he explained that "because I want to test the performance of ZEN in a fast game and most of ZEN's moves were exactly what I would play ".

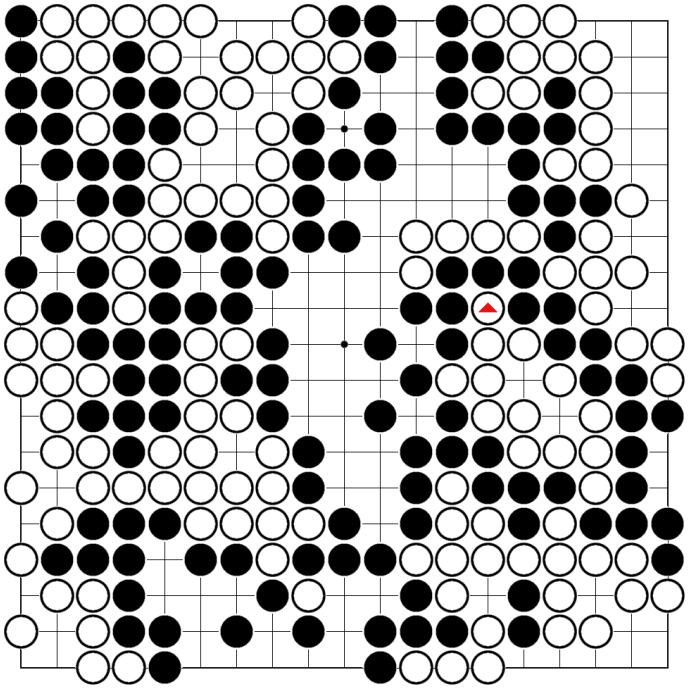

Figure 2.16: The final position of the match in the Computer Go Competition at the 2011 IEEE International Conference on Fuzzy Systems: Chun-Hsun Chou 9p (White) vs. ZEN (Black), with 6 handicap stones. Black won by 14.5 points.

2.4.7 Other Programs

Here we briefly and selectively introduce some other state-of-the-art and specific programs that are worth mentioning. PACHI by Petr Baudis and Jean-loup Gailly (Baudis and Gailly, 2010) is now the strongest open source program. VALKYRIA by Magnus Persson (Persson, 2010) features heavy and rich knowledge representation in the playout and is specifically competitive on the 9×9 board. LIBEGO by Łukasz Lew (Lew, 2010) is the fastest implementation of MCTS. OREGO by Peter Drake (Drake, 2011) is one of the popular test beds among computer Go researchers.

Chapter 3

ERICA

In this chapter, we introduce our Go-playing program ERICA. Section 3.1 briefly reviews the development history of ERICA, as well as its standings in some of the tournaments that ERICA participated up to July 2011. Section 3.2 investigates the implementation of MCTS of ERICA along with some of our own ideas. Finally, Section 3.3 gives several examples picked from the games that ERICA played against human players to indicate its strength.

3.1 Development History

3.1.1 First Version Created on May 2008

The first version of ERICA was created in May 2008, based on implementing MOGO's famous "UCT paper" (Gelly *et al.*, 2006). The work was motivated by the impressive performance of CRAZY STONE and MOGO in the 9×9 competition at the 2007 Computer Olympiad.

This earliest version of ERICA was written in pure C programming language. The speed was about 20,000 uniform random simulations per second on a single-core CPU of 2.26 GHz, on the 9×9 board. A *board* is realized by a single array, keeping the related information of a position, such as each string's color, liberty, owner[8] and size.

[8] The *owner* of a string is the representative stone of it.

MOGO-type, fixed-sequence simulation and RAVE form the basic MCTS framework of the program. In this period, the personal communications with Yizao Wang, one of the creators of MOGO, helped considerably for the author's understanding of the "UCT paper" and to make ERICA stronger.

In the Computational Intelligence Forum & World 9×9 Computer Go Championship held on September 25-27, 2008, in Taiwan, ERICA ended up in the *4th position*. Table 3.1 shows the result of this competition. In this tournament, ERICA won a game against GO INTELLECT but lost to JIMMY, the strongest Taiwanese Go-playing program at that time.

Position	Program	Wins	Country
1	MOGO	10	France
2	GO INTELLECT	6	America
3	JIMMY	6	Taiwan
4	**ERICA**	**6**	**Taiwan**
5	FUDO GO	6	Japan
6	CPS	6	Taiwan
7	GOSTAR	4	Taiwan
8	GOKING	4	Taiwan
9	HAPPYGO	2	Taiwan
10	CHANGJUAN1	0	Taiwan

Table 3.1: The result of the Computational Intelligence Forum & World 9×9 Computer Go Championship held on September 25-27, 2008, in Tainan, Taiwan.

In the 9×9 Go tournament at the 2008 Computer Olympiad held on September 28 to October 5, 2008, in Beijing, China, ERICA finished in *11th* place among the 18 participants. Figure 3.1 shows the game between ERICA (White) and AYA (Black) in round 2. In this game, thanks to the correct handling of *seki* in the playout, ERICA reversed the bad situation and won.

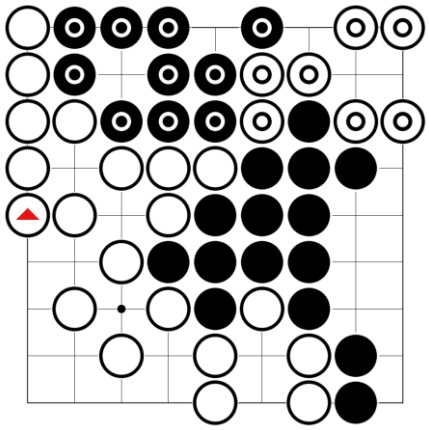

Figure 3.1: The final position of the match in round 2 in the 9×9 Go tournament at the 2008 Computer Olympiad: ERICA (White) vs. AYA (Black). The stone marked by Δ is the last move. The groups marked by ○ form a *seki*. White won by resignation.

At the 2009 Computer Olympiad held on May 10-18, 2009, in Pamplona, Spain, ERICA participated in the 9×9 Go tournament with the same version that played in the previous year. Finally, ERICA scored the *6th* position among the 9 participants.

3.1.2 Second Version Created on June 2009

In June 2009, a new version of ERICA was created, under the supervision of Rémi Coulom. The main advancement in this new version consists in the implementation of the Boltzmann softmax playout policy that was successful in Coulom's CRAZY STONE. In addition to RAVE, *prior information* was taken into account in the formulation of progressive bias (Chaslot *et al.*, 2007). The supervised learning algorithm Minorization-Maximization (MM) (Coulom, 2007) was used to train the pattern weights.

At the 2009 TAAI Computer Go Tournament held on October 30-31, 2009, in Taiwan, ERICA won the *3rd* and *2nd* position in the 9×9 Go (Table 3.2) and 19×19 Go (Table 3.3) tournaments respectively.

Position	Program	Country
1	ZEN	Japan
2	MOGO	France
3	**ERICA**	**Taiwan**

Table 3.2: The result of the 9×9 Go tournament at the 2009 TAAI Go Tournament.

Position	Program	Country
1	ZEN	Japan
2	**ERICA**	**Taiwan**
3	DRAGON	Taiwan

Table 3.3: The result of the 19×19 Go tournament at the 2009 TAAI Go Tournament.

In the next month, ERICA participated in the 3rd UEC Cup held on November 28-29, 2009, in Japan, and at last scored the *6th* position. In the round 2 of this tournament, ERICA for the first time defeated the well-known strong 19×19 Go-playing program AYA, as shown in Figure 3.2. The last move is marked by Δ. In this game, ERICA (Black) killed several White's groups, marked by ×, and acclaimed a great victory.

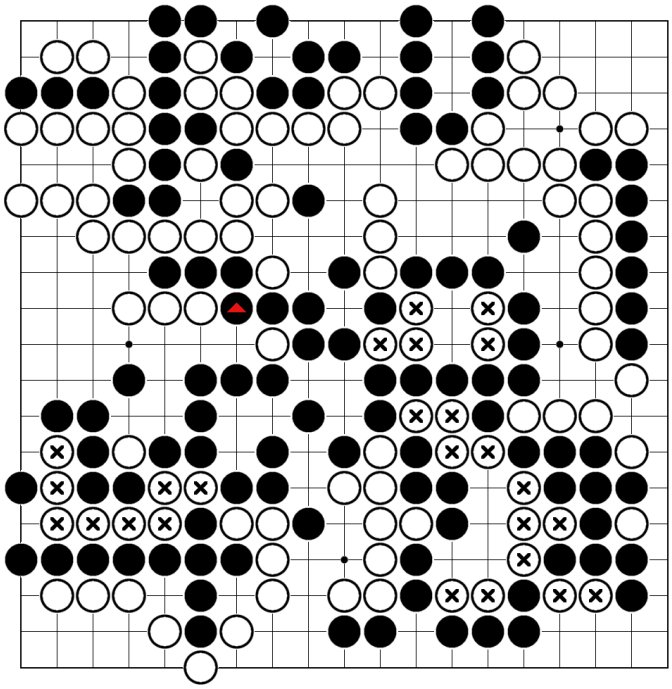

Figure 3.2: The final position of the match in round 2 of the 3rd UEC Cup: ERICA (White) vs.

AYA (Black).. Black won by resignation.

3.1.3 Third Version Created on February 2010

In February 2010, Łukasz Lew, the author of LIBEGO was a great help in speed optimization of ERICA. The author re-wrote many primary data structures and created a new version of ERICA, under the supervision of Coulom. For instance, *macro*s, a sort of preprocessor in C programming language, were used extensively for loop unrolling. The speed of the simulation was accelerated by a factor of 2 compared to the previous version. In this period, we concentrated on 19×19 Go, trying hard to make use of larger patterns in the tree and improve the quality of the playout.

At the 2010 Computer Olypmiad held on September 24 to October 2, 2010, in Kanazawa, Japan, ERICA won the *gold* and *silver* medals in the 19×19 Go (Fotland, 2010) and 9×9 Go tournaments respectively. In the 19×19 Go tournament, after the final round is finished, three programs, ZEN, THE MANY FACES OF GO and ERICA were in a tie. The final positions were decided in the second playoff, when both ZEN and ERICA defeated THE MANY FACES OF GO and ERICA defeated ZEN. This indicates that the three programs were competitive in playing strength. Figure 3.3 shows the final match between ZEN (White) and ERICA (Black). This game was decided by a large-scale semeai in the opening stage. ZEN misread the semeai so that ERICA killed White's big group (marked by ×) and secured the lead until the end.

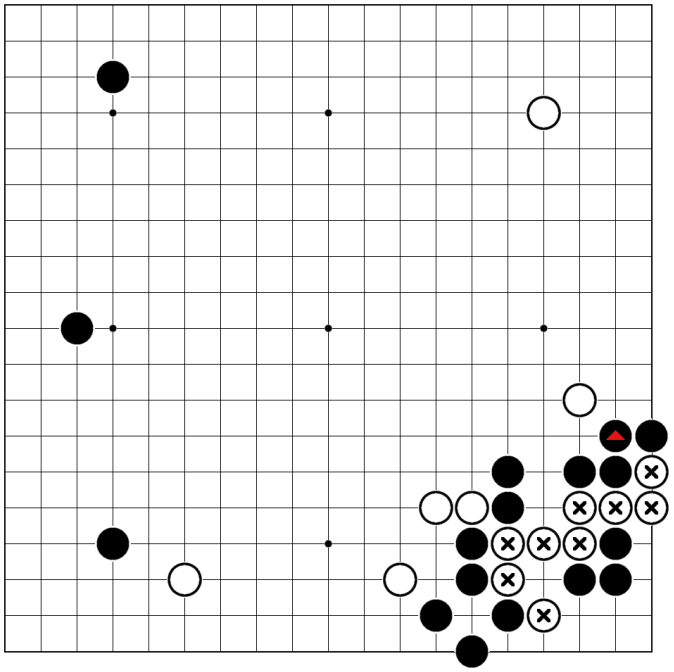

Figure 3.3: A position of the final match in the playoff of the 19×19 Go tournament at the 2010 Computer Olympiad: ZEN (White) vs. ERICA (Black). The previous move is marked by Δ. Black won by resignation.

In the 4th UEC Cup, held on 27th to 28th November 2010 in Japan, ERICA won the *3rd* position[9]. Table 3.4 shows the result of this interesting tournament. In this tournament, the games between the strong programs clearly indicated that handling semeai correctly is particularly crucial. Figure 3.4 shows the position of the match between THE MANY FACES OF GO (White) and ERICA (Black) in round 4 of the preliminaries in the first day. This game was decided by the large-scale semeai in the middle game. The Black's group, marked by ○, has 4 liberties A, B, C and D while the White's group, marked by ×, has only 3 liberties E, F and G. Finally, ERICA played correctly to win this capturing race and defeated the tough rival THE MANY FACES OF GO.

[9] Special thanks to Professor Tsan-Sheng Hsu, Research Fellow of Academia Sinica, Taiwan, who kindly provided us the hardware resources, an 8-core server with 64GB memory, for this tournament.

Position	Program	Country
1	FUEGO	Canada
2	ZEN	France
3	**ERICA**	**Taiwan**
4	AYA	Japan
5	THE MANY FACES OF GO	America
6	COLDMILK	Taiwan
7	CAREN	Japan
8	PERSTONE	Japan

Table 3.4: The result of the 4th UEC Cup, 2010.

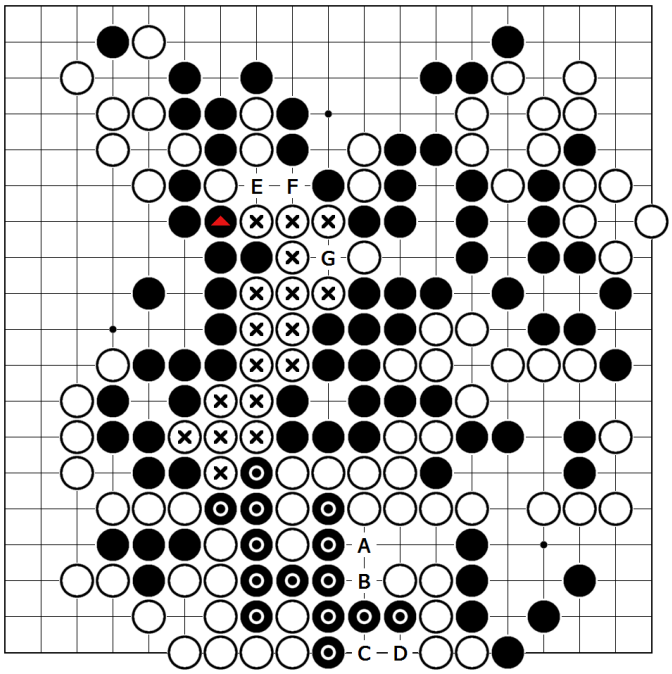

Figure 3.4: A position of the match in the 4th UEC Cup: THE MANY FACES OF GO (White) vs. ERICA (Black). The previous move is marked by Δ. Black won by resignation.

3.2 MCTS in ERICA

This section investigates the implementation of MCTS in ERICA, along with some of our own ideas. Note that these ideas might be re-inventions, since there are plenty of open source Go-playing programs to trace that we might overlook, not mentioning to the ones of unavailable source code.

3.2.1 Selection

The selection formula of ERICA is a combination of the strategies of UCT, RAVE and

progressive bias which maximizes the selection formula (3.1):

$$\text{Coefficient} \times \text{RAVE} + (1 - \text{Coefficient}) \times \text{UCT} + \text{progressive\_bias} \quad (3.1)$$

where *Coefficient* is the weight of RAVE computed by Silver's formula (Silver, 2009) and the exploration term of RAVE is taken off as Silver suggested. For the exploration term of UCT, C_{uct} is set to 0.6 for all board sizes.

The term progressive_bias is computed by the formula (3.2):

$$\text{progressive\_bias} = C_{PB} \times \frac{v_{prior}}{n_{uct}} \quad (3.2)$$

where C_{PB} is a constant which has to be tuned empirically. n_{uct}, initialized to 1, is the visit count of this node and v_{prior} is the prior value in [0,1]. After the end of search, the most visited candidate move in the root node is selected to play. For ERICA, the good value of C_{PB} on the 19×19 board is around 50. Note that the good values of C_{PB} can vary in different board sizes.

3.2.2 Expansion

ERICA uses *delayed node creation* (a node is expanded in the *n*th (*n*>1) visit) to reduce the memory overhead caused by RAVE, as explained in Section 2.2.2. For ERICA, the good value of *n* on the 19×19 board is around 5. In node creation, the prior computation takes into account various features which are partly listed in (Coulom, 2007), according to the pattern weights given by MM.

3.2.2.1 Larger Patterns

For ERICA, the first and foremost feature in prior computation on the 19×19 board is larger patterns of diamond-shape (Stern at al., 2006). Firstly, larger patterns of up to

size 9 (by the definition in (Stern at al., 2006)) are harvested from the game records according to their frequencies of appearance. Then, these patterns are trained by MM together with other features that participate in prior computation. In ERICA, larger patterns are only used in progressive bias, not in the playout. The improvement from larger patterns is measured to be over 100 Elo.

3.2.2.2 Other Features

Other useful features for the 19×19 board are, for instance, ladder, distance features (distance to the previous move, distance to the move before the previous move and *Common Fate Graph* (CFG) distance (Graepel *et al.*, 2001), etc) and various tactical features of semeai and life-and-death, such as "save a string by capturing".

3.2.3 Simulation

3.2.3.1 Boltzmann Softmax Playout Policy

In simulation stage, ERICA uses Boltzmann softmax playout policy (usually called *softmax policy* or *Gibbs sampling* (Geman and Geman; 1984)). Softmax policy was firstly applied to a Monte Carlo Go program in (Bouzy and Chaslot, 2006) and called *psdueo-random* moves that are generated by domain-dependent approach which uses a non-uniform probability. In the experiments of Bouzy and Chaslot, only 3×3 patterns along with one-liberty urgency were served as the features. This scheme of pseudo-random, non-uniform probabilistic distribution was further improved and extended by Coulom to multiple features (Coulom, 2007).

The softmax policy π_θ is defined by the probability of choosing action a in state s:

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^T\theta}}{\sum_b e^{\phi(s,b)^T\theta}}$$

where Φ(s, a) is a vector of binary features, and θ is a vector of feature weights.

To explain the softmax policy, Figure 3.5 gives an example of a position in the playout, Black to move. The previous move is marked by Δ. For Black, now the only legal moves are *A*, *B*, *C* and *D*[10].

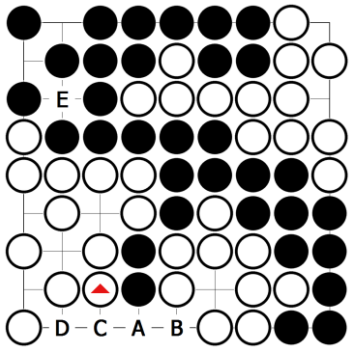

Figure 3.5: An example of a position in the playout. The previous move is marked by Δ. Black to move.

Suppose there two binary features (for simplicity, e^{θ_i} is denoted by γ_i):

1. **Contiguous to the previous move**. A candidate move that is directly neighboring to the previous move has this feature. The weight of this feature is γ_1. Point *A*, *C* and *D* have this feature.
2. **Save the string, put in atari by the previous move, by extending**. The weight of this feature is γ_2. Point *A* has this feature.

Then, the weight of each move is,

A: weight= $\gamma_1 \times \gamma_2$

B: weight= γ_1

C: weight= γ_1

D: weight= 1, with no corresponding feature.

[10] In ERICA, an empty point that fills a real eye, such as E, is also regarded as an illegal move, though they are legal according to the Go rules. Forbid "filling a real eye" in the playout is commonly used in current Mone Carlo Go-playing programs.

Consequently, the probability to choose each move is given by

A: $\frac{\gamma_1 \times \gamma_2}{\gamma_1 \times \gamma_2 + \gamma_1 + \gamma_1 + 1}$

B: $\frac{\gamma_1}{\gamma_1 \times \gamma_2 + \gamma_1 + \gamma_1 + 1}$

C: $\frac{\gamma_1}{\gamma_1 \times \gamma_2 + \gamma_1 + \gamma_1 + 1}$

D: $\frac{1}{\gamma_1 \times \gamma_2 + \gamma_1 + \gamma_1 + 1}$

3.2.3.2 Move Generator

The move generator in the playout of ERICA is depicted by the pseudocode shown in Table 3.5. The details are explained as follows.

```
MoveGenerator()
{
    ComputeLocalFeatures();
    for (;;)
    {
        If (TotalGamma == 0)
        {
            Move = PASS;
            break;
        }
        Move = ChooseMoveByProbability();
        If (ForbiddenMove(Move))
        {
            SetZeroGamma(Move);
            continue;
        }
        ReplaceMove(&Move);
        break;
    }
    RecoverMoves();
    Return Move;
}
```

Table 3.5: Pseudocode of the move generator in the playout of ERICA.

ComputeLocalFeatures deals with the *local features* (the features related to the

previous move or the move before the previous move, etc) and updates the gammas of the local moves which have the local features. 3×3 patterns and some of the local features of ERICA will be introduced in Section 4.3.2.

The move to be generated is decided in the "for loop". Firstly, if *TotalGamma*, the sum of the gammas of all the moves in this position, is equal to 0, *Move* is set to *pass* and returned immediately since no move has a nonzero probability. Otherwise, *ChooseMoveByProbability* chooses a move and assigns to *Move* by softmax policy as described in the previous section.

After *Move* is chosen, *ForbiddenMove* examines that if *Move* is forbidden, which means it has a feature of zero weight. If *Move* is detected to be forbidden, *SetZeroGamma* subtracts its gamma from *TotalGamma* and resets the gamma to zero. The mechanism of *ForbiddenMove* is a compromise for the features which are too costly to incrementally update. Note that it is also possible to check the legality of *Move* in *ForbiddenMove*. The next section will give an example of *ForbiddenMove*.

After the examination of *forbiddenMove*, *Move* is passed (call by reference) to *ReplaceMove* for further inspection. *ReplaceMove* is an extended version of *ForbiddenMove* in the sense that it not only examines if *Move* is forbidden or not, but also replaces it with a better move for the former case. Section 3.2.3.4 will give an example of *ReplaceMove*.

Outside the "for loop", when *Move* is ready to be returned, *RecoverMoves* sets back the gammas of the moves reset by *forbiddenMove*.

3.2.3.3 ForbiddenMove

Figure 3.6 gives an example of *ForbiddenMove* of ERICA's move generator in the playout, Black to move. In this example, point *A* is forbidden because it is a *self-atari* of 9 stones, which is a clearly bad move. In ERICA, a self-atari move is not forbidden

if it forms a *nakade* shape.

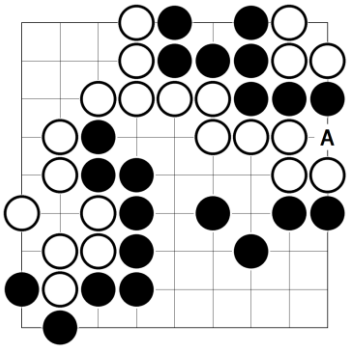

Figure 3.6: An example of *ForbiddenMove*, Black to move.

3.2.3.4 ReplaceMove

Figure 3.7 gives an example of *ReplaceMove* of ERICA's move generator in the playout, Black to move. In this example, point *A* is forbidden and replaced with *B* by the rule "when filling a false eye, if there is a capturable group in one of the diagonal point, then capture the group instead of filling the false eye".

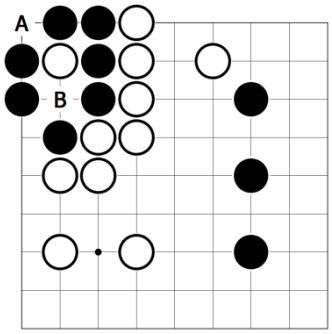

Figure 3.7: An example of *ForbiddenMove*, Black to move.

3.2.4 Backpropagation

In this section, we present two useful heuristics of RAVE to improve its performance. Section 3.2.4.1 presents the first heuristic, to bias RAVE updates by move distance. Section 3.2.4.2 presents the second heuristic, to fix RAVE updates for ko threats.

3.2.4.1 Bias RAVE Updates by Move Distance

When updating the RAVE values in a node, the heuristic "Bias RAVE Updates by

Move Distance" is to bias the simulation outcome according to how far the updated move was played away from this node. The number of the moves between this node and the updated move is defined as the *distance* of this move, denoted by *d*. The weight to bias the simulation outcome is defined as *distance weight*, denoted by *w*. If the simulation outcome is 1, then the updated outcome is $1-d*w$; if the simulation outcome is 0, then the updated outcome is $0+d*w$. Figure 3.8 gives an example.

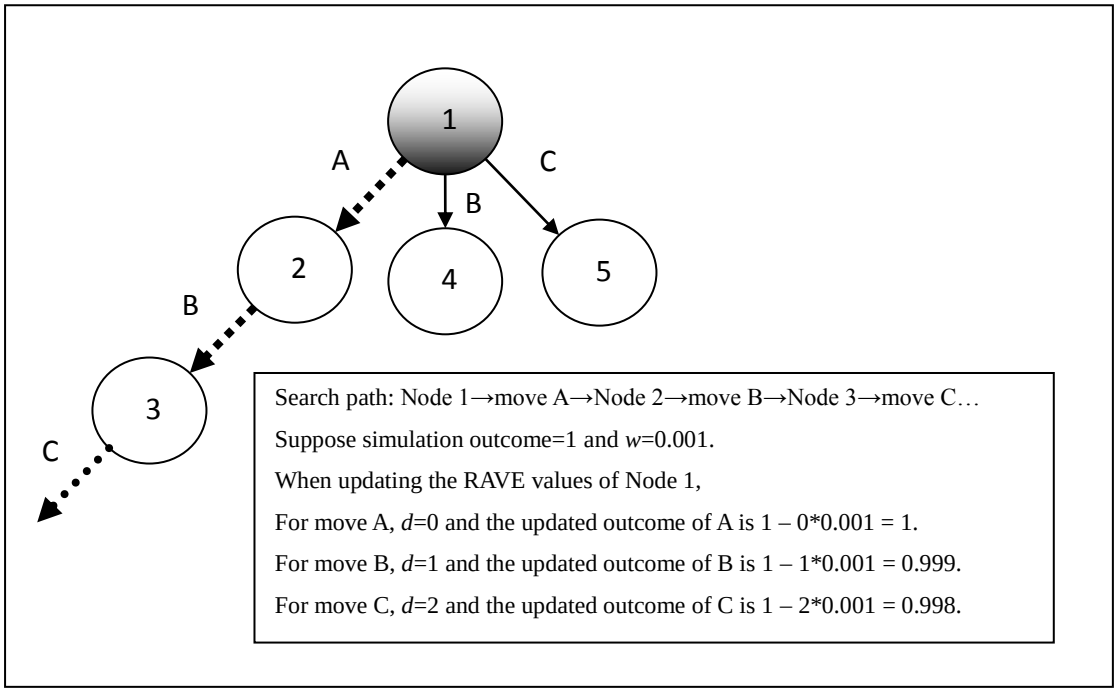

Figure 3.8: An example of "Bias RAVE Updates by Move Distance".

As far as we know, FUEGO was the first Go-playing program that proposed and used this idea[11]. This heuristic brings in the information of move sequence to RAVE. It is worth about 50 Elo in our experiments.

3.2.4.2 Fix RAVE Updates for Ko Threats

Figure 3.9 is an illustration to show the occasion where this heuristic is applicable. This position is selected from the game played on the KGS Go Server (KGS) between

[11] The details of FUEGO's approach to "Bias RAVE Updates by Move Distance" can be found in the documents of FUEGO in the official web site, http://fuego.sourceforge.net/.

ajahuang [6d] (White) and *Zen19D*[5d] (Black). The previous move (marked by Δ) played by ZEN is clearly meaningless, though it's a *sente* move or *ko threat* that forces White to respond. Apparently, the correct move in this moment is *A*, namely to capture the ko. But why ZEN played a ko threat before capturing the ko?

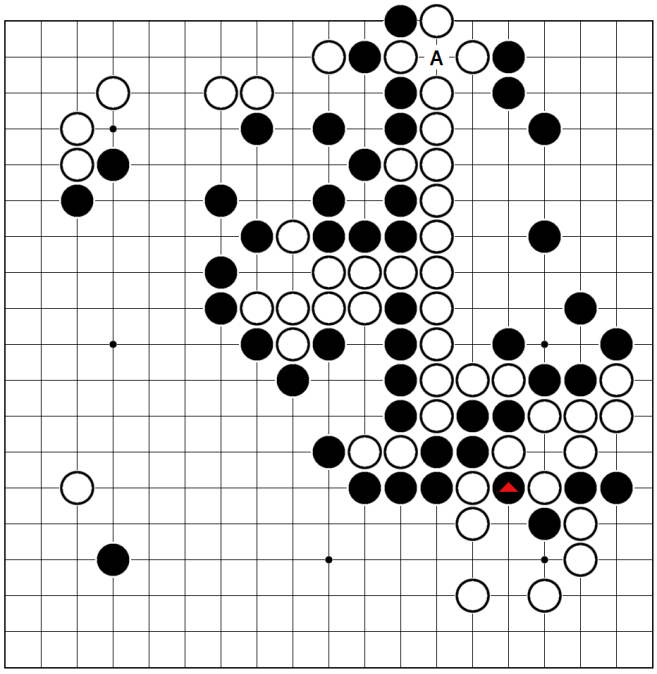

Figure 3.9: An example to show the occasion of the heuristic "Fix RAVE Updates for Ko Threats": *ajahuang* [6d] (White) vs. *Zen19D* [5d] (Black). White won by resignation.

The problem is (probably) out of RAVE. It is due to the intrinsic problem of RAVE that in the root node, the RAVE value of the ko threats (such as the previous move marked by Δ), which were searched in the lower levels of the tree, are also updated. But a ko threat is supposed to be played after a ko capture. So, in this example, the RAVE value of Black's ko threats (such as the previous move marked by Δ) should not be updated in the root node. This is the main idea of the heuristic "Fix RAVE updates for Ko Threats". Figure 3.10 gives an example of this heuristic to show how it works practically in the tree. In this example, the RAVE value of move E in Node 1 is not updated because it is detected as a ko threat move of Node 5. This

heuristic is worth about 30 Elo in our experiments.

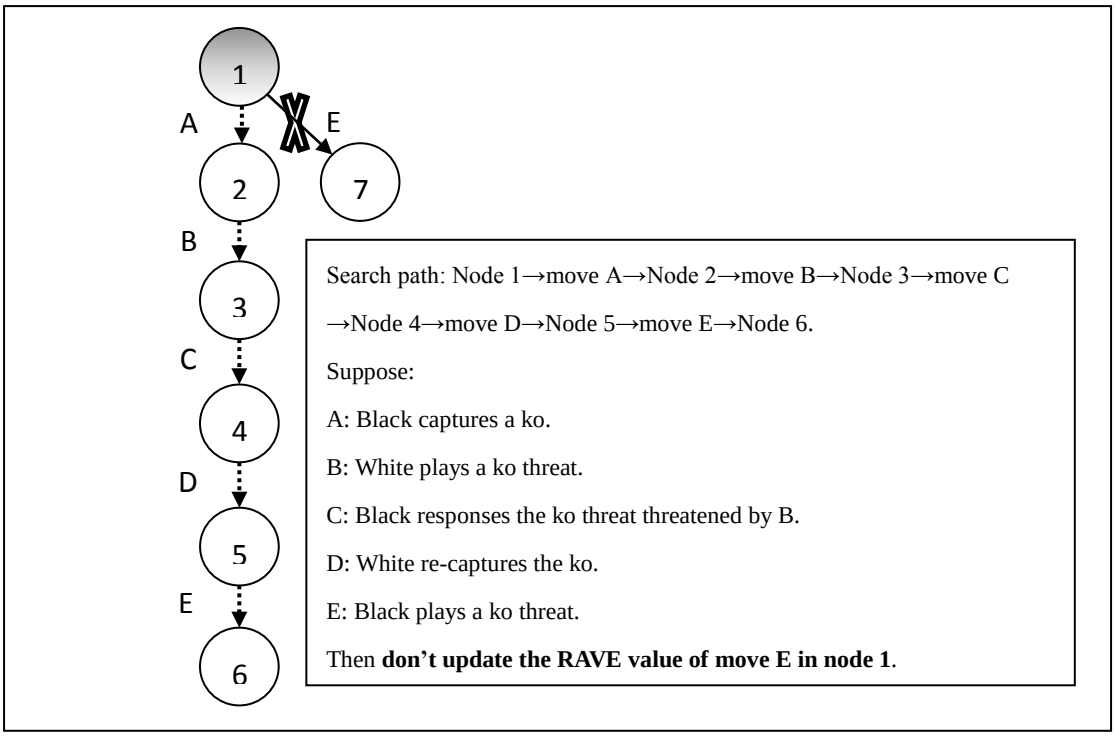

Figure 3.10: An example of "Fix RAVE Updates for Ko Threats".

3.3 KGS Games of ERICA

Starting December 13, 2010, ERICA played on the KGS Go Server (KGS) using the account *EricaBot*, running on a 4-core CPU of 3.07 GHz. With short time setting of 10×00:15 (15 seconds byo-yomi for 10 times), it was rated 3-dan in the beginning and about 3.75-dan on June, 2010, as shown in Figure 3.11.

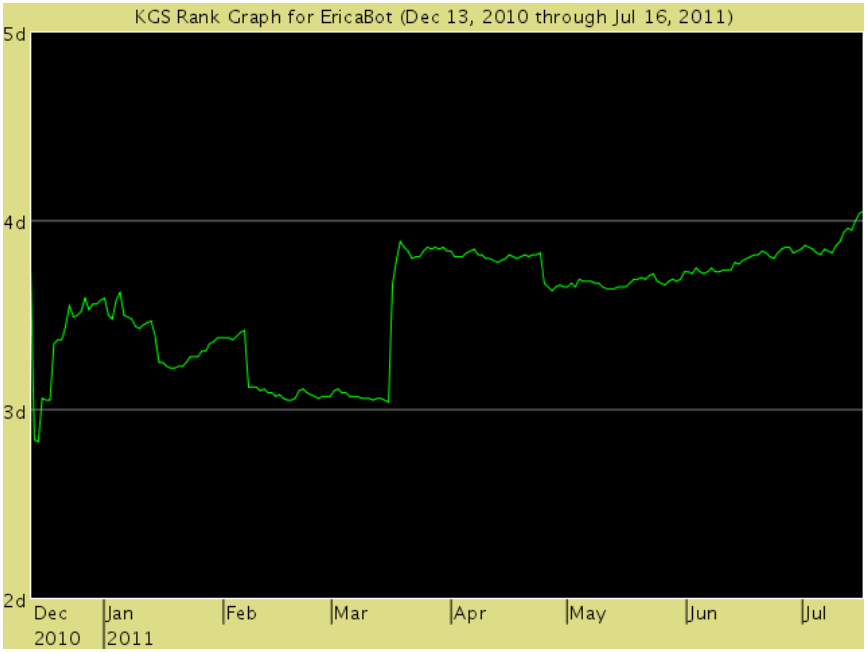

Figure 3.11: The KGS Rank Graph for *EricaBot*.

Figure 3.12 shows a 19×19 game between *EricaBot* (White) and a 2-dan human player *BOThater36*. In this game, ERICA captured the center group by ladder-atari (move 120) and won. This game shows that ERICA is a solid 3-dan player and features moderate opening play on the 19×19 board.

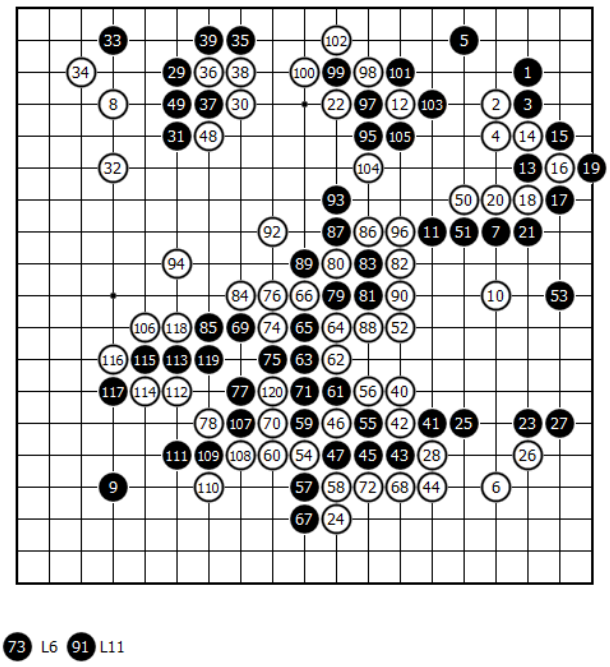

Figure 3.12: A 19×19 ranked game on KGS: *EricaBot* 3-dan (White) vs. *BOThater36* 2-dan (Black). White won by resignation.

Figure 3.13 shows a 9×9 game between *Erica9* (White) and a 5-dan human player *guxxan*. In this game, ERICA played a classical killing method (move 32 and 34) to kill the Black group in the top-left corner. This game shows that ERICA is already a solid high dan player on the 9×9 board.

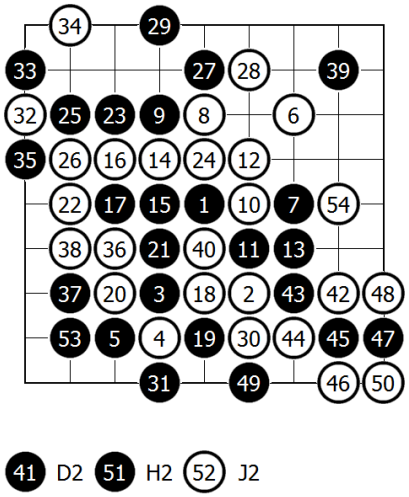

Figure 3.13: A 9×9 game on KGS: *Erica9* (White) vs. *guxxan* 5-dan (Black). White won by resignation.

Chapter 4

Monte Carlo Simulation Balancing Applied to 9×9 Go

4.1 Introduction

Monte Carlo evaluation of a position depends on the choice of a probability distribution over legal moves. A uniform distribution is the simplest choice, but produces poor evaluations. It is often better to play good moves with a higher probability, and bad moves with a lower probability. Playout policy has a large influence on the playing strength. Several methods have been proposed to optimize it.

The simplest approach to policy optimization is trial and error. Some knowledge is implemented in playouts, and its effect on the playing strength is estimated by measuring the winning rate against other programs (Bouzy, 2005; Gelly *et al.*, 2006; Chen and Chang, 2008; Chaslot *et al.*, 2009). This approach is often slow and costly, because measuring the winning rate by playing games takes a large amount of time, and many trials fail. It is difficult to guess what change in the playout policy will make the program stronger, because making playouts play better often causes the Monte Carlo program to become weaker (Bouzy and Chaslot, 2006; Gelly and Silver, 2007).

In order to avoid the difficulties of crafting a playout policy manually, some authors tried to establish principles for automatic optimization. We mention two of them. First, it is possible to optimize directly numerical parameters with generic stochastic optimization algorithms such as the cross-entropy method (Chaslot *et al.*, 2008). Such a method may work for a few parameters, but it still suffers from the rather high cost of measuring strength by playing games against some opponents. This cost may be overcome by methods such as reinforcement learning (Bouzy and Chaslot, 2006; Gelly and Silver, 2007; Silver and Tesauro, 2009), or supervised learning from good moves collected from game records (Coulom, 2007). Supervised learning from game records has been quite successful, and is used in some top-level Go programs such as ZEN and CRAZY STONE.

Second, among the reinforcement-learning approaches to playout optimization, a recent method is simulation balancing (SB) (Silver and Tesauro, 2009). It consists in tuning continuous parameters of the playout policy in order to match some target evaluation over a set of positions. This target evaluation is determined by an expert. For instance, it may be obtained by letting a strong program analyze positions quite deeply. Experiments reported by Silver and Tesauro indicate that this method is promising: they measured a 200-point Elo improvement over previous approaches.

Yet, the SB experiments were promising, but not completely convincing, because they were not run in a realistic setting. They were limited to 2×2 patterns of stone configurations, on the 5×5 and 6×6 Go boards. Moreover, they relied on a much stronger program, FUEGO (Enzenberger and Müller, 2009), that was used to evaluate positions of the training database. Anderson (2009) failed to replicate the success of SB for 9×9 Go, but may have had bugs, because he did not improve much over uniform-random playouts. So, it was not clear whether this idea could be applied successfully to a state-of-the-art program.

This chapter presents the successful application of SB to ERICA, a state-of-the-art Monte Carlo program. Experiments were run on the 9×9 board. The training set was made of positions evaluated by ERICA herself. So this learning method does not require any external expert supervisor. Experimental results demonstrate that SB made the program stronger than its previous version, where patterns were trained by minorization-maximization (MM) (Coulom, 2007). Besides a raise in playing strength, a second interesting result is that pattern weights computed by MM and SB are quite different from each other. For instance, SB patterns may wish to play some rather bad shape positions, which are evaluated quite badly by MM, but that helps to arrive at a correct playout outcome.

4.2 Description of Algorithms

This section is a brief reminder of the MM (Coulom, 2007) and SB (Silver and Tesauro, 2009) algorithms. More details about these algorithms can be found in the references.

4.2.1 Softmax Policy

Both MM and SB optimize linear parameters of a Boltzmann softmax policy, which was introduced in Section 3.2.3.1. The objective of learning algorithms is to find a good value for θ.

4.2.2 Supervised Learning with MM

MM learns feature weights by supervised learning over a database of sample moves (Coulom, 2007). MM is a maximization algorithm for computing maximum-a-posteriori values of θ, given a prior distribution and sample moves. The principle of this algorithm dates back to at least Zermelo (1929). Its formulation and convergence properties were studied recently in a more general case by Hunter

(2004).

When learning with MM, the training set is typically made of moves extracted from game records of strong players. It may also be made of self-play games if no expert game records are available.

4.2.3 Policy-Gradient Simulation Balancing (SB)

SB does not learn from examples of good moves, but from a set of evaluated positions. This training set may be made of random positions evaluated by a strong program, or a human expert. Feature weights are trained so that the average of playout outcomes matches the target evaluation given in the training set. Silver and Tesauro (Silver and Tesauro, 2009) proposed two such algorithms: policy-gradient simulation balancing and two-step simulation balancing. We chose to implement policy-gradient simulation balancing only, because it is simpler and produced better results in the experiments by Silver and Tesauro.

The principle of Policy-Gradient Simulation Balancing consists in minimizing the quadratic evaluation error by the steepest gradient descent. Estimates of the gradient are obtained with a likelihood-ratio method (Glynn, 1987), also known as REINFORCE (Williams, 1992).

The details of SB are given in Algorithm 1. In this algorithm, $\psi(s,a)$ is defined by:

$$\psi(s,a) = \nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - \sum_b \pi_\theta(s,b)\phi(s,b) \ .$$

$V^*(S_1)$ is the target value of position s_1. α is the learning rate of steepest descent. z is the outcome of one playout, from the point of view of the player who made action a_1 (+1 for a win, -1 for a loss, for instance). s_i and a_i are successive states and actions in a playout of T moves. M and N are integer parameters of the algorithm. V and g are

multiplied in the update of θ, so they must be evaluated in two separate loops, in order to obtain two independent estimates.

Algorithm 1 Policy-Gradient Simulation Balancing (SB)

$\theta \leftarrow 0$
for all $s_1 \in$ training set **do**
  $V \leftarrow 0$
  for $i = 1$ to M **do**
    simulate($s_1, a_1, \ldots, s_T, a_T; z$) using π_θ
    $V \leftarrow V + \frac{z}{M}$
  end for
  $g \leftarrow 0$
  for $j = 1$ to N **do**
    simulate($s_1, a_1, \ldots, s_T, a_T; z$) using π_θ
    $g \leftarrow g + \frac{z}{NT} \sum_{t=1}^{T} \psi(s_t, a_t)$
  end for
  $\theta \leftarrow \theta + \alpha(V^*(s_1) - V)g$
end for

4.3 Experiments

Experiments were run with the Go-playing program ERICA. The SB algorithm was applied repeatedly with different parameter values, in order to measure their effects. Playing strength was estimated with matches against FUEGO. The result of applying SB is compared to MM, both in terms of playing strength and feature weights.

4.3.1 ERICA

ERICA is developed by the author in the framework of his Ph.D. research. More details of ERICA can be found in Chapter 3.

4.3.2 Playout Features

This subsection and the remainder of this chapter uses Go jargon that may not be familiar to some readers. Explanations for all items of the Go-related vocabulary can be found in the Sensei's Library web site (http://senseis.xmp.net/). Still, it should be possible to understand the main ideas of this chapter without understanding that vocabulary. The playouts of ERICA are based on 3×3 stone patterns, augmented by the

atari status of the four directly connected points. These patterns are centred on the move to be played. By taking rotations, symmetries, and move legality into consideration, there is a total of 2,051 such patterns. In addition to stone patterns, ERICA uses 7 features related to the previous move (examples are given in Figure 4.1).

1. **Contiguous to the previous move**. Active if the candidate move is among the 8 neighbouring points of the previous move. Also active for all Features 2–7.

2. **Save the string in new atari, by capturing**. The candidate move that is able to save the string in new atari by capturing has this feature.

3. **Same as Feature 2, which is also self-atari**. If the candidate move has Feature 2 but is also a self-atari, then instead it has Feature 3.

4. **Save the string in new atari, by extending**. The candidate move that is able to save the string in new atari by extending has this feature.

5. **Same as Feature 4, which is also self-atari**.

6. **Solve a new ko by capturing**. If there is a new ko, then the candidate move that is able to solve the ko by capturing any one of the neighbouring strings has this feature.

7. **2-point semeai**. If the previous move reduces the liberties of a string to only two, then the candidate move that gives atari to its neighbouring string which has no way to escape has this feature. This feature deals with the most basic type of semeai.

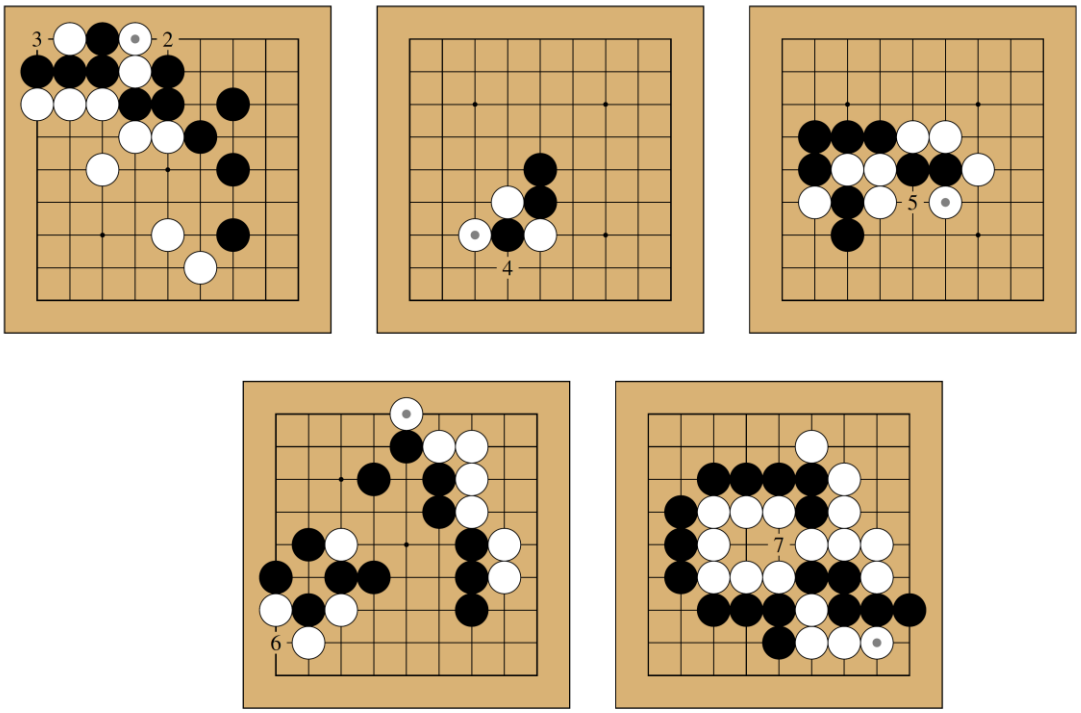

Figure 4.1: Examples of Features 2,3,4,5,6 and 7. Previous move is marked with a dot.

4.3.3 Experimental Setting

The performances of MM and SB were measured by the winning rate of ERICA against FUEGO 0.4 with 3,000 playouts per move for both programs. In the empty position, ERICA ran 6,200 playouts per second, whereas FUEGO ran 7,200 playouts per second. For reference, performance of the uniform random playout policy and the MM policy are shown in Table 4.1.

ERICA's Playout Policy	Winning Rate of ERICA
Uniform Random	$6.8\% \pm 1.6$
19×19 MM	$68.9\% \pm 2.9$
9×9 MM	$40.9\% \pm 3.0$

Table 4.1: Reference results against FUEGO 0.4, 1,000 games, 9×9, 3k playouts/move

For fairness, the trainings of MM and SB were both performed with the same features described above. The training of MM was accomplished within a day,

performed on 1,400,000 positions, chosen from 150,000 19×19 game records by strong players. The games were KGS games collected from the web site of Kombilo (Goertz and Shubert, 2007), combined with professional games collected from the web2go web site (Lin, 2009).

The production of the training data and the training process of SB were accomplished through ERICA without any external program. The training positions were randomly selected from the games self-played by ERICA with 3,000 playouts per move. Then ERICA with playouts parameters determined by MM, was directly used to evaluate these positions. It took over three days to complete merely the production and evaluation of the training positions. From this viewpoint, SB training costs much more time than MM.

The 9×9 positions were also used to measure the performance of MM in the situation equivalent to that of SB. The same 5k positions, that were served as the training set of SB, were trained on MM to compute the patterns.

The strength of these patterns was measured and shown in Table 1 as 9×9 MM.

4.3.4 Results and Influence of Meta-Parameters

SB has a few meta-parameters that need tuning. For the gradient-descent part, it is necessary to choose M, N, and α. Two other parameters define how the training set was built: number of positions, and number of playouts for each position evaluation. Table 4.2 summarizes the experimental results with these parameters.

Positions	5k	5k	5k	5k	5k	10k
Playouts	100k	100k	10k	100k	100k	100k
M	500	100	500	500	100	500
N	500	100	500	500	100	500
α	10	10	10	10	1	10
Outcome	0/1	-1/1	-1/1	-1/1	-1/1	-1/1
20	51.5%	69.2%	65.7%	69.3%	51.8%	71.2%
40	57.6%	75.5%	68.5%	75.4%	57.2%	76.0%
60	58.1%	70.1%	70.8%	77.9%	57.2%	74.0%
80	61.3%	78.2%	72.2%	76.8%	63.7%	76.9%
100	63.9%	76.2%	74.0%	73.5%	65.4%	76.0%
200	60.8%	77.4%	71.6%	76.3%	70.1%	74.1%
300	61.9%	73.9%	72.1%	75.0%	73.2%	
500	61.6%	72.3%		69.8%	75.4%	
700		71.6%			74.8%	
900		69.7%			74.3%	
1,100		70.5%			76.2%	
1,300		65.1%			76.6%	
1,500					77.3%	
1,700					76.0%	
1,900					74.2%	
2,100					76.4%	
Iteration	Winning Rate					

Table 4.2: Experimental results. The winning rate was measured 1,000 games against FUEGO 0.4, with 3,000 playouts per move. 95% condifence is ±3.1 when the winning rate is close to 50%, and ±2.5 when it is close to 80%.

Since the algorithm is random, it would have been better to replicate each experiment more than once, in order to measure the effect of randomness. Unlike MM, SB has no guarantee to find the global optimum, and may have a risk to get stuck at a bad local optimum. Because of limited computer resources, we preferred trying many parameter values rather than replicating experiments with the same parameters.

In the original algorithm, the simulations of outcome 0 are ignored when N simulations are performed to accumulate the gradient. The algorithm can be safely modified to use outcome -1/1 and replace z by (z - b), where b is the average reward, to make the 0/1 and -1/1 cases equivalent (Silver, 2009). The results of the 1st and 4th columns in Table 2 show that the learning speed of outcome -1/1 was much faster than 0/1, so that the winning rate of outcome -1/1 of Iteration 20 (69.2%) was even higher

than that of outcome 0/1 of Iteration 100 (63.9%). This is an indication that -1/1 might be better than 0/1, but more replications would be necessary to make a general conclusion.

The SB algorithm was designed to reduce the mean squared error (MSE) of the whole training set by stochastic gradient-descent. As a result, the MSE should gradually decrease if the training is performed on the same training set ever and again. Running the SB algorithm through the whole training set once is defined as an Iteration. Figure 4.2 shows that the measure MSE actually decreases.

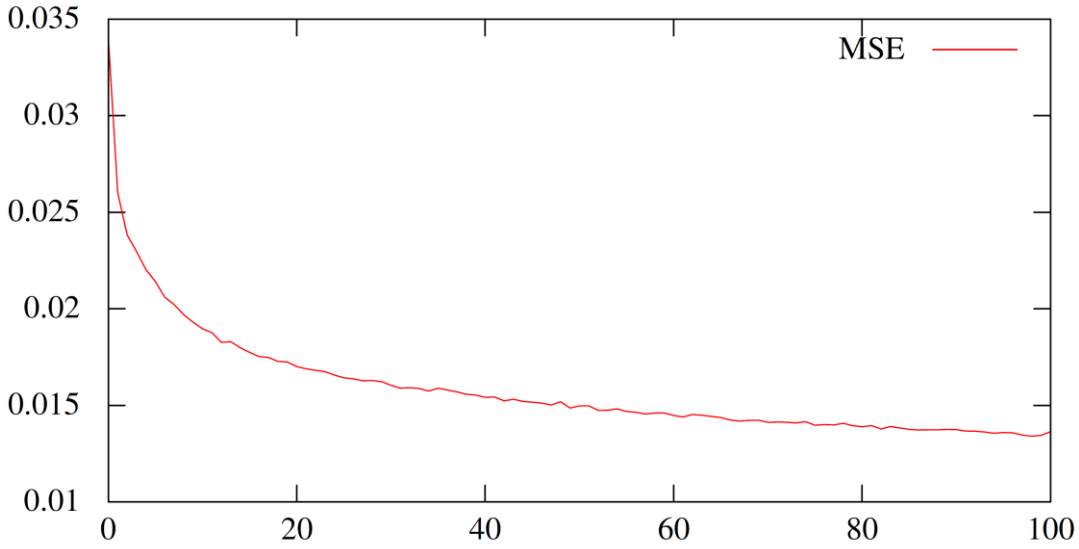

Figure 4.2: Mean square error as a function of iteration number. M=N=500, α =10, training set has 5k positions evaluated by 100 playouts. The error was measured by 1,000 playouts for every position for the training set.

4.4 Comparison between MM and SB Feature Weights

For all comparisons, SB values that scored 77.9% against FUEGO 0.4 were used (60 iterations, fourth column of Table 4.2). Table 4.3 shows the γ-values of local features ($\gamma_i = e^{\theta_i}$ is a factor proportional to the probability that Feature i is played). Table 4.4 shows some interesting 3×3 patterns (top 10, bottom 10, top 10 without atari, and

most different 10 patterns). Local features (Table 4.3) show that SB plays tactical moves such as captures and extensions in a way that is much more deterministic than MM. A possible interpretation is that strong players may sometimes find subtle alternatives to those tactical moves, such as playing a move in sente elsewhere. But those considerations are far beyond what playouts can understand, so more deterministic captures and extensions may produce better Monte Carlo evaluations.

Feature	Description	MM γ	SB γ
1	Contiguous	11.12	7.43
2	Save new atari by capturing	32.37	151.04
3	2 + self-atari	0.24	0.53
4	Save new atari by extending	6.71	23.11
5	4 + self-atari	0.05	0.02
6	Capture after ko	0.65	6.37
7	2-point semeai	32.07	141.80

Table 4.3: Comparison of local features, between MM and SB

Pattern weights obtained by SB are quite different from those obtained by MM. Figure 4.3 shows that SB has a rather high density of neutral patterns. Observing individual patterns on Table 4.4 shows that patterns are sometimes ranked in a completely different order. Top patterns (first two lines) are all captures and extensions. Many of the top MM patterns are ko-fight patterns. Again, this is because those occur quite often in games by strong human experts. Resolving a ko fight is beyond the scope of this playout policy, so it is not likely that ko-fight patterns help the quality of playouts. Remarkably, all the best SB patterns, as well as all the worst SB patterns (line 3) are border patterns. That may be because the border is where most crucial life-and-death problems occur.

The bottom part of Table 4.4 shows the strangest differences between MM and SB. Lines 5 and 6 are top patterns without atari, and lines 7 and 8 are patterns with the

highest difference in pattern rank. It is quite difficult to find convincing interpretations for most of them. Maybe the first pattern of line 7 (with SB rank 34) allows to evaluate a dead 2×2 eye. After this move, White will probably reply by a nakade, thus evaluating this eye correctly. Patterns with SB ranks 40, 119, and 15 offer White a deserved eye. These are speculative interpretations, but they show the general idea: playing such ugly shapes may help playouts to evaluate life-and-death correctly.

4.5 Against GNU Go on the 9×9 Board

The SB patterns of subsection 4.3.5 were also tested against GNU GO. For having more evident statistical observations, ERICA was set to play with 300 playouts per move to keep the winning rate as close to 50% as possible. The results presented in Table 4.5 indicate that SB performs almost identical to MM. The reason for this result is maybe that progressive bias still has a dominant influence to guide the UCT search within 300 playouts. Also, it is a usual observation that improvement against GNU GO is often much less than improvement against other Monte Carlo programs.

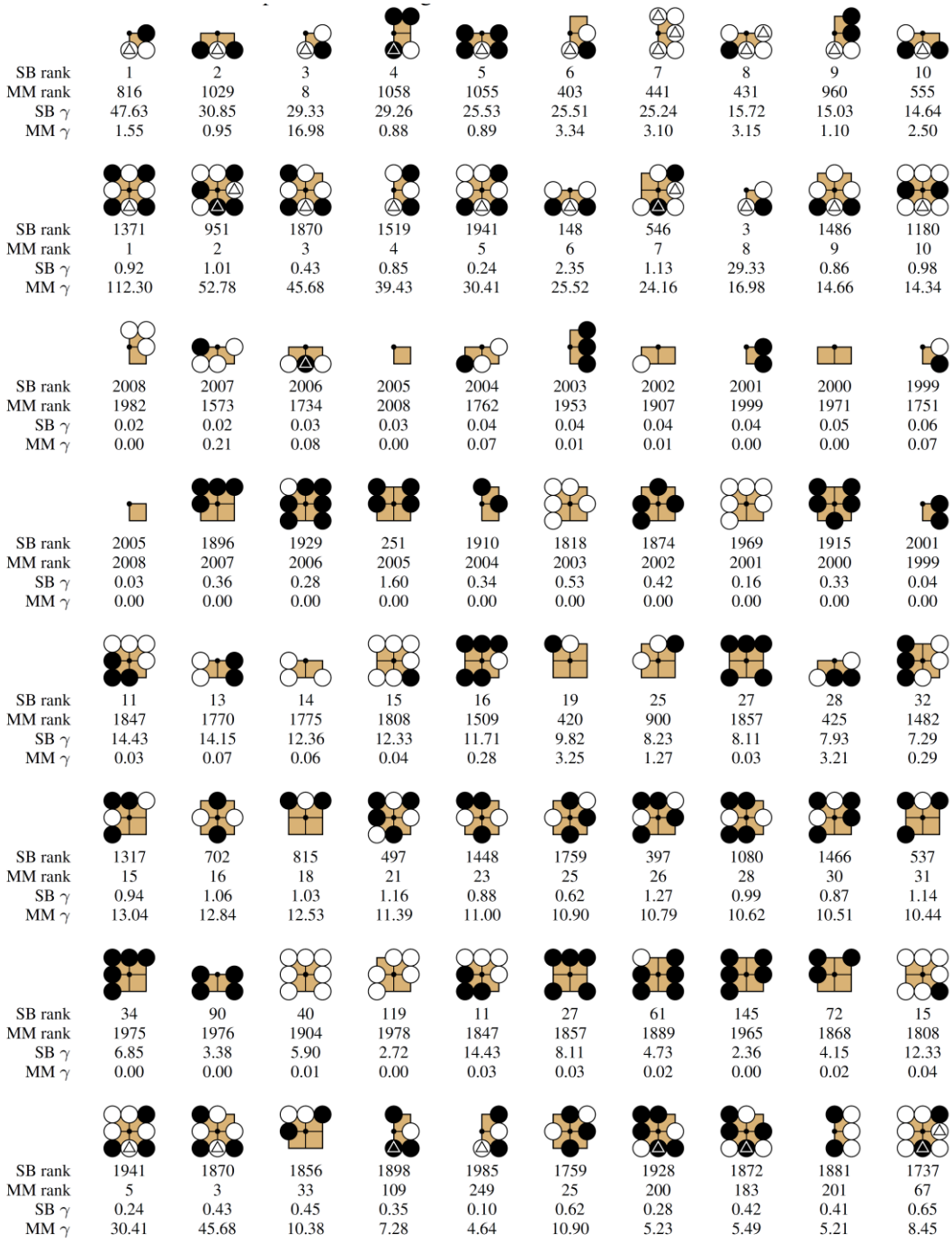

Table 4.4: 3×3 patterns. A triangle indicates a stone in atari. Black to move.

Playout Policy	Winning Rate
Uniform Random	$22.1\% \pm 2.6$
MM	$59.3\% \pm 3.0$
SB	$62.6\% \pm 3.0$

Table 4.5: Results against GNU GO 3.8 Level 10, 1,000 game, 9×9, 300 playouts/move

4.6 Playing Strength on the 19×19 Board

The comparison between MM and SB was also carried out on the 19×19 board by playing against GNU GO 3.8 Level 0 with 1,000 playouts per move. Although the foregoing experiments confirm that SB surpasses MM on the 9×9 board under almost every setting of M, N, and α, MM is still more effective on the 19×19 board. In Table 4.6, the original SB scored only 33.2% with patterns of which the winning rate was 77.9% on the 9×9 board. Even if the γ-values of all local features of SB are replaced by those of MM (MM and SB Hybrid), the playing strength still does not improve at all (33.4%). Nonetheless, the winning rate of SB raises to 41.2% if the γ-value of Feature 1 is manually multiplied by 4.46 (= (19×19)/(9×9)), which was empirically obtained from the experimental results. This clearly points out that patterns computed by SB on the 9×9 board are far from optimal on the 19×19 board. So far, it is not likely to use SB training directly on the 19×19 board, because even the top Go-playing programs are still too weak to offer good evaluations of the 19×19 training set. However, good performance of SB on the 13×13 board can be expected.

Playout Policy	Winning Rate
Uniform Random	$8.2\% \pm 2.4$
SB	$33.2\% \pm 4.1$
MM and SB Hybrid	$33.4\% \pm 4.1$
SB(4.46)	$41.2\% \pm 4.3$
MM	$42.0\% \pm 4.3$

Table 4.6: Results against GNU GO 3.8 Level 0, 500 game, 19×19, 1,000 playouts/move

4.7. Conclusions

Below we provide three conclusions.

(1) The experiments presented in this chapter demonstrate the good performance of SB on the 9×9 board. This is an important result for practitioners of Monte Carlo tree

search, because previous results with this algorithm were limited to more artificial conditions.

(2) The results also demonstrate that SB gives high weights to some patterns in a rather bad shape position. This remains to be tested, but it indicates that SB pattern weights may not be appropriate for progressive bias. Also, learning opening patterns on the 19×19 board seems to be out of the reach of SB, so MM is likely to remain the learning algorithm of choice for progressive bias.

(3) The results of the experiments also indicate that SB has the potential to perform even better. Many improvements seem possible. We mention two of them.

(3a) The steepest descent is an extremely inefficient algorithm for stochastic function optimization. More clever algorithms may provide convergence that is an order of magnitude faster (Schraudolph, 1999), without having to choose meta-parameters.

(3b) It would be possible to improve the training set. Using many more positions would probably reduce risks of overfitting, and may produce better pattern weights. It may also be a good idea to try to improve the quality of evaluations by cross-checking values with a variety of different programs, or by incorporating positions evaluated by a human expert.

Chapter 5

Time Management for Monte Carlo Tree Search Applied to the Game of Go

5.1 Introduction

One of the interesting aspects of MCTS that remains to be investigated is time management. In tournament play, the amount of thinking time for each player is limited. The most simple form of time control, called sudden death, consists in limiting the total amount of thinking time for the whole game. A player who uses more time than the allocated budget loses the game. More complicated time-control methods exist, like byo-yomi[12], but they will not be investigated in this Chapter. Sudden death is the simplest system, and the most often used in computer tournaments. The problem for the program consists in deciding how much of its time budget should be allocated to each move.

Some ideas for time-management algorithms have been proposed in the past (Hyatt, 1984; Markovitch and Sella, 1996; Baum and Smith, 1997; Yoshimoto *et al.*, 2006; Šolak and Vučković, 2009). Past research on this topic is mostly focused on classical programs based on the alpha-beta algorithm combined with iterative

[12] Wikipedia, http://en.wikipedia.org/wiki/Byoyomi.

deepening. The special nature of MCTS opens new opportunities for time-management heuristics. Many ideas have been discussed informally between programmers in the computer-Go mailing list (Sheppard *et al.*, 2009; Yamashita, 2010). This chapter presents a systematic experimental testing of these ideas, as well as some new improved heuristics.

One particular feature of MCTS that makes it very different from alpha-beta from the point of view of time management is its anytime nature. Every single random playout provides new information that helps to evaluate moves at the root. There is no need to wait for a move to finish being evaluated, or for a deepening iteration to complete.

In this Chapter, an enhanced formula and some heuristics are proposed. A state-of-the-art Go-playing program ERICA was used to run the experiments on 19×19 Go and the result shows significant improvement in her playing strength.

5.2 Monte Carlo Tree Search in ERICA and Experiment Setting

All the experiments were preformed on ERICA, running on Dual Xeon quad-core E5520 2.26 GHz. In the 19×19 empty position, ERICA ran 2,600 simulations per second on single core. No opening book is used by ERICA, so that the time allocation formula takes effect immediately from the first move, rather than being delayed by the opening book. The improvement of the playing strength was estimated by playing with GNU GO 3.8 Level 2 with time control of 40 secs sudden death for ERICA and no time limitation for GNU GO. ERICA was set to resign if the UCT mean value of the root node is lower than 30%. For the reference to the playing strength and search speed of ERICA, Table 5.1 shows the winning rate against GNU GO 3.8 Level 2, with

fixed 500 and 1000 playouts per move. ERICA spends 19.6 secs and 46.3 secs on average for each game, respectively. In a sense, fixed playouts per move is also a kind of time management that risks using too less time or losing by timeout.

Playouts	Win Rate	Erica's Time	GNU Go's Time
500	39±2.2%	19.6s	40.3s
1000	61±2.2%	46.3s	44.9s

Table 5.1: Fixed playouts per move against GNU GO 3.8, Level 2, 500 games, 19×19.

5.3 Basic Formula

In this chapter, the remaining time left to the program for the game is defined as RemainingTime. The allocated thinking time given by the time management formula for a position is defined as ThinkingTime. The most basic and intuitive time management formula is dividing the remaining time by a constant to allocate thinking time.

$$\text{ThinkingTime} = \frac{\text{RemainingTime}}{C}$$

According to the basic formula, most of the thinking time is allocated to the first move, then it is decreased gradually until the end of the game. Table 5.2 shows the result of various C for the basic formula. ERICA used less total thinking time on average when C is bigger, 36.5 secs for C = 20 and 27.2 secs for C = 80. However, these 9 seconds of additional thinking time did not bring any improvement to the playing strength (13.4% to 43.2%).

C	Win Rate	Erica's Time	GNU Go's Time
20	13.4±1.5%	36.5s	40.7s
50	36.6±2.2%	32.4s	44.2s
80	43.2±2.2%	27.2s	42.7s
100	43.2±2.2%	24.5s	43.7s
120	37.2±2.2%	21.9s	40.5s
200	32.4±2.1%	14.7s	39.6s
300	25.0±1.9%	10.5s	37.1s

Table 5.2: Basic formula against GNU GO 3.8, Level 2, 500 games, 19×19.

Two observations can be made for MCTS for 19×19 Go. Firstly, using more total thinking time is not bound to stronger playing. Conversely, using less total thinking time may be much stronger if the time is effectively and cleverly allocated. Secondly, allocating more thinking time in the beginning of the game, or the opening stage, is a waste, especially for a program that has not any form of opening or joseki database.

5.4 Enhanced Formula Depending on Move Number

The basic formula is reasonable since the characteristics of MCTS ensure the more accurate search result in the endgame. However, its main drawback is that it allocates too much time to the opening stage. To remedy such weak point, a simple idea is to make the denominator of the basic formula depend on the move count so as to allocate more time in the middle game, where the complicated and undecided semeai and life-and-death conditions appear most frequently. The following is an enhanced formula based on such an idea.

$$\text{ThinkingTime} = \frac{\text{RemainingTime}}{C + \max(\text{MaxPly} - \text{Ply}, 0)}$$

(Ply=0,1,2...)

By this formula, if the program is Black, RemainingTime/(C + MaxPly) is

assigned to the first move, RemainingTime/(C + MaxPly-2) to the second. It reaches the peak and goes back to the form of the basic formula in MaxPly with the value RemainingTime/C. Figure 5.1 gives an example of time per move for C = 80 and MaxPly = 160. The result of different MaxPly is shown in Table 5.3. For comparing with the best experimental performance of the basic formula, C is fixed at 80. The winning rate of ERICA is improved from 43.2% to 49.2% for MaxPly = 80. This strongly confirms the promising effectiveness of the enhanced formula.

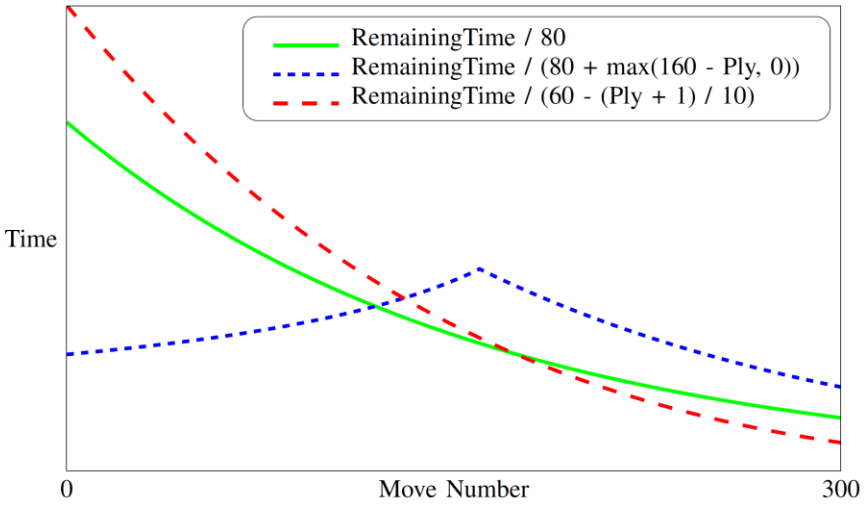

Figure 5.1: Thinking time per move, for different kinds of time-allocation strategies.

MaxPly	Win Rate	Erica's Time	GNU Go's Time
20	42.8±2.2%	24.9s	43.7s
40	46.4±2.2%	27.1s	43.3s
60	43.8±2.2%	26.5s	43.8s
80	49.2±2.2%	25.8s	42.8s
100	47.4±2.2%	24.9s	43.7s
120	46.0±2.2%	23.6s	42.9s
140	48.0±2.2%	22.2s	42.1s
160	47.4±2.2%	21.2s	41.7s
180	44.8±2.2%	19.6s	42.3s

Table 5.3: Enhanced formula (C=80) against GNU GO 3.8, Level 2, 500 games, 19×19.

A similar formula based on the principle of playing faster in the beginning was

proposed by Yamashita (Yamashita, 2010):

$$\text{ThinkingTime} = \frac{\text{RemainingTime}}{60 - (\text{Ply} + 1)/10}$$

(Ply=0,1,2...)

The winning rate of this time-allocation strategy was measured after 500 games to be 40.8±2.2%. It performs even worse than the basic formula. As shown on Figure 5.1, the Yamashita formula still wastes too much time in the beginning of the game.

5.5 Some Heuristics

In this section, the statistical information of the root node is used to dynamically adjust the thinking time. This makes the time allocation stochastic, depending on the internal situation of MCTS. This is correlated with the formula in selection stage. In the first section, the UCT selection formula in ERICA is introduced. The following sections described the heuristics that works successfully in the experiments.

5.5.1 UCT Formula in ERICA

The UCT formula in ERICA is a combination of UCT, RAVE and progressive bias as introduced in Section 3.2.1.

5.5.2 Unstable-Evaluation Heuristic

In the computer Go area, this heuristic was firstly suggested in (Sheppard *et al.*, 2009) and is used after the end of searching for ThinkingTime. The key concept is that if the most visited move has not the largest score, that means either the most visited move was becoming of low score or a better move was just being found or a potential good move was still exploited near the end of the search. For making sure which move is the best, search is performed for another ThinkingTime/2. The result in Table 5.4 shows that this heuristic is very useful. ERICA used around 4 secs more on average

and the winning rate was raised from 47.4% to 54.6% with C = 80 and MaxPly = 160. The improvement demonstrates that such additional 4 secs search effort was cleverly performed in the right occasions.

MaxPly	Win Rate	Erica's Time	GNU Go's Time
80	57.2±2.2%	30.2s	37.8s
100	55±2.2%	29.3s	37.6s
120	50.2±2.2%	27.7s	41.2s
140	54.8±2.2%	26.8s	42.1s
160	54.6±2.2%	25.1s	41s
180	49.4±2.2%	24.1s	41.2s
200	51.6±2.2%	23s	40.9s

Table 5.4: Enhanced formula (C=80) with Unstable-Evaluation heuristic against GNU GO 3.8, Level 2, 500 games, 19×19.

5.5.3 Think Longer When Behind

The main objective of time management for MCTS is to make the program think, in every position during the whole game, for appropriate time to maximize its performance and to win the game. As a result, time management when losing becomes meaningless, since the program will lose anyway, no matter how much time is allocated. This fact introduces the heuristic of thinking another P×ThinkingTime when behind, in which UCT mean of the root node is lower than a threshold T. Since this heuristic is also applied after the end of searching for ThinkingTime, the pseudo code of combining these two heuristics is described in Algorithm 1, to clarify the sequence.

Algorithm 1 Think Longer When Behind

```
ThinkingTime ← AllocateThinkingTime()
Think(ThinkingTime)
if Root.UCTmean < T then
  Think(P × ThinkingTime)
end if
if MostVisitedMove is not HighestValueMove then
  Think(ThinkingTime/2)
end if
Play(MostVisitedMove)
```

Table 5.5 shows this heuristic further improves the winning rate of Erica from 54.6% to 60% with T = 0:4 and P = 1. This indicates that it is effective to make the program think longer to try to reverse the situation when behind.

P	Win Rate	Erica's Time	GNU Go's Time
1.0	60.0±2.2%	27.5s	39.1s
1.5	58.8±2.2%	28.1s	44.0s
2.0	55.2±2.2%	28.0s	43.2s
2.5	53.8±2.2%	28.9s	44.0s
3.0	53.6±2.2%	28.8s	43.5s

Table 5.5: Enhanced formula (C=80, MaxPly=160) with Unstable-Evaluation heuristic and Think Longer When Behind (T=0.4) against GNU GO 3.8, Level 2, 500 games, 19×19.

5.6 Using Opponent's Time

This section discusses the policy of using opponent's thinking time, usually called pondering. The most basic type of pondering, to search as usual when opponent is thinking then re-use the subtree, is discussed in subsection 5.6.1. Subsection 5.6.2 presents another type of pondering, to search and focus on a fixed number of the guessed moves. A heuristic of pondering, reducing thinking time according to simulation percentage of the played move in pondering, is given in subsection 5.6.3. In all the experiments, the enhanced formula with C = 80 and the unstable-evaluation

heuristic are applied. MaxPly was set to 140, 160 and 180 for faster testing speed.

5.6.1 Standard Pondering

Pondering, thinking when the opponent is thinking, is a popular and important technique for Go-playing programs. It enables the program to make use of the opponent's time, rather than simply wait and do nothing. In MCTS, the simplest type of pondering, called Standard Pondering, is to search as if it's our turn to play and re-use the subtree of the opponent's played move. Table 5.6 shows the result of Standard Pondering. ERICA got a big jump in the playing strength (for example, from 54.8% to 67.4% with C =80 and MaxPly=140). GNU GO used more thinking time than ERICA during the game, so this experiment might over-estimate the effect of pondering. Still, it is a clear indication that pondering can have a strong effect on playing strength.

MaxPly	Win Rate	Erica's Time	GNU Go's Time
140	67.4±2.1%	28.6s	43.2s
160	66.2±2.1%	27.2s	43.9s
180	64±2.1%	25.8s	43s

Table 5.6: Standard Pondering against GNU GO 3.8, Level 2, 500 games, 19×19.

5.6.2 Focused Pondering

The other type of pondering, called Focused Pondering, consists in searching exclusively a fixed number N of selected moves, which are considered to be most likely to be played by the opponent. The priority of a move to be selected is the score of UCT formula as mentioned previously. If the opponent's played move is among the selected moves, it is called a ponder hit, otherwise a ponder miss. The prediction rate (PR) is defined as the proportion of the ponder hits, which is ponder hits/(ponder hits+ponder misses). The result of Focused Pondering with N = 10, shown in Table

5.7 indicates that its performance is very limited. For N = 5, shown in Table 5.8, the strength of Focused Pondering is almost identical to that of Standard Pondering. This is maybe because the prediction rate (42% for N = 5 and 57% for N = 10) is not high enough so that the actual played move was not searched as much on average as Standard Pondering would do. The score 69.8% for N = 5 and MaxPly = 160 is likely to be a noise. After a lot of testings, no good result can be found for Focused Pondering.

MaxPly	PR	Win Rate	Erica's Time	GNU Go's Time
140	42.1%	67±2.1%	28.2s	43.4s
160	41.7%	69.8±2.1%	27.6s	43.7s
180	42.8%	62±2.2%	25.8s	43.3s

Table 5.7: Focused Pondering (N=10) against GNU GO 3.8, Level 2, 500 games, 19×19.

MaxPly	PR	Win Rate	Erica's Time	GNU Go's Time
140	57.1%	62±2.2%	27.8s	43.4s
160	57.3%	62.2±2.2%	27.2s	44.1s
180	57.5%	61.8±2.2%	25.9s	43.6s

Table 5.8: Focused Pondering (N=5) against GNU GO 3.8, Level 2, 500 games, 19×19.

The performance of pondering is entirely decided by the search amount of the played move. The tradeoff is between N, the number of selected moves, and the expected search amount that will be performed on them. Strictly speaking, Standard Pondering is also a form of Focused Pondering, with selecting every legal move in the position and 100% prediction rate, to guarantees the played move is in the consideration anyway, even if it is rarely visited.

To evaluate a pondering strategy, it is better to test against an opponent that also

ponders. For this end, ERICA was set to play against herself with different pondering policies. The result of self-play given in Table 5.9 shows that the performance of Focused Pondering is still not significant. It scores 52.8% for N = 3, with a very high prediction rate (53.9%). This again shows the poor performance of Focused Pondering, since it will perform much worse when against a different program along with a much lower predication rate.

N	PR	Win Rate
1	33.7%	52.6±2.2%
3	53.9%	52.8±2.2%
5	66.4%	49.2±2.2%
10	86.2%	47.8±2.2%

Table 5.9: Self-play: Focused Pondering against Standard Pondering, both with Enhanced Formula (C=180, MaxPly=160), 500 games, 19×19.

5.6.3 Reducing ThinkingTime According to the Simulation Percentage

This heuristic is based on the popular idea in human playing: play faster if we guess right in pondering. In MCTS, the degree of rightness or wrongness for a guess can be quantified to the percentage of search performed on the opponent's played move during pondering. And the amount of search can be easily translated to the simulation percentage. In practice, thinking faster means reducing ThinkingTime by the search time spent on the played move. Assume that opponent's thinking time is t, and the simulation percentage of the played move in pondering is s ($0 \leq s \leq 1$), then the reduced time is $t \times s$.

Besides saving time for the rest of the game, playing faster also prevents the opponent from stealing thinking time. The experiment result of self play indicates that this heuristic does not improve performance significantly. Combined with Standard Pondering, its winning rate is 52.4%±2.2, after 500 games, against Standard Pondering. It scores 53.8%±2.2, combined with Focused Pondering (N = 3).

5.7 Conclusions

Experimental results presented in this chapter demonstrate the effectiveness of a variety of time allocation strategies. Playing strength can be improved very significantly with a clever management of thinking time.

An interesting direction for future research would consist in finding good time-allocation schemes for other board sizes. Time management policies are very different between 9×9 and 19×19 Go. In 9×9 Go, the opening book can be very deep to delay the occasion when the time allocation formula is applied. Besides, owning to much smaller search space, the search result of MCTS is usually much more accurate. This explains the preference for allocating more time in the opening stage.

Chapter 6

Conclusions and Proposals for Future Work

In this dissertation, we propose some new heuristics for MCTS applied to the game of Go, including two major contributions. These helped our Go-playing program ERICA win the 19×19 Go tournament at the 2010 Computer Olympiad.

6.1 Simulation Balancing (SB)

The first contribution is Simulation Balancing (SB), discussed in Chapter 4, applied to 9×9 Go. SB is a technique used for training the parameters of the simulation. The experiments demonstrated that ERICA, equipped with the SB parameters, was almost 90 Elo stronger than that with MM.

Although SB did not work well on the 19×19 board in our experiments, the good results on the 9×9 board strongly indicates that the gammas from Minorization-Maximization (MM) are far from optimal. This fact convinced us to manually tune the MM gammas, especially for the tactical features, on the 19×19 board and it contributed considerably to raise ERICA's playing strength. SB experiments also confirm the generally maintained conjecture about the playout

policy: to evaluate the playout correctly, local tactical responses should be absolute, and patterns only matter if there are no local tactics. This principle explains why MOGO-type simulation does work, though it still has much less flexibility than CRAZYSTONE-like simulation.

The reason why SB prefers to play so many bad patterns remains to be investigated. SB has more potential to perform even better by speeding up the learning algorithm or improving the evaluation of the training positions. Also, the good performance of SB on the 13×13 board can be appropriately expected.

6.2 Time Management

The second contribution of this research is the systematic experiments of the various time management schemes for 19×19 Go, discussed in Chapter 5. These time management algorithms were performed on ERICA with the time setting 40 seconds Sudden Death (SD). The experiments confirmed the effectiveness of several heuristics, such as the enhanced formula and the unstable-evaluation Heuristic. Pondering was also proved to be very crucial to playing strength.

One of the future and interesting directions is to investigate the time management schemes on other board sizes, such as 9×9, especially with a strong opening book. Other heuristics using the statistical data of MCTS also remains to be explored and tested. It is of great interest to try these heuristics with a longer time setting as well.

6.3 Other Prospects

We believe *adaptive playout* is a rising star of computer Go. To solve the semeai problems completely, heavy knowledge implementation scarcely can be the right way, because there are really too many cases for a perfect algorithm to exist. Letting the

playout learn from itself, under the framework of the softmax policy, is of great interest to investigate. Also, feeding the information from the tree to the playout in order to solve the semeai problems, as one author (Yamato, the author of ZEN) suggested recently in the computer Go mailing list, also deserves considerable attention. We hope this research can be inspiring and encouraging to other computer Go researchers.

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.

Allis, V. (1994). Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, Masstricht, The NetherLands.

Anderson, D.A. (2009). Monte Carlo search in Games. Technical report, Worcester Polytechnic Institute.

Audouard, P., Chaslot, G., Hoock, J.-B., Rimmel, A., Perez, J., and Teytaud, O. (2009). Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go. In *EvoGames, Tuebingen Allemagne. Springer*.

Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R. E. (1995). Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 322-331, IEEE Computer Society Press, Los Alamitos, CA.

Auer, P., Cesa-Bianchi, N. and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning Journal*, 47, pp. 235-256.

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

Baudis, P. (2011). Exploration formulas for UCT. Computer-Go mailing list, http://www.mail-archive.com/computer-go@dvandva.org/msg02154.html. Retrieved at 2011-07-17 T19:45:45+08:00.

Baudis, P. and Gailly, J.-l. (2011). Pachi: Software for the Board Game of Go / Weiqi /

Baduk.
http://pachi.or.cz/.
Retrieved at 2011-07-14 T14:38:10+08:00.

Baum, E. B. and Smith, W. D. (1997). A Bayesian Approach to Relevance in Game Playing. *Artificial Intelligence*, Vol. 97, No.1–2, pp. 195-242.

Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hérault, T., Hoock, J.-B., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P., and Yu, Z. (2010). Scalability and Parallelization of Monte-Carlo Tree Search. *Proceedings of the 5th International Conference on Computer and Games 2010*.

Bouzy, B. (2003). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, Vol. 175, pp. 247-257.

Bouzy, B. (2005a). Move Pruning Techniques for Monte-Carlo Go. In J*oint Conference on Information Sciences*. *Proceedings of the 11th Advances in Computer Games conference* (ACG-11), pp. 104-119.

Bouzy, B. (2005b). History and territory heuristics for Monte-Carlo Go. In *Heuristic Search and Computer Game Playing Session*, *Joint Conference on Information Sciences*, Salt Lake City.

Bouzy, B. and Cazenave, T. (2001). Computer Go: an AI-oriented Survey. *Artificial Intelligence*, Vol. 132, issues 1, pp. 39-103.

Bouzy, B. and Chaslot, G. (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 & Simon Lucas), pp. 176-181.

Bouzy, B. and Chaslot, G. (2006). Monte-Carlo Go Reinforcement Learning Experiments. *2006 IEEE Symposiumon Computational Intelligence and Games* (eds. G. Kendall and S. Louis), pp. 187-194, Reno, USA.

Bouzy, B. and Helmstetter, B. (2003). Monte-Carlo Go Developments. *Proceedings of the 10th Advances in Computer Games conference* (ACG-10), Graz 2003, Book edited by Kluwer, H. Jaap van den Herik, Hiroyuki Iida, Ernst A. Heinz, pp. 159-174.

Brügmann, B. (1993). Monte Carlo Go. Not formally published.
http://www.althofer.de/Bruegmann-MonteCarloGo.pdf.
Retrieved at 2011-07-18 T10:16:45+08:00.

Burmeister, J. and Wiles, J. (1995). The Challenge of Go as a Domain for AI Research: a Comparison Between Go and Chess. *Proceedings of the Third Australian and New Zealand Conference on Intelligent Information System*, pp. 181-186.

Campbell, M., Hoane, A. J. and Hsu, F. H. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, issues 1-2, pp. 57-83.

Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., and Teytaud, O. (2009). Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. *Proceedings of the Twelfth International Advances in Computer Games Conference*, pp. 1-13, Pamplona, Spain.

Chaslot, G., J-B. Hoock, J.-B., Perez, J., Rimmel, A., Teytaud, O. and Winands, 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.

Chaslot, G., Winands, M., Bouzy, B., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2007). Progressive Strategies for Monte-Carlo Tree Search. *Proceedings of the 10th Joint Conference on Information Sciences* (ed.P. Wang), pp. 655–661, Salt Lake City, USA.

Chaslot, G., Winands, M. and Herik, H. J. van den (2008). Parallel Monte-Carlo Tree Search. P*roceedings of the Conference on Computers and Games 2008*, Vol. 5131 of Lecture Notes in Computer Science, pp. 60-71.

Chen, K. (1989). Group identification in Computer Go. *Heuristic Programming in Artificial Intelligence*, Levy & Beal (Eds.), pp. 195-210.

Chen, K. and Chen, Z. (1999). Static analysis of life and death in the game of Go. *Information Science*, Vol. 121, pp. 113-134.

Coquelin, P.-A. and Munos, R. (2007). Bandit Algorithm for Tree Search, Technical Report 6141, INRIA.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th International Conference on Computer and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. J. Donkers), Vol. 4630 of Lecture Notes in Computer Science, pp. 72-83, Springer, Turin, Italy.

Coulom, R. (2007). Computing Elo Ratings of Move Patterns in the Game of Go. *ICGA Journal*, Vol. 30, No. 4, pp. 198-208.

Coulom, R. (2010). Bayesian Elo Rating.
http://remi.coulom.free.fr/Bayesian-Elo/#usage.
Retrieved at 2011-07-17 T17:19:35+08:00.

Drake, P. (2009). The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA Journal* Vol. 32, pp. 221-227.

Drake, P. (2011). Orego.
http://legacy.lclark.edu/~drake/Orego.html.
Retrieved at 2011-07-14 T14:43:55+08:00.

Enzenberger, M. and Müller, M. (2009). A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. In *Advances in Computer Games (ACG12)*, pp. 14-20, Pamplona Espagne, Berlin: Springer.

Enzenberger, M., Müller, M., Arneson B. and Segal R. (2010). Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 259-270. Special issue on Monte Carlo Techniques and Computer Go.

Fotland, D. (2011). Exploration formulas for UCT.
http://www.mail-archive.com/computer-go@dvandva.org/msg02143.html.
Retrieved at 2011-07-17 T19:47:50+08:00.

Fotland, D. (2010). ERICA wins Go 19x19 Tournament. *ICGA Journal*, Vol. 33, pp. 174-178.

Fotland, D. (2002). Static Eye in "The Many Faces of Go". *ICGA Journal*, Vol. 25, No. 4, pp. 203-210.

Fotland, D. (1996). World Computer Go Championships.
http://www.smart-games.com/worldcompgo.html.
Retrieved at 2011-07-17 T19:44:35+08:00.

Fotland, D. (1993). Knowledge Representation in The Many Faces of Go.
http://www.smart-games.com/knowpap.txt.
Retrieved at 2011-07-17 T19:43:01+08:00.

Gelly, S., Hoock, J.-B., Rimmel, Arpad, Teytaud, O. and Kalemkarian, Y. (2008). The Parallelization of Monte-Carlo Planning. In *International Conference on Informatics in Control, Automation and Robot*, Madeira, Portugal.

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. Proceedings of the 24th International Conference on Machine Learning, pp. 273-280, Corvallis Oregon USA.

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

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

Gaudel, R., Hoock, J.-B., Pérez, J., Sokolovska, N., Teytaud, O. (2010). A Principled Method for Exploiting Opening Books. *Proceedings of the 5th International Conference on Computer and Games 2010*, pp. 136-144.

Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *Readings in computer vision: issues, problems, principles, and paradigms*, pp. 564-584.

Graepel, T., Goutrié, M., Krüger, M. and Herbrich R. (2001). Learning on Graphs in the Game of Go. *Lecture Notes in Computer Science*, Vol. 2130/2001, pp. 347-352.

Herik, H. J. van den, Uiterwijk, J. W. H. M., Rijswijck, J. van. (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, pp. 277-311.

Helmbold, D. P. and Wood, A. P. (2009). All-Moves-As-First Heuristics in

Monte-Carlo Go. In *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI'09/ISBN 1-60132-108-2*, Editors: Arabnia, de la Fuente, Olivas, pp. 605-610, Los Vegas, USA.

Hendrik, B. (2010). Adaptive Playout Policies for Monte-Carlo Go. Master thesis, Institut für Kognitionswissenschaft, Universität Osnabrück.

Kocsis, L. and Szepesv´ari, C. (2006). Bandit Based Monte-Carlo Planning. In J. Furnkranz, T. Scheffer and M. Spiliopoulou (eds.), *Machine Learning: ECML 2006, Lecture Notes in Artificial Intelligence 4212*, pp. 282-293.

Jasiek, R. (1997). Elementary Rules.
http://home.snafu.de/jasiek/element.html.
Retrieved at 2011-07-17 T19:42:20+08:00.

Huang, S. C., and Yen, S. J. (2010). Many Faces of Go Wins Computer 13×13 Go Tournament. *ICGA Journal*, Vol. 33, No. 3, pp. 172-173.

Hyatt, R. M. (1984). Using time wisely. *ICCA Journal*, Vol. 7, No. 1, pp. 4-9.

Kato, H. (2008). More UCT / Monte-Carlo questions (Effect of rave). Computer-Go mailing list.
http://www.mail-archive.com/computer-go@computer-go.org/msg07135.html.
Retrieved at 2011-07-17 T19:48:49+08:00.

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

Lew, L. (2010). Library of effective Go routines.
http://github.com/lukaszlew/libego.
Retrieved at 2011-07-14 T14:27:45+08:00.

Lichtenstein, D. and and Sipser, M. (1978). Go is PSPACE-hard. *Foundations of Computer Science*, pp. 48-54.

Lin, C. H. (2009). Web2Go web site.
http://web2go.board19.com/.
Retrieved at 2011-07-26 T12:07:05+08:00.

Markovitch, S. and Sella, Y. (1996). Learning of resource allocation strategies for game playing. *Computational Intelligence*, vol. 12, pp. 88-105.

Müller, M. (2002). Computer Go. *Artificial Intelligence*, Vol. 134, issues 1-2, pp. 145-179.

Persson, M. (2010). Valkyria. Sensei's Library.
http://senseis.xmp.net/?Valkyria.
Retrieved at 2011-07-14 T14:33:12+08:00.

Rimmel, A., Teytaud, F. and Teytaud, O. (2010). Biasing Monte-Carlo Simulations through RAVE Values. In *Proceedings of the 5th International Conference on Computer and Games 2010*, pp. 59-68.

Rosin, C. D. (2010). Multi-armed bandits with episode context. *Proceedings of ISAIM 2010*.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, pp. 1203-1212.

Segal, R. (2010). On the Scalability of Parallel UCT. *Proceedings of the 5th International Conference on Computer and Games 2010*.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 7th series, Vol. 41, No. 314, pp. 256-275.

Sheppard, B., Williams, M., Dailey, D., Hillis, D., Nentwich, D., Persson, M., Fotland, D. and Boon, M. (2009). Tweak to MCTS selection criterion. Computer-go mailing list,http://groups.google.com/group/computer-go-archive/browse_thread/thread/7531bef033ca31f6, Jun. 2009.

Silver, D. (2009). Reinforcement learning and simulation-based search in computer Go. Ph.D. dissertation, University of Alberta.

Slate, D. J., and Atkin, L. R. (1977). Chess 4.5 - The Northwestern University Chess Program, *Chess Skill in Man and Machine*, P.W. Frey (ed.), Springer-Verlag, New

York, pp. 82-118.

Smith, A. (1908). The game of Go, the national game of Japan.

Šolak, R. and Vučković, R. (2009). Time management during a chess game. *ICGA Journal*, vol. 32, No. 4, pp. 206-220.

Stern, D., Herbrich, R. and Graepel, T. (2006). Bayesian Pattern Ranking for Move Prediction in the Game of Go. In *Proceedings of the International Conference of Machine Learning*.

Stogin, J., Chen, Y.-P., Drake, P., and Pellegrino, S. (2009). The Beta Distribution in the UCB Algorithm Applied to Monte-Carlo Go. In *Proceedings of the 2009 International Conference on Artificial Intelligence*, CSREA Press.

Tesauro, G., Rajan, V. T., Segal, R. (2010). Bayesian Inference in Monte-Carlo Tree Search , In *Proceedings of the Conference on Uncertainties in Artificial Intelligence 2010*.

Teytaud, O. (2011). News on Tromp-Cook.
http://www.mail-archive.com/computer-go@dvandva.org/msg02171.html.
Retrieved at 2011-07-17 T19:50:20+08:00.

Tromp, J. and Farnebäck, G. (2006). Combinatorics of Go. *Proceedings of 5th international Conference on Computer and Games*, pp. 241-249.

Xie, F., Liu, Z. (2009). Backpropagation Modification in Monte-Carlo Game Tree Search. *Intelligent Information Technology Application, 2009 (IITA 2009)*, pp. 125-128.

Yajima, T., Hashimoto, T., Matsui, T., Hashimoto, J. and Spoerer, K. (2010). Node Expansion Operators for the UCT Algorithm. *Proceedings of the 5th International Conference on Computer and Games 2010*, pp. 116-123.

Yamashita, H. (2010). time settings. Computer-Go mailing list.
http://dvandva.org/pipermail/computer-go/2010-July/000687.html.
Retrieved at 2011-07-17 T19:50:56+08:00.

Yamashita, H. (2011). Exploration formulas for UCT. Computer-Go mailing list. http://www.mail-archive.com/computer-go@dvandva.org/msg02155.html. Retrieved at 2011-07-17 T19:51:22+08:00.

Yamato. (2011). News on Tromp-Cook. http://www.mail-archive.com/computer-go@dvandva.org/msg02184.html. Retrieved at 2011-07-17 T19:51:43+08:00.

Yen, S. J. (1999). Design and Implementation of Computer Go Program Jimmy 5.0. Ph.D. dissertation, National Taiwan University.

Yoshimoto, H., Yoshizoe, K., Kaneko, T., Kishimoto, A. and Taura, K. (2006). Monte Carlo Go Has a Way to Go. In *Proceedings of the 21st National Conference on Artificial Intelligence*, AAAI Press, pp. 1070-1075.

Zobrist, A. (1969). A model of visual organization for the game of Go. *Proceedings of AFIPS Spring Joint Computer Conference*, Boston, AFIPS Press, Montvale, NJ, pp. 103-111.

Zobrist, A. (1970). Feature extractions and representation for pattern recognition and the game of Go. Ph.D. Thesis, Graduate School of the University of Wisconsin, Madison, WI.

Appendix A. Publication List

(a) Journal Papers

[1] Huang, S. C., Coulom, R. and Lin, S. S. (2010). Monte-Carlo Simulation Balancing Applied to 9×9 Go. *ICGA Journal*, Vol. 33, No. 4, pp. 191-201.

[2] Huang, L. T., Chen, S. T., Huang, S. C. and Lin, S. S. (2007). An Efficient Approach to Solve Mastermind Optimally. *ICGA Journal*, Vol. 30, No. 3, pp. 143-149.

(b) Conference Papers

[1] Huang, S. C., Coulom, R. and Lin, S. S. (2010). Monte-Carlo Simulation Balancing in Practice. In *Proceedings of the International Conference on Computers and Games 2010 (CG2010)*, JAIST, Kanazawa, Japan, September 24-26.

[2] Huang, S. C., Coulom, R. and Lin, S. S. (2010). Time Management for Monte-Carlo Tree Search Applied to the Game of Go. *International Workshop on Computer Games (IWCG 2010)*, Hsinchu, Taiwan, November 18-20.

[3] Lin, S. S., Chen, B. and Huang, S. C. (2004). Building a Multi-modal Multimedia Information Retrieval System for the National Museum of History. In *Proceedings of the 2004 International Conference on Digital Archive Technologies (ICDAT2004)*, pp. 121-130, Taipei, Taiwan.

(c) Tournament Report

[1] Huang, S. C. and Yen, S. J. (2010). Many Faces of Go wins Computer 13×13 Go Tournament. *ICGA Journal*, Vol. 33, No. 3, pp. 172-173.