WARSAW UNIVERSITY

Faculty of Mathematics,
Informatics and Mechanics

Modeling Go Game as a Large Decomposable Decision Process

PhD Thesis

Łukasz Lew

Supervisor:
prof. Krzysztof Diks

Warsaw, June 2011

Author’s declaration

Aware of legal responsibility I hereby declare that I have written this thesis myself and all its contents have been obtained by legal means.

Date	Author’s signature

Supervisor’s declaration

This thesis is ready to be reviewed.

Date	Supervisor’s signature

Abstract

Research on computer Go is almost as old as research on the other board games. But until recent developments of Monte Carlo Go, programs were not considered to be particularly strong. The key to the current programs strength is the quality of the moves in the game continuation samples. In the state of the art programs, moves in samples are chosen based on a local (to the move) Go heuristics. This heuristics are usually learnt from game records of good players using Minorization-Maximization (MM) algorithm. MM is the state of the art but it can process only a limited number of games due to memory constraints of existing hardware. In this thesis, we design and present a better, online algorithm, capable of learning from unconstrained data set. We prove its convergence.

State of the art Go programs play on a regular, 19×19 Go board on the level of a strong club player and on a small 9×9 board on a near professional level. Such discrepancy in playing strength is uncommon among human players, who tend to play on a similar level regardless of the board size. We attribute Monte Carlo Go weakness on a regular board to the ineffectiveness of a global Monte Carlo Tree Search (MCTS). 19×19 Go positions usually consist of a multiple smaller independent subgames that human players can analyze independently. MCTS is capable of analyzing any of them separately but globally it is ineffective due to combinatorial explosion when subgames' moves can interlace.

We review two decomposition-based, promising algorithms trying to work around that ineffectiveness by exploiting locality in Go positions. For each of them, we discover and describe the reason why it can't work well. Finally we propose a new, divide and conquer algorithm based on Combinatorial Game Theory. We show that it must converge to a near-optimal strategy, and experimentally show the convergence on a toy example.

Keywords: Monte Carlo Go, Monte Carlo Tree Search, Bradley-Terry model, Laplace$_q$ Marginal Propagation, Combinatorial Game Theory, Adaptive Playouts, Libego

ACM Classification: I.2.1, I.2.6, I.2.8

Streszczenie

Badania nad algorytmami grającymi w Go są prawie tak stare jak badania nad innymi grami planszowymi. Do niedawna, programy grające w Go nie były uważane za silne. Siła programów Go wzrosła dzięki rozwinięciu nowej techniki - Monte Carlo Go. Kluczem do siły gry obecnych programów jest jakość ruchów w próbkach kontynuacji gry. W najlepszych programach, ruchy w próbkach wybierane są z użyciem lokalnych (względem ruchu) heurystyk. Heurystyki te na ogół są tworzone na bazie zapisów gier dobrych graczy, za pomocą algorytmu Minorization-Maximization (MM). MM, mimo że jest obecnie najlepszym znanym algorytmem, może przetworzyć tylko ograniczoną ilość danych, ze względu na ograniczenia pamięciowe istniejącego sprzętu. W poniższej pracy projektujemy i prezentujemy nowy, lepszy algorytm typu “online”, który jest zdolny do uczenia się z nieograniczenie dużego zbioru danych. Dowodzimy że nasz algorytm zbiega.

Najlepsze programy grają na planszy 19×19 z siłą dobrego gracza klubowego, a na małej planszy 9×9 z siłą bliską zawodowego gracza. Taki rozrzut w sile gry nie jest spotykany wśród ludzi, którzy na ogół grają z podobną siłą, niezależnie od rozmiaru planszy. Naszym zdaniem powodem słabości algorytmu Monte Carlo Go na planszy 19×19 jest nieefektywność globalnego algorytmu Monte Carlo Tree Search (MCTS). Pozycje na planszy 19×19 składają się na ogół z wielu mniejszych i niezależnych podgier, które ludzie potrafią analizować niezależnie od siebie. MCTS jest zdolne do analizy każdej podgry z osobna. Ale globalnie, gdy ruchy podgier mogą się przeplatać, MCTS jest nieefektywne z powodu kombinatorycznej eksplozji ilości kombinacji.

W niniejszej pracy analizujemy dwa obiecujące algorytmy oparte o dekompozycje sytuacji, które próbują obejść tą nieefektywność, wykorzystując lokalność w sytuacjach Go. Dla każdego z nich odkrywamy i opisujemy powód, przez który te algotmy nie mogą dobrze działać. Proponujemy również nowy algorytm typu ’dziel i rządź’, oparty o Kombinatoryczną Teorię Gier. Pokazujemy, że algorytm zbiega w przybliżeniu do optymalnej strategii i potwierdzamy to doświadczalnie na małym przykładzie.

Słowa kluczowe: Monte Carlo Go, przeszukiwanie drzewa Monte Carlo, model Bradley’a-Terry’ego, propagacja rozkładów brzegowych z rzutowaniem Laplace$_q$, kombinatoryczna teoria gier, adaptatywne symualcje, Libego

Klasyfikacja tematyczna ACM: I.2.1, I.2.6, I.2.8

Acknowledgements

My research and interest in the topic of Monte Carlo Go began with my Master's thesis which was carried out under supervision of John Tromp. Most of the research for this thesis was carried out during my PhD studies at the University of Warsaw under supervision of Krzysztof Diks.

Most of the progress was made during the internship at Microsoft Research, Cambridge, in collaboration with Thore Greapel and Ralf Herbrich and during the internship in Inria, Lille in collaboration with Remi Coulom. I would like to thank my supervisor and collaborators.

I would like to express a special thanks to Remi. Your support went far beyond technical and without it I would not be able to complete this thesis.

I also would like to thank my girlfriend Kasia Kupniewska who supported me through the whole length of my research. Your support was essential for me.

I express my gratitude to Marcin Mucha and Witold Szczechla who helped me to complete the proof of the convergence of the LMP algorithm.

I would like to thank Krzysztof Stachlewski who spent a lot of time with me polishing the thesis's language.

Kasia Główczak, Monika Skowrońska, Cezary Kaliszyk, Diana Błaszczyk and Aja Huang. You all supported and helped me and I am very grateful to all of you.

Last but not least I would like to thank to my sister Ola and my parents who supported and believed in me, always.

Contents

Chapter 1

Introduction

The research on Go playing algorithms is an exciting topic. The main reason for this is the discovery of *Monte Carlo Go* (MC Go) in 2003 [BH03], a new approach for decision making in Go game and other domains.

MC Go replaces a traditional “brute-force” combination of alpha-beta tree search and evaluation function [KM75] with a statistical analysis of outcomes of random game continuations. MC Go is a simple and elegant algorithm, dramatically simpler than “classical” computer Go algorithms. Thanks to this simplicity it showed a great promise for improvement and made a great research topic. In fact, in the last few years it turned out that MC Go did improve and far overcame the playing strength of classical Go programs. Today, MC Go is a central framework in which new algorithms and improvements are being developed and tested. The limitations of MC Go are well known but despite the simplicity of MC Go, they are difficult to overcome.

Monte Carlo approach is being adopted by other games [SWH$^+$08, SCS09, FB08] and even seemingly unrelated combinatorial optimization problems [dMRVP09, RTC11, RST09, BSB11, GS10]. This stands in contrast with computer chess where years of research on specialized algorithms do not seem to benefit other domains. Compared to Monte Carlo framework, computer chess research may seem a “dead-end” in computer science.

One more reason which makes computer Go research an exciting research topic is the fact that Go is the last popular board game with a perfect information where computers do not dominate humans. Until a few years ago computer Go programs had played only on an average club player level. This level can be achieved in a few months by a motivated player and it is very far from the strength of top human players.

1.1 Place of game research in computer science

Computer science researchers always strived to create artificial intelligence that would surpass human beings. This task in its greatest generality is currently considered to be one of the most ambitious goals in computer science. The difficulty of it resulted in focusing the research on smaller and more closed domains.

Games were always one of the most convenient domains used in AI research due to well-defined rules and goals. Game of chess in particular attracted the most attention thanks to its popularity as a hobby and as a sport. The place of chess in AI was sometimes compared to the place of a "fruit fly" in genetics – the most popular domain to try new ideas. The game of chess became a popular benchmark of computers' ability to "think". The state of the art chess-playing programs rely on a "brute force" approach. It consists of a deep mini-max game tree search with an evaluation function used at the leaves.

The strength of such an algorithm comes mainly from its error-free calculation of the possible continuations of the game. The reoccurring comment of computer-human games is "the human eventually will make a mistake that will be exploited by the computer". The weaknesses of chess-playing algorithms, when compared to humans, is the lack of accurate judgement of the position's value.

In modern human-computer games, people always try to direct the game to so called closed positions. In such positions, tactics (sequences of moves) do not play such a big role in evaluation and human knowledge can be most useful. For this reason chess programs are often explicitly programmed to direct the game into open positions. In such positions, a dazzling number of sequences maximizes the probability of a human mistake. In 1997, a specialized super-computer Deep Blue beat the world chess champion Gary Kasparov [CHhH02, AChH90]. Since then the popularity of chess as a research topic has decreased.

Since the popularization of Monte Carlo framework, game of Go, which is known for its difficulty for computers, replaced chess in its position as a central AI research domain. Algorithms developed for the chess domain were sometimes used in other games but found little application in other fields of computer science. Monte-Carlo algorithms recently developed for the computer Go already found many interesting application in various decision domains [dMRVP09, RTC11, RST09, BSB11, GS10].

1.2 Overview of Monte Carlo Go approach

Monte Carlo Go is based on statistical analysis of results of a large number of *playouts*. Playouts are simulated game continuations that start at the given Go position and continue until there are no legal moves left. At that point the playout's result can be evaluated using only the game's rules. This stands in contrast with the classical approach based on a heuristic evaluation function.

The quality of a particular Monte Carlo Go algorithm depends mostly on the quality of the moves used in playouts. In state of the art Monte Carlo Go algorithms, the first few moves are chosen by *Monte Carlo Tree Search* (MCTS). In order to do so, MCTS stores statistics in the nodes of the in-memory game tree. These statistics summarize the results of previous playouts. We can see that MCTS is adaptive, i.e. the result of every single playout makes new playouts smarter. Given enough time, MCTS will converge to an optimal minimax tree.

During the playout, when the in-memory game tree ends, the playout is taken over by Heavy Playout, a fixed policy based on hard-coded Go knowledge. Its quality is crucial for the final algorithm's playing strength.

1.3 Contribution and outline of this thesis

In order to understand the challenges and complexities arising in computer Go, one must know not only the rules of the game but also common situations that can arise during the game. Chapter 2 explains the rules and shows a small bit of the complexity of Go by presenting typical Go situations on examples.

Chapter 3 describes the history and state of the art of computer Go. It also reviews the most important publications of the domain. Almost all the improvements developed for Monte Carlo Go rely on introducing a Go knowledge in various places of Monte Carlo framework. The notable exceptions are Monte Carlo Tree Search (MCTS) and RAVE heuristic. These algorithms are domain independent and found applications in several other domains than game playing.

The original goal of author's research was to create a domain-independent algorithm that would be able to learn the domain knowledge from experience in a given domain. This goal proved to be overly ambitious. Author's research focus shifted towards a narrower goal of learning from experience in the game of Go. This resulted in the *Laplace$_q$ Marginal Propagation* algorithm.

After some time, the author's research shifted towards the analysis of the fundamental weaknesses of the Monte Carlo Go framework. This lead to the rest of the results presented in this thesis and listed in the next paragraphs.

Learning a Heavy Playout policy. The first contribution of this thesis is a new algorithm for inferring a good Heavy Playout policy from game records of good players. In fact, this thesis provides a much more general algorithm. Chapter 4 describes:

- *Laplace$_q$ Marginal Propagation* (LMP)

LMP is a new efficient machine learning algorithm for Bayesian Models. It is applied to Bradley–Terry model to create an algorithm for learning the pattern-values from game records to be used in Heavy Playouts. It is a big improvement over the state of the art [Cou07a] because its space complexity is much smaller. This allows to process a much larger data sets which are available in Go domain.

Adaptive playouts. Even with a rich data, offline policy learning cannot replace online policy learning from a given Go position. This is a job of the Monte Carlo Tree Search. But MCTS, being a global search algorithm, has a very limited capability of searching through positions with many independent subgames which are typical for Go game. As a result, playouts are mostly done with Heavy Playout policy. In contrast to MCTS, Heavy Playouts are fixed and do not use results of previous playouts to improve. It takes an extremely large number of playouts for MCTS to overtake a bias of Heavy Playout policy. A concrete example is given in Chapter 5.

Adaptive Heavy Playouts are considered by many researchers a difficult and one of the most important research topics [Mü10, Cou07b]. There have been many attempts to use various techniques analogous to MCTS but all of them failed. It is not obvious why *naive adaptive playouts* do not work. Another promising approach for adaptive playouts was *Reinforcement Learning with a linear regression model for value function* [CGJB$^+$08, GS07]. But it does not work well either. The second, third and fourth contributions, described in Chapter 5, are:

- Explanation why widely tested, naive pattern-based adaptive playouts do not work.
- A counter example showing that the linear regression model is a bad model for value function in two-player games. A general explanation why the linear regression model cannot work well.
- A new model based on Combinatorial games for Monte-Carlo Reinforcement Learning.
- A new triangle approximation for thermographs.

Libego. Last but not least contribution of this thesis is:

- The Libego library.

Libego is a Go engine and a library that influenced many other Go engines. Libego is used in this thesis and several unrelated research papers. The library and the third party research using Libego are described in Chapter 6.

1.4 Research time-line

Author's research and interest in the topic of Monte Carlo Go began during his Master's thesis which was carried out under supervision of John Tromp. Most of the research for this thesis, time-wise, was conducted during author's PhD studies at the University of Warsaw in years 2005-2010. The biggest progress was made during the internship in Microsoft Research, Cambridge, in cooperation with Thore Greapel and Ralf Herbrich, and during the internship in Inria, Lille, in cooperation with Remi Coulom.

Chapter 2

Go Rules and Basic Strategies

In order to understand the algorithms presented in the next chapters, at least basic knowledge of the history, rules and difficulties of the Go game in necessary.

2.1 History

The game of Go is one of the oldest games in the world. According to the legend, it was invented around 2300BC by a Chinese Emperor Yao for his son Danzhu to teach him discipline, concentration and strategy. The earliest written references to the game of Go can be found in several Chinese books from the 4th and the 3rd century BC. It is believed that Go originated in China and spread to Korea and Japan around the 5th and the 7th centuries. In ancient China, Go game along with painting, calligraphy and playing musical instrument guqin were considered to be the four main arts required for a Chinese scholar.

Today, Go is a very popular game in many Asian countries. In China, Korea, Japan and Taiwan there are tournaments with prizes worth hundreds thousands of dollars. There is even a university major concentrated on Go in South Korea.

Most importantly, there is an institution of a professional player - a *pro*. This profession is sanctioned by each country's national Go institutes. The minimal standard to acquire a professional diploma is very high due to a tremendous competition. Big prizes in tournaments are a major incentive but so is the self-perfection and competition itself. A pro player's life concentrates on studying of Go game, improvement of his playing strength and participation in tournaments. Only few amateurs can compete with weakest professional players, and non of them can compete with top pros.

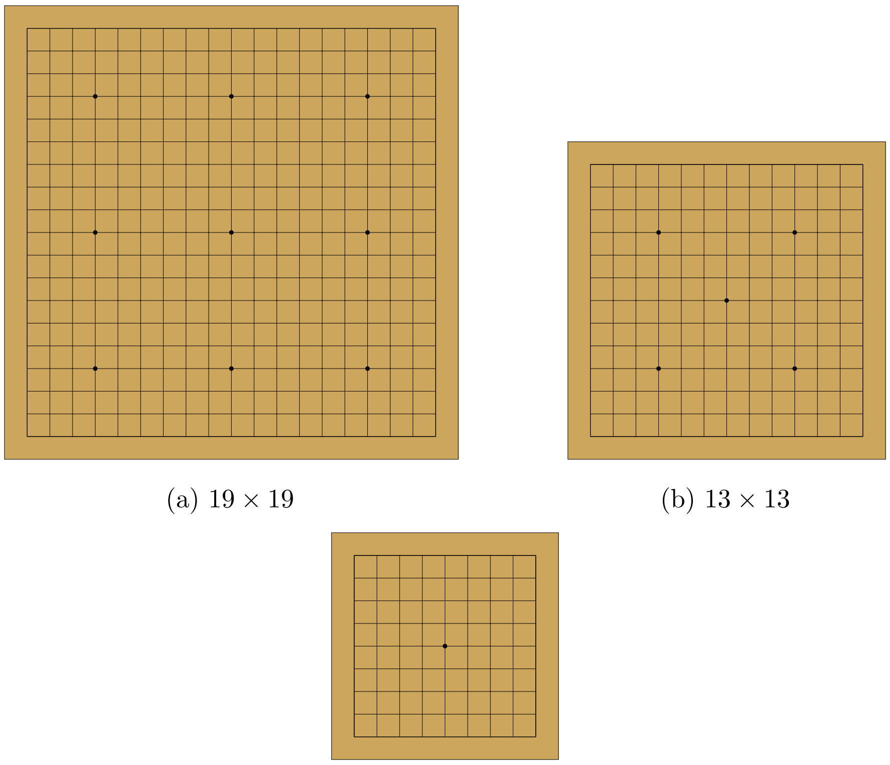
In the mentioned countries Go is very popular as a spectator sport and as an intellectual recreation. To illustrate the popularity we just mention that there are newspapers, magazines and even TV channels dedicated solely to Go game.

2.2 Rules

“One hour to learn, lifetime to master.” says old proverb describing the game of Go. In fact Go rules are very simple, but the game itself is deep and multi-layered.

There are many variants of Go rules. Fortunately it almost never affects the optimal strategy (except for rare end-game cases). Below we present a Tromp-Taylor variant of Go rules. This variant is widely used in computer Go tournaments thanks to its simplicity and mathematical preciseness.

2.2.1 Goban



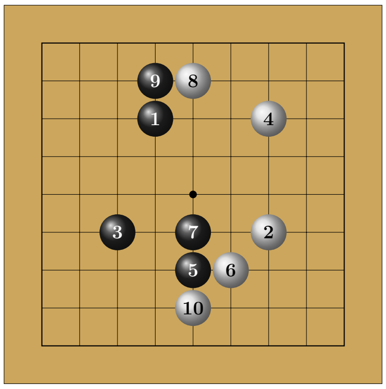
(a) 19×19

(b) 13×13

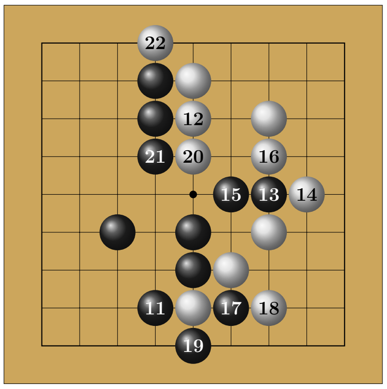
(c) 9×9

Figure 2.1: Empty gobans in the three most popular sizes.

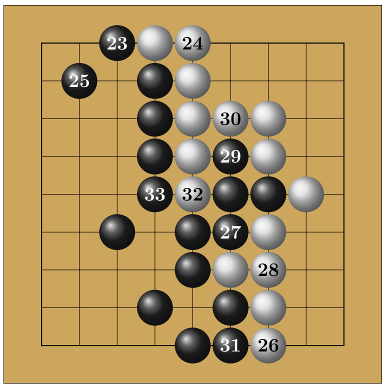
Go is played on a *goban* – a wooden board with a square grid painted on it. The board comes in three sizes as shown in Figure 2.1. The biggest, 19×19 board is the most popular one and by most players considered the only serious one. 9×9 board is very popular in the computer-Go research since computer engines seems to be much stronger on smaller sizes when compared with human players.



(a) Moves 1 – 10.



(b) Moves 11 – 22.



(c) Moves 23 – 33.

Figure 2.2: An exemplary game on 9 × 9 board. Move 19 captures one white stone.

2.2.2 The game

Two players, Black and White, alternately make a move or choose to pass (refuse to make a move). Black makes the first move. Player's move consists of putting a stone of his color on one of the unoccupied intersections on the board. The goal of each player is to surround more territory than the opponent. A simple example is shown in Figure 2.2.

It is possible for one player to play inside of the opponent's territory but such a stone will eventually be captured and the result of the game will remain unchanged.

2.2.3 Liberties

All intersections on board except edges have four *adjacent* intersections. Two stones are *connected* if they are placed on adjacent intersections. A *group* is a maximal set of connected stones. A group's *liberties* are all empty intersections adjacent to the group.

For instance, a single stone not touching other stones and lying on the edge of the board has 4 liberties - empty adjacent intersections. Stones on the edges, in the corners or

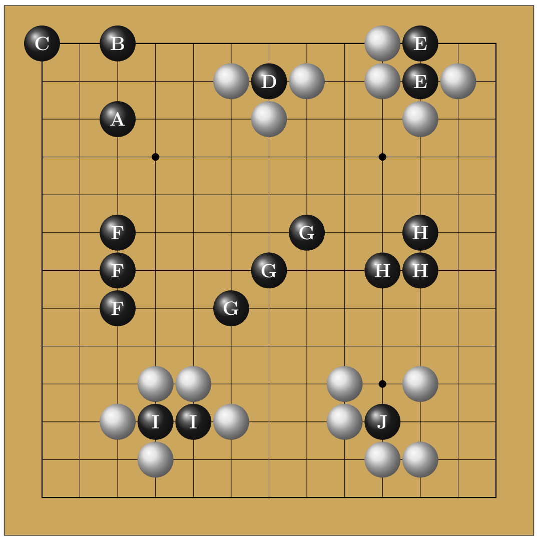


Figure 2.3: Groups and liberties.

touching opponent's stones have less liberties. The stones A, B, C, D and J in Figure 2.3 have 4, 3, 2, 1 and 2 liberties respectively.

Adjacent stones of the same color are connected and form a *group*. Stones in a group share liberties. In Figure 2.3, groups E, F, H and I have 1, 8, 7 and 1 liberties respectively. The stones marked with G are three separate groups.

A group having one liberty is said to be in *atari* because it is threatened to be captured.

2.2.4 Capturing

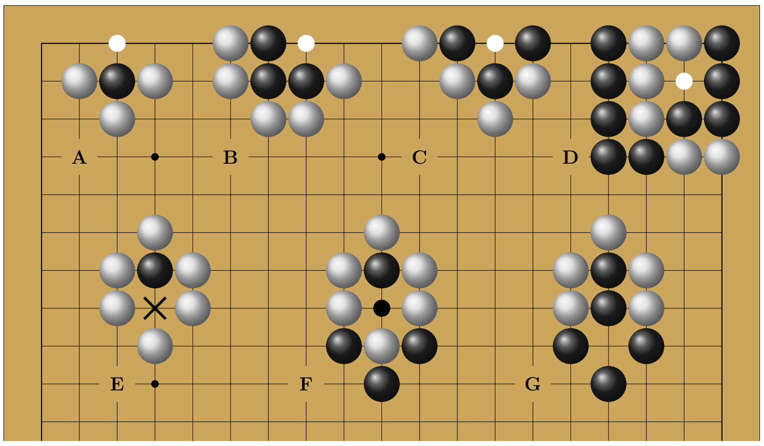


Figure 2.4: Situations where one or more Black's groups are in atari.

If a move takes away the last liberty of one or more of opponent's groups then these groups are captured and all of their stones are removed from the board. In Figure 2.4, in situations A, B, C and D, White can play a move at the intersection marked with the white dot. Such a move captures one of the Black's groups (two groups in case C). In situation D, Black could also play at the marked spot and capture the White's group.

After possibly capturing some opponent groups and connecting to some friendly groups, the new group needs to have at least one liberty to avoid self-capture. Self-capture is called *suicide* and is forbidden, i.e. such a move is illegal. Situation E is an example of this. Black cannot play at the spot marked with a cross due to lack of liberties.

Of course it is possible to play a move that looks like suicide if it results in capturing an opponent's group and effectively providing new liberties to the group that would otherwise be suicide. Situations F and G show an example of this before and after such a move done by Black.

2.2.5 Superko rule

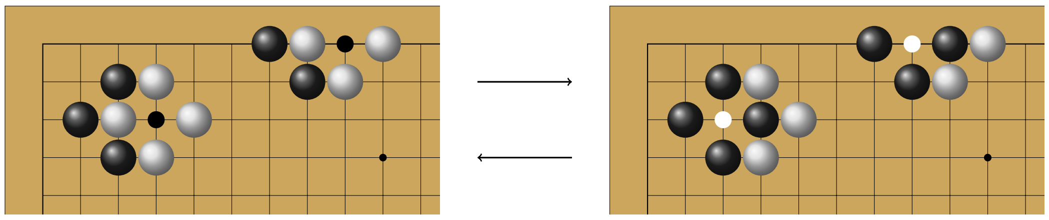


Figure 2.5: Ko - capturing a stone in one diagram leads to the other one. It is illegal to recapture the stone immediately as that would lead to a repetition of the position on the board and possibly to an infinite game.

A move that repeats a situation on the board is illegal. A situation where at least one move is forbidden due to this rule is called superko. The most common superko situation occurring in practice is called "ko" and it is shown in Figure 2.5.

In case of computer programs it is necessary to implement full superko detection as it may happen quite often when two weak algorithms play against each other.

2.2.6 Scoring

Game ends when both players pass. At this point players agree which stones cannot avoid capturing. Such stones are called dead and are removed from the board. In the Tromp-Taylor rules there is no end-game protocol and dead stones have to be explicitly captured and removed from board.

The score of each player is the number of his stones on the board plus the number of empty intersections surrounded by his stones. White receives *komi* – a compensation for Black making the first move. Usually $7\frac{1}{2}$ points are awarded to White. This method is called the area scoring. It is used in China and in the most computer Go competitions. In

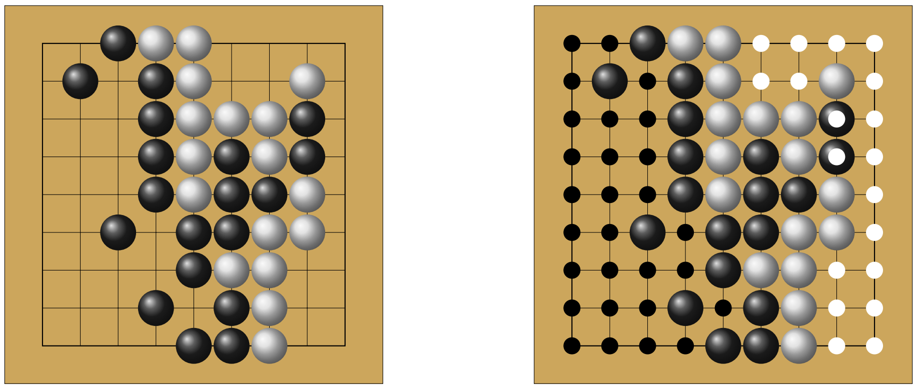


Figure 2.6: A finished game with the marked territory.

the example in Figure 2.6, Black has 28 territory points and 17 stones on board - together 45 points. White has 19 territory points and 17 stones on the board. Together with komi this yields $43\frac{1}{2}$ points for White. Black has won the game by $1\frac{1}{2}$ points.

An alternative to area scoring is territory scoring. Player's score is the number of empty points surrounded by his stones plus the number of prisoners (stones captured during the game and the dead stones). This method is used in Japan, Korea and Europe. In the most situations both methods give scores different by no more than one point. Choice of the particular rule variant usually does not affect the optimal strategy.

2.3 Elementary strategy and tactics

In this section we present basic ideas behind the strategy in the game of Go. It presents only a small fraction of the complexities arising in various Go positions. Classical Go programs try to mirror these and other ideas. This results in quite complex programs with many specialized algorithms. A case study of a classical Go program is given in section 3.1.3. On the other hand, Monte-Carlo programs are able to capture many of Go complexities in one simple algorithm.

Different situations may require different Go knowledge and different kind of tactical moves but they all share one common feature - they are local. Locality of tactical problems is something that currently the best Monte-Carlo algorithms cannot take advantage of. This limits their potential during the middle-game and is the cause of big mistakes in the end-game as there are many stable and independent sub-games.

In the next section we cover strategy of territory surrounding - the main goal of Go game. Later we cover a few of many important tactical situations that may occur in Go positions.

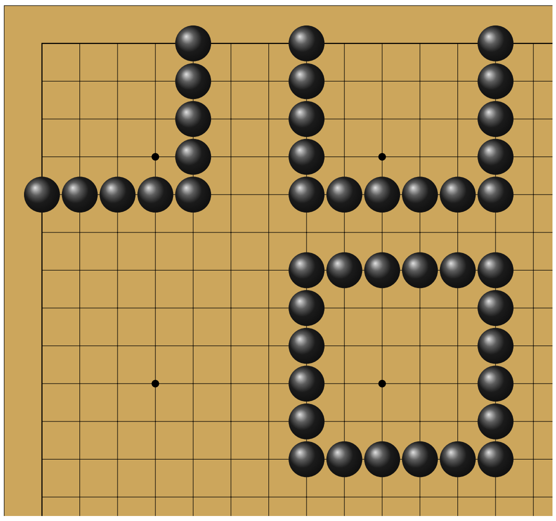


Figure 2.7: Less stones are needed to surround the same amount of territory in the corner and on the edge than in the center.

2.3.1 Territory

The goal of the game is to surround as much territory as possible with a limited (by the game length) number of moves. A good player tries to have all his moves as effective as possible. It is easier to surround the corners than the edges. The center is the most difficult part of the board to surround. It is enough to surround the corner from two sides. In order to surround the same amount of territory on the edge one needs to surround it from three sides. The center needs even more stones. Figure 2.7 illustrates this concept.

Obviously one does not place the stones next to each other during the opening as it is ineffective. Each player usually starts the game by placing his first two stones in the corners. When all the corners are taken, players attempt to extend their influence zones towards the edges. Figure 2.8 shows an example of an opening.

Area that is surrounded by the stones of one player does not have to become his territory. The other player can still try to invade it. It is a good strategy to place the stones in a way that it is not advantageous for the opponent to make an invasion.

Most of the moves in the opening phase are played on the third or the fourth line, counting from the edge of the board. Stones played on the first and the second line are not as effective in surrounding territory. The fifth and higher lines are too far away from the edge and make it easy for the opponent to jump in below (invade) and destroy territory one is trying to surround.

2.3.2 Life and death

Life and death is a situation that occurs often during Go games. It can arise when one player invades deeply into an area where the other player has many stones. In such a

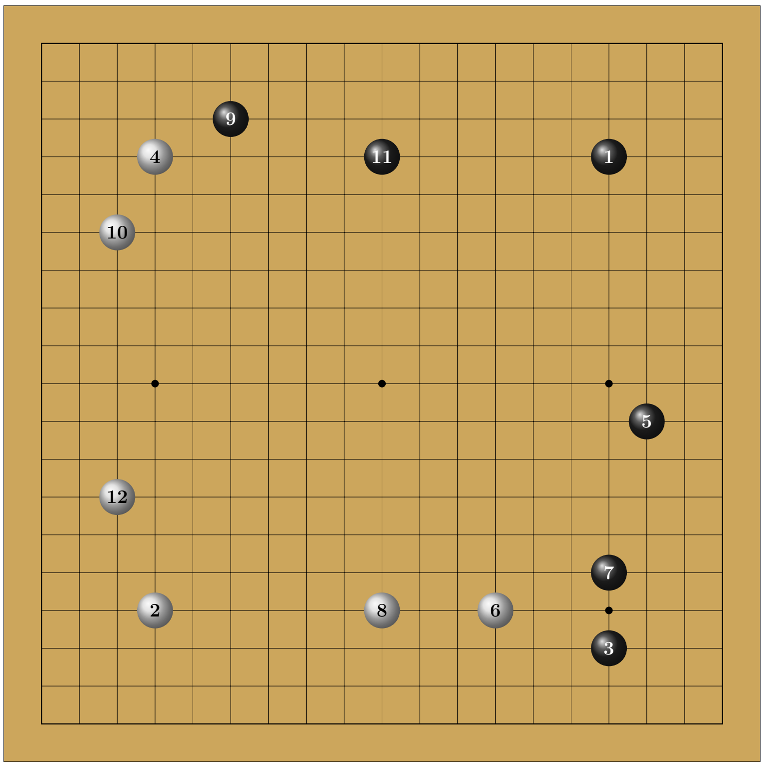


Figure 2.8: A peaceful Go opening.

situation the invader tries to make his new group *alive* while the opponent tries to *kill* it. Killing a group means that it cannot avoid being captured later in the game.

The understanding of the life and death concept is crucial since each player tries to surround as much territory as possible with a fixed number of stones. But if he becomes too greedy, his 'territory' will be too loose and an invasion by the opponent will be possible and will lead to a change in the ownership of this part of the board.

Classical Go programs and specialized algorithms are quite good at determining the life and death status of a group if the life and death area is well defined and not too big. Existing Monte Carlo algorithms are not yet able to deal with life and death situations effectively. They are also not able to exploit the fact that the life and death problem is usually independent from the rest of the board.

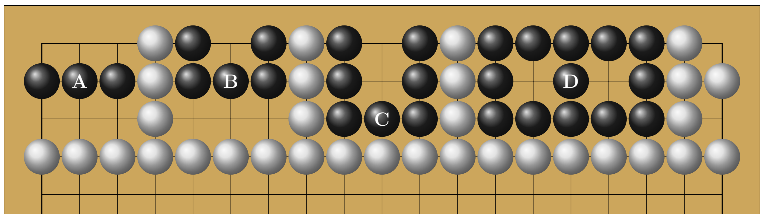


Figure 2.9: Groups A, B and C are dead. Group D is unconditionally alive.

A few basic situations of life and death situations are shown in Figure 2.9. The most elementary example of a dead but not captured group is group A. It has 6 liberties but any stone played in the neighborhood can only reduce the number of liberties. Capturing of such a group is imminent so there is no point in spending additional moves in order to kill it. In games between people it is customary to remove dead groups after the game is completed without an explicit capture.

Group B surrounds a single point of territory, the so-called *eye*. White cannot play a stone there as that would be a suicide. However, if he first fills out every other liberty of this group then playing in the eye takes away the last liberty and captures the group. There is no way that Black could interfere with White capturing his group, so group B is dead. Similarly, group A will be captured at the end of the game.

Group C has only one eye made of two neighboring intersections. It is still dead because White can place a stone inside the eye. Irrespective of Black capturing this stone or not, White can kill this group with his next move. Life and death status of groups such as groups A, B, C, is independent from the number of external liberties as long as Black cannot make a second eye.

Group D has two separate eyes. White obviously cannot place stones in both eyes at the same time and so the group is unconditionally alive. The surrounding white group is unconditionally alive too because it has two eyes at the right side of the board. Figure 2.10

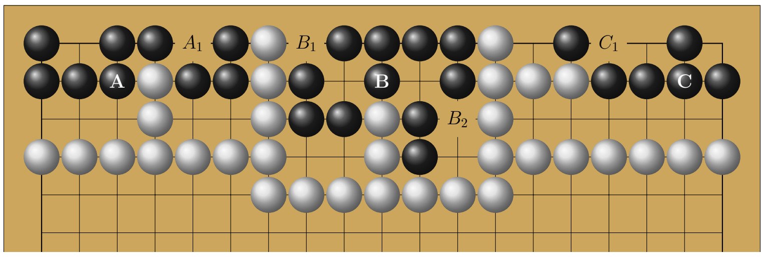


Figure 2.10: Examples of life and death problems.

shows more complex but still elementary examples of life and death situations. It illustrates concepts of *false eye* and *throw-in*. A false eye is an intersection that is surrounded by stones of one player but later in the game it will have to be filled due to atari of one of the surrounding groups.

For instance, group A is dead. Intersection A_1 will have to be filled by Black when white fills two external liberties of the three black stones putting them in atari. This will leave black group with one eye. Since there is nothing Black can do about it, the group is dead.

Group B in Figure 2.10 has two eyes. White has a way to make any of them false by playing at B_1 and B_2 respectively. If Black tries to secure the first eye by connecting at B_1, White will falsify the other eye at B_2. Similarly, if Black plays at B_2, White answers at B_1. Since there is no way Black can live, the group is dead and there is no hurry in capturing it.

Group C is dead or alive depending on the first player to play. If Black connects at C_1, it clearly has two eyes. If White plays at C_1 and Black captures it, then what remains is a false eye and Black is dead. Leaving White's stone at C_1 not captured does not help either. Thus, White's C_1 is a killing move.

2.3.3 Semeai - the capturing race

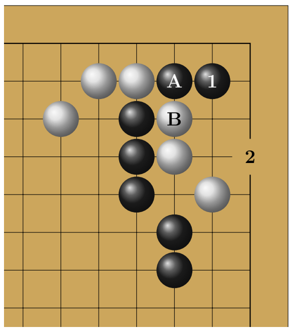


Figure 2.11: Example of capturing race.

Figure 2.11 shows an example of semeai or capturing race. Such a situation is as common and as important as life and death situations. Black just played move marked with 1. Neither of groups A or B have enough space to make two eyes and live. One of the groups have to die and be captured what guarantees the survival of the other one. Both players try to capture opponent's group as quickly as possible.

Semeai depicted in Figure 2.11 is a difficult problem. The only correct response to Black's 1 is White's play at the intersection marked with 2. Monte Carlo Go programs have big problems with handling of semeai situations for the same reasons they have problems with life and death problems. These are local problems while MC Go is a global algorithm.

2.3.4 Other tactics

There are many other local tactical situations in Go games:

- Capturing and escaping a group of stones.
- Cutting and connecting.
- Kikashi - a short forcing sequences.
- End-game situations.

These situations are very different in detail but all are local on board. Human Go players are able to analyze each "sub-game" separately and combine their findings to choose a globally good move. This divide-and-conquer approach guided the design of the "classical" Go programs. None of the modern Monte-Carlo programs are yet able to solve sub-games separately and combine the results. Ideas developed in this thesis in Chapter 5 help to close this gap.

2.3.5 Shape

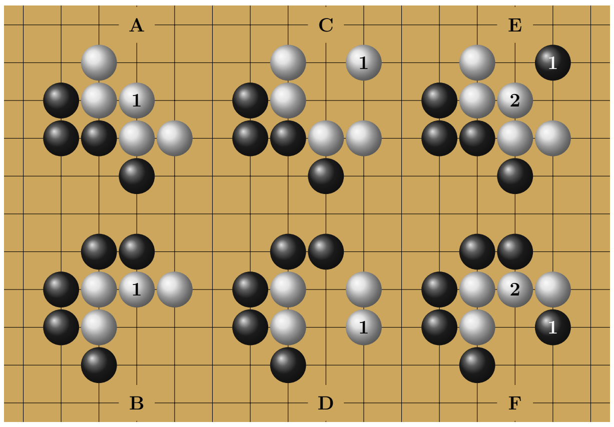


Figure 2.12: Situations A and B are examples of connecting stones in a bad shape. Good shape plays are shown in situations C and D. Situations E and F are examples of Black player forcing White to connect in a bad shape. Black moves marked with 1 are good as they force White into a bad shape.

The shape describes the effectiveness of a local group of stones. Figure 2.12 shows several examples of good and bad shapes. It is difficult to differentiate good shape moves from bad shape moves without playing experience. This is because there may be a large gap (counted in the number of game moves) between the moment of playing the move and moments where consequences of this move can be seen. For this reason the idea of playing good shape is difficult for beginner Go players to grasp. For the same reason it is difficult for tree search algorithms to find good shape moves.

In Go programs the notion of shape is encoded in valuated patterns. In classical Go programs the values of patterns are usually hard-coded by a human Go expert. In modern Monte-Carlo Go programs values of patterns are often automatically inferred from the game records of good Go players using Machine Learning algorithms.

The use of patterns is one of the most important components of Go programs, both classical and Monte Carlo. Introduction of patterns in move-choosing procedures in classical Go programs and later use of patterns in Monte Carlo playouts significantly improved playing strength.

Chapter 3

State of the Art

The history of computer Go research can be quite cleanly divided into two periods – the classical computer Go research and the Monte Carlo Go research. The idea of Monte Carlo Go, using long game simulations to evaluate a position, showed a lot of promise. Given its simplicity and the stagnation in classical computer Go research, Monte Carlo Go quickly became popular research topic and standard approach to Computer Go.

In this chapter we describe the history of Computer Go research and the current state of the art. We summarize achievements of classical computer Go using Gnu Go program as a case study. A more in depth study of state of the art classical computer Go can be found in [BC01]. Then, we describe a basic Monte Carlo Go algorithm followed by first enhancements such as All-Moves-As-First (AMAF) heuristic and Progressive Pruning algorithm. We describe the problems that face a tree search algorithm suitable for use with Monte Carlo evaluation and Monte Carlo Tree Search (MCTS) algorithm that overcame them. We conclude the chapter with a description of how the Go domain knowledge can be used within a Monte Carlo Go framework.

3.1 Classical approach in computer Go

3.1.1 Failure of the standard approach

The classical "brute force" approach of a global search combined with an evaluation function is widely used in other games but it fails in case of Go.

The problem with global search is the size of the game-tree. There are on average over 200 legal moves for each player yielding a game tree of size $O(200^D)$, where D is the depth of the search.

The evaluation function is an even harder problem. In chess, basic but strong evaluation function can be created just by summing values of the pieces on the board of each player. It can be further improved with features like mobility, evaluation of pawn structure, king safety, etc. In Go, the most natural evaluation function would be an estimate of safe territory. But even this task alone is hard enough to be researched separately [Mue97, Bou03, vdWvdHU04, Niu04, NM06, SGHM07].

3.1.2 History of classical computer Go

The first computer Go program was created around 1970 by Zobrist [Zob71]. It was based mainly on a computation of an influence “radiated” by the stones. It was able to beat a human (an absolute beginner) for the first time. At the time, all the programs were based on different variations of the influence function. The most popular variant of influence estimation is attributed to Bouzy [Bou03]. He applied operators borrowed from mathematical morphology – dilation and erosion – to a Go board. His algorithm is known under the name of 5/21 thanks to the respective numbers of dilations and erosions used.

The next important step was possible thanks to the analysis of human perception of Go [RKN$^+$75, Bru75]. Programs of that time started to use an abstract representation of the Go board and were able to reason about strength of groups of stones.

Probably the most significant breakthrough happened around 1990 when patterns were widely adopted to recognize typical situations and to suggest moves. Since then probably all the top classical programs combined those three and other techniques but little is known about the details of algorithms used because most of the best Go software is proprietary and commercial. The best classical Go programs are very complex. Most of them were developed by a single Go expert (and programmer) over a period of 5-15 years [Boo91, Fot92, Fot93]. A notable exception is Gnu Go [BGB05] - a program from the Free Software Foundation developed by a community of programmers.

3.1.3 The case of Gnu Go

Gnu Go is an open source Go program which has been actively developed for at least 10 years. The first version was published in March 1989. Strength of play of recent Gnu Go 3.8 on 19×19 board with human players is around 5 kyu on KGS server which is the rank of an average club player.

Gnu Go 3.8 is a huge program. It consists of more than 80000 lines of C code. A major part of the code is specialized, heuristic reading (game-tree searching) of life and death situations. There are another 60000 lines of human entered patterns that are the core part of the playing logic covering thousands of Go situations and shapes.

The process of selecting a move in Gnu Go tries to imitate human way of thinking. It consists of three phases: information gathering, move generation and move evaluation. All phases are very briefly described below to illustrate the complexity of the task and the amount of domain-dependant knowledge that had to be encoded in the form of patterns and algorithms. More details can be found in [BGB05].

3.1.4 Position analysis

The first phase gathers information about the position. First it finds a tactical state of all the chains (chain is Gnu Go terminology for a group of connected stones) on board. A specialized tactical search code is called for each chain. It searches until either the chain is captured or the chain gains 5 or more liberties.

Next, the influence function is called for the first time. It takes into account chains that cannot avoid capturing and finds areas of the board controlled by each player. It also helps to do the next step which finds groups (Gnu Go terminology) - sets of chains that are likely to end up as a single chain at the end of the game. Groups are found based on specialized patterns and a specialized minimax search algorithm that tries to ensure that connection can be maintained. Strength of each connection is saved for future use.

When all groups are found then their strength is estimated by crude estimation of number of eyes based on a yet another set of patterns and hard-coded heuristics. Each weak group also has its escape potential estimated and if it does not have any, then a life and death code is executed. This is one of the most complex parts of Gnu Go as it has to deal with open search situations. If two groups of opposite colors are found to be weak and next to each other, then yet another specialized search analyzing the capturing race is executed.

At this point the influence function is called again. This time it takes into account the whole accumulated information about the groups. Second influence call provides accurate information about territory and potential territory. This estimates are further improved with specialized break-in code that checks whether any invasions or reductions are possible. This concludes the first phase.

3.1.5 Move generation

The second phase generates move candidates. A list of moves is created by several different methods. Some of them were already executed during the previous phase. For instance, the candidate list already has chain-capturing and chain-defending moves and threats with capture/defend continuations. Analogous moves are stored for whole groups. This includes kill and life-making moves. Capture-race and break-in modules store their candidates as well.

During the opening phase a specialized code suggests moves based on the opening theory that can be found in Go books. Probably the most important is a pattern-based move generator. It works by matching the board position with human-generated pattern database. At the end a combination module is called. It searches for moves that serve multiple purposes at the same time.

If there is no move with a value greater than 6 points found, then specialized set of endgame patterns is used to accurately find endgame moves. Also if a program thinks it is winning the game, then safe-win moves are generated that may be suboptimal but ensure the tactical status of the groups. At the end of the game, neutral points are carefully generated by another specialized module.

3.1.6 Move evaluation

The third and final phase is a move evaluation. In the earlier versions of Gnu Go, the move values were estimated by move generators themselves. But as Gnu Go become more complex it turned out that a single algorithm is needed to evaluate all the moves at once. This was needed to ensure a fair comparison between moves produced by different move generators.

The primary value of a move is its territorial effect directly contributing to the final result of the game. Other factors evaluated and added to the move score are: growth of potential territory, change in strengths of groups, group connection value, changes in tactical statuses of chains and groups. There are a lot of heuristics implemented to avoid duplication in the scoring of the move.

3.1.7 Summary

Gnu Go is a very complex program with thousands of lines of heuristic knowledge. It is more and more difficult to improve its playing strength. Until the rediscovery of Monte Carlo Go, the research on Computer Go stalled.

3.2 Rediscovery of Monte Carlo Go

Discovery of Monte Carlo Go provided an algorithm that was easy to define and implement. It also gave hope for relatively easy improvements. Within a few years of its "rediscovery" 9×9 Go was dominated by Monte-Carlo programs. 19×19 Go followed shortly after. Recently other games such as Havannah, Lines of Action, Amazons, Hex and others become dominated by Monte-Carlo algorithms. Thanks to the simplicity and success of the Monte Carlo approach, it became popular in some non-game optimization domains.

In the next section we describe the early history of Monte Carlo Go. In the following sections we describe the first algorithms and improvements developed for computer Go. We give a lot of attention to the tree search algorithms compatible with Monte Carlo Go. The final search algorithm – Monte Carlo Tree Search proved to be very effective in many domains. We conclude the chapter with a description of domain heuristics specific for computer Go.

3.2.1 The First Monte-Carlo Go program

Monte-Carlo Go approach was first proposed by Bruegmann [Bru93]. He observed that in a given Go position some moves are good no matter when they are played as long as a player gets to play them. To find such moves he used a large number of *playouts* - randomized simulations of possible game continuations. Each playout begins at the same position and continues until there are no more legal moves. For a mid-playout move-selecting policy, Bruegmann used simulated annealing to find the best order of moves. Later it turned out that choosing the moves randomly with an uniform probability from the set of all legal moves is equally good and simpler. When the playout is finished Go rules determine the playout result.[1]

In order to evaluate a move Bruegmann used the average result of all the relevant playouts. Playout is relevant if the given move was played in it. Note that the given move does not have to be the first move, it can be played at any point of the playout. This definition of relevancy is important as the number of playouts according to the latter criterion is much larger.

Playing according to this simplest evaluation yields a surprisingly good Go-playing algorithm even though it almost does not have any Go knowledge except Go rules.

Bruegmann executed his experiments on a slow by today's standards 286 CPU. The limiting factor of his program was the number of the playouts per move he could afford. State of the art Monte Carlo algorithms described in the following sections scale very well. The number of playouts per move is still a very important factor of algorithm's strength. The next section describe the relationship between the number of playouts and algorithm's accuracy.

3.2.2 Statistical quality of averaging

By the law of large numbers, the standard deviation of the average of N independent samples, with the same mean and the standard deviation σ, is $\sigma/\sqrt{N}$. This means that in order to increase the accuracy of the statistics by two, one needs four times as many samples.

In each playout all empty intersections except eyes are played by one of the players. Each player gets to play around half of the legal moves in the starting position. So every statistic tracking a legal move on a given intersection is updated by approximately half of the playouts (given that the intersection is not an eye).

A standard deviation of a playout's result is around 45 points on 9×9 board. This is a very large number considering that the results fall into an interval $[-81, 81]$. This shows how little the starting position influences the result of the playout. Yet, with 2000 playouts the standard deviation of the average is $45/\sqrt{2000}$ which is around 1 point. This can be enough to differentiate most of bad moves. Figure 3.4 on page 41 is a good illustration of the huge variance of a light[2] playout.

[1]During the playout, moves that fill an eye of player's own group are not legal. This is necessary to ensure that at least some of the groups of stones that are provably uncapturable, remain uncaptured. This allows the playout to be finished and scored when there are no more legal moves.

[2]In a *light* playout, every legal move is played with the same probability

3.2.3 Measurement of strength

It was common in the early days of Monte-Carlo Go to measure the playing strength of a Go algorithm with an average score in points in games against a fixed opponent (usually Gnu Go). This measure later was replaced by a *winning rate* (the percentage of won games) against a fixed opponent. An average point difference is more sensitive than the winning rate because the latter returns only one bit of information per game. The winning rate became a dominating strength measure since programs learned how to maximize probability of win by sacrificing score advantage. Predicting the score accurately is difficult for non-professional human players because it is not easy to count all the points on 19×19 board without a mistake. Monte-Carlo programs are sacrificing possible additional points to maximize the probability of having more points than the opponent. This often leads to half-point wins, which Monte-Carlo programs are known for.

3.2.4 All moves as first heuristic

Bruegmann's work had remained mostly unnoticed for ten years until Cazenave and then Bouzy and Helmsetter in 2003 repeated and extended Bruegmann's experiments on a much more powerful hardware [BH03]. They decomposed Bruegmann's approach into several independent techniques. Each of them was separately evaluated for contribution to the playing strength.

Bruegmann evaluated each move by the average result of the playouts in which the move was played:

$$E\left(\text{result} \mid m \text{ played at some point}\right)$$

Bouzy evaluated a move by the average result of the playouts in which the move was played as *the first move*:

$$E\left(\text{result} \mid m \text{ played first}\right)$$

We call the latter algorithm *Vanilla Monte-Carlo Go* or simply the vanilla algorithm. Bruegmann's approach of averaging results of the playouts where a given move occurred at any time during the playout was named All-Moves-As-First (AMAF) heuristic.

It is important to understand the relation between both methods of evaluation. With enough playouts the vanilla value of a move m converges to the expected score of the game if the first move is m and all the rest of the moves are random. AMAF statistic converges to the expected result of the game that have m played at any point and all the rest of the moves are random. The goal of calculation of these statistics is to select a move *now*. It is the *first* move of a game continuation. This intuition suggests that the vanilla evaluation is better suited for the purpose than the AMAF evaluation. Indeed, the experiments performed by Bouzy on 9×9 board show that the algorithm selecting moves with the use of the vanilla evaluation is 10-15 points better than the one with the AMAF evaluation.

The drawback of the vanilla version is that each playout can update only one statistic (an average of playouts' results) while AMAF can update as many statistics as there are moves in one playout[3]. The vanilla needs many more playouts and computing power than AMAF to converge to the same accuracy. For instance, if there are on average 300 legal moves, then in order to evaluate each one of them with the same accuracy, vanilla needs 150 times more playouts then AMAF, making it a much slower algorithm.

3.2.5 Progressive pruning

Progressive Pruning(PP) by Bouzy [BH03, Bou05] is designed to focus distribution of the playouts on most promising moves. The key observation is that many of the moves are explored much more than it is needed to show that they are inferior to the best move. With a limited number of playouts it is never possible to prove that one statistic converges to a value smaller than other one, but it is possible to state that it happens with a high probability.

A *confidence interval* of a statistic is centered at the sample mean and has length proportional to the sample standard deviation. It is usually denoted as:

$$\mu \pm c \cdot \sigma_{AVG}$$

or equivalently:

$$\mu \pm c \cdot \frac{\sigma}{\sqrt{N}}$$

The probability that the true mean is in this interval is very high for $c > 3$. Chebyshev's inequality is one of many theorems formalizing this fact. We say that a move with a higher mean $c-$dominates another one if their respective confidence intervals are disjoint. I.e. the chance that their relative order will change is small (c dependent).

The vanilla algorithm assigns an equal number of playouts to each move. Progressive Pruning prunes any move that is dominated by another move. A pruned move is not sampled any further, so the rest of the moves are sampled with a greater number of playouts. With more playouts the unpruned moves' statistics have greater accuracy and the chance of selecting the best move is higher. The drawback of PP is that there always is a chance of pruning the best move.

The number of playouts per move is limited so there is a tradeoff with setting the value of c: with a low value of c many moves are pruned quickly so there are more playouts for the rest of the moves, but there is also a higher chance of pruning the best move. The best value of c has to be found empirically. Bouzy's experiments suggested $c = 2$.

With Progressive Pruning in place Bouzy found it is interesting to test the strength of the algorithm with respect to the number of playouts per game move. Compared to 10k

[3]In case of two player games such as Go half of the moves in the playout are opponent's, so only half of the updates are useful.

ppm (10000 playouts per move), algorithm with 1k ppm lost on average -12.7 points while 100k ppm variant won by 3.2 points. Bouzy concluded that 10k is a good compromise.

3.2.6 Future research as predicted by Bouzy

Bouzy predicted three main directions of further research: tree search to cover the tactical weakness of Monte-Carlo Go, use of a domain dependant knowledge in playouts and an evaluation of subgoals on a 19×19 board to exploit the locality of Go. As it is shown in the further part of this thesis, he turned out to be quite right. Tree search and domain knowledge in playouts were the key techniques to elevate Monte-Carlo Go to the status of state of the art technique for Go playing algorithms. We have yet to see whether local subgoals will play an important role in the development of Monte-Carlo Go.

3.2.7 Goal-based search algorithm

The only research tackling sub-goal evaluation was done by Cazenave [CH05]. In this work Monte-Carlo was used not to evaluate moves directly but to evaluate some tactical goals appearing on the board such as: group capture, eye making, group connecting and life-and-death problem solving. Each goal was evaluated by the difference in average playout results where the goal was achieved and where it was not achieved. Then the goal with a highest evaluation was chosen. Finally, an in-game move was chosen by a goal-specialized algorithms used in a classical program Golois.

Results of such a combination program were good. In an experiment against the vanilla MC Go program, using 10000 playouts each, the combination program won by 52 points on average. To account for lower performance another experiment was made in which combination program used only 1000 playouts per game move. Nevertheless, it still won on average by 24 points. The last experiment was against the original classical program Golois where the goal was selected using hard-coded heuristic knowledge. Combination program won on average by 26 points.

3.3 Search for the right tree search algorithm

A simple vanilla Monte-Carlo Go described in the previous section (with or without PP) is a surprisingly strong Go playing algorithm. But still it is too weak to compete with classical programs or humans. MC, while being a versatile position evaluator, does not take tactical sequences into account. Its single biggest weakness is lack of this tactical strength. This is not surprising as MC chooses moves using statistics that assume a random continuation of the game without any tactics.

The classical approach to automated game playing is a minimax tree search algorithm using an evaluation function at leaves of the search tree. Search finds all relevant tactical move sequences, an evaluation function judges the possible outcomes and the search combines all the results according to the minimax principle to find an optimal strategy. This way all possible tactics up to a certain number of moves (search depth) are taken into account when choosing a move.

It seems natural to try MC evaluation with a minimax algorithm. This was partially done (depth fixed to 2) by Bouzy already in the article described in the previous section [BH03]. Later he further extended his algorithm to an arbitrary depth [Bou04], but the results were not satisfactory. Both approaches are described in section 3.3.1.

The most important breakthrough came with selective variable-depth algorithms, i.e. algorithms that selectively grow the game tree based on the playouts' results. Two similar approaches were developed concurrently: Backup Operators [Cou06] and Upper Confidence Bound Applied to Trees (UCT) [KS06]. We concentrate on the latter as it turned out to be a better algorithm.

UCT was applied to Go in MoGo program [GWMT06, GW06] with a great success. MoGo soon became the strongest existing Go program which dominated computer-Go for more than a year. UCT search became de facto standard approach in computer Go. After a while in case of Go, the UCT formula was replaced by formulas that are more complicated and fine tuned to this domain. But in other domains UCT proved to be a useful tool that often works "out-of-the-box". UCT is described in section 3.3.2.

Soon another important improvement was developed. Rapid Action Value Estimation [GS07] (RAVE) was a generalization of AMAF heuristic to Monte Carlo Tree Search. This technique proved to be both very effective and domain-independent. It is described in section 3.3.2.

Finally, in practice it is invaluable to introduce domain-dependant knowledge to the search algorithm. A simple ad-hoc technique called Progressive Bias [CWvdH$^+$07] (PB) proved to be very effective. The knowledge itself is usually represented as a fixed pattern database with assigned numerical values and features that modify move values. PB is described in section 3.4.

3.3.1 Bouzy's minimax tree search

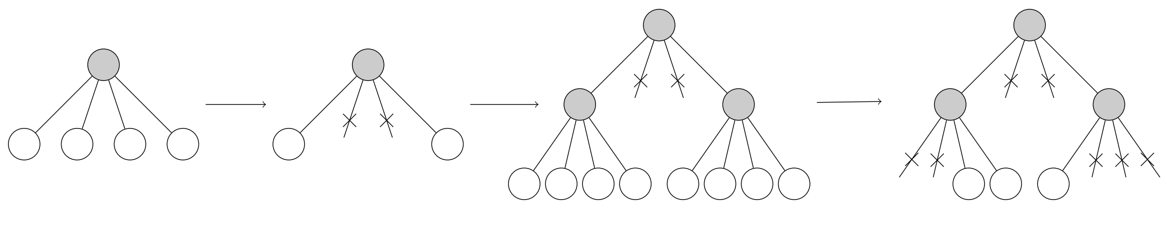


Figure 3.1: Depth-2 Progressive Pruning

In a game tree, the edges represent moves while the nodes represent positions, with the root representing the position in which the move decision has to be made. Depth-2 nodes (leaves of the tree) in each subtree rooted at depth-1 are evaluated using Progressive Pruning (with a modification for the minimizing player). At the depth-1 positions we need a move choice instead of a value. The most common solution for this problem was also adapted in Bouzy's experiment. It is assumed that the min player at depth-1 node chooses a depth-2 child that minimizes the evaluation. Therefore to each node at depth-1 algorithm assigns a value equal to the lowest Monte-Carlo mean of its children. The move chosen at depth-0 is a maximizer of the evaluation at depth-1.

The computational cost of depth-2 tree search is very high. Depth-2 tree has significantly more statistics to estimate than a simple Depth-1 tree (which is costly itself without PP). If n is the average number of legal moves, then depth-1 tree has n statistics and depth-2 has n^2 of them.

To take that into account in the experiment, program using depth-2 search received a budget of $n \cdot 10000$ playouts (10000 for each branch) while its opponent - simple Progressive Pruning had only 10000 playouts. Nevertheless, the results were surprisingly disappointing. Deeper search instead of being better was about 2 points worse on average.

The problem can be traced to the evaluation of depth-1 nodes as a minimum of the means. This evaluation is biased. To give a clear example let us assume that a depth-1 node have 10 children all with a true mean equal to zero and standard deviation equal to one. Minimizing player is indifferent which child to choose and we would expect the node to have value zero. While in fact, due to a random noise, about half of the nodes have positive value estimates and half of the nodes have negative value estimates. Choosing the minimum consequently underestimates the node's value. Or to put it simply, operation minimum on random variables amplifies the noise.

In a further research [Bou04] Bouzy refined his method, which allowed a deeper search (depth-3 and depth-4). Necessary selectivity was achieved through an iterative-deepening style algorithm. Firstly, depth-1 tree is explored, then few best moves are selected and expanded into depth-2. Then, after some random playouts, some of the depth-2 moves are pruned and the rest is expanded to depth-3 and so on. The scheme controlling the width of the tree and the speed of tree-growth is complicated. Also the moves for consideration in the tree were selected by a domain-dependent move generator, namely Indigo - a classical Go program.

The results of the self-play experiments in this setup were best for maximal depths 3 and 4. The optimal tree width was quite small. This can be explained by a good classical move generator. The results against Gnu Go were worse, but still the optimal tree shape parameters were the same. It is difficult to judge how would this search algorithm work without Indigo move generator.

3.3.2 Monte-Carlo Tree Search (MCTS)

A straightforward application of the Mini-Max algorithm with the Monte Carlo evaluation was not very successful. A discrepancy between the average operator used in Monte-Carlo evaluation and the maximum and minimum operators used in evaluation propagation proved to be the problem. The former operator cancels the noise while the latter amplifies it. In order to deal with it two new ideas were proposed.

Remi Coulom [Cou06] proposed a search tree that uses an operator dependent on the number of updates in a given node. Near the leaves, in the unexplored nodes, his search uses operators close to averaging. With a larger number of playouts the operator gradually changes to become minimum or maximum.

Approximately at the same time Kocsis et. al. [KS06] proposed UCT, an algorithm based on the averaging operator but with the allocation of the playouts arranged in such a way that the average converges to the maximum / minimum. The latter approach turned out to be superior to the former.

UCT was later modified many ways. Here we present UCT as a special case of the *Monte Carlo Tree Search (MCTS)* framework.

MDP terminology. MCTS is a relatively domain independent search algorithm. To retain generality of description we borrow the terminology from the Markov Decision Processes (MDP) [SB98].

- $\mathcal{S}$ is a set of states. Each MCTS tree node represents one state. In case of Go, states are Go positions. In practical applications like Go, state space is usually huge.

- $\mathcal{A}$ is a set of actions. In case of Go, actions are all legal moves. In MCTS tree, actions correspond to edges.

- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a deterministic transition function. $s' = T(s, a) = s + a$ is a state obtained by executing an action a in a state s. In general, MDP's transition function may be non-deterministic: $T : \mathcal{S} \times \mathcal{A} \to P(\mathcal{S})$. In case of Go, T represents rules of the game.

- $R : \mathcal{S} \to \mathbb{R}$ is the reward function. $R(s)$ is the reward for visiting the state s. In case of Go, $R(s)$ is non-zero only for final states where the game ends. In general, MDP's reward function may depend on a whole transition: previous state, action and obtained state: $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$.

- $\pi : \mathcal{S} \to \mathcal{A}$ is a deterministic policy. It assigns one action to each state. Using a policy and a starting state s_0 one can generate a playout: $s_0, a_0, s_1, a_1, s_2, \ldots, a_{n-1}, s_n, r$ where $a_i = \pi(s_i)$, $s_{i+1} = T(s_i, a_i)$, s_n is a final state and $r = R(s_n)$ is the score of the playout. Policy may be non-deterministic. In such case it assigns a probability distribution over actions to each state.

$$\begin{tikzcd}
s_0 \arrow[r, "a_0"] & s_1 \arrow[r, "a_1"] & s_2 \arrow[r, "a_2"] & s_3 \arrow[r, "a_3"] & \cdots \arrow[r, "a_{n-1}"] & s_n \arrow[r, dotted] & \boxed{R}
\end{tikzcd}$$

Figure 3.2: Example of playout with MDP notation: states, actions and reward.

MCTS tree statistics. Each playout starts from the root state and iteratively chooses next action according to a *tree policy* (defined below). When it leaves the tree it chooses the next action according to a different policy usually called a *playout policy*.

When a final state is reached, it is scored and the score is used to update all of the tree nodes the playout visited. Each tree node (state) tracks a number of playouts that visited it and an average of all their scores:

$$n : \mathcal{S} \to \mathbb{N} \quad \mu : \mathcal{S} \to \mathbb{R}$$

We use the notation:

- n_s - number of visits in node s.
- μ_s - average of all playouts' results that visited node s.

When a new result R needs to be averaged the following equations may be used to update the statistics:

$$\mu_s \leftarrow \frac{\mu_s \cdot n_s + R}{n_s + 1}, \quad n_s \leftarrow n_s + 1 \tag{3.1}$$

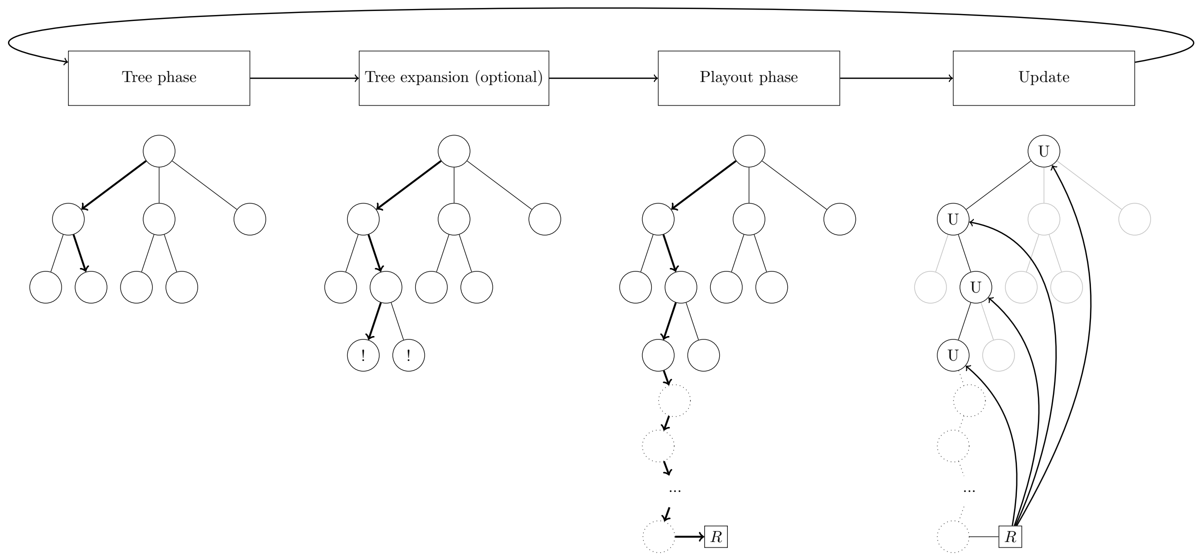


Figure 3.3: Four stages of every playout.

MCTS in an anytime algorithm. MCTS does not need to expand the tree using batches of playouts like Bouzy's algorithm (or iterative deepening alpha-beta used in chess). MCTS iteratively runs playouts and updates the tree after each one of them. It is an *anytime* algorithm, which means that it can be stopped after any playout and it returns an action that at that point is considered the best.

Tree growth algorithm. It is not possible to store a whole tree in the memory, not even the part ever visited by a playout. Only the nodes close to the root are stored. MCTS grows the tree starting only with the root and iteratively expands the leaves. The expansion consists in attaching all legal actions and children states to the leaf. A leaf s is expanded as soon as n_s - the number of playouts that left the tree through it - reaches a certain threshold n_{expand}. If $n_{\text{expand}} = 1$ then with each time a playout is leaving the tree, the leaf is expanded. The whole process is illustrated in Figure 3.3.

Exploration - exploitation dilemma. There are two conflicting goals in a tree policy: exploration and exploitation. The exploration promotes actions leading to the states s visited small number n_s of times in a hope they will turn out to be the best choice in the end. The exploitation promotes actions leading to the states that have a good average result μ_s to improve this estimation and check whether there is a good continuation.

Without the exploration the states that accidentally got a very low μ_s from a few first playouts could be never visited again. Without the exploitation many playouts would be wasted on probably uninteresting states. The exploration-exploitation problem is well researched in a literature under the name of *a multi-armed bandit* [ACBF02] and has many practical applications.

Tree policy: UCT formula. One asymptotically optimal solution to multi-armed bandit problem is an UCB1 (Upper Confidence Bound algorithm 1) formula [ACBFI00]. It was applied to Monte Carlo Game Tree by Kocsis and Szepesvari [KS06] yielding algorithm called UCT (UCB1 applied to Trees). The resulting policy chooses an action that maximizes UCB:

$$\pi_{\text{UCT}}(s) = \arg\max_a \ \mu_{s+a} + c\sqrt{\frac{\log n_s}{n_{s+a}}}$$

The first term in the formula represents the exploitation of the best actions, while the second is promoting the exploration of the actions with a small n_{s+a}. The constant c controls the tradeoff between the exploration and the exploitation. If c is zero an action with the highest mean is always chosen. If c is big enough, action with the smallest number of visits is always chosen.

The correctness of UCT. For UCT formula to be correct it has to guarantee that, given enough time, it finds the best action. It is proved in [KS06] that UCT based MCTS will eventually convergence to the min-max tree. This is shown by induction on the depth of the tree. Sketch of the proof follows.

Thanks to the exploration term (and specifically to the always growing $\log n_s$) each child of a node is visited infinitely many times. So each child of such node converges to its mini-max value by induction hypothesis.

The evaluation of each non-leaf state μ_s is almost (due to incremental tree expansion) weighted average of evaluation of children (it converges to it):

$$\mu_s \approx \frac{\sum_a n_{s+a} \cdot \mu_{s+a}}{\sum_a n_{s+a}}$$

To guarantee that μ_s eventually converges to $\max_a \mu_{s+a}$ (assuming the player on move is MAX), the UCT formula must choose the best action a_0 many more times than other actions:

$$\frac{n_{s+a_0}}{\sum_{a \neq a_0} n_{s+a}} \xrightarrow{n_s \to \infty} 1$$

which is indeed the case - the exploration terms eventually gets much smaller than the difference between the average of the best and the second-best states' evaluations.

MCTS is efficient. MCTS works well in practice thanks to saving a lot of the computation time on actions that have a low average return so far. It concentrates playouts on the most probable variants so that forced variants (tree branches where all except one or two actions are very bad) are searched deeply and accurately. This property makes MCTS a variable depth algorithm. Efficiency and simplicity of MCTS popularized it in other abstract games. Many first places in Computer Olympiad 2010 were taken by Monte-Carlo programs: Go (of course), Hex, Havannah, Amazons and others. MCTS is starting to be used in more distant domains, for instance:

- Samegame [SWH+08] - MCTS was adapted to solve a popular puzzle. A simple algorithm achieved the world record on a standardized problem set.

- Settlers of Catan [SCS09] - One of the most popular family board games - a simple MCTS agent was shown to be much stronger than advanced JSettlers AI.

- General Game Playing (GGP) [FB08] - A yearly competition where rules of the game are given to the program in the form of logic formulas. GGP is dominated by MCTS based programs.

- Fast Fourier Transform (FFT) algorithm optimization [dMRVP09] - A modified MCTS is used to search the algorithm space and find the fastest variant of FFT algorithm for a given data.

Domain knowledge - Progressive Bias. Large non-trivial domains like the game of Go cannot be solved by a brute-force search through the state-space because it is too big. Two most important heuristic improvements to MCTS are RAVE heuristics described later and an introduction of domain-dependent knowledge into the search. In contrast with RAVE averaging described later, there is no research on a principled knowledge introduction known to the author. Knowledge can be introduced in many ad hoc ways. The best way is usually determined by experiments. The most popular one - Progressive Bias (PB) [CWvdH+07] modifies the UCT formula with an additional knowledge term $K(s, a)$:

$$\pi_{\text{UCT+PB}}(s) = \arg\max_a \ \mu_{s+a} + c_1 \sqrt{\frac{\log n_s}{n_{s+a}}} + c_2 \frac{K(s, a)}{n_{s+a}}$$

$K(s, a)$ is a numerical value representing how good action a in state s is expected to be. The most popular implementation of K is the probability of choosing given action according to the *heavy* (heuristic, not uniform) playout policy:

$$K(s, a) = P(\pi_{\text{heavy}}(s) = a)$$

Progressive Bias is divided by n_{s+a} to constrain the influence of the heuristic knowledge to weakly explored actions. When n_{s+a} becomes large the evaluation from the playouts should and will dominate as it is much more accurate. This also allows MCTS to still converge to the minimax tree. Also note that if all means μ_{s+a} are equal and $c_1 = 0$, then the knowledge term dominates the tree policy. It chooses the actions so that the counts n_{s+a} are proportional to $K(s, a)$.

Progressive Bias helps to reduce computation by many orders of magnitude thanks to many simple heuristic rules. For instance in case of Go, a rule "search the moves near the last move" is very effective. It strongly reduces the effective branching factor of the tree by focusing the playouts on branches that are likely to be important.

RAVE statistic. RAVE [GS07] is a domain-independent and very effective heuristic. It works based on the fact that the order of actions in a playout could be usually changed and it would still yield the same result.

In order to explaining RAVE heuristic let us assume that a playout goes through a sequence of states and actions:

$$s_0, a_0, s_1, a_1, s_2, a_2, s_3, a_3, s_4, a_4, \ldots, R$$

Of course after it is finished and scored all MCTS statistics along its path are updated:

$$\mu_{s_0}, n_{s_0}, \mu_{s_1}, n_{s_1}, \mu_{s_2}, n_{s_2}, \ldots$$

Let's permute the actions a_1 and a_3. We will probably obtain a playout with the same result:

$$s_0, a_0, s_1, a_3, ?, a_2, ?, a_1, s_4, a_4, \ldots, R$$

Please note that the state after a_1 is the same state as the one in not permuted playout - s_4. By Markov property, the remainder of the playout and the result is identical.

This almost-permutation-invariance suggests that one playout could be used to update many tree branches. Unfortunately, the total number of permutations is too big to make this efficient. RAVE algorithm updates only the statistics of the first different states on alternative action-permuted paths. More precisely, if MCTS statistics are updated on the states:

$$s_i + a_i$$

then RAVE statistics are updated in states:

$$s_i + a_k, \quad k \geqslant i$$

RAVE uses the separate set of statistics: $\mu_s^{\text{RAVE}}, n_s^{\text{RAVE}}$ with the same formulas (3.1).

It is clear that RAVE statistics are updated much more often than MCTS statistics and converge faster. But since they are slightly differently defined, they do not converge to the correct values. Ideally we would like to use RAVE as long as the MCTS is noisy (n_s is small) and switch to MCTS when it gets more accurate. This idea is implemented using weighted average:

$$\pi_{\text{UCT+RAVE+PB}}(s) = \arg\max_a \quad \left((1-\alpha)\mu_{s+a} + \alpha\mu_{s+a}^{\text{RAVE}}\right) + c_1\sqrt{\frac{\log n_s}{n_{s+a}}} + c_2\frac{K(s,a)}{n_{s+a}}$$

In the original RAVE paper [GS07], α was some arbitrary function of n_s empirically found to be working. Later David Silver found a better equation for α based on some probabilistic assumptions:

$$\alpha = \frac{n_s^{\text{RAVE}}}{n_s^{\text{RAVE}} + n_s + c_3 \cdot n_s \cdot n_s^{\text{RAVE}}}$$

where c_3 is a constant representing the bias of RAVE statistic.

RAVE algorithm is an example of a horizontal tree knowledge transfer. It is similar to the killer heuristics used in the $\alpha - \beta$ search algorithm.

3.4 Domain knowledge in playouts

The quality of any non-exhaustive game-tree search depends on the quality of the evaluation function. For instance, if the evaluation function is random, the tree-search of any depth returns random results. If the evaluation is perfect, then a depth one tree search returns a perfect answer.

We can say that in the Monte Carlo Tree Search the playouts are the evaluation function. The quality of the score returned by playouts exiting the tree is the key to the quality of the whole algorithm. The first Monte-Carlo Go programs used a playout policy

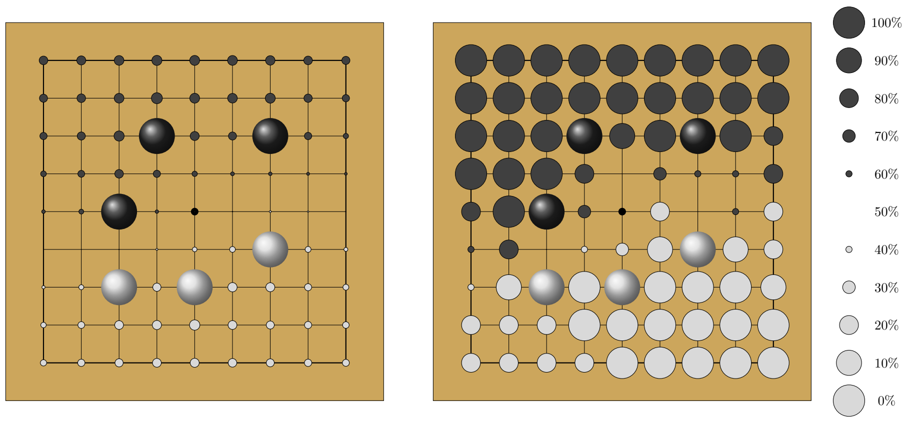


Figure 3.4: In the situation shown on the figures above, 10000 light playouts were performed and 10 self play games by Gnu Go. The figures show on each field of the board the percentage of 10000 playouts (on the left) and the percentage of 10 self play Gnu Go games (on the right) that finished with the given field owned by Black. This figures compare the power of light playout and full Gnu Go engine as an evaluation function. Note how weakly the played stones influence the light playout evaluation.

where the actions were chosen with a uniform probability distribution. Currently this approach is referred to as *light playouts*.

As we stated in section 3.2.2, the light playouts have a huge variance. Figure 3.4 shows how little from the starting position is retained in an average light playout. We can see that some of the areas of the board can be considered a "sure territory". In a real game they can be expected to be won by the dominating player in close to 100% of possible continuations. But in light playouts they are won in about 60% of samples, which is only slightly more than a coin toss. Such errors in judgement are consistent and direct the tree search into wrong branches, leaving the right ones unexplored. It seems that using smarter than random playouts can lead to a dramatic improvement in the evaluation quality and thus playing strength. Playouts with non-uniform probability distribution over their actions are called *heavy playouts*.

The MCTS can be seen as an algorithm that adaptively improves the policy in the prefix of the playout. It chooses better actions based on the results of the previous playouts. This suggests an idea to use MCTS to choose every single action of *meta-playout*. Such meta-playouts would be used with a global MCTS algorithm to find a good decision in the root. Unfortunately this approach would drastically reduce the number of available playouts from tens of thousands to few meta-playouts. It is not enough to explore the meta-tree properly. The nodes near the root will not be explored enough because time is spent on exploring nodes far from the root while choosing meta-playout actions. There is no known publication that shows meta-playouts to be beneficial in two-player game.

This example clearly shows the tradeoff between playout quality and the number of the playouts we can make in a given time. Statisticians call this tradeoff *bias-variance* tradeoff. Bias is a difference between expected result of the playout and the true position's value. The variance of the playout comes from the randomness of a single playout. The key difference is that variance can be averaged out by using more playouts and the bias cannot. In the perfect playout the bias is zero because it leads to a perfect score and the variance is also zero because it is deterministic. The bias of light playouts is very big as shown earlier. We search for improvements that reduce the bias but are efficient to compute so we can still make enough playouts to average out enough of the variance.

3.4.1 Domain knowledge in Go playouts

Most of the published methods for modifying playout policy rely on using hard-coded domain knowledge. In most programs this knowledge was entered manually (Many Faces of Go, MoGo, Fuego). In some of them this knowledge was automatically harvested from the games of strong human players (Crazy Stone, Libego, Erica). In both cases the knowledge is stored in two forms: patterns and features. The key difference between features and patterns is the way of implementation. The patterns are matched using hashing, while custom algorithms are used to find all the actions that have a certain feature. Every possible action on the board has exactly one pattern evaluating it and zero or more features.

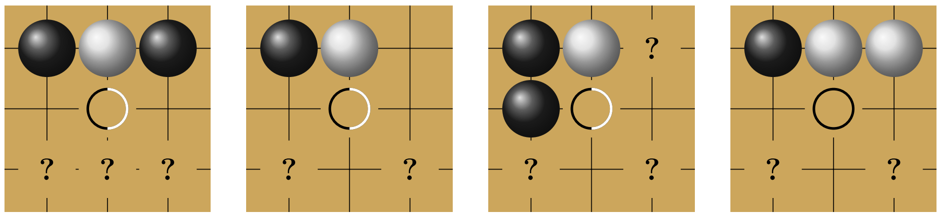


Figure 3.5: Basic patterns defining basic good shape moves for both Black and White. The question mark means that field can be either black, white or empty for a pattern to match. The color of the circle in the middle shows for which player a given pattern is good to be played in. All symmetric (rotation, mirror and color-inverse symmetries) patterns are considered good as well.

Heavy playouts usually use patterns sized 3 by 3 intersections. This size is a good compromise between the power to capture key Go ideas and the cost of the hash calculation. Figures 3.5, 3.6 and 3.7 show all the patterns used in MoGo program. These patterns were hand picked to maximize MoGo strength.

MoGo's playout policy [GWMT06] is:

1. If there is an opponent's group in atari, capture it.

2. If there is friendly group in atari, save it.

3. If there is at least one legal action next to the last played action that matches one of the patterns in Figures 3.5, 3.6 or 3.7, play a random one.

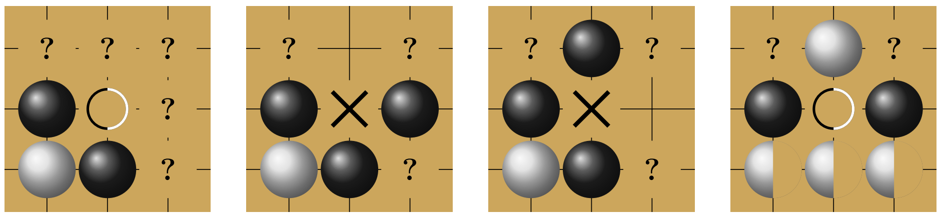


Figure 3.6: The first pattern means it is good to connect (for Black) and to cut (for White). The first pattern matches with the exception of situations shown on patterns 2 and 3, where cutting is a bad move as it can be immediately captured. The last pattern shows different kind of cutting/connecting. Half empty stone means that this location can be occupied by this stone or be empty for a pattern to match.

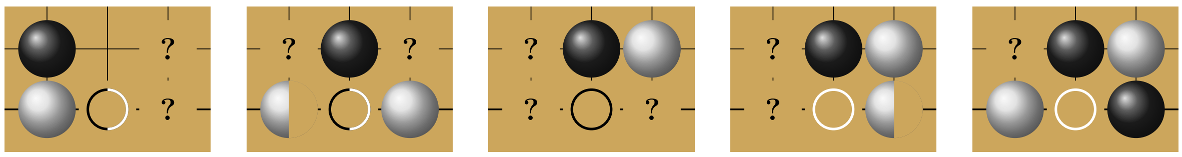


Figure 3.7: Pattern defining good on-edge moves.

4. Play randomly any legal action.

This algorithm was later extended, but it is a good illustration of use of patterns and features. The features used in MoGo are: "an action is capturing opponent's group", "an action is saving friendly group", "an action is played near the last played action". These are the most popular features and are used in all strong programs.

Crazy Stone and Libego use a richer playout policy representation. Each legal pattern p has assigned a positive number γ_p. Each feature f also has assigned a positive number γ_f. In a given state s (omitted in notation for simplicity), each action a has exactly one pattern $p(a)$ and a (possibly empty) set of features F_a. The action's "gamma" γ_a is a product of pattern's "gamma" at that intersection and all action's features' "gammas":

$$\gamma_a = \gamma_{p(a)} \prod_{f \in F_a} \gamma_f$$

The probability of heavy playout choosing action a in state s is proportional to "gamma" of the action a:

$$\pi_{\text{heavy}}(a|s) = \frac{\gamma_a}{\sum_{a \in \text{legalActions}(s)} \gamma_a}$$

Good gammas can be inferred from the games of strong players. If strong players play into a certain pattern or feature as soon as it appears, it should have a large gamma. Patterns and features rarely played should have gammas close to zero. One possible approach to find gammas from data is called *maximal likelihood*. It finds such gamma values that the probability of the data according to π_{heavy} policy is maximized. Implementation of *maximal likelihood* used in Crazy Stone program is called Minorization

Maximization [Cou07a]. It's main drawback is memory inefficiency. It needs a single computer with 24 GB of RAM memory to process 100000 games with only using a subset of $\frac{1}{11}$ game actions.

In Chapter 4 we describe in detail the Bayesian framework and describe a new algorithm that improves over Minorization Maximization. We also explain why maximum-likelihood is insufficient in some circumstances.

Chapter 4

Laplace$_q$ Marginal Propagation

4.1 Introduction

In this chapter we present *Laplace$_q$ Marginal Propagation (LMP)*, a new method for creating Bayesian inference algorithms. We use this method to derive a new algorithm for modeling preference data by inferring parameters of the Bradley–Terry (BT) model. This algorithm is used in the Libego system to infer gamma values (BT model parameters) for patterns as described in section 3.4.1.

LMP applied to Bradley–Terry model presents several improvements over Minorization-Maximization (MM) algorithm used for the same purpose in Crazy Stone [Cou07a]:

- MM is limited by the memory. A single computer with 24 GB of RAM memory is needed to process 100000 games with only a small fraction of $\frac{1}{11}$ of moves taken from each. LMP needs only around 15 MB for data of the same size.
- LMP can work on streaming data of potentially infinite size.
- LMP computes a full Bayesian distribution over gamma values instead of only the most probable gamma computed by MM.
- LMP is at least as simple in implementation as MM.
- LMP similarly to MM, contrary to SGD (Stochastic Gradient Descent), does not have any tunable parameters.
- LMP can function as an alternative to ELO, Glicko and Trueskill ranking algorithms.

The outline of the chapter is as follows. In the next section the idea of Bayesian modeling is explained. Then, we present the Bradley–Terry model that formalizes preference data such as action choices in Go games. Next, we present Maximal Likelihood (ML), an approach finding the most probable value of model parameters (gammas). We describe two algorithms for finding ML: Gradient Ascent and Minorization–Maximization. Then, we describe the shortcomings of ML when compared to a full Bayesian inference. Finally, we present a new Laplace$_q$ Marginal Propagation method and as an example its application to the Bradley–Terry model.

4.2 Bayesian modeling

The Cox theorem [Col04] shows that under very weak assumptions there is only one correct way of extending boolean logic to reasoning with uncertainty. The equations of that unique extension are identical to the ones used in the probability calculus. Within the context of reasoning, rules of probability are better known as *Bayesian modeling.*

The biggest difference between Bayesian modeling and the probability theory is the interpretation of probability distribution of a random variable. In case of probability theory, random variables can be sampled many times and probability distribution describes the *frequencies* of different outcomes. In case of Bayesian modeling, random variables represent unknown quantities and probability distributions represent *information* about their possible values. In Bayesian modeling, probability distribution is also called *belief.*

To illustrate the Bayesian inference, assume that a place on earth is chosen randomly. This could be modeled by a random variable "the position" valued with geodesic coordinates with uniform distribution over all points on the surface of the earth. Now you are given a photo taken in this place. On the photo there are several trees. It is possible to infer some facts almost certainly and some facts with a high probability. For instance it is almost certain that this place is not on the ocean and with a high probability it is not in the Sahara desert. In general this piece of information would *change your belief* (probability distribution) of "the position" random variable. This change of the belief bases on the model of the world. Given the prior belief (uniform in this case), the model (the knowledge about the location of trees on the earth) and the data (the photo), the posterior belief can be rigorously inferred using the *Bayes theorem.*

4.3 Bayes theorem

The Bayes theorem requires representing all unknown quantities as random variables. In our example they are valued in vector of real numbers $\vec{w} \in \mathbb{R}^n$, but in general Bayesian inference can deal with any objects that have a probability distribution defined on them. In case of continuous random variables, the belief is the probability density function (*pdf*).

$$P(\vec{w} \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \vec{w}) \cdot P(\vec{w})}{P(\mathcal{D})}$$

Figure 4.1: Bayes theorem.

$$f(\vec{w}) = \frac{g(\mathcal{D}, \vec{w}) \cdot h(\vec{w})}{\int g(\mathcal{D}, \vec{w}) \cdot h(\vec{w})\, d\vec{w}}$$

Figure 4.2: Bayes theorem in the form exposing argument $\vec{w}$ of all pdfs. Note that the denominator is just a constant that normalizes $f(\vec{w})$.

Bayes theorem allows to compute the *posterior* pdf, i.e. the belief of the model parameters (location on earth):

$$f(\vec{w}) = P(\vec{w} \mid \mathcal{D})$$

To compute the posterior, we need a model defined through the *likelihood* function (knowledge about the world):

$$g(\mathcal{D}, \vec{w}) = P(\mathcal{D} \mid \vec{w})$$

the data $\mathcal{D}$ (photo of the location with trees on it) and a *prior* pdf – an apriori (without data) belief of $\vec{w}$ (uniform location on earth):

$$h(\vec{w}) = P(\vec{w})$$

The *evidence* term in the Bayes theorem:

$$P(\mathcal{D}) = \int P(\mathcal{D} \mid \vec{w})\, P(\vec{w})\, d\vec{w}$$

is just a normalizing constant ensuring that the posterior is a proper pdf, i.e.

$$\int f(\vec{w})\, d\vec{w} = 1$$

$P(\mathcal{D})$ can be safely ignored in many algorithms. It can be also ignored when we want to compare probabilities of two values:

$$\frac{P(\vec{w}_1 \mid \mathcal{D})}{P(\vec{w}_1 \mid \mathcal{D})} = \frac{P(\mathcal{D} \mid \vec{w}_1) \cdot P(\vec{w}_1)}{P(\mathcal{D} \mid \vec{w}_2) \cdot P(\vec{w}_2)}$$

4.4 Reasons for doing the inference

Sometimes the posterior $P(\vec{w} \mid \mathcal{D})$ can be directly useful. Sometimes it can be used to predict some other quantity dependent on $\vec{w}$ (for instance expected temperature of the place on the photo). It can be also used to sample new data. But most often it is used to predict new data by computing their probability:

$$P(\mathcal{D}' \mid \mathcal{D}) = \int P(\mathcal{D}' \mid \vec{w}) \cdot P(\vec{w} \mid \mathcal{D})\, d\vec{w}$$

In case of the photo example, it would be the probability distribution over all possible photos taken in the same place (or the belief of what could be seen on the next photo).

4.5 Preference data and the Bradley–Terry model

Preference data consists of a series of choices between two or more alternatives. These can be choices of a human agent that we want to model and predict, but these can also be for example results of the matches in the Polish volleyball league (in such case the winner is "the choice").

The set of all choices is denoted $\mathcal{A}$. One observation d consists of a set of alternatives $A_d \subset \mathcal{A}$ and one observed preferred choice $c_d \in A_d$. We can write $d = (A_d, c_d)$, but sometimes we will omit the subscript d for simplicity.

The preference may not be deterministic. For instance, if there are two choices $\mathcal{A} = \{x, y\}$, the data series may consist of several choices of x out of $A = \{x, y\}$ and several choices of y out of the same set of alternatives. For instance:

$$\mathcal{D} = \left(\left(A, x\right), \left(A, x\right), \left(A, y\right)\right)$$

Preference data can be modeled with the Bradley–Terry model. The Bradley–Terry model is parameterized with a vector of weights, one for each choice: $\vec{w} = \{w_a\}_{a \in \mathcal{A}}$. It assumes (as most Bayesian models) that the preferred choices in all observations are dependent only on weights $\vec{w}$ and that with fixed weights the choices are independent:

$$P\left(\mathcal{D} \mid \vec{w}\right) = \prod_{d \in \mathcal{D}} P\left(d \mid \vec{w}\right)$$

The Bradley–Terry model assumes the probability of one observation d, i.e. the choice c out of A, to be:

$$P\left(d \mid \vec{w}\right) = P\left(\left(A, c\right) \mid \vec{w}\right) = \frac{w_c}{\sum_{a \in A} w_a}$$

The probability of any choice a is proportional to its weight w_a, but also depends on all the weights of the alternative choices. This equation is the heart of Bradley–Terry model.

The Bradley–Terry model can arise naturally in various settings. One example is a telephone router model. In this experiment there is a telephone routing machine with n telephone lines.

The time-to-next-call on line i can be modeled by an independent, exponential distribution function with a parameter λ_i:

$$P(T_i = t) = \lambda_i \exp(-\lambda_i t)$$

$$P(T_i > t) = \int_t^\infty \lambda_i \exp(-\lambda_i t')\, dt' = \exp(-\lambda_i t)$$

The probability that kth line calls first (before other lines) follows the Bradley–Terry model:

$$P(X_k \leqslant \min_{i \neq k} X_i) = \int_0^\infty P(X_k = x_k) P(x_k \leqslant \min_{i \neq k} X_i)\, dx_k$$

$$
\begin{aligned}
&= \int_0^\infty P(X_k = x_k) \prod_{i \neq k} P(x_k \leqslant X_i)\, dx_k \\
&= \int_0^\infty \lambda_k \exp(-\lambda_k x_k) \prod_{i \neq k} \exp(-\lambda_i x_k)\, dx_k \\
&= \lambda_k \int_0^\infty \exp(-\sum_i \lambda_i x_k)\, dx_k \\
&= \frac{\lambda_k}{\sum_i \lambda_i}
\end{aligned}
$$

This fact is used in section 4.7.

4.6 Parameter point estimation

Parameter point estimation simplifies the inference task by replacing the task of finding $P(\vec{w} \mid \mathcal{D})$ pdf with the task of finding maximum of it. I.e. instead of finding a whole belief function, point estimation just finds the most likely value. This conceptual simplification can significantly simplify and speedup an algorithm. This method is called *maximum aposteriori* (MAP):

$$
\begin{aligned}
\vec{w}_{\text{MAP}} &= \arg\max_{\vec{w}} f(\vec{w}) \\
&= \arg\max_{\vec{w}} P(\vec{w} \mid \mathcal{D})
\end{aligned}
$$

It is equivalent and usually easier to find maximizer of the logarithm of the posterior. Also, the evidence term $P(\mathcal{D})$ can be omitted as it is independent of $\vec{w}$.

$$
\begin{aligned}
\vec{w}_{\text{MAP}} &= \arg\max_{\vec{w}}\ \log P(\vec{w} \mid \mathcal{D}) \\
&= \arg\max_{\vec{w}}\ \log P(\mathcal{D} \mid \vec{w})\, P(\vec{w})
\end{aligned}
$$

It is common (but not theoretically correct) to assume the prior to be a constant $P(\vec{w}) = 1$. In such case, MAP method reduces to a very popular method called *maximal likelihood* (ML):

$$
\vec{w}_{\text{ML}} = \arg\max_{\vec{w}}\ \log P(\mathcal{D} \mid \vec{w})
$$

In case of the Bradley–Terry model, the log-likelihood is in the following form:

$$
\begin{aligned}
LL(\vec{w}) &= \log P(\mathcal{D} \mid \vec{w}) \\
&= \log \prod_{d \in \mathcal{D}} P(d \mid \vec{w}) \\
&= \sum_{d \in \mathcal{D}} \left(\log w_{c_d} - \log \sum_{a \in A_d} w_a \right)
\end{aligned}
$$

In order to find the maximum we may try to compare all partial derivatives against zero, but the system of equations is too complex to solve directly:

$$\begin{aligned} \frac{dLL}{dw_q}(\vec{w}) &= \sum_{d\in\mathcal{D}} \left(\frac{[q = c_d]}{w_{c_d}} - \frac{[q \in A_d]}{\sum_{a\in A_d} w_a} \right) \\ &= \frac{N_q}{w_q} - \sum_{d:q\in A_d} \frac{1}{\sum_{a\in A} w_a} = 0 \end{aligned} \tag{4.1}$$

N_q is the number of the observations where q was the preferred choice. The sum iterates over all data points d where q is one of the alternatives.

4.6.1 Gradient ascent

There are many iterative numerical procedures for finding a maximum of an arbitrary function. Each iteration i improves the current estimate $\vec{w}^{(i)}$ in the direction of the maximum. *Gradient ascent* methods are the most popular ones. The simplest gradient ascent starts with a guess $\vec{w}^{(0)}$ and moves it in the direction of the gradient.

$$\vec{w}^{(i+1)} \leftarrow \vec{w}^{(i)} + \alpha^{(i)} \frac{dLL}{d\vec{w}} \left(\vec{w}^{(i)} \right)$$

Given that learning rate $\alpha^{(i)}$ converge to zero not too fast and not too slow:

$$\sum_{i=0}^{\infty} \alpha^{(i)} = \infty$$

$$\sum_{i=0}^{\infty} \left(\alpha^{(i)} \right)^2 < \infty$$

The gradient ascent converges to a maximum of LL:

$$\vec{w}^{(i)} \longrightarrow_{i\to\infty} \vec{w}_{\mathrm{ML}}$$

Many log-likelihood functions including the one in the Bradley–Terry model are concave functions, so the maximum is global.

There are many problems with the gradient ascent methods. The most important is that it might be inefficient as it needs to go over all data to compute the gradient direction even if the data is redundant. Also, the speed of the convergence depends mostly on the learning rate schedule $\alpha^{(i)}$.

The former problem can be solved with the use of *Stochastic Gradient Descent*. In SGD, the gradient is computed based only on one random observation (one term from the LL sum) and $\vec{w}^{(i+1)}$ is updated immediately after.

The problem of finding the right learning rate schedule can be solved by using methods like the Newton's one. The learning rate is replaced by the inverse of Hessian of LL, but it is very costly to compute it. Usually the learning rate schedule has to be tuned to a particular function.

4.6.2 Minorization-Maximization

Minorization-Maximization (MM) [Hun04] algorithms, similarly to gradient methods, iteratively improves estimate of the maximum of LL. MM uses a concave minorizer $m(\vec{w}) \leqslant LL(\vec{w})$ tangent to LL in the current estimate and takes the maximum of m as a new estimate. This is illustrated in Figure 4.3. The trick is to use such a minorizer m so it is easy to find the maximum of m in a closed algebraic formula.

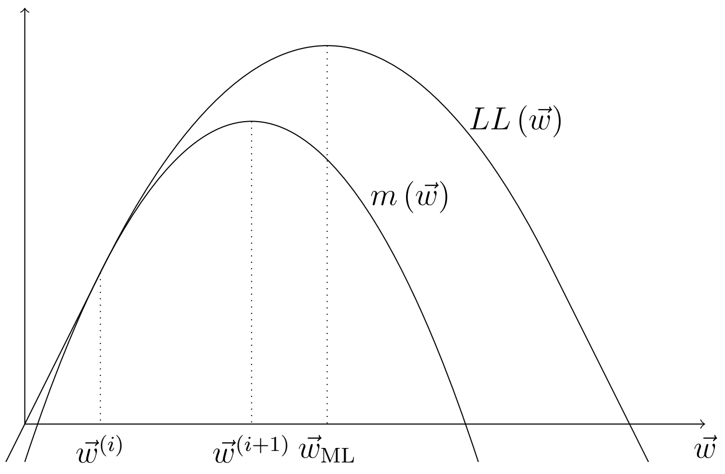


Figure 4.3: The idea of Minorization-maximization algorithm.

4.6.3 Advantages and disadvantages of point estimation

The maximum $\vec{w}_{\mathrm{MAP}}$ can be directly used for sampling and prediction. Instead of averaging over all possible $\vec{w}$ weighted with its pdf:

$$P(\mathcal{D}' \mid \mathcal{D}) = \int P(\mathcal{D}' \mid \vec{w}) \cdot P(\vec{w} \mid \mathcal{D})\, d\vec{w}$$

point estimation uses an approximation using only the most likely value:

$$P(\mathcal{D}' \mid \mathcal{D}) \approx P(\mathcal{D}' \mid \vec{w}_{\mathrm{MAP}})$$

This is not a theoretically correct approach but it works well in certain circumstances. For instance the full integral is well approximated by ML when posterior is unimodal with a small variance. When the variance of the posterior is large, i.e. when the data is scarce, ML may give bad results. This effect is sometimes called *overfitting.*

Nevertheless, MAP and ML are very popular in practice thanks to their simplicity and computational efficiency.

4.7 Laplace$_q$ Marginal Propagation

Point estimation methods are just an approximation of Bayesian inference since they only take into account one (the most likely) parameter value. Here we propose a method that approximates a whole $P(\vec{w} \mid \mathcal{D})$ distribution evaluated by Bayes theorem.

Laplace$_q$ Marginal Propagation (LMP) consists of several ideas combined in a novel way:

- Posterior factorization
- Laplace$_q$ approximation
- Iterative factor improvement
- Minorization-Maximization approximation

This techniques allow us to create a very efficient optimization algorithm without the disadvantages of the gradient methods and MM method described earlier.

4.7.1 Posterior factorization

The main idea of the proposed algorithm is to approximate the posterior with a product of the marginals:

$$P(\vec{w} \mid \mathcal{D}) \approx \prod_{a \in \mathcal{A}} P(w_a \mid \mathcal{D})$$

and to approximate each marginal with a product of factor functions $q_{a,d}$ parameterized with $\theta_{a,d}$, one for each data point and one for the prior:

$$P(w_a \mid \mathcal{D}) \approx \prod_{d \in \mathcal{D} \cup \{\text{-}\}} q_{a,d}(w_a, \theta_{a,d}) = q_a(w_a, \theta_a)$$

LMP finds such parameters $\vec{\theta}$ so that the approximation:

$$P(\vec{w} \mid \mathcal{D}) \approx \prod_{a \in \mathcal{A}} \prod_{d \in \mathcal{D} \cup \{\text{-}\}} q_{a,d}(w_a, \theta_{a,d})$$

is as good as possible in the sense of Laplace$_q$ approximation described in section 4.7.3.

Factors in the Bradley–Terry model

We have chosen an unnormalized Gamma distribution (another good choice would be Log-Normal distribution) to use as factors $q_{a,d}(w_a, \theta_{a,d})$ in the Bradley–Terry model. In such case, each factor has two parameters denoted as α and β, so we have:

$$\theta_{a,d} = (\alpha_{a,d}, \beta_{a,d})$$

$$q_{a,d}(w_a, \theta_{a,d}) = w_a^{\alpha_{a,d}} \exp(-\beta_{a,d} w_a)$$

There are several reasons for this choice. Gamma distribution is the most popular and tractable distribution defined on $\mathbb{R}_+$. For instance we get a simple form of a product of factors:

$$q_a(w_a, \theta_a) = \prod_d q_{a,d}(w_a, \theta_a) = w_a^{\alpha_a} \exp(-\beta_a w_a)$$

$$\text{where} \quad \alpha_a = \sum_d \alpha_{a,d} \quad \text{and} \quad \beta_a = \sum_d \beta_{a,d}$$

Since the factors model the belief of $\vec{w}$, ideally they should have a form of the conjugate prior[1] to the likelihood in the Bradley–Terry model. There is no simple conjugate prior to the Bradley–Terry likelihood, but as shown in section 4.5 preference data can be interpreted as telephone router data that is modeled as a set of exponential distributions. Gamma distribution is a conjugate prior to exponential distribution.

4.7.2 Laplace approximation

A classical Laplace approximation of pdf $p(x)$ works by finding (unnormalized) Normal distribution strongly tangent at the mode of $p(x)$. More precisely, to find an approximation of density $p(x)$ with a mode at x' with Normal distribution $\mathcal{N}(x, \mu, \sigma^2)$, we find such μ and σ^2 so that $\log p(x)$ is strongly tangent to $\log(c \cdot \mathcal{N}(x, \mu, \sigma^2))$, i.e.

$$\begin{aligned} \log p(x') &= \log(c \cdot \mathcal{N}(x', \mu, \sigma^2)) \\ \frac{d \log p(x)}{dx}(x') &= \frac{d \log(c \cdot \mathcal{N}(x, \mu, \sigma^2))}{dx}(x') \\ \frac{d^2 \log p(x)}{d^2 x}(x') &= \frac{d^2 \log(c \cdot \mathcal{N}(x, \mu, \sigma^2))}{d^2 x}(x') \end{aligned}$$

Parameter c is used only to satisfy the first equation and it is not used anywhere else. We omit it and the whole first equation in further considerations. The solution of the remaining two equations is:

$$\begin{aligned} \mu &= x' \\ \sigma^2 &= -\left(\frac{d^2 \log p(x)}{d^2 x}(x')\right)^{-1} \end{aligned}$$

We denote the solution using $\text{Laplace}_{\mathcal{N}}$ operator:

$$(\mu, \sigma^2) = \text{Laplace}_{\mathcal{N}}(p)$$

4.7.3 Laplace$_q$ approximation

The idea of *Laplace*$_q$ approximation is to use the factors $q(\vec{w}, \vec{\theta})$ instead of a Normal distribution:

$$\begin{aligned} \frac{d \log p(\vec{w})}{d\vec{w}}(\vec{w}') &= \frac{d \log q(\vec{w}, \vec{\theta})}{d\vec{w}}(\vec{w}') \\ \frac{d^2 \log p(\vec{w})}{d^2\vec{w}}(\vec{w}') &= \frac{d^2 \log q(\vec{w}, \vec{\theta})}{d^2\vec{w}}(\vec{w}') \end{aligned}$$

[1]Conjugate prior is a parameterized distribution that when multiplied by the likelihood yields the same distribution but with different parameters.

Thanks to the product form of $q(\vec{w}, \vec{\theta})$, if $p(\vec{w})$ can be also factored in the form $p(\vec{w}) = \prod_a p_a(w_a)$, then we can simplify the task and solve for each $\vec{\theta}_a$ separately:

$$\frac{d \log p(\vec{w})}{dw_a}(w'_a) = \frac{d \log q_a(w_a, \theta_a)}{dw_a}(w'_a) \tag{4.2}$$

$$\frac{d^2 \log p(\vec{w})}{d^2 w_a}(w'_a) = \frac{d^2 \log q_a(w_a, \theta_a)}{d^2 w_a}(w'_a) \tag{4.3}$$

We denote the solution using Laplace_q operator:

$$\vec{\theta} = \text{Laplace}_q(p)$$

Note that since there are two equations, $\vec{\theta}$ should be a vector of pairs of variables.

Laplace_q in the Bradley–Terry model with Gamma factors

Here we derive the mode and value of second derivative at mode (sdm) of $\log q_a(w_a, \theta_a)$ needed in Laplace_q approximation in our Bradley–Terry example. Mode:

$$\frac{d \log q_a(w_a, \theta_a)}{dw_a}(w'_a) = \frac{d}{dw_a}(\alpha_a \log(w_a) - \beta_a w_a)(w'_a) = \frac{\alpha_a}{w'_a} - \beta_a = 0$$

$$w'_a = \frac{\alpha_a}{\beta_b} \tag{4.4}$$

Value of sdm:

$$\frac{d^2 \log q_a(w_a, \theta_a)}{d^2 w_a}(w'_a) = -\frac{\alpha_a}{w'^2_a} = -\frac{\beta_a^2}{\alpha_a} \tag{4.5}$$

4.7.4 Laplace_q Marginal Propagation

Laplace_q Marginal Propagation (LMP) is an iterative procedure to find a good approximation of the posterior. It maintains and iteratively improves vector of parameters $\vec{\theta}$ so that $q(\vec{w}, \vec{\theta})$ is a better and better approximation of the posterior.

One iteration processes one data point $d \in \mathcal{D} \cup \{-\}$. Within product of factors $q(\vec{w}, \vec{\theta})$, the part responsible for modeling of d is :

$$\prod_{a \in \mathcal{A}} q_{a,d}(w_a, \theta_{a,d})$$

In each iteration, LMP first removes the factor responsible for d from $q(\vec{w}, \vec{\theta})$. The approximate factor is replaced with an appropriate factor from the likelihood, i.e. $P(d \mid \vec{w})$. We denote such a mixed product with $p_d(\vec{w}, \vec{\theta})$:

$$p_d(\vec{w}, \vec{\theta}) = \frac{q(\vec{w}, \vec{\theta})}{\prod_{a \in \mathcal{A}} q_{a,d}(w_a, \theta_{a,d})} \cdot P(d \mid \vec{w}) \tag{4.6}$$

$$= \prod_{a \in \mathcal{A}} \prod_{\substack{d' \in \mathcal{D} \\ d' \neq d}} q_{a,d'}(w_a, \theta_{a,d'}) \cdot P(d \mid \vec{w}) \tag{4.7}$$

Then Laplace_q operator is applied to obtain a new better approximation of this function with a new product of factors, i.e. a new $\vec{\theta}$:

$$\vec{\theta} \leftarrow \text{Laplace}_q\left(p_d(\vec{w}, \vec{\theta})\right)$$

We loop over data set (and prior) and perform this update until convergence. In order to make this computation efficient we need to evaluate Laplace_q operator efficiently.

The Bradley–Terry model

Laplace_q tries find such $\vec{\theta}_{\text{new}}$ so that mode and the second derivatives at the mode of $\log q(\vec{w}, \vec{\theta}_{\text{new}})$ and $\log p_d(\vec{w}, \vec{\theta})$ are equal. In order to evaluate Laplace_q, we need $\log p_d(\vec{w}, \vec{\theta})$ from equation 4.7 with $d = (A_d, c_d)$:

$$\begin{aligned} p_d(\vec{w}, \vec{\theta}) &= \frac{w_{c_d}}{\sum_{a \in A_d} w_a} \cdot \prod_{a \in \mathcal{A}} w_a^{(\alpha_a - \alpha_{a,d})} \exp(-(\beta_a - \beta_{a,d}) w_a) \\ \log p_d(\vec{w}, \vec{\theta}) &= -\log\left(\sum_{a \in A_d} w_a\right) + \sum_{a \in \mathcal{A}} \alpha'_a \log w_a - \beta'_a w_a \\ \text{where} \quad \alpha'_a &= \alpha_a - \alpha_{a,d} + [a = c_d] \quad \text{and} \quad \beta'_a = \beta_a - \beta_{a,d} \end{aligned}$$

Unfortunately, we are not able to solve the set of equations $\frac{d \log p_d(\vec{w}, \vec{\theta})}{d\vec{w}} = \vec{0}$ in a closed algebraic form.

4.7.5 Minorization-Maximization for Laplace_q

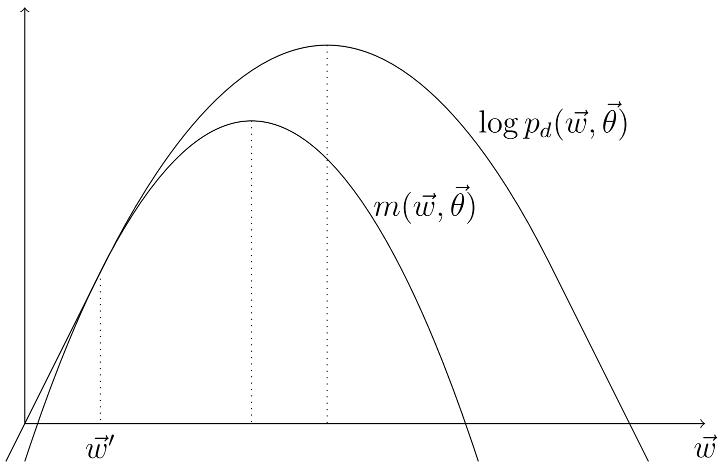


Figure 4.4: Minorization-Maximization for Laplace_q algorithm.

Since we cannot find the mode directly, we employ Minorization-Maximization algorithm. We minorize $\log p_d(\vec{w}, \vec{\theta})$, which is a concave with another concave function $m(\vec{w}, \vec{\theta})$

tangent to the former in $\vec{w}'$ which is a mode of $\log q(\vec{w}, \vec{\theta})$:

$$\begin{aligned}\log p_d(\vec{w}, \vec{\theta}) &\geqslant m(\vec{w}, \vec{\theta}) \\ \log p_d(\vec{w}', \vec{\theta}) &= m(\vec{w}', \vec{\theta}) \\ \vec{w}' &= \left(\frac{\alpha_a}{\beta_a}\right)_{a \in \mathcal{A}}\end{aligned}$$

This guarantees that the mode of $m(\vec{w}, \vec{\theta})$ is closer to the mode of $\log p_d(\vec{w}, \vec{\theta})$ than $\vec{w}'$. Iterating this procedure with $\vec{w}'$ being the mode of m from previous iteration would yield Minorization-Maximization(MM) algorithm described in section 4.6.2. MM would yield a convergence to the mode of $\log p_d(\vec{w}, \vec{\theta})$, but in practice the second step of the iteration is not worth the cost. It is better to move to a next data point and come back to the current data point later.

The minorizer is used in the Laplace_q operator instead of p_d:

$$\vec{\theta} \leftarrow \text{Laplace}_q\left(m(\vec{w}, \vec{\theta})\right)$$

To define a suitable minorizer we apply inequality:

$$-\log x \geqslant 1 - \frac{x}{s} - \log s$$

to $\log p_d(\vec{w}, \vec{\theta})$. The only point where both sides are equal is $x = s$. We choose s so that the equality coincides with the old estimation of the mode, i.e. $s = \sum_{a \in A_d} \frac{\alpha_a}{\beta_a}$ and we get:

$$\begin{aligned}\log p_d(\vec{w}, \vec{\theta}) &\geqslant 1 - \frac{\sum_{a \in A_d} w_a}{\sum_{a \in A_d} \frac{\alpha_a}{\beta_a}} - \log \sum_{a \in A_d} \frac{\alpha_a}{\beta_a} + \sum_{a \in \mathcal{A}} \alpha'_a \log w_a - \beta'_a w_a \\ &= m(\vec{w}, \vec{\theta})\end{aligned}$$

The mode and second derivative at the mode of $m(\vec{w}, \vec{\theta})$:

$$\begin{aligned}\frac{dm(\vec{w}, \vec{\theta})}{dw_a}(w'_a) &= -\frac{[a \in A_d]}{\sum_{a \in A_d} \frac{\alpha_a}{\beta_a}} + \frac{\alpha'_a}{w'_a} - \beta'_a = 0 \\ w'_a &= \frac{\alpha'_a}{\frac{[a \in A_d]}{\sum_{a \in A_d} \frac{\alpha_a}{\beta_a}} + \beta'_a} \\ \frac{d^2 m(\vec{w}, \vec{\theta})}{d^2 w_a}(w'_a) &= -\frac{\alpha'_a}{w'^2_a} \\ &= -\frac{\left(\frac{[a \in A_d]}{\sum_{a \in A_d} \frac{\alpha_a}{\beta_a}} + \beta'_a\right)^2}{\alpha'_a}\end{aligned}$$

Next, as in equations 4.2 and 4.3, we match mode w'_a and sdm with mode and sdm of new factored approximation of posterior q: 4.4 and 4.5.

$$\frac{\alpha'_a}{\frac{[a\in A_d]}{\sum_{a\in A_d}\frac{\alpha_a}{\beta_a}}+\beta'_a} = \frac{\alpha_a^{\text{new}}}{\beta_a^{\text{new}}}$$
$$-\frac{\left(\frac{[a\in A_d]}{\sum_{a\in A_d}\frac{\alpha_a}{\beta_a}}+\beta'_a\right)^2}{\alpha'_a} = -\frac{(\beta_a^{\text{new}})^2}{\alpha_a^{\text{new}}}$$

Solving for θ_a^{new} we get a simple updating rule:

$$\alpha_a^{\text{new}} = \alpha'_a$$
$$\beta_a^{\text{new}} = \frac{[a\in A_d]}{\sum_{a\in A_d}\frac{\alpha_a}{\beta_a}}+\beta'_a$$

We can see that if $[a \in A_d] = 0$ then θ_a does not change. This is in agreement with the intuition, i.e. if an alternative does not compete in the current data point it should not change. Of course, even a data point not containing alternative a can influence a's weight indirectly. It can happen by changing weights of alternatives competing with a in other data points. This information *propagates through marginals* and is taken into account next time the other data points containing a are processed.

Every $\alpha_{a,d} = 0$, except $\alpha_{p,d} = 1$, and does not change, also every $\beta_{a,d}$ has the same value for the same d and different a. This allows to further simplify the algorithm and keep just a single $\delta_d = \beta_{a,d}$ for each data point. Algorithm 1 shows the pseudo-code.

4.7.6 The algorithm

For the sake of completeness of this section we repeat below the description of the data and the problem.

The problem statement. Preference data consists of a series of choices between two or more alternatives. One data point $d = (A_d, c_d)$ consists of a set of alternatives A_d and one observed preferred choice $c_d \in A_d$. For instance:

$$\mathcal{D} = ((\{x,y\},x),(\{x,y\},x),(\{x,y\},y))$$

Such data can be modeled with the Bradley–Terry model. To each possible alternative we assign one hidden variable. In the example it would be w_x, w_y. In the Bradley–Terry model the probability of every data point is:

$$P(d \mid \vec{w}) = \frac{w_{c_d}}{\sum_{a\in A_d} w_a}$$

Algorithm 1 Laplace$_q$ Marginal Propagation for the Bradley–Terry model.

```
Input: $\mathcal{D}, \vec{\theta}_{\text{prior}}$
  $\vec{\alpha}, \vec{\beta} \leftarrow \vec{\theta}_{\text{prior}}$

  while not converged do
    choose any $d = (A_d, c_d) \in \mathcal{D}$
    if $d$ is chosen for the first time then
      $\alpha_{c_d} \leftarrow \alpha_{c_d} + 1$
      $\delta_d \leftarrow 1$
      for all $a \in A_d$ do $\beta_a \leftarrow \beta_a + \delta_d$
    end if

    $\delta'_d \leftarrow \left(\sum_{a \in A_d} \frac{\alpha_a}{\beta_a}\right)^{-1}$
    for all $a \in A_d$ do
      $\beta_a \leftarrow \beta_a - \delta_d + \delta'_d$
      $\delta_d \leftarrow \delta'_d$
    end for
  end while

  return $\vec{\alpha}, \vec{\beta}$
```

The probability of the whole data set is the product of probabilities of all data points (independence assumption). For our toy data set $\mathcal{D}$ we have:

$$P(\mathcal{D} \mid \vec{w}) = \frac{w_x}{w_x + w_y} \frac{w_x}{w_x + w_y} \frac{w_y}{w_x + w_y}$$

We can apply Bayes theorem to derive distribution of $\vec{w}$ given the data:

$$P(\vec{w} \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \vec{w}) \cdot P(\vec{w})}{P(\mathcal{D})} \propto P(\mathcal{D} \mid \vec{w}) \cdot P(\vec{w})$$

We model this distribution with the product of Gamma distributions.

$$\begin{aligned} P(\mathcal{D} \mid \vec{w}) \cdot P(\vec{w}) &\propto \prod_{a \in \mathcal{A}} w_a^{\alpha_a} \exp(\beta_a w_a) \\ &= \prod_{a \in \mathcal{A}} q_a(w_a, \alpha_a, \beta_a) \\ &= q(\vec{w}, \vec{\alpha}, \vec{\beta}) \end{aligned}$$

4.8 Convergence proof

In this section we show that the mode of $q(\vec{w}, \vec{\alpha}, \vec{\beta})$ distribution returned by Algorithm 1 converges to the mode of likelihood $P(\mathcal{D} \mid \vec{w})$. For the sake of simplicity of the proof we assume an uniform prior and that each action is preferred at least once.

Definition 1. *Datapoints and alternatives form a bipartite graph. The set of alternatives of a data point d is denoted A_d. D_a denotes a set of all data points where the alternative a is in the set of alternatives. Note that we have:*

$$D_a = \{d \in \mathcal{D} : a \in A_d\}$$

$$A_d = \{a \in \mathcal{A} : d \in D_a\}$$

Definition 2. *$\vec{\alpha}$ are the parameters of the algorithm. α_a is the number of datapoints where alternative a was preferred.*

Definition 3. *We denote normalization operation with $n(\cdot)$ that casts a point in $\mathbb{R}^n$ onto a simplex:*

$$n : \mathbb{R}^n \to \mathbb{S}_n$$
$$n(\vec{a}) = \frac{\vec{a}}{|\vec{a}|_1}$$

Definition 4. *One step of Algorithm 1 updates coordinate d of vector $\vec{\delta}$ using this formula:*

$$u_d(\vec{\delta}) = \left(\sum_{a \in A_d} \frac{\alpha_a}{\sum_{\hat{d} \in D_a} \delta_{\hat{d}}} \right)^{-1}$$

U_d is defined as a single step of Algorithm 1 composed with normalization to the simplex:

$$U_d(\vec{\delta}) = n\left(\left(\delta_1, \delta_2, \ldots, u_d(\vec{\delta}), \ldots, \delta_{|D|} \right) \right)$$

We denote the normalized sequence produced by the algorithm as $\vec{\delta}^{(i)}$:

$$\vec{\delta}^{(i+1)} = U_d(\vec{\delta}^{(i)})$$

Definition 5. *Log-likelihood is the potential function in the alternatives' simplex:*

$$LL(\vec{w}) = \sum_{d \in \mathcal{D}} \log \frac{w_{c_d}}{\sum_{a \in A_d} w_a} = \sum_{a \in \mathcal{A}} \alpha_a \log w_a - \sum_{d \in \mathcal{D}} \log \left(\sum_{a \in A_d} w_a \right)$$

The potential function in the datapoints' simplex:

$$f(\vec{\delta}) = \sum_{d \in \mathcal{D}} \log \frac{\delta_d}{\sum_{d \in D_{c_d}} \delta_d} = \sum_{d \in \mathcal{D}} \log \delta_d - \sum_{a \in \mathcal{A}} \alpha_a \log \left(\sum_{d \in D_a} \delta_d \right)$$

Lemma 1. *Function* $g(\vec{x}) = \log\left(\sum_{i=1}^{n} \exp(x_i)\right)$ *is convex. The convexity is strict, except for lines* $g(\vec{x} + \vec{c})$ *where the function is constant.*

Proof. Definition of convexity applied to $g(\vec{x})$ is equivalent to Holder's inequality. The conditions of strictness of convexity of $g(\vec{x})$ follow from the conditions of strictness of Holder's inequality. □

Lemma 2. *Functions* $f(\exp(\vec{x}))$ *and* $LL(\exp(\vec{x}))$ *are concave. The concavity is strict, except for lines* $f(\exp(\vec{x} + \vec{c}))$ *and* $LL(\exp(\vec{x} + \vec{c}))$ *where the functions are constant.*

Proof. In Definition 5, we can see that $f(\exp(\vec{x}))$ and $LL(\exp(\vec{x}))$ are sums of terms. Logarithm of the numerator of each term is linear. Logarithm of the denominator is concave as it is negation of a function shown to be convex in Lemma 1. □

Lemma 3. *Functions* $f(\vec{\delta})$ *and* $LL(\vec{w})$ *have unique maximums in the datapoints' simplex and the alternatives' simplex respectively. We will denote these maximums as* $\vec{\delta}^*$ *and as* $\vec{w}^*$ *respectively.*

Proof. It is enough to show that functions $f(\exp(\vec{x}))$ and $LL(\exp(\vec{x}))$ have a unique maximum on the logarithm of the respective simplexes.

Lemma 2 proves that these functions are strictly concave on their respective logarithmized simplexes. The exponent of points at the boundaries of the logarithm of a simplex are in boundaries of the simplex and therefore have at least one coordinate equal to zero. In case of $LL(\cdot)$, since every action is preferred at least once, for every a, $\log(w_a)$ will appear at least once in the sum, driving a whole sum towards negative infinity. In case of $f(\cdot)$, every $\log(\delta_d)$ appears in the sum explicitly.

If a function is strictly concave on a set and tends towards negative infinity on its boundaries then it has a unique global maximum. □

Definition 6. *The datapoints' space is dual to the alternatives' space. Coordinate transformations into dual spaces are defined as follows:*

$$\vec{w} : \mathbb{R}^D \to \mathbb{R}^A \qquad \qquad \vec{\delta} : \mathbb{R}^A \to \mathbb{R}^D$$

$$\vec{w}(\vec{\delta}) = \left(\frac{\alpha_a}{\sum_{d \in D_a} \delta_d}\right)_{a \in \mathcal{A}} \qquad \vec{\delta}(\vec{w}) = \left(\frac{1}{\sum_{a \in A_d} w_a}\right)_{d \in \mathcal{D}}$$

Note that the coordinate transformations are not inverse functions, i.e. usually $\vec{w}(\vec{\delta}(\vec{w})) \neq \vec{w}$.

Lemma 4. $n(\vec{w}(\vec{\delta}^*)) = \vec{w}^*$

Proof. Lemma 3 shows that $\vec{w}^*$ and $\vec{\delta}^*$ are the only critical points of LL and f respectively. So it is enough to show that $\frac{df}{d\vec{\delta}}(\vec{\delta}) = \vec{0}$ implies $\frac{dLL}{d\vec{w}}(\vec{w}(\vec{\delta})) = \vec{0}$

Criticality of $f(\vec{\delta})$:

$$\frac{df}{d\delta_{\hat{d}}}(\vec{\delta}) = \frac{1}{\delta_{\hat{d}}} - \sum_{a\in\mathcal{A}} \alpha_a \frac{[\hat{d} \in D_a]}{\sum_{d\in D_a}\delta_d} = \frac{1}{\delta_{\hat{d}}} - \sum_{a\in A_{\hat{d}}} \frac{\alpha_a}{\sum_{d\in D_a}\delta_d} = 0$$

$$\delta_{\hat{d}} = \left(\sum_{a\in A_{\hat{d}}} \frac{\alpha_a}{\sum_{d\in D_a}\delta_d}\right)^{-1} = \left(\sum_{a\in A_{\hat{d}}} (\vec{w}(\vec{\delta}))_a\right)^{-1} \tag{4.8}$$

Criticality of $LL(\vec{w}(\vec{\delta}))$:

$$\frac{dLL}{dw_{\hat{a}}}(\vec{w}) = \frac{\alpha_{\hat{a}}}{w_{\hat{a}}} - \sum_{d\in D_{\hat{a}}} \left(\sum_{a\in A_d} w_a\right)^{-1}$$

$$\frac{dLL}{dw_{\hat{a}}}(\vec{w}(\vec{\delta}))) = \frac{\alpha_{\hat{a}}}{\frac{\alpha_{\hat{a}}}{\sum_{d\in D_{\hat{a}}}\delta_d}} - \sum_{d\in D_{\hat{a}}} \left(\sum_{a\in A_d} (\vec{w}(\vec{\delta}))_a\right)^{-1} = 0$$

The last line is shown by substituting equation 4.8. Criticality of $LL(n(\vec{w}(\vec{\delta})))$ is trivial as rescaling of the argument does not change LL's gradient. □

Lemma 5. *Sequence $f(\vec{\delta}^{(i)})$ is monotonic:*

$$f(\vec{\delta}^{(i+1)}) \geqslant f(\vec{\delta}^{(i)})$$

It is constant only when $\vec{\delta}^{(i)}$ is a critical point:

$$f(\vec{\delta}^{(i+1)}) = f(\vec{\delta}^{(i)}) \implies \frac{df}{d\vec{\delta}}(\vec{\delta}^{(i)}) = \vec{0}$$

Proof. For the first part, it is enough to prove that:

$$f(U_{\hat{d}}(\vec{\delta})) \geqslant f(\vec{\delta})$$

The following inequality holds with both sides being equal only for $x = s$:

$$-\log x \geqslant -\left(\log s - 1 + \frac{x}{s}\right)$$

We apply it to the second logarithm in f with the following s:

$$s = \sum_{d\in D_a} \delta_d^{(i)}$$

$$f(\vec{\delta}) = \sum_{d\in\mathcal{D}} \log\delta_d - \sum_{a\in\mathcal{A}} \alpha_a \log\left(\sum_{d\in D_a}\delta_d\right)$$

$$f(\vec{\delta}) \geqslant \sum_{d\in\mathcal{D}} \log\delta_d - \sum_{a\in\mathcal{A}} \alpha_a \left(\log s - 1 + \frac{\sum_{d\in D_a}\delta_d}{s}\right) = m(\vec{\delta})$$

We find a unique maximum $\vec{\delta}^M$ of $m(\vec{\delta})$ on the line where all coordinates are fixed except $\hat{d}$:

$$\frac{dm}{d\delta_{\hat{d}}}(\vec{\delta}^M) = \frac{1}{\delta^M_{\hat{d}}} - \sum_{a \in \mathcal{A}} \alpha_a \frac{[\hat{d} \in D_a]}{s} = \frac{1}{\delta^M_{\hat{d}}} - \sum_{a \in A_{\hat{d}}} \frac{\alpha_a}{s} = 0$$

$$\delta^M_{\hat{d}} = \left(\sum_{a \in A_{\hat{d}}} \frac{\alpha_a}{\sum_{d \in D_a} \delta^{(i)}_d} \right)^{-1}$$

Because $\delta^M_{\hat{d}} = u_{\hat{d}}(\vec{\delta})$ we have $\vec{\delta}^{(i+1)} = n(\vec{\delta}^M)$ and:

$$f(\vec{\delta}^{(i+1)}) = f(n(\vec{\delta}^M)) = f(\vec{\delta}^M) \geqslant m(\vec{\delta}^M) \geqslant m(\vec{\delta}^{(i)}) = f(\vec{\delta}^{(i)})$$

If $\vec{\delta}^{(i)}$ is a maximum of f then we have $\vec{\delta}^M = \vec{\delta}^{(i)}$. Otherwise the inequality is strict. □

Lemma 6. $\vec{\delta}^{(i)} \to \vec{\delta}^*$

Proof. The sequence $\vec{\delta}^{(i)}$ has a convergent subsequence because the simplex - Co-domain of n is compact.

$$\vec{\delta}^{(i_k)} \to \vec{\delta}^L$$

Since f is continuous, we have:

$$f(\vec{\delta}^{(i_k)}) \to f(\vec{\delta}^L)$$

$f(\vec{\delta}^{(i)})$ is monotonic by Lemma 5. So by squeeze theorem we have:

$$f(\vec{\delta}^{(i)}) \to f(\vec{\delta}^L)$$

The last thing to show is that $\vec{\delta}^L$ is at a maximum of f. Let us take any coordinate d. By continuity of U_d and f we have:

$$f(U_d(\vec{\delta}^{(i_k)})) \to f(U_d(\vec{\delta}^L))$$

Since d appears infinitely many times in the sequence of updates, we have:

$$f(U_d(\vec{\delta}^{(i_k)})) \to f(\vec{\delta}^L)$$

So:

$$f(U_d(\vec{\delta}^L)) = f(\vec{\delta}^L)$$

By Lemma 5, $\vec{\delta}^L$ is the maximum of f. □

Lemma 7.

$$n(\vec{w}(\vec{\delta}^{(i)})) \to \vec{w}^*$$

Proof. From Lemma 6

$$\vec{\delta}^{(i)} \to \vec{\delta}^*$$

By applying a continuous function $n(\vec{w}(\vec{\delta}))$ to both sides and Lemma 4 we get:

$$n(\vec{w}(\vec{\delta}^{(i)})) \to n(\vec{w}(\vec{\delta}^*)) = \vec{w}^*$$

□

Theorem 1. *The mode of $q(\vec{w}, \vec{\alpha}, \vec{\beta})$ returned by Algorithm 1 converges to a mode of likelihood $P(\mathcal{D} \mid \vec{w})$.*

Proof. The likelihood $P(\mathcal{D} \mid \vec{w})$ is uniform, i.e.

$$P(\mathcal{D} \mid \vec{w}) = P(\mathcal{D} \mid c \cdot \vec{w})$$

The mode of the likelihood is a line crossing the simplex in exactly one point. So it is enough to show that mode of $q(\vec{w}, \vec{\alpha}, \vec{\beta})$ after being normalized to the simplex, converges to the same point. That is exactly what Lemma 7 shows:

The left side: $n(\vec{w}(\vec{\delta}^{(i)}))$ is equal to the mode of q equal $\frac{\vec{\alpha}}{\vec{\beta}}$.

The right side: $\vec{w}^*$ is a mode of log-likelihood in the simplex. $\square$

Chapter 5

Decomposition-based Combinatorial Game Search

In 1997, a thousand dollar bet was made between John Tromp and Darren Cook. John claimed that he would not be defeated by a Go program at least until the end of the year 2010 and Darren accepted the bet. In December 2010, John convincingly won the series 4 – 0. Games from that series are a good illustration of the main weakness of the state of the art Monte Carlo Go programs. Figure 5.1 shows the final position from the first game.

Independent subgames. Like most 19×19 Go positions, the one shown in Figure 5.1 can be separated into many independent subgames. Independent means that optimal strategies can be defined on local, disjoint parts of the board. Several of the subgames are marked with letters A-K:

- Groups marked with A, B, C, E, F are dead (cannot avoid capture, regardless of who plays first).
- White group D can always connect and therefore black group D is dead as well.
- White can locally ensure that stone G is connected to the big white group above it.
- Black can locally ensure that white group I will not be able to break out to the center.
- White can ensure that if Black plays in the area around J, he is not be able to live there and therefore this whole area is White's territory.
- Area around K can remain as Black's territory but White can reduce it from left and right. Even though subgame K is not settled, it is still independent from other subgames.

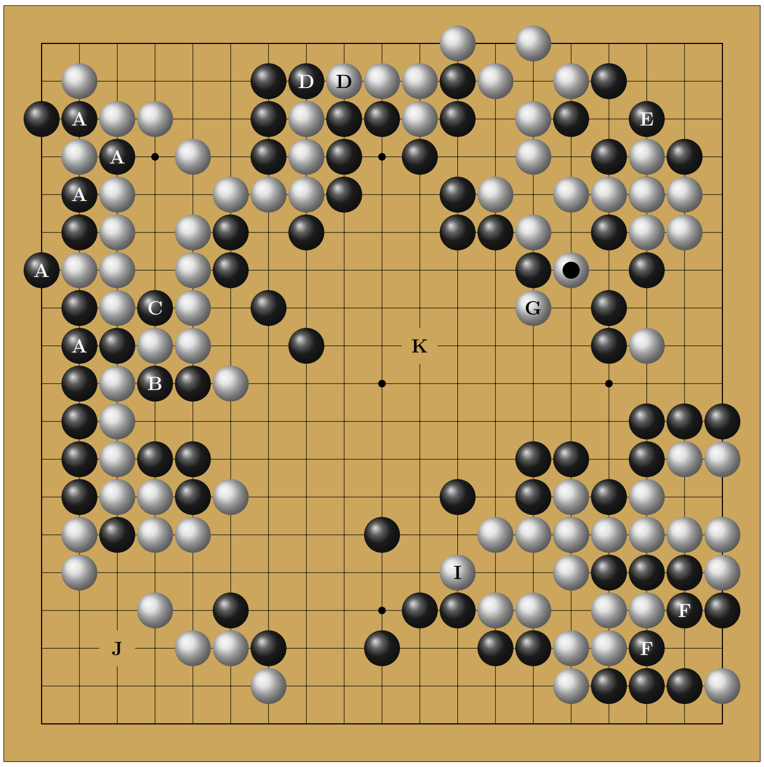


Figure 5.1: The final position from the “Shodan Go Bet” series. Computer (Black) resigned. The last move is marked with a black dot. Several independent subgames are marked with letters A-K.

The main problem in the state of the art Monte Carlo Go. The state of the art Monte Carlo Go program consists of a global Monte Carlo Tree Search (MCTS) grown with Heavy Playouts. This approach is hugely inefficient in Go positions consisting of many independent or almost independent subgames.

In a typical 19×19 Go position there are a few complicated and dozens of simple subgames. Human players deal with such situations by searching each subgame independently and judging the value of each move in each subgame. Monte Carlo Tree Search (MCTS) is capable of reading out any of these sequences (finding good local strategies), but it has difficulty reading all of them at the same time. If there are k subgames with average game tree size of $O(n)$, then the size of the global game tree is approximately $O(n^k)$. In practice, there are not enough computational resources to search the whole game tree. A divide and conquer algorithm that could take advantage of this structure would work exponentially faster.

The horizon effect. MCTS is able to search only through one or two subgames in every branch of the tree. Playing out the rest of the subgames is left to the heuristics of the heavy playouts. Heavy playouts are able to play out most of the simple, small subgames reasonably well. But they are unable to correctly play out a huge variety of complex subgames occurring in real Go games, even though MCTS would have no problem with finding good variations. This problem is called *a horizon effect*.

In Go game, there are many subgames that have a little or zero value in playing a move in them but with a property that if one player plays a move, then a correct answer must be played to maintain the subgame's status quo. In case of the position in Figure 5.1 that would be all the marked subgames except K. Heavy playouts would almost always play out correctly subgames C, D, I. Subgames J and G would be played out correctly most of the times. But pattern and feature based heavy playouts are unable to play out complex strategies required to maintain a status quo in the rest of the subgames. This leads to a consistent misevaluation of the position's value.

The horizon effect is a central problem in the Monte Carlo Go framework. The best known way of circumventing this problem is adding more and more knowledge to the heavy playouts. It takes a form of a specialized, hard-coded Go heuristics to cover a wider variety of situations occurring in the game. But as the history of the classical Go has taught us, there are diminishing returns in such an approach.

The adaptive patterns idea. The heavy playouts are mainly driven by patterns such as the ones described in section 3.4.1. The weights attached to the patterns are learned from games of good players and are a crude approximation of the value of playing in the center of the pattern. Similar weights are stored in nodes of MCTS tree but these are derived from the results of the playouts. In order to deal with horizon effect of a global search tree, it is natural to try to learn pattern weights based on playouts' results. We describe this approach in section 5.2 and explain why it cannot succeed. This is the first result of this chapter.

Linear model idea. MCTS tree stores playout knowledge in weights, one for each state. Generalization is accomplished by replacing such a representation with one that can generalize to new, unseen but similar states. This is accomplished with a use of machine learning algorithms, usually a linear regression model. This approach is very natural and it is well researched [BBK96].

In section 5.4 we show that linear model fails to converge even in very simple games. This is the second and an important result of this chapter.

A new model based on Combinatorial Games. Go board tends to decompose into several independent subgames. The linear model is unable to capture the essence of the sum of the subgames. The right model for the game of Go is a sum of Combinatorial Games [BCG82]. *Combinatorial Game Theory* has been applied to a game of two professional players [Spi02]. The analysis has detected many previously unnoticed mistakes in play, each worth enough points to change the winner of the game. This example clearly proves the usefulness of CGT.

The algorithm presented in the following sections is based on CGT. It can search through each subgame separately using playouts and find the correct strategy for a sum of games. This is the third result of this chapter.

Triangle approximation. In order to find the sum-of-games strategy, each of them needs to have its *thermograph* evaluated. To avoid complex and noise prone calculations, we propose a triangle approximation for the thermograph. This is the fourth result of this chapter.

Chapter outline. There are four results described in this chapter:

- Explanation of the failure of pattern-based naive adaptive playouts - section 5.2.
- Unsuitability of a linear model for certain two-player games - section 5.4.
- A triangle approximation for the thermographs - section 5.7.
- A new, Combinatorial Games based model and algorithm - section 5.8.

One other contribution is a simple explanation of the key concepts of Combinatorial Game Theory and Thermography in particular. It can be found in section 5.6. This section is partially based on author's publication [LC10].

5.1 A simple artificial problem

The algorithms are tested on a simple artificial sum of games. Such a sum is depicted in Figure 5.2. Figure 5.3 shows an equivalent Go position which can be separated into three completely independent areas. This is an extremely simple situation that is not representative of the complexity of the game of Go. But this situation is complex enough to be a challenge for adaptive-playout Monte-Carlo algorithms and to explain their failure.

For such a simple problem, it is easy to determine an optimal strategy. The total number of possible states in this sum of games is equal to the product of the number of states in each game, that is to say $3\times3\times5 = 45$. The principal variation of optimal strategy is: $(G+H+I) \to (G+H^L+I) \to (G+H^L+I^R) \to (G+H^L+I^{RL}) \to (G^R+H^L+I^{RL})$. This yields a score of $-2+4+0 = 2$ for Left.

5.2 Failure of adaptive patterns.

The *tree policy* is adaptive because it depends on the weights stored in the tree nodes which depend on results of previous playouts. The *playout policy* is fixed and it is based on

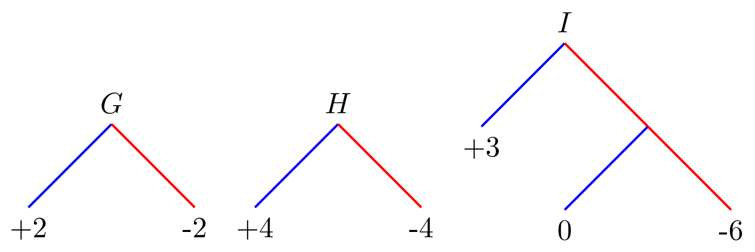


Figure 5.2: A simple sum of three combinatorial subgames (G, H, and I), each being represented by a tree. Two players, Left (Max) and Right (Min), alternatively make a move. The game starts in the root position of all subgames denoted $(G + H + I)$. A move consists in choosing a subgame and moving from its current position to the left or right child, depending on whose turn it is to play. It is not possible to play in a subgame whose current state is a leaf. The game is over when all the subgames have reached a leaf. The final score is the sum of leaf values. For instance, if Left starts by playing in subgame I, then he would reach the leaf $I^L = +3$. Then, Right could play in subgame H, and reach the leaf $H^R = -4$. The only move for Left would be then +2, in subgame G. The final score of this game would be $G^L + H^R + I^L = 2 - 4 + 3 = 1$.

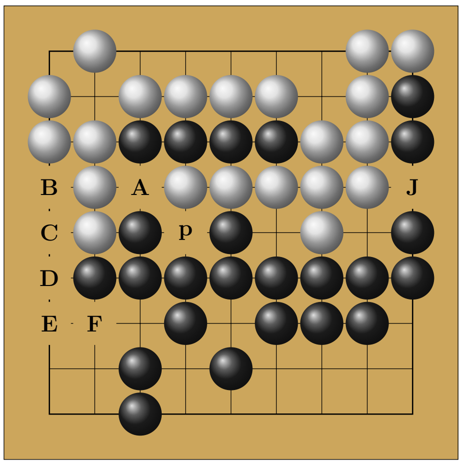


Figure 5.3: This Go board is similar to the abstract sum of games of Figure 5.2. Black is Left, and White is Right. Leaves of the abstract games correspond to the following move sequences: Leaf G^L in game G is ●J; G^R is ○J; H^L is ●A; H^R is ○A. Game I is a little more complicated. I^L is the sequence ●C-○B-●D. I^R is ○D-●E-○C. Leaf I^{RL} is reached after additional ●F. Leaf I^{RR} continues with ○F and all Black's corner territory is destroyed.

pattern and feature values, which leads to the horizon effect. The idea of adaptive patterns is to make the playout policy adaptive by changing the pattern and feature weights based on playout results. It is natural to try the same rules and equations as the ones used in MCTS.

Weights. MCTS node weight stores an average of all playouts' results that go through that state:

$$\mu_s = \mathbb{E}\,(\text{playout result} \mid \text{state } s \text{ was visited})$$

The adaptive pattern weight would be an average of all the playouts that had this pattern played (a pattern appeared and a move in the middle was played):

$$\mu_p = \mathbb{E}\,(\text{playout result} \mid \text{pattern } p \text{ was played})$$

Policies. In the tree phase there are several possible next-states. MCTS uses UCT policy (section 3.3.2) to focus on moves that lead to states with a high μ_s. Within the playout phase there are several possible patterns to play. The adaptive pattern can use the same policy to focus on the moves in the patterns with a high μ_p.

The failure. This and similar approaches were tried by many researchers. None of them published good results and many of them reported failures and expressed the surprise that it did not work on a computer-go mailing list. To my best knowledge, nobody offered an explanation why this approach does not work. I stumbled on this problem as well during a much more complex experiment (dynamic, growing decision trees in place of patterns). The setup described in this section is simpler and gives a better illustration of the inherent problem in this policy.

The explanation. In my experiment whenever a large group was captured, the policy chose to play many bad moves around that group. A closer inspection of the weights and playouts that led to them exposed a fundamental problem: Pattern weights *do not* represent value of the move in the middle. They represent value of the move *and* the value of the pattern. But the policy *cannot influence* the set of observed patterns, it can only influence the move choice.

An example explaining this problem is shown in Figure 5.4. It shows that comparing weight of one pattern to another is like comparing apples to oranges. In MCTS this comparison works thanks to the fact that all the compared weights share the same circumstances - a previous position.

5.3 Temporal-Difference Search

MCTS, as a global search algorithm, maintains a separate weight for each global state. In our simple example, this would mean having $3 \times 3 \times 5 = 45$ weights, one per each combination of subgames' states. It is clear that 45 weights are too many to completely represent such a simple and structured game and that most of the weights are somehow redundant.

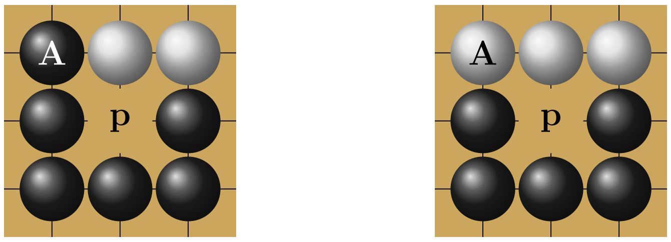


Figure 5.4: Patterns from position shown in Figure 5.3 around point p. Left side shows a situation where Black got to play move A. Right side shows a situation where White got to play move A. Black (White) move on A implies rescue (capture) of 4 black stones in Figure 5.3. The weight associated with the pattern on the left (right) is the average of all the playouts where A was played by Black (White) and p was played (a fixed player). Therefore the difference between weights of left and right patterns amounts to at least 8 points. Both weights should be estimating the value of playing move p (by a fixed player) in a given local circumstances. But in fact the weights are evaluating the circumstances themselves.

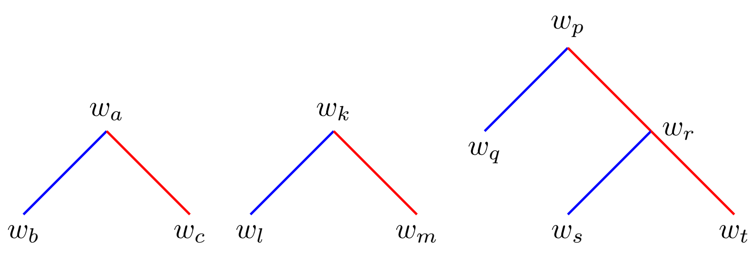


Figure 5.5: Linear model weights associated with each state in each subgame in example from Figure 5.2.

One approach to reducing the complexity of the sum of games is using a linear model. It consists in approximating value of each state by the sum of the subgames' values. In our simple example, this would mean estimating instead of 45 parameters, only $3+3+5 = 11$, one per each subgames' states. Figure 5.5 shows and names the 11 weights.

The complexity of this method for approximating the value function is equal to the sum of game complexities, instead of its product. When the number of games is high, this decomposition can produce a huge simplification.

To represent value of state s with weights $\vec{w}$ we use function $V(\vec{w}, s)$. For instance the value of the global root is:

$$V(\vec{w}, G + H + I) = w_a + w_k + w_p$$

The value of the final state from the principal variation is:

$$V(\vec{w}, G^R + H^L + I^{RL}) = w_c + w_l + w_s$$

Algorithm 2 Monte Carlo Policy Iteration

```
$\vec{w}_\pi \leftarrow \vec{w}$
loop
  for $n = 1$ to $N$ do

    {Make a playout using model weights $\vec{w}_\pi$.}
    $s \leftarrow$ initial state
    while $s$ is not terminal do
      if random() $< \epsilon$ then
        $a \leftarrow$ random action
      else
        {greedy action}
        $a \leftarrow \arg\max_a(V(\vec{w}_\pi, s.\text{play}(a)))$
      end if
      $s \leftarrow s.\text{play}(a)$
    end while

    {Update model weights $\vec{w}$.}
    for $s \in$ all visited states do
      $\vec{w} \leftarrow \vec{w} - \alpha(V(\vec{w}, s) - R)\frac{\partial V(\vec{w},s)}{\partial \vec{w}}$
    end for

  end for
  $\vec{w}_\pi \leftarrow \vec{w}$ {policy update}
end loop
```

Algorithm 3 TD(λ) Policy Iteration

```
$\vec{\theta}_\pi \leftarrow \vec{\theta}$
loop
  for $n = 1$ to $N$ do
    $\vec{e} \leftarrow 0$ {Clear eligibility traces}
    $s_0 \leftarrow$ initial state
    $i \leftarrow 0$
    while $s_i$ is not terminal do
      if random() $< \epsilon$ then
        $\vec{e} \leftarrow 0$
        $a_i \leftarrow$ random action
      else
        {greedy action}
        $\vec{e} \leftarrow \lambda \vec{e} + \partial V(\vec{\theta}, s_i)/\partial \vec{\theta}$
        $a_i \leftarrow \arg\max_a(V(\vec{\theta}_\pi, s_i.\text{play}(a)))$
      end if
      $s_{i+1} \leftarrow s_i.\text{play}(a_i)$
      $\delta \leftarrow V(\vec{\theta}, s_{i+1}) - V(\vec{\theta}, s_i)$
      $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
      $i \leftarrow i + 1$
    end while
  end for
  $\vec{\theta}_\pi \leftarrow \vec{\theta}$ {policy update}
end loop
```

Now the task is to create an algorithm that does the playouts based on $V(\vec{w}, s)$ and updates $\vec{w}$ based on the playouts' results. The tree policy in MCTS was based on the μ_s and n_s. Here we use $V(\vec{w}, s)$ in place of μ_s and we do not have n_s. n_s was used in exploration term so we switch to probabilistic epsilon-greedy exploration. This change is irrelevant to the results presented in this section.

The last piece is an updating equation analogous to equation 3.1 in section 3.3.2. This equation is equivalent to the following one:

$$\mu_s \leftarrow (1 - \alpha)\mu_s + \alpha R, \quad \alpha = \frac{1}{n_s + 1}$$

Since we do not have n_s we use a fixed α. The last obstacle is that we do not have μ_s but a linear combination of weights. The common solution in such circumstances is to update

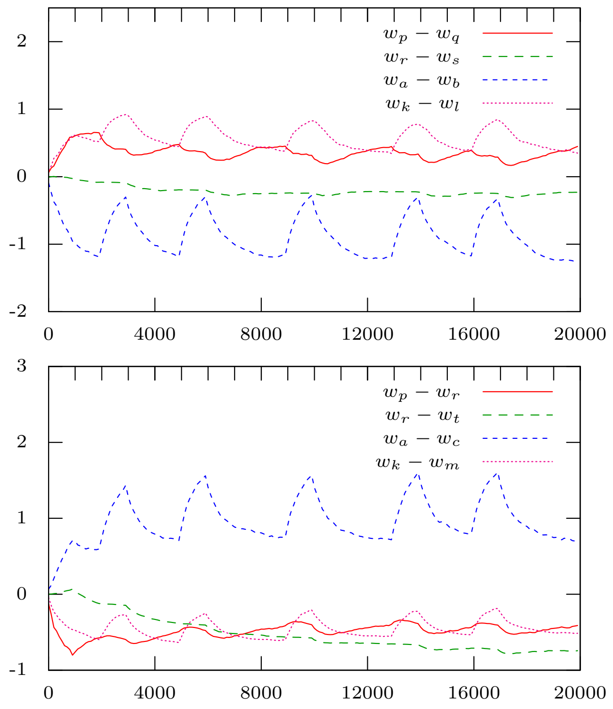


Figure 5.6: Change in value function (incentive) for different moves of Left (Right) player. Greedy policy chooses a move with the highest (lowest) value.

each weight proportionally to its influence on $V(\vec{w}, s)$. This is equivalent to the gradient descent algorithm and results in such an update equation:

$$\vec{w} \leftarrow \vec{w} - \alpha \cdot (V(\vec{w}, s) - R) \cdot \frac{\partial V(\vec{w}, s)}{\partial \vec{w}}$$

Please note that in our simple example $V(\vec{w}, s)$ is a linear combination of weights $\vec{w}$ with exactly 3 coefficients equal to 1 and all the rest equal 0. Therefore $\frac{\partial V(\vec{w},s)}{\partial \vec{w}}$ is a vector consisting of zeros and 3 ones.

This leads us to the Algorithm 2 - Monte Carlo Policy Iteration. This algorithm is equivalent to a Temporal Difference algorithm TD(1) which can be generalized to TD(λ). TD(λ) is shown as Algorithm 3. TD(λ) is a well researched algorithm with thousands of publications. It was also researched in the context of computer Go [CGJB$^+$08, GS07] resulting in a Go program called RLGO. RLGO can play without global search based only on the linear model and can still compete with classical programs. But RLGO is still much weaker than state of the art Monte-Carlo programs that use a static playout policy.

The author did his own experiments with linear model algorithms. Only a few of them managed to converge to optimal policy on such a simple example. In order to understand this problem we used parameter N larger than one. This effects in epochs of N playouts of learning with a fixed policy. This way for each policy occurring we can inspect closely what parameters are learnt and what new policy they produce.

Figure 5.6 shows the result of applying Algorithm 3 to the sum of combinatorial games. It shows the incentives - changes in value function after certain moves. Greedy policy always chooses a move with the highest incentive. This particular experiment used $N = 1000$, $\epsilon = 0.2$, $\alpha = 0.001$, $\lambda = 0.9$. Many other parameter values were tried, on a variety of combinatorial games, and oscillations were observed most of the time.

This demonstrates that Temporal-Difference Search fails to converge to any policy when state value is approximated by the sum of the values of independent games.

5.4 Explanation of linear model failure

A close inspection of the algorithm working on long epochs with a fixed policy revealed a fundamental problem. The problem is caused by using a linear model in place of one-weight-per-global-state MCTS approach. The main instance of the problem can be illustrated with this property: *The weights of the linear model learned with a fixed optimal policy necessarily produce a non-optimal policy even though the optimal policy is possible to represent.*

In our example the optimal policy leads to a following playout:

$$\begin{aligned}
&(G + H + I) \rightarrow \\
&(G + H^L + I) \rightarrow \\
&(G + H^L + I^R) \rightarrow \\
&(G + H^L + I^{RL}) \rightarrow \\
&(G^R + H^L + I^{RL}) \rightarrow \\
&R
\end{aligned}$$

In this linear model there are enough degrees of freedom (weights) to represent any policy that can be represented by some order on the moves of each player (from best to worst). This in particular includes the optimal policy. The value function learned using this optimal playout tries to make the value function constant on all the states of the playout and equal to the playout result.

$$\begin{aligned}
&V(\vec{w}, G + H + I) \approx R \\
&V(\vec{w}, G + H^L + I) \approx R \\
&V(\vec{w}, G + H^L + I^R) \approx R \\
&V(\vec{w}, G + H^L + I^{RL}) \approx R \\
&V(\vec{w}, G^R + H^L + I^{RL}) \approx R
\end{aligned}$$

There are enough degrees of freedom in the model so that the errors in the above approximations converge to zero. This is equivalent to the following limits:

$$\begin{aligned}
w_a + w_k + w_p &\longrightarrow R \\
w_k - w_l &\longrightarrow 0 \\
w_p - w_r &\longrightarrow 0 \\
w_r - w_s &\longrightarrow 0 \\
w_a - w_c &\longrightarrow 0
\end{aligned}$$

The above limit equations forces indifference between moves occurring on the principal variation and many other moves. For instance, in the global root position the algorithm converges to indifference between playing in G^R an playing in I^R:

$$V(\vec{w}, G^R + H + I) - V(\vec{w}, G + H + I) = w_a - w_c \longrightarrow 0$$

$$V(\vec{w}, G + H + I^R) - V(\vec{w}, G + H + I) = w_p - w_r \longrightarrow 0$$

Note that this indifference is independent of the state of subgame H.

To give another intuition, the linear model cannot represent the crucial ingredient of the policy: *the order in which the subgames are played out.*

There are many publications describing algorithms designed to prevent oscillations and force convergence $TD(\lambda)$ used with linear models. But in the author's opinion it is clear that the problem lies in the use of the linear model and not in the algorithm that trains it.

5.5 Combinatorial game theory

A classical value function assigns to each state only a single number – the expected reward. Such a function must depend on a policy and evaluates one policy only.

The previous section shows that even simple state spaces arising in Go game (or more generally in combinatorial games) may lead to oscillation problems in classical Temporal Difference Learning. One way to overcome this problem is to have the knowledge accumulated from simulated experience stored in a form independent of the policy used. To achieve this while simultaneously benefiting from a divide-and-conquer approach, we need a representation of subgames that is richer than a term in the linear model and can reflect the dynamics of the sum of subgames.

Combinatorial subgames can be represented accurately using *thermographs.* Next section explains what thermograph is and how it can be recursively built.

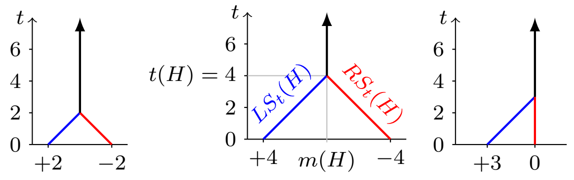


Figure 5.7: Thermographs of G, H, and I. Thermographs of *gote* games like G and H are symmetric triangles. Thermographs of *sente* games like I are asymmetric.

5.6 Thermographs

The simplest approximation to represent combinatorial game would be two numbers - left and right stop. Stops are the results of the game if Left or Right plays the first move and an alternating optimal play follows. For instance in Figure 5.2: $LS(I) = 3$ and $RS(I) = LS(I^R) = 0$ (I^R denotes subgame I after Right has played a move in it).

Stops of the subgames would be sufficient if after one player starts, his opponent would always respond in the same subgame or pass if there is no response. But in the *sum* of combinatorial games, it may happen that one player plays two or more moves in a row in the same subgame while his opponent plays elsewhere.

In order to model this we assume that there exists a dense *environment* consisting of many simple subgames and a player may choose to either play a move in a subgame G or to play it in the environment and get t points. This is equivalent to giving t points of *"tax"* to opponent with each move. A value t is called *ambient temperature. Taxed stops* $LS_t(G)$ and $RS_t(G)$ are the results of a taxed game under optimal play when Left or respectively Right player plays first.

If the tax is high enough, then neither player is willing to play first in the subgame (or equivalently, when the ambient temperature is high enough, both players prefer to play in environment.) They wait until the tax is low enough. Such point is called *temperature* of the subgame. It is denoted $t(G)$ and corresponds to notion of urgency of play.

When a current tax is equal to the temperature of subgame, then both stops are equal. This value is called mean of a subgame: $m(G) = LS_{t(G)}(G) = RS_{t(G)}(G)$. Because waiting for the tax to be low enough is optimal strategy, we have: $LS_t(G) = RS_t(G) = m(G)$ for all $t >= t(G)$.

A *thermograph* is a graphical plot of stop functions. On the horizontal axis is the result of the subgame with positive values on the left. On the vertical axis is tax (or ambient temperature).

It is easy to recursively compute thermographs. If G^L and G^R are sets of left and right options of G then:

$$LS_t(\{G^L|G^R\}) = \max(RS_t(G^L) - t, m(G)) \ , \tag{5.1}$$

$$RS_t(\{G^L|G^R\}) = \min(LS_t(G^R) + t, m(G)) \ , \tag{5.2}$$

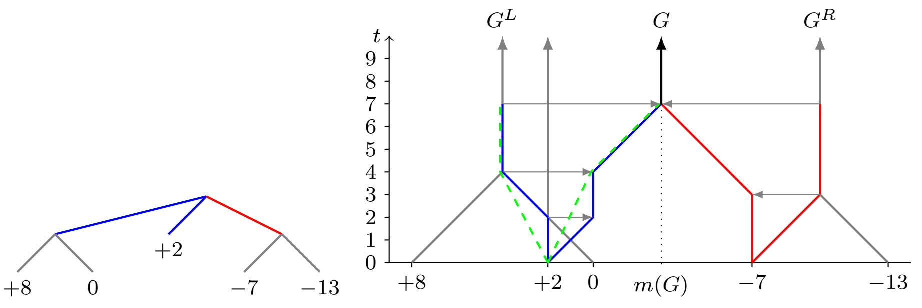


Figure 5.8: A simple combinatorial game and a recursive calculation of its thermograph. Green dashed line show an approximation used in CG-TD(0) algorithm. Horizontal gray arrow represent taxing.

and $(t(G), m(G))$ are coordinates of intersection of $RS_t(G^L) - t$ and $LS_t(G^R) + t$. In case one player has more than one option, the maximum in $\max(RS_t(G^L) - t, m(G))$ goes over right stops of all the options in G^L and $m(G)$.

An example of thermographs and process of recursive computation is graphically shown in Figure 5.8.

Given the thermographs of all the combinatorial subgames, one strategy that is close to the optimum is *HOTSTRAT*. HOTSTRAT advises to always play in the hottest subgame. Other strategies like SENTESTRAT or THERMOSTRAT differ only slightly and can be found in [BCG82].

[Caz02] gives empirical comparison of different strategies on simple combinatorial games.

5.7 Approximation of thermographs

Thermographs of big games can be quite complex - can have many corners. We propose a triangle approximation. Each thermograph is represented by four numbers only: temperature $t(G)$, mean $m(G)$, left stop and right stops $LS(G) = LS_0(G)$, $RS(G) = RS_0(G)$. Left and right taxed stops are assumed to be vertical *mast* above $(t(G), m(G))$ and straight lines below going through $(0, LS(G))$ and $(0, RS(G))$ respectively.

Given triangle-approximations of all options (children) it is straightforward to compute all four parameters needed for the triangle-approximation of the parent. Firstly, an approximation of right stop of all the left options must be done using Equation 5.1. With triangle-approximated thermographs it is sufficient to find $\max(RS(G^L))$, mean and temperature of a subgame with the highest mean.

In Figure 5.8 a triangle approximation of G^L (both left options in G) is marked with a green dashed line. A taxed triangle approximation is marked with green dashed line as well. Horizontal arrows represent the taxation used in recursive computation of parent's thermograph.

The same steps must be repeated for G^R. Intersection of two approximated stops yields mean and temperature of G needed for the triangle approximation of the parent. The stops of the parent are trivial to compute.

5.8 Combinatorial Games Temporal Difference Learning

Algorithm 4 CG-TD(0)

```
while not converged do
    $s \leftarrow (s[1], s[2], ..., s[N])$ {*sum of games*}
    while $\exists i : s[i]$ has options do
        $i \leftarrow \pi(s)$ {*choose subgame*}
        $G \leftarrow s[i]$
        $H \leftarrow \pi(G)$ {*choose option of G to play*}
        $s[i] \leftarrow H$ {*make the move*}
        UpdateThermograph($G$)
    end while
    UpdateLeaves($s$, Result($s$))
end while
```

Algorithm 5 UpdateLeaves(s, R)

```
$\delta \leftarrow R - \sum_{i=1}^{N} m(s[i])$ {*prediction error*}
for $i = 1$ to $N$ do
    $t(s[i]) \leftarrow 0.0$
    $m(s[i]) \leftarrow m(s[i]) - \frac{\alpha\delta}{N}$
    $LS(s[i]) \leftarrow m(s[i])$
    $RS(s[i]) \leftarrow m(s[i])$
end for
```

Algorithm 6 UpdateThermograph(G)

```
{*update direction based on left options*}
$L \leftarrow \arg\max_{L \in G^L}(m(L))$ {*highest mean*}
$lrs \leftarrow \max(RS(G^L))$ {*highest right stop*}
$rs \leftarrow m(G) - t(G)$
{*intersection with taxed mast*}
$t_1 \leftarrow \frac{m(L)-rs}{2}$
{*intersection with taxed right stop*}
$t_2 \leftarrow t(L)\frac{lrs-rs}{lrs-(m(L)-2 \cdot t(L))}$
$t' \leftarrow \min(t_1, t_2)$ {*first intersection*}
$m' \leftarrow rs + t'$ {*mean of first intersection*}

{*update the thermograph of G*}
$m(G) \leftarrow m(G) + \alpha(m' - m(G))$
$t(G) \leftarrow t(G) + \alpha(t' - t(G))$
$LS(G) \leftarrow t(G) + \alpha(lrs - LS(G))$

{*update direction based on right options*}
... {analogous code}

{*update the thermograph of G*}
... {analogous code}
```

In this section we propose a new algorithm - Combinatorial-Games Temporal-Difference Search (*CG-TD*). It replaces linear model with triangle-approximated thermographs. Similarly to $TD(0)$ algorithm, it modifies the thermograph in a state based on the thermographs in the state that is visited after it.

With each node of each subgame of the game sum algorithm maintains approximated thermograph represented with four numbers: mean, temperature, left and right stops. A

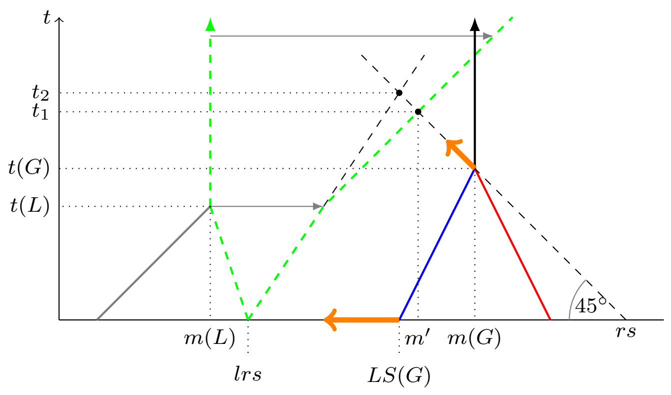


Figure 5.9: Illustration of UpdateThermograph(G). The left side of approximate thermograph G is moved in the direction of big arrows, in order to match the taxed thermograph of L.

state is a position in all the subgames' trees. The starting state consists of roots of all the subgames. A state is final if none of the subgames have any options (children).

The next state is chosen according to some policy π: firstly, the subgame is chosen then subgame's option. The state is then updated by replacing chosen subgame by its chosen option (child in the tree). At this point the triangle thermograph of subgame G is updated.

UpdateThermograph(G) procedure is best understood by looking at the Figure 5.9. The taxed line approximating all the left options should match left stop function of G subgame $LS_t(G)$. If it does not, the algorithm adjust left stop $LS(G)$, mean $m(G)$ and temperature $t(G)$ in the direction of correct position on the taxed line. The amount of adjustment is controlled by the learning rate. Next, similar update is done for the right options (not shown). The exact equations are given in Algorithm 6.

Leaf nodes are updated using classical gradient descent to match the result. They converge to correct (up to a global constant) values as long as policy visits enough of variety of final states.

This algorithm converges to correct triangle-thermographs in all the nodes if the policy ensures that all the nodes are visited infinitely many times. This is the only constraint for the policy used. Otherwise it should be beneficial to tune the policy for a fast convergence. A policy similar to UCT should do well.

Figure 5.10 shows very fast convergence of the temperatures of subgames in our toy example to the correct values. HOTSTRAT strategy (play in the hottest subgame) is optimal not only for the principal variation but it is optimal for both players in all the 45 possible configurations of subgames' states.

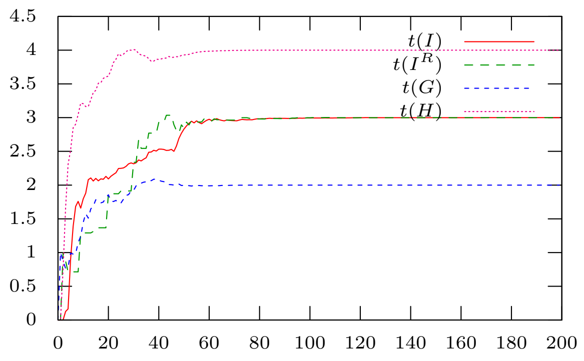


Figure 5.10: Temperatures

5.9 Summary

We have shown that it is not trivial to create adaptive playout algorithm for the game of Go. We presented two natural approaches for adaptive playouts and explained why they cannot work well. The insights provided there point at the model as the source of the problems. A new model was proposed based on Combinatorial Games which model game of Go very well. A triangle approximation was proposed to simplify the computations needed in evaluation of thermographs. A Temporal Difference algorithm based on the new model was shown to converge very quickly to the right values providing optimal strategy on a toy example that was challenging for preexisting algorithms. This algorithm, when scaled to large trees, should lead to a correct evaluation of Go positions with many independent subgames like the one presented in Figure 5.1. It would cover the main weakness of state of the art playout based Go programs.

Applicability of the algorithm and future work. The proposed algorithm is not directly applicable to Go game, since it relies on an explicit separation of the game state into independent combinatorial subgames. It also only shows a correct principle - the thermograph estimation, that can serve to build a large scale algorithm capable of searching large complicated subgames. Design of the policy π is crucial for an efficient searching of the subgames.

It is also interesting to design and test the performance of RAVE-like heuristics within $CG - TD$ framework. In case of Go, this algorithm can be further extended to cover zugzwang situations where neither player wants to play first. This would allow players to pass when temperature of all subgames is zero or negative, and finish game early. This would cover situations like *seki* in the game of Go.

Chapter 6

Libego - The Library of Effective Go Routines

The Library of Effective Go Routines [Lew11] is a framework for performing experiments in the computer Go domain. It was created to improve author's understanding of computer Go algorithms and to facilitate experiments with new algorithms. The source of Libego was published on GPL licence. It was used in few formal and several informal research projects.

In the next section we describe motivation for creating a new library. Section 6.2 presents some features of Libego. Section 6.3 describes uses of Libego library known to the author.

6.1 Motivation

Before the development of Libego started, the author had tried to use one of the existing Go board (Go rules) implementations. The most promising library was Gnu Go's board implementation. It turned out to be designed with classical Go algorithms in mind. Gnu Go implementation is too slow for the needs of new Monte-Carlo Go algorithms, the interface was too complicated and it contained too much of a legacy code. It turned out that it is simpler to create an efficient implementation from scratch than to use Gnu Go's board. The performance of a Monte-Carlo Go program is crucial as it needs as many simulated games as possible to gather knowledge. The strength of the program scales quite well with the number of simulations allocated to it.

Libego started as an efficient implementation of Go game rules as possible. The task is not trivial and specific details of implementation were discussed many times on computer-go list. Later, more advanced algorithms were build on the top of the library. As Libego grew bigger some design decisions proved to be inadequate, increasing the complexity of the code. A lot of refactoring was applied to Libego to make it as simple and easy to use for other researchers as possible.

6.2 Features

Major features of Libego are:

- A very efficient implementation of game logic, move sampler and a whole playout algorithm.
- Various Monte Carlo Go algorithms, such as: AMAF, MCTS, RAVE, patterns and features in heavy playouts.
- Standard Go engine based on Go Text Protocol (GTP).
- Distributed testing framework allowing to make tests on computer clusters provided by MiM UW department and Grid5k cluster.
- Scripts for testing the engine on Computer Go Server (CGOS) and KGS Go Server (KGS).
- Graphics user interface implemented using Qt GUI framework with various Go visualizations.
- Minorization-Maximization and LMP algorithms for training the Bradley-Terry model on pattern data.

Finally, Libego became a quite complex piece of software. A large part of its complexity comes from supporting features such as GUI, testing scripts, data parsing etc. A lot of effort was made to make it as easy as possible to use. That could be a factor in making it popular among other researches and hobbyists.

6.3 Usage of Libego in third party research

Academic use of Libego. There are several publications that use Libego.

- Flynn et al. [Fly08a, Fly08b] experimented with Libego extended with 5×5 patterns engine for heavy playouts.
- Helmbold et al. [HPW09] created and tested several more powerful variants of AMAF heuristic. They were implemented and tested within Libego library.
- Anderson [And09] used Libego to test Apprenticeship Learning, Policy Gradient Reinforcement Learning and Policy Gradient Simulation Balancing on a Go domain.
- Childs, Brodeur et al. [CBK08, Bro08] implemented and tested two algorithms using Libego. They tested effectiveness of Transposition Tables that turn the game tree into a game DAG to save computations and Move Groups to more effectively explore the game tree.

- Takeuchi et al. [ST08, ST10] created an algorithm for fast evaluation of the engine strength based on database of games of good players. They performed their experiments in four games: Chess, Go, Othello and Shogi. Libego was used as one of Go implementations.

Non-academic use of Libego. When asked on computer-go mailing list, several people declared that they used Libego or that Libego influenced their own implementations.

- Heikki Levanto performed several experiments using Libego.
- Peter Drake reimplemented Libego in Java and used in Orego program.
- Petr Baudis created a strong engine Pachi. Libego's implementation of Go rules inspired Pachi's but no source code was reused.
- Aja Huang, creator of top Go program Erica declared "*Erica is also benefitted a lot from Libego, your implementation and ideas of speed optimization.*".
- Christian Nentwich declared that Libego influenced his CUDA implementation of Go (engine on a graphics card).
- Lars Schäfers redesigned his program Gomorra based on Libego implementation.
- Jason House reused implementation of several algorithms from Libego in his Go engine.
- René van de Veerdonk created BitmapGo – implementation of Go rules based on bitmask arithmetic – declared that his design was influenced by Libego. Libego was used as a reference benchmark in his research on fast Go rules implementations.

Chapter 7

Conclusion

This thesis focuses on new, promising Go playing algorithm - Monte Carlo Go. It identifies key elements of Monte Carlo Go framework. One of them, the expert knowledge, is usually extracted from game records of good players by decomposing each decision to a set of available patterns. We described a new learning algorithm that is an online variant of the state of the art batch algorithm. It overcomes the memory limitation and allows to learn from arbitrarily large data set. We also proved the new algorithm always to converge - a rare result among online learning algorithms. A future work should concentrate on applying the described method to other models applicable in other domains and compare its performance experimentally on standard data set recognized by Machine Learning community.

This work also identifies the main weakness of Monte Carlo Go - globality of the tree search. Monte Carlo Go is terribly inefficient when the Go position decomposes into multiple independent subgames (which is usually the case on 19×19 board). Adaptive playouts and other local algorithms are considered crucial research topic but it is very hard to get right algorithms. This thesis identifies and describes reasons why natural algorithms do not work. It presents a divide and conquer algorithm based on combinatorial game theory that overcomes the identified problems on toy examples. It relies on a decomposition being known. Adapting this algorithm to a large scale real game problem should include decomposition search and a UCB-like exploration and is an important followup research.

The state of the art Go programs can play on small 9×9 Go board with a strength of a professional player and on a full 19×19 board the same programs are “only” on a low dan level. Based on that fact we judge that exploiting decomposability of big Go positions is the key to bringing 9×9 strength to 19×19 board.

Bibliography

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, May 2002.

[ACBFI00] Peter Auer, Nicolò Cesa-Bianchi, Paul Fischer, and Lehrstuhl Informatik. Finite-time Analysis of the Multi-armed Bandit Problem, 2000.

[AChH90] Thomas Anantharaman, Murray S. Campbell, and Feng hsiung Hsu. Singular extensions : Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99 – 109, 1990.

[And09] David A. Anderson. Monte Carlo Search in Games, 2009.

[BBK96] Steven J. Bradtke, Andrew G. Barto, and Pack Kaelbling. Linear least-squares algorithms for temporal difference learning. In *Machine Learning*, pages 22–33, 1996.

[BC01] Bruno Bouzy and Tristan Cazenave. Computer Go: an AI oriented survey, 2001.

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*, volume I and II. Academic Press, 1982.

[BGB05] Daniel Bump, Farneback Gunnar, and Arend Bayer. GnuGo, 2005.

[BH03] Bruno Bouzy and Bernard Helmstetter. Monte Carlo Go developments. In Ernst, editor, *Advances in Computer Games conference (ACG-10), Graz 2003*, pages 159–174. Kluwer, 2003.

[Boo91] Mark Boon. Overzicht van de ontwikkeling van een Go spelend programma. Master's thesis, University Amsterdam, 1991.

[Bou03] Bruno Bouzy. Mathematical morphology applied to computer go. *IJPRAI*, 17(2), 2003.

[Bou04] Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In *4rd Computer and Games Conference, Ramat-Gan*, 2004.

[Bou05] Bruno Bouzy. Move pruning techniques for Monte-Carlo Go. In *11th Advances in Computer Game conference, Taipei*, 2005.

[Bro08] James H. Brodeur. Random Search Algorithms, 2008.

[Bru75] Bruce. Perception and representation of spatial relations in a program for playing Go. In *ACM Annual Conference*, pages 37–41, 1975.

[Bru93] Bernd Bruegmann. Monte Carlo Go, 1993.

[BSB11] S.R.K Branavan, David Silver, and Regina Barzilay. Non-Linear Monte-Carlo Search in Civilization II, 2011.

[Caz02] Tristan Cazenave. Comparative evaluation of strategies based on the values of direct threats. In *Board Games in Academia V, Barcelona*, 2002.

[CBK08] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search. In Philip Hingston and Luigi Barone, editors, *IEEE Symposium on Computational Intelligence and Games*, pages 389–395. IEEE, December 2008.

[CGJB$^+$08] Louis Chatriot, Sylvain Gelly, Hoock Jean-Baptiste, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combining expert, offline, transient and online knowledge in monte-carlo exploration, 2008.

[CH05] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and Monte-Carlo in the game of Go. In *IEEE CIG 2005*, 2005.

[CHhH02] Murray Campbell, A. Joseph Hoane, and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57 – 83, 2002.

[Col04] Mark Colyvan. The philosophical significance of Cox's theorem. *International Journal of Approximate Reasoning*, 37:71–85, 2004.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In Jaap H. van den Herik, Paolo Ciancarini, and (jeroen) H. H. L. M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630/2007 of *Lecture Notes in Computer Science*, pages 72–83, Turin, Italy, June 2006. Springer.

[Cou07a] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, December 2007.

[Cou07b] Remi Coulom. Monte-Carlo Tree Search in Crazy Stone, 2007.

[CWvdH$^+$07] Guillaume Chaslot, Mark Winands, Jaap H. van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*, 2007.

[dMRVP09] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 729–736, New York, NY, USA, 2009. ACM.

[FB08] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, pages 259–264. AAAI Press, 2008.

[Fly08a] J. Flynn. SlugGo Spring 08 Report: Using Patterns in Libego Playouts, 2008.

[Fly08b] J. Flynn. SlugGo Winter 08 Report: Using 5x5 Tiles with Libego, 2008.

[Fot92] David Fotland. Many Faces of Go. 1st Cannes/Sophia-Antipolis Go Research Day, Februar 1992.

[Fot93] David Fotland. Knowledge representation in The Many Faces of Go, 1993.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference on Machine Learning, ICML 2007*, 2007.

[GS10] Romaric Gaudel and Michèle Sebag. Feature Selection as a One-Player Game, 2010.

[GW06] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go, December 2006.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[HPW09] David P. Helmbold and Aleatha Parker-Wood. All-Moves-As-First Heuristics in Monte-Carlo Go, 2009.

[Hun04] R. Hunter. MM algorithms for generalized Bradley-Terry models. *The Annals of Statistics*, 32:2004, 2004.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, FebApr 1975.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML-06*, 2006.

[LC10] Lukasz Lew and Remi Coulom. Simulation-based Search of Combinatorial Games. In *ICML 2010 Workshop on Machine Learning and Games, Israel*, 2010.

[Lew11] Lukasz Lew. Libego library. `https://github.com/lukaszlew/libego`, 2011.

[Mue97] Martin Mueller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In *Game Programming Workshop in Japan '97, MATSUBARA, H. (ed.), Computer Shogi Association, Tokyo, Japan*, 1997.

[Mü10] M. Müller. Challenges in Monte Carlo Tree Search, 2010.

[Niu04] Xiaozhen Niu. Recognizing safe territories and stones in computer Go. Master's thesis, Department of Computing Science, University of Alberta, 2004.

[NM06] Xiaozhen Niu and Martin Müller. An open boundary safety-of-territory solver for the game of Go. In *5th Conference on Computer and Games, CG2006*, 2006.

[RKN$^+$75] Walter Reitman, James Kerwin, Robert Nado, Judith Reitman, and Bruce Wilcox. Goals and plans in a program for playing Go. In *ACM Annual Conference*, 1975.

[RST09] P. Rolet, M. Sebag, and O. Teytaud. Upper Confidence Trees and Billiards for Optimal Active Learning, 2009.

[RTC11] Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows, 2011.

[SB98] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning I: Introduction, 1998.

[SCS09] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo Tree Search in Settlers of Catan. In *ACG*, pages 21–32, 2009.

[SGHM07] Scott Sanner, Thore Graepel, Ralf Herbrich, and Tom Minka. Learning CRFs with hierarchical features: An application to Go. In *Proceedings of the Workshop on Constrained Optimization and Structured Output Spaces (at ICML-07)*, 2007.

[Spi02] William L. Spight. Go Thermography: The 4/21/98 Jiang-Rui Endgame WILLIAM L., 2002.

[ST08] K. Yamaguchi S. Takeuchi, T. Kaneko. Evaluation of Monte Carlo tree search and the application to Go, 2008.

[ST10] K. Yamaguchi S. Takeuchi, T. Kaneko. Evaluation of Game Tree Search Methods by Game Records, 2010.

[SWH$^+$08] Maarten P. D. Schadd, Mark H. M. Win, H. Jaap Van Den Herik, Guillaume M. J b. Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *In Computers and Games, volume 5131 of Lecture Notes in Computer Science*, pages 1–12. Springer, 2008.

[vdWvdHU04] E. C. D. van der Werf, H. J. van den Herik, and J. W. H. M. Uiterwijk. Learning to estimate potential territory in the game of go. In *4th International Conference on Computers and Games (CG'04)*, LNCS. Springer-Verlag, 2004.

[Zob71] Albert L. Zobrist. Complex preprocessing for pattern recognition. In *ACM annual conference 1971*, 1971.