"Al. I. Cuza" University
Faculty of Computer Science

Bachelor's Degree Thesis

Artificial Intelligence in Computer Go

by

Florin Chelaru

Supervisors:
Liviu Ciortuz, Ovidiu Gheorghies

Iași, June 2008

Contents

Contents **i**

1 Motivation and goals **1**
1.1 Introduction 1
1.2 Heuristics for Monte Carlo Go 2
1.2.1 Upper Confidence for Trees (UCT) 3
1.2.2 All-moves-as-first 4
1.2.3 Rapid Action Value Estimation (RAVE) 4
1.2.4 Grandparent knowledge 5
1.3 The Analyze-after approach to random simulations 5
1.4 Integration of Monte Carlo with GNU Go 7
1.4.1 GNU Go engine overview 7
1.4.2 Adding a Monte Carlo module to GNU Go 7

2 Preliminaries **9**
2.1 The game of Go 9
2.1.1 Basic rules 9
2.1.2 Important consequences 12
2.1.3 Ranks and ratings 13
2.2 Computer Go as a field of Artificial Intelligence 14

3 Computer Go **17**
3.1 General overview 17
3.1.1 How does a good state look like? 17
3.1.2 The size of the territory 18
3.1.3 Tactical information (example: weak points) 18
3.1.4 The *expected outcome* of the game 19
3.2 Old-fashioned Computer Go vs Monte Carlo Go 19
3.2.1 Old-fashioned Computer Go 19
3.2.2 Basic Monte Carlo Go 21

4 Combining Old-fashioned Go with MC Go 23
4.1 Basic ideas and goals . . . 23
4.2 General data structures . . . 23
4.2.1 Array List . . . 23
4.2.2 Heap Array . . . 25
4.2.3 Visited List . . . 26
4.3 Theoretic base and enhancements to Monte Carlo Go . . . 28
4.3.1 The Random Simulation . . . 28
4.3.1.1 The *naïve* approach . . . 28
4.3.1.2 The Common Fate Graph approach . . . 33
4.3.1.3 The Analyze-after approach . . . 35
4.3.2 The Multi-armed Bandit Problem . . . 40
4.3.3 Upper Confidence for Trees . . . 41
4.3.4 Key Positions Priority . . . 46
4.3.4.1 Creating a distribution using Bézier curves . . . 47
4.3.5 First-play urgency . . . 49
4.3.6 Learning from past experience . . . 49
4.3.6.1 Grandparent knowledge . . . 50
4.3.6.2 The Experience Tree . . . 51
4.3.7 All-moves-as-first . . . 53
4.3.8 Rapid Action Value Estimates . . . 54
4.4 Adding Monte Carlo to GNU Go . . . 55
4.4.1 GNU Go engine overview . . . 55
4.4.1.1 Gathering information . . . 55
4.4.1.2 Move Generators . . . 56
4.4.1.3 Move Valuation . . . 57
4.4.2 Adding a Monte Carlo module to GNU Go . . . 58

5 Results and conclusions 61
5.1 The random simulation . . . 61
5.2 Monte Carlo GNU Go versus GNU Go 3.6 . . . 62

Bibliography 67

List of Figures 73

List of Tables 75

Index 77

Chapter 1

Motivation and goals

Throughout our work, we discuss the idea of combining old-fashioned Computer Go with Monte Carlo Go. We explain our approach to adding a Monte Carlo module to the GNU Go 3.6 engine. We also briefly present the features of our present Monte Carlo with UCT implementation. We then discuss an analyze-after approach to random simulations, and finally show some preliminary results of the entire work, ideas for future work and conclusions.

1.1 Introduction

Go is an ancient Chinese game, its first historic references dating around the 4th century B.C. In spite of its apparently simple rules[1], its combinatorial complexity is high, making it a challenge to create a strong engine for playing the game. As a result, the best Computer Go programs reach only average human level of performance.

Although, at a first glance, other board games, like Chess seam to be harder and more complex, the game of Go raises above them, having both the branching factor and the game length significantly larger. In other words, the average number of legal positions in an arbitrary state of Go ranges from slightly below 100, for the 9×9 board to a few hundred, in the case of the 19×19 board, while the maximum number of possible choices in Chess barely surpasses a few dozens. Moreover, the problem of finding an evaluation function is a difficult one, thus making it hard for programmers to use the classic artificial intelligence techniques like alpha-beta search in order to find good strategies.

[1] Visit Sensei's library (http://senseis.xmp.net/) for a detailed description of the game.

Important efforts have been made for the purpose of creating a good Computer Go playing program, which are commonly separated into two general categories: Old-fashioned Computer Go, encapsulating classic artificial intelligence techniques and strategies, and Monte Carlo Go, a recent approach based on probabilities and heuristics.

First studies and research in old-fashioned Go started about forty years ago[2] focusing mostly on state representation, breaking down the game in goal-oriented sub-games, local searches and local results, functions for evaluation and determining influence and base knowledge for pattern matching [6]. There are several programs using this approach, of which one of the best and the only one with available sources and documentation is GNU Go.

Since the beginning of 1990's[3], and more intensely within the last 10 years, the attention has moved over some probabilistic approaches, the most important being Monte Carlo Go. Monte Carlo is a simple algorithm, based on *approximating the expected outcome of the game*. At each step, before generating a move, the program launches a number of random simulations, starting with each available move, which are evaluated. The move with the best average score is picked as the best move and played. Used along with various heuristics, it turned out to give birth to impressive results. On 9×9 boards, the *standard deviation* of the random games is approximated to 35. For a one point precision evaluation, 1000 games give 68% statistical confidence, while 4000 games 95%. Present CPU's are able to compute about $10,000$ random simulations per second, which means that the method works in reasonable time and with enough statistical confidence [4].

Our focus falls upon three aspects. First of all, the most important Monte Carlo heuristics which have already proven to be efficient. Secondly, finding a fast enough algorithm for the Monte Carlo random simulation, so it becomes suitable for the 19×19 Go board, and thirdly, combining the two approaches by adding a Monte Carlo module to the engine of GNU Go.

1.2 Heuristics for Monte Carlo Go

Being a relatively new approach, research is still open on the subject of Monte Carlo applied to Go. Still, several heuristics have already shown themselves to give positive results, of which some more than others. In this section we discuss about the most

[2] The first Go program was written by Albert Zobrist in 1968 as part of his thesis on pattern recognition [22].

[3] The general Monte Carlo model was advanced by Bruce D. Abramson, who used it on games of low complexity, such as 6x6 Otello [4]. In 1993, Bernd Brügmann created the first 9×9 MC Go program, Gobble [10].

important of them, which either we have implemented or we consider that should be implemented in a future version of our work.

1.2.1 Upper Confidence for Trees (UCT)

UCT represents an algorithm based on the *Multi-armed Bandit* problem[4]. The problem basically refers to the *Exploration versus Exploitation* dilemma, which consists of searching for a balance between exploring the environment to find profitable actions and taking the empirically best action as often as possible.

Formally, a K-armed bandit is represented by random variables $X_{i,i=\overline{1,K}}$, where each i is the index of a gambling machine, which on successive plays yields rewards $X_{i_1}, X_{i_2}, \ldots$. The rewards are independent and identically distributed according to an unknown law with unknown expectation μ_i. Independence also holds for rewards across machines; i.e., X_{i_s} and X_{j_t} are independent for each $1 \leq i < j \leq K$ and each $s, t \geq 1$. The purpose of the gambler is to find a strategy of maximizing his winnings.

It has been proven that, choosing at each step the machine which maximizes the following formula (called UCB1-TUNED, or simply UCB1) ensures the play of the overall best machine exponentially more often than the others:

$$\overline{X}_j + \sqrt{\frac{\ln n}{T_j(n)} \cdot \min\left\{0.25, V_j\Big(T_j(n)\Big)\right\}}, \tag{1.1}$$

where:

- $T_j(n)$ is the number of times machine i has been played after the first n plays,
- $\overline{X}_{i,s} = \frac{1}{s}\sum_{t=1}^{s} X_{i_t}$; $\overline{X}_i = \overline{X}_{i,T_i(n)}$ and
- $V_j(s) = \left(\frac{1}{s}\sum_{t=1}^{s} X_{jt}^2\right) - \overline{X}_{j,s}^2 + \sqrt{\frac{2\ln n}{s}}$ (an estimated upper bound of the variance of machine j).

UCT consists of treating every node in the Monte Carlo tree as a bandit problem, all move choices representing the machines. At first, for each available move, a random simulation is launched. Then, at each step, the algorithm chooses the one for which (1.1) has the greatest value[5] and plays the next random game starting with it. The rewards may be 1, if the game ended in favor of the color playing the first move or 0 otherwise.

[4]See [1] for a detailed description of the problem and the solution we use in our implementation of UCT.

[5]See [13] for an extended presentation of UCT.

1.2.2 All-moves-as-first

This heuristic has an important word to say for the Monte Carlo Go playing engines, since it allows the process of evaluating a move to divide the response time by the size of the board. The idea is simple: after a random game with a certain score, instead of just updating the mean of the first move of the random game, the heuristic updates all moves played first on their intersections with the same color as the first move. It also updates with the opposite score the means of the moves played first on their intersections with a different color from the first move [8].

Basically, this updates the means of almost all moves in the game. Of course, the heuristic isn't entirely correct, since various moves may have different effects on the game depending on the time they were played. Still, the speedup is worth taking into consideration.

1.2.3 Rapid Action Value Estimation (RAVE)

RAVE[6] is a heuristic used for generalizing the value of a move across all positions in the subtree below a certain Go state. It is closely related to the all-moves-as-first idea. Moreover, it provides a way of sharing experience between classes of related positions. The strategy uses the following formula:

$$\hat{Q}(s,a) = \frac{1}{\hat{n}(s,a)} \sum_{i=1}^{N} \hat{I}_i(s,a) z_i, \tag{1.2}$$

where:

- z_i is the outcome of the ith simulation: $z = 1$ if the game was won and $z = 0$ otherwise;
- $\hat{I}_i(s,a)$ yields 1 if position s was encountered at any step k of the ith game, and move a was selected at any step $t \geq k$, or 0 otherwise;
- $\hat{n}(s,a) = \sum_{i=1}^{N} \hat{I}_i(s,a)$ counts the total number of simulations used to estimate the RAVE value; and
- $\hat{Q}(s,a)$ represents the average outcome of all simulations where move a was selected in the position s, or in any subsequent position.

The idea is to use (1.2), which is biased but with lower variance at the beginning of the game, and gradually shift to the classic Monte Carlo value, unbiased and with higher variance.

[6] A detailed description of RAVE and integration within Monte Carlo and UCT is provided in [12].

1.2.4 Grandparent knowledge

Grandparent knowledge is an approach related to the two previous ones. It basically uses the fact that moves at the same intersection, which are close in time, have close influence over the outcome of the game. Let s be the current state and g be its grandparent node. Then, at the time g was the current state, Monte Carlo explored the uncles of s, i.e. the alternative moves to the parent of s, the one eventually chosen. The idea is to combine the values of the sons of s with the values of its uncles whenever s becomes the current node. Although the results of this technique are not spectacular, we consider it worth mentioning.

1.3 The Analyze-after approach to random simulations

Our random simulation idea, which behaved the best in our tests, was to just play, with no concern whatsoever for the rules, except that of not filling friendly eyes. Then, when all the board is full, reanalyze the game and decide which moves were illegal, which stones were captured and which territory belongs to whom. In order to present this approach, we will formalize the concepts in the game of Go, so it is easier to see the solution.

Let N the size of the board. Then $\mathcal{B}$, the set of intersections and $\mathcal{C}$, the set of colors, can be written as:

$$\mathcal{B} = \Big\{(x,y) \ \Big| \ 1 \le x, y \le N\Big\} \text{ and } \mathcal{C} = \{black, white\}.$$

We introduce a general concept of game:

$$\mathrm{G}_{k,k\ge 1} = \Big\{g_k \ \Big| \ g_k : \{1,\dots,k\} \to \mathcal{B} \times \mathcal{C}\Big\}$$

G_k is the set of all games with k moves.

We aim to find a way of knowing the state of the board at any point of the game. For that, we first define time, $\mathrm{time}_{g_k} : \mathcal{B} \to \{1,\dots,k\}$, as

$$\mathrm{time}_{g_k}(p) = \begin{cases} \min\Big\{j \le k \ \Big| \ \exists c \in \mathcal{C} : g_k(j) = (p,c)\Big\} \text{ or} \\ \infty \text{ , if no such j as above exists,} \end{cases}$$

which represents the first moment during g_k when the intersection p was occupied by a stone of color c.

A first notion of board state, $\hat{\mathrm{b}}_{g_k} : \mathcal{B} \to \mathcal{C} \cup \{empty\}$ (one with no rules) is defined below:

$$\hat{\mathrm{b}}_{g_k}(p) = \begin{cases} c \in \mathcal{C} & \text{, if } g_k(\mathrm{time}_{g_k}(p)) = (p,c) \\ empty & \text{, otherwise.} \end{cases}$$

Now, to introduce rules, we start with $N_{p,p=(x,y)}$ the set of neighbors for intersection p:

$$\mathrm{N}_p = \mathcal{B} \cap \Big\{(x+1,y),(x,y+1),(x-1,y),(x,y-1)\Big\}.$$

The notion of *worm*, $w_{g_k,p} \subseteq \mathcal{B}$, is defined inductively:

Base

$$w_{g_k,p} = \begin{cases} \{p\} & \text{, if } \hat{\mathrm{b}}_{g_k}(p) \in \mathcal{C} \\ \emptyset & \text{, otherwise.} \end{cases}$$

Inductive step

$$\text{Let } p' \in w_{g_k,p}; \quad \left.\begin{array}{l} p'' \in \mathrm{N}_{p'} \\ \hat{\mathrm{b}}_{g_k}(p'') = \hat{\mathrm{b}}_{g_k}(p') \end{array}\right\} \Rightarrow p'' \in w_{g_k,p}.$$

Moreover the worm liberties, $\mathrm{L}(w_{g_k,p}) \subseteq \mathcal{B}$, are written as:

$$\mathrm{L}(w_{g_k,p}) = \Big\{p'' \in \mathcal{B} \;\Big|\; \exists p' \in w_{g_k,p} \;:\; p'' \in \mathrm{N}_{p'} \wedge \hat{\mathrm{b}}_{g_k}(p'') = \mathit{empty}\Big\}.$$

In order to obtain the board, we use $\mathrm{newness} : \mathcal{G} \times \mathcal{P}(\mathcal{B}) \to \mathbb{N}$ giving the time the latest stone of the worm was played:

$$\mathrm{newness}(g_k, w) = \max\Big\{t \le k \;\Big|\; \exists p \in w : t = \mathrm{time}_{g_k}(p)\Big\}.$$

Finally, the state of a board, based on a game, $\mathbf{b}_{g_k} : \mathcal{B} \to \mathcal{C} \cup \{\mathit{empty}\}$, is defined inductively as:

Base

$$(\forall p \in \mathcal{B})\ \hat{\mathrm{b}}_{g_k}(p) = \mathit{empty} \Rightarrow \mathbf{b}_{g_k}(p) = \mathit{empty}$$

Inductive step

$$\text{Let } w_{g_k,p} = \operatorname*{argmin}_{w_{g_k,p}} \Big\{\mathrm{newness}(w_{g_k,p}) \Big| \exists p' \in w_{g_k,p} \;: \mathbf{b}_{g_k}(p') \text{ undefined}\Big\}.$$

$$\text{Then, } \forall p' \in w_{g_k,p},\ \mathbf{b}_{g_k}(p') = \begin{cases} \mathit{empty} & \text{, if } L(w_{k,p}) = \emptyset \\ \hat{\mathrm{b}}_{g_k}(p') & \text{, otherwise.} \end{cases}.$$

This result tells us that, having a sequence of completely arbitrary moves, we can generate a valid Go board state, as if the rules had been followed from the beginning. In other words, if we just placed stones on the board until there is literally no place to move, there would be a way to extract a correct Go endgame board, having clearly delimited territories, and the resulted board would be easy to evaluate.

The complexity of this approach is $O(|\mathcal{B}| + |\mathcal{W}| \log |\mathcal{W}|)$, where $O(|\mathcal{B}|)$ is the time required by filling the board and extracting the worms after and $O(|\mathcal{W}| \log |\mathcal{W}|)$ is the time needed to sort the worms by their newness.

1.4 Integration of Monte Carlo with GNU Go

1.4.1 GNU Go engine overview

GNU Go starts by trying to get a good understanding of the current board position. Using the information found in this first phase, and using additional move generators, a list of candidate moves is generated. Finally, each of the candidate moves is valued according to its territorial value (including captures or life-and-death effects), and possible strategic effects (such as strengthening a weak group).

Although GNU Go does a lot of reading to analyze possible captures, life and death of groups etc., it does not have a full-board lookahead and this is the main point where improvements can be made[7].

1.4.2 Adding a Monte Carlo module to GNU Go

We try to solve this latter problem of GNU Go, by adding the Monte Carlo module. The functionality of this module is as follows.

A separate thread runs random simulations during opponent time, exploring the UCT tree. Whenever a move is to be generated, the thread pauses, waiting for GNU Go's engine to generate a list of moves. Each of the moves has associated reasons summing up to a value estimating how good it is. After the list is generated, Monte Carlo resumes for a given amount of time, so that the confidence of its evaluation is good enough. Then again, it pauses. At this point, every available move has an associated winning probability. The next step is intuitive. For every move with positive score in the list generated by GNU Go we take its value and multiply it by its winning probability. This way, moves considered good both by GNU Go and Monte Carlo are automatically chosen. Moreover, moves estimated by GNU Go to have the same local influence, which in fact have different global importance in the game are overall ranked accordingly by Monte Carlo. Also, inherent errors which appear in Monte Carlo-only applications, due to lack of local precision, are eliminated thanks to GNU Go's *old-fashioned* Computer Go approach.

[7] For a full description of the GNU Go engine, visit the GNU Go documentation page: http://www.gnu.org/software/gnugo/gnugo_toc.html

Chapter 2

Preliminaries

2.1 The game of Go

Go[1] is an ancient Chinese board game, its first historic references dating some time before the 4th century BC. Its original name, *Wéiqí*, which basically means *board game of surrounding* [17], is rarely used in the western part of the world. Here, the game is known as *Go*, the Japanese translation of *Wéiqí*. This is due to the fact that early western players learned the game from Japanese sources. Along with the name, most of Go concepts and terms have also become known through their Japanese names.

2.1.1 Basic rules

There are several sets of rules for the game, of which the Chinese and the Japanese ones are the most popular[2]. However, the differences are minor, not changing the game in what tactics or strategies are concerned.

The game consists of two players, *Black* and *White*, alternatively placing *stones*[3] of their own color at the intersections of a square board. The standard board size is 19×19, but there are also other popular sizes, like 9×9 or 13×13. Unlike most of

[1] See [24] for a more detailed description of the game - history, rules, tactics, strategy and more.

[2] Throughout this paper we will refer only to the Japanese set of rules.

[3] Game piece.

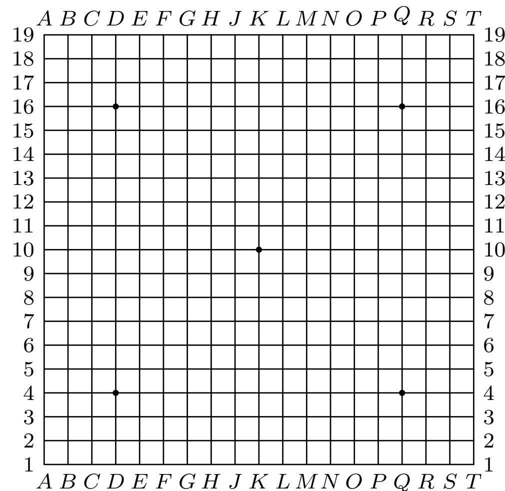

Figure 2.1: The board of Go

board games, *Black* starts. Because he moves second, *White* receives a compensation called *komi*[4]. The komi usually has fractional values, to prevent *jigo*[5].

The objective of the game is to control a larger part of the board. To achieve this, each player tries to place his stones in such a way that they cannot be captured. A stone (or a *chain*[6]) can be captured if it is completely surrounded by enemy stones[7]. Each player's final score consists of the number of empty intersections completely surrounded by his color added to the number of captured enemy stones. The other empty intersections are considered neutral and are not counted.

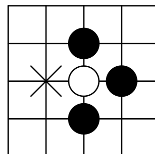

Figure 2.2: A stone in atari

In Go stones can be placed almost everywhere on the board. Still, there are two kinds

[4]A predetermined number of points added to the score of *White* at the end of the game. [28] According to professional Go players, the *komi* should be set to 5.5, not depending on the board size. However, the most popular value is 6.5.

[5]The result of a game where *Black* and *White* have an equal score, i.e., e drawn game. [28]

[6]Also called *string* or *worm*, a chain represents a group of game pieces of the same color, directly connected on the board.

[7]The empty intersections around a chain are called *liberties*. When a chain has only one liberty left, it is said to be in *atari*. If the chain loses all its liberties, it is removed from the board (*captured*).

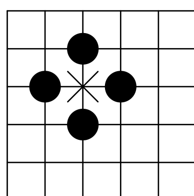

Figure 2.3: A suicide situation.

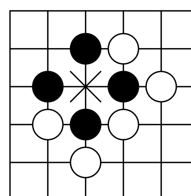

Figure 2.4: A *false* suicide move.

of moves considered illegal. Take, for instance, the situation illustrated in Figure 2.3. If *Black* moved at the marked intersection, his stone would have no liberty. That situation is called *suicide* and is forbidden. Notice however, in Figure 2.4, that if *White* placed his stone at the marked intersection, he would capture two black stones. Thus, in this case the move would prove not to be suicide.

The other illegal move situation, called *Ko*[8], is a move which produces an already seen situation on the board. See the example in Figure 2.5: if *White* places his stone in the marked intersection and *Black* moves back to $D4$, we reach the exact same board configuration as we had at the beginning. As a consequence, *Black* can only move to $D4$ after he places a piece somewhere else on the board. This way, the old board position is not repeated.

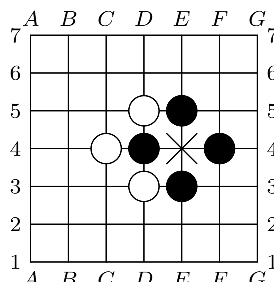

Figure 2.5: A *Ko* situation

The game finishes when both players pass. Beginners pass when there is nowhere else to move, except filling their own *eyes*[9]. Experienced players however, pass long before all legal moves have ran out. In Figure 2.6 we have a board configuration which allows the game to go on, and yet two advanced players would call it a game. Supposing one of the players contested the other's territory and the game went on, then all his *invading* stones would end up being captured (Figure 2.7).

[8]The Japanese word for *eternity*.

[9]An eye is an empty intersection on the board surrounded by friendly stones where the enemy cannot move, due to the suicide rule.

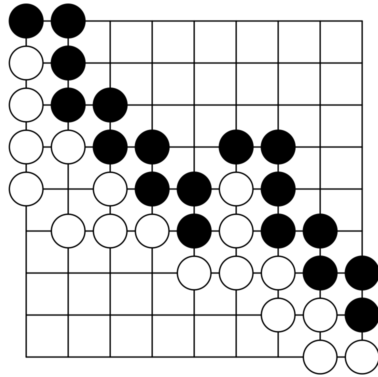

Figure 2.6: A situation in which two advanced players pass.

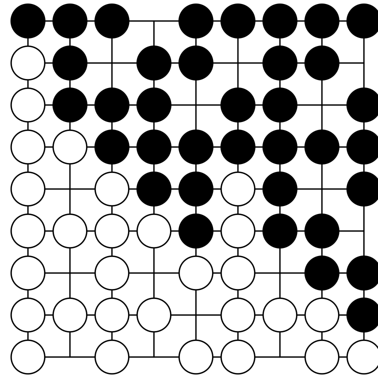

Figure 2.7: If the game went on, it would finally end up in a state like this.

2.1.2 Important consequences

Although the rules of the game themselves are few, they imply important consequences which can be regarded as *derived rules*.

The most important of these consequences are the concepts of *life* and *death*. A group of stones which has no possibility of connecting with other friendly stones can be classified as *alive* or *dead*.

A group of stones is considered to be *alive* if it cannot be captured even if the opponent is allowed to move first. Conversely, a group of stones is said to be dead if the owner cannot avoid capture even if he is allowed to make the first move. If the status of the stones depends on who makes the next move, the group is called *unsettled*. The player who makes the first move either *kills* it, or *makes it alive*.

For a group of stones to be *alive*, it needs to have at least two *eyes*. This means that there exists one stone within the group which has at least two liberties. The opponent won't be able to move inside, due to the *suicide rule*, and thus, he can never capture it (or the stones in the same chain). Only one eye is not enough for a group to be *alive*, since the enemy *can* eventually place a stone inside, taking the last liberty of the group (see Figure 2.4 for a single eye capturing situation).

There is one exception where a group of stones is also considered to be *alive*. When two or more groups of stones share the same liberties, it may come to a situation where moving first would allow the opponent to capture. Thus, the groups remain on the board. Such situations are called *seki*[10].

[10] Japanese word, meaning *mutual life*.

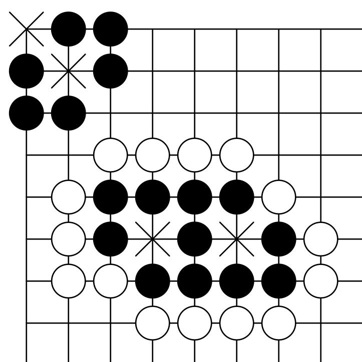

Figure 2.8: An example of *alive* groups (*eyes* are marked with X).

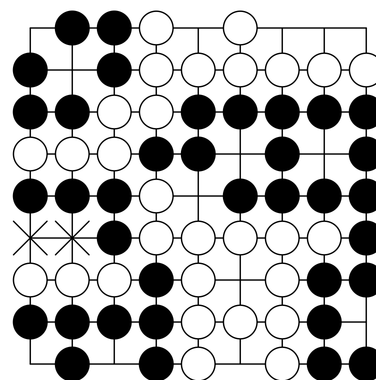

Figure 2.9: An example of *seki* (the common liberties are marked with X).

2.1.3 Ranks and ratings

In Go, players are ranked similarly to martial arts (since the Go ranking system has the same origin), by *kyu* and *dans*. Kyu grades (abbreviated *k*) are distinctions for Go students and amateurs. Dan grades are for advanced amateurs and professionals. When playing together, two players of different ranks play using *handicap stones*[11]. The number of handicap stones played is equal to the rank difference between the two players. For instance if a $5k$ plays a game with a $1k$, the $5k$ would need a handicap of four stones.

Here is a list of all rankings (lowest to highest):

Table 2.1: Go rankings (lowest to highest)[24]

Rank Type	**Range**	**Skill Level**
Double-digit *kyu* (*geup* in Korean)	$30 - 20k$	Beginner
Double-digit *kyu*	$19 - 10k$	Casual player
Single-digit *kyu*	$9 - 1k$	Intermediate amateur
Amateur *dan*	$1 - 7d$ (where $8d$ is special title)	Advanced amateur
Professional *dan*	$1 - 9p$ (where $10p$ is special title)	Professional player

[11]Handicap stones are stones placed on the board before the game starts, to even the odds of winning for both players.

2.2 Computer Go as a field of Artificial Intelligence

Computer Go is the term used to denote the part of *artificial intelligence*[12] dedicated to creating a computer program which plays Go [22][13].

In spite of legends and movies with robots fighting against mankind, in the world of the early 21st century computer intelligence is still in its youth. In fact, *artificial intelligence* in the proper sense hasn't yet been born. The field of Computer Science called artificial intelligence is about knowledge insertion and learning algorithms, lacking one of men's most important qualities, *intuition*.

Depending on the complexity of the problem, knowledge can be injected into a program in various ways. For example, in a *simple* game, like *Tic-Tac-Toe*, the winning strategy may be *hard coded*[14]. However, as problems grow more complex, new strategies need to be developed. Computer science has witnessed important AI developments. One by one, various board games - of which *Chess*[15] - became no longer obstacles in front of AI[16]. One step further, Computer Go comes into discussion, as its combinatorial complexity is exponentially larger than of any other board game.

The traditional AI methods used in Go require more speed and memory than any present computer has to offer[17]. Although having simple rules, Go brings a challenge to computer science, as no program has yet been created, which ranks higher than an average human player. On a short analysis, taking into consideration the *branching factor*, B, and *game length*, L, here is an estimation of the combinatorial complexity of Go [6]:

$$\begin{cases} B \approx 200 \\ L \approx 200 \end{cases} \Rightarrow B^L \approx 10^{400} \gg 10^{123}(Chess) > 10^{58}(Otello) > 10^{32}(Checkers)$$

This makes *global tree search*[18] *intractable* and non terminal position evaluation *hard*.

[12] Abbreviated AI.

[13] See, for further reference, [7].

[14] This means that the developer will write code for every particular case which may occur during the game.

[15] In May 1997, IBM's *Deep Blue Supercomputer* defeated the world champion Garry Kasparov, using deep tactical tree search. [2]

[16] In 1994, *Chinook* beat Marion Tinsley at *Checkers* and in 1998, *Logistello* defeated the best human at *Otello* [6].

[17] It is estimated that, for a game on a regular Go board of size 19×19, there are about $2.081 \cdot 10^{170}$ legal game positions [27].

[18] Global tree search consists of iterating through all possibilities of play and identifying the best one available.

Thus, when approaching architecture design of Computer GO, one should look for a way of injecting intuition and learning capacity.

When learning to play a game, a human constantly asks himself - consciously or unconsciously - questions like *What should I move next? Would this be a good move? What would I achieve by making this move?* Beginners usually can't see many moves in advance and most often, *just make the best apparent move available*. But along with experience and practice, they *learn* to recognize similar moves played before, to choose those which previously turned out good and avoid the bad ones. Also, they learn to predict more or less favorable situations a few moves in advance.

In the present work, we'll try to accomplish an artificial intelligence capable of doing just what we do, *learn to play GO*.

Chapter 3

Computer Go

3.1 General overview

3.1.1 How does a good state look like?

There are several ways to describe a good state. A competitive AI could take into account as many of them as possible. Ideally, there would exist a way of completely describing a good board configuration, and how it may be compared to others. However, although there are several unanimous accepted opinions, most features which make a state better or worse are subjective and may differ, depending on the player and his goals regarding the game[1]. We will, in the following, enumerate some of these features, speaking a few words about each.

Here are some of a good state's main characteristics[2]:

- The size of the territory.
- Tactical information (for example, *weak points*).
- The *expected outcome* of the game.
- The distance up to a good/bad state.
- The evaluation of professional players (base knowledge).

[1] For instance, a player may weigh a particular configuration of the board depending on whether at the end of the game he wins or loses, while some other evaluates it as better if at the end he owns larger territory than expected.

[2] These will be taken into consideration later on, when talking about algorithms.

3.1.2 The size of the territory

This is probably the most intuitive measure of a board configuration. Any Go player should be able to look at a board and say "it seams like you're winning" or "you're lost". Moreover, without such a measure one can't tell if the game is over, and if it is over, what the final score was.

It is easy to notice that this parameter has a strong word to say at the end of the game, when the possibilities of gaining over the opponent are fewer and the outcome is more predictable (there are fewer chances of any turn of situation, and thus, the evaluation is safer). On the other hand, at the beginning of the game, the size of the territory may make all the difference in choosing one state over another.

In consequence, when looking for some winning path in the moves tree, one may try to find those which get close to large territory end-states, while when trying to find the next move at the beginning of the game, he'll choose one which gains more territory.

However, apart from the present, this evaluation doesn't give us any information about what is yet to come. In Go, turns of situation occur very often, and a greedy strategy has no winning odds.

3.1.3 Tactical information (example: weak points)

A weak point is a point where the territory can be *cut*. Take, for instance, the situation in Figure 3.1.

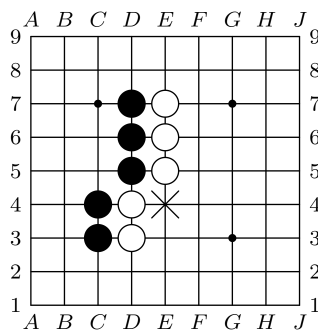

Figure 3.1: A cutting point situation

$E4$ is a weak point for *white*. A dangerous move is *cutting* the enemy's territory by moving to his weak point. To illustrate the importance of weak points, it is enough to assume, in the previous example, that it is black's turn. Although white owns

more territory, black's move at $E4$ leads, eventually, to winning the game. If it is white's turn, then any move elsewhere than at $E4$ becomes fatal[3].

3.1.4 The expected outcome of the game

The *expected outcome* is defined as the expected value[4] of game's outcome given random play from that node on[4].[5]

The expected outcome is used in *Monte-Carlo* heuristics to help predict what may happen within the next moves. If we accept that a good state leads, in general, to other good states, in other words, that good states are interdependent, then we can imagine that in the game tree there are whole clusters of similarly (good) nodes. Which means that in those clusters, there is a great probability of *accidentally* making a good move. We concentrate on this idea in the latter part of our work. For now, we just enumerate two of the important aspects of the expected value.

The first is straightforward: it gives a glance of the result of the game. If, after a few moves the expected outcome increased, it means we're on the right track.

Second is that if the expected outcome of the next few moves is good, it is likely that the current state is good too, and also, chances are that the next move will lead us to another good state.

It is, thus, fair to say that the expected outcome gives a somewhat good feel of *how good a state is*.

3.2 Old-fashioned Computer Go vs Monte Carlo Go

3.2.1 Old-fashioned Computer Go

The first noted research in the field of Computer Go dates almost fifty years ago[6]. Since then, there has been work in several areas of AI involving the game, including board state representation[7], pattern matching, tree search, automatic generation of knowledge, and the list goes on.

[3] Notice that, in this case, if white chooses to extend territory instead of defending the weak points, he loses the game.
[4] The expected value of a discrete random variable (not. $E(X)$) is the sum of the probability of each possible outcome of the experiment multiplied by the outcome value (or payoff).[23]
[5] See Sections 3.2.2 and 4.3 for an in-depth use of the expected value concept.
[6] Lefkovits, 1960 [6].
[7] Zobrist hashing, 1969 [6]; Mathematical morphology [5]

When talking about *old-fashioned* Computer Go, we refer to the algorithms discovered and researched up to the middle of 1990's. Up to that point, the interest and effort was concentrated onto injecting as much knowledge as possible inside the Go playing engine.

The main factors of decision for evaluating a move were based on *influence*[8], evaluating the status of groups[9] and knowledge bases[10], in other words, territory and tactical information.

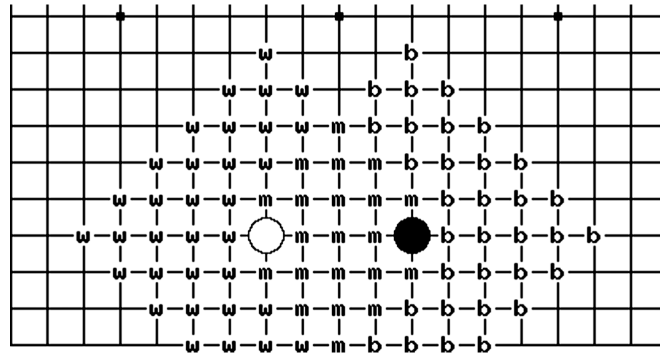

Figure 3.2: Influence on the board of Go. w=white influence, b=black influence, m=mutual influence [16]

The underlining idea was to *break the game into goal-oriented sub-games*[11] *and the board state into sub-states*. Every one of the sub-problems would be assumed and treated independently. For each move, there would be generated several reasons, each having a certain score, for and against playing it. In the end, the move with the highest summed score would be chosen as the next move.

The upsides of these techniques were numerous of which we'll mention just three. First of all, the computation time for each move generation would be rather small. Secondly, in most cases, solving the sum of all sub-problems would come close to solving the problem itself. Thirdly, on local situations we would get very much accuracy.

The results are indisputable, raising old-fashioned Computer Go programs in the top of Go-playing AI's.

Still, with the advantages come a series of disadvantages, one of them being that neither of the breaking stages developed so far are proven to be 100% correct. That is,

[8]Influence is closely related to territory. Basically, each stone on the board has a potential to help other stones. This potential is called influence. [17]

[9]Is the group *alive*, *dead* or *unsettled*.

[10]Which means *pattern matching* over small regions of the board.

[11]String capture, connections, dividers, eyes, life and death, etc.

in part, because sub-games are hardly ever independent. Also, because the algorithms are based on domain-dependent knowledge, any potential new situation becomes impossible to identify. Results aside, implementations also tend to be laborious and time consuming.

3.2.2 Basic Monte Carlo Go

Since the beginning of 1990's[12], and more intensely within the last 10 years, the attention has moved over some probabilistic approaches, the most important being Monte Carlo Go.

Monte Carlo is a simple algorithm, based on *approximating the expected outcome of the game*. At each step, before generating a move, the program launches a number of random simulations, starting with each available move, which are evaluated. The move with the best average score is picked as the best move and played.

The idea itself is intuitive and not at all spectacular. However, used along with various heuristics, it turned out to give birth to impressive results. On 9×9 boards, the *standard deviation* of the random games is approximated to 35. For a one point precision evaluation, $1,000$ games give 68% statistical confidence, while $4,000$ games 95%. Present CPU's are able to compute about $10,000$ random simulations per second, which means that the method works in reasonable time and with enough statistical confidence. [4]

There are quite several ways of improving the speed of Monte Carlo, which we will discuss in Chapter 4.

[12] The general Monte Carlo model was put forward by Bruce D. Abramson, who used it on games of low complexity, such as 6x6 Otello [4]. In 1993, Bernd Brügmann created the first 9×9 MC Go program, Gobble [10].

Chapter 4

Combining Old-fashioned Go with MC Go

4.1 Basic ideas and goals

We aim to use both the capacities of old-fashioned Computer Go and the advantages given by the probabilistic approach offered by Monte Carlo. To accomplish this, we use the engine of GNU Go 3.6, to which we have added a Monte Carlo module. Each move considered by GNU Go has an associated winning probability, which helps deciding upon the best move. The observed results were made on the 19×19 board, which is the one that brings the toughest challenge to Computer Go.

4.2 General data structures

Since GNU Go's implementation is entirely ANSI C based, we decided to go on the same approach, preserving the coding style used so far and using the existent data structures and methods as much as possible. We added, however, a series of new structures which we knew to be useful from the experience of working with higher level and object-oriented languages.

4.2.1 Array List

The Array List, as the name suggests, is a data structure combining the advantages of an array, basically related to indexing, and those of a list, related to having flexible

size according to the needs of the developer. In other words, the Array List looks to set a balance between speed (retrieving an item fast) and space (using no more memory than needed). Particularly, it behaves as an indexed stack.

Figure 4.1 shows the structure as we designed it, where:

```
1 struct array_list
2 {
3     void* inner_list;
4     int count;
5     int capacity;
6     size_t element_size;
7     int (*compare)(const void *, const void *);
8 };
9
10 typedef struct array_list* array_list_t;
```

Figure 4.1: The Array List structure

- `inner_list` is used for holding the actual items of the array.
- `count` is the number of elements in the array, initialized to `0`.
- `capacity` represents the total number of items which can be stored inside the array.
- `element_size` stores the size in bytes of an item, and finally,
- `compare` is a function which compares two items of the array (returning `0` if they are considered equal, `1` if the first is greater than the second and `-1` otherwise).

Initially, the capacity is set to 2. Whenever an attempt is made to add an item to the list, if the capacity doesn't allow it (the inner list is full), then we double the memory allocated for the elements of the list. After that, the item is just pushed at the end of the list:

```
1 /* Returns index of the element added */
2 int
3 array_list_add(array_list_t list, void* element)
4 {
5     if (list->count == list->capacity)
6     {
7         list->capacity *= 2;
```

```
 8          list->inner_list = realloc(
 9                                  list->inner_list,
10                                  list->capacity * list->element_size);
11      }
12
13      memcpy(list->inner_list + list->count * list->element_size,
14             element,
15             list->element_size);
16
17      list->count++;
18
19      return list->count-1;
20  }
```

Let's analyze the complexity of this approach. If $|add(n)|$ is the number of operations required to add a number of n elements to the list, then

$$|add(n)| = c(n + \sum_{i=1}^{\lceil \log_2^n \rceil} 2^i) = c(n + 2^{\lceil \log_2^n \rceil + 1} - 1) < 5c \cdot n, \text{ where } c \text{ is constant.}$$

Thus, the time complexity remains within the same order as that of the stack, $O(n)$.

Also, if $size(n)$ is the size used by an array list containing n elements, then

$$size(n) = \texttt{element_size} \cdot 2^{\lceil \log_2^n \rceil} < \texttt{element_size} \cdot 2n,$$

which means that the size complexity is also within $O(n)$. Obviously, the items are retrieved in $O(1)$ time.

4.2.2 Heap Array

A *heap* is a binary complete tree in which the parent nodes are always *better*[1] then their descendants. It is used in situations where the *best* item retrieval is required in minimum time.

We combined the Array List with the notion of heap, resulting into this new simple structure.

---

[1]Depending on the items stored, the notions of *better* and *worse* may vary. For example, we may consider the best number in an array, the one having the greatest value. So the higher number, the *better*.

```
1 struct heap_array
2 {
3     array_list_t inner_list;
4     int (*better_than)(const void *, const void *);
5 };
6
7 typedef struct heap_array *heap_array_t;
```

Figure 4.2: The Heap Array structure

Every node is an element in the array at index `i`, having its descendants at `2i + 1` and `2i + 2`.[2]

The complexity of retrieving the *best* node is, evidently $O(1)$. Inserting and removing have both complexities of $O(\log n)$, where n is the number of nodes in the heap.

4.2.3 Visited List

The Visited List represents a bit array used to keep track of whether a certain number has been processed before. For example, when building information about worms, we iterate from one stone to another through its neighbors. It is desirable that we don't visit a stone twice, since we want to make as few operations and use as little memory as possible. For that purpose, we use a Visited List, which keeps a bit of `1` for each visited stone. Whenever we meet a new neighbor, we first check whether it has been visited before, and if it has, it is skipped.

In Figure 4.3 we have a glance of the base structure, where:

- `min` represents the value of the minimum value we keep track of. Conversely,
- `max` is the maximum value kept
- `mid` is the first item in the `right_list`
- `left_list` and `right_list` are bit arrays storing the data.

We can look at this structure as a union of two sets: $[\texttt{min}, \texttt{mid}) \cup [\texttt{mid}, \texttt{max}]$, the first one being stored by `left_list`, while the second one by `right_list`.

[2] For more information about heaps and algorithm design see [18] [25].

```
1  struct visited_list
2  {
3      int mid;
4      int min;
5      int max;
6      unsigned int *left_list;
7      unsigned int *right_list;
8  };
9
10 typedef struct visited_list *visited_list_t;
```

Figure 4.3: The Visited List structure

Here too, we aim to balance speed and memory. Initially, both `left_list` and `right_list` are $\emptyset$ (`NULL`). When the first number is visited, `mid` takes its value and the `right_list` is initialized, allocating 4 bytes (32 bits). So `max` becomes $\texttt{mid}+32-1$ (the Visited List becomes $\emptyset \cup [\texttt{mid}, \texttt{mid}+31]$). Whenever a new value is added, if it is less than `min` or greater than `max`, then the corresponding set is reallocated so that it may store the new value. The values are stored by setting their corresponding a bit. Figure 4.4 shows an example of the algorithm for the case when a value smaller than `min` is added to the list.

```
1  void visited_list_set(int x) {
2      int dword_size = sizeof(int); // 4
3      int bits_per_dword = dword_size * 8; // 32
4
5      if (x < mid) {
6          if (x < min) {
7              int nr_bits = log_2^(mid-x-1) + 1;
8              int nr_dwords_left = max(1, 2^(nr_bits) / bits_per_word);
9
10             left_list = realloc(nr_dwords_left*dword_size);
11
12             min = mid - nr_dwords_left*bits_per_dword;
13         }
14
15         bit = mid-x-1;
16         SET_BIT(left_list, bit);
17     }
18     ...
19 }
```

Figure 4.4: Adding a value smaller than `min` to the Visited List

Let $|set(n)|$ be the maximum number of operations needed to store n numbers into the Visited List. Then[3]

$$|set(n)| = n + \sum_{i=1}^{\lceil\delta\rceil} = n + \frac{\lceil\delta\rceil(\lceil\delta\rceil + 1)}{2}, \text{ where } \delta = \min\{\frac{\texttt{max} - \texttt{min} + 1}{32}, n\}.$$

On the average, for a relatively compact set of values, the time complexity falls within $O(n)$. Also, the maximum amount of memory needed to store any number n of values is 4δ bytes.

On the particular case where we used the Visited List, which is storing board intersections, $1 \leq \texttt{min}$ and $\texttt{max} \leq 361$. The above formula becomes

$$|set(n)| = n + \frac{\lceil\delta\rceil(\lceil\delta\rceil + 1)}{2} \leq n + \min\{78, \frac{n(n+1)}{2}\}.$$

4.3 Theoretic base and enhancements to Monte Carlo Go

In order to be able to use Monte Carlo on a 19×19 board and to expect some results, we first need to focus on enhancing the algorithm by coming up with speed optimizations and scoring heuristics.

We will break the current section into subsections concentrated on each aspect considered in our work.

4.3.1 The Random Simulation

The purpose of the random simulation is to play a game with legal moves avoiding to fill friendly eyes. All worms should be removed from board when captured and the score evaluation, at the end of the game, should be accurate.

We will talk about three approaches, presenting advantages and disadvantages of each, and concluding with results.

4.3.1.1 The naïve approach

The naïve approach consists of two steps: initialization and iteration.

In the initialization step, we create a list of all empty intersections on the board. They are all, at first, considered legal potential moves.

[3] We consider the number of bits in an `int` to be 32.

```
1     array_list_t moves = CREATE_INT_ARRAY();
2
3     for (i=0; i<board_size; i++)
4         for (j=0; j<board_size; j++)
5             if (board[POS(i, j)] == EMPTY) {
6                 array_list_add(moves, POS(i, j));
7             }
```

Figure 4.5: Initialization in the naïve random simulation approach

Then, in the iteration step, we extract at random, from the list of available moves, one at a time. If the move is legal, does not fill any eyes and doesn't place any friendly worm into atari, then it is played and the game goes on. If the move doesn't fulfill all these requirements, it is disposed of and another one is selected. Whenever a worm is captured, all its intersections are added to the list of available moves, since they become empty. The process finishes when there are no available moves left in the list.

```
1     int pos, n_passes = 0;
2
3     while (moves->count > 0 && n_passes < 2)
4     {
5         pos = play_move(color, moves);
6
7         if (pos != PASS_MOVE) {
8             n_passes = 0;
9         } else n_passes++;
10
11        color = OTHER_COLOR(color);
12    }
```

Figure 4.6: Iteration step in the naïve random simulation approach

The implementation of this approach tends to get harder than its basic idea.

First of all, for the purpose of not altering the real state of the game, i.e. the board, the number of captured stones of each color, etc., we need to store all this information inside an auxiliary structure[4].

Also, in order to know, at each moment in the iteration, whether the move is legal or not, it captures any worms, puts friendly worms into atari, fills any friendly

[4]In our implementation, we used an existent structure in the GNU Go engine, called `board_state`.

```
1 struct board_state {
2   int board_size;
3   float komi;
4   int black_captured;
5   int white_captured;
6
7   Intersection board[BOARDSIZE];
8   ...
9 };
```

Figure 4.7: The board state structure

```
1 struct worm_data
2 {
3   int origin;
4   int liberties;
5 };
```

Figure 4.8: The worm data structure

eye, we need an additional data structure, containing basic information about worms, i.e. the *origin*[5] and *liberties*. We called it `worm_data`.

The move selection algorithm expands into the following sequence of steps:

```
1 play_move(
2   board_state* state, colors color,
3   array_list_t moves, worm_data worms[])
4
5 while (!found_legal_move && moves->count > 0)
6 {
7     move = array_list_pop_random(moves);
8
9     if (move has no stones around) {
10        found_legal_move = true;
11    }
12
13    for (worm in adiacent enemy worms) {
```

If the move captures a worm, it is most likely legal (the only case where it weren't would be if the move were *ko*).

```
14        if (worms[worm].liberties == 0) {
15            found_legal_move = true;
16            capture(worm);
17            update(board);
18        }
19    }
20
```

[5]The origin of a worm represents the top left-most stone belonging to it.

```
21     if (!found_legal_move) {
22         if ((the stone is atari, or has no liberty) ||
23           (the stone is placed inside an eye))
24             continue;
25     } else {
```

The current stone, when placed, may connect two or more friendly strings, resulting into a single worm. Before we may decide whether the move is legal or not, we should first see if this worm loses its last liberty by this move. If it isn't then the move is considered safe.

```
26         worm =
27           connect_adiacent_frienly_worms_with_stone();
28
29         if (worm.liberties != 0) {
30             found_legal_move = true;
31         } else {
32             // suicide
33             continue;
34         }
35     }
36 }
37
38 if (found_legal_move) {
39     state->board[move] = color;
40     update(worms);
41 }
```

This routine runs on an average, about 300 times per game, so we'll analyze the number of operations performed, to eventually find out how many random simulations we may get in one second. We will use the following notations and assumptions:

- *rand* - selecting a random move (line 7). This yields constant time, about 3-10 operations[6].
- *neighbors* - iteration through neighbors to see their color, also constant time consuming, yielding about another 10 operations.

[6]Depending, however, on the chosen random number generation algorithm.

- *capture* - capturing a worm and updating the board accordingly (lines 16-17). This comes to a number of $c \cdot |worm|$ operations, where $c >= 10$ and $|worm|$ is the size of $worm$.

- *connect* - connecting the corresponding friendly worms (lines 26-27), coming to $\sum_{neighbor\ friendly\ worm}(c \cdot |worm|)$, and finally,

- *update* - updating the worms (line 40), which counts about $\sum_{worm\ neighbor}(c \cdot |worm|)$ operations.

Let $|play_move|$ be the approximate number of operations performed by `play_move`. Then,

$$rand + neighbors \leq |play_move| \leq rand + neighbors + capture + connect + update.$$

At the beginning of the simulation, $|play_move|$ gets close, on an average, to $rand + neighbors$, but at the end it almost always goes through *connect* and *update*. Also, at the end, worms tend to be very long, thus yielding a greater number of operations per move.

Suppose, on the average, the length of a worm is $\mu_{|worm|} = 20$, a stone has about 3 neighbor worms and the number of operations necessary to process a worm is about $c = 6$. Then,

$$\begin{aligned} \mu_{|play_move|} &\approxeq rand + neighbors + \sum_{i=1}^{nr_neighbors} c \cdot \mu_{|worm|} \\ &\approxeq 10 + 10 + \sum_{i=1}^{3} 6 \cdot 20 \\ &= 380 \end{aligned}$$

is the average number of operations performed by this routine. As mentioned before, the average number of moves generated by this approach per game is about 300. If the CPU performs approximately 10,000,000 operations per second it means that we get, on the average

$$\frac{cpu}{nr_moves \cdot \mu_{|play_move|}} = \frac{10,000,000}{300 \cdot 380} \approxeq 90 \text{ games in one second.}$$

Indeed, in practice, this approach didn't achieve a level of more than 90-100 simulations per second, which made us look for a better idea[7].

[7] See, for results of the naïve approach Section 5.1.

4.3.1.2 The Common Fate Graph approach

The Common Fate Graph[8] term refers to a representation of the board of Go as an undirected graph.

A board position can be represented by its *full graph representation*[9], a graph with the structure of an $N \times N$ square grid. Formally, FGR is defined as $G_{FGR} = (P, E)$, an undirected connected graph, $G_{FGR} \in \mathcal{G}_{uc}$. The set $P = \{p_1, p_2, \ldots, p_{N_P}\}$ represents the intersections on the board, with each node $p \in P$ having one of three given labels $l : P \rightarrow \{black, white, empty\}$. The set $E = \{e_1, e_2, \ldots, e_{N_E}\}$ represents the vertical and horizontal neighborhood relation between intersections.

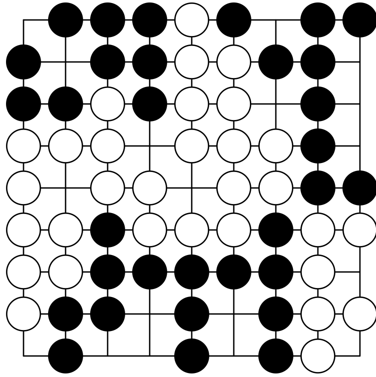

Figure 4.9: A board position (FGR)

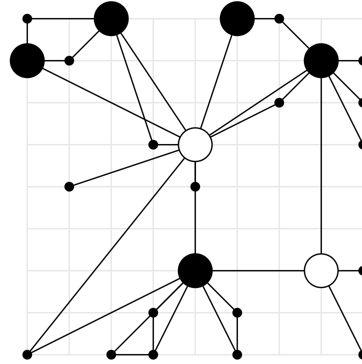

Figure 4.10: The board state in Figure 4.9 transformed into a Common Fate Graph

An important observation is that stones belonging to the same worm have *the same fate* on the board. This means that all the stones in the worm share the same liberties and neighbors. A stone in the worm is captured only when the whole worm is captured. Based on this observation, each worm is represented as a single node and each two nodes are connected by a single edge, representing their neighborhood relation. The resulting *reduced graph representation* is called a *Common Fate Graph*. Formally, we define the graph transformation $T : \mathcal{G}_{uc} \rightarrow \mathcal{G}_{uc}$ by the following rule. Let p_i and p_j be two neighbor nodes, $\{p_i, p_j\} \in E$ with the same label $l(p_i) = l(p_j) \neq empty$. Then,

1. $P \mapsto P \setminus \{p_j\}$;[10]

[8] Abbreviated CFG, see [14] and [19] for more details.
[9] Abbreviated FGR [14].
[10] The node p_j *melts into* node p_i.

2. $E \mapsto (E \setminus \{\{p_j, p_k\} \in E\}) \cup \{\{p_i, p_k\} | \{p_j, p_k\} \in E\}$.[11]

The graph resulted from repeating T over G_{FGR} until no neighbor nodes have the same label is the Common Fate Graph, G_{CFG}.

This representation has its bad sides, one of them being the fact that the shape of worms, the number of stones it contains and their structure are lost. These disadvantages influence in some manner the outgoing of the random simulation, but not in an essential way. Details are given below.

Even from the beginning, this approach shows itself better than the naïve one, as far as complexity is concerned. The move selection process still has the same steps, of which we'll talk about the two of them, having the most important word to say in determining complexity: *capture* and *connect*. Before we get into counting operations, a few details of how we see the implementation. The graph is represented by both adjacency lists and an adjacency matrix, since we need to favor speed over space.

At every connection of two friendly nodes p_i and p_j, we iterate through the neighbors of p_j and add them to p_i's list. For every entry p_k, we need to iterate through its neighbors and change every occurrence of p_j into p_i.

Capturing is a bit harder, since, as mentioned earlier, the structure of worms is lost. In other words, we don't have an easy way to remove a worm from the board, i.e. to replace one single worm (node) having the label *white* or *black* with the nodes representing the stones belonging to it, having the label *empty*. There are two ways of solving this problem. The first is to run backwards the process of transformation, until we find out all the intersections belonging to the worm. This approach, however, is time expensive, which we cannot afford at this step. The second is to just consider the worm as part of the enemy territory and to play on without moving inside it. This variant has the disadvantage of making the game somehow inaccurate in what game play authenticity is concerned. The accuracy can, however, be improved: when there is no legal move left except those inside worms, we create the FGR graph, restore the empty intersections and then re-create the CFG, after which the game may go on. This can be done several times, as long as needed to balance accuracy with speed.

Supposing that one will use our second approach for capturing, which consumes as much time as connecting friendly worms (that is, the time required to update the neighbors), we will only talk about the time complexity of this process. More exactly, we'll talk about connecting two worms with a friendly stone. Let *connect* be the number of operations required to perform this job. Also, let p_1, p_2 and p_3

[11] Connect the node p_i to all the neighbors of p_j.

be the three nodes which have to be connected. Then,

$$connect = \sum_{i=1}^{3} \Big(|neighbors(p_i)| + \sum_{p \in neighbors(p_i)} |neighbors(p)| \Big).$$

4.3.1.3 The Analyze-after approach

The CFG approach eliminates, indeed a lot of overhead, and intuition tells us to just use it. Still, there is the stone capturing problem and the relatively high amount of operations required by *connect*.

Our third idea, which turned out to behave the best, was to just play, with no concern whatsoever for the rules, except that of not filling friendly eyes. Then, when all the board is full, reanalyze the game and decide which moves were illegal, which stones were captured and which territory belongs to who. In order to present this approach we will formalize the concepts in the game of Go, so it is easier to see the solution.

Let N the size of the board. We define $\mathcal{B}$, the set of intersections on the board and $\mathcal{C}$, the set of possible colors for a stone:

$$\mathcal{B} = \Big\{ (x, y) \;\Big|\; 1 \le x, y \le N \Big\} \text{ and } \mathcal{C} = \{black, white\}.$$

Then, the set of possible moves in a game can be written as $\mathcal{M} = \mathcal{B} \times \mathcal{C}$. Having these defined, we introduce a general concept of game:

$$\mathrm{G}_{k, k \ge 1} = \Big\{ g_k \;\Big|\; g_k : \{1, \ldots, k\} \to \mathcal{M} \Big\}$$

G_k is the set of all games with k moves. Particularly, $G_0 = \{g_0 \mid g_0 : \emptyset \to \mathcal{M}\}$ is the set containing the game where no moves have been made (leaving the board empty). Also, $\mathcal{G} = \bigcup_{k \ge 0} \mathrm{G}_k$ is the set of all possible games (having no rules), where any $g \in \mathcal{G}$ is called a game.

In order to know what is the state of the board in a game g_k, at any moment of time t, we define the function $\mathrm{time}_{g_k} : \mathcal{B} \to \{1, \ldots, k\}$, as

$$\mathrm{time}_{g_k}(p) = \min \Big\{ 1 \le j \le k \;\Big|\; \exists c \in \mathcal{C} \;:\; g_k(j) = (p, c) \Big\}.$$

If there is not such j, then $\mathrm{time}_{g_k}(p) = \infty$. The value $\mathrm{time}_{g_k}(p)$ represents the first moment during g_k when the intersection p was occupied by a stone of color c.

Now we can define a first notion of board state, $\hat{\mathrm{b}}_{g_k} : \mathcal{B} \to \mathcal{C} \cup \{empty\}$ (one with no rules applied to it):

$$\hat{\mathrm{b}}_{g_k}(p) = \begin{cases} c \in \mathcal{C} & \text{, if } \mathrm{time}_{g_k}(p) < \infty \text{ and } g_k(\mathrm{time}_{g_k}(p)) = (p, c) \\ empty & \text{, otherwise.} \end{cases}$$

Having all these, the only thing that remains is to introduce rules, and finally have the state of the board based on a game, all rules respected. First, the set N_p containing the neighbors of the intersection p:

$$\mathrm{N}_p = \mathcal{B} \cap \Big\{(x+1,y),(x,y+1),(x-1,y),(x,y-1)\Big\}, \text{ where } p=(x,y).$$

Then, the notion of *worm*, $w_{g_k,p} \subseteq \mathcal{B}$, defined inductively:

Base

$$w_{g_k,p} = \begin{cases} \{p\} & \text{, if } \hat{\mathrm{b}}_{g_k}(p) \in \mathcal{C} \\ \emptyset & \text{, otherwise.} \end{cases}$$

Inductive step

$$\left.\begin{array}{l} p' \in w_{g_k,p} \\ p'' \in \mathrm{N}_{p'} \\ \hat{\mathrm{b}}_{g_k}(p'') = \hat{\mathrm{b}}_{g_k}(p') \end{array}\right\} \Rightarrow p'' \in w_{g_k,p}.$$

Let $w_{g_k,p} \subseteq \mathcal{B}$ be a worm in the game g_k. Then its liberties, $\mathrm{L}(w_{g_k,p}) \subseteq \mathcal{B}$, are:

$$\mathrm{L}(w_{g_k,p}) = \Big\{p'' \in \mathcal{B} \;\Big|\; \exists p' \in w_{g_k,p} \;:\; p'' \in \mathrm{N}_{p'} \wedge \hat{\mathrm{b}}_{g_k}(p'') = \mathit{empty}\Big\}.$$

Similarly, we can write the set of worm neighbors, $\mathrm{N}(w_{g_k,p}) \subseteq \mathcal{B}$:

$$\mathrm{N}(w_{g_k,p}) = \Big\{p'' \in \mathcal{B} \;\Big|\; \exists p' \in w_{g_k,p} \;:\; p'' \in \mathrm{N}_{p'}\Big\} \setminus w_{g_k,p}.$$

Each worm has a coefficient of newness, $\mathrm{newness} : \mathcal{G} \times \mathcal{P}(\mathcal{B}) \to \mathbb{N}$ giving the time the latest stone of the worm was played:

$$\mathrm{newness}(g_k, w) = \max\Big\{1 \le t \le k \;\Big|\; \exists p \in w \;:\; t = \mathrm{time}_{g_k}(p)\Big\}.$$

The coefficient of newness helps determine the correct state of the board at each moment in time. Finally the state of a board, based on a game, $\mathbf{b}_{g_k} : \mathcal{B} \to \mathcal{C} \cup \{\mathit{empty}\}$, is defined inductively as:

Base

$$\forall p \in \mathcal{B}, \hat{\mathrm{b}}_{g_k}(p) = \mathit{empty} \Rightarrow \mathbf{b}_{g_k}(p) = \mathit{empty}$$

Inductive step

$$\text{Let } w_{g_k,p} = \underset{w_{g_k,p}}{\operatorname{argmin}}\Big\{\mathrm{newness}(w_{g_k,p}) \Big|\; \exists p' \in w_{g_k,p} \;:\; \mathbf{b}_{g_k}(p') \text{ is undefined}\Big\}.$$

$$\text{Then, } \mathbf{b}_{g_k}(p') = \begin{cases} \mathit{empty} & \text{, if } \forall p'' \in \mathrm{N}(w_{g_k,p}) \;:\; \mathbf{b}_{g_k}(p'') \neq \mathit{empty} \\ \hat{\mathrm{b}}_{g_k}(p') & \text{, otherwise.} \end{cases} \quad , \forall p' \in w_{g_k,p}.$$

This result tells us that, having a sequence of completely arbitrary moves (their order not respecting any rule of Go), we can generate a valid Go board state, as if

the rules had been followed from the beginning. In other words, if we just placed stones on the board until there is literally no place to move, there would be a way to extract a correct Go endgame board, having clearly delimited territories. The resulted board would be easy to evaluate and the road to get to it is a simple and especially, fast one.

Having all of that said, the algorithm idea comes straightforward. Initialization remains the same as the one of the naive random simulation approach. Iteration changes slightly, to include computation of time (lines 2 and 7):

```
1     int pos, n_passes = 0;
2     int moment = 0;
3
4     while (moves->count > 0 && n_passes < 2)
5     {
6         pos = play_move(color, moves);
7         time[move] = ++moment;
8
9         if (pos != PASS_MOVE) {
10            n_passes = 0;
11        } else n_passes++;
12
13        color = OTHER_COLOR(color);
14    }
```

Also, after iteration, there are two more steps which need to be performed before the evaluation of the board: building of worm information and removing from the board the *captured* strings.

```
15
16    array_list_t worms = CREATE_WORM_ARRAY();
17    build_worm_infos(state, worms, time);
18    capture_worms(state, worms);
```

And scoring becomes a simple formality:

```
19
20    // positive result means white wins
21    return state->komi + state->white_captured
22                       - state->black_captured;
```

where `state->white_captured` and `state->black_captured` are calculated as:

$$\texttt{white_captured} = \left|\{p \in \mathcal{B} \mid \hat{\mathrm{b}}_{g_k}(p) = \mathit{black} \wedge \mathbf{b}_{g_k}(p) = \mathit{empty}\}\right|$$

and

$$\texttt{black_captured} = \left|\{p \in \mathcal{B} \mid \hat{\mathrm{b}}_{g_k}(p) = \mathit{white} \wedge \mathbf{b}_{g_k}(p) = \mathit{empty}\}\right|$$

Move selection changes to include only the *random* and *neighbors* parts:

```
1  play_move(board_state state, colors color,
2            array_list_t moves)
3  {
4      while (!found_legal_move && moves->count > 0)
5      {
6          move = array_list_pop_random(moves);
7
8          if (move fills an eye) {
9              continue;
10         } else {
11             found_legal_move = true;
12         }
13     }
14
15     if (found_legal_move) {
16         state->board[move] = color;
17         return move;
18     } else {
19         return PASS_MOVE;
20     }
21 }
```

For the purpose of implementing `build_worm_infos`, we created a structure called intuitively `worm_info` given in Figure 4.11, where

- `stones` contains the indexes of all stones belonging to the worm.
- `neighbors` contains the indexes of the neighbors in the array with all worms.
- visited_neighbors, as the name suggests, is used for tracking the already processed neighbors, so we don't iterate through any of them twice.
- latest_stone is the value of newness($worm$).

```
1  struct worm_info
2  {
3      array_list_t stones;    // the one dimensional index
4      array_list_t neighbors; // the indexes of the neighbor
5                               // worms
6
7      visited_list_t visited_neighbors;
8
9      int latest_stone;        // the latest stone in the worm
10     unsigned has_liberties : 1;
11     unsigned color : 2;
12     unsigned : 5;
13 };
14
15 typedef struct worm_info *worm_info_t;
```

Figure 4.11: The Worm Info structure

- has_liberties is 1 if the worm has at least one liberty and 0 otherwise. And
- color is 1 if the worm is *white* and 2 if it is *black*[12]

Let's analyze the complexity of this approach. First of all, the initialization time, $init = c_1 \cdot |\mathcal{B}|$, which is within $O(|\mathcal{B}|)$. Then, the iteration part, $iter = rand \cdot |\mathcal{B}|$, also within $O(|\mathcal{B}|)$. Building worm information, $build_worm_infos = c_2 \cdot |\mathcal{B}|$ has the same complexity, $O(|\mathcal{B}|)$. Capturing worms, $capture_worms$, encapsulates two steps: sorting the worms by their lateness and removing stones from the board. The first has complexity $O(|worms| \cdot \log_2^{|worms|})$, where $|worms|$ represents the number of strings on the board, while the second, $O(|\mathcal{B}|)$. In consequence, $capture_worms = c_3 \cdot |worms| \cdot \log_2^{|worms|} + c_4 \cdot |\mathcal{B}|$. Summing up, if $analyze_after$ represents the number of operations performed by this approach, then

$$analyze_after = (c_1 + rand + c_2 + c_4) \cdot |\mathcal{B}| + c_3 \cdot |worms| \cdot \log_2^{|worms|}.$$

In our implementation, this sum raised to about $8,000$, meaning $8,000$ operations per simulation, which yielded approximately $1,200$ games in a second. Although not perfectly accurate and missing some game plays, this approach behaved wonderful, giving excellent results, for which reason we decided upon using it.

[12]We used 2 bits for storing explicitly 1 and 2, since 0 is in general reserved for *empty*.

4.3.2 The Multi-armed Bandit Problem

The multi-armed bandit problem[13] is associated with the *Exploration versus Exploitation* dilemma, which can shortly be described as searching for a balance between *exploring* the environment to find profitable actions and *exploiting* the best of the already known part of it.[14]

The problem of the multi-armed bandit is the simplest instance of this dilemma, having thus been extremely studied in statistics (Berry & Fistedt, 1985), but also in several areas of artificial intelligence like reinforcement learning (Sutton & Barto, 1998) and computer optimization, such as genetic algorithms. It can be formulated as follows. A multi-armed bandit, also called a *K-armed bandit* is a slot machine having more than just one lever (K levers), or otherwise, a series of K classical slot machines. When played, each machine provides a reward drawn from a distribution associated to that specific machine. The objective of the gambler is to maximize the collected reward sum through iterative pulls. The gambler has no prior knowledge of the machines. The crucial trade-off he faces at each trial is between *exploitation* of the machine with the highest expected payoff and *exploration* - getting more information about the expected payoffs of the other machines.

Formally, the multi-armed bandit is defined by random variables $X_{i,\ i=\overline{1,K}}$, where each i is the index of a slot machine (i.e. the *arm* of the bandit). Successive plays of machine i yield rewards $X_{i_1}, X_{i_2}, \ldots$ which are independent and identically distributed according to an unknown law with unknown expectation μ_i. Independence also holds for rewards across machines; i.e., X_{i_s} and X_{j_t} are independent (and usually not identically distributed) for each $1 \leq i < j \leq K$ and each $s, t \geq 1$.

Algorithms choose the next machine to play depending on the obtained results of the previous plays. Let $T_i(n)$ be the number of plays the machine i has been played after the first n plays ($\sum_{i=1}^{K} T_i(n) = n$). Since the algorithm doesn't always make the best choice, its expected loss is studied. Then, the *regret* after n plays is defined by

$$\mu^* \cdot n - \sum_{j=1}^{K} \mu_j E[T_j(n)], \text{ where } \mu^* = \max_{1 \leq i \leq K} \mu_i.$$

$E[\cdot]$ denotes expectation. Authors of [1] propose a strategy called UCB1, based on maximizing the upper confidence bound of each machine. The resulting algorithm

[13]See [26] for a more detailed description of the multi-armed bandit problem and [1], [13] for its solving approach, referenced in our paper.

[14]See also [15].

is proved to achieve logarithmic regret uniformly over n, when rewards are in $[0,1]$. Let

$\overline{X}_{i,s} = \frac{1}{s}\sum_{t=1}^{s} X_{i_t}$ be the average reward of machine i after s plays and

$\overline{X}_i = \overline{X}_{i,T_i(n)}$, its average reward, in n overall number of plays done so far.

Then we have:

Algorithm 1. *Deterministic policy: UCB1*

- *Initialization: Play each machine once.*
- *Loop: Play machine j that maximizes* $\overline{X}_j + \sqrt{\frac{2\ln n}{T_j(n)}}$.

A formula with better experimental results is suggested in [1] and [13]. Let

$$V_j(s) = \Big(\frac{1}{s}\sum_{t=1}^{s} X_{j_t}^2\Big) - \overline{X}_{j,s}^2 + \sqrt{\frac{2\ln n}{s}}$$

be an estimated upper bound on the variance of machine j. Then the new value to maximize becomes:

$$\overline{X}_j + \sqrt{\frac{\ln n}{T_j(n)} \cdot \min\Big\{0.25, V_j\Big(T_j(n)\Big)\Big\}}. \qquad (4.1)$$

According to [1] and [13], the policy maximizing (4.1), named UCB1-TUNED, performs substantially better than UCB1 in all experiments. For this reason, we only used UCB1-TUNED[15] in our work.

4.3.3 Upper Confidence for Trees

As mentioned in Section 3.2.2, given a state, basic Monte Carlo follows the next simple steps: it launches random simulations starting with each available move. After a certain amount of time, it picks the move having the best average outcome and plays it. Having presented the problem of the multi-armed bandit and the exploration versus exploitation dilemma, it is now easy to see a certain resemblance between it and the problem of choosing the next move in the Monte Carlo approach.

Let's suppose that, after a few random simulations, some of the moves keep giving bad results, while others, better ones. The algorithm, as presented so far, doesn't

[15] Which will be, however, referenced as UCB1, for simplicity.

use this information in any way, and keeps *exploring* the bad moves with the same probability as the good ones. What we would need is to *exploit* the good moves by running simulations starting with them more often, while not wasting time with the bad ones. So here too, we need to set a balance between *exploration* and *exploitation* when choosing from moves with different winning probabilities.

What we will do is to associate each current Go position to a bandit problem. Playing a machine means launching a random simulation starting with a certain move derived from the current position. The reward may either be 1, if the game is won, or 0 otherwise. So the algorithm goes as follows. Initially, every available move is played once. At each iteration, instead of playing a randomly chosen move, we will elect the best one corresponding to the UCB1 formula.

In order to implement UCT, we created two specific structures, `uct_tree` and `uct_node`, and associated with each of them various operations. In Figure 4.12 we have the `uct_node` structure of which we will discuss in the following lines.

```
1  /* A node in the UCT tree */
2  struct uct_node
3  {
4      short move; /* the index of the move in the 1-D board
5                     or 0 if the node represents the state of
6                     the empty board */
7      unsigned color : 2;
8      unsigned has_unvisited_moves : 1;
9
10     float value;            /* the sum of all values */
11     int n_visits;
12
13     heap_array_t children;  /* the children of the node,
14                                stored inside a max-heap */
15     struct uct_node *parent;/* parent of the node */
16 };
17
18 typedef struct uct_node* uct_node_t;
```

Figure 4.12: The UCT node structure

A UCT node represents a state of the board in a particular game. Thus, it is composed of:

- `move`, a number between 1 and N^2, representing the index of an intersection on the board, counting from the upper-left corner to bottom-right;

- `color`, which may be *empty*, if the node is the root of the tree, or *black/white* otherwise;
- `has_unvisited_moves`, a flag which is set if there are unvisited legal moves available from this node;
- `value`, representing the sum $\sum_{t=1}^{T_i(n)} X_{i_t}$, where i is the index of the current node and X_{i_t} is the outcome of the simulation t, which may be 1, if the game was won by the player corresponding to the node and 0 otherwise;
- `n_visits`, the number of simulations containing this node, namely $T_i(n)$;
- `children`, a heap containing the nodes deriving from this one. The *best* item in the heap is the one with the greatest value for UCB1[16];
- `parent`, the parent UCT node of the current one, if there is such (the current node isn't the root of the tree).

In order to build the tree, we used the algorithm advanced in [13], which we present in Figure 4.13. In common words, the algorithm descends in the UCT tree, by choosing, at each step, the node with the greatest value for UCB1. When a leaf is encountered (a node yet unvisited), it launches a random simulation starting with it, and updates the entire line starting with the leaf, and up to the current node, with the value retrieved.

```
1 void uct_play_one_sequence(
2     uct_node_t current_node, board_state *state)
3 {
4     float outcome;
5
6     uct_node_t iter = current_node;
7     do
8     {
9         iter = uct_descend_by_ucb1(iter, state);
10    }
11    while (iter->visited);
12
13    outcome = uct_node_update_value_by_mc(iter, state);
14    uct_update_value(iter, outcome);
15 }
```

Figure 4.13: UCT as described in [13]

Here's how it works[17].

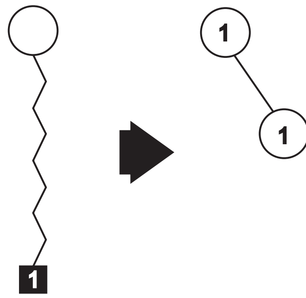

Figure 4.14: UCT: The beginning of the algorithm. A random simulation is launched. Its outcome was a favorable one, so the mean value of the node becomes 1. The mean value of the parent node is also 1.

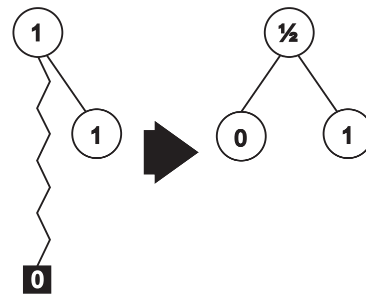

Figure 4.15: UCT: Legal moves are still available. Thus, another game is launched. This time, the current color lost, so the mean value for the newly created node is 0, while its parent's becomes $\frac{1}{2}$.

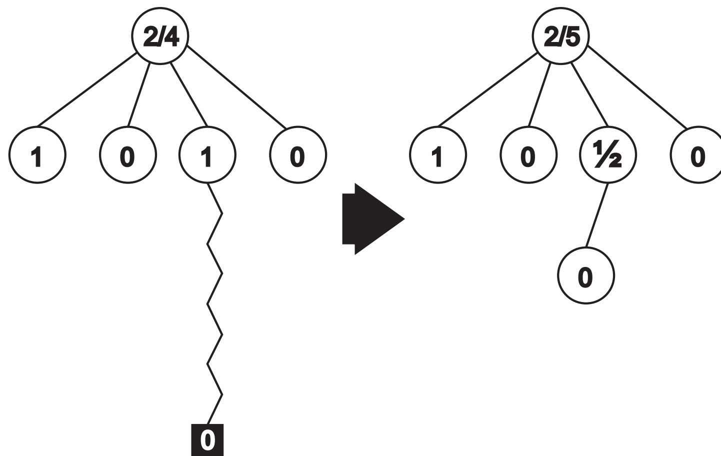

Figure 4.16: UCT: No available legal move remains, so the node with the greatest UCB1 value is chosen. A random game is played beginning with it as the first move. The corresponding color lost, so the mean value of its new son is 0. Notice that all its ancestors are updated as well (its father has now the mean value of $\frac{1}{2}$, while its grandparent, $\frac{2}{5}$).

At first, a random simulation is launched, and its value stored inside the root node. A new child node is created (Figure 4.14). Then, another game is launched for another unvisited move, its value backed up and another child node created (Figure 4.15). The process goes on until there is no unvisited legal move. From this point, at each step, the node with the best UCB1 value is chosen, and a new random simulation is launched starting with it. The outcome is saved, and its

[16]See Section 4.2.2 for the way a heap array is built.
[17]See [6] for reference.

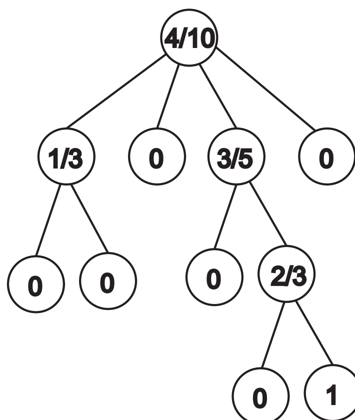

Figure 4.17: UCT: The algorithm after a few iterations. Notice how the node yielding better results has been visited more often.

ancestors updated (Figure 4.16). The algorithm runs until a certain amount of time runs out (Figure 4.17).

UCT runs in a minimax fashion, since the best move for one color is the one that minimizes the other color's chances of winning. The goal of the search is the optimal branch at the root node. It is acceptable, however, if one branch with score near to the optimal one is found, especially in the case of Go, where the depth of the tree is large and the branching factor is big, as it is often too difficult to find the optimal branch within short time. As shown in [13], in this sense, UCT outperforms alpha-beta search from at least three perspectives.

First of all, it works in an anytime manner, meaning that we can stop the algorithm at any moment, and its performance can be somehow good. This is not the case of alpha-beta search. Figures 4.18 and 4.19 show the difference between the way each of the trees, UCT and alpha-beta, evolves in time. With alpha-beta, if we stop it prematurely, some moves at the first level remain unexplored. So the chosen move can be far from optimal. Of course, iterative deepening can be used, and partially solve this problem. Still, the anytime property is stronger for UCT, making it easier for it finely control time.

Secondly, UCT is robust as it automatically handles uncertainty in a smooth way. At each node, the computed value is the mean of the value for each child weighted by the frequency of visits. Then the value is a smoothed estimation of max, as the frequency of visits depends on the difference between the estimated values and the confidence of these estimates. Then, if one child-node has a much higher value than the others, and the estimate is good, this child-node will be explored much more often than the others. Also, UCT will select most of the

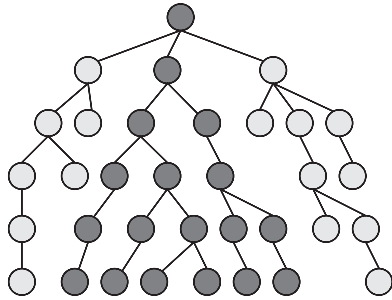

Figure 4.18: The way a UCT tree grows. The shape of the tree enlarges asymmetrically, according to the best explored move. Yet, all the legal moves are also explored. If, at this point, the algorithm stops, a good decision can be made based upon the way the tree evolved so far.

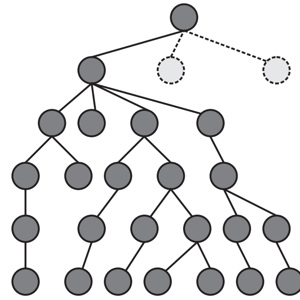

Figure 4.19: The way an alpha-beta tree grows. The algorithm only explores few possible moves in a limited time (the light gray nodes represent unvisited nodes). This happens often during large-sized tree search, where entire search is impossible.

time the maximum child node. However, if two child-nodes have a similar value, or a low confidence, then the value will be closer to an average.

Thirdly, the tree grows in an asymmetric manner: it explores more deeply the good moves and this is achieved in an automatic manner.

4.3.4 Key Positions Priority

Since the Go moves tree can get so large, many of the nodes remain unvisited. Because of this, some of their parents create a wrong impression (either bad or good) because none of their significant children got to be explored in time.

One minor fix to this problem is to change the order in which moves are visited, favoring *zones* of the board where playing is considered better most of the time. In other words, the order of exploring child moves for the first time should be dictated by the importance of the position on the board. For example, a move in the corner, or on the edge is generally considered to be a bad move, no matter what the state of the game is, while the ones close to the marked intersections ($D4$, $Q4$, etc.) are, in general, better. So these moves should have priority when exploring for the first time.

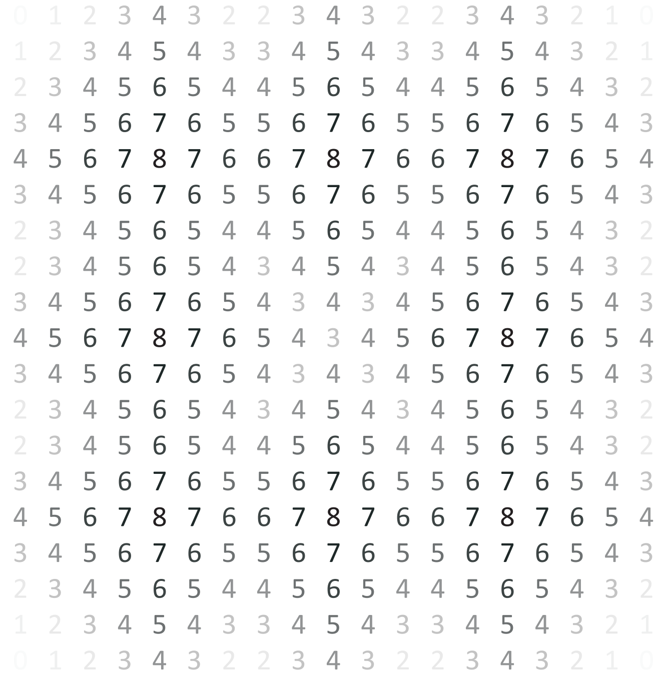

Figure 4.20: Intersections, according to Key Positions Priority. The better intersections (strong gray) have better probabilities of being chosen.

4.3.4.1 Creating a distribution using Bézier curves

In order to have a distribution according to intersection importance, we decided to generate a graphic which could be tuned according to our needs. The graphic would have to look like the one in Figure 4.21. What we did was to use the equation of a Bézier curve, which is easily customizable.

In the mathematical field of numerical analysis, a Bézier curve is a parametric curve important in computer graphics and related fields[18]. Four points $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ in the plane or in three-dimensional space define a cubic Bézier curve. The curve starts at $\mathbf{P}_0$ going toward $\mathbf{P}_1$ and arrives at $\mathbf{P}_3$ coming from the direction of $\mathbf{P}_2$. Usually, it will not pass through $\mathbf{P}_1$ or $\mathbf{P}_2$; these points are only there to provide directional information. The distance between $\mathbf{P}_0$ and $\mathbf{P}_1$ determines *how long* the curve moves into direction $\mathbf{P}_2$ before turning towards $\mathbf{P}_3$.

The parametric form of the curve is:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3t(1-t)^2\mathbf{P}_1 + 3t^2(1-t)\mathbf{P}_2 + t^3\mathbf{P}_3 \ , \ t \in [0,1].$$

[18] See [21] for a detailed description of Bézier curves.

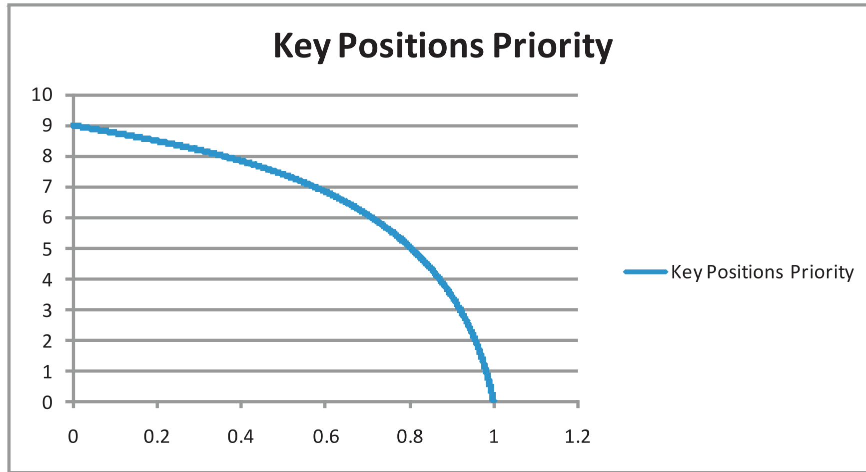

Figure 4.21: Key Positions Priority: Distribution. To every move category $i \in \{0, \ldots, 8\}$ in Figure 4.20 corresponds an interval $[a, b] \subset [0, 1]$, $[a, b] = \Big\{x \in [0, 1] \Big| (x, y) \in f \Rightarrow y \in [i, i + 1]\Big\}$, where f is the function which generated the graphic.

Determining $f = \{(x, y) \in \mathbf{B}(t) | t \in [0, 1]\}$ based on $\mathbf{B}$ can be a time consuming problem, since it involves solving a cubic equation. However, in our case this isn't necessary, since we don't need exact precision. All we need is that, given an $x \in [0, 1]$, we find an interval $[a, b]$ with $y \approxeq f(x) \in [a, b]$, which corresponds to the move category we want to select.

The algorithm goes as follows. Initially, we define a number of intervals proportional to the precision we want to reach, let's say, max_t. We then split $[0, 1]$ into max_t intervals, t_i, generating a point (x_i, y_i) for each: $(x_i, y_i) = \mathbf{B}(t_i)$. We store the points in an array, sorted by x. Whenever we need the approximate value of $f(x)$ for a given x, we binary search through our point array and find i such that $x_i \leq x < x_{i+1}$. The corresponding $y_i = f(x_i) \approxeq f(x)$, is the value we're looking for.

Coming back to setting a move priority, choosing a move according to our defined distribution becomes a trivial job. Whenever we need to select an available legal intersection for the next move, we generate a random number $r \in [0, 1]$. Then, we find $[a, b]$ and i such that $r \in [a, b]$ and $[a, b] = \{x | f(x) \in [i, i + 1]\}$. Finally, we select at random a move from the category i and launch the random game starting with it.

4.3.5 First-play urgency

UCB1 algorithm begins by exploring each arm once, before using the formula (4.1). This can sometimes be inefficient, especially if the number of trials is not large comparing to the number of children. This is the case for numerous nodes in the tree (number of visits is small comparing to the number of moves). For example, if an arm keeps returning 1 (win), there is no good reason to explore other arms.

As suggested in [13], we used an heuristic called first-play urgency, which consists of using a threshold, FPU for each node, which controls how urgent it is. If the value of the node for formula (4.1) is greater than this threshold, then the move has priority over unvisited moves. The FPU is by default set to ∞ for each legal move before first visit (see line 11 in Figure 4.13). Any node, after being visited at least once, has its urgency updated according to UCB1 formula. We play the move with the highest urgency. Thus, the FPU ∞ ensures the exploration of each move once before further exploitation of any previously visited move. On the other way, smaller FPU ensures earlier exploitations if the first simulations lead to an urgency smaller than FPU (in this case the other unvisited nodes are not selected).

```
1     // First play urgency (FPU)
2     if (N_CHILDREN(node) > 0)
3     {
4         uct_node_t son = uct_node_max_ucb1_child(node);
5
6         if (UCB1(son) >= FPU)
7         {
8             return son;
9         }
10    }
```

Figure 4.22: First-play urgency

In our implementation, we added to the function `uct_play_one_sequence` the code in Figure 4.22, before checking for unvisited legal moves.

4.3.6 Learning from past experience

Although, for the 9×9 and 13×13 boards, the algorithms and heuristics presented so far are sufficient for an engine to to give impressive results, the 19×19 board

is more demanding, needing a bit more in order to have coherence in its actions. For this, we used two improvements to the ideas presented so far.

4.3.6.1 Grandparent knowledge

The first idea is the one presented in [13], called *grandparent knowledge*, which basically uses the fact that moves on the same intersection, close in time, have similar influence over the outcome of the game. Let c be the current state and g be its grandparent node. Then, at the time g was the current state, Monte Carlo explored the uncles of c, i.e. the alternative moves to the parent of c, the one eventually chosen. The idea is to combine the values of the sons of c with the values of its uncles whenever c becomes the current node.

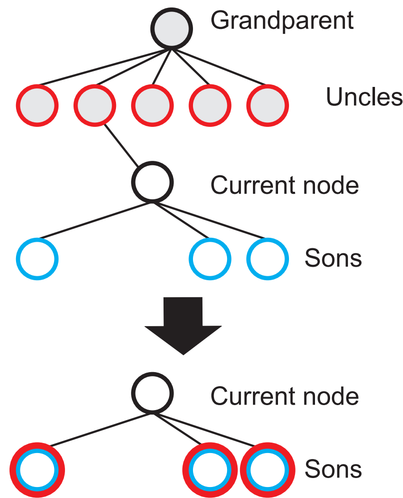

Figure 4.23: Grandparent knowledge: Monte Carlo explores moves as usual. With each actual move, the current node changes, and before we start exploring its children, all the values of the uncles update the corresponding sons.

Figure 4.23 illustrates how it works. The formula we used is as follows.

$$\texttt{son->value} \leftarrow \frac{\texttt{son->value} + \texttt{uncle->value}}{\texttt{son->n_visits} + \texttt{uncle->n_visits}} \cdot \texttt{son->n_visits}$$

Formally, we can write the mean of the current node, taking account of the above formula, as:

$$\overline{X}_s = \frac{1}{T_s(n_c) + T_u(n_g)} \Big(\sum_{t=1}^{T_s(n_c)} X_{s_t} + \sum_{t=1}^{T_u(n_g)} X_{u_s} \Big),$$

where:

- c represents the current node;
- s represents the son currently updating;
- g is the grandparent of the current node;
- u is the uncle of the current node, corresponding to s. Then,
- $T_s(n_c)$ is the number of visits for the son, and
- $T_u(n_g)$ is the number of visits for the uncle.

In other words, the new mean of the son being updated is the average value of all the simulations launched both from the son and from the uncle. This approach works very well, especially on the 19×19 board, where the results are very satisfactory.

4.3.6.2 The Experience Tree

The *experience tree*, or *memory tree*, as we called it, represents a UCT tree created and saved in past games. When exploring a move, before visiting all its sons, our algorithm first loads from the memory tree any sons this node may have as if they were already visited.

This speeds up the beginning of the game and increases the accuracy of Monte Carlo, since it allows it to search deeper into the tree, the first nodes being already explored. Of course, the results can be seen only at the beginning of the game, since towards the end, boards differ very much and it is very hard to run into an exact same position as in a previous game.

In order to implement this concept, so that we can store a tree in a file, we created a structure called `uct_memory_node`, shown in Figure 4.24, where:

- `next_sibling` is the index of the next sibling of the current node, if there is such, or 0 otherwise;
- `first_child` is the index of the first child, or 0, if the current node has no children;
- `n_children`, `move`, `color`, `value` and `n_visits` store the corresponding data of a UCT node.

```
1  struct uct_memory_node
2  {
3      int next_sibling;
4      int first_child;
5
6      short n_children;
7      short move;
8      enum colors color;
9
10     float value;
11     int n_visits;
12 };
13
14 typedef struct uct_memory_node *uct_memory_node_t;
```

Figure 4.24: The memory node structure

The nodes are stored in a file in the same order as the BFS[19] search. Figure 4.25 shows an example of how a UCT memory tree is saved in a file.

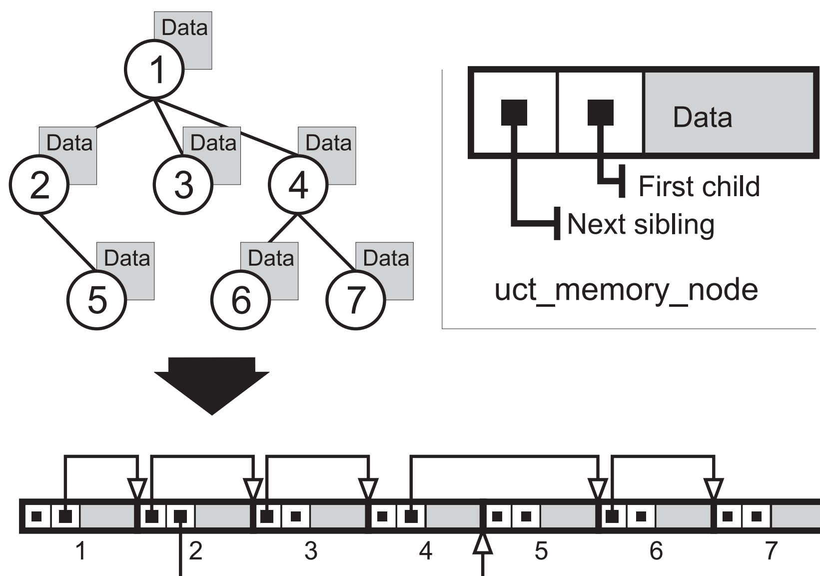

Figure 4.25: UCT Memory Tree: Saving

Finally, we added a flag in the `uct_node` structure, called `has_unloaded_children`, which is used for monitoring the state of the node. If it was loaded from the

[19] Visit http://en.wikipedia.org/wiki/Breadth-first_search for a detailed description of the breadth-first search algorithm.

memory tree, and its children haven't, then the flag is set to *true*, otherwise it is *false*.

4.3.7 All-moves-as-first

This heuristic has an important word to say for the Monte Carlo Go playing engines, since it allows the process of evaluating a move to divide the response time by the size of the board. The idea is simple: after a random game with a certain score, instead of just updating the mean of the first move of the random game, the heuristic updates all moves played first on their intersections with the same color as the first move. It also updates with the opposite score the means of the moves played first on their intersections with a different color from the first move [8].

First of all, let's write the average outcome of all simulations in which move a was selected in position s, like[20]

$$\begin{aligned} Q(s,a) = \overline{X}^s_a &= \frac{1}{T^s_a(n)} \sum_{t=1}^{T^s_a(n)} X^s_{a_t} \\ &= \frac{1}{T^s_a(n)} \sum_{i=1}^{n} I^s_i(a) z_i, \text{ where} \end{aligned} \tag{4.2}$$

- $T^s_a(n)$ is the number of simulations in which a was selected in position s[21];
- $X^s_{a_t}$ is the outcome of the tth random game in which move a was selected in position s;
- $I^s_i(a)$ is an indicator function returning 1 if move a was selected in position s for the ith simulation, and 0 otherwise. Notice that $T^s_a(n) = \sum_{i=1}^{n} I^s_i(a)$;
- $z_i = X^s_{a_{T^s_a(i)}}$ represents the outcome of the ith simulation.

Then, we define $Q^*(s,a)$, the average outcome of the random games in which move a was played *for the first time in its intersection*, in position s, or any subsequent position,

$$Q^*(s,a) = \frac{1}{T^{s*}_a} \sum_{i=1}^{n} I^{s*}_i(a) z_i, \text{ where} \tag{4.3}$$

[20] See [12] for reference.
[21] See Section 4.3.2.

$$I_i^{s*}(a) = \begin{cases} 1 & \text{if } s \text{ encountered at step } k \text{ of the } i^{th} \text{ simulation, and move } a \text{ was} \\ & \text{selected at step } t \geq k \text{ for the first time in its intersection;} \\ 0 & \text{otherwise,} \end{cases}$$

and $T_a^{s*} = \sum_{i=1}^n I_i^{s*}(a)$.

Using $Q^*(s,a)$ we can update the means of almost all moves in the game. Of course, the heuristic isn't entirely correct, since various moves may have different effects on the game depending on the time they were played, but the speedup it brings to evaluation is worth taking into consideration.

4.3.8 Rapid Action Value Estimates

UCT learns a unique value for each node in the search tree, estimated online from experience simulated from the current position. However, it cannot generalize between related positions. The RAVE algorithm provides a simple way to share experience between classes of related positions, resulting in a rapid, but biased value estimate [12].

The RAVE value $\hat{Q}(s,a)$ is the average outcome of all simulations in which move a is selected in position s, or in any subsequent position,

$$\hat{Q}(s,a) = \frac{1}{\hat{T}_a^s} \sum_{i=1}^n \hat{I}_i^s(a) z_i, \text{ where} \tag{4.4}$$

$$\hat{I}_i^s(a) = \begin{cases} 1 & \text{if } s \text{ encountered at step } k \text{ of the } i^{th} \text{ simulation, and move } a \text{ was} \\ & \text{selected at any step } t \geq k; \\ 0 & \text{otherwise,} \end{cases}$$

and $\hat{T}_a^s = \sum_{i=1}^n \hat{I}_i^s(a)$ counts the total number of simulations used to estimate the RAVE value.

The RAVE value generalizes the value of move a across all positions in the subtree below s. We can easily notice the close relation to the all-moves-as-first heuristic.

The Monte Carlo value, $Q(s,a)$ is unbiased, but may be high variance if insufficient experience is available. The RAVE value $\hat{Q}(s,a)$ is biased, but lower variance; it is based on more experience, but this generalization may not always be appropriate. Hence, the RAVE value is used initially, while gradually shifting to the Monte Carlo value, by using a linear combination of these values with a decaying weight.

4.4 Adding Monte Carlo to GNU Go

4.4.1 GNU Go engine overview

GNU Go starts by trying to get a good understanding of the current board position. Using the information found in this first phase, and using additional move generators, a list of candidate moves is generated. Finally, each of the candidate moves is valued according to its territorial value (including captures or life-and-death effects), and possible strategic effects (such as strengthening a weak group).

Although GNU Go does a lot of reading to analyze possible captures, life and death of groups etc., it does not have a full-board lookahead and this is the main point where improvements can be made[22].

4.4.1.1 Gathering information

This is by far the most important phase in the move generation, which is done by the function `examine_position()`. It first calls `make_worms()`.

After knowing which worms are tactically stable, GNU Go makes a first picture of the balance of power across the board: the *influence* code is called for the first time.

This is to aid the next step, the analysis of *dragons*[23]. Naturally the first step in the responsible function `make_dragons()` is to identify these dragons, i.e. determine which worms cannot be disconnected from each other. This is partly done by patterns, but in most cases the specialized *readconnect* code is called. This module does a minimax search to determine whether two given worms can be connected with, respectively disconnected from each other.

Then GNU Go computes various measures to determine how strong or weak any given dragon is:

- A crude estimate of the number of eyes is made.
- A guess is made for the potential to escape if the dragon got under attack.

[22] For a full description of the GNU Go engine, visit the GNU Go documentation page: http://www.gnu.org/software/gnugo/gnugo_toc.html

[23] By a *dragon* we mean a group of stones not necessarily directly connected, that cannot be cut.

For those dragon that are considered weak, a life and death analysis is made (the *Owl* code). If two dragons next to each other are found that are both not alive, the situation is handled with the *semeai* module.

The influence code is then called second time to make a detailed analysis of likely territory. Of course, the life-and-death status of dragons are now taken into account.

The territorial results of the influence module get corrected by the *break-in* module. This specifically tries to analyze where an opponent could break into an alleged territory, with sequences that would be too difficult to see for the influence code.

4.4.1.2 Move Generators

Once it has found out all about the position, GNU Go generates the best move. Moves are proposed by a number of different modules called *move generators*. The move generators themselves do not set the values of the moves, but enumerate justifications for them, called *move reasons*. The valuation of the moves comes last, after all moves and their reasons have been generated.

There are some move generators that only extract data found in the previous phase, examining the position:

- `worm_reasons()`. Moves that have been found to capture or defend a worm are proposed as candidates.
- `owl_reasons()`. The status of every dragon, as it has been determined by the owl code in the previous phase, is reviewed. If the status is critical, the killing or defending move gets a corresponding move reason.
- `semeai_move_reasons()`. Similar to `owl_reasons`, this function proposes moves relevant for semeais.
- `break_in_move_reasons()`. This suggests moves that have been found to break into opponent's territory by the break-in module.

The following move generators do additional work:

- `fuseki()`. Generate a move in the early fuseki, either in an empty corner of from the fuseki database.

- `shapes()`. This is probably the most important move generator. It finds patterns from `patterns.db`, `patterns2.db`, `fuseki.db`, and the joseki files in the current position. Each pattern is matched in each of the 8 possible orientations obtainable by rotation and reflection. If the pattern matches, a so called "constraint" may be tested which makes use of reading to determine if the pattern should be used in the current situation. Such constraints can make demands on number of liberties of strings, life and death status, and reading out ladders, etc. The patterns may call helper functions, which may be hand coded (in `patterns/helpers.c`) or auto-generated.

  The patterns can be of a number of different classes with different goals. There are e.g. patterns which try to attack or defend groups, patterns which try to connect or cut groups, and patterns which simply try to make good shape.[24]

- `combinations()`. See if there are any combination threats or atari sequences and either propose them or defend against them.

- `revise_thrashing_dragon()`. This module does not directly propose move: If GNU Go is clearly ahead, and the last move played by the opponent is part of a dead dragon, we want to attack that dragon again to be on the safe side. This is done be setting the status of this *thrashing dragon* to *unknown* and repeating the shape move generation and move valuation.

- `endgame_shapes()`. If no move is found with a value greater than 6.0, this module matches a set of extra patterns which are designed for the endgame. The endgame patterns can be found in `patterns/endgame.db`.

- `revise_semeai()`. If no move is found, this module changes the status of opponent groups involved in a semeai from `DEAD` to `UNKNOWN`. After this, genmove runs `shapes` and `endgame_shapes` again to see if a new move turns up.

- `fill_liberty()`. Fill a common liberty. This is only used at the end of the game. If necessary a backfilling or back-capturing move is generated.

4.4.1.3 Move Valuation

After the move generation modules have run, each proposed candidate move goes through a detailed valuation by the function `review_move_reasons`. This invokes some analysis to try to turn up other move reasons that may have been missed.

[24]In addition to the large pattern database called by `shapes()`, pattern matching is used by other modules for different tasks throughout the program.

The most important value of a move is its territorial effect. This value is modified for all move reasons that cannot be expressed directly in terms of territory, such as combination attacks (where it is not clear which of several strings will get captured), strategic effects, connection moves, etc. A large set heuristics is necessary here, e.g. to avoid duplication of such values.

4.4.2 Adding a Monte Carlo module to GNU Go

We try to solve the problem of not having a global view over the board by adding the Monte Carlo module. The functionality of this module is as follows.

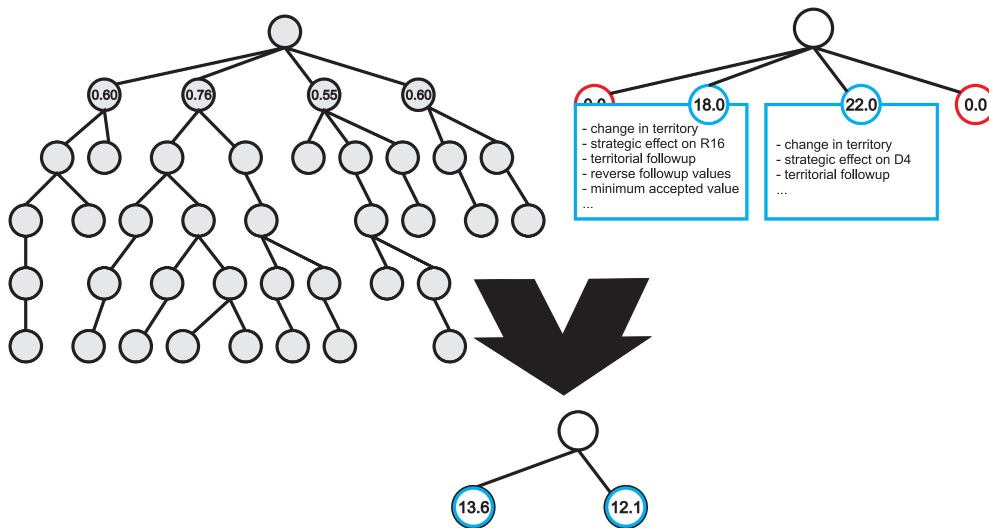

Figure 4.26: GNU Go evaluation combined with Monte Carlo: GNU Go makes its evaluation, generating reasons and summing them to obtain the final value. Monte Carlo, on the other hand, explores the UCT tree, finding winning probabilities. The results of the two are then combined, to give the final evaluation.

A separate thread runs random simulations during opponent time, exploring the UCT tree. Whenever a move is to be generated, the thread pauses, waiting for GNU Go's engine to generate a list of moves. Each of the moves has associated reasons summing up to a value estimating how good it is. After the list is generated, Monte Carlo resumes for a given amount of time, so that the confidence of its evaluation is good enough. Then again, it pauses. At this point, every available move has an associated winning probability. The next step is intuitive. For every move with positive score in the list generated by GNU Go we take its value and multiply it by its winning probability (see Figure 4.26). This way, moves considered good both by GNU Go and Monte Carlo are automatically chosen. Moreover, moves estimated by GNU Go to have the same local influence, which

in fact have different global importance in the game are overall ranked accordingly by Monte Carlo. Also, inherent errors which appear in Monte Carlo-only applications, due to lack of local precision, are eliminated thanks to GNU Go's *old-fashioned* Computer Go approach.

We added the Monte Carlo evaluation in `genmove()`, just before computation of the best value (see Figure 4.27).

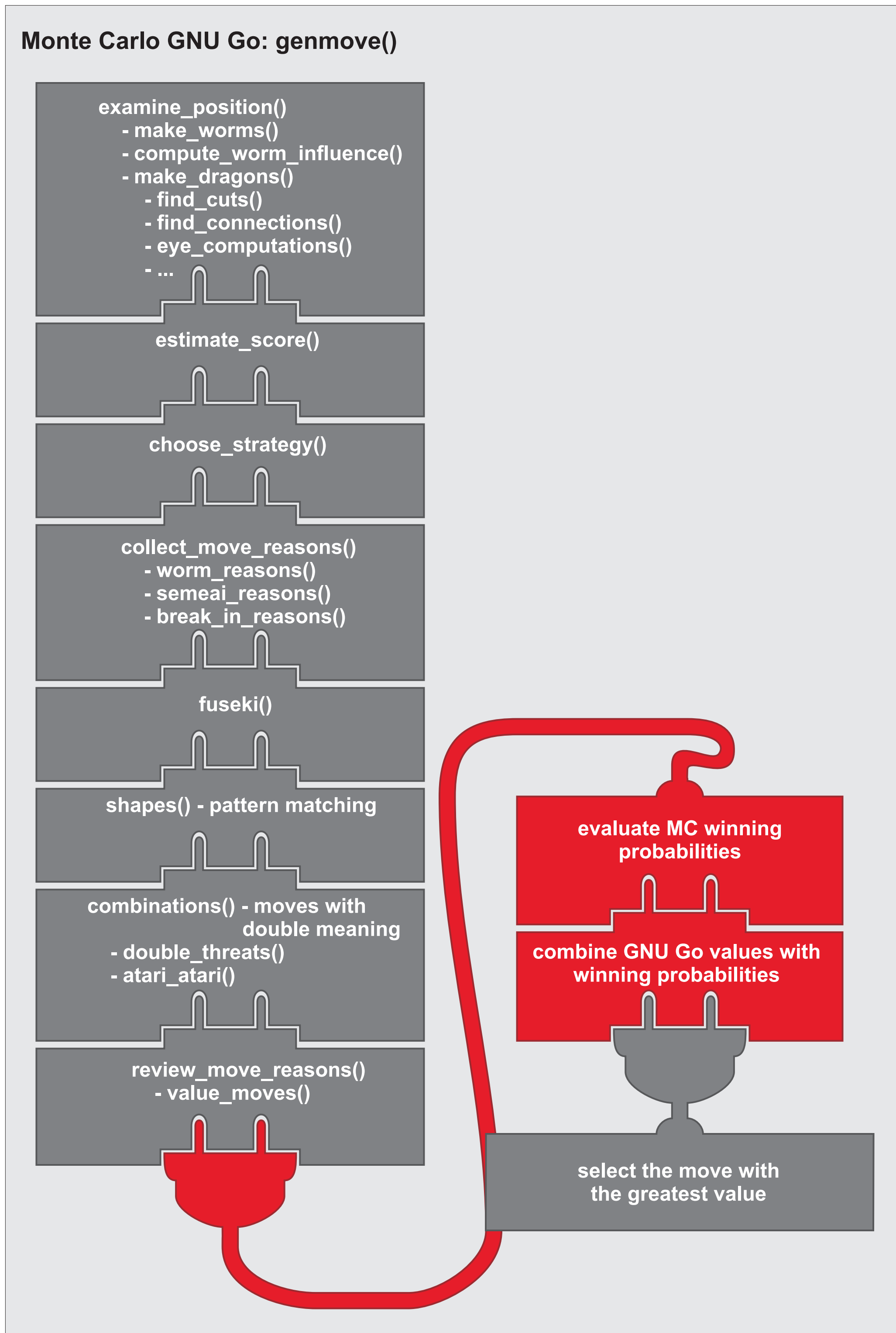

Figure 4.27: Integration of Monte Carlo in `genmove()`

Chapter 5

Results and conclusions

Our application was implemented in ANSI C, respecting the coding style and conventions of GNU Go 3.6, specified in [3], Section 4.6. The sources were compiled under Microsoft® Windows XP® Service Pack 2, using Microsoft® Visual Studio® 2005 Team Suite. For testing we used a system with an AMD® Athlon® XP 2700+ (2.17 GHz) CPU[1].

5.1 The random simulation

We have tested the speed of random simulations, comparing two of the presented algorithms, Naïve and Analyze-after. The tests were made on both of the CPU's mentioned in the introduction of the chapter, which we will call, for short, *Athlon* and *Pentium*[2]. Table 5.1, along with Figures 5.1 and 5.2 show a comparison of the two approaches on the Athlon processor, while Table 5.2 and Figures 5.3 and 5.4, on Pentium.

The results clearly show the speedup of the Analyze-after algorithm over the Naïve one[3].

[1] In some cases, we used an Intel® Pentium® III CPU (1 GHz). However, unless clearly specified, we will refer to the Athlon® CPU.

[2] Keep in mind that the Pentium processor we used is much slower than the Athlon.

[3] See Sections 4.3.1.1 and 4.3.1.3 for a complete analysis of the two algorithms and explanation of the results presented here.

Table 5.1: **Random Simulations**: Algorithm Performance Comparison - Athlon. The statistics are made over 100 tests for each algorithm, with 5 second running time per test, on the Athlon CPU. The values represent average numbers of simulations per second (in 5 second time). μ represents *mean value* and σ denotes *standard deviation*.

Algorithm	**Min**	**Max**	μ	σ
Naïve	90.99	98.19	95.41	1.38
Analyze-after	1028.4	1205.2	1058.62	38.47

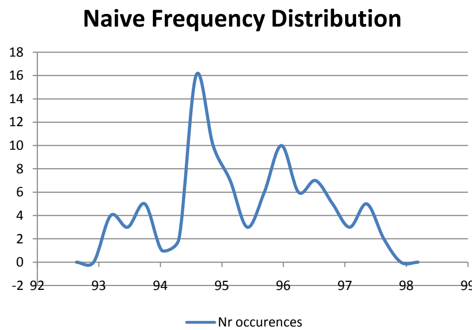

Figure 5.1: Random Simulations: Naïve Frequency Distribution - Athlon

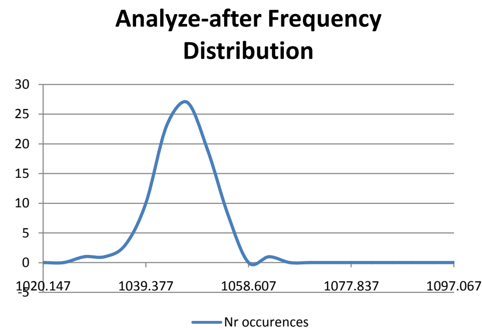

Figure 5.2: Random Simulations: Analyze-after Frequency Distribution - Athlon

Table 5.2: **Random Simulations**: Algorithm Performance Comparison - Pentium. The statistics are made over 100 tests for each algorithm, with 5 second running time per test, on the Pentium CPU. The values represent average numbers of simulations per second (in 5 second time). μ represents *mean value* and σ denotes *standard deviation*.

Algorithm	**Min**	**Max**	μ	σ
Naïve	40.87	44.46	42.84	0.78
Analyze-after	399.42	439.96	412.75	5.88

5.2 Monte Carlo GNU Go versus GNU Go 3.6

In the current section we present the outcomes of the games played by our program, called Monte Carlo GNU Go (or McGnuGo, for short) and GNU Go 3.6 (or GnuGo). These represent preliminary results, since the number of games tested so far is still small. However, we created a statistic based on this data, and we are able to explain different ups and downs of our approach, which resulted from these results. Also, we will suggest some possible enhancements to our work, which, we are confident, will improve the performance of our application.

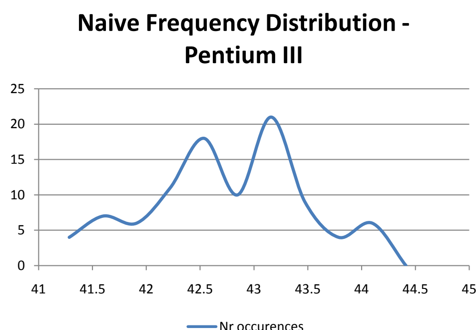

Figure 5.3: Random Simulations: Naïve Frequency Distribution - Pentium

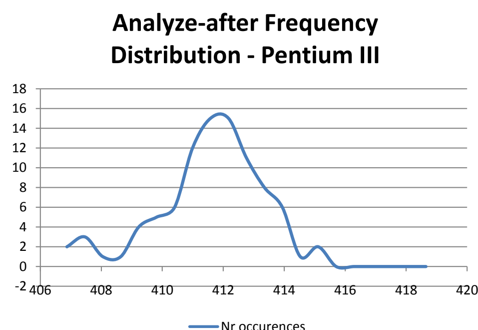

Figure 5.4: Random Simulations: Analyze-after Frequency Distribution - Pentium

Table 5.3 shows the results of the games played so far.

Table 5.3: McGnuGo vs GnuGo: We created a statistic over the values $\delta = score(McGnuGo) - score(GnuGo)$. Positive outcomes were counted as McGnuGo victories, while negative scores as defeats.

Win percentage as White	68.18%
Win percentage as Black	45.45%
Overall win percentage	60.60%
Average outcome (μ)	1.106
Standard deviation (σ)	24.54

We try to analyze first the problems present in our approach, leaving the advantages, which we thing come more straightforward, to the end.

The first thing that we noticed was that when playing with black, our program lost more games than it won, and especially, the results differed a lot from the ones recorded when playing with white. Another thing we noticed, was the fact that most of the times, when losing a little advantage at the beginning of the game, our engine started making bad moves, thus losing even more, eventually being defeated by a big difference. One last thing we were able to pick up from the tests so far is the fact that, whenever in a tactical situation, where the opponent is clearly in advantage, our program tends to favor moves extending territory somewhere else on the board, ignoring the thread. This eventually leads to losing the game.

We try to explain these disabilities in the following paragraphs.

One of the most important issues and reasons for wrong decisions is the fact that, in many cases, when speaking about random simulations, $average_outcome \neq$

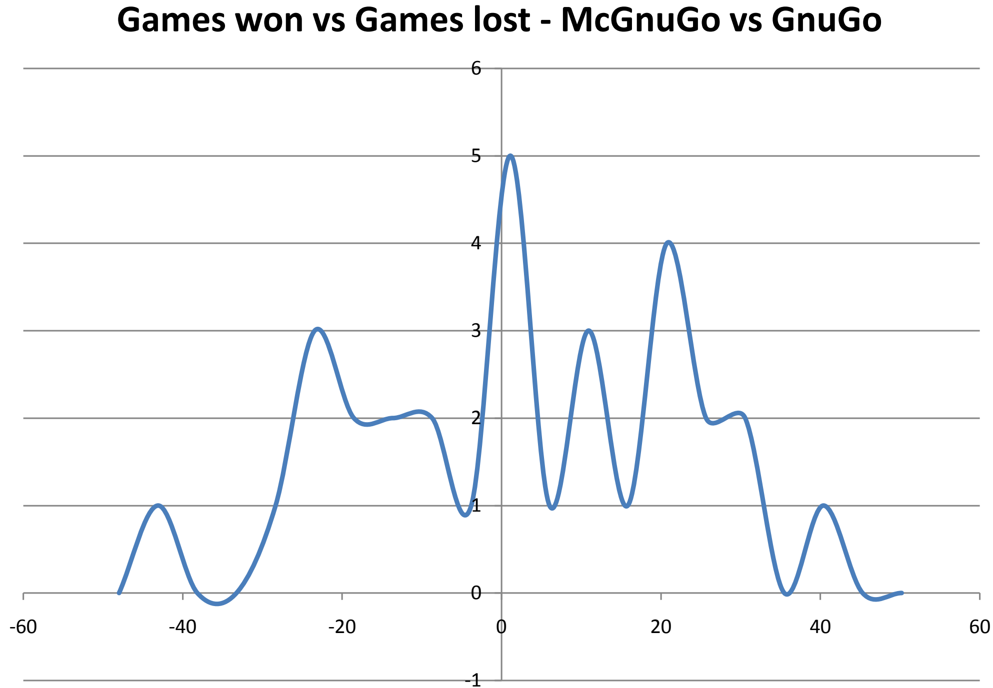

Figure 5.5: Distribution of δ in McGnuGo vs GnuGo: Positive values on the x axis represent McGnuGo wins, while negative values, defeats. The y values give the number of victories.

actual_outcome. In other words, most of the results yielded by random games are far from what even an average Go player would play, most of them having no sense at all. This makes the mean of the random simulations to get far from the actual situation on the board when a tactical situation shows up, and only get close to it when a clear territorial advantage is present. This explains the fact that McGnuGo ignores, in most cases, the strategic threat situations.

A potential fix we suggest is to use a temperature variable, similar to the one in the Simulated Annealing algorithm, which, taking account of previous plays and current evaluations, favors the moves advised by the GnuGo valuation over the ones advised by the Monte Carlo module. This variable starts with a higher chance of deciding over the move for Monte Carlo, and during the game, adapts to the actual situation.

Another issue, which we consider worth speaking about, is one concerning the fact that the outcomes of the random simulations, as far as Monte Carlo is concerned, come in two values: 0, for defeat and 1 for victory, no intermediate values. Whenever our program loses some advantage, most of the random simulations

start yielding 0, telling the program one thing: the game is most likely lost. So instead of trying to maximize his score, our program prematurely abandons hope. The same observation can be made when we are clearly in advantage, which causes McGnuGo not to make the difference between a potential bad move, and a better move.

The fix for this problem, we think, comes also by associating a temperature variable to Monte Carlo, helping it decide whether to make strict evaluations (0 and 1) or differentiated evaluations, depending on the actual score which emerged from the simulation. When the situation is balanced on the board, the variable would lean onto the 0/1 evaluation, while when in clear advantage/disadvantage, it would try to choose the moves yielding better scores rather than just wins/loses.

The inaccuracy of random games, associated with the abandoning hope, also explains, in part, the fact that black loses more than white: when playing randomly, the advantage given by the fact that black places the first stone is much smaller than the standard value of the *komi*[4]. This makes Monte Carlo believe that the game is lost, and acts as described earlier.

A solution which partially solves the problem is setting a score average for the last number of random simulations and, if the next score is over the average, decide it to be a victory, while if it is below average, defeat. This should make the program constantly choose the better moves most of the time.

Another solution to the inaccuracy of the random games is implementing a pattern-matching approach. See, for further reference, [13], [20], [9] and [11].

Still, most of the times, our Go player is an offensive one, trying to expand territory as much as possible, and moreover, to invade and capture enemy large zones. This gives the good results observed in Table 5.3. The balance between GnuGo's move choices and Monte Carlo's enhancements is acceptable and overall, we consider this to be a first step towards an excellent Go player.

[4]We used $komi = 6.5$.

Bibliography

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002. http://homes.dsi.unimi.it/~cesabian/Pubblicazioni/ml-02.pdf. [cited at p. 3, 40, 41]

[2] Jeffrey Bagdis. A machine-learning approach to computer go. May 2007. http://www.cs.princeton.edu/~jbagdis/jp.pdf. [cited at p. 14]

[3] Arend Bayer, Daniel Bump, Evan Berggren Daniel, David Denholm, Jerome Dumonteil, Gunnar Farnebäck, Paul Pogonyshev, Thomas Traber, Tanguy Urvoy, and Inge Wallin. *Documentation for the GNU Go Project*. Free Software Foundation Inc, 59 Temple Place and Suite 330 and Boston and MA 02111-1307 USA, 3.6 edition, July 2003. http://www.gnu.org/software/gnugo/gnugo_toc.html. [cited at p. 61]

[4] B. Bouzy and B. Helmstetter. Monte-carlo go developments. http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-helmstetter.pdf. [cited at p. 2, 19, 21]

[5] Bruno Bouzy. Mathematical morphology applied to computer go. http://www.math-info.univ-paris5.fr/~bouzy/publications/Bouzy-IJPRAI.pdf. [cited at p. 19]

[6] Bruno Bouzy. Old-fashioned computer go vs monte-carlo go. April 2007. http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf. [cited at p. 2, 14, 19, 44]

[7] Bruno Bouzy and Tristan Cazenave. Computer go: An ai oriented survey. http://www.ai.univ-paris8.fr/~cazenave/CG-AISurvey.pdf. [cited at p. 14]

[8] Bruno Bouzy and Guillaume Chaslot. Monte-carlo go reinforcement learning experiments. http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-chaslot-cig06.pdf. [cited at p. 4, 53]

[9] Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19×19 go. *IEEE Symposium on Computational Intelligence in Games*, pages 176–181, 2005. http://www.math-info.univ-paris5.fr/~bouzy/publications/bouzy-chaslot-cig05.pdf. [cited at p. 65]

[10] Bernrd Brügmann. Monte carlo go. October 1993. http://www.ideanest.com/vegos/MonteCarloGo.pdf. [cited at p. 2, 21]

[11] Rémi Coulom. Computing elo ratings of move patterns in the game of go. *Computer Games Workshop*, 2007. http://remi.coulom.free.fr/Amsterdam2007/MMGoPatterns.pdf. [cited at p. 65]

[12] Sylvain Gelly and David Silver. Achieving master level play in 9×9 computer go. 2008. http://www.cs.ualberta.ca/~silver/research/publications/files/MoGoNectar.pdf. [cited at p. 4, 53, 54]

[13] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. *Institut National de Recherche en Informatique et en Automatique*, (6062), November 2006. http://hal.inria.fr/docs/00/12/15/16/PDF/RR-6062.pdf. [cited at p. 3, 40, 41, 43, 45, 49, 50, 65]

[14] Thore Graepel, Mike Goutrié, Marco Krüger, and Ralf Herbrich. Learning on graphs in the game of go. *Computer Science Department - Technical University of Berlin and Berlin and Germany.* http://research.microsoft.com/~rherb/papers/graegoukrueher01.ps.gz. [cited at p. 33]

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. 2006. http://zaphod.aml.sztaki.hu/papers/ecml06.pdf. [cited at p. 40]

[16] Sensei's Library. About influence. http://senseis.xmp.net/?AboutInfluence. [cited at p. 20]

[17] Sensei's Library. Etymology of go. http://senseis.xmp.net/?EtymologyOfGo. [cited at p. 9, 20]

[18] Emanuela Mateescu and Ioan Maxim. *Arbori.* Ţara Fagilor, S.C. ROF S.A. Suceava - 5800 P.O. 1 Box 148 ROMANIA, 1996. [cited at p. 26]

[19] Livia Ralaivola, Lin Wu, and Pierre Baldi. Svm and pattern-enriched common fate graphs for the game of go. http://www.ics.uci.edu/~lwu/go.2005.ESANN.pdf. [cited at p. 33]

[20] David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of go. *Proceedings of the International Conference of Machine Learning*, pages 873–880, 2006. http://research.microsoft.com/~dstern/papers/sternherbrichgraepel06.pdf. [cited at p. 65]

[21] Wikipedia, the free encyclopedia. Bézier curve. http://en.wikipedia.org/wiki/B%C3%A9zier_curve. [cited at p. 47]

[22] Wikipedia, the free encyclopedia. Computer go. http://en.wikipedia.org/wiki/Computer_Go. [cited at p. 2, 14]

[23] Wikipedia, the free encyclopedia. Expected value. http://en.wikipedia.org/wiki/Expected_value. [cited at p. 19]

[24] Wikipedia, the free encyclopedia. Go (board game). http://en.wikipedia.org/wiki/Go_(board_game). [cited at p. 9, 13]

[25] Wikipedia, the free encyclopedia. Heap (data structure). http://en.wikipedia.org/wiki/Heap_(data_structure). [cited at p. 26]

[26] Wikipedia, the free encyclopedia. Multi-armed bandit. http://en.wikipedia.org/wiki/Multi-armed_bandit. [cited at p. 40]

[27] John Tromp. Number of legal go positions. http://homepages.cwi.nl/~tromp/go/legal.html. [cited at p. 14]

[28] Erik van der Werf. *AI techniques for the game of Go.* Datawyse b.v., Maastricht, The Netherlands, 2004. ISBN 9052784450 - Universitaire Pers Maastricht. [cited at p. 10]

Appendices

List of Figures

2.1 The board of Go . . . 10
2.2 A stone in atari . . . 10
2.3 A suicide situation. . . . 11
2.4 A *false* suicide move. . . . 11
2.5 A *Ko* situation . . . 11
2.6 A situation in which two advanced players pass. . . . 12
2.7 What happens if the two players don't pass. . . . 12
2.8 Example of *alive* groups. . . . 13
2.9 Example of *seki*. . . . 13

3.1 A cutting point situation . . . 18
3.2 Influence on the board of Go . . . 20

4.1 The Array List structure . . . 24
4.2 The Heap Array structure . . . 26
4.3 The Visited List structure . . . 27
4.4 Adding a value smaller than `min` to the Visited List . . . 27
4.5 Initialization in the naïve random simulation approach . . . 29
4.6 Iteration step in the naïve random simulation approach . . . 29
4.7 The board state structure . . . 30
4.8 The worm data structure . . . 30
4.9 A board position (FGR) . . . 33
4.10 Transformation of FGR to CFG . . . 33
4.11 The Worm Info structure . . . 39
4.12 The UCT node structure . . . 42
4.13 Pseudocode for UCT . . . 43
4.14 UCT: The beginning of the algorithm . . . 44
4.15 UCT: Legal moves are still available . . . 44
4.16 UCT: The node with the greatest UCB1 value is chosen . . . 44

4.17 UCT: The algorithm after a few iterations 45
4.18 The way a UCT tree grows 46
4.19 The way an alpha-beta tree grows 46
4.20 Intersections, according to Key Positions Priority 47
4.21 Key Positions Priority: Distribution 48
4.22 First-play urgency 49
4.23 Grandparent knowledge 50
4.24 The memory node structure 52
4.25 UCT Memory Tree: Saving 52
4.26 GNU Go evaluation combined with Monte Carlo 58
4.27 Integration of Monte Carlo in `genmove()` 60

5.1 Random Simulations: Naïve Frequency Distribution - Athlon 62
5.2 Random Simulations: Analyze-after Frequency Distribution - Athlon . 62
5.3 Random Simulations: Naïve Frequency Distribution - Pentium 63
5.4 Random Simulations: Analyze-after Frequency Distribution - Pentium 63
5.5 Distribution of results in McGnuGo vs GnuGo 64

List of Tables

2.1 Go rankings 13

5.1 Random Simulations: Algorithm Performance Comparison - Athlon . 62
5.2 Random Simulations: Algorithm Performance Comparison - Pentium . 62
5.3 McGnuGo vs GnuGo 63

Index

alive, 13
all-moves-as-first, 50
alpha-beta search, 44
Analyze-after random simulation, 34
Array List, 23

Bézier curve, 45
board state, 29

Common Fate Graph, 33

dan, 14
dead, 13
death, 13
dragon, 52

expected outcome, 19, 21
expected value, 40
exploration vs exploitation dilemma, 39
eye, 13

first-play urgency (FPU), 47
full graph representation, 33

Go, 11
grandparent knowledge, 48

handicap stone, 14
heap, 25
Heap Array, 25

influence, 20

jigo, 12

key positions priority, 45
kill, 13
knowledge bases, 20
Ko, 12
komi, 12

kyu, 14

life, 13

make alive, 13
minimax tree, 44
Monte Carlo, 19, 20, 28
move reasons, 53
Multi-armed Bandit problem, 39

naïve random simulation, 28

old-fashioned Computer Go, 19

random simulation, 21, 28
Rapid Action Value Estimates (RAVE), 51

seki, 14
stone, 12
suicide, 12

UCB1, 41
UCT Node, 41
UCT Tree, 41
uncertainty, 44
unsettled, 13
Upper Confidence for Trees (UCT), 41

Visited List, 26

Wéiqí, 11
worm origin, 30