

Monte Carlo Tree Search applied to Go

Reaching Ω

Dissertation presented by
Thibault VANDERMOSTEN

for obtaining the Master's degree in
Computer Science
Options: Artificial Intelligence and Software Engineering

Supervisor
Yves DEVILLE

Readers
Pierre SCHAUS, Francois AUBRY , Michael SAINT-GUILLAIN

Academic year 2015-2016

Dedication

To you as a reader, enjoy it.

To Google, who beat me close to the finish line.

Declaration

I hereby declare that this thesis is solely my original work. I have acknowledged all sources of information used in this thesis.

Acknowledgements

First of all, I would like to thank Pr Deville, my supervisor, for believing in me, pushing me to give my best and for his guidance during the year. I would also like to thank him for having aroused my interest for artificial intelligence during my studies.

Then, I would like to thank my readers Francois Aubry and Michael Saint-Guillain for their help, notably with the redaction of this paper.

Also, I would like to thank my family for their continual support, notably my sister, Alice, for her patience and motivation to help me.

Finally, I would like to thank my friends and more specifically Maciek Skoniescka for having my back during this year and for his advice.

Abstract

This thesis is about designing an artificial intelligence Go player based on Monte Carlo Tree Search, or MCTS, techniques, with the lower bound complexity. Go is a strategy board game with a high complexity, having 10^{600} possible games. For this reason, artificial intelligence applied to Go is a challenging field, where progress can still be done. Before the recent match of AlphaGo versus a worldwide champion of Go, MCTS was considered as the state of the art and performed better than other methods. As the quality of the results of MCTS is proportional to the number of iterations, it is important to have the fastest iteration possible, leading to a more exhaustive search for an amount of time. This is the main focus of this master thesis.

It was a success by making use of specific data structures for the Go board representation, resulting in the end in an optimal model. These structures are detailed and explained in this master thesis, as well as the way their combination allows to reach an optimal complexity.

In addition to giving an implementation that reached Ω , we expose one flaw of the core basic technique of MCTS when used alone. Moreover, we find and examine some leads. On the one hand, some of them correct the mentioned flaw, and on the other hand, some could be used in general to improve MCTS.

Contents

1	Introduction	8
2	Basics	10
2.1	Rules of Go - [1]	10
2.2	Computational Complexity - [2]	12
2.2.1	Basic Example	12
2.2.2	Several Types of Complexity	12
2.3	Monte-Carlo [3] [4] [5] [6]	13
2.3.1	Multi Armed Bandit Problem	14
2.4	Tree Search - [7]	15
3	MCTS : State of the art	18
3.1	Monte Carlo and Tree Search combined	18
3.1.1	One iteration of the MCTS algorithm	18
3.1.2	Basic Strategies for each step - [3]	19
3.1.3	Benefits and Drawbacks	22
3.2	Monte Carlo Tree Search applied to Go	23
3.2.1	MC-RAVE, Monte Carlo Rapid Action Value Estimation - [8], [9]	23
3.2.2	Bradley and Terry model - [10]	24
3.2.3	AlphaGo - [11]	25
4	Naive version	27
4.1	General Monte Carlo	27
4.2	Go Methods	27
4.2.1	Clone	27
4.2.2	GetMoves	28
4.2.3	DoMove	28
4.2.4	GetResult	29
4.3	Total Complexity	29
5	Optimization	30
5.1	Abstract Representation of Group	30
5.1.1	Presentation of Union Find	30
5.1.2	Using Union-Find for Go	31
5.2	Ko	32
5.3	Dynamically Maintained List	32
5.4	Numerical Liberties	34
5.4.1	Main Idea	34
5.4.2	Validity of the Model	34
5.4.3	Maintaining the List of Possible Moves	35
5.5	Conclusion	35

6	Implementation	36
6.1	General Monte Carlo	36
6.1.1	Game State	36
6.1.2	Node Class	37
6.1.3	Simple Pseudo Code	37
6.2	Overall Program	37
6.2.1	End Game	39
7	Experiments and results	41
7.1	General	41
7.2	Start of Game	41
7.3	Tsumego	43
7.4	Hard tsumego problem, symptom of the main flaw of V-Run	43
7.5	Majority Voting	44
7.6	Binary Victories	47
7.7	Complexity	47
8	Conclusions	49

List of Figures

1.1	Boards of games played by V-Run (black) against an amateur level player	9
2.1	Liberties [12]	10
2.2	Group with two eyes	11
2.3	Scoring	11
2.4	Ko : White cannot immediately play in the cross	11
2.5	Monte Carlo π estimation [13]	14
2.6	Tree of tic tac toe	16
3.1	MCTS tree	19
3.2	Selection step	19
3.3	Expansion	20
3.4	Backpropagation	20
3.5	Performance of Go AI - [8]	23
3.6	Simulations for RAVE	24
5.1	Before and after Find(4)	31
5.2	Before and after Union(2,5)	32
5.3	The group of stones a,b,c remember having respectively 4,6 and 8 liberties	35
6.1	Interface between bot and GUI	38
6.2	Infinite game	39
7.1	Games played by V-Run(black) against a GoFree bot(white)	42
7.2	Heatmap of an empty board: number of times the move was picked out of 1000 searches	42
7.3	Heatmap of an empty board: number of times the move was picked out of 1000 searches	43
7.4	Different level of tsumego	43
7.5	Tree for the hard tsumego	44
7.6	Why 3,2 is found as a valid move	45
7.7	Greedy move picked	45
7.8	reaching a depth 2 with 3 possible moves	45
7.9	Tuning of C parameter in UCT formula	46
7.10	Time complexity test depending on i iterations	48
7.11	Time complexity test depending on n^2 the size of the board	48

Chapter 1

Introduction

Artificial intelligence(or A.I) aims to understand and produce intelligent behaviour. It is a large field of research already present all around us. A.I helps us automatize tasks, predicts diseases, talks with humans or beats us at our own games. It is a developing filed where much has still to be achieved. Birth is given to new A.Is everyday and they are continually improved. Self driving cars (Google and tesla cars), personal assistants (Cortana), diagnosticians (Watson [14]) or lawyers (Ross [15]) are only a fraction of what A.Is are able to do. Unlike some scientific branches, where one could feel a lot was already done, the domain of A.I has still room for an Einstein; has plenty of challenges that still need to be explored and cracked.

The game of Go was one of them for a long time. It is a 4000 years old strategy board game. Contrary to chess, where the best human player was beaten by Deep Blue in 1999, human masters of Go remained undefeated till recently. The main problem of Go was its complexity. A game of Go lasts longer than a game of chess, and has more or less 200 moves available at each turn (37 for chess). There is approximately 10^{600} possible game of Go and 10^{200} games of chess. The techniques used by A.I for chess failed on Go because they needed far to much time in order to be effective. The A.I players couldn't get through amateur level.

Then came the day when Monte Carlo Tree Search (or MCTS) appeared. It started in 1987, but was only really applied to Go in 2006, known now as the year of the Monte Carlo revolution in Go. This technique allowed the A.I to beat low level professional. It blends two methods. The first one is the Monte Carlo method, that estimates values with random data samples. It was then merged with a standard A.I method, the tree search. The standard tree search failed on Go, because the tree of possibility was to large to be computed in reasonable time. Monte Carlo Tree Search begins to explore the tree at random, one path at a time, because at first it doesn't know anything. Then the more knowledge is found, the more it tries to exploit this knowledge. Thus, searching and exploring the tree only where it finds it interesting. In addition to the core method that was refined a lot in the last 10 years, there were additional heuristics found that complement it nicely.

Then, this year came another Go revolution. Indeed, the world champion of Go was beaten by an A.I, DeepMind AlphaGo ([16]) in the end of march 2016. AlphaGo had already beaten the European champion of Go in the end of 2015, and was victorious against the other top level A.I 99.8% of the time. It was truly an incredible breakthrough that would have never been expected for at least the coming ten years. The most incredible part of it, is that DeepMind is not really programmed to play a specific game. It is programmed to learn, thanks to neural networks(a machine learning method), from simulation and other recorded matches. Then it is continually improved by playing against itself. Go AI were stuck to the problem : "How to search a space far too big for our computing ability". Where MCTS tried to find an answer,

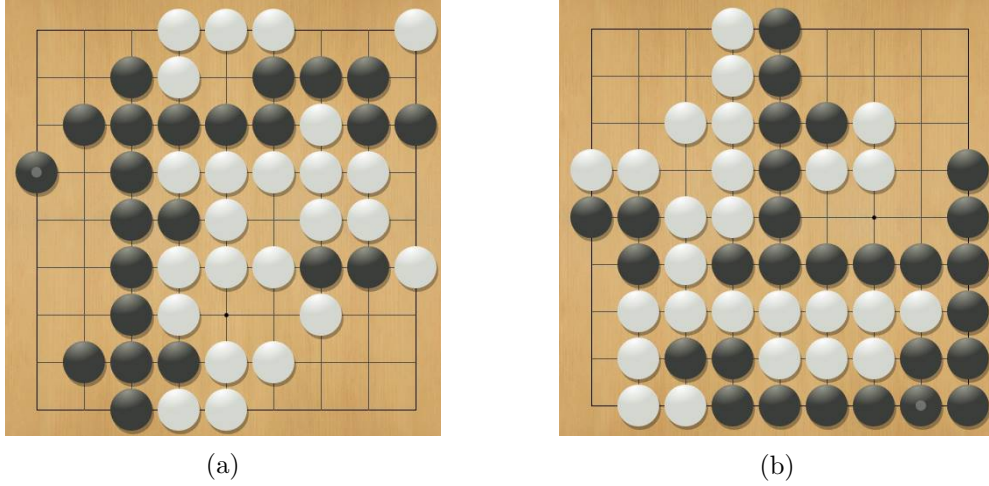


Figure 1.1: Boards of games played by V-Run (black) against an amateur level player

AlphaGo took a totally different route and outperformed it.

The goal of this master thesis was to design an artificial intelligence player able to play Go thanks to Monte Carlo Tree Search technique. This A.I computer program, or bot, named V-Run, went through several stages of development. At the very first, it was bad in terms of strength of plays but also in terms of time taken. But, thanks to various improvements, presented one by one, its time complexity progressively improved and went from something impractical to the lower bound, the best complexity reachable. The final version is more than 7500 times faster than the original one. It is really useful because a faster search means more precision for the same time used. Furthermore, to my knowledge, there is no actual public implementation model of MCTS for Go with the lower bound complexity. In the end, V-Run does not only perform the fastest it could, but it also performs well (figure 1.1). It reached the level of an amateur, succeeding to make intelligent plays when it only has a core technique that wasn't nor refined, nor upgraded with heuristics or database.

Nevertheless, there is still room for improvement: one of the possible upgrade presented is majority voting. It aims to exploit the randomness with which MCTS functions. We show some leads that could yield promising results.

Chapter 2

Basics

This chapter contains the basic knowledge needed to fully understand this master thesis. It starts by describing the rules of Go. Then, it explains the concept of computational complexity. Finally, it ends by explaining the two main concepts of MCTS, Monte-Carlo technique and Tree search methods.

2.1 Rules of Go - [1]

The rules of Go are extremely simple, and their simplicity is exactly what allows a real complexity and depth of play. The game is played on a so-called go-ban, which is a board composed of 9*9, 13*13 or 19*19 intersections. There are two players, one who plays the white stones, the other playing the black stones. Each player places a stone on an intersection of the go-ban alternatively.

They are two important concepts to understand, the liberties and the groups. A stone has as many liberties as there are free intersections next to it. In figure 2.1a, the black stone has initially four liberties, and it progressively decreases to only one liberty as white stones are added next to it :

Two or more stones can fusion together to form a group in which they share the liberties. In figure 2.1b, we can see three groups, one black and two white. The black one has five liberties and each white group has four liberties.

An important concept is the capture, the board being representative (as a lot of ancient board games, like chess e.g) of a real situation of war. If a group is surrounded by the opponent, it has no more liberties, and it is considered as dead. All the stones of the group are taken as prisoners, leaving free the intersections where they used to be. When a group of stones has only one remaining liberty, we say that the opponent put it in atari, meaning threatens to capture it at his/her next move. In figure 2.1aD, the black stone is in atari. A group is said safe, if it cannot be captured. This is when a group has at least two eyes, two single separate empty



Figure 2.1: Liberties [12]

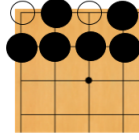


Figure 2.2: Group with two eyes

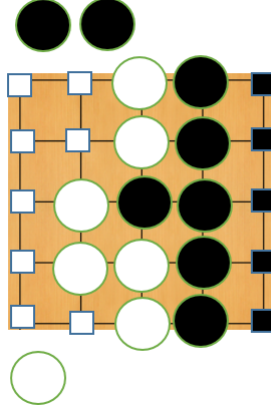


Figure 2.3: Scoring

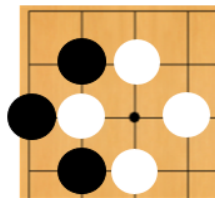
intersections surrounded by the group. To be killed, the opponent should play two moves at the same time. As it is impossible, the group is sure to be safe. In the figure 2.2, the white player should play at the same turn in the two circled intersection to capture the black stones group, after having surrounded the outside of it.

In order to win, a player must have more points than its opponent. A player gains one point for each prisoner he has captured and one point by territory. A territory is a free intersection surrounded at least partially by your stones and potentially by the sides of the board.

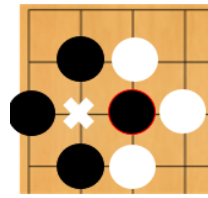
In the example in 2.3, the white player wins with 10 points (8 from territories and 2 from prisoners) against 6 points for the black player (5 from territories and 1 prisoner).

To prevent Go games from going infinite, the ko rules forbid any play that recreates the last board position. So if A and B are two distinct states of the board, the game cannot follow a path $A \rightarrow B \rightarrow A$, see figure 2.4.

The official ranking system for Go players is described in table 2.1.



(a) A



(b) B

Figure 2.4: Ko : White cannot immediately play in the cross

Table 2.1: Go rankings- [17]

Ranks	Stage
30-21 kyu	Beginner
20-11 kyu	Casual
10-1 kyu	Intermediate amateur
1-7 amateur dan	Advanced amateur
1-9 professional dan	Profesional player

2.2 Computational Complexity - [2]

Complexity is a concept in computer science that allows for an estimation of the space or of the time taken by a program depending on the size of the input data. There are thus two kinds of complexity. Spatial complexity aims to predict the space taken in memory by an algorithm. Time complexity, on the other hand, estimates the number of operations needed for an algorithm to be finished.

2.2.1 Basic Example

For each problem, there are several answers and rarely one optimal solution. Often, some solutions are better in terms of time while others perform better in terms of space complexity, or of other criteria.

Given a desktop with n drawers, a pen in one of them and the goal is to find a pen in particular. One solution is to open each drawer looking for the pen. It will take at worst n operations. But, if at the moment of putting the pen in the drawer, we leave a note mentioning which drawer contains the pen, we would later directly open the right drawer, thus making only one operation. The second way is faster than the first one, because the time taken to solve the problem is constant and thus does not depend on the number n of drawers.

2.2.2 Several Types of Complexity

We can estimate a complexity in different ways. By the possible input :

- Worst case complexity : Complexity of an algorithm, given the worst possible input for it. In the first method of the previous example, the pen would be in the last drawer, leading thus to n operations.
- Best case complexity: Complexity given the best possible input. In the first method of the previous example, the pen would be in the first drawer, the complexity will be 1.
- Average case complexity: Complexity of solving the algorithm on average. It takes into account the probability of the input. For the first method, if the placement of the pen follows a uniform distribution, on average finding the pen would take $\frac{n}{2}$ operations.

And by its bounds:

- $\mathcal{O}$ represents the upper bound complexity : the real complexity is lower or equal to this one
- Ω represents the lower bound complexity : the real complexity is greater or equal.
- Θ represents when the lower and the upper bound are identical.

Amortized complexity It is the actual complexity of a method, evaluated over a sequence of operations, as the worst complexity can be too pessimistic. Suppose a bag with n element. We have a remove operation, taking at max n elements from the bag. If the operation is repeated i times, our complexity is $\mathcal{O}(i * n)$. But we know for sure that we can't take more elements than what was already in the bag, the amortized complexity for i removal is $\mathcal{O}(n)$.

As the complexity is only an estimation, it helps us knowing a bound on time or space taken. As a consequence, the complexity of an algorithm can be simplified of all constant factors, considering only the largest one. E.g : $2n^2 + n$ becomes n^2 , $3n \log n + n + d$ becomes $n \log n + d$.

Here's the average complexity for some operation on the data structures used later (n is the actual size of the data structure, “-” means that the operation is impossible within that data structure) :

Data structure	Access	Search	Insert	Remove	Union	Get a random element
Array	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n + m)$	$\mathcal{O}(1)$
Linked list	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Set/hashmap	-	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(n + m)$	-

*expected

2.3 Monte-Carlo [3] [4] [5] [6]

Monte Carlo methods consist of a big set of algorithms that works by using a repeated random sample to estimate a solution. They are widely used in a lot of domains such as physics, computational biology, computer graphics, computer science, artificial intelligence, and even economics.

Monte Carlo methods work by sampling a sufficient amount of data to estimate with enough precision a value. To give a very simple example, if the aim is to find the mean value of a six sided dice, it samples a lot of values between 1 and 6 and finds the mean of all these values. With sufficiently high amount of samples, it will tend to 3,5.

There are four steps in a Monte Carlo algorithm:

- 1: Define a domain that will contain all the possible values
- 2: Sample random values over the domain following a probability distribution. Stop when there are enough values according to the level of precision required.
- 3: With the data sampled, compute results using theses values.
- 4: With the results found, deduce other variables of importance using equations or theorems (depending on the domain of application).

We show this four steps through an example, the computation of π or how to estimate π with a random generation Monte Carlo method.

- Define a domain of inputs: the square of coordinates $(-1,-1),(-1,1),(1,1)$ and $(1,-1)$.
- Generate random values over the domain following a probability distribution: Generate points following a uniform distribution in the square.
- Compute the result: compute x where $x = \frac{In}{Total}$ and where $Total$ is the number of points sampled and In is the number of points which fall within the circle of radius 1.

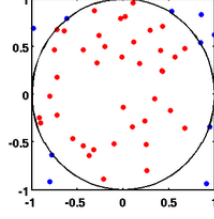


Figure 2.5: Monte Carlo π estimation [13]

- Use the result to get what you wanted: with a sufficiently high number of $Total$ we can approximate π thanks to the law of large numbers :

$$\frac{In}{Total} \simeq \frac{\text{area of the circle}}{\text{area of the square}}$$

As the area of the circle is $\pi * 1^2$, and the area of the square is $2 * 2 = 4$, we get our final approximation:

$$\frac{In}{Total} \simeq \frac{\pi}{4}$$

Results have shown that with a large enough number of samples, the estimation can get really close of the real value of π ([3]).

2.3.1 Multi Armed Bandit Problem

Multi armed bandit problem is a type of problem where one must choose one element amongst n , knowing that it wants to maximize its gain, and that each element yield a random reward following a probability distribution.

Given a strategy A, the regret (rg) of this strategy is the expected maximum reward minus the actual gain.

$$rg = N\mu^* - \sum_{n=1}^N r_n$$

- N is the number of rounds
- μ^* is the maximal reward mean
- r_n is the actual reward gained at round n

The goal is to minimize the regret, thus to optimize the gain. But trying to achieve this aim, a dilemma arises: exploration versus exploitation.

Exploitation is capitalizing on data already acquired and doing what is thought to be the optimal action while exploration is trying to check if there isn't any other course of action more efficient.

UCB strategy is used to balance the trade-off. Knowing that each element yields a reward that is a random variable, UCB compares the elements according to their upper confidence bound, the best value expected of an element.

$$UCB_i = \bar{X}_i + k\sqrt{\frac{\ln n}{n_i}}$$

- UCB_i is the upper confidence bound of the element i .
- n is the total number of plays until now.
- n_i is the total number of plays on i until now.
- $\bar{X}_i$ is the average reward of i . This value becomes progressively closer to the real mean reward of i while n increases. It is the term encouraging the exploitation.
- k is a constant parameter tunable. Basically, $k = \sqrt{2}$.
- $\sqrt{\frac{\ln n}{n_i}}$ is a representation of the deviation of the reward of i with respect to the estimated mean $\bar{X}_i$. This is the term encouraging exploration. Each time we choose the element i , this factor decreases :

$$\frac{\ln n}{n_i} > \frac{\ln(n+1)}{n_i+1} \text{ for } n > 1$$

While for each other element j the factor increases :

$$\frac{\ln n}{n_j} < \frac{\ln(n+1)}{n_j} \text{ for } n \geq 1$$

2.4 Tree Search - [7]

In this section we explain the basics of tree search in artificial intelligence. We present notably the mini-max algorithm which is widely use in artificial intelligence and we expose its drawbacks.

A tree is an abstract representation of a game. Each node represents a state of the game. The root is the initial state. Then there is one child for each possible move from that state.

Some assumptions need to be made:

- Game played turn by turn, not in real time.
- Two players.
- Fully observable

Figure 2.6 is an example of tree for the game of tic-tac-toe. Each player can alternatively take a decision. In our example, the cross player chooses at even levels(0-the root,2,4...) and the circle player at odd levels. The tree is not showed exhaustively due to space reasons.

Mini-max

The mini-max algorithm, see algorithm 1, is an algorithm used in artificial intelligence games. This algorithm associates to each state (or node) of the game a score. One player tries to maximize it while the other tries to minimize it. E.g :a greater score is a better situation for player one. Each action or move impacts the score. Therefore, the algorithm chooses the action that eventually brings it into the best state, assuming the other player plays the best moves it can. If possible, the algorithm continues to search the tree until a situation where the game has ended. Unfortunately, there is not always enough time to go until the end, and the algorithm must thus limit itself to a certain depth.

Here is an example where the algorithm begins with the player seeking to maximize the score and with a depth limit of two.

At first, the algorithm has no knowledge:

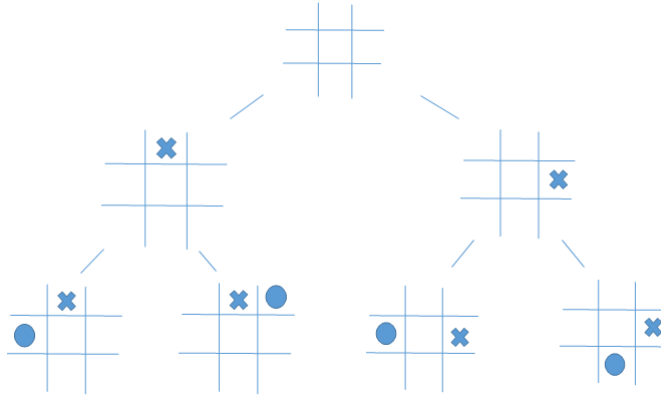
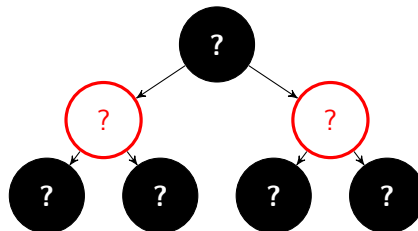
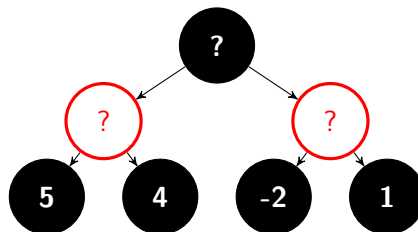


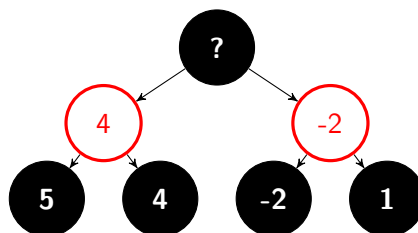
Figure 2.6: Tree of tic tac toe



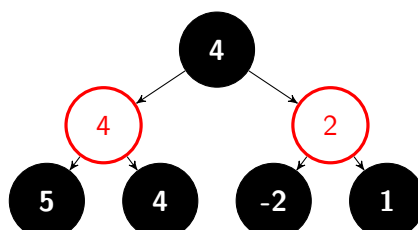
Then the values of the leaves are evaluated.



Thereafter the algorithm computes the value for the red node, knowing that the player is looking to minimize the score. Thus, the algorithm assumes that the player will choose at each time, the minimal node possible.



Finally, the algorithm finds the value of the root node.



This is the basis principle of a minimax search. There are several benefits to this technique but it also suffers some major drawbacks.

- Mini-max needs to explore in breadth first search, meaning all nodes from the top, depth after depth. In some cases, this is just impossible. The complexity of the minimax is $(b^d) * e$ where b is the branching factor (number of possible moves), d is the cut off depth, and e is the complexity of the evaluation function. In Go, the initial branching factor is 361 and the mean duration of a game is 200 plays. With such a complexity, 361^{200} , it's impossible to compute the tree in reasonable time. Using a cut off depth is a possibility, but it would be so small (due to the time constraint) that the tree won't be deep enough to make any interesting plays.
- The algorithm needs an evaluation function which returns a certain score for each node. It requires a deep knowledge of the game to produce a correct and precise evaluation. In Go, providing a score to a board or even assessing which player is leading is sometimes practically impossible. The issue of the game can change depending on a single move.

Algorithm 1: minimax function with a cut off depth

Input: node, depth, maximizingPlayer
Output: returns minimax value for the node

```

1 if depth == 0 or node is terminal then
2   | return node heuristic value
3 if Maximizing player then
4   | best =  $-\infty$ 
5   | for each child of node do
6   |   | best = max(best, minimax(child, depth-1, False))
7   | return best
8 else
9   | best =  $+\infty$ 
10  | for each child of node do
11  |   | best = min(best, minimax(child, depth-1, False))
12  | return best

```

Chapter 3

MCTS : State of the art

This chapter explains Monte Carlo Tree Search, step by step, through one iteration. Then it details the basic strategies for each step. Finally it ends by describing the actual performance of state of the art programs, and some of the most used heuristics.

3.1 Monte Carlo and Tree Search combined

We show how to combine these two principles to form the Monte Carlo Tree Search(MCTS), The idea is relatively simple. As the total tree of the game is far too big, MCTS computes it step by step, trying to explore only interesting parts. MCTS computes the tree node by node, with information on what nodes build next. In the end, there is a far smaller tree than the exhaustive one, indicating the best move possible.

3.1.1 One iteration of the MCTS algorithm

This subsection presents how one execution of the MCTS occurs, through four distinct steps. As these steps were called differently in the literature, we mentioned them both to help the reader. We begin with a tree already partially built(3.1). There are two colors, one for each player. The first number in the nodes is the number of times the owner of the root node won passing by that state¹. The second is the number of times the algorithm passed by that state. E.g : the leftmost leaf, where we can see that the black player won zero times on two tries.

1. **Selection(descend)**: The first step is the selection, the algorithm selects the best node following a specific criterion until a leaf is reached. This criterion can vary, and the different strategies concerning its choice will be explained later. For the example, this criterion is simply the best ratio won/played. The chosen moves are highlighted in a diamond shape(figure 3.2). First, the best move for black is chosen, 4/6 is better than 1/4. Then the best move for white is chosen. Remember that the value in the node are the chances of winning for the root player, here black. Thus the best move for white is the node with 0/2.
2. **Expansion(growth)**: Once the selection phase reaches a leaf node of the tree, the expansion phase starts. If the leaf is not a terminal state of the game, the basic strategy is to select a random move and to add it to the tree with null statistics (as shown on figure 3.3).
3. **Simulation(roll-out)**: The simulation step plays until the end of the game with random moves for both players.

¹In programming, as the technique works, the value in the node must be relative to the player. As it concretely does not change anything, we allowed ourselves to standardize in the example for clarity, and showing only the victory chances relative to the root player.

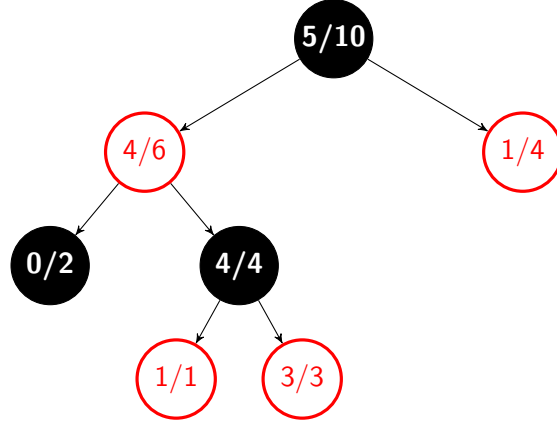


Figure 3.1: MCTS tree

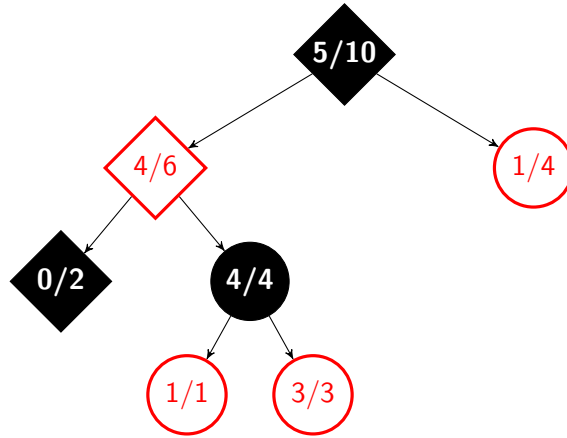


Figure 3.2: Selection step

4. **Backpropagation(update)**: Once the game is ended, the algorithm determines the winner. After, it can update the nodes on the path taken, depending on the result. In our example, the black player won, so the values of all the diamond nodes will be updated accordingly (see figure 3.4).

The execution of one iteration is now finished. There are two possibilities, either time is not over, and the algorithm goes on, or time ran out, and the algorithm must return a move. In the last case, it returns a node following a criterion. With the criterion of the best ratio won over played, it would return the move on the left. Indeed, by choosing that move, the algorithm has 5 out of 7 chances to win, where in the other case, it would have a smaller probability to win.

Search only where it's promising and take the best decisions according to the results, such is the principle of MCTS. With a number of iterations high enough, the built tree becomes statistically representative, allowing thus to find interesting moves.

Now we explore the different strategies possible for each of the steps detailed before.

3.1.2 Basic Strategies for each step - [3]

There are several strategies for each step. To fully understand the strategies for the selection step, a big dilemma in the application of MCTS must be acknowledged. The problem is between

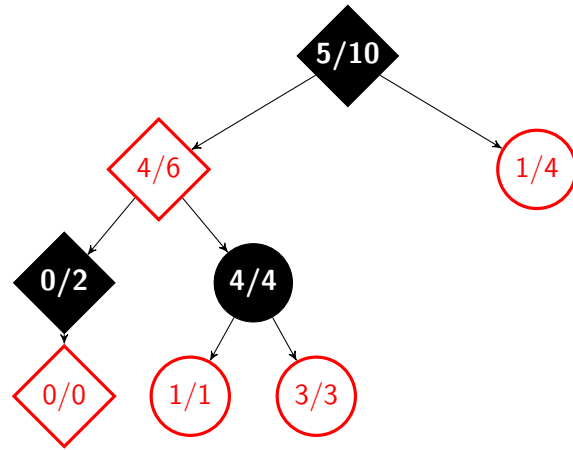


Figure 3.3: Expansion

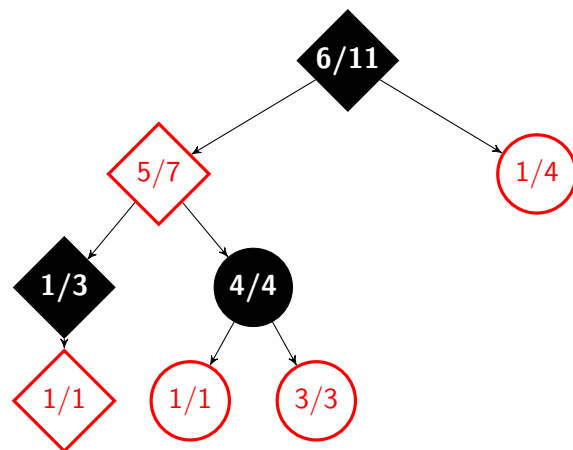


Figure 3.4: Backpropagation

two elements : Exploitation, taking profits of what you already know, and Exploration, checking elsewhere for a better possibility. When you exploit, you cannot explore and reciprocally. Therefore there is always the following question : “Am I on the right path ? Should I go further on it, or should I try something else?”.

Selection - UCT

The UCT technique, or upper confidence bound applied to trees, is an easy and simple strategy to answer the dilemma of exploitation versus exploration. It’s the adaptation of UCB technique to trees, described in Chapter basics, Monte carlo.

Starting from a node p , we consider I child nodes. The UCT formula will select the node i which has the maximum value :

$$V(i) = \frac{w_i}{v_i} + \sqrt{C * \frac{\ln v_p}{v_i}}$$

- w_i is the number of wins of that node.
- v_i is the number of visits of node i .
- C is a parameter, tuned experimentally but usually set to 2.
- v_p is the total number of visits for the parent node p .

The idea behind this formula is to estimate the value with a deviation. It selects the node i with the highest $V(i)$. Then over the visits on i , $\frac{\ln v_p}{v_i}$ decreases. Thus $V(i)$ decreases while $V()$ increases for the all other nodes (for another node j , v_p progressively increments while v_j stay the same). After exploiting the node i , another node will potentially become more interesting. As a consequence, the exploration come into balance, and the search selects another node to exploit.

Expansion

There exist strategies where more than one node is added by expansion step to the MCTS tree. Nevertheless, those strategies proved not to be more effective [3] and here we will consequently keep expanding one node by iteration.

Simulation

For this step, there are several techniques varying between two majors methods: choose a move at random or with heuristics.

	Random	Heuristics
Benefits	Speed	Reflect real play
Drawbacks	Non representative	Order of magnitude slower

In our case, to simplify the situation, we choose the random method, that allow for the fastest iterations.

Backpropagation

Again, for this step several strategies exist, but as none proved to be more efficient [3] than simply keeping the average value V for each node, knowing that N is the number of games and R_i is the result for the game i (-1 in defeat, +1 in victory):

$$V = \frac{\sum_i^n R_i}{N}$$

Selection of the final move

There are several methods to choose the move after the MCTS tree is built.

Here are a few examples:

- Max child: Select the move with the highest count of wins divided by the counts of visits.
- Robust child: Select the move with the highest visits count known as Robust child.
- Robust-Max child: Select the move with the both highest wins and visit count.

As the goal of this master thesis is not the exploration of the efficiency of the different strategies, our MTCS A.I was built using only the basic strategy for each step. Which means basic UCT selection, random moves for the simulation step, and the robust child selection for the final move.

3.1.3 Benefits and Drawbacks

Benefits

MCTS draws a lot of advantages from its special technique of using tree search and random simulations.

- Aheuristic: There is absolutely no need for heuristic in the core version of MCTS. All that is necessary to know about the game are the possible moves from a state, and the winning situations for a player. Except from that, which are the basic rules, there is no need to know how to play, to be a professional or to have a really good vision of the game to implement an artificial intelligence player.
- Asymmetric: Instead of having a gigantic tree, which would take an incredible amount of space, there is only the needed tree, significantly smaller than the complete one. As the tree is built only depending on the needs, it can be completely asymmetric.
- Any time: One of the beautiful parts about MCTS is that no specific amount of time is needed. There is always an output. Some algorithms systematically take a certain amount of seconds or minutes before delivering an answer. But here, the algorithm can always, at any time, return an answer. Of course, the more time is given, the better could be the answer. This system allow to stop the search either after a certain time or number of iterations.
- The basic version of MCTS can be implemented and coded easily.

Drawbacks

Here comes the second part about the big dilemma mentioned earlier. There are several difficult choices. For example, if the simulation heuristic is too complicated, more time is necessary, leading to fewer iterations and statistically unrepresentative results. On the other hand, if it

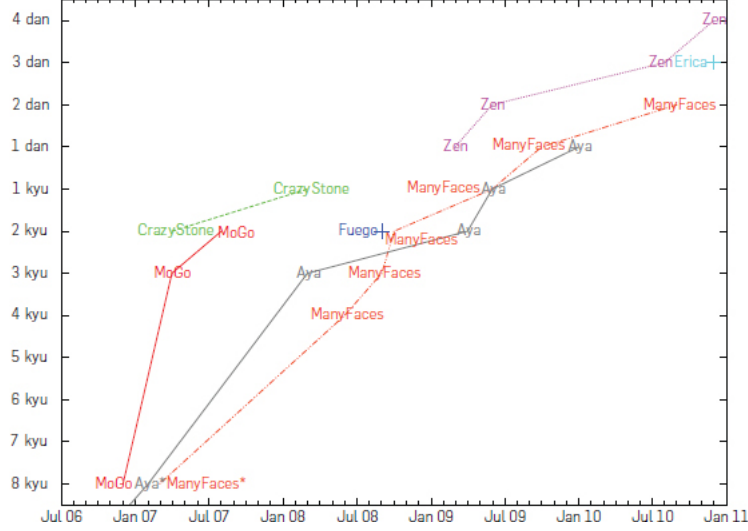


Figure 3.5: Performance of Go AI - [8]

is too simple, the simulations won't reflect real plays and even with a lot of those, won't be representative, again.

There are heuristics, added to the core method of MCTS, that improve dramatically the results. They will be explained in the next section.

3.2 Monte Carlo Tree Search applied to Go

Before the introduction of Monte Carlo, the progress in the field of artificial intelligence in Go was stuck. The programmers couldn't make a bot player better than an average good human player. Then, with Monte Carlo, all bots reached radically higher levels, as shown on figure 3.5. To clarify, a 8 kyu Go player corresponds to a good amateur player, and a first dan player to a very good player, approaching professional level (see table 2.1).

We present here two heuristics for MCTS in Go, we then finish with Alpha Go, a Go-playing agent revealed this year, totally different from what already existed. The first heuristic is general and the second one is for the expansion step.

3.2.1 MC-RAVE, Monte Carlo Rapid Action Value Estimation - [8], [9]

MCTS estimates a value for each move of each state indistinctly. The idea of the RAVE algorithm is to allow for the generalization of information thanks to the "all moves as first" heuristic (AMAF).

The basic intuition behind AMAF is that each move a has an AMAF value of $\frac{\tilde{W}(a)}{\tilde{N}(a)}$ where $\tilde{N}(a)$ is a number of simulation where the move a was played at any time, and $\tilde{W}(a)$ is the number of times the black player won. This AMAF value reflects the expected reward of a independently of the state of the board.

RAVE applies the AMAF principle to search tree. It shares the value of a move between the search tree. The Rave value of a node will be :

$$\tilde{R}(s, a) = \frac{\tilde{W}(s, a)}{\tilde{N}(s, a)}$$

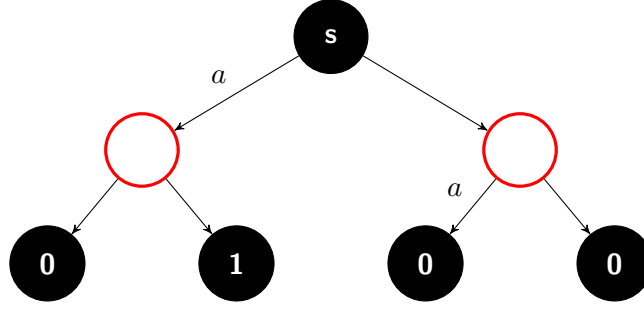


Figure 3.6: Simulations for RAVE

where $\tilde{W}(s, a)$ is the reward for all the simulations starting from s with a played eventually and $\tilde{N}(s, a)$ is the number of simulations starting from s with a played eventually.

In the figure 3.6, the standard MC count is $\frac{W(s, a)}{N(s, a)} = \frac{1}{2}$ while the Rave value of (s, a) is $\frac{\tilde{W}(s, a)}{\tilde{N}(s, a)} = \frac{1}{3}$ because a was played in three simulations and won only in one.

RAVE allows for a really fast learning but ends up frequently wrong. Indeed, its main assumption, that a move has the same value all game long, is wrong. The position of neighbouring stones can strongly impact the results. MC-RAVE chooses a trade-off between the rapid learning of RAVE and the accuracy of MCTS. Each node has a value depending on each technique with a factor β that weight the importance given.

$$\tilde{V}(s, a) = (1 - \beta)MC + \beta(RAVE)$$

$$\tilde{V}(s, a) = (1 - \beta(s, a)) \frac{W(s, a)}{N(s, a)} + \beta(s, a) \frac{\tilde{W}(s, a)}{\tilde{N}(s, a)}$$

Where β decreases from 1 to 0 over time, inversely proportional to $N(s, a)$.

3.2.2 Bradley and Terry model - [10]

Initially, this model is used to estimate the chance of winning of participants in a competition. First, it gives an estimation of a certain strength α_i for each player i , and then it predicts the chances to win of the player i against the player j as follow:

$$P(i \text{ vs } j) = \frac{\alpha_i}{\alpha_i + \alpha_j}$$

The consequences of these definition are that :

- The probability of victory depends solely on the strength of the players implied.
- The stronger i is compared to j , the better are his/her chances of winning.
- There is always a non null probability for each player to win a match.

The Bradley and Terry model also allows to represent teams of players against each other. A team has a strength equal to the product of the strength of its members :

$$P((1 + 2) \text{ vs } (3 + 4)) = \frac{\alpha_1 * \alpha_2}{\alpha_1 * \alpha_2 + \alpha_3 * \alpha_4}$$

Now the application to Go is presented. Each possible move has several features depending on the board and the move. A move can have characteristics such as :

- Putting Atari a group (only one liberty remaining).
- Having a certain distance of intersections with the previous move.
- Securing a group, making it safe whatever happen.
- Being next to a group.

Each one of theses features i can be given a strength α_i . The strength of a move s_a can be described as the product of the strength of all its features :

$$\prod_{\text{features } i \text{ in } s_a} \alpha_i$$

Following the logic of the model, if we want to compare the different possible moves from the state s , we evaluate a probability associated with each move s_a such as :

$$P(s_a) = \frac{\prod_{\text{features } i \text{ in } s_a} \alpha_i}{\sum_{\text{legal moves from } s} \left(\prod_{\text{feature } i \text{ in } s_j} \alpha_i \right)}$$

Once this is done, we have effectiveness probability for each move. Then, during the expansion phase, we can select a move in the T best effectiveness probability, because there is a high probability to find a good move in it.

But as the game goes on, the distinction between moves diminishes and we need to inspect more moves. Thus we will progressively increase T . The following formula is used by the developer of AYA, a well-known Go program, to compute T where n is the number of plays, and μ a parameter :

$$T = 1 + \frac{\log n}{\log \mu}$$

3.2.3 AlphaGo - [11]

During this year, a brand new bot was released by Google under the name of AlphaGo, generating a wide surprise among both programmers and Go players. Indeed, not only did it won against the European champion, Fan Hui, in October 2016 but it also defeated the world best player, Lee Sedol, in March 2016. Therefore the emblematic challenge posed by Go, one of the last board game where the best humans were still undefeated by machines, is now, in a certain way, over. This was unpredictable, as experts claimed that this level of A.I was not to be expected before a decade. Google really nailed it in a beautiful and successful way. We present shortly Google's method.

Google used deep learning, a machine learning brain inspired technique. The AlphaGo program wasn't coded to know how to play Go. Instead, it learned automatically from a large database of 30 millions professional's moves what is a good play and what is not.

Furthermore, it also played against itself, becoming better through each iteration. This technique is called reinforcement learning.

Alpha-Go closed a decades-old challenge. It succeeded by mimicking human-like knowledge and by training instead of trying to crack it through sheer computational power. As it is not the first challenge solved by machine learning, it really shows the capacity of deep learning to solve one AI problem after another.

One of the remaining challenges of deep learning is to successfully recover the knowledge from the self-learning algorithm and transmit it to something else. For example a master in Go

won't have difficulties to use his skills acquired in Go for another similar fields, while this is still not possible for AlphaGo.

Chapter 4

Naive version

In this chapter, we present the first naive version of a bot player with Monte Carlo Tree Search mechanism. We display the base structure for a general Monte Carlo algorithm and then the first really naive adaptation for Go. Finally we show that this version has a poorly performing complexity. It is only the representation of Go that is naive, not the structure of the MCTS.

4.1 General Monte Carlo

As explained before, one iteration of MCTS is composed of four steps.

1. Make a copy of the actual state.
2. Do the move chosen by the policy of the selection step until a leaf node of the MCTS tree is reached.
3. Get a random move, apply it and create a node in the MCTS tree for it (expansion step).
4. Get a move at random and apply it until the end of the game is reached (simulation step).
5. Backpropagate the information to the root of the MCTS tree.

So we need several programming methods relative to the state of the board of Go.

- Clone a state.
- Get the possible moves from a state, `GetMoves()`.
- Apply a move m on a state, `DoMove( $m$ )`.
- Get the score of a state, `GetResult()`.

4.2 Go Methods

We now present the general rules for each situation. For instance, `GetMoves` return all the possible moves from a state, we thus show in which situation a move is allowed in the rules of Go. Then we present a naive implementation for this behaviour.

4.2.1 Clone

To clone a state of the game, we simply need to create a copy of each information in that state. The chapter 6 implementation will detail the `GoState`. What really matters for the moment is that the largest data in that game state is the board, of size $n * n$ where n is typically equals to 9, 13 or 19 in Go. As a consequence, the complexity of our Clone method is $\mathcal{O}(n^2)$, because we will need at worst n^2 operations to clone the state.

4.2.2 GetMoves

In Go

The possible moves are seemingly simple. We can play only on free intersection, and we cannot make plays that will only kills one of our group. Thus, we can define formally a legal move as:

$$F \wedge (E \vee K \vee G) \Rightarrow Possible$$

Where F means that the intersection is free, E means there is an empty intersection in the neighbourhood, K means the move kills and enemy group and finally G means there is a friendly group in the neighbourhood with more than one liberty.

The rule still need to take into account the Ko (see Basics: rules of Go, figure 2.4).

So the final definition for a possible move is :

$$F \wedge (E \vee K \vee G) \wedge NoKo \Rightarrow Possible$$

In programming

The board is represented by a matrix of integers : a free intersection is represented by a zero, a black stone by one, and a white stone by two. Checking F and E is easy and is in $\mathcal{O}(1)$.

Next comes the hard part: check if a group is alive, ally or enemy. With a stack, we explore each group. To start with, we take the first stone and put in on the stack. Then for each stone on the stack, we look at his neighbourhood and we put on it the stones of the same color that wasn't already explored. It is a depth first search(DFS) algorithm in $\mathcal{O}(l)$ where l is the length of a group.

In order to verify if a move is possible we have to

1. Check if the intersection is free.
2. Check if at least one intersection in the neighbourhood is free.
3. If no, check if, after the possible move, there is an allied group alive, or a dead enemy group.

But for the moment this method doesn't check Ko. Thus for each move, the naive algorithm had to check that it wouldn't produce the last board, recorded in memory. Applying a move and comparing the two board takes $n^2 * l$ operations, the ko check thus had a $\mathcal{O}(n^2 * l)$ per move where n is the size of the board.

So a simple call of GetMoves requires each position, n^2 position in total, to be checked. As the complexity of checking one position is $\mathcal{O}(n^2 * l)$, it leads to a complexity of $\mathcal{O}(n^2 * (n^2 * l))$. As a consequence, the greatest factor is n^4 , which is unacceptable if we want a feasible computation time. In the following chapter, we will see how to overcome this problem.

4.2.3 DoMove

In Go

Of course the primary effect of placing a stone, is playing the stone on the board. Then if that stone deprived an enemy group from its last liberty, the dead group is captured and its stones are removed from the board.

In programming

As explained in the precedent part, playing a move changes the value of the intersection ($\mathcal{O}(1)$) and then we check for enemy groups in the neighbourhood and whether they are alive or not thanks to the stack system ($\mathcal{O}(l)$).

4.2.4 GetResult

In Go

The game is over because no more moves can be played, or because the two players passed. Indeed, a player has the right to pass in Go, notably if he thinks that they he has no more interesting plays.

Once the game is over, the count of points begins, depending on the territories defined and on the number of prisoners.

In programming

This is a tricky part as it's really hard for a computer to define an empty zone as a territory with certitude. Thus the only way for the algorithm to simulate until the end is to simply run out of possible moves. Therefore the board is filled with groups with two eyes. Thus, the count of points becomes simple, in addition of the prisoners, the algorithm just have to look for all the free intersections and their neighbourhood stones. If an intersection is surrounded by white/black stones, one point goes to the white/black player.

4.3 Total Complexity

We study here the final complexity of one iteration of the four steps of Monte Carlo Tree Search.

The process of an iteration begins with a Clone of the state. When the tree and the board are empty, the selection step is immediately over, we expand a move then simulate a game until the end. Which means get a move, then perform it, until the board is complete, which takes at worst n^2 repeats, the number of free intersections on an empty board. next we have to get the result and consequently update the node on the way to the root of the MCTS tree. As MCTS simulates a complete game, and that a game can have a length of n^2 moves, we know for sure that MCTS will do at least n^2 operations. Thus, it appears that the lower bound complexity for MCTS is $\Omega(n^2)$.

- n : the size of the board
- GM: complexity of getmoves= $n^4 * l$
- DM: Complexity of Domove = l (check neighbourhood)
- GW: complexity of getResult = n^2 (check each position)
- CL: Complexity of clone = n^2
- d: depth of Monte Carlo tree ($d < n$)

Total worst case complexity = $(CL + n^2 * (GM + DM) + GW + Update)$

Which gives = $\mathcal{O}(n^2 + n^6 * l + n^2 * l + n^2 + d)$

As d cannot exceed n , the complexity can be simplified to : $\mathcal{O}(n^6 * l)$

Such a complexity is impossible to keep as the number of operations for $n = 9$ is 531 441 for only one iteration. With such a value, the results reached by the algorithm will not be statistically representative. As a consequence, the following chapter explains improvements for this version.

Chapter 5

Optimization

*In this chapter, we explore how we improved the previous version complexity from $\mathcal{O}(n^6 * l)$ to $\mathcal{O}(n^2)$. This was done by reducing the complexities of *GetMoves* ($n^4 * l$) and *DoMove*(l) to constant time thanks to several upgrades. At the start of each improvement, we detail the operation affected (*GetMoves* or *DoMove*) and its complexity both before and after the upagred.*

For simplicity reasons, this chapter presents only the final version. Other approaches were tried (Union-Find-Delete, chained list of pointers), but as they proved less efficient, they were not included in the final version and thus not detailed here.

5.1 Abstract Representation of Group

Operation affected	GM before	GM after
GetMoves(GM)	$\mathcal{O}(n^4 * l)$	$\mathcal{O}(n^4 * l)$

The absence of representation of a group on the board in the naive version led to many drawbacks. We needed a data structure to represent a group optimally : Union-Find or Disjoint-set is a data structure that allows to represent easily and efficiently groups of elements. Even if it does not immediately reduce the complexity, it allows the future improvements that will.

5.1.1 Presentation of Union Find

The union find is a special data structure characterized by the following advantages :

- You can find the representative element of a group in $\mathcal{O}(\alpha(n))$
- You can make the union of two groups in $\mathcal{O}(1)$
- Note : $\alpha(n)$ is the inverse of the Ackerman function $A(x, x)$, an extremely fast growing function($A(1, 1) = 3$ while $A(4, 2) = 2^{65536} - 3$). Thus $\alpha(n)$ can so be legitimately approximated to a small constant, 1.

Union-Find works thanks node with special attributes. Each node has two attributes, a rank and a parent. Every node has one and only one parent that is either himself (in case the node is alone) or another node. One node can be the parent of several other nodes. Finally all the nodes have a rank corresponding to the maximum depth in their subtree.

Find(x)

The find operation take as input a node and return the representative element of a group.

At each execution of *Find*(x), the tree is balanced. All the nodes on the path from the root to x included have their parent set as the root. Thus potentially reducing the depth of the tree and optimizing the next *Find*() operations.

Algorithm 2: Find algorithm

Input: A node x **Output:** The representative node of the group of x

```
1 if  $x.parent == x$  then
2   | return  $x$ 
3 else
4   |  $x.parent = \text{Find}(x.parent)$ 
5   | return  $x.parent$ 
```

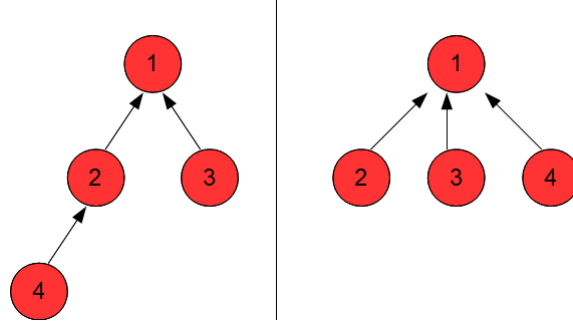


Figure 5.1: Before and after Find(4)

Union(x,y)

Union method has the following pseudo code : algorithm 3. With this the root's parent of the group with the lowest rank become the root of the other group. Thus it is always the shallowest tree that is attached to the tree with the maximum depth. In consequence, we minimize the increase of depth of the total group. The only situation where the depth increases is when ranks of the roots of the two group are equal.

Algorithm 3: Union algorithm

Input: A node x and a node y **Output:** Nothing returned, the two groups were unified

```
1 xRoot=Find( $x$ )
2 yRoot=Find( $y$ )
3 if  $xRoot.rank > yRoot.rank$  then
4   | yRoot.parent=xRoot
5 else if  $xRoot.rank < yRoot.rank$  then
6   | xRoot.parent=yRoot
7 else if  $xRoot != yRoot$  then
8   | yRoot.parent=xRoot
9   | xRoot.rank+=1
```

5.1.2 Using Union-Find for Go

The main idea is to use the Union-Find data structure to represent the different groups on the board. Each stone start behaving as a single node. Then, when two stones are placed next to each other, we union the two of them, and go forth as more stones are added to the group. The GoState does not represent the board with a matrix of integer but with a matrix of Union-Find nodes. Each node contains the basics attributes for Union-Find and the color of the intersection.

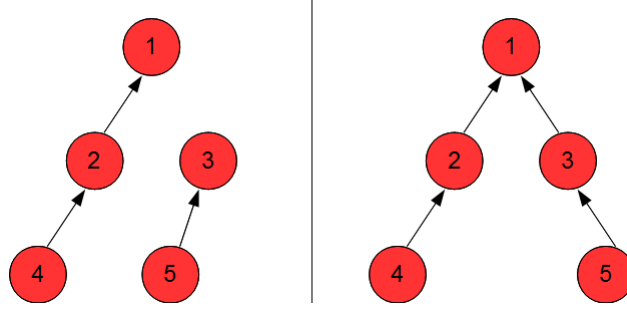


Figure 5.2: Before and after Union(2,5)

Furthermore, the parent node of every group conserves two data structures. The first one is a set and maintains all the liberties of the group itself. The second one is a chained list of the elements of the group, needed when a group is destroyed. The chained list attribute doesn't affect the complexity of union, because the Union operation of two chained lists takes a constant time.

Playing a stone has a complexity of $\mathcal{O}(1)$. Checking if a node is free is in $\mathcal{O}(1)$. Checking if an enemy/ally group in the neighbourhood is alive is in $\mathcal{O}(1)$: we look at the size of the liberty set, if it's zero, it's dead. The only problem with DoMove and this method is during the union operation. As we changed the Union-Find data structure to add the set of liberties to the nodes, the complexity changed as well. Because we also need to union the sets, the complexity of the union operation becomes $\mathcal{O}(n + m)$, where n and m represent the size of the sets for each group united.

5.2 Ko

Operation affected	GM before	GM after
GetMoves(GM)	$\mathcal{O}(n^4 * l)$	$\mathcal{O}(n^2)$

The rule of Ko : “One may not play in such a way as to recreate the board position following one's previous move” ([18]). Such a statement can be rephrased. Indeed the only possibility for the Ko rule to enter in application is in case a single stone is captured. Then the player who lost his stone cannot immediately play again in that intersection. Thus the new statement is : “One may not capture just one stone, if that stone was played on the previous move, and that move also captured just one stone”. Therefore in practice, instead of comparing the old and new boards, when player a captured a single stone i , player b cannot play at this turn in the intersection that contained i . It is feasible to both remember which intersection is concerned and to forbid the move there in a constant time.

5.3 Dynamically Maintained List

Operation affected	GM before	GM after
GetMoves(GM)	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

A complexity of n^2 for GetMoves and thus at least n^4 in total is still too large. To reduce this value, we can keep a list of possibles moves in the state and transfer this list between the states. In order to maintain this list, we update it each time a move is done (removing now unvalid moves, like suicidal ones and adding the now possible ones).

The problem is that inserting or removing an element of a array is in $\mathcal{O}(l)$ where l is the length of the array. Those operations are executed in expected constant time in a set. But with a set, there would still be a problem. The function GetMoves is used in this Monte Carlo algorithm for two distinct things: either in the simulation step to select a move at random, either in the expansion step to expand a random node. So we must be able to select a random move but this is not possible in a set.

Therefore I used a special data structure that combines an array and a hashmap. I'll present here how the methods insert, remove and GetRandom works in $\mathcal{O}(1)$

Insert x

To insert an element x:

Array.append(x) $\Rightarrow \mathcal{O}(1)$

Hashmap(x) = index of x in Array $\Rightarrow \mathcal{O}(1)$

Example : insert the value "blue"

Hashmap	Array		
{ red : 1, green : 2 }	red	green	
becomes			
Hashmap	Array		
{ red : 1, green : 2, blue : 3 }	red	green	blue

Remove x

To remove an element x :

We exchange the last element of the array with the one that needs to be suppressed(step 1 to 4). Next we pop the end of the array(step 5) and we finally suppress the value in the hashmap(step 6). This is the pseudo code :

Input: A value x to remove

- 1 last= Array[len(Array)-1]
 - 2 toRemove = hashmap[x]
 - 3 Array[toRemove]=last
 - 4 Hashmap[Last] = toRemove
 - 5 Array.pop
 - 6 Hashmap.delete(x)
-

Example : Remove green from the structure.

- Take the initial structure :

Hashmap	Array		
{ red : 1, green : 2, blue : 3 }	red	green	blue

- Switch with the last value :

Hashmap	Array		
{ red : 1, green : 2, blue : 3 }	red	blue	green

- Adapt the hashmap for the previous last value (here blue) :

Hashmap	Array		
{ red : 1, green : 2, blue : 2 }	red	blue	green

- Pop the end of the array and suppress the value in the hashmap :

Hashmap	Array	
{ red : 1, blue : 2 }	red	blue

Random an x

To get a random element :

Array [random(Array.size)]

The following operations were needed on GetMoves :

- Get a random move : Get a random element from the data structure $\Rightarrow$ constant time ✓
- Check if there are still other possible moves : Check if the size of the hashmap $> 0 \Rightarrow$ constant time ✓

Consequently, each time we call GetMoves in the MCTS algorithm or we maintain it due to a DoMove operation, it is in constant time.

5.4 Numerical Liberties

Operation affected	DM before	DM after
DoMove(DM)	$\mathcal{O}(l)$	$\mathcal{O}(1)$

Currently, the liberties are remembered by each group thanks to a set. But this representation leads to l operations when two groups unite. Thus the liberties model presented in section 5.1.2 needs to be changed for DoMove to be in $\mathcal{O}(1)$. Our final idea to manage the liberties of a group should allow for a size check in $\mathcal{O}(1)$ and a maintenance in $\mathcal{O}(1)$. In order to accomplish that, each group will simply remember the number of its liberties thanks to an integer.

5.4.1 Main Idea

When a stone is placed on the board:

- it gains as many liberties as there are free intersections next to it
- it removes one liberty to each group in each direction. If case the same group is concerned x times, the group liberties integer is decreased by x .
- Last but not least, it unites with the ally groups in the neighbourhood. The number of liberties of the united group is the sum of the two sub-groups liberties.

5.4.2 Validity of the Model

Let's start with a single stone. This stone has $l = 4 - nn$ liberties where nn is the number of neighbouring stones. If another stone is grouped with this one, their liberties are added up. If an intersection has the same group a times in its neighbourhood, the group will count this intersection as a liberties.

As a consequence, the number of liberties of a group is $\sum_{i=firstlib}^{n=lastlib} nb_i$ where nb_i is the number of times the intersection i counts the group in his neighbourhood.

However, when a stone is placed on the intersection i , it removes nb_i to the number of liberties of the group. Thus, the number of liberties of a group is zero if and only if the group is completely surrounded by enemy stones or edges.

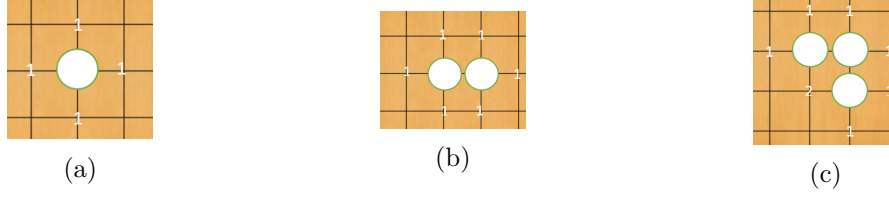


Figure 5.3: The group of stones a,b,c remember having respectively 4,6 and 8 liberties

5.4.3 Maintaining the List of Possible Moves

Our last concern is determining whether a move is playable or not. Previously, we had at our disposal different mechanics to suppress invalid moves. Nevertheless, they worked only because they knew which nodes were liberties of which group (thanks to the set of liberties). Therefore, it was needed to come up with a new method in order to maintain the list of playable moves :

On the one hand, a move (x, y) is removed from the data structure of possible moves when a stone is placed on (x, y) . On the other hand, a move (x, y) is added to the data structure when a stone is removed from the board on (x, y) . Ultimately, when the MCTS algorithm intends to play a move, it first checks if this move is playable, in case it is not, it is withdrawn from the possible moves for this state. Then, the algorithm picks another move at random and tries it.

5.5 Conclusion

The total complexity of one iteration was :

$$\mathcal{O}(CL + n^2 * (GM + DM) + GW + Update)$$

Once we had established the Union-Find data structure to represent the group and the other data structure to maintain a list of possible moves, the complexity of GetMoves was in constant time. Thanks to the numerical representation of liberties, DoMove, which includes all the following operations, has a complexity in constant time.

- Place a stone on the board on x, y : Change node(x, y) color value ✓
- Remove liberty for adjacent group : Decrease liberties integer ✓
- Union with another group of the same color : Our final Union-Find is constant ✓
- Check liveness of enemy group : Is the liberties integer > 0 ✓
- Suppress dead group : Pass through list of the group's components and remove each of them. On a game on average we won't remove more than one stone by intersection $\Rightarrow$ amortized constant time by intersection (one operation to place it, one to remove it, see Basics : complexity) ✓

Here's a tabular to show the evolution of the total complexity.

Version	GetMoves	DoMove	Final ($n^2 * (GM + DM)$)
Naive	$n^4 * l$	l	$n^6 * l$
Union Find	$n^4 * l$	l	$n^6 * l$
Ko	n^2	l	$n^4 * l$
Maintaining possible moves	constant	l	$n^2 * l$
Numerical liberties	constant	constant	Amortized n^2

Chapter 6

Implementation

This chapter concerns itself with the technical python implementation of the algorithm, along with the overall structure, the interaction with Graphics user interfaces and the technical tools used. We present here our implementation of the MCTS part only. The entire project, with the different previous version is available on Github : <https://github.com/Fioelkais/MCTS.git>

6.1 General Monte Carlo

6.1.1 Game State

The game state corresponds to the state of the game at a particular moment. The game state is transferred and modified to reflect the execution of plays. Here are the principal methods needed for its execution:

- Initialization.
- Clone.
- Execute move (m) on the state.
- Get possible moves from that state.
- Get result (p), return 1 if the player p won in that state, 0 otherwise.

Here are its attributes :

- The board: an array of array of Node.
- The player who just moved : to know which has to play now.
- The possible moves for each players: in the data structure describes in chapter 5.3
- The actual points for each player.
- The potential forbidden move due to the Ko rule.
- A boolean set to true if the last player passed, decided to play no move.

6.1.2 Node Class

The node class represents a node of the Monte Carlo(MC) tree. Each node must have the several attributes listed:

- Move. The move that lead to the node, that applied on the parent node board would give the actual node board.
- The parent node.
- The children nodes.
- The number of times the player won passing by this node.
- The number of visits.
- All the possible moves still not expanded into the tree: untried moves.
- The player who did the last move. To know the actual player.

The following operations must be feasible on a Node :

- Select a child following a formula (in our case UCT formula).
- Add a child to the node.
- Update the value of wins/visits.

6.1.3 Simple Pseudo Code

We detail here a pseudo code for the main method, UCT. In V-Run, it is the method, programmatically speaking, that receive a state as input and return the best move found as output. We called it UCT as this method used the UCT formula for the selection step (see algorithm 4). Note : the algorithm used in reality is a slightly modified version of the one presented here. We didn't show it exhaustively due to space reasons. The real version includes the mechanic to remove a potentially invalid move from the list of moves, and to pick a new one if it was the case. You're welcome to look it up on the Github.

6.2 Overall Program

With the precedent algorithm, we can return a specific move given a state. To obtain an AI able to play against other AI or even players, we needed a Grapical User Interface or GUI. We used GoGui [19], a standard user interface that allows humans or bots to play between them. To communicate with the bots, this GUI uses a very common protocol, the Go Text Protocol or GTP.

We create a function main, that simply opens a prompt for GoGui, in which a series of command were implemented. The prompt is invisible to the user, as it is only a mean to allow the GUI and the programm to communicate (see figure 6.1).

- $\text{boardsize}(x)$: define a new board with size $x * x$.
- $\text{clearboard}()$: clean the actual board.
- $\text{komi}(x)$: set a komi of x for white.
- $\text{play}(P, L, X)$: play the stone of color P in the row L and in the column X .

Algorithm 4:

Input: A state of the game rootstate and a number n of iterations

Output: The best child move of the root state

```
1 rootnode = Node(state=rootstate)
  /* Create the root of the MCTS tree, for the moment it is the only node in
  it */
2 for i ← to n do
3   node=rootnode
4   state=rootstate.Clone()
  /* Selection step */
5   while node.children not empty && node.untriedmoves is empty do
6     node=node.UCTselect()
7     state.DoMove(node.move)
  /* Expansion step */
8   if node.untriedmoves not empty then
9     m=random.choice(node.untriedmoves)
10    state.DoMove(m)
11    node.addChild(m,state)
  /* Simulation step */
12   while state.GetMoves() not empty do
13     state.DoMove(random.choice(state.GetMoves()))
  /* Backpropagation step */
14   while node is not None do
15     node.update(state.GetResult(node.playerJustMoved))
16     node=node.parent
17 return best(root.childs) /* Where best(x) return the childe node with the most
visits */
```

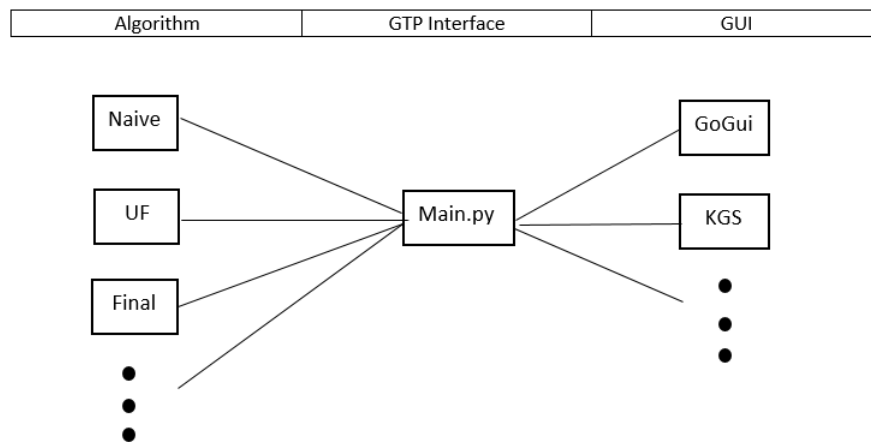


Figure 6.1: Interface between bot and GUI

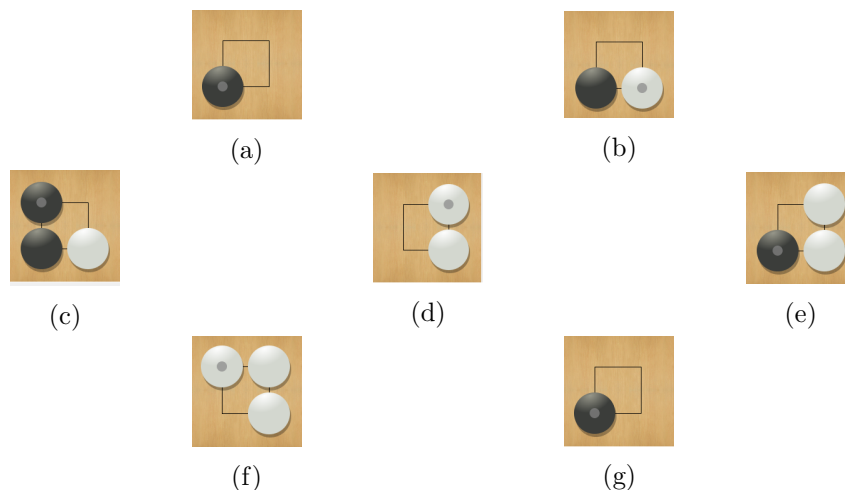


Figure 6.2: Infinite game

- `genmove( $p$ )` : ask the A.I to give a move for p .

Each time the player will take an action in GoGui, the corresponding command will be issued by GoGui. For instance, if the player clicks on the GoGui button “ask the AI to select a move for black”, GoGui enters the following command: `genmove(b)`. After a certain time, the A.I answers for example: A6. A6 is the GTP translation for: first line, sixth column.

Thanks to GTP and this implementation, we can easily try different GUI or even different algorithm versions. By handling GTP, the main program articulates between the GUI and the algorithm. Changing the version of UCT called in `genmove()` allows to try different versions of MCTS, with different number of iterations.

6.2.1 End Game

In general, a game of Go reaches its end when the two players have consecutively decided to pass, judging there are no more interesting moves to be played on the board. Such an evaluation is really hard for a program. So we manage the end game differently depending on two cases.

When MCTS simulates the game

In this case, the algorithm will simply play random moves until the end, meaning no more moves are possible, as explained in the chapter 3. Nevertheless, such a play will simply lead the algorithm to fill completely the board with stones until group are captured and so on and so forth, see figure 6.2 for an example on a 2×2 board. To solve this problem, we forbid V-Run to fill an intersection that is fully surrounded by the same group. As this type of move is never helping, we can thus safely remove it. Therefore, on boards of standard size, the white and black players will eventually form safe groups with at least two eyes leading to a closed game.

This modification allows us to improve the time taken by the method `GetWinner`. Indeed, before this change, `GetWinner` needed to check the whole board, looking for free intersection and then to determine to which player they belonged. Now, we progressively remove from the possible moves all the intersections that will still be free at the end of the game, without exception. Thus we simply record these intersections, and when the game is finished, those are the only ones to be checked.

When playing against a human

On the other hand, we cannot as well detect that the game is over when V-Run is playing against a human. Furthermore going until a total end game is laborious for the human player. Therefore we simply leave the role of the end game detection to the human. When the human player pass his/her turn, thinking the game is over, V-Run imitates the player behaviour.

Chapter 7

Experiments and results

This chapter firstly presents the results of our experiments in three different parts : the overall evaluation of V-Run level (see basics : Go rankings), the quality of its plays at the start of the game (on an empty board), and its answers in tsumego (special Go problem). Then we compare the results of the basic V-Run with a majority voting version. To finish, we show that the practical complexity evolve accordingly to the predicted complexity.

7.1 General

Following several test games (10 per match up, see next tabular) against another bot of different levels [20], we estimate the level of V-Run to approximatively correspond to 15 kyu given thirty seconds to one minute per play (between 5000 and 10 000 iterations). V-run played relatively well with regards to its simple implementation, making overall good choices and often finding the critical moves, see a figure 7.1 of boards won by V-Run (black). For figure 7.1 d, the game was already over at that point in regard to the level of the white bot. There was no reason to pursue it.

Against	18 kyu	15 kyu	11 kyu
V-Run 5000 iterations	90 %	40%	0%
V-Run 10 000 iterations	100 %	80%	10%

We also tested V-Run's capacities when playing against itself, with different amounts of iterations. In the next tabular, the cell of row i and column j represents the percentage of victories for a black V-Run with i iterations against a white V-Run with j iterations. Fifty games took places for each match. We can observe that the number of iterations is crucial to the program level. The more iterations V-Run is allowed to perform, the more exhaustive is its search, and the better are its plays.

Nbr iter	1000	2000	5000	10 000
1000	44 %	22%	0%	0%
2000	84%	44%	2%	0%
5000	100%	98%	46%	24%
10 000	100%	100%	76%	48%

7.2 Start of Game

Starting from an empty board is the worst situation for MCTS as it corresponds to the largest possible search tree. Thus, in order to examine V-Run's performance at the start of the game,

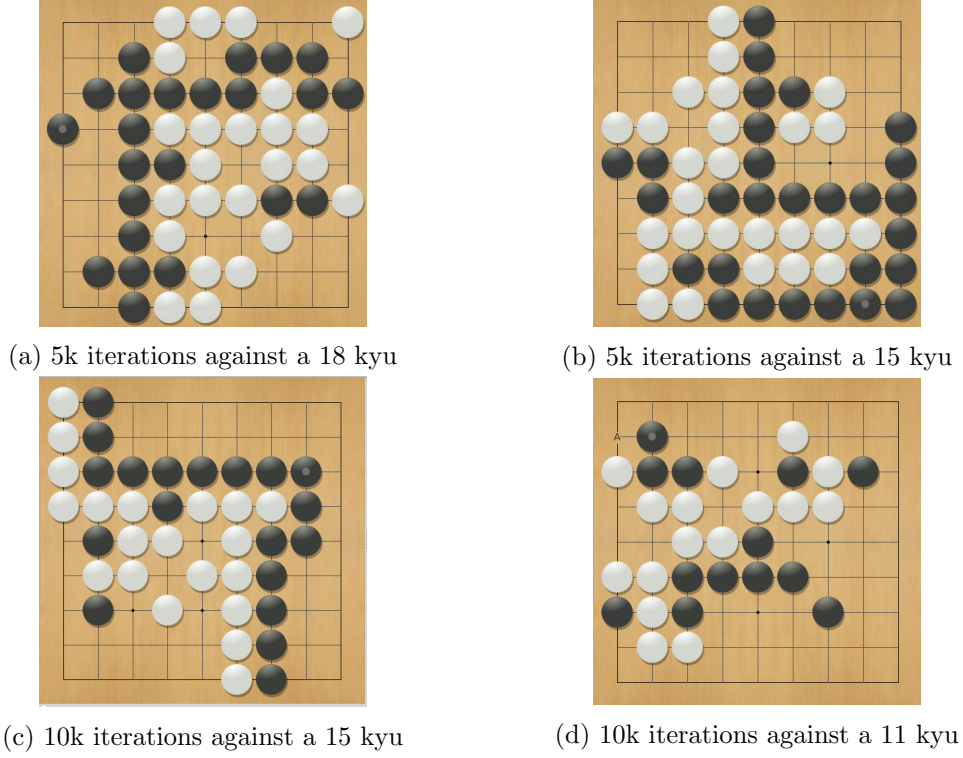


Figure 7.1: Games played by V-Run(black) against a GoFree bot(white)

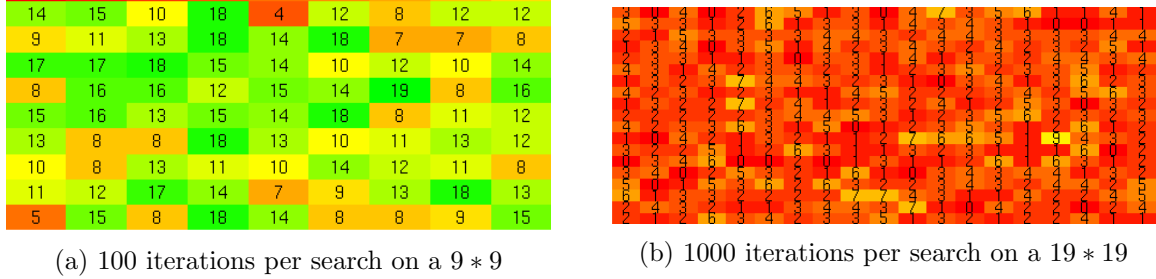
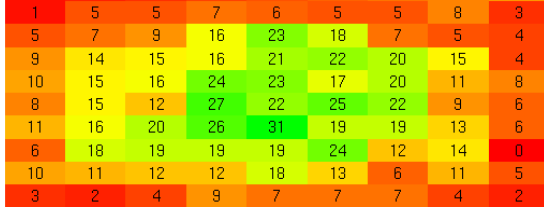


Figure 7.2: Heatmap of an empty board: number of times the move was picked out of 1000 searches

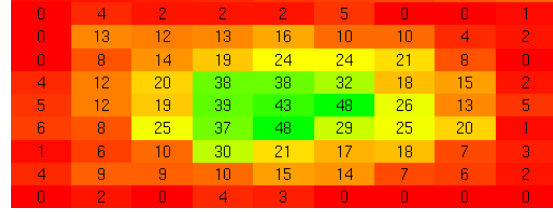
we ask it to choose a first move a thousand times, with different amount of iterations.

The results with 100 iterations for a $9 * 9$ board or with 1000 iterations for a $19 * 19$ board have proved disappointing. The number of iterations is too low relatively to the size of the board. Thus the results are very close to an uniform distribution, figure 7.2. The heatmap represents a board with, on each square, the amount of times V-Run selected this square as the first best move from an empty board out of 1000 times.

On the other hand, on a small board ($9 * 9$) and with enough iterations, respectively 1000 and 5000, better results are reached: see figure 7.3 a and b. We can observe that the higher is the number of iterations, the greatest is the probability for the returned move to be in the center of the board. It shows efficiency from V-Run, as on a $9 * 9$ board, an opening move in the center zone is strong. It has a lot of influence due to the smaller size of the board (compared to a $19 * 19$ board, where it is more interesting to play near the edges at start).

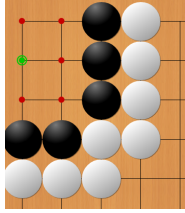


(a) 1000 iterations per search on a 9 * 9

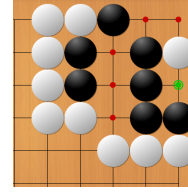


(b) 5000 iterations per search on a 9 * 9

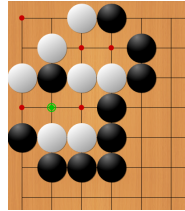
Figure 7.3: Heatmap of an empty board: number of times the move was picked out of 1000 searches



(a) Easy tsumego



(b) Average tsumego



(c) Hard tsumego

Figure 7.4: Different level of tsumego

7.3 Tsumego

Tsumego are special problem situations in Go. Often described as life and death problems, it shows a board where the life of a group, friend or foe, is at stake. The goal is to find the right move that will either save our group, or kill the enemy.

We tested V-Run on three different level of tsumego. Each one of them was tested with three different numbers of iterations. The tsumego came from “tsumego pro” [21]. We compare V-Run results to what a random move generator would have given.

We present here (figure 7.4 a and b) the two first tsumego. The green play is the right move to play. The black player plays first.

	100	1000	10 000	Random chance
Easy	44 %	64%	81%	16,66%
Average	66%	69%	74%	16,66%
Hard	7%	7%	8%	5.26%

7.4 Hard tsumego problem, symptom of the main flaw of V-Run

The results are on the hard tsumego are intriguing. We can see the right move (3,1) in green in the figure 7.4c. The main problem for its low selection count is that if black play the move (3,2), to the right of the correct move, the only move allowing victory for white is (3,1). In all other responses of white, black wins in most cases. So black considers (3,2) like a high value move. In fact, V-Run selects it 87% of the time.

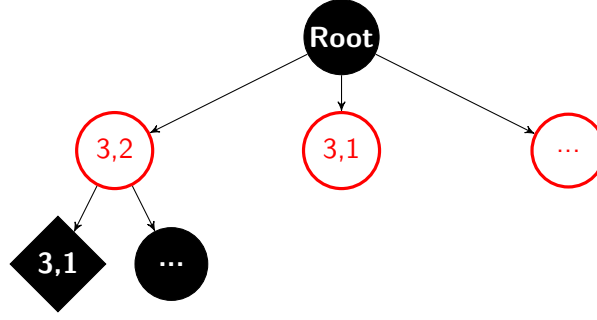


Figure 7.5: Tree for the hard tsumego

To illustrate this case, see figure 7.6. The left board is when white answer correctly. The right board is when white answer elsewhere, leaving 3,1 for the black player and so a nearly guaranteed victory.

It would seem that V-Run often do the mistake of playing a greedy move, expecting its opponent to react poorly (see figure 7.7, in the two situations, V-Run tried to play a move that could yield promising results but was easily counter by the white bot). As we saw in the hard tsumego example, if the white player failed its response, V-Run could win without difficulties. It would be as if in the MCTS tree, the correct answer for black(3,2), which is white(3,1) wasn't picked often(in diamond shape on figure 7.5), thus giving a high value to black(3,2).

But on the other hand, when we apply the move black(3,2), and we ask V-Run for the best white answer, it returns the correct answer (3,1), more than 80% of times(for 1000 iterations, out of 100 tries). The only difference is that in the base hard tsumego case the white choice was at the second level of the MCTS tree, while it is now on the first level. As a minimum of iterations must be done on each move possible from a node before the UCT formula becomes effective, a node will have significantly less visits the deeper it is in the MCTS tree. It is one of the consequence of our expansion step strategy. Actually, each time a node is reached, we expand all of its children before selecting one of them to pursue further, because it needs the minimum knowledge to choose accordingly. Meaning that if there is x possible move from a situation and we want to go to a depth of y , we will have to do at least $x * y$ iterations in the best case. On the figure 7.8, maybe only the diamond shape path is interesting. But actually to explore it, V-Run take 6 iterations, where 2 are in reality needed to build this precise path.

A solution to this problem would be to use the Bradley and Terry(B&T) model described in chapter 3.2.2 to expand only the interesting moves. Less nodes to expand would mean less iterations to go deeper, in addition to the initial advantages of B&T model. Going deeper into the tree would mean a more accurate and precise search.

In the end, V-Run is able to pick a crucial move if it has direct impact, if it is at the first level of its tree, but will sometimes fail to anticipate correctly the answer from its opponent in other situations and will try play expecting its opponent to play badly. It is probably due to our choice of strategies for each step. As mentioned before B&T model could help. The others possibilities include guiding the simulations to make them less random and more representative or testing other selections techniques.

7.5 Majority Voting

MCTS is based on probabilities and randomness. This method attempts to exploit the random search in an optimized way. But despite well thought method, we cannot prevent the program

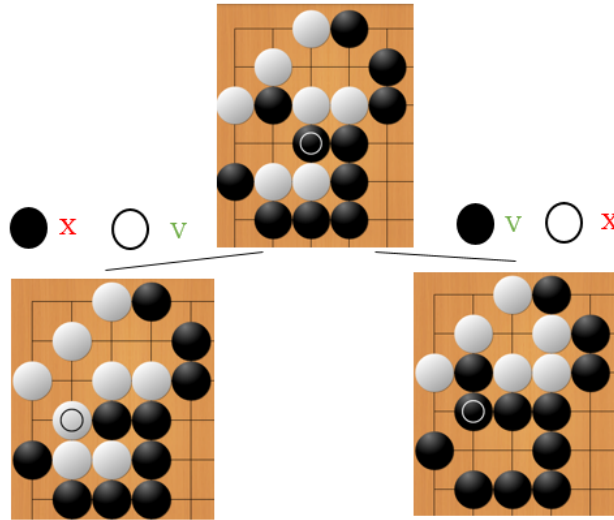
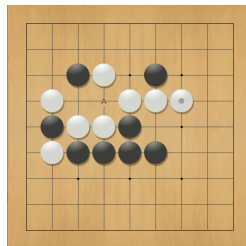
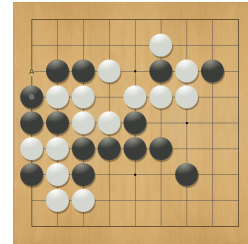


Figure 7.6: Why 3,2 is found as a valid move



(a) V-Run chose the move with the letter A



(b) V-Run just placed the stone with a grey dot, white then answered A and captured

Figure 7.7: Greedy move picked

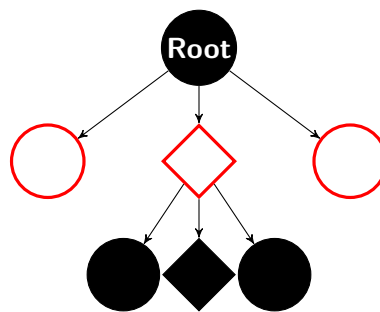


Figure 7.8: reaching a depth 2 with 3 possible moves

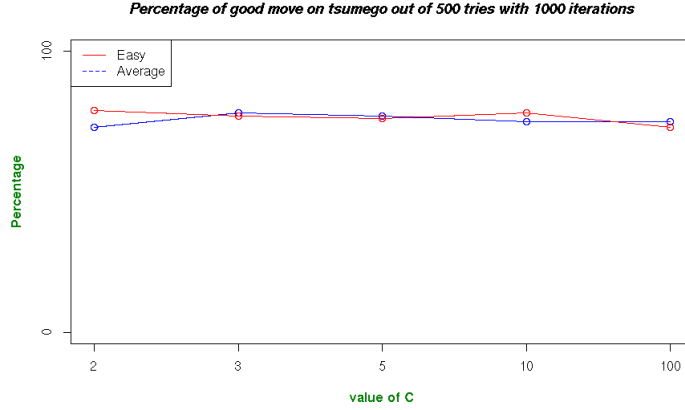


Figure 7.9: Tuning of C parameter in UCT formula

	10*500	5*1000	5000
10*500	48%	38%	2%
5*1000	62%	50%	0%

Table 7.1: Matches between 10*500, 5*1000 and 5000 iterations-50 games

to fail occasionally. If we observe the figures 7.3, it appears that increasing the number of iterations improves the quality of the moves, but a bad one can still happen. Thus we could increase the number of iterations until no bad moves are made anymore, but this strategy can be extremely time consuming. As a consequence, it is needed to find another solution. As shown by theory and related experiments, MCTS provides satisfying results on average. Our goal is to try to exploit this feature. Majority voting repeats a search several times, and selects the most returned move from all the searches. For example, instead of running 10 000 iterations, we can run ten times 1000 iterations, and choose the most selected move. The time cost is the same and this method could be more effective. Bad luck (a large search that failed), could be prevented. To check that the percentage of failure wasn't due to an unbalanced exploitation versus exploration in the UCT formula (see MCTS state of the art : selection step), we tested the results for different values of C . The results ensured that tuning C would not help here, meaning would not significantly improve the percentage of right answers, see figure 7.9.

When we compare the results on tsumego, table 7.2, majority voting performs better. Nevertheless, when we test a V-Run standard against a V-Run using majority voting 7.1, with the same total number of iterations, the standard V-Run wins in most cases.

When the number of iterations is by far superior to the number of possible moves, majority voting appears to perform better while it is the opposite when the other situation arises. The next formula could be used to balance and select the most profitable version for a given state:

$$F = \begin{cases} S & \text{if } \frac{i}{m} < k \\ D & \text{if } \frac{i}{m} > k \end{cases}$$

Where F is the technique to choose when there are still m move possible, S is the standard MCTS with i iterations, D is the majority voting method with d search of $\frac{i}{d}$ iterations, and d and k are parameters. Currently, we cannot provide the optimal values for d and k and leave it to future research.

	10*10	10*100	10*1000
Easy	33%  7%	80%  16%	96%  15%
Medium	48%  18%	74%  5%	77%  3%

Table 7.2: Comparison of the results between different majority voting V-Run with 10 times k iterations and $10 * k$ total iterations standard V-Run

7.6 Binary Victories

One of the problem encountered with MCTS is due to the binary evaluation of victory. At the end of the simulation step, when V-Run check the winner, the value put in the node is either 1 for a win, or 0 for a loss. In a situation where V-Run is completely loosing a game, and have no more possibility to win, the MCTS will return a random move, as all the nodes would have a null value. In this case, it would be interesting for V-Run to choose the moves that would minimize the score difference, as much as trying to maximize the score difference in winning situation. This attitude would be closer to the one of a real player.

More over, if V-Run prefers a path that would maximize its score, it would have more margin in case of a bad play later, due to a search that would have gone statistically wrong.

What we propose to add this mechanics to MCTS without disturbing the core technique is to balance the reward found. As mentioned before, for the moment the values of wins in the node is updated by 1 or 0. Our proposition is :

$$r_i = w_i + \frac{(p_i - p_j)}{k}$$

Where r_i is the reward for the node for the player i , w_i is 0 if the player i lost in the simulation from that node, 1 in the other case, p_i and p_j are the points of the players i and j at the end of the simulation and k is a parameter to optimize.

In 50 matches between a standard V-Run and a V-Run with this adaptation ($k = 100$), both searching with 1000 iterations, the last one won in 56% of the cases. Further testing on different k value is needed.

With this method, between two winning nodes, the one where the difference of score is the highest is promoted. As it is, the reward r_i can be higher than 1 or lower than 0, maybe it's better to set w_i to 0.1 for a loss and 0.9 for a victory, with a large enough k such that $0 < r_i < 1$. We leave further testing to future research.

7.7 Complexity

For now, we tested the play level of V-Run. In this section we verify our announced complexity. The complexity of one iteration of MCTS with our final version is n^2 where n is the size of the board, or the number of possible moves (at the start, all the intersections on the board are available). The complexity for i iterations is thus $\mathcal{O}(i * n^2)$.

Therefore, we tested V-Run with different sizes of board for the same number of iterations and with different number of iterations for the the same size. We can effectively notice (see figure 7.11) that time evolves linearly with regard to i iterations but in a quadratic way with regard to the size n^2 .



Figure 7.10: Time complexity test depending on i iterations

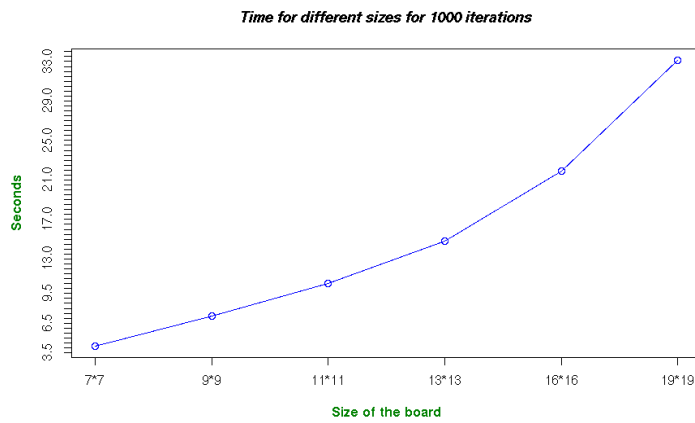


Figure 7.11: Time complexity test depending on n^2 the size of the board

Chapter 8

Conclusions

The goal of this master thesis was to implement an A.I for Go, based on the Monte-Carlo Tree Search techniques, and to reach the lower bound complexity.

Go is a strategy board game that always challenged A.I due to its large branching factor. Indeed a game last usually 200 turns and at start of the game, 361 different moves are possible. The standard A.I algorithm failed to reach a professional level.

Monte Carlo Tree Search helped tackling this problem. As it is based on probability and does not require to explore the full tree, this method allowed a lot of progress in the area, taking the bot from amateur player to low level professional. MCTS builds the tree step by step. At each step, it explores the present game until the end, examine the result and take it into account. After a few iterations, the program can make a rough estimate of the probability to win for each possible move (explored x times, won y times), and thus chooses accordingly.

The higher is the number of iterations, the better is the results as it will be closer to the real value. Therefore, each iteration needs to be executed the fastest possible. As a consequence, reaching the lower bound complexity for an iteration in Go was one of the main objectives of this thesis.

In order to decrease as much as possible the theoretical computational time, we started from a naive version of the program that was gradually improved. We made use of special combinations of data structure (Union-Find, combination of arrays and sets, chained list) and different algorithms such that an iteration of MCTS would take at worst $k * n^2$ for any board, where k is a constant and n is the size of the board.

We implemented this program in python and called it V-Run. V-Run provided several interesting and performing results. V-Run, without any heuristics, database or other methods used to upgrade the level of bot nowadays, was able to reach a level of at least 15 kyu, given a time of one minute per play. In other words, the level of a rather good beginner in the game. Moreover, it was able to solve special Go problems, tsumego, and when asked to select a first move on an empty board, it also returns good answers on average.

It is really interesting to see that with a core basic method well optimized, MCTS can yield a certain level of play. As V-Run works only with the core of the technique, a lot of improvements are still possible: some parameters can be tuned, some part of the algorithm have various strategy and heuristics or database can be added.

Among the future improvements, we looked further into two. The first one would be trying to take advantage of the probability and randomness part of MCTS. MCTS is an algorithm relying partly on a random exploration with some mechanics ensuring to exploit at best the randomness. Actual techniques doesn't make use of the majority voting. We observed that MCTS give good answers on average given a high enough number of iterations. On the other hand, we also noticed that from time to time, V-Run failed to build a tree representative of the

reality. Maybe that one, two, ten time out of one hundred, the algorithm will give a bad answer. To compensate this aspect, it might be more efficient to repeat x times y iterations, and to pick the most found move instead of playing the answer to $x * y$ iterations. Majority voting scored better given a sufficiently high ratio of the number of iterations over a number of possible moves. The precise ratio and the optimal value of x are left to future research.

The second one would be to try to correct the major flaw of V-Run -its tendency to play greedy move, believing its opponent will fail to counter them. The causes of this problem could originate from several parts of V-Run, but we gave some specific leads that could attempt to solve it, including the simulations policy or the Bradley & Terry model.

The last one try to give an improvement to the binary estimation of the winner in the end of the simulation step, by balancing with the score difference.

These are interesting leads to research.

In the end, V-Run is a bot, able to play Go against other humans or A.Is, on GUI. Its level approaches that of a good amateur and its theoretical execution could not be faster. Despite having a lower play level than a lot of other bots, there is no actual public MCTS applied to Go that reached Ω . Thus, from this perspective, V-Run brought a true contribution.

Despite the A.Is' field in Go being highly disturbed by the outstanding performance of AlphaGo this year, we don't know much yet about its evolution. Therefore, it is possible that the deep learning approach will progressively outstands the other methods, but also that techniques like MCTS will have their utility in other specific fields.

Bibliography

- [1] J. V. D. Steen, “The Rules of Go,” no. February, 1995. [Online]. Available: <http://gobase.org/studying/rules/doc/a4.pdf>
- [2] S. Arora and B. Barak, “Computational complexity: a modern approach,” *Complexity*, no. January, p. 489, 2007. [Online]. Available: <http://retarget.googlecode.com/svn-history/r160/trunk/temp/Complexity/book.pdf> \n<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7207&rep=rep1&type=pdf>
- [3] Q. M. Ha, “Monte Carlo tree search and its application in Quoridor,” Ph.D. dissertation, Université Catholique de louvain, 2014.
- [4] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, “Why the Monte Carlo Method is so important today Uses of the MCM,” *WIREs Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014.
- [5] R. E. Caflisch, “Monte Carlo and quasi-Monte Carlo methods,” *Acta numerica*, 1998, vol. 7, pp. 1–49, 1998.
- [6] Wikipedia, “Monte Carlo method.” [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method
- [7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 1995, vol. 9, no. 2. [Online]. Available: <http://portal.acm.org/citation.cfm?id=773294>
- [8] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud, “The grand challenge of computer Go: Monte Carlo tree search and extensions,” *Commun. ACM*, vol. 55, no. 3, pp. 106–113, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2093548.2093574> \n<http://cacm.acm.org/magazines/2012/3/146245-the-grand-challenge-of-computer-go/fulltext>
- [9] S. Gelly and D. Silver, “Monte-Carlo tree search and rapid action value estimation in computer Go,” *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1876, 2011.
- [10] K. Ikeda and S. Viennot, “Efficiency of Static Knowledge Bias in Monte-Carlo Tree Search,” 2014. [Online]. Available: <http://hdl.handle.net/10119/12799>
- [11] B. Y. E. Gibney, “Google masters Go,” pp. 8–9. [Online]. Available: http://www.nature.com/polopoly_fs/1.19234!/menu/main/topColumns/topLeftColumn/pdf/529445a.pdf
- [12] T. B. Game, “The Board Game,” 2012. [Online]. Available: <https://kopoint.files.wordpress.com/2012/12/go-the-game.pdf>
- [13] Wikipedia, “Monte Carlo Integration.” [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_integration
- [14] Wired, “The Three Breakthroughs That Have Finally Unleashed AI on the World.” [Online]. Available: <http://www.wired.com/2014/10/future-of-artificial-intelligence/>

- [15] Futurism, “Artificially Intelligent Lawyer “Ross” Has Been Hired By Its First Official Law Firm.” [Online]. Available: ArtificiallyIntelligentLawyer.com/RossHasBeenHiredByItsFirstOfficialLawFirm
- [16] Theatlantic, “How Google AlphaGo beat a Go world champion.” [Online]. Available: <http://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/>
- [17] Wikipedia, “Go ranks and ratings.” [Online]. Available: https://en.wikipedia.org/wiki/Go_ranks_and_ratings
- [18] —, “Rules of Go.” [Online]. Available: https://en.wikipedia.org/wiki/Rules_of_go
- [19] “Gogui.” [Online]. Available: <http://gogui.sourceforge.net/>
- [20] AI Factory, “Go Free.” [Online]. Available: <https://play.google.com/store/apps/details?id=uk.co.aifactory.gofree>
- [21] L. Studios, “Tsumego pro.” [Online]. Available: <https://play.google.com/store/apps/details?id=net.lrstudios.android.tsumego-workshop&hl=fr>
- [22] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, “Monte-Carlo Tree Search: A New Framework for Game AI.” *Aiide*, pp. 216–217, 2008. [Online]. Available: <http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf>
- [23] G. M. J.-B. Chaslot, *Monte-Carlo tree search*, 2010, vol. 174, no. 11.
- [24] A. Ben-Amram and S. Yoffe, “A simple and efficient Union-Find-Delete algorithm,” *Theoretical Computer Science*, vol. 412, no. 4-5, pp. 487–492, 2011.
- [25] U. Alstrup, Stephen and Gortz, Inge Li and Rauhe, Theis and Thorup, Mikkel and Zwick, “Union-Find with Constant Time Deletions.pdf,” 2005. [Online]. Available: <https://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/Union-FindwithConstantTimeDeletions.pdf>
- [26] “General implementation of MCTS in python.” [Online]. Available: www.mcts.ai
- [27] “Wikipedia Go(jeu).” [Online]. Available: https://fr.wikipedia.org/wiki/Go_%28jeu%29
- [28] C. Browne and E. Powley, “A survey of monte carlo tree search methods,” *IEEE Transactions on Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–49, 2012. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6145622

