Monte Carlo tree search applied to the game of Go

UCL - Ecole polytechnique de Louvain

Vandermosten Thibault

Day Month Year

Abstract

Dedication

Declaration

Acknowledgements

Contents

1 Introduction **7**
- 1.1 Go and artificial intelligence . . . 7
- 1.2 Rules of Go . . . 8

2 State of the art **10**
- 2.1 Monte Carlo and Tree search . . . 10
  - 2.1.1 Monte Carlo . . . 10
  - 2.1.2 Tree search . . . 12
  - 2.1.3 Monte Carlo and Tree search combined . . . 15
  - 2.1.4 Basic Strategies for each step . . . 18
  - 2.1.5 Benefits and Drawbacks . . . 20

3 MCTS applied to Go **22**
- 3.1 Monte Carlo tree search applied to Go . . . 22
  - 3.1.1 Improvement . . . 22
  - 3.1.2 Alpha-Go . . . 25

4 Naive version **26**
- 4.1 General Monte Carlo . . . 26
  - 4.1.1 Game State . . . 26
  - 4.1.2 Node class . . . 27
  - 4.1.3 Simple pseudo code . . . 27
- 4.2 Adaptation to Go . . . 27
  - 4.2.1 GetMoves . . . 29
  - 4.2.2 DoMove . . . 30
  - 4.2.3 GetResult . . . 30
- 4.3 Total complexity . . . 31

5 Optimization **32**
5.1 Union Find, or how to solve the group problems 32
5.1.1 Presentation of Union Find 32
5.2 Improvement 1 34
5.2.1 Special structure to solve the GetMoves complexity . . 35
5.2.2 Complexity 36
5.3 Improvement 2 37
5.3.1 Remember the component 37
5.3.2 Remember the liberties 37
5.3.3 Complexity 38
5.4 Another improvement 39
5.4.1 Union-Find-Delete Model 39
5.4.2 Drawbacks 40
5.5 Final improvement 41
5.5.1 Main idea 41
5.5.2 Proof 41
5.5.3 Conclusion 41

6 Conclusion **43**

A Appendix Title **44**

Chapter 1

Introduction

1.1 Go and artificial intelligence

Go is a really ancient board game. It is more than 2500 years old. Born in ancient china, it has relatively simple rules but a deep game complexity. Actually they are more than 40 millions of players in the world. As chess, it's a game of strategy, which take its roots in military play.

Artificial intelligence on the other way, is the study and design of intelligent agent. There are several approaches behind the word intelligent. Systems can try to act or think, like a human or rationally. Each of the four cases has its own approach and way of thinking.

Here we will present the development of an artificial intelligence that will finally be able to play Go. It will be more related to the part of A.I that is trying to act rationally. A.I and Go were always an interesting match, because until recently (march 2016) the human players were better than the machine. Go was one of the last board games where it was the case. It was due to his complexity (we estimate the number of possible games in chess at 10^200 compared to 10^700 in Go). The usual algorithms couldn't take the challenge. It forced programmers to think deeper and about new techniques. That's where Monte Carlo tree search came. It helped a lot and allowed big progress in this field. The challenge was close this semester, thanks to Alpha Go, the bot of Google who defeated the world champion of Go. It used deep learning (it learned from a lot of humans games to know the best plays) to

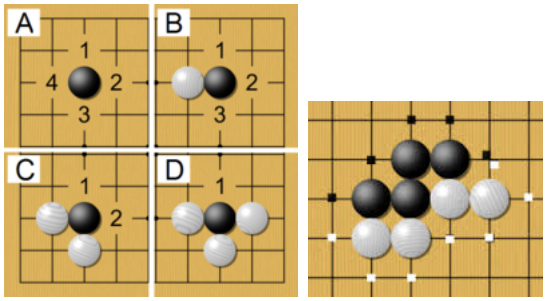


Figure 1.1: Liberties of a stone and of a group

crack the problem.

In this chapter thesis, we will try to focus on the complexity of the modelling of Go in A.I. We'll show the state of the art and explain the main concept, the Monte Carlo tree search. Then we will present how we can progressively reached the lower bound of the complexity for representing a game of Go, software speaking.

1.2 Rules of Go

The rules of Go are extremely simple, and their simplicity 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 play the white stones, the other the black stones. Each player puts a stone on an intersection on the go-ban alternatively.They are two important concepts to understand, the liberties and the group.

A stone has as many liberties as there are free intersections next to it. In figure 1, the black stone has initially four liberties, and it progressively decreases to one :

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

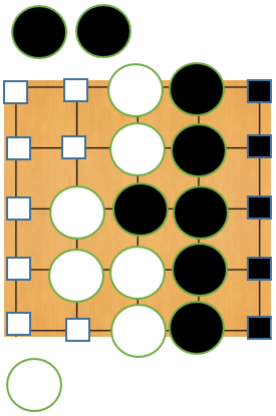


Figure 1.2: Scoring

group is surrounded by the opponent, it has no more liberty, it is dead. All the stones of the group are taken as prisoners, leaving free the intersections where they used to be.

In order to win, a player must have more points than the opponent. A player gains one point by prisoner 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 1.2, the white player win with 10 points (8 from territories and 2 from prisoners) against 6 points for black(5 from territories and 1 prisoner).

Chapter 2

State of the art

In this chapter, we talk about what is Monte Carlo tree search (or MCTS) and how it is applied to Go. First we present the Monte Carlo part (2.1) and the tree search part (2.2) separately, then explain how the fusion works (2.3). Then, in section 2.4, we see the benefits and drawbacks as well as a dilemma in MCTS, exploitation versus exploration.

2.1 Monte Carlo and Tree search

2.1.1 Monte Carlo

Monte Carlo methods are 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 like Physics, computational biology, computer graphics, computer science, Artificial intelligence, and even economics.

The Monte Carlo Method works by sampling enough data to estimate with enough precision a value. To give a very simple example, if it seeks the mean between zero and one, it samples a lot of value between 0 and 1 and find the mean of all this values. With sufficiently high number, it will tend to 0.5.

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. It stops when there are enough value according to the level of precision required.
- 3: Using the data sampled, it can compute results using theses values.
- 4: With the results found, we can deduce other variable of importance using equations or theorems(depending of the domain of application).

We show this four steps through an example, the computation of π. 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,-11).
- Generate random value 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.
- Use the result to get what you wanted: With a sufficient high number of $Total$ we can approximate π thanks to the law of big numbers and assuming

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

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

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

Results have shown that with enough sampled values, it can get really close to the real value of π.

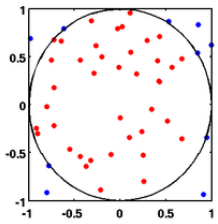


Figure 2.1: Monte Carlo π estimation

2.1.2 Tree search

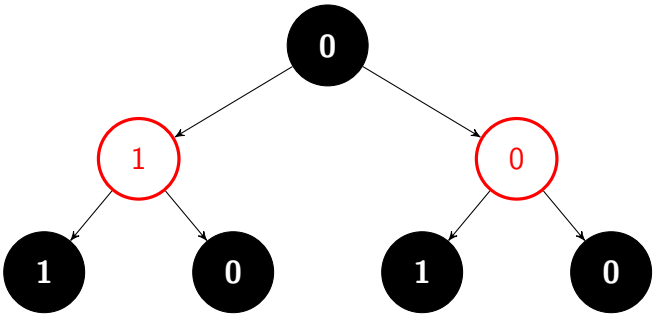
In this section we explain the basics of tree search in artificial intelligence, some techniques already used and how they differ from MCTS. We present notably the mini max algorithm which is widely use in Artificial intelligence to show how his behaviour really differ from MCTS.

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 children for each move possible since that state.

Some assumptions need to be made:

- The game is played turn by turn, not in real time.
- Two players.
- Fully observable

Here's an example of a tree for a game where two players flip a coins and try to maximize the number of heads. The game begin with zero head for each player and the first to play is the red player. Either the coin is head then the game move to the node with one head or it is a tail then the state is the 0 blue node. Then the black player plays and so on. The number in the node represent the number of heads since the beginning for the player of the same color as the node.

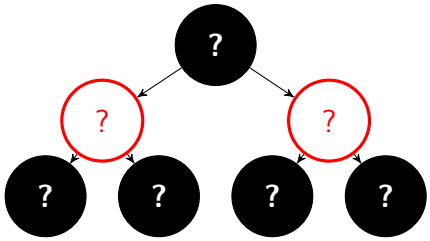


Mini-max

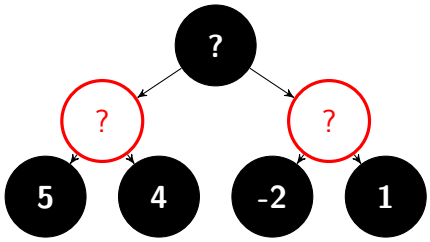
The minimax algorithm is an algorithm used in artificial intelligence games. It gives to each state (or node) of the game a score. Then one player tries to maximize it while the other will try to minimize it. So a bigger score is a better situation for player one. Each action or move will impact the score. Thus, the algorithm chooses the action that eventually bring it into the best state for him, assuming the other player plays the best moves he can. If possible, the algorithm continues to search the tree until a won end game situation but as there is not always enough time, it can limit itself to a certain depth.

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

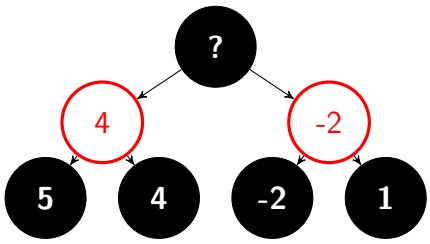
At first, it doesn't know anything :



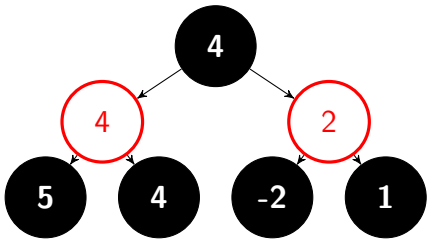
Then it evaluates the value of the leaves.



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



Finally, it 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 suffers some major drawbacks.

- It needs to explore in breadth first search, meaning all nodes from the top, depth after depth. In some case it's just impossible. The complexity of the minimax is $(b^d) * e$ where b is the branching factor(number of possible moves), d is the depth wished, and e is the complexity of the evaluate function. Example give : 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. Of course, there can be a depth limit, but to be computable, the found tree won't be deep enough to make interesting plays.

- The algorithm needs an evaluate function which return a value, a score for a node. It needs a deep knowledge of the game to produce a correct and precise heuristic when it's possible which is not always the case. In Go, it's practically impossible to be able to give a value to a board, or even to say which player is at an advantage. The game can change on a single move.

2.1.3 Monte Carlo and Tree search combined

We now see how to combine these two principles to form the Monte Carlo tree search. The idea is relatively simple. As the total tree of the game is far to big, it only computes the necessary. Which means computing the tree node by node, with information on what nodes build next. And in the end, there is a far smaller tree indicating the thought best move. In the following sections, the process for building the tree is detailed.

Four Steps

This section presents how one execution of the Monte Carlo tree search occurs, through the four steps. Two names are given each time for the steps. As theses steps wears different names, the two names found were put to help the reader. We begin with a tree already partially built(2.2). There's 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 state. The second is the number of times the algorithm passed by that state. E.g : the leftmost leaf, where we can see that the red player won zero times on two tries.

Selection(descent)

The first step is the selection, the algorithm selects the best node following a criteria until a leaf is reached. This criteria can vary, and the different strategies will be explain further. For the moment, it simply is the one with the best ratio won over played. The chosen moves are highlighted in a diamond shape(2.3).

Expansion(growth)

Once the selection phase reaches a leaf node of the tree built, it is the turn of the expansion phase.If the leaf is not a terminal state of the game, the basic

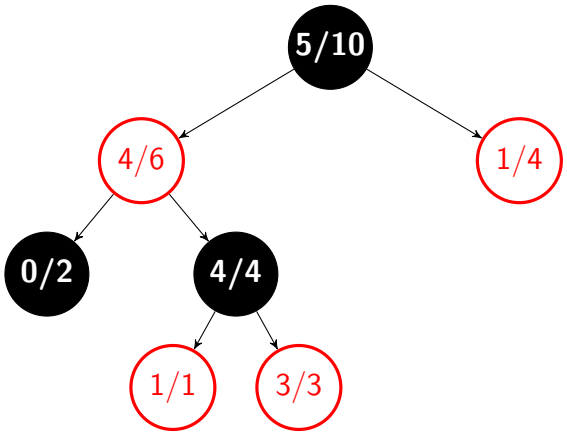


Figure 2.2: MCTS tree

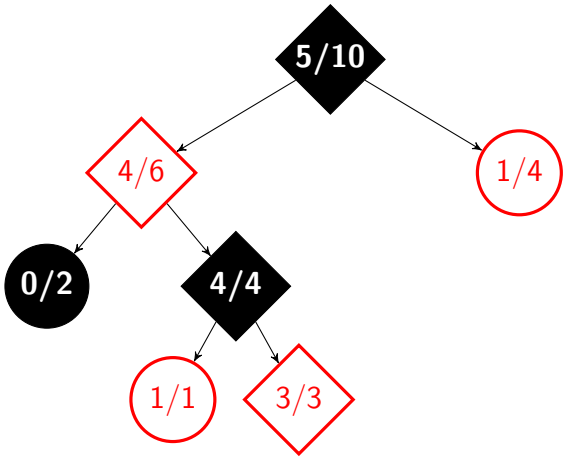


Figure 2.3: Selection step

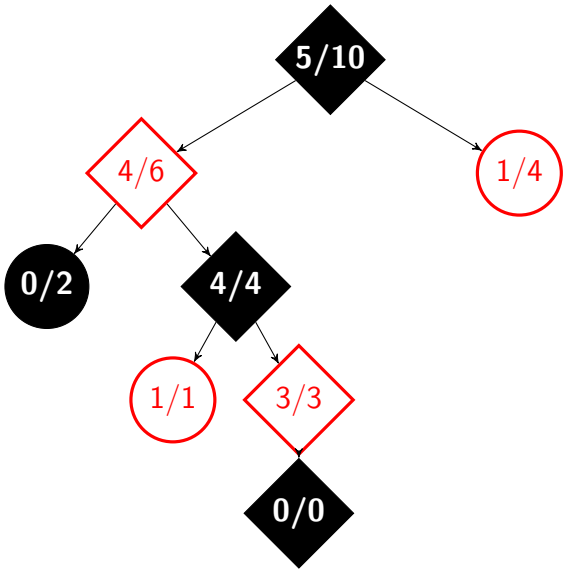


Figure 2.4: Expansion

strategy is to select a random move and add it to the tree with null statistics (2.4).

Simulation(roll-out)

The simulation steps plays until the end of the game with random possible moves for both players.

Backpropagation(update)

Once the game is ended, the algorithm computes the winner. After, it can update the nodes on the path taken, depending of the result. In the example, black won, so the values of all the diamond nodes will be updated accordingly (2.5).

The execution of one iteration is now finished. There is two possibilities, either there's still time, and the algorithm go on, or it ran out of time, and it must choose a move. In that case, it returns the a node following a criteria. With the criteria of the best ratio won over plays, it would return the move on the left. Indeed, by choosing that, it has 5 out of 7 chances to win, where

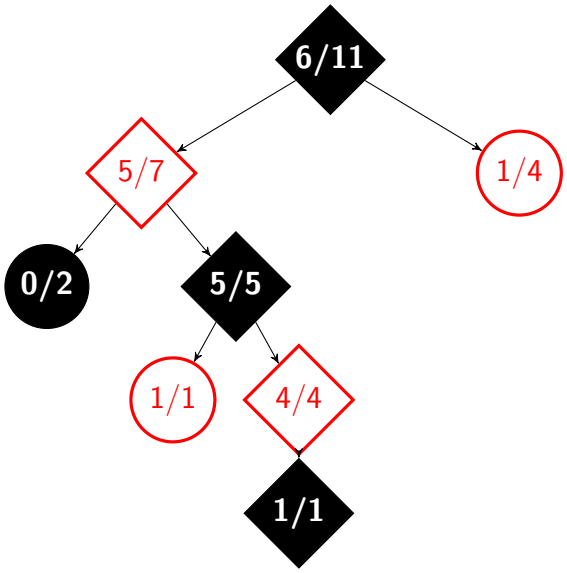


Figure 2.5: Backpropagation

in the other case, he would have less probability to win.

Thus, it's the principle of the Monte Carlo tree search. Only searching where it's interesting. By doing enough iterations, like with the computation of π, we can compute a tree that is big enough to be statistically representative.

Now we explore the different strategies possible for each steps.

2.1.4 Basic Strategies for each step

There are several strategies for each step. To understand well the strategies for the selection step, it must be clear that there's a big dilemma in the application of Monte Carlo tree search. The problem is between two elements : Exploitation, which means taking profits of what you already know, and Exploration, which means go see elsewhere for better. When you exploit, you can't explore and reciprocally. So there's always that question : "Am I on the right path ? should I go further on it, or should I try something else". The following section, with the strategies for the selection step enlightens

some answers.

Selection - Objective Monte Carlo

The goal of this technique is to compute a fairness function for each move. It attributes a probability to each possible move and then selects one randomly, following the probabilities found. The fairness function reflects the balance between the exploitation and exploration.

First the fairness function for each move m is the following, where n_p, n_m is the count of visit for the parent of m, for m respectively, PM is all the possible moves and U(x) is the urgency function of x.

$$f_m = \frac{n_p U(m)}{n_m \sum_{j \in PM} U(j)}$$

U(x) is a function that allows estimating if a move must be played. To be exact, it's :

$$U(m) = erfc(\frac{v_0 - v_m}{\sqrt{2}\sigma_m}$$

v_x is the value of the node x

0 is the best move.

σ_x is the standard deviation of x. The standard deviation for a discrete variable with the same chance for each value is :

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_i)^2}$$

TODO

Expansion

There exist strategies where there is more than one node added by expansion step to the MCTS tree, but as it didn't prove to be more effective, here we keep expanding only one node by iteration.

Simulation

For this step, there are several techniques that are more or less between two extremes :choose a move at random or with heuristics.

	Random	Heursitics
Benefits	Speed	Reflect real play
Drawbacks	Non representative	order of magnitude slower

Backpropagation

They are several strategies again for this step. But no one showed most efficient 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.

- Select the move with the highest count of wins divided by the counts of visits known as Max child
- Select the move with the highest visits count known as Robust child
- Select the move with the both highest count (wins and visits), known as Robust-Max child

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

2.1.5 Benefits and Drawbacks

Benefits

Monte Carlo tree search draws a lot of advantages from his special technique of using tree search and random simulations.

- Aheuristic : There's absolutely no need for heuristic in the refined version of Monte Carlo tree search. All you need to know about the game are the possible moves from a state, and the winning situations for a player. Except that, which are the basic rules, you don't need to know how to play, to be a professional or having a really good vision of the game to implement an artificial intelligence player.

- Asymmetric : In place of having a gigantic tree, taking an incredible amount of space, there's only the needed tree, much much smaller than the complete one. As it's build only depending of the needs, it can be completely asymmetric.

- Any time. One of the beautiful part of the Monte Carlo tree search is that you don't need a specific amount of time. There always is an output. Some algorithms take a certain amount of seconds or minutes, then delivers an answer. Here, the algorithm can always, at any time return an answer. Of course, the more time given, the better the answer. But it allows to stop the search, either after a certain time or a number of iterations through the four steps.

- The basic version of Monte Carlo tree search can be implemented really simply. But his efficient adaptation to complex games is difficult.

Drawbacks

Here's come the second part about the big dilemma talked sooner. There are several difficult choices. By example if the simulation heuristic is too complicated, it takes too much time, leading to fewer iterations and so to not statistically representative results. On the other hand, if it's too simple, it won't reflect real plays and even with a lot of simulation, won't be representative.

There are heuristics, added to the core method of MCTS, that improves dramatically the results. It is explained in the next chapter.

Chapter 3

MCTS applied to Go

In this chapter, we present the actual performance of MCTS in Go, and some heuristics combine with MCTS that deepen and strengthen it

3.1 Monte Carlo tree search applied to Go

Before the introduction of Monte Carlo, 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, the level of all bot jump trough the sky as we can see on the representative figure. For reference, a 8 kyu go player is a good amateur player, and a first dan player is a very good player, approaching professional level.

3.1.1 Improvement

Here we present some heuristic specifically applied to the case of the Monte Carlo tree search on the game of go. The first one is general and the second one is for the expansion step.

MC-RAVE

The idea behind the MC-Rave is rather simple. The basic Monte Carlo tree search algorithm always considers a move dependently from a state. But in Go, they are a lot of situations where a part of the state won't influence the quality of a move. By example there's really little chance that the stones put in a corner of the board will affect the stones in the other corner. That's

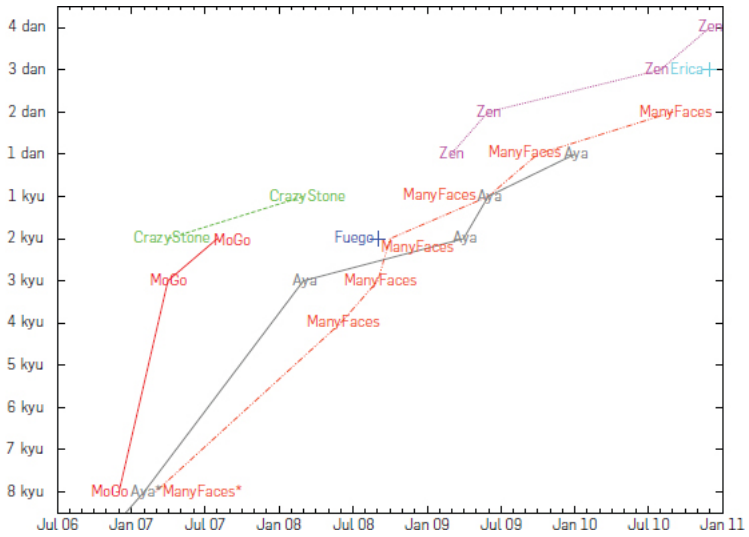


Figure 3.1: Performance of Go AI

where come from the idea of AMAF, or all moves as first. Thus, each move will have a rating consisting of $\frac{\tilde{W}(a)}{\tilde{N}(a)}$ where $N(a)$ is a number of simulation where the move a was played at any time, and $W(a)$ is the number of times the black player won.

Once we got Amaf value for each move, we could make it play with that for all game, but as the games progress, there's more and more influence from the state of the board. Thus, the MC-Rave heuristic corrects that by giving less weight to the AMAF value as the number of plays increase. Here's the equation for the Value of a move "a" from a state s.

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

Where β is a parameter decreasing from 1 to 0 as the game progress, and so the visit count $N(a)$ increases.

Bradley and Terry model

Suppose some people in a competition. This model assign to each player a strength α then it predicts the chance to win of the player i against the

player j to :

$$P(ivsj) = \frac{\alpha_i}{\alpha_i + \alpha_j}$$

There's also possibility to play in team :

$$P(1 + 2vs3 + 4) = \frac{\alpha_1 * \alpha_2}{\alpha_1 * \alpha_2 + \alpha_3 * \alpha_4}$$

As in go there are several types of moves, each move has different feature. Here's a possible list :

- Atari
- Distance of x of previous move
- Secure group
- Next to group/alone

The idea is to give to each of theses features a strength α so the probability of a move a from state s :

$$P(s_a) = \frac{\prod\limits_{features\ i\ in\ s_a} \alpha_i}{\sum\limits_{legal\ moves\ from\ s} (\prod\limits_{feature\ i\ in\ s_j} \alpha_i)}$$

Then we can compute this probability for all legal moves. To limit the search, progressive widening is used. In a lot of case in A.I, the algorithm pruned the search space, here's it's the opposite. The goal is to un-pruned as plays increase. The algorithm will look into the T best moves following the Bradley and Terry model The following formula is used by the developer of AYA, a well known go program, where's n is the number of plays, and μ a parameter :

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

3.1.2 Alpha-Go

During this year, a new bot was released by Google, named alpha go. It was a real surprise for everyone in the field because not only it won against the best European player, Fan Hui, in October 2016 but it also defeated the number one professional, Lee Sedol, in March 2016. So this challenge of Go, one of the last board game where the best humans were still undefeated versus the machine, is now, in a certain way, over. This was unpredictable. Experts said that this kind of level of A.I wasn't expected before a decade. Google really nailed it in a beautiful and successful way. We present shortly their method to accomplish that.

They used deep learning. Deep learning is a machine learning technique. The program alpha go wasn't coded to know how to play Go. Instead, it learned automatically from a lot (30 millions) of plays of professional what is a good plays and what is not. Furthermore, it played against itself to improve further.

Alpha-Go closed a challenge that was running since twenty years. Here the question is, what deep learning cannot do ? The future will tell us.

Chapter 4

Naive version

In this chapter, we present the first naive version of a bot player with Monte Carlo tree search mechanism. We show the base structure for a general Monte Carlo algorithm and then the first really naive adaptation for Go. We finish by showing how bad of a complexity it is. Please note that the naive part is in representing Go, not the structure of the basic Monte Carlo Tree search, which, even basic, is correct.

4.1 General Monte Carlo

4.1.1 Game State

The game state is what represent the state of the game at a moment. This is what is transferred and modified to follow the execution of plays. Here's the principal needed methods :

- 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

4.1.2 Node class

The node class represent a node into the built Monte Carlo tree. Each node must have several attributes listed here :

- Move : The move that lead to the node
- 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 that just moved. To know the actual player

The following operations must be doable on a Node :

- Select a child following a formula, generally UCT
- Add a child to the node
- Update the value of wins/visits

4.1.3 Simple pseudo code

This is a pseudo code for one iteration. It takes in argument a state and return the best move.

4.2 Adaptation to Go

The real problem of adaptation, and the only needed knowledge of the game is to answer : What will I get in GetMoves ? How will a move m impact the board ? And who won in GetResult ?

Theses are three methods that need to be adapted.

We first answer for the game, then show a bad programming representation.

Note : when we talk about the neighbourhood of a stone, it's the four intersections in the following directions : up, down, left and right.

```
   Input: A state of the game r
   Output: The best child move of the root
 1 root=Node(r)
 2 node=root
 3 state=r
   /* Selection step                                          */
 4 while node.children not empty do
 5 |   node=node.UCTselect
 6 |   state.DoMove(node.parentmove)
   /* Expansion step                                          */
 7 if node.untriedmoves not empty then
 8 |   m=untriedmove.random
 9 |   state.DoMove(m)
10 |   node.addChild(m,state)
   /* Simulation step                                         */
11 while state.GetMoves not empty do
12 |   state.DoMove(state.GetMoves.random)
   /* Backpropagation step                                    */
13 while node is not None do
14 |   node.update(state.GetResult)
15 |   node=node.parent
16 return best(root.childs)
```

4.2.1 GetMoves

In general

The possible moves are seemingly simple. We can play only on free intersection and we cannot make suicide plays except if it kills a group. We can define formally a legal move as next :

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

Where F means that the intersection is free, nothing on it, E is for empty intersection in the neighbourhood, K is if it kills and enemy group and G is if there is a friendly group in the neighbourhood with more than one liberty.

Except there only one more problem, the ko. To prevent the game of Go being unbounded, this rule was added. It says that you can't make a move that makes the board the same as the play before. So A and B being state of the board the game cannot make $A \Rightarrow B \Rightarrow A$.

So the correct proposition for a possible move is :

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

In programming

The board was represented by a double arrays of integer. 0 for a free intersection, 1 for player black stone, 2 for player white stone. Checking F is easy and in $\mathcal{O}(1)$. What about the rest : checking E is also easy and in $\mathcal{O}(1)$.

Now comes the hard part. How to check is a group is alive, being and ennemy, or a friend. To do this badly, it used a stack to explore each group. It takes a first stone, put in on the stack. Then for each stone on the stack, it looks at his neighbourhood and put the stone of the same color that wasn't already explored. It's in $\mathcal{O}(l)$ where l is the length of a group.

So to check the conditions, the algorithm first check the intersection if it's free, then check the neighbourhood, and if's free it's ok, if it's only stones, it checks their liveliness.

Moreover, it doesn't solve Ko problems, to do that, the naive algorithm keep in memory the precedent board and for each move check each position between the last and the new and verify it's not the same. It's in $\mathcal{O}(n^2)$ where n is the size of the board.

So a simple call of GetMoves needs each position to be checked, leading to a complexity of $\mathcal{O}(n^2 * (n^2 + l)$ and so we have a factor n^4 which is unacceptable. We'll see on the following chapter the solutions found.

4.2.2 DoMove

In Go

Of course the primary effect of putting a stone, is playing the stone on the board. Then if that stone deprived from liberty an enemy group, it suppress it.

in programming

As explained in the precedent part, playing a move changes the integer of the intersection ($\mathcal{O}(1)$) and then checks for enemy groups in the neighbourhood and check their liveliness $\mathcal{O}(l)$.

4.2.3 GetResult

In Go

The game is over because they are no more playable moves, or because the two players passed. You have the right to pass in go, if they are no more interesting plays possible.

Then the count of points begin, as a human, you simply counts the territory defined and the prisoners.

In programming

This is a tricky part. Because it's really hard for a computer to define a territory with certitude. So the only way for the algorithm to simulate until the end is to simply run out of possible moves. So the board is filled with stones and living group with two eyes. Thus, the count of point become simple, in addition of the prisoners, you just have to look for free intersection and look for the neighbourhood stones. Then give one point to the owner of the closes stones.

4.3 Total complexity

We study here the total complexity of an iteration of the four steps of Monte Carlo tree search.

The process of an iteration is first a Clone of the state. Then in the worst case, at the start where the selection step is null, we have to expand one node then simulate until the end. Which means get a move, then do it, until the board is complete, which we can estimate to n^2 times, the number of intersections. Then it has to Get the result and then update the node on the way

- n is 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

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

Which gives = $n^2 + n^6 * l + n^2 * l + n^2 + d$

Worst complexity = n^6

This is impossible to maintain, the number of operations for $n = 9$ is 531441 for one iteration. The algorithm will never be able to reach statistically representative solutions. That's why this solution had to be dramatically improved, which is the following chapter.

Chapter 5

Optimization

In this chapter we present solutions to the problems encounter in the last part. We show the abstract structure used to drop the complexity.

We ultimately show the progress in complexity made

5.1 Union Find, or how to solve the group problems

5.1.1 Presentation of Union Find

The union find is a special data structure that present the next several advantages :

- You can find the representative element of a group in $\mathcal{O}(l)$
- You can make the union of two group in $\mathcal{O}(l)$
- Note : To be completely exact, the complexity is the inverse of the Ackerman function which can be approximated to one.

It works thanks to a structure of node. Each node has two attributes, a rank and a parent. Here's how it works

Find(x)

The find operation take into argument a node and return the representative element of a group.

Algorithm 1: 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 = Find(x parent)
5 |   return x parent
```

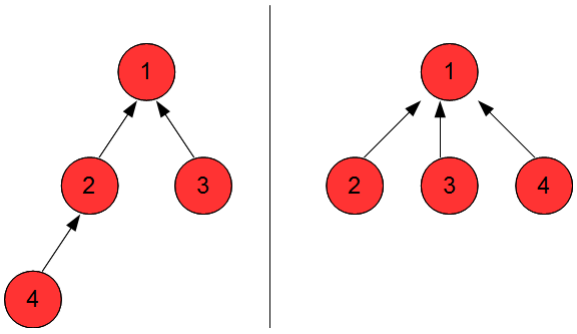


Figure 5.1: Before and after Find(4)

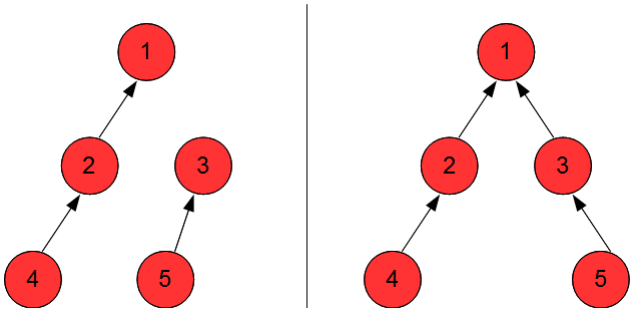


Figure 5.2: Before and after Union(2,5)

We can see (5.1) that each time a find is done, it balances the tree, to put all the nodes on the path closer to the root which optimizes the Find time the next operations.

Union(x,y)

This is the pseudo code for the union method. We can see that's the group with the bigger rank who become the root because it makes a smaller total depth.

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

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

5.2 Improvement 1

The idea is to use the union find data structure to represent the groups. All stones begin as a single node. Then when a stone is put next to it, we can

union the two and so on. The GoState won't represent the board with a double array of integer but with a double array of Union-Find nodes. Each node contains the basic attribute and his color. Further more the parent node of each group conserve two sets. One maintains all the nodes actually in the group and the other maintains all the liberties of the group itself.

So what is the complexity of DoMove with theses data structures ?

Playing a stone is in $\mathcal{O}(1)$. To check if an enemy group in the neighbourhood is alive, we look at the size of the liberty set, if it's zero, it's dead. This operation is in $\mathcal{O}(1)$. The only problem with DoMove and this method is during the union operation. As we changed the Union-Find data structure to add two sets (components and liberties), the complexity changed as well. Because we need to union the sets as well, the complexity of the union operation become $\mathcal{O}(n+m)$, where n and m are the size of the sets of the two groups.

5.2.1 Special structure to solve the GetMoves complexity

A big problem of the previous naive version was the GetMoves with a complexity of $\mathcal{O}(n^4)$ because the algorithm had to check each position n^2 and then for the ko each time each position again n^2.

Ko problem complexity can be easily solved, by keeping in memory a forbidden intersection. The only possibility for ko to happen is to capture a solo stone that just captured an enemy solo stone. So in this situation, we can simply remember the intersection and forbid to play on it the next turn.

This take the n^4 to n^2. But it's still a problem because as in one iteration of the Monte Carlo tree search we still have to simulate the game until the end, we have as a lower bound $\mathcal{O}(n^2 * (GetMoves + DoMove)$. If the complexity of get moves is n^2, we'll have a n^4 in total, which is to much for size of board of 19 by example ($19^4 = 130321$ operations for only one iteration.)

So it became clear that it was no longer possible to check each time each intersection. The first idea is to maintain a list of possibles moves in the state and to transfer this list between the states. To maintain it, there's two possibilities,either updating it each time a move is done(removing now unvalid moves,adding possible ones) or each time the method GetMoves is called check if the returned move is ok, if not remove it from the list and

random a new one.

The problem is that inserting or removing an element of a list is in a complexity of the length of the list. Using a set would be nice to get the complexity of insertion and removing in $\mathcal{O}(1)$. There's still a problem. The function GetMoves is used in this Monte Carlo algorithm for two things, either in the simulation steps 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. This isn't possible in a set.

So I used a special data structure which is the combination of a list and a hashmap. I'll present here how the methods insert, remove and getRandom works in $\mathcal{O}(1)$

To insert an element x:
List.append(x)$\Rightarrow \mathcal{O}(1)$
Hashmap(x)= index of x in List $\Rightarrow \mathcal{O}(1)$

To remove an element x :

We exchange the last element of the list with the one to suppress(step one to four), we pop the end of the list(step 5) and we suppress the value in the hashmap(step 6). It gives something like that:

Input: A value toRemove
1 last= List[len(List)-1]
2 ToRemove = hashmap[x]
3 List[ToRemove]=last
4 Hashmap[Last] = ToRemove
5 List.pop
6 Hashmap.delete(x)

To get a random element :
List [random(List.size)] simply.

5.2.2 Complexity

Remember the total complexity is $(CL + n^2 * (GM + DM) + GW + Update)$

Where GetMoves is actually in $\mathcal{O}(1)$ and DoMove $\mathcal{O}(s1 + s2)$

Our actual worst complexity is $\mathcal{O}(n^2 * (s1 + S2))$ which is quite good, and a lot better but still not the lower bound ($\mathcal{O}(n^2)$).

5.3 Improvement 2

5.3.1 Remember the component

So, actually our complexity is $\mathcal{O}(n^2 * l)$ where l is the size of the set of the liberties of the group after union. To reduce that factor, we had several challenges. First, destroy a group in $\mathcal{O}(1)$. In order to achieve that, we had to remember somewhere the components of the group. We keep it so in memory of each group in the form of a linked list. Thus, when two group unite, the linked list of one is simply appended to the other, this is done in constant time. Then when we want to effectively destroy the group, we pass by each node and reinitialize his attributes. This takes l operations, where l is the size of the group. But we can amortize this complexity. In place of considering l operation to remove a group, we consider one operation per stone. As it cost one operation to play a stone, and one to remove it, we see it as constant.

5.3.2 Remember the liberties

Now there's still the problem of remembering the liberties. To achieve this, we also used a linked list of pointers to the neighbourhood. We show in the next figure how it's done.

First when a stone is played, it adds to his list of liberties the four intersection next to it, See figure 5.3 (the list of liberties is represented in green circle in the pictures).

When a stone of the same color is played next to it, it first adds the four intersection in his neighbourhood then union with the same color stone. It can lead to duplicate or to pointers to node where there is stone of the same group(as in the picture, the white group has in his neighbourhood list the two white stones as well).

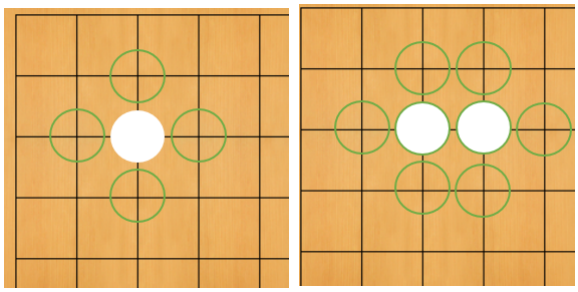


Figure 5.3: First play (left) and Union(right)

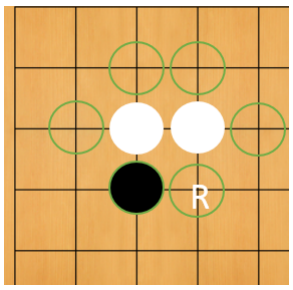


Figure 5.4: Opponent plays

What happen when an opposing stone is played ? That's the interesting part. Here, black played next to the white group. So, in order to verify if the white group is alive, it runs through the pointers, until it finds a free intersection. If not, the group is dead. During this process, it also removes from the pointers the intersection with the same color as the group tested. Because it means theses are stones of the group, and it's useless to keep them in memory. Furthermore, we remember the intersection that were free. Like that, for the next check of this group, we have high probability to find directly the free intersection. See figure 5.5 (The R node is the node remembered for next check).

5.3.3 Complexity

What is our complexity at this point ?

If we suppose we don't delete group during a game, our complexity is

the lower bound n^2. Because we do only a constant number of operations in the worst case. Playing a stone cost one operation, checking the liveliness of a group cost at worst the whole group and his neighbourhood thanks to the RESIDUS(translate in english) system for all the game, which is n^2 operations amortized on n^2 plays. But in Go there is group suppression. So even if this model is better than the previous one, because the multiplicative factor show more occasionally, it's still not good enough.

5.4 Another improvement

To solve the problem of reaching the lower bound, we had to use another version of the Union-Find data structures. It's a version that allow to delete an element of the tree in constant time, while keeping the union and find operation in constant time too. We present it now how it would have work with Go.

5.4.1 Union-Find-Delete Model

Assuming we have a union find delete(UFD) data structure, it allows us to make a model that reach the lower bound.

The GoState is represented by a double array of Node. They are used in UFD. Each group conserves his liberties with another UFD data structure. There will be the UFD tree with the stones and group of stones and each of this group will have a reference to an UFD with the liberties.

As an intersection can be a liberty in different group, there cannot be only one node to represent it. So we model each intersection with 5 node. One for the stone, used to model the group, and four for the liberties, one for each direction(Right, Left, Up and Down). By example, the node (X,Y,R) is the node in coordinate x,y and that is used as a liberty by a node (X,Y+1).

In the example, the stone is in (1,1). His group is with one stone, his liberties with four.

DoMove

There are several operations that need to be in constant time when a move is played.

- Put a stone on the board on x,y $\Rightarrow$ constant time ✓

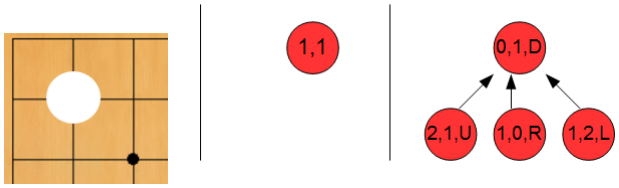


Figure 5.5: Stone on the board, UFD of the group, UFD of the liberties

- Remove liberty for adjacent group ⇒ Delete in UFD is in constant time ✓
- Union with another group of the same color ⇒ Union of UFD is in constant time ✓
- Check liveliness of enemy group ⇒ Are there still element in UFD of liberty ? Constant time ✓
- Suppress dead group ⇒ One deletion by element, amortized in constant time ✓

Our DoMove operation complexity is so $\mathcal{O}(1)$. And so our final complexity for one iteration in the worst case is $\mathcal{O}(n^2)$, the lower bound.

5.4.2 Drawbacks

The major inconvenient with this data structure is his complexity to implement it correctly and to adapt it efficiently to the model used. By reasoning about this data structure and his different implementations, I found an implementations using vacant nodes, which means the deleted nodes stayed in the tree, it's just his data that is deleted. I then went step by step on this reasoning. First, maybe I could maintain a size of all the real node, without the vacant ones. Secondly, in this case, I don't need a Union find delete, a union find with maintaining the real size is enough. And last but not least, I realized I didn't even needed an union find set for the liberties. We explain the concept in the next section.

5.5 Final improvement

Our final idea to manage the liberties of a group should allow a check in $\mathcal{O}(1)$ and a maintain in $\mathcal{O}(1)$. So, to conclude on all the ideas that were presented here, we finish with a very simple one. Indeed, to maintain the liberties of a group, we simply remember how many of them each group has, only numerically.

5.5.1 Main idea

Each stone when put, removes one liberty four times to his neighbour group in each direction. If it's the same group in three directions, it reduces the liberties of this group by three. Then, the stone gains one liberty by empty intersection next to it. And finally, it unites with any neighbouring groups. When two groups make a union operation, the resulting group has as many liberties as the sum of his two components.

5.5.2 Proof

Let's start with a single stone. It has $l = 4 - nn$ liberties where nn is the number of neighbouring stones. If another stone is grouped with this one, it adds up their liberties. So we can see that if an intersection is a neighbour of the same group in more than one direction, it counts several times.

So 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 has the group in his neighbourhood.

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

5.5.3 Conclusion

And so it is complete. Playing a stone, removing a group and checking if a group is alive are all in amortized $\mathcal{O}(1)$.

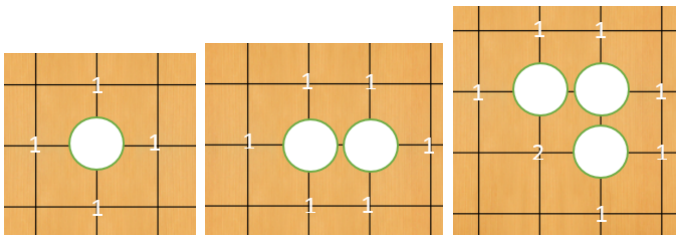


Figure 5.6: The group of stones remembers having respectively 4,6 and 8 liberties

The only problem left is knowing if a move is playable. Before we had mechanism to suppress invalid moves but they worked only because we knew which nodes were the liberties of which group. So, we had to come with a new method to maintain the list of playable moves.

As before, a move (x,y) is removed from the list of possible moves when a stone is put on (x,y) and added to the list when a stone is remove from the board on (x,y).

But now, when the MCTS algorithm tries to play a move, it first checks if it is playable, if it's not, it is withdrawn of the possible moves. Then the algorithm random another move and try it.

Chapter 6

Conclusion

Appendix A

Appendix Title