Monte Carlo Tree Search applied to Go

reaching Ω

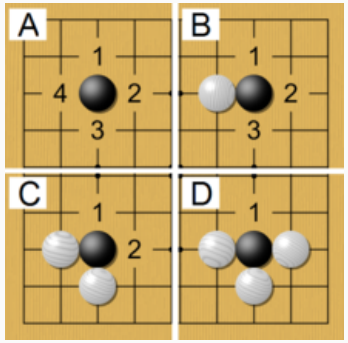
Thibault Vandermosten

July 5, 2016

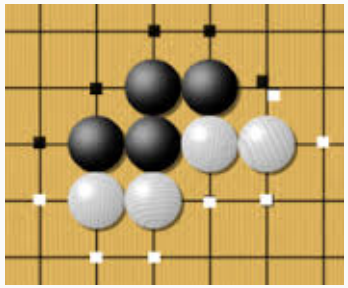
Table of contents

Rules of Go

Rules of Go : Liberties and group



(a) for a stone



(b) for a group

Figure 1: Liberties

(a) for a stone

(b) for a group

Figure 1: Liberties

Go is played by 2 players. One with black stones, the other with white stones. It's turn-based. The only possible move for each player is to place a stone on an empty intersection.

A stone has as many liberties as free intersections next to it. In the left picture, the black stone begins with four liberties and it progressively decreases to one.

In the figure 1.b, there are different groups. A group is composed of several stone of the same color in contact altogether. There are one black group of four stones, and two white group of two stones. Group share their liberties. Each white group has four liberties.

When a group has no more liberties (when it is completely surrounded by opponent's stones), it is captured, meaning the stones are removed from the board and are taken as prisoner.

Rules of Go : Score

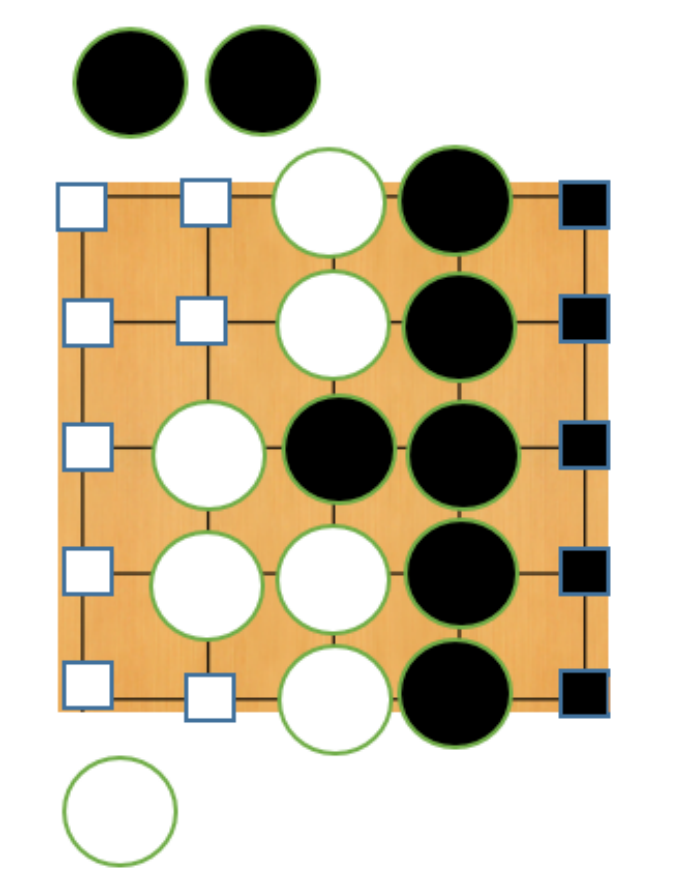


Figure 2: Scoring

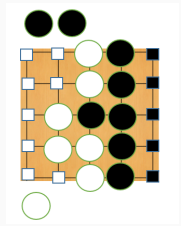
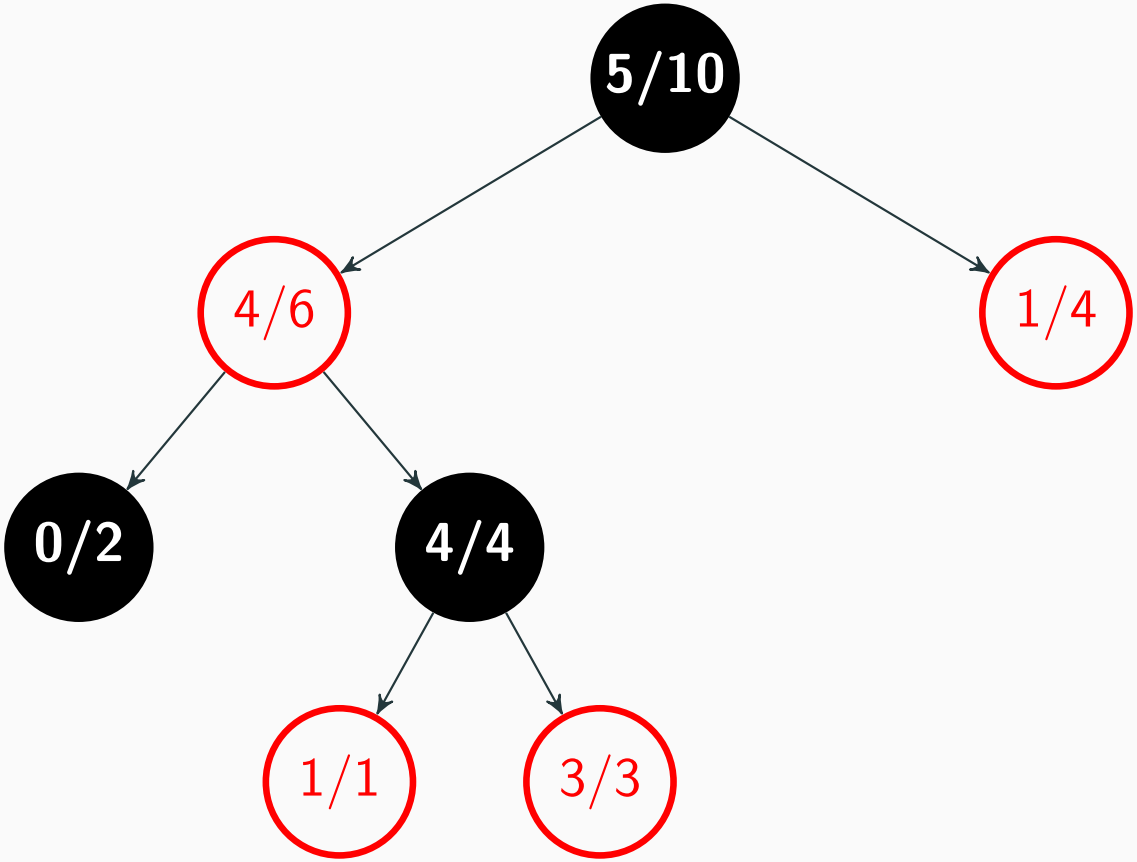


Figure 2: Scoring

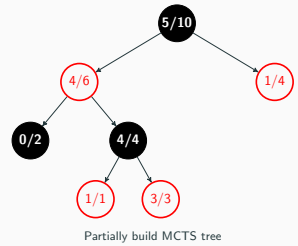
The game ends when there is no more possible moves, or when each player passed , estimating there is no more interesting moves.
Each player gains one point for each prisoner, and one point for each territory.
A territory is a free intersection indirectly bordered by the edges or by stones of the player.
In the figure 2, the black player has 6 points : 5 points of territory (the black squares) and 1 prisoner.

MCTS

MCTS



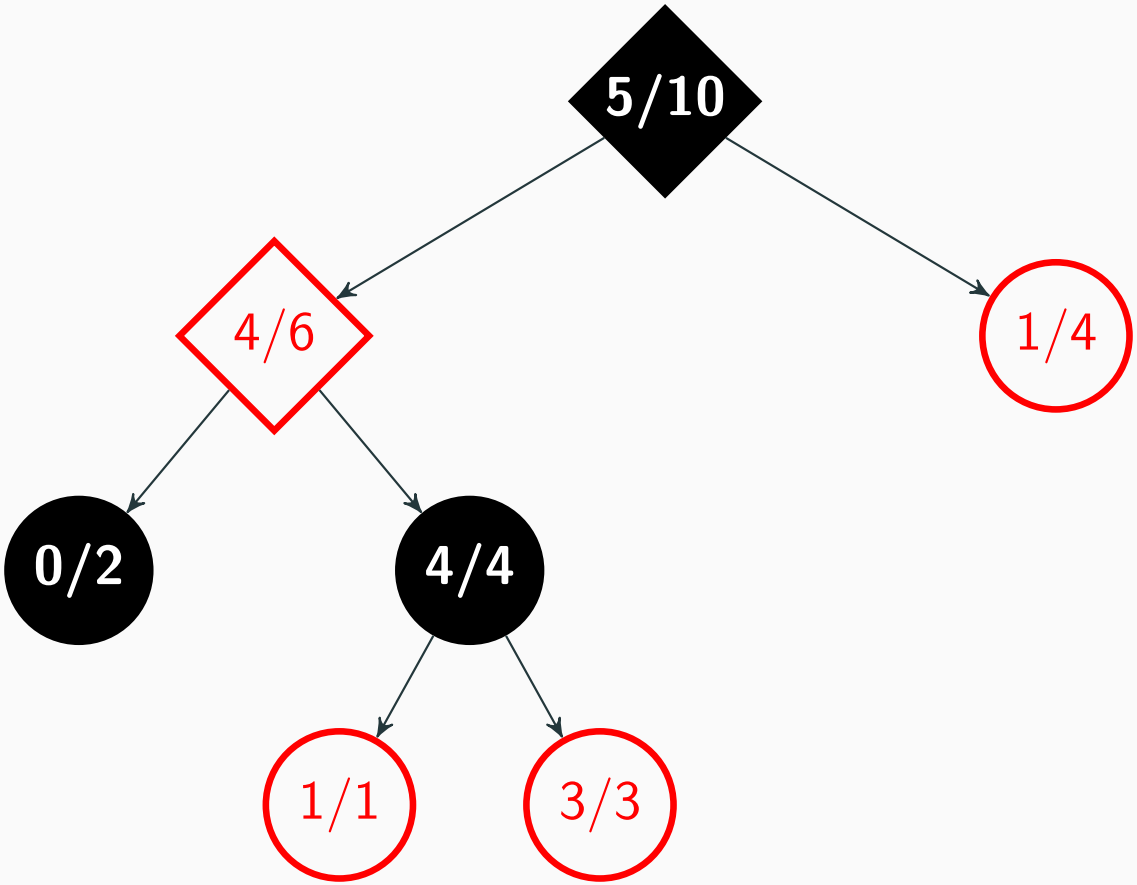
Partially build MCTS tree



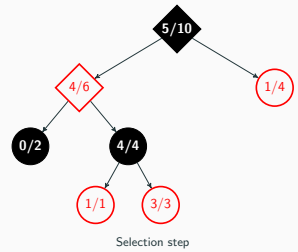
Monte Carlo Tree Search (MCTS) is an algorithm that takes a board situation and try to return the best move. It progressively builds a tree, iteratively. Each node represent a board situation, and each transition is a move. Each node contains two values. The first one is the number of time the black player won passing by that node, the second one is the number of times the node was visited.

To explain simply one iteration, we start with a partially pre-built tree.

MCTS



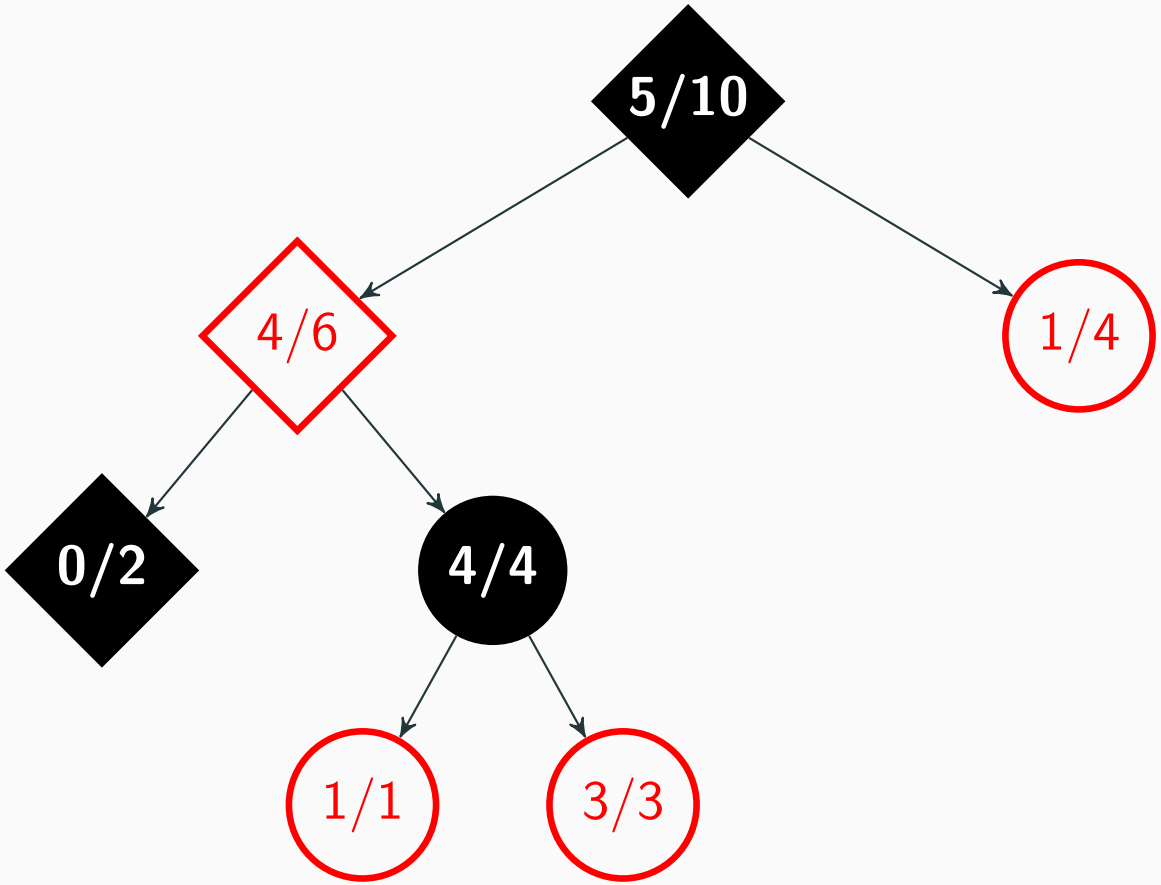
Selection step



The first step, the selection step, is going down the tree until a leaf, or a node not completely expanded. We start at the root, the black node with 5/10. Black has to play. MCTS can choose between a node with an estimated chance of winning of 4/6 or 1/4 (1 win on 4 visits). Thus, it chooses the 4/6 node.

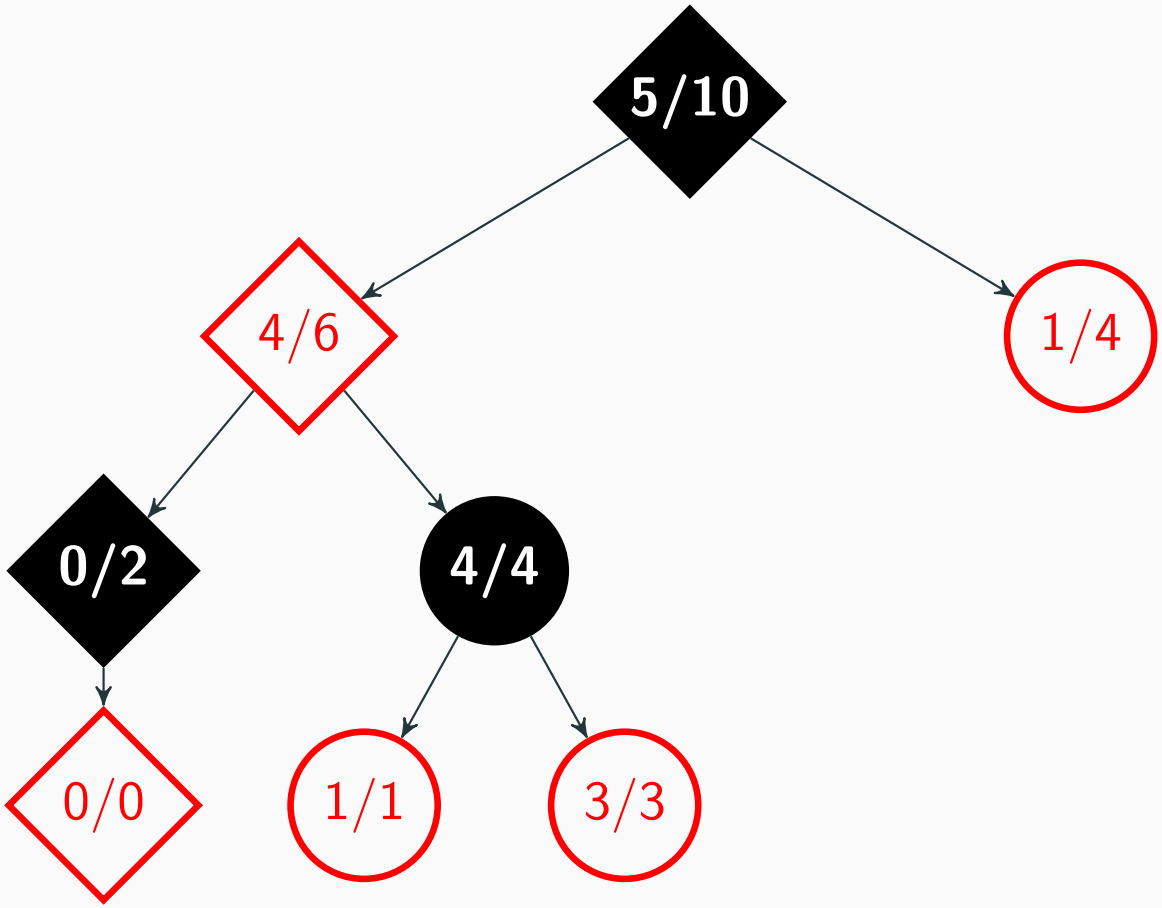
White has now to play. MCTS can choose between 4/4 or 0/2, remember it is the values for black. Thus, it is better for white to select 0/2.

MCTS

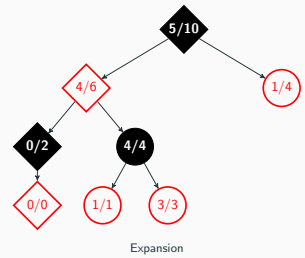


Selection step

MCTS



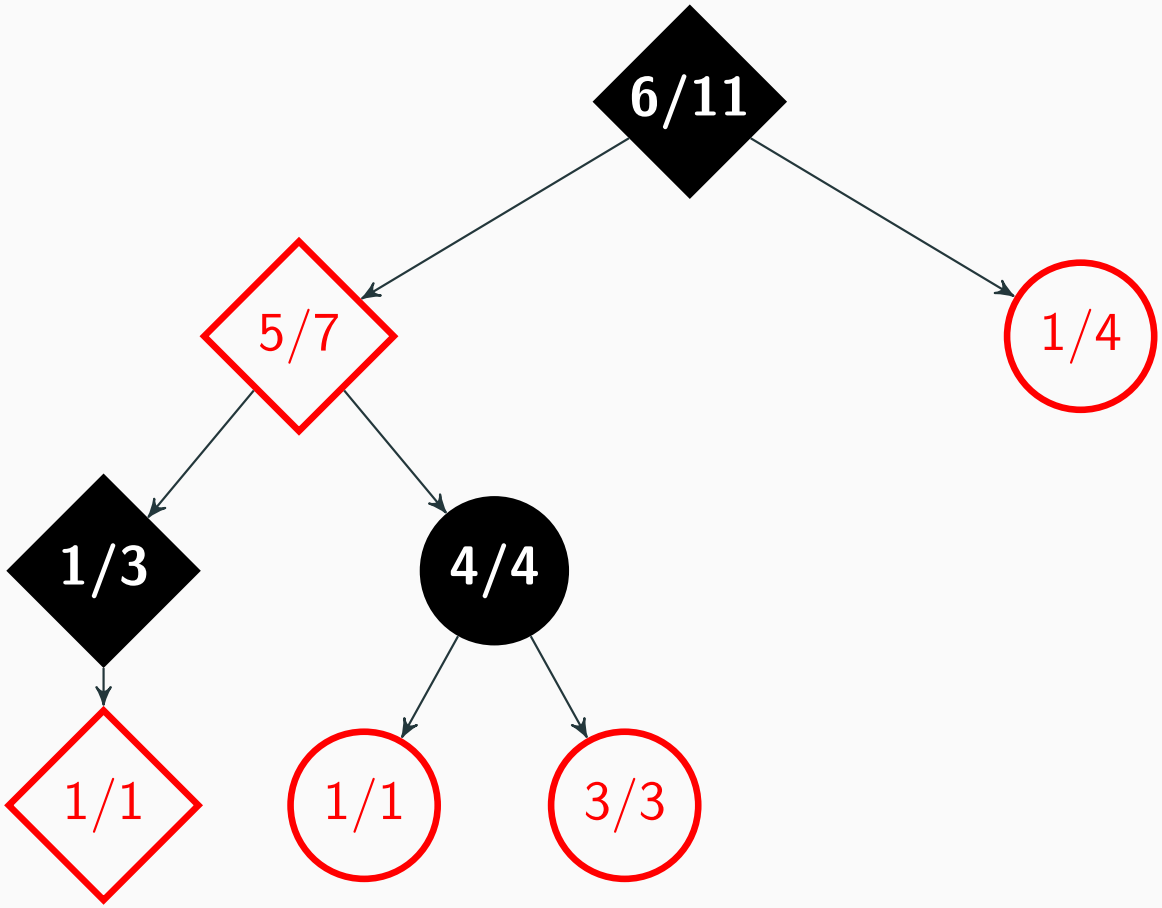
Expansion



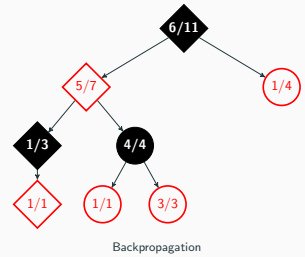
Now that we reached the end of the tree, MCTS picks randomly a possible move from that state and add it to three with 0/0 statistics.

Once done, there is the simulation step. The algorithm simulates the game from that state with random moves until an end game situation. We suppose here that black wins in the end, leading to a 1/1 value for the node.

MCTS



Backpropagation



The last step, the backpropagation rides up to the root of the tree on the path selected in the first step, and update the values of the nodes. 0/2 becomes 1/3 and so on.

MCTS : Why

Aheuristic : No need for knowledge of the game outside of the rules.

Non-exhaustive : No need to explore all the tree.

Size of the tree : b^d (b = average branching factor; d = average *depth*).

Go tree	Chess tree
b=250	b=35
d=200	d=37
Size $\simeq 10^{479}$	Size $\simeq 10^{57}$

MCTS operation on game state

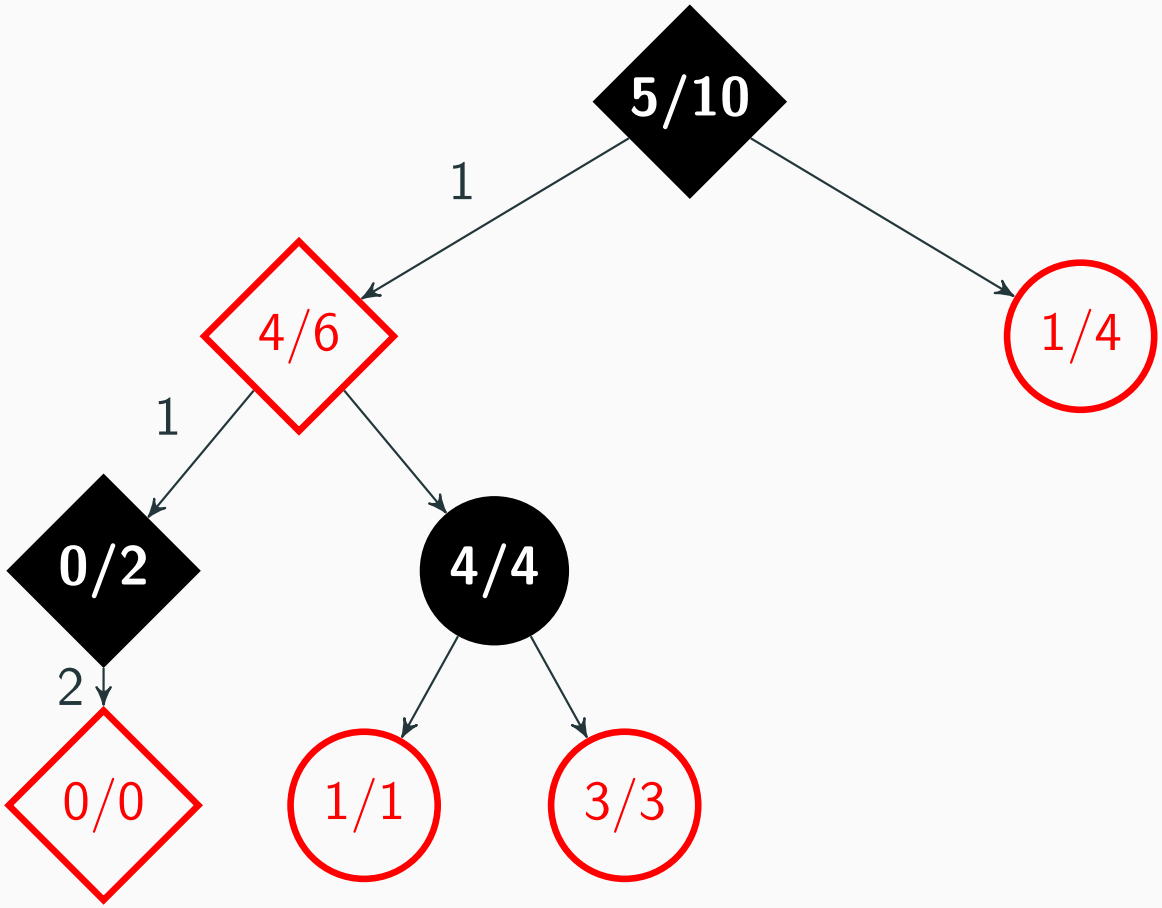


Figure 4: 1 : Do Move ; 2 : Get Random Move

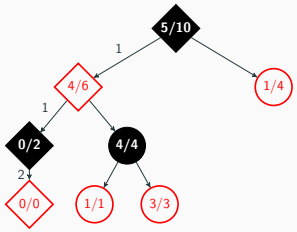


Figure 4: 1 : Do Move ; 2 : Get Random Move

The only operations needed on the game state are DoMove during the selection step, and Get a random move during the expansion step. The simulation step is an alternation of DoMove and GetRandomMove and the backpropagation doesn't affect the game state.

MCTS operation on game state

Get a move at random :

- Get the list of possibles moves
- Randomize one of them

MCTS operation on game state

DoMove:

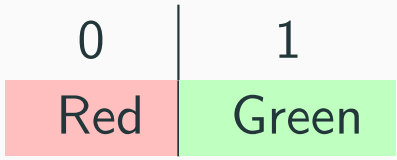
- Put a stone
- Remove from the possible moves
- Updates liberties of adjacent group
- Remove enemy group and update possible moves if needed
- Union with allied groups

Abstract representation

We needed a data structure with the following operations in constant time : insert, remove and get random. Nor the set or the array allowed all of them in constant time. We thus used a combination of them.

Array/set

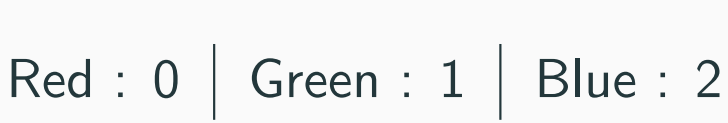
Red : 0	Green : 1



Array/set

The data structure use a dictionary (on the left) and an array (on the right). It contains two value : Red and Green. Each value is the key in the dictionary with a value corresponding to their index in the array. Red is in index 0, thus it has 0 as value in the dictionary.

Array/set : Insert



(a) Add

(b) Append at the end

Figure 6: Insert Blue

(a) Add

(b) Append at the end

Figure 6: Insert Blue

If we want to insert a value, we append it to the array, and insert it in the dictionary with a value equals to the array size minus one.

Array/set : Remove Green

Red : 0	Green : 1	Blue : 2

Red : 0	Green : 1	Blue : 1

(a) Adapt set

(b) Switch value to remove with last

Remove Green

Red : 0	Blue : 1

(a) Remove

(b) Pop last value

(a) Remove

(b) Pop last value

If we want to remove a value, here green, we switch the last value of the array with it. Then the value switched we won't remove is updated in the dictionary (Blue : 2 becomes Blue : 1). And we can finally remove from the dictionary and pop the last value of an array, both operations doable in constant time.

Get Random

Get a random element between 0 and Array size

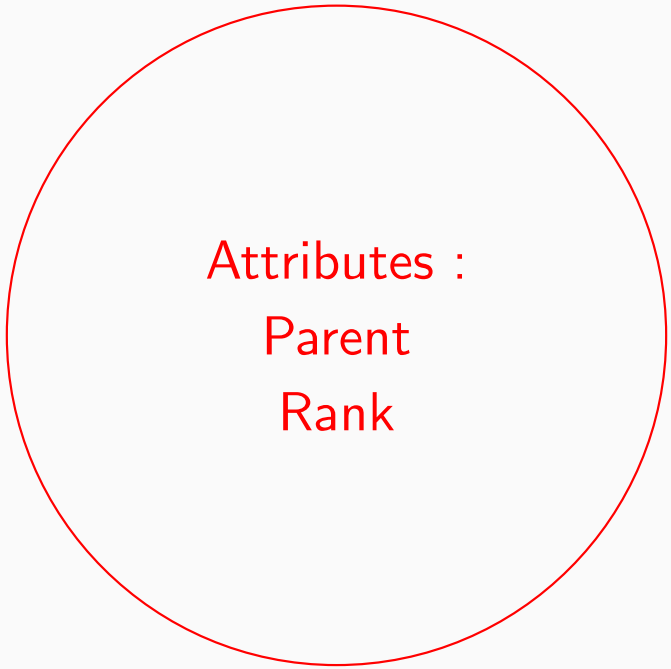
Figure 10: return Array[random(0,1)]

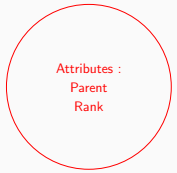
Get random move in 0(1)

Get a possible move at random :

- Get the list of possible moves $\Rightarrow$ ✓
- Randomize one $\Rightarrow$ ✓

Optimal representation : Union Find





We had to use a structure to represent the group of stones. We used the Union Find. It allows two operations : Union of two groups in constant time and Find the group of an element in almost constant time. For this purpose, each node has two attributes : its parent and its rank.

Optimal representation: Find

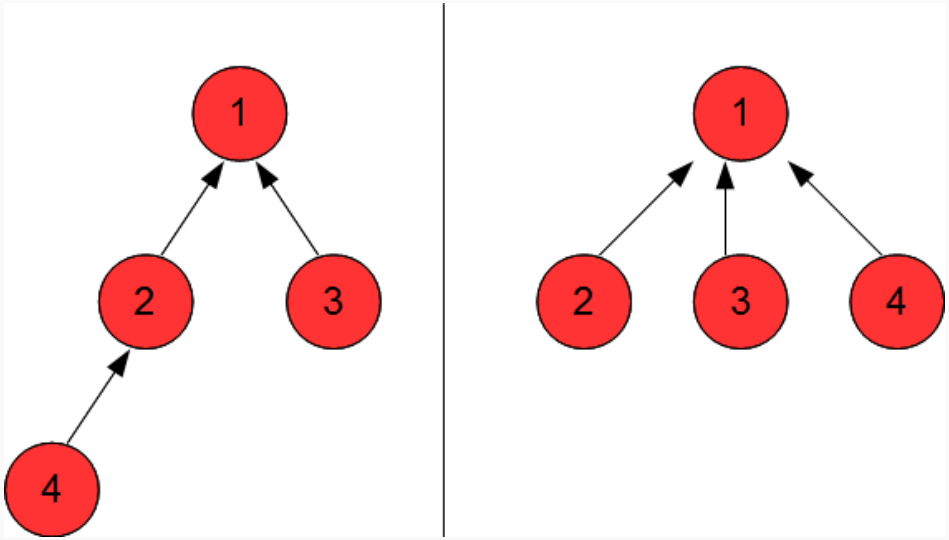


Figure 11: Find(4) return 1

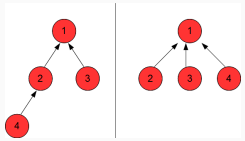


Figure 11: Find(4) return 1

Here we have an union-find group with 4 elements. The parent of 3 is 1, e.g. If we try Find(4), we ride up to the parent until we reach the root : 1 (parent of 4 is 2, parent of 2 is 1). To optimize the following operations, each node on the path has its parents set to the root. The next time Find(4) will be called, it will take only one operation instead of two.

Optimal representation : Union

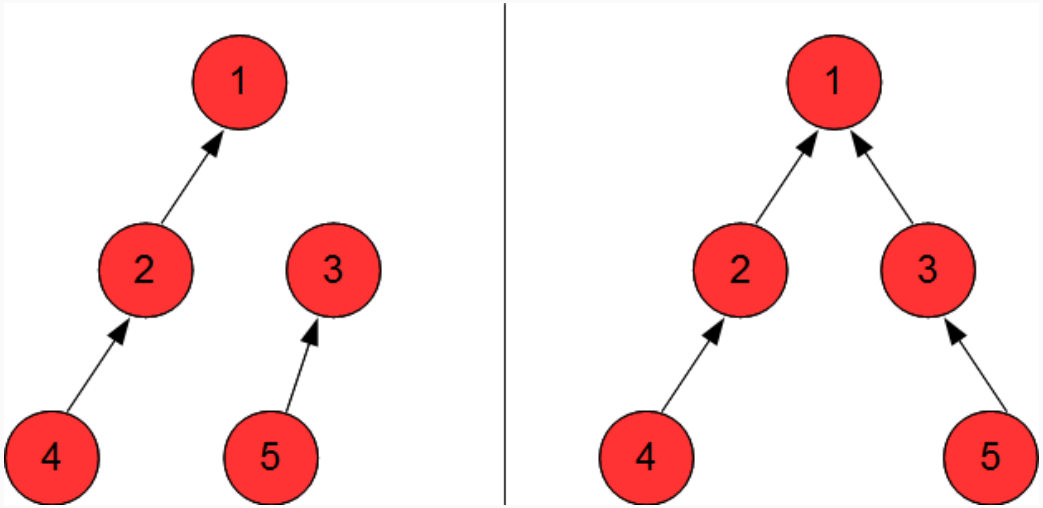


Figure 12: Always produce minimum final depth

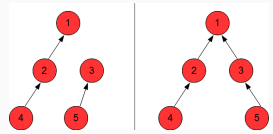


Figure 12: Always produce minimum final depth

Thanks to the rank mechanics, it will always be the tree with the smaller depth that will be linked to the root of the other. Thus the resulting tree will have the smallest depth possible. As it only consist in setting the parent of the root of one tree as the root of the other tree, it is in constant time.

Optimal representation

- Find complexity $\simeq$ constant time
- Union in constant time

We can add attributes to the node, as long as Union stay in constant time.

$\Rightarrow$ Add a linked list to remember members of a group

Optimal representation : Liberties

Represent numerically : each group remembers an integer relative to the number of liberties.

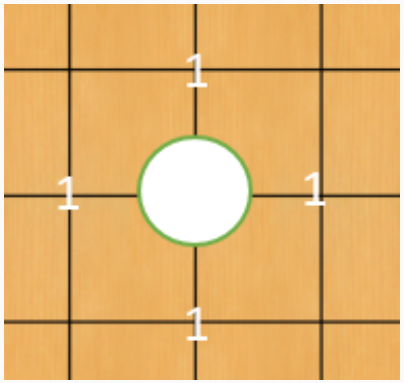
In practice, we just need to know if the group has > 0 liberties.

Optimal representation : Liberties

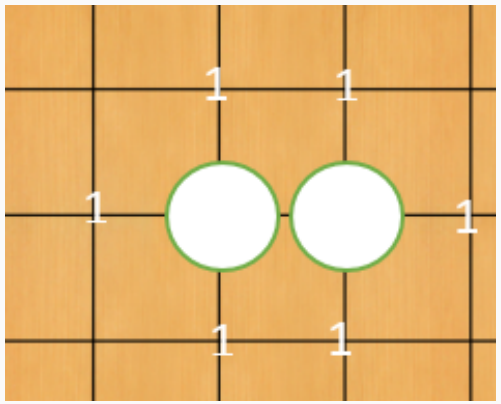
When a stone s is placDat :

- CountLib(s) $= 0$.
- CountLib(s)+=1 for each free intersection next to it.
- CountLib of each neighbouring group g -= a, a is the number of times g and the stone connect.
- Union : CountLib of $g \cup s$=CountLib(g)+CountLib(s).

Optimal representation : Liberties



(a) CountLib=4



(b) CountLib=6

Optimal representation : Liberties

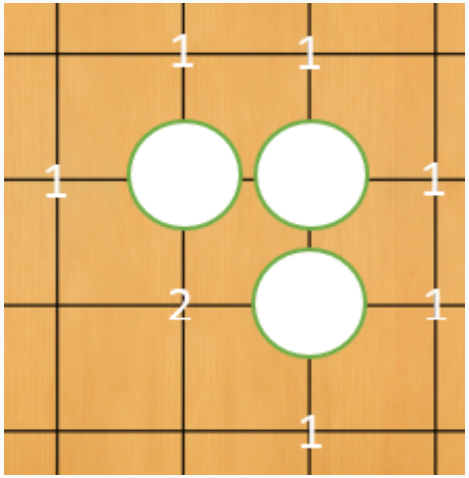


Figure 14: CountLib=8

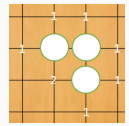


Figure 14: CountLib=8

With this representation of liberties, we will be sure that a group will have CountLib==0 if and only if it is completely surrounded.

DoMove in 0(1)

DoMove:

- Place a stone $\Rightarrow$ ✓
- Remove from the possible move$\Rightarrow$ ✓
- Updates liberties of neighbouring groups$\Rightarrow$ ✓
- Remove dead enemy groups$\Rightarrow$ ✓
- Union with allied groups $\Rightarrow$ ✓

DoMove in 0(1)

DoMove:

- Place a stone ⇒ ✓
- Remove from the possible move⇒ ✓
- Updates liberties of neighbouring groups⇒ ✓
- Remove dead enemy groups⇒ ✓
- Union with allied groups ⇒ ✓

All of the above operations are in constant time thanks to our data structures, except the removal of a dead group. Indeed, remove a group will cost as many operations as they are stones in the group. But, in one game, we will never remove more stones than there are intersections. Thus we can consider that for each intersection we will play on it once, and remove a stone once. Leading to 2 operations per intersection and so to a amortized constant time per intersection.

Results

Result and experiments : Overall level

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

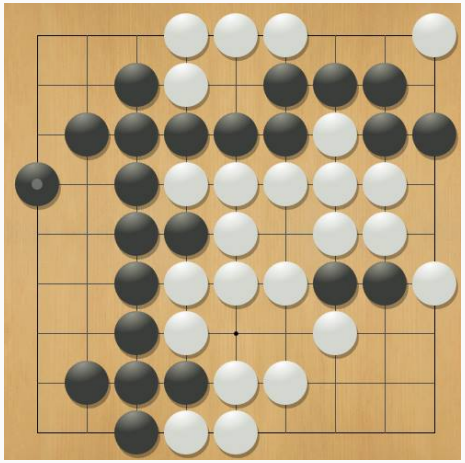
Percentage of wins on a 9*9

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

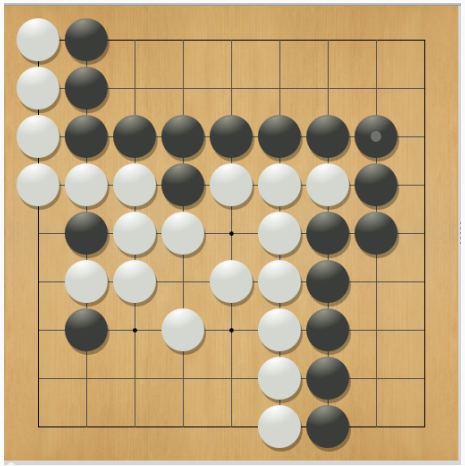
Percentage of wins on a 9*9

V-Run is the name of the bot of this master thesis. 5000 thousands iterations corresponded more or less to 30 seconds per move. 18 kyu is a good beginner, 15 kyu a low level amateur, 11 kyu a good amateur.

Result and experiments



(a) 5k iterations against a 18 kyu



(b) 10k iterations against a 15 kyu

Figure 15: V-Run(black) VS GoFree(white)

Result and experiments: Complexity $i * n^2$

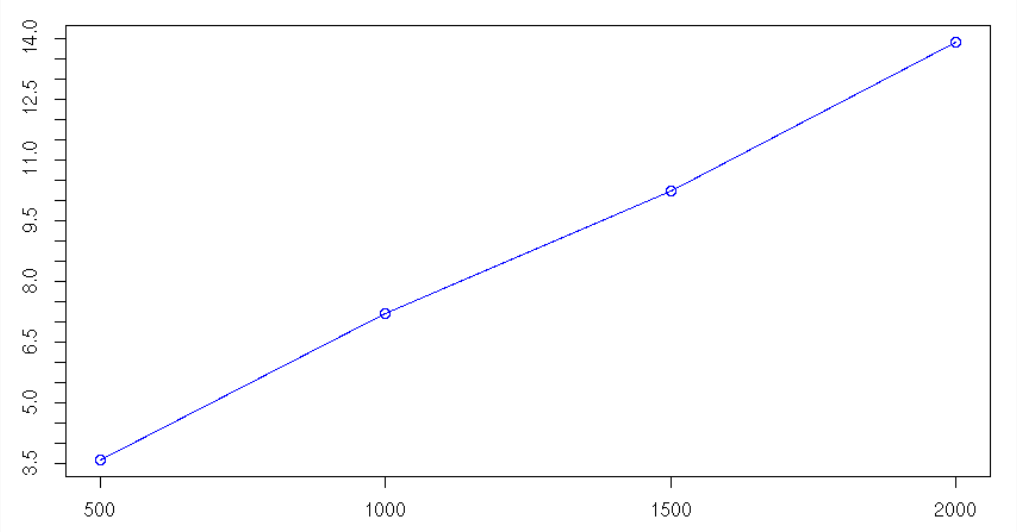


Figure 16: Time complexity test depending on i iterations

Result and experiments: Complexity $i * n^2$

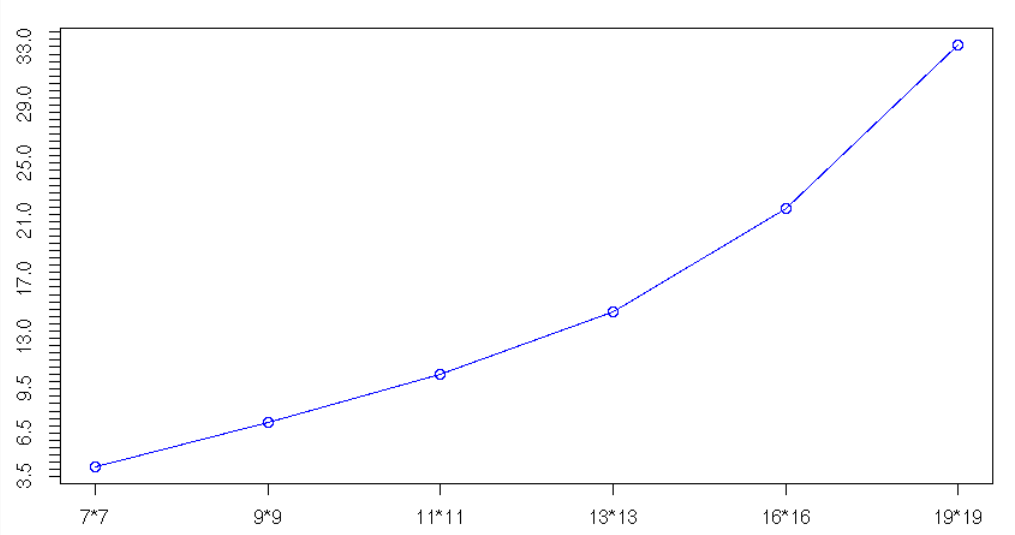


Figure 17: Time complexity test depending on n^2 the size of the board

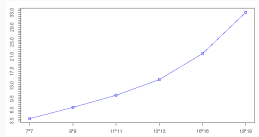
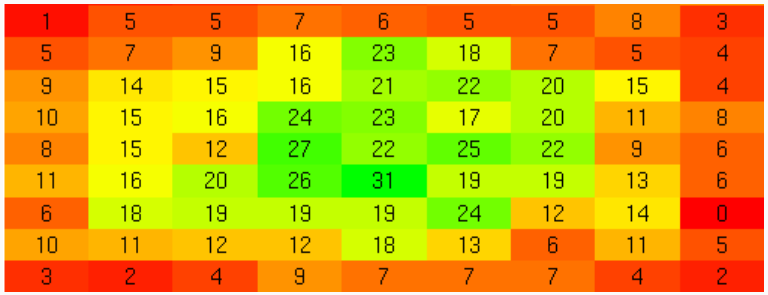


Figure 17: Time complexity test depending on n^2 the size of the board

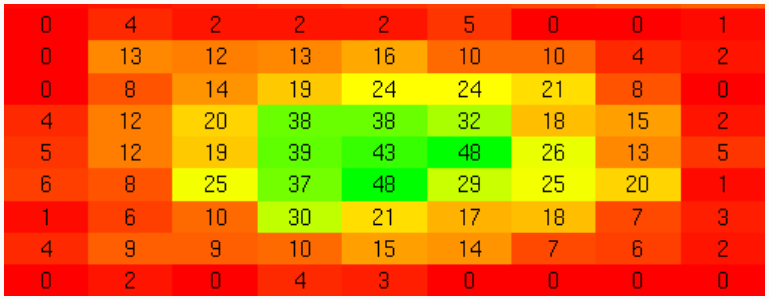
As the goal was to minimize complexity, we wanted to test the actual complexity to compare it to the theoretical one.

As all operations on the board are in constant time, one iteration will cost at worst n^2 operations. Because in the worst case, one iteration will have to play a complete game, fulfilling $n * n$ intersections, where n is the length of the board. Thus i iterations costs $i * n^2$. The time taken evolves linearly depending on i and in a quadratic manner depending on n, asserting the the theoretical complexity.

Result and experiments : Start of the game

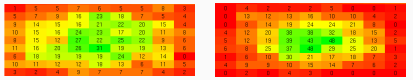


(a) 1000 iterations



(b) 5000 iterations

Figure 18: Heatmap of the move chosen an empty board

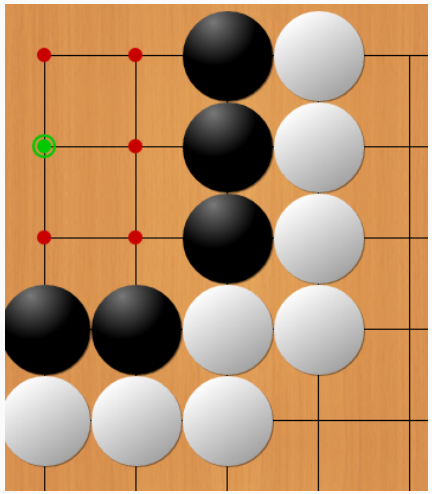


(a) 1000 iterations (b) 5000 iterations

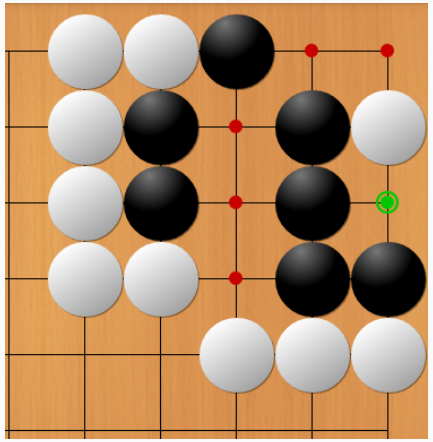
Figure 18: Heatmap of the move chosen an empty board

We tested the level on an empty board. The heatmaps show the results of asking one thousand times a first move. Each square is an intersection, and its number is the number of times MCTS picked this intersection out of 1000. We can observe that the more iterations MCTS searched, the more it played at the center. It is good, because on a $9 * 9$ board, plays at the center are stronger.

Result and experiments: Tsumego



(a) Easy tsumego



(b) Average tsumego

Result and experiments : Tsumego

	100	1000	10 000
Easy	44 %	64%	81%
Average	66%	69%	74%

	100	1000	10 000
Easy	44 %	64%	81%
Average	66%	69%	74%

We also tested V-Run on small problem of Go, where you have to find the critical move. It shows relatively good results.

Result and experiments: Hard Tsumego

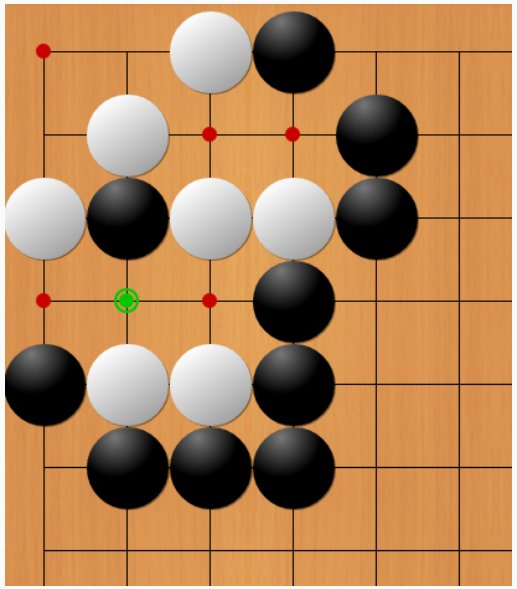


Figure 20: Hard tsumego - 7% of success

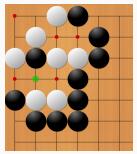
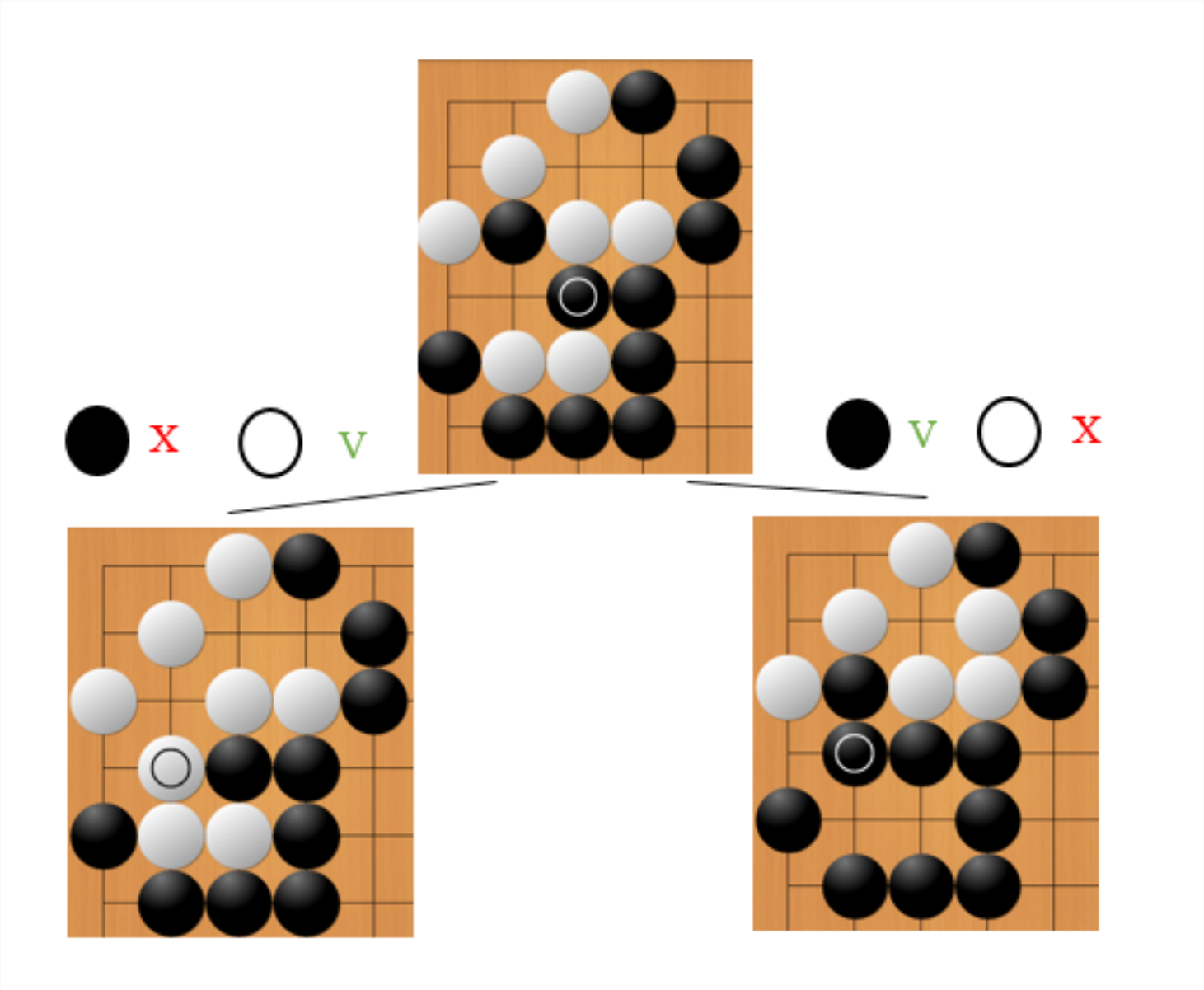
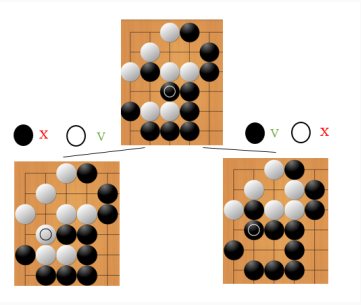


Figure 20: Hard tsumego - 7% of success

But on one tsumego, it had only a 7% success rate. In fact, V-Run returned another move 80% of times.

Result and experiments : Hard Tsumego





The move returned in 80% of the cases is showed on the top board(the black stone circled). It is a move that allows for an almost guaranteed victory if move answer badly (right bottom board), but is easily countered if white answered correctly (left bottom board)

Result and experiments : Greedy move problem

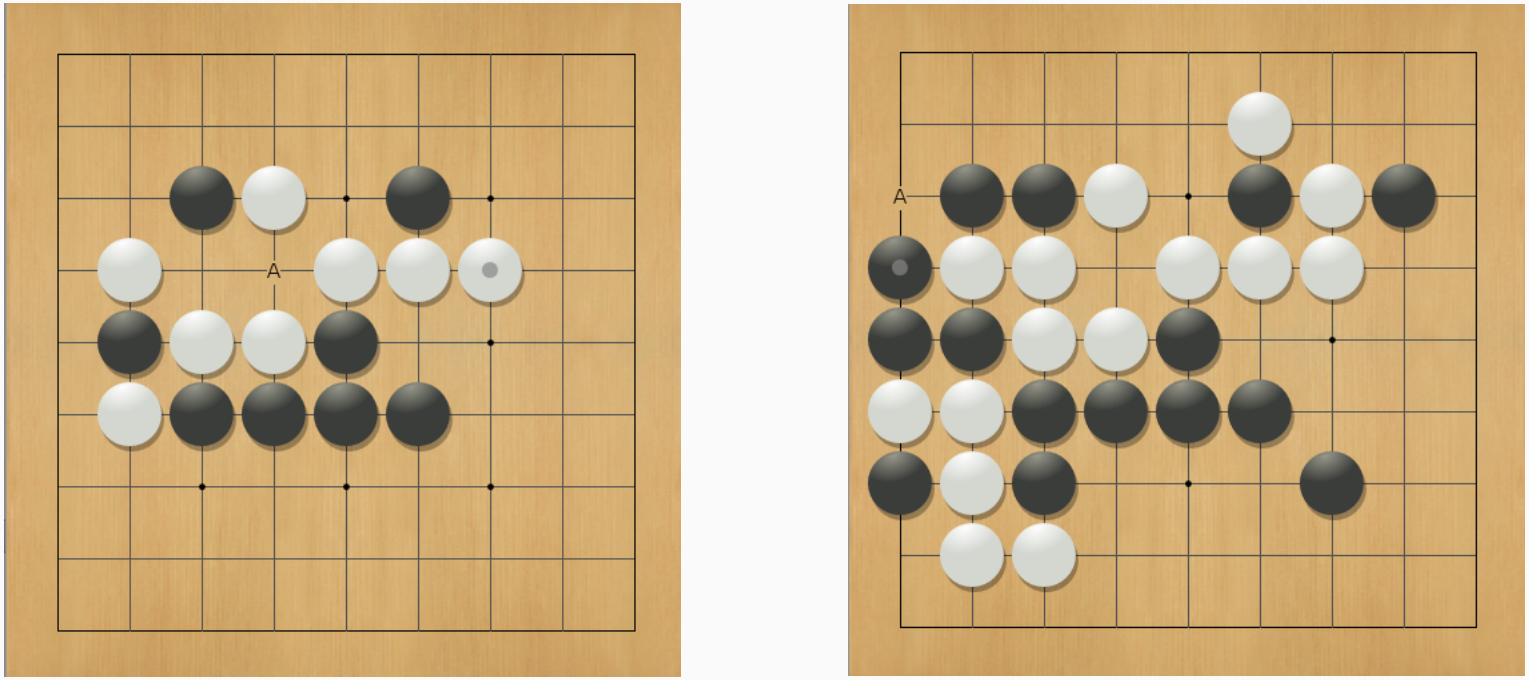


Figure 21: Greedy move picked

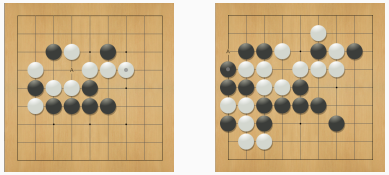


Figure 21: Greedy move picked

We realized that V-Run played regularly greedy moves, that could yield promising results but that are easily countered.

Result and experiments: Greedy move problem

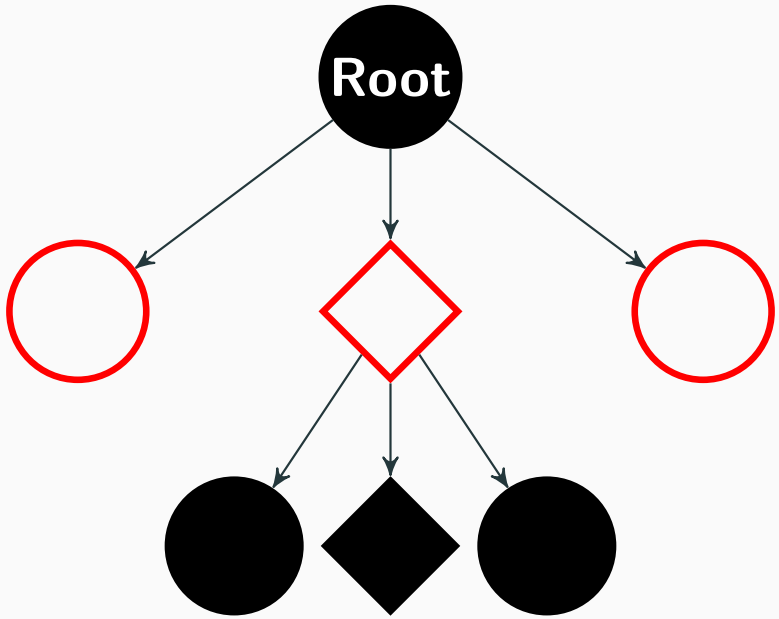


Figure 22: reaching a depth 2 with 3 possible moves

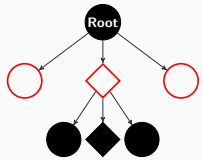


Figure 22: reaching a depth 2 with 3 possible moves

The potential reason is that in the core technique of MCTS, each time you reach a node in the selection step, you have to expand all of its possible children before going down further. Indeed, how could it decide to explore further a node when there is no knowledge on another children node ? Thus, going down cost a lot of iterations. Exploring each level cost at least five or six times the branching factor, as we need to have a minimal knowledge of each node before exploiting the promising ones. With few thousands iterations, V-Run can find a first good move, but won't have enough iterations to find the counter move at the second level of the tree. As it didn't find the counter, the greedy move is considered as strong.

Improvement

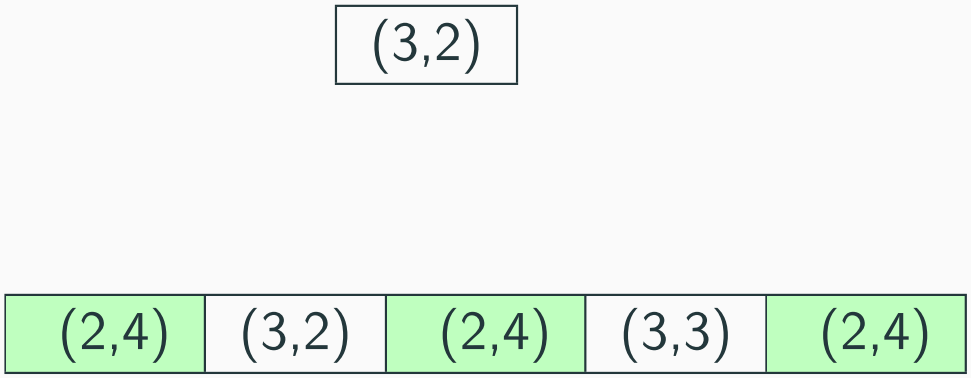
Improvement : Bradley & Terry

Solution to the greedy problem : Only expand t nodes based on their potential strength.

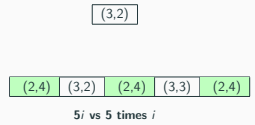
Solution to the greedy problem : Only expand t nodes based on their potential strength.

This heuristic estimates the strength of moves. It allows to expand only the best. If we expand less nodes at each level, we can have a more precise and deeper search.

Improvement : Majority voting



5i vs 5 times i



As we saw in the experiments, V-Run will generally find a good answer (80% of right move on tsumego, most of the plays in the center), but can still fail sometimes. Instead of doing one big search, why not divide the number of iterations by k, do k smaller search and take the most returned result.

Improvement : Majority voting

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

Table 1: Comparison of the results between different majority voting V-Run with 10 times k iterations and $10 * k$ total iterations standard V-Run

Improvement : Majority voting

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

Table 2: Matches between 10*500, 5*1000 and 5000 iterations-50 games

Improvement : Majority voting

Combine versions

- Standard if few iterations relatively to the number of moves possible
- Majority voting in the opposite case

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

Combine versions

- Standard if few iterations relatively to the number of moves possible
- Majority voting in the opposite case

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

Majority voting showed more efficient on tsumego, but performs worse when opposed to standard version on a complete game. The reason is maybe the ratio number of iterations on moves possible. If we have a lot of iterations for a small search space, it is probably better to divide the search to prevent a failed exploration. But in the opposite case, it is probably better to have one larger search, to allow for a more representative result.

Result and experiments : Binary victories

Ponderate the reward depending on the final board

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

Ponderate the reward depending on the final board

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

Another problem is that the reward at the end of the simulation step is binary : 0 for a loss, 1 for a win. One possible drawbacks is that in a losing situation, V-Run will begin to play at random, as every node wins attribute will be zero. To balance that reward, we suggest to weight it with the difference of score at the end of the simulation step. w_i is the initial reward (0 or 1), p_i is the score of the player i and k is a constant to optimize.

It should make V-Run promote more winning or less loosing moves. Even if theoretically, it shouldn't change the outcome of win with a perfect search, it could practically change the results. Indeed, by choosing better moves, V-Run would give himself more margin in case of a future bad play.

Github

Free access model on Github :

`https://github.com/Fioelkais/MCTS.git`

Github

Free access model on Github :

https://github.com/Fioelkais/MCTS.git

Everything (Code, master thesis, slides) is available on GitHub. As I didn't find any optimal public representation of Go, I believe this to be the first one available publicly.

For the bibliography, please look in the master thesis.

The following slides were included to answer to eventual questions. The first one tries to precise numerically the expansion of all node before going down further. The second where it is better to use MCTS. The third the technical details, and the last one is there to remember that all the operations cited are in constant time on a python list.

Thank you for your attention.

Questions?

Questions : Expand all

On 5000 iterations, with 81 moves possible, best move explored : 90 times

As +- 200 moves possible on average $\Rightarrow$ best counter not enough explored

Questions : MCTS

Advantages

- Aheuristic
- Non exhaustive search
- Any time

When to use:

- Board games : Hex, Havannah
- Real time video games : Fable legends, Total War :Rome
- Non deterministic games : Poker, Magic

Questions : Technical details

- Implemented in Python
- + 15 000 lines, - 9500 (misleading due to similar files)

Questions : Python list

Time complexity

- Get item $\Rightarrow$ ✓constant time
- Set item $\Rightarrow$ ✓constant time
- Pop last item $\Rightarrow$ ✓constant time
- Append one item at the end $\Rightarrow$ ✓constant time