Monte Carlo tree search applied to the game of Go

UCL - Ecole polytechnique de Louvain

Vandermoste Thibault

Day Month Year

Abstract

Dedication

Declaration

Acknowledgements

Contents

Chapter 1

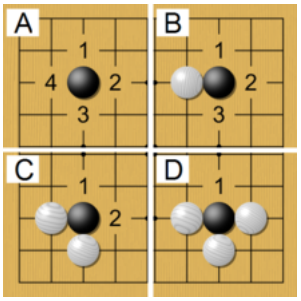
Introduction

1.1 start of intro

1.2 Rules of Go

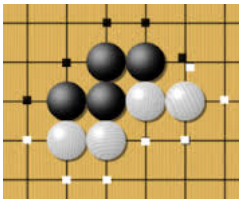
The rules of Go are extremely simple, and their simplicity allows a real complexity and depth of play. The game is played on go-ban, which are board of 9*9,13*13 or 19*19 intersections. There is two players, one who play the white stones, the other the black stones. Each player put a stone on an intersection on the go-ban alternatively.They are two important concept to understand, the liberties and the group.

A stone has as many liberties as free intersection next to it. In the following case, the black stone has initially four liberties, and it progressively decrease to one :



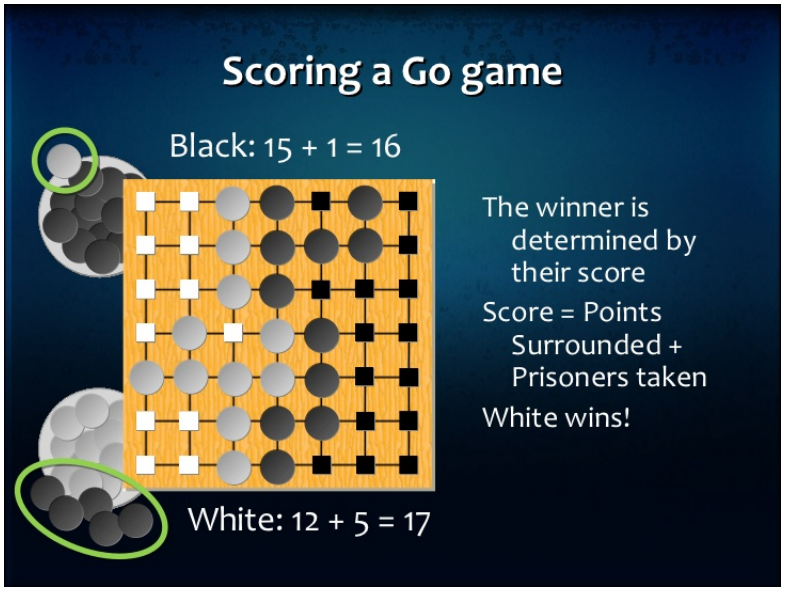
Two or more stones can fusion together and form a group who share the

liberties. Here we can see three group, one black and two white. The black one has five liberties by example.



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, understand has no more liberties, it's dead. All the stones of the group are taken as prisoners, leaving the intersections where they were free.

To win, you must have more point than your opponent. You gain one points by prisoner and one points by territory. A territory is a free intersection surrounded at least partially by your stones and potentially by the sides of the board.



Chapter 2

State of the art

In this chapter, we will talk about what is Monte Carlo tree search is and how it applied to the game of Go.

First we will talk about the Monte Carlo part and the tree search part separately, then explain how the fusion work.

The second section is about the benefits and drawbacks of MCTS

Thirdly, we will see the big dilemma of MCTS : Exploitation versus Exploration

And last but not least, there is a part where we see the application of MCTS to the game of go and what are the actual heuristics used today.

2.1 Monte Carlo 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 or science, Artificial intelligence, even economics.

There is four steps to a Monte Carlo algorithm :

- Define a domain of inputs
- Generate random value over the domain following a probability distribution
- Compute the result
- Use the result to get what you wanted

We'll show this four steps through an example : the computation of π. How to estimate π with a random generation.

- Define a domain of inputs : the square of coordinates (0,0),(1,0),(0,1) and (1,1).
- Generate random value over the domain following a probability distribution : Generate points randomly into this square.
- Compute the result : compute x where $x = \frac{In}{Total}$ and where $Total$ is the number of points generated and In is the number of points inside the circle of radius 1.
- Use the result to get what you wanted : With a sufficient high number of $Total$ we can approximate thanks to the law of big numbers and assume

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

As the area of the circle is $\pi * 1^2$, the quarter is $\frac{\pi}{4}$. And so we got our final equation :

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

Results have show that with enough tries, we can get really close to the real value of π.

2.1.2 Tree search

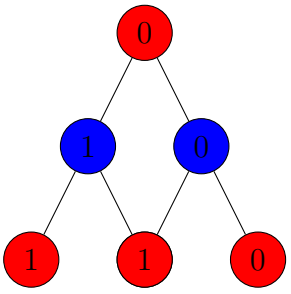
In this section we will explain the basics of tree search in artificial intelligence, some techniques already used and how they differ of MCTS.

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 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 0 heads for each player and the first to play is the blue player. Either the coin is head then the game move to the node with 1 head either it's tail then the state is the 0 blue node. Then the red player plays and so on.



Mini-max

The minimax algorithm is an algorithm in artificial intelligence games. It will try to win by giving to each state of the game a score. Following the name of the algorithm player one will try to maximize the score, 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 will choose the action that eventually bring him into the best state for him, assuming the other player plays the best moves he can. In the best cases, the algorithm

continue to search the tree until he finds a finished won situation but as it's not always possible to compute such depth, he limits himself to the best score.

```
function minimax(node, depth, maximizingPlayer)
   if depth = 0 or node is a terminal node
       return the heuristic value of node

   if maximizingPlayer
      bestValue := -minus infinity
   else
      bestValue := infinity

   for each child of node
       val := minimax(child, depth - 1, !maximizingPlayer)
       if maximizingPlayer
           bestValue := max(bestValue, val)
       else
           bestValue := min(bestValue, val)
   return bestValue
```

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. In the case of Go by example, the initial branching factor is 361. We easily understand that such a complexity wouldn't be applicable.

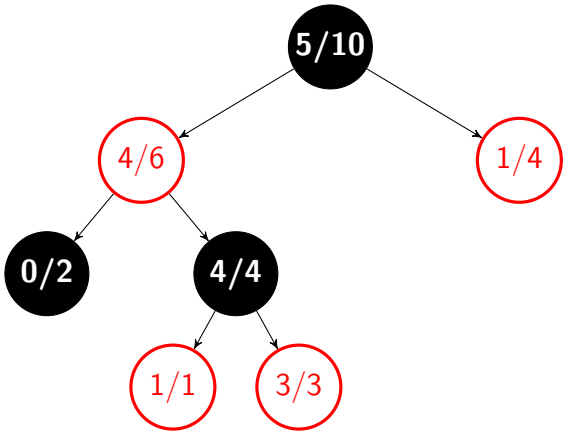
- The algorithm needs an evaluate function which will return a value, a score for a node. It needs a deep knowledge of the game to produce a correct and precise heuristic when it's possible which is not always the case. In the case of Go, it's practically impossible to always being able give a value 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 will now see how to combine these two principle to form the Monte Carlo tree search. The idea is relatively simple. As the total tree of the game is far to big to be computed, it will only compute the necessary tree. Which means computing the tree node by node, with information on what node build next. And in the end, there will be a far smaller tree indicating the best moves. In the following sections, the process of building the tree will be detailed.

Four Steps

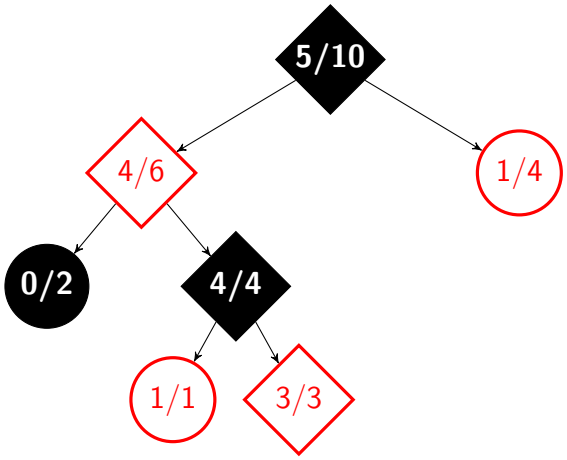
This section will present with the help of figure how one execution of the Monte Carlo tree search occur, through the four steps. Two names are given each time for the steps, it doesn't mean something different. It's just that theses steps are not called the same everywhere. The two names found were put to help a reader. We present an initial tree that already was partially built. 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. By example if you take right leaf node, we can see the red players won 0 times on 4 passages.



Selection or descent

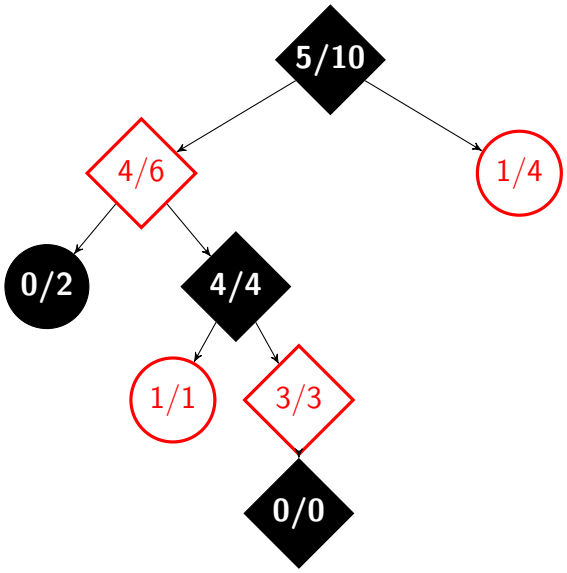
The first step is the selection, the algorithm will select node after node the best moves for him. The criteria of best moves can vary, and the different strategies will be explain further. For the moment, it simply selects the one

with the best ratio won/played. The moves chosen are highlighted by being in diamond shape.



Expansion or growth

Once the selection phase reach either a terminal state of the game or a leaf node of the tree built, we arrive at the expansion phase.If this is not a terminal state, the basic strategy from this point is to select a random move and add it to the tree with a 0/0 statistics.

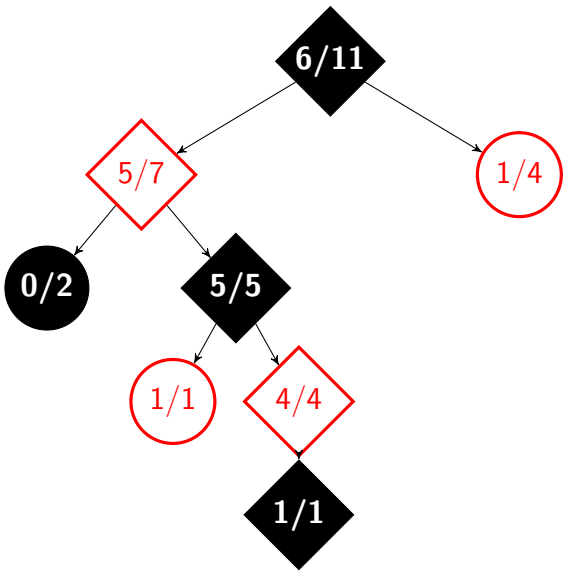


Simulation or roll-out

From this state of the game the simulation step plays random move one after another for each player until reaching a final state.

Backpropagation or update

Once a final state is reached, it computes who's the winner and then, node by node, update the value of the tree. In the example, the black player will win. It will then update all the nodes on the path from the leaf node to the root.



The execution is now finished. There is two possibilities, either there's still time, and the algorithm can process one more, or the algorithm must choose a move and will select by example the one with the best ratio, here the move on the left, 5/7. Indeed, by choosing that, he got 5 out of 7 chance to win, where in the other case, he would have loose for sure.

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 will be big enough to be statistically representative.

Now we will 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 notably 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 if there's not something better. When you exploit, you can't explore and the inverse is also true. 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 will enlighten some answer.

Selection - Objective Monte Carlo

The goal of this technique is to compute a fairness function for each move. It will attribute a probability to each possible move and will then select randomly one in the possible, following the probabilities found. The fairness function reflect the balance between the exploitation and exploration. It will be shown here.

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 the ensemble of 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

TODO

Simulation

TODO

Backpropagation

TODO

Selection of the final move

Todo

2.1.5 Benefits and Drawbacks

Benefits

Monte Carlo tree search draw 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 is 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 how to play 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.
- Non time limited. One of the beautiful part of the Monte Carlo tree search is that you don't need a specific amount of time. The output of the program is not binary depending of the time. Some algorithm will take a certain amount of seconds or minutes, then deliver an answer. Here, the algorithm can always, at any time deliver an answer. Of course, the more time is given, the better the answer. But it allows

stopping the search, either after a certain time or a number of iterations trough the four steps.

- The basic version of Monte Carlo tree search can be implemented really simply. It's his efficient adaptation to complex games that makes it difficult.

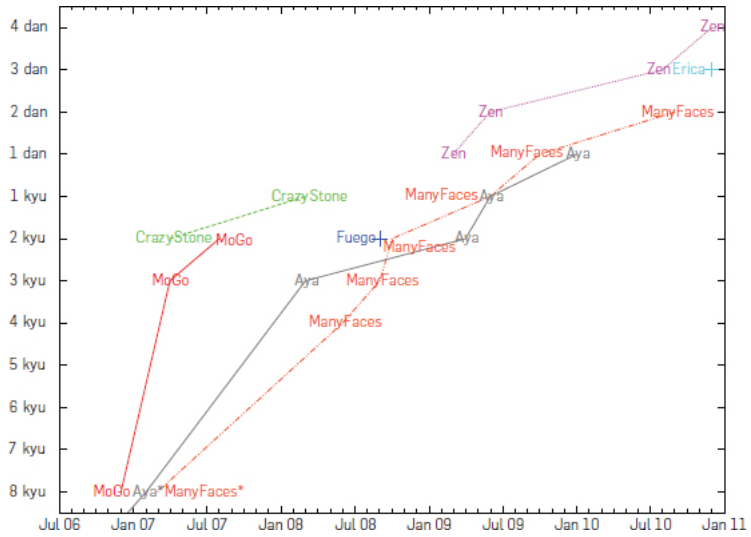
Drawbacks

Here's come the second part about the big dilemma talked sooner. There are several moments, as seen in the strategies, that give difficult choices. By example, in the simulation step, if it's too complicated, it will take 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 moreover, it can create the same problem than a complicated heuristic, as it do a lot of random bad plays, it won't represent a good player, even with a high number of play.

There are some heuristics that were thought for this, one of them, the MC-RAVE will be explained later for the Go.

2.2 Monte Carlo tree search applied to Go

Before the introduction of Monte Carlo, the field of artificial intelligence in go was kinda 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



2.2.1 Improvement

Here we will 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 consider 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 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 big and well known go program, where's n is the number of plays, and μ a parameter :

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

Chapter 3

Naive version

In this chapter, we will present the first naive version of a bot player with Monte Carlo tree search mechanism

This naive version was done in Python. We will present the base structure for a general Monte Carlo algorithm then the first really naive adaptation for Go. 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

We will finish by showing how bad of a complexity it is.

3.1 General Monte Carlo

3.1.1 Game State

The game state is what represent the state of the game at a moment. This is what will be 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

3.1.2 Node class

The node class represent a node into the Monte Carlo tree that will be built. Each nodes must have several attribute listed here :

- Move : The move that leads to the node
- The parent node
- The children node
- 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

3.1.3 Simple pseudo code

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

```
root = Node(r)
node= root
state = r

Selection Step
while node.children is not empty do        ▷ Here's a problem that i don't
understand, would like to talk to you about it.
    node= node.UCTSelect
    state.DoMove(node.parentmove)
```

```
end while

Expand step
if node.untriedmove is not empty then
    m = untriedmove.random
    state.doMove(m)
    node.addChild(m,state)
end if

Simulation step
while state.getMoves is not empty do
    state.doMove(random.state.getMoves)
end while

Backpropagation step
while Node is not None do
    Node.update(state.getResult)
    node= node.parent
end while
return best root.child
```

3.2 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 impact a move m on the board ? And who won in GetResult ?

Theses are three methods that need to be adapted.

We will 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 intersection in the following directions : up, down, left and right.

3.2.1 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. 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 rules was added. It says that you can't make a move that will make 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 will need 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.

3.2.2 DoMove

In Go

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

in programming

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

3.2.3 GetResult

In Go

When 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 count 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 will be 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.

3.3 Total complexity

We will 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 will 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 will have 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
- GW : complexity of getResult = n^2
- 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

We easily understand that is completely unthinkable, 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 4

Optimization

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

We will ultimately show the progress in complexity made

4.1 Union Find, or how to solve the group problems

4.1.1 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 roughly gives 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.

```
if x.parent==x then
    return x
else
    x.parent=Find(x.parent)
    return x.parent
end if
```

We can see that each time a find is done, it will balance the tree, to put all the nodes on the path closer to the root which will optimize 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 will make a smaller total depth.

```
xRoot=Find(x)
yRoot=Find(y)
if xRoot.rank > yRoot.rank: then
    yRoot.parent = xRoot
else if xRoot.rank < yRoot.rank: then
    xRoot.parent = yRoot
else if xRoot != yRoot: then
    yRoot.parent = xRoot
    xRoot.rank += 1
end if
```

4.1.2 How to use it in Monte-Carlo tree search

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.

This is the basic thinking, we can then have easily the group of stones and their representative. The GoState won't represent the board with a double array of integer but with a double array of Union-Find nodes. Each node will contain the basic attribute and his color, a set of his liberties and unfortunately his children because there's the problem of the deletion of a group.

This part is still in TODO following a find a way or not to suppress a group in O(1)

4.2 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 ko rules you can't have the same state of the board as just before. The only possibility for this situation to happen is to capture a solo stone that just capture a solo stone. So in this situation, we can simply remember the intersection and forbid to play on it the next turn.

This take back 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. So the first idea is to maintain a list of the possibles moves in the state, to transfer this list between the state and updating it each time a move was done. A second possibility is to not change the list at each move, but either when I get a random move from it, verifying its correctness, and if it's not, suppress it from the list.(**is it interesting to explain what you have to check when a move is done, or is it to technical ?**)

The problem is that inserting or removing an element of a list is in a complexity of the length of the list. So it wouldn't change anything. 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, we pop the end of the list and we update the value of the hashmap. It gives something like that:

last= List[len(List)-1] ▷ Remember the last value of the list
ToRemove = hashmap[x] ▷ remember the index of x
List[ToRemove]=last ▷ replace the value in the list to remove by the last value of the list
Hashmap[Last] = ToRemove ▷ updathe the index of the last value in the hashmap
List.pop ▷ Suppress the last value
Hashmap.delete(x) ▷ suppress in the Hashmap the value of x

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

Chapter 5

Conclusion

Appendix A

Appendix Title