Advanced Machine Learning Project

Monte Carlo Tree Search applied to the Game of Go

Madriss Seksaoui
Flavie Vampouille

MSc in Data Sciences & Business Analytics
CentraleSupélec & ESSEC Business School

Objectives of the project

The game of Go is one of the board games for which computer programs have long been unable to beat experienced human players. When play on the classical 19x19 board, the complexity of the game, in terms of legal positions is really huge, meanwhile with a restricted 9x9 grid, it is inferior to the complexity of the game of Chess. However, chess-programming techniques fail to produce a player stronger than experienced human. One of the reasons is that tree search cannot be stopped easily at a quiet position, as it is done in chess. Even when no capture is available, most of the positions in the game of Go are very dynamic.

Monte Carlo tree Search (MCTS) is one of the more recent and successful development in reinforcement learning with tree policy. It is a method to find optimal solution in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems. The MCTS is the first method to be competitive against very good players on a restricted board size for the game of Go.

There are several facts that make it hard to implement a good AI player for the game of Go: games are long, have a large branching factor (at each move a large number of actions are available) and it is difficult to evaluate the quality of a non-terminal node.

The objective of the project is to implement an agent that learns to play the Game of Go using the MCTS method combine with a UCB selection.

Due to the huge size of the state space with a 19x19 board, we will restrain our project to a 9x9 board size.

First task. Implementation of a Random Agent

Using the OpenAI Gym environment we have all the positions of the board detailed in 3 Boolean matrices which contains respectively the positions of the Whites, the positions of the Blacks and the empty positions. The action space is comprised of 83 total actions which correspond to the positions of the board (9x9), the possibility to abandon and the possibility to pass the turn.

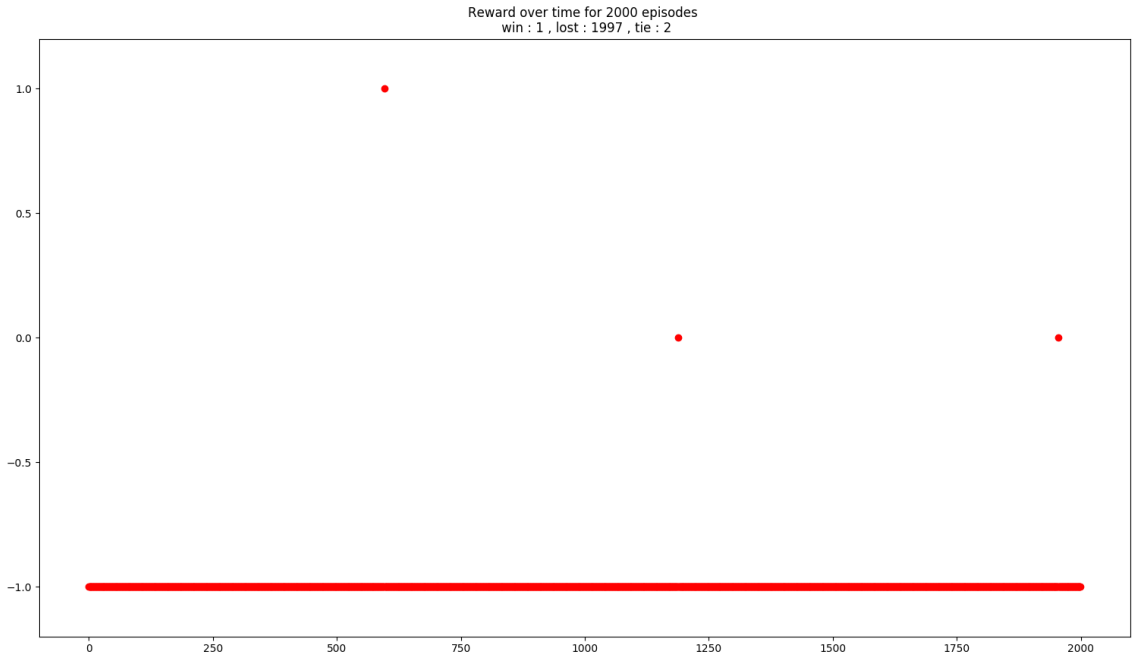
The game is said terminal if there is a winner, if it ends on a tie or if a player try to do an illegal move and hence lost the game.

The reward is set to zero when the game begins and change only once, when a terminal state is reached. It then takes value

- -1 if the agent loses the game or tries to play an illegal move
- +1 if the agent wins the game
- 0 if the game ends in a tie.

The first agent we create (***RandomAgent.py***) can choose randomly any move possible in the game (on the 81 positions, abandon or pass). However, this agent was really bad, even when playing against the random agent of OpenAI Gym:

Pure Random Agent vs. OpenAI Gym Random Agent

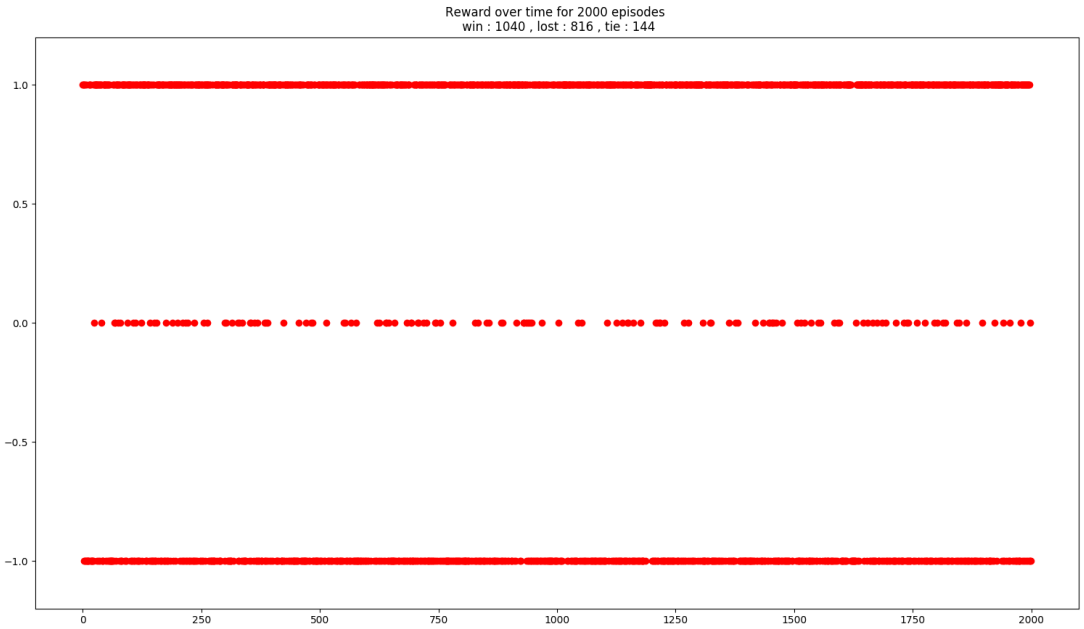


We can see that this first agent nearly always loses the game. It appears that as this agent is purely random, he very quickly (within a few steps) tries to play a forbidden action and thus loses the game.

Hence the first task was to create an intelligent random agent, who knows the rules of the game, and who, when playing against another random agent, win or lost with the same probability. Our second

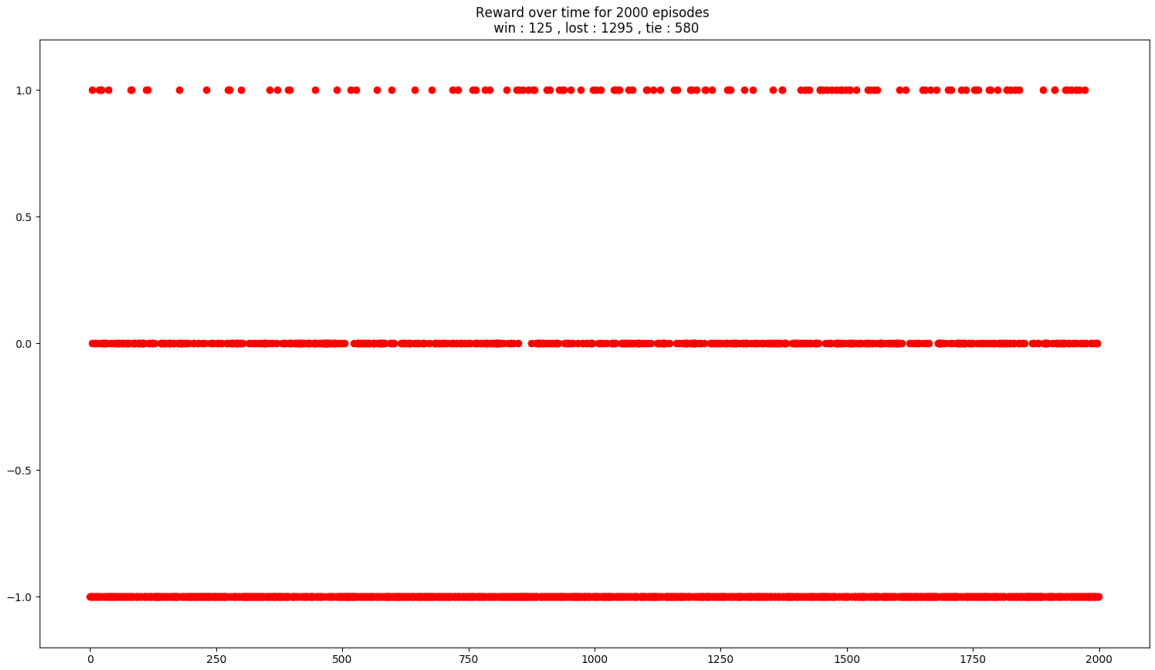
agent (***WiseRandomAgent.py***) interact more accurately with the OpenAI Gym environment and choose randomly is next action on the panel of allowed moves. His score looks more like what was expected in a random versus random game:

Wise Random Agent vs. OpenAI Gym Random Agent



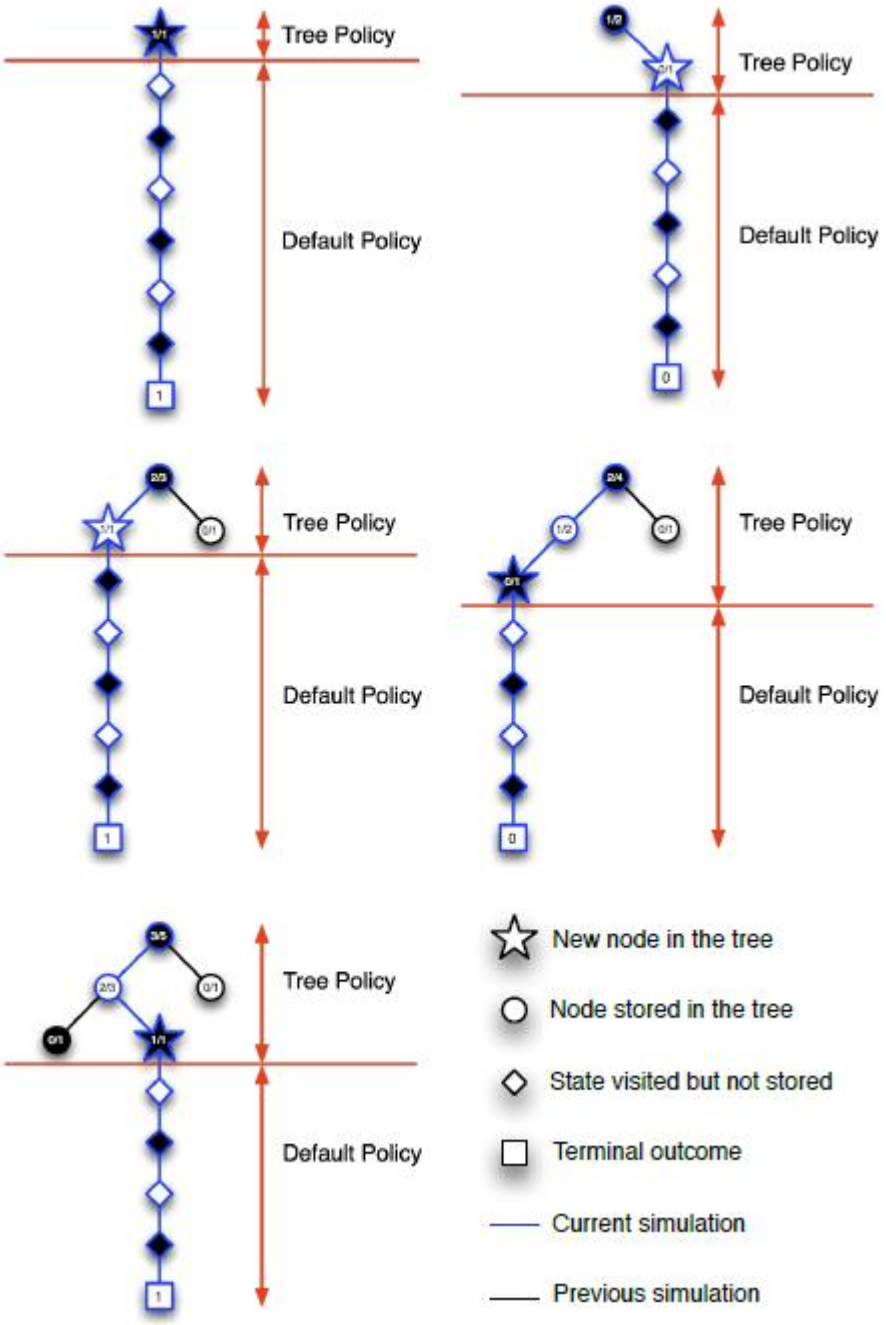
However, when playing against the UCT agent of OpenAI Gym, our agent is, as could be expected, awfully bad:

Wise Random Agent vs. OpenAI Gym UCT Agent



Second Task. Monte Carlo Tree Search Algorithm Implementation

The basic MCTS concept is that a tree is built in an incremental way using a tree policy which uses both exploration (look at the area that have not been well sampled yet) and exploitation (look in area which appear to be promising). A simulation is then run from the selected node and the search tree updated according to the result (a child node is added to the selected node and all the statistics of its ancestor are updated). The simulation is run according to a default policy (the wise random agent in our case).



Sutton & Barto, An Introduction to Reinforcement Learning
(Monte Carlo Tree Search on a problem with binary returns)

Tree Policy : Selection / Expansion

Starting from the root note (the empty board), a child UCB selection is recursively applied to descend through the tree until a not already explore node is reached.

An attribute is created on the nodes to store the information of knowing if a node already exists in the search tree or not: a node is said non-terminal if it has been explored at least once, and terminal if it is not yet in the tree.

At each step of this recursion, we consider the current node:

- While the node is non-terminal and has been visited more than twice, all its children already exist and one of them is added to the tree using the UCB selection policy;
- If the node is non-terminal and has been visited only once (or never in the case of the root node for the first iteration), all of its children are created with a default terminal state and one of them is selected randomly, as they all share the same initialization reward of zero.
- This node is then terminal, the Selection / Expansion stops here, a simulation is run from this node and its state is updated to non-terminal.

During the expansion process, the nodes are created and stored for both players alternatively.

One particular case can occur: if the last node selected is already non-terminal but no more action is available (in other words if the game ended here). We then return the current node itself in the tree policy.

Default Policy : Simulation

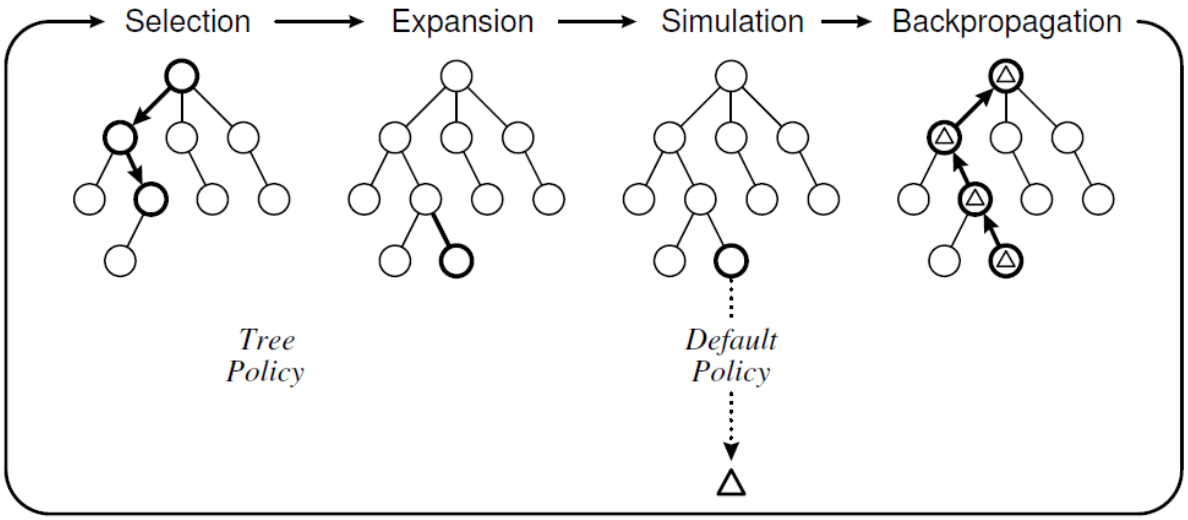
A simulation is run from the selected node of the previous step: starting from this node, a game is played using our wise random agent to obtain a potential reward.

If the selected node is fully expanded (it means if the game terminates here) we just return the reward. If not, to run the simulation we first check if it is our turn to play or the turn of OpenAI Gym. If it is OpenAI Gym turn we make it take a random action chosen randomly on the panel of the available legal actions before playing the simulated game.

Back-propagation

The result of the simulated game (that we will call delta and which is the value of the reward for the played game) is backpropagated through the selected nodes to update their statistics.

Each parent of the node from which starts the simulation is incremented of +1 in the number of its visits and of +delta in the value of its reward.



A Survey of Monte Carlo Tree Search Methods

Details on the UCB Policy

To select iteratively the nodes already creating in the partial tree search during the tree policy step, several ways are possible (max child, robust child, secure child…). Here we choose to iterate the choice of the child node as a multi-armed bandit problem, as seen in the previous class works.

When a node is created it is initialize with a reward value of zero and a number of visits of 10^{-100} (the real value would be zero but as python doesn't support the zero division we set it near to zero only, when the number of visits at the node is incremented by +1, this value would be neglectable).

A child node *j* is selected to maximise

$$UCT = \frac{cumulative\ rewards\ on\ the\ child\ j}{number\ of\ visits\ of\ the\ child\ j} + c * \sqrt{\frac{2\log\left(number\ of\ visits\ of\ the\ parent\ node\right)}{number\ of\ visits\ of\ the\ child\ j}}$$

If more than one child achieved a maximum value, one them is selected randomly.

The

$$\frac{cumulative\ rewards\ on\ the\ child\ j}{number\ of\ visits\ of\ the\ child\ j}$$

term holds information on the theoretic value of the child node and promotes the *exploitation* of nodes with superior results.

while the

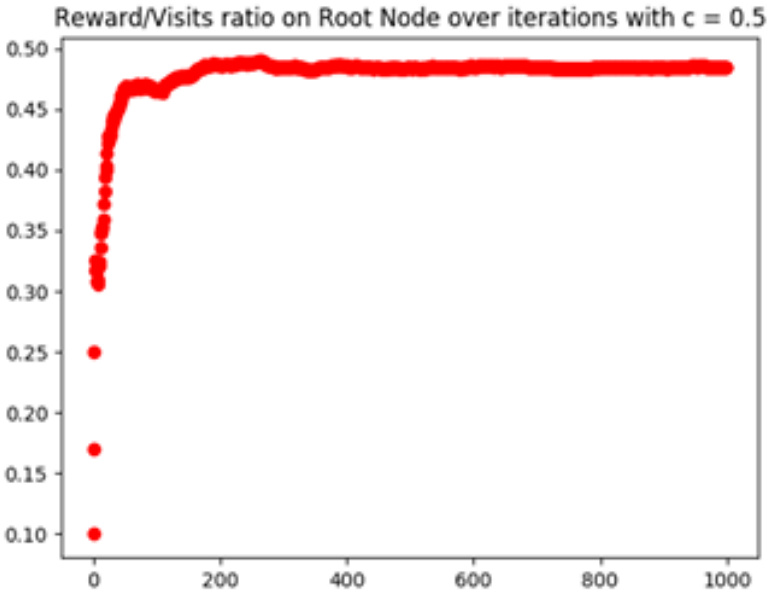
$$c * \sqrt{\frac{2\log\left(number\ of\ visits\ of\ the\ parent\ node\right)}{number\ of\ visits\ of\ the\ child\ j}}$$

term holds information on the number of times the node has already been visited, with respect to the number of times its parent has been visited. It promotes the *exploration* of new nodes or nodes that has been explored only few times. The initialization of the number of visits near zero makes sense here

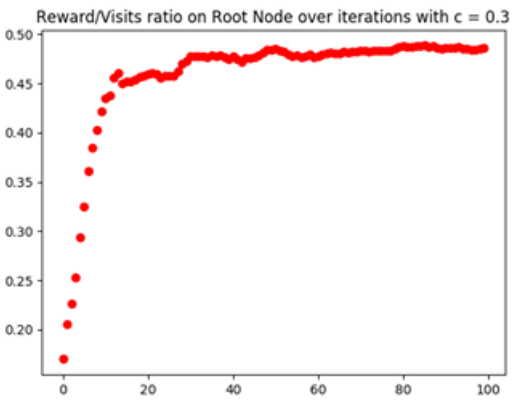
as we want all the nodes to be explore at least once, and the UCT will have a value near infinity for nodes never tried.

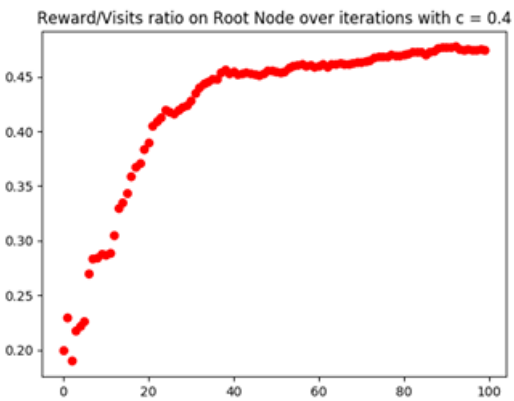
There is a smart balance between the exploitation and exploration terms the UCT formula: when a child is visited the denominator of the exploration term increases and thus the exploration term is lowered and the other children are privileged. But when a new child is explored, the numerator of the exploration term increases and the child has more chance to be played again. Hence, all the children are guaranteed to be choose if the time of test is long enough. The constant *c* change the balance between the exploitation and exploration: large *c* will increase exploration while small *c* will increase exploitation. After exploring some literature, we choose to set the value at c = 0.5. Choosing a better value will be possible in running a lot of tests to optimize it.

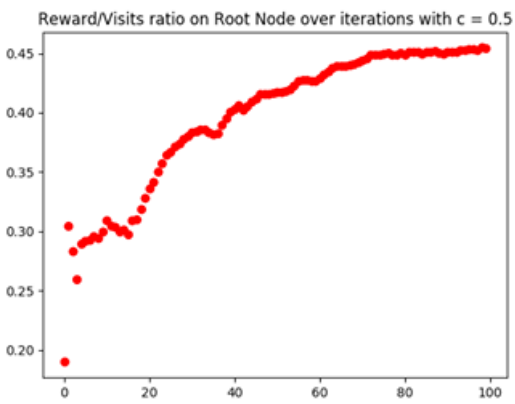
Such a policy will ensure that all nodes are tested, even the ones which doesn't obtain a good reward at a first stage. To obtained an efficient search tree, it is however necessary to train it a long time. Plotting the reward/visits ration of the Root Node over times shows that the tree allows preference to the best explored nodes as this value increases.

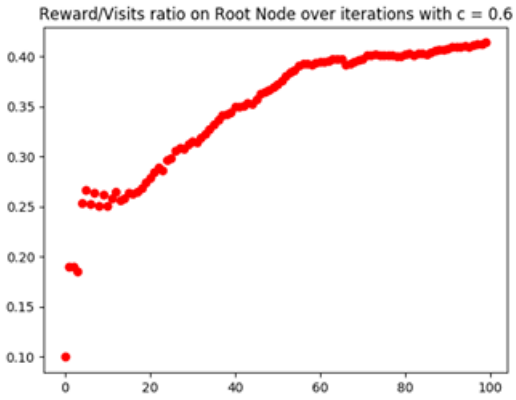


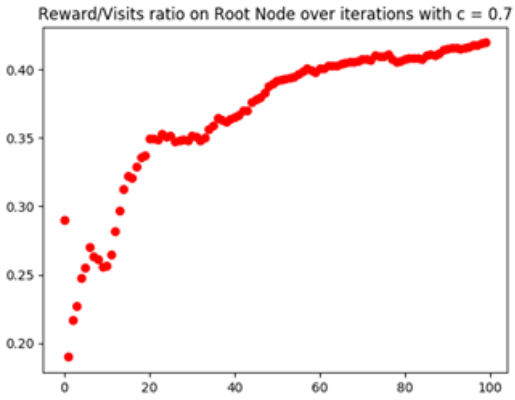
We run a few test on this Reward/Visits value to see the impact of the constant c.











These graphs are not unique, they change slightly at each new run of the UCT_Tree. However we can see that with small c the tree quickly test the best nodes while with the bigger values of c, the exploration is more frequent. This confirm that our UCB policy works fine. . Note the x axis doesn't reflect the total number of iterations, we reported the reward/visits ratio for every 100 iterations.

Third Task. Play Against OpenAI Gym Environment

Once the Monte Carlo Tree Search is done, the game tree hence created is stored in a UCT_Tree variable. The objective is to use it to play against the OpenAI Gym random agent. We hope to obtain a result in reverse of what happens when our own random agent plays against the OpenAI Gym UCT agent.

At first step, started from the root node we chose the best of our child nodes and the environment reacted in playing a new move. We then pruned the top of the UCT_Tree to be placed in the current node and once again choose the best move available. Doing so iteratively we play 2 000 games (like with the random agent) and stored the rewards of these games to plot a win / lost graph.

MCTS Agent vs. OpenAI Gym Random Agent

…

Unfortunately, our implementation for playing against OpenAI Gym crashed at some point.

References

A Survey of Monte Carlo Tree Search Methods

IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1, MARCH 2012

Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

Rémi Coulom, Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search

5th International Conference on Computer and Games, May 2006, Turin, Italy. 2006. <inria-00116992>

Paolo Ciancarini and H. Jaap van den Herik

Modification of UCT with Patterns in Monte-Carlo Go

[Research Report] RR-6062, INRIA. 2006. <inria-00117266v3>

Sylvain Gelly, Yizao Wang, R_emi Munos, Olivier Teytaud

Reinforcement Learning: An Introduction

MIT press (draft)

Richard S. Sutton and Andrew G. Barto