Reinforcement Learning

Ahmed BEJAOUI - Aymen MEJRI - Mohamed Rostom GHARBI

1- Project introduction :

This project is realized by: Ahmed BEJAOUI, Aymen MEJRI and Mohamed Rostom GHARBI. We are based mainly on the article **"From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning (2014) Rémi Munos"**.

The latter proposes a theoretical study of the "Monte-Carlo Tree Search" method using the UCT algorithm (Upper Confidence Bounds applied to Trees). This algorithm allows an efficient tree search by performing a bandit algorithm to each node of the search tree and use an optimistic search strategy that explores in priority the most promising path of the tree.

In this project, we propose an implementation of the "Connect4" game using the UCT algorithm. It's a two-player connection game. So in our case, you are going to play against the computer. The goal of the game is to align as quick as possible a series of 4 successive pawns of the same shapes on a grid with 7 columns and 6 rows. Each player has at most 21 pawns and alternately takes turns placing a pawn in the column of their choice.

2- Theoretical side of the project :

The application of Monte-Carlo methods to reflection games involves many random simulations. The exploitation of these simulations is now done by the UCT algorithm which appeared in the context of bandits problems where the only way to get information about a state is to make random selections.

UCT is an adaptation of the Upper Confidence Bound (UCB) algorithm to tree problems; board games is concrete example; in this kind of game, each of the possible positions is considered as an arm, the different evaluations made from the position are the rewards of the controller.

However it is possible to take into account the tree structure of these games, so that there is an advantage for the parent node, thanks to the different simulations made on the child nodes. This illustrates how the UCT algorithm works. The formula that gives the UCT value of a node, as a function of the mean μ of the scores obtained and the number of simulations that are passed through this node and the number of simulations s performed in the node's father is given by :

$$\mu + C\sqrt{\frac{\log n}{s}}$$

such that C is UCT constant set by the user (we chose the value of C equal to $\sqrt{2}$ for our project).

UCT develops at the beginning of each random simulation the gameplays that lead to the position whose UCT value is maximum. The constant C allows to adjust the degree of exploration of the algorithm. The higher the constant, the more the algorithm will tend to explore the low-rated moves.

The pseudocode for one iteration of the UCT algorithm for a game tree is given below:

Algorithm 1 One iteration of the UCT algorithm

```
1: if root = NULL then
2:     root ← MAKENODE                          ▷ Initializes the root of the tree
3: end if

4: traverser ← root
5: while not ISLEAF(traverser) do                ▷ ISLEAF(node) tests if
                                                   the given node is a leaf
6:     expandLeaf ← TRUE
7:     for i = 1 to number[traverser.children] do
8:         if traverser.children[i].counter = 0 then   ▷ Will be incremented in
                                                         UPDATEVALUES(...)
9:             values[i] ← ∞                    ▷ a vector of reward values for all players
10:            expandLeaf ← FALSE
11:        else
12:            values[i] ← traverser.children[i].values + C*sqrt(log(traverser.counter) / traverser.children[i].counter)
13:        end if
14:    end for
15:    traverser ← traverser.children[arg max values]
16: end while

17: if expandLeaf and not ISTERMINAL(traverser) then
                                 ▷ ISTERMINAL(node) tests if the given node
                                   corresponds to a terminal state
18:    EXPANDNODE(traverser)     ▷ Adds all children of the node and sets their
                                   counters to zero, but does not expand them
19:    traverser ← RANDOMCHILD(traverser)
20: end if

21: if not ISTERMINAL(traverser) then
22:    outcome ← DOMONTECARLOSIMULATION(traverser)
                                 ▷ Does a Monte Carlo simulation (with random
                                   action choice) until a terminal node is reached
23: else
24:    traverser.values ← GAMEVALUE
                                 ▷ GAMEVALUE is a vector of rewards for all players,
                                   computed from the value of the completed game
25:    outcome ← traverser.values
26: end if

27: UPDATEVALUES(traverser, outcome)
                 ▷ Updates the accumulated rewards values and increments the
                   number of visits counters of the nodes along the path to the root
```

To summarize, the MCTS algorithm consists of the four following phases:

1- Selection : the algorithm starts with a root node and selects a child node such that it picks the node with maximum win rate The idea is to keep selecting optimal child nodes until we reach the leaf node of the tree. To select such a child node, we use UCT formula.

2- Expansion : When it can no longer apply UCT to find the successor node, it expands the game tree by appending all possible states from the leaf node.

3- Simulation : the algorithm arbitrarily selects a child node and randomly simulates a game from the selected node to the final state.

4- Backpropagation : Once the algorithm reaches the end of the game, it evaluates the state to figure out which player has won. It traverses upwards to the root and increments visit score for all visited nodes. It also updates win score for each node if the player for that position has won the playout.

3- Practical side of the project :

During this part of the report, we will talk about the technical side of the project, and we will have a deeper look into the different functions used. The project is divided into 4 parts : Main program, GamePlay, Games and AI. We will talk more about these 4 parts in details.

3.1 Main.py :

This is the main entry point into the game framework. There are different functions that were used in this file, such as, *load_module_from_path(path)* which takes a python module of the form path/to/module and return the actual module after importing it.

To run the main program of the game, we need to put this command in our terminal after acceding to the game folder '**python3 main.py games/thegame.py ai/uct.py**'.

3.2 Games Folder :

In this folder, there are 4 different python files that are important to define the game we are playing. These different files are named as follow : board.py, gamestate.py, metadata.py, thegame.py.

The first file named board.py is the most important one, because it actually defines what the user sees during the game. The file gamestate.py is actually a class, that is used to hold the intermediate data during the game, this class also holds one of the most important functions of this project which is the reward function. The file metadata.py is also a class to hold the metadata of the game (Metadata is the data that provides information about other data) such as each player's symbol or who's going to start first. And finally, the last file is

thegame.py, which is the referee of the game, it organizes the turns between the players, and tells when the game is over for example.

3.3 Gameplay folder :

This folder contains only one file named gameplay.py. It is actually the main entry to the game. It takes into account both the game module and the ai module in order to run everything together.

3.4 AI Folder :

This folder is what is the project about. In this folder, we find two files that are so important to run the computer side of the game. The files that were used here are node.py and uct.py. We will talk more about both these files in details in the following section.

3.4.1 Details about uct.py

We find here the code for the UCT (Upper Confidence Bounds for Trees) algorithm which is the main topic of our project. This file contains many functions to define this algorithm, in the following section, we will name these functions and explain their roles.

Function	Role
get_best_move(cur_state, reward_function)	With this function, we will try to get the best move that the computer have to play.
_uct_search(game_state, reward_function)	We do this in order to get the action that leads to the node child with the best reward.
_search_helper(root, reward_function)	This function returns the best child within the computational budget we set (equal to 2 seconds in our project).
_child_is_not_most_visited(child, root)	This function return True if the child is not the most visited node, else it returns False.
_back_up(v, delta)	With this function we get, Delta which is the value of the terminal node that we reached through the node V.
_best_child(v)	With this function, we compute the UCT values of the different nodes, and we return the node with the maximum value.
_choose_untried_action_from(available_actions, already_chosen_actions)	This function helps us to choose an un-tried action, rather than a random or uniform one.

_default_policy(game_state, reward_function)	When ever we can not compute the optimal move, we need a default policy. This actually a random choice to follow.
_delta_function(delta, v)	Denotes the component of the reward vector delta associated with the current player p at node v.
_expand(v)	This function helps us expand the MC Tree with all possible actions that we can do.
_tree_policy(v)	This is a complementary function to the above one. Here we check if the node we're in is terminal or not, if not, keep expanding.
_within_computational_budget(start)	With this function, we can control the time in which our algorithm converges. The more we add more time to the algorithm, the harder it will be to beat it.

3.4.2 Details about node.py :

This file is actually a class where we will define different game states as nodes . In the next section, we will have a deeper look into the different attributes and functions of this class.

The attributes of this class are : Name, State, Parent, Children (A list), total_reward, num_times_visited, already_tried_actions.

The different functions are in the following table. We will also talk about their different roles :

Function	Role
available_actions(self)	This function helps us to get the possible moves given a Node.
derive_child(self, action)	This function helps us to get a new Node from a current Node given the action we made.
move_that_derived_this_node(self)	With this function, we are able to get the action from a Node in the MCT. This is an important function since this gives us the optimal play.
is_non_terminal(self)	This function returns True, if the node we are in is not terminal. A terminal node means that its a node where the game is over.

is_not_fully_expanded(self)	Returns True unless all possible children have been added to this Node's childrens.

3.5 View on the game play :

During this section of the report, we will dive into the game play of our game. We will also put some screenshots to explain more about the game.

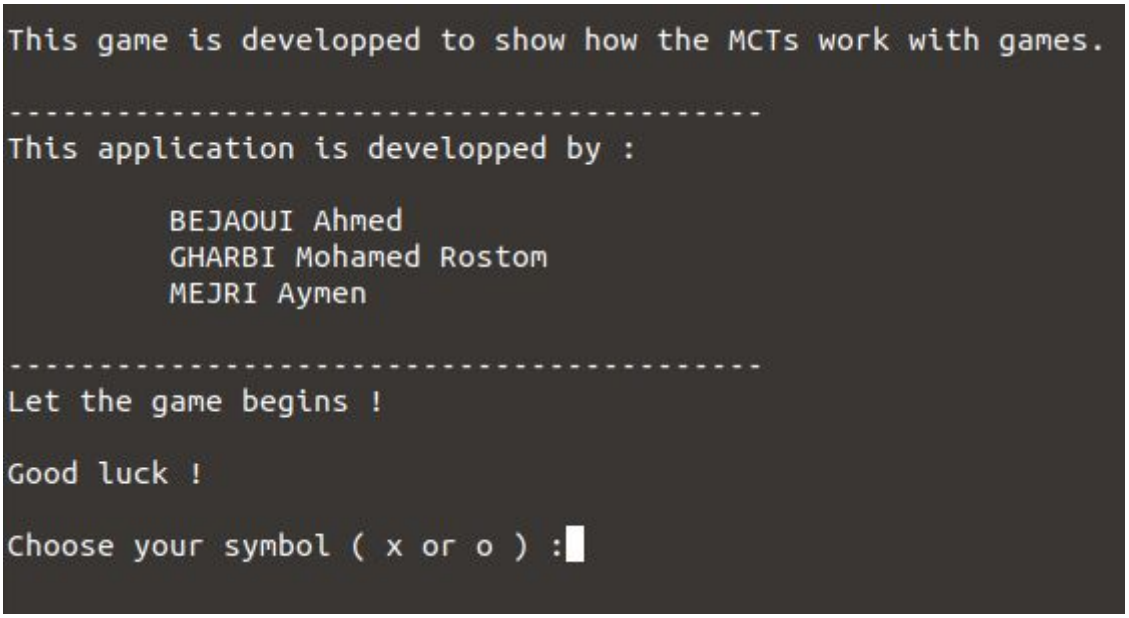
3.5.1 Running the game :

To run the game, we need to use this command in our terminal :
python3 main.py games/thegame.py ai/uct.py.

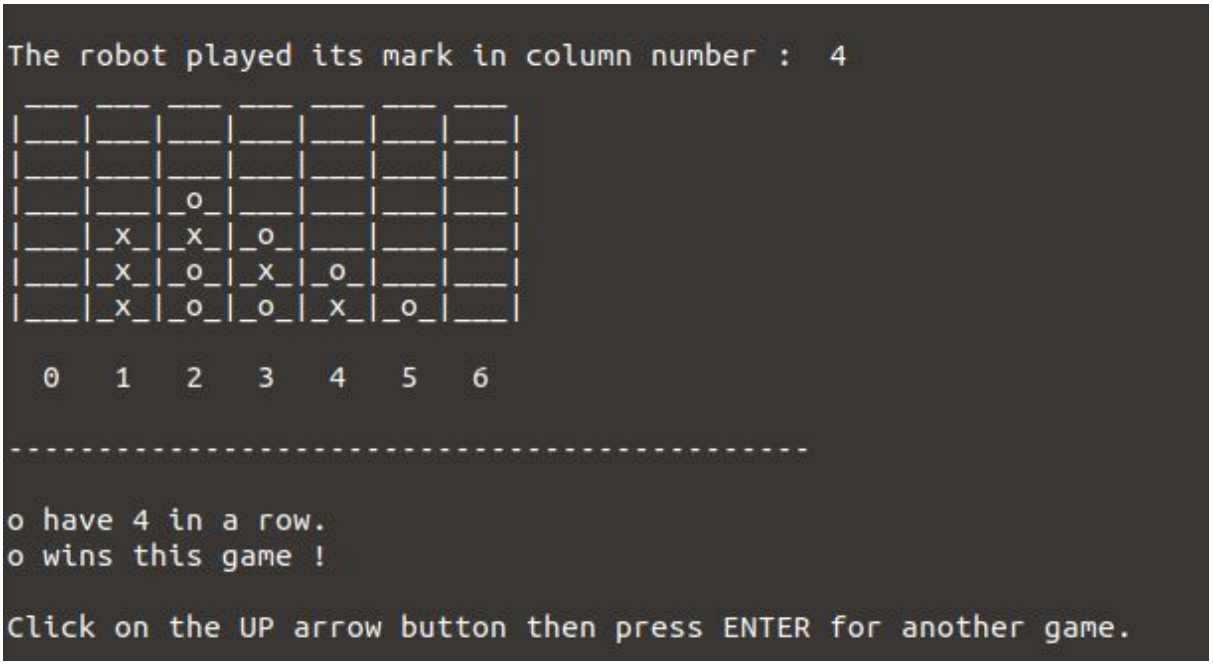
3.5.2 Different states during the game :

At first, when we run the game, the user needs to choose it's symbol, it whether an 'x' or an 'o'.

This is the first view of the game :



Then, the user has to choose a column between 0 and 6, in which, he will put it's symbol. The goal of the game is the have 4 connected symbols, whether horizontally, vertically, or diagonally.



As we can see on the picture above, the game has ended when the robot had 4 connected symbols diagonally (‘o’ symbols).

4- Advantages and Drawbacks :

The main advantages of the monte Carlo method are:

- It does not require any pre-knowledge or predefined strategy about the game.
- Determine the most important node which are potentially capable of winning the game.
- Suitable for problems with high branching factor as it only focuses on the most promising branches.

The major drawback of the Monte Carlo method is that it may fail to suggest reasonable moves. It may occur if nodes are not visited adequately which could result in inaccurate estimates.

5- Conclusion :

At first, it was difficult to say, or to trust that an algorithm that relies mainly on random choices can lead to a very smart Artificial Intelligence.

However, thoughtful implementation of MCTS can indeed provide us a solution which can be used in many games as well as in decision-making problems. It is also used when

playing a so-called perfect information game. In short, perfect information games are games in which, at any point in time, each player has perfect information about all event actions that have previously taken place.

6- References and bibliography:

(2009, 01). Utilisation de la recherche arborescente Monte-Carlo au Hex. Récupérée 01, 2019, à partir de : https://pdfs.semanticscholar.org/086c/63b8d7a87bfa6e222631a1f70d89950a3176.pdf

Sievers, S. (2012, 04). Implementation of the UCT Algorithm for Doppelkopf. Récupérée 01, 2019, à partir de https://ai.dmi.unibas.ch/papers/sievers-master-12.pdf