# DPRL Assignment 3: MCTS Tic-Tac-Toe

Group 13: Geoffrey van Driessel , 2639310, Erkin Yildirim, 2600101

13 December 2020

## 1 Introduction

During this assignment we will focus on the creation and implementation of a Monte Carlo Tree Search (MCTS) algorithm that can reliably beat a random agent during a game of tic-tac-toe, using Python programming language. We aim to analyze the convergence process which should provide meaningful information about the search algorithm in beating a random agent. For exploration, we have implemented UCB (Upper Confidence Bound) algorithm using multiple values for the confidence value to see how exploration ultimately influences the accuracy. The program considers a full game of tictactoe (starting at turn 0).

## 2 Method

Our MCTS implementation consist of three main parts. The first parts creates a tree object where each node in the tree represents a possible state of the tic-tac-toe board. And where the childs of this node are all the remaining actions for that state. For instance, the root tree object represents the state [], thus all possible actions are still open and therefore its childs are 1 until 9. Then for node 1 its childs are from 2 until 9, etc,.

Then the second part is the training. Initially the selection of actions will be random, but after some games UCB will start to decide whether to exploit or to explore, more on this later. After a game is finished, it returns the state and its reward. Where the reward is -1 for a loss, 0 for a tie and 1 for a win.

The third part is the updating of the tree. This simply takes the end state and the reward and traverses the tree using the state. Then for each traversed node it increments the total amount of tries by 1 and the the total amount of rewards by the reward.

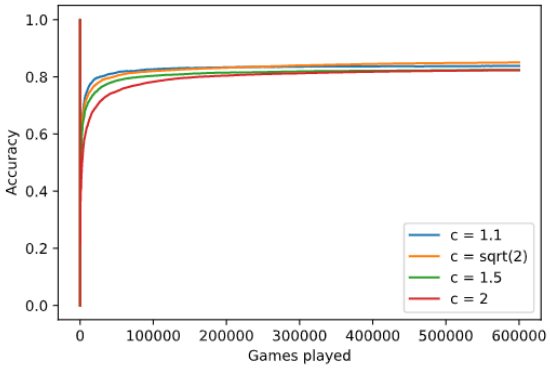
As you can see after playing the game a number of times and updating the tree, UCB starts to exploit an action based on the confidence value. In the next section we discuss the results in regards to the convergence per confidence value.

The evaluation metric that we use is the accuracy, which is defined by the total amount of wins divided by the total amount of games played.

## 3 Interpretation of results

After running the total amount of iterations, we can comment on the probability of picking a certain action starting from the root of the tree. We observe that one action is heavily exploited when compared to the others. We tested this for one certain confidence value ($\sqrt{2}$) and it shows that in this case, the first action (the upper left cell of the tic-tac-toe board) is heavily favoured over other actions (481.842 successes out of 599.869 tries). This means that all other actions were tried only a total of 600.000 - 599.869 = 131 times. This is almost a negligible difference. We can interpret this as the algorithm experiencing no problems with picking one starting action and dedicating to it, as multiple runs have showed that this heavily exploited action can be a different cell on the board as well.

As we can see from figure **a**, there is a large spike at the very first or first few games played. This can be explained by the fact that there is a small probability that the algorithm randomly wins the first few games and interprets the win accuracy as 100%. Quickly after that, it becomes more clear that the amount of played games show that the accuracy is in fact much lower and starts climbing as the games progress.




(a) Accuracy convergence over the amount of games played

As a higher confidence value tends to lean towards a lower mean total reward, we chose to plot iterations with different confidence values to visualize the effects. In figure **a** we can see that for each confidence value, convergence starts at a different point in time (games played). Even though all 4 plots converge around the same accuracy, there is still some difference in performance. We can see that, in the end, a confidence value of $\sqrt{2}$ achieves the highest accuracy, with a confidence value of 2 achieving the lowest accuracy. A noteworthy observation is that a confidence value of 1.1 converges faster than any other confidence value, but is eventually beaten by a confidence value of $\sqrt{2}$, but still performs very well despite the low exploration value. This could be attributed to the fact that tic-tac-toe is not a very high dimensional problem, which may explain why exploration isn't as favoured as in highly dimensional problems.

We found the eventual best starting actions for each confidence value (1.1, $\sqrt{2}$, 1.5, 2) to be 3, 1, 3, 3 respectively. Interpretation is that order does not matter in tic-tac-toe, so for instance if you take state: [1,6,3], then this is the same as state [3,6,1]. So the explanation of the different optimal action per confidence value is that they simply found a different optimal solution first and kept exploiting that.

# 4 Discussion

Because our current tree has many duplicate states (e.g. state [3,5,6,8] is the same as [6,8,3,5]), these duplicate states eat away a lot of memory and time, as they require computations. We can reduce the tree size by acknowledging these duplicate states to reduce the depth and breadth of the tree. We propose that the tree size can be significantly reduced by implementing an algorithm that resembles "alpha-beta pruning" like in minimax algorithms using a tree structure. The way this would be implemented is by scanning the tree for states that occupy a similar position in the game and pruning them from there, as they would have the exact same child node possibilities. Then if you would have to traverse the tree for a certain state, you can simply order the actions in the state per player. Using pruning, you would not only preserve memory, but also time. This would present a good topic for future research or implementation in similar code to ours.