

**Name:** Sarthak Dalmia Artificial Intelligence (CS F407)

**ID:** 2018B3A70290G BITS, Pilani, Goa Campus

**Assignment No.** 2 December 2, 2021

Training Q-Learning using Monte Carlo Tree Search to play Connect-4

Table of Contents

[1. Abstract 2](#_Toc89308251)

[2. Overview 2](#_Toc89308252)

[2.1 Problem Specification 2](#_Toc89308253)

[2.2 Some Basic Representations 2](#_Toc89308254)

[3. Monte Carlo Tree Search 3](#_Toc89308255)

[3.1 Implementation 3](#_Toc89308256)

[3.2 Parameters and Evaluation 3](#_Toc89308257)

[3.3 Final MCTS Parameters 4](#_Toc89308258)

[4. Q-Learning 4](#_Toc89308259)

[4.1 Implementation 4](#_Toc89308260)

[4.1.1 Vanilla Q-Learning Value Updation 4](#_Toc89308261)

[4.1.2 Afterstates Q-Learning Value Updation 4](#_Toc89308262)

[4.2 Parameters and Evaluation 5](#_Toc89308263)

[4.3 Modified Q-Learning Algorithm 5](#_Toc89308264)

# Abstract

This report focuses on implementing both, the Monte Carlo Tree Search and Q-learning algorithms to play the game of Connect 4. We run MCTS algorithm for 40 and 200 simulations and show that MC200 wins most of the time against MC40. Lastly, we train Q-learning algorithm against our MCn algorithm, for n ranging from 0 to 25, for a simplified Connect 4 game of size 5 columns x r rows using afterstates.

# Overview

The report is organized as follows. Section 1 contains the problem specification and implementation of the MCn and Q-Learning algorithms. Section 2 contains the detailed implementation of the Monte Carlo algorithm for a game of 5 columns x 6 rows for 200 and 40 simulations. Section 3 contains the implementation of Q-learning in way that expedites its convergence against MCn for n between 0 and 25. Lastly, Section 4 contains the modified Q-Learning algorithm to win against MCn for a range of n.

## Problem Specification

In this assignment, Monte Carlo tree search and Q-Learning is to be implemented to play the game of Connect 4. For Monte Carlo, the board size is 5 columns x 6 rows and number of simulations are 40 and 200. For Q-Learning, Monte Carlo must be used to train the algorithm and should win against the later.

## Some Basic Representations

Each state of the game (or board) is represented such that first player is “1” and second player is a “2” in the grid. Now we concatenate all the elements in each row to make one string, which is a integer. This is then saved to a file using the pickle function which stores objects directly as their binary representation rather than converting them into strings. This is saves memory and reduces file size.

# Monte Carlo Tree Search

## 3.1 Implementation

A tree of depth 4 is created via simulations at each time step including the first one. This entails a possibility of having a child-node already existing in the tree (with some value) before we begin its simulations. As usual we return the move to take, as the move with maximum number of playouts. Each node’s value is calculated via UCT function with rewards rather than wins as a parameter. If multiple nodes have the same value, then we choose a move (or node) randomly. In terms of expansion policy, we can either choose to create all the immediate children or only one child. Here, all the children are created. This is simply because it makes it easy to code as we now don’t have to check each time whether we have all the children from a particular node and or not and if not do we select from the existing child node or expand via some other policy.

## 3.2 Parameters and Evaluation

The algorithm with 200 simulations benefits first and foremost with the **number of simulations.** More number of simulations helps in converging into the best action at each step more accurately. With 40 simulations, for an epsilon, there is a chance that the exploration part of the algorithm heavily effects our final choice, which may not then be the best choice. The **terminal rewards** granted were changed to make the game finish in fewer moves and to increase win percentage of the MC200 algorithm. These were changed for both the 40 simulations and 200 simulations algorithm and calculated over a range of **UCT constant** values for 100 iterations each.

The above graph shows the average win loss and draw percentage over different reward maps. It was found that with changing of the UCT constant value there is no drastic change in percentage value and thus the default of 1.4 is taken for further runs of the algorithm. After choosing the UCT value as 1.4 we tabulate the win, draw, loss percentage and number of moves for a few of the

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rewards | | | Statistics for MC200 | | | |
| Loss | **Draw** | **Average No. of Moves** | | **% Losses** | **% Draws** | **% Wins** |
| -1 | **0** | 22.03 | | 14.75 | 13.04 | 72.21 |
| -0.5 | **-0.5** | 21.21 | | 19.21 | 0.77 | 80.02 |
| 0 | **0** | 19.58 | | 19.21 | 0.84 | 79.95 |
| -1 | **-1** | 20.71 | | 17.76 | 1.53 | 80.71 |
| -1.5 | **-1.5** | 20.63 | | 19.38 | 1.21 | 79.41 |

reward maps tried.

It was found that the reward structure of -1, 0, 1 for Loss, Draw and Win respectively results in a minimum percentage of loss for the 200 simulations algorithm. Moreover, for the reward structure -1, -1, 1 for loss, draw and win respectively it seen that the win percentage is maximum and the number of moves required are fairly close to the minimum found. Also, in the later case the draw percentage was considerably lower than that of former case.

## 3.3 **Final MCTS Parameters**

Finally, the reward map of **-1, -1, 1 for Draw, Loss and Win** respectively is chosen. An **UCT constant value of 1.4** is chosen.

# Q-Learning

## 4.1 Implementation

Both the vanilla Q-Learning with training state-action pairs is trained and an Afterstates algorithm of Q -Learning is trained. These two algorithms are compared on the bases of there win percentage against MCTS algorithm. For both the algorithms epsilon greedy policy is used for next state selection. That is after finding the action for maximum state- action pair value for vanilla Q-Learning and maximum afterstates value for afterstates Q-Learning, we choose that action with an epsilon greedy policy.

### 4.1.1 Vanilla Q-Learning Value Updation

### 4.1.2 Afterstates Q-Learning Value Updation

## 4.2 Parameters and Evaluation

**Afterstates** were useful because firstly they save memory, it saves training time because it condenses multiple state-action pairs which lead to same single value which will themselves be updated several times. Also, from the graphs shown below we see that wins of afterstates algorithm is higher than that of vanilla algorithm. Next, different **reward maps**, **gamma(γ)** and **epsilon(ε)** values are tried. **Learning rate (α)** was fixed at 0.1 for each of these trials. Parameters chosen for the MCTS are that from the previous part, that is, reward map of -1, -1, 1 for loss, draw, and win and UCT constant of 1.4. From this we take out the best possible gamma, epsilon and reward map for Q-learning based on the time it takes to converge faster. One liberty taken to reduce the number of iterations is that we train our Q-learning against MC20 instead of MC25.

Following are some of the graphs found during the above evaluation.

## 4.3 Modified Q-Learning Algorithm

For choosing the best possible value for gamma and epsilon, the number of wins Q-learning secures against Monte Carlo tree search for 0 to 25 simulations was considered. Adding to the previous criterion, training time and number of moves to win is also considered. Q-Learning is pitted against MC for 0 to 25 with an interval of 5. It is found out that the less the number of simulations for the MCTS, the higher is the win percentage of Q-Learning is, which was as expected. Moreover, running the algorithms for a greater number of iterations increases the win percentage suggesting that convergence was not yet achieved.