Artificial Intelligence

Assignment 2

Kushal Joseph Vallamkatt

2019A7PS0135G

The Game: Connect-4

Connect-4 is a two-player connection board game, in which the players choose a color and then take turns dropping colored discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs. Connect Four is a solved game. The first player can always win by playing the right moves.

Introduction

In this assignment I used 2 algorithms: **Monte Carlo Tree Search (MCTS)** and **Q-Learning** to create a program that plays a smaller version of Connect-4. Although the algorithms and their implementations are straightforward, it is essential to find the optimal tuning of their parameters for a game-playing program to perform better, i.e, have a greater number of wins. In order to do so, I have tried various different parameter values for testing the performance of the program and by observing a trend in the resultant plots, I have chosen the values for the same.

Game, State and Action Representation

Game: In code, I have represented a game of Connect-4 as a series of States where there is a State-Transition when a player makes a move (drops a coin). Since Connect-4 is traditionally a 2-player game, I have used the symbol “1” to represent the player who plays first and “2” for the other.

State: The state is represented as a 2-dimensional array (or Python List) where the first index represents the current state’s row and the second, the column. The array is filled with “1”s, “2”s indicating that the location is occupied by a coin from the respective player, and “0”s indicating that the location is empty

Action: At each stage, a player has at most 5 possible actions, since the number of columns is fixed for all experiments in this report. The actions can hence be appropriately represented as the integers 0-4, where an action “2” would mean dropping a coin in the 2nd column (0-indexed). Sometimes, if a particular column is already full, the player cannot make a move there.

Monte Carlo Tree Search

The Monte Carlo Tree Search Algorithm uses a **sequence of “rollouts”** or “simulations”, in its essence, to try to find the best move given a particular state of the game.

In the first part of this report, I investigate the games played between 2 versions of MCTS agents: One that uses 40 rollouts before deciding its move, and another that uses 200. Some of the essential points with regards to this match-up are:

- ★ It seems fairly obvious that MC200 would consistently beat MC40, however **this was not always the case**.
- ★ The more times MC200 defeats MC40, we can say that our algorithm and its parameters are indeed better, because if MC40 matches MC200, it shows that our algorithm is highly random and is not really “learning” the optimal moves for a given game-state.
- ★ So, I tried to tune the parameters and introduce new parameters to try and **improve the ratio of wins: MC200’s wins/MC40’s wins.**
- ★ I concluded that the better the above ratio is for a given set of parameters, the better the MCTS algorithm.

Mentioned in the next subsections are some of the parameters I tuned, their optimal values and my inferences on why certain values worked better than others.

c, the UCT equation Parameter

Like many standard implementations of the Monte Carlo Tree Search, I have used the **Upper-Confidence bound (UCT)** formula for selecting a child node of a given parent. The formula is:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

The main idea behind selecting a node on the basis of its UCB value comes from the notion of uncertainty in a particular node's value when it has not been visited enough. As we visit a node more number of times, we move closer to its actual value, however, if we have not visited a node enough number of times, and hence are not aware about its true value, and may hence make an incorrect decision (by exploiting our already gained knowledge about the search space).

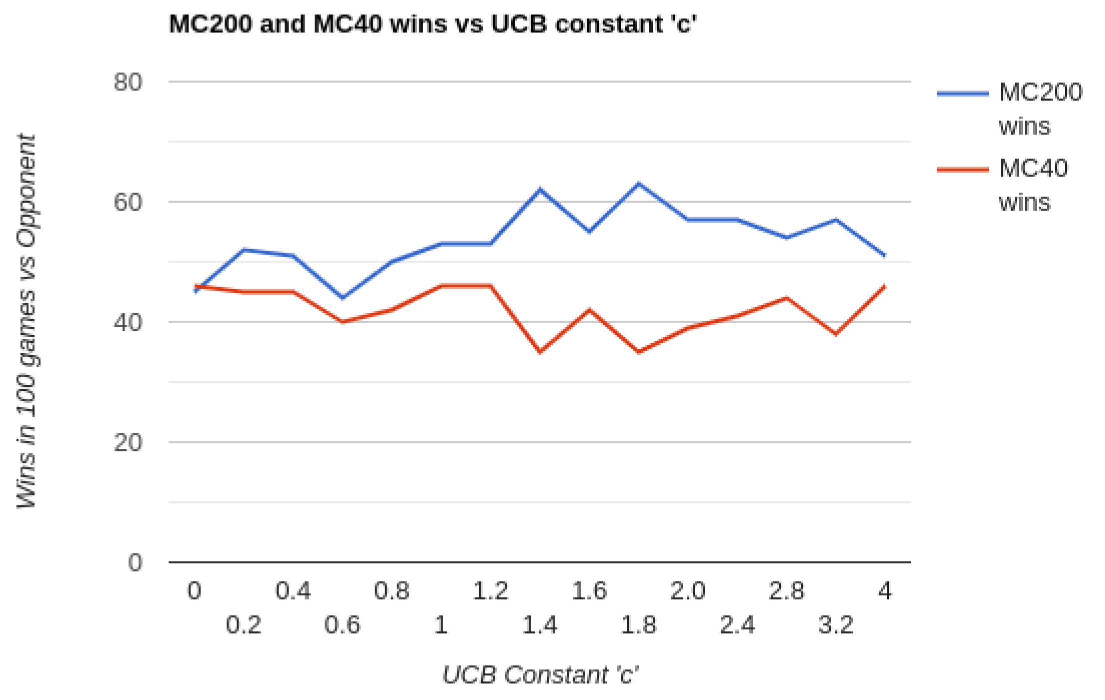
Hence, by adding a factor in the term to account for how many times a particular node has been visited before, we are essentially balancing our exploration and exploitation. The square-root term is weighted by the parameter, 'c', hence we can say:

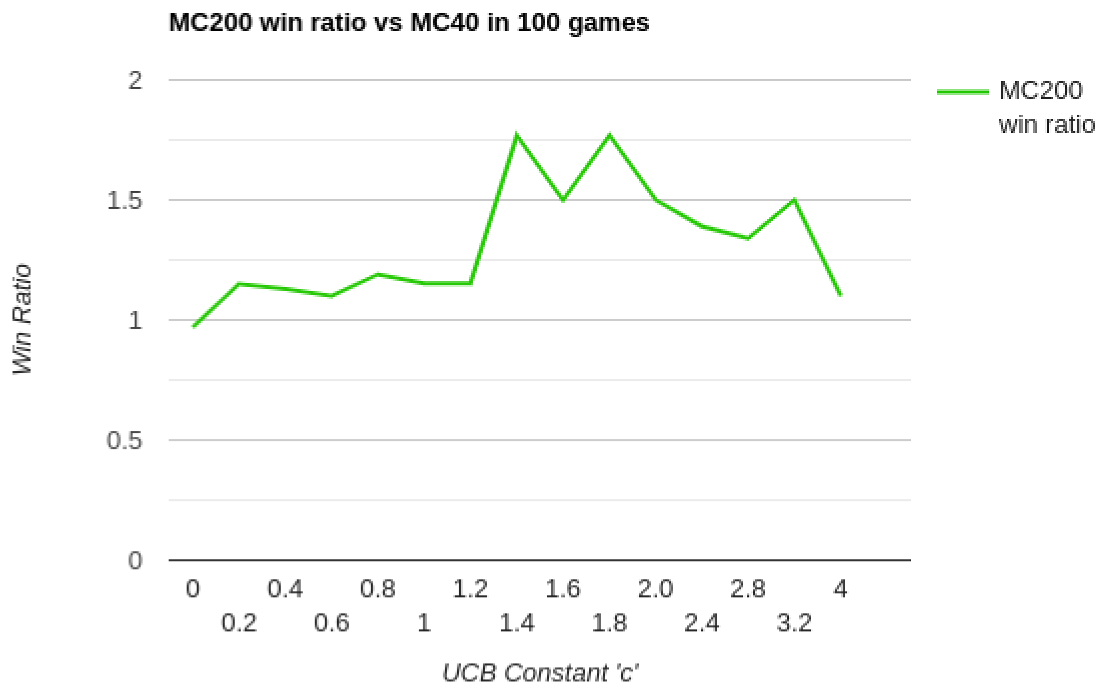
★ If c is low, (say c = 0), we are ignoring the second term in the sum, thereby ignoring a particular node's visit count. Hence, we are only **exploiting without exploring.**

★ Higher the value of c, more is the emphasis on the second term, which is inversely proportional to the visit count of a child node. In other words, as we increase c, we tend to favour nodes that have been visited less.

★ If a particular child node has not being visited for some iterations (instead it's sibling nodes are selected from it's parent), the value of the numerator term increases without any change to the denominator, hence boosting the UCB value for this particular child, giving it a fair chance to be selected.

★ We may also note that if N(n), the number of visits for a child is 0, the value of its UCB tends to ∞. Hence, in our MCTS algorithm, we first fully expand a given node (thereby visiting each of its children once), and only after a node is fully expanded, we may use the children's UCB values to select the best child to simulate on.


MC200 and MC40 wins vs UCB constant 'c'
Wins in 100 games vs Opponent
UCB Constant 'c'
MC200 wins
MC40 wins
0
20
40
60
80
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2.0
2.4
2.8
3.2
4


MC200 win ratio vs MC40 in 100 games
Win Ratio
UCB Constant 'c'
MC200 win ratio
0
0.5
1
1.5
2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2.0
2.4
2.8
3.2
4

Note that each value on the above graph is the average (over 2-3 trials) of the result of 100 games played between MC200 and MC40 on the given value of ‘c’.

From the plots we can notice that MC200 performed best against MC40 for **higher values of c (>1.2)**, and the performance peaked at c = 1.4 and c = 1.8. However, the performance dropped after, and at c = 4, MC200 and MC40 performed almost similarly.

Inferences from the plots

MC200 can win against MC40 by utilizing the parameter of which it has a larger value: **The number of rollouts**. Hence, MC200 has a much longer available time to try and “explore” the search space for better rewards as compared to MC40. So we must **promote exploration to favour MC200** against MC40.

As seen from the plots, lower values of c, i.e, c < 1.2 show almost similar performance (in terms of wins) between MC40 and MC200. This is because since c is small, exploration is not favoured as much as exploitation, so, MC200 is not able to fully utilize it’s high amount of rollouts to explore and will instead perform similar moves as MC40.

For higher values of c, MC200 will perform better, as now it has more “freedom” to explore, MC40 will also explore (since parameters are changed for both versions), but since 40 < 200, it will have much less rollouts to do so and figure out an optimal move. I have taken an average of 2-3 runs (each of 100 games) to plot the graphs, but in certain other runs of 20-30 games I found instances where MC40 didn’t win even a single game against MC200, for values of c = 1.4 and above.

When we further increase the value of c, we essentially reduce the contribution of exploitation, which is obviously essential. Without exploitation, both MC200 and MC40 act more or less as “random” agents, hence we cannot say which one would perform better, but on an average, they perform similarly.

Rewards Received

I implemented MCTS with the following reward scheme for all values of ‘c’:

Reward on WIN: +1, Reward on LOSS: -1, Reward on Tie: 0.25.

I also tried the following reward schemes (in similar ratios)

- ★ WIN: +3, LOSS: -1, Tie: 0.25 (Large Reward on Win)
- ★ WIN: +1, LOSS: -3, Tie: 0.25 (Large Negative Reward on Loss)
- ★ WIN: +1, LOSS: -1, Tie: 0 (No reward on Tie)
- ★ WIN: +1, LOSS: -1, Tie: 0.5 (Slightly higher reward on Tie)

I found that among the schemes mentioned above, the first three **do not affect the final result by a large factor.** Hence I decided to go ahead with the original scheme.

Another observation came from the value of ‘c’ and the appropriate reward scheme. I noticed that c = 1.4 ≃ (sqrt(2)) worked best for **positive rewards in the range [0, 1].** For higher values of c however, positive rewards greater than 1 worked better.

For example, for **c** = **1.8**, I noticed:

Reward Scheme ➔ W: +1, L: -1, D: 0.25	Reward Scheme ➔ W: +2, L: -2, D: 0.25
MC200 Win ratio: 57/40 = 1.425	MC200 Win ratio: 63/33 = 1.74

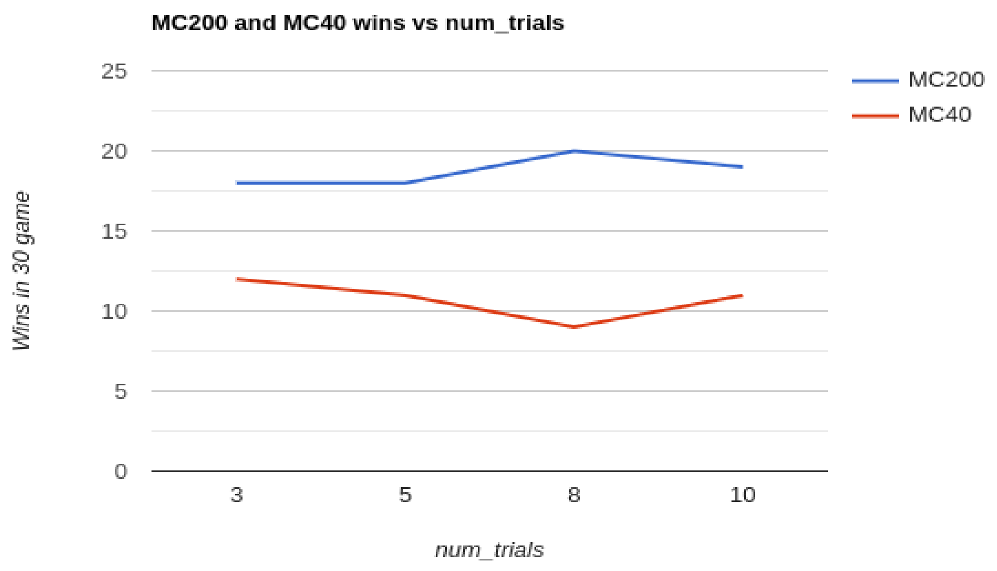
So, in my ‘c’ value testing plots, the rewards for c >= 1.814 were those of the second column of the above table. However, for c < 1.814, the rewards I chose for an optimal algorithm were those from the first column.

Number of Trials “num_trials” parameter:

During my experimentation on MC200 vs MC40, I tried to print the game board at each move and notice the reasons why MC200 loses (by noticing the non-optimal move at a particular state). I then **separately** tried to make MC200 play a single move on the same state where it lost previously. I noticed that about **3-4 out of 5 times, MC200 actually made the optimal move** from that position.

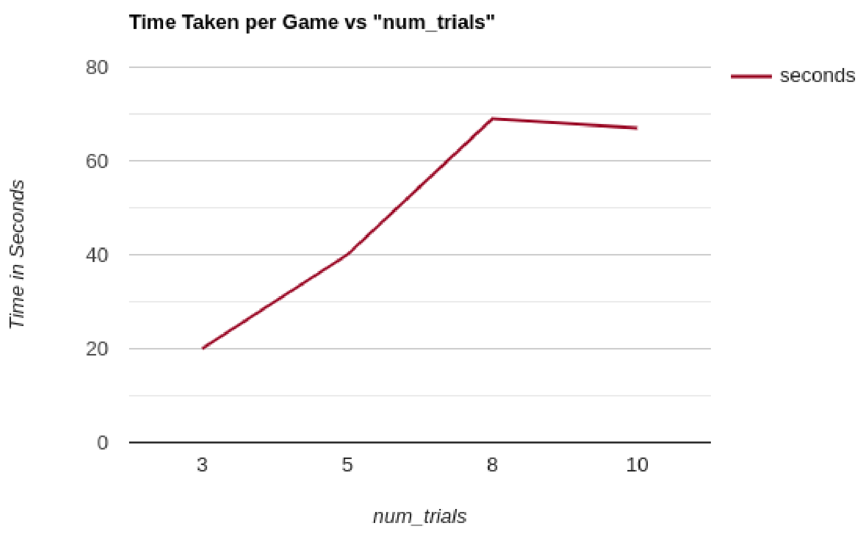
Hence, some of the times that MC200 lost to MC40, the reason came down to an unfortunate wrong move (this is possible because of the **non-determinism** involved in certain areas of the algorithm where we use random numbers).

So, before every move, I decided to run the algorithm some “**num_trials**” number of times, storing the action taken in each “trial”. Finally, the algorithm picks the **most repeated action** in these num_trials trials. The results have been plotted:



Due to increasing time (discussed below) and the fact that num_trials = 3 and 5 performed similarly, I decided to go ahead with num_trials = 3.

Clearly, it is most optimal to increase num_trials to produce a better move, in order to completely remove “unfortunate” moves caused due to nondeterminism. However this comes at the cost of increasing playing time: The time taken per game of MC200 vs MC40 increased **3 times** by increasing “num_trials” from 3 to 8. In order to balance the time constraint to produce an overall optimal and time-conserving algorithm, I concluded that num_trials = 3 is a good value.



Q-Learning

Q-learning is a **model-free reinforcement learning algorithm** to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. It can identify an **optimal action-selection policy** for any given Finite Markov Decision Process (FMDP), given infinite exploration time and a partly-random policy. "Q" itself refers to the function that the algorithm computes – the expected rewards for an action taken in a given state.

Size of our Game State-Space

In the version of connect-4 implemented for testing MCTS (6 rows and 5 columns), we try to calculate (or at least form a rough upper bound) on the number of states possible, this is important because Q-Learning tries to find out for each state the optimal action that results in better performance in terms of game-wins.

There are 30 holes in the board and each hole may be filled with “1”, “2” or it may be empty “0”. Hence a very **rough upper bound** on the number of states required is 3^{30}. Obviously lots of these states would be “invalid” because we cannot have a

“1” or a “2” above a hole that reads “0”, but it gives us a rough idea about the scale of the state-space.

This is a very large number and in general, would take days to train up a Q-Learning model that converges to the optimal values for state-action pairs. So I have reduced the Connect-4 board to a size: 5 columns and “**r**” rows.

I have tried training Q-Learning for r = 2, and r = 3. For r = 3, I was able to create a Q-mapping that performs considerably well, and can beat certain versions of the MC algorithm.

Even for r = 3, an upper bound on the number of states would be 3^{15}, which is also an extremely large value.

The following **plots and discussion** shows us how the Q-Table generated from training the Q-Learning algorithm performs against the MCTS game playing family.

Q-Learning Training Details:

ε, Exploration Rate = 0.3, tells us how much we explore the search space. I decided to go with a slightly higher value because from my experience in MCTS, I found that it was **important to explore to find a better future move**. Due to the nature of the Connect-4 game, we usually cannot say much about a particular state until and **unless we are 2-3 moves away from winning/losing**. So it is important to have a slightly higher exploration rate

α, Learning Rate = **0.35,** again a slightly higher value because we can only know a particular state's (and state-action pair) worth from it's children states. So it is essential to change a state-action pair's value by a larger amount (hence learning rate is slightly larger than usual) depending upon the result or reward of following the path of it's children's states.

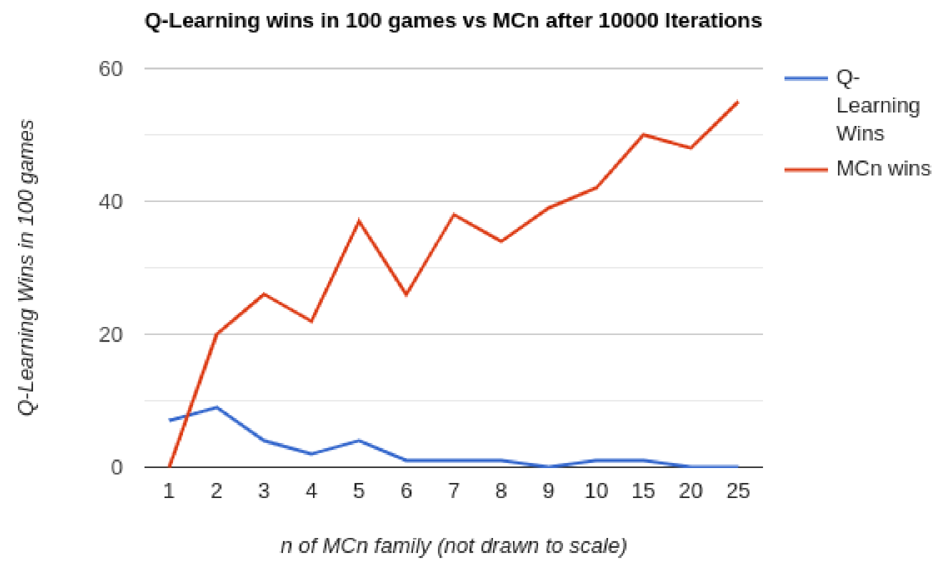
γ, Discount Factor = **0.9,** we do not want to discount future rewards too much, as they are indeed important for the current state. So we keep it at a moderately higher value

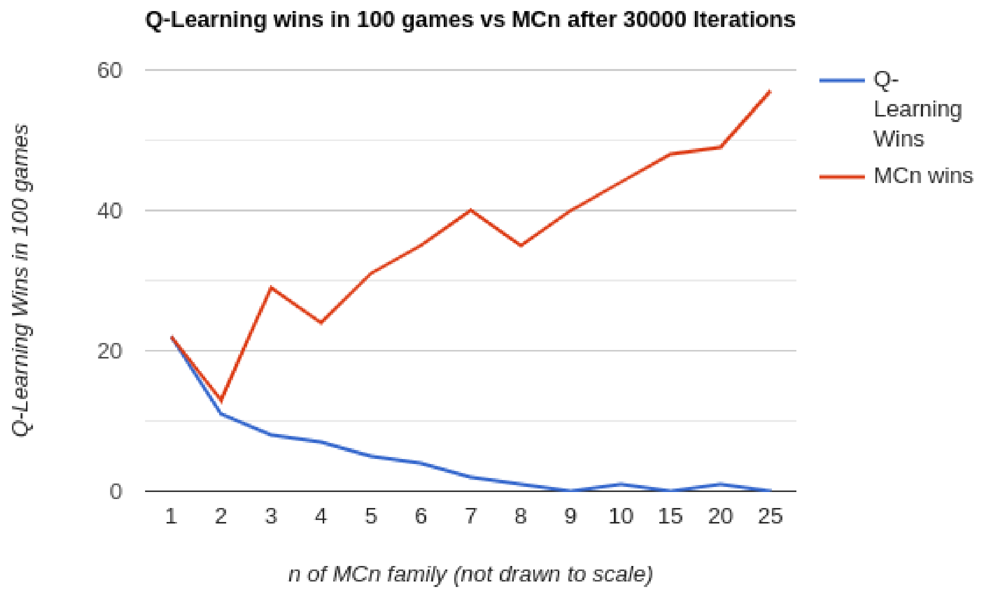
State Representation in the Data file:

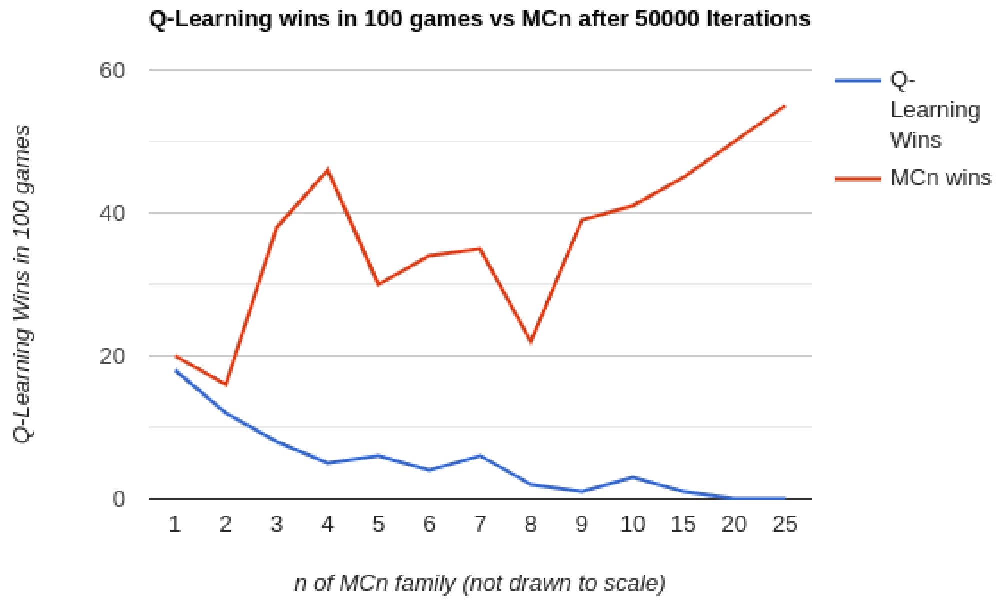
The data file consists of a **python dictionary**. The keys are tuples: (State, Action) where state is the string version of the 2D list state (because Python cannot hash lists), and Action is an integer 0-5. The value is the value of that state action pair

Iterations (Episodes):

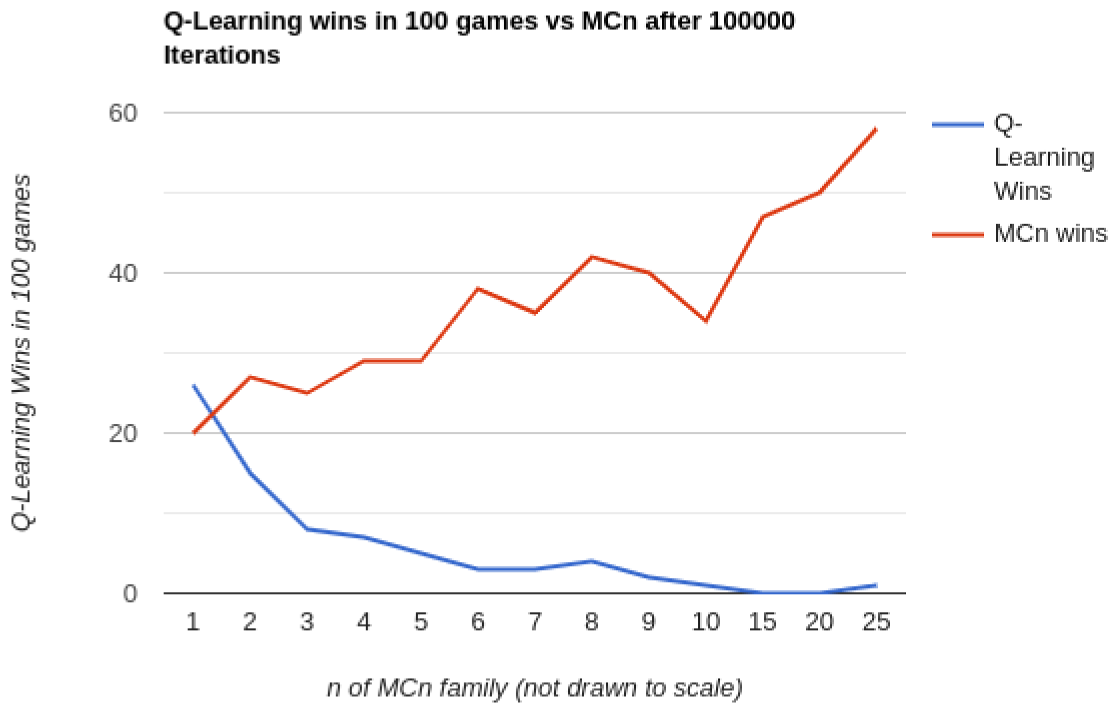
Depicted below are the plots of the performance of the trained model after a certain number of episodes had passed:







Finally, after 100000 iterations:



We notice that after 100000 episodes, Q learning was able to **comfortably defeat MC1 and play at par with MC2.** It was also able to win matches upto about MC5, and after n = 5, its performance deteriorates when compared with MCn.

Afterstates

The concept of afterstates means that we **evaluate a state's value after the agent makes its move**. Indeed, I have used this concept during my Q-Learning training because we first allow the agent to make a move, and then validate the state of the board to be a winning/losing state. In fact, in most board games like Tic-Tac-Toe, we use the concept of afterstates while training Reinforcement Learning Solutions

Conclusion

I believe that the version of MCTS I implemented is an extremely good algorithm and perfectly apt for the game of Connect-4. Instead of simply having a "table" of state-action pair values and deciding a move on the basis of this (as in Q-Learning), MCTS uses actual playouts from a given state (at every move). Even a smaller value of n, MCn, can predict and perform the basic moves of detecting and completing a Four-Connect (win), and slightly higher values of n can predict the opponent's win approaching and block their opponent's Four-Connect.

Even after **100000 episodes,** which takes roughly **8 hours to train,** Q-Learning was only able to consistently beat MC1 - MC3, occasionally beating MC4-MC10, however after n > 10, it lost almost all games, showing the capabilities of the MCTS algorithm and the shortcomings of Q-Learning.

Indeed, to make Q-Learning perform better, we need to adopt some **modern improvements,** like deep-Q-Learning, which uses Deep Neural Networks. Critically, Deep Q-Learning replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a q-value, a neural network maps input states to (action, Q-value) pairs.