CS F407: ARTIFICIAL INTELLIGENCE REPORT

Programming Assignment 2

Omkar Garad

2019A7PS1010G

ABSTRACT

In programming assignment 2, we were expected to implement a Connect 4 board and implement a few RL agents to play the game.

The assignment was divided into 3 main parts:

a) Create 2 versions of the Monte Carlo Tree Search algorithm to play Connect 4 on a 6 rows x 5 columns board, with one version of the algorithm using 40 simulations and the other using 200 simulations. These 2 versions of MCTS should play against each other for 100 games, with MC_{40} being the starting player for 50 games, and MC_{200} being the starting player for the rest 50 games.
b) Implement a Q-Learning algorithm for playing a simplified Connect 4 game, with the number of rows *r* ranging between 2 and 4. Player 1 will be MC_n where n varies between 0 and 25, and Player 2 will be the Q-Learning algorithm.
c) Train the Q-Learning algorithm so that it can win games against the MC_n algorithm (where n again varies between 0 and 25)

I have merged part B and C together in this report, so that it flows easier.

INTRODUCTION

CONNECT 4:

The Connect 4 game is a two-player perfect information game in which the two players try to stack 4 coins in a row vertically, horizontally or diagonally in a traditional 6 row and 7 column grid. It is a perfect information game as each player knows the set of actions available to the other player at each state of the game. The state of the game after the possible actions are taken is also known. It is also a zero-sum game, as the pay-off of one player decreases as the other player's increases.

For Part A, we were told to create a 6 row and 5 column board. As we have 3 possible choices in each slot, i.e, blank, player 1 coin and player 2 coin, we have a total of 3^{30} states present for this board. For a generic board of *r* rows and *c* columns, the total number of states is 3^{r*c}. But out of these states, some are invalid.

Valid states are those where the number of coins of player 1 is either equal to the number of coins of player 2 or one more than the number. Also, it cannot be possible to have a coin "levitate" in the board and we cannot have a state after a certain player has won.

MCTS:

I represented each state as a 2-D list with *r* rows and *c* columns, present as a node of the tree. Each player (MC_{200} and MC_{40}) have a separate game tree constructed with 200 and 40 play-offs respectively.

In the very first game before selecting the very first action, the game trees are created by carrying out 40 and 200 simulations as suggested in the question, so that the subsequent simulations would help improve the player's decision. This is because the values of the nodes are improved before the first move is made (as the value of the nodes become more optimal as they're being chosen more).

The win reward was set to +5, the draw reward was set to +1 and the lose reward was set to -5.

Q-LEARNING:

Here, each state is mapped to an array of 5 actions, each containing the Q-value of the state-action pair. Each state contains it's reward, which is set to -0.01 for non-terminal states, +2 for win states, 0 for draw states and -2 for lose states.

The initial Q values are set to +1 as they were observed to converge slightly faster than keeping initial Q values to 0. These 2 were significantly better than having an initial Q value as -1.

The negative reward for non-terminal states is so that the Q-Learning algorithm converges at a faster rate and it doesn't take actions that lead it to non-terminal states.

I have assumed that convergence of Q-Learning occurs when the win percentage against the MC_n stagnates over a period of iterations of the game.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

We can also say that Q-Learning converges if the α*temporal difference term reduces to a certain threshold for each state action pair. But this isn't feasible to check for each state-action pair, during every iteration in a 3^{r*c} Q-table. Hence, I have assumed the above condition for Q-Learning's convergence.

PART A

What choice of parameters give a clear advantage for MC_{200} algorithm in terms of number of wins in 100 games?

1. **NUMBER OF SIMULATIONS**

In MCTS, as the number of simulations increases, the more different branches (and hence nodes and states) are explored. As we use the Upper Confidence Bound Algorithm in order to select the next move for the player, the values of the nodes must be a clear indication of the best move the player can play.

In this case,

> **Value stored in Node= sum of total rewards/ total number of times each node was visited**

The UCB algorithm has an exploitation term (a term that exploits a greedy move) and an exploratory term (a term that explores unexplored branches and nodes). The more a node is visited, the more the value of the node converges to the optimal value. Also, the more sure the player is that the greedy move they're playing is the *correct* greedy move. In the ideal scenario, if the number of simulations tended to infinity, then the values of the nodes would stagnate to the optimal value and the player would just need to choose the next moves (states) greedily in each turn.

Hence, as MC_{200} has 200 simulations before making a move as compared to MC_{40}'s 40, it has an advantage with respect to number of wins. Therefore, ideally MC_{200} must win most of its games against MC_{40}.

2. **VALUE OF C IN UCB ALGORITHM**

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

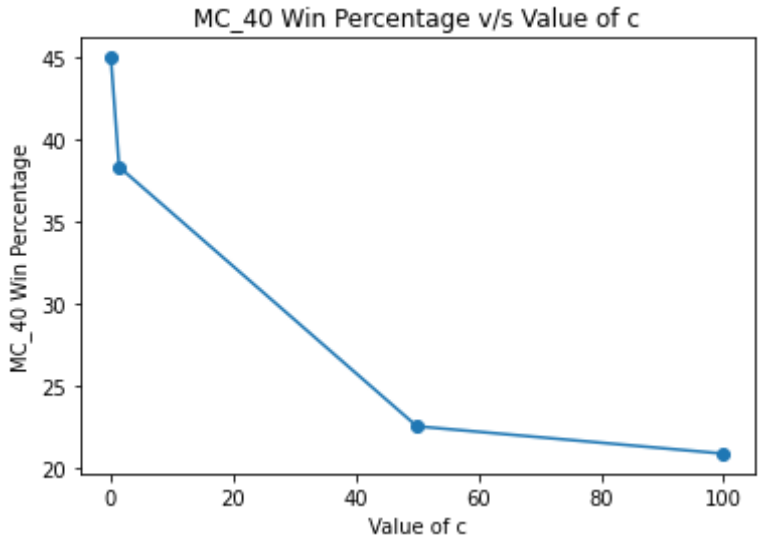
As mentioned above, the UCB algorithm gives more preferences to actions that haven't been taken yet. Due to the exploratory term, actions that have been chosen less have a larger value (as the denominator is small) and hence are more likely to be chosen. Hence, if any node is unexplored it is definitely chosen as the expression value is the highest.

c is a parameter that controls the weightage to be given the exploratory term.

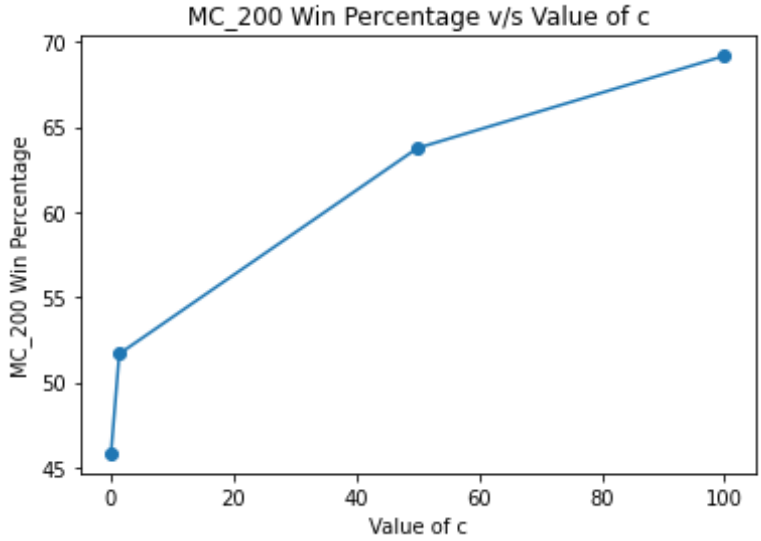
For large values of c, exploration is high.

For small values of c, exploration is low.

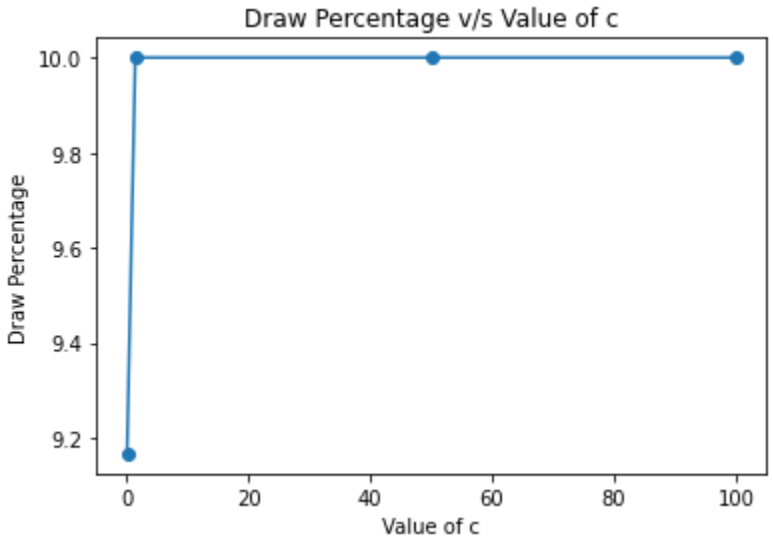
I changed the value of c to 0.1, √2, and 100 to observe the impact it would have on MC_{200}'s win percentage.



Here, we can see that as the value of c increases, the win percentage of MC_{40} decreases.



Here, we can see that as the value of c increases, the win percentage of MC_{200} increases.



Here, we can see that as the value of c increases, the draw percentage stagnates at around 10%.

For c=0.1, the win percentages of MC_{200} didn't have a significantly higher win percentage as compared to MC_{40}.

For c=√2, the winning percentages of MC_{200} increased and that of MC_{40} decreased.

For c=100, the winning percentages of MC_{200} increased much higher than that of MC_{40}.

This is because, as my rewards are +5 and -5, and the visit count increases by 1 every time a node is visited, the **ratio of rewards to visits (the exploitation term)** is 5 (a relatively high value). Hence, to counteract the high exploitation term, a large value of c gives importance to the exploratory term as well, thus allowing MC_{200} to make well informed moves.

For small values of c, MC_{200} performs worse as it doesn't explore as many branches.

Hence, it isn't sure whether the greedy move it makes is the best move or not.

For large values of c, MC_{200} performs better as it has explored many branches and hence various states and is well informed about the move it has to take.

3. ADDING REWARDS FOR DRAWS

In this assignment, I gave a +5 reward for every win and a -5 reward for every loss. I also incorporated a +1 reward for a draw, so that the player would not be tempted to play moves that could potentially lead to a win in future moves, and not pay attention to certain losses.

I calculated the average win percentage for each player as well as the draw percentage of 6 runs of 100 games each when the reward for drawing a match was set to 1 and when it was set to 0. All other parameters were kept constant. The results obtained were:

Percentages Calculated (over 6 runs of 100 games)	draw_reward=0	draw_reward=1
Average MC_{40} win percentage	22.5%	20.83%
Average MC_{200} win percentage	64.167%	69.167%
Average draw percentage	13.33%	10%

Here, we can see that incorporating a positive reward for draws, increases the average win percentage for MC_{200} and also decreases the average win percentage for MC_{40} and the average draw percentage.

This is because the player now avoids going for an uncertain win in a future state and rather blocks a certain loss in the next move.

Hence, including a small positive reward for drawing the game (that is obviously less than the reward obtained from the player winning) gives MC_{200} an advantage.

4. **CHANGING THE VALUES OF REWARDS**

As mentioned above, changing the values of the rewards obtained would affect the exploitation term. Hence, we would have to change the parameters that affect the exploratory term in order to balance out the decisions that the player makes, so that they aren't skewed towards either greedy or exploratory moves.

For low values of rewards, we would need to reduce the value of c.

For large values of rewards, we would need to increase the value of c.

PART B AND C

Two-player games are always stochastic. Even in perfect information games, where each player knows the opponent's possible actions and next possible states, there is no certainty for which state will be chosen. Hence, there is still a stochastic element. The deterministic part is our actions, as we know the state after our turn.

The concept of afterstates is that only the afterstate is saved. It stores the data of the previous state and how to get to that particular state. If we don't consider afterstates, then the algorithm will consider the same state, visited by two or more different paths, as the same state.

Afterstates are effective in many games, are a useful concept and hence are effective in connect 4 as well.

I have tweaked various parameters that affect how the Q-Learning algorithm learns.

1. **CONSTANT OR DECREASING EPSILON**

The action is selected with the ϵ-greedy policy, while the state-action values are updated greedily. Hence, Q-Learning is an off-policy learning algorithm.

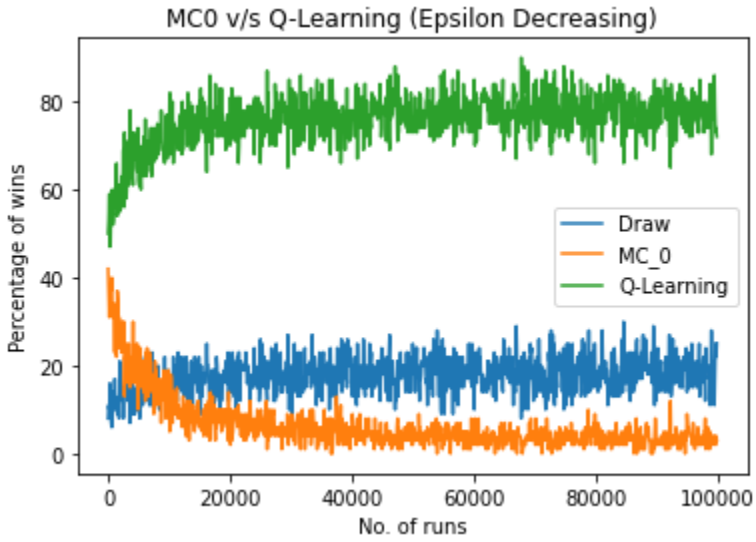
With a probability of ϵ, a random action is chosen from the set of all possible actions.

With a probability of 1-ϵ, the greedy action is selected. The random action is to allow

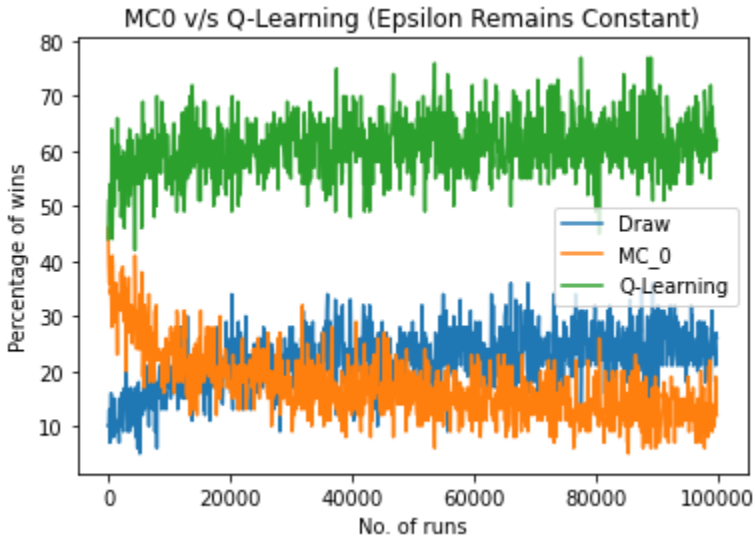
exploration to occur. Instead of always choosing the greedy action, with a small probability the player chooses an exploratory move so that they are sure that the greedy moves are the optimal moves.

However, the value of ϵ should decrease with time, so that the player can initially explore various possible moves, and finally when they are assured of the greedy action, they can continue to **ONLY** take the greedy action.

Win percentage of Q-Learning with the value of epsilon decreasing per 500 games



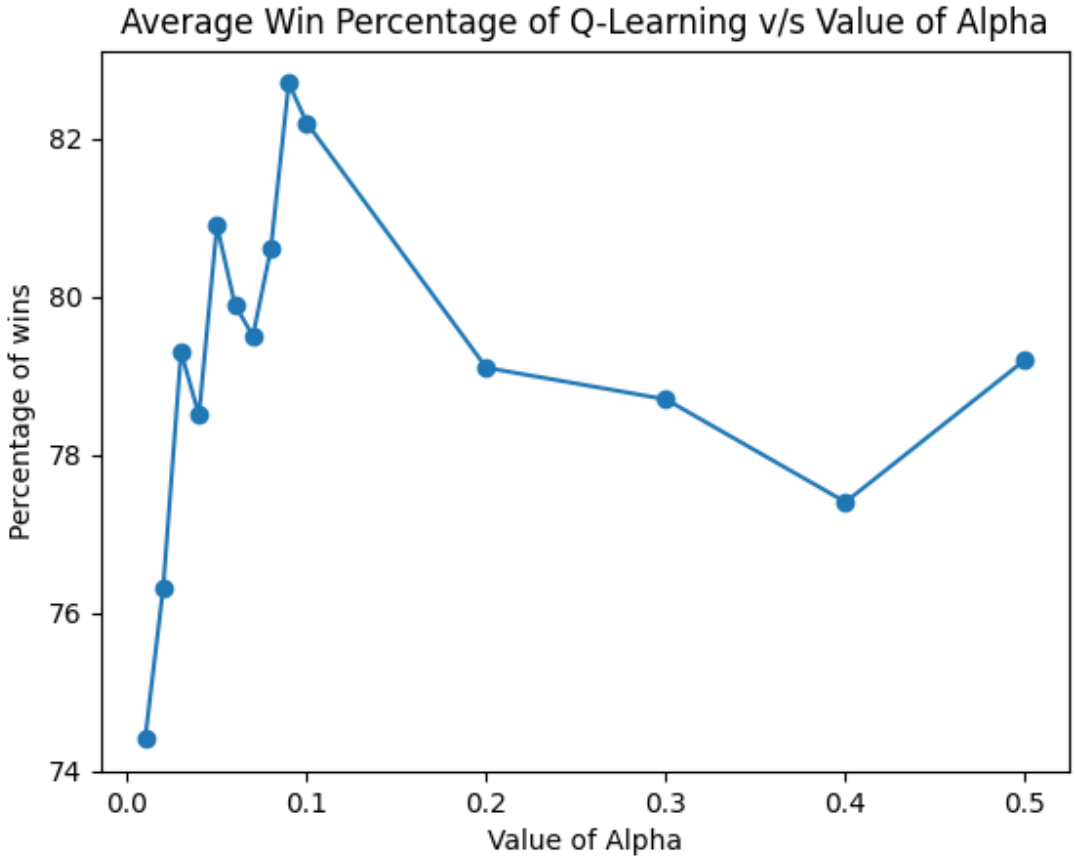
Win percentage of Q-Learning with the value of epsilon remaining constant.



Here, we can see for a decreasing value of ϵ, the Q-Learning algorithm converges faster than if the value of ϵ remains constant, due to reasons stated above.

2. **OPTIMAL LEARNING RATE (α)**

α is the learning rate, i.e., the rate at which the Q-Learning algorithm learns (usually between 0 and 1). If the value of α is high, then learning occurs at a very large rate. However, if α is low and tends to 0, then learning occurs very slowly and the Q-values won't be updated. As seen in the graph below, an optimal value for α is 0.09.



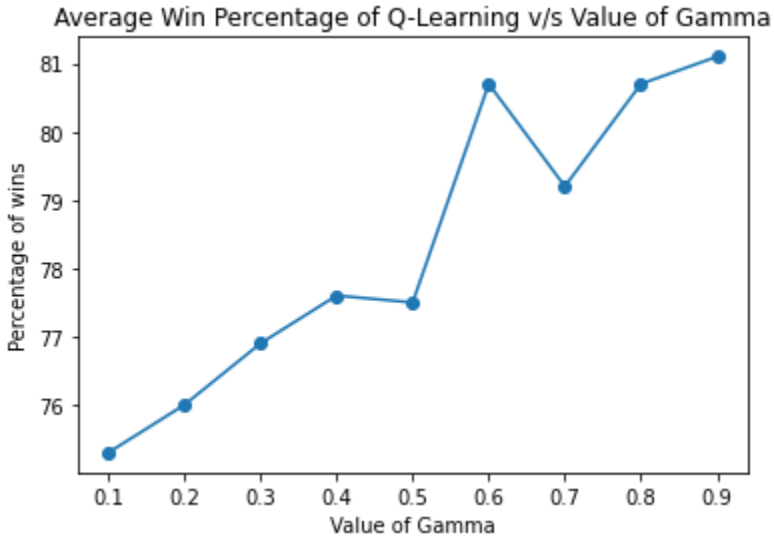
3. **OPTIMAL DISCOUNT RATE (γ)**

The discount rate γ determines how much importance is to be given to the future rewards. It determines how much the Q-Learning agent cares about rewards in the distant future relative to those in the immediate future. If γ=**0**, the agent will be completely myopic and only learn about actions that produce an immediate reward. If γ=1, then the agent will evaluate each of its actions based on the sum total of all of its

future rewards.

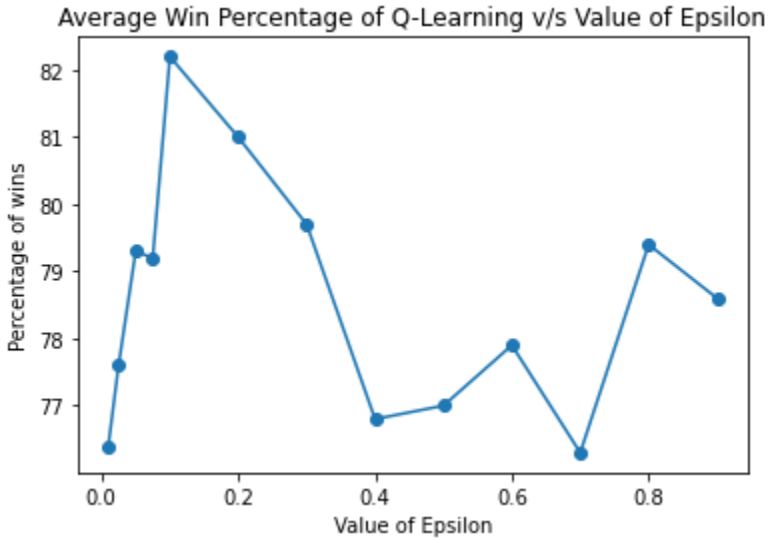
As the player is playing a game, it wants to see how its actions affect the rewards it gets in the future (whether a win or a loss). Hence, a large value of γ is essential for the algorithm to make moves in the game.

In the graph given below, we see that the optimal value for γ is 0.9. Hence, for the Q-Learning algorithm to converge quickly, it requires a large γ value as it considers future rewards while playing the Connect-4 game.



4. OPTIMAL EPSILON VALUE

As mentioned above, the value of ϵ decides the probability with which the player selects a random action. The probability shouldn't be too high, else the player will largely take random moves rather than choosing greedy moves and shouldn't be too less, else the player will choose only greedy moves and won't explore. Hence, according to the graph given below, the optimal value of ϵ is 0.1. **The X-axis has values of: 0.01, 0.25, 0.5, 0.75, 0.1, 0.2, 0.3, 0.4 and 0.5.**

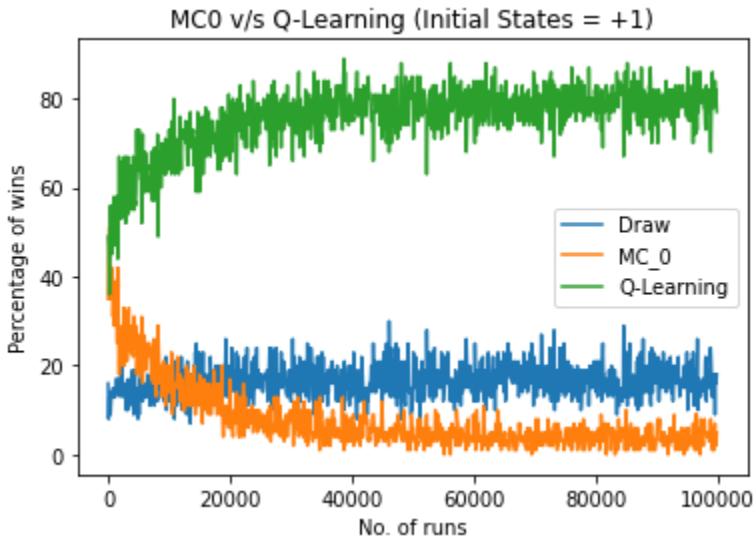


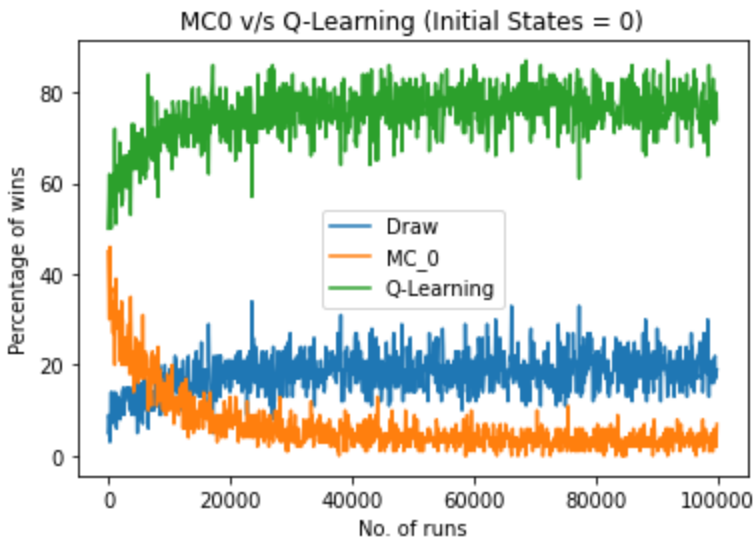
5. INITIAL Q VALUES

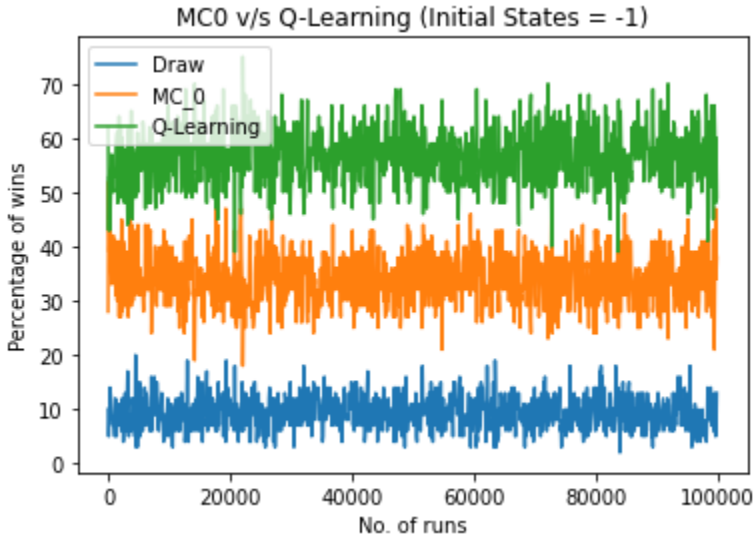
As seen below, having a positive initial value of states for the state-action pairs (Q values), leads to slightly better convergence than if the Q values were set to 0.

However, we see if the initial value of the state-action pairs is set to -1, then convergence doesn't occur.

Q Value = +1



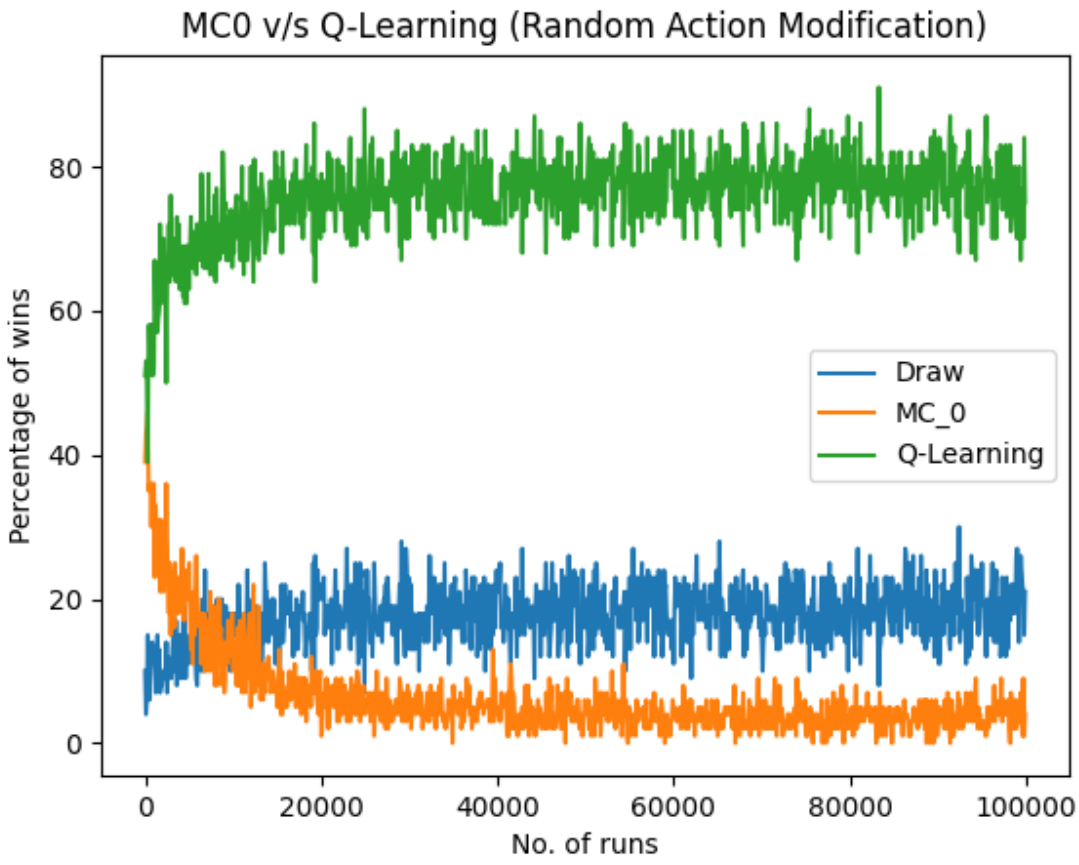




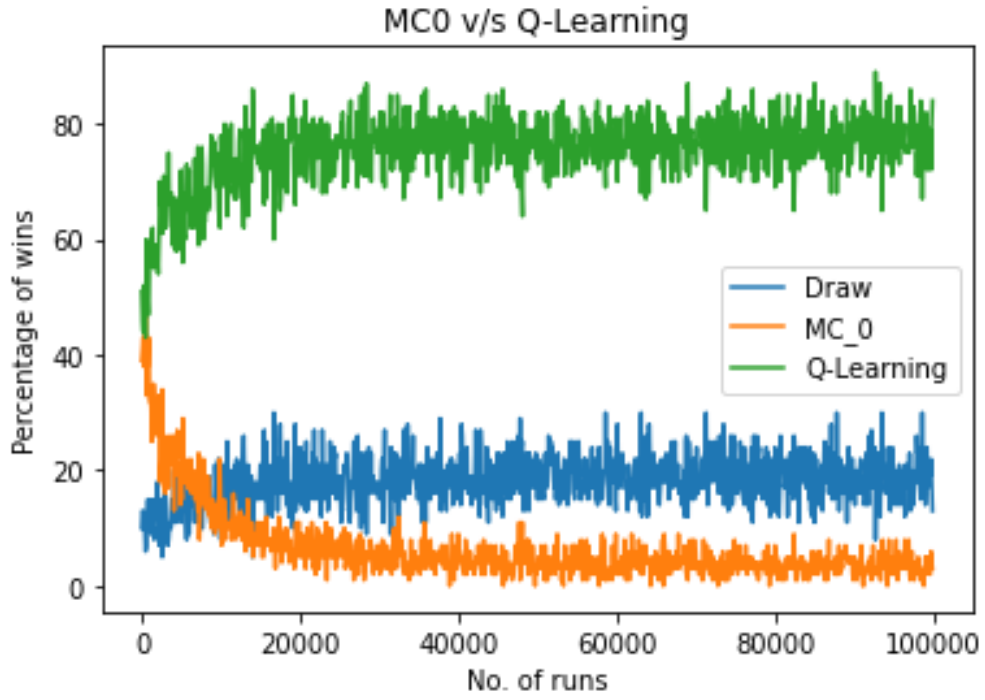
6. SELECT ACTION VARIATION

Instead of the ϵ-greedy policy randomly choosing from all actions possible at a given state, I made the policy choose from all actions that haven't been tried out yet, in order to encourage more exploration. As seen in the graphs below, there doesn't seem to be much difference in convergence rate for the ϵ-greedy policy and my modified ϵ-greedy policy.

Only actions that haven't been selected before



All possible actions

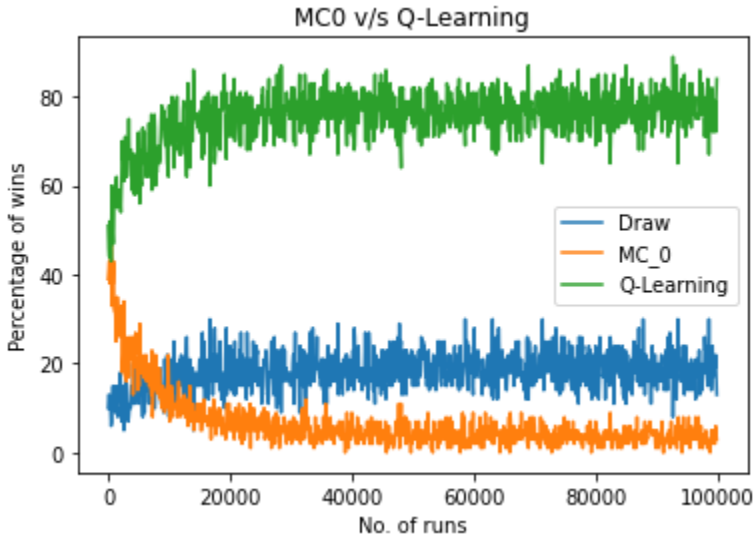


After selecting the optimal values, I obtained a convergence rate of around 80% wins for the Q-Learning algorithm against MC_0.

Increasing the exploration for MC_n allowed it to make better and more well informed moves against Q-Learning and hence could beat it, rather than with MC_0 which chooses random moves.

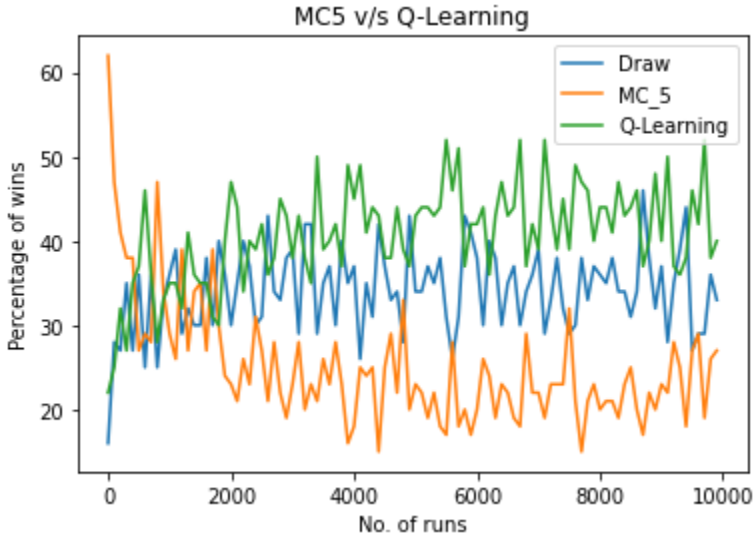
RESULTS

The Q-Learning algorithm performs very well against the MC_0 algorithm and converges to an average win percentage of around 80%.

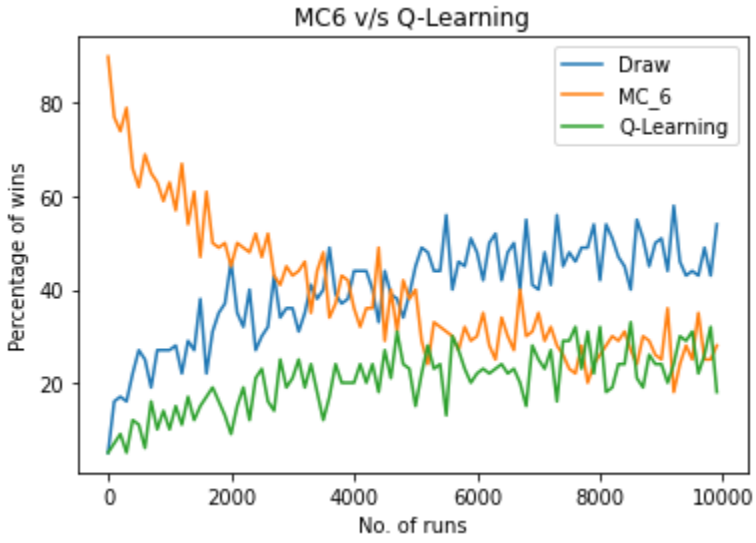


MC_0 v/s Q-Learning in 4x5 grid

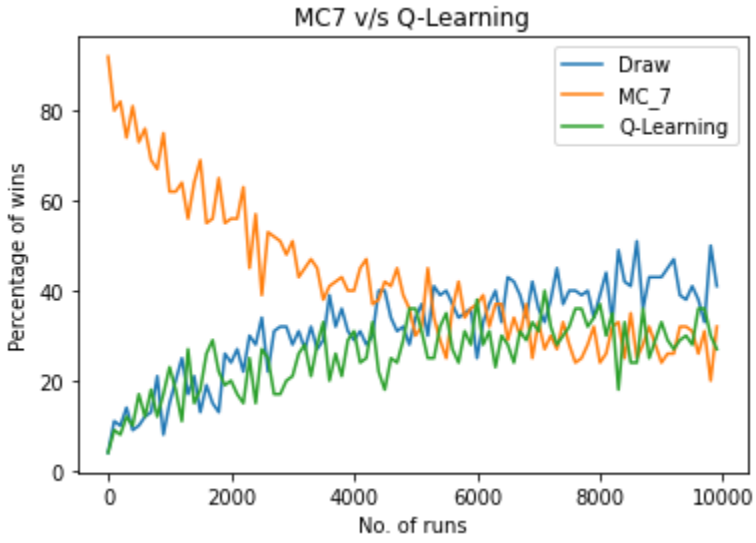
However, as n in MC_n increases, the average win percentage of Q-Learning decreases and it performs worse than the MC_n algorithm. This is because the MC_0 algorithm takes completely random moves, while if n increases, the MC_n algorithm performs many simulations, allowing it to make more informed decisions.



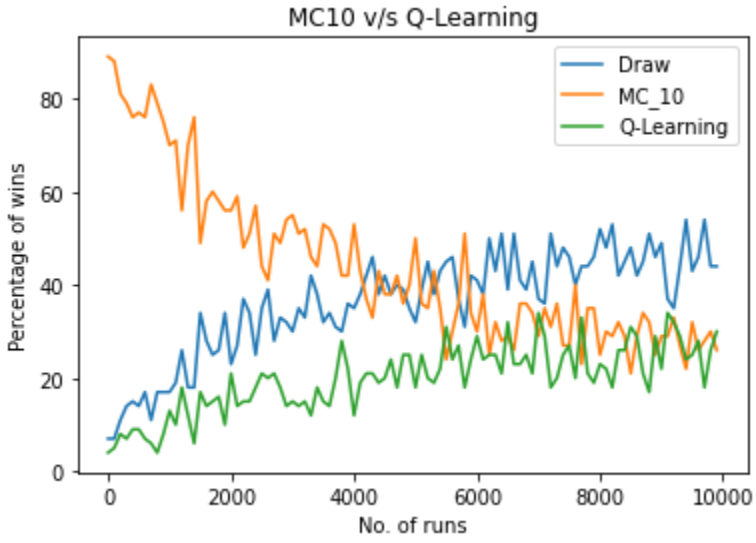
MC_5 v/s Q-Learning in 4x5 grid



MC_6 v/s Q-Learning in 4x5 grid

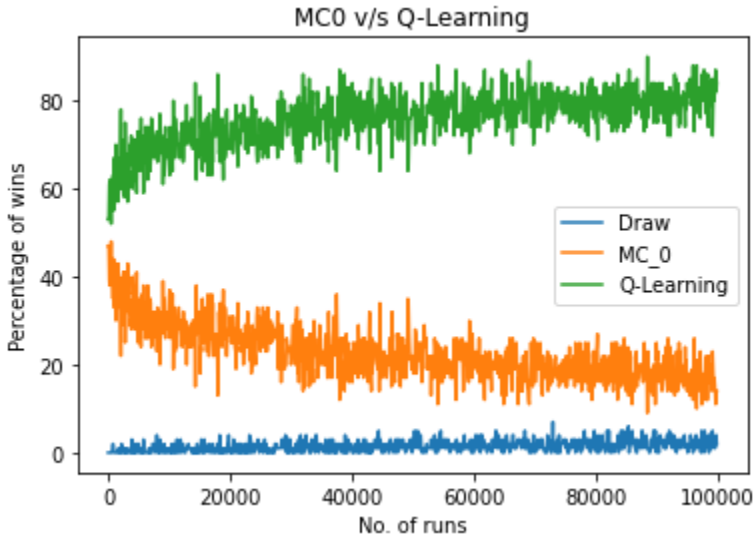


MC_7 v/s Q-Learning in 4x5 grid

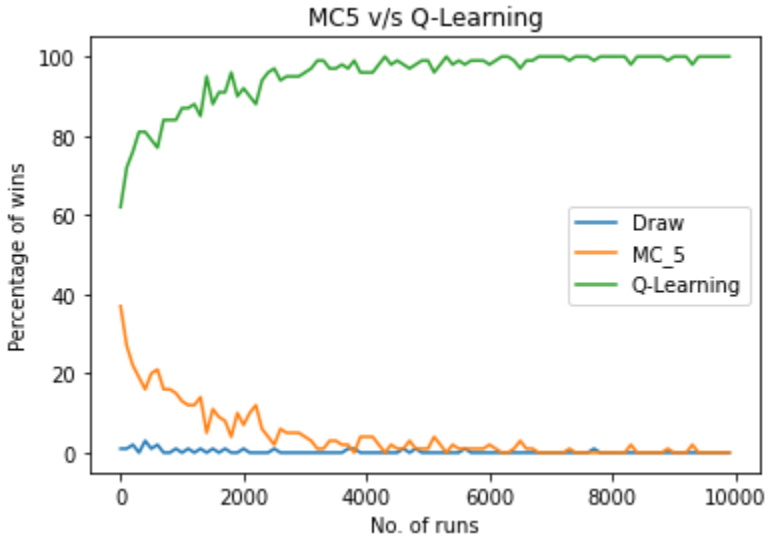


MC_{10} v/s Q-Learning in 4x5 grid

However, while experimenting with different values of *r* in the *r*x5 grid, I observed that for larger values of *r*, the draw percentage against MC_n decreased to near 0 values and the win percentages of Q-Learning increased and that of MC_n decreased.



MC_0 v/s Q-Learning in 6x5 grid



MC_5 v/s Q-Learning in 6x5 grid (for 10,000 runs)

In this case (against MC_5), the algorithm converged to 100% win percentage.