ARTIFICIAL INTELLIGENCE

ASSIGNMENT - 2 REPORT

BY -

AMAN SHARMA

2019A7PS0053G

INTRODUCTION -

Connect 4 -

Connect 4 is a classic 2 - player board game. Two players take turns dropping their discs in a vertical grid. A piece, when dropped, occupies the lowest available row in the column. The first player to form a line of four pieces wins. The line can be vertical, horizontal, or diagonal.

Monte Carlo Search Tree -

Monte Carlo Tree Search is an algorithm that evaluates the value of a state of a game. It involves 4 steps, selection, expansion, simulation, and backpropagation.

Simulation: Simulation is used to select nodes on the tree that have the highest possibility of winning. For every node, the algorithm store the number of times it has been explored, and the number of times it resulted in victory. An important consideration in the process of selection is the balancing of exploitation with exploration. This is usually done using a UCB policy.

$$UCB(node_i) = \overline{x}_i + c\sqrt{\frac{\log N}{n_i}}$$

$\overline{x}_i$: mean node value; n_i: #visits of node i; N #visits parent;

The node value in the above formula is a value function depending on the number of wins achieved from the node and the total number of visits to the node.

Expansion: After expansion, we pick an unexplored child node of the selected node.

Simulation: From the node selected during expansion, we start a game and play with random moves until we reach a win, lose, or a draw. We get rewards according to the final state of the board.

Backpropagation: The nodes visited during the selection and expansion phase are then updated with the reward received at the end of the simulation phase.

These steps are repeated a number of times, and the child node of the root node which is visited the most during the process is selected as the next move.

Q-Learning -

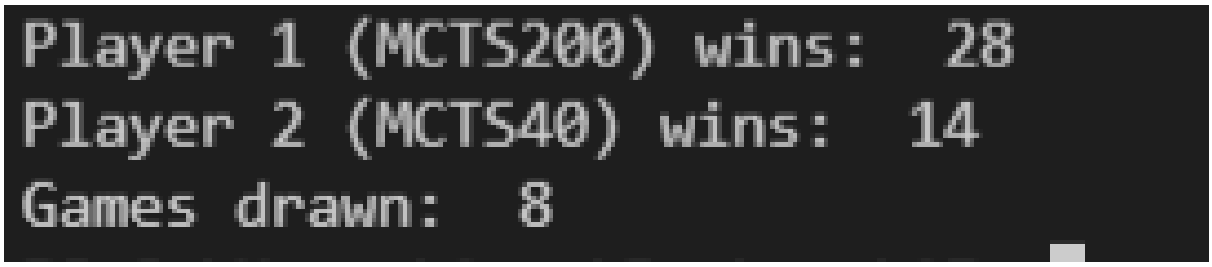
Q-learning is a reinforcement learning algorithm that seeks to find the best action to take given the current state. Q-learning tries to balance exploitation with exploration using an Epsilon-Greedy policy.

The policy is stored in form of a table with the value of every state-action pair. The algorithm uses these values to decide its move. These values are updated after every move.

Part A - (MCTS200 vs MCTS40)

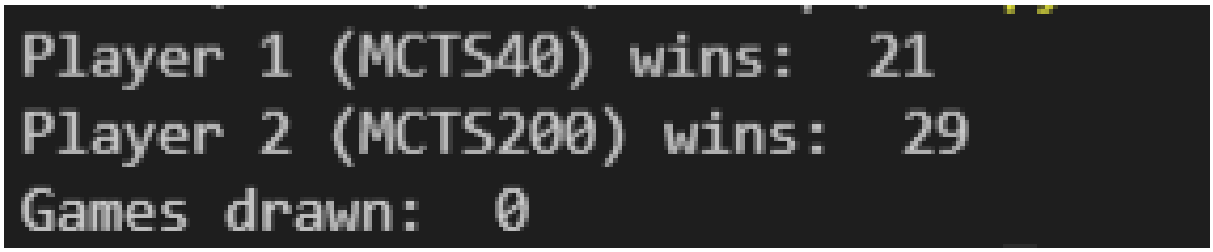
MCTS200 is a Monte Carlo Tree Search algorithm that performs 200 iterations of the algorithm before selecting a move. Similarly, MCTS40 performs 40 iterations before performing a move.

When 50 games are played with MCTS200 as the first player and MCTS40 as the second player, the results are:



It should be noted that these results vary with different runs of the program.

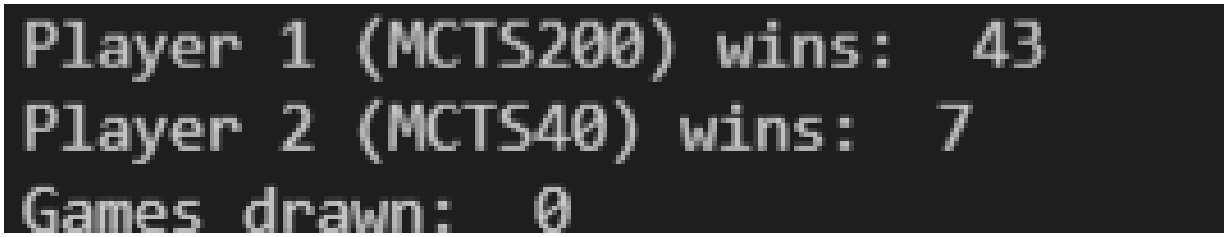
Similarly, when 50 games are played with MCTS40 as the first player and MCTS200 as the second player, the results are:



Multiple Rollouts -

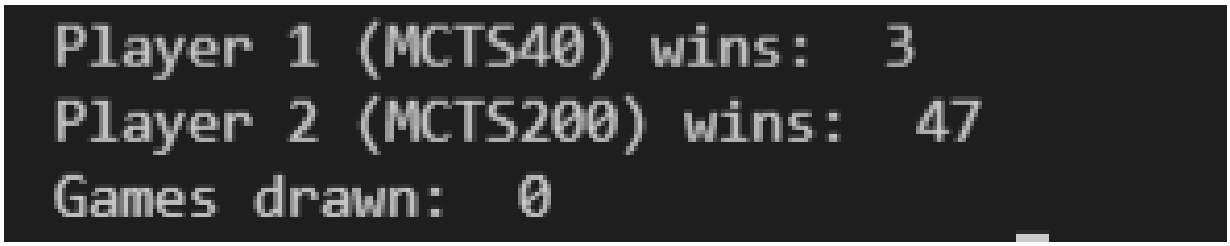
To enhance the algorithm, we can do multiple rollouts in the rollout stage of the Monte Carlo Tree Search algorithm. This way, we the algorithm gets more data points and hence makes better decisions.

After implementing multiple rollouts in our algorithm, when we play 50 games with MCTS200 as the first player and and MCTS40 as the second player, the results are:



It can be easily observed that MCTS200 gets a huge advantage with the implementation of multiple rollouts per iteration.

With MCTS40 as first player and MCTS200 as second player, when we play 50 games, the results are:



As we can see, even as second player, MCTS200 dominates the game when there are multiple rollouts per iteration.

Reward Function-

The reward function in the UCB equation for MCTS algorithms depends on the ratio of the number of wins and the total number of node visits.

As a modification, we can also give negative rewards on loss instead of zero. This should make the algorithm prefer draws over loses.

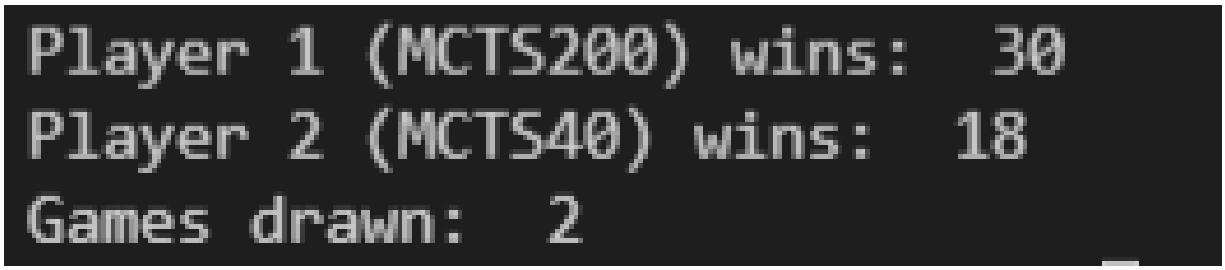
Original Formula -

2*sqrt(log(parentnode.count)/node.count) + node.wins/node.count

Modified Formula -

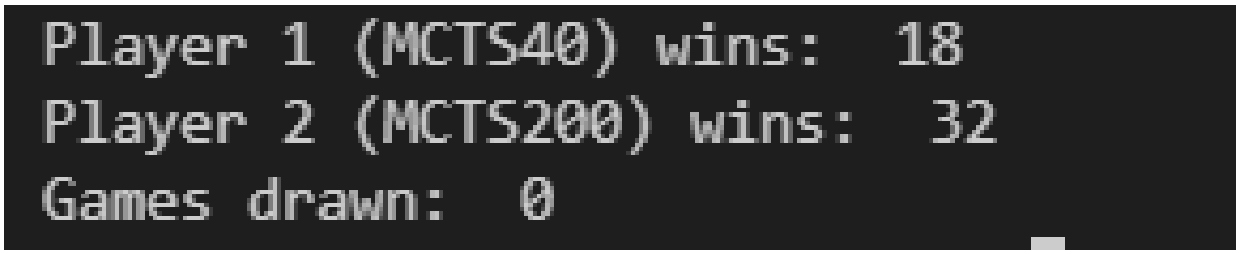
2*sqrt(log(parentnode.count)/node.count) + (node.wins-node.loses)/node.count

When 50 games are played with MCTS200 as the first player and MCTS40 as the second player, with a modified reward function, the results are:



As we can observe, the number of draws has decreased significantly compared to the original algorithm.

When 50 games are played with MCTS40 as the first player and MCTS200 as the second player, with a modified reward function, the results are:



Conclusion: Having multiple rollouts gives a very large advantage to MC200. It achieves a much greater win percentage with multiple rollouts compared to single rollout.
Modified reward value gives a slight advantage to MC200 which is probably because as the number of draws decreases with modified reward, more of those games are won by MC200 as compared to MC40.

Part B - (MCn vs Q-learning)

Afterstates :

Q-learning algorithm seeks to find the values of state-actions pairs. In our case, we can use after-states to map the values of state-action pairs to save memory and reduce the complexity of our program. We can look at state-action pair values as the value of the state we reach if we perform the given action at the given state.

To get the best action from a given state, we can compare the state values of possible next states, and choose the action that leads to the state with maximum value.

Game - state representation :

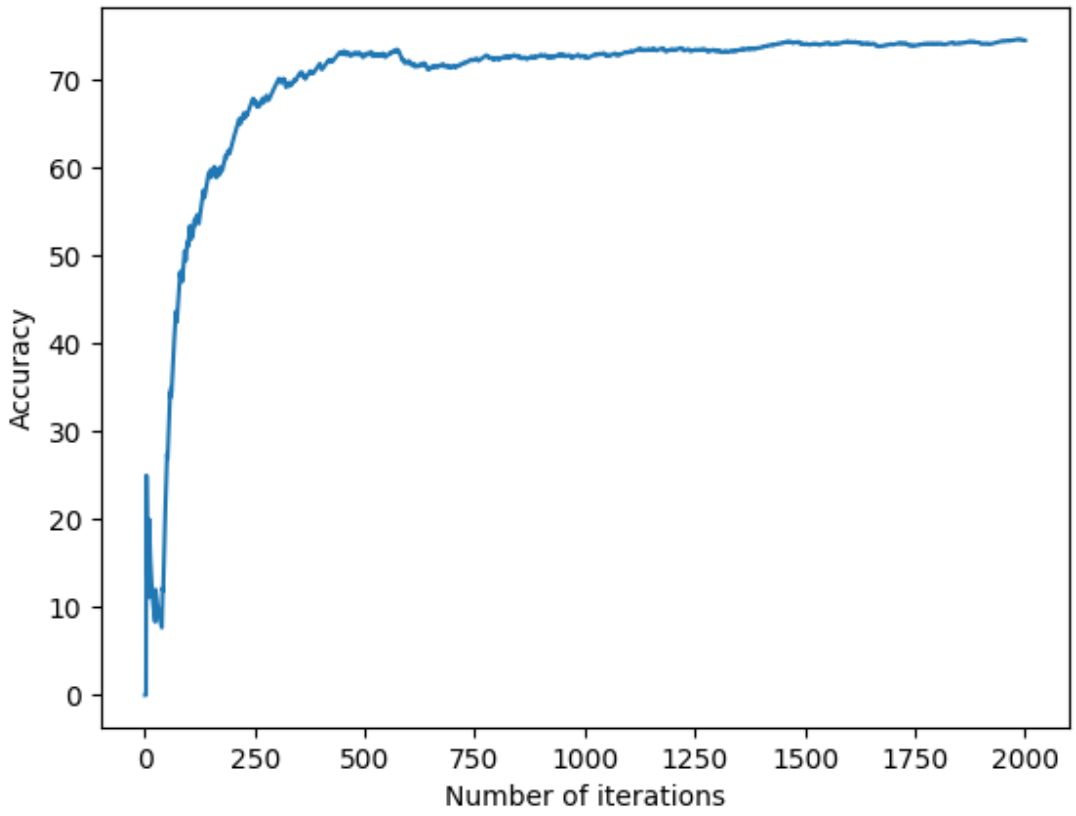
To map the states to their values, the game states have to be stored in a way that is hashable. We can do that by converting our NumPy board to a string using the function numpy.array2string(). This function converts takes a NumPy matrix and returns it in form of a string. This way we can directly store the values of afterstates in a dictionary.

To store the data generated by the Q-learning algorithm, the dictionary is written to a gzip file. It is retrieved from the file when the q-learning algorithm has to run.

Training the Q-learning algorithm:

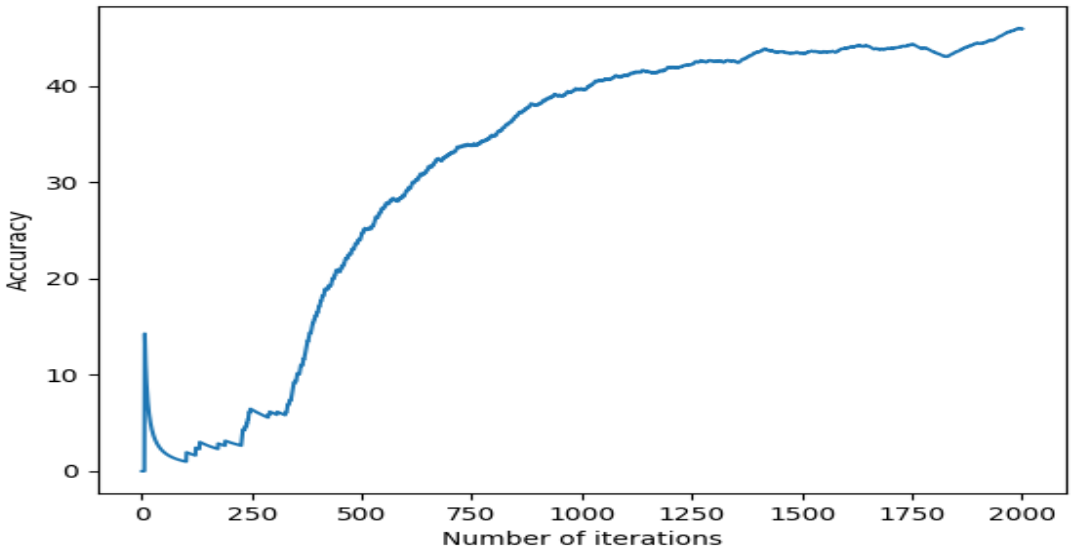
We train the Q-learning algorithm by making it play against our MCn algorithms.

Q-learning vs MC5 (ROWS=2):

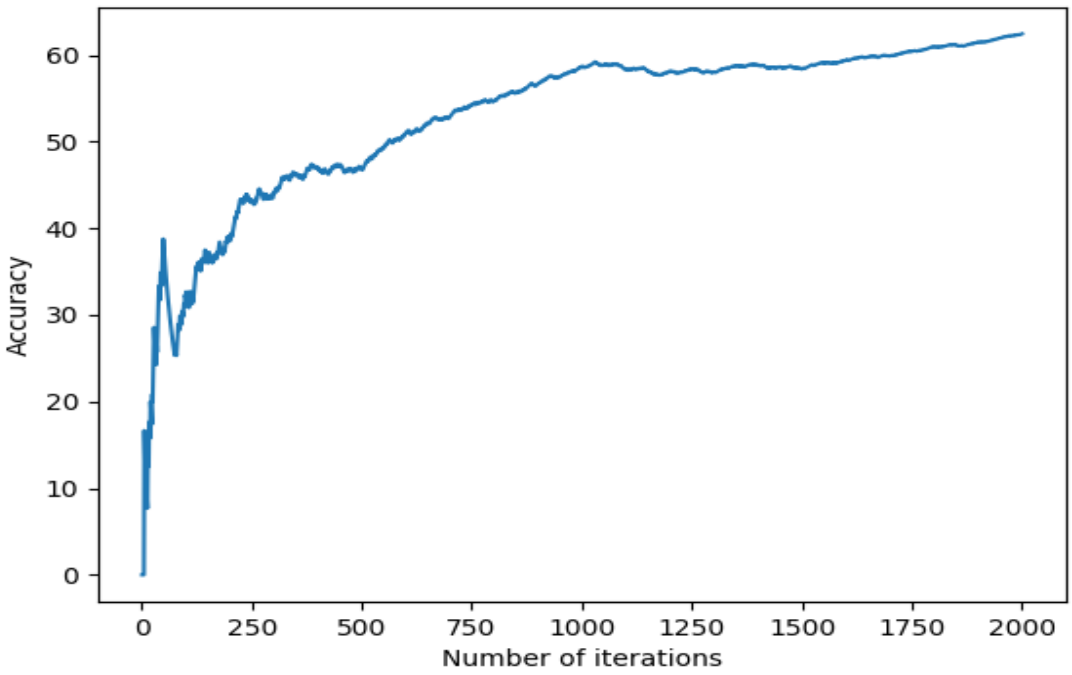


The vertical axis shows the win percentage (accuracy), and the horizontal axis shows the number of games played.

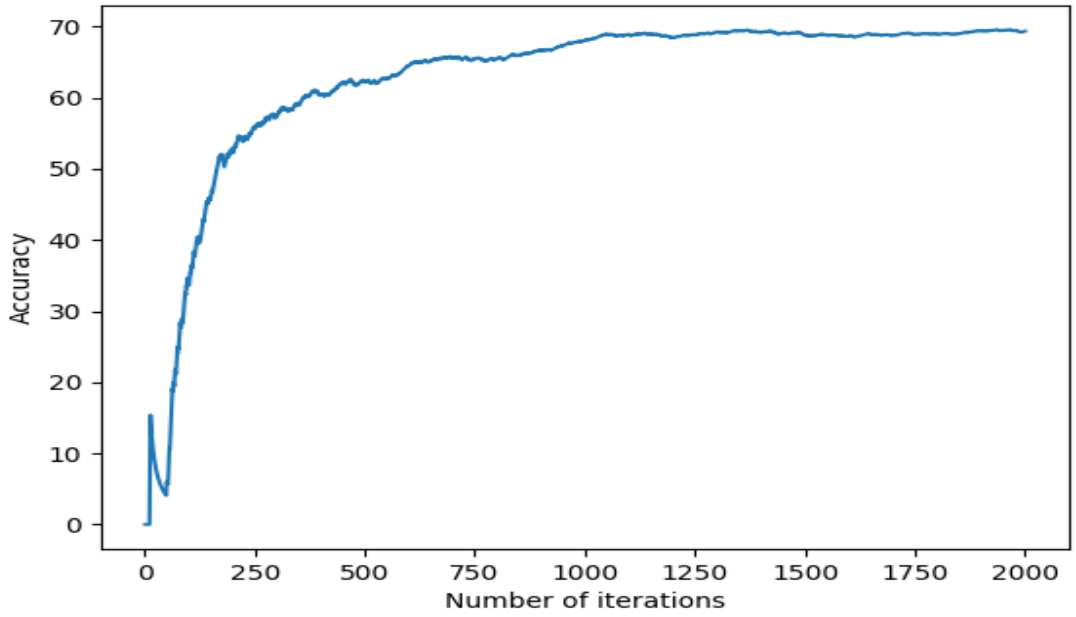
Q-learning vs MC5 (ROWS=3):



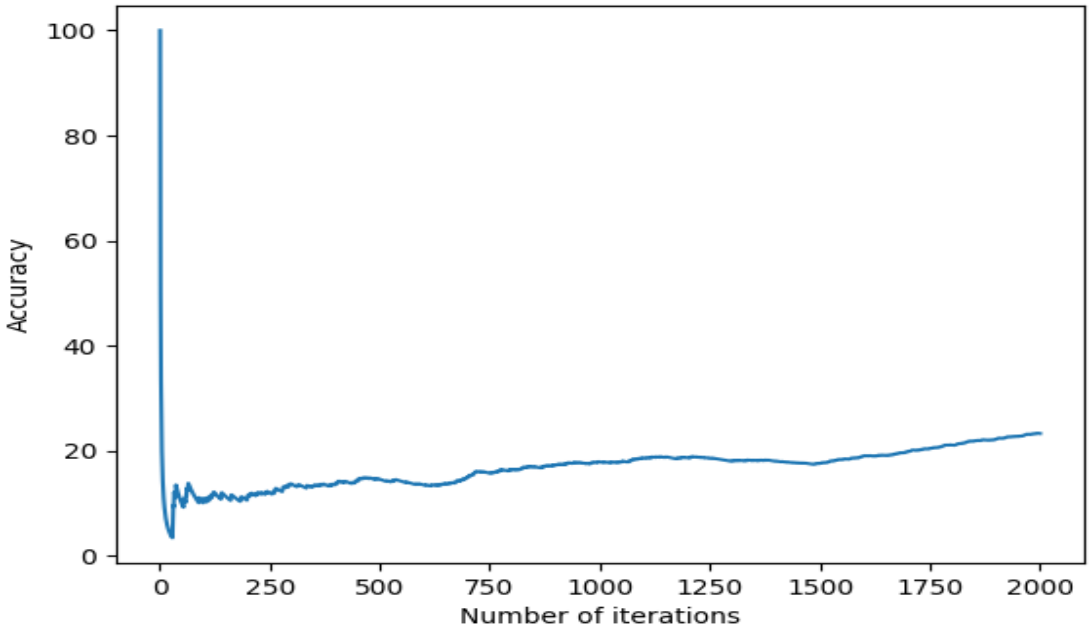
Q-learning vs MC5 (ROWS=4):



Q-learning vs MC10 (ROWS=2):

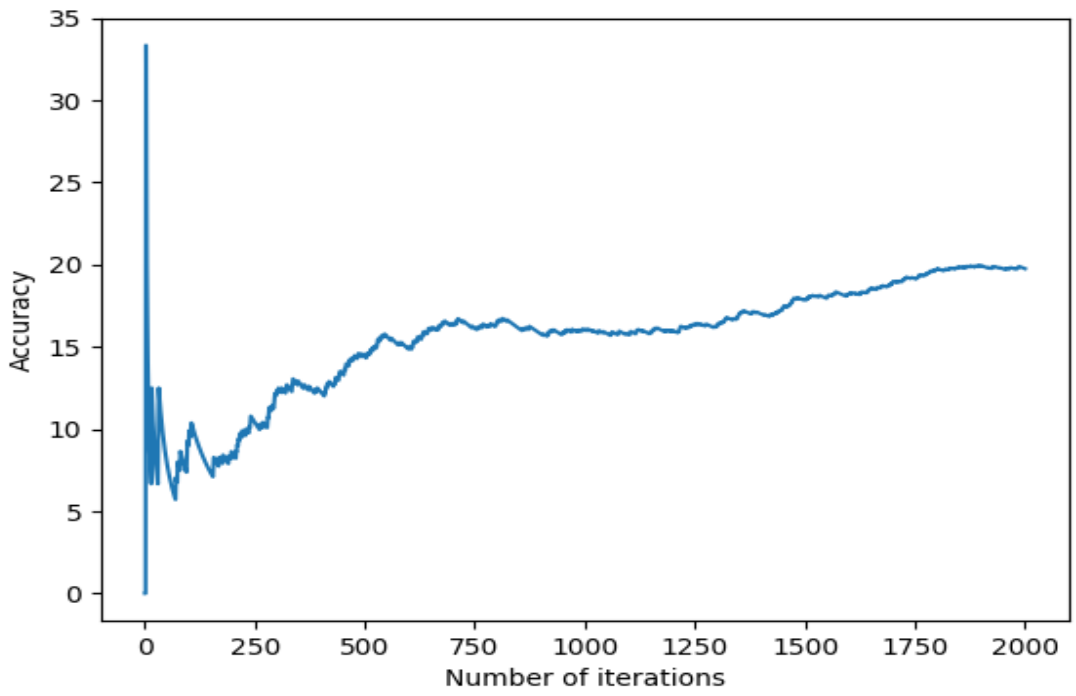


Q-learning vs MC10 (ROWS=3):

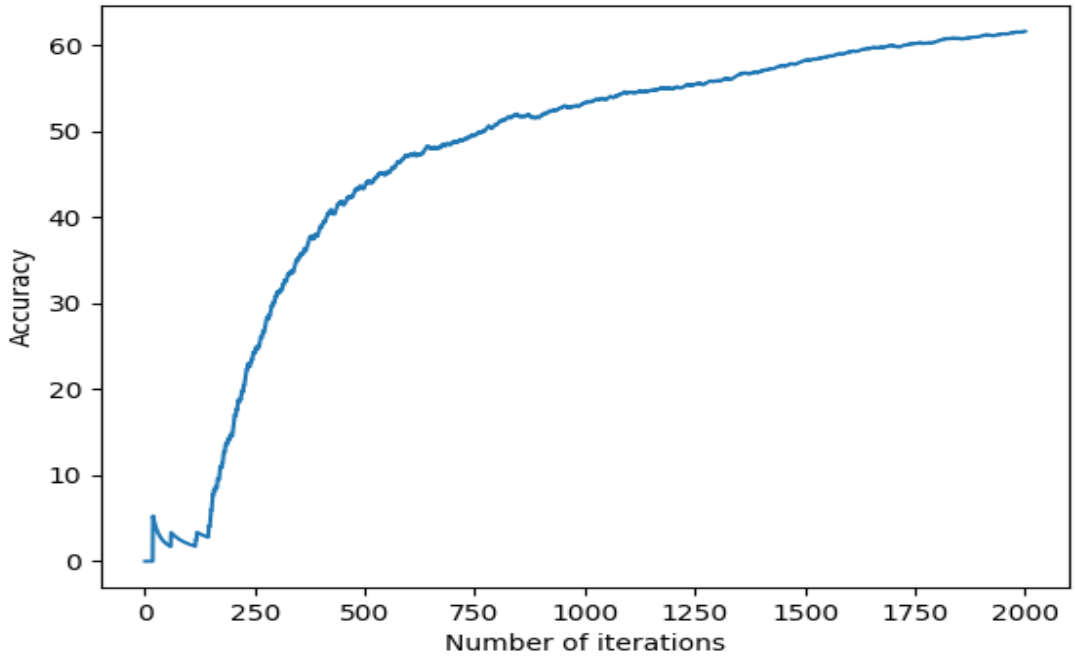


Note - In MC10 (ROWS=3), the q learning algorithm could not get many wins, but it achieved a large number of draws.

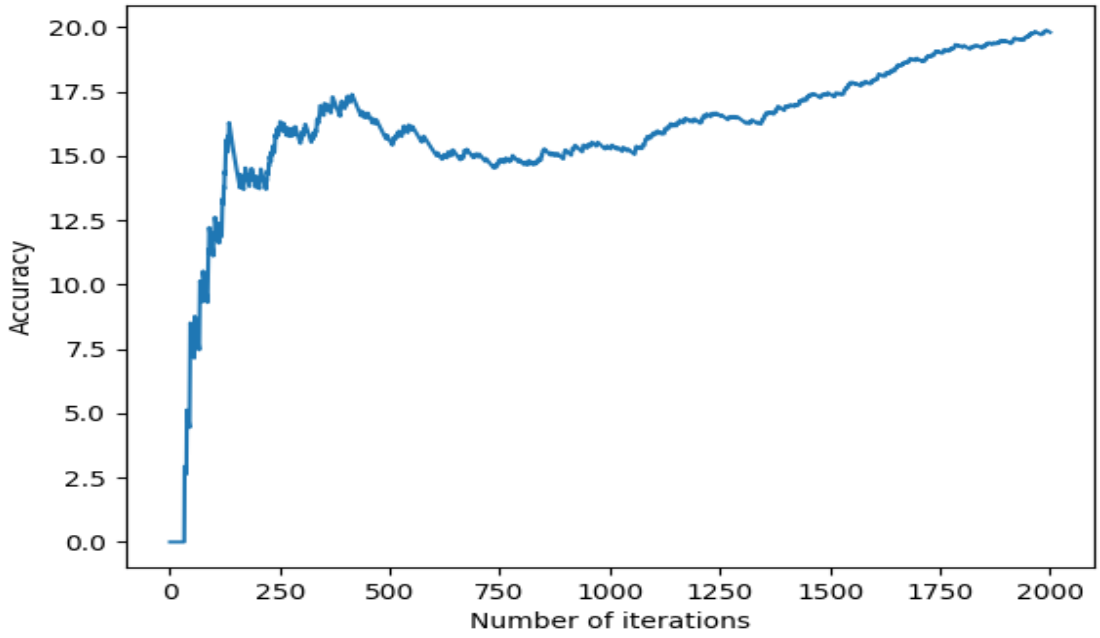
Q-learning vs MC10 (ROWS=4):



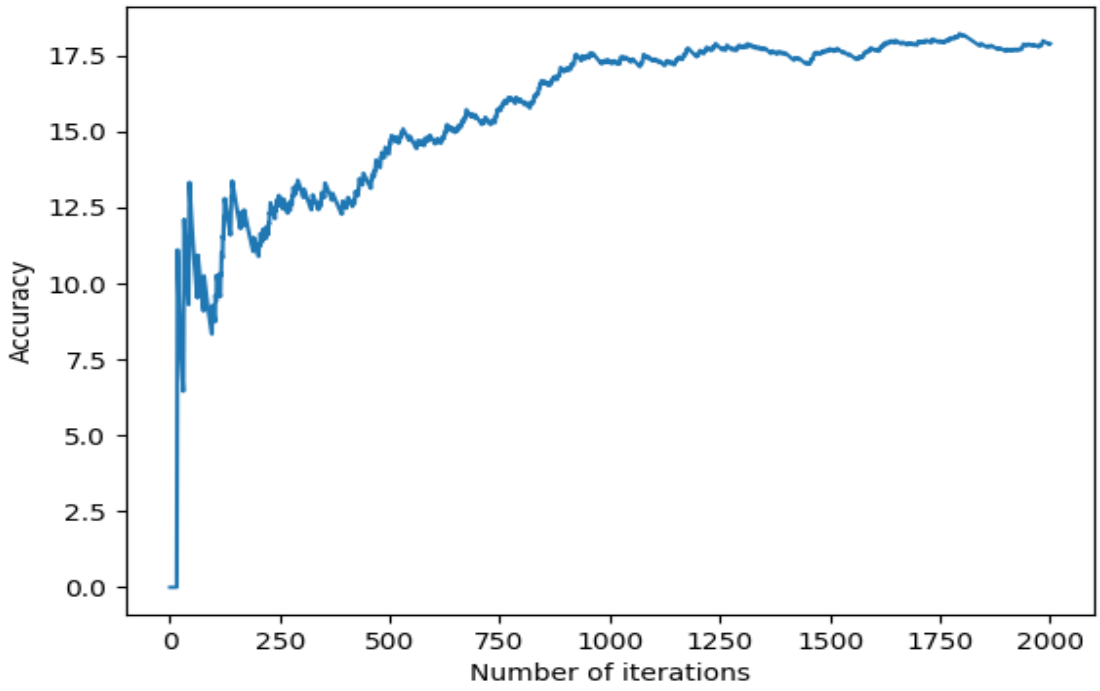
Q-learning vs MC20 (ROWS=2):



Q-learning vs MC20 (ROWS=3):



Q-learning vs MC20 (ROWS=4):

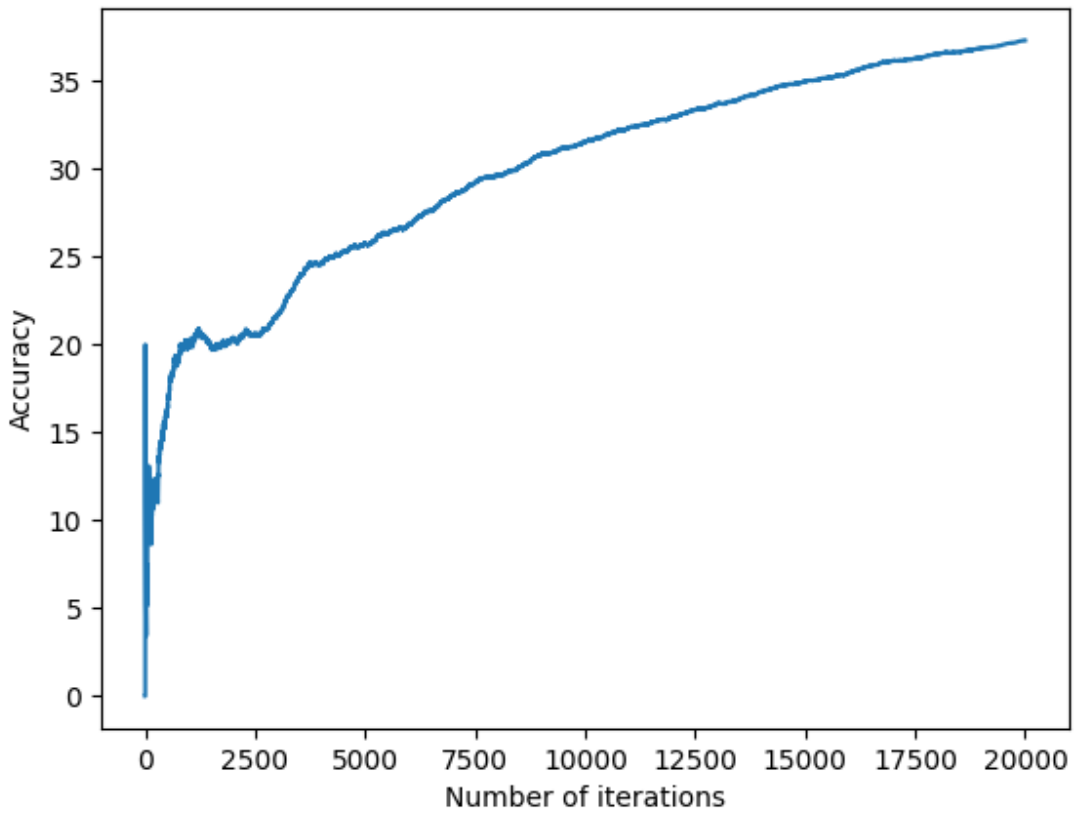


Note - Similar to MC10 (ROWS=3), against MC10(ROWS=4) and MC20(ROWS=3) also, the q-learning algorithm achieved high draw percentage despite having low win percentage. On the other hand, the q-learning algorithm could not compete with MC20(ROWS=4), and achieved very low draw as well as win percentages.

Conclusion: We can conclude that Q-learning algorithm converges faster when the number of rows is ferwer. As the number of rows increases, the rate of convergence slows down. Similarly, the rate of convergence is smaller against stronger opponents (i.e. MCn algorithms with higher valuse of n).

Part C - (Final Training)

To make sure that the Q-learning algorithm works well against a range of MCn algorithms, it is trained by playing against an MCn opponent where n is selected randomly from 0 to 25. The number of games required to be played using this approach to train the q-learning algorithm against all of these algorithms is very high.



As we can see, the accuracy of the q-learning algorithm keeps increasing even after 20000 games.

The q learning algorithm in the code has been trained this way, hence it has an overall accuracy of around 35% against all MCn algorithms. It can be trained for a longer time against these algorithms for better accuracy.

Conclusion - The Q-learning algorithm can converge against MCn algorithms, but it will take a very long time and a large number of games to do so.