BITS Pilani
K K Birla Goa Campus

Name: Shikha Bhat

ID: 2019A7PS0063G

Artificial Intelligence (CS F407) Assignment 2

1 Introduction

In this study, we design the game of Connect-4, implement 2 reinforcement learning agents and observe the results as they play against each other with varying parameters. Reinforcement algorithms have an agent that finds itself in various situations, or **states**. The agent performs an **action** to go to the next state. A **policy** is the strategy of choosing which action to take, given a state, in expectation of better outcomes. After the transition, the agent receives a **reward** (positive or negative) in return. This is how the agent learns.

The game of Connect 4 has a special property - there is an exponential increase in the number of possible actions that can be played. To combat this we have used 2 methods of reinforcement learning, namely Monte Carlo Tree Search (MCTS) and Q-Learning with the concept of afterstates. Monte Carlo Tree Search constructs a 'game tree' and performs a number of simulations to predict the next move that should be taken by the player. Q-Learning estimates values for each state of the game and learns this value (Q-value) by convergence after training through many episodes. Then the state with the best Q value is selected. Let us study these in detail in the next sections.

2 Game Implementation

A smaller version (5 columns × 6 rows) of Connect 4 game has been implemented using a class **'Board'.** (*Table 1*) The class describes the current state of the board as a 2D grid. The 'state' is represented as a **board object** in MCTS, and as a **2D integer matrix** in Q-learning. On their turn, a '**Player**' object (*Table 2*) evaluates all the empty positions available in the board (actions) using the method **getEmptyPositions()**, the MCTS or Q-Learning algorithm helps them decide which move to play, and then the **play()** method is called to execute the move, and the next state of the board is returned. After every move, the **checkWinner()** method is called to evaluate if any player has won. The class and method descriptions are given below.

Class Name: Board **Attributes: int rows, int columns, int board[][][]**			
Method Name	**Input**	**Returns**	**Description**
play()	playerNo, position	board, col	executes move, enters playerNo in the position given and returns the next state of the board and the column of the chip played.
getEmptyPositions()		array of positions	evaluates board and finds possible moves for next player
checkWinner()		-1: no one has won yet 0: board complete, no winner (draw) 1: player1 has won 2: player2 has won	evaluates the board and checks if there is a row, column or diagonal with 4 chips of the same player, and returns accordingly.
deepcopy()		board	Returns a new board object with copied attributes
printBoard			Prints the current board.

Table 1. Class description of the Board class.

Class Name: Player **Attributes: int playerNo, int playouts, Player opponent**

Table 2. Class description of the Player class.

3 Monte Carlo Tree Search Algorithm

The Monte Carlo Tree Search algorithm is a popular algorithm that figures out the best move out of a set of moves for a player. We construct a **game tree** with a root node, and then we keep expanding it using simulations. **Each node represents the state of the board in the game**. (*Table 4*) **The edges between the nodes represent the actions taken to reach that state**. In the process, we maintain **visit count** and **win count** for each node. This method is repeated for a **number of playouts** and then we select the node with the most promising value for the next move. There are 4 main steps to an MCTS algorithm. (*Figure 1*) (*Table 3*)

Step 1: Selection - We use tree policy to construct a path from root to the most promising leaf node. Each node has an associated value and during selection, we always chose the child node with the highest value.

Step 2: Expansion - We randomly pick an unexplored child of the leaf node.

Step 3: Simulation - We play out one or more simulations of the game after the expanded child node, until a winner is reached. The action selection policy here is random, such that it is fast to execute.

Step 4: Backpropagation - In this step, after we have a winner, we trace back the path we came through to reach the expanded child node, and update the visit and win values of all the nodes on this path.

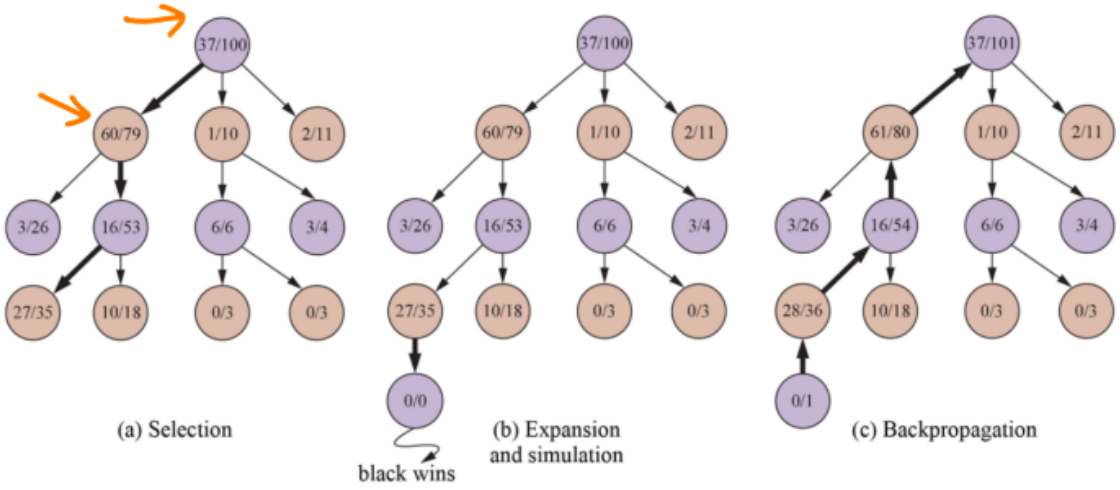

Figure 1. Demonstrating the MCTS Algorithm's game tree and the 4 different stages of the process.

Implementation

Two versions of Monte Carlo Tree Search (MCTS) algorithm have been implemented for playing the Connect 4 game. Version MC_{40} uses 40 simulations (playouts) before picking an action, and version MC_{200} uses 200 simulations before picking an action. The two versions maintain separate game trees and run the MCTS algorithm with their respective number of playouts to select the next move.

The state space is each **board object** generated when a player plays. Each player has their own separate state space. The action space is the **positions in the board (i, j)** that can be played. The value of each state is evaluated by the **number of wins/number of playouts** from that particular state.

The tree selection policy used is **Upper Confidence Bound Selection**, giving more preference to those actions whose values are uncertain.

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

The value of C is varied and its detailed comparison is done in the results section. For the value of each node, was used.

Pseudocode

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 2. The Monte Carlo tree search algorithm. A game tree, tree, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until the given number of playouts, and return the move that led to the node with the highest number of playouts.

Classes

Class Name: MonteCarloTreeSearch **Attributes: Player player, int playouts, Node tree**			
Method Name	**Input**	**Returns**	**Description**
select()		leaf node	Repeatedly applies the UCT method on the tree, and returns the leaf node reached.
uct()	node	child node	Evaluates all child nodes and returns the child node with the maximum ucb value.
expand()	leaf node	node to simulate from	Evaluates all possible actions from the given state and creates child nodes for all of them. Returns one of those created nodes to simulate further in the playouts.
simulate()	node to simulate from	winner playerNo	Simulates a playout from the given state until someone wins. Returns the winner.
back_propagate()	winner, node to simulate from		Updates the number of playouts and wins in the game tree, which changes the values of the nodes for the next playout.
nextState()	board	nextnode	Evaluates the best action for the current player using the MCTS algorithm and returns the next node selected in the game tree.

Table 3. Class Description of the MonteCarloTreeSearch class.

Class Name: Node **Attributes: Board board, Player player, int col, Node parent, Node children[], int playouts, int wins**			
Method Name	**Input**	**Returns**	**Description**
deepcopy()		node	Makes new copies of all attributes of the current node and returns a new node object with these copied attributes

Table 4. Class Description of the Node class.

Results

The two versions of MCTS played against each other for 100 games with each version being the first player (Player 1) for 50 games, and c = sqrt(2). (*Figure 3*) (*Figure 4*) (*Table 5*)

Game	MC_{40} wins	MC_{200} wins	Draws
MC_{40} vs MC_{200}	12 (24%)	36 (72%)	2 (4%)
MC_{200} vs MC_{40}	10 (20%)	40 (80%)	0 (0%)

Table 5. Game results for the MCTS algorithm

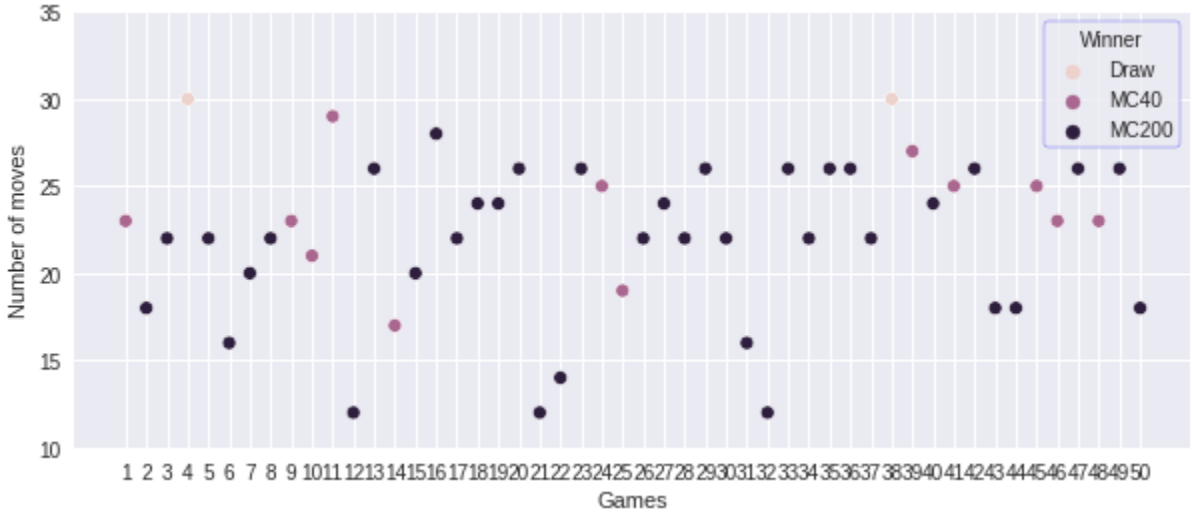

Figure 3. Game statistics of 50 games with MC_{40} as first player.

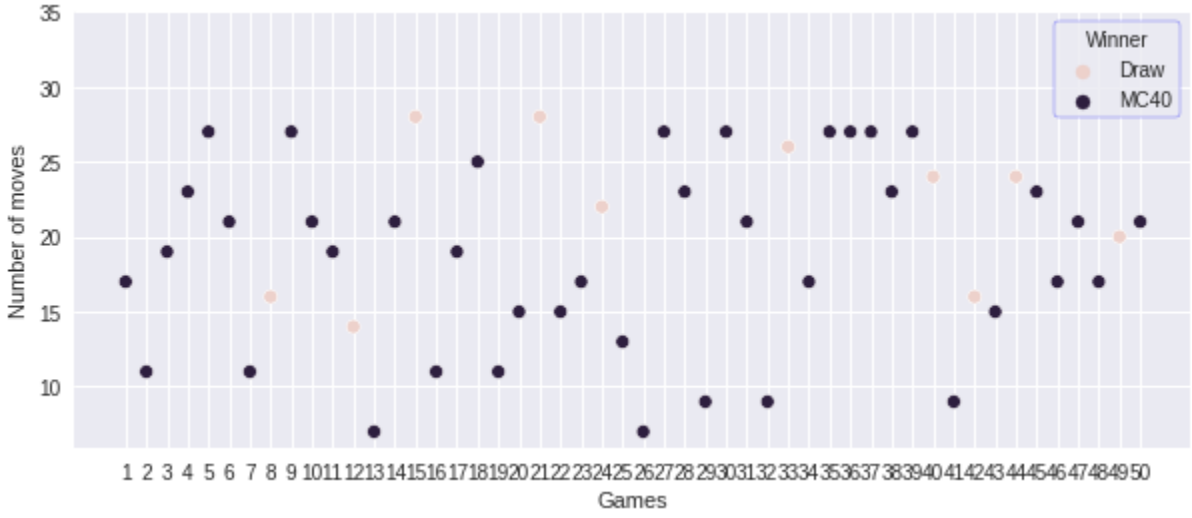

Figure 4. Game statistics of 50 games with MC_{200} as first player.

It can be seen that MC_{200} has a clear advantage over MC_{40}, and **more so when it is the first player.** This is because MC_{200} **uses more playouts and thus has more accurate values of number of wins/number of playouts for each node in its game tree**. Thus **it can select the next action with more confidence** than MC_{40} (*Figure 5*), and so it performs better. **Also, when the MC_{200} algorithm plays first, it tends to win more in a lesser number of moves.**

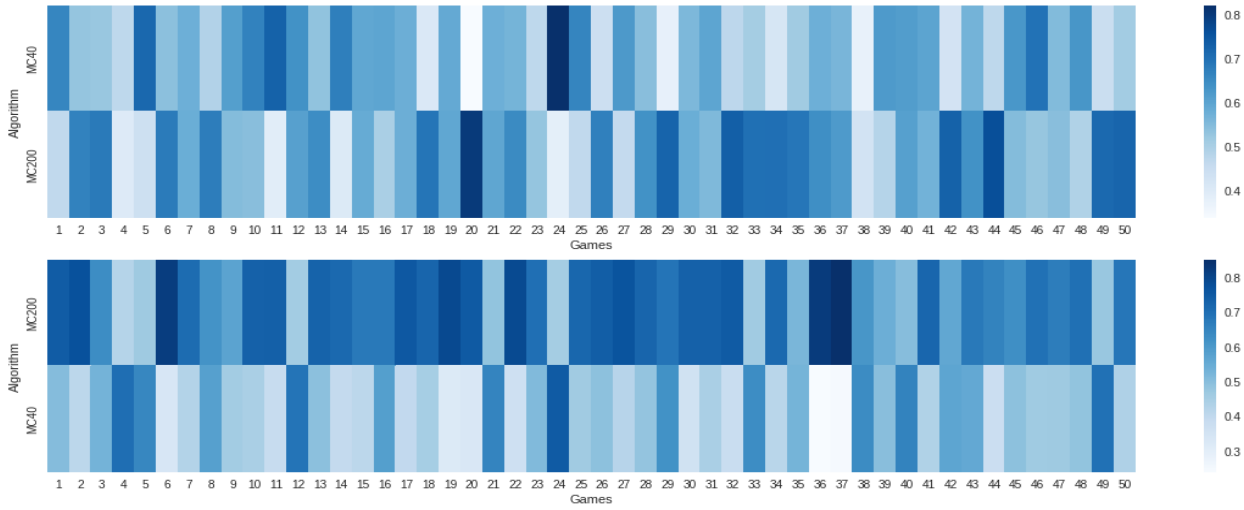

Figure 5. Average value of the next states as estimated by each algorithm over 100 games, with each version being the first player (Player 1) for 50 games. The MC_{200} algorithm has greater number of playouts, therefore more confidence for its next move for almost all games.

Choice of Parameter c

c is the **degree of exploration** in the UCB tree selection policy. We experiment with the choice of parameter c and how it affects the win percentage of MC_{200}. The value of parameter c in the MCTS algorithm was varied from 0 to 3 in steps of 0.2 over 100 games, with each version being the first player for 50 games. (*Figure 6*). It can be observed that **c = 2** is favourable for the MC_{200} algorithm, while **lesser values of c are less favourable** (except for 1 outlier at c=1), which indicates that **greater exploration is advantageous for the MC_{200} algorithm**. Given the large number of states possible in the game, this seems right. We also observe that being the second player reduces the advantage that the MC_{200} algorithm has.

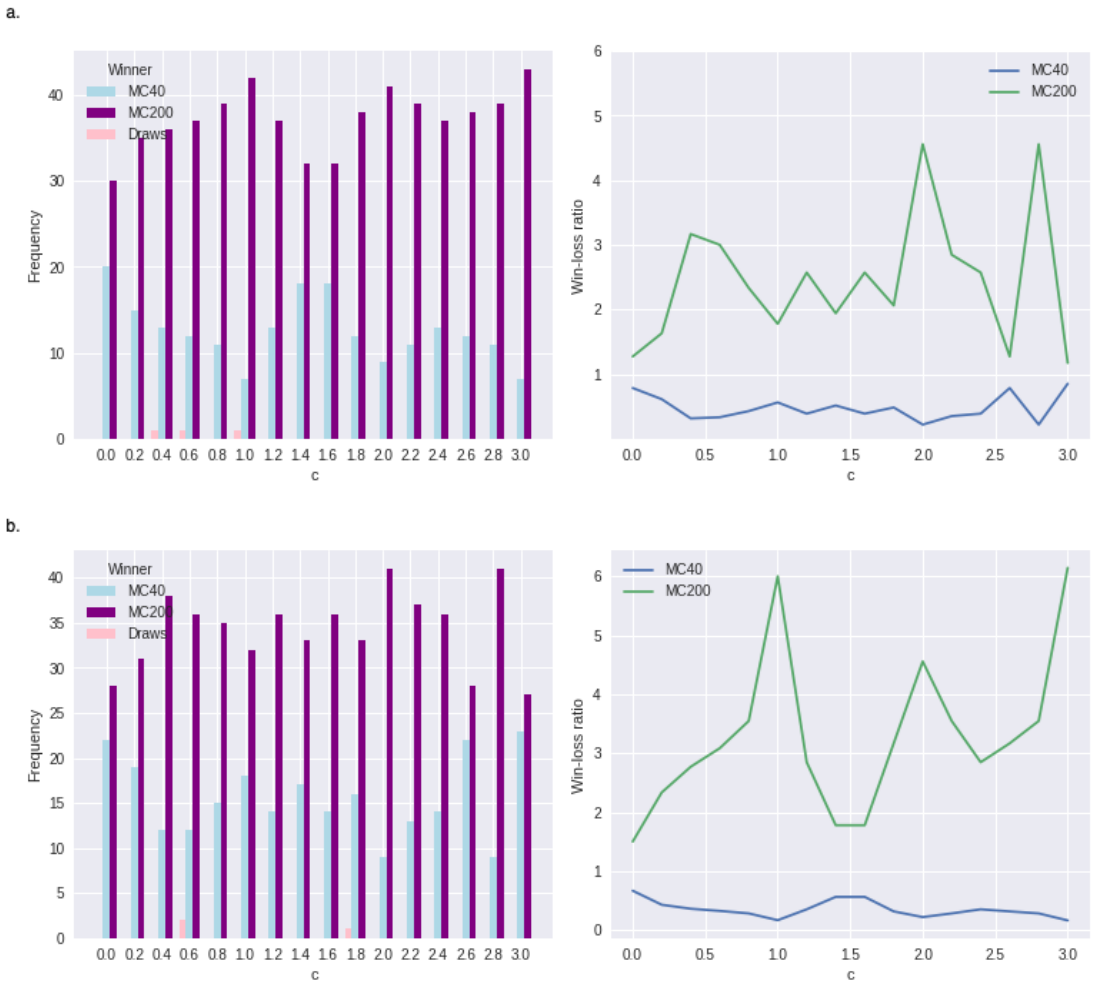

Figure 6. Summarizing the wins for each algorithm over varying values of C
(a) MC_{40} as first player. (b) MC_{200} as first player.

4 Q-Learning Algorithm

Q-Learning is a reinforcement learning algorithm that estimates values for each state of the game and learns this value (Q-value) by convergence after training through many episodes. Every time the Q-learning agent encounters a state visited before, it incrementally updates the action it previously took based on whether its previous action earned a positive or negative reward.

Implementation

The Q-learning algorithm has been implemented for playing the simplified Connect-4 game with the number of rows ranging from 2 to 4.

The state space is each **2D integer matrix board** generated when a player plays. The action space is the **positions in the board (i, j)** that can be played. The rewards are **+2 if the agent wins, -2 if the agent loses, +0.5 in case of a draw and 0 otherwise.**

Since there are many possible values of state-action pairs in a game like Connect-4, the concept of **afterstates** is used. This gives us a Q value for each state after a move and the **incremental updating** of the values is done by the following Q-Learning equation:

$$Q(s) = Q(s) + \alpha * (R(s) + \gamma * Q(s') - Q(s))$$

where Q is the Q function, s is the previous state, s' is the current state, α is the learning rate, R is the reward and γ is the discount factor.

The goal is to learn the **Q function,** a function that maps states to a value that represents the expected mean of the future rewards that each player will get after playing this action in the specific state. The playing policy is an **ε-greedy policy** which defines the chance to take a random move, or optimal move that maximizes the Q-value.

Initially, Q values of all states are set to **1,** to encourage exploration. The other parameters such as the learning rate (α), discount factor (γ) and ε (for ε-greedy action selection policy) are varied to see how they affect the win percentage of the Q-learning agent.

Pseudocode

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s,a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

Figure 7. Pseudocode for an exploratory Q-learning agent. In our implementation, we modified this using the concept of afterstates - only state values instead of state-action pairs.

Class Name: QLearning **Attributes: Player player, float epsilon, float alpha, float gamma, dict q**			
Method Name	**Input**	**Returns**	**Description**
getQ()		q-value	Fetches q-value of a state. If it is accessed for the first time, initializes the q value to 1 (to encourage exploration).
getReward()	state	2: player has won -2: opponent has won 0.5: draw 0: not a terminal state	Checks if the current state is a terminal state and returns reward values accordingly
nextAction()	state, actions	action to be taken (position to play chip in)	Chooses an action based on the ε-greedy policy.
updateTable()	prevState, q value of next state		Updates values in the q-table by using the q-learning equation.
nextState()	board	next state of board, column played	Calls appropriate functions to return the next state of the board after playing a move and updating the q-table.

Table 6. Class Description of the QLearning class

Training

The Q-learning algorithm is trained by allowing it to play the game against the MC_{25} algorithm for each value of rows [2, 4]. MC_{25} is always Player 1 and Q-learning is always Player 2, as this reduces the number of values that Q-learning needs to estimate, and trains it more rigorously. The value of **C is kept 2**, conferring with the results found in the previous section. The parameters for the Q-learning algorithm are varied to examine which values help the algorithm converge to an optimal value faster. For the Q-learning algorithm to converge, it is important that α approaches 0 at infinity. Thus, α is reduced by 50% and ε increased by 0.01 in every update. The difference in the values after updating are very small, so we devise another method for training. MCTS is tough to beat, so for checking the speed of the algorithm with each parameter, convergence is said to be reached when the **Q-learning algorithm wins at least 100 games against MC_{25}.**

Results

For each value of rows [2, 4], the Q learning algorithm converged with the number of games shown in *Figure 8*. From these graphs, we can observe the optimal values of each parameter, and how long it took to converge. These parameter values shown in *Table 7* give us the fastest convergence because Connect-4 is heavily dependent on future rewards, given the large number of states that can be formed. We shall use these values found in the next section.

Number of Rows	Exploration Factor (Epsilon)	Learning Rate (Alpha)	Discount Factor (Gamma)
2	0.6	0.4	0.3
3	0.7	0.1	0.9
4	0.4	0.5	1

Table 7. Optimal values of parameters for fast convergence

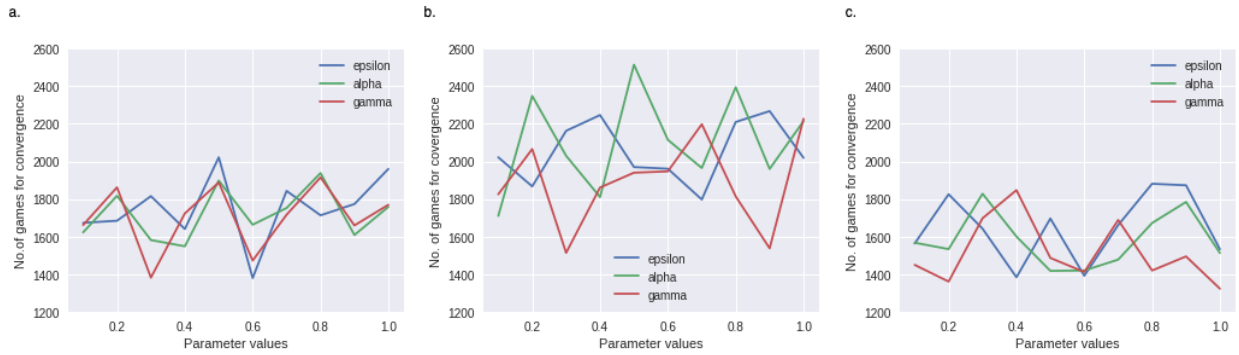

Figure 8. Plotting the number of games taken for the Q Learning agent to converge as the parameters exploration factor (ε), learning rate (α) and discount factor (γ) are varied.

(a) no.of rows = 2 (b) no.of rows = 3 (c) no. of rows = 4

5 Q Learning vs MC_n

Training

The Q-learning algorithm is trained by allowing it to play the game against MC_n algorithm, where n is the number of playouts, which can vary between 0 and 25. MC_n was always Player 1 and Q-learning was always Player 2, as this reduces the number of values that Q-learning needs to estimate. The value of **C = 2**, conferring with the results found in the previous sections.

We want to make sure that the Q-learning algorithm is able to beat the MC_n algorithm and that it should work well in the whole range of n = [0, 25]. Thus we train the Q-learning against the MC_n algorithm, **with different values of n in each game**. The difference in the values after updating are very small, so we devise another method for training. **Convergence is said to be reached when the Q-learning algorithm wins at least 10000 games against MC_n. The values of the states have been attached in a .dat file.** Using the results from the previous section, the best values of each parameter are selected for optimal performance and faster convergence. For the Q-learning algorithm to converge, it is important that α approaches 0 at infinity, and ε increases gradually. Thus, α is reduced by 50% and ε increased by 0.01 in every update. **The γ value helps reduce the number of moves that the Q-learning agent will take to reach a winning state.**

Results

The Q-learning agent played 500 games against the MC_n algorithm, **where n was randomly selected from the range [0, 25] in each game.** The results are shown in *Table 8*.

Number of Rows	Exploration Factor (Epsilon)	Learning Rate (Alpha)	Discount Factor (Gamma)	MC_n wins	Q learning wins	Draws
2	0.6	0.4	0.3	95 (19%)	34(6%)	371 (74%)
3	0.7	0.1	0.9	247 (49%)	56 (11%)	197 (39%)
4	0.4	0.5	1	385 (77%)	95 (19%)	20 (4%)

Table 8. Wins of Q-learning and MC_n over 500 games, with different values of n in each game

6 Conclusion

The MCTS agent takes a lot more playouts to learn than the Q learning agent, but the learning is a lot faster, as the Q learning agent takes very long for convergence. Connect 4 has a large number of possible states, which means that probably every game ever played is unique. We see that as the number of rows increase, the number of states increase and then the MCTS algorithm starts performing much better than the Q-learning algorithm, due to the convergence factor.

Both the agents learn to generalize and play well. The Q learning agent learns to generalize by learning similarities between different states and the MCTS agent learns to take the best path statistically. Both of these are promising methods for gameplay and are widely used in RL.

7 Code

The code is attached in 2019A7PS0063G_SHIKHA.py. All the classes described above are in the program, as well as the training method used to train the Q learning agent. There are separate methods for Part A and C, which have been called through the main function. In Part C, the data from the 2019A7PS0063G_SHIKHA.dat.gz file is unzipped and read. The file contains estimates of values of 81,371 states of the board. Each state is represented as a tuple of tuples with the player chip number 1, 2 or 0 (if not filled yet) in the position (i,j). These values are put into the Q-table dictionary of the Q-learning algorithm.