AI ASSIGNMENT 2

MILAN MUKHERJEE 22303417

The 6 files implements q_learning vs random, minimax vs random and q_learning vs minimax for TicTacToe and Connect4 game.

Random Player:
This game is implemented using python.
The default player is a semi-intelligent player which puts a winning move if there is any. In case there is no winning move,it puts a blocking move(which will prevent the opponent from immediate win) if there is any. Otherwise it makes a random move.

Design Choice:
Minimax algorithm:
Before each step in the game, the algorithm goes through every possible step.
Also, alpha-beta pruning has been implemented which helps to reduce the states visited by the algorithm significantly and consumes less time.

Q_learning:

1) Here, it has been assumed that the Q learning algorithm is playing with opponent which is part of the environment. So, after q_learning takes a step, the intermediate state where opponent takes an action is ignored. The next state is where q_learning again takes a step. For example:
   For tictactoe, if the board_state is denoted as an string where each character denotes the value of element in the board at that position.

X	o	0
0	0	0
0	0	0

Let's say there is a board with above state which is denoted as "Xo0000000" . Qlearning plays first and put 'X', opponent plays next and put 'o' . Next it is Q learning's turn and it puts 'X' in the 3rd position. So now the state is: "XoX000000". Then again the opponent takes a move and puts "o" at 4th position. So now the state is: "XoXo00000"

So, according to the algorithm implemented, the
Q_learning agent considers one state as: "xo0000000". Then it takes an action and opponent(part of the environment also takes an action) then next state is : "XoXo00000"

Also, it is easy to determine the current player since, when q learning plays first with "x" odd number of "o" would denote it is opponent's turn and even number of "o" would denote it is q_learning's turn.
Also, if the q-learning agent wants to change position and play in the opponent's position, it needs to be trained accordingly. But it is to note that, the states while playing first and playing second would never overlap.

For ease of understanding current player could have been added to the state but it is not mandatory.

2) In Connect4, to make the algorithm converge faster, for each game, depending on the game's result, all the previous states are updated simultaneously once each game finishes. So, to implement this, a history is needed to be maintained which just stores all the state details for a particular game. Once the game is finished, the q_learning agent backtracks all the moves of the agent and updates the Q value accordingly. It makes the convergence faster.

3) For each step, depending on the value of epsilon, the algorithm either explores or exploits. It has been ensured in the code that the algorithm always tries to choose least visited state during exploration.

Analysis:

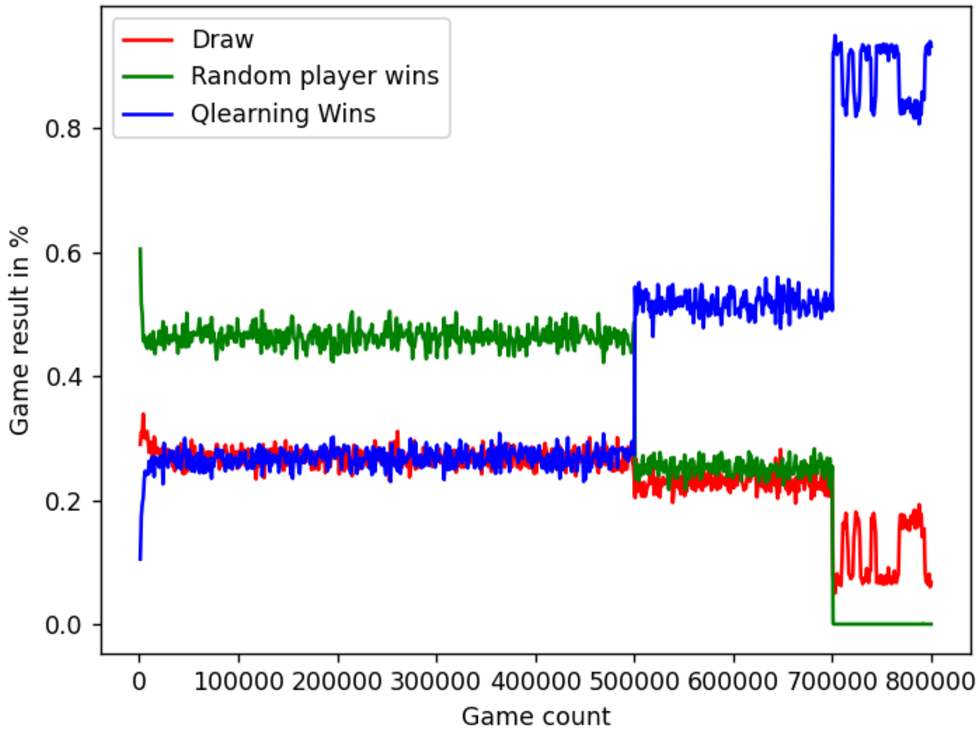
<u>Q-learning vs Random semi intelligent player:</u>

It can be observed in the below two figures that accuracy of Q_learning agent against random player is more than 90%
It is to note that this accuracy is achieved against the semi-intelligent random player which has been mentioned earlier.
Against completely random player win percentage is almost 100%

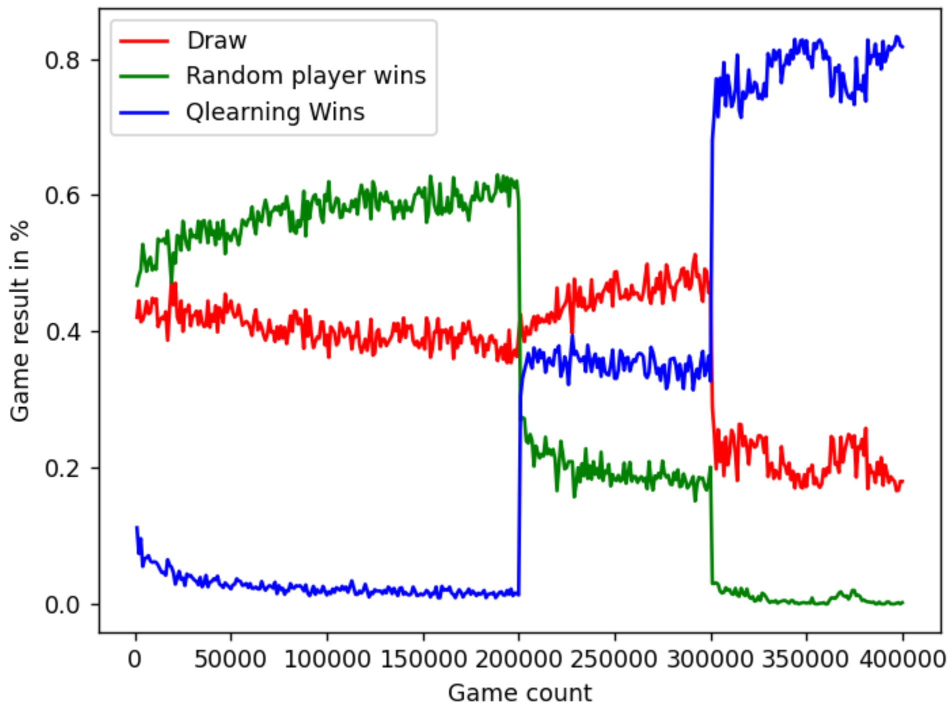
There is a sudden increase in win percentage two times in the below two graphs. It is because initially the epsilon value was set high(0.9) to let the agent explore as many states as possible and reduce the number of surprise states(previously unexplored). It has been performed because it is noticed that the algorithm performs really bad if there is a surprise state in the game.

After a certain iteration, the epsilon value is reduced to 0.2 which gives the first boost in win percentage.
Secondly, after another few iterations, the epsilon value is set as zero which means the algorithm only exploits.

Furthermore, it was observed that change alpha provides the best result at 0.3 value. Increasing alpha means more weight to edge cases which is not good for algorithm training. Discount was set to 0.9 . Decreasing discount more would mean less weight to the (state,action) which may lead to good results but are away from the final state.



Game = TictacToe (multiply with 100 for percentage value)

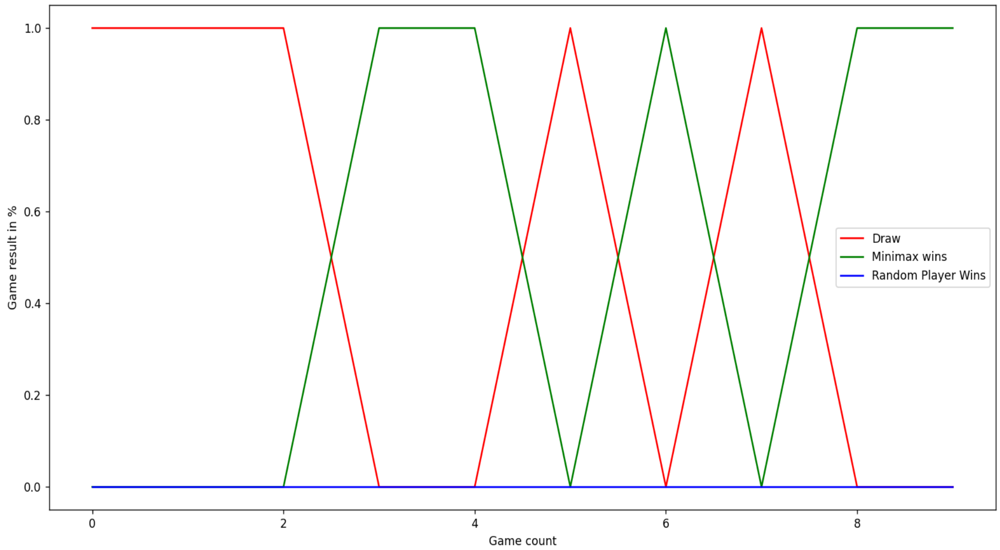


Game = Connect4 (multiply with 100 for percentage value)

Minimax vs Random semi intelligent Player both games:

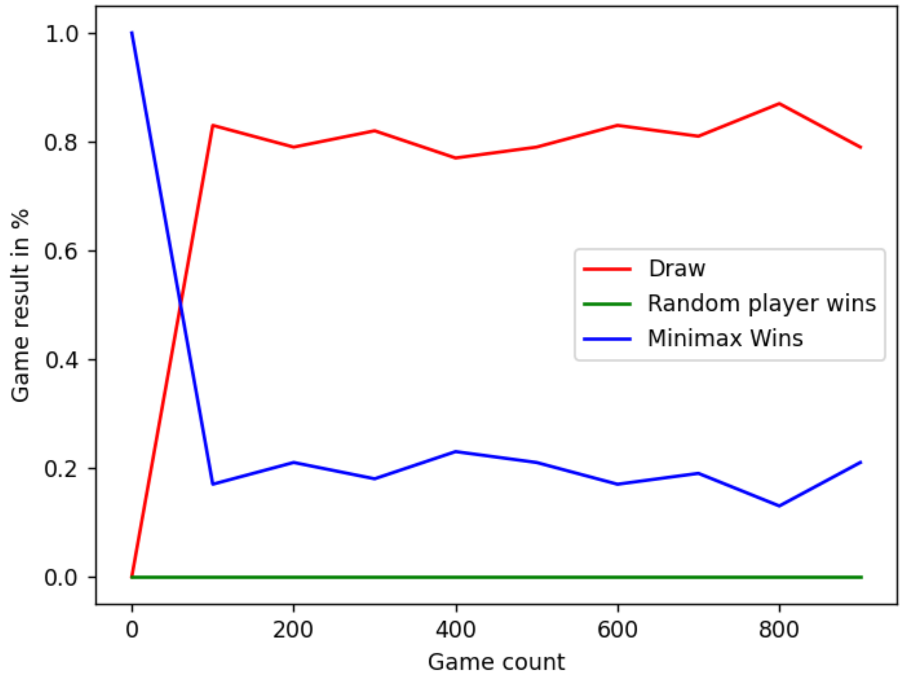
Minimax wins in most of the cases while playing against random players in Connect4. It also draws in a few cases. For a complete game, the algorithm visits 311127 states among which 100606 states are with maxValue.

For further running the algorithm, depth is needed to be set. There is not any heuristic implemented for this algorithm in the code. But one possible heuristic function is to calculate the total number of streaks or continuous similar numbers horizontally,vertically and diagonally on the board of connect4.



Game = Connect4, 10 games (multiply with 100 for percentage value)

It can be observed that both for connect4 and TicTacToe the algorithm leads to optimal solutions. It can either end up in win state or in draw state. It will never lose. This is because the algorithm visits all the possible states to find the best optimal path.



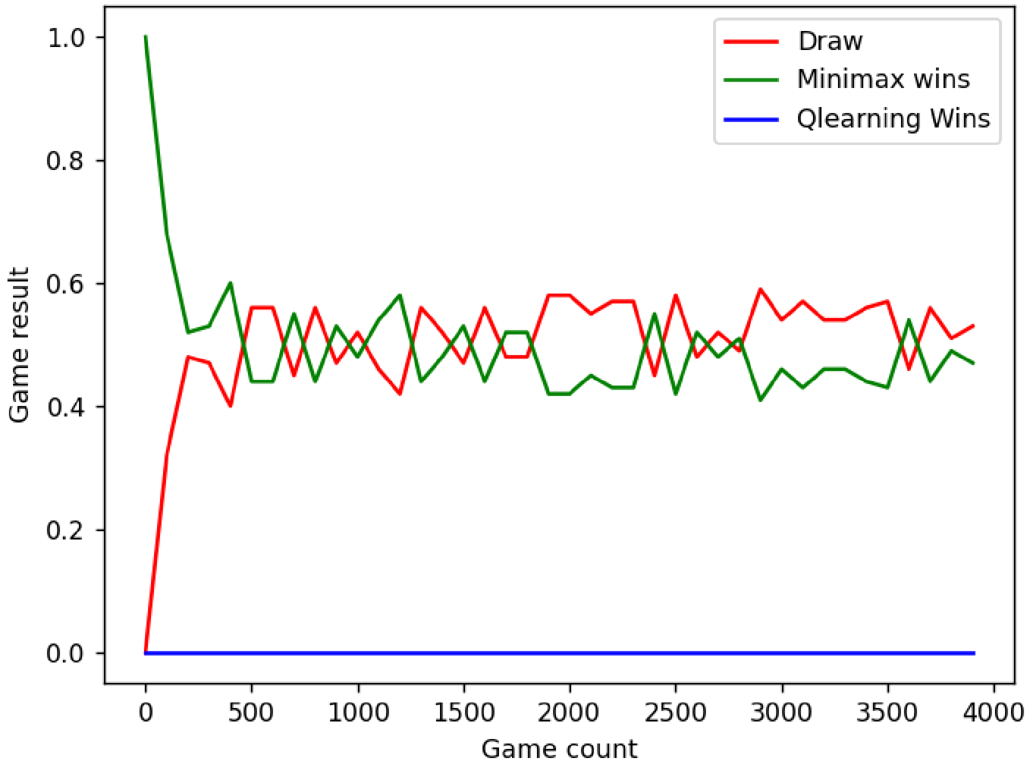
Game = Tictactoe ((multiply with 100 for percentage value)

Q-learning vs Minimax Tictactoe:

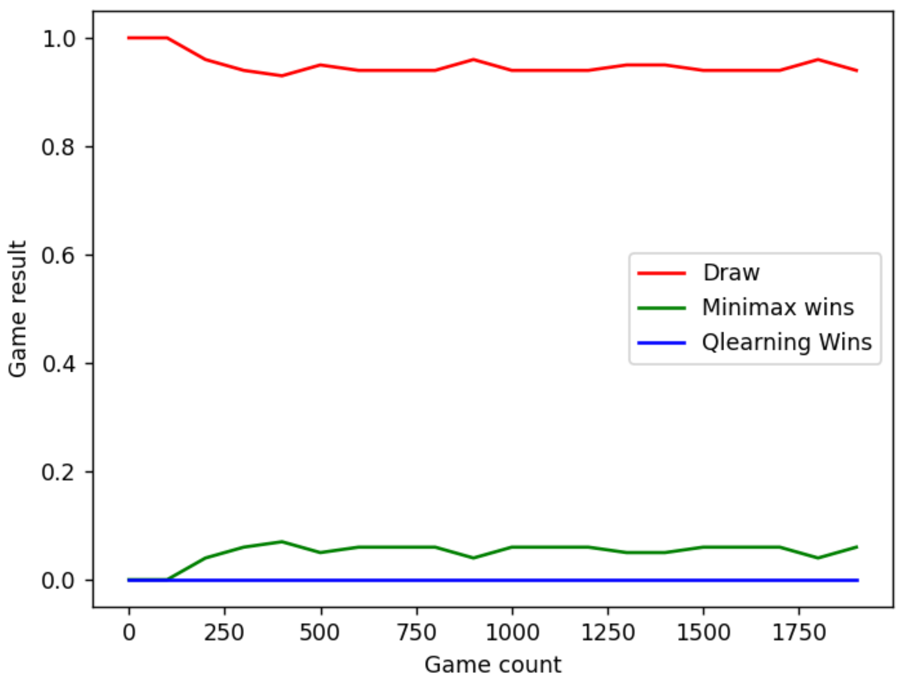
Q without training:
Next, TicTacToe was played between Q-learning agent vs Minimax agent. Initially, the Q is assigned to null and it is trained as the games are played. It is observed that even though Q_learning agent q_learning plays first, it can never win. It either draw the game or lose. Epsilon value was set as 0.3 Increasing it would result in better unbiased training for Q_agent but it would also take significant time to reach optimum values.
<u>Reason behind so many draw:</u>
1)Q learning plays first move
2) it plays in a certain algorithm
3) also since minimax tries to play optimum path for each state
Due to the above reasons, after a few games most of the relevant Q values are already trained. Next games almost follow the pattern. (Unlike Qlearning vs semi intelligent random matches where random can put any random move)

Q with training:

On the other hand, if Q_learning agent vs minimax game is played with the Q values which were trained earlier (for Qlearning vs Random case) with 800000 iteration, Q_agent performs clearly much better than previous case. All the matches are drawn and Q_agent successfully prevents minimax from winning. It is to note that here epsilon is set to zero which means the agent only exploits.



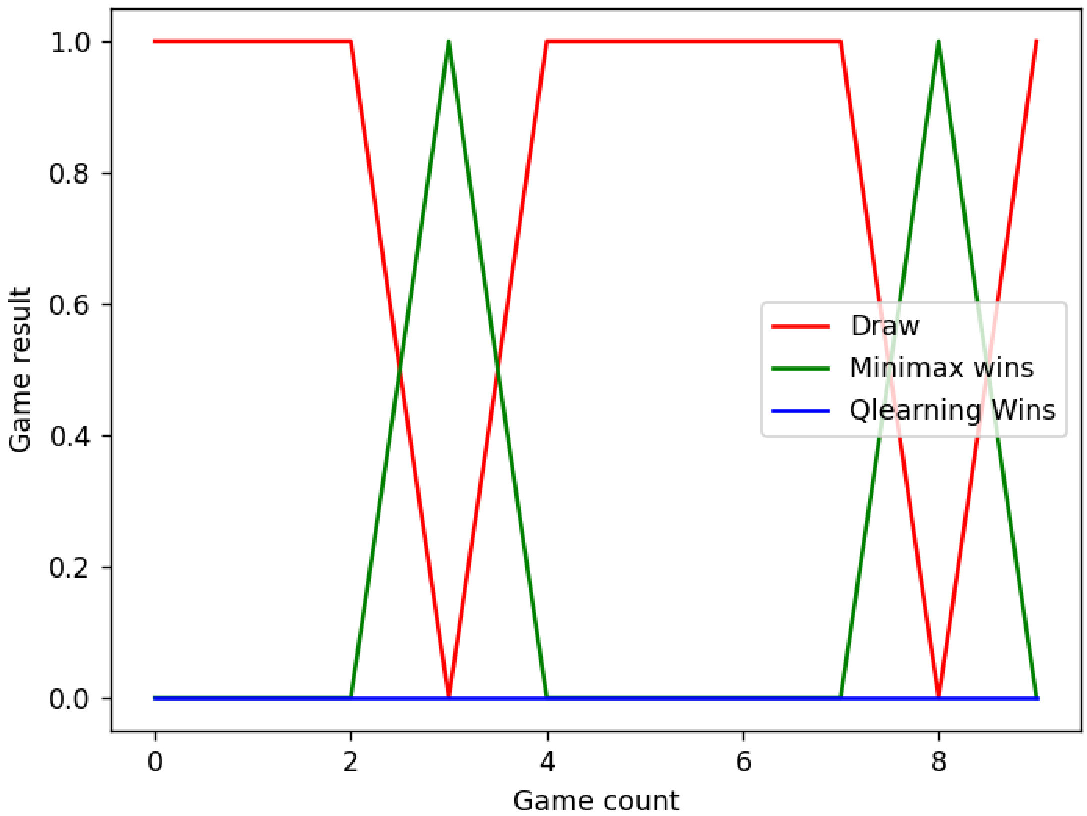
Game = TicTacToe without Q trained



Game = TicTacToe with Q trained(epsilon 0)

Q-learning vs Minimax Connect4:
If Connect4 is played between trained Q-learning agent and minimax player, minimax either wins or draws. As expected, q learning agent never wins since minimax always play optimal move.

For all the above three cases q_learning agent was played first and minimax agent was played second. Win rate for minimax would increase if opposite sequence was followed since minimax would have gained beginner's advantage.



Game = Connect4 (Q trained)

Overall, it is observed that:

Minimax is time consuming but more optimal
Q learning agent takes less time once trained. But not 100% optimal.

Also, alpha-beta pruning reduces the number of the states visited by the miimax algorithm significantly. As the number of states increases, it is hard to run Minimax without depth consideration. In this implementation, it is preferred to reduce the board size rather than sacrificing the minimax agent's optimality.

Also, if the number of states increases, it is better to use trained q_learning rather than minimax agent with depth consideration unless a really effective heuristic function is found.

A few inspiration was taken from the internet online during designing the connect4 board and tictactoe board.

Appendix:
Minimax_tictac:
```
import random

def PrintGame(game):
    for i in range(len(game)):
        if game[i] == 0:
            value = "_"
        else:
            value = game[i]
        print(value, end=" ")
        if((i+1)%3 == 0):
            print("\n")

def check_result(game):
    for sequence in winning_sequence():
        if(game[sequence[0]] != 0 and game[sequence[0]] ==
game[sequence[1]] and game[sequence[1]]==game[sequence[2]]):
            if game[sequence[0]] == "x":
                return "minimax_wins"
            else:
                return "random_player_wins"
    if 0 in game:
        return "no_winner_yet"
    return "draw"

def minimax_algo(game, alpha, beta, maxmin):
    global total_state,total_max
    total_state = total_state+1

    result = check_result(game)
    if(result == "draw"):
        return 0
    elif(result == "minimax_wins"):
        return 100
    elif(result == "random_player_wins"):
        return -100
    if maxmin == True:
        total_max= total_max+1
        maxValue = float('-inf')
```

```
        for square_index in range(len(game)):
            if(game[square_index] == 0):
                game[square_index] = "x"
                result = minimax_algo(game, alpha, beta,False)
                game[square_index] = 0
                maxValue = max(maxValue,result)
                alpha = max(alpha,maxValue)
                if beta<=alpha:
                    break

        return maxValue
    else:
        minValue = float('inf')
        for square_index in range(len(game)):
            if(game[square_index] == 0):
                game[square_index] = "o"
                result = minimax_algo(game,alpha,beta,True)
                game[square_index] = 0
                minValue = min(minValue,result)
                beta = min(minValue,beta)
                if(beta <= alpha):
                    break
        return minValue

def minimax_play(game):
    max_result = float('-inf')
    index = -1
    for i in range(len(game)):
        if(game[i] == 0):
            game[i] = "x"
            result =
minimax_algo(game,float('-inf'),float('inf'),False)
            game[i] = 0
            if(result>max_result):
                max_result = result
                index = i

    game[index] = "x"
    return game

def winning_sequence():
    return
[[2,4,6],[0,4,8],[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8]]
```

```
def find_prob_loss(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"x" and game[sequence[index1]] == game[sequence[index2]])
def find_prob_win(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"o" and game[sequence[index1]] == game[sequence[index2]])
def random_play(game):
    for sequence in winning_sequence():
        if find_prob_win(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
            return sequence[2]
        elif find_prob_win(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_win(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]

    for sequence in winning_sequence():
        if find_prob_loss(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
            return sequence[2]
        elif find_prob_loss(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_loss(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]
    list1 = []
    for i in range(len(game)):
        if(game[i] == 0):
            list1.append(i)
    pos = random.randint(0,len(list1)-1)
    return list1[pos]

def user_play(game):
    position = input('input the box number you chose. range(1-9) \n')
    if game[int(position)-1] != 0:
        print("Wrong Move")
        exit(1)
    game[int(position)-1] = 'o'
    return game
```

```python
def start_game():
    global count_q,count_mini,count_draw
    game = [0] * 9
    play = 1

    result = "no_winner_yet"
    for square_index in range(len(game)):
        result = check_result(game)
        if(result != "no_winner_yet"):
            break;
        elif((square_index+play)%2 != 1):

            print("minimax turn")
            PrintGame(game)
            game = minimax_play(game)
        else:
            print("random play turn")
            PrintGame(game)
            #random_play(game)
            pos = random_play(game)
            game[pos] = "o"
    result = check_result(game)
    if(result == "minimax_wins"):
        count_q = count_q+1
        PrintGame(game)
        print("minimax_wins")
    elif(result == "random_player_wins"):
        count_mini = count_mini+1
        PrintGame(game)
        print("random_player_wins")
    elif(result == "draw"):
        count_draw = count_draw+1
        PrintGame(game)
        print("It's a draw")

count_mini = 0
count_q = 0
count_draw = 0
list_qwin = []
list_rwin = []
list_draw = []
list_count = []
total_state = 0
```

```
total_max = 0

for i in range(1000):
    start_game()
    if i%10 == 0 and count_q+count_mini != 0:
        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("q-learning = " +str(count_qi) )
        print("random = "+ str(count_minii))
        print("draw = "+ str(count_drawi))

import matplotlib.pyplot as plt
plt.ylabel('Game result in %')
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Random player wins')
plt.plot(list_count, list_qwin, 'b-', label='Minimax Wins')
plt.legend()
plt.show()
```

Qlearning_tictac:

```
import copy
import random

def PrintGame(game):
    for i in range(len(game)):
        if game[i] == 0:
            value = "_"
        else:
            value = game[i]
```

```python
        print(value, end=" ")
        if((i+1)%3 == 0):
            print("\n")
def PrintGame2(game):
    for i in range(len(game)):
        if game[i] == 0:
            value = "_"
        else:
            value = game[i]
        print(value, end=" ")
        if((i+1)%3 == 0):
            print("\n")

def check_result(game):
    for sequence in winning_sequence():
        if(game[sequence[0]] != 0 and game[sequence[0]] ==
game[sequence[1]] and game[sequence[1]]==game[sequence[2]]):
            if game[sequence[0]] == "x":
                return "qlearning_wins"
            else:
                return "random_player_wins"
    if 0 in game:
        return "no_winner_yet"
    return "draw"

def get_state(game):
    return "".join(str(value) for value in game )

def maxQ_value(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return Q[(state,final_action)]

def maxQ(state,game):
    global Q,N,reward,total_states , actions
```

```
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return final_action

def least_used_Q(state,game):
    global Q,N,reward,total_states, actions
    min_n = float('inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in N:
                N[(state,action)] = 0
                final_action = action
                break
            else:
                if N[(state,action)] < min_n:
                    min_n = N[(state,action)]
                    final_action = action

    return final_action



def explore_exploit(epsilon,action,state,game):
    global Q,N,reward,total_states
    if  random.uniform(0,1) < epsilon :
        action = least_used_Q(state,game)
        N[(state,action)] = N[(state,action)]+1
    return action

def
update_state(current_state,last_state,game,last_action,alpha,discount,e
psilon_first_step):
    global total_states
    total_states = total_states+1
    try:
```

```
        abcd  = (1-alpha)*Q[(last_state,last_action)] + alpha *
(N[(last_state,last_action)]/total_states)*
(discount*maxQ_value(current_state,game) - Q[(last_state,last_action)])
        if(Q[(last_state,last_action)] >5):

            Q[(last_state,last_action)] = abcd
        else:
            Q[(last_state,last_action)] = abcd
    except:
        breakpoint()
def
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step):
    global Q,N,reward,total_states
    if not (first_step ):

update_state(current_state,last_state,game,last_action,alpha,discount,e
psilon)

    action = maxQ(current_state,game)
    if (current_state,action) not in N:
        N[(current_state,action)] = 1

    else:
        N[((current_state,action))] = N[((current_state,action))]+1
    action = explore_exploit(epsilon,action,current_state,game)
    game[action] = "x"
    return action

def winning_sequence():
    return
[[2,4,6],[0,4,8],[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8]]
def find_prob_loss(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"x" and game[sequence[index1]] == game[sequence[index2]])
def find_prob_win(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"o" and game[sequence[index1]] == game[sequence[index2]])

def random_play(game):
    for sequence in winning_sequence():
        if find_prob_win(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
```

```
            return sequence[2]
        elif find_prob_win(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_win(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]

    for sequence in winning_sequence():
        if find_prob_loss(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
            return sequence[2]
        elif find_prob_loss(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_loss(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]
    list1 = []
    for i in range(len(game)):
        if(game[i] == 0):
            list1.append(i)
    pos = random.randint(0,len(list1)-1)
    return list1[pos]

def user_play(game):
    position = input('input the box number you chose. range(1-9) \n')
    if game[int(position)-1] != 0:
        print("Wrong Move")
        exit(1)
    game[int(position)-1] = 'o'
    return game

def start_game():
    global count_q,count_mini,count_draw, alpha, discount, epsilon
    game = [0] * 9
    #play = int(input('type 1 for minimax to play first, type 2 for
minimax to play second \n'))
    play = 1
    result = "no_winner_yet"
    current_state = get_state(game)
    action = 0
    first_step = True
```

```
    while check_result(game) == "no_winner_yet":
        if((play+1)%2 != 1):
            print("q-agent turn")
            PrintGame(game)

            last_state = current_state
            current_state = get_state(game)
            last_action = action
            result = check_result(game)
            last_game = copy.deepcopy(game)
            reward[get_state(game)] = 0
            action =
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step)
            first_step = False
            #last_game = copy.deepcopy(game)

            last_state = current_state
        else:
            print("random play turn")
            PrintGame(game)
            pos = random_play(game)
            game[pos] = "o"
            #game = user_play(game)
        play = (play+1)%2
    result = check_result(game)

    last_action = action
    if (result == "qlearning_wins"):
        Q[(last_state,action)]  = 100
        count_q = count_q+1
        PrintGame(game)
        #reward[state] = 200
    elif (result == "random_player_wins"):
        Q[(last_state,action)]  = -100
        count_mini = count_mini+1
        PrintGame(game)
        #reward[state] = -200
    elif(result == "draw"):
        count_draw = count_draw+1
        PrintGame(game)
        Q[(last_state,action)]  = 0
```

```
Q = {}
N = {}
reward = {}
total_states = 0
count_mini = 0
count_q = 0
count_draw = 0
epsilon = 0.4
alpha = 0.9
discount = 0.9
actions = [0,1,2,3,4,5,6,7,8]
list_qwin = []
list_rwin = []
list_draw = []
list_count = []
for i in range(1,800001):
    start_game()

    if i%10000 == 0 and count_q+count_mini != 0:
        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("q-learning = " +str(count_qi) )
        print("random = "+ str(count_minii))
        print("draw = "+ str(count_drawi))
    if(i >500000):
        epsilon = 0.2
    if(i>700000):
        epsilon = 0



import matplotlib.pyplot as plt
```

```
plt.ylabel('Game result in %')
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Random player wins')
plt.plot(list_count, list_qwin, 'b-', label='Qlearning Wins')
plt.legend()
plt.show()
```

Minimax Connect4:

```
import numpy as np
import random
import copy
count = 0

def minimax_algo(game,turn, alpha, beta, maxmin):
    global total_state,total_maxmove
    total_state = total_state+1
    global count
    op_turn = change_turn(turn)
    result = check_result(game)
    if(result != "no_winner_yet"):
        #count = count+1
        pass
    if(result == "draw"):
        return 0
    elif(result == "1st_player"):
        return 100
    elif(result == "2nd_player"):
        return -100
    if maxmin == True:
        total_maxmove = total_maxmove+1
        maxValue = float('-inf')
        for col in range(len(game)):
            if(game[rows-1][col] == 0):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, turn)

                result = minimax_algo(game,op_turn, alpha, beta,False)
```

```
                reverse_drop(game, row, col, turn)
                maxValue = max(maxValue,result)
                alpha = max(alpha,maxValue)
                if beta<=alpha:
                    break
        return maxValue
    else:
        minValue = float('inf')
        for col in range(len(game)):
            if(game[rows-1][col] == 0):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, turn)
                result = minimax_algo(game,op_turn,alpha,beta,True)
                reverse_drop(game, row, col, turn)
                minValue = min(minValue,result)
                beta = min(minValue,beta)
                if(beta <= alpha):
                    break
        return minValue


def minimax_play(game,turn):
    max_result = float('-inf')
    final_row = -1
    final_col = -1
    for i in range(len(game[0])):
        if(game[rows-1][i] == 0):
            row = get_next_open_row(game, i)
            drop_piece(game, row, i, turn)
            op_turn = change_turn(turn)
            result =
minimax_algo(game,op_turn,float('-inf'),float('inf'),False)
            reverse_drop(game, row, i, turn)
            if(result>max_result):
                max_result = result
                final_row = row
                final_col = i

    return final_col

import numpy as np
```

```
def create_game(rows,cols):
    game = np.zeros((rows, cols))
    return game

def drop_piece(game, row, col, piece):
    game[row][col] = piece
def reverse_drop(game, row, col, piece):
    game[row][col] = 0
def is_valid_location(game, col):
    try:
        return game[rows-1][col] == 0
    except:
        breakpoint()
def get_next_open_row(game, col):
    r = -1
    for r in range(rows):
        if game[r][col] == 0:
            return r



def print_game(game):
    print(np.flip(game, 0))

def winning_move(game, piece):
    # check horizontal locations
    for c in range(cols-3):
        for r in range(rows):
            if game[r][c] == piece and game[r][c+1] == piece and
game[r][c+2] == piece and game[r][c+3] == piece:
                return True

    # check vertical locations
    for c in range(cols):
        for r in range(rows-3):
            if game[r][c] == piece and game[r+1][c] == piece and
game[r+2][c] == piece and game[r+3][c] == piece:
                return True

    # check diagonal positive slope locations
    for c in range(cols-3):
        for r in range(rows-3):
            if game[r][c] == piece and game[r+1][c+1] == piece and
game[r+2][c+2] == piece and game[r+3][c+3] == piece:
```

```
                return True

    # check diagonal negative slope locations
    for c in range(cols-3):
        for r in range(rows-3, rows):
            if game[r][c] == piece and game[r-1][c+1] == piece and
game[r-2][c+2] == piece and game[r-3][c+3] == piece:
                return True

def check_result(game):

    if  winning_move(game, 1):
        return "1st_player"
    elif(winning_move(game, 2)):
        return "2nd_player"
    elif( np.all(game != 0)):
        return "draw"
    return "no_winner_yet"
def change_turn(turn):
    if turn == 1:
        turn = 2
    else:
        turn = 1
    return turn

def random_player(game, piece,op_piece):
    possible_moves = []
    for col in actions:
        if is_valid_location(game, col):
            row = get_next_open_row(game, col)
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, op_piece)
            if winning_move(test_game, op_piece):
                return col
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, piece)
            if winning_move(test_game, piece):
                return col
            possible_moves.append(col)
    if possible_moves:
        return random.choice(possible_moves)
```

```python
rows=4
cols=4
total_maxmove = 0
total_state = 0
def start_game():
    global count_q,count_mini,count_draw
    game = create_game(rows,cols)
    game_over = False
    turn = 0
    while check_result(game) == "no_winner_yet":
        # Ask for player input
        if turn == 0:
            print("minimax play")
            col = minimax_play(game,1)
            if is_valid_location(game, col):
                row = get_next_open_row(game, col)
                last_game = copy.deepcopy(game)
                drop_piece(game, row, col, 1)
                if winning_move(game, 1):
                    print("Player 1 wins!!!")
                    game_over = True
        else:
            print("random player play")
            col = random_player(game,2,1)
            while not (is_valid_location(game, col)):
                col = int(input("Player 2 make your selection (0-rows):
"))
            if is_valid_location(game, col):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, 2)

        print_game(game)
        turn += 1
        turn = turn % 2
    result = check_result(game)
    if (result == "1st_player"):
        count_mini = count_mini+1
        print("minimax wins")
        #reward[state] = 200
    elif (result == "2nd_player"):#
        print("random player wins")
        count_q = count_q+1
        #reward[state] = -200
```

```python
    elif(result == "draw"):
        count_draw = count_draw+1


    print("Thanks for playing Connectcols-3!")
actions = [*range(cols)]
count_mini = 0
count_q = 0
count_draw = 0
list_qwin = []
list_rwin = []
list_draw = []
list_count = []
for i in range(100):
    start_game()
    if i%10 == 0:
        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("minimax = " +str(count_qi) )
        print("random = "+ str(count_minii))
        print("draw = "+ str(count_drawi))



import matplotlib.pyplot as plt
plt.ylabel('Game result in %')
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Minimax wins')
plt.plot(list_count, list_qwin, 'b-', label='Random Player Wins')
plt.legend()
plt.show()
```

Qlearning Connect4:

```
import numpy as np
import random
count = 0
import copy

def get_state(game):
    state = ""
    for i in range(len(game)):
        for j in range(len(game[0])):
            state = state + str(int(game[i][j]))

    return state

def maxQ_value(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(is_valid_location(game, action)):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return Q[(state,final_action)]

def maxQ(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(is_valid_location(game, action)):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
```

```
                max_value = Q[(state,action)]
                final_action = action
    return final_action

def least_used_Q(state,game):
    global Q,N,reward,total_states, actions
    min_n = float('inf')
    final_action = -1
    for action in actions:

        if(is_valid_location(game, action)):
            if (state,action) not in N:
                N[(state,action)] = 0
                final_action = action
                break
            else:
                if N[(state,action)] < min_n:
                    min_n = N[(state,action)]
                    final_action = action

    return final_action



def explore_exploit(epsilon,action,state,game):
    global Q,N,reward,total_states
    if  random.uniform(0,1) < epsilon :
        action = least_used_Q(state,game)
        N[(state,action)] = N[(state,action)]+1
    return action

def
update_state(current_state,last_state,game,last_action,alpha,discount):
    global total_states
    total_states = total_states+1
    try:
        Q[(last_state,last_action)]  =
(1-alpha)*Q[(last_state,last_action)] + alpha *
(discount*maxQ_value(current_state,game) - Q[(last_state,last_action)])
    except:
        breakpoint()
```

```
def
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step,play):
    global Q,N,reward,total_states

    #if not (first_step ):


#update_state(current_state,last_state,game,last_action,alpha,discount,
epsilon)

    action = maxQ(current_state,game)
    if (current_state,action) not in N:
        N[(current_state,action)] = 1

    else:
        N[((current_state,action))] = N[((current_state,action))]+1
    col = explore_exploit(epsilon,action,current_state,game)
    row = get_next_open_row(game, col)
    drop_piece(game, row, col, play)
    return col


#connect4 board


import numpy as np


def create_game(rows,cols):
    game = np.zeros((rows, cols))
    return game

def drop_piece(game, row, col, piece):
    game[row][col] = piece
def reverse_drop(game, row, col, piece):
    game[row][col] = 0
def is_valid_location(game, col):
    try:
        return game[rows-1][col] == 0
    except:
        breakpoint()
def get_next_open_row(game, col):
```

```python
    for r in range(rows):
        if game[r][col] == 0:
            return r

def print_game(game):
    print(np.flip(game, 0))

def winning_move(game, piece):
    # check horizontal locations
    for c in range(cols-3):
        for r in range(rows):
            if game[r][c] == piece and game[r][c+1] == piece and
game[r][c+2] == piece and game[r][c+3] == piece:
                return True

    # check vertical locations
    for c in range(cols):
        for r in range(rows-3):
            if game[r][c] == piece and game[r+1][c] == piece and
game[r+2][c] == piece and game[r+3][c] == piece:
                return True

    # check diagonal positive slope locations
    for c in range(cols-3):
        for r in range(rows-3):
            if game[r][c] == piece and game[r+1][c+1] == piece and
game[r+2][c+2] == piece and game[r+3][c+3] == piece:
                return True

    # check diagonal negative slope locations
    for c in range(cols-3):
        for r in range(rows-3, rows):
            if game[r][c] == piece and game[r-1][c+1] == piece and
game[r-2][c+2] == piece and game[r-3][c+3] == piece:
                return True

def check_result(game,turn):
    if(turn == 1):
        opturn = 2
    else:
        opturn = 1
    if  winning_move(game, turn):
        return "1st_player"
```

```python
    elif(winning_move(game, opturn)):
        return "2nd_player"
    elif( np.all(game != 0)):
        return "draw"
    return "no_winner_yet"
def change_turn(turn):
    if turn == 1:
        turn = 2
    else:
        turn = 1
    return turn
rows=4
cols=4

def random_play(game):
    list1 = []
    for col in actions:
        if(is_valid_location(game,col)):
            list1.append(col)
    pos = random.randint(0,len(list1)-1)
    col = list1[pos]
    return col

def random_player(game, piece,op_piece):
    possible_moves = []
    for col in actions:
        if is_valid_location(game, col):
            row = get_next_open_row(game, col)
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, op_piece)
            if winning_move(test_game, op_piece):
                return col
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, piece)
            if winning_move(test_game, piece):
                return col
            possible_moves.append(col)
    if possible_moves:
        return random.choice(possible_moves)


def start_game():
    global count_q,count_mini,count_draw, alpha, discount, epsilon
```

```
    game = create_game(rows,cols)
    game_over = False
    #turn = int(input('type 0 for minimax to play first, type 1 for
minimax to play second \n'))
    play = 1
    turn = 1
    result = "no_winner_yet"
    current_state = get_state(game)
    action = 0
    first_step = True
    update_list = []
    while check_result(game,play) == "no_winner_yet":
        # Ask for player input
        if((turn)%2 == 1):
            last_state = current_state
            current_state = get_state(game)
            last_action = action
            result = check_result(game,play)
            last_game = copy.deepcopy(game)
            reward[get_state(game)] = 0
            if not(first_step):

update_list.append((current_state,last_state,copy.deepcopy(game),last_a
ction))
            action =
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step,play)
            first_step = False
            last_state = current_state
            turn=2
        else:
            try:
                col = random_player(game,2,1)
            except:
                breakpoint()
            while not (is_valid_location(game, col)):
                col = int(input("Player 2 make your selection (0-rows):
"))
            if is_valid_location(game, col):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, 2)
                if winning_move(game, 2):
                    #print("Player 2 wins!!!")
```

```
                    game_over = True
            turn = 1
        print_game(game)
    result = check_result(game,play)

    last_action = action

    if (result == "1st_player"):
        Q[(last_state,action)]  = 100
        count_q = count_q+1
        #reward[state] = 200
    elif (result == "2nd_player"):
        count_mini = count_mini+1
        Q[(last_state,action)]  = -100
        #reward[state] = -200
    elif(result == "draw"):
        count_draw = count_draw+1
        Q[(last_state,action)]  = 0
    update_list.reverse()
    for entry in range(len(update_list)):

update_state(update_list[entry][0],update_list[entry][1],update_list[en
try][2],update_list[entry][3],alpha,discount)
    #print("Thanks for playing Connect4!")
Q = {}
N = {}
reward = {}
total_states = 0
count_mini = 0
count_q = 0
count_draw = 0
epsilon = 0.9
alpha = 0.4
discount = 0.9
actions = [*range(cols)]
list_qwin = []
list_rwin = []
list_draw = []
list_count = []

for i in range(100):
    start_game()
    if i%10 == 0 and count_q+count_mini != 0:
```

```
        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("q-learning = " +str(count_qi) )
        print("random = "+ str(count_minii))
        print("draw = "+ str(count_drawi))
    if(i >200000):
        epsilon = 0.2
    if(i>300000):
        epsilon = 0


import matplotlib.pyplot as plt
plt.ylabel('Game result in %')
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Random player wins')
plt.plot(list_count, list_qwin, 'b-', label='Qlearning Wins')
plt.legend()
plt.show()
```

Qlearning vs minimax tictac:

```
import copy
import random
import pickle

def minimax_algo(game, alpha, beta, maxmin):
    result = check_result(game)
    if(result == "draw"):
```

```python
        return 0
    elif(result == "1st_player"):
        return -100
    elif(result == "2nd_player"):
        return 100
    if maxmin == True:
        maxValue = float('-inf')
        for square_index in range(len(game)):
            if(game[square_index] == 0):
                game[square_index] = "o"
                result = minimax_algo(game, alpha, beta,False)
                game[square_index] = 0
                maxValue = max(maxValue,result)
                alpha = max(alpha,maxValue)
                if beta<=alpha:
                    break

        return maxValue
    else:
        minValue = float('inf')
        for square_index in range(len(game)):
            if(game[square_index] == 0):
                game[square_index] = "x"
                result = minimax_algo(game,alpha,beta,True)
                game[square_index] = 0
                minValue = min(minValue,result)
                beta = min(minValue,beta)
                if(beta <= alpha):
                    break
        return minValue

def minimax_play(game):
    max_result = float('-inf')
    index = -1
    for i in range(len(game)):
        if(game[i] == 0):
            game[i] = "o"
            result = 
minimax_algo(game,float('-inf'),float('inf'),False)
            game[i] = 0
            if(result>max_result):
                max_result = result
                index = i
```

```
    return index

def PrintGame(game):
    for i in range(len(game)):
        if game[i] == 0:
            value = "_"
        else:
            value = game[i]
        print(value, end=" ")
        if((i+1)%3 == 0):
            print("\n")


def check_result(game):
    for sequence in winning_sequence():
        if(game[sequence[0]] != 0 and game[sequence[0]] ==
game[sequence[1]] and game[sequence[1]]==game[sequence[2]]):
            if game[sequence[0]] == "x":
                return "1st_player"
            else:
                return "2nd_player"
    if 0 in game:
        return "no_winner_yet"
    return "draw"

def get_state(game):
    return "".join(str(value) for value in game )

def maxQ_value(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return Q[(state,final_action)]
```

```
def maxQ(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return final_action

def least_used_Q(state,game):
    global Q,N,reward,total_states, actions
    min_n = float('inf')
    final_action = -1
    for action in actions:
        if(game[action] == 0):
            if (state,action) not in N:
                N[(state,action)] = 0
                final_action = action
                break
            else:
                if N[(state,action)] < min_n:
                    min_n = N[(state,action)]
                    final_action = action

    return final_action



def explore_exploit(epsilon,action,state,game):
    global Q,N,reward,total_states
    if  random.uniform(0,1) < epsilon :
        action = least_used_Q(state,game)
        N[(state,action)] = N[(state,action)]+1
    return action

def
update_state(current_state,last_state,game,last_action,alpha,discount,e
psilon_first_step):
    global total_states
```

```
    total_states = total_states+1
    try:
        abcd  = (1-alpha)*Q[(last_state,last_action)] + alpha *
(N[(last_state,last_action)]/total_states)*
(discount*maxQ_value(current_state,game) - Q[(last_state,last_action)])
        if(Q[(last_state,last_action)] >5):

            Q[(last_state,last_action)] = abcd
        else:
            Q[(last_state,last_action)] = abcd
    except:
        breakpoint()
def
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step):
    global Q,N,reward,total_states
    if not (first_step ):

update_state(current_state,last_state,game,last_action,alpha,discount,e
psilon)

    action = maxQ(current_state,game)
    if (current_state,action) not in N:
        N[(current_state,action)] = 1

    else:
        N[((current_state,action))] = N[((current_state,action))]+1
    action = explore_exploit(epsilon,action,current_state,game)
    game[action] = "x"
    return action

def winning_sequence():
    return
[[2,4,6],[0,4,8],[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8]]
def find_prob_loss(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"x" and game[sequence[index1]] == game[sequence[index2]])
def find_prob_win(index1,index2,index3,sequence,game):
    return (game[sequence[index3]] == 0 and game[sequence[index1]] ==
"o" and game[sequence[index1]] == game[sequence[index2]])

def random_play(game):
    for sequence in winning_sequence():
```

```
        if find_prob_win(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
            return sequence[2]
        elif find_prob_win(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_win(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]

    for sequence in winning_sequence():
        if find_prob_loss(0,1,2,sequence,game) :
            game[sequence[2]] = "o"
            return sequence[2]
        elif find_prob_loss(1,2,0,sequence,game):
            game[sequence[0]] = "o"
            return sequence[0]
        elif find_prob_loss(0,2,1,sequence,game):
            game[sequence[1]] = "o"
            return sequence[1]
    list1 = []
    for i in range(len(game)):
        if(game[i] == 0):
            list1.append(i)
    pos = random.randint(0,len(list1)-1)
    return list1[pos]



def user_play(game):
    position = input('input the box number you chose. range(1-9) \n')
    if game[int(position)-1] != 0:
        print("Wrong Move")
        exit(1)
    game[int(position)-1] = 'o'
    return game

def start_game():
    global count_q,count_mini,count_draw, alpha, discount, epsilon
    game = [0] * 9
    #play = int(input('type 1 for minimax to play first, type 2 for
minimax to play second \n'))
    play = 1
```

```
    result = "no_winner_yet"
    current_state = get_state(game)
    action = 0
    first_step = True
    while check_result(game) == "no_winner_yet":
        if((play+1)%2 != 1):
            print("q-agent turn")
            last_state = current_state
            current_state = get_state(game)
            last_action = action
            result = check_result(game)
            last_game = copy.deepcopy(game)
            reward[get_state(game)] = 0
            action =
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step)
            first_step = False
            #last_game = copy.deepcopy(game)

            last_state = current_state
        else:
            print("minimax play turn")
            PrintGame(game)
            minimax_game = copy.deepcopy(game)
            pos = minimax_play(game)
            game[pos] = "o"
            #game = user_play(game)
        play = (play+1)%2
    result = check_result(game)
    last_action = action
    if (result == "1st_player"):
        Q[(last_state,action)]  = 100
        count_q = count_q+1
        PrintGame(game)
        #reward[state] = 200
    elif (result == "2nd_player"):
        Q[(last_state,action)]  = -100
        count_mini = count_mini+1
        PrintGame(game)
        #reward[state] = -200
    elif(result == "draw"):
        count_draw = count_draw+1
        PrintGame(game)
```

```
        Q[(last_state,action)]  = 0


with open('q_agent_tictactoe.pickle', 'rb') as handle:
    Q = pickle.load(handle)

N = {}
reward = {}
total_states = 0
count_mini = 0
count_q = 0
count_draw = 0
epsilon = 0
alpha = 0.4
discount = 0
actions = [0,1,2,3,4,5,6,7,8]
list_qwin = []
list_rwin = []
list_draw = []
list_count = []

for i in range(2000):
    start_game()
    if i%100 == 0 and count_q+count_mini+count_draw != 0:
        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("q-learning = " +str(count_qi) )
        print("minimax = "+ str(count_minii))
        print("draw = "+ str(count_drawi))


import matplotlib.pyplot as plt
plt.ylabel('Game result')
```

```
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Minimax wins')
plt.plot(list_count, list_qwin, 'b-', label='Qlearning Wins')
plt.legend()
plt.show()
```

Qlearning vs Minimax Connect4:

```
import numpy as np
import random
count = 0
import copy



def get_state(game):
    state = ""
    for i in range(len(game)):
        for j in range(len(game[0])):
            state = state + str(int(game[i][j]))

    return state


def minimax_algo(game,turn, alpha, beta, maxmin):
    global count
    op_turn = change_turn(turn)
    result = check_result(game)
    if(result != "no_winner_yet"):
        #count = count+1
        pass
    if(result == "draw"):
        return 0
    elif(result == "1st_player"):
        return -100
    elif(result == "2nd_player"):
        return 100
    if maxmin == True:
```

```
        maxValue = float('-inf')
        for col in range(len(game)):
            if(game[rows-1][col] == 0):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, turn)
                result = minimax_algo(game,op_turn, alpha, beta,False)
                reverse_drop(game, row, col, turn)
                maxValue = max(maxValue,result)
                alpha = max(alpha,maxValue)
                if(maxValue < -150):
                    breakpoint()
                if beta<=alpha:
                    break
        return maxValue
    else:
        minValue = float('inf')
        for col in range(len(game)):
            if(game[rows-1][col] == 0):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, turn)
                result = minimax_algo(game,op_turn,alpha,beta,True)
                reverse_drop(game, row, col, turn)
                minValue = min(minValue,result)
                beta = min(minValue,beta)
                if(beta <= alpha):
                    break
        return minValue

def minimax_play(game,turn):
    max_result = float('-inf')
    final_row = -1
    final_col = -1
    for i in range(len(game[0])):
        if(game[rows-1][i] == 0):
            row = get_next_open_row(game, i)
            drop_piece(game, row, i, turn)
            op_turn = change_turn(turn)
            result =
minimax_algo(game,op_turn,float('-inf'),float('inf'),False)
            reverse_drop(game, row, i, turn)
            if(result>max_result):
                max_result = result
                final_row = row
```

```
                final_col = i

    return final_col

def maxQ_value(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(is_valid_location(game, action)):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return Q[(state,final_action)]

def maxQ(state,game):
    global Q,N,reward,total_states , actions
    max_value = float('-inf')
    final_action = -1
    for action in actions:
        if(is_valid_location(game, action)):
            if (state,action) not in Q:
                Q[(state,action)] = 0
            if(Q[(state,action)]> max_value):
                max_value = Q[(state,action)]
                final_action = action
    return final_action

def least_used_Q(state,game):
    global Q,N,reward,total_states, actions
    min_n = float('inf')
    final_action = -1
    for action in actions:

        if(is_valid_location(game, action)):
            if (state,action) not in N:
                N[(state,action)] = 0
                final_action = action
                break
            else:
```

```
                if N[(state,action)] < min_n:
                    min_n = N[(state,action)]
                    final_action = action

    return final_action



def explore_exploit(epsilon,action,state,game):
    global Q,N,reward,total_states
    if  random.uniform(0,1) < epsilon :
        action = least_used_Q(state,game)
        N[(state,action)] = N[(state,action)]+1
    return action

def
update_state(current_state,last_state,game,last_action,alpha,discount):
    global total_states
    total_states = total_states+1
    try:
        Q[(last_state,last_action)]  =
(1-alpha)*Q[(last_state,last_action)] + alpha *
(discount*maxQ_value(current_state,game) - Q[(last_state,last_action)])
    except:
        breakpoint()
def
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step,play):
    global Q,N,reward,total_states

    #if not (first_step ):


#update_state(current_state,last_state,game,last_action,alpha,discount,
epsilon)

    action = maxQ(current_state,game)
    if (current_state,action) not in N:
        N[(current_state,action)] = 1

    else:
        N[((current_state,action))] = N[((current_state,action))]+1
    col = explore_exploit(epsilon,action,current_state,game)
```

```
    row = get_next_open_row(game, col)
    drop_piece(game, row, col, play)
    return col


#connect4 board


import numpy as np


def create_game(rows,cols):
    game = np.zeros((rows, cols))
    return game

def drop_piece(game, row, col, piece):
    game[row][col] = piece
def reverse_drop(game, row, col, piece):
    game[row][col] = 0
def is_valid_location(game, col):
    try:
        return game[rows-1][col] == 0
    except:
        breakpoint()
def get_next_open_row(game, col):
    for r in range(rows):
        if game[r][col] == 0:
            return r

def print_game(game):
    print(np.flip(game, 0))

def winning_move(game, piece):
    # check horizontal locations
    for c in range(cols-3):
        for r in range(rows):
            if game[r][c] == piece and game[r][c+1] == piece and
game[r][c+2] == piece and game[r][c+3] == piece:
                return True

    # check vertical locations
    for c in range(cols):
        for r in range(rows-3):
```

```
            if game[r][c] == piece and game[r+1][c] == piece and
game[r+2][c] == piece and game[r+3][c] == piece:
                return True

    # check diagonal positive slope locations
    for c in range(cols-3):
        for r in range(rows-3):
            if game[r][c] == piece and game[r+1][c+1] == piece and
game[r+2][c+2] == piece and game[r+3][c+3] == piece:
                return True

    # check diagonal negative slope locations
    for c in range(cols-3):
        for r in range(rows-3, rows):
            if game[r][c] == piece and game[r-1][c+1] == piece and
game[r-2][c+2] == piece and game[r-3][c+3] == piece:
                return True

def check_result(game):

    if  winning_move(game, 1):
        return "1st_player"
    elif(winning_move(game, 2)):
        return "2nd_player"
    elif( np.all(game != 0)):
        return "draw"
    return "no_winner_yet"
def change_turn(turn):
    if turn == 1:
        turn = 2
    else:
        turn = 1
    return turn
rows=4
cols=4

def random_play(game):
    list1 = []
    for col in actions:
        if(is_valid_location(game,col)):
            list1.append(col)
    pos = random.randint(0,len(list1)-1)
    col = list1[pos]
```

```
    return col

def random_player(game, piece,op_piece):
    possible_moves = []
    for col in actions:
        if is_valid_location(game, col):
            row = get_next_open_row(game, col)
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, op_piece)
            if winning_move(test_game, op_piece):
                return col
            test_game = copy.deepcopy(game)
            drop_piece(test_game, row, col, piece)
            if winning_move(test_game, piece):
                return col
            possible_moves.append(col)
    if possible_moves:
        return random.choice(possible_moves)


def start_game():
    global count_q,count_mini,count_draw, alpha, discount, epsilon
    game = create_game(rows,cols)
    game_over = False
    #turn = int(input('type 0 for minimax to play first, type 1 for
minimax to play second \n'))
    play = 1
    turn = 1
    result = "no_winner_yet"
    current_state = get_state(game)
    action = 0
    first_step = True
    update_list = []
    while check_result(game) == "no_winner_yet":
        # Ask for player input
        if((turn)%2 == 1):
            last_state = current_state
            current_state = get_state(game)
            last_action = action
            result = check_result(game)
            last_game = copy.deepcopy(game)
            reward[get_state(game)] = 0
            if not(first_step):
```

```
update_list.append((current_state,last_state,copy.deepcopy(game),last_a
ction))
            print("qlearning_turn")
            action =
q_learning(game,alpha,discount,epsilon,current_state,last_state,last_ac
tion,first_step,play)
            first_step = False
            last_state = current_state
            turn=2
        else:
            try:
                print("minimax_turn")
                col = minimax_play(game,2)
            except:
                breakpoint()
            while not (is_valid_location(game, col)):
                col = int(input("Player 2 make your selection (0-rows):
"))
            if is_valid_location(game, col):
                row = get_next_open_row(game, col)
                drop_piece(game, row, col, 2)
                if winning_move(game, 2):
                    print("Player 2 wins!!!")
                    game_over = True
            turn = 1
        print_game(game)
    result = check_result(game)
    last_action = action
    if (result == "1st_player"):
        Q[(last_state,action)]  = 100
        count_q = count_q+1
        print("player1 wins")
        #reward[state] = 200
    elif (result == "2nd_player"):
        count_mini = count_mini+1
        Q[(last_state,action)]  = -100
        #reward[state] = -200
    elif(result == "draw"):
        count_draw = count_draw+1
        Q[(last_state,action)]  = 0
    update_list.reverse()
    for entry in range(len(update_list)):
```

```
update_state(update_list[entry][0],update_list[entry][1],update_list[entry][2],update_list[entry][3],alpha,discount)
    #print("Thanks for playing Connect4!")

import pickle
with open('q_agent_connect4.pickle', 'rb') as handle:
    Q = pickle.load(handle)

N = {}
reward = {}
total_states = 0
count_mini = 0
count_q = 0
count_draw = 0
epsilon = 0
alpha = 0.4
discount = 0.9
list_qwin = []
list_rwin = []
list_draw = []
list_count = []
actions = [*range(cols)]
if __name__ == '__main__':
    for i in range(10):
        start_game()

        print("i ="+str(i))
        sum = count_q+count_mini+count_draw
        count_qi = count_q/sum
        count_minii = count_mini/sum
        count_drawi = count_draw/sum
        count_q=0
        count_mini=0
        count_draw=0
        list_qwin.append(count_qi)
        list_rwin.append(count_minii)
        list_draw.append(count_drawi)
        list_count.append(i)
        print("q-learning = " +str(count_qi) )
        print("minimax = "+ str(count_minii))
        print("draw = "+ str(count_drawi))
```

```python
import matplotlib.pyplot as plt
plt.ylabel('Game result')
plt.xlabel('Game count')

plt.plot(list_count, list_draw, 'r-', label='Draw')
plt.plot(list_count, list_rwin, 'g-', label='Minimax wins')
plt.plot(list_count, list_qwin, 'b-', label='Qlearning Wins')
plt.legend()
plt.show()
```