AI Assignment

Submitted by: Chirag Saxena
Student number: 22312293
Email: saxenac@tcd.ie

Implement in Python (or reuse existing open source) games of Tic Tac Toe and Connect 4.

I took help to implement Tic Tac Toe (Min-Max) from the geeks for geeks website and Then I did major changes to that code to use it for Connect 4 (Min-Max). I wrote Q-learning on my own and took some help from YouTube videos.

Implement Minimax algorithm (with alpha-beta pruning) and tabular Q-learning Reinforcement Learning algorithm for playing each of the two games above.

I took help to implement Tic Tac Toe (Min-Max) from the geeks for geeks website and Then I did major changes to that code to use it for Connect 4 (Min-Max). I wrote Q-learning on my own and took some help from YouTube videos. Code and reference is given below in the form of classes.

Implement a default opponent that will play these games against your algorithms. It does not have to be very advanced but it has to be better than fully random (eg, opponent must select a winning move if it exists. It must select a blocking move (i.e. prevent opponent from winning) if it exists.

For Tic-Tac-Toe, I wrote algorithm which checks winning combinations (if found) and place the blocking move If not found then it will place the move to random position available. For Tic-Tac-Toe , if I represents it in 2-D format then winning moves will be given below.

[(0, 0), (0, 1), (0, 2)], [(1, 0), (1, 1), (1, 2)],

[(2, 0), (2, 1), (2, 2)], [(0, 0), (1, 0), (2, 0)],

[(0, 1), (1, 1), (2, 1)], [(0, 2), (1, 2), (2, 2)],

[(0, 0), (1, 1), (2, 2)], [(0, 2), (1, 1), (2, 0)]

For Connect 4. The algorithm will be the mostly the same but winning combinations will be vast. It will check the winning combination in Horizontal, Vertical and Diagonal positions (if Found) Otherwise will return the next random move available.

<u>Algorithms and Design Choice</u>

Minmax : One of the most popular algorithm to determine best possible move from all possible move. At every level it generates max and min nodes then it calculates the best score (Heuristic Function) recursively by going down to up (root node – where game started). The Time complexity is O(b^d) where d is the depth of the game tree and b is the branching factor (average number of children per node).

The number of nodes that must be explored can be decreased through alpha-beta pruning and other optimizations, which reduces the worst-case time complexity. The time complexity of the Minimax algorithm is still exponential in the depth of the game tree in the general case.

Design choice (MinMax) : The Minmax function (*board, depth, alpha, beta, isMax, isPrun)* parameters. Where alpha and beta parametes are for the pruning. *isPrun* parameter is Boolean so if we want to calculate time with or without pruning we can do that so. Also depth parameter will limit the search to certain depth provided. This makes testing very easy. In main Function we can choose who can play first MinMax or default opponent.

Design of Tic Tac Toe is similar to Connect 4 but the major difference is that how it calculate reward. Because legal moves are different in these games.

Q – Learning : The goal of Q-learning, a type of reinforcement learning algorithm, is to identify the best course of action for an agent to take in order to maximize rewards. The steps are to calculate Q table is: Select the current state by epsilon greedy method and observe it. Then take the action and observe the next state. Now here we use modified bellman equation to update the Q-Value of the current state. The equation is :

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma(\max Q(s', a')) - Q(s, a)]$$

Here ***'s'*** is a current state, α (alpha) determines how much new information overrides previous knowledge in the Q-table. γ (Gamma) is the discount factor, regulates how important future rewards are to the agent when making decisions. **s'** is next state.

The time complexity of Tic Tac Toe depends on the size of the state space, the number of actions available in each state, and the number of training episodes necessary for convergence. There are a total of 39 possible states in the state space, or roughly 19,683 states. Q-learning has an O(SAN) time complexity, where S is the number of states, A is the number of actions, and N is the number of training episodes necessary to converge. Q-time learning's complexity for the game of Tic Tac Toe would be O(19,683 * 9 * N).

Design Choice (Q-learning) : In my case QAgent Function takes care of everything. We can pass argument to run it for number of episodes.

The game state is initialized at the beginning of the algorithm as a tuple of nine integers, where 0 denotes an empty cell, 1 denotes a move made by player 1 (O), and 2 denotes a move made by player 2. (X).

The algorithm then alternates between player 1's turn (O) and player 2's turn (X), starting with player 1. Using the find mov() function, the algorithm first creates a list of all moves that are currently available for use (i.e., empty cells in the current state). It then makes a copy of the current state and plays a move by setting the corresponding cell to the player's number (1 for player 1, 2 for player 2).

Depending on whether it is player 1 or player 2's turn, the algorithm then calls either the min action() or max action() function to calculate the Q-value of the resulting state. The minimum or maximum Q-value among all feasible actions in the specified state is returned by these functions. Here by alternatively providing min action() and max action () we are forcing the agent to choose minimum Q value for the opponent. Doing this, will provide the optimal strategy.

Also Q table is the form of dictionary. I could have used the array or another data structure to store the states. Dictionary is important because the searching time for each state is O(1) and searching through array is O(n) operation. As we know During Connect 4 the legal moves are reaching Trillions so O(n) is not practical.

The main difference between Tic Tac Toe and Connect 4 is game state is initialized as a tuple of 41 integer (7x6) and how to calculates reward to legal moves.

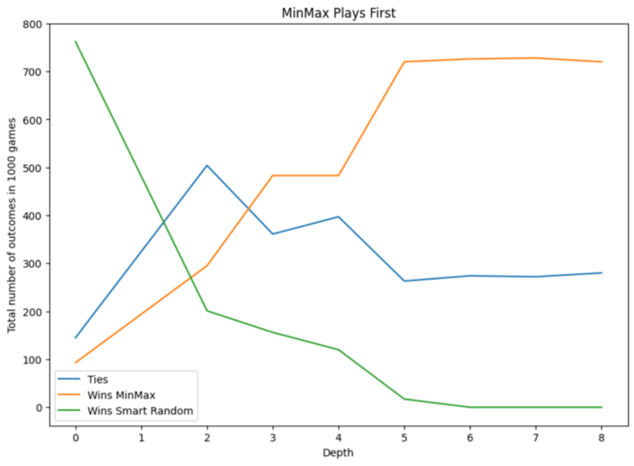
Compare the following, presenting graphs and analysis/conclusions, making sure you let agents play a sufficient number of games to be able to draw conclusions:

1. How do your algorithms compare to each other when playing against default opponent in Tic Tac Toe.

The following results are when **MinMax plays first against default player**. In this case I ran total 1000 games on multiple depths. I am comparing total wins of Minmax, total wins of Default Player and total ties between 2 algorithms out of 1000 games.

Talking about wins of minmax we can see it outperforms the default player at higher depth. At depth = 2, wins of minmax and wins of default is same and at depth = 0 default player starts to win.

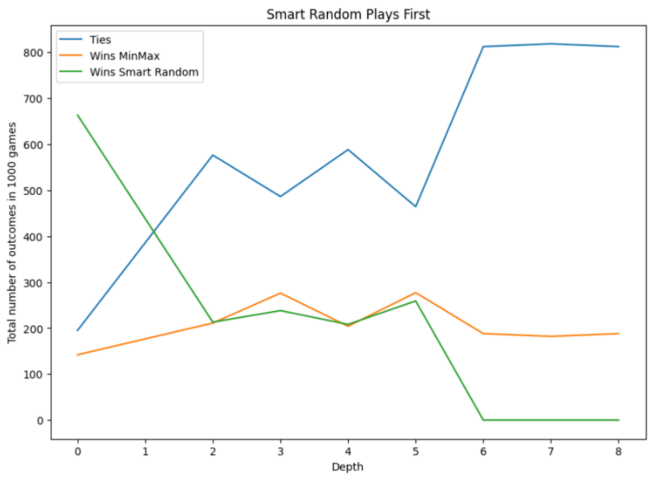
Depth	Wins (Minmax)	Wins (Default)	Ties (Minmax vs Default)
8	720	0	280
7	728	0	272
6	726	0	274
5	720	17	263
4	483	120	397
3	483	156	361
2	295	201	762
0	93	762	145



Talking about wins of default player at higher depths it is losing every time. But at depth = 5, it starts to win and depth = 0, it outperforms the minmax.

Talking about the ties we can see that as depth decreases, number of ties increases. Also, we can see that at depth = 7 number of wins of minmax is higher than default player this is due to randomness introduced in the game so every time we play the game will results in different outcomes.

The following results are when **Default plays first against minmax.**



Here we can see that wins of minmax is around 188 at depth = 8, 7, 6. It outperforms the default player in this case also. But ties are very high around 800.

Depth	Wins (Minmax)	Wins (Default)	Ties (Minmax vs Default)
8	188	0	812
7	182	0	818
6	188	0	812
5	277	259	464
4	204	208	588
3	276	238	486
2	211	213	576
0	142	663	195

Wins of default player is 0 at higher depths but starts to increase as depth become lower. At 2<=depth<=5, wins of minmax is fairly equal to wins of default but at depth = 0, default outperforms the minmax.

The following results are when **Q learning plays first against default player** at 5,00,000 episodes. Approximate number of states = 181700.

Alpha	Beta	Wins (Q-L)	Wins (Default)	Ties (Q-L vs Default)
0.1	0.8	733	0	267
0.5	0.8	727	25	248
0.8	1	707	43	250

We can see that alpha = 0.1, gamma = 0.8 gives wins for Q-learning. Where wins of default is 0. Ties at every alpha, gamma are around 250. We can see that as alpha , gamma increase the wins of Q-L is decreasing and wins of default is increasing.

The following results are when **default player plays first against Q - L** at 5,00,000 episodes.

Alpha	Beta	Wins (Q-L)	Wins (Default)	Ties (Q-L vs Default)
0.1	0.8	883	28	89
0.5	0.8	886	30	84
0.8	1	876	30	94

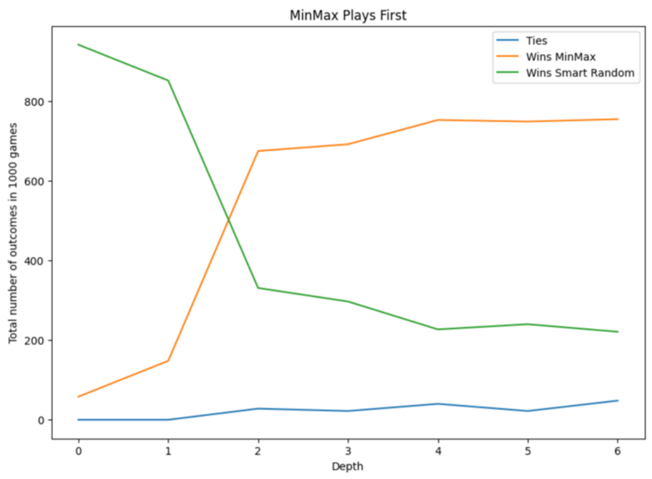
Here, changing alpha and gamma doesn't create any difference. Wins Q-L are average around 850, wins default are average around 30 and ties are around 85.

2. How do your algorithms compare to each other when playing against default opponent in Connect 4?

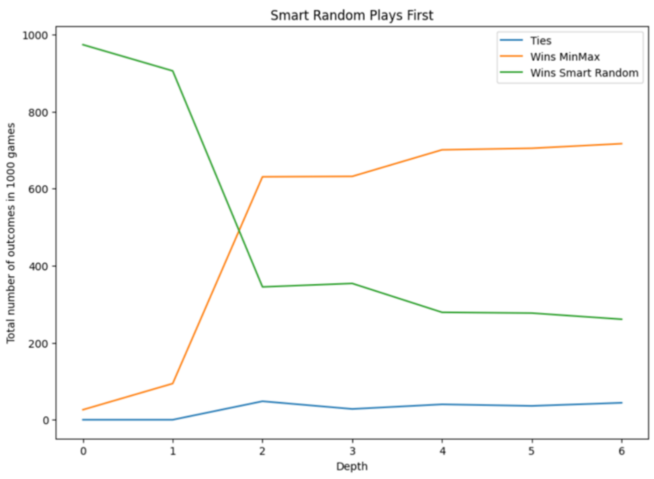
The following results are **when minmax plays first against default player** on multiple depths. I was not able to take very high depths because of computational limitations. I am comparing total wins of Minmax, total wins of Default Player and total ties between 2 algorithms out of 1000 games.

Depth	Wins (Minmax)	Wins (Default)	Ties (Minmax vs Default)
6	755	221	48
5	749	240	22
4	753	227	40
3	692	297	22
2	675	331	28
1	148	852	0
0	58	942	0

As we can see at higher depths minmax is performing good. At depth = 6, 5, 4 the results are similar. Trend starts to change when depth<=3 and at depth = 0, 1 default player outperforms minmax.



The following results are **when default plays first against minmax.**



Depth	Wins (Minmax)	Wins (Default)	Ties (Minmax vs Default)
6	717	261	44
5	705	277	36
4	701	279	40
3	632	354	28
2	631	345	48
1	94	906	0
0	26	974	0

Talking about wins of minmax, it performing better at higher depth but at lower depths default outperforms minmax. We can see that there is similar trends between **minmax plays first and default plays first** but slightly more wins of default player.

The following results are when **Q learning plays first against default player** at 20,000 episodes.

Here the number of episodes are less because of com putational limit. There are 41 Trillion states in Connect 4. Storing all the states is next to impossible on my ma chine (8 GB ram). That means the performance of Q-L will not be the good compared to minmax. The approx imate number of states are `2774177`.

Alpha	Beta	Wins (Q-L)	Wins (Default)	Ties (Q-L vs Default)
0.1	0.8	498	494	8
0.5	0.8	451	546	3
0.8	1	402	593	5

Here we can see that at alpha = 0.1 and gamma = 0.8, Q-L outperforms the default player and number of ties are very low. As we increase alpha and gamma, the number of wins Q-L starts to decrease and wins default starts to increase but no significant effect on the number of ties.

The following results are when **default player plays first against Q-L** at 20,000 episodes.

Alpha	Beta	Wins (Q-L)	Wins (Default)	Ties (Q-L vs Default)
0.1	0.8	348	647	5
0.5	0.8	338	655	7
0.8	1	344	646	10

Here is also the same trend as Q-L tic tac toe. No significant effect seen on changing the alpha and gamma values.

3. How do your algorithms compare to each other when playing against default opponent overall ?

Tic Tac Toe (MinMax) : On analysing the data given we can say that depth = 6 is the max depth of the game tree because graph is flat after that. We can see that result of depth = 7,8 is same as the depth = 6.

Connect4 (MinMax) : On analysing the data given, we can see that depth = 6 is not the max depth of the game tree because default player is still winning (even the number of low). The graph is flat after depth = 4 but we can see is changing if I would have tested it on more depth which is not possible on my system.

Tic Tac Toe and Connect 4 (MinMax) : In minmax if someone plays first, the time taken to generate next move by minmax is decreased because there will be less states combination.

Tic Tac Toe and Connect 4 (Q-L): We can see changing the alpha changes the result this is because a high alpha value indicates a faster update of the agent's Q-values, whereas a low alpha value indicates a slower update of the agent's Q-values. The rate of learning and the speed at which the agent finds the best policy can both be significantly impacted by changing the alpha value. Also The agent's ability to learn and make decisions can be significantly impacted by changing the gamma value. For instance, the agent may behave more conservatively in the short term if the gamma value is set to a high value because it will prioritize long-term rewards over immediate rewards. On the other hand, if the gamma value is set to a low value, the agent will place more weight on short-term rewards than long-term ones, which could result in riskier behaviour in the short run.

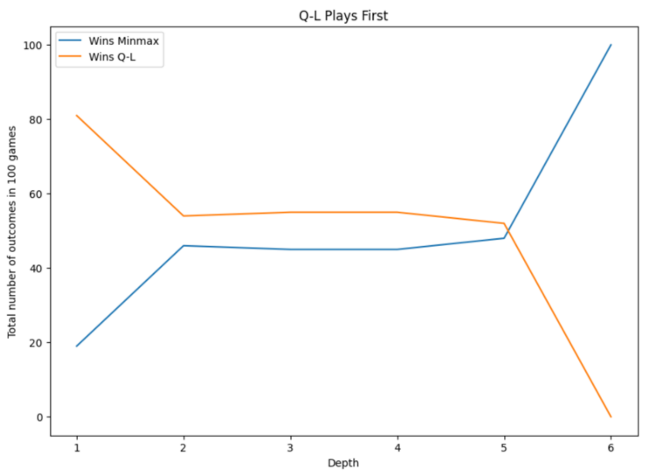
Why the results shows more wins for who played first in Tic Tac Toe and Connect4 ?

The first player in Tic Tac Toe has an advantage because they get to place their first mark in the center square, which is the most important square from a strategic standpoint. The first player may be able to prevent the second player from making a winning pattern in any direction by placing their mark in the center square. The first player also has the option of developing the mark they put in the center square into their own winning pattern.

The first player in Connect Four has an advantage because they can always guarantee a victory if they play flawlessly. This is so that the first player, who has an extra move, can force a win by assembling a row of four pieces in any direction. No matter what the second player does, if the first player plays the best possible game, they will always win or force a draw.

4. How do your algorithms compare to each other when playing against each other in Tic Tac Toe?

Now Q-L is a trained model and minmax is very powerful due to optimal nature. Here total number of game played = 100 only (not 1000 like previously)



Depth	Wins (Q-L)	Wins (Min Max)
1	19	81
2	46	55
3	45	55
4	45	55
5	48	52
6	0	100

Here we can see that at lower depth minmax is performing poorly against Q-L but at 2<=depth<=5 is a Tie and at depth = 6 minmax outperformed the Q-L.

Talking about when minmax plays first, In my case minmax is always winning.

So the best case scenario should be Tie. Tie can happen if Wins Q-L is equal to Wins Minmax or No one wins (Tie = 100) but no one wins (Tie) scenario cannot happen because there is no randomness both of the algorithms and will play optimally so there is winning or losing only.

5. How do your algorithms compare to each other when playing against each other in Connect 4 ?

In my case Connect 4 minmax vs Connect 4 Q-L is performing very bad. Connect 4 minmax is winning every time even at depth = 1 which is the minimum depth. I think this is because due to low number of states present in Q table leads to infinite loss. I ran Q-L for total 50000 episodes which has 220 millions. The total legal states in Connect4 is around 41 trillion. If we calculate percentage, our Q-table is missing 95.5% states. This is reason we have less number of states.

6. How do your algorithms compare to each other when playing against each other overall?

My Tic Tac Toe minmax vs Q-learning is performing very well because we can see tie in every from 2 to 5 depth. That means when minmax plays at average difficulty it average outs the Q-L. at depth = 1, Q-L performs well and worst at depth = 6.

Now, final verdict about Connect 4 minmax vs Connect 4 Q-L is if I had better system then I would have been able to store all the states and results would be better.

Final Verdict : Tic Tac Toe has a smaller state space than Connect 4, making it easier for a Q-learning agent to explore every possible state quickly. Tic Tac Toe is also a game of perfect information, which means that at all times, both players are fully aware of the game's current state.

Tic Tac Toe is a great game for Q-learning because of these elements, and using this method to solve the game is fairly simple. In fact, using Q-learning, it is possible to build the ideal Tic Tac Toe playing agent because the best strategy can be found by examining every move and how it will turn out.

Contrarily, Connect 4 has a much larger state space and a more complicated set of rules, making it more difficult for a Q-learning agent to discover the best course of action. Additionally, because it is not always possible to predict your opponent's moves, Connect 4 is an imperfect information game. This calls for the agent to develop the ability to decide based on ambiguous information, which can be more challenging.

The performance of a Q-learning agent in Connect 4 can be improved using a variety of methods, including feature engineering to condense the state space, modifying the reward function to reward good play, and using more sophisticated algorithms like Monte Carlo Tree Search or Deep Reinforcement Learning.

References:

1. https://www.geeksforgeeks.org/q-learning-in-python/

2. https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/

3. **https://github.com/ShekMaha/connect4-reinforcement-learning/blob/master/connect4.py**

# Code for TicTacToe MinMax

```
class TicTacToe:
    def __init__(self):
        self.player, self.opponent = 'o', 'x'

    def printboard(self,board):
        print(' {} | {} | {}'.format(board[0][0], board[0][1], board[0][2]))
        print('---+---+---')
        print(' {} | {} | {}'.format(board[1][0], board[1][1], board[1][2]))
        print('---+---+---')
        print(' {} | {} | {}'.format(board[2][0], board[2][1], board[2][2]))
        print('\n')

    def isMovesLeft(self,board) :
        for i in range(3) :
            for j in range(3) :
                if (board[i][j] == '_') :
                    return True
        return False

    def check_winner(self, board):
        for row in board:
            if row[0] != '_' and row[0] == row[1] == row[2]:
                return row[0]

        for col in range(3):
            if board[0][col] != '_' and board[0][col] == board[1][col] == board[2][col]:
                return board[0][col]

        if board[0][0] != '_' and board[0][0] == board[1][1] == board[2][2]:
            return board[0][0]
        if board[0][2] != '_' and board[0][2] == board[1][1] == board[2][0]:
            return board[0][2]

        return None


    def evaluate(self,b) :
        for row in range(3) :
            if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
                if (b[row][0] == self.player) :
                    return 10
                elif (b[row][0] == self.opponent) :
                    return -10

        for col in range(3) :
            if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
                if (b[0][col] == self.player) :
                    return 10
                elif (b[0][col] == self.opponent) :
                    return -10

        if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
            if (b[0][0] == self.player) :
                return 10
            elif (b[0][0] == self.opponent) :
                return -10

        if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
            if (b[0][2] == self.player) :
                return 10
            elif (b[0][2] == self.opponent) :
                return -10

        return 0

    def minimax(self, board, depth, alpha, beta, isMax, isPrun) :
        score = self.evaluate(board)
        if (score == 10) :
            return score

        if (score == -10) :
            return score

        if (self.isMovesLeft(board) == False) :
            return 0

        if depth==0:
            return 0


        if (isMax) :
            best = -1000
            for i in range(3) :
                for j in range(3) :
                    if (board[i][j]=='_') :
                        board[i][j] = self.player
                        best = max( best, self.minimax(board,depth - 1, alpha, beta, not isMax,
isPrun))
                        board[i][j] = '_'
                        if not isPrun:
                            alpha = max(alpha, best)
                            if beta <= alpha:
                                break
            return best
        else :
            best = 1000
            for i in range(3) :
                for j in range(3) :

                    if (board[i][j] == '_') :
                        board[i][j] = self.opponent
                        best = min(best, self.minimax(board, depth - 1, alpha, beta, not isMax,
isPrun))
                        board[i][j] = '_'
                        if not isPrun:
                            alpha = max(alpha, best)
                            if beta <= alpha:
                                break

            return best

    def get_best_move(self, board, depth, isPrun):
        bestVal = -1000
        bestMove = (-1, -1)
        alpha = -1000
        beta = 1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j] == '_') :
                    board[i][j] = self.player
                    moveVal = self.minimax(board, depth-1, alpha, beta, False, isPrun)
                    board[i][j] = '_'
                    if (moveVal > bestVal) :
                        bestMove = (i, j)
                        bestVal = moveVal
                    if not isPrun:
                        alpha = max(alpha, bestVal)
                        if beta <= alpha:
                            break

        return bestMove
```

# Code for TicTacToe Qlearning

```
class TicTacToeQ:
    def __init__(self, alpha, gamma, epsilon):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.Q = dict()
        self.winning_combinations = [
                    (0, 1, 2), (3, 4, 5), (6, 7, 8),
                    (0, 3, 6), (1, 4, 7), (2, 5, 8),
                    (0, 4, 8), (2, 4, 6)
                    ]


    def printboard(self, state):
        state = [' ' if x==0 else x for x in state]
        state = ['x' if x==1 else x for x in state]
        state = ['o' if x==2 else x for x in state]
        print(' {} | {} | {}'.format(state[0], state[1], state[2]))
        print('---+---+---')
        print(' {} | {} | {}'.format(state[3], state[4], state[5]))
        print('---+---+---')
        print(' {} | {} | {}'.format(state[6], state[7], state[8]))
        print('\n')



    def endgame(self, state):
        check = 1
        if state[0] == check and state[4] == check and state[8] == check:
          return check
        if state[2] == check and state[4] == check and state[6] == check:
          return check
        for i in range(3):
          if (
            state[i * 3] == check
            and state[i * 3 + 1] == check
            and state[i * 3 + 2] == check
          ):
            return check
        for i in range(3):
          if state[i] == check and state[3 + i] == check and state[6 + i] == check:
            return check
        check = 2
        if state[0] == check and state[4] == check and state[8] == check:
          return check
        if state[2] == check and state[4] == check and state[6] == check:
          return check
        for i in range(3):
          if (
            state[i * 3] == check
            and state[i * 3 + 1] == check
            and state[i * 3 + 2] == check
          ):
            return check
        for i in range(3):
                if state[i] == check and state[3 + i] == check and state[6 + i] == check:
                    return check
            if len([i for i in range(len(state)) if state[i] == 0]) == 0:
                return -1
            return 0

        def get_computer_move(self, state):
            for combination in self.winning_combinations:
                if state[combination[0]] == state[combination[1]] == 1 and state[combination[2]]
== 0:
                return combination[2]
            elif state[combination[0]] == state[combination[2]] == 1 and
state[combination[1]] == 0:
                return combination[1]
            elif state[combination[1]] == state[combination[2]] == 1 and
state[combination[0]] == 0:
                return combination[0]
        for combination in self.winning_combinations:
            if state[combination[0]] == state[combination[1]] == 2 and state[combination[2]]
== 0:
                return combination[2]
            elif state[combination[0]] == state[combination[2]] == 2 and
state[combination[1]] == 0:
                return combination[1]
            elif state[combination[1]] == state[combination[2]] == 2 and
state[combination[0]] == 0:
                return combination[0]
        while True:
            move = random.randint(0, 8)
            if state[move] == 0:
                return move

    def check_win(self, state):
        for i in range(3):
            if state[i*3] == state[i*3+1] == state[i*3+2] != 0:
                return state[i*3]

        for i in range(3):
            if state[i] == state[i+3] == state[i+6] != 0:
                return state[i]

        if state[0] == state[4] == state[8] != 0:
            return state[0]
        if state[2] == state[4] == state[6] != 0:
            return state[2]

        return 0
```

```
def get_all_actions(self, state):
    for i in range(9):
        if (state, i) in self.Q:
            print(i, ":", self.Q[(state, i)])


def find_mov(self, state, turn):
    side = 1 if turn == 1 else -1
    empty_pos = [i for i in range(len(state)) if state[i] == 0]
    if random.random() < self.epsilon:
        first_try = True
        for pos in empty_pos:
            if (state, pos) not in self.Q:
                self.Q[(state, pos)] = random.random() * 2 - 1
            if first_try:
                max_Q = side * self.Q[(state, pos)]
                max_a = pos
                first_try = False
            else:
                if side * self.Q[(state, pos)] > max_Q:
                    max_Q = side * self.Q[(state, pos)]
                    max_action = pos
        return pos
    return random.choice(empty_pos)



def reward(self, state):
    ret = self.endgame(state)
    if ret == 1:
        return 1000
    if ret == 2:
        return -1000
    return 0



def max_action(self, Q, state):
    state = tuple(state)
    empty_pos = [i for i in range(len(state)) if state[i] == 0]
    first_try = True
    max_action = -1
    max_Q = 0
    for pos in empty_pos:
        if (state, pos) not in Q:
            Q[(state, pos)] = random.random() * 2 - 1
        if first_try:
            max_Q = Q[(state, pos)]
            max_action = pos
            first_try = False
        else:
            if Q[(state, pos)] > max_Q:
                max_Q = Q[(state, pos)]
                max_action = pos
    return max_action, max_Q


def min_action(self, Q, state):
    state = tuple(state)
    empty_pos = [i for i in range(len(state)) if state[i] == 0]
    first_try = True
    min_a = -1
    min_Q = 0
    for pos in empty_pos:
        if (state, pos) not in Q:
            Q[(state, pos)] = random.random() * 2 - 1
        if first_try:
            min_Q = Q[(state, pos)]
            min_a = pos
            first_try = False
        else:
            if Q[(state, pos)] < min_Q:
                min_Q = Q[(state, pos)]
                min_a = pos
    return min_a, min_Q


def show_state(self, state):
    for i in range(3):
        for j in range(3):
            print(state[i * 3 + j], end=" ")
        print()
    print()


def rem(self, array, val):
    for item in array:
        if item ==val:
            array.remove(item)


def QAgent(self, it):
    for i in range(it):
        current_state = (0, 0, 0, 0, 0, 0, 0, 0, 0)
        while self.endgame(current_state) == 0:

            # O Moves (1's Turn)
            turn = 1
            next_state = list(current_state)
            mov = self.find_mov(current_state, turn)
            next_state[mov] = turn
            m_a, m_Q = self.min_action(self.Q, next_state)
            if (current_state, mov) not in self.Q:
               self.Q[(current_state, mov)] = random.random() * 2 - 1
            self.Q[(current_state, mov)] = self.Q[(current_state, mov)] + self.alpha *
(self.reward(next_state) + self.gamma * m_Q - self.Q[(current_state, mov)])
            current_state = tuple(next_state)

            if self.endgame(current_state) != 0:
               break

            current_state = tuple(next_state)

      #        X Moves (2's Turn)
            turn = 2
            next_state = list(current_state)
            mov = self.find_mov(current_state, turn)
            next_state[mov] = turn
            m_a, m_Q = self.max_action(self.Q, next_state)
            if (current_state, mov) not in self.Q:
               self.Q[(current_state, mov)] = random.random() * 2 - 1
            self.Q[(current_state, mov)] = self.Q[(current_state, mov)] + self.alpha *
(self.reward(next_state) + self.gamma * m_Q - self.Q[(current_state, mov)])

            current_state = tuple(next_state)

            if self.endgame(current_state) != 0:
               break

            current_state = tuple(next_state)
```

# Code for Connect4 MinMax

```
class Connect4:
   def __init__(self):
      self.player, self.opponent = 'o', 'x'

   def printboard(self, board):
      rows = len(board)
      cols = len(board[0])
      board = [[' ' if x == '_' else x for x in row] for row in board]
      for row in range(len(board)):
         print('|', end=' ')
         for col in range(len(board[row])):
            print('{} |'.format(board[row][col]), end=' ')
         print()
      print()
```

```
    def isMovesLeft(self, board):
        rows = len(board)
        cols = len(board[0])
        for i in range(rows):
            for j in range(cols):
                if board[i][j] == '_':
                    return True
        return False

    def check_winner(self, board):
        rows = len(board)
        cols = len(board[0])

        for i in range(rows):
            for j in range(cols - 3):
                if (board[i][j] != '_' and
                    board[i][j] == board[i][j+1] == board[i][j+2] == board[i][j+3]):
                    return board[i][j]

        for j in range(cols):
            for i in range(rows - 3):
                if (board[i][j] != '_' and
                    board[i][j] == board[i+1][j] == board[i+2][j] == board[i+3][j]):
                    return board[i][j]

        for i in range(rows - 3):
            for j in range(cols - 3):
                if (board[i][j] != '_' and
                    board[i][j] == board[i+1][j+1] == board[i+2][j+2] == board[i+3][j+3]):
                    return board[i][j]

        for i in range(3, rows):
            for j in range(cols - 3):
                if (board[i][j] != '_' and
                    board[i][j] == board[i-1][j+1] == board[i-2][j+2] == board[i-3][j+3]):
                    return board[i][j]

        return None


    def evaluate(self, board):

#         self.opponent = 'o' if self.player == 'x' else 'x'
        rows = len(board)
        cols = len(board[0])

        for i in range(rows):
```

```
        for j in range(cols - 3):
            if (board[i][j] != '_' and
                board[i][j] == board[i][j+1] == board[i][j+2] == board[i][j+3]):
                if board[i][j]==self.player:
                    return 10
                elif board[i][j]==self.opponent :
                    return -10

    for j in range(cols):
        for i in range(rows - 3):
            if (board[i][j] != '_' and
                board[i][j] == board[i+1][j] == board[i+2][j] == board[i+3][j]):
                if board[i][j]==self.player:
                    return 10
                elif board[i][j]==self.opponent :
                    return -10

    for i in range(rows - 3):
        for j in range(cols - 3):
            if (board[i][j] != '_' and
                board[i][j] == board[i+1][j+1] == board[i+2][j+2] == board[i+3][j+3]):
                if board[i][j]==self.player:
                    return 10
                elif board[i][j]==self.opponent :
                    return -10

    for i in range(3, rows):
        for j in range(cols - 3):
            if (board[i][j] != '_' and
                board[i][j] == board[i-1][j+1] == board[i-2][j+2] == board[i-3][j+3]):
                if board[i][j]==self.player:
                    return 10
                elif board[i][j]==self.opponent :
                    return -10

    return 0

def getValidRow(self, board, column):
    row = 6
    while row >= 0:
        if board[row][column] == '_':
            break
        row -= 1
    if row < 0:
        return -1
    return row

def minimax(self, board, depth, alpha, beta, isMax, isPrun):
    rows = len(board)
```

```
        cols = len(board[0])
        score = self.evaluate(board)

        if score == 10:
            return score

        if score == -10:
            return score

        if self.isMovesLeft(board) == False:
            return 0

        if depth == 0:
            return 0

        if isMax:
            best = -1000
            for j in range(cols):
                i = self.getValidRow(board, j)
                if i != -1:
                    board[i][j] = self.player
                    best = max(best, self.minimax(board, depth - 1, alpha, beta, not isMax,
isPrun))
                    board[i][j] = '_'
                    if not isPrun:
                        alpha = max(alpha, best)
                        if beta <= alpha:
                            break
            return best

        else:
            best = 1000
            for j in range(cols):
                i = self.getValidRow(board, j)
                if i != -1:
                    board[i][j] = self.opponent
                    best = min(best, self.minimax(board, depth - 1, alpha, beta, not isMax,
isPrun))
                    board[i][j] = '_'
                    if not isPrun:
                        beta = min(beta, best)
                        if beta <= alpha:
                            break
            return best



    def get_best_move(self, board, depth, isPrun):
        rows = len(board)
        cols = len(board[0])
        bestVal = -1000
        bestMove = (-1, -1)
        alpha = -1000
        beta = 1000
        for j in range(cols):
            i = self.getValidRow(board, j)
            if i != -1:
                board[i][j] = self.player
                moveVal = self.minimax(board, depth - 1, alpha, beta, False, isPrun)
                board[i][j] = '_'



                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal
                if not isPrun:
                    alpha = max(alpha, bestVal)
                    if beta <= alpha:
                        break

        return bestMove
```

# Code for Connect4 QLearning

```
import numpy as np
import random

def endgame(state):
    rows = 7
    cols = 6
    check = [1, 2] # players are represented by 1 and 2
    for p in check:
        # check for horizontal win
        for r in range(rows):
            for c in range(cols - 3):
                if state[r*cols+c] == p and state[r*cols+c+1] == p and state[r*cols+c+2] == p
and state[r*cols+c+3] == p:
                    return p
        # check for vertical win
        for r in range(rows - 3):
            for c in range(cols):
                if state[r*cols+c] == p and state[(r+1)*cols+c] == p and state[(r+2)*cols+c] == p
and state[(r+3)*cols+c] == p:
                    return p
        # check for diagonal win (positive slope)
        for r in range(rows - 3):
```

```
        for c in range(cols - 3):
            if state[r*cols+c] == p and state[(r+1)*cols+c+1] == p and state[(r+2)*cols+c+2]
== p and state[(r+3)*cols+c+3] == p:
                return p
        # check for diagonal win (negative slope)
        for r in range(3, rows):
            for c in range(cols - 3):
                if state[r*cols+c] == p and state[(r-1)*cols+c+1] == p and state[(r-2)*cols+c+2]
== p and state[(r-3)*cols+c+3] == p:
                    return p
    # check if the board is full
    if all(x != 0 for x in state):
        return -1
    # if no winner and the board is not full, the game is still ongoing
    return 0


# In[77]:


def find_mov(state, turn):
    side = 1 if turn == 1 else -1
    empty_pos = [i for i in range(len(state)) if state[i] == 0]
    if random.random() < epsilon:
        first_try = True
        for pos in empty_pos:
            if (state, pos) not in Q:
                Q[(state, pos)] = random.random() * 2 - 1
            if first_try:
                max_Q = side * Q[(state, pos)]
                max_a = pos
                first_try = False
            else:
                if side * Q[(state, pos)] > max_Q:
                    max_Q = side * Q[(state, pos)]
                    max_action = pos
        return pos
    return random.choice(empty_pos)


# In[78]:


def reward(state):
    ret = endgame(state)
    if ret == 1:
        return 1000
    if ret == 2:
        return -1000
    return 0


# In[79]:


def available_moves(board):
    moves = []
    rows = 7
    cols = 6
    for i in range(cols):
        for j in range(rows-1,-1,-1):
            if board[(j*cols) + i] == 0:
                moves.append((j*cols) + i)
                break
    return moves


# In[80]:


def max_action(Q, state):
    state = tuple(state)
    empty_pos = available_moves(state)
    first_try = True
    max_action = -1
    max_Q = 0
    for pos in empty_pos:
        if (state, pos) not in Q:
#             print('sd')
            Q[(state, pos)] = random.random() * 2 - 1
        if first_try:
            max_Q = Q[(state, pos)]
            max_action = pos
            first_try = False
        else:
            if Q[(state, pos)] > max_Q:
                max_Q = Q[(state, pos)]
                max_action = pos
    return max_action, max_Q


# In[81]:


def min_action(Q, state):
    state = tuple(state)
    empty_pos = available_moves(state)
    first_try = True
    min_a = -1
    min_Q = 0
    for pos in empty_pos:
        if (state, pos) not in Q:
            Q[(state, pos)] = random.random() * 2 - 1
        if first_try:
            min_Q = Q[(state, pos)]
            min_a = pos
            first_try = False
        else:
            if Q[(state, pos)] < min_Q:
                min_Q = Q[(state, pos)]
                min_a = pos
    return min_a, min_Q


# In[82]:


def show_state(state):
    for i in range(3):
        for j in range(3):
            print(state[i * 3 + j], end=" ")
        print()
    print()


# In[83]:


def QAgent(alpha, gamma, epsilon, itera):
    Q = dict()
    for i in range(itera):
        current_state = (0,0,0,0,0,0,
             0,0,0,0,0,0,
             0,0,0,0,0,0,
             0,0,0,0,0,0,
             0,0,0,0,0,0,
             0,0,0,0,0,0,
             0,0,0,0,0,0)

        while endgame(current_state) == 0:
            # O Moves (1's Turn)
            turn = 1
            next_state = list(current_state)
            mov = find_mov(current_state, turn)
            next_state[mov] = turn
            m_a, m_Q = min_action(Q, next_state)
            if (current_state, mov) not in Q:
```

```
            Q[(current_state, mov)] = random.random() * 2 - 1
        Q[(current_state, mov)] = Q[(current_state, mov)] + alpha * (
            reward(next_state) + gamma * m_Q - Q[(current_state, mov)]
        )
        current_state = tuple(next_state)
        if endgame(current_state) != 0:
            break
        current_state = tuple(next_state)

        # X Moves (2's Turn)
        turn = 2
        next_state = list(current_state)
        mov = find_mov(current_state, turn)
        next_state[mov] = turn
        m_a, m_Q = max_action(Q, next_state)
        if (current_state, mov) not in Q:
            Q[(current_state, mov)] = random.random() * 2 - 1
        Q[(current_state, mov)] = Q[(current_state, mov)] + alpha * (
            reward(next_state) + gamma * m_Q - Q[(current_state, mov)]
        )
        current_state = tuple(next_state)
        if endgame(current_state) != 0:
            break
        current_state = tuple(next_state)
    return Q
```