CS7IS2 Artificial Intelligence Assignment 2

Sai Bala Subrahmanya Lakshmi kanth Rayanapati
School of Computer Science and Statistics
Computer Science MCS
Trinity College Dublin
Dublin, Ireland
Student Id: 19304511
Email: kanthras@tcd.ie

I. Introduction

Reinforcement learning (RL) in games is widely explored to improve decision-making processes, strategy development, and algorithm efficiency. This paper focuses on implementing various AI player algorithms for two well-known games: Tic Tac Toe and Connect Four in Python. The performance of the algorithms Minimax (without Alpha Beta Pruning), Minimax (with Alpha Beta Pruning), and Q Learning are compared against each other and against a default agent that performs better than a fully random opponent.

II. Game Environment

A. Tic Tac Toe

For the implementation of the TicTacToe game, I used an open-source TicTacToe library (python-tictactoe) [1]. This library offers various useful functions for setting up and validating the game board. The "python-tictactoe" library offers a suite of functions tailored for managing and interacting with the game board:

- `Board()` is used to create a game board of our preferred dimensions (3x3 in this scenario).
- `Board.copy()` allows the creation of an identical copy of the current game board.
- `Board.turn()` returns the current player of the board (1 represents player X and 2 represents player O).
- `Board.possible_moves()` returns a list of all the possible moves for the current state of the board.
- `Board.push()` places the current player's game piece on the game board based on the specified position passed.
- `Board.set_mark()` allows to place a game piece on the game board based on the player mark (X or O) passed.
- `Board.result()` returns the current status of the board. Returns "None" if the game is still active and no terminal state is reached. Returns "1" if player1 (X) wins, "2" if player2 (O) wins and "0" if the game is a draw.

B. Connect 4

The implementation of the connect4 board is based on the "Board()" generated by python-tictactoe with significant modifications in the environment to suit the game of Connect4. In contrast to the original TicTacToe implementation, which identifies the winning states by checking 3 pieces in a row, the modified board checks for 4 pieces in a row adjusting the board size to the Connect4 standard of 7 columns by 6 rows.

- `Board.possible_moves()` is modified to return the list of columns where a move is possible.
- `Board.push()` is altered to accept a column number as its input to place the player piece and place the piece at an empty position in that specified column.
- `Board.set_markc4()` is a variation of `Board.set_mark()` that takes the column input to place a player mark (X or O) accordingly.
- `Board.get_column()` returns a list of the values of the player pieces placed in the specified column.
- `Board.get_row()` returns a list of the values of the player pieces placed in the specified row.

III. Agents

A. Random Agent

The random agent samples all the possible moves for a given board state. It then selects and returns a random move from the possible moves collection of the board.

B. Default Agent

The default agent is designed in a way to return a winning move if one exists, or it blocks the winning move of the opponent if it exists otherwise, it returns a random move.

IV. Algorithms

A. Minimax Algorithm without Alpha Beta Pruning

Minimax is a backtracking algorithm that is used in decision making and game trees to find the most optimal move for a player. In a Minimax algorithm there are 2 players called the "maximizer" and the "minimizer player". The maximizer tries to get the highest score possible while the minimizer tries to get the lowest score possible. [2]

My implementation of this code is inspired by [3]. The `minimax_best_move()` function assesses the current game board, a Boolean indicating whether it's the maximising player's turn, and the AI player as the input parameters. It then calculates the score for all the possible states by calling

the `minimax` function at each of the possible states. The `minimax` first checks for the terminal states and assigns a score of +10 for player wins, -10 for player loses and a 0 for a drawn game. Inside the `minimax` function, the `minimax` function is called recursively for all the possible states of the board and appends the scores to a list. If the player is the maximizer player, `minimax` returns the maximum score from the list otherwise it returns the minimum score from the list. Finally, the `minimax_best_score()` returns the move associated with the highest score as the next move for the AI player.

B. Minimax Algorithm with Alpha Beta Pruning

Alpha beta pruning is a modified version of the minimax algorithm used in game trees to reduce the number of nodes evaluated, making the search process more efficient. It achieves the efficiency by using two parameters, alpha and beta, which represent the minimum and maximum score that a maximizing player is guaranteed. These values are updated during the search to narrow the search space.

Following a similar structure to the implementation of the minimax algorithm (without alpha beta pruning), the `minimax_best_move()` recursively calls the `minimax` function. The updated `minimax` function takes additional parameters like maximum search depth, alpha and beta values. Coming to the logic of the algorithm, in addition to checking for terminal states, in the minimax algorithm (with alpha beta pruning) the depth of the search is also checked to limit the game's search space. If the search depth is reached before the game reaches a terminal state, the score is calculated using an evaluation function called `score_position()` that returns score based on the completeness of the game windows (4 adjacent pieces). Further by updating alpha and beta values, the best score of the board is returned in a much quicker way when compared to the minimax algorithm (without alpha beta pruning).

For the evaluation function, I have tried different approaches. The first approach I tried was hardcoding the values for different states of the board inspired by the approach [4]. However, the performance of the boat was not particularly impressive. So, inspired from the approach [5], I created a simple evaluation function that searches the board in windows of size 4 checking for possible terminal states and assigning scores based on the number of moves required to get to a terminal state. This approach, especially prioritises the centre column, aligns with common winning strategies and can contribute towards a win in most scenarios.

1) Tic Tac Toe: Due to the game tree size for Tic Tac Toe being small, Alpha-Beta pruning is utilised directly without the implementation of a depth-limited search. The minimax algorithm, with the help of Alpha-Beta pruning, effectively returns the best score by updating the Alpha and beta values. Alpha value is set to -infinity and Beta is set to infinity.

2) Connect 4: The scenario for implementing minimax algorithm for Connect4 presents a different situation due to its larger dimensions of 7x6 game board, resulting in a significantly expanded search space. To address this complexity, I implemented a logic that includes depth-limited search alongside an evaluation function. This approach allows the algorithm to manage a larger number of possible game states more efficiently by limiting the search depth to a reasonable level.

C. Tabular Q Learning Algorithm

QLearning [6] is a model-free reinforcement learning algorithm designed to learn the value of an action in a particular state. It does not require a model of the environment and can handle problems with stochastic transitions and rewards without requiring adaptations.

The core of the QLearning algorithm is the Q-function, $Q : \text{State} \times \text{Action} \rightarrow R$, which gives the value of taking an action a in a state s. The Q-function is updated as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- s' is the state following state s after action a is taken.
- r is the reward received after moving from s to s'.
- α is the learning rate.
- γ is the discount factor.

The Q-Learning agent updates its policy to choose the action with the highest Q-value in each state, with exploration managed by an ϵ-greedy strategy.

In my Q-learning implementation for a game-playing agent, I started by initializing the Q-learning agent with parameters such as the exploration rate (epsilon), learning rate (alpha), and discount factor (gamma). For my experiments, I set alpha = 0.3, gamma = 0.9 and epsilon = 0.9 with the value of the epsilon beginning at 0.9 and then reducing by a decay rate of 10% every 2000 or 5000 episodes.

Key functions include the `get_rewards()` function that calculates the rewards based on the game states. And the `choose_action()` function that selects an action based on the epsilon-greedy strategy. The `minmax()` function is a helper function for `choose_action()` that returns a move based on maximizing or minimizing. The `state_to_str()` function returns the unique state of the board as a list. The `init_QTable()` function returns the current state of the board creating a Q-table entry for the current state and initializes it to 0. The `update_q_values()` function updates the Q-table based on the observed transition from state to next_state with action and reward. It uses the Q-learning update rule to update the Q-value for the state-action pair. Finally, `get_q_table()` and `set_q_table()` functions are used for retrieving and updating the Q-table, respectively.

1) Tic Tac Toe: The Q Learning agent for Tic Tac Toe is trained for 200,000 episodes with the epsilon value decaying by 10% every 5000 episodes. The trained model is stored in a file called `"QTable_TicTacToe.pkl"`.

2) Connect 4: As the search space of Connect 4 is much bigger, the training process took a longer time. Considering the processing power of my laptop, I could only train the

agent for 10,000 episodes with the epsilon value decaying by 10% every 2000 episodes. The trained model is stored in a file called "QTable_connect4.pkl".

V. Results

A. Tic Tac Toe

1) ***How do your algorithms compare to each other when playing against the default opponent in Tic Tac Toe****:* Below are the results of running Minimax (without alpha beta pruning), Minimax (with alpha beta pruning), and Q learning agent for 100 games against the default player.

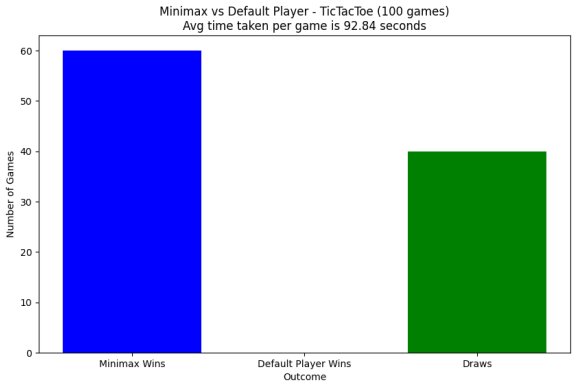

Fig. 1. Minimax (Without alpha beta pruning) vs Default - Tic Tac Toe

The above figure is the output of Minmax (without alpha beta pruning) algorithm playing against a default agent for 100 games. We can observe that the Minimax algorithm only won or drawn the game but never lost the game against the default player. This trend is followed even when the number of games are increased. It is worth nothing that even though this algorithm is not losing a game, each game takes an average of 92.84 seconds to complete.

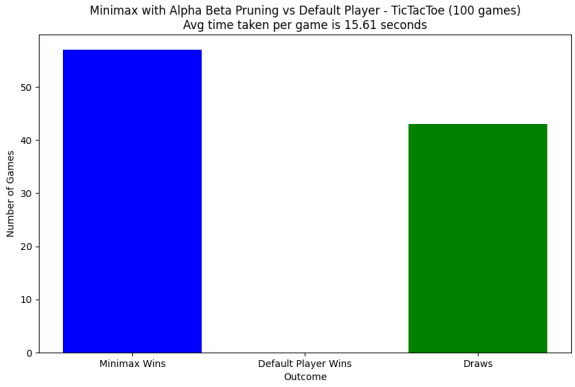

Fig. 2. Minimax (With alpha beta pruning) vs Default - Tic Tac Toe

The above is the output of the updated minimax algorithm, Minmax (with alpha beta pruning), playing against a default player for 100 games. We can observe that the performance (in terms of wins) of this algorithm is similar to that of Minmax (without alpha beta pruning) but one big difference is that this updated minimax with alpha beta pruning takes significantly lesser time (average of 15.61 seconds per game) when compared to the minimax without alpha beta pruning.

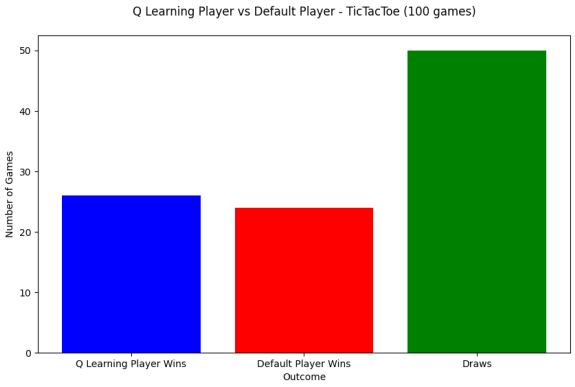

Fig. 3. Q Learning vs Default - Tic Tac Toe

The above is the output of 100 games played by the Q Learning agent against the default agent. Interestingly we can notice here that some of the games were won by the default player. This indicates that more training is required for the Q learning player to outperform the default player.

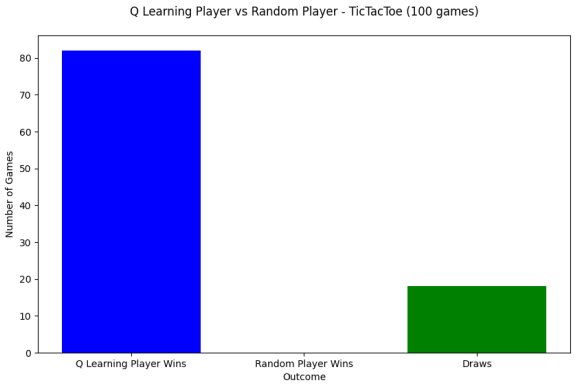

Fig. 4. Q Learning vs Random - Tic Tac Toe

The above is the output of 100 games played by the Q Learning agent against the random agent. Here, the Q Learning agent doesn't lose a single game against the random player highlighting that the agent is learning properly but requires more training to outperform the default player.

2) ***How do your algorithms compare to each other when playing against each other in Tic Tac Toe****:* As the performance of both the minimax algorithms is very similar in terms of the win percentage, I chose to play the Q Learning agent only against the Minimax algorithm with alpha beta pruning as it was much quicker to compute.

Below are the results of 100 games played between the Q Learning and the Minimax agents. It is clear from the results that Minimax has the upper hand when compared to the Q

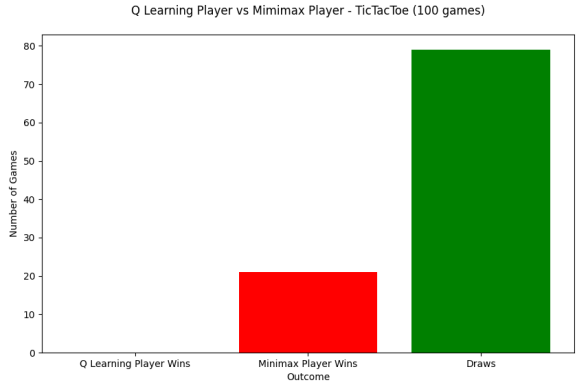

Fig. 5. Q Learning vs Minimax - Tic Tac Toe

Learning agent. However, most of the games ended as a draw indicating that more training of the Q Learning agent will improve its performance.

B. Connect 4

1) ***How do your algorithms compare to each other when playing against default opponent in Connect 4****:* Considering the large search space of the connect 4 game and the time taken to play each game of Connect 4, I decided to evaluate the results based on the performance in 10 games where minimax algorithm is involved. Additionally, the Q Learning agent is trained only for 10,000 episodes due to time and hardware constraints.

Note: Given the larger game space that the minimax algorithm without Alpha-Beta pruning needs to search, it is understandable that the process is time-consuming. As a result of these challenges to run such simulations, the graphs depicting the performance or outcomes of this algorithm have not been generated.

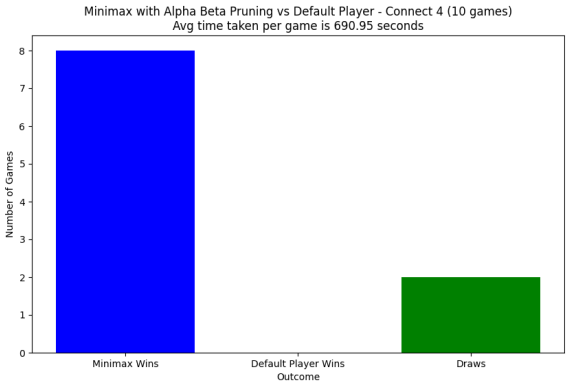

Fig. 6. Minimax (With alpha beta pruning) vs Default - Connect 4

The above are the result of 10 games played between Minimax (with alpha beta pruning) and the default agent. As expected, the Minimax algorithm outperforms the default agent with ease. It is worth noting that each game takes an average of 690.95 seconds to play. The reason being the search space is huge for a connect 4 game. Scaling up the games would still result in similar results.

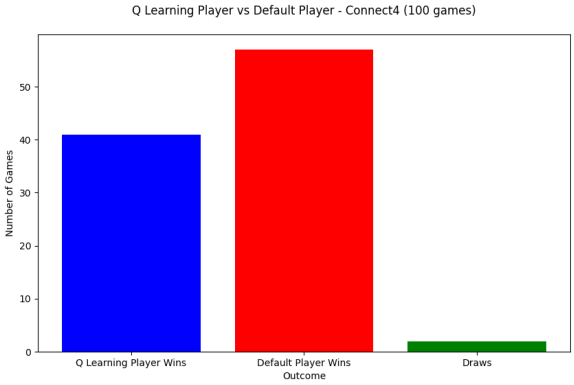

Fig. 7. Q Learning vs Default - Connect 4

The above are the result after playing 100 games between Q Learning agent and the default agent. As the Q learning agent is not completely trained, the default player outperforms the Q learning agent. Increasing the number of games in the training will improve the performance of the Q learning agent.

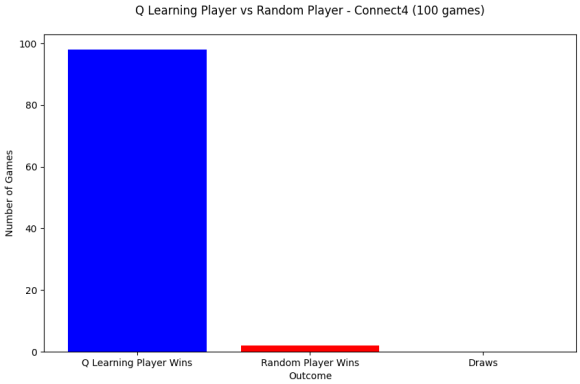

Fig. 8. Q Learning vs Random - Connect 4

The above are the result of playing 100 games between Q learning agent and the random player. As we can observe, from the limited training, the Q Learning agent performs extremely well against the random player. Increasing the no of training episodes will further help in the improvement of the Q learning agent.

2) ***How do your algorithms compare to each other when playing against each other in Connect 4****:* Below are the results of Q Learning agent playing 10 games against the minimax agent. As expected, the minimax agent outperforms the Q Learning agent. A more well trained Q Learning agent will help in reducing the number of wins of the minmax agent and increasing the number of draws. Scaling the total games played will provide with a similar result as the Q learning agent is only trained for 10,000 episodes.

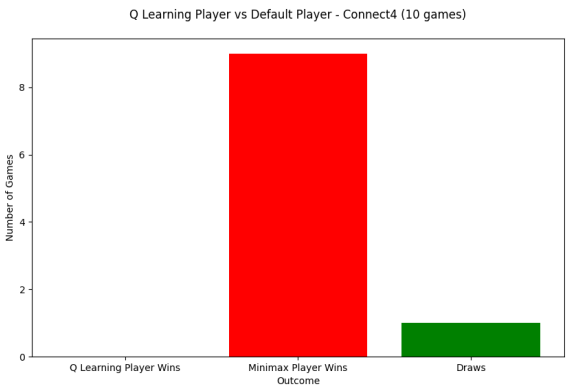

Fig. 9. Q Learning vs Minimax (With alpha beta pruning) - Connect 4

C. Overall

1) ***How do your algorithms compare to each other when playing against default opponent overall****:* By comparing all the results, we can say that minimax algorithm with Alpha-beta pruning is the best performing (win percentage) agent against the default agent in both the games. However, the time taken by minmax algorithm is very high. One solution to this problem can be a well trained Q Learning agent. When trained against a high number of episodes, the performance of the Q Learning agent will improve and the time taken to train will still be less than that of minimax algorithm.

2) ***How do your algorithms compare to each other when playing against each other overall****:* When comparing the performance of Q Learning and Minimax agents against each other in games like TicTacToe and Connect4, it is evident that Minimax consistently has the upper hand in terms of the number of games won. This is primarily because in both TicTacToe and Connect 4 games, the number of episodes the Q Learning agent was trained for is much lesser than the total unique possible games. Training the agents to cover all the possible states would require a lot of computational power and time.

Similarly, despite the efforts to optimize the Minimax algorithm considering to reduce the search space in Mimimax by limiting the maximum depth and implementing Alpha-Beta pruning, Mimimax algorithm still requires significant time to determine the optimal move for each game situation.

Both the algorithms when required to play optimally, would require a significant time to play the game moves. Minimax algorithm, although takes a lot of time, outperforms the Q Learning agent in most of the scenarios as the Q Learning agent is not trained for all the states. As the Q Learning agent undergoes more training, its performance would increase, narrowing the gap with the Minimax algorithm.

VI. Conclusion

In conclusion, we can observe that Minimax algorithm with Alpha Beta Pruning consistently outperforms other algorithms in game wins but requires significant time and computational resources. While a well-trained Q Learning algorithm shows promise in environments with smaller search spaces, expanding these spaces significantly increases its time complexity. Thus, despite the efficiency of Minimax with Alpha Beta Pruning, the potential of Q Learning to challenge Minimax is evident with adequate training.

References

[1] "Python tictactoe package," https://pypi.org/project/python-tictactoe/, accessed: 06/04/2024.

[2] "Minimax algorithm in game theory — set 1 (introduction)," https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/, Year of Publication, accessed: Date of Access.

[3] "Mastering tic-tac-toe with minimax algorithm," https://levelup.gitconnected.com/mastering-tic-tac-toe-with-minimax-algorithm-3394d65fa88f, Year of Publication, accessed: Date of Access.

[4] "Alpha-beta pruning for connect 4," https://www.sciencebuddies.org/science-fair-projects/project-ideas/ArtificialIntelligence_p014/artificial-intelligence/alpha-beta-connect4, Access Year, accessed: Date of Access.

[5] K. Galli, "Connect4 with ai implementation," https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py#L85, Access Year, accessed: Date of Access.

[6] "Q-learning," https://en.wikipedia.org/wiki/Q-learning, Access Year, accessed: Date of Access.

Appendix

A. Minimax.py

```
import math


# Code inspired from https://levelup.gitconnected.
    com/mastering-tic-tac-toe-with-minimax-algorithm
    -3394d65fa88f
def minimax(board, is_maximizer_turn, ai_player):
    game_result = board.result()

    if game_result is not None:
        # Tie
        if game_result == 0:
            return 0

        if ai_player == 1:
            # Win
            if game_result == 1:
                return 10
            # Loose
            elif game_result == 2:
                return -10

        if ai_player == 2:
            # Loose
            if game_result == 1:
                return -10
            # Win
            elif game_result == 2:
                return 10

    scores = []

    for move in board.possible_moves():
        board_copy = board.copy()
        board_copy.push(move)
        scores.append(minimax(board_copy, not
            is_maximizer_turn, ai_player))

    if is_maximizer_turn:
        return max(scores)
```

```
    else:
        return min(scores)


def minimax_best_move(board, is_maximizer_turn,
    ai_player):
    best_score = -math.inf
    best_move = None

    for move in board.possible_moves():
        board_copy = board.copy()
        board_copy.push(move)
        score = minimax(board_copy,
            is_maximizer_turn, ai_player)

        if score > best_score:
            best_score = score
            best_move = move

    return best_move
```

B. Minimax_Alpha_Beta_TicTacToe.py

```
    import math


def minimax_alphabeta(board, is_maximizer_turn,
    ai_player, alpha, beta):
    game_result = board.result()

    if game_result is not None:
        # Tie
        if game_result == 0:
            return 0

        if ai_player == 1:
            # Win
            if game_result == 1:
                return 10
            # Loose
            elif game_result == 2:
                return -10

        if ai_player == 2:
            # Loose
            if game_result == 1:
                return -10
            # Win
            elif game_result == 2:
                return 10

    else:
        if is_maximizer_turn:
            best_score = -math.inf
            for move in board.possible_moves():
                board_copy = board.copy()
                board_copy.push(move)
                score = minimax_alphabeta(board_copy
                    , False, ai_player, alpha, beta)
                best_score = max(best_score, score)
                alpha = max(alpha, best_score)
                if beta <= alpha:
                    break
            return best_score
        else:
            best_score = math.inf
            for move in board.possible_moves():
                board_copy = board.copy()
                board_copy.push(move)
                score = minimax_alphabeta(board_copy
                    , True, ai_player, alpha, beta)
                best_score = min(best_score, score)
                beta = min(beta, best_score)
                if beta <= alpha:
                    break
            return best_score


def minimax_best_move(board, is_maximizer_turn,
    ai_player):
    best_score = -math.inf
    best_move = None

    for move in board.possible_moves():
        board_copy = board.copy()
        board_copy.push(move)
        score = minimax_alphabeta(board_copy,
            is_maximizer_turn, ai_player, -math.inf,
             math.inf)

        if score > best_score:
            best_score = score
            best_move = move

    return best_move
```

C. Minimax_Alpha_Beta_Connect4.py

```
import math


# Code inspired from https://levelup.gitconnected.
    com/mastering-tic-tac-toe-with-minimax-algorithm
    -3394d65fa88f and
# https://github.com/KeithGalli/Connect4-Python/blob
    /master/connect4_with_ai.py#L85
def minimax_alphabeta(board, depth,
    is_maximizer_turn, ai_player, alpha, beta):
    game_result = board.result()

    if depth == 0 or game_result is not None:
        # Tie
        if game_result == 0:
            return 0

        elif game_result == 1:
            if ai_player == 1:
                return 10
            elif ai_player == 2:
                return -10
        elif game_result == 2:
            if ai_player == 1:
                return -10
            elif ai_player == 2:
                return 10
        else:
            return score_position(board, ai_player)

    else:
        if is_maximizer_turn:
            best_score = -math.inf
            for move in board.possible_moves():
                board_copy = board.copy()
                board_copy.push(move)
                score = minimax_alphabeta(board_copy
                    , depth - 1, False, ai_player,
                    alpha, beta)
                best_score = max(best_score, score)
                alpha = max(alpha, best_score)
                if beta <= alpha:
                    break
            return best_score
        else:
            best_score = math.inf
            for move in board.possible_moves():
                board_copy = board.copy()
```

```
                board_copy.push(move)
                score = minimax_alphabeta(board_copy
                    , depth - 1, True, ai_player,
                    alpha, beta)
                best_score = min(best_score, score)
                beta = min(beta, best_score)
                if beta <= alpha:
                    break
            return best_score


def minimax_best_move(board, is_maximizer_turn,
    ai_player):
    best_score = -math.inf
    best_move = None

    for move in board.possible_moves():
        board_copy = board.copy()
        board_copy.push(move)
        score = minimax_alphabeta(board_copy, 6,
            is_maximizer_turn, ai_player, -math.inf,
             math.inf)

        if score > best_score:
            best_score = score
            best_move = move

    return best_move


def evaluate_window(window, player):
    score = 0
    opp_player = 2 if player == 1 else 1

    if window.count(player) == 4:
        score += 10
    elif window.count(player) == 3 and window.count
        (0) == 1:
        score += 5
    elif window.count(player) == 2 and window.count
        (0) == 2:
        score += 2

    if window.count(opp_player) == 3 and window.
        count(0) == 1:
        score -= 4

    return score


def score_position(board, player):
    score = 0

    ROW_COUNT = board.dimensions[1]
    COLUMN_COUNT = board.dimensions[0]

    # Score center column
    center_array = board.get_column(COLUMN_COUNT //
        2)
    center_count = center_array.count(player)
    score += center_count * 3

    # Score horizontal
    for r in range(ROW_COUNT):
        row_array = board.get_row(r)
        for c in range(COLUMN_COUNT - 3):
            window = row_array[c:c + 4]
            score += evaluate_window(window, player)

    # Score vertical
    for c in range(COLUMN_COUNT):
        col_array = board.get_column(c)
        for r in range(ROW_COUNT - 3):
            window = col_array[r:r + 4]
            score += evaluate_window(window, player)

    # Score positive sloped diagonal
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board.get_mark_at_position((c
                + i, r + i)) for i in range(4)]
            score += evaluate_window(window, player)

    # Score negative sloped diagonal
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board.get_mark_at_position((c
                + i, r + 3 - i)) for i in range(4)]
            score += evaluate_window(window, player)

    return score
```

D. Q_Learning_TicTacToe.py

```
import numpy as np
import random
from Agents.default_agent_tictactoe import
    default_agent as default_tictactoe


class QLearningAgent:
    def __init__(self, epsilon, alpha, gamma):
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.q_table = dict()

    def get_reward(self, board):
        game_result = board.result()
        if game_result is not None:
            if game_result == 1:
                return 10
            elif game_result == 2:
                return -10
            elif game_result == 0:
                return 1
        else:
            return 0

    def state_to_str(self, board):
        return str(board.board.flatten().tolist())

    def init_QTable(self, board):
        state = self.state_to_str(board)
        if state not in self.q_table:
            possible_moves = board.possible_moves().
                tolist()
            self.q_table[state] = {tuple(move): 0.0
                for move in possible_moves}
        return state


    def update_q_values(self, board, action):
        current_state = self.init_QTable(board)
        reward = self.get_reward(board)
        future_board = board.copy()
        future_board.push(action)
        future_state = self.init_QTable(future_board
            )
        old_q_value = self.q_table[current_state][
            action]

        if future_board.result() is not None:
            new_q_value = old_q_value + self.alpha *
                 (reward - old_q_value)
        else:
            future_q_values = self.q_table[
                future_state]
```

```
            if future_board.turn == 1:
                expected_q_values = max(
                    future_q_values.values())
            else:
                expected_q_values = min(
                    future_q_values.values())
            new_q_value = old_q_value + self.alpha *
                (reward + self.gamma *
                expected_q_values - old_q_value)

        self.q_table[current_state][action] =
            new_q_value

    def minmax(self, q_values, minmax):
        optimal_value = minmax(list(q_values.values
            ()))
        count = list(q_values).count(minmax)
        if count > 1:
            best_moves = [move for move in list(
                q_values.keys()) if q_values[move]
                == optimal_value]
            move = best_moves[np.random.choice(len(
                best_moves))]
        else:
            move = max(q_values, key=q_values.get)
        return move

    def choose_action(self, board):
        if random.uniform(0, 1) < self.epsilon:
            return tuple(default_tictactoe(board,
                board.turn))
        else:
            current_state = self.init_QTable(board)
            q_values = self.q_table[current_state]
            if board.turn == 1:
                return self.minmax(q_values, max)
            elif board.turn == 2:
                return self.minmax(q_values, min)

    def train(self, board, episodes):
        decay_rate = 0.9
        for episode in range(episodes):
            print("Episode no:", episode)
            board_copy = board.copy()
            if episode != 0 and episode % 5000 == 0:
                self.epsilon = self.epsilon *
                    decay_rate
                print(self.epsilon)
            while board_copy.result() is None:
                action = self.choose_action(
                    board_copy)
                self.update_q_values(board_copy,
                    action)
                board_copy.push(action)

    def get_q_table(self):
        return self.q_table

    def set_q_table(self, q_table):
        self.q_table = q_table
```

E. Q_Learning_Connect4.py

```
from Agents.default_agent_connect4 import
    default_agent as default_connect4
import numpy as np
import random


class QLearningAgent:
    def __init__(self, epsilon, alpha, gamma):
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.q_table = dict()

    def get_reward(self, board):
        game_result = board.result()
        if game_result is not None:
            if game_result == 1:
                return 10
            elif game_result == 2:
                return -10
            elif game_result == 0:
                return 1
        else:
            return 0

    def state_to_str(self, board):
        return str(board.board.flatten().tolist())

    def init_QTable(self, board):
        state = self.state_to_str(board)
        if state not in self.q_table:
            possible_moves = board.possible_moves()
            self.q_table[state] = {move: 0.0 for
                move in possible_moves}
        return state

    def update_q_values(self, board, action):
        current_state = self.init_QTable(board)
        reward = self.get_reward(board)
        future_board = board.copy()
        future_board.push(action)
        future_state = self.init_QTable(future_board
            )
        old_q_value = self.q_table[current_state][
            action]

        if future_board.result() is not None:
            new_q_value = old_q_value + self.alpha *
                (reward - old_q_value)
        else:
            future_q_values = self.q_table[
                future_state]
            if future_board.turn == 1:
                expected_q_values = max(
                    future_q_values.values())
            else:
                expected_q_values = min(
                    future_q_values.values())
            new_q_value = old_q_value + self.alpha *
                (reward + self.gamma *
                expected_q_values - old_q_value)

        self.q_table[current_state][action] =
            new_q_value

    def minmax(self, q_values, minmax):
        optimal_value = minmax(list(q_values.values
            ()))
        count = list(q_values).count(minmax)
        if count > 1:
            best_moves = [move for move in list(
                q_values.keys()) if q_values[move]
                == optimal_value]
            move = best_moves[np.random.choice(len(
                best_moves))]
        else:
            move = max(q_values, key=q_values.get)
        return move

    def choose_action(self, board):
        if random.uniform(0, 1) < self.epsilon:
            return default_connect4(board, board.
                turn)
        else:
```

```
        current_state = self.init_QTable(board)
        q_values = self.q_table[current_state]
        if board.turn == 1:
            return self.minmax(q_values, max)
        elif board.turn == 2:
            return self.minmax(q_values, min)

def train(self, board, episodes):
    decay_rate = 0.9
    for episode in range(episodes):
        print("Episode no:", episode)
        board_copy = board.copy()
        if episode != 0 and episode % 2000 == 0:
            self.epsilon = self.epsilon *
                decay_rate
            print(self.epsilon)
        while board_copy.result() is None:
            action = self.choose_action(
                board_copy)
            self.update_q_values(board_copy,
                action)
            board_copy.push(action)

def get_q_table(self):
    return self.q_table

def set_q_table(self, q_table):
    self.q_table = q_table
```