Assignment 3 – Adversarial search and RL
Module: CS7IS2-202324 Artificial Intelligence

Student Name: Indrajeet Kumar
Student Number: 23345983

1. Introduction

In this assignment, I will discuss the development and analysis of Minimax (both with and without alpha-beta pruning) and tabular Q-learning Reinforcement Learning algorithms applied to the games of Tic Tac Toe and Connect 4. I will also evaluate the performance of these algorithms in game-playing scenarios and comparing their effectiveness and efficiency. In the following section, I will explore the implementation of Tic Tac Toe and Connect 4 games, along with a detailed comparison of the algorithms when playing against a default opponent. Additionally, I will provide an overall comparison of the algorithms used across both games.

2. Tic Tac Toe

I. Human vs. computer with Minimax and Alpha-Beta Pruning

Firstly, I have implemented the game of tic tac toe where human competes against the computer. The computer utilizes the two variations of the minimax algorithm : one with and one without Alpha-Beta pruning. The main goal of this implementation is to show and compare and how well the minimax algorithm works both with and without the added improvement of Alpha-Beta pruning.

The game board is represented as a dictionary with keys from 1 to 9, corresponding to positions on a 3x3 grid. Initially, all positions are marked with a space (' ') indicating they are free as shown in figure 1.

```
board = {i: ' ' for i in range(1, 10)}
```

Figure 1: Initialization of the Tic Tac Toe game board

I am using '*printBoard(board)*` function which is used to visually represent the current state of the game board in the console. It organizes the board's dictionary values into a structured grid format. I have added the code snippet for '*printBoard(board)*` function which is shown in figure 2.

```
def printBoard(board):
    print("\nTic Tac Toe Board:")
    print(" " + board[1] + " | " + board[2] + " | " + board[3] + " ")
    print("---+---+---")
    print(" " + board[4] + " | " + board[5] + " | " + board[6] + " ")
    print("---+---+---")
    print(" " + board[7] + " | " + board[8] + " | " + board[9] + " ")
    print()
```

Figure 2 : Function to display the Tic Tac Toe game board

The core function of this tic tac toe game include '*spaceIsFree(position*)', which checks if a board position is unoccupied, '*insertLetter(letter, position*)', which places a marker and checks for a win or draw, '*checkForWin()'* , which identifies winning combinations and 'checkDraw()', which determines if the board is full and the game ends in a draw. Together, these functions manage gameplay and enforce game rules.

The computer's moves are determined using either a straightforward Minimax algorithm or minimax with Alpha-Beta pruning, which can be selected by the user at the start of the game.

- ***minimax****(board, depth, isMaximizing, call_count):* The *minimax* function is a recursive algorithm used in the tic tac toe game to calculate the best move. It updates a count each time it runs, checks if the game has a winner or is a draw, and returns a score based on these outcomes. If maximizing, it looks for the highest possible score by trying every possible move, updating the board, and calling itself as minimizing. It does the opposite when minimizing, trying to achieve the lowest score. After evaluating all options, it undoes the moves, resets the board, and returns the highest or lowest score along with the total count of recursive calls. This helps determine the best move at each step of the game.

- ***minimaxAlphaBeta****(board, depth, isMaximizing, alpha, beta, call_count):* This function is a refined version of the minimax algorithm used in tic tac toe, which is used to speed up the decision-making process by skipping unnecessary calculations. This function counts each recursive call and checks for a win or draw, returning specific scores if either occurs. For maximizing turns, it tries every possible move, updates the board, calls itself to anticipate the opponent's response, and updates the highest score found. If this score exceeds a certain limit, it stops checking further to save time. For minimizing turns, it similarly seeks the lowest possible score and stops early if the score is too low, ensuring efficiency. After all possible moves, the function returns the best score and the total count of recursive calls.

Figure 3 displays the results of a Tic Tac Toe game where a human competes against the computer. In this game, the human player is denoted by 'O', while 'X' represents the moves made by the computer.

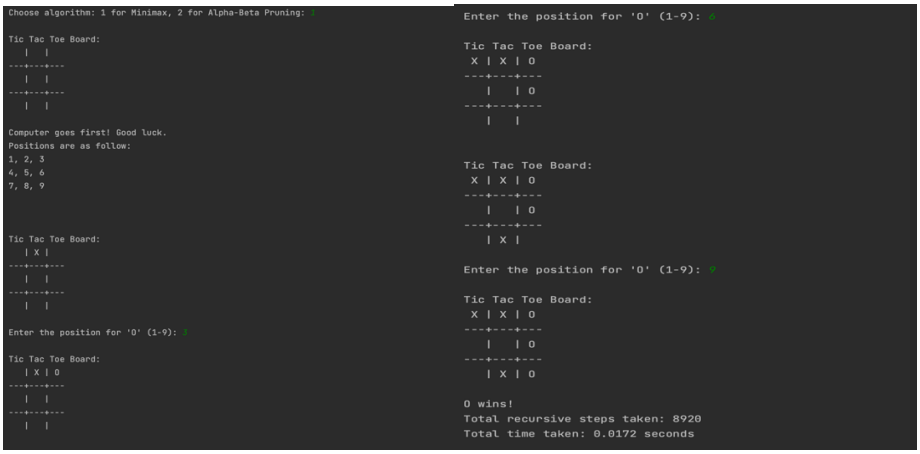

Figure 3 : Human vs. Computer with Minimax Algorithm

Algorithms	Total Time Taken	Total Recursive Steps
Minimax	0.0172 seconds	8920
Minimax with alpha beta pruning	0.0049 seconds	2278

Table 1 : Comparison between minimax and minimax with alpha beta pruning

We see a clear difference when we compare the two methods as seen in Table 1. The regular Minimax took 0.0172 seconds and made 8920 recursive steps to make a decision. When we used Minimax with Alpha-Beta pruning, it only took 0.0049 seconds and 2278 steps. The performance improvement is substantial , the version with Alpha-Beta pruning is almost 3.5 times faster and reduces the number of recursive calls by approximately 75%.

II. Default Player vs. computer with Minimax and Alpha-Beta Pruning

As explained in the above section, we have already discussed the tic tac toe game mechanics. In this section, we will discuss default AI opponent for the computer to play against when it's not the computer's turn. The default opponent is a simple AI designed to simulate a human player.

When it's the default AI opponent's turn in the game, the first thing we do is check if there's a move that can win the game immediately. If there is, the opponent will make that move to win. If not, we then check if we need to block the computer from winning on its next turn. If we find a spot where the computer could win, the opponent will play there to block the win.

If neither a winning nor a blocking move is available, the opponent uses a strategy called Minimax. This strategy helps the opponent think ahead by simulating future moves to figure out the best possible outcome. We set a limit on how deep this strategy can go to keep it from taking too long.

On the computer's side, the first move is chosen randomly to make the game start faster. For all other moves, the computer uses the Minimax strategy to decide the best move. This includes timing how long it takes and tracking how many moves it considers to help analyze how well the strategy is working.

The game goes on with each side taking turns until someone wins or all spaces are filled, leading to a draw.

Figure 4-5 displays the results of a tic tac toe game where a default player competes against the computer using regular minimax algorithm and minimax with alpha beta algorithm. In this game, the default player is denoted by ‘O’, while ‘X’ represents the moves made by the computer.

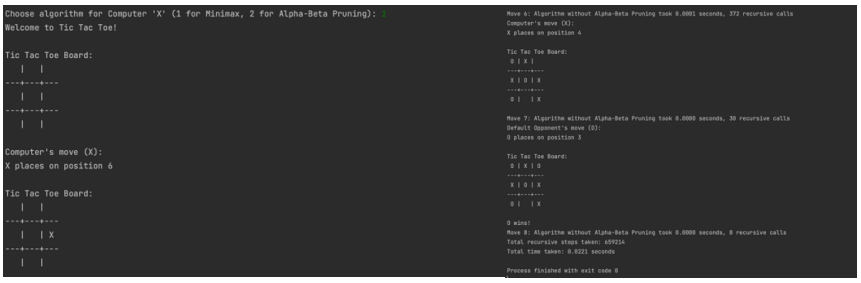

Figure 4 : *Default vs. Computer with Minimax Algorithm*

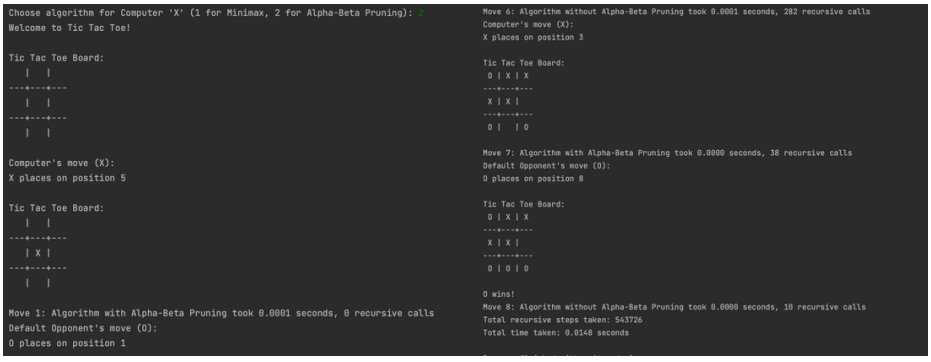

Figure 5 : Default vs. Computer with Minimax alpha beta pruning

Now, we will compare the performance of the Minimax algorithm and the Minimax algorithm with Alpha-Beta pruning when playing against the default opponent.

The Table 2 compares the performance of the Minimax algorithm versus Minimax with Alpha-Beta pruning in a series of Tic Tac Toe games against a default opponent. The metrics used for comparison include the total number of recursive steps taken and the total computational time required. As we can see from the table In all cases, Minimax with Alpha-Beta pruning consistently requires fewer recursive steps compared to the standard Minimax algorithm. This reduction in recursive steps illustrates the efficiency of Alpha-Beta pruning in eliminating unnecessary branches and reducing the search space. And also Time taken for computations is also consistently lower with Alpha-Beta pruning across all game scenarios. This indicates better performance, likely due to lesser computations. So, The Alpha-Beta pruning enhances the performance of the Minimax algorithm by reducing both the computation time and the number of recursive calls needed to reach a decision.

Case Description	Algorithm Type	Outcome	Recursive Steps	Time Taken (Seconds)
1st case	Minimax	O wins	677640	0.0223
1st Case	Minimax with Alpha beta	O wins	582972	0.0187
2nd case	Minimax	Draw	658210	0.0222
2nd case	Minimax with Alpha beta	Draw	564790	0.0186
3rd Case	Minimax	X wins	656476	0.0215
3rd Case	Minimax with Alpha beta	X wins	574908	0.0190

Table 2 : Performance comparison of Minimax and Minimax with Alpha-Beta Pruning in Tic Tac Toe

III. Default vs. computer with Q-learning Algorithms

In our Tic Tac Toe game, we start with both the Q-learning agent ('X') and the default opponent ('O') playing randomly to ensure a variety of initial game states. As games progress, I utilize a Q-table to guide 'X's decisions, updating this table based on game outcomes to refine the agent's strategy. This involves choosing moves at random to balance between exploiting known strategies and exploring new ones.

The default opponent, on the other hand, first looks for winning moves or blocks 'X' from winning if possible. If neither option is available, it employs a basic minimax strategy to decide its moves.

In the gameplay it alternates between 'X' and 'O' until one wins or the board fills up, resulting in a draw. Outcomes influence the Q-table adjustments, enhancing 'X's decision-making over time. To maintain continuity across sessions, we use the pickle library to save and reload the Q-table, ensuring 'X' builds on previous learning. This ongoing improvement helps the Q-learning agent become more skilled, aiming for a higher win rate against the less sophisticated default opponent.

Figure 6-8 displays the results of a Tic Tac Toe game where a default player competes against the computer using q-learning algorithm in various scenarios. In this game, the default player is denoted by 'O', while 'X' represents the moves made by the computer.

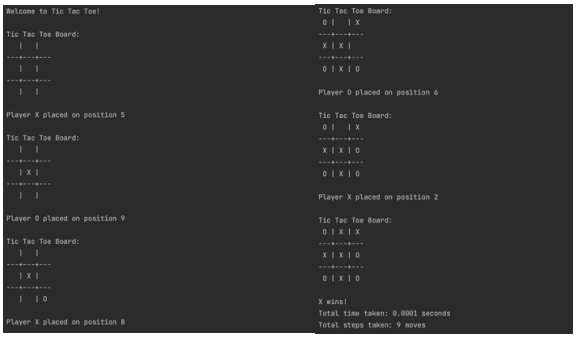

Figure 6 : Default Opponent vs. Computer Using Q-Learning Algorithm - Scenario Where the Computer Wins

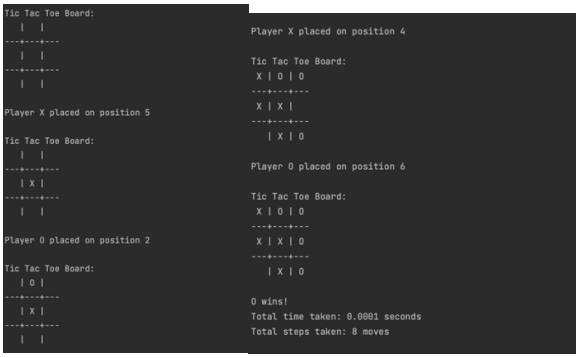

Figure 7 : Default Opponent vs. Computer Using Q-Learning Algorithm - Scenario Where the Default opponent Wins

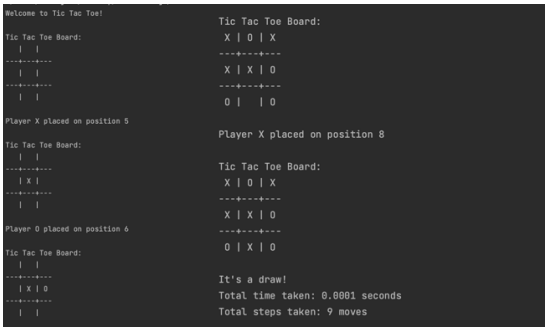

Figure 8 : Default Opponent vs. Computer Using Q-Learning Algorithm - Scenario Where it's Draw

Now, we will compare the performance of the Q-learning algorithm when playing against the default opponent in a series of 100 Tic Tac Toe games.

Outcome	Frequency	Average Time Taken	Average Steps Taken
Wins by 'X'	12	0.002 secs	7
Wins by 'O'	72	0.001 secs	6
Draws	16	0.0017 secs	9

Table 3 : Comparative Performance of Q-learning Algorithm vs. Default Opponent in 100 Tic Tac Toe Games

From the table 2 , we can see how well the Q-learning algorithm did against the default opponent in 100 Tic Tac Toe games. It won 12 times, taking about 0.002 seconds and 7 moves on average. The opponent won 72 times, taking about 0.001 seconds and 6 moves on average. There were 16 draws, taking about 0.0017 seconds and 9 moves on average. So, the Q-learning algorithm won fewer times than the opponent, but it managed to draw quite a few games.

IV. Overall Comparison of the Algorithms When Playing Against Default Opponent

When comparing the performance of different algorithms against the default opponent in Tic Tac Toe, Minimax without and with Alpha-Beta Pruning, and the Q-learning Algorithm yield the following results. I played 50 games of Minimax agent with and without Alpha-Beta Pruning against the default opponent, and 100 games of the Q-learning agent against the default opponent. These are the results I have noted down in the below table 4.

Algorithm	Wins by 'X'	Wins by 'O'	Draws	Average Time (secs)
Minimax	5 (out of 50)	43 (out of 50)	2 (out of 50)	0.0187-0.0223
Minimax with Alpha Beta pruning	7 (out of 50)	40 (out of 50)	3 (out of 50)	0.0186-0.0190
Q-learning Algorithm	12 (out of 100)	72 (out of 100)	16	0.001 – 0.002

Table 4 : Comparative Performance of Algorithms Against Default Opponent in Tic Tac Toe

Overall, Minimax with Alpha-Beta Pruning performed slightly better than Minimax without Alpha-Beta Pruning, with both algorithms winning fewer games than the Q-learning Algorithm against the default opponent in Tic Tac Toe. The Q-learning Algorithm achieved a significant number of wins by 'O' and draws, although it took longer on average compared to both Minimax variants.

3. Connect4

Connect 4 is a game where two players take turns dropping their symbols (here we are using 'X' and 'O') into a vertical board with several slots. The goal is to lines up four of your pieces in a row, either horizontally, vertically, or diagonally, before the opponent does. The game ends when a player achieves this or when all slots are filled and no one has won.

I. Human vs. computer with Minimax and Alpha-Beta Pruning

The Connect 4 game begins by asking the user to input the size of the board in terms of rows and columns. The user then chooses between two algorithms (Minimax or Minimax with Alpha-Beta pruning) for the computer's decision-making process. Lastly, the user decides who starts the game, either the human player or the computer.

Figure 9 displays the results of a connect 4 game where a human competes against the computer. In this game, the human player is denoted by 'X', while 'O' represents the moves made by the computer.

Figure 9 : *Human vs. Computer game in connect4 game using Minimax algorithm*

Algorithms	Total Time Taken	Total Recursive Steps
Minimax	0.28 seconds	2712
Minimax with alpha beta pruning	0.09 seconds	782

Table 5 : Comparison between minimax and minimax with alpha beta pruning in connect4 game

The table 5 shows the performance of two different algorithms when playing Connect 4, with the computer starting the game and the human playing second. The standard Minimax algorithm took 0.28 seconds and required 2712 recursive steps to complete the game. In contrast, the Minimax algorithm with Alpha-Beta pruning was faster and more efficient, taking only 0.09 seconds and needing just 782 recursive steps. So , the game runs faster and uses fewer resources when using Alpha-Beta pruning compared to the standard Minimax algorithm.

II. Default player vs. computer with Minimax and Alpha-Beta Pruning

In connect4 game, a default player represented by 'X' and a computer opponent denoted by 'O'. First, we check each column from the center outward to see if placing a token there would allow 'X' to win. This is done by simulating a token drop in each column and checking for four in a row. If a winning move is identified, 'X' will play in that column.

If there's no immediate winning move, we next look to block the computer from winning on its next move. This is achieved by simulating the computer's moves in the same way—dropping a token in each column and checking if it would result in a win for 'O'. If such a threat is detected, 'X' will place its token there to block the computer.

If neither a winning move nor a blocking move is necessary, 'X' simply places its token in the column closest to the center that is still open. This approach leverages the natural advantage of controlling the center in Connect 4, which facilitates access to the most possible four-in-a-row combinations.

When it's the computer's turn, its strategy varies based on the game's progress. If it's not the computer's first move, it uses either minimax or alpha-beta pruning to predict the best move by considering future possibilities and both players' potential actions. If it is the computer's first move, it randomly selects a column to ensure variability in the game.

And then We alternate turns between 'X' and 'O' until one player connects four tokens in a row or the board fills up, resulting in a draw. We also track the number of moves and the total time taken to offer insights into each player's strategy effectiveness.

Figure 10-11 displays the results of a connect4 game where a default player competes against the computer using regular minimax algorithm and minimax alpha beta algorithm. In this game, the default player is denoted by 'X', while 'O' represents the moves made by the computer.

Figure 10 : *Default vs. Computer in connect4 game using Minimax Algorithm*

Figure 11 : Default vs. Computer in connect4 game using Minimax Alpha beta Algorithm

Now, we will compare the performance of the Minimax algorithm and the Minimax algorithm with Alpha-Beta pruning in table 6 when playing against the default opponent in connect4 game, I have selected different board sizes and starting players.

Board Size	Starting Player	Algorithm	Time (Seconds)	Steps
4x5	Default	Minimax	0.09	1825
4x5	Default	Minimax with Alpha Beta Pruning	0.04	727
5x5	Default	Minimax	0.12	2561
5x5	Default	Minimax with Alpha Beta Pruning	0.04	818
6x7	Default	Minimax	0.60	14165
6x7	Default	Minimax with Alpha Beta Pruning	0.60	3365
4x5	Computer	Minimax	0.11	2322
4x5	Computer	Minimax with Alpha Beta Pruning	0.04	853
5x5	Computer	Minimax	0.12	2659

5x5	Computer	Minimax with Alpha Beta Pruning	0.06	1022
6x7	Computer	Minimax	0.65	15908
6x7	Computer	Minimax with Alpha Beta Pruning	0.21	3954

Table 6 : Performance Comparison of Minimax and Minimax with Alpha-Beta Pruning in Connect 4 Against Default Opponent

The comparison between the Minimax algorithm with and without Alpha-Beta pruning against the default opponent in Connect 4 shows that pruning significantly reduces the number of steps required to make decisions. For example, on a 4x5 board, steps decrease from 1,825 to 727, and on a 6x7 board, from 14,165 to 3,365. Time is also reduced on smaller boards, with the 4x5 game dropping from 0.09 to 0.04 seconds. However, on the 6x7 board, despite fewer steps, the time remains unchanged at 0.60 seconds. This indicates that Alpha-Beta pruning enhances efficiency, particularly in simpler or less complex scenarios.

III. Default player vs. computer with Q-learning Algorithms

In our Connect4 game, Player X, also known as the Q-agent, which utilizes a Q-learning algorithm to determine its moves. This algorithm stores values in a Q-table, storing the outcomes of previous moves. During gameplay, the Q-agent decides whether to experiment with new moves randomly or to stick with the best moves it has previously learned, depending on the *epsilon* setting.

After each move, the Q-agent evaluates the outcome and receives a reward based on the result—whether it's a win, a draw, or the game continues. We then update the Q-table with these rewards to help the agent learn which moves yield the best outcomes over time.

Our default opponent, Player O, focuses on securing wins by checking if any move can directly lead to a victory. If there's no immediate winning move, Player O shifts its strategy to block Player X from possibly winning in the next turn. This approach ensures that we always aim to either win or prevent a loss by making the most strategic move available based on the current state of the board.

Figure 12-14 displays the results of a connect4 game where a default player competes against the computer using q-learning algorithm in various scenarios. In this game, the default player is denoted by 'O', while 'X' represents the moves made by the Q-Agent.

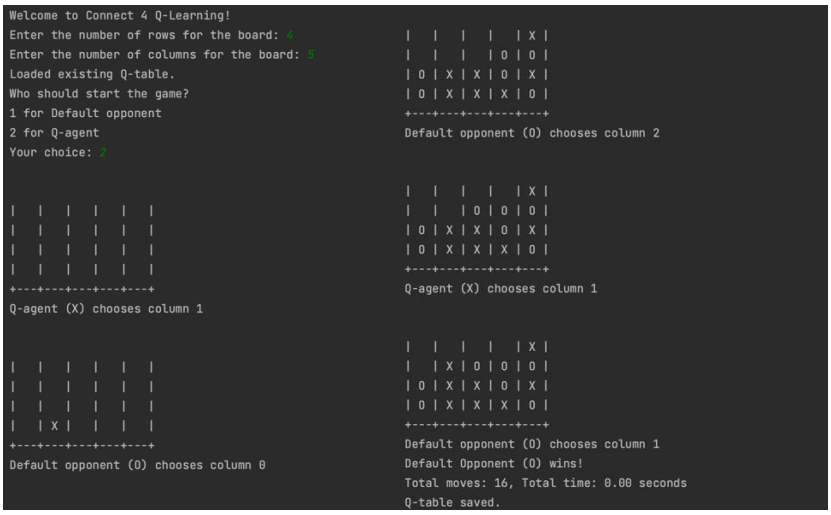

Figure 12 : *Default Opponent vs. Computer Using Q-Learning Algorithm - Scenario Where the default player Wins in connect4 game*

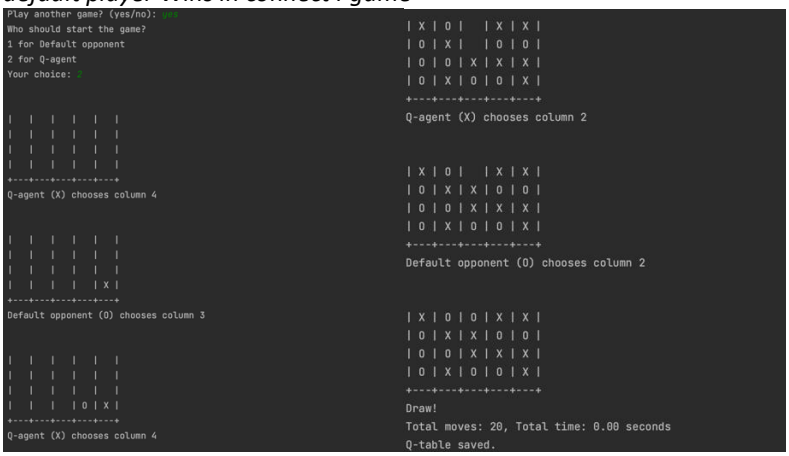

Figure 12 : *Default Opponent vs. Computer Using Q-Learning Algorithm – Scenario when it's draw in connect4 game*

Figure 13 : *Default Opponent vs. Computer Using Q-Learning Algorithm - Scenario Where the Q-agent player Wins in connect4 game*

Now, we will compare the performance of the Q-learning algorithm when playing against the default opponent in a series of 100 connect4 games.

Outcome	Frequency	Average Time Taken	Average Steps Taken
Wins by 'X'	7	0.01 secs	25
Wins by 'O'	84	0.01 secs	24
Draws	9	0.04 secs	28

Table 7 : Comparative Performance of Q-learning Algorithm vs. Default Opponent in 100 connect4 games

In a comparison of 100 Connect4 games, the default opponent won 84 times, significantly more than the 7 wins by the Q-learning agent ('X'). There were also 9 draws. Both types of players took about the same average time per move, 0.01 seconds, when they won, but draws took longer at 0.04 seconds per move. The average number of moves taken per game was slightly lower for the default opponent at 24 compared to 25 by 'X', with draws requiring 28 moves. So, the default opponent performs much better than the Q-learning agent.

IV. Overall Comparison of the Algorithms When Playing Against Default Opponent

In connect4 game, As we can see in table 6 , Minimax with Alpha-Beta Pruning consistently outperforms the standard Minimax algorithm in terms of efficiency. The pruning reduces the number of steps significantly across all board sizes. And from table 7 we can see that the Q-learning algorithm struggles against the default opponent. It wins only 7 out of 100 games, while the default opponent wins 84 times, and there are 9 draws. The average steps and time per game are similar for wins but increase for draws, indicating that more complex decision-making occurs in drawn games. When comparing the Q-learning algorithm with the Minimax approaches, the Minimax (especially with Alpha-Beta Pruning) shows a higher level of efficiency and effectiveness in terms of steps required to potentially win or draw the game. However, even with the optimized Minimax with Alpha-Beta Pruning, the default opponent still wins most of the time, suggesting it is a strong competitor. So , Overall the Minimax algorithm with Alpha-Beta Pruning performs the best in terms of efficiency against the default opponent in Connect 4. However, all algorithms still show challenges in outperforming the default opponent significantly in terms of overall game wins, with the Q-learning algorithm showing particular areas for potential improvement.

4. Overall comparison of Minimax (with and without alpha beta pruning) algorithms And Q-Learning Algorithms

In both the games, Tic Tac Toe and Connect 4, the Minimax algorithm with alpha-beta pruning consistently outperforms the standard Minimax, as shown in Tables 1 and 5. It significantly reduces both time and recursive steps, making it faster and more efficient. For example, in Tic Tac Toe, the alpha-beta version reduces steps from 8920 to 2278 and time from 0.0172 seconds to 0.0049 seconds as shown in Table 1.

The Q-learning algorithm, while efficient in processing games quickly, does not win as often. In Connect 4, it won only 7 out of 100 games, as highlighted in Table 7, indicating it might need adjustments to compete better against strategic opponents.

Overall, Minimax with alpha-beta pruning is more effective in these strategic games, showing better performance against default opponents compared to Q-learning, which, despite its speed, struggles to secure wins, as detailed in Tables 3 and 7.

5. Conclusion and Future work

In this assignment , we examined how different strategies perform in tic tac toe and Connect 4 games, focusing on Minimax, Minimax with Alpha-Beta Pruning, and Q-learning. The results show that Minimax with Alpha-Beta Pruning is the most effective, significantly reducing the time and steps needed to make decisions, as seen in Tables 1 and 5. This approach works very well against opponents in both games.

On the other hand, Q-learning, while fast, does not win as many games. It needs improvement to perform better in complex games like Connect 4, as indicated in Tables 3 and 7.

In summary, Alpha-Beta Pruning is the best method we tested because it's both quick and effective. This study helps us understand which game-playing strategies are most successful and where improvements are needed.

For future work, we can enhance the user experience by using the Tkinter library to create a more visually appealing interface. This will make the games more engaging and accessible to users, providing a better interaction platform while they play against these advanced algorithms.

6. References

[1] "Minimax Algorithm & Alpha-Beta Pruning." HackerEarth, https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/.

[2] Keith Galli, "How to Program Connect 4 in Python! (part 1) - Basic Structure & Game Loop," YouTube, uploaded by Keith Galli, https://youtu.be/UYgyRArKDEs?si=rhozVRVwqK4ITctJ.

[3] Jeremy Zhang, "Reinforcement Learning — Implement TicTacToe: Introduction of two Agent Game Playing," Towards Data Science, May 19, 2019, https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542.

[4] Abish Pius, "Writing in the World of Artificial Intelligence: Advanced AI Alpha-Beta Pruning in Python Through Tic-Tac-Toe," Medium, May 16, 2023, https://medium.com/chat-gpt-now-writes-all-my-articles/advanced-ai-alpha-beta-pruning-in-python-through-tic-tac-toe-70bb0b15db05

[5] Java Coding Community, "Tic Tac Toe AI with MiniMax using Python | Part 1: Programming Tic Tac Toe," YouTube, uploaded by Java Coding Community , https://youtu.be/JC1QsLOXp-I?si=YBo0MiVHjupttOgx.

[6] freeCodeCamp.org, "Develop an AI to play Connect Four - Python Tutorial," YouTube, uploaded by freeCodeCamp.org https://youtu.be/8392NJjj8s0?si=SLGuwWf3dniGxJCB.

[7] Glenn Lenormand, "Mastering Decision-Making in AI: An Introduction to Q-Learning," Medium, December 11, 2023, https://medium.com/@glennlenormand/mastering-decision-making-in-ai-an-introduction-to-q-learning-27afa8f0d856.

APPENDIX:

1. Tic Tac Toe

Human vs Computer with Minimax (with and without alpha beta pruning)

```
import random
import time

def printBoard(board):
    print("\nTic Tac Toe Board:")
    print(" " + board[1] + " | " + board[2] + " | " + board[3] + " ")
    print("---+---+---")
    print(" " + board[4] + " | " + board[5] + " | " + board[6] + " ")
    print("---+---+---")
    print(" " + board[7] + " | " + board[8] + " | " + board[9] + " ")
    print()

def spaceIsFree(position):
    return board[position] == ' '

def insertLetter(letter, position):
    if spaceIsFree(position):
        board[position] = letter
        printBoard(board)
        if checkForWin():
            print(f"{letter} wins!")
            return True
        if checkDraw():
            print("Draw!")
            return True
    else:
        print("This space is occupied!")
    return False

def checkForWin():
    return (board[1] == board[2] == board[3] != ' ' or
            board[4] == board[5] == board[6] != ' ' or
            board[7] == board[8] == board[9] != ' ' or
            board[1] == board[4] == board[7] != ' ' or
            board[2] == board[5] == board[8] != ' ' or
            board[3] == board[6] == board[9] != ' ' or
            board[1] == board[5] == board[9] != ' ' or
            board[7] == board[5] == board[3] != ' ')

def checkDraw():
    return ' ' not in board.values()

def playerMove():
    position = int(input("Enter the position for 'O' (1-9): "))
    return insertLetter(player, position)

def compMove(first_move):
    global total_steps, total_time
    if first_move:
        position = random.choice([k for k in range(1, 10) if spaceIsFree(k)])
        return insertLetter(bot, position), 0, 1
    else:
        bestScore = -float('inf')
        bestMove = None
        count_calls = 0
        start_time = time.time()

        for key in board.keys():
```

```
            if spaceIsFree(key):
                board[key] = bot
                score, calls = minimax(board, 0, False, -float('inf'), float('inf'), 0) if use_alpha_beta else minimax(board, 0, False, 0)
                board[key] = ' '
                count_calls += calls
                if score > bestScore:
                    bestScore = score
                    bestMove = key

        elapsed_time = time.time() - start_time
        move_made = insertLetter(bot, bestMove)
        total_time += elapsed_time
        total_steps += count_calls
        return move_made, elapsed_time, count_calls

def minimax(board, depth, isMaximizing, alpha=None, beta=None, call_count=0):
    call_count += 1
    if checkForWin():
        return (1 if isMaximizing else -1), call_count
    if checkDraw():
        return 0, call_count

    if isMaximizing:
        bestScore = -float('inf')
        for key in board.keys():
            if spaceIsFree(key):
                board[key] = bot
                if use_alpha_beta:
                    score, count = minimaxAlphaBeta(board, depth + 1, False, alpha, beta, call_count)
                else:
                    score, count = minimax(board, depth + 1, False, call_count=call_count)
                board[key] = ' '
                bestScore = max(bestScore, score)
                call_count = count
                if use_alpha_beta and beta is not None:
                    alpha = max(alpha, bestScore)
                    if beta <= alpha:
                        break
        return bestScore, call_count
    else:
        bestScore = float('inf')
        for key in board.keys():
            if spaceIsFree(key):
                board[key] = player
                if use_alpha_beta:
                    score, count = minimaxAlphaBeta(board, depth + 1, True, alpha, beta, call_count)
                else:
                    score, count = minimax(board, depth + 1, True, call_count=call_count)
                board[key] = ' '
                bestScore = min(bestScore, score)
                call_count = count
                if use_alpha_beta and alpha is not None:
                    beta = min(beta, bestScore)
                    if beta <= alpha:
                        break
        return bestScore, call_count

def minimaxAlphaBeta(board, depth, isMaximizing, alpha, beta, call_count):
    return minimax(board, depth, isMaximizing, alpha, beta, call_count)

def chooseAlgorithm():
    global use_alpha_beta
    choice = input("Choose algorithm: 1 for Minimax, 2 for Alpha-Beta Pruning: ")
    use_alpha_beta = (choice == '2')

board = {i: ' ' for i in range(1, 10)}
player = 'O'
bot = 'X'
use_alpha_beta = False
total_time = 0
```

```
    total_steps = 0
    game_over = False
    firstComputerMove = True

    chooseAlgorithm()
    printBoard(board)
    print("Computer goes first! Good luck.")
    print("Positions are as follow:")
    print("1, 2, 3 ")
    print("4, 5, 6 ")
    print("7, 8, 9 ")
    print("\n")

    while not game_over:
      game_over, time_taken, steps_taken = compMove(firstComputerMove)
      firstComputerMove = False
      if not game_over:
        game_over = playerMove()

    print(f"Total recursive steps taken: {total_steps}")
    print(f"Total time taken: {total_time:.4f} seconds")
```

Default vs Computer with minimax (with and without alpha beta pruning)

```
import random
import time
depth_limit = 4
def print_board(board):
  print("\nTic Tac Toe Board:")
  print(" " + board[1] + " | " + board[2] + " | " + board[3] + " ")
  print("---+---+---")
  print(" " + board[4] + " | " + board[5] + " | " + board[6] + " ")
  print("---+---+---")
  print(" " + board[7] + " | " + board[8] + " | " + board[9] + " ")
  print()

def space_is_free(board, position):
  return board[position] == ' '

def insert_letter(board, letter, position):
  board[position] = letter
  print(f"{letter} places on position {position}")
  print_board(board)
  if check_draw(board):
    print("Draw!")
    return True  # Indicate game end
  if check_for_win(board):
    print(f"{letter} wins!")
    return True  # Indicate game end
  return False  # Game continues

def check_for_win(board):
  return (board[1] == board[2] == board[3] != ' ' or
      board[4] == board[5] == board[6] != ' ' or
      board[7] == board[8] == board[9] != ' ' or
      board[1] == board[4] == board[7] != ' ' or
      board[2] == board[5] == board[8] != ' ' or
      board[3] == board[6] == board[9] != ' ' or
      board[1] == board[5] == board[9] != ' ' or
      board[7] == board[5] == board[3] != ' ')

def check_draw(board):
  return ' ' not in board.values()

def find_winning_move(board, mark):
  for key in range(1, 10):
    if space_is_free(board, key):
      board[key] = mark
      if check_for_win(board):
```

```
            board[key] = ' '
            return key
        board[key] = ' '
    return None

def select_strategic_move(board):
    for key in [5, 1, 3, 7, 9, 2, 4, 6, 8]:  # Center, corners, then sides
        if space_is_free(board, key):
            return key

def opponent_move(board, player, opponent):
    # First, check if the opponent can win in the next move
    winning_move = find_winning_move(board, opponent)
    if winning_move is not None:
        insert_letter(board, opponent, winning_move)
        print(f"Opponent ({opponent}) plays winning move at position {winning_move}")
        return

    # Next, check if it needs to block the player from winning
    blocking_move = find_winning_move(board, player)
    if blocking_move is not None:
        insert_letter(board, opponent, blocking_move)
        print(f"Opponent ({opponent}) blocks player ({player}) at position {blocking_move}")
        return

    # If no immediate win or block is possible, use Minimax to determine the best move
    best_move = None
    best_score = float('inf')
    for key in range(1, 10):
        if space_is_free(board, key):
            board[key] = opponent
            score, _ = minimax(board, 0, True, opponent, player, -float('inf'), float('inf'), [0])
            board[key] = ' '
            if score < best_score:
                best_score = score
                best_move = key

    # Make the best move determined by Minimax if no direct win or block was available
    if best_move is not None:
        insert_letter(board, opponent, best_move)
        print(f"Opponent ({opponent}) makes a strategic move at position {best_move}")

def minimax(board, depth, is_maximizing, player, opponent, alpha, beta, count_calls):
    count_calls[0] += 1
    if depth == depth_limit or check_for_win(board) or check_draw(board):
        return evaluate_board(board, player, opponent, depth), count_calls[0]

    if check_for_win(board):
        return (1 if is_maximizing else -1), count_calls[0]
    if check_draw(board):
        return 0, count_calls[0]

    if is_maximizing:
        best_score = -float('inf')
        for key in range(1, 10):
            if space_is_free(board, key):
                board[key] = player
                score, calls = minimax(board, depth + 1, False, player, opponent, alpha, beta, count_calls)
                board[key] = ' '
                best_score = max(best_score, score)
                if alpha is not None and beta is not None:
                    alpha = max(alpha, score)
                    if beta <= alpha:
                        break
        return best_score, count_calls[0]
    else:
        best_score = float('inf')
        for key in range(1, 10):
            if space_is_free(board, key):
                board[key] = opponent
```

```
            score, calls = minimax(board, depth + 1, True, player, opponent, alpha, beta, count_calls)
            board[key] = ' '
            best_score = min(best_score, score)
            if alpha is not None and beta is not None:
                beta = min(beta, score)
                if beta <= alpha:
                    break
        return best_score, count_calls[0]
def evaluate_board(board, player, opponent, depth):
    if check_for_win(board):
        if board[next(iter(board))] == player:
            return 10 - depth
        else:
            return depth - 10
    return 0
def comp_move(board, player, opponent, use_alpha_beta, move_count):
    start_time = time.time()
    count_calls = [0]

    if move_count == 1:
        free_positions = [pos for pos in range(1, 10) if space_is_free(board, pos)]
        first_move = random.choice(free_positions)
        insert_letter(board, player, first_move)
    else:
        best_score = -float('inf')
        best_move = None
        for key in range(1, 10):
            if space_is_free(board, key):
                board[key] = player
                if use_alpha_beta:
                    score, calls = minimax(board, 0, False, player, opponent, -float('inf'), float('inf'), count_calls)
                else:
                    score, calls = minimax(board, 0, False, player, opponent, None, None, count_calls)
                board[key] = ' '
                if score > best_score:
                    best_score = score
                    best_move = key
                count_calls[0] += calls  # Accumulate total calls

        if best_move is not None:
            insert_letter(board, player, best_move)

    elapsed_time = time.time() - start_time
    return count_calls[0], elapsed_time



def main():
    board = {key: ' ' for key in range(1, 10)}
    player = 'X'  # Computer's marker
    opponent = 'O'  # Default AI opponent's marker
    use_alpha_beta = int(input("Choose algorithm for Computer 'X' (1 for Minimax, 2 for Alpha-Beta Pruning): ")) == 2
    total_steps = 0
    total_time = 0
    move_count = 1

    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while True:
        if move_count % 2 != 0:
            print("Computer's move (X):")
            steps, time_taken = comp_move(board, player, opponent, use_alpha_beta, move_count)
            total_steps += steps
            total_time += time_taken
            if check_for_win(board) or check_draw(board):
                break  # Break the loop if the game is over
        else:
            print("Default Opponent's move (O):")
            steps, time_taken = comp_move(board, opponent, player, False, move_count)
```

```
            total_steps += steps
            total_time += time_taken
            if check_for_win(board) or check_draw(board):
                break  # Break the loop if the game is over
        move_count += 1

    print(f"Total recursive steps taken: {total_steps}")
    print(f"Total time taken: {total_time:.4f} seconds")

if __name__ == "__main__":
    main()
```

Default vs Computer with Q-learning Algorithms

```
import random
import pickle
import time

def print_board(board):
    print("\nTic Tac Toe Board:")
    print(" " + board[1] + " | " + board[2] + " | " + board[3] + " ")
    print("---+---+---")
    print(" " + board[4] + " | " + board[5] + " | " + board[6] + " ")
    print("---+---+---")
    print(" " + board[7] + " | " + board[8] + " | " + board[9] + " ")
    print()


def space_is_free(board, position):
    return board[position] == ' '

def check_for_win(board):
    return (board[1] == board[2] == board[3] != ' ' or
            board[4] == board[5] == board[6] != ' ' or
            board[7] == board[8] == board[9] != ' ' or
            board[1] == board[4] == board[7] != ' ' or
            board[2] == board[5] == board[8] != ' ' or
            board[3] == board[6] == board[9] != ' ' or
            board[1] == board[5] == board[9] != ' ' or
            board[7] == board[5] == board[3] != ' ')

def check_draw(board):
    return ' ' not in board.values()

def get_state(board):
    return "".join([board[i] if board[i] != ' ' else '0' for i in range(1, 10)])


def load_q_table(filename='q_table.pkl'):
    try:
        with open(filename, 'rb') as f:
            return pickle.load(f)
    except FileNotFoundError:
        return {}


def find_winning_move(board, player):
    for i in range(1, 10):
        if space_is_free(board, i):
            board[i] = player
            if check_for_win(board):
                board[i] = ' '
                return i
            board[i] = ' '
    return None


def minimax(board, depth, is_maximizing, player, opponent):
    if check_for_win(board):
        return -1 if is_maximizing else 1
```

```
    if check_draw(board):
        return 0

    if is_maximizing:
        best_score = -float('inf')
        for i in range(1, 10):
            if space_is_free(board, i):
                board[i] = opponent
                score = minimax(board, depth + 1, False, player, opponent)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(1, 10):
            if space_is_free(board, i):
                board[i] = player
                score = minimax(board, depth + 1, True, player, opponent)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score


def best_move(board, player, opponent):
    best_val = -float('inf')
    move = None
    for i in range(1, 10):
        if space_is_free(board, i):
            board[i] = opponent
            move_val = minimax(board, 0, False, player, opponent)
            board[i] = ' '
            if move_val > best_val:
                best_val = move_val
                move = i
    return move


def save_q_table(q_table, filename='q_table.pkl'):
    with open(filename, 'wb') as f:
        pickle.dump(q_table, f)


def choose_action(board, q_table, epsilon=0.1):
    state = get_state(board)
    if random.random() < epsilon:  # Explore
        return random.choice([k for k in range(1, 10) if space_is_free(board, k)])
    else:  # Exploit
        all_q_values = q_table.get(state, {})
        if not all_q_values:
            return random.choice([k for k in range(1, 10) if space_is_free(board, k)])
        return max(all_q_values, key=all_q_values.get)


def update_q_table(q_table, state, action, reward, next_state, alpha=0.1, gamma=0.9):
    old_value = q_table.get(state, {}).get(action, 0)
    next_max = max(q_table.get(next_state, {}).values(), default=0)
    new_value = old_value + alpha * (reward + gamma * next_max - old_value)
    if state not in q_table:
        q_table[state] = {}
    q_table[state][action] = new_value


def player_move(board, q_table, player='X'):
    move = choose_action(board, q_table)
    board[move] = player
    print(f"Player {player} placed on position {move}")
    print_board(board)
    return move
```

```
def opponent_move(board, opponent='O', player='X'):
    win_move = find_winning_move(board, opponent)
    if win_move:
        board[win_move] = opponent
        print(f"Player {opponent} placed on winning position {win_move}")
        print_board(board)
        return win_move

    block_move = find_winning_move(board, player)
    if block_move:
        board[block_move] = opponent
        print(f"Player {opponent} blocks on position {block_move}")
        print_board(board)
        return block_move

    move = best_move(board, player, opponent)
    board[move] = opponent
    print(f"Player {opponent} plays on position {move} using minimax")
    print_board(board)
    return move


def main():
    board = {i: ' ' for i in range(1, 10)}
    q_table = load_q_table()
    player, opponent = 'X', 'O'
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    total_steps = 0
    start_time = time.time()
    is_first_move_player = True
    is_first_move_opponent = True

    while True:
        if is_first_move_player:
            player_move_position = random.choice([i for i in range(1, 10) if space_is_free(board, i)])
            board[player_move_position] = player
            print(f"Player {player} makes a random first move on position {player_move_position}")
            is_first_move_player = False
        else:
            player_move_position = player_move(board, q_table, player)

        print_board(board)
        total_steps += 1
        if check_for_win(board):
            print(f"{player} wins!")
            break
        elif check_draw(board):
            print("It's a draw!")
            break

        if is_first_move_opponent:
            opponent_move_position = random.choice([i for i in range(1, 10) if space_is_free(board, i)])
            board[opponent_move_position] = opponent
            print(f"Opponent {opponent} makes a random first move on position {opponent_move_position}")
            is_first_move_opponent = False
        else:
            opponent_move_position = opponent_move(board, opponent, player)

        print_board(board)
        total_steps += 1
        if check_for_win(board):
            print(f"{opponent} wins!")
            break
        elif check_draw(board):
            print("It's a draw!")
            break

    total_time = time.time() - start_time
```

```
    print(f"Total time taken: {total_time:.4f} seconds")
    print(f"Total steps taken: {total_steps} moves")

    save_q_table(q_table)


if __name__ == "__main__":
    main()
```

2. Connect4 Game

Human vs Computer with Minimax (with and without alpha beta pruning)

```
import time
import random
class Connect4:
    def __init__(self, rows, columns, ai_with_pruning, ai_starts):
        self.rows = rows
        self.columns = columns
        self.board = [[' ' for _ in range(columns)] for _ in range(rows)]
        self.ai_with_pruning = ai_with_pruning
        self.turn = 'O' if ai_starts else 'X'
        self.total_steps = 0
        self.total_time = 0
        self.first_move = True

    def display_board(self):
        print("\n")
        for row in self.board:
            print('| ' + ' | '.join(row) + ' |')
        print('+---' * self.columns + '+')

    def insert_token(self, column):
        if column < 0 or column >= self.columns or self.board[0][column] != ' ':
            return False

        for i in range(self.rows - 1, -1, -1):
            if self.board[i][column] == ' ':
                self.board[i][column] = self.turn
                return True
        return False

    def check_winner(self):
        directions = [(0, 1), (1, 0), (1, 1), (1, -1)]
        for r in range(self.rows):
            for c in range(self.columns):
                if self.board[r][c] == ' ':
                    continue
                for dr, dc in directions:
                    line = []
                    for i in range(4):
                        rr, cc = r + dr * i, c + dc * i
                        if 0 <= rr < self.rows and 0 <= cc < self.columns:
                            line.append(self.board[rr][cc])
                        else:
                            break
                    if len(line) == 4 and all(x == self.turn for x in line):
                        return self.turn
        return None

    def switch_turn(self):
        self.turn = 'O' if self.turn == 'X' else 'X'

    def evaluate_board(self):
        winner = self.check_winner()
        if winner == 'O':
            return 1  # Computer wins
        elif winner == 'X':
            return -1  # Human wins
```

```
        return 0  # No winner yet

    def is_full(self):
        return all(self.board[0][c] != ' ' for c in range(self.columns))

    def minimax(self, depth, maximizingPlayer, alpha=None, beta=None):
        self.total_steps += 1
        if depth == 0 or self.check_winner() or self.is_full():
            return self.evaluate_board()

        if maximizingPlayer:
            maxEval = float('-inf')
            for c in range(self.columns):
                if self.board[0][c] == ' ':
                    self.insert_token(c)
                    self.switch_turn()
                    start_time = time.time()
                    eval = self.minimax(depth - 1, False, alpha, beta)
                    elapsed = time.time() - start_time
                    self.total_time += elapsed
                    self.remove_token(c)
                    self.switch_turn()
                    maxEval = max(maxEval, eval)
                    if self.ai_with_pruning:
                        if alpha is not None and beta is not None:
                            alpha = max(alpha, eval)
                            if beta <= alpha:
                                break
            return maxEval
        else:
            minEval = float('inf')
            for c in range(self.columns):
                if self.board[0][c] == ' ':
                    self.insert_token(c)
                    self.switch_turn()
                    start_time = time.time()
                    eval = self.minimax(depth - 1, True, alpha, beta)
                    elapsed = time.time() - start_time
                    self.total_time += elapsed
                    self.remove_token(c)
                    self.switch_turn()
                    minEval = min(minEval, eval)
                    if self.ai_with_pruning:
                        if alpha is not None and beta is not None:
                            beta = min(beta, eval)
                            if beta <= alpha:
                                break
            return minEval

    def remove_token(self, column):
        for i in range(self.rows):
            if self.board[i][column] != ' ':
                self.board[i][column] = ' '
                break

    def best_move(self, depth=4):
        best_score = float('-inf')
        best_column = None
        for c in range(self.columns):
            if self.board[0][c] == ' ':
                self.insert_token(c)
                self.switch_turn()
                start_time = time.time()
                score = self.minimax(depth - 1, False, float('-inf'), float('inf'))
                elapsed = time.time() - start_time
                self.total_time += elapsed
                self.remove_token(c)
                self.switch_turn()
                if score > best_score:
                    best_score = score
```

```
                best_column = c
        return best_column

    def play_game(self):
        while True:
            self.display_board()
            if self.turn == 'X':
                col = int(input(f"Player {self.turn}, enter column (0-{self.columns - 1}) to place your token: "))
            else:
                print("Computer is making its move...")
                if self.first_move:
                    col = random.choice([i for i in range(self.columns) if self.board[0][i] == ' '])
                    self.first_move = False
                else:
                    col = self.best_move()



            if self.insert_token(col):
                if self.check_winner():
                    self.display_board()
                    print(f"Player {self.turn} wins!")
                    break
                elif self.is_full():
                    self.display_board()
                    print("It's a draw!")
                    break
                self.switch_turn()
            else:
                print("Invalid move, try again.")
        print(f"Total steps taken: {self.total_steps}")
        print(f"Total time taken: {self.total_time:.2f} seconds")


# Game setup and initialization from user input
print("Welcome to Connect 4!")
rows = int(input("Please enter board size:\nRows: "))
cols = int(input("Columns: "))

print("Choose algorithm: \n1 for Minimax\n2 for Alpha-Beta Pruning")
algorithm_choice = input("Your choice: ")
pruning = algorithm_choice == '2'

print("Who should start the game? \n1 for Human\n2 for Computer")
start_choice = input("Your choice: ")
ai_starts = start_choice == '2'

game = Connect4(rows, cols, ai_with_pruning=pruning, ai_starts=ai_starts)
game.play_game()
```

Default vs Computer with minimax (with and without alpha beta pruning)

```
import random
import time

class Connect4:
    def __init__(self, rows, columns, ai_with_pruning, ai_starts):
        self.rows = rows
        self.columns = columns
        self.board = [[' ' for _ in range(columns)] for _ in range(rows)]
        self.ai_with_pruning = ai_with_pruning
        self.turn = 'O' if ai_starts else 'X'
        self.total_steps = 0
        self.total_time = 0
        self.first_move = True
        self.computer_wins = 0

    def display_board(self):
        print("\n")
        for row in self.board:
```

```
        print('| ' + ' | '.join(row) + ' |')
    print('+---' * self.columns + '+')

def insert_token(self, column):
    if column < 0 or column >= self.columns or self.board[0][column] != ' ':
        return False, -1
    for i in range(self.rows - 1, -1, -1):
        if self.board[i][column] == ' ':
            self.board[i][column] = self.turn
            return True, i
    return False, -1

def remove_token(self, column):
    for i in range(self.rows):
        if self.board[i][column] != ' ':
            self.board[i][column] = ' '
            break

def check_winner(self, check_turn=None):
    check_turn = check_turn or self.turn
    directions = [(0, 1), (1, 0), (1, 1), (1, -1)]
    for r in range(self.rows):
        for c in range(self.columns):
            if self.board[r][c] != check_turn:
                continue
            for dr, dc in directions:
                line = []
                for i in range(4):
                    rr, cc = r + dr * i, c + dc * i
                    if 0 <= rr < self.rows and 0 <= cc < self.columns and self.board[rr][cc] == check_turn:
                        line.append(self.board[rr][cc])
                    else:
                        break
                if len(line) == 4:
                    return check_turn
    return None

def switch_turn(self):
    self.turn = 'O' if self.turn == 'X' else 'X'

def evaluate_board(self):
    winner = self.check_winner()
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1
    return 0

def is_full(self):
    return all(self.board[0][c] != ' ' for c in range(self.columns))

def minimax(self, depth, maximizingPlayer, alpha=None, beta=None):
    self.total_steps += 1
    if depth == 0 or self.check_winner() or self.is_full():
        return self.evaluate_board()

    if maximizingPlayer:
        maxEval = float('-inf')
        for c in range(self.columns):
            if self.board[0][c] == ' ':
                _, row = self.insert_token(c)
                self.switch_turn()
                start_time = time.time()
                eval = self.minimax(depth - 1, False, alpha, beta)
                elapsed = time.time() - start_time
                self.total_time += elapsed
                self.board[row][c] = ' '
                self.switch_turn()
                maxEval = max(maxEval, eval)
                if self.ai_with_pruning:
```

```
                if alpha is not None and beta is not None:
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return maxEval
    else:
        minEval = float('inf')
        for c in range(self.columns):
            if self.board[0][c] == ' ':
                _, row = self.insert_token(c)
                self.switch_turn()
                start_time = time.time()
                eval = self.minimax(depth - 1, True, alpha, beta)
                elapsed = time.time() - start_time
                self.total_time += elapsed
                self.board[row][c] = ' '
                self.switch_turn()
                minEval = min(minEval, eval)
                if self.ai_with_pruning:
                    if alpha is not None and beta is not None:
                        beta = min(beta, eval)
                        if beta <= alpha:
                            break
        return minEval

def best_move(self, depth=4):
    best_score = float('-inf')
    best_column = None
    for c in range(self.columns):
        if self.board[0][c] == ' ':
            _, row = self.insert_token(c)
            self.switch_turn()
            start_time = time.time()
            score = self.minimax(depth - 1, False, float('-inf'), float('inf'))
            elapsed = time.time() - start_time
            self.total_time += elapsed
            self.board[row][c] = ' '
            self.switch_turn()
            if score > best_score:
                best_score = score
                best_column = c
    return best_column

def opponent_move(self):
    col = self.best_move(depth=2)
    return col

def play_game(self):
    while True:
        self.display_board()
        if self.turn == 'X':
            print("Player X (Default opponent) is making its move...")
            col = self.opponent_move()
        else:
            print("Computer is making its move...")
            if self.first_move:
                col = random.choice([i for i in range(self.columns) if self.board[0][i] == ' '])
                self.first_move = False
            else:
                col = self.best_move()

        if col is None or not self.insert_token(col)[0]:
            print("Invalid move, try again.")
            continue

        winner = self.check_winner()
        if winner:
            self.display_board()
            print(f"Player {winner} wins!")
            if winner == 'O':
```

```
                self.computer_wins += 1
                break
            elif self.is_full():
                self.display_board()
                print("It's a draw!")
                break
            self.switch_turn()
        print(f"Total steps taken: {self.total_steps}")
        print(f"Total time taken: {self.total_time:.2f} seconds")

# Game setup and initialization from user input
print("Welcome to Connect 4!")
rows = int(input("Please enter board size:\nRows: "))
cols = int(input("Columns: "))

print("Choose algorithm: \n1 for Minimax\n2 for Alpha-Beta Pruning")
algorithm_choice = input("Your choice: ")
pruning = algorithm_choice == '2'

print("Who should start the game? \n1 for Default opponent\n2 for Computer")
start_choice = input("Your choice: ")
ai_starts = start_choice == '2'

game = Connect4(rows, cols, ai_with_pruning=pruning, ai_starts=ai_starts)
game.play_game()
```

Default vs Computer with Q-learning Algorithms

```
import random
import numpy as np
import pickle
import time
class QAgent:
    def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.1):
        self.q_table = {}
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon

    def get_q_value(self, state, action):
        return self.q_table.get((state, action), 0)

    def set_q_value(self, state, action, value):
        self.q_table[(state, action)] = value

    def choose_action(self, state, possible_actions):
        if random.random() < self.epsilon:
            return random.choice(possible_actions)
        else:
            q_values = [self.get_q_value(state, a) for a in possible_actions]
            max_q_value = max(q_values)
            actions_with_max_q_value = [a for a, q in zip(possible_actions, q_values) if q == max_q_value]
            return random.choice(actions_with_max_q_value)

    def learn(self, state, action, reward, next_state, possible_actions):
        current_q = self.get_q_value(state, action)
        max_future_q = max([self.get_q_value(next_state, a) for a in possible_actions], default=0)
        new_q = current_q + self.alpha * (reward + self.gamma * max_future_q - current_q)
        self.set_q_value(state, action, new_q)

    def save_q_table(self, file_path='q_table.pkl'):
        with open(file_path, 'wb') as file:
            pickle.dump(self.q_table, file)

    def load_q_table(self, file_path='q_table.pkl'):
        with open(file_path, 'rb') as file:
            self.q_table = pickle.load(file)

class Connect4:
```

```
def __init__(self, rows, columns, start_player='X'):
    self.rows = rows
    self.columns = columns
    self.start_player = start_player
    self.reset()
    self.step_count = 0
    self.time_spent = 0

def reset(self):
    self.board = [[' ' for _ in range(self.columns)] for _ in range(self.rows)]
    self.turn = self.start_player

def display_board(self):
    print("\n")
    for row in self.board:
        print('| ' + ' | '.join(row) + ' |')
    print('+---' * self.columns + '+')

def get_state(self):
    return ''.join(sum(self.board, []))

def insert_token(self, column):
    if column < 0 or column >= self.columns or self.board[0][column] != ' ':
        return False, self.get_state()
    for i in range(self.rows - 1, -1, -1):
        if self.board[i][column] == ' ':
            self.board[i][column] = self.turn
            return True, self.get_state()
    return False, self.get_state()

def remove_token(self, column):
    # Remove the topmost token from the column
    for i in range(self.rows):
        if self.board[i][column] != ' ':
            self.board[i][column] = ' '
            break

def check_winner(self):
    directions = [(0, 1), (1, 0), (1, 1), (1, -1)]
    for r in range(self.rows):
        for c in range(self.columns):
            if self.board[r][c] == ' ':
                continue
            for dr, dc in directions:
                line = []
                for i in range(4):
                    rr, cc = r + dr * i, c + dc * i
                    if 0 <= rr < self.rows and 0 <= cc < self.columns and self.board[rr][cc] == self.turn:
                        line.append(self.board[rr][cc])
                    else:
                        break
                if len(line) == 4:
                    return self.turn
    return None

def switch_turn(self):
    self.turn = 'O' if self.turn == 'X' else 'X'

def play_game(self, q_agent, start_player):
    self.start_player = start_player
    self.reset()
    game_over = False
    state = self.get_state()
    while not game_over:
        self.display_board()
        possible_actions = [i for i in range(self.columns) if self.board[0][i] == ' ']
        if not possible_actions:
            print("Draw!")
            break
        start_time = time.perf_counter()
```

```
            if self.turn == 'X':  # Q-learning agent
                action = q_agent.choose_action(state, possible_actions)
                print(f"Q-agent (X) chooses column {action}")
                _, new_state = self.insert_token(action)
                reward = 0
                if self.check_winner() == 'X':
                    reward = 1
                    game_over = True
                    print("Q-agent (X) wins!")
                elif not possible_actions:
                    reward = 0.5
                    game_over = True
                    print("Draw!")
                q_agent.learn(state, action, reward, new_state, possible_actions)
                state = new_state
            else:  # Default opponent random play
                action = self.intelligent_opponent_move()
                print(f"Default opponent (O) chooses column {action}")
                _, state = self.insert_token(action)
                if self.check_winner() == 'O':
                    reward = -1
                    game_over = True
                    print("Default Opponent (O) wins!")
            elapsed_time = time.perf_counter() - start_time
            self.time_spent += elapsed_time
            self.step_count += 1

            self.switch_turn()
        print(f"Total moves: {self.step_count}, Total time: {self.time_spent:.2f} seconds")

    def intelligent_opponent_move(self):
        # Prioritize winning moves and blocking moves
        for action in range(self.columns):
            if self.board[0][action] != ' ':  # Skip full columns
                continue
            _, _ = self.insert_token(action)
            if self.check_winner() == self.turn:  # Winning move
                self.remove_token(action)
                return action
            self.remove_token(action)

            self.switch_turn()
            _, _ = self.insert_token(action)
            if self.check_winner() == self.turn:  # Blocking move
                self.remove_token(action)
                self.switch_turn()
                return action
            self.remove_token(action)
            self.switch_turn()

        return random.choice([c for c in range(self.columns) if self.board[0][c] == ' '])

if __name__ == "__main__":
    print("Welcome to Connect 4 Q-Learning!")
    rows = int(input("Enter the number of rows for the board: "))
    columns = int(input("Enter the number of columns for the board: "))
    q_agent = QAgent()
    try:
        q_agent.load_q_table()
        print("Loaded existing Q-table.")
    except (FileNotFoundError, EOFError):
        print("No existing Q-table found, starting fresh.")

    while True:
        print("Who should start the game? \n1 for Default opponent\n2 for Q-agent")
        start_choice = input("Your choice: ")
        start_player = 'O' if start_choice == '1' else 'X'
        game = Connect4(rows, columns, start_player=start_player)
        game.play_game(q_agent, start_player)
```

```
q_agent.save_q_table()  # Save progress after each game
print("Q-table saved.")

continue_playing = input("Play another game? (yes/no): ")
if continue_playing.lower() != 'yes':
    break
```