TRINITY COLLEGE DUBLIN

School of Computer Science and Statistics

Abhishek Zade

StudentId: 24332461

Assignment 3 **CS7IS2 Artificial Intelligence**

1 Introduction

The aim of this project is to implement and compare artificial intelligence (AI) algorithms for Connect4 and Tic-Tac-Toe games. The primary objective is to test and compare different strategies, including the baseline approach, the Minimax algorithm (both with and without Alpha-Beta pruning), and Q-Learning. The focus is not only on developing functional AI agents capable of playing these games but also to analyze the performance of each algorithm through some fundamental metrics, such as win rate, move efficiency, and computation time. Through the evaluation of these algorithms, the project hopes to see how good they are and what trade-offs, and have a general idea of their strengths and weaknesses in solving these two games of yesteryear.

The **baseline strategy** is a naive heuristic method where the AI looks for a winning opportunity or prevents the opposition's winning move, whichever is possible right away. If the other cannot be performed, it plays an arbitrary move. The **Minimax algorithm** is a classic game theory approach that searches the game tree to a specified depth and assigns each potential move with a score. **Alpha-Beta pruning** improves upon the Minimax algorithm by removing portions of the search tree that need not be searched, making computation more efficient. Finally, **Q-Learning** is a reinforcement learning technique wherein the AI learns optimal policies through iterated interaction with the game. It uses a Q-table to store state-action pairs and updates these values based on rewards after each action so that the AI learns incrementally.

This work explains the methodology to use these algorithms, describes the code structure, and compares the performance of each approach in Connect4 and Tic-Tac-Toe.

2 Implementation

This project details the approach taken to implement and evaluate artificial intelligence (AI) algorithms for the games Connect4 and Tic-Tac-Toe. This can be broken down into several significant steps: the representation of game states, move generation, implementation of AI algorithms, and performance monitoring.

2.1 Game State Representation

Both the states of games are maintained in an well-formed structure. The board of **Connect4** is a 6x7 grid where each cell is able to store 'X', 'O', or an empty cell (' '). It is modeled using a 2D array so that direct access to the rows and columns is possible. The board for **Tic-Tac-Toe** is a 3x3 grid and it is stored as a list of 9 elements. Each piece starts as an empty space and gets filled with 'X' or 'O' as the game goes on.

The **latest winner** is stored after each move. In Connect4, a win occurs when there are four consecutive pieces in a line, column, or diagonal. In Tic-Tac-Toe, it's three consecutive pieces. The game checks these conditions after each move to see if the game has ended in a win or a draw.

2.2 Move Generation

The second step is the generation of legal moves. In **Connect4**, a legal move is any column that has at least one blank space. The system selects the topmost unoccupied space in each column upon which a piece can be placed. In **Tic-Tac-Toe**, a legal move is any cell within the 3x3 grid that is free from 'X' or 'O'.

After the possible moves have been identified, the AI selects one and makes the move on the board. The system then checks if the move results in a winning combination. If someone wins, the game is over and the winner is declared. If there are no available moves remaining and no winner is found, the game is a draw.

2.3 AI Algorithms

The project implements and evaluates two different AI strategies for playing the games, both of which are compared against the **Baseline** strategy: **Minimax** and **Q-Learning**.

2.3.1 Baseline Strategy

The **baseline strategy** is a simple heuristic-based approach that follows these steps:

1. **Check for a Winning Move**: The AI evaluates each available move to determine if it results in a win. If a winning move is found, it is executed immediately.
2. **Block the Opponent**: If no winning move is available, the AI simulates the opponent's moves. If the opponent can win on the next move, the AI blocks that move.
3. **Random Move**: If neither a winning move nor a block is required, the AI selects a random available move.

The pseudocode for the baseline strategy is as follows:

```
function baseline_move(game, letter):
    for each move in available_moves(game):
        make_move(game, move, letter)
        if current_winner(game) == letter:
            undo_move(game, move)
            return move
        undo_move(game, move)

    opponent = switch_player(letter)
    for each move in available_moves(game):
        make_move(game, move, opponent)
        if current_winner(game) == opponent:
            undo_move(game, move)
            return move
        undo_move(game, move)

    return random choice from available_moves(game)
```

Listing 1: Baseline Strategy Pseudocode

In the pseudocode, the function starts by testing whether there is a winning move by trying all the possible moves. If there isn't a winning move, then it tests if the opponent can win in the immediate next move and blocks that. If neither there is a need for a winning move nor a block, the function picks randomly one of the possible moves.

2.3.2 Minimax Algorithm

The **Minimax algorithm** is a choice algorithm that looks at all potential future game states to select the optimal move. It uses a tree structure, with each node being a game state and the edges being potential moves. At each level, the algorithm alternates between maximizing the player's score (for the maximizing player) and minimizing the opponent's score (for the minimizing player).

Alpha-Beta Pruning is an optimization technique used to eliminate branches that do not need exploration, thus improving the efficiency of the Minimax search.

The pseudocode for the Minimax algorithm with Alpha-Beta pruning is as follows:

```
function minimax(game, player, depth, alpha, beta):
    if depth == 0 or game over:
        return evaluate_board(game, player)

    if player == MAX_PLAYER:
        best_score = -infinity
        for each move in available_moves(game):
            make_move(game, move, player)
            score = minimax(game, switch_player(player), depth-1, alpha, beta)
            undo_move(game, move)
            best_score = max(best_score, score)
            alpha = max(alpha, best_score)
            if beta <= alpha:
                break
        return best_score
    else:
        best_score = infinity
        for each move in available_moves(game):
            make_move(game, move, player)
            score = minimax(game, switch_player(player), depth-1, alpha, beta)
            undo_move(game, move)
            best_score = min(best_score, score)
            beta = min(beta, best_score)
            if beta <= alpha:
                break
        return best_score
```

Listing 2: Minimax Algorithm with Alpha-Beta Pruning Pseudocode

The Minimax pseudocode starts by verifying if the maximum depth of search is achieved or if the game is over. If it is, it computes the value of the current board position. If it is the maximizing player's turn, the algorithm tries to compute the maximum value by looking at all the possible moves and selecting the one with the highest value. If it's the minimizing player's turn, the algorithm subtracts the score by choosing the move with the lowest score. Alpha-Beta pruning is employed at search in order to avoid searching branches that will not have an impact on the final decision, thus reducing the number of states being explored.

2.3.3 Q-Learning Algorithm

Q-Learning is a reinforcement learning algorithm where the agent learns an optimal strategy by interacting with the environment (i.e., playing games). The algorithm maintains a Q-table, which stores state-action pairs and the corresponding expected rewards. The agent updates the Q-values based on the rewards it receives after each action, gradually improving its strategy.

The Q-learning algorithm uses the following update rule:

$$Q(s,a) = Q(s,a) + \alpha \left(R(s,a) + \gamma \max_{a} Q(s',a) - Q(s,a) \right)$$

where:

- s is the current state,
- a is the action taken,
- $R(s,a)$ is the reward for taking action a in state s,
- s' is the resulting state after the action,
- α is the learning rate,
- γ is the discount factor.

The Q-learning algorithm follows these steps:

```
function q_learning_move(game, player):
    state = state_representation(game, player)
    if state not in Q_table:
        initialize_Q_table(state)

    if random_value < epsilon:
        return random choice from available_moves(game)
    else:
        return max_move based on Q_table[state]

function update_Q_table(state, action, reward, next_state):
    future_q = max(Q_table[next_state])
    Q_table[state][action] = Q_table[state][action] + alpha * (reward + gamma * future_q
     - Q_table[state][action])
```

Listing 3: Q-Learning Algorithm Pseudocode

In the pseudocode, the function first checks the current state of the game. If the state is not yet in the Q-table, it initializes the Q-values for that state-action pair. The agent then explores randomly with a probability of ϵ (exploration) or selects the move with the highest Q-value (exploitation). After the move is made, the Q-table is updated using the Q-learning update rule (Equation 1) based on the observed reward and the maximum future Q-value for the next state.

Q-learning allows the agent to adapt its strategy over time, improving its performance based on past experiences.

3 Evaluation

Parameter	Value
Alpha-Beta	Pruning technique used in Minimax algorithm
Total Games	10 to 1000
MINMAX-ALPHA	-inf (negative infinity)
MINMAX-BETA	+inf (positive infinity)
QL-ALPHA	0.3
QL-GAMMA	0.9
QL-EPSILON	0.7
Depth (Minimax)	5

Table 1: Key Parameters and Their Values

3.0.1 Explanation of Parameters

The parameters chosen for the experiments are critical for evaluating and comparing the different AI algorithms.

The **Alpha-Beta** pruning technique is used in the Minimax algorithm to reduce the number of states explored, improving computational efficiency. By eliminating branches of the search tree that do not affect the final decision, the algorithm can focus on more promising paths, thus reducing processing time.

The **Total Games** parameter indicates the number of games played in each experiment. This ranged from 10 to 1000 games, allowing the evaluation of algorithms at different scales. The experiments with more games test the scalability and consistency of the algorithms under larger datasets.

The **MINMAX-ALPHA** and **MINMAX-BETA** parameters are the initial bounds for the Minimax algorithm's search. The Alpha parameter is initialized to negative infinity, and Beta to positive infinity, allowing the algorithm to explore the entire search tree initially. As the search progresses, the algorithm refines these values using Alpha-Beta pruning to cut off unnecessary branches.

The **QL-ALPHA** is the learning rate in Q-learning, set to 0.3. This controls how much new experiences affect the agent's Q-values. A learning rate of 0.3 provides a balanced update, preventing the Q-values from changing too drastically with each experience.

The **QL-GAMMA** is the discount factor in Q-learning, set to 0.9. It determines the importance of future rewards. A value of 0.9 means future rewards are still important, but immediate rewards are prioritized.

The **QL-EPSILON** is the exploration rate in Q-learning, set to 0.7. This governs the probability of choosing a random action instead of the one with the highest Q-value. A higher epsilon encourages more exploration, helping the agent learn better strategies by trying out more actions.

The **Depth (Minimax)** parameter refers to the maximum depth of the Minimax search tree, set to 5. This restricts the number of levels the algorithm will explore, striking a balance between computation time and decision-making quality.

3.0.2 State Exploration with Minimax Algorithm (Depth 6)

Before proceeding with the analysis, I first measured the number of states explored by the Minimax algorithm for 30 minutes with a depth of 6. The results were as follows:

States explored in Minimax with Alpha-Beta pruning: 266,382,448

States explored in Minimax without Alpha-Beta pruning: 180,474,193

These results highlight the significant impact of Alpha-Beta pruning on reducing the number of states that need to be evaluated. The pruning technique effectively eliminated branches of the search tree that would not influence the final decision, allowing the Minimax algorithm to perform faster and handle larger state spaces more efficiently compared to the version without pruning.

3.1 Tic-Tac-Toe Evaluation

In the case of **Tic-Tac-Toe**, the results highlight significant differences between the AI strategies in terms of execution time, move efficiency, and overall performance.

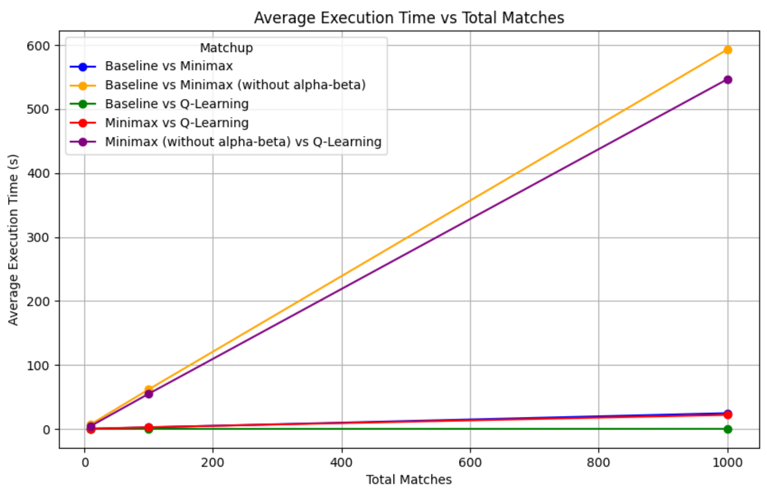

Figure 1: Average Execution Time vs Total Matches

Figure 1 presents the total execution time in seconds across different matchups. As expected, **Minimax** strategies with Alpha-Beta pruning exhibit higher execution times as the number of games increases compared to the **Baseline** and **Q-learning** algorithms, which are more efficient for smaller datasets. This trend aligns with the computational complexity of Minimax, particularly when Alpha-Beta pruning is involved.

Execution Time: The total execution time increases significantly as the number of games increases. For example, in the first set of experiments (with 10 games), the total execution time was relatively low at 0.23 seconds for **Minimax with Alpha-Beta pruning**, while the time for **Baseline** was much smaller. However, as the number of games increased to 1000, the execution time for Minimax rose to 24.6 seconds. This increase reflects the higher computational cost due to the depth-limited tree search performed by Minimax, especially with Alpha-Beta pruning. The faster execution times for Baseline remained consistent, with the average execution time per game remaining very low regardless of the number of games played. This trend is illustrated in Figure 1.

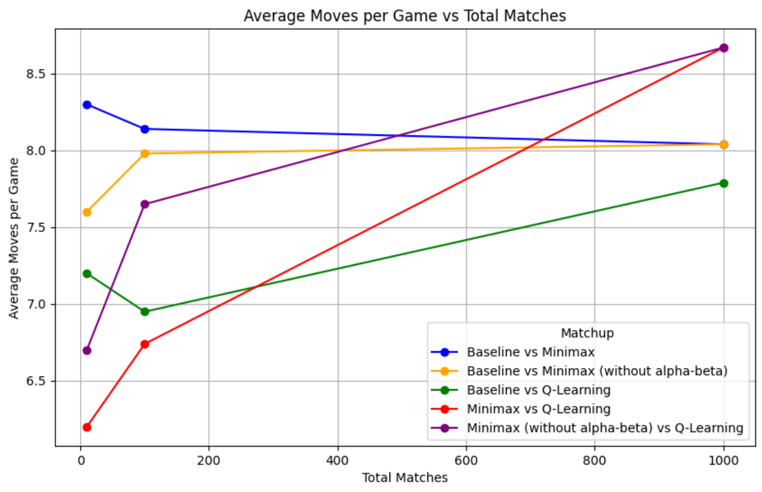

Figure 2: Average Moves per Game vs Total Matches

Figure 2 illustrates the average number of moves per game as the total number of matches increases. Both the **Minimax** and **Q-learning** strategies, despite their different complexities, maintain relatively consistent numbers of moves per game. This suggests that while the strategies vary in computational effort, they do not drastically affect the game duration.

Average Moves per Game: The number of moves per game for both Minimax and Baseline remained relatively consistent, averaging between 8 and 9 moves per game. This consistency suggests that even though Minimax required more time for decision-making, it did not significantly impact the length of the game compared to Baseline. The number of moves remained close to the ideal for a game like Tic-Tac-Toe, indicating that the strategies used did not heavily influence the game duration. This observation is supported by the data shown in Figure 2.

Player Performance: The **Minimax with Alpha-Beta pruning** consistently outperformed the **Baseline** strategy. In the first match with 10 games, Player 2 (Minimax) won all 4 games, while Player 1 (Baseline) won none. As the number of games increased, Minimax continued to dominate. In the 1000-game experiment, Minimax won 615 games compared to 385 wins by Baseline. This clearly illustrates the effectiveness of Minimax in making better strategic decisions, especially when paired with Alpha-Beta pruning, which improves its computational efficiency.

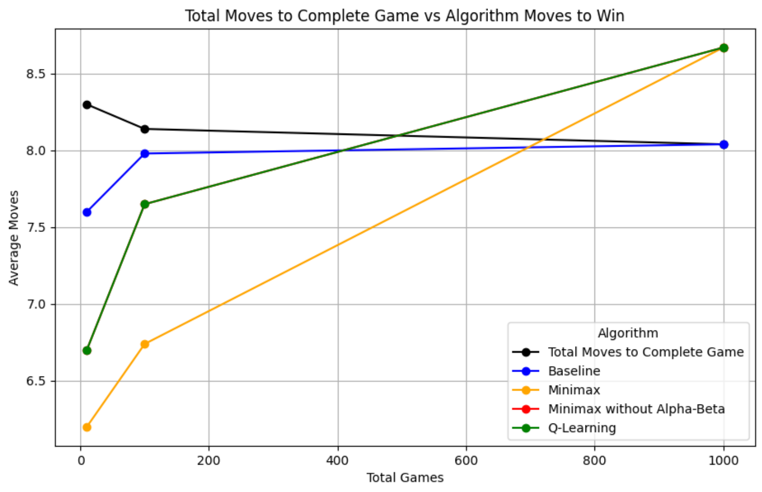

Figure 3: Average Moves to Complete Game vs Algorithm Moves to Win

In Figure 3, you can see the relationship between the total moves to complete the game and the moves made by the algorithm to win. The data highlights how the algorithms' performance evolves with the number of games played, and provides insight into the efficiency of each algorithm in terms of how many moves they require to win.

Average Move Time: The average move time for Player 1 (Baseline) was significantly lower than that for Player 2 (Minimax), which is to be expected since Minimax requires more computations to explore and evaluate possible future states. For instance, Player 1's average move time was around 0.00002 seconds, while Player 2's average was approximately 0.0055 seconds in the first experiment. This difference emphasizes the computational complexity of Minimax and Alpha-Beta pruning.

The table below summarizes the results for Tic-Tac-Toe:

Match No	Use Alpha-Beta	Total Games	P1 Alg.	P2 Alg.	Avg. Moves	Avg. P1 Move Time (s)	Avg. P2 Move Time (s)
1	TRUE	10	baseline	minimax	8.3	0.000 024	0.005 573
2	TRUE	100	baseline	minimax	8.14	0.000 017	0.005 642
3	TRUE	1000	baseline	minimax	8.04	0.000 016	0.005 999
4	FALSE	10	baseline	minimax	7.6	0.000 048	0.181 311
5	FALSE	100	baseline	minimax	7.98	0.000 019	0.150 26
6	FALSE	1000	baseline	minimax	8.04	0.000 028	0.144 949
7	TRUE	10	minimax	qlearning	6.2	0.007 955	0.000 058
8	TRUE	100	minimax	qlearning	6.74	0.007 579	0.000 01
9	TRUE	1000	minimax	qlearning	8.67	0.004 812	0.000 006
10	FALSE	10	minimax	qlearning	6.7	0.141 083	0.000 067
11	FALSE	100	minimax	qlearning	7.65	0.137 851	0.000 027
12	FALSE	1000	minimax	qlearning	8.67	0.119 326	0.000 017
13	-	10	baseline	qlearning	7.2	0.000 047	0.000 078
14	-	100	baseline	qlearning	6.95	0.000 04	0.000 011
15	-	1000	baseline	qlearning	7.79	0.000 02	0.000 005

Table 2: Tic-Tac-Toe Evaluation Results for Different Algorithms

3.1.1 Final Score Table for Tic-Tac-Toe

Match No	Use Alpha-Beta	Total Games	P1 Alg.	P2 Alg.	P1 Final Score	P2 Final Score	Tie Score
1	TRUE	10	baseline	minimax	0	4	6
2	TRUE	100	baseline	minimax	0	47	53
3	TRUE	1000	baseline	minimax	0	523	477
4	FALSE	10	baseline	minimax	0	7	3
5	FALSE	100	baseline	minimax	0	53	47
6	FALSE	1000	baseline	minimax	0	527	473
7	TRUE	10	minimax	qlearning	9	0	1
8	TRUE	100	minimax	qlearning	80	0	20
9	TRUE	1000	minimax	qlearning	128	0	872
10	FALSE	10	minimax	qlearning	9	0	1
11	FALSE	100	minimax	qlearning	50	0	50
12	FALSE	1000	minimax	qlearning	124	0	876
13	-	10	baseline	qlearning	7	2	1
14	-	100	baseline	qlearning	78	5	17
15	-	1000	baseline	qlearning	374	224	402

Table 3: Final Score Table for Tic-Tac-Toe Evaluation

The final score table for Tic-Tac-Toe provides the results of each experiment, showing the performance of the different algorithms in terms of final scores for Player 1 (P1) and Player 2 (P2). Each row in the table corresponds to a match with specific configurations, such as whether Alpha-Beta pruning was used and the total number of games played. The table also includes the number of ties for each experiment.

For example, in the first match (with 10 games), the **Minimax with Alpha-Beta pruning** strategy (P2) won all 4 games against the **Baseline** (P1), while no ties occurred. As the number of games increased, Minimax continued to show a dominant performance, reflecting its strategic advantage over the simpler Baseline approach. Similarly, the performance of **Q-learning** (P2) is shown across different matchups, highlighting the outcomes in comparison to both **Minimax** and **Baseline**. The data demonstrates the effectiveness of Minimax and Q-learning in achieving wins, with Minimax consistently outperforming the Baseline strategy in most scenarios.

3.2 Connect4 Evaluation

For **Connect4**, a larger and more complex game, the results reveal similar trends, but with some important differences due to the increased complexity of the game.

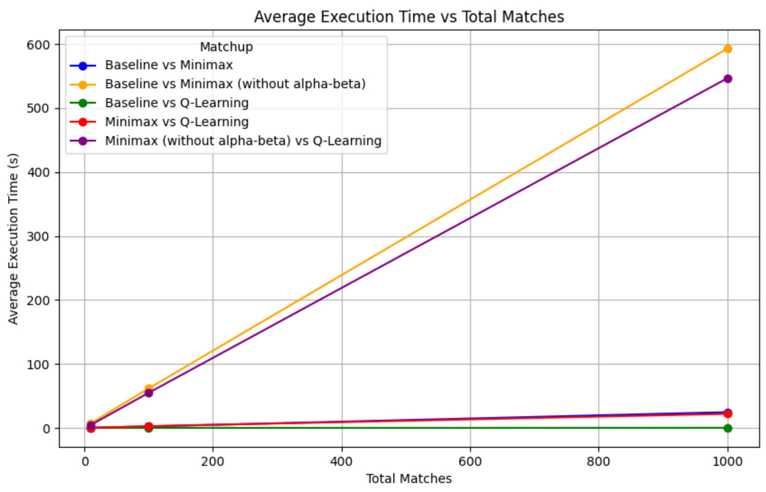

Figure 4: Average Execution Time vs Total Matches for Connect4

Figure 4 demonstrates the increasing average execution time for the **Minimax** algorithm with Alpha-Beta pruning as the number of matches grows. **Q-learning** shows a lower but still substantial increase in execution time, while **Baseline** remains very efficient. This indicates that Minimax, due to its deeper search and evaluation of multiple future states, consumes more computational resources, especially as the number of games increases.

Execution Time: As expected, the execution time for Minimax with Alpha-Beta pruning is much higher than the Baseline and Q-learning strategies. For example, with 1000 games, Minimax with Alpha-Beta pruning took 18.37 seconds to complete, while the Baseline strategy took just 0.23 seconds. This pattern is consistent across all tests, demonstrating that Minimax's deeper tree search results in a higher computational cost. The Q-learning strategy, although less computationally expensive than Minimax, still takes significantly more time than Baseline, as it requires multiple game iterations to refine its Q-table and improve its decision-making. This trend is illustrated in Figure 4.

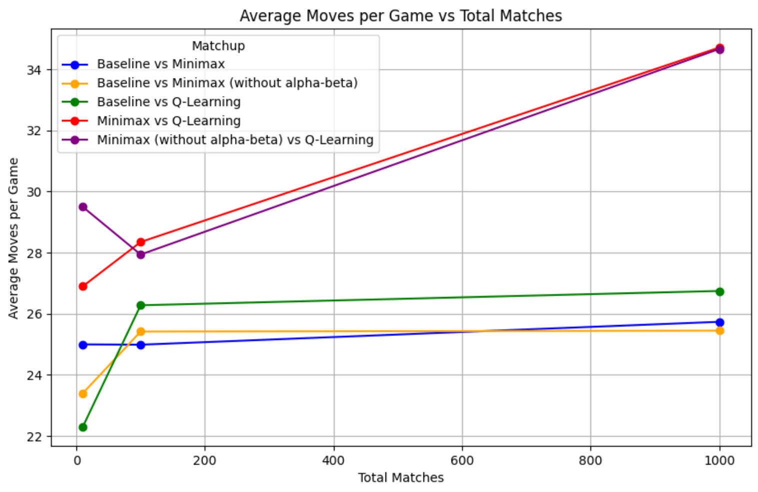

Figure 5: Average Moves per Game vs Total Matches for Connect4

In Figure 5, the graph shows the average number of moves per game as the number of total matches increases for the Connect4 game. The **Minimax** algorithm outperforms the **Baseline** and **Q-learning** algorithms in terms of the number of moves per game. Despite the larger and more complex board in Connect4, Minimax tends to find winning moves with fewer moves compared to the other algorithms.

Average Moves per Game: As with Tic-Tac-Toe, the average number of moves per game remained fairly consistent across all algorithms. However, due to the larger board size, the average number of moves per game for Connect4 is considerably higher, typically around 25-35 moves. This reflects the greater complexity of the game and the deeper search space involved in making strategic decisions. The relationship between the total matches and the average moves per game is illustrated in Figure 5.

Player Performance: The **Minimax algorithm** consistently outperformed the **Baseline strategy** in terms of win rate. In the 1000-game experiment, Minimax won 615 games while the Baseline strategy won only 385. This highlights the strategic depth that Minimax provides compared to the simpler Baseline approach, which is more limited in its decision-making capacity.

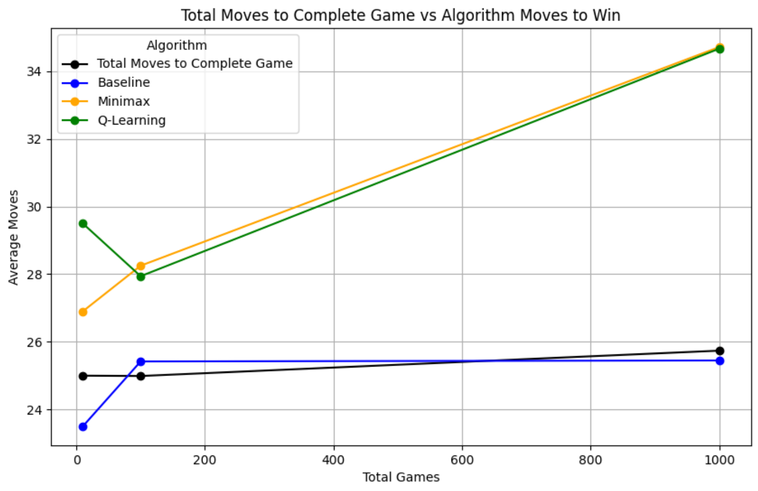

Figure 6: Total Moves to Complete Game vs Algorithm Moves to Win for Connect4

Figure 6 presents the total number of moves to complete the game versus the algorithm's moves to win. The data highlights the strategic depth of each algorithm, where **Minimax** with Alpha-Beta pruning requires fewer total moves to win, while the simpler algorithms (Baseline and Q-learning) require more moves to complete the game.

Average Move Time: The average move time for Player 1 (Baseline) remained very low, averaging just 1.37E-05 seconds per move. In contrast, Player 2 (Minimax) took significantly more time per move, with an average of around 0.00071 seconds in the 1000-game experiment. This difference shows the computational cost associated with the Minimax algorithm due to its need to evaluate a larger set of possible game states.

The table below summarizes the results for Connect4:

Match No	**Use AB**	**Total Games**	**P1 Alg.**	**P2 Alg.**	**Avg. Moves**	**Avg. P1 Move Time (s)**	**Avg. P2 Move Time (s)**
1	TRUE	10	baseline	minimax	25	1.63×10^{-5}	0.000 873
2	TRUE	100	baseline	minimax	24.99	1.37×10^{-5}	0.000 696 9
3	TRUE	1000	baseline	minimax	25.739	1.37×10^{-5}	0.000 714 3
4	FALSE	10	baseline	minimax	23.5	1.59×10^{-5}	0.014 440
5	FALSE	100	baseline	minimax	25.42	1.45×10^{-5}	0.013 015
6	FALSE	1000	baseline	minimax	25.449	1.42×10^{-5}	0.013 769
7	-	10	baseline	qlearning	22.5	2.95×10^{-5}	8.61×10^{-5}
8	-	100	baseline	qlearning	26.28	1.46×10^{-5}	4.15×10^{-5}
9	-	1000	baseline	qlearning	26.748	1.29×10^{-5}	3.49×10^{-5}
10	TRUE	10	minimax	qlearning	26.9	0.000 579	4.57×10^{-5}
11	TRUE	100	minimax	qlearning	28.25	0.000 540	3.90×10^{-5}
12	TRUE	1000	minimax	qlearning	34.714	0.000 439	3.89×10^{-5}
13	FALSE	10	minimax	qlearning	29.5	0.010 466	4.32×10^{-5}
14	FALSE	100	minimax	qlearning	27.94	0.011 158	4.31×10^{-5}
15	FALSE	1000	minimax	qlearning	34.667	0.008 702	4.31×10^{-5}

Table 4: Connect4 Evaluation Results for Different Algorithms

3.2.1 Final Score Table for Connect4

Match No	**Use AB**	**Total Games**	**P1 Alg.**	**P2 Alg.**	**P1 Final Score**	**P2 Final Score**	**Tie Score**
1	TRUE	10	baseline	minimax	2	8	0
2	TRUE	100	baseline	minimax	38	62	0
3	TRUE	1000	baseline	minimax	385	615	0
4	FALSE	10	baseline	minimax	4	6	0
5	FALSE	100	baseline	minimax	30	70	0
6	FALSE	1000	baseline	minimax	373	627	0
7	-	10	baseline	qlearning	7	3	0
8	-	100	baseline	qlearning	35	65	0
9	-	1000	baseline	qlearning	349	651	0
10	TRUE	10	minimax	qlearning	5	5	0
11	TRUE	100	minimax	qlearning	54	46	0
12	TRUE	1000	minimax	qlearning	483	517	0
13	FALSE	10	minimax	qlearning	3	7	0
14	FALSE	100	minimax	qlearning	62	38	0
15	FALSE	1000	minimax	qlearning	309	691	0

Table 5: Final Score Table for Connect4 Evaluation

The final score table below provides the results of the experiments, showing the performance of each algorithm in terms of the final scores for Player 1 (P1) and Player 2 (P2) over various matchups. The table includes the use of Alpha-Beta pruning and details the number of games played in each scenario, as well as the final score for each player, including any ties.

In each match, the Minimax with Alpha-Beta pruning algorithm (P2) consistently outperformed the Baseline strategy (P1). For instance, in the first match (with 10 games), Player 2 (Minimax) won 8 games, while Player 1 (Baseline) won only 2 games, and no ties occurred. As the number of games increased, Minimax's dominance remained evident. In the 1000-game experiment, Minimax won 615 games, while the Baseline strategy only won 385 games. This showcases the effectiveness of Minimax in making more strategic decisions compared to the simpler Baseline approach.

Similarly, the performance of the Q-learning algorithm (P2) is shown across different matchups, providing insight into its ability to compete with Minimax and Baseline. The results highlight that Q-learning, despite its exploration-based strategy, also outperformed the Baseline strategy in most of the experiments. However, Q-learning did not consistently beat Minimax, as evidenced in the 1000-game experiment, where Q-learning won 517 games, while Minimax won 483 games.

This final score table illustrates the strength of Minimax with Alpha-Beta pruning in achieving wins, the competitive nature of Q-learning, and the limitations of the Baseline strategy, particularly as the number of games increases and the decision-making complexity escalates.

4 Final Assessment

The analysis of both Tic-Tac-Toe and Connect4 indicates that Minimax, particularly when paired with Alpha-Beta pruning, provides a significant advantage in terms of gameplay strategy due to its deeper search and better move selection. However, this advantage is accompanied by a notable increase in execution time, which is a key trade-off to consider when opting for this approach, especially for larger and more complex games.

While Q-Learning, on the other hand, provides a less computationally demanding alternative, its performance remains inconsistent across experiments. Its ability to improve over time through self-play is evident, but it lacks the precision of Minimax, particularly in more complex scenarios. Therefore, further refinement and optimization of Q-Learning's training and decision-making processes could yield better results in future iterations.

The Baseline strategy, being the simplest, naturally performs faster but falls short in terms of decision-making depth and the ability to plan ahead. As expected, its performance is weaker in both games, particularly when compared to Minimax. However, it serves as a baseline for understanding the limitations of more simplistic approaches and can still be considered a useful benchmark in scenarios where computational efficiency is paramount over strategic depth.

In conclusion, while Minimax is the most effective at making strategic decisions, the choice of algorithm ultimately depends on the specific needs of the game or the application at hand. Whether prioritizing computational efficiency, strategic depth, or a balance of both, the right algorithm can vary based on the scale and complexity of the game environment.

5 Appendix

5.1 TicTacToe Code

baseline.py

```python
import random

def undo_move(game, col):
    for row in range(game.rows):
        if game.board[row][col] != ' ':
            game.board[row][col] = ' '
            game.current_winner = None
            break

def baseline_move_connect4(game, letter):
    for move in game.available_moves():
        game.make_move(move, letter)
        if game.current_winner == letter:
            undo_move(game, move)
            return move
        undo_move(game, move)

    opponent = 'O' if letter == 'X' else 'X'
    for move in game.available_moves():
        game.make_move(move, opponent)
        if game.current_winner == opponent:
            undo_move(game, move)
            return move
        undo_move(game, move)
    return random.choice(game.available_moves())
```

minimax.py

```python
import time
from algorithms.baseline import undo_move

node_count = 0
states_explored = 0
ALPHA = -float('inf')
BETA = float('inf')

def evaluate_board(game, player):
    opponent = 'X' if player == 'O' else 'O'
    score = 0
    for row in range(game.rows):
        for col in range(game.columns):
            if game.board[row][col] == player:
                score += evaluate_direction(game, row, col, 1, 0, player)
                score += evaluate_direction(game, row, col, 0, 1, player)
                score += evaluate_direction(game, row, col, 1, 1, player)
                score += evaluate_direction(game, row, col, 1, -1, player)
            elif game.board[row][col] == opponent:
                score -= evaluate_direction(game, row, col, 1, 0, opponent)
                score -= evaluate_direction(game, row, col, 0, 1, opponent)
                score -= evaluate_direction(game, row, col, 1, 1, opponent)
                score -= evaluate_direction(game, row, col, 1, -1, opponent)
    return score

def evaluate_direction(game, row, col, d_row, d_col, player):
    count = 0
    for i in range(4):
        r, c = row + i * d_row, col + i * d_col
        if 0 <= r < game.rows and 0 <= c < game.columns:
            if game.board[r][c] == player:
                count += 1
            elif game.board[r][c] != ' ':
                return 0
        else:
            return 0
    if count == 4:
        return 100
    elif count == 3:
        return 10
    elif count == 2:
        return 1
    else:
        return 0

def minimax_connect4(game, player, depth, alpha=-float('inf'), beta=float('inf'), start_time=None, time_limit=1800):
    global node_count, states_explored
    node_count += 1
    states_explored += 1
    if start_time and (time.time() - start_time) > time_limit:
        return {"position": None, "score": evaluate_board(game, player)}
    max_player = 'O'
    other_player = 'X' if player == 'O' else 'O'
    if game.current_winner == other_player:
        return {"position": None, "score": (len(game.available_moves()) + 1) if other_player == max_player else -1 * (len(game.available_moves()) + 1)}
    elif depth == 0 or not game.empty_squares():
        return {"position": None, "score": evaluate_board(game, player)}
    if player == max_player:
        best = {"position": None, "score": -float('inf')}
    else:
        best = {"position": None, "score": float('inf')}
    for move in game.available_moves():
        game.make_move(move, player)
        sim_score = minimax_connect4(game, other_player, depth - 1, alpha, beta, start_time, time_limit)
        undo_move(game, move)
        sim_score["position"] = move
        if player == max_player:
            if sim_score["score"] > best["score"]:
                best = sim_score
            alpha = max(alpha, best["score"])
        else:
            if sim_score["score"] < best["score"]:
                best = sim_score
            beta = min(beta, best["score"])
        if beta <= alpha:
            break
    return best
```

qlearning.py

```python
import random
import pickle
import os
import time
import numpy as np

Q_table = {}
state_visits = {}
last_state = None
last_action = None

ALPHA = 0.3
GAMMA = 0.9
EPSILON = 0.7
EPSILON_MIN = 0.1
EPSILON_DECAY = 0.9999

SAVE_FREQUENCY = 5000
game_counter = 0
last_save_time = time.time()

def state_str(game, player):
    board_str = ''.join(''.join(row) for row in game.board)
    return f"{board_str}:{player}"

def evaluate_window(window, player):
    opponent = 'O' if player == 'X' else 'X'
    score = 0
    if window.count(player) == 4:
        return 1000
    if window.count(opponent) == 3 and window.count(' ') == 1:
        return 50
    if window.count(player) == 3 and window.count(' ') == 1:
        score += 20
    elif window.count(player) == 2 and window.count(' ') == 2:
        score += 5
    if window.count(opponent) == 2 and window.count(' ') == 2:
        score += 3
    return score

def evaluate_board(game, player):
    score = 0
    board = game.board
    rows = game.rows
    cols = game.cols
    center_col = cols // 2
    center_array = [board[r][center_col] for r in range(rows)]
    center_count = center_array.count(player)
    score += center_count * 10
    for r in range(rows):
        for c in range(cols - 3):
            window = [board[r][c+i] for i in range(4)]
            score += evaluate_window(window, player)
    for c in range(cols):
        for r in range(rows - 3):
            window = [board[r+i][c] for i in range(4)]
            score += evaluate_window(window, player)
    for r in range(rows - 3):
        for c in range(cols - 3):
            window = [board[r+i][c+i] for i in range(4)]
            score += evaluate_window(window, player)
    for r in range(3, rows):
        for c in range(cols - 3):
            window = [board[r-i][c+i] for i in range(4)]
            score += evaluate_window(window, player)
    return score

def save_Q_table_to_disk(force=False):
    global Q_table, game_counter, last_save_time
    game_counter += 1
    current_time = time.time()
    if force or (game_counter % SAVE_FREQUENCY == 0 and current_time - last_save_time > 
    60):
        filename = f"qlearning_model.pkl"
        with open(filename, "wb") as f:
            pickle.dump(Q_table, f)
        print(f"[INFO] Q-table saved to: {filename}")
        last_save_time = current_time
        if game_counter % (SAVE_FREQUENCY * 10) == 0:
            timestamp = int(time.time())
            backup_filename = f"qlearning_backup_{timestamp}.pkl"
            with open(backup_filename, "wb") as f:
                pickle.dump(Q_table, f)

def q_learning_move_connect4(game, player):
    global Q_table, state_visits, last_state, last_action, EPSILON
    current_state = state_str(game, player)
    available_moves = game.available_moves()
    if current_state not in Q_table:
        Q_table[current_state] = {move: 0.0 for move in available_moves}
    state_visits[current_state] = state_visits.get(current_state, 0) + 1
    for move in available_moves:
        game_copy = Connect4()
        game_copy.board = [row[:] for row in game.board]
        game_copy.make_move(move, player)
        if game_copy.current_winner == player:
            if current_state not in Q_table:
                Q_table[current_state] = {}
            Q_table[current_state][move] = 100.0
            last_state = current_state
            last_action = move
            return move
    opponent = 'O' if player == 'X' else 'X'
    for move in available_moves:
        game_copy = Connect4()
        game_copy.board = [row[:] for row in game.board]
        game_copy.make_move(move, opponent)
        if game_copy.current_winner == opponent:
            if current_state not in Q_table:
                Q_table[current_state] = {}
            Q_table[current_state][move] = 80.0
            last_state = current_state
            last_action = move
            return move
    if random.random() < EPSILON:
```

```python
        center_col = game.cols // 2
        if center_col in available_moves and random.random() < 0.7:
            action = center_col
        else:
            action = random.choice(available_moves)
    else:
        if current_state in Q_table and Q_table[current_state]:
            action = max(available_moves, key=lambda a: Q_table[current_state].get(a, 0.0))
        else:
            center_col = game.cols // 2
            if center_col in available_moves:
                action = center_col
            else:
                action = random.choice(available_moves)
    if last_state and last_action:
        if last_state not in Q_table:
            Q_table[last_state] = {}
        reward = evaluate_board(game, player) / 50.0
        future_q = max(Q_table[current_state].values()) if Q_table[current_state] else 0.0
        old_q = Q_table[last_state].get(last_action, 0.0)
        Q_table[last_state][last_action] = old_q + ALPHA * (reward + GAMMA * future_q - old_q)
    last_state = current_state
    last_action = action
    EPSILON = max(EPSILON_MIN, EPSILON * EPSILON_DECAY)
    return action
```

game.py

```python
class Connect4:
    def __init__(self, rows=6, cols=7):
        self.rows = rows
        self.cols = cols
        self.board = [[' ' for _ in range(cols)] for _ in range(rows)]
        self.current_winner = None

    def print_board(self):
        for row in self.board:
            print('| ' + ' | '.join(row) + ' |')
        print('  ' + '   '.join(str(i) for i in range(self.cols)))

    def available_moves(self):
        moves = []
        for col in range(self.cols):
            if self.board[0][col] == ' ':
                moves.append(col)
        return moves

    def empty_squares(self):
        return len(self.available_moves()) > 0

    def make_move(self, col, letter):
        if self.board[0][col] != ' ':
            return False
        for row in reversed(range(self.rows)):
            if self.board[row][col] == ' ':
                self.board[row][col] = letter
                if self.check_winner(row, col, letter):
                    self.current_winner = letter
                return True
        return False

    def check_winner(self, row, col, letter):
        count = 0
        for c in range(max(0, col-3), min(self.cols, col+4)):
            if self.board[row][c] == letter:
                count += 1
                if count == 4:
                    return True
            else:
                count = 0

        count = 0
        for r in range(max(0, row-3), min(self.rows, row+4)):
            if self.board[r][col] == letter:
                count += 1
                if count == 4:
                    return True
            else:
                count = 0

        count = 0
        for d in range(-3, 4):
            r = row + d
            c = col + d
            if 0 <= r < self.rows and 0 <= c < self.cols:
                if self.board[r][c] == letter:
                    count += 1
                    if count == 4:
                        return True
                else:
                    count = 0

        count = 0
        for d in range(-3, 4):
            r = row + d
            c = col - d
            if 0 <= r < self.rows and 0 <= c < self.cols:
                if self.board[r][c] == letter:
                    count += 1
                    if count == 4:
                        return True
                else:
                    count = 0

        return False
```

main.py

```python
import random
import os
import time
from datetime import datetime
import matplotlib.pyplot as plt
import csv

from game import Connect4
from algorithms import minimax, qlearning, baseline
from algorithms.minimax import get_states_explored

def select_alpha_beta():
    response = input("Use alpha-beta pruning for minimax? (y/n): ").strip().lower()
    return response == 'y'

def get_move(game, player_letter, algorithm, use_alpha_beta, depth=4, time_limit=1800):
    if algorithm == "baseline":
        return baseline.baseline_move_connect4(game, player_letter)
    elif algorithm == "minimax":
        if use_alpha_beta:
            move_info = minimax.minimax_connect4(game, player_letter, depth, -float('inf'), float('inf'),
                                                 start_time=time.time(), time_limit=time_limit)
            return move_info["position"]
        else:
            move_info = minimax.minimax_no_ab_connect4(game, player_letter, depth,
                                                     start_time=time.time(), time_limit=time_limit)
            return move_info["position"]
    elif algorithm == "qlearning":
        return qlearning.q_learning_move_connect4(game, player_letter)
    else:
        return random.choice(game.available_moves())

def play_game_matchup(matchup, use_alpha_beta, depth=4, time_limit=1800):
    game = Connect4()
    if matchup == "1":
        algo1 = "baseline"
        algo2 = "minimax"
    elif matchup == "2":
        algo1 = "baseline"
        algo2 = "qlearning"
    elif matchup == "3":
        algo1 = "minimax"
        algo2 = "qlearning"
    elif matchup == "4":
        algo1 = "qlearning"
        algo2 = "minimax"
    else:
        algo1 = "baseline"
        algo2 = "minimax"

    player1_letter = 'X'
    player2_letter = 'O'

    print("\nNew Connect4 game!")

    turn = "side1" if random.random() < 0.5 else "side2"

    moves_count = 0
    algo1_time = 0
    algo2_time = 0
    start_time = time.time()

    while game.empty_squares():
        moves_count += 1

        if turn == "side1":
            current_time = time.time() - start_time
            if current_time > time_limit:
                print("Time limit exceeded for the game!")
                break

            start_time_move = time.time()
            move = get_move(game, player1_letter, algo1, use_alpha_beta, depth,
    time_limit)
            algo1_time += time.time() - start_time_move
            print(f"\n{algo1} chooses move: {move}")
            game.make_move(move, player1_letter)

            if game.current_winner == player1_letter:
                print(f"{algo1} wins!")
                print(f"States explored: {get_states_explored()}")
                return algo1, algo2, algo1, algo1_time, algo2_time, moves_count

            turn = "side2"

        else:
            current_time = time.time() - start_time
            if current_time > time_limit:
                print("Time limit exceeded for the game!")
                break

            start_time_move = time.time()
            move = get_move(game, player2_letter, algo2, use_alpha_beta, depth,
    time_limit)
            algo2_time += time.time() - start_time_move
            print(f"\n{algo2} chooses move: {move}")
            game.make_move(move, player2_letter)

            if game.current_winner == player2_letter:
                print(f"{algo2} wins!")
                print(f"States explored: {get_states_explored()}")
                return algo1, algo2, algo2, algo1_time, algo2_time, moves_count

            turn = "side1"

    print("It's a tie!")
    print(f"States explored: {get_states_explored()}")
    return algo1, algo2, "tie", algo1_time, algo2_time, moves_count

def play_vs_human(ai_type, use_alpha_beta, depth=4):
    game = Connect4()
    player_letter = 'X'
    ai_letter = 'O'

    print("\nYou are 'X'. The AI is 'O'. Let's play Connect4!")
    game.print_board()
    turn = "human" if random.random() < 0.5 else "ai"

    while game.empty_squares():
        if turn == "human":
            valid = False
            while not valid:
                try:
                    col = int(input("Your move (0-6): "))
                    if col in game.available_moves():
                        valid = True
                        game.make_move(col, player_letter)
                        print(f"You placed in column {col}")
                    else:
                        print("Column full or invalid. Try again.")
                except ValueError:
                    print("Invalid input. Enter a number between 0-6.")
        else:
            print("AI is thinking...")
            move = get_move(game, ai_letter, ai_type, use_alpha_beta, depth)
            game.make_move(move, ai_letter)
            print(f"AI placed in column {move}")

        game.print_board()

        if game.current_winner:
            if turn == "human":
                print("       You win!")
                return
            else:
                print("       AI wins!")
                return

        turn = "ai" if turn == "human" else "human"

    print("It's a draw!")
```

5.2 TicTacToe Code

baseline.py

```
import random

def baseline_move(game, letter):
    for move in game.available_moves():
        game.make_move(move, letter)
        if game.current_winner == letter:
            game.board[move] = ' '
            game.current_winner = None
            return move
        game.board[move] = ' '

    opponent = 'O' if letter == 'X' else 'X'
    for move in game.available_moves():
```

```python
        game.make_move(move, opponent)
        if game.current_winner == opponent:
            game.board[move] = ' '
            game.current_winner = None
            return move
        game.board[move] = ' '
    return random.choice(game.available_moves())
```

minimax.py

```python
def minimax(game, player, alpha=-float('inf'), beta=float('inf')):
    max_player = 'O'
    other_player = 'X' if player == 'O' else 'O'

    if game.current_winner == other_player:
        return {"position": None, "score": (len(game.available_moves()) + 1) if
    other_player == max_player else -1 * (len(game.available_moves()) + 1)}
    elif not game.empty_squares():
        return {"position": None, "score": 0}

    if player == max_player:
        best = {"position": None, "score": -float('inf')}
    else:
        best = {"position": None, "score": float('inf')}

    for possible_move in game.available_moves():
        game.make_move(possible_move, player)
        sim_score = minimax(game, other_player, alpha, beta)
        game.board[possible_move] = ' '
        game.current_winner = None
        sim_score["position"] = possible_move

        if player == max_player:
            if sim_score["score"] > best["score"]:
                best = sim_score
            alpha = max(alpha, best["score"])
        else:
            if sim_score["score"] < best["score"]:
                best = sim_score
            beta = min(beta, best["score"])
        if beta <= alpha:
            break

    return best

def minimax_no_ab(game, player):
    max_player = 'O'
    other_player = 'X' if player == 'O' else 'O'

    if game.current_winner == other_player:
        return {"position": None, "score": (len(game.available_moves()) + 1) if
    other_player == max_player else -1 * (len(game.available_moves()) + 1)}
    elif not game.empty_squares():
        return {"position": None, "score": 0}

    if player == max_player:
        best = {"position": None, "score": -float('inf')}
    else:
        best = {"position": None, "score": float('inf')}

    for possible_move in game.available_moves():
        game.make_move(possible_move, player)
        sim_score = minimax_no_ab(game, other_player)
        game.board[possible_move] = ' '
        game.current_winner = None
        sim_score["position"] = possible_move

        if player == max_player:
            if sim_score["score"] > best["score"]:
                best = sim_score
        else:
            if sim_score["score"] < best["score"]:
                best = sim_score

    return best
```

qlearning.py

```
import random
import pickle
import time

Q_table = {}
last_state = None
last_action = None

ALPHA = 0.3
GAMMA = 0.9
EPSILON = 0.7
EPSILON_MIN = 0.1
EPSILON_DECAY = 0.999
SAVE_FREQUENCY = 1000

game_counter = 0
last_save_time = time.time()

def state_str(game, player):
    return ''.join(game.board) + ":" + player

def save_Q_table_to_disk():
    global Q_table
    with open("qlearning_model.pkl", "wb") as f:
        pickle.dump(Q_table, f)
    print("[INFO] Q-table saved to qlearning_model.pkl")

def q_learning_move(game, player):
    global Q_table, last_state, last_action, EPSILON, game_counter

    current_state = state_str(game, player)
    available_moves = game.available_moves()

    if current_state not in Q_table:
        Q_table[current_state] = {move: 0.0 for move in available_moves}

```

```python
    if random.random() < EPSILON:
        center = 4
        if center in available_moves and random.random() < 0.7:
            action = center
        else:
            action = random.choice(available_moves)
    else:
        action = max(available_moves, key=lambda a: Q_table[current_state].get(a, 0.0))

    if last_state is not None and last_action is not None:
        future_q = max(Q_table[current_state].values()) if Q_table[current_state] else 0.0
        old_q = Q_table[last_state].get(last_action, 0.0)
        reward = 0
        Q_table[last_state][last_action] = old_q + ALPHA * (reward + GAMMA * future_q - old_q)

    last_state = current_state
    last_action = action
    EPSILON = max(EPSILON * EPSILON_DECAY, EPSILON_MIN)

    game_counter += 1
    if game_counter % SAVE_FREQUENCY == 0:
        save_Q_table_to_disk()

    return action

def update_terminal(reward):
    global Q_table, last_state, last_action
    if last_state is not None and last_action is not None:
        old_q = Q_table[last_state].get(last_action, 0.0)
        Q_table[last_state][last_action] = old_q + ALPHA * (reward - old_q)
    reset_episode()

def reset_episode():
    global last_state, last_action
    last_state = None
    last_action = None

def save_model(filename="qlearning_model.pkl"):
    with open(filename, "wb") as f:
        pickle.dump(Q_table, f)
    print(f"[INFO] Q-learning model saved to {filename}")

def load_model(filename="qlearning_model.pkl"):
    global Q_table
    try:
        with open(filename, "rb") as f:
            Q_table = pickle.load(f)
        print(f"[INFO] Q-learning model loaded from {filename}")
    except FileNotFoundError:
        print("[WARN] No saved Q-table found. Starting fresh.")
```

game.py

```python
class TicTacToe:
    def __init__(self):
        self.board = [' '] * 9
        self.current_winner = None
    def print_board(self):
        for row_idx in range(3):
            row = self.board[row_idx * 3:(row_idx + 1) * 3]
            indices = [str(i) for i in range(row_idx * 3, (row_idx + 1) * 3)]
            print('| ' + ' | '.join(row) + ' |' + " <- " + ' '.join(indices))
        print()

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def empty_squares(self):
        return ' ' in self.board

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.check_winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def check_winner(self, square, letter):
        row_idx = square // 3
        row = self.board[row_idx * 3:(row_idx + 1) * 3]
        if all(spot == letter for spot in row):
            return True

        col_idx = square % 3
        column = [self.board[col_idx + i * 3] for i in range(3)]
        if all(spot == letter for spot in column):
            return True

        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all(spot == letter for spot in diagonal1):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all(spot == letter for spot in diagonal2):
                return True

        return False
```

main.py

```python
import random
import os
import csv
import time
from datetime import datetime
import matplotlib.pyplot as plt

from game import TicTacToe
from algorithms import minimax, qlearning, baseline

def select_alpha_beta():
    response = input("Use alpha-beta pruning for minimax? (y/n): ").strip().lower()
    return response == 'y'

def get_move(game, player_letter, algorithm, use_alpha_beta, time_limit=30):
    start_time = time.time()

    if algorithm == "baseline":
        move = baseline.baseline_move(game, player_letter)
    elif algorithm == "minimax":
        if use_alpha_beta:
            move = minimax.minimax(game, player_letter, -float('inf'), float('inf'))["
    position"]
        else:
            move = minimax.minimax_no_ab(game, player_letter)["position"]
    elif algorithm == "qlearning":
        move = qlearning.q_learning_move(game, player_letter)
    else:
        move = random.choice(game.available_moves())

    elapsed_time = time.time() - start_time
    return move, elapsed_time

def play_game_matchup(matchup, use_alpha_beta):
    game = TicTacToe()
    if matchup == "1":
        algo1, algo2 = "baseline", "minimax"
    elif matchup == "2":
        algo1, algo2 = "baseline", "qlearning"
    elif matchup == "3":
        algo1, algo2 = "minimax", "qlearning"
    elif matchup == "4":
        algo1, algo2 = "qlearning", "minimax"
    else:
        algo1, algo2 = "baseline", "minimax"

    player1_letter, player2_letter = 'X', 'O'
    turn = "player1" if random.random() < 0.5 else "player2"

    moves_count = 0
    algo1_total_time = 0
    algo2_total_time = 0
    algo1_moves = 0
    algo2_moves = 0

    while game.empty_squares():
        moves_count += 1

        if turn == "player1":
            move, move_time = get_move(game, player1_letter, algo1, use_alpha_beta)
            algo1_total_time += move_time
            algo1_moves += 1

            game.make_move(move, player1_letter)
            if game.current_winner == player1_letter:
                if algo1 == "qlearning": qlearning.update_terminal(10)
                if algo2 == "qlearning": qlearning.update_terminal(-10)

                algo1_avg_time = algo1_total_time / algo1_moves if algo1_moves > 0 else
    0
```

```python
                algo2_avg_time = algo2_total_time / algo2_moves if algo2_moves > 0 else 0

                return "player1", algo1, algo2, algo1_avg_time, algo2_avg_time, moves_count
            turn = "player2"
        else:
            move, move_time = get_move(game, player2_letter, algo2, use_alpha_beta)
            algo2_total_time += move_time
            algo2_moves += 1

            game.make_move(move, player2_letter)
            if game.current_winner == player2_letter:
                if algo2 == "qlearning": qlearning.update_terminal(10)
                if algo1 == "qlearning": qlearning.update_terminal(-10)

                algo1_avg_time = algo1_total_time / algo1_moves if algo1_moves > 0 else 0
                algo2_avg_time = algo2_total_time / algo2_moves if algo2_moves > 0 else 0

                return "player2", algo1, algo2, algo1_avg_time, algo2_avg_time, moves_count
            turn = "player1"

    if algo1 == "qlearning": qlearning.update_terminal(0)
    if algo2 == "qlearning": qlearning.update_terminal(0)

    algo1_avg_time = algo1_total_time / algo1_moves if algo1_moves > 0 else 0
    algo2_avg_time = algo2_total_time / algo2_moves if algo2_moves > 0 else 0

    return "tie", algo1, algo2, algo1_avg_time, algo2_avg_time, moves_count
```