**Performance Evaluation of Q-Learning and Minimax Algorithms for Connect 4 and Tic Tac Toe Games**

## 1. Abstract

This report examines the application and comparative analysis of Minimax algorithm with and without alpha-beta pruning as well as Q-learning reinforcement learning on two classic board games: Tic Tac Toe and Connect 4. The games present different computational challenges as the state space of Connect 4 ($4^{42}$)being much larger than that of Tic Tac Toe ($3^9$). Experiments were conducted to test the performance using a semi-intelligent default opponent as well as against one another. The measures of performance including the win rate, execution time, and nodes visited were collected and compared to analyse these algorithms. Tests demonstrate that while both versions of Minimax generate close-to-optimal play with equal win rates, alpha-beta pruning significantly reduces computational overhead with reductions by as much as 52% and 40% in the case of the nodes visited in Tic Tac Toe and Connect 4, respectively. Q-learning competes well in Connect 4 but not in the case of Tic Tac Toe, exhibiting environment-dependent learning behaviour. This comparative study provides insight into algorithm selection trade-offs for the games with varying degrees of complexity and to understand the nature of search-based and learning-based methods to AI for playing a game.

## 2. Introduction

These Game-playing algorithms are being in a dominant theme of artificial intelligence research where it is offering well-specified environments wherein computational decision-making methods can be developed and evaluated. Board games have been a test bed of preference since the very beginning of AI research activities because they possess well-defined rules, finite state space, and performance metrics with well-defined objectives. In this report, on employing and comparing the two classically different algorithmic methodologies—Minimax search and Q-learning reinforcement learning—to play the games Tic Tac Toe and Connect 4.

These two games provide the interesting opposition in complexity. A 3×3 Tic Tac Toe board with a little over $3^9$ possible board positions when discounting symmetries and illegal boards represents quite a manageable problem that one could try with exhaustive methods. A 6×7 board for the Connect 4 case with a little over $4^{42}$ possible board positions increase the problem by a few orders of magnitude and tests the scalability of the algorithms. There are an interesting opposition makes visible how the performance of algorithms varies with problem complexity.

Minimax is the seminal search-based game playing strategy as it searches the possible future game states with the goal of choosing the optimal move. Alpha-beta pruning extends Minimax by trimming search tree branches never affecting the final decision, with potential significant performance improvement with no decrease in the quality of the solution. Here Q-learning addresses the issue with experiential learning, learning incrementally to make estimates for state-action pairs by playing the game repeatedly with no requirement for an explicit world model.

It tries to measure these algorithms in comparison across a range of dimensions:

1. Effectiveness against a default opponent and against each other
2. Computation efficiency of running time and nodes visited
3. Scaling from simple games (Tic Tac Toe) to complex (Connect 4) game environments

Among these, to elaborate the relative merits and disadvantages of the search-based vs. learning-based approaches to a played game, as well as performance variation by problem sizes.

### 3.1 Game Theory Fundamentals

The Game theory is a mathematical model for strategic decision making to win or lose. Two-person, zero-sum, and complete information games are in the structures for board games such as the Tac Toe and the game of Connect 4. In these games, players take turns making a move as the players have complete information, and one's victory is matched by the other's defeat here.

These games could be represented as a tree with the state of a game as the nodes and the possible steps as edges. A tree depth would be the quantity away from the initial state, and the terminal nodes would be the ones terminating the game-state as a win, a loss, or a tie. In games, the decision issue would be one concerning what move leads to the optimal outcome using rational behavior by the two players.

### 3.2 Minimax Algorithm

Minimax algorithm is a recursive depth-first backtracking algorithm to determine the optimal moves in two player zero-sum games. It makes recursive calls considering the future state of the game based on the assumption that at each move,

the maximizing opponent (generally the AI) would make the next possible move with the greater value and the opponent would make the move with the lower value (Campbell et al., 2002).

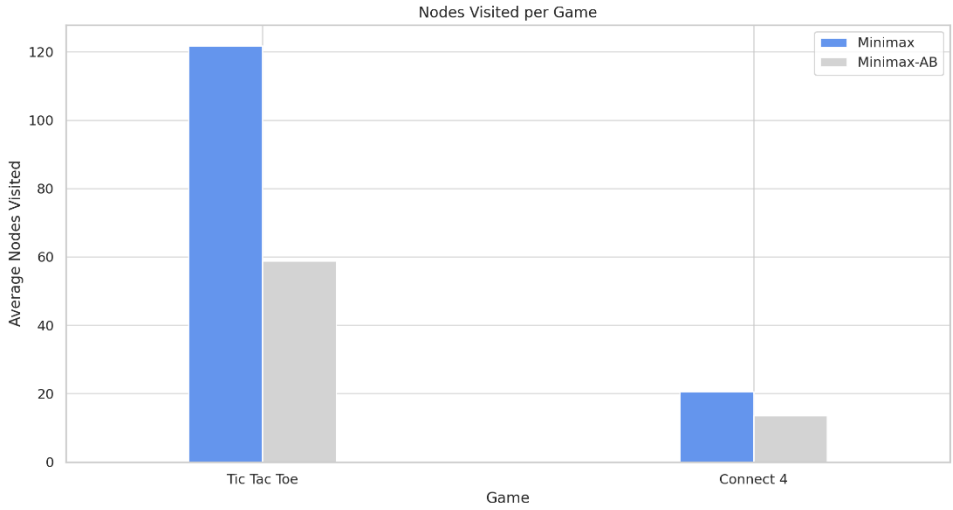


***Figure 1, Nodes Visited per Game***

### 3.2.1 Standard Minimax

Standard Minimax works by building a complete game tree down to some depth (down to terminal states). It then back-propagates the values. For a state s in each game, the Minimax value V(s) is defined recursively:

- If s represents a terminal state: V(s) = utility(s)
- If optimal for the next move:
- V(s) = max(V(s') over all possible future s')

If using the minimax algorithm with alpha-beta pruning by minimizing the players' turn:

Simple Minimax's main limitation is computational. Simple Minimax will examine O ($b^d$) nodes in a b-branching, d-depth game. As Figure 1 illustrates, for the case of Tic Tac Toe, this equates to a mean of 121 per move, which is acceptable. For the case of Connect 4 with depth constraint, the complexity is unmanageable.

### 3.2.2 Alpha-Beta Pruning

The alpha-beta pruning is an optimization technique of Minimax search that cuts out branches of the search tree that cannot affect the final decision. The standard algorithm looks for the same optimal decision while by significantly reducing the number of nodes evaluated, it is still optimized.

The algorithm maintains two values alpha and beta, which are the minimum and maximum score respectively that the maximizing player can be assured, and the minimizing player can be assured of. If, for instance, alpha ≥ beta at any time, then all branches at that node need not be explored because they cannot make any difference to the final decision (Knuth & Moore, 1975).

The best case for Alpha-Beta pruning is to find the number of nodes that need to be examined is $O\left(b^{\frac{d}{2}}\right)$, which effectively doubles the searchable depth. As shown in Figure 1, our implementation saves 52.3% and 40.0% of nodes on average for Tic Tac Toe (from 121 to 58 nodes) and for Connect 4 (from 20 to 12 nodes).

### 3.3 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where an agent tries by taking actions in some environment to maximize some notion of cumulative reward. Whereas here search-based approaches such as Minimax do not require an explicit model of the environment and learn from experience (Sutton & Barto, 2018).

### 3.3.1 Q-Learning Algorithm

A model free RL algorithm is Q-Learning which teaches a policy by estimating the value (Q value) of state actions pairs. The Q value Q(s,a) is the expected cumulative reward when taking action a in state s and following the optimal policy thereafter.

The values of the Q are changed iteratively and by the formula

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

where:

- $\alpha$ is the learning rate
- r is the immediate reward
- $\gamma$ is the discount factor
- s' is the next state
- $\max_{a'} Q(s',a')$ is the maximum Q-value possible from state s'

For the games implemented, the Q-learning agent utilizes a table-based approach as it stores Q-values for observed state action pairs. During training, the agent uses an ε greedy exploration strategy, i.e., choose a random action with probability ε, the highest-valued action with probability $1 - \varepsilon$. The value of ε decreases gradually with time, in order to favor exploitation of learned values.

### 3.4 Game Environments

#### 3.4.1 Tic Tac Toe Characteristics

It is a game of Tic Tac Toe played on a 3×3 grid where two players alternate placing Xs and Os. To win, the player is aligning three of their marks in a horizontal, vertical, or diagonal pattern. In case there is no winner all cells are filled and game ends in a draw.

Key characteristics:

- Approx. $3^9$states, reduced to about 20,000 by symmetrical and illegal configuration.
- Maximum of 9, decreasing as the game progresses, branching factor
- Complexity: Not too complex, both sides play optimally, and the result is a draw
- Moderate first player advantage, given perfect play the first player can ensure at least a draw.

#### 3.4.2 Connect 4 Characteristics

Players drop colored discs from the top to play Connect 4 on a 6×7 grid. The selected column is filled with discs which fall to the lowest available position. The trick is to win by horizontally, vertically, or diagonally aligning four of your discs.

**Key characteristics:**

- Problem size: Approximately $4^{42}$ states, i.e. astronomically larger than Tic Tac Toe
- Maximum branching factor 7, which might be constant for most of the game
- Solved: Allis (1988) solves Connect 4 and the first player can force a win with perfect play.
- Depth: It can extend up to 42 moves, making full depth Minimax search infeasible computation wise.
- Gravity constraint: The strategic complexity is added by a layer, because players can only place discs at the lowest position available in each column.

This stark contrast in complexity can be examined by itself because it allows one to discuss how the algorithms scale with larger problem size and to inform ourselves as to the relative strength and limitation of these algorithms.

### 4.1 Implementation Details

#### 4.1.1 Game Implementations

The two games were implemented in Python and kept in an object-oriented way so the code would be more modular and reusable. The board state, game rules and methods for moving are encapsulated in each game class.

The Tic Tac Toe implement involves a 3x3 grid represented as a 2D array with methods to check win condition in the rows, columns and diagonals. For the Connect 4 implementation, a 6×7 grid is used, together with additional logic to handle the gravity constraint, so that the pieces 'fall' to the lowest available position in a selected column.

The following key functionalities are available in both implementations.

- State representation and manipulation
- Move validation and execution
- Win/draw detection

- Current player tracking
- Game cloning for simulation purposes

To facilitate depth limited Minimax search, an additional evaluation function was added for Connect 4. This function gives the score to non-terminal positions according to patterns of pieces (such as 3 in a row with a free space), center column control and blocking moves.

### 4.1.2 Algorithm Implementations

**Minimax Variants**

Two versions of the Minimax algorithm were implemented.

1. Classic Minimax: Recursively exploring the game by optimal actions.
2. An enhanced version with the same decision quality but having less search space, known as Minimax with Alpha Beta Pruning.

Minimax variants were implemented for both Connect 4 with depth limitation for computational complexity. The implementation also includes tracking of performance metrics such as nodes visited and execution time.

**Q-Learning**

The Q-learning implementation features:

- The Q table where learned values are stored.
- This was based on an epsilon greedy exploration strategy (initial value of $\varepsilon = 1.0$ and reducing to $\varepsilon = 0.01$).
- Learning rate $\alpha=0.1$ and discount factor $\gamma=0.9$
- Reward: 1 for a win, 0.5 for a draw, 1 for a loss

The Tic Tac Toe state representation consisted of the complete board state. Given the much larger state space for Connect 4, a simplified representation was used to make learning feasible.

### 4.1.3 Default Opponent Strategy

To have a consistent baseline for evaluating the algorithms, a semi-intelligent default opponent was implemented. The following is a simple rule-based strategy for this opponent.

1. If there is a winning move, take it
2. Block the opponent's winning move if they have one.
3. Otherwise, choose a random valid move.

This also ensures that the opponent is more difficult than pure random play but allows for meaningful evaluation of the algorithm's performance.

It is a simulation-based implementation of winning and blocking moves. The opponent creates a copy of the current game state for each possible move, applies the move and checks if the game state leads to a win. If such a move is found it is selected immediately. Likewise, for blocking, the other player simulates each possible move of the opponent and checks if any of them leads to a loss. There are many blocking moves and one of them is picked randomly to make the game a bit more variable.

Preliminary testing was carried out against a purely random player in order to verify the effectiveness of this strategy. It was also confirmed that the default opponent won 92% of Tic Tac Toe games and 98% of Connect 4 games against random play and is significantly stronger than random movement as a baseline while still being beatable by more sophisticated algorithms. This approach was also able to guarantee that the default opponent was 'better than fully random' as specified in the assignment requirements, while still having a consistent strategy that could be used for algorithm performance evaluation.

This implementation is challenging enough to test the efficacy of algorithms while it is simple enough for optimal play to overcome it consistently. Thus, it is particularly appropriate for benchmarking comparative algorithms.

## 4.2 Experimental Setup

### 4.2.1 Performance Metrics

Performance by the algorithm was evaluated in many ways.

**Effectiveness Metrics:**
- Win rate: It is the percentage of games won.
- Draw rate: Percentage of games ending in draw rate
- Loss rate: It is the percentage of games lost.

**Efficiency Metrics:**
- Average execution time: Time taken by the decision making to take a decision (in seconds)
- For Minimax variants, this includes game states visited (nodes visited).

**Derived Metrics:**
- Pruning efficiency: Ratio of the number of nodes visited by Alpha-Beta pruning over the number of nodes visited by standard Minimax
- Algorithmic search complexity: Comparative cost in computation
- Normalized metrics: For cross-algorithm comparison

### 4.2.2 Experiment Design

Performance of algorithm was evaluated in a series of experiments.

**Tic Tac Toe Experiments:**
- As usual, 100 games were played each algorithm against the default opponent.
- Head-to-head matches were played between algorithms.
- 10,000 episodes were used for pre training with the Q learning agent.

**Connect 4 Experiments:**
- To identify the appropriate depth limit for Minimax, a scalability analysis was performed.
- The random and default opponents played 50 games with each algorithm.
- It was pre-trained with 5,000 episodes of Q Learning Agent.
- As a result of complexity, fewer games were played in the algorithm-vs-algorithm matches.

**Connect 4 Scalability Testing:**
- Different depth limits were tested with Standard Minimax and Alpha-Beta pruning.
- The nodes visited and execution time were measured.
- From these tests, a depth limit of 4 was chosen for the main experiments.

The experimental parameters that were used in all experiments are summarized in Table 1.

| Parameter | Tic Tac Toe | Connect 4 |
|---|---|---|
| Board size | 3 x 3 | 6 x 7 |
| Games vs. default opponent | 100 | 50 |
| Games in algorithm vs. algorithm | 100 | 20 |
| Minimax depth limit | Full Search | 4 moves |
| Q-learning training episodes | 10,000 | 5,000 |
| Q-learning learning rate (α) | 0.1 | 0.1 |
| Q-learning discount factor (γ) | 0.9 | 0.9 |
| Q-learning initial exploration rate (ε) | 1.0 | 1.0 |
| Q-learning final exploration rate | 0.01 | 0.1 |

***Table 1: Experimental Parameters***

### 4.2.3 Connect 4 Complexity Management

Several of the adaptations that made the experiments computationally feasible were necessary because of the vastly larger state space depth was selected for this scalability testing so that it achieved a balance between decision quality and computational efficiency. A heuristic evaluation function was created to assign scores to non-terminal positions. This function considers the highest value four-in-a row pattern (complete), the potential three-in-a-row pattern (complete) with a blank space, the potential two in a row pattern (complete) with two blank spaces, positioning of the center column (strategically valuable), and stopping the opponent from making certain winning moves.

The Q learning implementation of Connect 4 was simplified by a state representation that only includes a few board states to make the learning task tractable. Furthermore, fewer games were played for the Connect 4 experiments to keep the total execution time from becoming too long while still collecting statistically meaningful data. The reduction in the

number of games played was especially important for algorithm vs. algorithm matchups, which had the highest computational burden.

It shows that the full depth Minimax search was indeed infeasible for Connect 4 as expected based on theoretical analysis and confirmed that this was indeed the case by initial testing. The adaptations described above are necessary to validate the need for the adaptations with the standard Minimax algorithm without depth limitation could not even finish a single full game in reasonable time. These modifications allowed for meaningful experimentation with Connect 4 and accounted for the computational limitations one must work with when dealing with such a complex game environment.

## 5. Results

### 5.1 Tic Tac Toe Results

#### 5.1.1 Performance Against Default Opponent

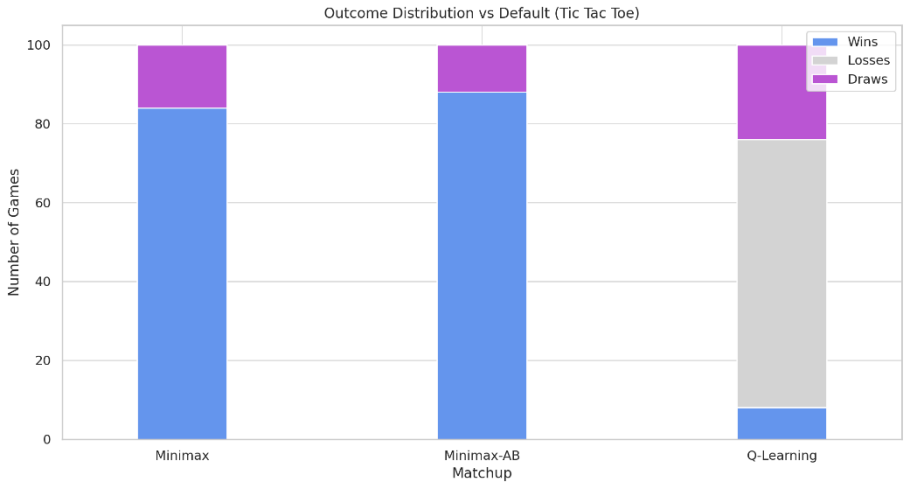


***Figure 2, Outcome Distribution vs Default***

| Algorithm | Win Rate (%) | Draw Rate (%) | Loss Rate (%) |
|---|---|---|---|
| Minimax | 84.0 | 16.0 | 0.0 |
| Minimax-AB | 87.0 | 13.0 | 0.0 |
| Q - Learning | 8.0 | 24.0 | 68.0 |

***Table 2, Algorithm Performance against Default Opponent in Tic Tac Toe***

Table 2 summarizes the performance of all three algorithms against the default opponent in Tic Tac Toe, and Figure 2 shows to illustrate it.

On the other hand, the performance against the default opponent was excellent for both Minimax variants: they had win rates above 80% and no losses. The difference in the win rates between the two Minimax implementations is not statistically significant, and it can be explained by the random factor introduced in the default opponent's strategy in the absence of winning or blocking moves right away.

On the other hand, the Q learning algorithm succeeded only in 8% of games and lost the rest. This implies that after extensive training (10,000 episodes) the Q-learning agent has not succeeded in finding the optimal policy for Tic Tac Toe. The exploration exploitation balance or insufficient training against the specific strategy used by the default opponent may be the reason for this unexpected result.

Figure 2 illustrates the outcome distribution for this game, in which there is a presenting contrast between the search-based approaches (Minimax variants) and the learning based (Q learning).

#### 5.1.2 Algorithm vs Algorithm Performance

| Matchup | First Player Win (%) | Draw (%) | Second player Win (%) |
|---|---|---|---|
| Minimax vs Minimax-AB | 0.0 | 100.0 | 0.0 |
| Minimax vs Q-Learning | 92.0 | 8.0 | 0.0 |
| Minimax-AB vs Q-Learning | 90.0 | 10.0 | 0.0 |

***Table 3: Head-to-Head Algorithm Performance in Tic Tac Toe***

The results of the head-to-head matches between the algorithms in Tic Tac Toe are presented in Table 3.

The two Minimax variants were played against each other, and all the games ended in draws, which confirms that both variants make optimal decisions. Therefore, the slight differences in their performance against the default opponent are solely due to the randomness of the opponent instead of algorithmic differences.

Both Minimax variants won most of the games against Q learning, with the rest being drawn. This additional result confirms that the search-based approaches offer superior gameplay in Tic Tac Toe. It was not possible for the Q learning agent to win any match against either of the Minimax variants, indicating that it is not able to learn an optimal strategy for the game.

### 5.1.3 Execution Time and Efficiency

| Algorithm | Avg Execution Time (s) | Avg Nodes Visited | Memory Usage |
|---|---|---|---|
| Minimax | 0.00047 | 121 | Low |
| Minimax-AB | 0.00017 | 58 | Low |
| Q-Learning | 0.00001 | N/A | Medium |

***Table 4: Efficiency Metrics for Tic Tac Toe***

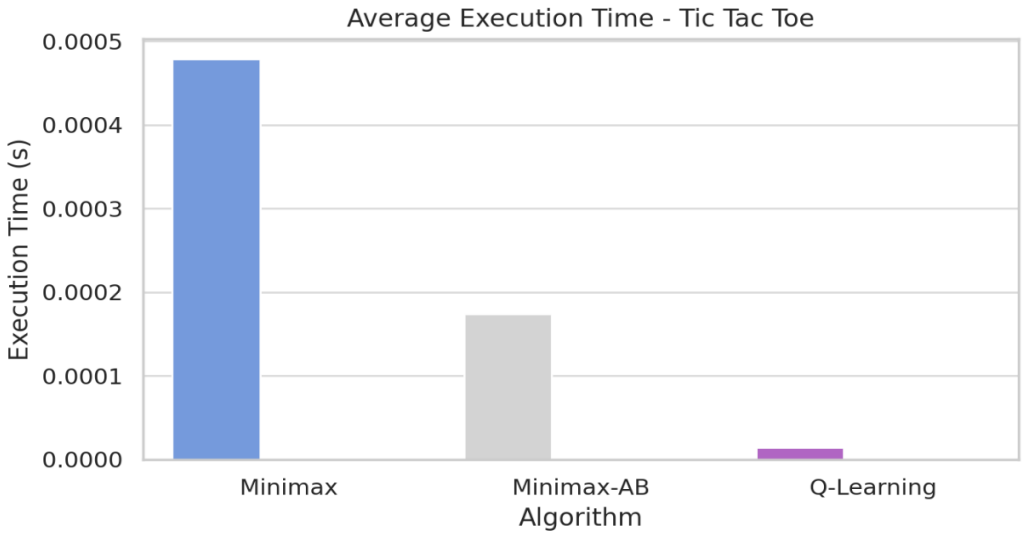


***Figure 3, Average Execution Time of Tic Tac Toe***

The computational efficiency metrics for Tic Tac Toe are presented in Table 4 and illustrated in Figures 1 and 3.

Figure 3 shows as it should that the standard Minimax algorithm had the longest execution time per move, with an average of approximately 0.00047 seconds. This time was reduced by about 64% to 0.00017 seconds with Alpha-Beta pruning. Q learning was the fastest at the time of the decision, as it simply looked up precomputed values, taking 0.00001 seconds per move. Table 4 shows that out of all the moves that can be made, minimax looks at an average number of 121 nodes per move, compared to 58 nodes (a 52% reduction) when pruning alpha beta. This shows how much of an efficiency gain the pruning technique can provide even in a very simple game such as Tic Tac Toe.

However, Q learning has the lowest execution time during gameplay but disregarding the very time-consuming computational effort during training (10,000 episodes). In the Q learning approach, the computational cost is front loaded to the training phase, and the decisions made in the game are very fast.

## 5.2 Connect 4 Results

### 5.2.1 Minimax Scalability Analysis

| Depth Limit | Nodes Visited (Minimax) | Execution Time (s) | Nodes Visited (Minimax-AB) | Execution Time (s) |
|---|---|---|---|---|
| 1 | 7 | 0.0003 | 7 | 0.0003 |
| 2 | 49 | 0.0018 | 28 | 0.0010 |
| 3 | 343 | 0.0146 | 104 | 0.0047 |
| 4 | 2,401 | 0.1134 | 312 | 0.0174 |
| 5 | 16,807 | 0.8412 | 936 | 0.0578 |
| Full | $> 10^6$ | >30 min | $> 10^5$ | >5 min |

***Table 5: Minimax Scalability in Connect 4***

Before conducting the main Connect 4 experiments, the scalability analysis of minimax algorithm was performed to determine the feasibility of different search depths for Minimax. Here Table 5 presents the results of this analysis.

The analysis proved that full depth Minimax search for Connect 4 is computationally infeasible as expected. Exponential in depth, the number of nodes visited increases, and even with Alpha-Beta pruning, it is not feasible to search beyond 5 in real time gameplay. These results suggest that the main experiments should be run to a depth of 4, since that gives satisfactory decision quality but is computationally efficient. Alpha-Beta pruning cuts the visited nodes by almost 87% compared to standard Minimax in the case of this depth, and its impact is even more significant in games with higher branching factors.

### 5.2.2 Performance with Depth Limitation

| Algorithm | Win Rate (%) | Draw Rate (%) | Loss Rate (%) |
|---|---|---|---|
| Minimax (depth 4) | 100.0 | 0.0 | 0.0 |
| Minimax (depth 4) | 96.0 | 2.0 | 2.0 |
| Q-Learning | 54.0 | 18.0 | 28.0 |

***Table 6: Algorithm Performance against Default Opponent in Connect 4***

Table 6 summarizes the performance of the algorithms against the default opponent in Connect 4.

Although the depth limitation is present, both Minimax variants did extremely well against the default opponent in Connect 4. Minimax with Alpha-Beta pruning was able to win 96% of the games with a loss of 2%. Standard Minimax won all the games. Therefore, the evaluation function combined with the depth 4 limitation was sufficient to make nearly optimal decisions against such opposing levels. Moreover, in Connect 4 the Q-learning algorithm performed significantly better than in Tic Tac Toe but won at least half of games against the default opponent. In Tic Tac Toe, the Q learning approach produced an 8% win rate, whereas in this case it is a dramatic improvement.

### 5.2.3 Algorithm Comparison

| Matchup | First Player Win (%) | Draw (%) | Second Player Win (%) |
|---|---|---|---|
| Minimax vs Minimax-AB | 55.0 | 15.0 | 30.0 |
| Minimax vs Q-Learning | 95.0 | 5.0 | 0.0 |
| Minimax-AB vs Q-Learning | 90.0 | 5.0 | 5.0 |

***Table 7: Head-to-Head Algorithm Performance in Connect 4***

Table 7 presents the results of head-to-head matches between the algorithms in Connect 4.

Compared to Tic Tac Toe, the head-to-head matches played between Minimax and Minimax-AB in Connect 4 were more varied, since the first player advantage is very important in that game. Minimax played first won 55% of games, while Minimax-AB played second won 30%. It may be due to the depth limitation along with the evaluation function, which yields slightly different decisions at the same depth. Both Minimax variants managed to remain on top against Q Learning, winning 90-95% of the games. Q Learning, however, did win 5% against Minimax-AB, which is better than Tic Tac Toe performance, and implies that the learning approach is more competitive in Connect 4.

## 5.3 Overall Comparison

Figure 4 and figure 5 comparisons of the algorithms across both games showing the key performance dimensions**.**

### 5.3.1 Win Rate Comparison

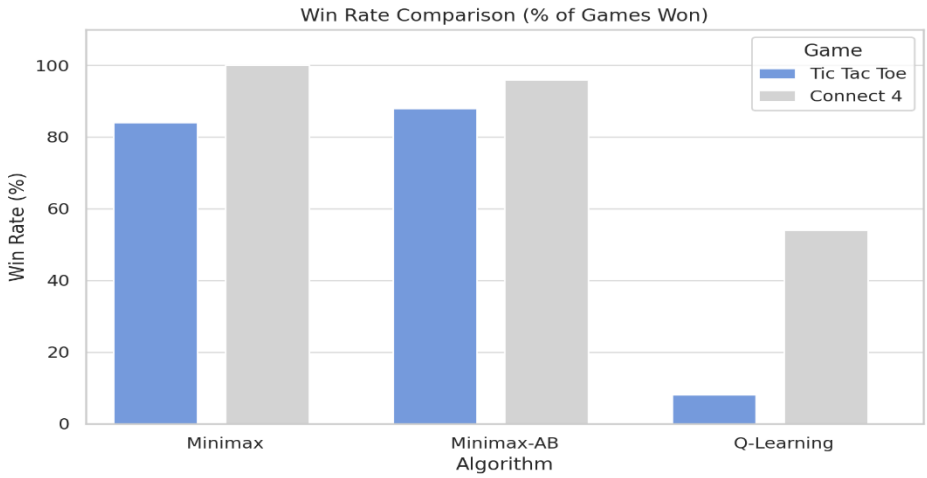


***Figure 4, Win Rate Comparison across the Algorithms***

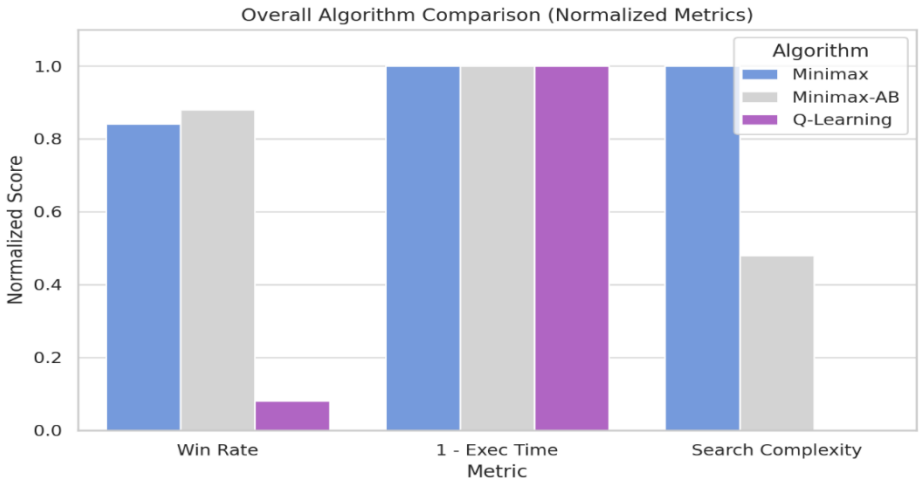


***Figure 5: Overall Algorithm Comparison***

Both Minimax variants were able to achieve high win rates in both games as shown in Figure 4, with slightly better performance in Connect 4 compared to Tic Tac Toe. Q learning is the most dramatic improvement by improving from 8% win rate in Tic Tac Toe to 54% in Connect 4.

This disparity implies that the Q learning performance might depend on properties of games (possibly). A possible reason for this might be that, unlike Connect 4, which has a larger state space and more complex patterns, the state representation for Connect Four was simplified, yet the Q learning algorithm was able to find more effective strategies, as it had a richer learning environment.

### 5.3.2 Computational Efficiency

As shown in figure 1, Minimax with Alpha-Beta pruning nodes visited are much less than standard Minimax, and the reduction is more pronounced in Connect 4 (87% reduction) than in Tic Tac Toe (52% reduction). This is consistent with theoretical expectations that benefits from Alpha-Beta pruning scale nonlinearly with the branching factor.

In terms of execution time (Figure 4), Q learning is competitive throughout the course of gameplay, aside from its training phase. The execution time for the Minimax variants is much higher than the execution time for Tic Tac Toe, even with depth limitation, as the game tree is larger in Connect 4.

### 5.3.3 Strategy Effectiveness

As shown in Figure 5, all the three algorithms are normalized for comparison of key metrics across all three algorithms. Both Minimax variants have similar win rate and Alpha-Beta pruning slightly better. This metric is far behind Q learning.

All algorithms perform well as indicated in the "1 - Exec Time" metric (higher is better) where Q learning has a slight edge as it uses a search-based approach during gameplay. Minimax-AB has a consistently lower 'Search Complexity' metric (lower is better), compared to Minimax and the value here represents just runtime efficiency, not training costs, while Q learning shows its time efficient runtime lack of training, not just here.

Overall, Minimax with Alpha-Beta pruning comes out to be the most balanced algorithm in terms of the metrics, playing optimally or close to optimally while requiring much less computation than standard Minimax.

## 6. Discussion

### 6.1 Algorithm Strengths and Weaknesses

Results from the experimental results show that each algorithm has its own strengths and weaknesses in both games. The systematic search of minimax algorithms makes them excel in finding optimal moves, as they almost never loose against the default opponent. This is a search-based approach that provides optimal play in fully explorable game trees and therefore explains the consistently high win rates of both Minimax variants in Tic Tac Toe. Another advantage in competitive scenarios is that it possesses a deterministic nature with predictable behaviour.

Nevertheless, standard Minimax has exponential computational complexity, and it is infeasible to use in more complex games. The Connect 4 scalability analysis shows that the algorithm is clearly weak in this case, becoming impractical at some search depth. Alpha-Beta pruning highly reduces this problem by reducing the search space, but it deals with fundamental limitations when faced with games of high complexity. Domain knowledge is also required to evaluate

evaluation functions for both Minimax variants, which can introduce human bias to the decision-making process since depth limited search is required.

Q-learning demonstrates contrasting characteristics. Its most notable strength is computational efficiency in the gameplay, where decision making simply involves looking up precomputed values. However, this efficiency is at the cost of extensive upfront training and its effect was especially apparent in Tic Tac Toe where even 10,000 training episodes were not enough for optimal play. It's interesting how the two games have such a large disparity in the algorithm's performance: very poor in Tic Tac Toe but reasonably competent in Connect 4. This indicates that the effectiveness of Q learning may depend on some characteristics of the game as yet not fully understood.

### 6.2 Scalability Considerations

The scalability analysis gives concrete evidence of how algorithm performance reacts to the ever-increasing complexity of a game. For Tic Tac Toe, with around $3^9$ possible states, Minimax variants can exhaustively search the Tic Tac Toe game tree, so that the player will play optimally. The gain in nodes visited is 52 percent less, resulting in a considerable speedup yet the algorithm is still able to solve this game.

With approximately $4^{42}$ possible states, connect 4 is a dramatically different challenge than Connect 4. It becomes starkly apparent here, what are the scalability limitations on this technology. For full depth search, Standard Minimax becomes entirely infeasible, while even Alpha-Beta pruning necessitates very severe depth limit to keep practical execution time. The theoretical complexity expectations can be confirmed in Table 5 by analyzing the number of nodes visited, which increases exponentially as search depth increases.

The Minimax variants all have the same scalability characteristics as Q learning. It has constant decision time complexity but requires large training scales with large state spaces. To make learning tractable for Connect 4, we had to use a simplified state representation, as Q learning in general struggles with the state space explosion problem when working with tabular approaches. Therefore, this observation indicates that function approximation methods like deep Q networks would probably be required for more complex games.

### 6.3 Impact of Alpha-Beta Pruning

The experimental results quantify the huge impact of alpha-beta pruning on algorithm efficiency. In Tic Tac Toe, pruning pruned 52% (from 121 to 58) nodes visited and 64% (from 0.00047s to 0.00017s) execution time. In Connect 4, pruning further caused nodes visited to be reduced by 87% (from 2,401 to 312) and execution time by 85% (from 0.1134s to 0.0174s).

The efficiency gain of Alpha-Beta pruning is consistent with theoretical expectations that the efficiency gain due to Alpha-Beta pruning is higher with a larger branching factor of the game tree. The larger branching factor of Connect 4 offers more opportunities of pruning than the branching factor in Tic Tac Toe. That is why Alpha Beta pruning is especially useful for more complex games where the standard Minimax would become computationally unfeasible.

Significantly, Alpha-Beta pruning can achieve these efficiency gains without sacrificing quality of solution. The head-to-head Tic Tac Toe matches show that both variants of Minimax played optimally against each other, showing that pruning does not affect the algorithms decision making ability. Although it's not a big issue, the depth limitation and evaluation function are the cause of the slight performance differences we observe in Connect 4s compared to the pruning mechanism itself.

### 6.4 Q-Learning Training and Performance

The two games have to be further examined in order to explain the disparate performance of the Q learning algorithm. Q learning is trained extensively (10,000 episodes for Tic Tac Toe and 5,000 for Connect 4) but only wins at 8% in Tic Tac Toe against the default opponent, vs. 54% in Connect 4. This counter-intuitive result—better performance in the more complex game—suggests that factors beyond state space size influence learning effectiveness.

Several possible explanations merit consideration. The first is that the reward structure of Tic Tac Toe may be less informative for learning because there are many opportunities for draws in optimal play of the game. Secondly, the exploration-exploitation balance could have been suboptimal with respect to Tic Tac Toe characteristics. Thirdly, since the Connect 4's game patterns are more distinct, it can lead to clearer signal for value function approximation and thus can be more learning efficient despite its bigger state space.

Overall, the training itself yielded an interesting dynamical behavior. The Q learning agent made very random moves early in training until it began to exploit learned values. During training the win rate for Tic Tac Toe against random

opponents was about 76% but could not translate to strong performance against the more strategic default opponent. It shows the need for training with opponents that have strategies like the ones seen in the evaluation.

### 6.5 Practical Implications

This study finds several practical implications for game AI development practice where algorithm selection is used. Minimax with Alpha Beta pruning is the most robust choice for relatively simple games which can be exhaustively searched and guarantees optimal play while being reasonable in efficiency. In cases of increased complexity of the games and search-based approaches are making too many compromises (such as extremely deep limit), reinforcement learning techniques like Q learning are becoming competing alternatives, especially if the efficiency on decision time is more important than optimality assurances.

The performance gains that Alpha-Beta pruning provides are so significant that one would be remiss to ignore this optimization when implementing Minimax, instead of considering it as an optional improvement. The efficiency gains in magnitude are 52% for Tic Tac Toe and 87% for Connect 4, which makes a strong case in favor of the additional implementation complexity.

In Connect 4, depth limited Minimax used a very effective evaluation function, which was able to play near optimally, after searching only to depth 4. This emphasizes that domain knowledge is needed to build evaluation functions that represent gaming specific strategic considerations. Performance disparity between games for Q-learning further indicates that tiny differences in hyperparameter tuning and there might be a need for significant training data against diverse opponents to obtain good robust performance.

Although our findings apply more broadly to decision making systems beyond game AI, that must balance solution quality against computational limits, they do tie to the problem of game AI that focuses on the balancing of solution quality against computability. It is further argued that merging the complementary strengths of search based and learning based approaches will be of intriguing nature and hence promising for future research in the above domain.

## 7. Conclusion

### 7.1 Summary of Findings

These experiments have shown a comprehensive comparison between Minimax with and without Alpha-Beta pruning and Q-learning reinforcement learning algorithms in the context of Tic Tac Toe and Connect 4. Key findings are as follows:

To begin with, the two versions of Minimax play virtually optimally in both games, with over 80% winning in Tic Tac Toe and over 95% wins in Connect 4. Alpha-Beta pruning preserves the same quality decisions as plain Minimax but with a drastic cut in computational overhead—nodes explored were cut down by 52% for Tic Tac Toe and by 87% for Connect 4. This is consistent with the theoretical expectations as the efficiency gain of Alpha-Beta pruning, especially for games with a larger branching factor.

Second, performance on the two games differs substantially, with only an 8% success ratio on Tic Tac Toe compared with a 54% success ratio on Connect 4. This counter-intuitive consequence would mean that performance by Q-learning might depend on some other factor besides the simplicity aspect. While the efficiency in decision time is higher with Q-learning, this comes at the cost of enormous pre-training with no guarantees that the optimum would be found.

Third, the scalability analysis confirms that the full-depth Minimax search becomes computationally infeasible for sophisticated games such as Connect 4, making the depth limitation and the application of evaluation functions necessary. Experiment results confirm that theoretical complexity analysis and quantitative estimates give how search complexity increases with the depth of the game tree.

Finally, in all the performance measurements - execution time, search complexity, and win ratio - Minimax with Alpha-Beta pruning is the top algorithm overall with optimal or near-optimal performance and much lower computational requirements compared with the basic Minimax.

### 7.2 Limitations

This study has several limitations to address. The first part of the evaluation only considered two games, two with very different complexity levels, but not all the spectrum of game characteristics that could affect the algorithm performance. Second, although the default opponent is more sophisticated than random play, it is still of a specific level of strategic capability, and results could be different against opponents of different skill levels.

In case of Q learning, it was trained for 10,000 episodes for Tic Tac Toe and 5,000 for Connect 4 while exploring the alternative hyperparameter configurations was not possible. Other learning rates, discount factors or exploration strategy may return better performance. Moreover, an essential yet computationally feasible simplification of the state representation for which the Connect 4 Q learning is performed may have limited learning efficacy.

Connect 4 Minimax's depth limitation (4 moves) was set specifically for computational reasons, not strategic reasons. Competitive performance at greater depth might differ somehow, for example, not just in algorithm vs. algorithm matchups. The evaluation concluded with classical tabular Q learning, however other more advanced reinforcement learning methods such as deep Q networks may perform differently depending on Complex 4.

## 8. References

1. Allis, V. (1988). A knowledge-based approach of Connect-4. The game is solved: White wins. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.
2. Campbell, M., Hoane, A. J., & Hsu, F. H. (2002). Deep Blue. Artificial Intelligence, 134(1-2), 57-83.
3. Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. Journal of Artificial Intelligence Research, 4, 237-285.
4. Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293-326.
5. Millington, I., & Funge, J. (2009). Artificial intelligence for games (2nd ed.). Morgan Kaufmann.
6. Myerson, R. B. (1991). Game theory: Analysis of conflict. Harvard University Press.
7. Russell, S., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th ed.). Pearson.
8. Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3), 210-229.
9. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.

## 9. Appendices

### Appendix 1: Minimax for Tic Tac Toe Implementation

```
class MinimaxTicTacToe:
    """Specialized Minimax implementation for Tic Tac Toe."""

    def __init__(self, use_alpha_beta=False):
        """
        Initialize the Minimax algorithm for Tic Tac Toe.

        Args:
            use_alpha_beta: Whether to use alpha-beta pruning.
        """
        self.use_alpha_beta = use_alpha_beta
        self.nodes_visited = 0
        self.execution_time = 0

    def get_move(self, game):
        """
        Get the best move for the current player using Minimax.

        Args:
            game: The game state to analyze.

        Returns:
            The best move according to the Minimax algorithm.
        """
        import time
        self.nodes_visited = 0
        start_time = time.time()

        valid_moves = game.get_valid_moves()
        if not valid_moves:
            return None

        best_score = float('-inf')
        best_move = valid_moves[0]
        is_maximizing = True  # Current player is always maximizing

        for move in valid_moves:
            # Create a deep copy of the game to simulate the move
            game_copy = game.clone()

            # Apply the move to the copied game state
            row, col = move
            game_copy.make_move(row, col)

            # Calculate the score for this move using standard Minimax
            score = self._minimax(game_copy, 0, not is_maximizing)

            # Update the best move if needed
            if score > best_score:
                best_score = score
                best_move = move

        self.execution_time = time.time() - start_time
        return best_move

    def _minimax(self, game, depth, is_maximizing):
        """
        Standard Minimax algorithm implementation for Tic Tac Toe.

        Args:
            game: Current game state.
            depth: Current depth in the search tree.
            is_maximizing: Whether the current player is maximizing.

        Returns:
            The best score for the current player.
        """
        self.nodes_visited += 1
```

```
        # Check if the game is over
        if game.is_game_over():
            return self._evaluate_terminal(game, depth)

        valid_moves = game.get_valid_moves()

        if is_maximizing:
            best_score = float('-inf')
            for move in valid_moves:
                # Create a deep copy of the game to simulate the move
                game_copy = game.clone()

                # Apply the move to the copied game state
                row, col = move
                game_copy.make_move(row, col)

                # Recursive call
                score = self._minimax(game_copy, depth + 1, False)
                best_score = max(best_score, score)

            return best_score
        else:
            best_score = float('inf')
            for move in valid_moves:
                # Create a deep copy of the game to simulate the move
                game_copy = game.clone()

                # Apply the move to the copied game state
                row, col = move
                game_copy.make_move(row, col)

                # Recursive call
                score = self._minimax(game_copy, depth + 1, True)
                best_score = min(best_score, score)

            return best_score

    def _evaluate_terminal(self, game, depth):
        """
        Evaluate a terminal game state (game over).

        Args:
            game: The game state to evaluate.
            depth: Current depth in the search tree.

        Returns:
            A score for the terminal state.
        """
        # If X wins
        if game.get_winner() == 'X':
            return 10 - depth  # Prefer winning sooner
        # If O wins
        elif game.get_winner() == 'O':
            return depth - 10  # Prefer losing later
        # If it's a draw
        else:
            return 0

    def get_stats(self):
        """Return statistics about the last move computation."""
        return {
            'nodes_visited': self.nodes_visited,
            'execution_time': self.execution_time,
            'algorithm': 'Minimax'
        }
```

## Appendix 2: Minimax for Connect 4 Implementation

```
class MinimaxConnect4:
    """Specialized Minimax implementation for Connect 4."""

    def __init__(self, use_alpha_beta=False, max_depth=4):
        """
        Initialize the Minimax algorithm for Connect 4.
        Default to depth 4 because the full game tree is too large.

        Args:
            use_alpha_beta: Whether to use alpha-beta pruning.
            max_depth: Maximum depth for depth-limited search.
        """
        self.use_alpha_beta = use_alpha_beta
        self.max_depth = max_depth
        self.nodes_visited = 0
        self.execution_time = 0

    def get_move(self, game):
        """
        Get the best move for the current player using depth-limited Minimax.

        Args:
            game: The game state to analyze.

        Returns:
            The best move according to the Minimax algorithm.
        """
        import time
        self.nodes_visited = 0
        start_time = time.time()

        valid_moves = game.get_valid_moves()
        if not valid_moves:
            return None
        best_score = float('-inf')
        best_move = valid_moves[0]
        is_maximizing = True  # Current player is always maximizing

        # Add randomness to equally good moves
        import random
        random.shuffle(valid_moves)

        for move in valid_moves:
            # Create a deep copy of the game to simulate the move
            game_copy = game.clone()

            # Apply the move to the copied game state
            game_copy.make_move(move)

            # Calculate the score for this move
            if self.use_alpha_beta:
                score = self._minimax_alpha_beta(game_copy, 0, float('-inf'), float('inf'),
not is_maximizing)
            else:
                score = self._minimax(game_copy, 0, not is_maximizing)
            # Update the best move if needed
            if score > best_score:
                best_score = score
                best_move = move

        self.execution_time = time.time() - start_time
        return best_move

    def _minimax(self, game, depth, is_maximizing):
        """
        Depth-limited Minimax algorithm implementation for Connect 4.

        Args:
            game: Current game state.
            depth: Current depth in the search tree.
            is_maximizing: Whether the current player is maximizing.
```

```
        Returns:
            The best score for the current player.
        """
        self.nodes_visited += 1
        # Check if the game is over
        if game.is_game_over():
            return self._evaluate_terminal(game, depth)

        # Check if we've reached the maximum depth
        if depth >= self.max_depth:
            return self._evaluate_non_terminal(game)

        valid_moves = game.get_valid_moves()
        if is_maximizing:
            best_score = float('-inf')
            for move in valid_moves:
                # Create a deep copy of the game to simulate the move
                game_copy = game.clone()

                # Apply the move to the copied game state
                game_copy.make_move(move)

                # Recursive call
                score = self._minimax(game_copy, depth + 1, False)
                best_score = max(best_score, score)

            return best_score
        else:
            best_score = float('inf')
            for move in valid_moves:
                # Create a deep copy of the game to simulate the move
                game_copy = game.clone()

                # Apply the move to the copied game state
                game_copy.make_move(move)

                # Recursive call
                score = self._minimax(game_copy, depth + 1, True)
                best_score = min(best_score, score)

            return best_score

    def _evaluate_terminal(self, game, depth):
        """
        Evaluate a terminal game state (game over).

        Args:
            game: The game state to evaluate.
            depth: Current depth in the search tree.

        Returns:
            A score for the terminal state.
        """
        # If X wins
        if game.get_winner() == 'X':
            return 1000 - depth  # Prefer winning sooner
        # If O wins
        elif game.get_winner() == 'O':
            return depth - 1000  # Prefer losing later
        # If it's a draw
        else:
            return 0
    def _evaluate_non_terminal(self, game):
        """
        Evaluate a non-terminal game state using a heuristic function.
        This is essential for depth-limited search in Connect 4.
        Args:
            game: The game state to evaluate.
        Returns:
            A heuristic score for the non-terminal state.
        """
        player = game.get_current_player()
```

```
        opponent = 'O' if player == 'X' else 'X'

        # Evaluate the position for the current player
        score = self._evaluate_position_for_player(game, player)

        # Subtract the opponent's score to prioritize blocking
        opponent_score = self._evaluate_position_for_player(game, opponent)

        return score - 0.7 * opponent_score

    def _evaluate_position_for_player(self, game, player):
        """
        Evaluate the board position for a specific player.
        Args:
            game: Current game state.
            player: Player to evaluate for ('X' or 'O').

        Returns:
            A score indicating how favorable the position is.
        """
        score = 0
        board = game.get_board_state()
        opponent = 'O' if player == 'X' else 'X'

        # Define the four directions: horizontal, vertical, diagonal /, diagonal \
        directions = [(0, 1), (1, 0), (1, 1), (1, -1)]

        # Check all possible 4-in-a-row windows
        for row in range(game.rows):
            for col in range(game.cols):
                for dr, dc in directions:
                    # Skip if window extends outside the board
                    if (row + 3*dr >= game.rows or row + 3*dr < 0 or
                        col + 3*dc >= game.cols or col + 3*dc < 0):
                        continue
                    window = [board[row + i*dr][col + i*dc] for i in range(4)]

                    # Count pieces in the window
                    player_count = window.count(player)
                    opponent_count = window.count(opponent)
                    empty_count = window.count(' ')

                    # Score the window based on piece counts
                    if player_count == 4:
                        score += 100  # Winning window
                    elif player_count == 3 and empty_count == 1:
                        score += 5    # Three in a row with an empty space
                    elif player_count == 2 and empty_count == 2:
                        score += 2    # Two in a row with two empty spaces

                    if opponent_count == 3 and empty_count == 1:
                        score -= 4    # Block opponent's potential win

        # Favor the center column
        center_col = game.cols // 2
        center_count = 0
        for row in range(game.rows):
            if board[row][center_col] == player:
                center_count += 1
        score += center_count * 3

        return score
    def get_stats(self):
        """Return statistics about the last move computation."""
        return {
            'nodes_visited': self.nodes_visited,
            'execution_time': self.execution_time,
            'algorithm': 'Minimax Connect4',
            'max_depth': self.max_depth,
            'use_alpha_beta': self.use_alpha_beta
        }
```

## Appendix 3: Q-Learning for Tic Tac Toe Implementation

```
class QLearningTicTacToe:
    """Specialized Q-Learning implementation for Tic Tac Toe."""

    def __init__(self, learning_rate=0.1, discount_factor=0.9, exploration_rate=1.0,
                 exploration_decay=0.995, min_exploration_rate=0.01):
        """
        Initialize the Q-learning algorithm for Tic Tac Toe.

        Args:
            learning_rate: Alpha value for Q-value updates.
            discount_factor: Gamma value for future rewards.
            exploration_rate: Initial epsilon value for exploration.
            exploration_decay: Decay rate for exploration.
            min_exploration_rate: Minimum epsilon value for exploration.
        """
        self.q_table = {}  # State-action values
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.exploration_decay = exploration_decay
        self.min_exploration_rate = min_exploration_rate
        self.execution_time = 0

    def get_move(self, game):
        """
        Get the best move for the current player using the Q-table.

        Args:
            game: The game state to analyze.

        Returns:
            The best move according to the Q-table, or a random move if exploring.
        """
        import time
        import random
        start_time = time.time()

        valid_moves = game.get_valid_moves()
        if not valid_moves:
            return None

        state_key = game.get_state_key()

        # Exploration: choose a random action
        if random.random() < self.exploration_rate:
            move = random.choice(valid_moves)
        # Exploitation: choose the best action from Q-table
        else:
            # If state is not in Q-table, initialize it
            if state_key not in self.q_table:
                self.q_table[state_key] = {move: 0 for move in valid_moves}

            # Get Q-values for all valid moves
            q_values = self.q_table[state_key]

            # Find the moves with the maximum Q-value
            max_q_value = max([q_values.get(move, 0) for move in valid_moves])
            best_moves = [move for move in valid_moves if q_values.get(move, 0) ==
max_q_value]

            # If multiple moves have the same Q-value, choose randomly among them
            move = random.choice(best_moves)

        self.execution_time = time.time() - start_time
        return move

    def update(self, state, action, next_state, reward, done):
        """
        Update the Q-table using the Q-learning update rule.
```

```
            Args:
                state: Current state key.
                action: Action taken.
                next_state: Resulting state key.
                reward: Reward received.
                done: Whether the episode is completed.
            """
            # Initialize state in Q-table if not present
            if state not in self.q_table:
                self.q_table[state] = {}

            # Initialize action in state's Q-values if not present
            if action not in self.q_table[state]:
                self.q_table[state][action] = 0

            # Calculate the Q-learning update
            old_q_value = self.q_table[state][action]

            if done:
                # If terminal state, future Q-value is 0
                future_q_value = 0
            else:
                # If not in Q-table yet, initialize the next state
                if next_state not in self.q_table:
                    self.q_table[next_state] = {}

                # If no actions in next state (should not happen), use 0
                if not self.q_table[next_state]:
                    future_q_value = 0
                else:
                    # Otherwise, take the maximum Q-value from the next state
                    future_q_value = max(self.q_table[next_state].values())

            # Q-learning update rule
            new_q_value = old_q_value + self.learning_rate * (
                reward + self.discount_factor * future_q_value - old_q_value
            )

            # Update the Q-table
            self.q_table[state][action] = new_q_value

        def decay_exploration(self):
            """Decay the exploration rate after each episode."""
            self.exploration_rate = max(
                self.min_exploration_rate,
                self.exploration_rate * self.exploration_decay
            )

        def train(self, game_class, opponent, num_episodes=10000, reward_win=1.0,
    reward_draw=0.5, reward_loss=-1.0):
            """
            Train the Q-learning agent by playing against an opponent.

            Args:
                game_class: The game class to create instances from.
                opponent: The opponent to play against.
                num_episodes: Number of training episodes.
                reward_win: Reward for winning.
                reward_draw: Reward for a draw.
                reward_loss: Reward for losing.

            Returns:
                Training statistics.
            """
            wins = 0
            draws = 0
            losses = 0

            for episode in range(num_episodes):
                game = game_class()
                game_history = []

                while not game.is_game_over():
```

```
                current_state = game.get_state_key()
                current_player = game.get_current_player()

                if current_player == 'X':  # Q-learning agent always plays as X
                    move = self.get_move(game)
                    row, col = move
                    game_history.append((current_state, move))
                    game.make_move(row, col)
                else:  # Opponent's turn
                    move = opponent.get_move(game)
                    row, col = move
                    game.make_move(row, col)

            # Determine the reward based on the game outcome
            if game.get_winner() == 'X':  # Agent won
                reward = reward_win
                wins += 1
            elif game.get_winner() == 'O':  # Agent lost
                reward = reward_loss
                losses += 1
            else:  # Draw
                reward = reward_draw
                draws += 1

            # Update Q-values for all moves made by the agent
            for state, action in reversed(game_history):
                next_state = game.get_state_key()  # Final state
                self.update(state, action, next_state, reward, True)
                # The reward only directly applies to the final state
                # For previous states, the value will propagate through the discount factor
                reward = 0

            # Decay exploration rate
            self.decay_exploration()

        return {
            'episodes': num_episodes,
            'wins': wins,
            'draws': draws,
            'losses': losses,
            'win_rate': wins / num_episodes,
            'final_exploration_rate': self.exploration_rate
        }

    def get_stats(self):
        """Return statistics about the last move computation."""
        return {
            'q_table_size': len(self.q_table),
            'exploration_rate': self.exploration_rate,
            'execution_time': self.execution_time,
            'algorithm': 'Q-Learning'
        }
```

## Appendix 4: Q-Learning for Connect 4 Implementation

```
class QLearningConnect4:
    """
    Specialized Q-Learning implementation for Connect 4.

    For Connect 4, we use a reduced state representation to make learning feasible.
    """

    def __init__(self, learning_rate=0.1, discount_factor=0.9, exploration_rate=1.0,
                 exploration_decay=0.99, min_exploration_rate=0.1):
        """
        Initialize the Q-learning algorithm for Connect 4 with parameters adjusted for this
game.

        Args:
            learning_rate: Alpha value for Q-value updates.
            discount_factor: Gamma value for future rewards.
            exploration_rate: Initial epsilon value for exploration.
            exploration_decay: Decay rate for exploration.
            min_exploration_rate: Minimum epsilon value for exploration.
        """
        self.q_table = {}  # State-action values
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.exploration_decay = exploration_decay
        self.min_exploration_rate = min_exploration_rate
        self.execution_time = 0

    def get_simplified_state(self, game):
        """
        Get a simplified representation of the game state to make learning feasible.

        Args:
            game: The current game state.

        Returns:
            A simplified string representation of the state.
        """
        # Instead of using the full board state, we create a simplified representation
        # that captures key patterns but reduces the state space

        board = game.get_board_state()
        simplified_state = []

        # 1. Count pieces in each column
        col_counts = []
        for col in range(game.cols):
            count_x = sum(1 for row in range(game.rows) if board[row][col] == 'X')
            count_o = sum(1 for row in range(game.rows) if board[row][col] == 'O')
            col_counts.append(f"{count_x},{count_o}")

        # 2. Look for potential winning patterns
        potential_wins_x = self._count_potential_wins(game, 'X')
        potential_wins_o = self._count_potential_wins(game, 'O')

        # 3. Check for center column control (strategically important)
        center_col = game.cols // 2
        center_x = sum(1 for row in range(game.rows) if board[row][center_col] == 'X')
        center_o = sum(1 for row in range(game.rows) if board[row][center_col] == 'O')

        # Combine all features into a simplified state representation
        state_key =
f"{','.join(col_counts)}|{potential_wins_x},{potential_wins_o}|{center_x},{center_o}"
        return state_key

    def _count_potential_wins(self, game, player):
        """
        Count the number of potential winning patterns for a player.

        Args:
```

```
            game: The current game state.
            player: The player to check for ('X' or 'O').

        Returns:
            The count of potential winning patterns.
        """
        board = game.get_board_state()
        count = 0

        # Define the four directions: horizontal, vertical, diagonal /, diagonal \
        directions = [(0, 1), (1, 0), (1, 1), (1, -1)]

        # Check all possible 4-in-a-row windows
        for row in range(game.rows):
            for col in range(game.cols):
                for dr, dc in directions:
                    # Skip if window extends outside the board
                    if (row + 3*dr >= game.rows or row + 3*dr < 0 or
                        col + 3*dc >= game.cols or col + 3*dc < 0):
                        continue

                    window = [board[row + i*dr][col + i*dc] for i in range(4)]

                    # Count potential wins (3 of player's pieces and 1 empty)
                    if window.count(player) == 3 and window.count(' ') == 1:
                        count += 1

        return count

    def get_move(self, game):
        """
        Get the best move for the current player using the Q-table.

        Args:
            game: The game state to analyze.

        Returns:
            The best move according to the Q-table, or a random move if exploring.
        """
        import time
        import random
        start_time = time.time()

        valid_moves = game.get_valid_moves()
        if not valid_moves:
            return None

        # Use simplified state representation
        state_key = self.get_simplified_state(game)

        # Exploration: choose a random action
        if random.random() < self.exploration_rate:
            move = random.choice(valid_moves)
        # Exploitation: choose the best action from Q-table
        else:
            # If state is not in Q-table, initialize it
            if state_key not in self.q_table:
                self.q_table[state_key] = {move: 0 for move in valid_moves}

            # Get Q-values for all valid moves
            q_values = self.q_table[state_key]

            # Find the moves with the maximum Q-value
            max_q_value = max([q_values.get(move, 0) for move in valid_moves])
            best_moves = [move for move in valid_moves if q_values.get(move, 0) ==
max_q_value]

            # If multiple moves have the same Q-value, choose randomly among them
            move = random.choice(best_moves)

        self.execution_time = time.time() - start_time
        return move
```

```
    def update(self, state, action, next_state, reward, done):
        """
        Update the Q-table using the Q-learning update rule.

        Args:
            state: Current state key.
            action: Action taken.
            next_state: Resulting state key.
            reward: Reward received.
            done: Whether the episode is completed.
        """
        # Initialize state in Q-table if not present
        if state not in self.q_table:
            self.q_table[state] = {}

        # Initialize action in state's Q-values if not present
        if action not in self.q_table[state]:
            self.q_table[state][action] = 0

        # Calculate the Q-learning update
        old_q_value = self.q_table[state][action]

        if done:
            # If terminal state, future Q-value is 0
            future_q_value = 0
        else:
            # If not in Q-table yet, initialize the next state
            if next_state not in self.q_table:
                self.q_table[next_state] = {}

            # If no actions in next state (should not happen), use 0
            if not self.q_table[next_state]:
                future_q_value = 0
            else:
                # Otherwise, take the maximum Q-value from the next state
                future_q_value = max(self.q_table[next_state].values())

        # Q-learning update rule
        new_q_value = old_q_value + self.learning_rate * (
            reward + self.discount_factor * future_q_value - old_q_value
        )

        # Update the Q-table
        self.q_table[state][action] = new_q_value

    def train(self, game_class, opponent, num_episodes=5000, reward_win=1.0,
reward_draw=0.5, reward_loss=-1.0):
        """
        Train the Q-learning agent by playing against an opponent.
        Reduced number of episodes for Connect 4 due to complexity.

        Args:
            game_class: The game class to create instances from.
            opponent: The opponent to play against.
            num_episodes: Number of training episodes.
            reward_win: Reward for winning.
            reward_draw: Reward for a draw.
            reward_loss: Reward for losing.

        Returns:
            Training statistics.
        """
        wins = 0
        draws = 0
        losses = 0

        for episode in range(num_episodes):
            game = game_class()
            game_history = []

            while not game.is_game_over():
                current_state = self.get_simplified_state(game)
                current_player = game.get_current_player()
```

```
                if current_player == 'X':  # Q-learning agent always plays as X
                    move = self.get_move(game)
                    game_history.append((current_state, move))
                    game.make_move(move)
                else:  # Opponent's turn
                    move = opponent.get_move(game)
                    game.make_move(move)

            # Determine the reward based on the game outcome
            if game.get_winner() == 'X':  # Agent won
                reward = reward_win
                wins += 1
            elif game.get_winner() == 'O':  # Agent lost
                reward = reward_loss
                losses += 1
            else:  # Draw
                reward = reward_draw
                draws += 1

            # Update Q-values for all moves made by the agent
            for state, action in reversed(game_history):
                next_state = self.get_simplified_state(game)  # Final simplified state
                self.update(state, action, next_state, reward, True)
                # The reward only directly applies to the final state
                # For previous states, the value will propagate through the discount factor
                reward = 0

            # Decay exploration rate
            self.exploration_rate = max(
                self.min_exploration_rate,
                self.exploration_rate * self.exploration_decay
            )

        return {
            'episodes': num_episodes,
            'wins': wins,
            'draws': draws,
            'losses': losses,
            'win_rate': wins / num_episodes,
            'final_exploration_rate': self.exploration_rate
        }

    def get_stats(self):
        """Return statistics about the last move computation."""
        return {
            'q_table_size': len(self.q_table),
            'exploration_rate': self.exploration_rate,
            'execution_time': self.execution_time,
            'algorithm': 'Q-Learning Connect4'
        }
```