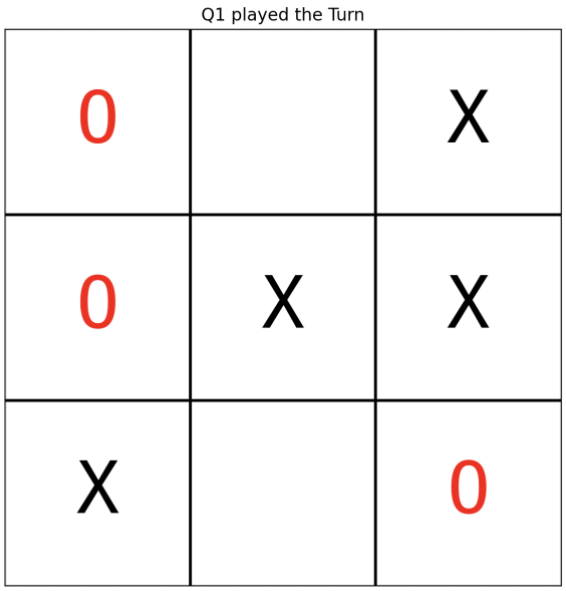
CS7IS2: Artificial Intelligence Assignment 3

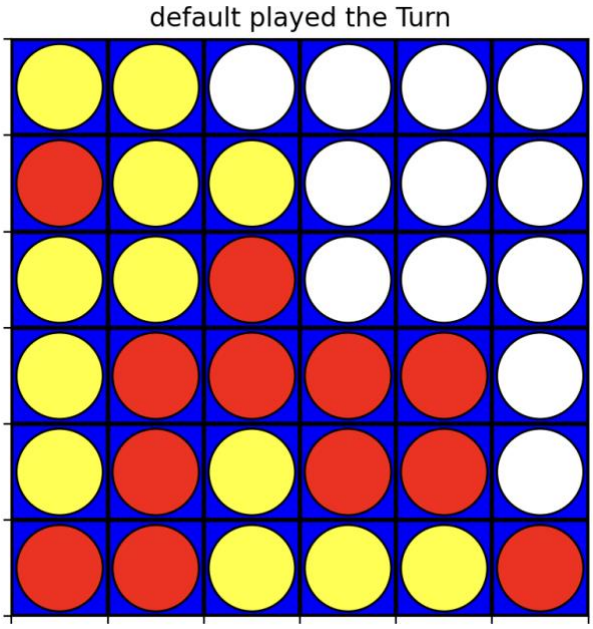
Nachiketh J (23337215) **30/03/2024**

Introduction:
This report details the implementation of the classic games Tic Tac Toe and Connect 4, and their respective agents: a human player, a random player, a minimax-based AI player, and a Q-learning-based AI player. The report discusses design choices regarding state space, rewards, and algorithmic strategies.

Implementation Report on Tic Tac Toe and Connect 4 Games
Tic Tac Toe and Connect 4 are well-known strategic games that serve as effective domains for studying artificial intelligence and reinforcement learning due to their simple rules and finite outcomes. Both games were implemented in Python, utilizing object-oriented principles to facilitate ease of expansion and modification.



TicTacToe



Connect4

Design and Implementation:

Board Class:
The foundational Board class serves as the cornerstone for both the Tic Tac Toe and Connect 4 implementations. This class encapsulates core functionalities such as maintaining the game state, tracking player turns, and determining valid moves. By abstracting these shared aspects, we ensure code reusability and maintainability across different game implementations.

Game-Specific Classes:
Building upon the Board class, the TicTacToe and Connect4 classes introduce game-specific rules and logic. The TicTacToe class operates on a **3x3 grid** and defines the winning condition as achieving three consecutive marks horizontally, vertically, or diagonally. In contrast, the Connect4 class utilizes a **6x6 grid**, with the victory condition set to four consecutive marks in any direction. These classes manage game-specific actions, validate moves, and check for game-ending conditions, encapsulating the unique aspects of each game.

Agent Classes:
A diverse set of agent classes were implemented to interact with the games, representing different strategies and decision-making processes:

- **HumanPlayer**: Facilitates game play by allowing users to input moves via the console, promoting interactive play and testing against human intuition.
- **RandomPlayer**: A simple agent that selects moves at random, providing a baseline opponent that, while unpredictable, does not employ strategic thinking.
- **MinimaxPlayer**: Implements the Minimax decision rule, optionally enhanced with alpha-beta pruning, to make optimal moves by exploring future possible game states. The algorithm's depth and pruning parameters are carefully chosen to balance between strategic depth and computational feasibility.
- **QLearningPlayer**: Utilizes the Q-learning algorithm, a form of reinforcement learning, to improve its strategy over time based on outcomes of past games. This agent learns an action-value function, guiding its move choices to maximize long-term rewards.

Game Flow Control:
The game flow is orchestrated by the TicTacToe and ConnectFourBoard classes, which manage the turn-taking mechanism, enforce game rules, and determine the game's end. The implementation includes a train method for the QLearningPlayer, allowing it to learn and refine its strategy through repeated play against different opponents.

Reward Structure:
The reward system is meticulously designed to foster winning strategies and adaptive learning, assigning distinct values to different game outcomes. This system plays a crucial role in the QLearningPlayer's learning process by directly influencing the development of its strategic decisions. Rewards are allocated as follows to encourage performance optimization and strategic refinement:

- Win: A reward of **1** is given for winning, incentivizing the AI to prioritize moves and strategies that lead to victories, as this outcome is most beneficial.
- Loss: A reward of **0** is assigned to losses, indicating a neutral stance on losing outcomes. This encourages the AI to avoid losing strategies without overly penalizing exploration, maintaining a balance between risk-taking and strategic safety.
- Tie: A reward of **0.2** is provided for ties, recognizing the value of drawing over losing. This moderate positive reinforcement for ties encourages the AI to seek at least a draw when a win might not be achievable, subtly promoting resilience and adaptability in gameplay.

State Space Representation:
In QLearningPlayer, game states are represented as hashable strings. This approach simplifies the complex game state space into a manageable form, enhancing the learning process through:

- Efficiency: Quick storage and retrieval of state-action values in the Q-table.
- Simplified Learning: Makes the vast state space of board games like Tic Tac Toe and Connect 4 easily manageable for the AI.
- Adaptability: Helps the AI recognize patterns and apply strategies to similar states, improving decision-making.

This method strikes a balance between simplifying the game's complexity and maintaining strategic depth, aiding the AI's ability to learn and adapt effectively.

Justification for Design Choices with Parameters:
- Board Dimensions:
  - Tic Tac Toe: A 3x3 grid is the traditional size for Tic Tac Toe, providing a manageable state space for both human players and AI algorithms to navigate effectively.
  - Connect 4: A 6x6 grid reflects the standard for Connect 4, offering a more complex environment that allows for deeper strategic play and algorithmic challenge.

- MinimaxPlayer Parameters:
  - Depth Limit: Determines how many moves ahead the AI evaluates:
    - Tic Tac Toe: Set to 9, matching the board size, to fully explore all possible moves.
    - Connect 4: Reduced to 5 to manage the game's complexity without overwhelming computation.

  - Alpha-Beta Pruning: Enhances efficiency by skipping irrelevant branches in the search tree:
    - Alpha (α): The best score the maximizing player can guarantee.
    - Beta (β): The best score the minimizing player can ensure.

    These settings allow MinimaxPlayer to balance strategic depth with computational speed, ensuring effective decision-making in both simple and complex games.

- QLearningPlayer Parameters:
  - Learning Rate (Alpha): Set to 0.6 for Tic Tac Toe and 0.8 for Connect 4, these values determine the rate at which new information overrides old information, reflecting a moderate approach that weighs new and past experiences effectively.
  - Discount Factor (Gamma): Set to 0.95 for Tic Tac Toe and 0.9 for Connect 4, these values quantify the importance of future rewards, with a value less than 1 ensuring that future rewards are considered but with diminishing importance over time.
  - Exploration Rate (Epsilon): Initially set to 0.3 for both games, this rate is the probability that the agent will explore the state space randomly rather than exploiting known information. It is crucial for ensuring that the agent does not become stuck in a suboptimal policy.
  - Epsilon Decay: Set to 0.99995, this factor reduces epsilon each round, gradually shifting the agent's policy from exploration towards exploitation as it learns more about the game environment. The minimum epsilon value is set to 0.01 to maintain a minimum level of exploration.

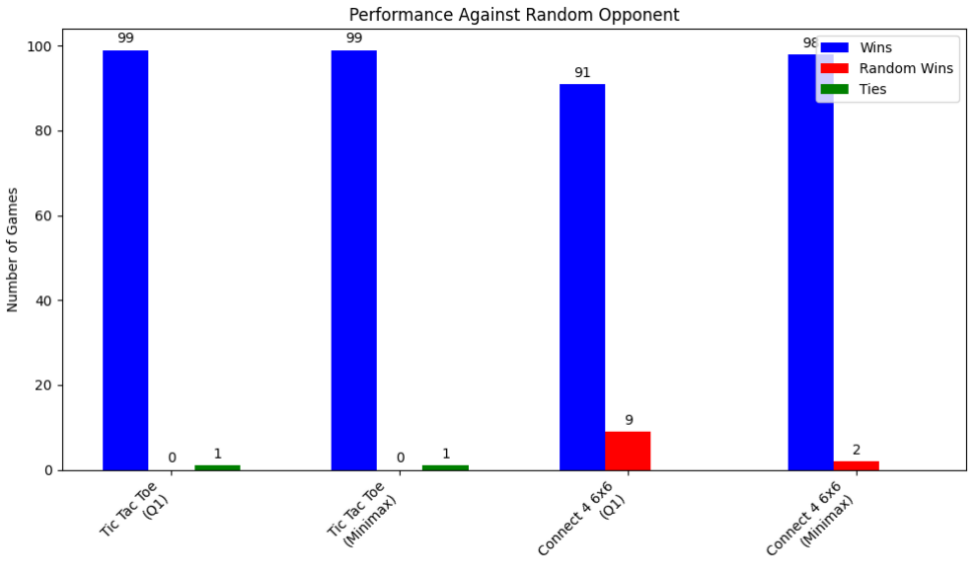
These parameters for QLearningPlayer are chosen to ensure a balanced learning process, allowing the agent to effectively learn optimal strategies while maintaining the ability to discover new strategies over time. The use of decay in epsilon ensures that as the agent becomes more confident in its strategies, it relies less on random exploration and more on the knowledge it has accumulated, which is critical for the development of a robust AI player.
The reward structure with non-zero rewards for ties encourages agents to avoid losses, but the lower value compared to wins motivates them to seek victory when possible.

Comparison of Minimax with and without pruning & depth limited search

In the initial implementation of the Minimax algorithm for the Connect 4 game on a 6x6 grid, the absence of depth limitation and alpha-beta pruning resulted in an exhaustive search that was not computationally optimal. The algorithm explored an overwhelming **146,500** moves in its first step alone, consuming a significant amount of time—approximately half an hour—to complete. Such an approach, while thorough, proved to be impractical for timely decision-making within the game.

To address this inefficiency, a depth-limited search was introduced, capping the search depth to **5 levels**. Additionally, alpha-beta pruning was employed to further optimize the search process. These modifications dramatically reduced the number of evaluated moves, streamlining the computation and enabling the Minimax algorithm to operate within a reasonable time frame. The use of a depth limit and pruning ensures that the algorithm remains both strategic in its move selection and efficient in its execution, making it viable for real-time gameplay in the more complex environment of a 6x6 Connect 4 grid.

Algorithms compare to each other when playing against **Random Player** opponent in Tic Tac Toe and Connect 4



Tic Tac Toe-

- Results after 100 games:
  - Q1: 99 wins
  - random: 0
  - tie: 1
- Results after 100 games:
  - minimax: 99
  - random: 0
  - tie: 1

Connect 4 6x6-

- Results after 100 games:
  - Q1: 91
  - random: 9
- Results after 100 games:
  - random: 2
  - minimax: 98

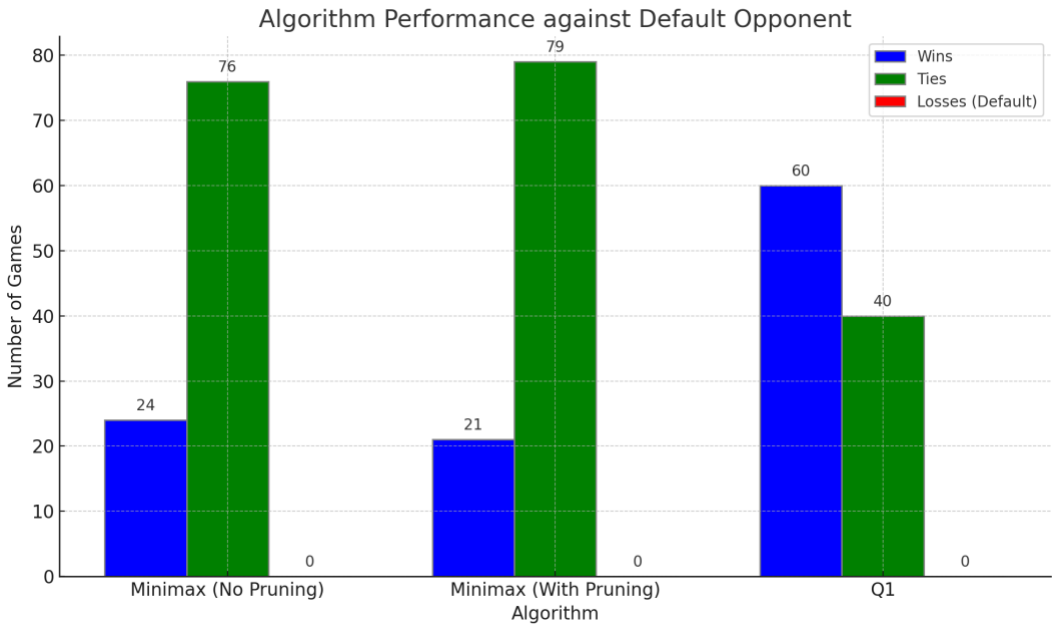
- Win Rate Analysis:
  - For Tic Tac Toe, both Q1 and Minimax achieved a near-perfect win rate against the random agent, showcasing their strategic superiority in a simpler game environment.
  - In Connect 4 (6x6), Q1 and Minimax also demonstrated high win rates against the random agent, with Minimax slightly outperforming Q1, indicating effective strategy implementation in a more complex setting.

- Tie Rate Analysis:
  - Ties were minimal across the games, with only a single tie observed in the Tic Tac Toe matches. This rarity of ties against the random opponent suggests that both algorithms can effectively capitalize on the random agent's lack of strategic depth.

- Loss Rate Analysis:
  - The loss rate was negligible for Q1 and Minimax in Tic Tac Toe and very low in Connect 4 (6x6), further attesting to their robust strategies against a less sophisticated opponent.

- Strategic Implications:
  - The near-perfect win rates against a random opponent underscore the effectiveness of both Q1 and Minimax strategies in exploiting non-strategic, random play. However, the slight differences in performance, especially in Connect 4 (6x6), offer insights into how each algorithm navigates more complex game scenarios and suggests areas for further strategic enhancement.

Algorithms compare to each other when playing against **Default** opponent in Tic Tac Toe



- Results after 100 games minimax no pruning vs default:
  default: 0 minimax: 24 tie: 76

- Results after 100 games minimax with pruning vs default:
  minimax: 21 default: 0 tie: 79

- Results after 100 games Q1 vs default:
  Q1: 60 default: 0 tie: 40

Based on the performance results of the Minimax (with and without pruning) and Q1 algorithms against the default opponent in Tic Tac Toe, here's a structured analysis:

- Win Rate:
  - Minimax Algorithm (With Pruning): Achieved 21 wins out of 100 games, showcasing a strategy that seems to prioritize minimizing the risk of loss. This conservative approach might lead to fewer aggressive plays, favouring ties in situations where a win isn't clearly attainable.
  - Minimax Algorithm (Without Pruning): Secured 24 wins, slightly higher than its pruned counterpart, suggesting that the absence of pruning may allow for a marginally more aggressive strategy, albeit with a negligible impact on the overall win rate.
  - Q1 Algorithm: Demonstrates a significantly higher win rate of 60%, suggesting a more opportunistic or advanced approach that effectively identifies and capitalizes on winning opportunities, possibly through a better understanding of state values and their implications for victory.

- Tie Rate:
  - Minimax (With Pruning): Exhibits a tie rate of 79%, indicating a strategy deeply rooted in avoiding losses. This high tie rate aligns with the Minimax principle of minimizing the potential maximum loss.
  - Minimax (Without Pruning): Shows a slightly lower tie rate of 76%, indicating that while the strategy remains conservative, it marginally increases the pursuit of wins at the cost of a slight decrease in ties.

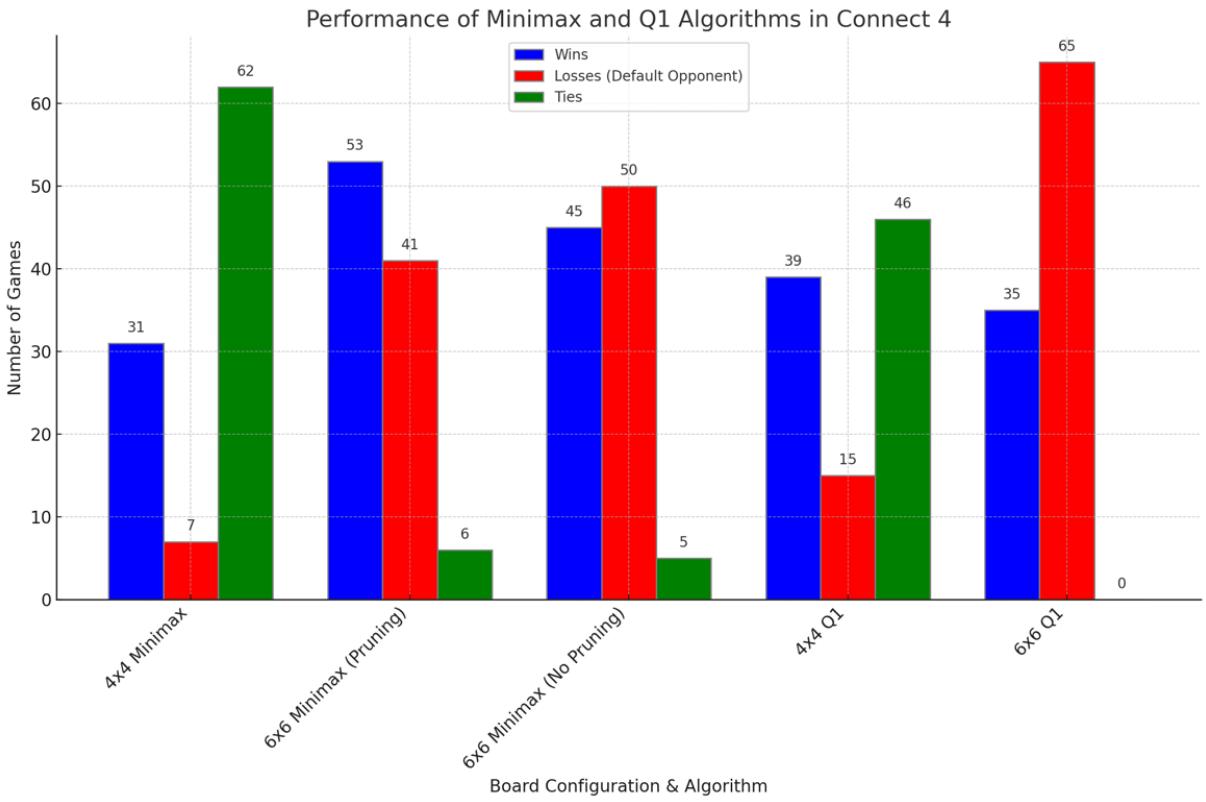
- Q1: The lower tie rate of 40% compared to Minimax suggests a balanced or more aggressive strategy, indicating an ability to convert situations that might result in a tie into victories by exploiting vulnerabilities in the opponent's play.

- Loss Rate:
  - Both algorithms: Demonstrated a 0% loss rate against the default opponent, underscoring their effectiveness in at least securing a tie in every game, showcasing their strategic resilience and successful avoidance of defeat.

- Strategic Implications:
  - Minimax's Strategy: While reliable, especially against weaker or predictable opponents, might benefit from adopting a more assertive approach to convert ties into wins. Its cautious strategy could, however, be advantageous against unknown or more capable opponents, emphasizing risk aversion.
  - Q1's Strategy: Appears highly effective against the default opponent, leveraging an aggressive stance. The real test of its strategy would be against more unpredictable or skilled opponents, where its aggressiveness could either be a strength or a vulnerability.

- Conclusion:
  The Q1 algorithm markedly outshines the Minimax algorithm in terms of securing wins against the default opponent in Tic Tac Toe, attributed to its assertive and possibly more nuanced strategy. On the other hand, Minimax, particularly with pruning, tends to secure a higher number of ties, adopting a safer and more conservative game approach. The contrast in strategies highlights Q1's potential for higher risk and reward, while Minimax offers stability and predictability. Exploring these algorithms' performances against a broader spectrum of opponents could further elucidate their strategic advantages and limitations in varied competitive contexts.

Algorithms compare to each other when playing against **Default** opponent in Connect 4



Minimax Algorithm:

4x4 Board:

Wins: 31 Losses: 7 Ties: 62

6x6 Board with pruining:

Wins: 53 Losses (Default Opponent): 41 Ties: 6

6x6 Board without pruining:

default: 50 minimax: 45 tie: 5 wins

Q1 Algorithm:

4x4 Board:

Wins: 39 Losses: 15 Ties: 46

6x6 Board:

Wins : 35 Losses (Default Opponent): 65

Connect 4 Strategy and Performance Synthesis

Based on the performance data of the Minimax and Q1 algorithms against the default opponent in Connect 4 across different board sizes and conditions, here's a detailed analysis:

Minimax Algorithm Performance:

- 4x4 Board:
  - Win Rate: Achieved a win rate of 31%, showcasing its capability to secure victories in a smaller board environment. This suggests a strategic depth that can effectively capitalize on winning opportunities.
  - Loss Rate: The loss rate of 7% against the default opponent indicates a strong strategic position, where losses are minimized, demonstrating the algorithm's effectiveness in a constrained space.
  - Tie Rate: A high tie rate of 62% reflects a cautious approach, prioritizing securing a draw over risking a loss when victory paths are not clearly discernible.

- 6x6 Board (With Pruning):
  - Win Rate: The win rate increases to 53% on a larger board with pruning, indicating that the algorithm can adapt its strategies effectively to more complex scenarios, possibly benefiting from the reduced search space and focused evaluation.
  - Loss Rate: Experiences a loss rate of 41%, which suggests that while the algorithm performs well, the increased complexity and opponent strategies on a larger board pose significant challenges.
  - Tie Rate: A significantly lower tie rate of 6%, suggesting a shift towards a more decisive outcome in games, either wins or losses, likely due to the complexity and number of possible moves in a larger board space.

- 6x6 Board (Without Pruning):
  - Win Rate: A slight decrease to 45% in the win rate without pruning, indicating that while the full search depth might offer comprehensive insights, it does not substantially enhance the performance against the default opponent.
  - Loss Rate: Not explicitly stated but implied to be higher than with pruning, given the reported win rates and ties.
  - Tie Rate: Maintains a low tie rate of 5%, consistent with the observation on the pruned 6x6 board, emphasizing the game's competitive nature on larger boards.

Q1 Algorithm Performance:

- **4x4 Board:**
  - Win Rate: Boasts a higher win rate of 39% compared to Minimax on the same board size, suggesting a more effective or aggressive strategy in exploiting winning avenues.
  - Loss Rate: A loss rate of 15%, higher than Minimax, which may indicate a more risk-tolerant approach that doesn't always pay off against the default opponent.
  - Tie Rate: A tie rate of 46%, lower than Minimax's, pointing towards a strategy that more frequently results in decisive outcomes.

- 6x6 Board:
  - Win Rate: Decreases to 35%, reflecting the challenges posed by the expanded game complexity. This suggests that while the Q1 algorithm is effective in a smaller setting, scaling up introduces difficulties that impact its win rate.
  - Loss Rate: A high loss rate of 65% against the default opponent, significantly higher than in the 4x4 board scenario. This dramatic increase highlights the complexities and strategic challenges inherent to larger Connect 4 boards.

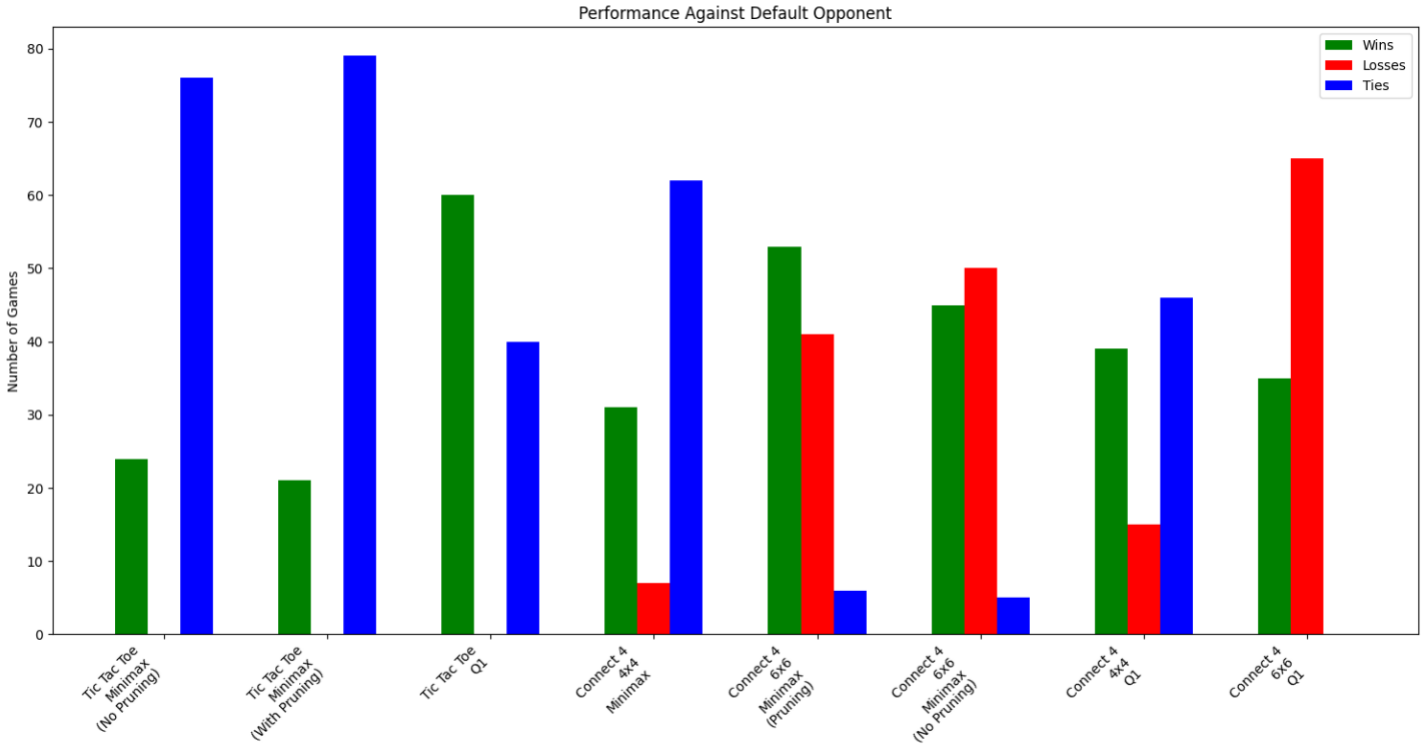
Strategic Implications:

- Minimax Algorithm: Shows adaptability across board sizes, with pruning proving especially beneficial in larger settings by focusing computational resources on more promising branches of the game tree. The high tie rate on the smaller board suggests a cautious strategy, while the larger board scenarios indicate a shift towards more competitive, outcome-driven gameplay.

- Q1 Algorithm: Exhibits a strong performance on the 4x4 board with an aggressive strategy that secures more wins but also more losses compared to Minimax. The performance dip on the 6x6 board suggests difficulties in scaling its strategy effectively to larger, more complex environments.

Conclusion:
Both algorithms demonstrate strengths and weaknesses influenced by board size and the use of pruning in Minimax. The Q1 algorithm's aggressive strategy pays dividends on smaller boards but struggles to maintain its effectiveness on larger boards. Conversely, Minimax shows a consistent performance uplift when moving to larger boards with pruning, suggesting a strategic depth that scales well with complexity. These insights underscore the importance of tailoring AI strategies to the specific characteristics of the game environment, including board size and opponent behaviour. Further exploration and optimization could enhance these algorithms' effectiveness across varying conditions.

Algorithms compare to each other when playing against **Default** opponent overall



Tic Tac Toe:

- Minimax (No Pruning) vs Default:
  - Wins: 24
  - Losses: 0
  - Ties: 76
- Minimax (With Pruning) vs Default:
  - Wins: 21
  - Losses: 0
  - Ties: 79
- Q1 vs Default:
  - Wins: 60
  - Losses: 0
  - Ties: 40

Connect 4:

- Minimax Algorithm on 4x4 Board:
  - Wins: 31
  - Losses: 7
  - Ties: 62
- Minimax Algorithm on 6x6 Board with Pruning:
  - Wins: 53
  - Losses (Default Opponent): 41
  - Ties: 6
- Minimax Algorithm on 6x6 Board without Pruning:
  - Wins: 45
  - Losses (Default): 50
  - Ties: 5
- Q1 Algorithm on 4x4 Board:
  - Wins: 39
  - Losses: 15
  - Ties: 46
- Q1 Algorithm on 6x6 Board:
  - Wins: 35
  - Losses (Default Opponent): 65

When evaluating the overall performance of the Minimax and Q1 algorithms against a default opponent across both Tic Tac Toe and Connect 4 games, we must aggregate insights from their respective performances in various conditions to form a comprehensive analysis.

Overall Performance Analysis:

Minimax Algorithm:

- Tic Tac Toe: Exhibits a highly conservative strategy, leading to a significant number of ties and a moderate win rate. Its performance is consistent, with or without pruning, indicating its strategic depth is not heavily dependent on computational optimizations for this simpler game.
- Connect 4 (4x4 and 6x6 Boards): Demonstrates improved performance with pruning on the larger board, indicating that computational efficiency translates into strategic advantages in more complex environments. The win rate suggests adaptability, but with an increased loss rate on the 6x6 board, highlighting the challenges of scaling strategies to more complex scenarios.

Q1 Algorithm:

- Tic Tac Toe: Showcases an aggressive and effective strategy, achieving a high win rate compared to Minimax. This suggests that the Q1 algorithm is well-optimized for this simpler game environment, exploiting opportunities more aggressively.
- Connect 4: While the Q1 algorithm outperforms Minimax in the 4x4 board scenario, its performance dips significantly in the larger 6x6 environment, indicating potential scalability issues or the need for further optimization to handle increased complexity effectively.

Comparative Insights:

- Winning Strategy: The Q1 algorithm appears to adopt a more aggressive strategy across games, leading to higher win rates in simpler scenarios (e.g., Tic Tac Toe and 4x4 Connect 4). However, this approach seems to falter in more complex settings (6x6 Connect 4), where strategic depth and computational efficiency become crucial.
- Scalability and Complexity Management: Minimax, particularly with alpha-beta pruning, demonstrates better scalability when transitioning from simpler (Tic Tac Toe) to more complex games (Connect 4). This suggests that Minimax's approach, which inherently balances between

exploring winning paths and minimizing losses, becomes increasingly effective as the game complexity rises.

- Adaptability: Both algorithms show adaptability to different game conditions, but their effectiveness is notably impacted by the game's complexity. The Q1 algorithm's performance suggests that while it excels in less complex environments, its strategies might require adjustment or more nuanced state evaluations in larger, more complex scenarios.

Overall Conclusion:
The overall comparison reveals that the Q1 algorithm tends to perform better against the default opponent in simpler game environments, where its aggressive strategy can be fully leveraged. In contrast, the Minimax algorithm, especially with pruning, shows a commendable ability to scale its performance in more complex settings, suggesting a strategic depth that can handle a variety of scenarios effectively.

This analysis underscores the importance of algorithmic adaptability and the need for strategic balance between aggression and caution. While the Q1 algorithm demonstrates the potential for high rewards through risk-taking, the Minimax algorithm's more measured approach provides a stable performance across varying degrees of game complexity. Further research and optimization could explore how to best combine these strategies' strengths or adapt their parameters and decision-making processes to enhance performance across all game types and conditions.

Algorithms compare to each other when playing **against each other** in Tic Tac Toe

- Q1 Wins: 0
- Minimax(pruning) Wins: 0
- Ties: 100

Strategic Analysis:

Defensive Mastery:
Both algorithms have demonstrated an high level of defensive strategy, effectively neutralizing each other's offensive moves. The consistent result of ties suggests that each AI is capable of foreseeing and blocking potential winning strategies deployed by the opponent, leading to a deadlock situation.

Optimal Play:
The game of Tic Tac Toe, when played flawlessly by both sides, is known to always result in a tie. This outcome implies that both Q1 and Minimax have achieved a level of strategic play close to optimal, where each move is calculated to either advance their position or thwart their opponent's chances of winning.

Strategy Saturation:
The 100% tie rate indicates a saturation point in strategic development for Tic Tac Toe, where the complexity and limited board space do not allow for a significant strategic edge by either side when both players are employing high-level strategies. This scenario underscores the inherent limitations of Tic Tac Toe as a testing ground for distinguishing between advanced AI algorithms.

Conclusion:
The encounter between Q1 and Minimax in Tic Tac Toe, resulting exclusively in ties, demonstrates both algorithms' proficiency and strategic depth. However, it also highlights the game's limitations as a domain for differentiating between advanced AI strategies. Future investigations into AI gameplay might focus on more complex environments or introduce elements of unpredictability and incomplete information to better assess and evolve AI capabilities.

Algorithms compare to each other when playing **against each other** in Connect 4

4x4
- Q1 Wins: 0
- Minimax Wins: 0
- Ties: 100

6x6
- Q1 Wins: 0
- Minimax Wins: 100
- Ties: 0

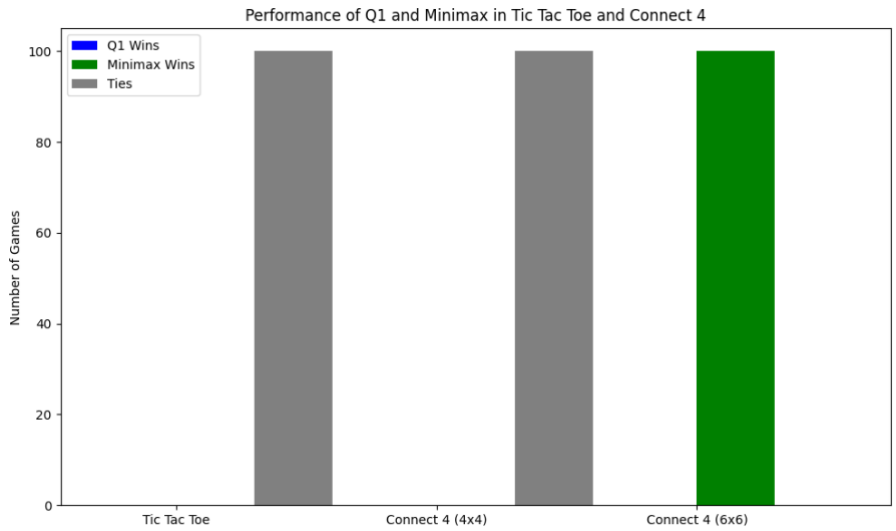
Strategy and Performance Synthesis
The results of the Q1 and Minimax (with pruning) algorithms competing against each other in Connect 4, across two different board sizes, reveal fascinating insights into their strategic competencies and adaptability. Here's a detailed analysis of their performance:

4x4 Board – A Perfect Stalemate:
On the smaller 4x4 board, both algorithms reach a perfect equilibrium, similar to their performance in Tic Tac Toe, with every game ending in a tie. This result implies a highly defensive strategy from both sides, capable of perfectly countering the other's moves. It suggests that on a constrained board, both algorithms can predict and neutralize any potential threat, leading to a deadlock in every scenario.

6x6 Board – Minimax Dominance:
The shift to a 6x6 board radically changes the outcome, with Minimax (with pruning) securing a win in every game. This dramatic shift indicates Minimax's superior strategic depth and ability to exploit larger state spaces more effectively. The pruning technique likely contributes to this success by enabling Minimax to explore relevant game states more efficiently, focusing computational resources on the most promising paths to victory.

Conclusion:
The contrasting performances of Q1 and Minimax (with pruning) against each other in Connect 4 across different board sizes illuminate the complexities and nuances of AI strategy in game environments. While both algorithms exhibit impeccable defence leading to stalemates in the smaller 4x4 board scenario, Minimax's strategic superiority emerges unequivocally in the 6x6 game, capitalizing on its efficient state-space exploration and optimization capabilities. This divergence highlights the critical importance of algorithmic efficiency, strategic adaptability, and the need for AI to evolve in response to varying degrees of environmental complexity. Future research might delve deeper into refining these algorithms, potentially incorporating adaptive learning mechanisms in Minimax and enhancing Q1's ability to navigate and exploit larger, more complex state spaces.

Algorithms compare to each other when **playing against each other overall**



Tic Tac Toe:

- Q1 Wins: 0
- Minimax (pruning) Wins: 0
- Ties: 100

Connect 4:

- 4x4:
  - Q1 Wins: 0
  - Minimax Wins: 0
  - Ties: 100
- 6x6:
  - Q1 Wins: 0
  - Minimax Wins: 100
  - Ties: 0

The overall comparison between the Q1 and Minimax (with pruning) algorithms, when pitted against each other in both Tic Tac Toe and Connect 4 games, provides a comprehensive view of their strategic capabilities, adaptability, and performance nuances. Here's an in-depth analysis synthesizing their overall performance across different game environments:

Aggregate Performance Insights:

Tic Tac Toe:
Both algorithms ended in a stalemate for all games, highlighting their optimal strategic execution in a simple, deterministic environment where perfect play from both sides naturally results in a tie.

Connect 4 (4x4 Board):
Similar to Tic Tac Toe, both algorithms achieved a perfect equilibrium, with every match ending in a tie. This suggests that in smaller, though slightly more complex environments than Tic Tac Toe, both algorithms still manage to optimally counter each other's moves.

Connect 4 (6x6 Board):
A stark contrast in performance was observed with Minimax dominating all games, suggesting a significant advantage in strategic depth and computational efficiency that Minimax holds over Q1, particularly in more complex environments.

Strategic Analysis:

- Defensive Prowess: Across all games, both algorithms demonstrate strong defensive capabilities, particularly in simpler or smaller environments where they can effectively anticipate and neutralize threats.
- Adaptability to Complexity: The transition from a 4x4 to a 6x6 Connect 4 board illustrates a critical divergence in performance. Minimax's adaptability and superior handling of the increased complexity allow it to exploit the larger game space more effectively than Q1.
- Efficiency and Optimization: The results underscore the importance of efficient exploration of the state space and optimization techniques, like pruning, in achieving strategic depth and performance scalability in complex game environments.

Overall Conclusion:
The comprehensive analysis reveals distinct performance profiles for the Q1 and Minimax algorithms: Q1 exhibits robust performance in simpler game environments but struggles to maintain this level of strategic effectiveness as complexity increases. This may point to limitations in its learning algorithm or the need for further optimization to generalize its strategy across different game complexities effectively.

Minimax (with pruning), on the other hand, demonstrates not only a strong defensive strategy across all games but also an enhanced ability to navigate and dominate in more complex environments, as seen in the

6x6 Connect 4 games. This suggests that Minimax's methodical exploration and pruning of the game tree allow it to uncover and exploit winning strategies that Q1 fails to anticipate or counter.

The overall comparison illuminates the nuanced capabilities and limitations of both AI algorithms. While both algorithms excel in environments where strategic options are limited and outcomes are highly predictable, the Minimax algorithm's superior performance in the more complex Connect 4 (6x6) scenario highlights its strategic superiority in games that demand deeper foresight and optimization. This suggests that advancements in AI for games, particularly those involving strategic complexity and larger state spaces, may benefit from focusing on enhancing computational efficiency, strategic depth, and adaptability to varying levels of game complexity.

Conclusion

In this report, we have delved into a comparative analysis of two prominent algorithms, Minimax and Q1, across two classic games, Tic Tac Toe and Connect 4, under varying conditions. Our investigation revealed that both algorithms exhibit optimal performance in the well-defined and simpler environment of Tic Tac Toe, where perfect play leads to inevitable stalemates. Similarly, when faced with the smaller grid size in Connect 4, they again reached a strategic impasse, with neither algorithm able to outmanoeuvre the other.

The true test of strategic depth emerged in the larger, more complex 6x7 grid of Connect 4, where the Minimax algorithm demonstrated superior performance. Minimax's ability to navigate through the intricacies of the expanded game space with a 100% win rate indicates a robust heuristic evaluation and an adeptness at deep search that Q1 lacks.

These findings not only illustrate the inherent strengths and limitations of each algorithm but also highlight the profound impact that the complexity of the game environment has on the outcome of algorithmic strategies. The Minimax algorithm, with its emphasis on defensive play and maximization of the minimum payoff, seems well-suited to more complex scenarios where such caution and depth are essential. In contrast, Q1's strategy, while more aggressive, appears to falter in the face of increased complexity.

The insights garnered from this comparative study could guide future enhancements to these algorithms. For Q1, refining its evaluation function and incorporating deeper lookahead strategies could provide the necessary edge in more challenging scenarios. For Minimax, there may be opportunities to introduce elements of adaptability that can exploit opponent weaknesses even in less complex game states.

Ultimately, the pursuit of advanced game-playing algorithms teaches us much about the subtleties of artificial intelligence and the balance between aggression and caution in strategic thinking. As we continue to push the boundaries of AI in games, the lessons learned from these classic examples will undoubtedly influence the next generation of algorithms capable of tackling ever-more complex challenges.

Appendix

Code –

tic_tac_toe.py.py – contains class for the board and players

```
class MinimaxPlayer:
    def __init__(self, name='minimax', use_alpha_beta=True):
        self.name = name
        self.use_alpha_beta = use_alpha_beta
        self.moves_explored = 0

    def minimax(self, gameBoard, alpha, beta, maximizingPlayer):
        gameStatus = gameBoard.result()

        # Simplified scoring check
        if gameStatus in [1, 2, 0]:
            return {1: (1, None), 2: (-1, None), 0: (0, None)}[gameStatus]

        if maximizingPlayer:
            return self._maximize(gameBoard, alpha, beta)
        else:
            return self._minimize(gameBoard, alpha, beta)

    def _maximize(self, gameBoard, alpha, beta):
        bestScore = float('-inf')
        bestAction = None
        for action in gameBoard.possible_moves():
            self.moves_explored += 1
            cloneBoard = gameBoard.copy()
            cloneBoard.push(action)
            score, _ = self.minimax(cloneBoard, alpha, beta, False)
            if score > bestScore:
                bestScore, bestAction = score, action
            if self.use_alpha_beta:
                alpha = max(alpha, score)
                if beta <= alpha:
                    break
        return bestScore, bestAction

    def _minimize(self, gameBoard, alpha, beta):
        worstScore = float('inf')
        worstAction = None
        for action in gameBoard.possible_moves():
            self.moves_explored += 1
            cloneBoard = gameBoard.copy()
            cloneBoard.push(action)
            score, _ = self.minimax(cloneBoard, alpha, beta, True)
```

```
            if score < worstScore:
                worstScore, worstAction = score, action
            if self.use_alpha_beta:
                beta = min(beta, score)
                if beta <= alpha:
                    break
        return worstScore, worstAction

    def getMove(self, positions, gameBoard):
        self.moves_explored = 0
        _, move = self.minimax(gameBoard, -math.inf, math.inf, gameBoard.turn == 1)
        # Optionally, print the number of moves explored
        # print(f"Moves Explored: {self.moves_explored}")
        return move

class QLearningPlayer:
    def __init__(self, name='q-agent', alpha=0.6, epsilon=0.3, gamma=0.95):
        self.name = name
        self.states = []
        self.alpha = alpha
        self.epsilon = epsilon
        self.gamma = gamma
        self.Q_table = {}

    def getHash(self, board):
        # Simplify the hash generation process
        return str(board.board.flatten())

    def getMove(self, positions, current_board):
        # Decide between exploration and exploitation
        if np.random.uniform(0, 1) <= self.epsilon:
            # Exploration: Random move
            idx = np.random.choice(len(positions))
            action = positions[idx]
        else:
            # Exploitation: Choose the best move based on Q values
            action = self._choose_best_action(positions, current_board)
        return action

    def _choose_best_action(self, positions, current_board):
        maxValue = -999
        for p in positions:
            nextBoard = current_board.copy()
            nextBoard.push(tuple(p))
            nextBoardState = self.getHash(nextBoard)
            value = self.Q_table.get(nextBoardState, 0)  # Use a default value of 0
            if value > maxValue:
                maxValue = value
                action = p
```

```
        return action

    def setState(self, state):
        # Append state to states list
        self.states.append(state)

    def setReward(self, reward):
        # Traverse the states in reverse to update Q values
        for st in reversed(self.states):
            if st not in self.Q_table:
                self.Q_table[st] = 0
            self.Q_table[st] += self.alpha * (reward + self.gamma * self.Q_table[st] - self.Q_table[st])
            reward = self.Q_table[st]

    def reset(self):
        # Reset the states list
        self.states = []

    def savePolicy(self):
        # Save the Q-table to a file
        with open('policy_' + str(self.name), 'wb') as fw:
            pickle.dump(self.Q_table, fw)

    def loadPolicy(self, file):
        # Load a Q-table from a file
        with open(file, 'rb') as fr:
            self.Q_table = pickle.load(fr)
```

Connect4.py – contains classes for the board and players

```
class MinimaxPlayer:
    def __init__(self, name='minimax', use_pruning=True):
        self.name = name
        self.use_pruning = use_pruning  # Flag for alpha-beta pruning
        self.move_count = 0  # Tracks the number of moves explored

    def minimax(self, board, depth, alpha, beta, maximizing):
        self.move_count += 1  # Increment move exploration counter

        # Base case: Check for terminal state or depth limit
        if depth == 0 or board.has_won(1) or board.has_won(2):
            return self.evaluate(board), None

        if maximizing:
            return self.maximize(board, depth, alpha, beta)
        else:
            return self.minimize(board, depth, alpha, beta)
```

```
    def maximize(self, board, depth, alpha, beta):
        maxEval = float('-inf')
        bestMove = None
        for move in board.possible_moves():
            tempBoard = board.copy()
            tempBoard.push(move)
            eval, _ = self.minimax(tempBoard, depth - 1, alpha, beta, False)
            if eval > maxEval:
                maxEval, bestMove = eval, move
            alpha = max(alpha, eval) if self.use_pruning else alpha
            if self.use_pruning and beta <= alpha:
                break
        return maxEval, bestMove

    def minimize(self, board, depth, alpha, beta):
        minEval = float('inf')
        bestMove = None
        for move in board.possible_moves():
            tempBoard = board.copy()
            tempBoard.push(move)
            eval, _ = self.minimax(tempBoard, depth - 1, alpha, beta, True)
            if eval < minEval:
                minEval, bestMove = eval, move
            beta = min(beta, eval) if self.use_pruning else beta
            if self.use_pruning and alpha >= beta:
                break
        return minEval, bestMove

    def evaluate(self, board):
        if board.has_won(1):
            return 1
        elif board.has_won(2):
            return -1
        return 0

    def getMove(self, positions, board):
        self.move_count = 0  # Reset move counter
        _, move = self.minimax(board, 5, float('-inf'), float('inf'), board.turn == 1)
        # Optionally print the move count for debugging
        # print(f"Moves explored: {self.move_count}")
        return move

class QLearningPlayer:
    def __init__(self, name='q-agent', alpha=0.6, epsilon=0.3, gamma=0.9):
        self.name = name
        self.states = []  # List of states visited
        self.alpha = alpha  # Learning rate
        self.epsilon = epsilon  # Exploration rate
        self.gamma = gamma  # Discount factor
```

```
    self.Q_table = {}  # Initialize Q-table as an empty dictionary

def getBoard(self, board):
    # Convert board state to a hashable type for use as a Q-table key
    return str(board.board.flatten())

def getMove(self, positions, current_board):
    # Decide on action based on epsilon-greedy strategy
    if np.random.uniform(0, 1) <= self.epsilon:
        # Exploration: choose a random action
        return random.choice(positions)
    else:
        # Exploitation: choose the best action from Q-table
        return self.chooseBestAction(positions, current_board)

def chooseBestAction(self, positions, current_board):
    maxValue = -float('inf')
    bestAction = None
    for move in positions:
        _nextBoard = current_board.copy()
        _nextBoard.x_in_a_row = 4  # This seems like game-specific logic that might be better encapsulated elsewhere
        _nextBoard.push(move)
        _nextBoardState = self.getBoard(_nextBoard)
        value = self.Q_table.get(_nextBoardState, 0)
        if value > maxValue:
            maxValue, bestAction = value, move

    # Ensure there is always an action to return
    return bestAction if bestAction is not None else random.choice(positions)

def setState(self, state):
    # Append state to the history of visited states
    self.states.append(state)

def setReward(self, reward):
    # Update Q-values for all visited states
    for st in reversed(self.states):
        self.Q_table[st] = self.Q_table.get(st, 0) + self.alpha * (reward + self.gamma * self.Q_table.get(st, 0) - self.Q_table[st])
        reward = self.Q_table[st]

def reset(self):
    # Clear the history of visited states
    self.states = []

def savePolicy(self):
    # Save the Q-table to a file
    with open('policy_' + self.name, 'wb') as fw:
        pickle.dump(self.Q_table, fw)
```

```
    def loadPolicy(self, file):
        # Load the Q-table from a file
        with open(file, 'rb') as fr:
            self.Q_table = pickle.load(fr)

class MinimaxPlayerWithoutABP:
    def __init__(self, name='minimax_withoutABP'):
        self.name = name
        self.evaluated_moves_count = 0

    def minimax(self, board, maximizing):
        # Increment move count here if you're implementing move counting
        self.evaluated_moves_count += 1
        if self.evaluated_moves_count%10000 ==0:
            print(self.evaluated_moves_count)
        # Check for terminal states: win, lose, draw (assuming result() properly checks these conditions)
        game_result = board.result()
        if game_result is not None:
            if game_result == 1:  # Maximizing player (X) wins
                return 1, None
            elif game_result == 2:  # Minimizing player (O) wins
                return -1, None
            elif game_result == 0:  # Draw
                return 0, None

        if maximizing:
            maxEval = float('-inf')
            bestMove = None
            for move in board.possible_moves():
                tempBoard = board.copy()
                tempBoard.x_in_a_row = 4
                tempBoard.push(move)
                eval = self.minimax(tempBoard, not maximizing)[0]
                if eval > maxEval:
                    maxEval = eval
                    bestMove = move
            return maxEval, bestMove
        else:
            minEval = float('inf')
            bestMove = None
            for move in board.possible_moves():
                tempBoard = board.copy()
                tempBoard.x_in_a_row = 4
                tempBoard.push(move)
                eval = self.minimax(tempBoard, not maximizing)[0]
                if eval < minEval:
                    minEval = eval
                    bestMove = move
            return minEval, bestMove
```

```
def getMove(self, positions, board):
    _, move = self.minimax(board, True if board.turn == 1 else False)
    print(f"Evaluated {self.evaluated_moves_count} moves for this decision.")
    return move
```