An Overview and Analysis of Reinforcement Learning Algorithms for Blackjack

Introduction

Reinforcement Learning (RL) is a branch of machine learning that focuses on decision-making processes. It involves an agent that learns to make decisions by interacting with an environment, taking actions that maximize cumulative reward. This learning paradigm has been applied to a wide range of problems, from robotics to game playing. In this study, we focus on the application of RL algorithms to the game of Blackjack, a popular card game.

We implemented and compared the performance of three well-known RL algorithms: Monte Carlo, SARSA (State-Action-Reward-State-Action), and Q-learning. These algorithms represent different approaches to learning an optimal policy, which is a mapping from states to actions that maximizes the expected cumulative reward. The performance of each algorithm was measured based on win, draw, and loss counts over multiple episodes, providing a comprehensive evaluation of their effectiveness in the game of Blackjack.

Background

Blackjack

Blackjack is a popular card game played in casinos worldwide. The objective of the game is to have a hand value as close to 21 as possible without exceeding it. The game involves a player and a dealer, with the player making decisions based on their hand and the dealer's visible card. The player can choose to "hit" (draw another card), "stand" (stop drawing cards). The dealer follows a fixed policy, typically standing on 17 or higher and hitting otherwise. The game is particularly interesting for RL because it involves a mix of chance (due to the random draw of cards) and strategy (in the choice of actions).

Reinforcement Learning Algorithms

Reinforcement Learning algorithms learn an optimal policy by interacting with the environment and observing the results of their actions. Here, we briefly describe the three algorithms used in this study:

1. **Monte Carlo Methods**: These methods learn by averaging the returns (cumulative rewards) observed after visiting each state-action pair. They do not require a model of the environment and are particularly effective for episodic tasks, where the episodes (games) are all independent and identically distributed.
2. **SARSA (State-Action-Reward-State-Action)**: SARSA is a Temporal Difference (TD) learning algorithm that updates the value of a state-action pair based on the immediate reward plus the value of the next state-action pair. It is an on-policy algorithm, meaning it learns the value of the policy being followed.
3. **Q-Learning**: Q-Learning is another TD learning algorithm, but it is off-policy: it learns the value of the optimal policy regardless of the policy being followed. It updates the value of a state-action pair based on the immediate reward plus the maximum value of the next state over all possible actions.

Implementation

Overview of Algorithms

The project implements three reinforcement learning (RL) algorithms for playing the game of Blackjack: Monte Carlo Control with exploring starts, SARSA (State-Action-Reward-State-Action), and Q-Learning. Each algorithm follows a distinct approach to learning optimal policies through interaction with the environment.

1. **Monte Carlo On-Policy Control**:
    - **Exploring Starts**: One configuration of Monte Carlo uses Exploring Starts. This technique ensures exploration by starting episodes with both states and actions sampled uniformly. This guarantees that all state-action pairs are visited infinitely often in the limit, which is a necessary condition for convergence to the optimal policy.
    - **Epsilon-Greedy Policy**: During action selection, it balances exploration (random actions) and exploitation (greedy actions based

on learned Q-values) based on a specified epsilon parameter. This allows the algorithm to explore different actions while gradually favoring those that have yielded higher rewards in the past.
    - **Q-Value Update**: Q-values are updated after completing episodes using the average return observed for each state-action pair encountered. This is a simple and effective way to estimate the expected return of each state-action pair under the current policy.
2. **SARSA (State-Action-Reward-State-Action) On-Policy Control**:
    - **Epsilon-Greedy Policy**: Similar to Monte Carlo, SARSA uses an epsilon-greedy policy to balance exploration and exploitation. This allows the algorithm to gradually improve its policy while ensuring sufficient exploration.
    - **On-Policy Update**: Q-values are updated after each action using the current action and the next action derived from the policy. This makes SARSA an on-policy algorithm, meaning it learns the value of the policy it follows.
    - **Terminal State Handling**: Adjustments are made in terminal states to ensure the correct calculation of Q-values based on rewards and next state values. This is necessary because the value of the terminal state is always zero in episodic tasks.
3. **Q-Learning Off-Policy Control**:
    - **Epsilon-Greedy Policy**: Q-Learning also employs an epsilon-greedy policy but learns from the maximum Q-value of the next state rather than the action actually taken. This allows it to learn the optimal policy regardless of the policy being followed.
    - **Off-Policy Update**: Q-values are updated irrespective of the action actually chosen, based on the maximum Q-value of the next state. This makes Q-Learning an off-policy algorithm, capable of learning the optimal policy while following a different (exploratory) policy.
    - **Comparison with SARSA**: Q-Learning tends to converge to the optimal policy faster than SARSA due to its off-policy nature but may overestimate Q-values in certain scenarios. This is known as the "maximization bias" and can lead to instability in some cases.

Implementation Details

Each algorithm is encapsulated within a respective class inheriting from `PlayerRLAgent`, which extends the `Player` class. This design allows for modular implementation and testing of different RL algorithms against the same environment.

- **PlayerRLAgent Class**:
    - Manages Q-tables (`q_table` and `q_table_counts`) to store and update Q-values for state-action pairs. The Q-table is a data structure that stores the estimated value of each state-action pair, while `q_table_counts` keeps track of the number of times each state-action pair has been visited. This information is used to update the Q-values in a way that ensures convergence to the true values under certain conditions.
    - Defines methods for action selection (`choose_action`) based on epsilon-greedy policies and updating Q-values (`update_q_table`) based on rewards and next states. The `choose_action` method selects an action to take in the current state, either randomly (with probability epsilon) or greedily (with probability 1-epsilon), based on the current Q-values. The `update_q_table` method updates the Q-value of the current state-action pair based on the immediate reward and the value of the next state-action pair.
- **MonteCarloOnPolicyControl Class**:
    - Inherits from `PlayerRLAgent`.
    - Implements Monte Carlo Control with methods specific to handling returns, exploring starts, and updating Q-values accordingly. The `handle_return` method calculates the return (cumulative discounted reward) of an episode, which is used to update the Q-values. The `exploring_starts` method ensures that all state-action pairs are visited infinitely often in the limit by starting episodes with both states and actions sampled uniformly.
- **SARSAOnPolicyControl Class**:
    - Inherits from `PlayerRLAgent`.
    - Implements SARSA with methods for updating Q-values based on state transitions and rewards, considering both exploration and exploitation. The `update_q_table` method updates the Q-value of the

current state-action pair based on the immediate reward and the value of the next state-action pair, following the policy derived from the current Q-values.

- **QLearningOffPolicyControl Class**:
    - Inherits from `SARSAOnPolicyControl`.
    - Extends SARSA by using an off-policy approach to update Q-values, learning from the maximum Q-value of the next state-action pair. The `update_q_table` method updates the Q-value of the current state-action pair based on the immediate reward and the maximum value of the next state over all possible actions, regardless of the action actually taken.

Algorithm Configurations

Each algorithm instance is configured with specific parameters such as learning rate (`alpha`), discount factor (`gamma`), and exploration rate (`epsilon`). These parameters are crucial in determining how aggressively the agent explores new actions versus exploiting known good actions.

- **Learning Rate (α)**: This parameter determines how much the Q-value of a state-action pair is updated at each step. A high learning rate means that recent rewards have a greater influence on the Q-value, while a low learning rate means that past rewards are given more weight. The learning rate is typically set to a small positive value (e.g., 0.1) to ensure gradual convergence.
- **Discount Factor (γ)**: This parameter determines the present value of future rewards. A high discount factor means that future rewards are considered almost as valuable as immediate rewards, while a low discount factor means that immediate rewards are given much more weight. The discount factor is typically set close to 1 (e.g., 0.9) to encourage long-term optimization.
- **Exploration Rate (ε)**: This parameter determines the probability of taking a random action at each step. A high exploration rate means that the agent explores the environment more, while a low exploration rate means that the agent exploits its current knowledge more. The exploration rate is typically set to a small positive value (e.g., 0.1) to ensure sufficient exploration.

Exploration vs Exploitation

Exploration and exploitation are two fundamental concepts in reinforcement learning. They represent the trade-off between trying out new actions to discover potentially better strategies (exploration) and sticking with the best action currently known to maximize reward (exploitation).

- **Exploration**: In the context of Blackjack, exploration means the agent chooses a random action, regardless of the current Q-values. This ensures the agent explores various actions and states, preventing premature convergence to suboptimal policies. For example, even if the agent has learned that standing is the best action when the hand value is 20, it might still choose to hit occasionally to explore the possibility of getting an Ace (which would result in a hand value of 21, the best possible outcome).
- **Exploitation**: Exploitation, on the other hand, means the agent chooses the action with the highest Q-value for the current state, according to its current knowledge. This allows the agent to make the best decision based on what it has learned so far. For instance, if the agent has learned that hitting when the hand value is 15 usually results in a better outcome than standing, it will choose to hit more often when the hand value is 15.

Balancing exploration and exploitation is crucial for the success of RL algorithms. Too much exploration can lead to unnecessary losses, as the agent keeps trying out suboptimal actions. On the other hand, too much exploitation can lead to suboptimal learning, as the agent might miss out on discovering better strategies.

In our implementation, we control the balance between exploration and exploitation using an epsilon-greedy policy. The agent chooses a random action with probability epsilon (exploration) and the action with the highest Q-value with probability 1-epsilon (exploitation). We also experimented with different decay strategies for epsilon, which gradually decrease the amount of exploration over time.

Experiment Setup

Parameters

We used the following parameters for our experiments:

- **Alpha (α)**: The learning rate, set to 0.1. This parameter determines how much the agent updates its Q-values based on new experiences. A higher alpha means the agent gives more weight to recent experiences, while a lower alpha means the agent gives more weight to past experiences.
- **Gamma (γ)**: The discount factor, set to 0.9. This parameter determines how much the agent values future rewards compared to immediate rewards. A higher gamma means the agent values future rewards more, promoting strategies that may not yield immediate rewards but are beneficial in the long run.
- **Epsilon (ε)**: The exploration rate, set to 0.1. This is the default value of the parameter, and is overridden by the appropriate epsilon configuration as needed. This parameter determines the probability of the agent choosing a random action (exploration) over the action with the highest Q-value (exploitation).

Configurations

We tested each algorithm with different epsilon configurations to study the effect of exploration decay on the performance:

1. **1/k**: Epsilon decreases over time as `1/k`, where `k` is the episode number. This configuration ensures that the amount of exploration gradually decreases over time, allowing the agent to exploit its learned knowledge more as it gains experience.
2. **e^(-k/1000)**: Epsilon decreases exponentially over time as `e^(-k/1000)`. This configuration results in a more gradual decrease in exploration, forcing the agent to explore more states at the start.
3. **e^(-k/10000)**: Epsilon decreases exponentially over time as `e^(-k/10000)`, but at a slower rate compared to `e^(-k/1000)`. This configuration allows for more exploration in the early stages of learning.

Metrics

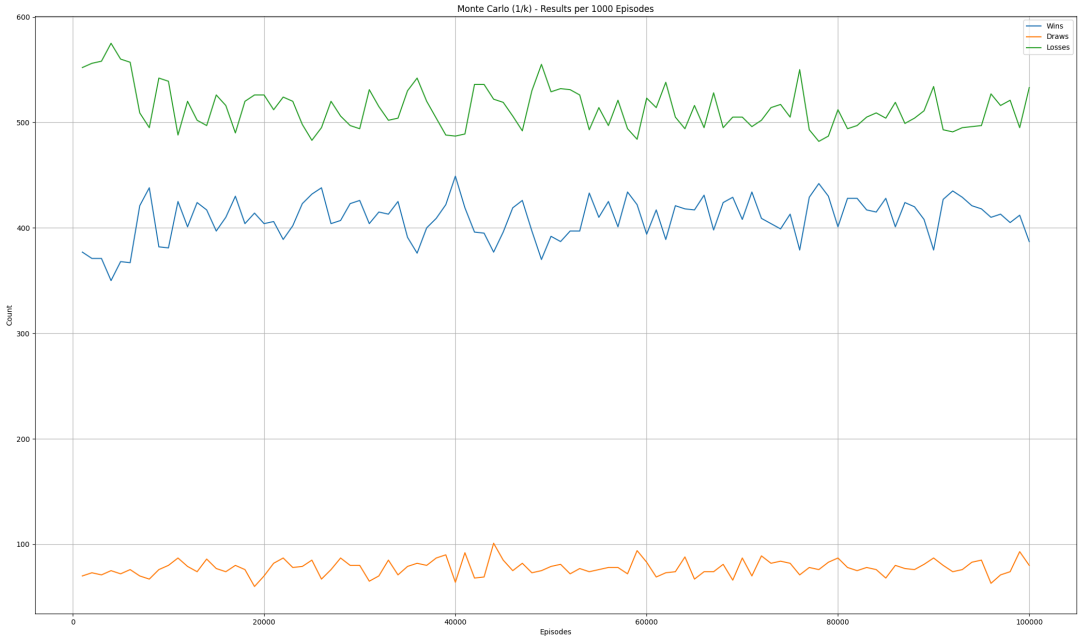
We measured the performance of each algorithm based on the number of wins, draws, and losses over 100000 episodes. These metrics provide a comprehensive evaluation of the effectiveness of each algorithm in the game of Blackjack. We also plotted these metrics over time to visualize the learning process and understand the behavior and stability of each algorithm.

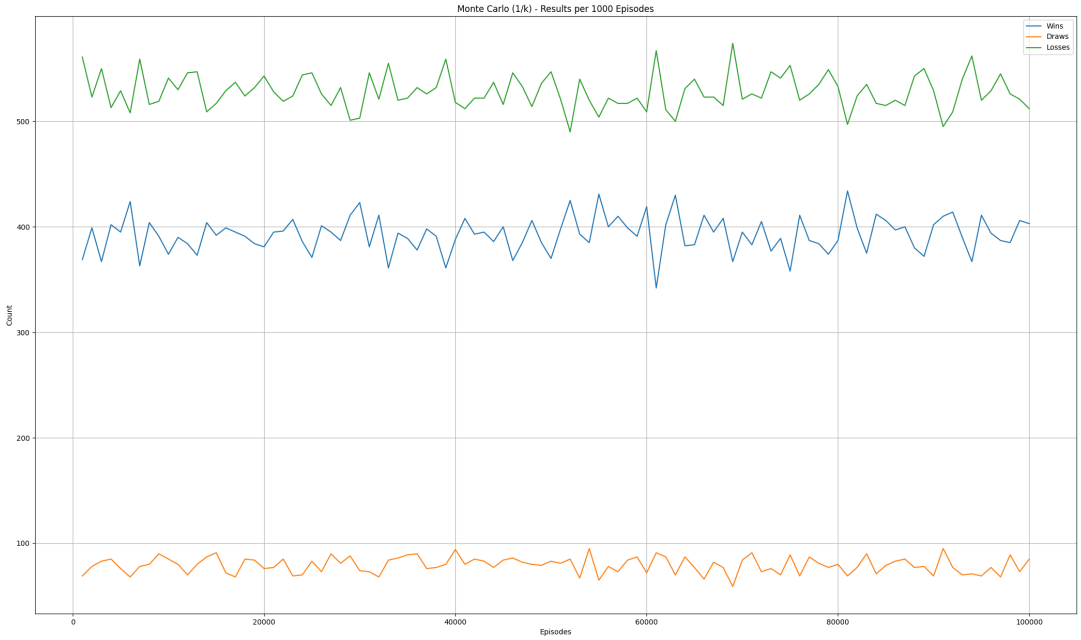
Results and Analysis

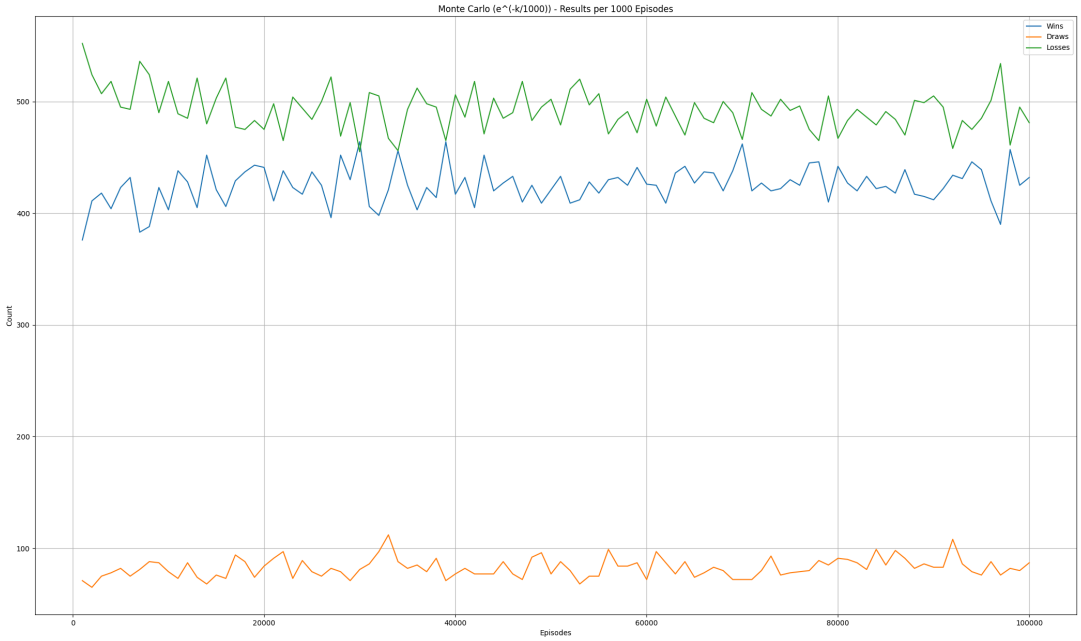
We analyzed the performance of each algorithm based on the win-loss-draw counts and the state-action pair counts. The win-loss-draw counts provide a direct measure of the effectiveness of each algorithm in winning the game. The state-action pair counts, on the other hand, provide insights into the learning process of each algorithm, showing how often each state-action pair was visited.

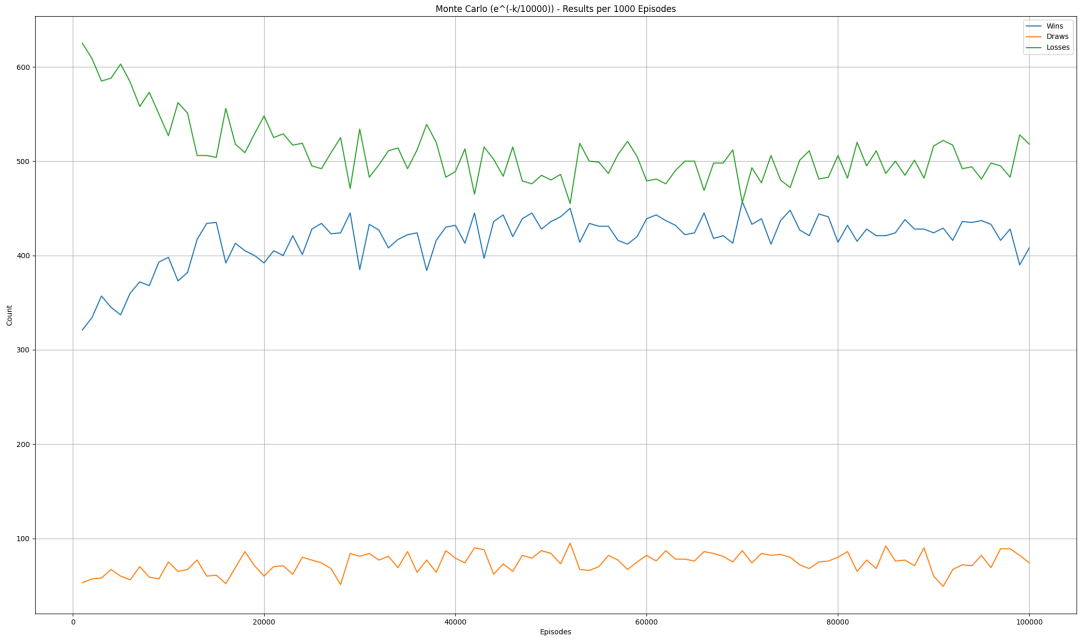
Monte Carlo Method

Win-Loss-Draw Counts



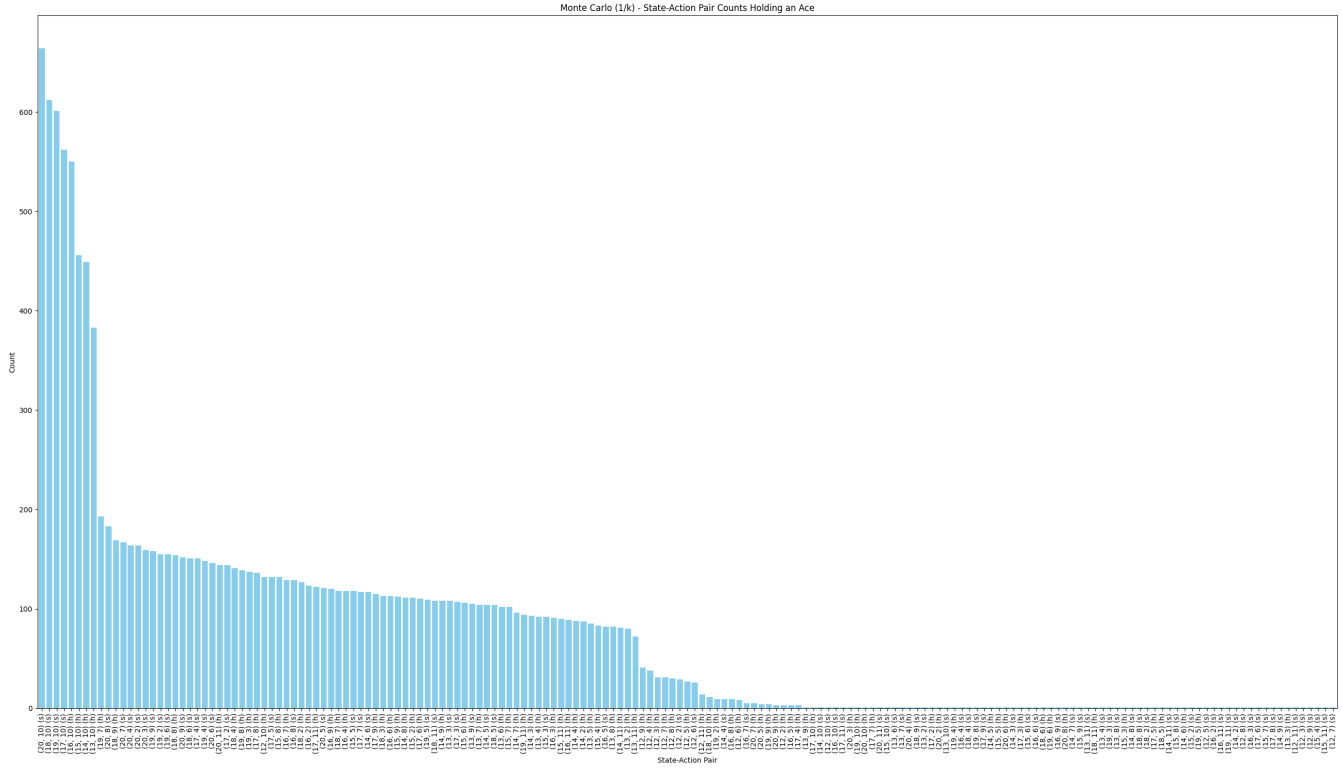

Monte Carlo (1/k) - Results per 1000 Episodes
Wins
Draws
Losses
Count
Episodes
0
20000
40000
60000
80000
100000
100
200
300
400
500

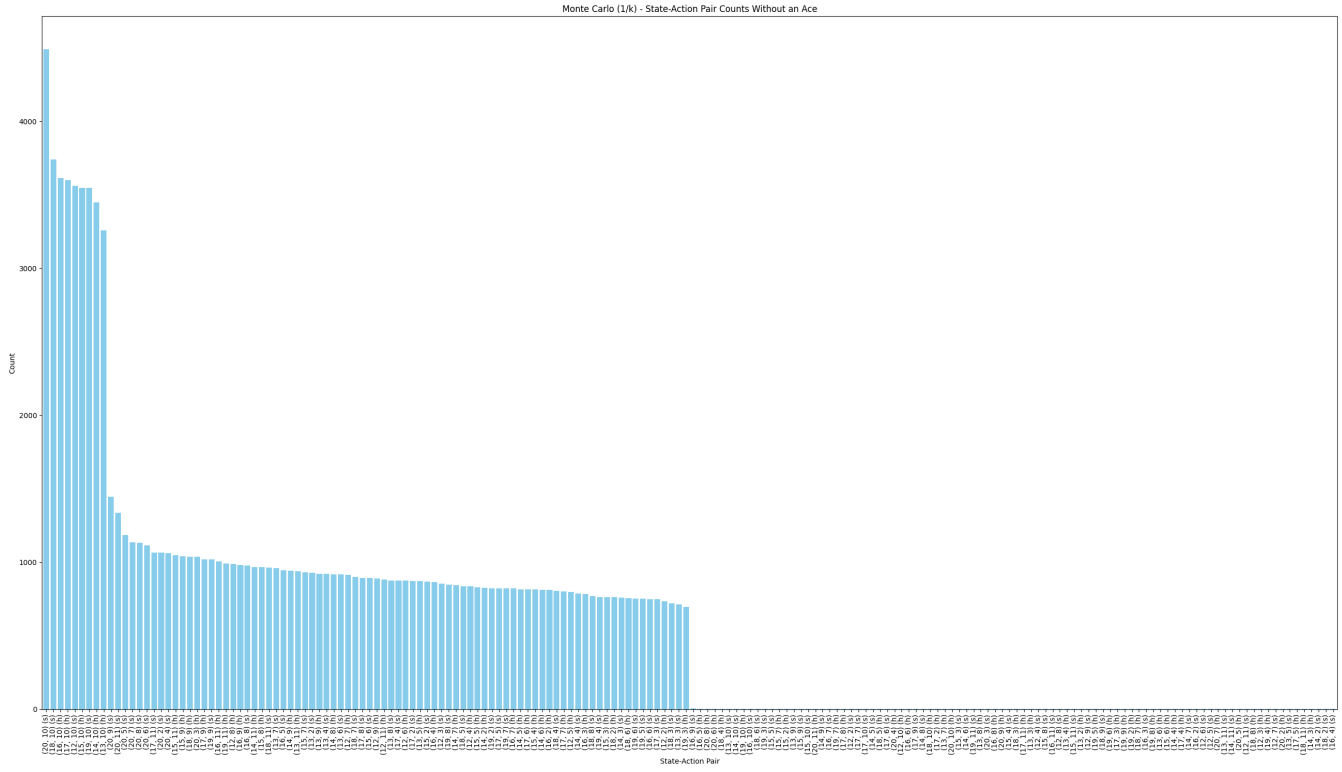

Monte Carlo (e^(-k/1000)) - Results per 1000 Episodes
Wins
Draws
Losses
Count
Episodes
0
20000
40000
60000
80000
100000
100
200
300
400
500



The Monte Carlo method with `1/k` showed a steady learning curve but required many episodes to converge. The `e^(-k/1000)` and `e^(-k/10000)` configurations converged faster, with the former being slightly more aggressive in exploration. This indicates that the rate of exploration decay plays a significant role in the speed of convergence and overall performance of the algorithm.

State-Action Pair Counts

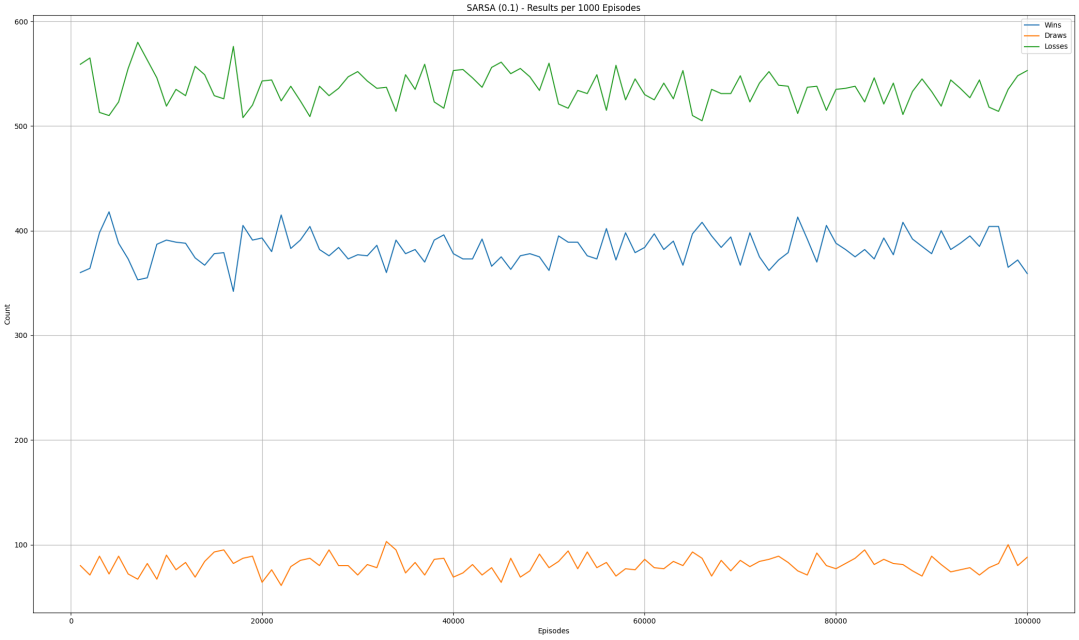


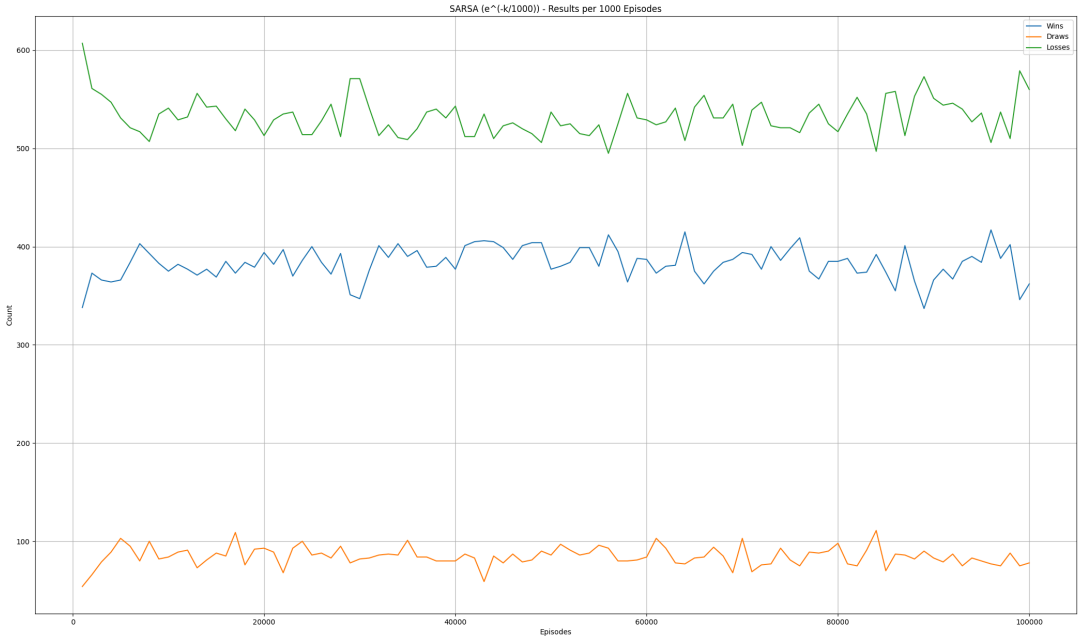


The state-action pair counts reveal the frequency of each action taken in different states. This provides insight into the learned policy and the effectiveness of the exploration strategy.

SARSA

Win-Loss-Draw Counts

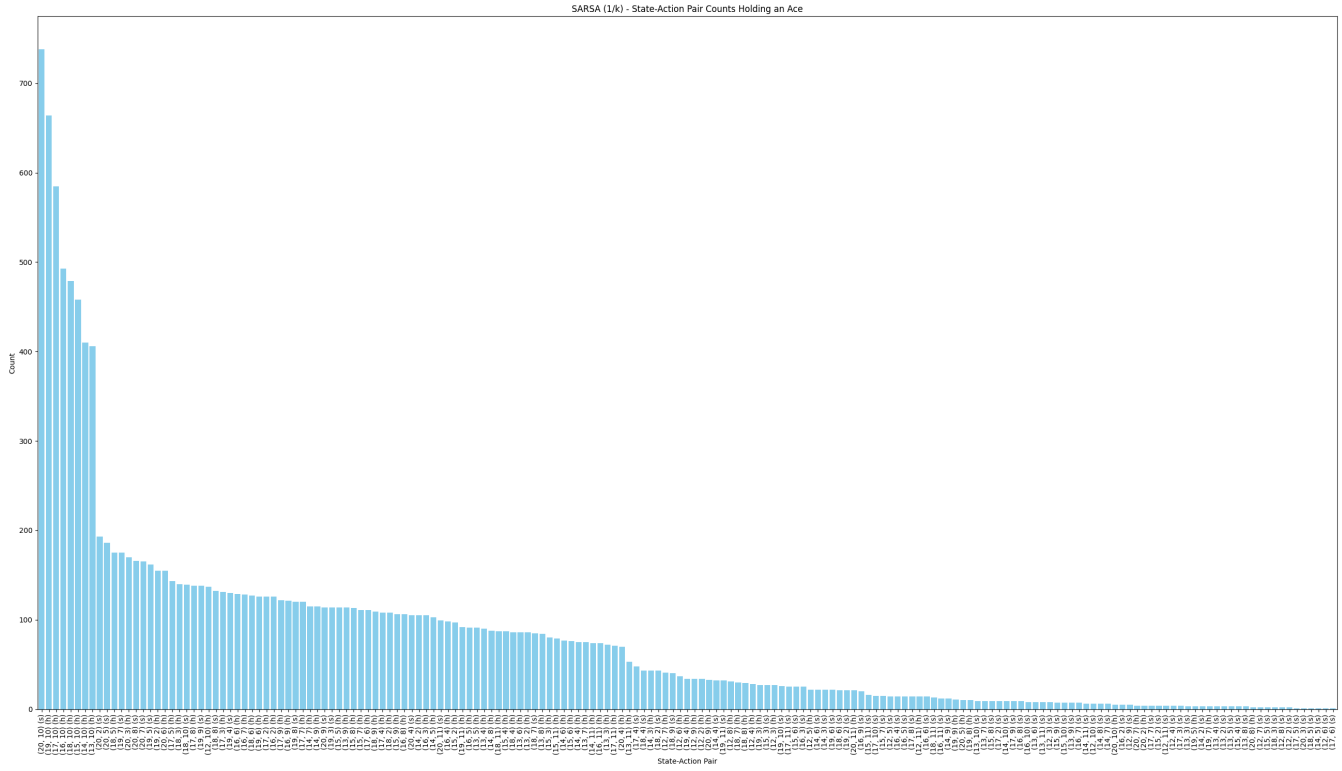


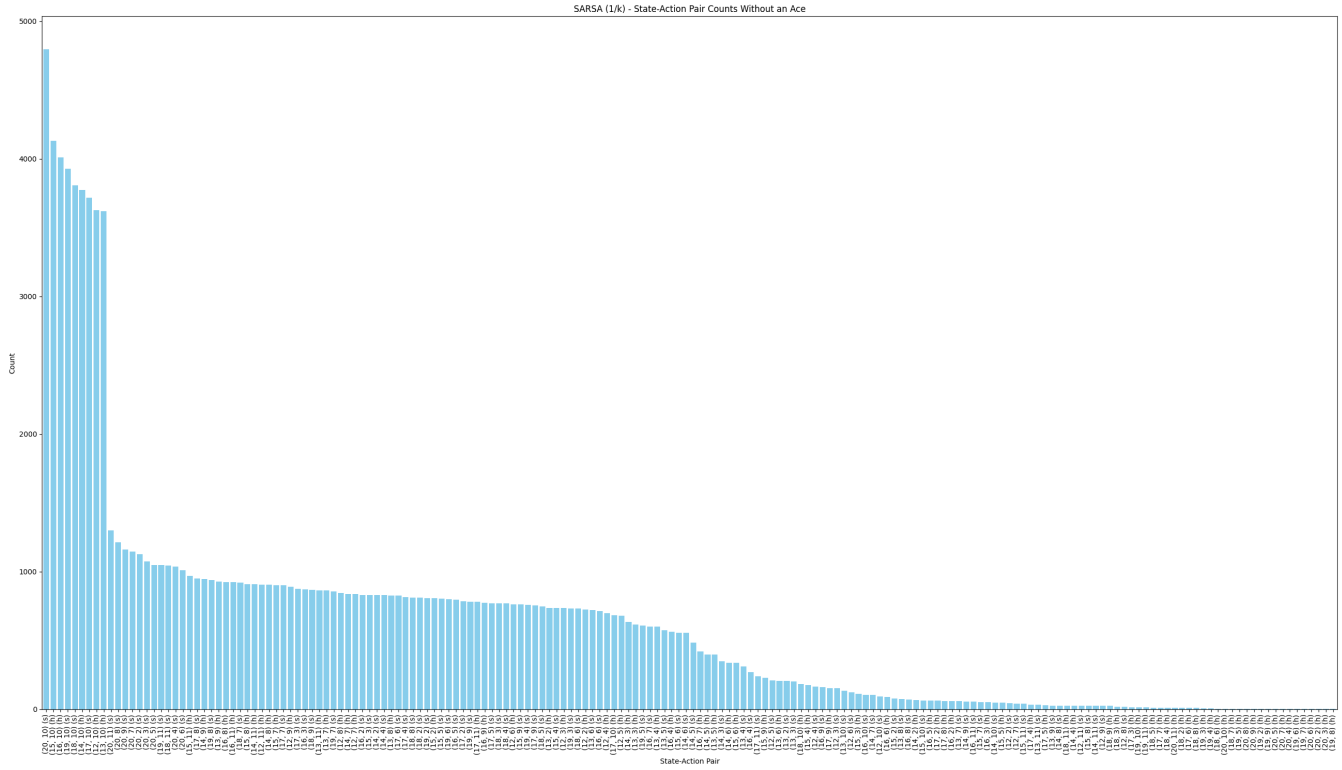


SARSA's performance was stable across all configurations. The `1/k` configuration showed a gradual improvement, while `e^(-k/1000)` and `e^(-k/10000)` converged faster. This suggests

that SARSA, being an on-policy algorithm, benefits from a balance between exploration and exploitation, which is effectively managed by the epsilon decay strategies.

State-Action Pair Counts

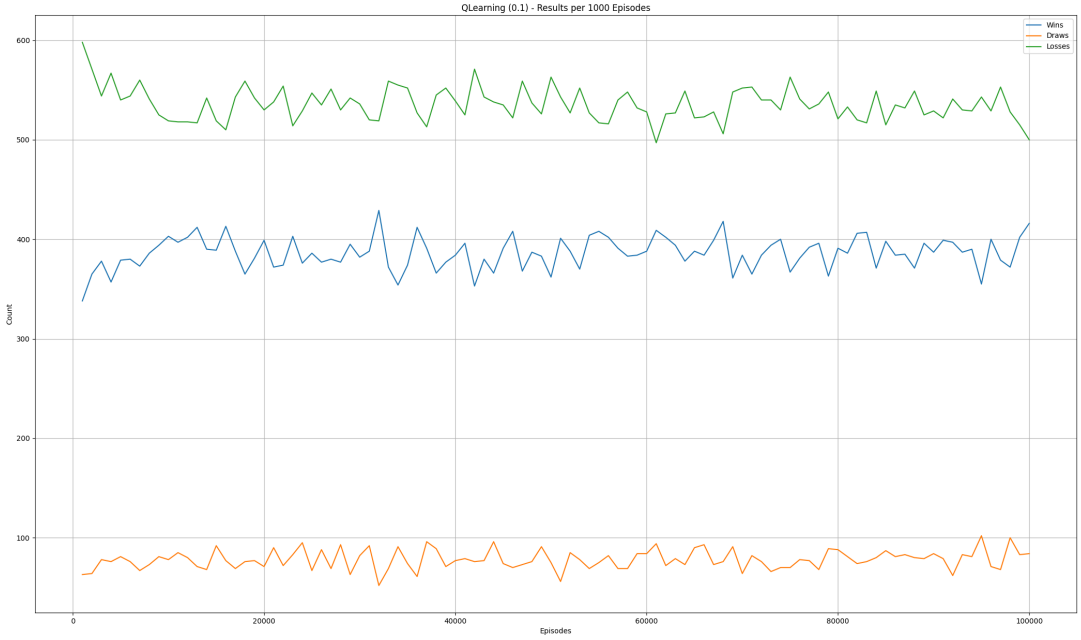


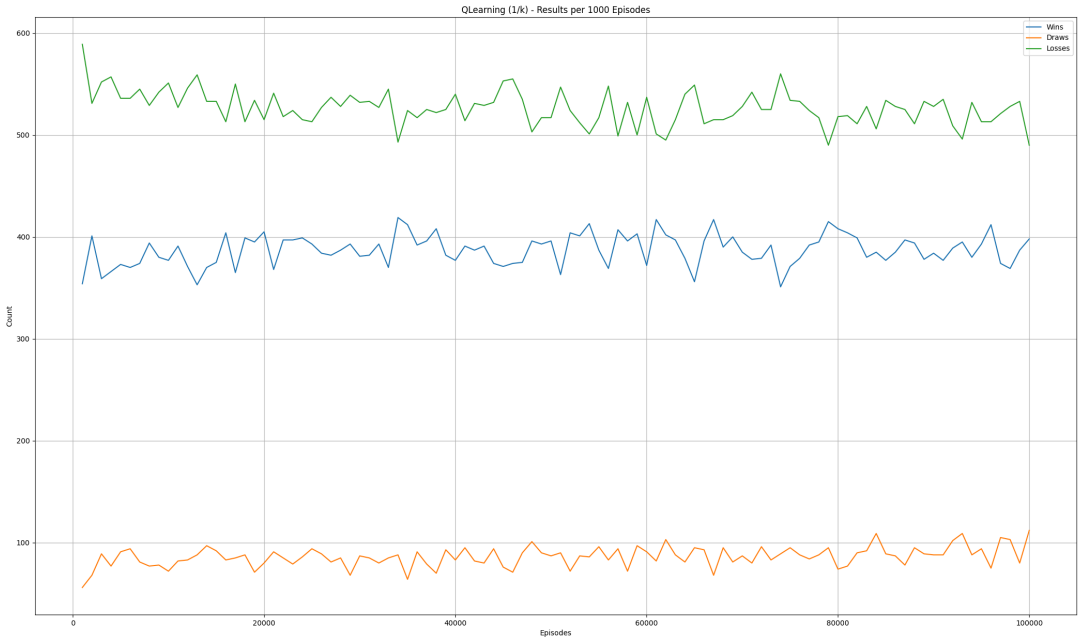


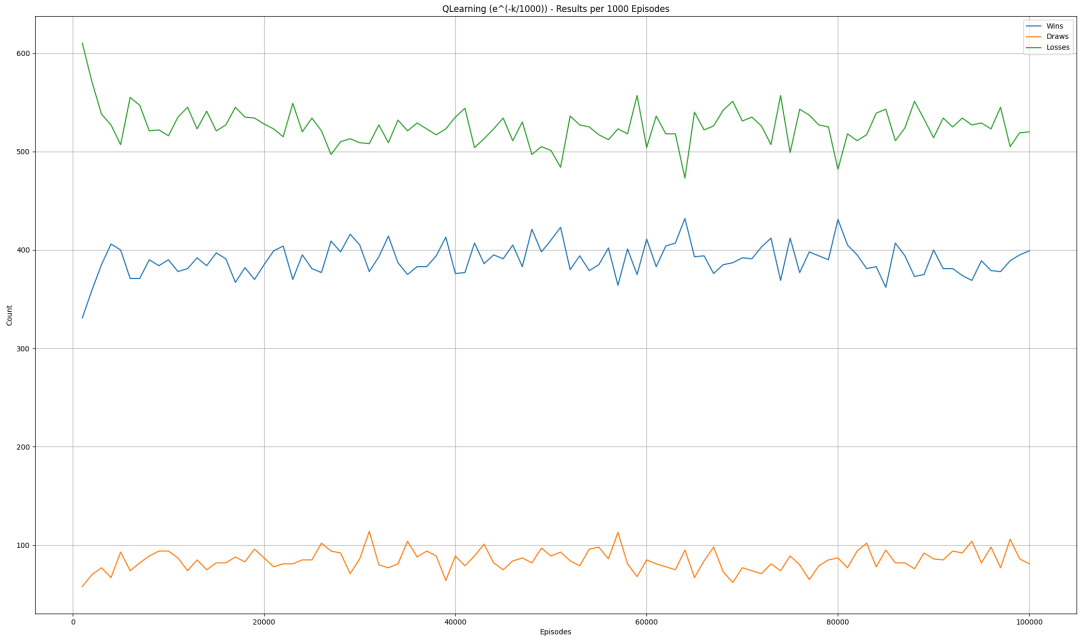
The state-action pair counts for SARSA further illustrate the balance between exploration and exploitation. The distribution of actions across states shows the learned policy's adaptability to different game scenarios.

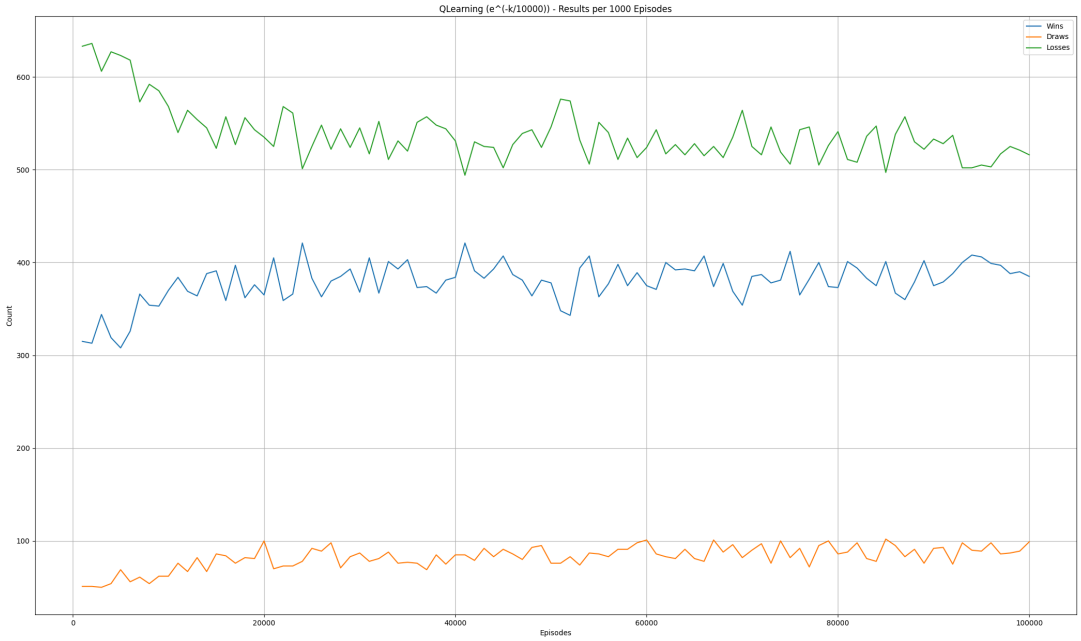
Q-Learning

Win-Loss-Draw Counts





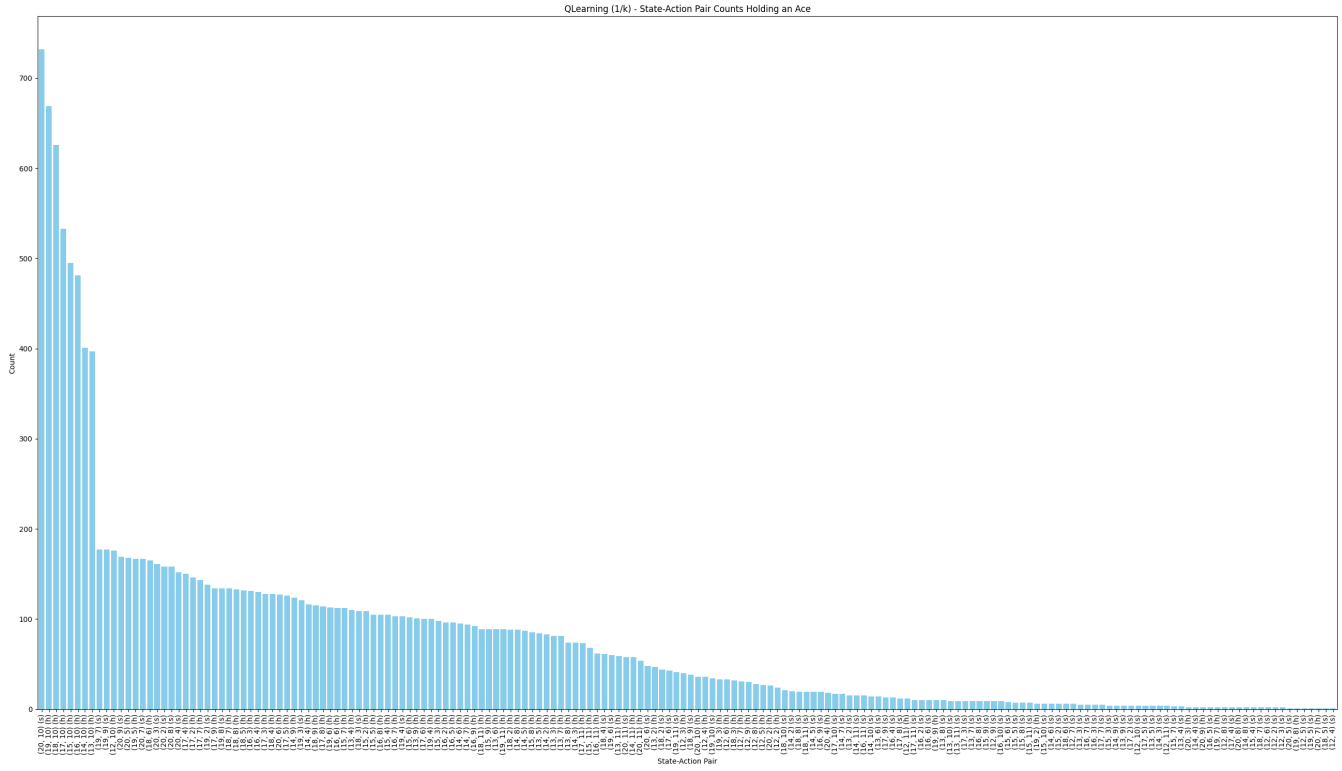


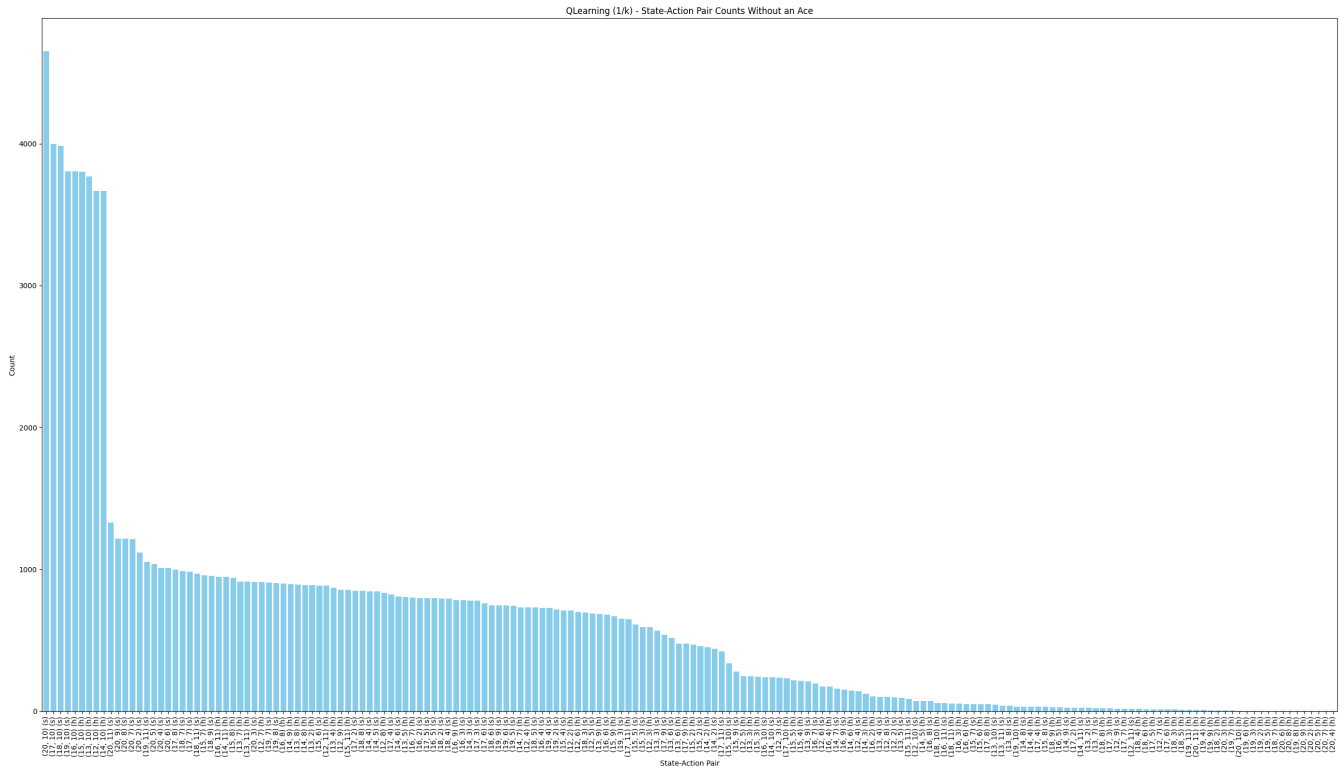


Q-learning showed the fastest convergence, particularly with the `e^(-k/1000)` configuration. The `1/k` configuration, while effective, was slower compared to exponential decay strategies.

This rapid convergence can be attributed to Q-learning's off-policy nature, which allows it to learn from the maximum Q-value of the next state-action pair, irrespective of the action actually taken.

State-Action Pair Counts



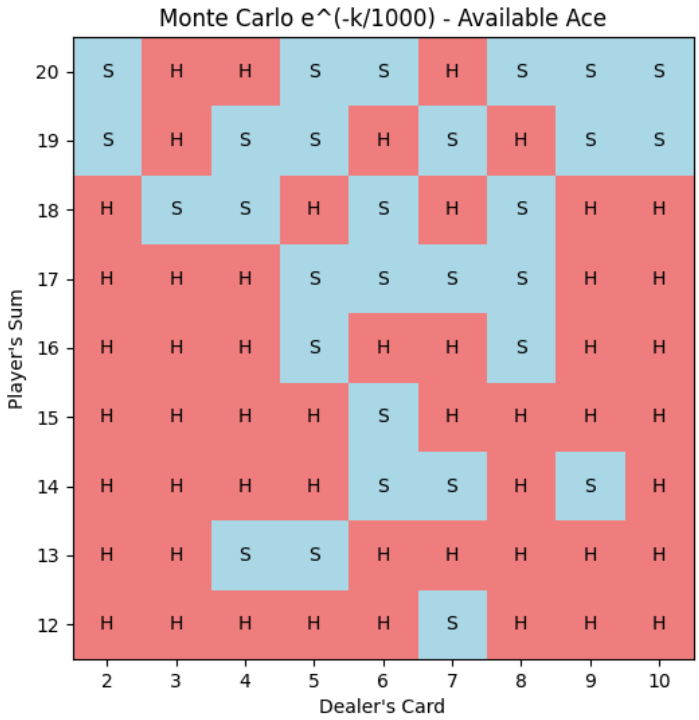


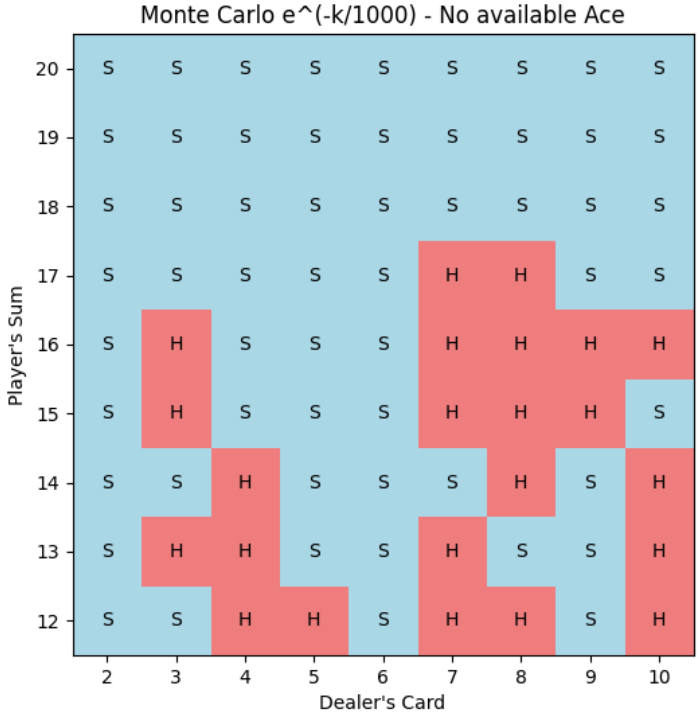
The state-action pair counts for Q-learning highlight the aggressive nature of its learning strategy. The distribution of actions across states shows a strong bias towards certain actions, indicating a high degree of exploitation.

Strategy Tables

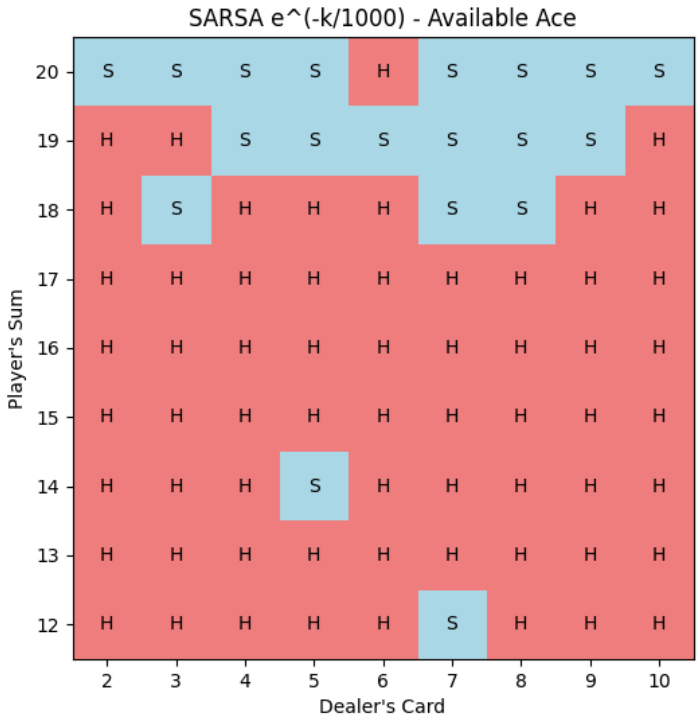
The strategy tables provide a visual representation of the optimal actions in different states as learned by each algorithm. These tables serve as a valuable tool for understanding the learned policies and their implications on the game strategy. The ones shown below use the same epsilon configuration.

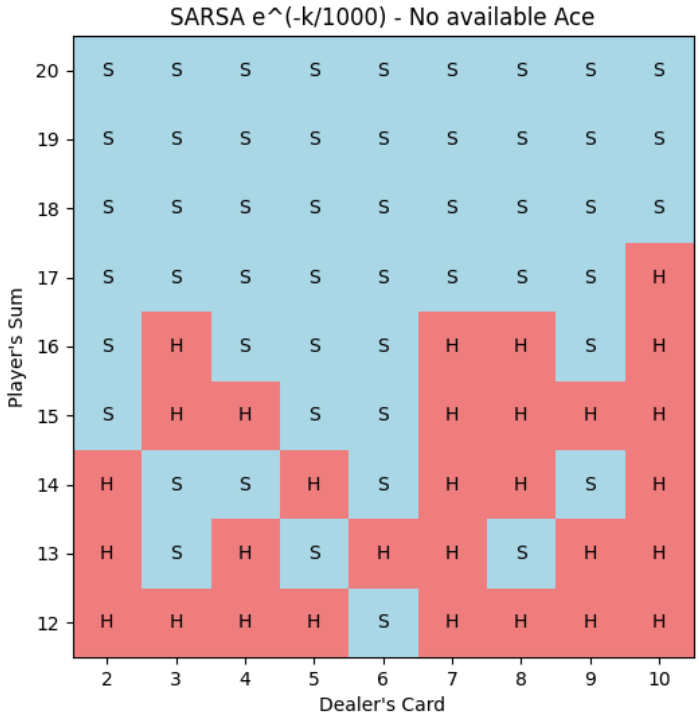
Monte Carlo Strategy Table



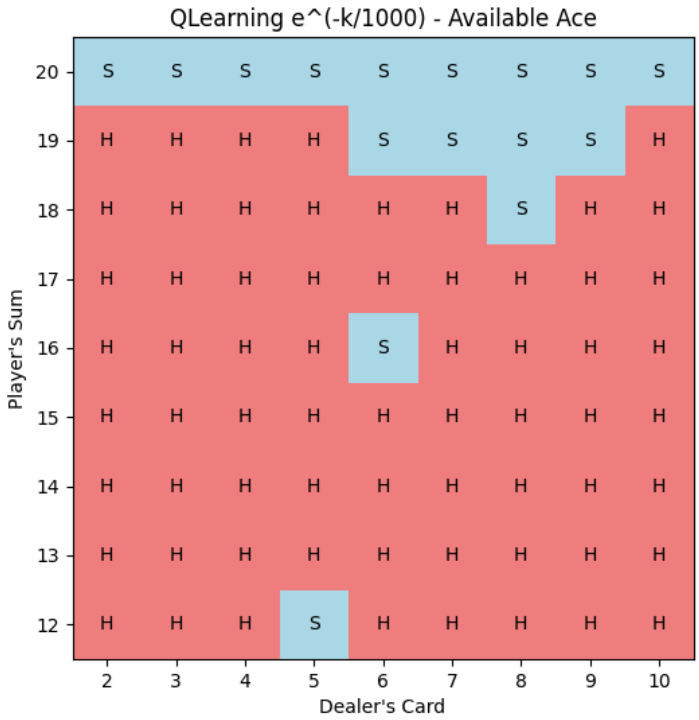


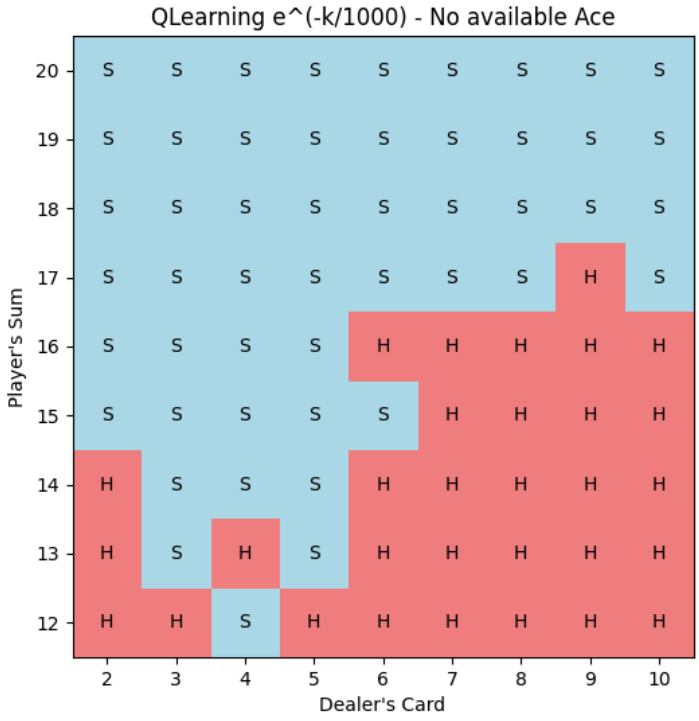
SARSA Strategy Table





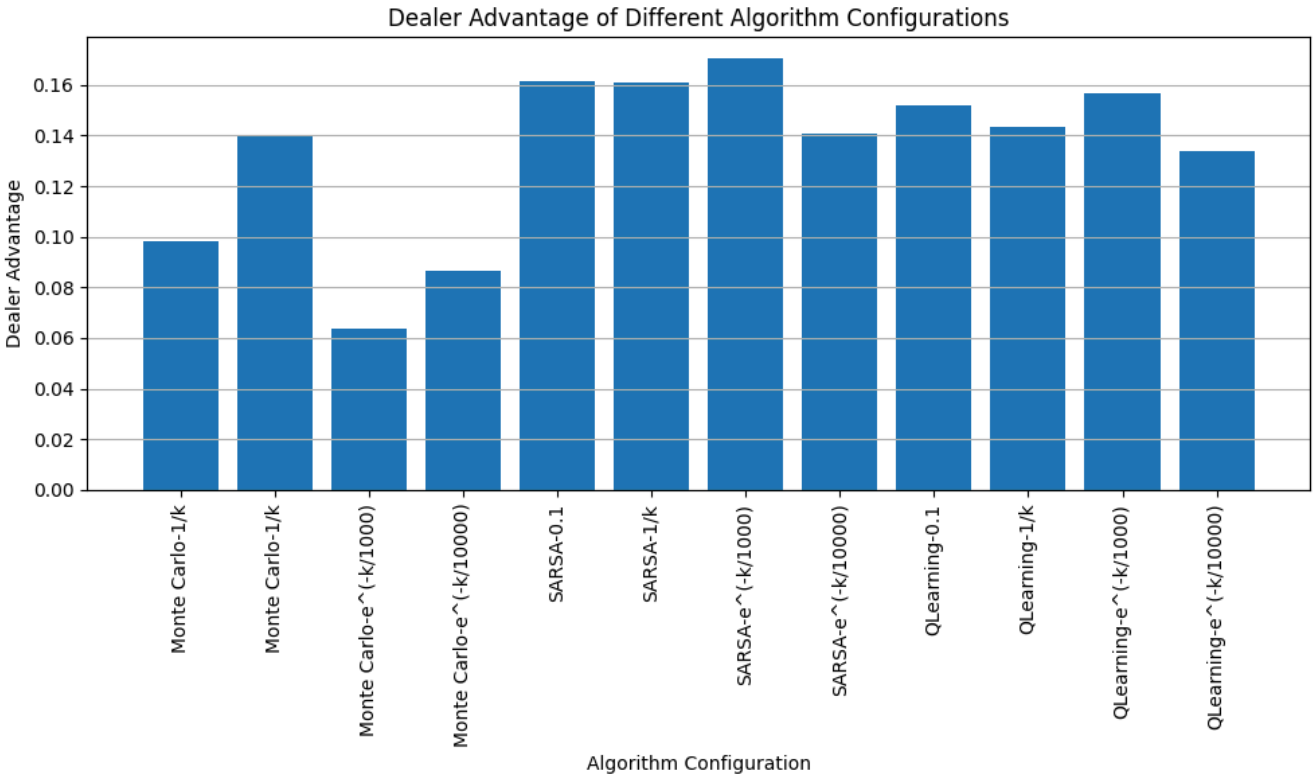
Q-Learning Strategy Table





You can access the complete set of graphs and strategy tables in the appendix file.

Discussion



The above plot is the calculated dealer advantage for each algorithm configuration. On average, Monte Carlo learns to minimize the dealer advantage best, especially when more states are explored during early exploration.

Exploration vs. Exploitation

The balance between exploration and exploitation significantly impacted each algorithm's performance. Exponential decay (`e^(-k/1000)` and `e^(-k/10000)`) allowed for faster convergence by reducing exploration more gradually compared to the `1/k` method. This resulted in slower stabilization of policies and might risk inoptimal exploitation until the algorithm converges.

In terms of algorithm comparison, Monte Carlo, being suitable for problems where episodes can be fully simulated, showed effective but slow convergence. SARSA, being an on-policy algorithm, demonstrated reliable and stable learning with moderate convergence speed. Q-

Learning, being an off-policy algorithm showed fast convergence, even with the use of exponential decay strategies, but can be less stable.

Algorithm Comparison

- **Monte Carlo**: Suitable for problems where episodes can be fully simulated. Effective but slow convergence.
- **SARSA**: Reliable and stable learning with moderate convergence speed. Suitable for on-policy learning.
- **Q-Learning**: Fast convergence, even with exponential decay strategies, but can be less stable. Best for off-policy learning.

Conclusion

This study demonstrates the application of Monte Carlo, SARSA, and Q-learning algorithms in Blackjack. Each algorithm's configuration significantly affects its learning performance. Exponential decay exploration strategies generally lead to slower convergence but enable better exploration of the search space. Future work can explore combining these methods or using more advanced techniques like Deep Q-Learning for further improvements. The results of this study provide valuable insights into the dynamics of reinforcement learning algorithms and their application in complex environments like Blackjack.

References

1. Geiser, J., & Hasseler, T. (n.d.). **Beating Blackjack - A Reinforcement Learning Approach**. Stanford University. This research seeks to develop various learning algorithms for Blackjack play and to verify common strategic approaches to the game. He present these implementation of Sarsa, and Q-Learning.
2. Avish Buramdoyal, Tim Gebbie. (2023). **Variations on the Reinforcement Learning performance of Blackjack**. *arXiv:2308.07329 [cs.AI]*. Retrieved from arXiv:2308.07329

Distribution of Work

- Florent Cadet :
    - Creation of the code project structure
    - Creation of `Card` and `Deck` classes and implementation of card logic
    - Implementation of the Blackjack game, with its rules and game logic
    - Implementation of basic RL scripts such as `PlayerRLAgent` and `BlackJackWithRL`.
    - Writing project reports and documentation
- Federico Oliveri:
    - Implementation of `MonteCarloOnPolicyControl` class
    - Implementation of `SARSAOnPolicyControl` class
    - Implementation of `QLearningOffPolicyControl` class
    - Implementation of the plotting functions

Signature

Florent Cadet, Federico Oliveri