L-Università ta' Malta
Faculty of Information & Communication Technology

Department of Computer Information Systems

Reinforcement Learning Project

Cristina Cutajar* (230802L), Britney Vella* (188102L), Nicole Shaw* (66502L)

*B.Sc. (Hons) Artificial Intelligence

Study-unit: **Reinforcement Learning**

Code: **ARI2204**

Lecturer: **Dr Josef Bajada**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Cristina Cutajar	
Student Name	Signature
Britney Vella	
Student Name	Signature
Nicole Shaw	
Student Name	Signature
Student Name	Signature

ARI2204	Reinforcement Learning Project
Course Code	Title of work submitted

19/05/2022

Date

Implementation of Algorithms

In order to implement and execute the algorithms, we implemented the policy. Whenever a new environment is created, the Game class initialises an Agent state to be used by the policy. In the start_game() function, a check function from the Agent class is called to check if the player has an ace. If the player has an ace, a boolean flag will be set to True. However if the sum is greater than 21, the sum will be decreased by 10 and the boolean flag will be set to False. The new player sum and boolean flag will be returned. Then, the policy will choose hit if the player sum is less than 12, stand if the player sum is greater than 20, or execute either the Monte Carlo On-Policy Control Algorithm or SARSA On-Policy Control Algorithm if the player sum is between 12 and 20.

If Q-Learning (SARSAMAX) Off-Policy Control Algorithm is chosen, it will be executed no matter what the player sum is.

Afterwards, the state and corresponding chosen action will be appended to an array which is then returned at the end of the episode along with the reward of the episode (1, 0, -1) in order to calculate the Q(s,a).

In the main function, two dictionaries will be initialised both containing the player_sum, True or False (depending if ace is being used as 11) and dealer sum. For Monte Carlo and SARSA, the player_sum value will be between 12 and 20, whereas for Q-Learning, it will be between 1 and 21. Each state in these two dictionaries will have 2 values, one for hit and one for stand which will all be initialised to 0. The Q dictionary will store the Q(s,a) value whilst the N dictionary will store the N(s,a) value.

Monte Carlo On-Policy Control Algorithm

The Monte Carlo On-Policy Control Algorithm was implemented in the start_game() function of the Game class to execute if the player sum is between 12 and 20. Depending on whether the exploring_starts variable was set to True or False, the algorithm will execute differently.

If the exploring_starts variable is set to True, a random action is always chosen for the first state of the episode. For the other states, epsilon will be set to 1/num_of_episodes which includes the current episode. Otherwise, if the exploring_starts variable is set to False, the e_config variable is checked. If e_config was set to 1, then epsilon will be set to 1/num_of_episodes. If e_config was set to 2, then epsilon will be set to e^(-num_of_episodes/1000) and if e_config was set to 3, then epsilon will be set to e^(-num_of_episodes/10000).

Afterwards, a random number between 0 and 1 will be generated. If the random number is smaller than epsilon, then the action will be randomly chosen. Otherwise, the ϵ-Greedy algorithm will be executed which will loop through the Q(s,a) values and finds the corresponding state and chooses the action with the highest value for that state. If both values for Hit and Stand are equal, then the action will be randomly chosen.

After each episode, the algorithm uses the outcome value and game_actions array to calculate the Q(s,a). It will loop through the game_actions array and through the keys and values of the Q and N dictionaries and for each state-action pair in the game_actions array, the value in the N dictionary will be increased by 1 and then the value in the Q dictionary will be updated using Q(s,a) = Q(s,a) + (1/N(s,a))*(G-Q(s,a)) where G is the outcome/reward value.

SARSA On-Policy Control Algorithm

The SARSA On-Policy Control Algorithm was implemented in the start_game() function of the Game class. If the value of the players hand is between 12 and 20 then the ϵ-Greedy policy will run based on the numeral configuration that was entered in the main function. Entering 1 would set the epsilon to the first configuration where it will execute with ϵ = 0.1. If 2 was entered in the configuration slot, the program will run with $\epsilon = 1/k$. Additionally, the value 3 will execute with $\epsilon = e^{-k}/1000$ and the value 4 will execute with $\epsilon = e^{-k}/10000$.

After the epsilon is calculated with the given ϵ, a random number is generated to compare the two. If the random number is smaller than epsilon then another random number will be generated of either 1 or 2 to choose which action to take. However, if the epsilon is smaller than the random number then it will loop through the Q(s,a) values to choose whether to hit or stand based on that value. If the hit and stand values are equal then the action taken is chosen randomly.

The main program is where the Q dictionary is updated with the state-action pairs and the outcome of the game. The dictionary N is used to calculate the alpha value using the equation $\alpha = 1/N(s,a) + 1$. The number of times the state-action pair was encountered including in the current episode is found within that dictionary. In addition, the Q(s,a) is calculated and appended to the Q dictionary using the equation Q(s,a) = Q(s,a) + α* [R + Q(s', a') - Q(s,a)]. Here, the terminating Q is set as 0 since the Q value of a terminal state is always 0.

Q-Learning (SARSAMAX) Off-Policy Control Algorithm

The Q-Learning Off- Policy Control Algorithm was also implemented within the start_game() function of the Game class like the previous algorithms. However, since this is an Off-Policy algorithm it always chooses the actions based on the ϵ-Greedy policy.

Similar to the SARSA implementation, the ϵ-Greedy policy will run based on the numeral configuration that was entered in the main function when starting the game. The values for the ϵ-Greedy policy that are inputted in the start_game() function are 1, 2, 3 and 4. These numbers will execute with the ϵ value of 0.1, $1/k$, $e^{-k}/1000$ and $e^{-k}/10000$ respectively.

The epsilon is then calculated and a random number is generated to compare. If epsilon value is larger then the action taken will be randomised. Otherwise, it will go through the Q dictionary to take the corresponding state-action pairs' Q(s,a) value into consideration to be

able to choose which action to take. In the event that the hit and stand values are equal, then the action is chosen randomly.

The main program initialises the game and updates the Q and N dictionaries which are used to calculate the Q(s,a) values. The N dictionary is used to get the value of alpha α = 1/$N(s,a)$+1 and also the maxa.Q(s',a) value. This is done through a check that finds the corresponding state-action pair and takes the Q(s,a) value of the maximum action value of that pair. That value is then used to calculate the Q(s,a). The Q dictionary stores the state-action pairs, results of the episodes along with the computed Q(s,a) values according to the equation Q(s,a) = Q(s,a) + α*[R + maxa.Q(s',a) - Q(s,a)].

Plot Results

For the plot results of the SARSA implementation, the total episode count of 100,000 was used since having a higher number was resulting in the Jupyter Notebook crashing and took a very long time. The Monte Carlo and Q-Learning Implementations, were run for a total episode count of 20,000 for the same reasons. Monte Carlo was also executed for a total episode count of 100,000 for the first configuration but took a total of 2 hours and 20 minutes to execute and hence all the Monte Carlo configurations were then run for 20,000 episodes.

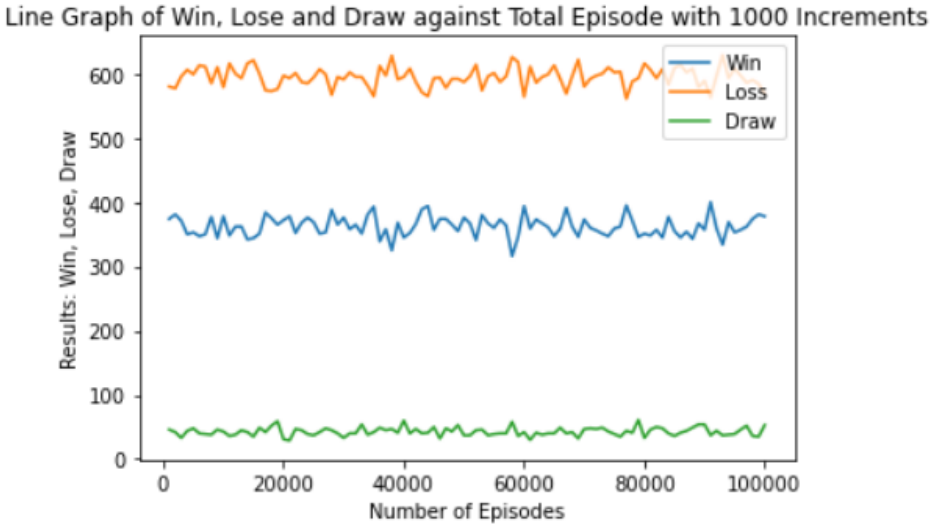
The first line graphs show the number of times the agent won, lost or tied against the number of episodes played with increments of 1000 episodes. The second line graph is a different implementation of what we had to do for the first graph that shows a line representing the total amount of wins, losses and ties against the total amount of episodes it executed. The third graph, the bar chart, represents all of the unique state-action pairs against the number of times they appeared in the total episodes.

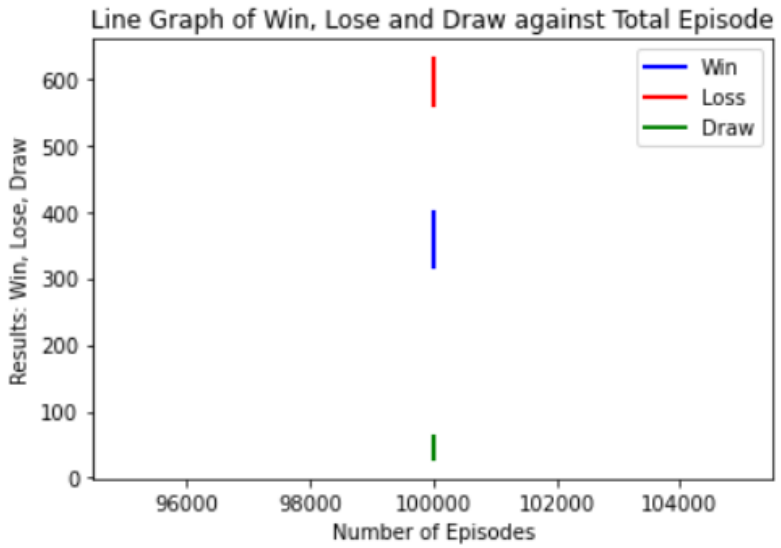
Finally, the bar graph represents the total unique state-action pairs that appeared during the runs of the four configurations and the histogram chart shows the dealer's advantage in all four configurations.

Monte Carlo On-Policy Control Algorithm

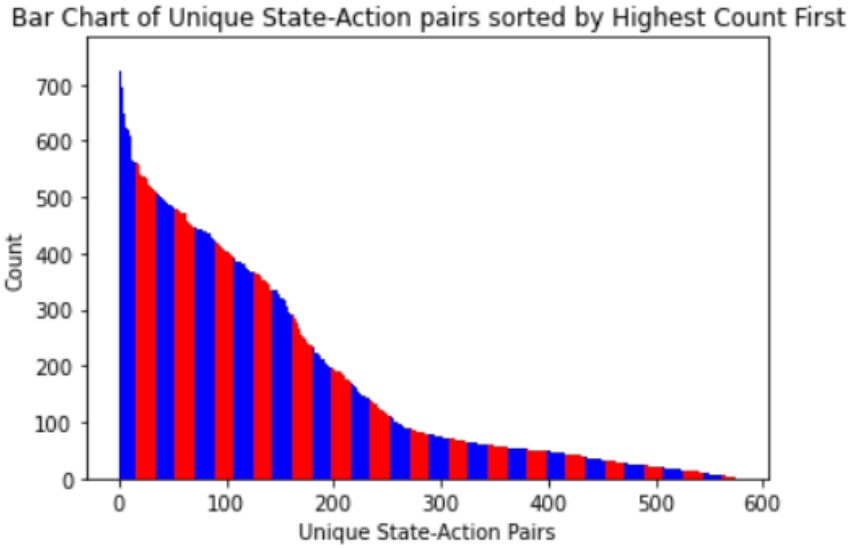
Configuration 1: Exploring Starts $\epsilon = 1/k$

For 100,000 episodes:

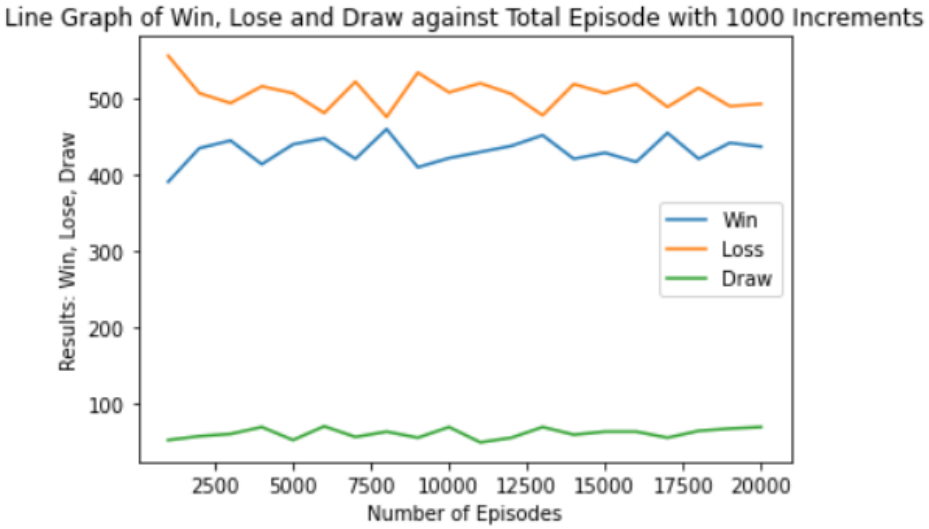


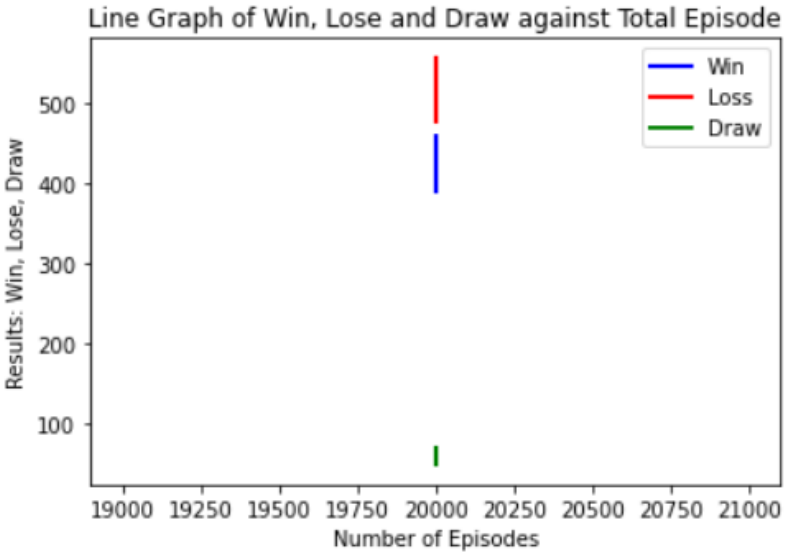


The above graphs show that the configuration resulted in the algorithm to win, lose and draw at a mildly changing rate where the algorithm is mostly losing. Unfortunately, the number of wins is much lower than the number of losses. This could be due to the Exploring Starts where exploration is happening more than exploitation. There was a small error where a random action was being assigned to not only the first state. The error was fixed before running the configurations for 20,000 episodes.

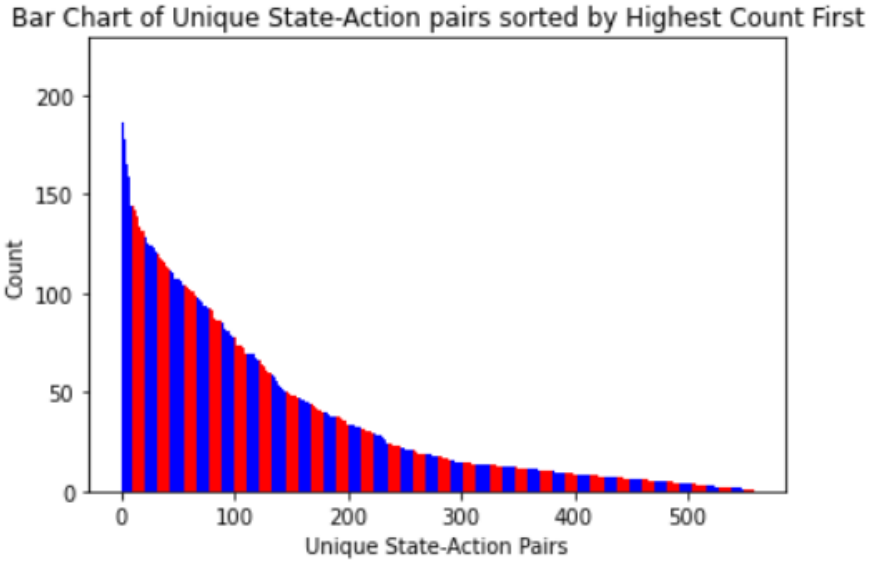


For 20,000 episodes:

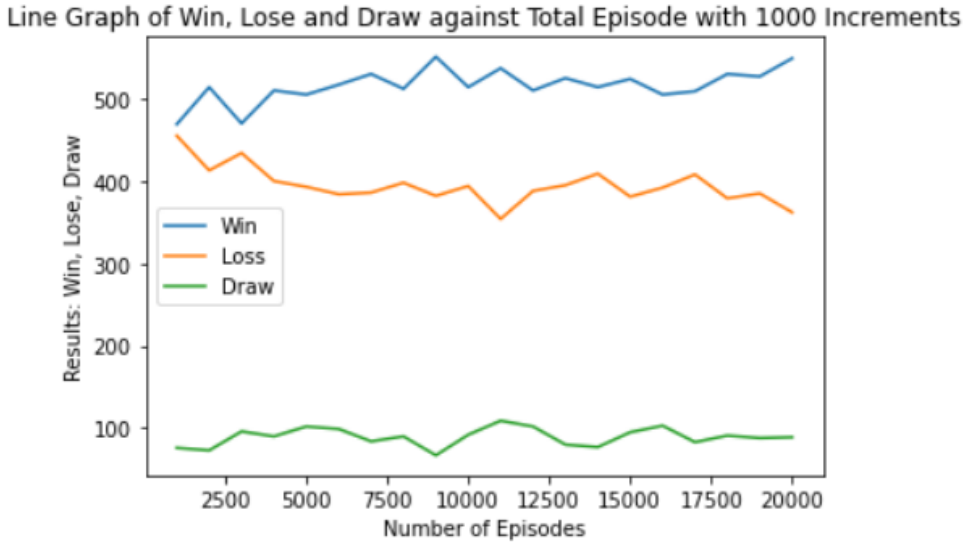


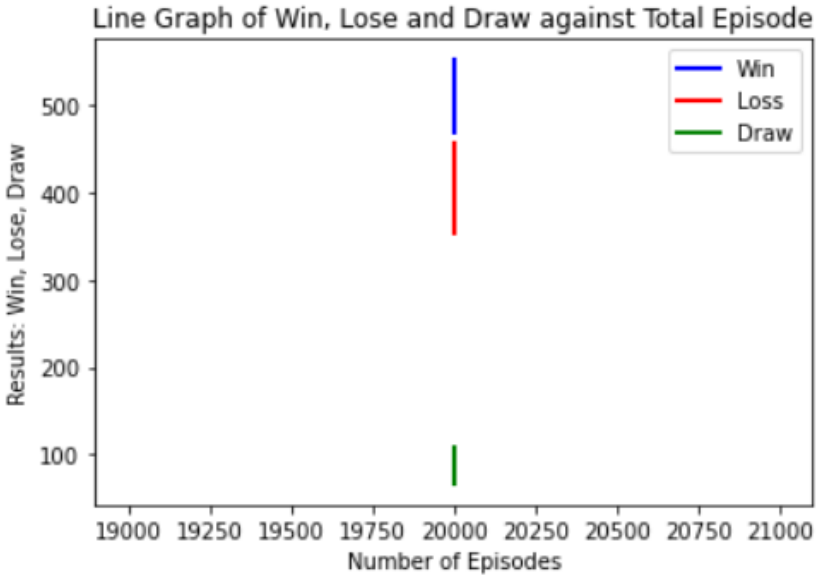


Similar to that for 100,000 episodes, they also show that the win, lose and draw are mildly changing and the algorithm is mostly losing. However the difference between wins and losses is less than that of 100,000 episodes.

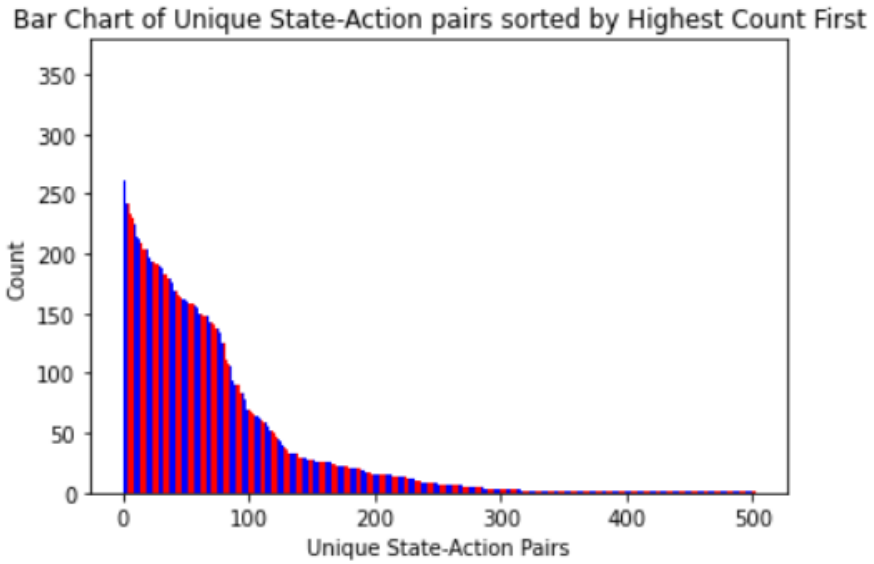

Bar Chart of Unique State-Action pairs sorted by Highest Count First
Count
200
150
100
50
0
0
100
200
300
400
500
Unique State-Action Pairs

Configuration 2: Not Exploring Starts $\epsilon = 1$

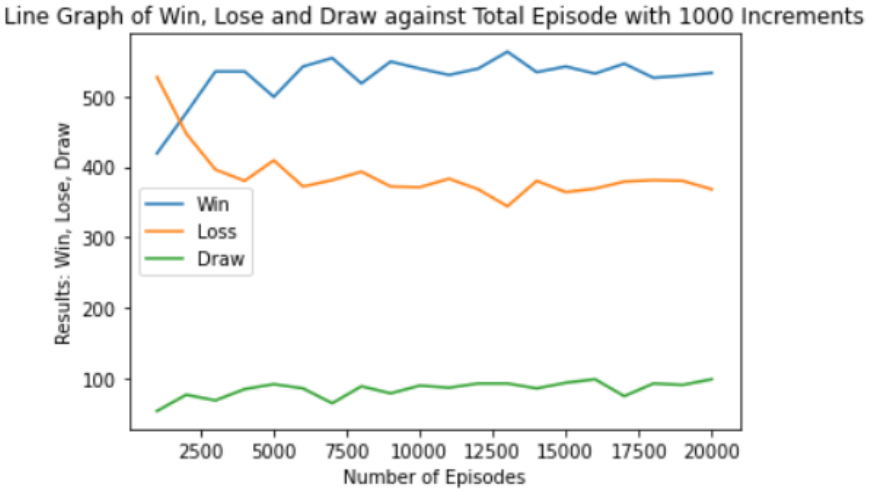


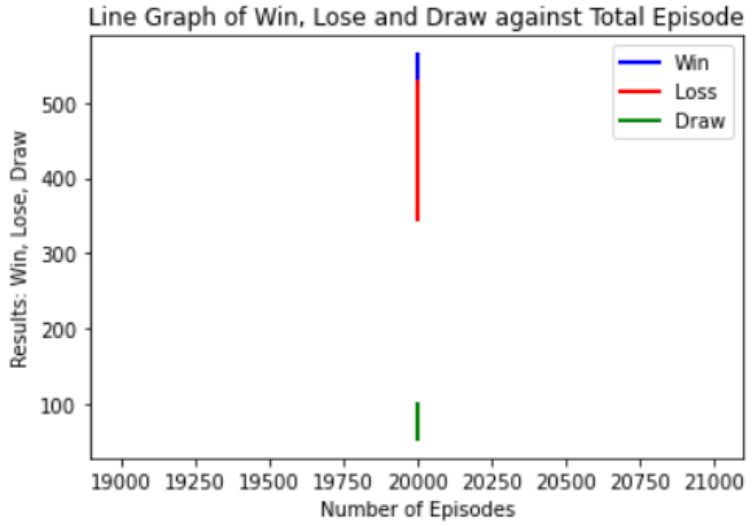


The graphs above show that unlike for the previous configuration, the number of wins was higher than the number of losses. The number of draws is very low with mild changes throughout the episodes.

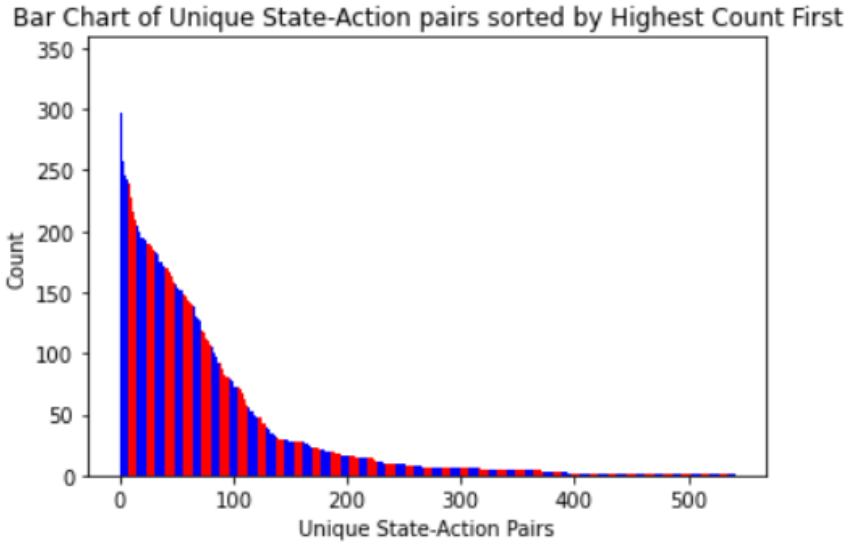


Configuration 3: Not Exploring Starts $\epsilon = e^{\wedge} -k/1000$

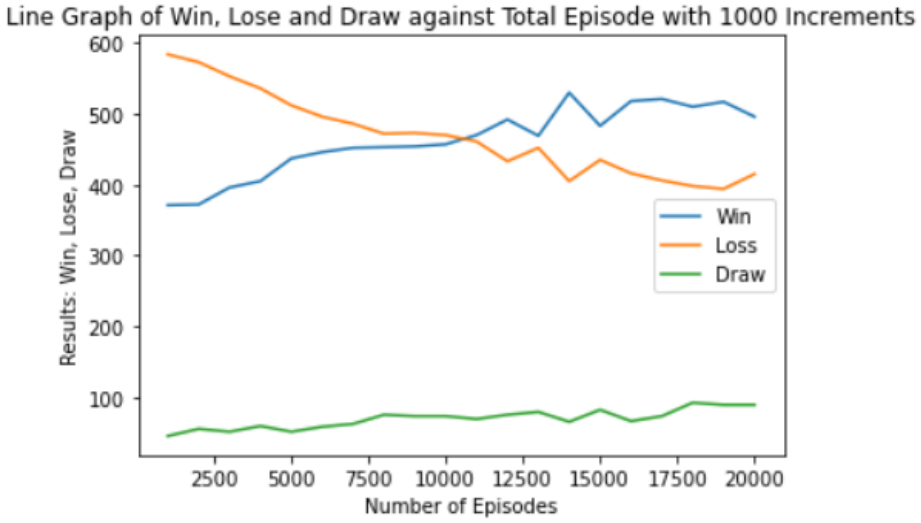


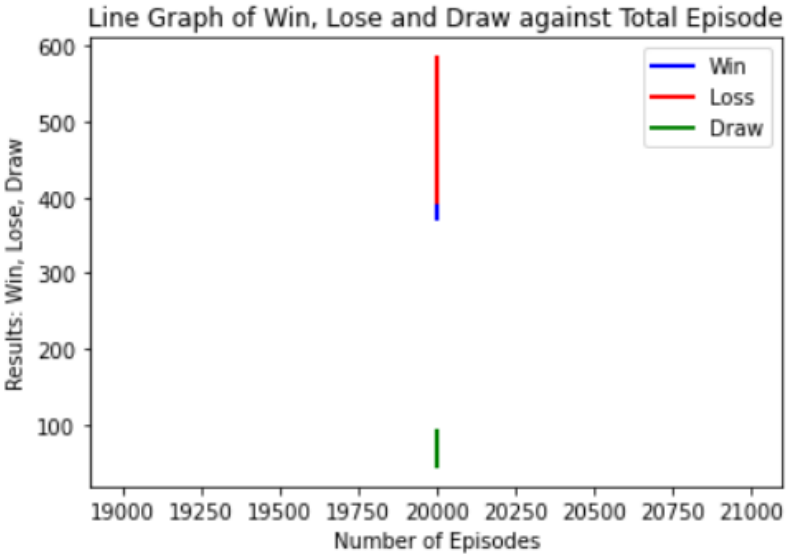


The above graphs show that at first the configuration started with more losses however then ended up resulting with more wins. The difference is less when compared to the previous configuration. The number of ties increased a bit along the episodes.

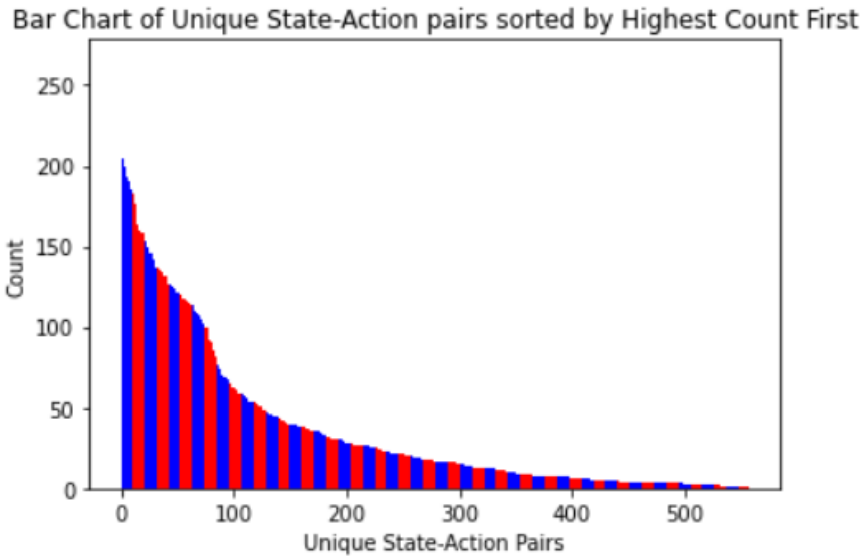


Configuration 4: Not Exploring Starts $\epsilon = e^{\wedge} -k/10000$

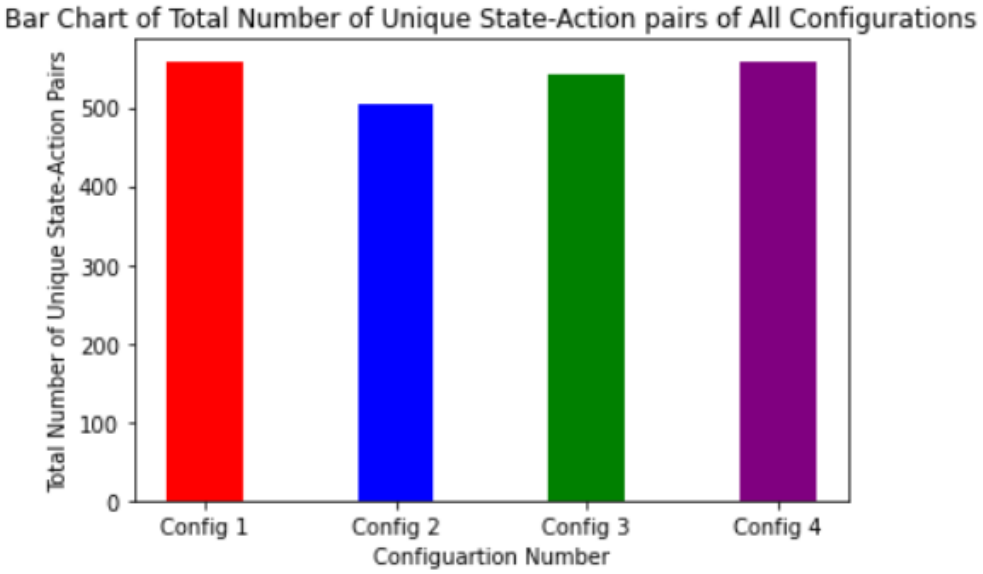




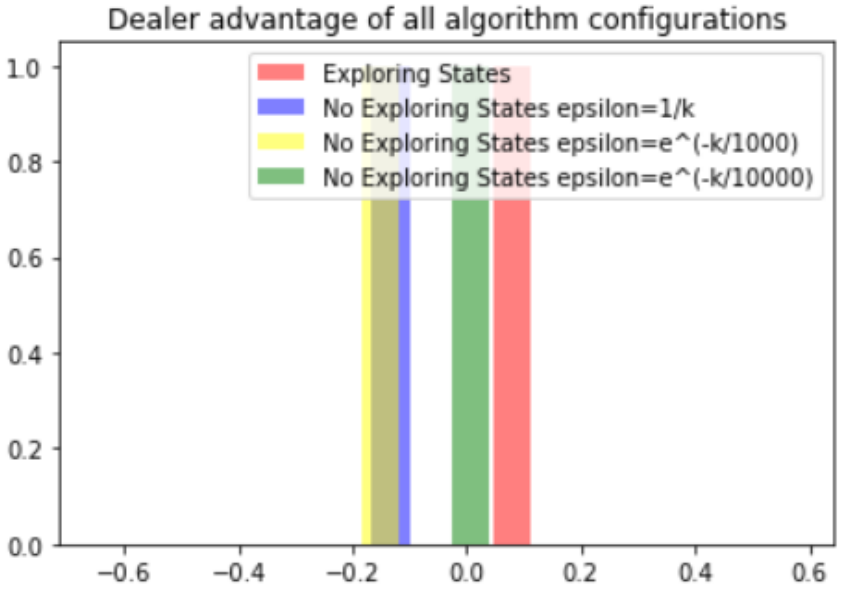
The above graphs show that in total, there were more losses than wins however after 10,000 episodes, the wins increased more than the losses. The draws kept increasing at a slight incline.



Graphs Based on All 4 Configurations



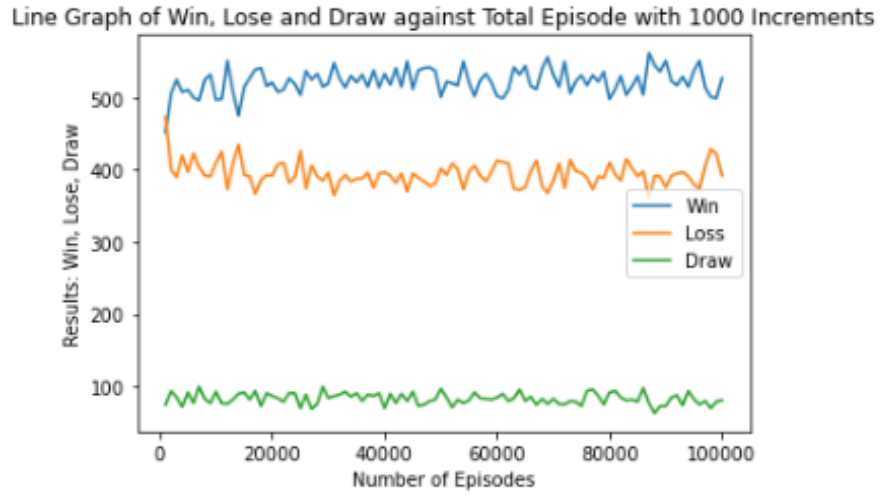
The above bar chart shows that configuration 1 had the most unique state-action pairs. Meanwhile configuration 2 had the least unique state-action pairs.

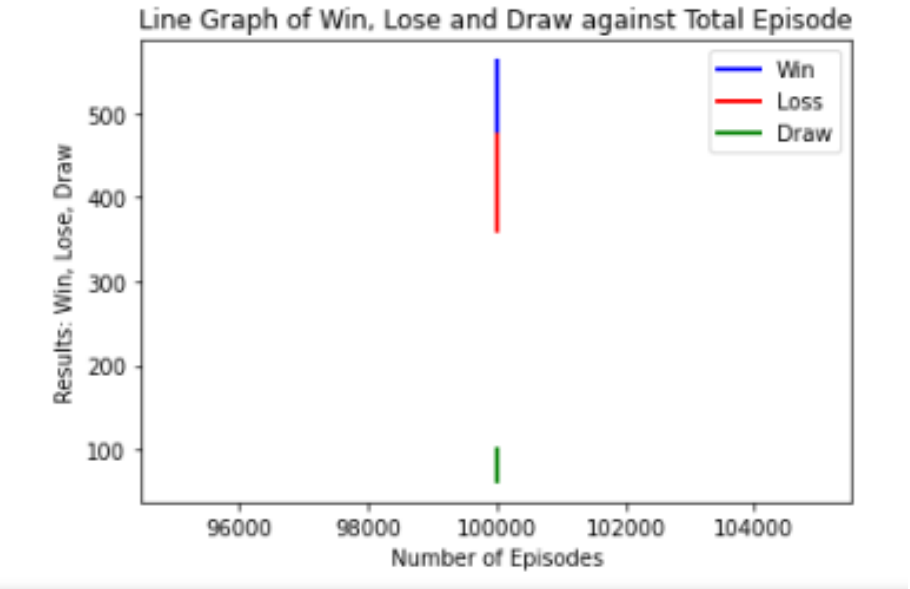


From the above histogram, one can see that the exploring starts configuration had the highest dealer advantage whilst configuration 3 had the lowest.

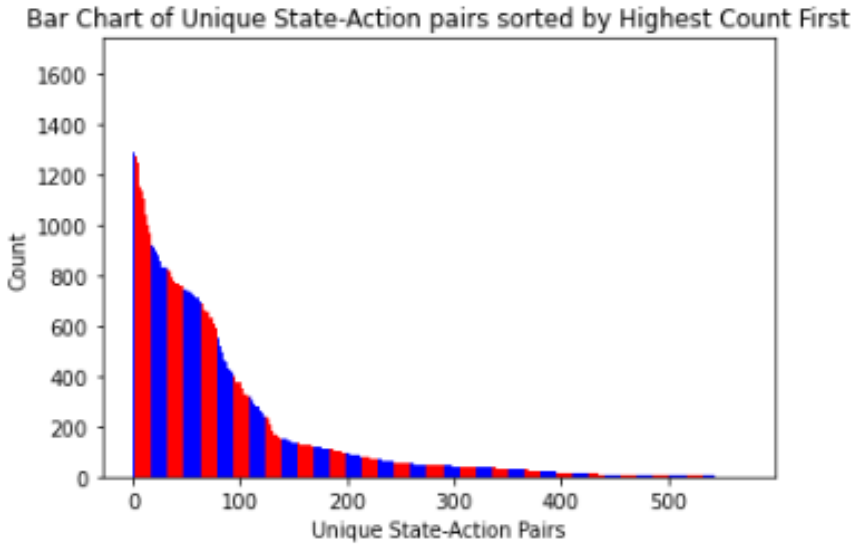
SARSA On-Policy Control Algorithm

Configuration 1: $\epsilon = 0.1$

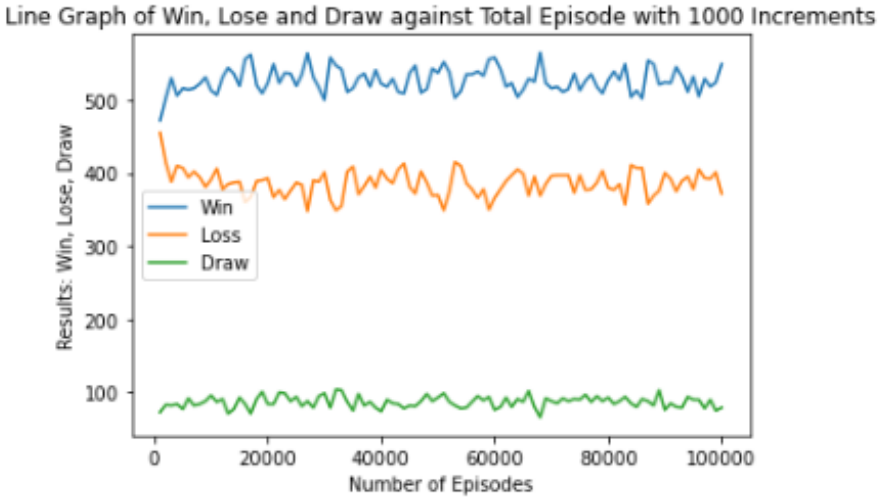


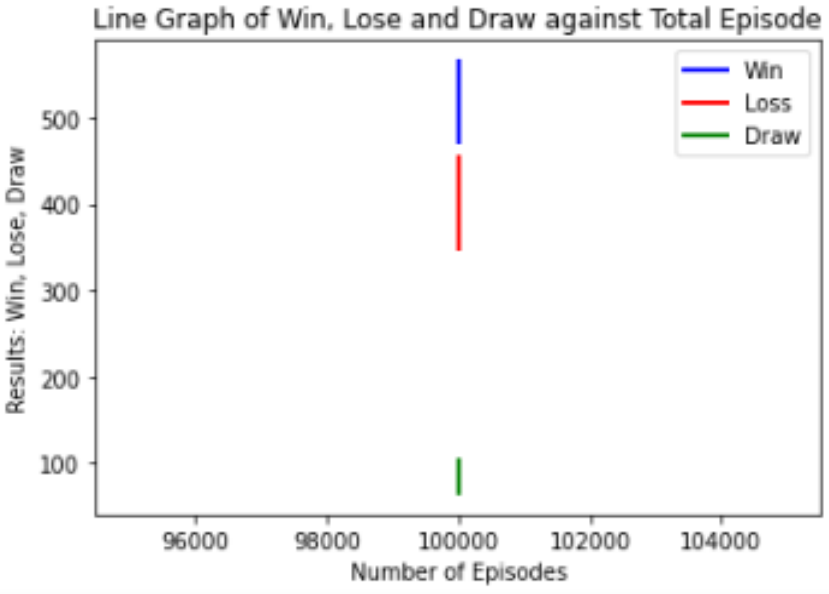


From the above charts one can see that the agent mostly won and rarely tied with the dealer. One can also note the steep decline and incline at the beginning of the loss and win lines. This shows that the agent was learning when to hit or stand correctly and improved its chances of winning.

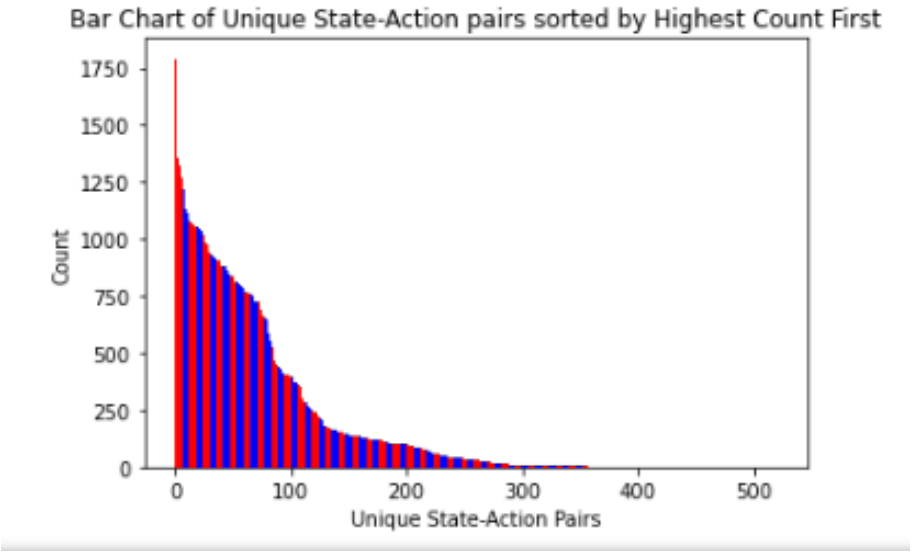


Configuration 2: $\epsilon = 1/k$

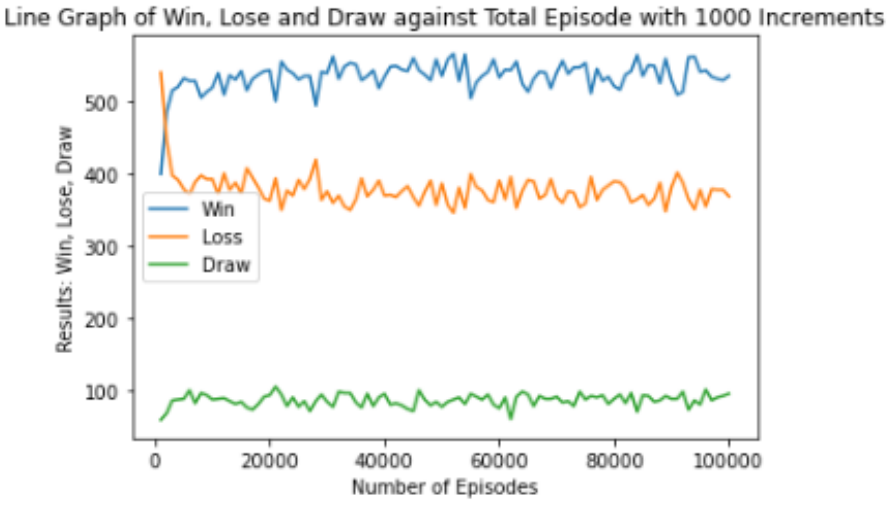


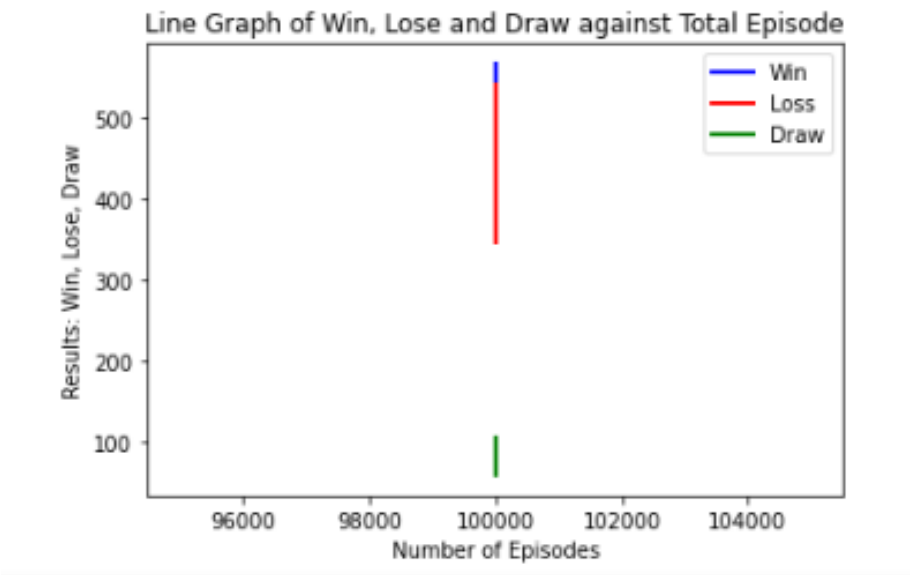


In the above graphs, it can be seen that the agent mostly won and rarely tied. There is also another steep decline and incline at the beginning of the loss and win lines. The gap between the win and lose lines proves that the win rate was much higher than the losing rate.

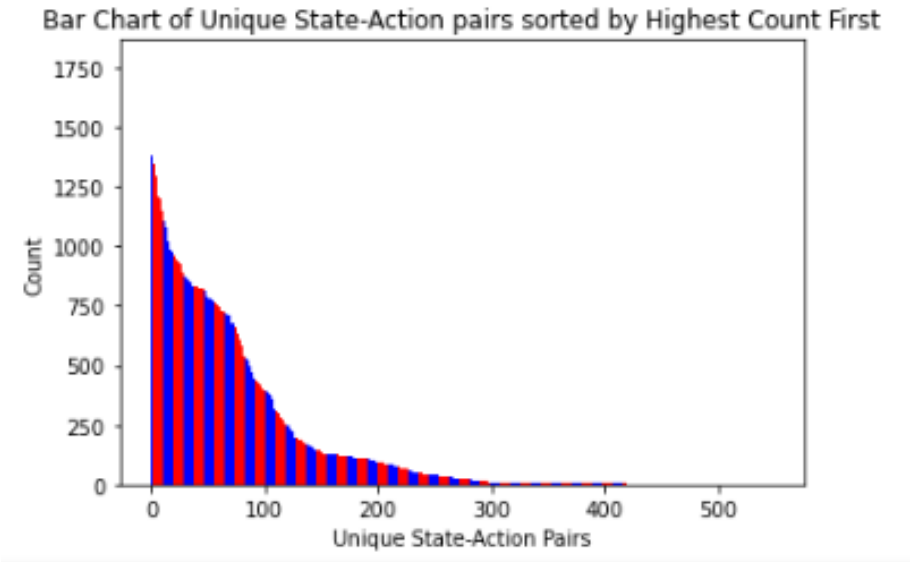


Configuration 3: $\epsilon = e^{\wedge} -k/1000$

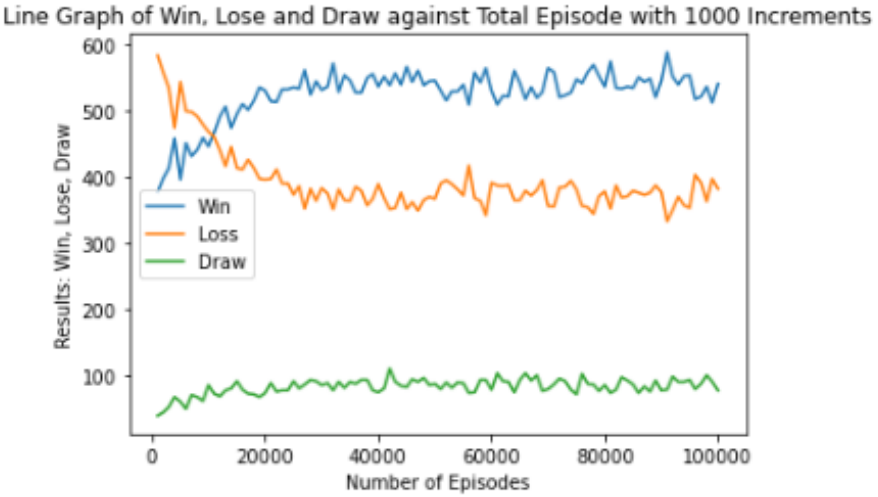


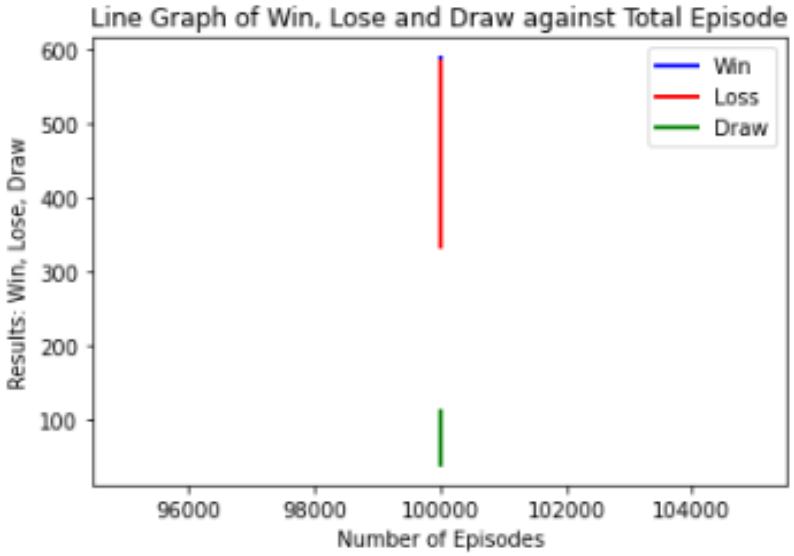


These graphs evidently show that the agent started off with a very high losing rate and low winning rate. Especially when compared to the previous results. However, there is a much steeper change in both the win and lose lines showing that the agent quickly learned and started to win more. It can also be stated that the draw rate started off low and increased slightly as the losing rate was declining.

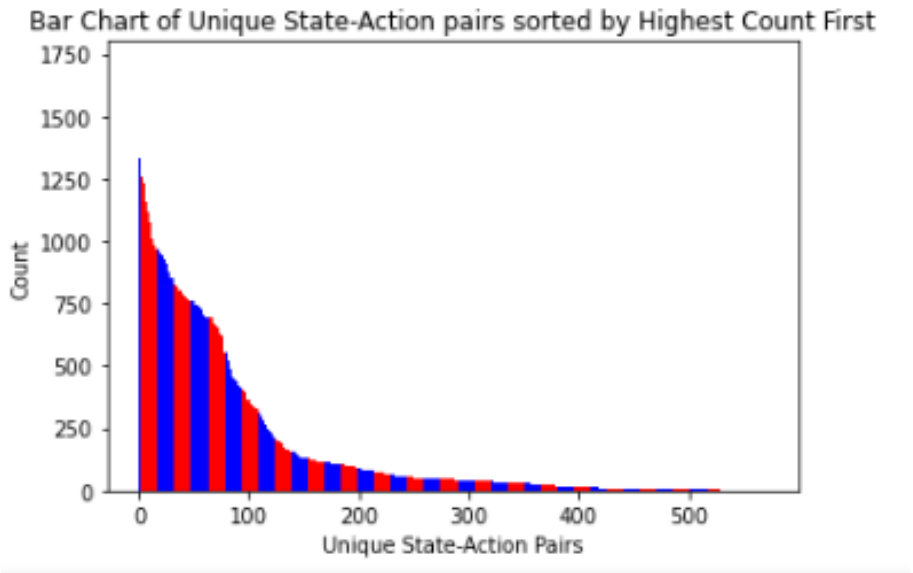


Configuration 4: $\epsilon = e^{\wedge} -k/10000$

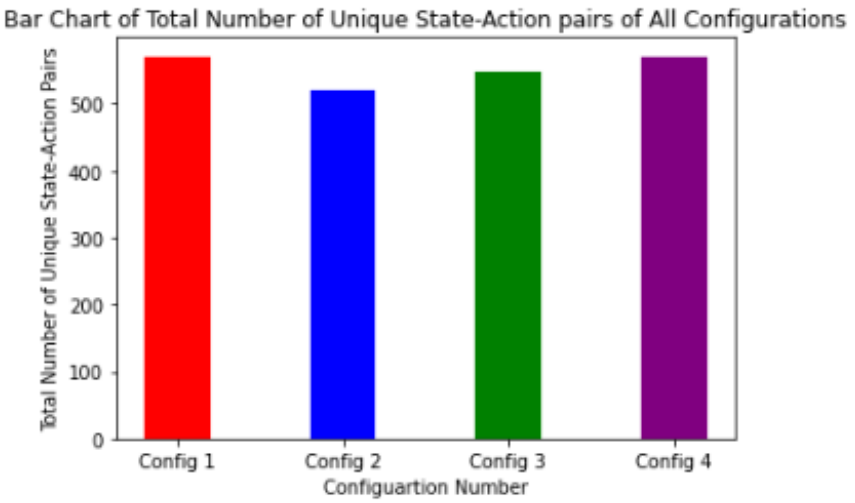




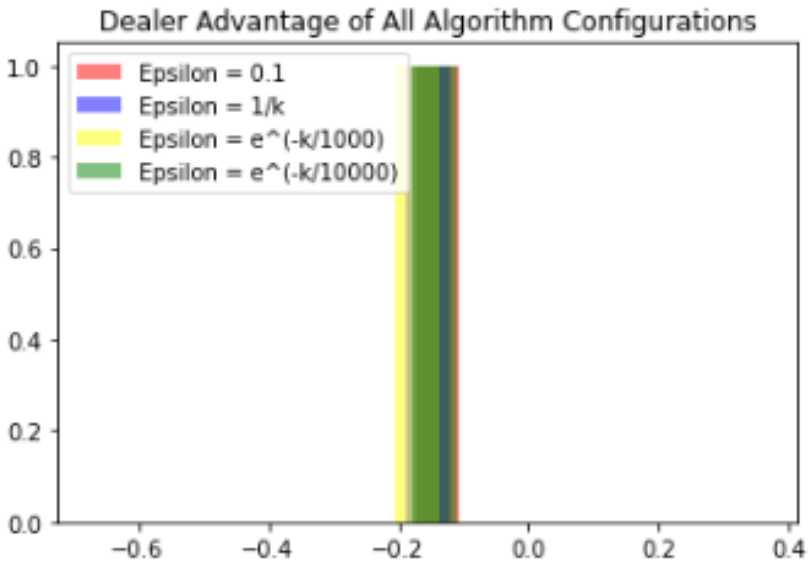
The graphs clearly show the learning algorithm having the win rate and lose rate increase and decrease significantly within the first 20,000 episodes. Furthermore, the draw rate is also increased at a slower rate than the others.



Graphs Based on All 4 Configurations



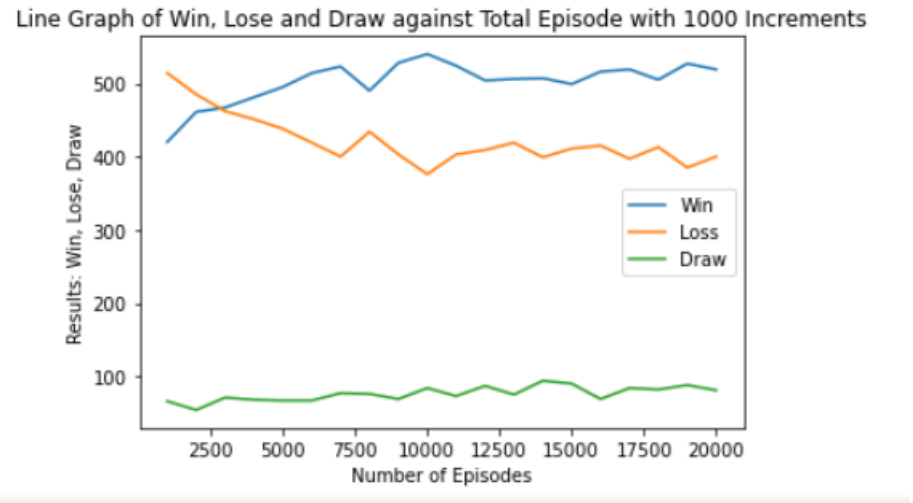
From the bar chart above, it is evident that the configurations that have the highest appearances of unique state-action pairs are 1 or 4 while configuration 2 seems to have the least different unique state-values.

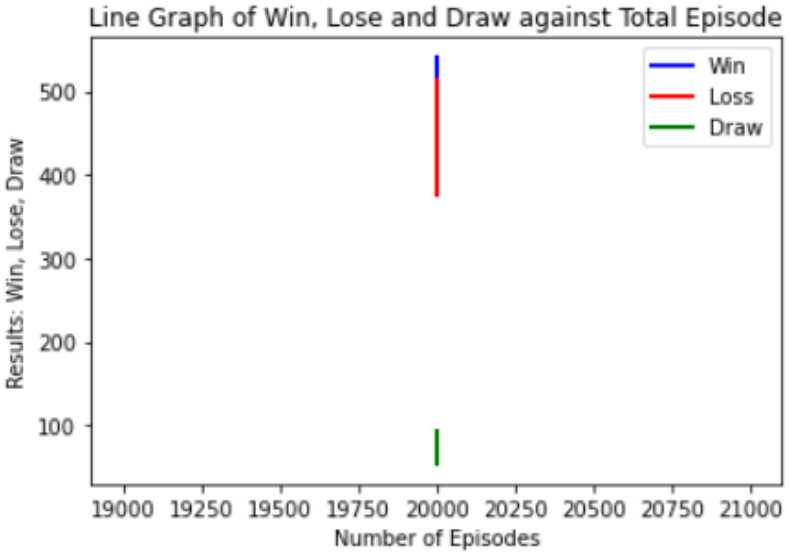


The above histogram shows that the third epsilon results in the dealer having the least advantage whilst the first epsilon value resulted in the dealer having the most advantage.

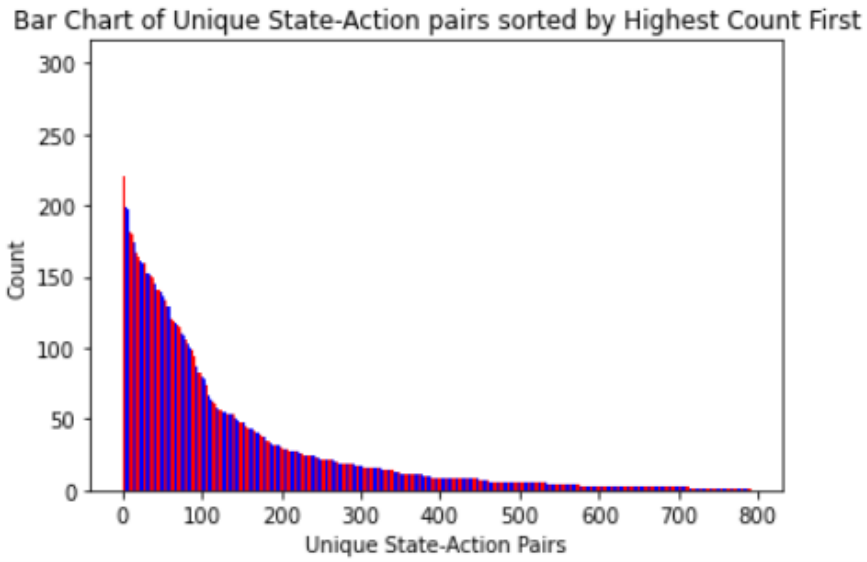
Q-Learning (SARSAMAX) Off-Policy Control Algorithm

Configuration 1: $\epsilon = 0.1$

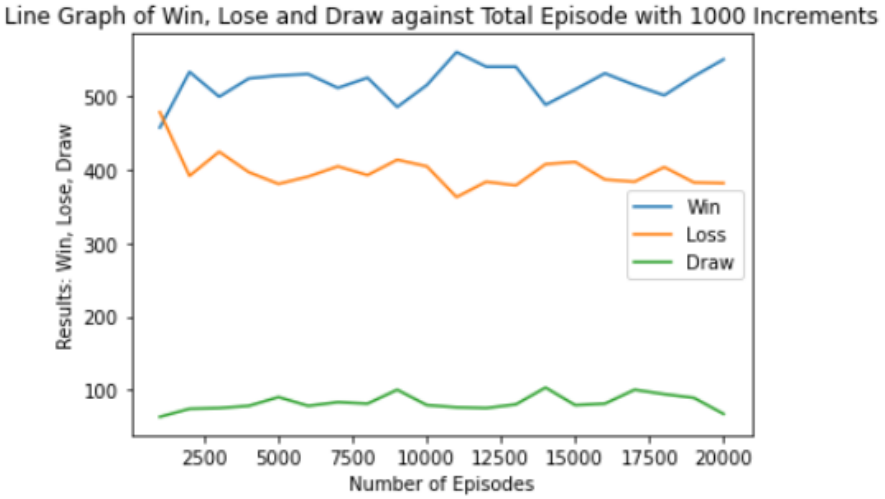


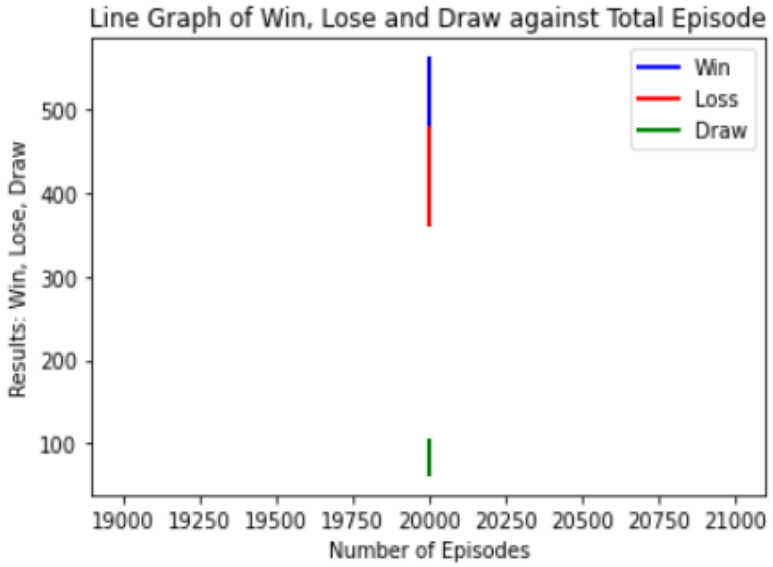


These plots show that this configuration started with having more losses but by around 2500 episodes, the algorithms' win rate increased significantly while the lose rate decreased. Finally, the draw rate seems to remain at a steady rate with a slight increase at the end.

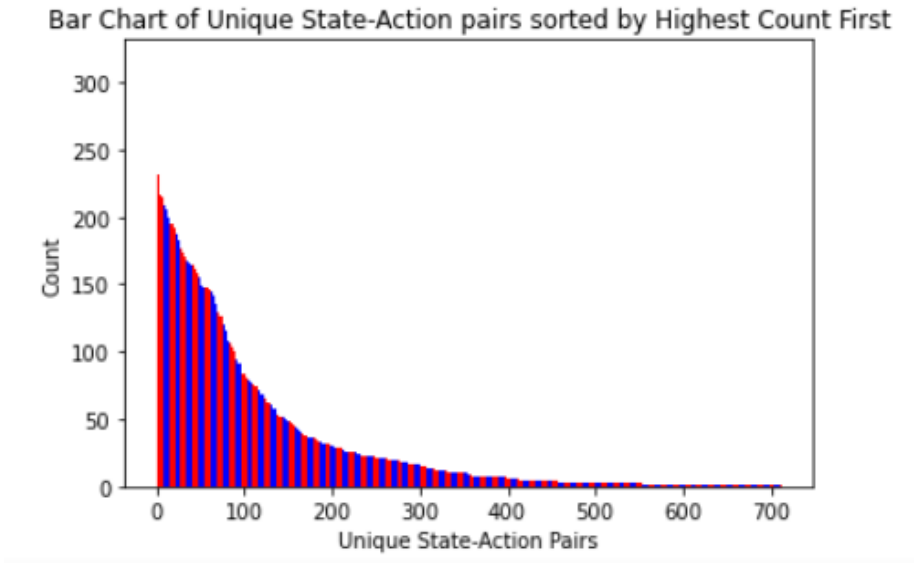


Configuration 2: $\epsilon = 1/k$

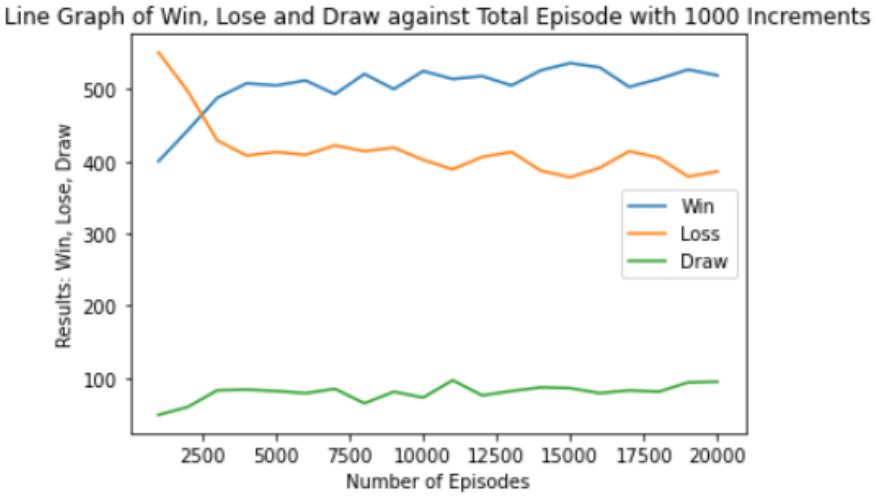


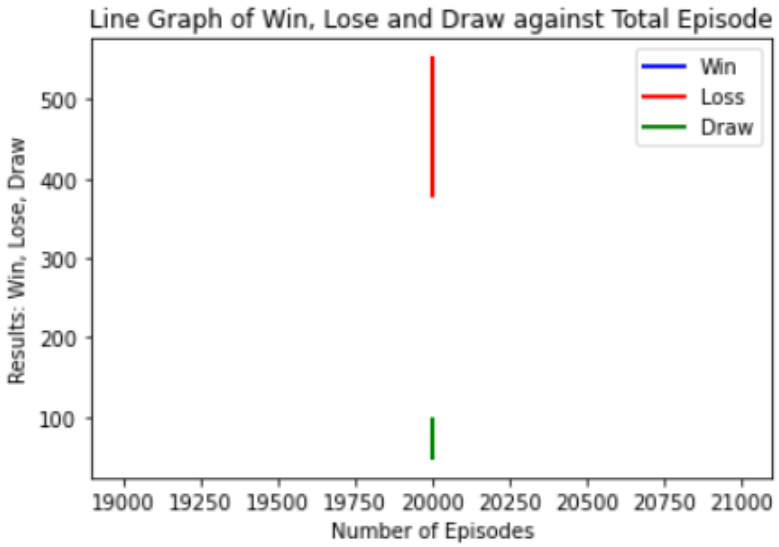


The above graphs show that this algorithm almost immediately had a higher win rate than lose rate. It is apparent that the win rate is increasing while the lose rate is decreasing. Meanwhile the number of ties is at a constant rate.

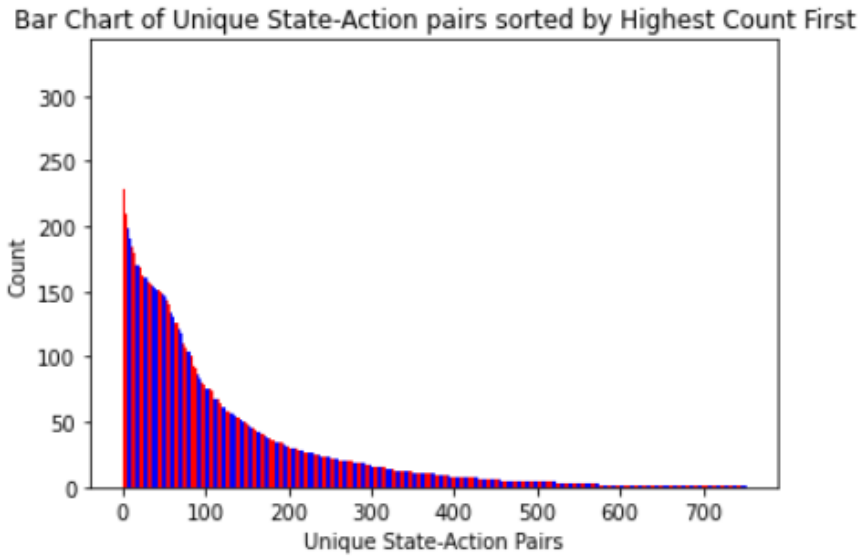


Configuration 3: $\epsilon = e^{\wedge} -k/1000$

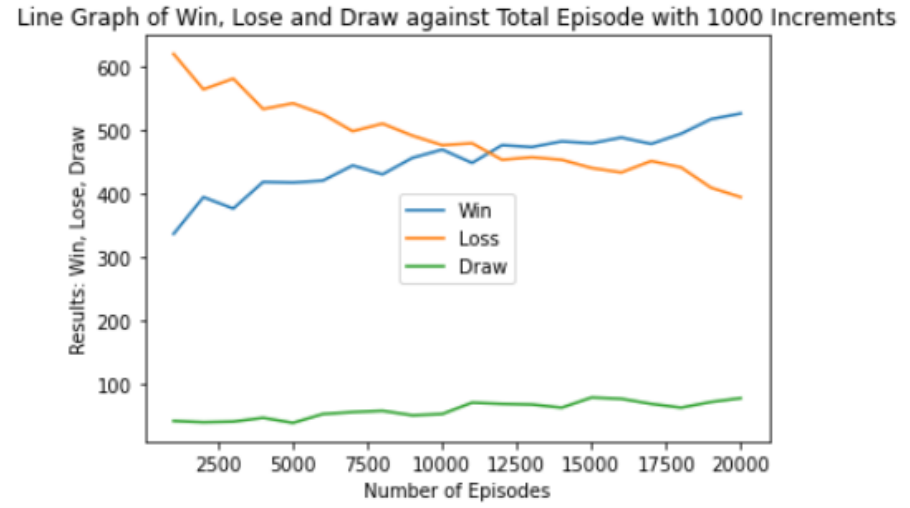


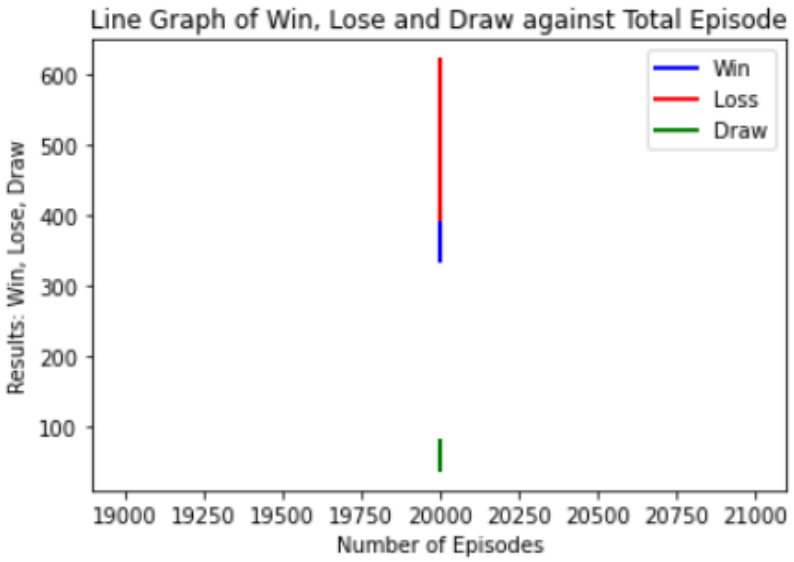


The above graphs indicate that the configuration started with a higher losing rate however, after 2500 episodes the win rate became higher and both lines remained steadily on the same average. The tie line shows a small increase as well.

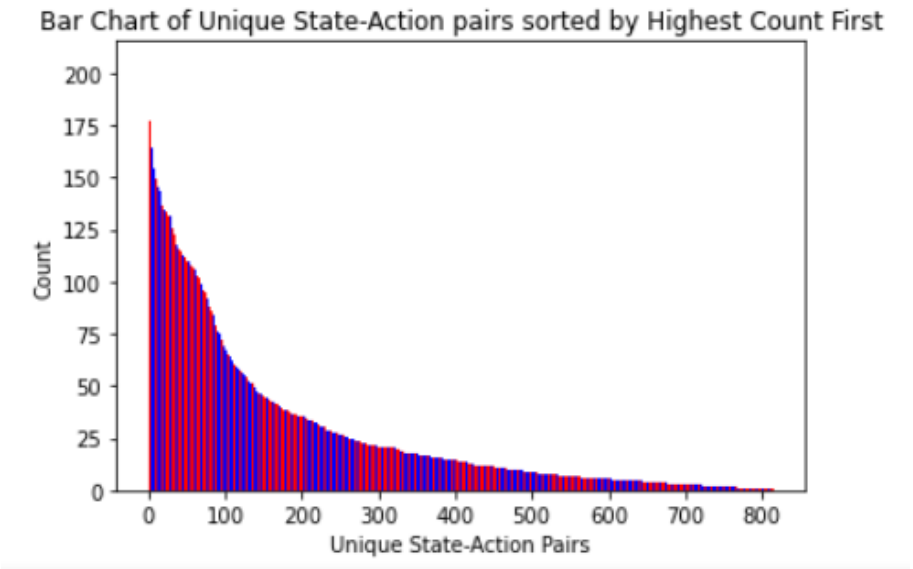


Configuration 4: $\epsilon = e^{\wedge} -k/10000$

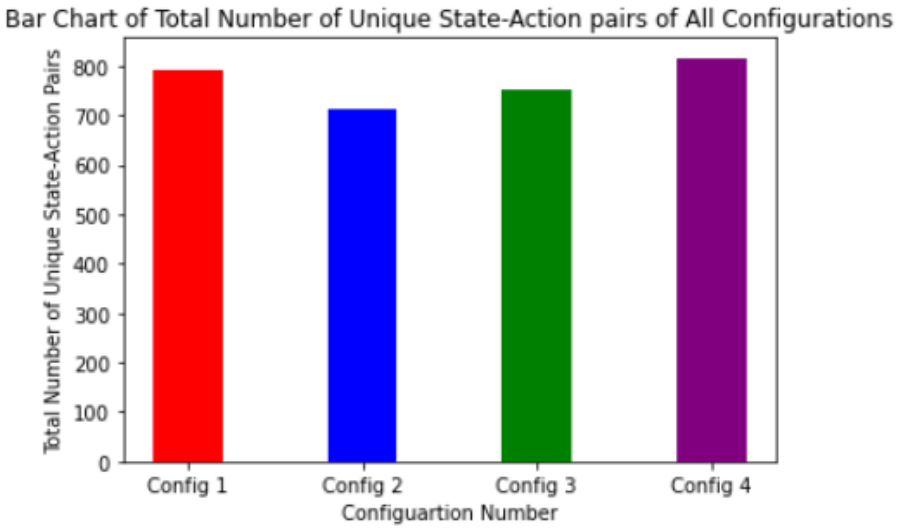




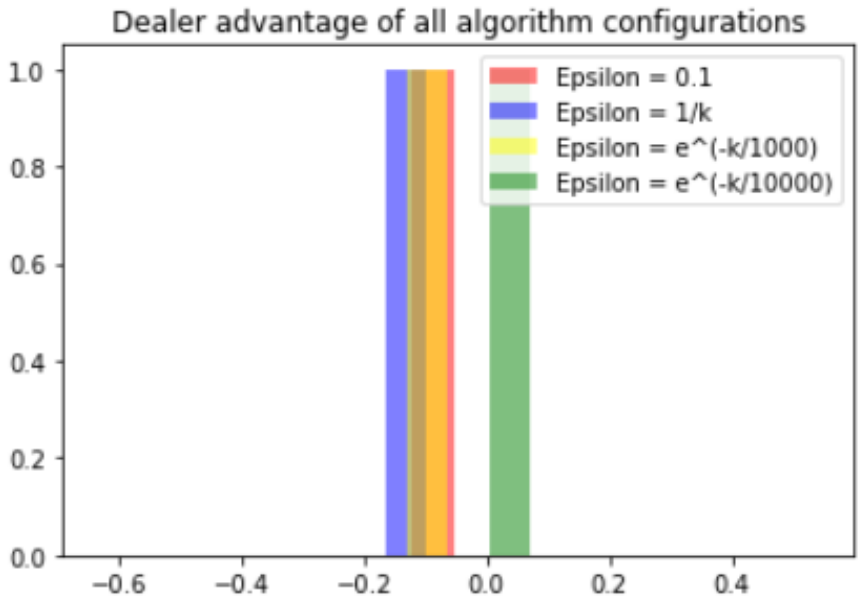
The above graphs illustrate that in Configuration 4, the lose rate started very high, however, after around 10,000 episodes, the algorithm started to win more than lose but it is apparent that the win rate did not exceed the initial lose rate. This could be due to the limited episodes we had to run.



Graphs Based on All 4 Configurations



The above shows that Configuration 4 used the most unique state-action pairs while the second configuration used the least.



This graph clearly indicates that the final configuration gave the dealer the most advantage while for the first, the dealer had the least advantage.

Results

Monte Carlo

For the Exploring Starts configuration of Monte Carlo algorithm, the win rate was increasing more slowly than for the Not Exploring Starts configurations. This could be because Exploring Starts has more exploration than exploitation unlike the other configurations. In fact, the Exploring Starts algorithm used the most unique state-action pairs whilst configuration 2 used the least. The dealer advantage was the highest in the first configuration and lowest in the third configuration. The dealer advantage of the second configuration was slightly more than the third configuration. Hence, the third configuration seems to be the best for 20,000 episodes.

SARSA

The SARSA algorithm starts off with a high lose rate and a lower win rate which quickly learns and starts winning more than losing. The draws start increasing, however, at a much slower pace. It also seems the case that it does not explore many of the unique state-action values and some state-action values appear a large number of times. Between the configurations, it is concluded that the fourth configuration had the highest winning rate, the second configuration used the least unique state-action pairs and the third configuration gave the dealer the least advantage. Therefore, the second or third configuration could be considered the best for 100,000 episodes.

Q-Learning

In the Q-Learning algorithm, it usually starts off with a higher loss rate which proceeds to change over the episodes resulting in a higher win rate. The spacial difference between the two lines varies, however, from configuration to configuration. Between the configurations, it can be seen that the second configuration had the highest winning rate and used the least unique state-action pairs whilst the first configuration gave the dealer the least advantage. The second configuration gave the best results for 20,000 episodes.

Final Remarks

Q-Learning goes through the most unique state-actions pairs having almost 800 each with highest values between 150 and 200. Monte Carlo and SARSA had the least unique state-action pairs ranging around less than 600 which could be due to them being On-Policy unlike Q-Learning which was Off-Policy Control. For the 100,00 episode runs SARSA had the most counts in the unique state-action pairs while for the 20,000 episode runs both Monte Carlo and Q-Learning were both roughly in the same range.

SARSA had the best minimal dealer advantage of -0.2 but it could also be the case that this was due to SARSA having had more episodes. Meanwhile, Q-Learning and Monte Carlo had the most dealer advantage which could have both been due to exploration. Configurations

with more exploration than exploitation resulted in lower win rates. This is due to more random actions being chosen instead of actions with the highest Q(s,a) values.

In this case, SARSA had the best results but this could have been because it was run for 100,000 episodes whilst the others were run for 20,000. It could also be noted that SARSA is the fastest algorithm. Meanwhile, Q-Learning seemed to take the most time to execute.

References

[1] J. Bajada. Reinforcement Learning [PowerPoint Slides]. Available: Course: ARI2204-SEM2-A-2122: Reinforcement Learning (um.edu.mt) (Accessed May 17, 2022)

[2] Quantsol. "Simple-Blackjack-Game" github.com. https://github.com/Quantsol/Simple-Blackjack-Game/blob/main/blackjack.py (Accessed Apr. 26, 2022)

[3] rafalk342. "BlackjackRL" github.com. https://github.com/rafalk342/BlackjackRL (Accessed Apr. 27, 2022)

[4] ShangtongZhang. "reinforcement-learning-an-introduction" github.com. https://github.com/ShangtongZhang/reinforcement-learning-an-introduction (Accessed Apr. 27, 2022)

[5] abken601. "Blackjack-Reinforcement-Learning" github.com. https://github.com/abken601/Blackjack-Reinforcement-Learning (Accessed May 6, 2022)

[6] "Reinforcement learning" en.wikipedia.org. https://en.wikipedia.org/wiki/Reinforcement_learning (Accessed May 6, 2022)

[7] "Q-learning" en.wikipedia.org. https://en.wikipedia.org/wiki/Q-learning#Influence_of_variables (Accessed May 7, 2022)

[8] "Q-Learning and SARSA, with Python" towardsdatascience.com. https://towardsdatascience.com/q-learning-and-sasar-with-python-3775f86bd178 (Accessed May 7, 2022)

[9] "Python – Add values to Dictionary of List" www.geeksforgeeks.org. https://www.geeksforgeeks.org/python-add-values-to-dictionary-of-list/ (Accessed May 17, 2022)

[10] "Note : Convert a list of Tuples into Dictionary" geeksforgeeks.org. https://www.geeksforgeeks.org/python-convert-list-tuples-dictionary/ (Accessed May 17, 2022)

[11] "how to plot bar chart for a list in python" stackoverflow.com. https://stackoverflow.com/questions/34029865/how-to-plot-bar-chart-for-a-list-in-python (Accessed May 19, 2022)

[12] "How to plot a histogram with various variables in Matplotlib in Python?" geeksforgeeks.org. https://www.geeksforgeeks.org/how-to-plot-a-histogram-with-various-variables-in-matplotlib-in-python/ (Accessed May 19, 2022)

[13] “Python Histogram” pythongeeks.org. https://pythongeeks.org/python-histogram/ (Accessed May 19, 2022)

Distribution of Work

Cristina Cutajar

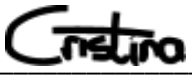
- Main Code
- Monte Carlo On-Policy Control Algorithm Implementation
- Code to extract the required information for 500,000 episodes
- Blackjack Strategy Table and calculating and plotting the dealer advantage
- Documentation

Britney Vella

- Main Code
- SARSA On-Policy Control Algorithm Implementation
- Q-Learning (SARSAMAX) Off-Policy Control ALgorithm Implementation
- Code to plot the extracted information
- Documentation

Nicole Shaw

- Main Code
- Q-Learning (SARSAMAX) Off-Policy Control ALgorithm Implementation

Cristina Cutajar 

Britney Vella 

Nicole Shaw