Reinforcement Learning

Study Unit ARI2204

Mario Vella (81502L)
Kian Parnis (0107601L)
Matteo Sammut (0206002L)

Contents

Algorithms

Monte Carlo On-Policy Control

The Monte Carlo method is a Reinforcement Learning algorithm which looks at complete episodes and learns from its experience. This specific algorithm calculates the mean return (Q) at the end of every episode. For each state-action pair that exists, two values are kept and these are the counter/frequency of every state-action pair across all the episodes and the sum of all the returns for a state.

When it came to implementing the Monte Carlo Algorithm, the user was given a choice of which configuration they wanted to use. These being firstly, the choice between Exploring Starts and Non-Exploring Starts and if the Non-Exploring Starts variant is chosen, the user is then given a choice between the 3 configurations used in that variant. The 3 configurations include a different value to Ɛ which are:

- $\varepsilon = \frac{1}{k}$
- $\varepsilon = e^{\frac{-k}{1000}}$
- $\varepsilon = e^{\frac{-k}{10000}}$

Figure 1: Epsilon Value configurations for Non-Exploring Starts

Once this choice is made, the game function is called for the main game to start. Before the true algorithm can start running several things are done. Three separate dictionaries are created, the Q Values dictionary which contains all the q values for every state-action that the algorithm may encounter, the frequency dictionary which contains the frequency of how many times a certain state-action pair is encounter and the policy table dictionary which contains a boolean of whether the algorithm should hit for a certain state. These three dictionaries are vital to the Monte Carlo Policy Control algorithm working correctly.

Once these dictionaries are created, the training of the algorithm commences. Every time the algorithm is run, it has a specific number of episodes which it must run for. In this case, the algorithm is running for 500,000 episodes. After each 1000 episodes, the program will be printing the number of wins, losses and draws achieved by the Monte Carlo Algorithm with that specific variant and configuration.

Firstly, the algorithm will create a brand-new environment each episode. At the start, this includes a 52-card deck, the first two cards the player will draw and the first card that the dealer will draw. These values are sent back to the main game function and the policy checking starts. Firstly, the algorithm will make sure that the player has between 12 and 20 by hitting until the player is within this range. Once this is done, the current state is inputted into a state-action list.

The next part of the algorithm is calculating the probability of the player hitting or standing. This all depends on the configuration chosen as the values will be completely different from each other. Once the epsilon value is computed, the action required is then chosen.

If the player hits, there are two ways that this action may affect the player. Firstly, the player might bust, meaning that the player's total card value has surpassed that of 21 and the player has lost. If that happens, the number of losses is incremented and the policy and q-value of the specific path is improved so that next time this state-action is encountered, a better-informed decision might be taken. If the player does not pass 21, then the algorithm goes onto the next state where a new action is chosen.

Secondly, if the player stands, there are 4 different ways that the game can play out. The dealer policy will keep on hitting until the value passes 17. If, coincidentally, the dealer surpasses 21, the dealer loses and thus the player wins. Even though this has arrived at a win, the q-values and policy are still improved so that a win can always be assured. If the dealer doesn't win, the result all depends on luck. The final three ways the game can play out are if the player has a higher card value than the dealer, the dealer has a higher card value than the player or the player and dealer have equal values. For each destination, the policy is improved using the respective reward.

After an episode finishes, the algorithm then creates a brand-new environment and starts the process, which was described above, all over again.

SARSA On-Policy Control and Q-Learning (SARSAMAX) Off-Policy Control

Description of Algorithm

SARSA-MAX, also known as Q-Learning, together with SARSA both fall under a class of Temporal Difference (TD) learning algorithms, these types of algorithms are referred to as model-free algorithms, these algorithms do not rely on the transition probability distribution of a Markov decision process i.e., they do not require the full model of the MDP to be known (probabilities & rewards) such as Dynamic Programming [1]. Thus model-free learning relies solely on 1) Model-free Prediction where the value function of an unknown MDP is estimated given some policy π and 2) Model-free Control where the policy π of the unknown MDP gets estimated.

The way TD methods can learn are directly from each episode, episodes consist of a finite sequence of steps as well as a final step where an episode would terminate. For example, for a game such as flappy bird, steps can be achieved whenever the bird goes through each pipe successfully, whereas the terminal condition would be hitting the pipe and losing the game. Applying TD learning to this example would lead to a TD algorithm forming some initial estimate, instead of relying on the episode to be fully complete, it will update periodically after every successful step (On-Policy) and in the end when hitting the pipe, based on the feedback (rewards) it receives. This procedure is also known as bootstrapping [2] and SARSA / SARSAMAX both adopt these techniques in their algorithms.

Going further to the algorithms themselves they both start by initialising a Q table Q (s, a) where every state action pair would initially be set as a value 0 and overtime depending on the outcome of each step/episode these algorithms would update these Q (s, a) values depending on their respective equations. The Q-Table itself would be fully loaded into memory and depending on the scale of the problem these can instead be switched out with differentiable function approximators (such as ANN) which utilise gradient descent to approximate the policy value function instead, lowering the special cost required [3].

After the Q-Table has been initialised, each algorithm consists of two main loops, an outer loop which depends on the amount of episodes training would require and an inner loop which takes a certain number of steps specified within that episode. For the outer loop SARSA / SARSAMAX would need to first initialise the base modelled environment with starting state S. S does not require to be the same exact state from the episode prior. Following this, an action would have to be chosen from this starting using the policy derived from Q (ex: e-greedy). E-greedy policies are stochastic policies which with some probability e, would choose a random action, and some probability 1-e choose a greedy action argmax q (π) (s, a) [4]. After the action is taken, an immediate reward is returned based on the outcome of that action, followed by an observation for what the future state S' will be.

The main difference between SARSA and SARSAMAX is in how they observe the future action A', for SARSA the same e-greedy policy is used again to determine the action A' which would still lead to some probability of a random action being chosen. SARSAMAX however, only considers the max value taken so far (hence the MAX in SARSAMAX) which is referred to as a greedy policy for the successor state S'. With the current and future observations for the state action values as well as the reward, the Q (S, A) will be updated following the equation Q (S, A) <- Q (S, A) + α(R + γmaxQ (S', A') – Q (S, A)) or Q (S, A) <- Q (S, A) + α(R + Q(S', A') – Q (S, A)) for SARSAMAX and SARSA respectfully, where α is the learning rate and γ is the discount factor. Finally, for Q-learning the current state S is set to the future S' and SARSA would also set the current action A to the future A'. The inner loop terminates once S is a terminal state, or the step size is reached while the outer loop terminates once the episodes finish [2].

Implementation of Algorithms

When it came to implementing SARSA and SARSAMAX for the blackjack problem, both algorithms were implemented together with the user deciding which one to use. This would change the inner loop of the algorithm to take an e-greedy action if the user chose SARSA and a greedy action if the user chose Q-Learning (SARSAMAX) . After picking which algorithm to use, the user also has the choice of choosing from four different configurations which decides what the epsilon value will be: e = 0.1, e = 1/k', e^-k/1000 or e^-k/1000. The main block *Train ()* gets called which runs the algorithm and returns to the user the final Q-Table.

The algorithm is set to a base episode count of 500000 episodes and starts by calling *IniQ ()*, this function initialises the Q-Table using a dictionary and loops to initialise all possible states to 0. These states consist of the tuple *playerHand* [12 .. 20], *dealerHand* [2..11], the Ace flag [0,1] and finally, all possible actions [HIT, STAND]. *IniQ ()* also returns the max steps, learning rate and all the possible actions to be utilised by the algorithm. We then proceed to the outer loop where the environment is initialised by the helper function *environment ()*, this instantiates the deck and draws twice for the player and once for the dealer. Once the environment is set the player policy is called which automatically draws cards for the player until they have at least a sum of twelve. When checking the total for the player/dealer, *total ()* is called, which sums up all the cards. Depending on the current hand size, this function has each ace set to eleven by default, but if the total exceeds twenty-one, any number of aces are set to one up until the sum is less than twenty-one.

If just by running the player policy, the player reaches twenty-one, then that episode is skipped since nothing can be learnt. The state action values are then considered, the state is set to the tuple composed of the player's hand size, the dealer's hand size and if any ace the

player has is set as eleven. An action is taken using e-greedy (*takeActionEpsilon ())* which either returns a random action or the best action value with some probability depending on e. For the inner loop, while max step hasn't been reached and the player hasn't chosen to stand, the algorithm checks via *takestep ()* what information we can get about the environment, this returns the immediate reward if the player decides to continue or the final reward if the player stands as well as if the final state has been reached. The idea behind the rewards are as follows:

When the player decides to hit:

· If the player hits and by doing so, they exceed twenty-one then the player is penalised

· If when hitting the player has a total of twenty-one, they automatically stand and then dealer plays, the dealer's policy is called which keeps hitting if his sum is less than seventeen. If the dealer busts the player is rewarded, if they both end with a sum of twenty-one it's a draw thus a reward of 0 is given and finally, if the player is at a sum of twenty-one and the dealer stands with a sum less than the player, then the player is rewarded.

· If the player doesn't bust or exceed twenty-one after hitting, he's given a neutral reward of zero.

When the player decides to stand:

· The dealer plays and depending on the outcome, the player is rewarded for winning, penalised for losing and finally given a neutral reward if they drew.

If the player stands then that is the final state, hence the next state action pair is not considered. If they decide to hit, the next state S' is taken and then depending on the configuration chosen by the user, the next A' value is either taken using e-greedy again (SARSA) or argmax is taken (Q-learning). Finally, the Q-Values are updated based on the formula $Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma \max Q(S', A') - Q(S, A))$ *or* $Q(S, A) \leftarrow Q(S, A) + \alpha (R + Q(S', A') - Q(S, A))$ for SARSAMAX and SARSA respectfully. If the user is at a final state, then the formula is changed to $Q(S, A) \leftarrow Q(S, A) + \alpha (R - Q(S, A))$ since no future state exists. Finally, before the inner loop is repeated the S, A values are changed to the S', A' values. This algorithm would then continue repeating the inner and outer loop till the episode count is reached.

Evaluation

1000 Episode Graph

SARSA Algorithm

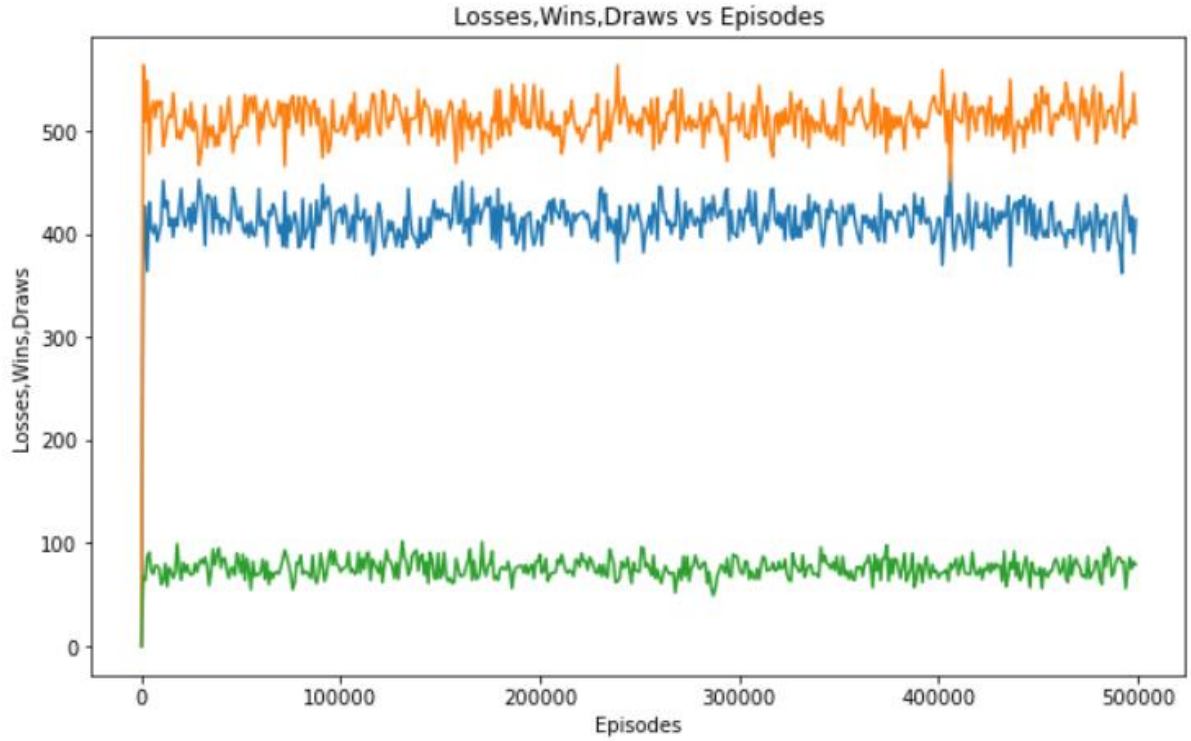

Figure 2:SARSA, e = 0.1

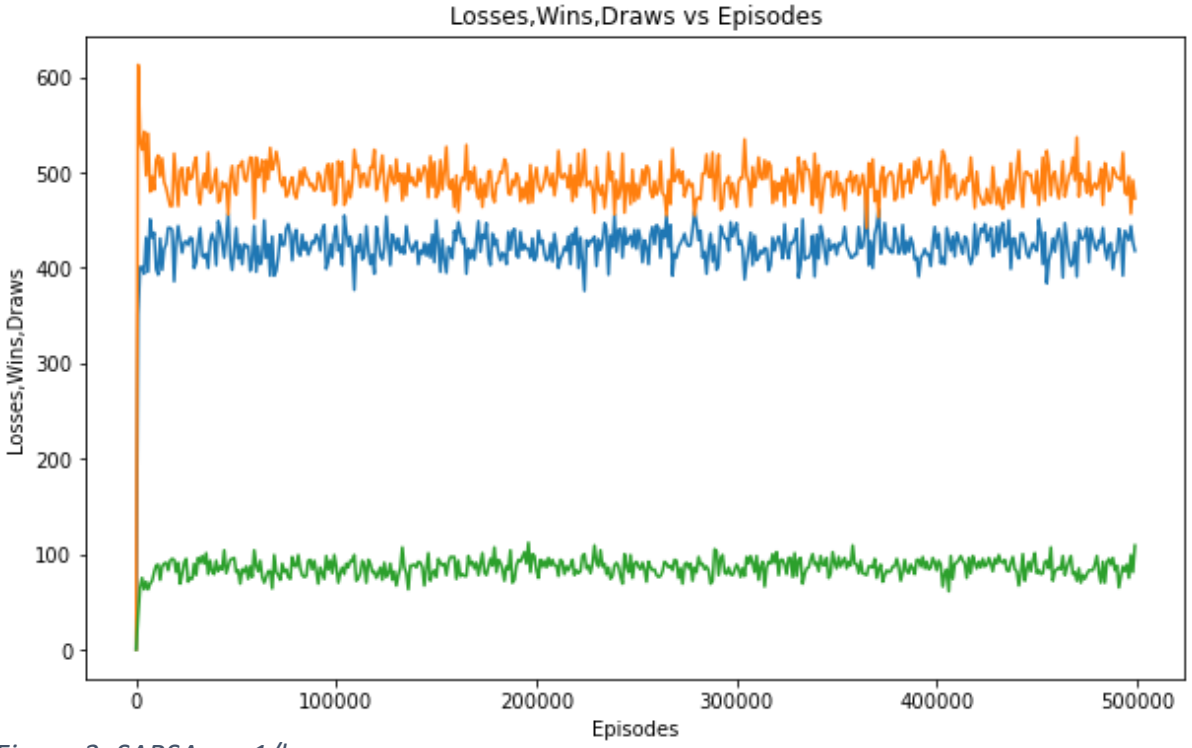

Figure 3: SARSA e = 1/k

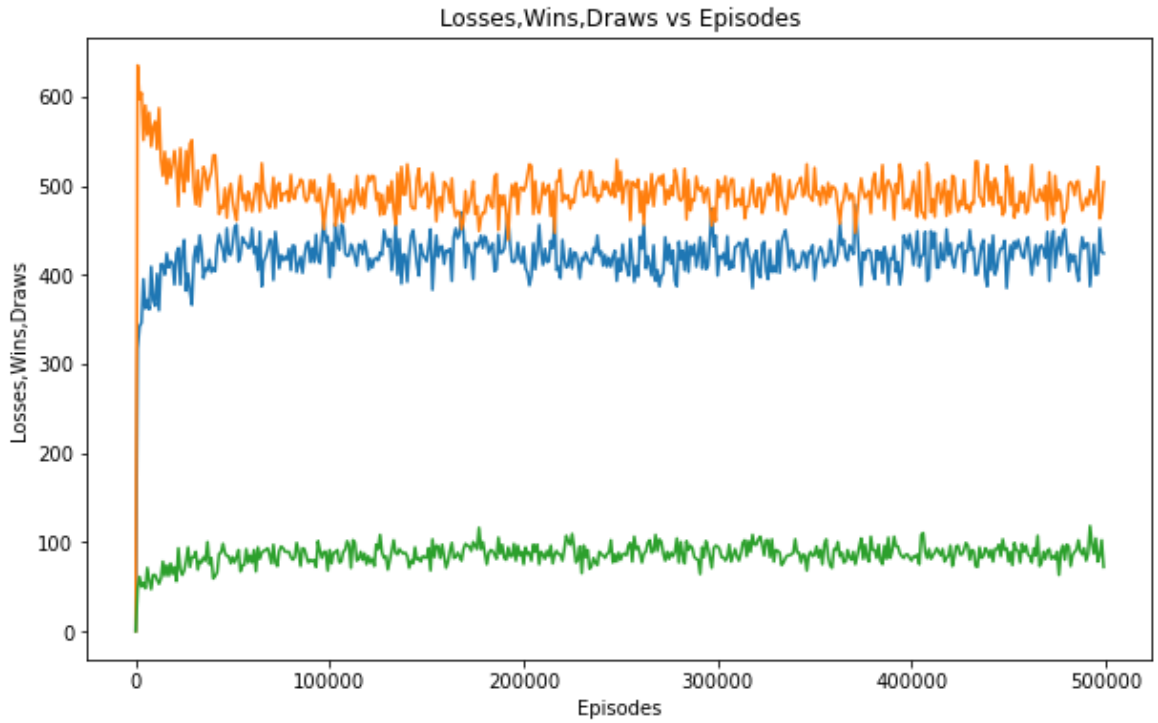

Figure 4: SARSA, e^-k/1000

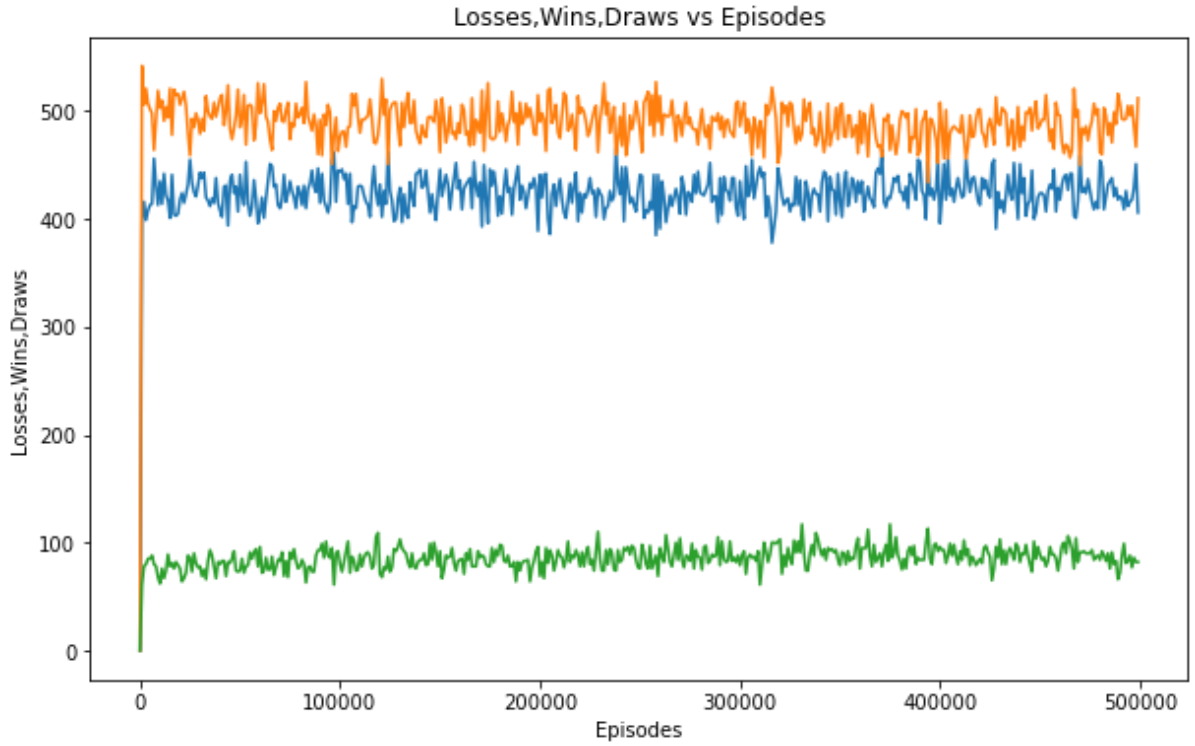

Figure 5: SARSA, e^-k/10000

SARSAMAX Algorithm

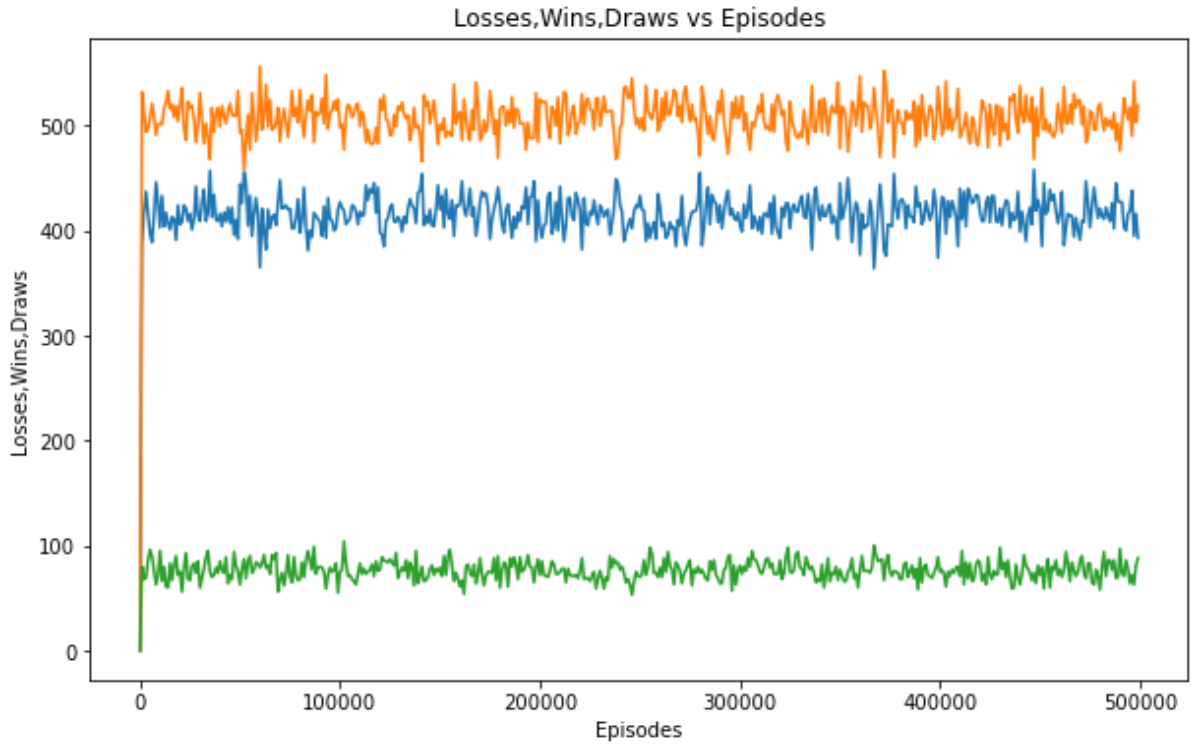

Figure 6: SARSAMAX e = 0.1

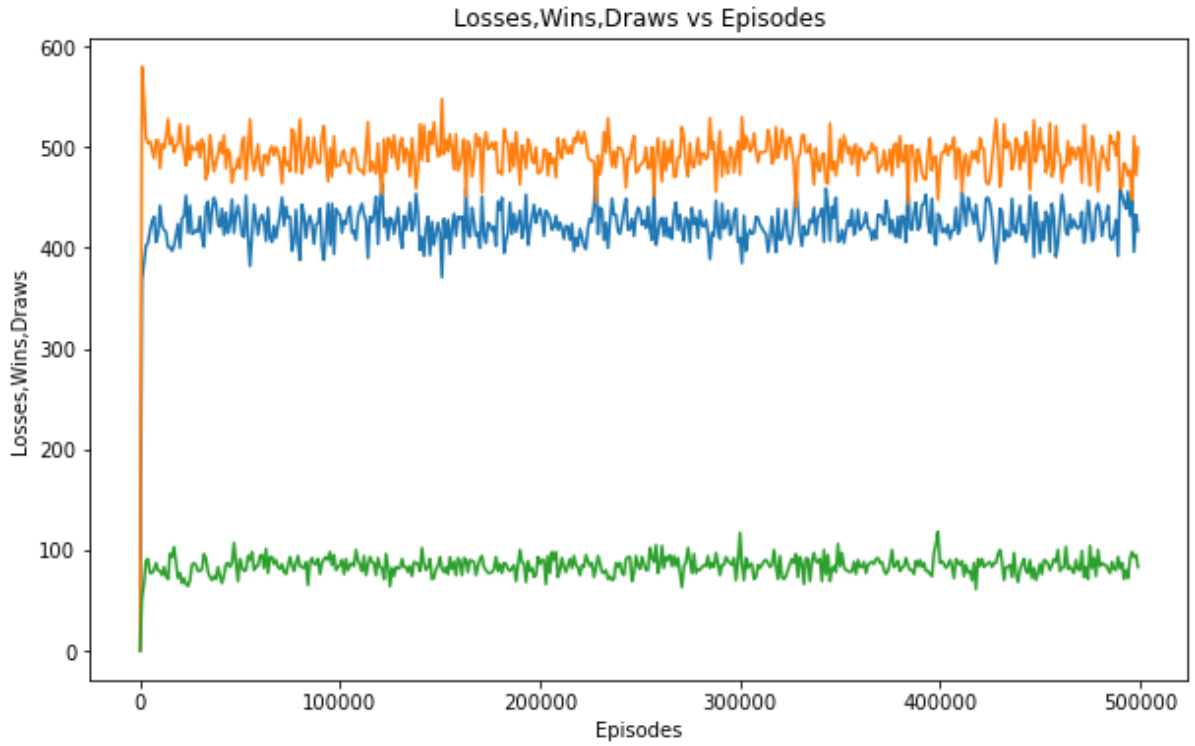

Figure 7: SARSAMAX e= 1/k

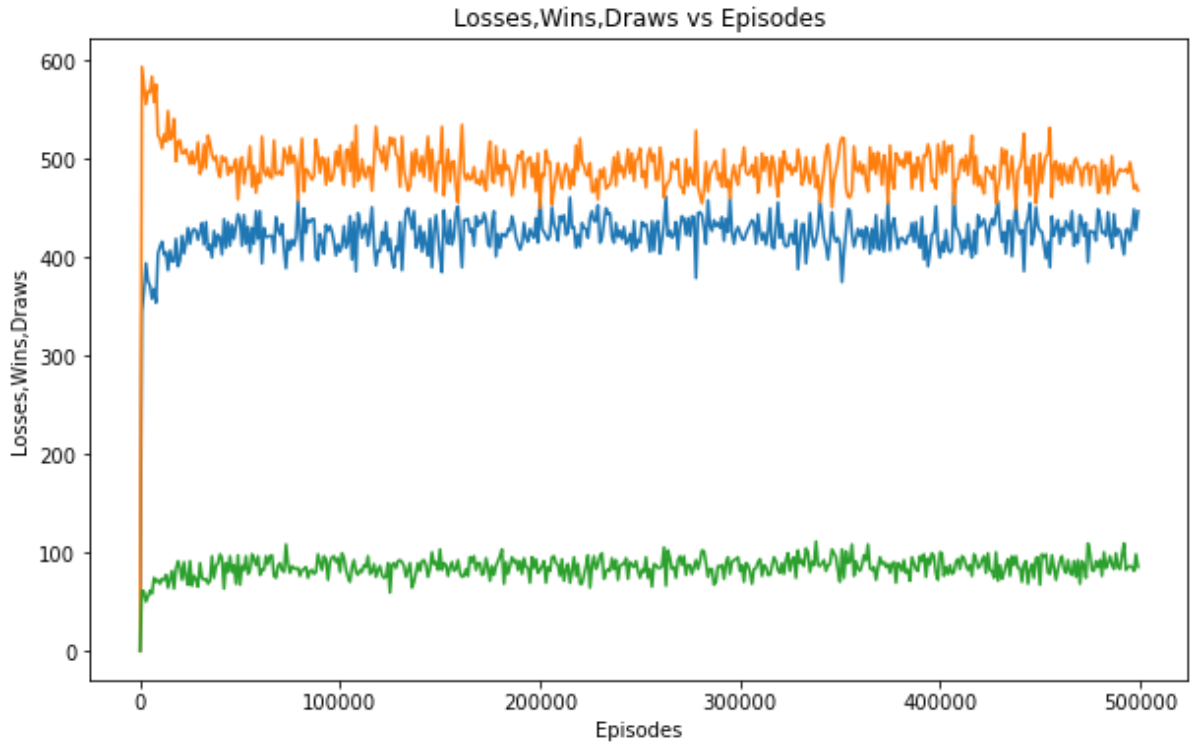

Figure 8: SARSAMAX e^-k/1000

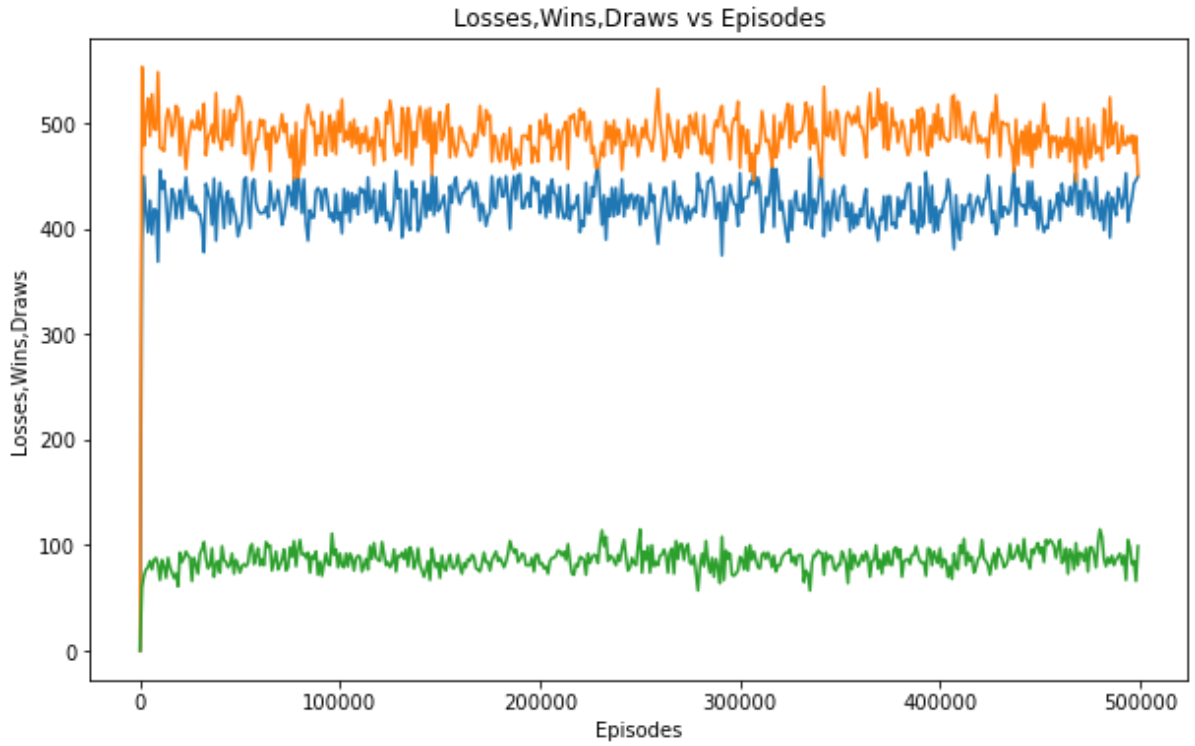

Figure 9: SARSAMAX e^-k/10000

Monte-Carlo Algorithm

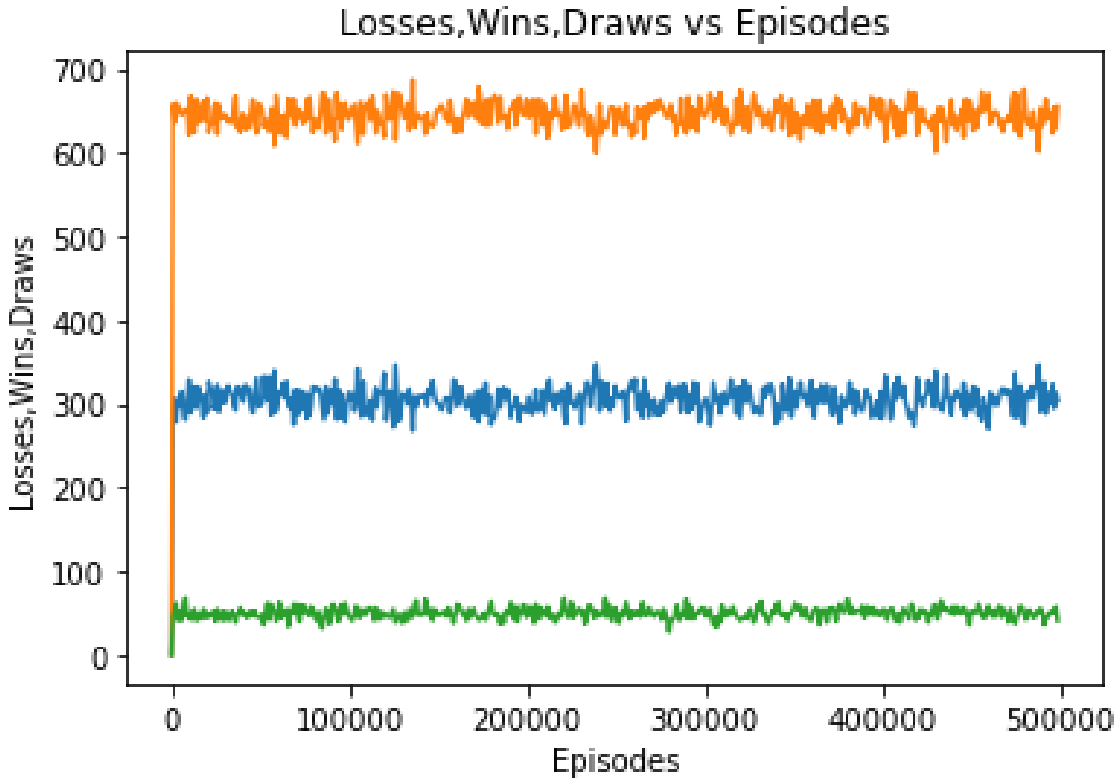

Figure 10: Monte Carlo, Exploring Starts

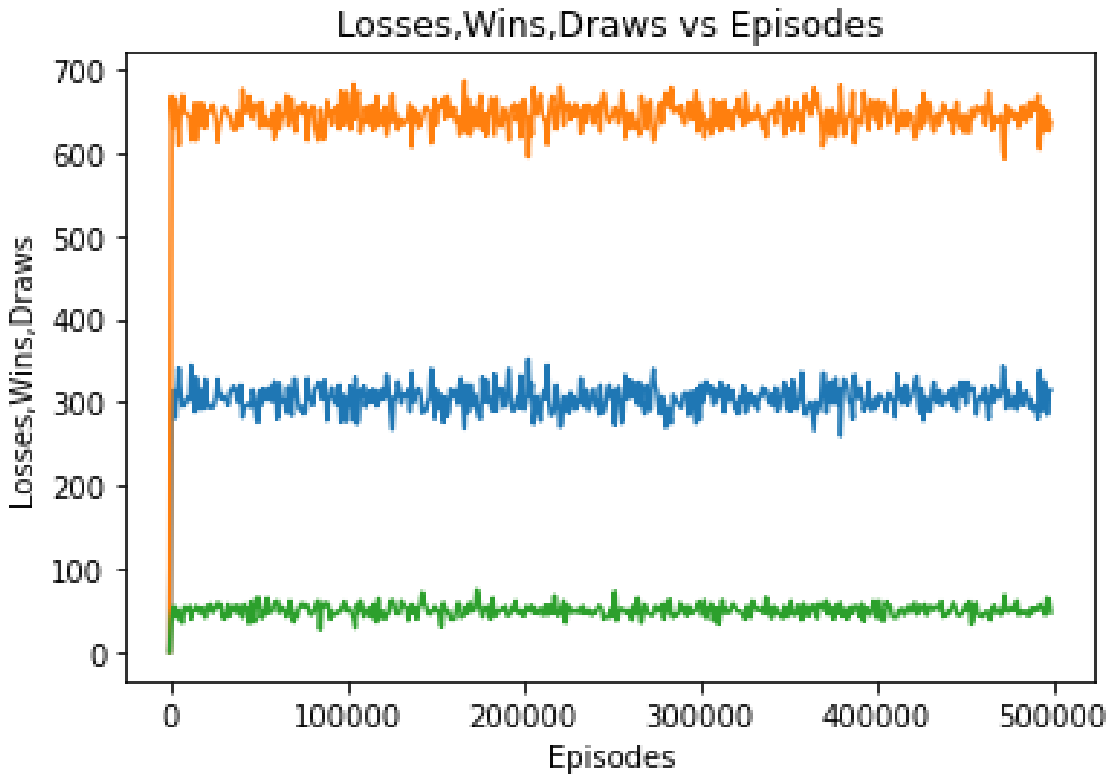

Figure 11: Monte Carlo e = 1/k

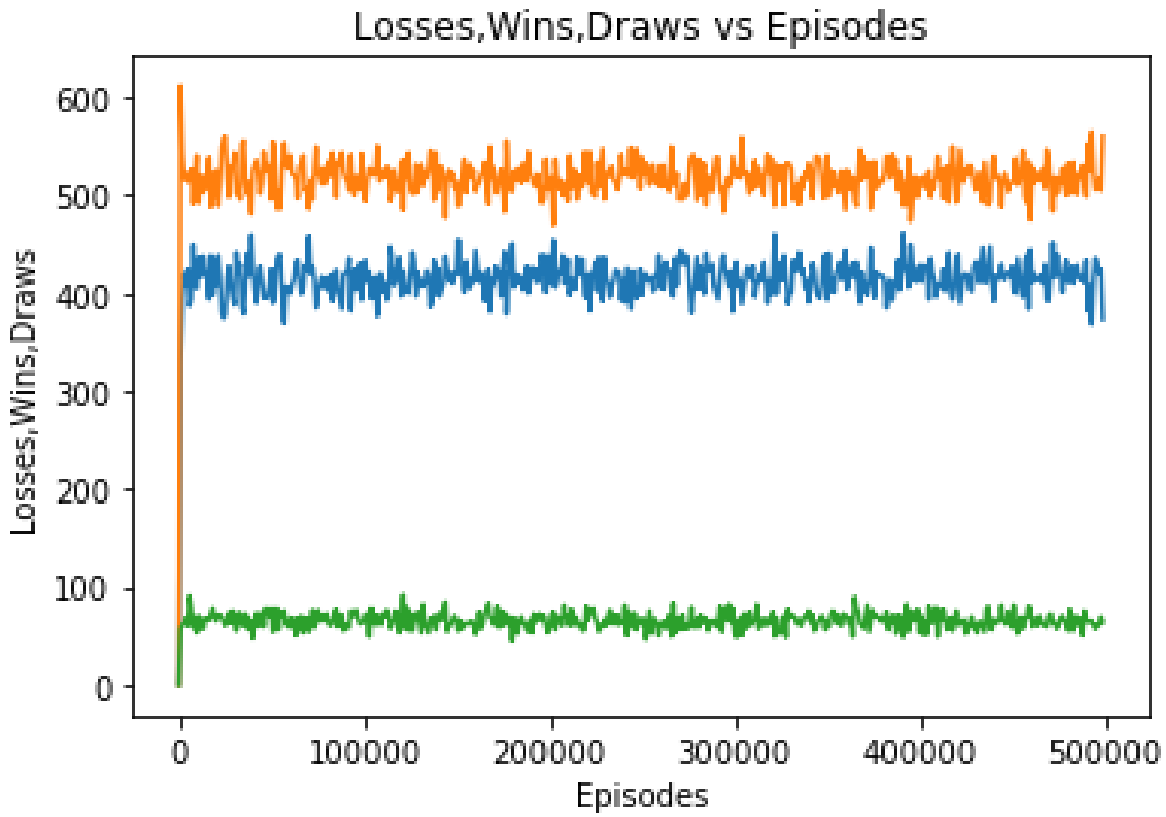

Figure 12: Monte Carlo e^ -k/1000

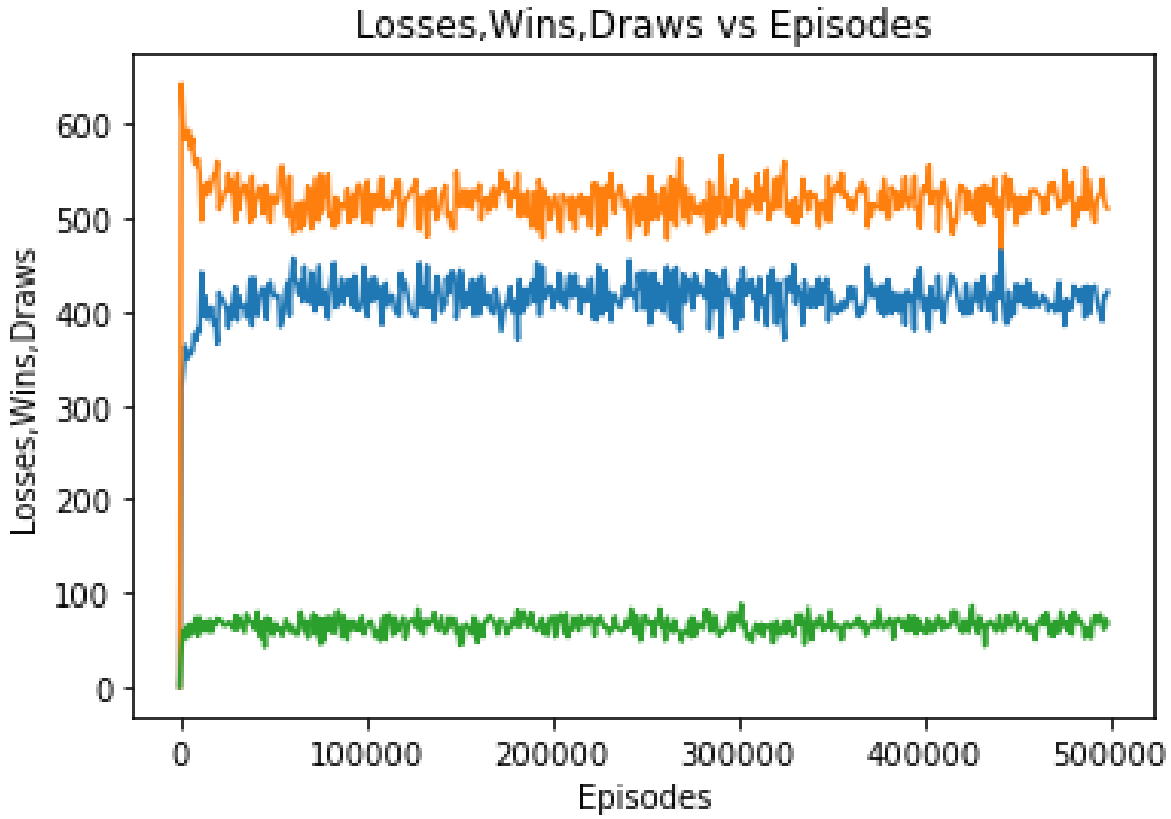

Figure 13: Monte Carlo e^-k/10000

Unique State-Action Pairs

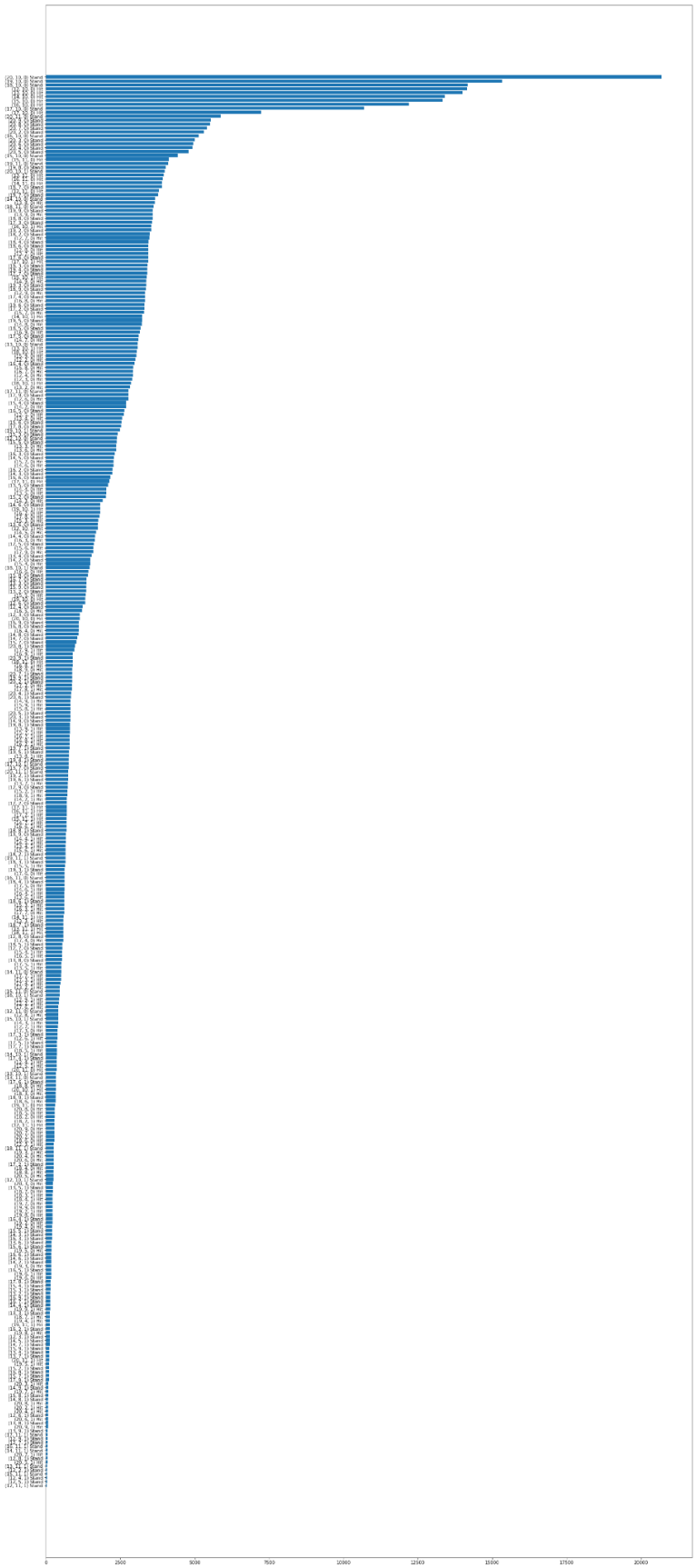

Figure 14: SARSA e = 0.1

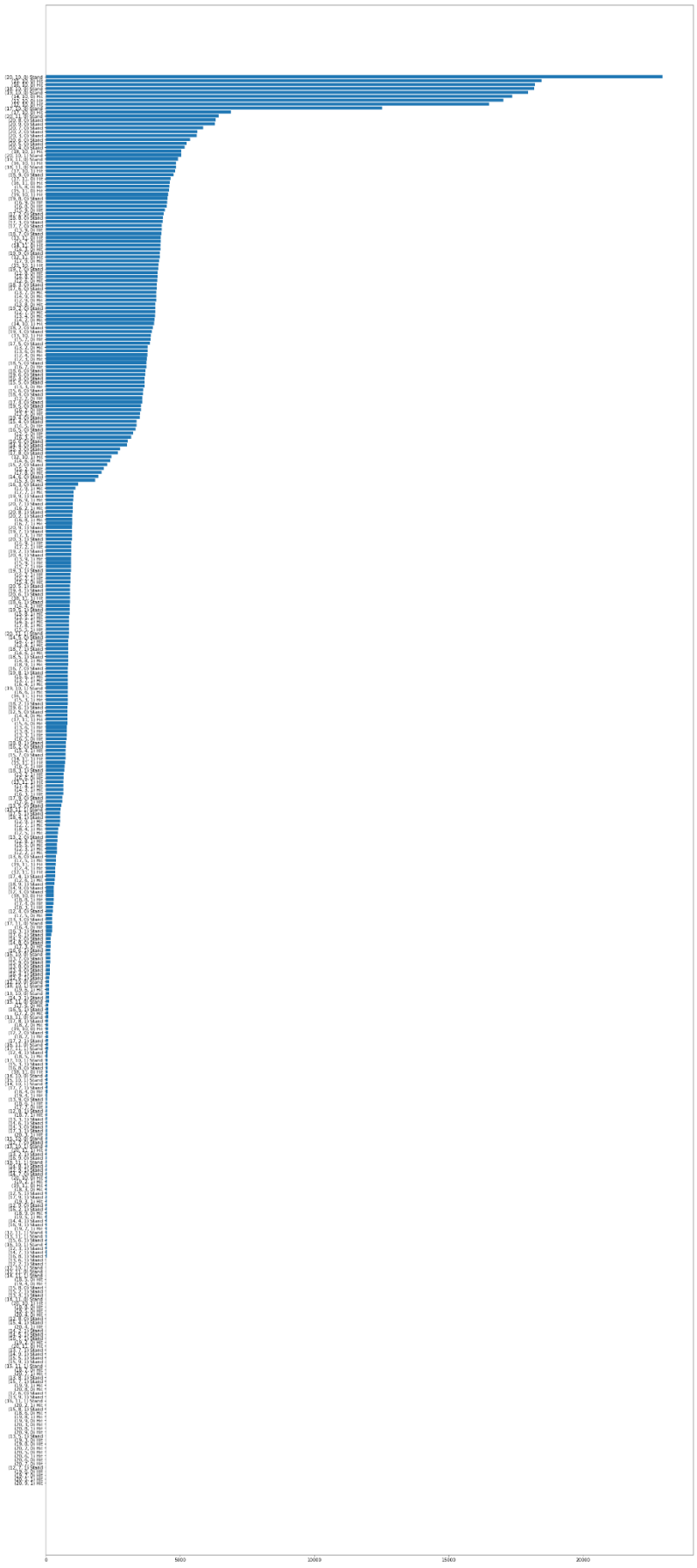

Figure 15: SARSA e =1/k

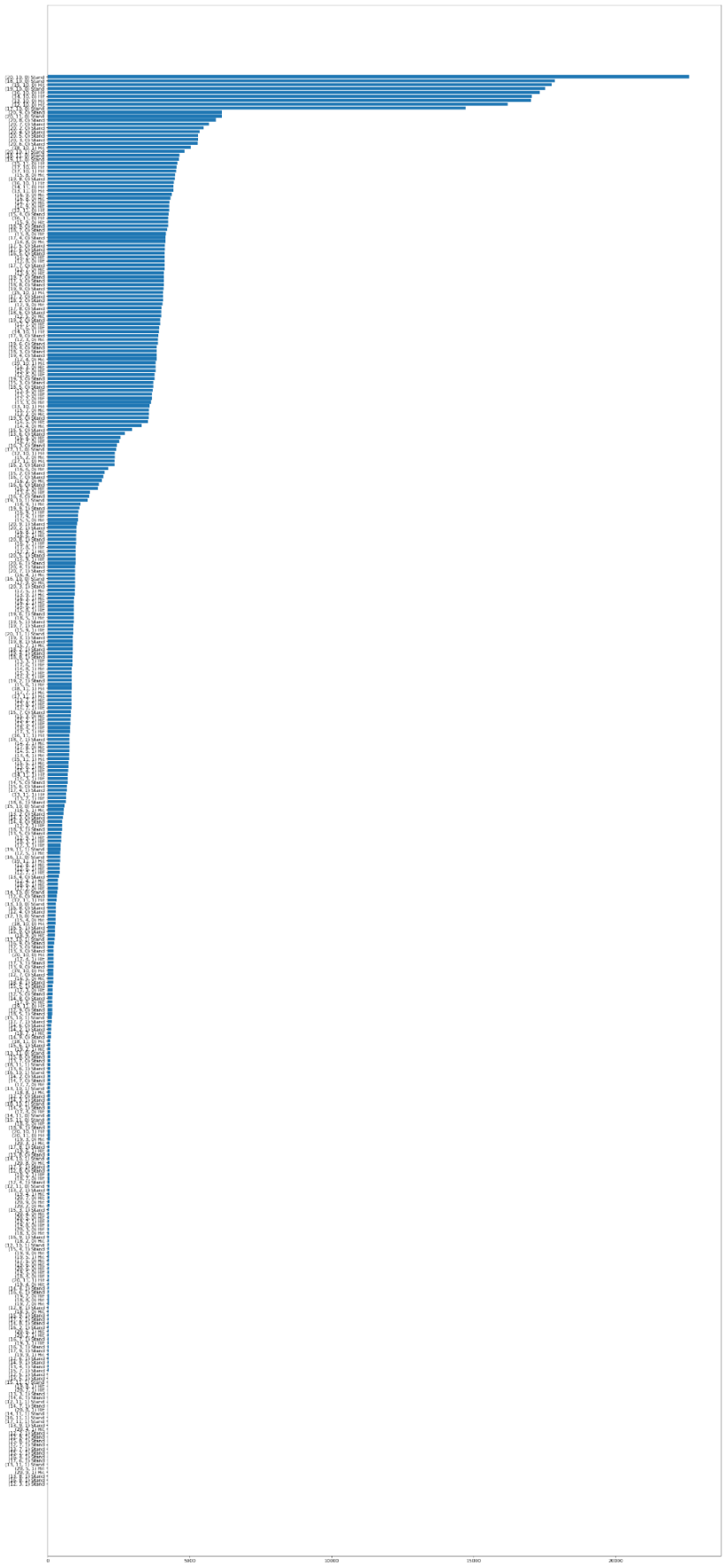

Figure 16: SARSA e^-k/1000

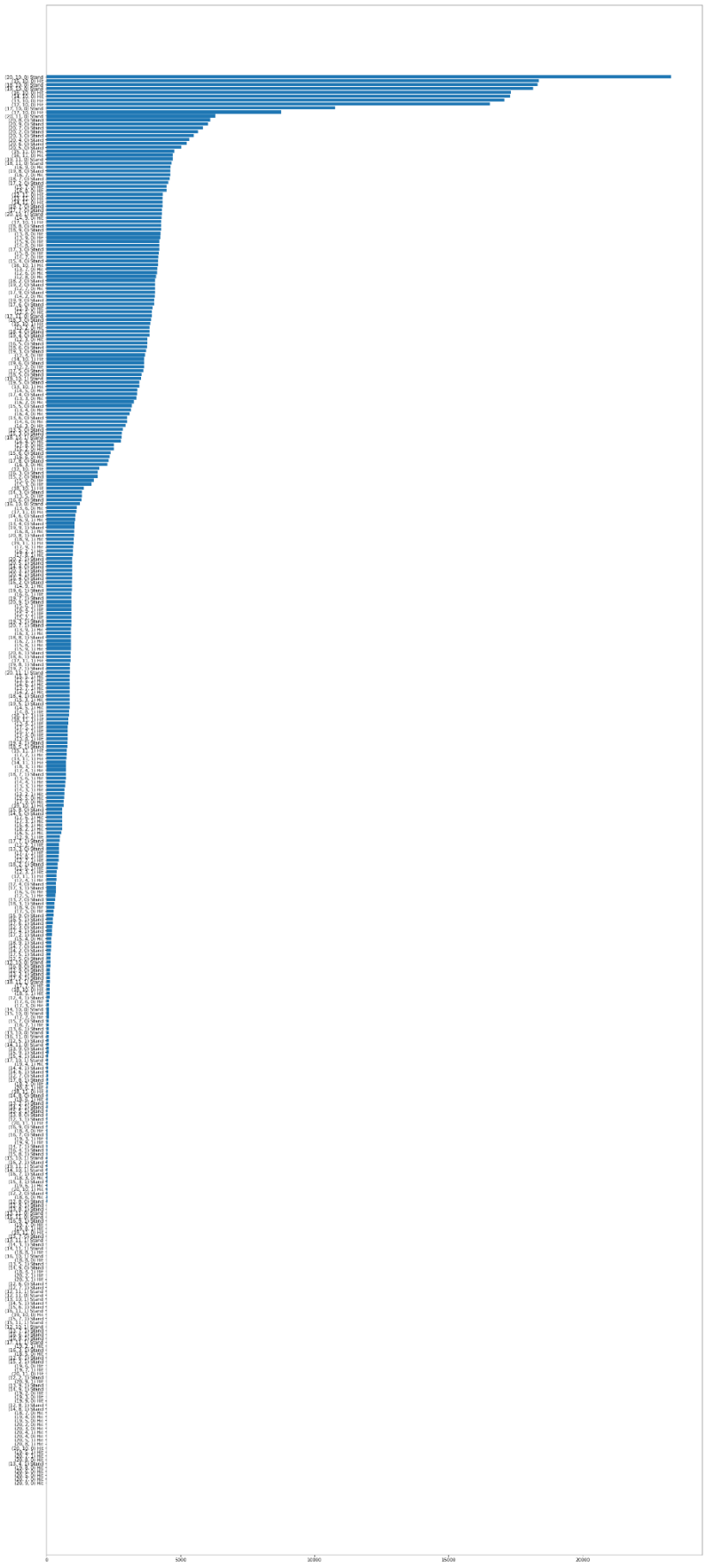

Figure 17: SARSA e^-k/10000

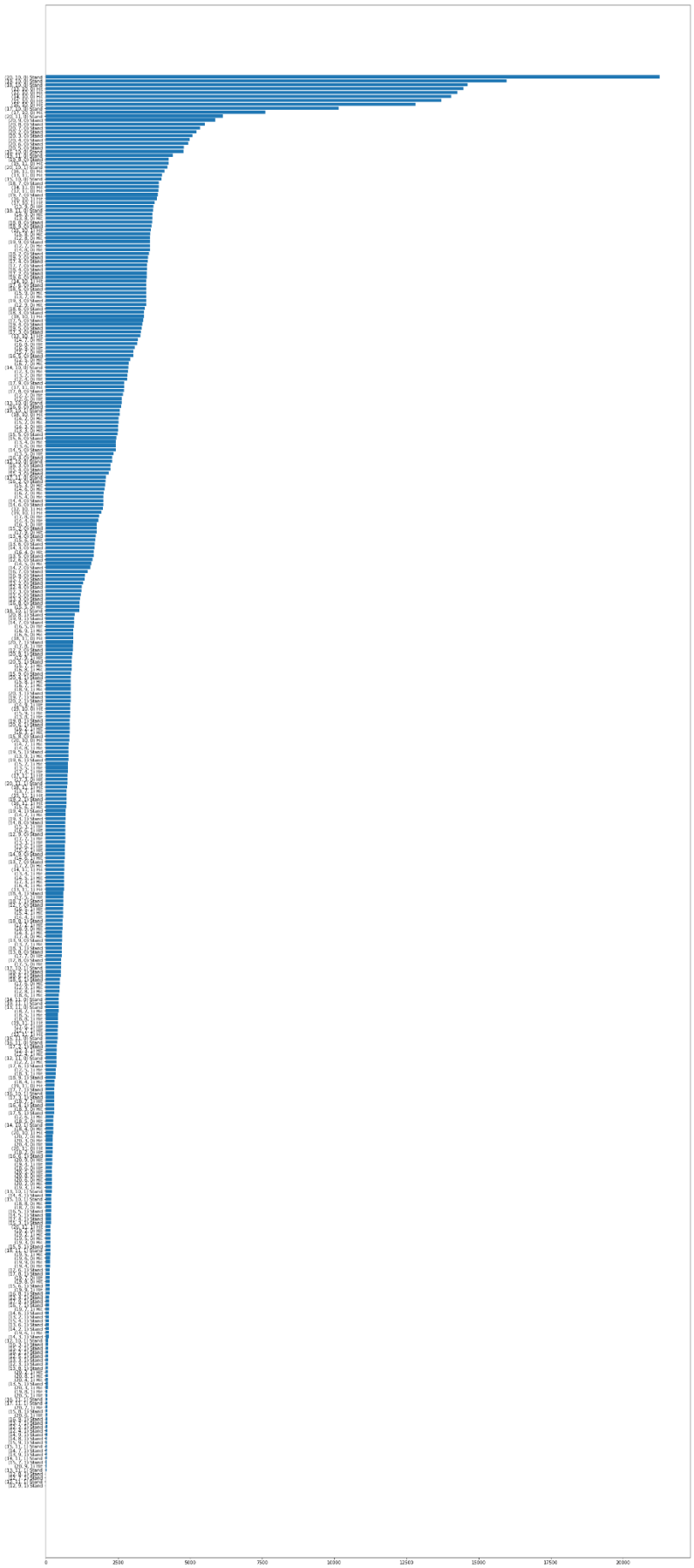

Figure 18: SARSAMAX, e = 0.1

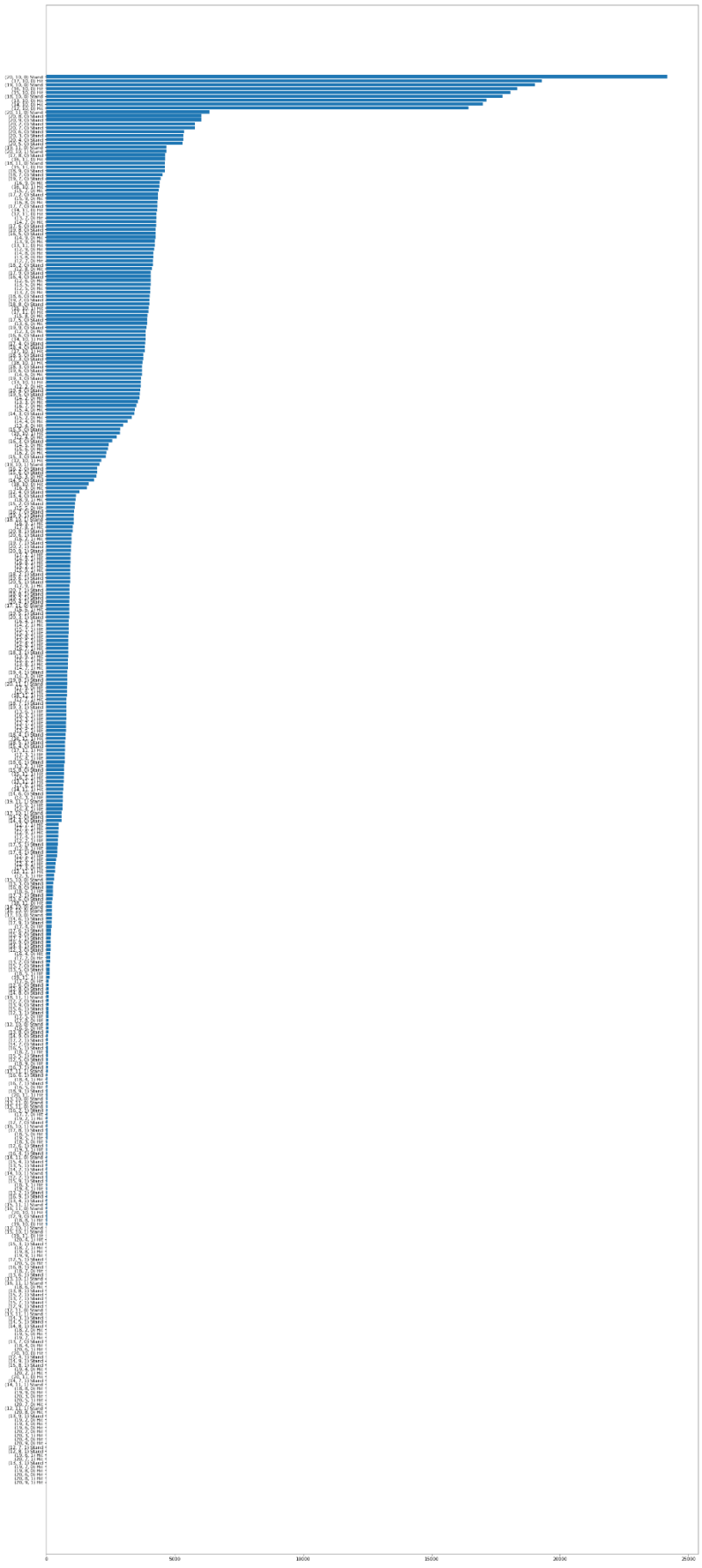

Figure 19: SARSAMAX, e = 1/k

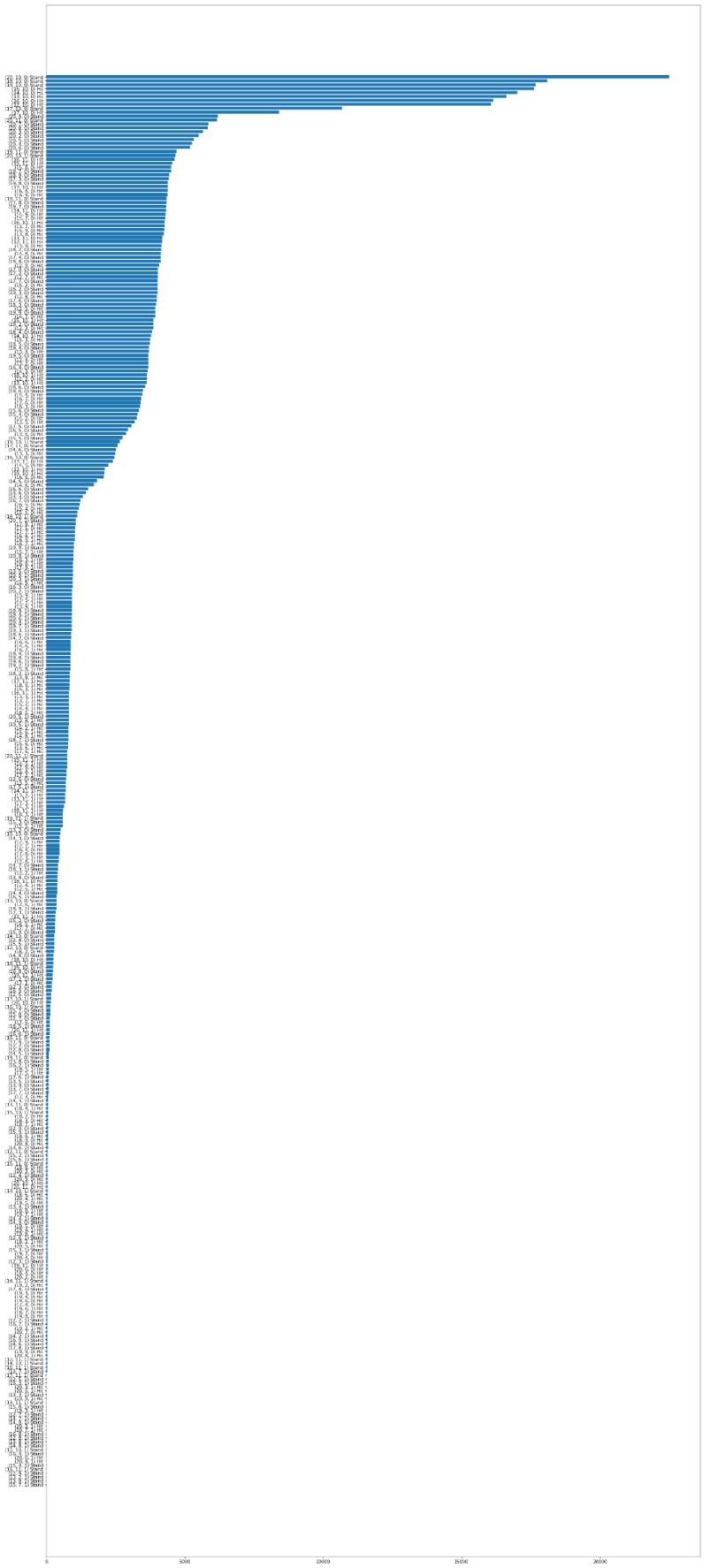

Figure 20: SARSAMAX, e^-k/1000

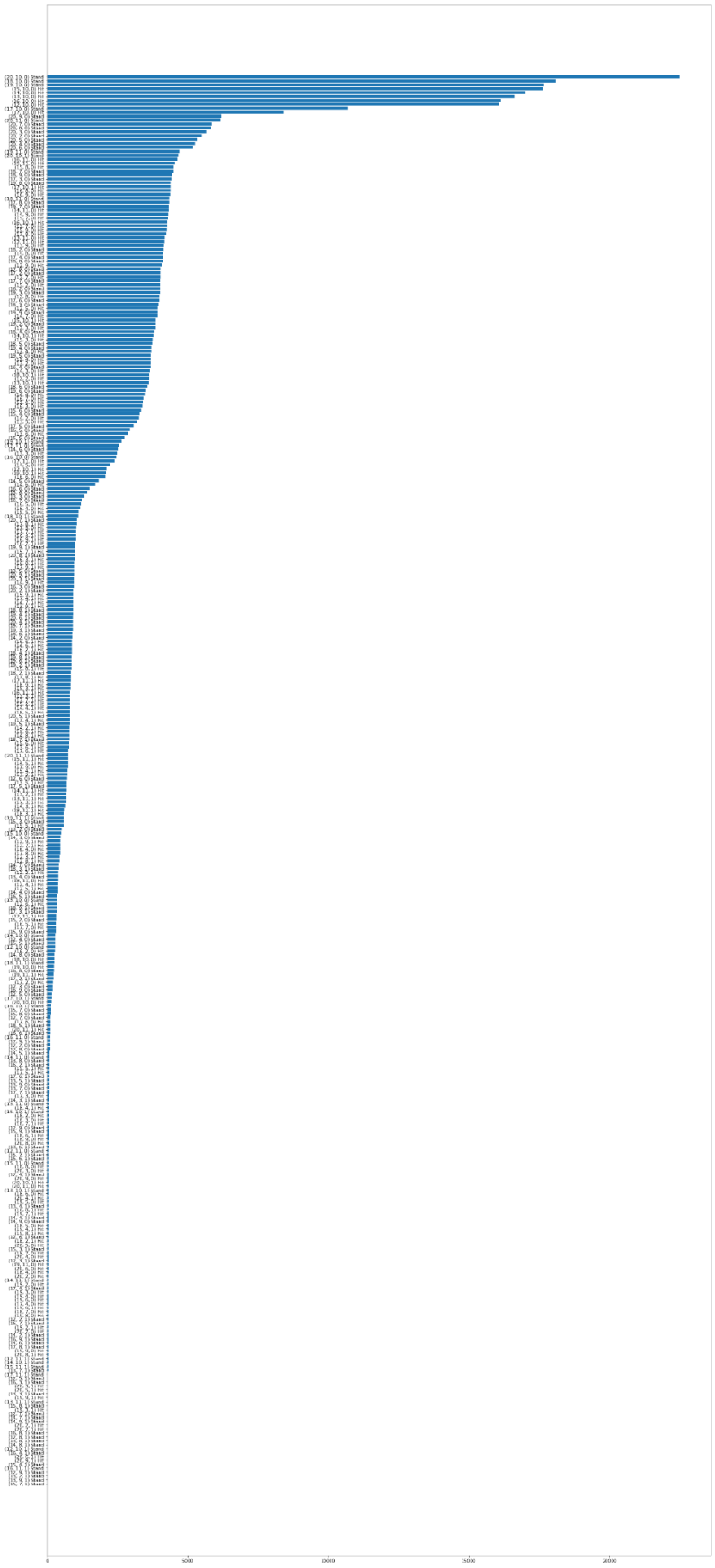

Figure 21: SARSAMAX e^-k/10000

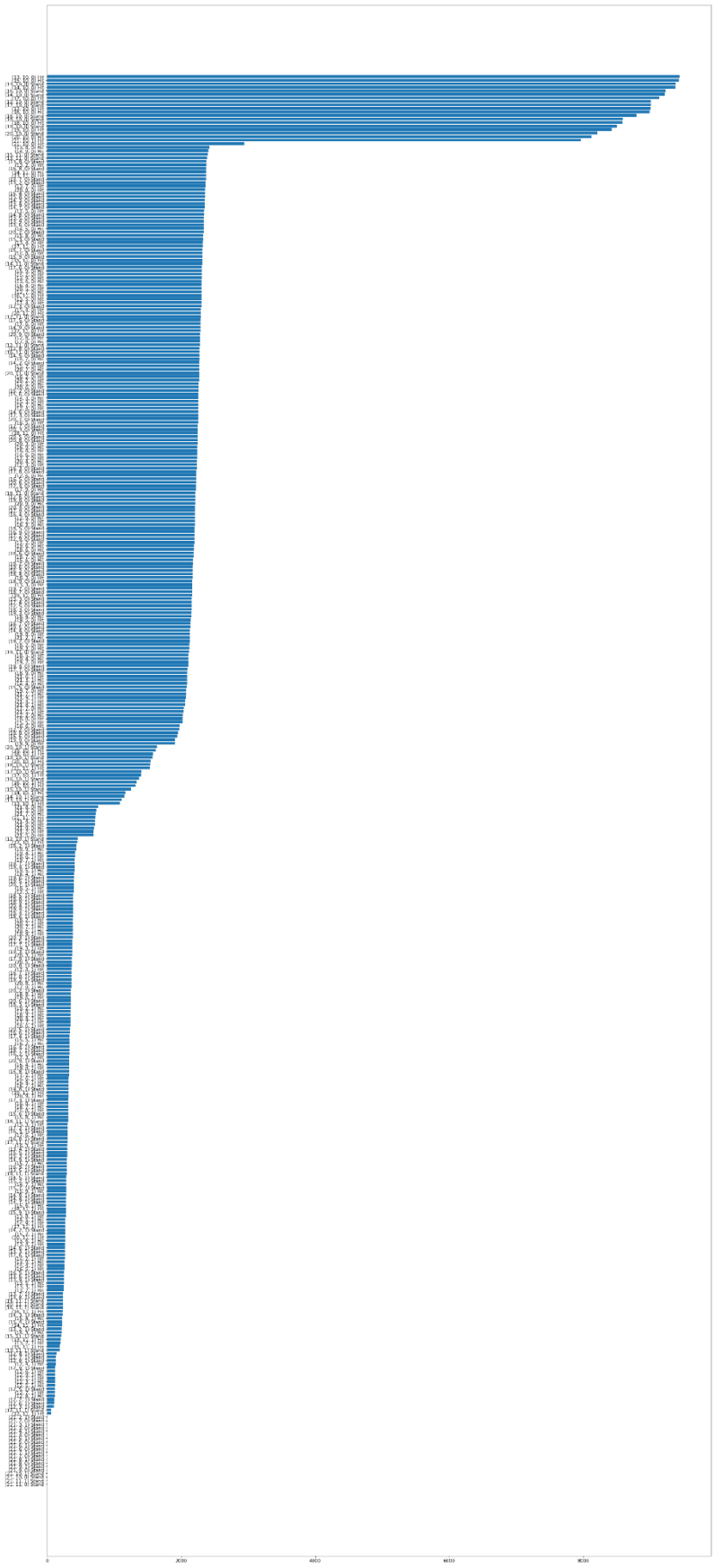

Figure 22 Monte Carlo, Exploration Starts

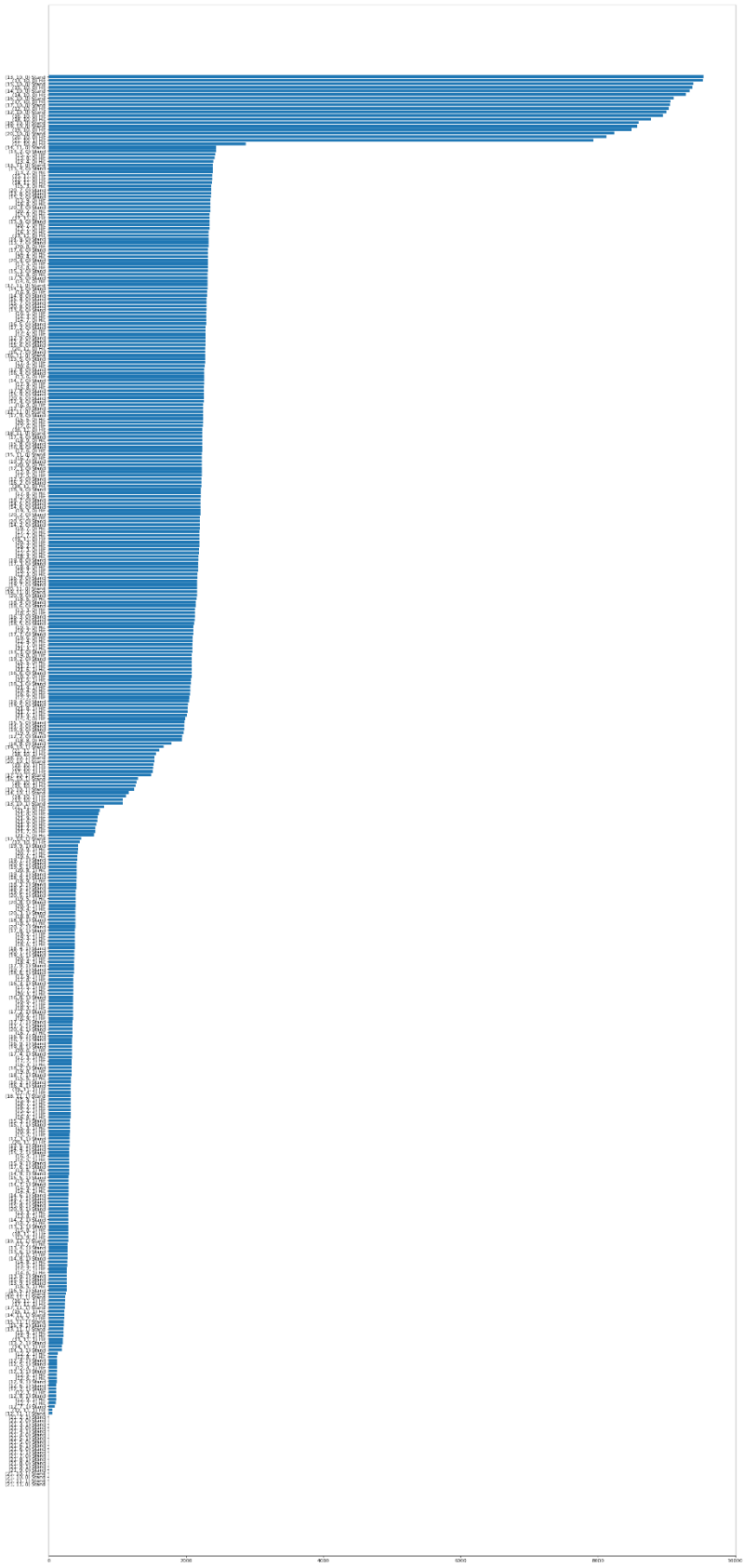

Figure 23: Monte Carlo e = 1/k

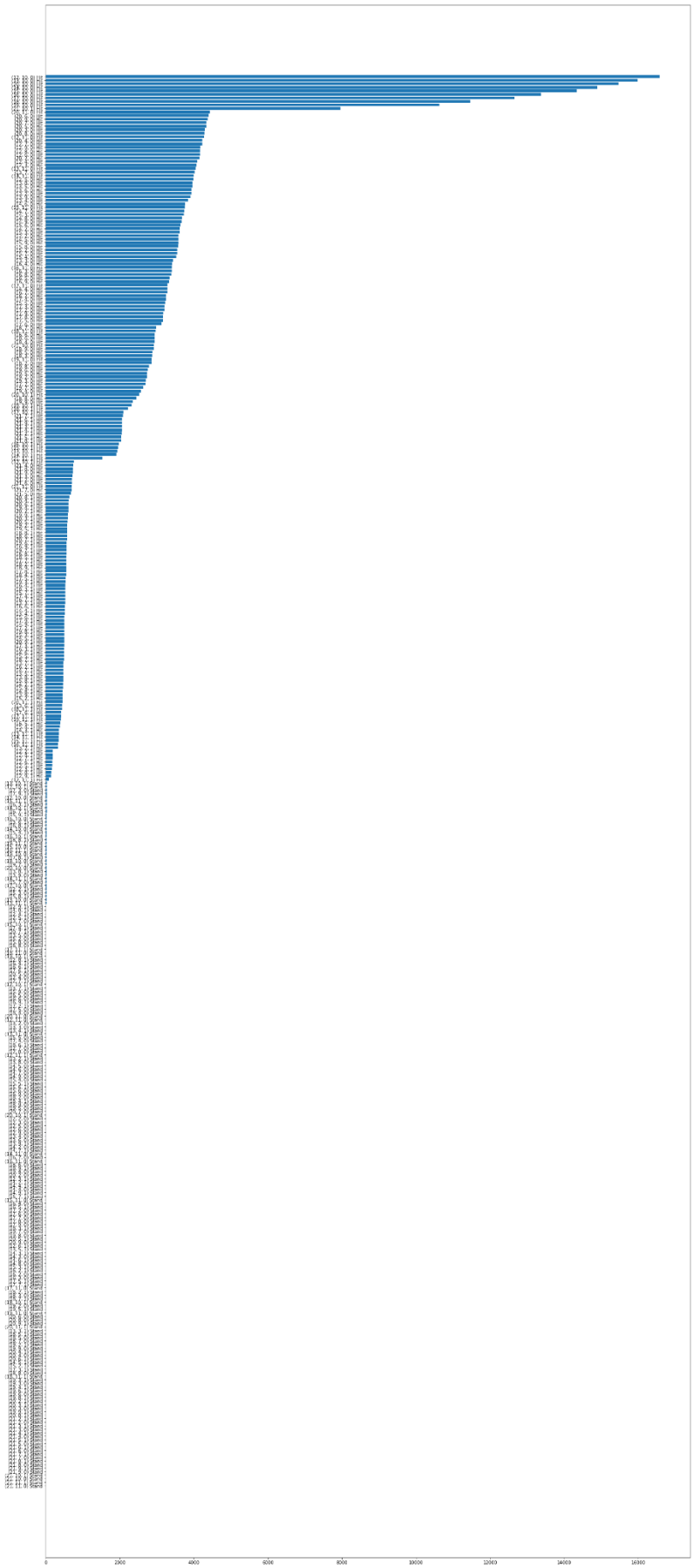

Figure 24: Monte Carlo e^-k/1000

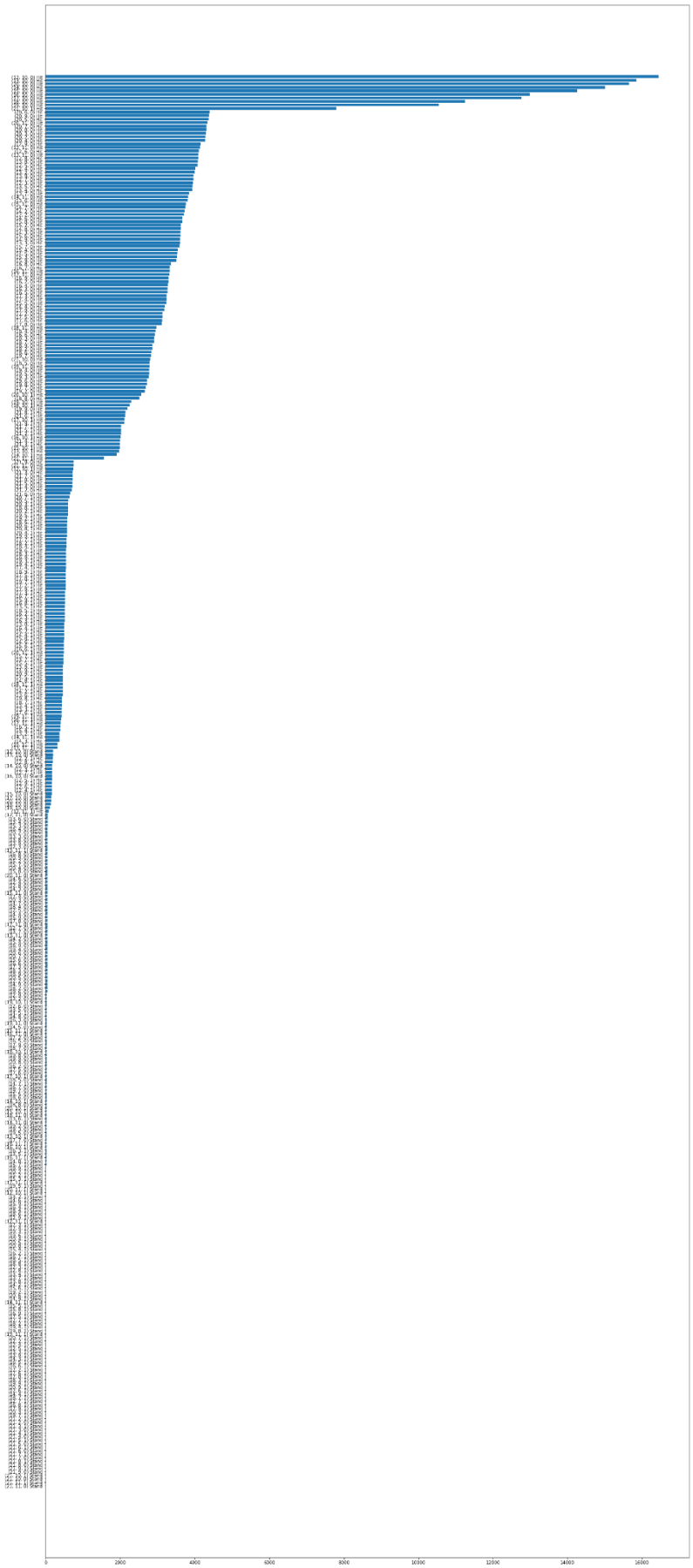

Figure 25: Monte Carlo e^-k/10000

Q-Values

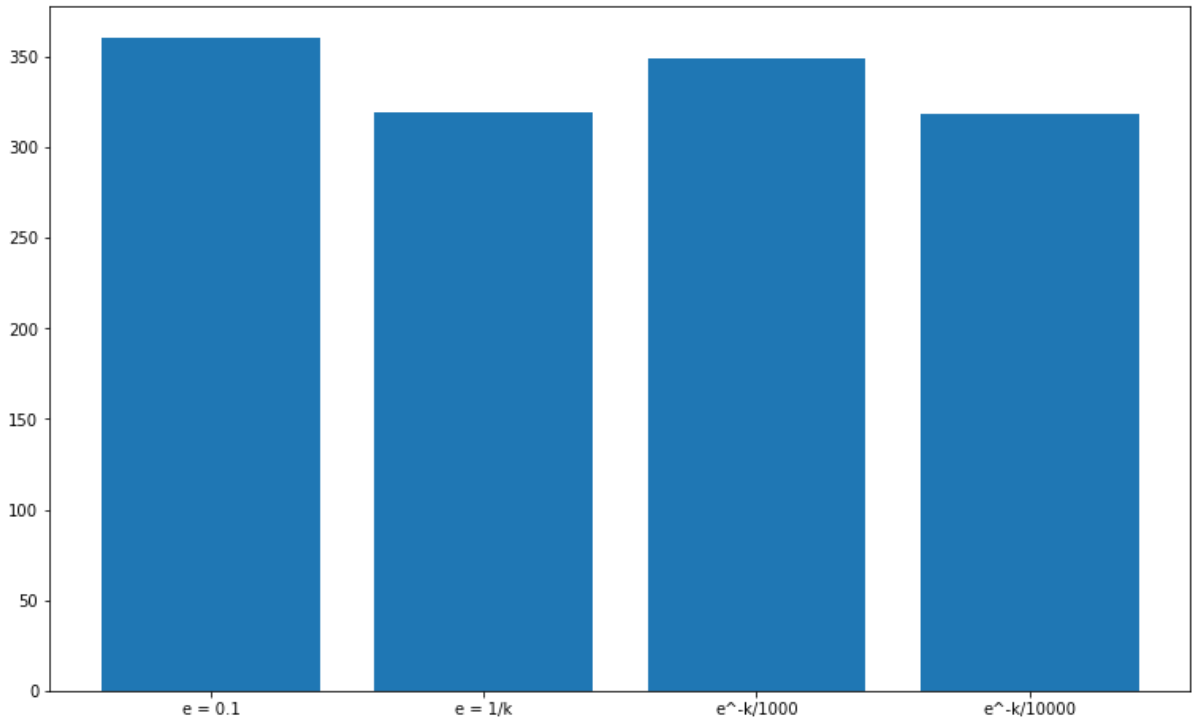

Figure 26: SARSA

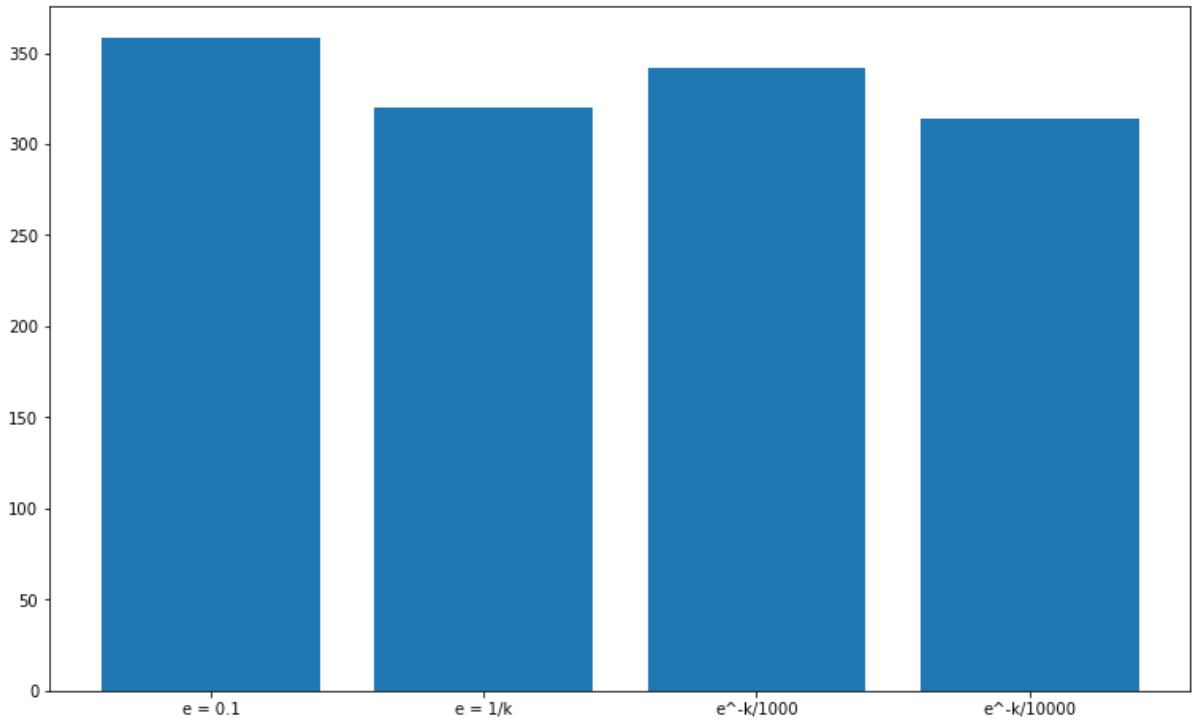

Figure 27:SARSAMAX

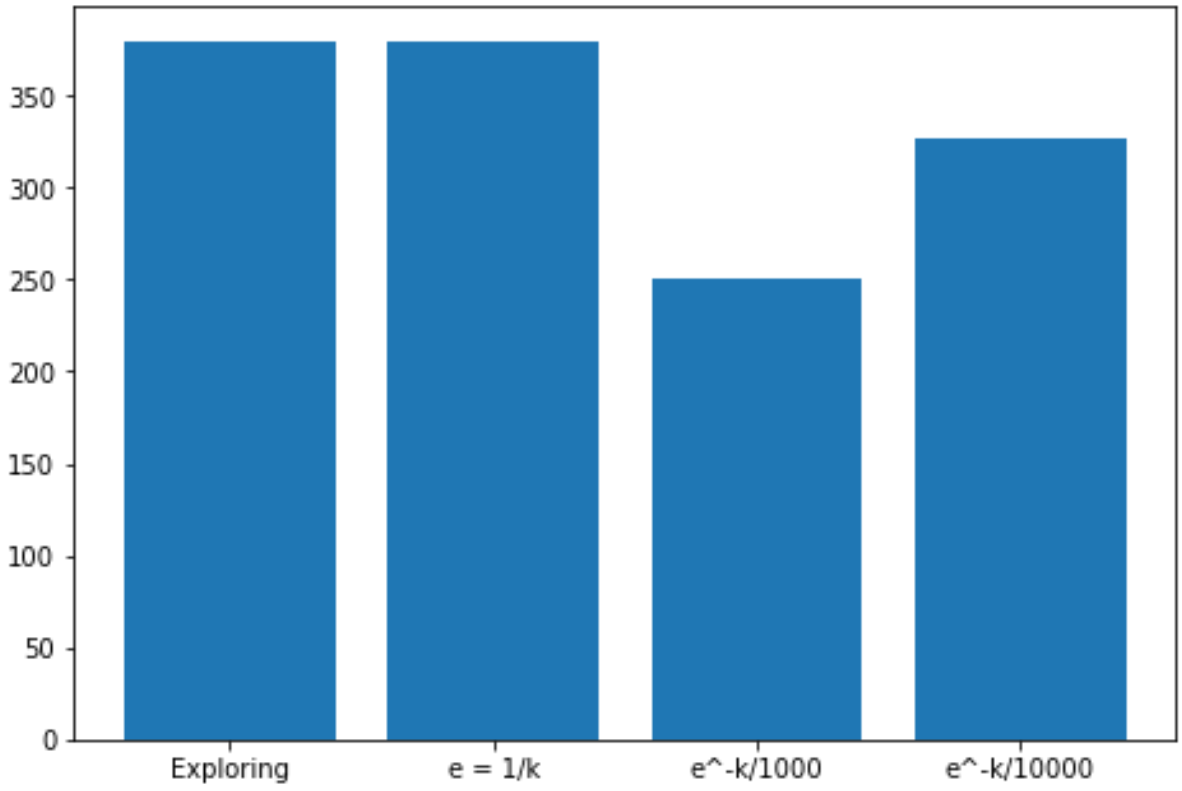

Figure 28: Monte Carlo

Blackjack Strategy Tables

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	H	S	S	H	S	S	H	S	S
17	S	H	S	S	H	H	H	H	H	H
16	H	S	S	H	H	H	H	H	H	H
15	H	S	H	H	S	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	S	H	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	H	S	S	H	S	S	H	S	S
17	S	H	S	S	H	H	H	H	H	H
16	H	S	S	H	H	H	H	H	H	H
15	H	S	H	H	S	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	S	H	H	H	H	H	H	H

Figure 29: SARSA e = 0.1

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	H	S	S	H
16	H	S	S	S	S	H	H	H	H	H
15	H	H	H	H	S	H	S	H	H	H
14	H	S	S	H	S	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	H
18	H	S	H	S	H	S	S	H	H	H
17	H	H	S	H	S	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 30: SARSA e = 1/k

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	H
16	H	H	H	H	S	H	H	H	H	H
15	H	S	H	H	S	H	H	H	H	H
14	S	H	H	S	H	H	H	H	H	H
13	H	H	H	S	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	H	H	S	S	S	S	S	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 31: SARSA e^-k/1000

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	S
16	S	S	H	S	H	H	H	H	H	H
15	H	H	H	S	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	S	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	H	S
18	S	H	S	H	S	S	S	S	H	S
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	S	H	H	H	H	S	H	H	H
14	H	H	S	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 32: SARSA e^-k/10000

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	S
16	S	S	H	S	H	H	H	H	H	H
15	H	H	H	S	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	S	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	H	S
18	S	H	S	H	S	S	S	S	H	S
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	S	H	H	H	H	S	H	H	H
14	H	H	S	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 33: SARSAMAX e = 0.1

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	H
16	S	H	H	S	S	H	H	S	H	H
15	H	H	S	S	S	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	H	H
18	S	S	S	S	S	S	S	H	H	H
17	H	S	H	H	H	S	H	H	H	S
16	H	H	H	H	S	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 34: SARSAMAX e = 1/k

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	H	S
16	S	H	H	S	H	H	H	H	H	H
15	S	H	S	S	S	H	H	H	H	H
14	H	H	H	H	S	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	H	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	H	H	H
17	H	H	S	H	H	H	H	S	H	H
16	H	H	H	S	S	H	H	H	H	H
15	S	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 35: SARSAMAX e^-k/1000

Player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	H	S	S	H
16	S	S	S	H	S	H	H	H	H	H
15	H	S	H	S	H	H	H	H	H	H
14	H	H	H	S	S	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	S	H	H	H	H	H	H

Player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	H
18	S	H	S	S	S	S	S	S	H	H
17	H	H	H	S	H	H	H	H	H	H
16	H	H	H	H	S	H	H	H	H	H
15	H	H	H	H	S	H	H	H	H	H
14	H	S	H	H	S	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Figure 36: SARSAMAX e^-k/10000

Dealer Advantage Graphs

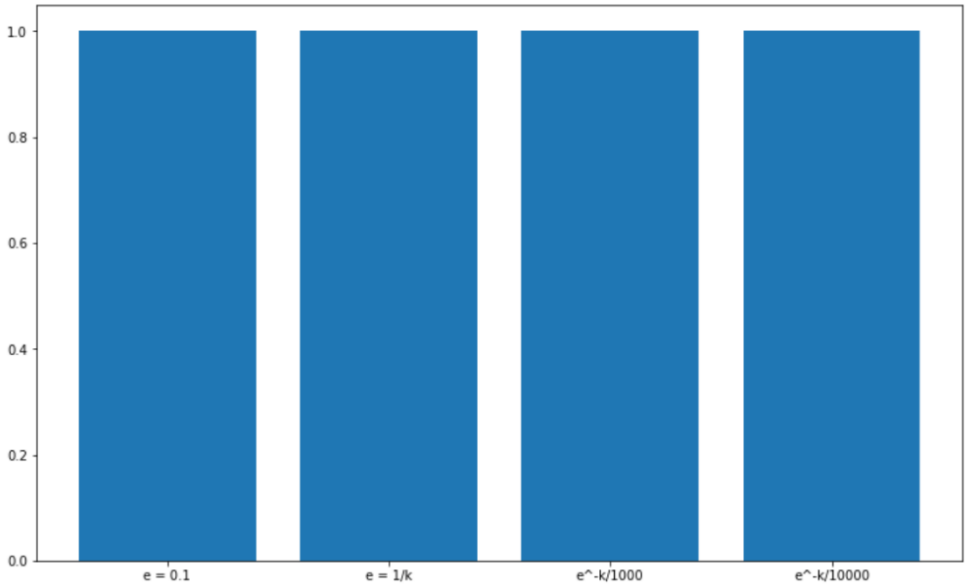

Figure 37: SARSA

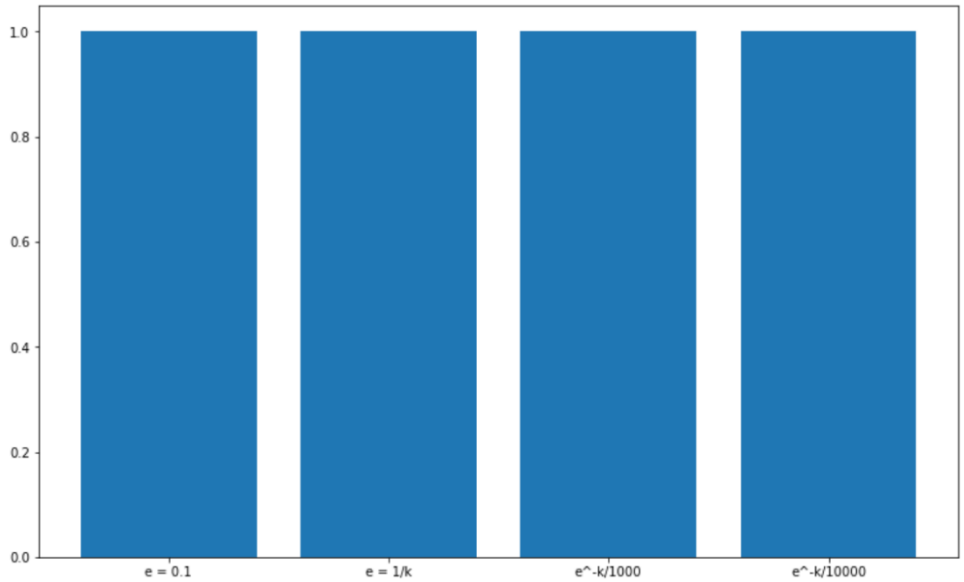

Figure 38: SARSAMAX

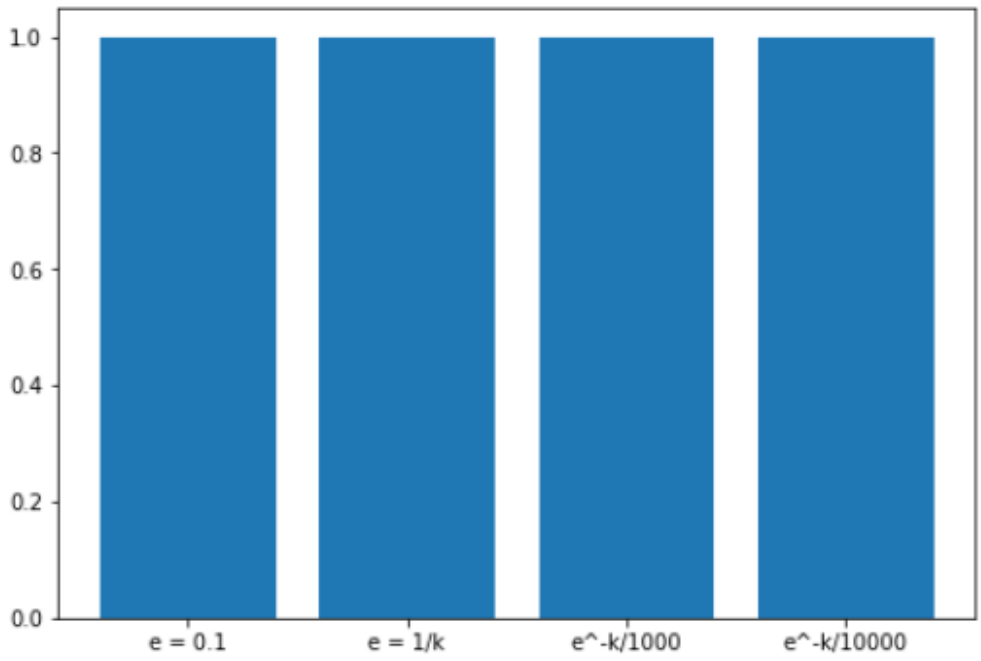

Figure 39: Monte Carlo

Analysis and Evaluation of Graphs

It is easy to decern from the graphs that the less explorative a graph is the greater the win chance on average. While exploration may be good in a game like monopoly for example, where being a bit spontaneous may be of advantage, in Blackjack, having a 20 and then hitting just for the sake of exploration is never going to be the ideal play. When looking at these win graphs, one may think that none of the algorithms are working exceptionally well. However, being that the best strategy for Blackjack has a win rate of less than 50% and playing at random yields around a 30%-win rate, it is natural to expect our result to be in the high 30 low 40 range.

This is also reflective with the state action pairs, with less explorative algorithms being steeper. Although all algorithms learnt the basic strategy of, hit on low cards stand on high cards, less explorative algorithms would choose these safer options more.

It was noted from the state action pairs, that the algorithms tended to play more aggressively (hit more) when they had an ace, compared to what they would do with the same card count without the ace. This is quite similar to what humans do when they have an ace in their hand, despite this, hitting on 20 even with an ace is strongly unrecommended.

For the population of the Q table, we disregarded state actions that had less than 10 selections, this was done to exaggerate the results. In reality, if a state action is only hit 9 times in 500,000 episodes where the average amount of selections is around 3, it might as well have been hit 0 times.

With exploration policies a policy of 1 would mean that the algorithm is playing random, and 0 would mean it never takes risks. All the dynamic policies start initially with a policy of 1, however as the number of episodes increases the exploration rate decreases, each varying how fast it approaches a ‘0’ learning rate. e^-k/10000 approaches 0 very fast so it has little time to explore a lot of the state action pairs, 0.1 is static and will always choose randomly 10% of the time so it is hitting all the state action pairs is given.

As expected, when it comes to comparing the different policies for each configuration, the strategy shifts when an ace is considered to be the value, one. The agent will still mostly hit when he has a lower card total (12 - 14) and stand when the total is 19 and over, but for the values between 15 and 18 start changing in favour of hitting more if any Ace is set as the value 11. Comparing the different configurations themselves, in a configuration that keeps a higher exploration rate such as figure 33 and figure 29, the policy tends to favour hitting more when the total is around 19. On the other hand, a policy that favours exploitation more (such as figure 32 and figure 36) ends up having a policy more logically sound in what the optimal strategy should be.

With regards to the dealer advantage, for each configuration little to no difference can be shown. This is expected as all configurations running at 500k episodes would lead to all of them converging, hence the very minor difference.

Bibliography

[1] BAJADA DYNAMIC PROGRAMMING NOTES

[2] BAJADA TEMPORAL DIFFERENCE METHODS

[3] BAJADA FUNCTION APPROXIMATION NOTES

[4] BAJADA MONTE CARLO METHODS NOTES

Plagiarism Declaration Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), 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 making a false declaration may include, but are not limited to, 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).

Kian Parnis	
Student Name	Signature
Matteo Sammut	
Student Name	Signature
Mario Vella	
Student Name	Signature
Student Name	Signature

ARI2204	ARI2204 - Assignment Submission
Course Code	Title of work submitted

19/5/2022
Date

Distribution of Work

Kian Parnis

- Implementation of SARSA and SARSAMAX
- Documentation of SARSA and SARSAMAX

Mario Vella

- Implementation of Monte Carlo
- Documentation of Monte Carlo

Matteo Sammut

- Implementing Blackjack environment
- Coding of the Evaluation Section
- Documentation of Results and Evaluation