Module Name: ME5406, Introduction to Reinforcement Learning

Project Title: Project 1, Frozen Lake

Student Name: Lee Hong De Moses

Student Number: A0199876X

Student Email Address: E0406857@u.nus.edu

Overview

For this project, each algorithm was improved through independent learning. In the first section of the report, we will analyse only the “regular” algorithm, based on the pseudo code given with alterations in the epsilon / learning in terms of decay or warm up respectively. In the next sections, we will go more in depth on the realisations and the reasoning for implementations put in place to solve the Frozen Lake problem. There will also be graphs provided in the Appendix of the report for visualisation (and is not counted as part of the report page limit of 10 pages).

Performance of Reinforcement Techniques

Looking at the 3 Reinforcement Techniques, for Environment 1 (4x4 Grid), we can see from **Appendix 1** that the policy for all 3 algorithms converges. We can see that the rewards per episode for each algorithm converages to a value indicating that the 3 algorithms are able to solve for the 4x4 grid environment. Coupled with analysis of the Qtables, we also notice that by taking the max Q value of the actions in each state, we will be able to take the right actions to reach the end state goal. Hence it could be said that the agent has found the optimal policy since each state gave “instructions” to the agent on how to get to the end goal.

When applying these algorithms to Environment 2 (10x10 grid) however, the results were unfortunately not the same. It was impossible for the Monte Carlo Algorithm to solve Environment 2, most of the time leaving many states unexplored. Increasing the number of episodes for the simulation did improve results and the agent was able to explore more states, however, looking at the rewards in **Appendix 2** for Monte Carlo, we can see that the rewards are all negative, indicating that the agent never did find the end goal state.

Another thing to take note of is the speed at which the policy converges for each algorithm. For Environment 1, We can see that for the *SARSA* and *Q-Learning* algorithms, the policies took less than 20 episodes to converge, however for *Monte Carlo*, it took significantly longer - about 400 episodes before rewards started to become positive. For Environment 2, *Monte Carlo* did not converge to the optimal policy, and *SARSA* became too computationally expensive to even continue simulation. *Q-Learning* was the only Algorithm that converged and continued to perform similarly to that when applied to Environment 1. The average time taken per episode was also analysed, and in general for both Environments, time taken was ranked as such: ***Monte Carlo < Q-Learning < SARSA.***

Similarities and Differences (Explanation)

For a small grid environment like Environment 1, it is not surprising that the 3 algorithms were able to solve the optimal policy. The Frozen Lake Problem has a state space with sparse rewards, so the Q values of each state-action will be more heavily penalised (become more negative) given that the probability of falling into a hole yielding negative rewards is much higher than reaching the end goal state, which yields a positive reward of +1. In a 4x4 Grid Environment, the probability a random action sequence reaches the end is at worst 1/(4^6) = **1/4096.** In a larger 10x10 Grid Environment, the probability is alot smaller, at worst 1/(4^20) = **1/1,099,511,600,000**. This means that the agent is **1,048,576 times** more unlikely to reach the end goal state. Hence, though in Environment 1, all 3 algorithms were able to solve the Frozen Lake problem, not all 3 were able to solve the problem when simulated on Environment 2. This also explains why the rewards for *Monte Carlo* and *SARSA* were negative for all episodes when simulating for Environment 2, because the agent never reached the end goal state and hence never had positive rewards.

The speed of policy convergence can be explained easily by analysing the algorithms. The 2 Environments pose different problems, hence we will isolate simulation for Environment 1 to analyse the speed of convergence. Analysing the individual codes of the algorithms, we can understand why *SARSA* and *Q-Learning* quickly converged. This is because the agent didn't need to wait till the end of anepisode to update the Q values. The agent was "looking ahead" at every step of the episode and consistently updating the Q values to prompt the agent to take better actions towards the end goal state. Our data analysis also supports this hypothesis: The agents for *SARSA* and *Q-Learning* had already explored the entire 4x4 state space by the 10th episode, while the *Monte Carlo* agent had only explore 11/16 environment by the end of the 1000 episodes.

Now, Isolating Environment 2, the performance of *Q-Learning* remained consistent and had converged within 20 episodes. The *SARSA* and *Q-Learning* are similar, but yielded different results. The *SARSA* algorithm updated the Q values based on the next step action and state, but the *Q-Learning* algorithm updated its Qtable based on the max Q value of the actions in the next step state. This meant that the Q values of the *Q-learning* algorithm was more likely to have a clearer path, since the Q values of a specific action at a specific state will be updated to have a greater Q value than other actions at the same state. On the flip side, the Q values are more likely to oscillate for each step of the *SARSA* simulation, hence the agent would thus move within visited states more regularly than the *Q-Learning* agent. This also explains the reason why the computational time of simulation of *SARSA* was much longer than *Monte Carlo* and *Q-Learning*.

Tying in the above points, we can now explain why the Average Time Taken per Episode for the different algorithms are as such. *Monte Carlo* takes the shortest amount of time because each episode completes the moment the agent reaches an **terminal state (hole / end goal state)**. However, for *SARSA* and *Q-Learning*, the algorithms could be visualised as a sub-simulation within a main-simulation. Hence, each episode will end only when the sub-simulation and main-simulation agents have reached a terminal state. *SARSA* simulation took longer than *Q-Learning* because of the oscillation in the Q values, causing the agent to take unnecessary steps, hence leading to longer episode durations.

Performance Conclusion

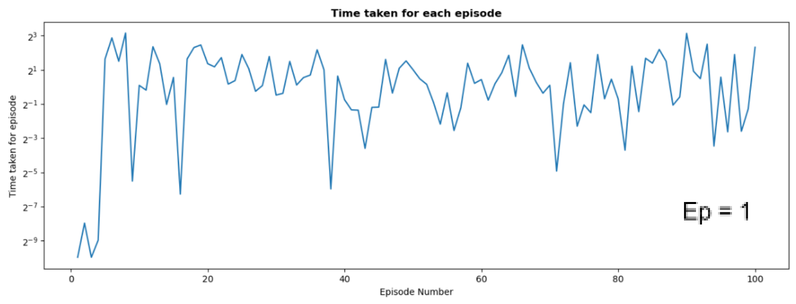
While the performance of the 3 algorithms performed relatively well on the 4x4 grid environment, having implemented them on the 10x10 grid environment allowed us to evaluate the performance of the Reinforcement Learning Techniques more in depth. It is clear that the performance of *SARSA & Q-Learning* in terms of speed of convergence and exploration efficiency is much better than *Monte Carlo*. In terms of scalability, only *Q-Learning* is seen to be robust enough to be able to solve the Frozen Lake 10x10 grid problem. Analysing computational speed, it would vary depending on the environment being used, but generally, *Q-Learning* seems to have good computational speed for both environments.

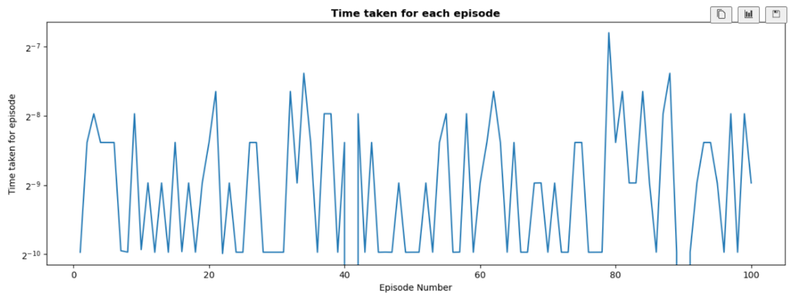
In general, *Q-Learning* seems to be the best algorithm thus far, and is able to provide a good balance between convergence speed, scalability, and exploration efficiency. Due to time constraints, the other performance metrics such as generalisation, learning and sample efficiency were not explored, but would also be points of consideration when choosing a efficient and suitable algorithm for problem solving.

Unexpected Outcomes (With Explanation)

There were unexpected outcomes during the analysing of the simulations of the 3 algorithms. These unexpected outcomes were mainly due to the settings of the parameters such Epsilon and the Learning Rate. However, through multiple testings and tweaking, I was able to formulate some hypothesis as to the behaviour of the agent in certain simulations. Some of these unexpected outcomes had allowed me to derive the explanations in the previous section, but I will talking more in depth about these unexpected simulation results.

One of these unexpected outcomes was seen when running the *Monte Carlo* simulation on Environment 2. Epsilon was set to 0.1 and Gamma was set at 0.9. I had set my number of episodes to be 1000. I noticed that the agent was taking a significantly longer time to complete simulation and when the simulation was completed, many Q values in the Q table were 0 (unexplored). Using pygame package to render the environment and agent, i noticed that the agent was moving back and forth between the states visited and new states - and hence i decided to try to test for a similar behaviour in my smaller 4x4 grid environment. I suspected that the reason for the higher duration for simulation was because the agent was less likely to explore and because the Q values of taking an action backward was lower, hence the agent by having a higher probability of taking the greedy action, was more likely to choose a step backward rather than explore the environment. To show an extreme example of this, I analysed two scenarios in Environment 1 where Epsilon was 0.01 and 1.





The Time Taken per Episode for Ep = 0.01 was orders of magnitude greater than when Ep = 1.

Looking at these 2 extreme scenarios, it becomes clear that choosing a suitable Epsilon value is vital in training the agent to solve the Frozen Lake Problem. Ideally, we would want the agent to explore the state early on in the simulation, but take the greedy action more often in later parts of the simulation. This will be talked about and implemented in the next section of the report.

Another unexpected outcome was the increased duration taken by the *SARSA* agent to complete simulations on the 10x10 grid environment. The explanation for *Monte Carlo* and *SARSA* is similar. Because the Q values were constantly being updated, and because of the sparse rewards in the environment, each visited state became more and more negative to the point where choosing a backward action may be more optimal as the algorithm chose to select the greedy action. The only point in time the agent becomes more “confident” in moving towards the end goal state is after it has visited that state and received the +1 reward and backpropagated the positive rewards back to earlier states. However given such a large 10x10 state space, it may take many episodes before the agent even finds the end state goal, and by then, the Q values may be too negative that reaching the end goal state would not change these Q values as much anymore, especially if learning rate is constant. This is the reason why in the next section, you will see the implementation of the “warm up” learning rate under the *SARSA* algorithm section.

Difficulties & Self Initiatives

Algorithmic specific difficulties was especially prominent in the extended implementation. This drove me to implement some initiatives, and hence I am combining difficulties and self initiatives (talk about problems and how a solution was derived). Apart from difficulties in the extended implementation, it was challenging to implement this project because of the lack of experience and skillset. I will elaborate in the following.

Skill Level

There was a significant level of difficulty in implementing this project due to the lack of experience in python. Much online resources was used for learning and research and alot of documentation was read. Online video resources like Youtube was used as well to understand certain concepts. On top of that, the RL notes were useful and having a full picture of the math along with the given pseudo code, crafting the algorithm became a task that seemed doable. Due to the lack of experience, alot more time was spent figuring out how to implement certain functions, and over countless attempts, trial and errors, the project was finally completed. Oh, and an honorary mention that chatGPT was also vital in assisting quicker learning (:

Originally, the entire Python Notebook was only made up of functions, with only the environment being a class. However, after realising that extended implementation was needed, it was clear that having each component as classes would shorten the duration for implementation of the 10x10 grid. This proved slightly challenging as i had no prior experience with classes and it took a few test Python Notebooks with examples before i could understand how classes actually worked. However, i don't regret it as i can see the usefulness and elegance in the use of classes now and will likely use it again in the future.

Extended Implementation

More in depth as touched on before: It became quite challenging to solve the extended implementation when the grid had now expanded to a 10x10 state-action space. In a 10x10 grid space, it was alot more difficult for the agent to find the end state especially because it is the only state that provided a positive reward. Hence the only time when the Q(s, a) of any state will start to turn positive will be the moment the agent has found the end goal state. Even then, it is quite easy for the Qtable to remain majority negative as each episode iteration that ended at a hole would cause Qvalues to become even more negative. Hence the self initiative for each of these algorithms was to have a better algorithm. The common change for all 3 algorithm was to have an exponential epsilon decay as the number of episodes increased starting from the max epsilon (1) till the min epsilon. This meant that at the early stages of simulation, the agent was able to explore the state space more freely and hopefully find the end goal state. But through the number of episodes of simulation, the agent will start to choose the optimal Q(s, a), allowing the agent to approach find the more desirable state each time. I will talk about the individual algorithms and the difficulties faced with each of them as well as the self implemented initiatives.

Monte Carlo

When increasing the state size, there was a huge problem the initial algorithm had in finding the end state. Implementing epsilon decay did help the agent explore more states, but due to the size of the environment, it took multiple attempts in tweaking the decay rate before the end state could even be found. Even then, there was still a high probability the agent did not find the end state. As such, we had to bump up the number of simulations being runned. The idea was to have a decay rate that was low enough so that in the first 30% - 40% of the episodes, the agent was more inclined to explore the space. Approaching the 50% mark, the agent would start to take the greedy action towards the end goal state. By the 80 - 90% mark, the agent would have found the end goal and would be taking the greedy action most of the time, updating the Qtable to be more positive values. Based on online reading, there are other ways to solve this problem, such as function approximation using neural networks, or monte carlo tree search, however due to time constraints, i wasn't able to implement these strategies. Instead i decided to increase the number of episodes in the simulation, which arguably may not be the best method since it is computationally expensive and requires a significantly greater amount of time to complete the simulation.

SARSA

The simulations took extremely long especially since the SARSA algorithm works is basically a sub simulation within a main simulation - since we are looking forward one step up till a terminal state in the sub simulation and then updating the main step based on the updated Q values until the main simulation terminates. The number of actual steps taken by the agent is hence plentiful, and in a 10x10 grid environment, due to the sheer size, it was alot harder for the agent to find the end state, and simulations were taking alot longer to run. As such, there were a few implementations in place apart from having epsilon decay.

Firstly, I implemented a fixed number of step sizes, to reduce the computational duration required for the algorithm. Secondly, a 'warm-up' learning rate was implemented.

Originally, i implemented a decaying learning rate, and when plotting graphs, realise that more often than not, the rewards of the agent tended to a negative value. This is not surprising as this means that more often than not, the agent was reaching a terminal state that was a hole much more often than the end state goal. Because learning rate decay meant that the agent would give more weight to the negative values (holes) at the earlier stages, when the end goal is found, there will be less weight placed on the positive reward. Hence i decided that while i want my agent to explore the environment in the early stages, I also want less weight to be placed on the negative rewards it received. However as number of episodes went by, and when the goal state is found, i want the agent to place more weight on that positive reward to update the Q table with more positive values. The learning rate would increase linearly from the minimum learning rate to the maximum learning rate on the last episode of the simulation. The difficulty with having these new parameters was then fine tuning them, to ensure there was a good balance between exploration-exploitation as well as the learning of the agent, to prevent the agent from oscillating or diverging from the optimal policy.

It was challenging when tuning the parameters. The max step size also played a role in affecting the overall rewards of the agent. Though not as obvious when running on Environment 1, it became clear when implementing on the extended 10x10 grid where some spaces were not explored - hence the max steps had to also be fine tuned. Eventually, we were able to get some decent results, with the end state being found and having positive rewards.

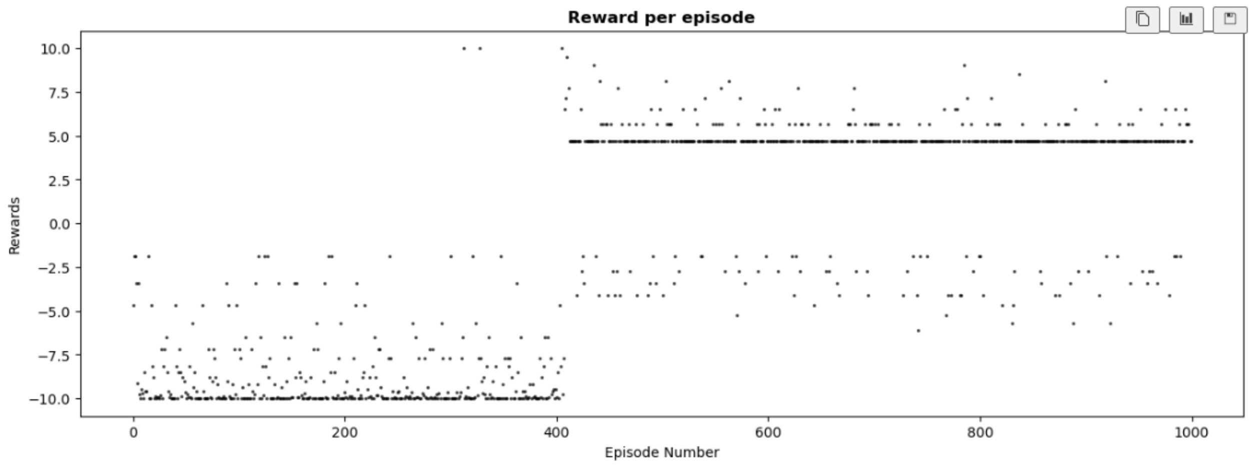
Q-Learning

The self initiative for Q-Learning was the same as SARSA. I noticed that Q-Learning was able to efficiently explore the states in the 10x10 grid environment, however rewards were still negative. Though the Q table showed that an agent would be able to take optimal actions to reach the end goal, our rewards graph was still showing negative rewards. Initially, Epsilon decay was first implemented, and this yielded more positive rewards, however still negative. Hence i decided that maybe be implementing a warm up learning rate, the results for this might improve. Though results improved, and our agent still converged to the optimal policy, fine tuning of the parameters were paramount in achieving these results.

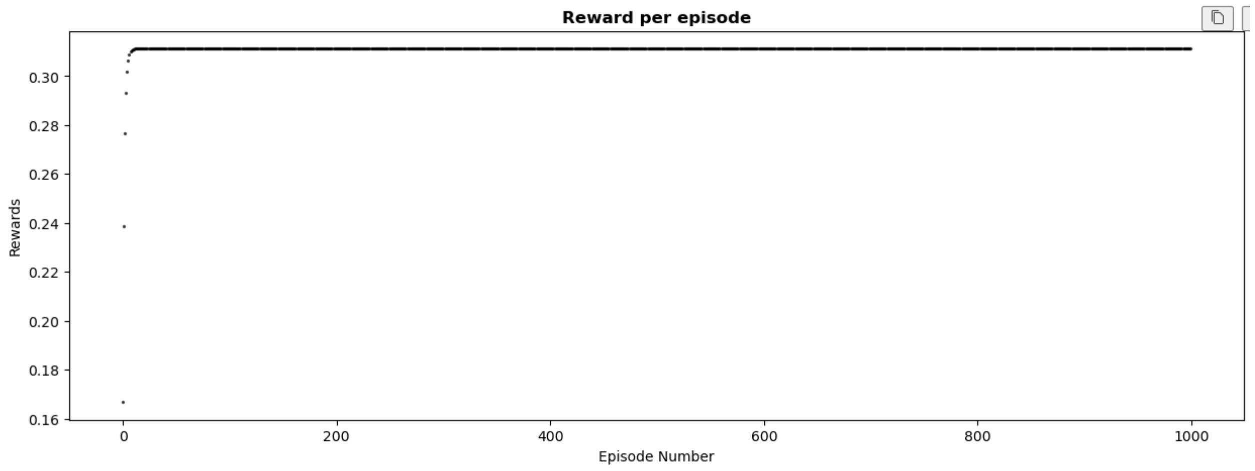
Closing Note

While there are self initiatives to improve the individual algorithms, the fine tuning of parameters is still essential in ensuring that the agent functions well and solves the Frozen Lake problem especially in a larger state spaces. However experts have argued that the fine tuning of parameters may not be the most ideal method of solving problems especially when it comes to achieving generalised artificial intelligence. The argument is that by fine tuning parameters to solve specific problems, we are forcing our model / algorithm to solve a problem that we already know the solution to - when in reality, if we want to apply these model to solve new and unique problems, then it would not be the case. Hence as we proceed to part 2 of this module, it is vital to bear this in mind whenever implementing algorithms or AI solutions such as Convolutional Neural Networks or other AI algorithms.

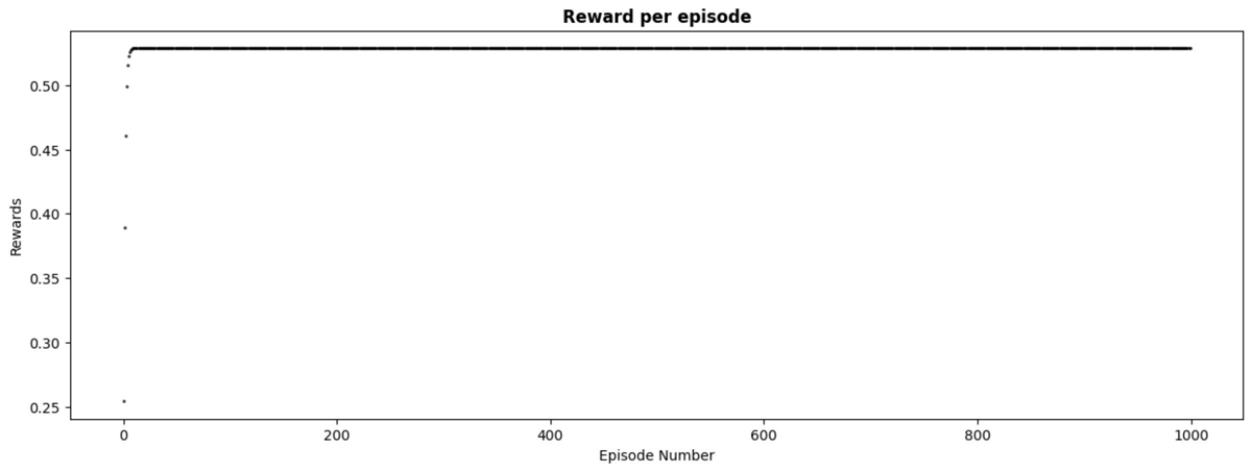
Appendix 1



Monte Carlo (Env 1): **Average Time Taken / Episode = 0.012s**

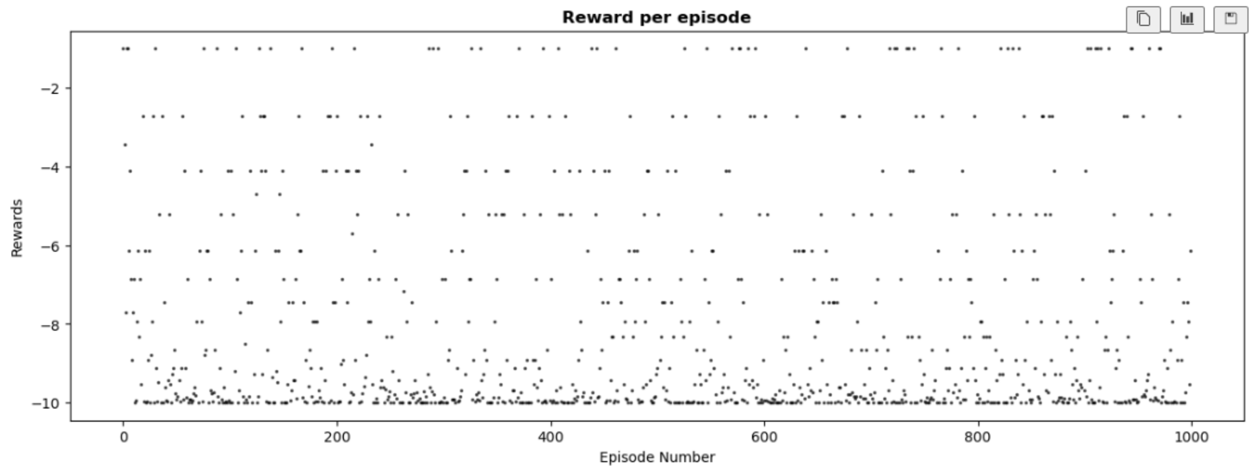


SARSA (Env 1): **Average Time Taken / Episode = 0.075s**

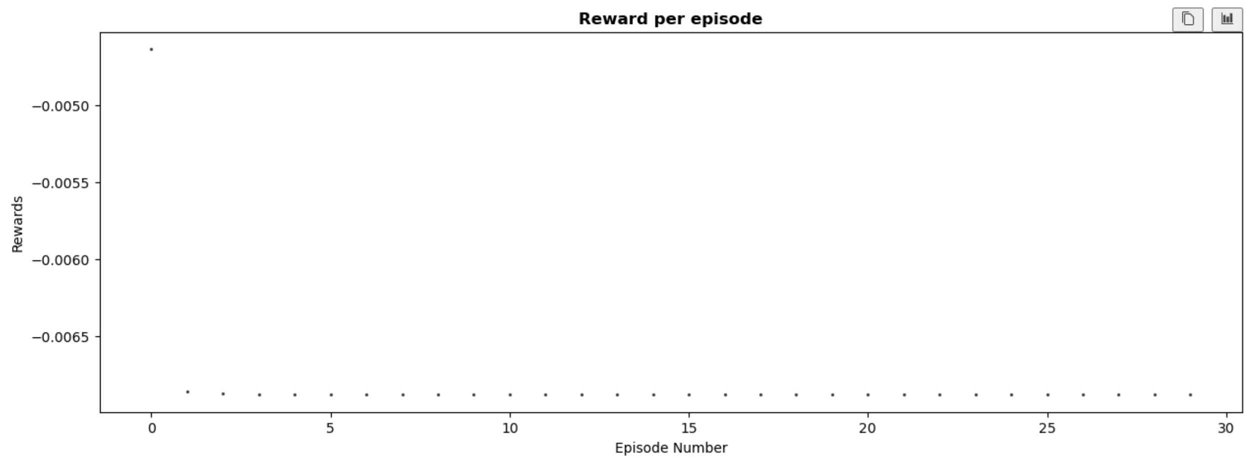


Q-Learning (Env 1): **Average Time Taken / Episode = 0.016s**

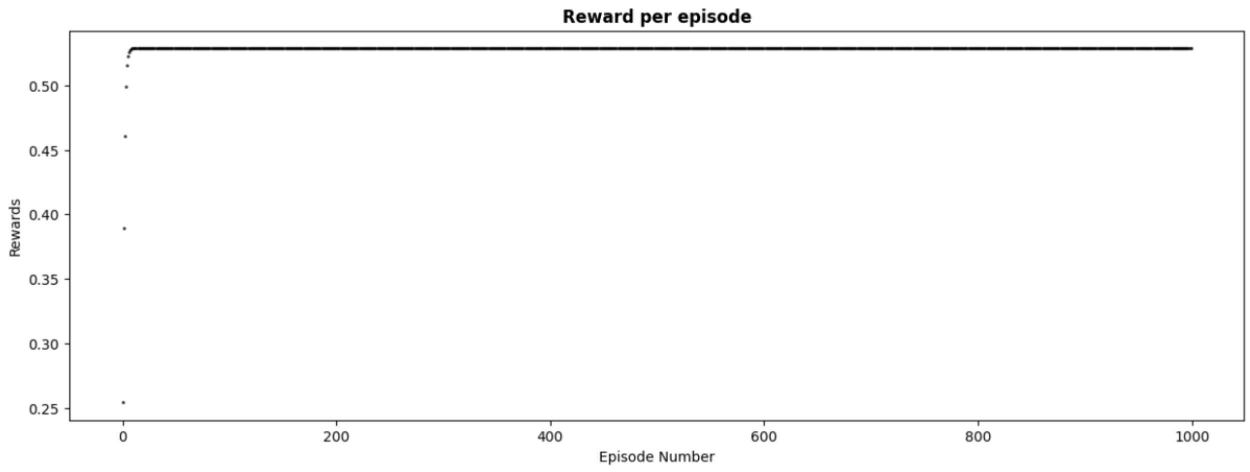
Appendix 2



Monte Carlo (Env 2): **Average Time Taken / Episode = 0.0042s**



SARSA (Env 2): **Average Time Taken / Episode = 12.2s**



Q-Learning (Env 2): **Average Time Taken / Episode = 0.122s**