ME5406 Deep Learning for Robotics

Project for Part 1: The Froze Lake Problem and Variations

Student Name: Penglin Cai

Student Number: A0313903M

Student Email: E1499298@u.nus.edu

March 3, 2025

1 Introduction

The purpose of this project is to discuss the applications of three typical reinforcement learning algorithms Monte Carlo control, SARSA and Q-learning in discrete robot path planning, specifically, in solving navigation problems in a "Frozen Lake" environment. The problem background is set as a "Frozen Lake" environment with a limited size of grids, each grid is treated as a single state for the agent while some states are covered with thin ice forming holes. The agent has four action spaces which are left, right, up and down, in which the agent is trained within the map. The ultimate objective is to train the agent to navigate itself from the starting point (top left corner) to the endpoint (lower right corner) to get the target object (Frisbee) without stepping into any holes. In this process, the agent will only gain a reward and terminate the episode when they fall into the hole (-1) and reach the goal state (+1), reward for any other cases is 0. The training environment is under sparse reward.

The project is divided into two main tasks:

- Task 1: Implement three reinforcement learning algorithms in a 4×4 grid environment, and analyze the corresponding performance differences.
- Task 2: Repeat task 1, but in a 10x10 grid with random holes whose amount is 25% of the grid size. To see the issues that occur and evaluate the performances when the state space scales up.

In order to accomplish the above concerns, this project adopts the following three reinforcement learning algorithms and discusses their performance under given environment:

- The first visit Monte Carlo control without the initiation of exploration (using ϵ-greedy strategy): With a fixed initial state, gradually approach the optimal policy by gathering the cumulative reward of the first-visit state-action pairs.
- SARSA (using ϵ-greedy strategy): With on-policy updates of the current Q value, the update process strictly relies on the actions actually taken by the agent. Exploitation and exploration are combined so that the policy can gradually converge to the relative optimal one while ensuring randomness to avoid insufficient exploration of the environmental states.
- Q-learning (using ϵ-greedy strategy): Belongs to the off-policy algorithm, that is, when updating, the current behavior strategy is not considered but only obeys "greedy" to select future state action pairs so that the optimal policy can be theoretically obtained after convergence.

2 Methodology and Implementation

2.1 Program Structure

The code for realizing this project is divided into five parts generally for modularity, organization, and reusability:

- An `Environment.py` file which simulates the "Frozen Lake" environments with different grid sizes, defines the agent action space, and visualization functions showing the training processes and policy obtained.
- A `Parameters.py` file which stores the important parameters for setting up the training such as grid size, map configuration, gamma, epsilon, etc.
- A `First_Visit_Monte_Carlo_Control.py` file which is the standard algorithms of First-visit Monte Carlo control without exploring starts.
- A `SARSA.py` file which is the standard algorithm of SARSA with a ϵ-greedy behavior policy.
- A `Q_learning.py` file which is the standard algorithm of Q-learning with a policy of ϵ -greedy behavior.

- A `main.py` file which contains the functions such as training the agent various reinforcement learning algorithms, plotting for evaluation, visualization of training, etc.

2.2 Training Environment Setup

To satisfy the requirements without using any external repositories to build the environment, the "low-level" simulation environment is built from scratch. The important main functions relating to the agent behavior will be explained as follows:

Firstly, the agent's action space and its numbers of states are defined such that the agent can move in four directions left, right, up and down with the states equal to the multiplication of the grid height and grid width. Each grid corresponds to a state s.

Secondly, two functions called `generate_4x4_environment()` and `generate_10x10_environment()` are created to simulate the environment in which the holes position are fixed for 4x4 grid map and it uses the random hole positions for 10x10 grid map. The random seed is used for the reproducibility for 10x10 grid map that every time the holes positions are fixed when using the same random seed. The agent and frisbee are placed at the top-left corner and bottom-right corner respectively for all map size when the episode starts. And most importantly, there must a `reset()` function to reset the agent back to the original starting point when new episode begins during training to ensure the consistency.

Lastly, it will be necessary to define how the agent will act and receive rewards under the action space and the given environment. The `step()` function is created to calculate the above. An array is created to initialize the agent's horizontal and vertical displacements to 0. A logical reasoning loop was introduced to determine the agent's action while ensuring that the agent does not move out of the map, updating the displacement array accordingly after taking the action. Finally, another loop was used to determine the current state of the agent to decide whether the episode should end or not. When the agent steps into holes, it gains a -1 reward and the episode ends. When the agent reaches the goal state, it receives a +1 reward and the episode ends. Any other situations will receive 0 rewards and the agent keeps executing actions until reaching one of the above conditions.

In summary, `Environment.py` file uses `Tkinter` to create a visualized "Frozen Lake" simulation environment meets the requirements for the implementation of simulation environments from low-level. It provides an intuitive experimental platform for subsequent test of reinforcement learning algorithms in the following manner:

- Modularity and parameter settings: The random seeds ensures the random distribution of holes remain consistent every time during test, and to set different parameters in organized way.
- Creation of GUI environment with Tkinter: An intuitive way to visualize the agent movements during training process and optimal route given by the optimal policy, accelerating the understanding of the performance of different algorithms.
- Unified state representation and action control: Provide convenience to the reinforcement learning algorithms to manage the environmental states and actions.
- Path record: For the later evaluations, the optimal route and least optimal route which lead agent to the goal will be record in terms of time steps consumed.

2.3 Parameters Interpretation

This section will interpret the parameters which are crucial to the performances of the reinforcement learning.

2.3.1 ϵ in greedy policy

For three different algorithms, each algorithm has a set of ϵ parameters for balancing exploitation (select the best action) and exploration (select random actions). It can be represented as follows:

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{for } a = A^* \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{for } a \neq A^* \end{cases}$$

Higher ϵ increases the probability of exploration and decreases the exploitation during training. The ϵ is set to decrease gradually as the episode increases during training to ensure the environment is explored in the early stages and utilize the policy learning to make optimized decisions in the later stages. A minimum value of ϵ is given to make sure the probability of exploration does not decay to 0, therefore preventing the local optimality.

2.3.2 Discount factor γ

Discount factor γ is used to balance the importance of current rewards with future rewards. The closer the value is to 1, the more the algorithm focuses on long-term returns. When the value is lower approaching 0, the algorithm focuses more on the current reward with "short-sighted". In an environment where longer paths or delayed rewards exist, a higher γ can make the policy more "visionary" but can cause slower convergence. In the case of this project with sparse rewards, the γ is set to a high value close to 1 since there will only be a positive reward when reaching the goal state.

2.3.3 Learning rate α

Whether SARSA or Q-learning, the core idea is to approximate the optimal policy by updating the state-action value function *Q* constantly. The learning rate determines the transfer ratio between the newly received information in the update and the current *Q* value. If the α is close to 1, the new information has a high weight in updating the current *Q* value, the value will be adjusted dramatically with new information, resulting in volatility or unstable convergence. On the other hand, if the α is close to 0, the new information has a lower weight, and the value will be adjusted slowly with new information, this can bring more tolerance to the environment noise and robustness, but with slower convergence speed. To learn the policy gradually and accurately, the learning rate is set to a mild value which is close to 0.01.

2.4 First-visit Monte Carlo control without exploring starts (ϵ-greedy behavior policy)

The algorithm is created by defining a `MonteCarlo` class. At the beginning of the algorithm, the initial function receives three parameters which are environment settings, ϵ and γ. The basic environment parameters include the map configuration, number of actions and number of states. To evaluate the training processes and the performances, several variables are recorded inducing time steps used per episode, the cumulative *Q* value per episode, the cumulative average rewards of the current episode, the number of episodes with positive rewards, number of episode with negative rewards and number of episode reaching the goal state. Several tables are created to store the Q value for every state-action pair, the number of "first-visit" of each state-action pair (i.e. only the first-visit pairs will be used to update the returns) and the cumulative returns.

After the initial setting, to solve the conflict between the exploring starts assumption and the situation faced in this project, the ϵ-greedy policy is used defined in `epsilon_greedy_policy()`, a number between 0 to 1 is generated randomly if it is less than ϵ, the random action (exploration) is executed. If the number is greater than ϵ, select the action with the largest *Q* value. If there are multiple optimal actions, randomly select one of them to have some randomness.

The training process of the agent is divided into two parts. Firstly, the function `generate_episode()` generates episode which loops through each time steps:

- Using ϵ-greedy policy to select action based on the state from last time step.
- The selected action is passed to `step()` in `environment.py` which will return the next state *s'*, reward *r* and flag indicates whether the episode is done or not.

- Add the state, action and reward into the list and starts for the next time step.

Secondly, the core function in training agent called `first_visit_monte_carlo_control()` is created which uses the episode generated to train for many episodes:

- For each created episode, traverse the episode in reverse order, compute the return *G* and update the *Q* table, using the formula to compute return for each step until the starting state:

$$G \leftarrow \gamma G + R_{t+1}$$

  Where G is return and R_{t+1} is the reward received.

- After finishing each episode, the return (*Q* value) for each state is updated using only the "first-visit" state-action pair value. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N_{1\text{st}}(s,a)}\big(G - Q(s,a)\big)$$

  Where $N_{1\text{st}}(s,a)$ is the number of first-visit state-action pairs in each episode.

- The training will terminate when reach the maximum set episode, and the *Q* table that stores each state-action pair *Q* value is finalized. The optimal policy then can be extract using:

$$\pi^*(s) \leftarrow \arg\max_a Q(s,a)$$

  Where $\pi^*(S_t)$ is the optimal policy extracted from action values.

2.5 SARSA with an ϵ-greedy behavior policy

The algorithm is created by defining a `SARRSA` class. At the beginning of the algorithm, the initial function receives four parameters which are environment settings, learning rate α, ϵ and γ. The basic environment parameters include the map configuration, number of actions and number of states. These parameters inherit the settings from the environment and parameter files. The variables captured for evaluation during training are the same as discussed in the previous section. A *Q* table is created to constantly store the updated *Q* value during training. Similarly, the ϵ-greedy policy is also used to ensure exploitation and exploration. The core interaction process of the SARSA algorithm is to continuously interact with the environment until the end of each episode. The training process is much more straightforward then that in *Monte Carlo* method, and is defined in `train()` as following steps:

- Initialize the environment for a new episode and assign a random action to the agent by using ϵ-greedy policy.

- Execute the current action to get the next state, reward, and flag indicates whether the episode is done or not from `step()` defined in `environment.py`.

- Select the next action according to the next state based on the ϵ-greedy policy to maintain the "on-policy" characteristic of the algorithm.

- Use computed $Q(s',a')$ and $Q(s,a)$ to update the $Q(s,a)$ for each time step for each state in each episode until training ends based on the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma\, Q(s',a') - Q(s,a)\big).$$

- Eventually, the finalized Q table obtained after training ends and optimal policy can be extracted using:

$$\pi^*(s) \leftarrow \arg\max_a Q(s,a)$$

2.6 Q-learning with an ϵ-greedy behavior policy

The algorithm is created by defining a `Q-learning` class. At the beginning of the algorithm, the initial function receives four parameters which are environment settings, learning rate α, ϵ and γ. The basic environment parameters include the map configuration, number of actions and number of states. These parameters inherit the settings from the environment and parameter files. The same evaluation scheme is used as that in *SARSA* method. A *Q* table is created to constantly store the updated *Q* value during training. Similarly, the ϵ-greedy policy is also used to ensure exploitation and exploration. The process of training using *Q-learning* is similar to *SARSA* except that the $Q(s', a')$ used is the largest value at the next state to update. The process is described as follows:

- Initialize the environment for a new episode and the agent selects a random action based on ϵ-greedy policy.
- Execute the current action to get the next state, reward, and flag indicates whether the episode is done or not from `step()` defined in `environment.py`.
- The update of the *Q* value is defined in function `learn()` which takes the current state-action pair, reward received and the next state as input. Use the next state-action pair that gives the largest *Q* value, the update rule is given as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r \ + \ \gamma \max_{a'} Q\big(s', a'\big) \ - \ Q(s,a)\big].$$

- After looping through all the episode in `train()`, the updated *Q* table can be used to provide the optimal policy:

$$\pi^*(s) \leftarrow \arg\max_a Q(s,a)$$

3 Evaluation and Discussion

3.1 Parameter Settings

In this section, the training results of the reinforcement learning algorithms are given based on the following common performance criteria:

- Average Reward: As the episode accumulates, the average rewards for all episodes are computed cumulatively to observe the overall improvement of the learned policy. The impact of reward fluctuations in a single episode can be reduced, making the evaluation easier to observe the overall trend.
- Time Steps per Episode: Measure the number of steps to finish an episode which can reflect the efficiency of the current policy.
- Success Rate: The proportion of statistics on reaching the goal state for every 50 episodes, this is important to indicate the stability of the algorithm.
- Mean Squared Error (MSE) of *Q* Value: Standard deviation of all Q values for each episode, by observing the standard deviation curve to determine whether the Q value converges and obtain intuitive feedback of volatility and learning process stability.
- Success and Failure Bar Chart: Intuitively indicate the quality, the stability and convergence of trained *Q* table in the learning process.

The parameters set during training are:

- `RANDOM_SEED=31`, `GRID_SIZE=[4,10]`, `NUM_STEPS=200` and `NUM_EPISODES=20000`
- For *Monte Carlo*: $\epsilon = 0.5$, $\epsilon_{min} = 0.01$, $Decay\ fator = 0.999$ and $\gamma = 0.99$.

- For *SARSA*: $\epsilon = 0.4$, $\epsilon_{min} = 0.01$, $Decay\ fator = 0.9$, $\alpha = 0.01$ and $\gamma = 0.99$.
- For *Q-learning*: $\epsilon = 0.4$, $\epsilon_{min} = 0.01$, $Decay\ fator = 0.9$, $\alpha = 0.01$ and $\gamma = 0.99$.

The full training results can be found in `Results` folder.

3.1.1 Training Results for 4x4 Environment

The training results for the 4x4 grid will be provided in this section by the figures below. From left to right are average rewards curve, time steps used in each episode, average success rate over 50 episodes curve, and the Mean Squared Error (MSE) of Q Value for each episode curve for three reinforcement learning algorithms.

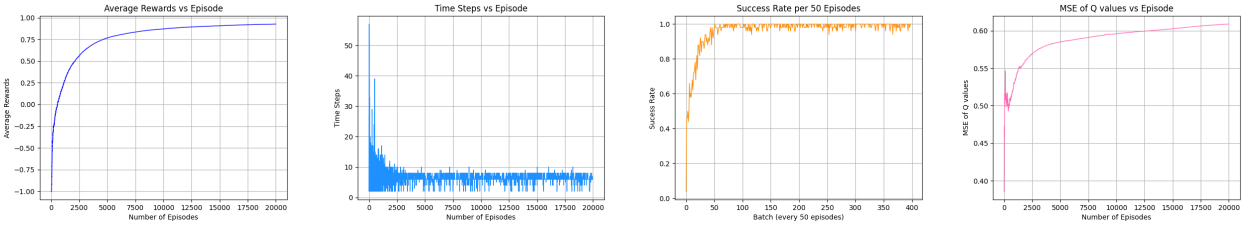


Figure 1: First Visit Monte Carlo Control Training Results (4x4)

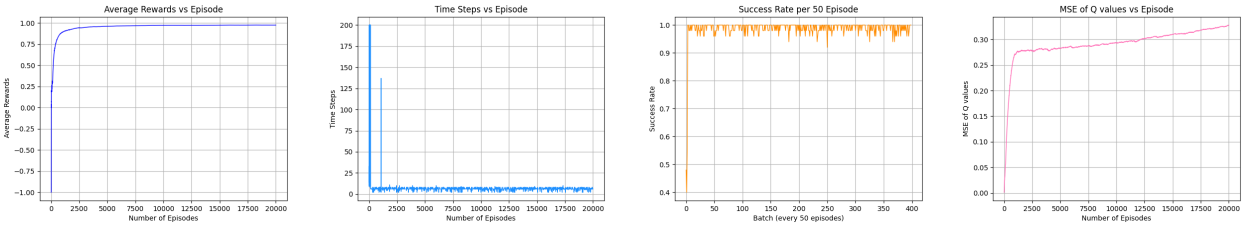


Figure 2: SARSA Training Results (4x4)

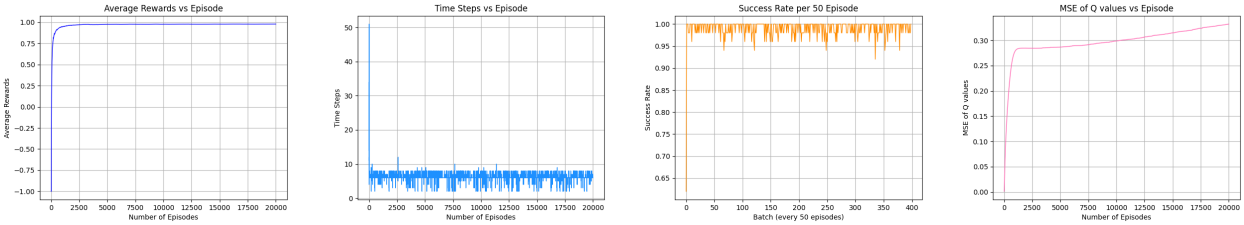
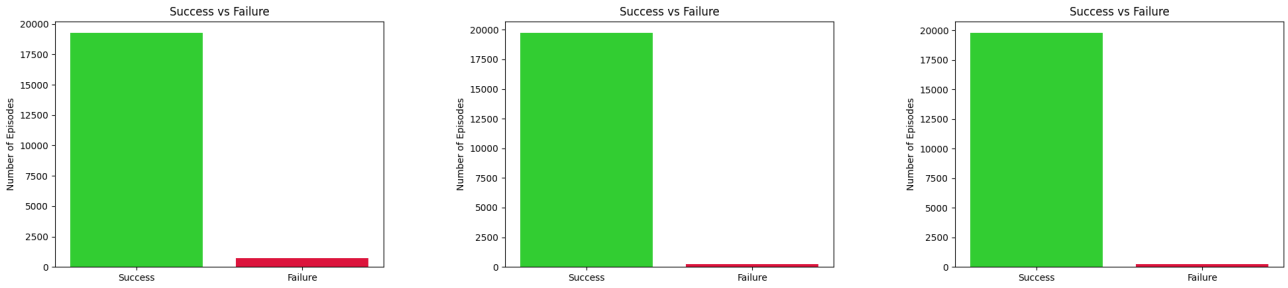
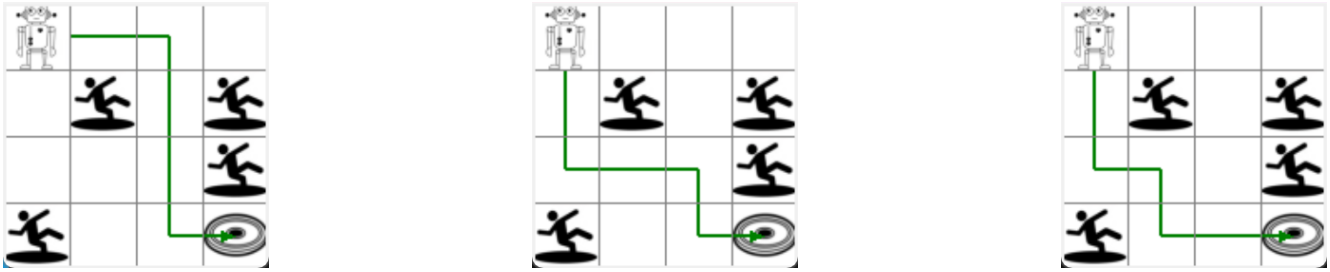


Figure 3: Q-learning Training Results (4x4)



(a) Monte Carlo (b) SARSA (c) Q-learning

Figure 4: Sucess vs Failure Bar Chart (4x4)



(a) Monte Carlo (b) SARSA (c) Q-learning

Figure 5: Optimal Policies (4x4)

3.1.2 Training Results for 10x10 Environment

The training results for the 10x10 grid will be provided in this section by the figures below. From left to right are average rewards curve, time steps used in each episode, average success rate over 50 episodes curve, and the Mean Squared Error (MSE) of Q Value for each episode curve for three reinforcement learning algorithms.

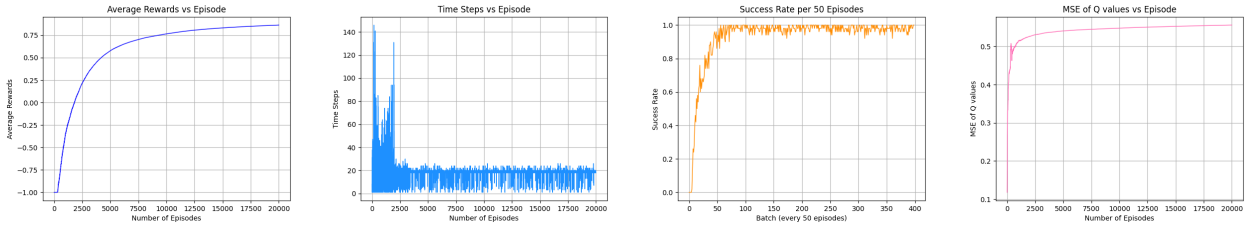


Figure 6: First Visit Monte Carlo Control Training Results (10x10)

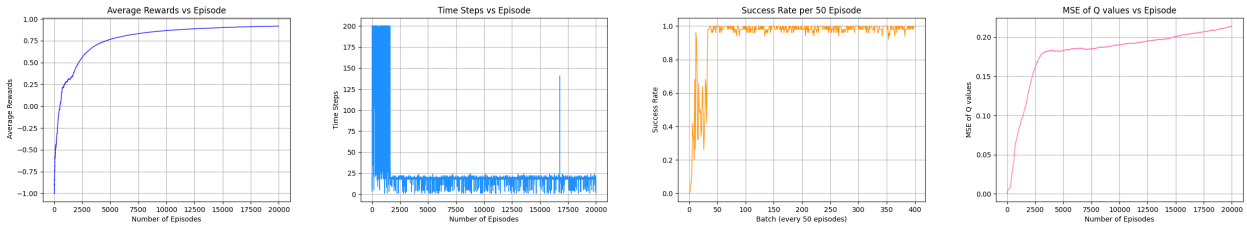


Figure 7: SARSA Training Results (10x10)

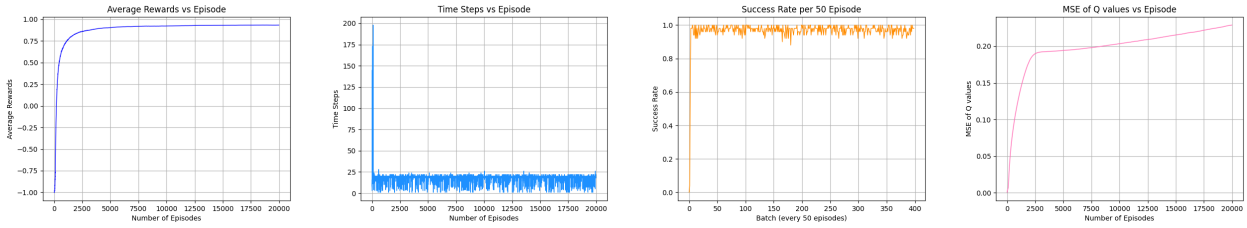
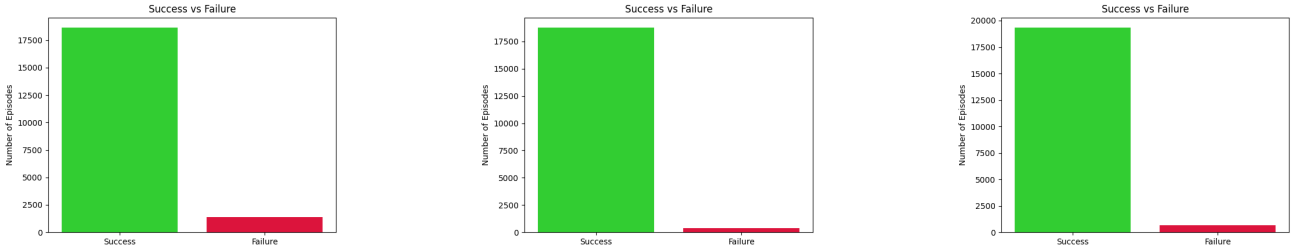
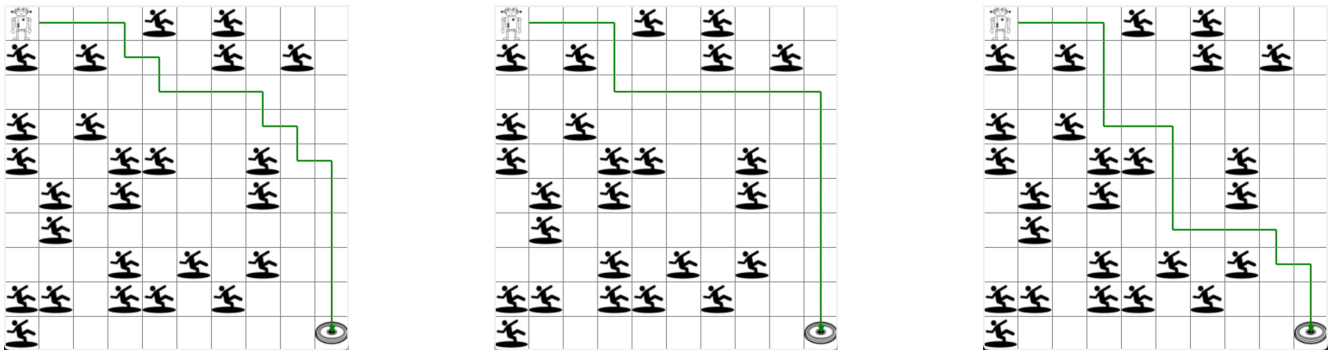


Figure 8: Q-learning Training Results (10x10)



(a) Monte Carlo (b) SARSA (c) Q-learning

Figure 9: Sucess vs Failure Bar Chart (10x10)



(a) Monte Carlo (b) SARSA (c) Q-learning

Figure 10: Optimal Policies (10x10)

3.1.3 Results Comparison

A detailed comparison of different algorithms will be carried out in this part. From Figures below, we can see that *Q-learning* has the fastest increase rate of the average reward and the success rate curves followed by *SARSA* and *Monte Carlo*. This is because that *Monte Carlo* requires a large amount of environment sampling in early stages that brings drastic fluctuations. The unstable observation in *SARSA* is because of the relative characteristics of "conservative" on-policy updates. On the other hand, the reason that *Q-learning* converges so fast is due to its characteristics of "optimistic" off-policy updates to maximize future returns. It can also be observed that the convergence of time steps taken to reach the goal from *Monte Carlo*, *SARSA* to *Q-learning* shows a decreasing trend. This phenomenon is consistent with the action selection strategy discussed before. After the convergence, the time steps used of *Q-learning* is the least means that it found a more efficient route to the goal than that from other methods. The MSE of Q value of *Monte Carlo* gives a huge gap from *SARSA* and *Q-learning*, this is because the returns only update after a complete episode, so deviation is high. It gradually stabilizes when approaching the middle stages of training, indicating that the Q value distribution converges, and the policy approaches the final one. The The MSE of Q value of *Q-learning* becomes stable quickly because the update for each time step is based on $maxQ(s', a')$, it tends to converge the difference in the distribution of the Q values quicker.

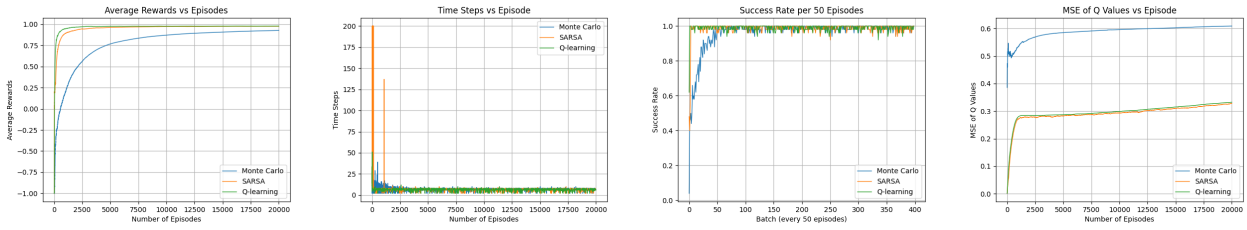


Figure 11: Combined Training Results for Three Algorithms (4x4)

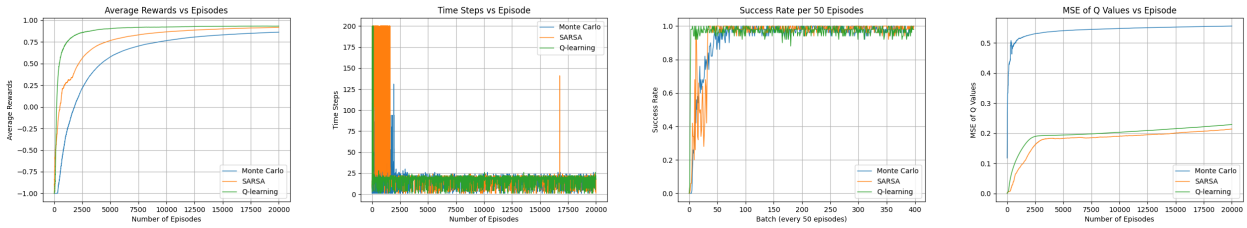


Figure 12: Combined Training Results for Three Algorithms (10x10)

In conclusion, under the same environment, *Q-learning* can usually converge the fastest and maintain a high success rate. *SARSA* updates the policy in a more conservative way, although there are more fluctuations in the early stages of training, but it is safer in uncertain environments. *Monte Carlo* It relies solely on heavy sampling and the return of the entire episode for updates, and the initial fluctuations are large, but once a better policy is derived, the success rate can be quickly increased and stabilized at a high level. *Q-learning* performances the best, and *SARSA* then the *Monte Carlo*. Under current given environments, three methods can eventually learn reasonable policy whose success rate can be close to 100%. The similarities observed are:

- All methods use ϵ greedy policy to balance exploration and exploitation. The reward curve and success rate curve in the figure have an identical trend from low to high.
- All methods tend to have a high success rate, high average reward, and relatively low time steps after training, which means that a good policy can be converged under reasonable settings of parameters.
- As the training progresses, the standard deviation (MSE) curve of the Q value converges for all methods, which means that the value prediction are gradually stable.

The differences observed are:

- Convergence Speed: *Q-learning* can bring average rewards and success rates to a high level after fewer episodes while *Monte Carlo* and SARSA relatively require more episodes to reach similar levels.
- Fluctuation: *Monte Carlo* fluctuates more at begining due to sampling activities. *SARSA*'s time steps and average reward curve may experience more unstable in the middle of training; *Q-learning* often presents a fast and relatively stable behavior.
- On-policy and Off-policy: *SARSA* is more "careful" about negative rewards, so a longer period of conservative exploration can be seen. *Q-learning* pursues maximum Q value, it converges quickly under appropriate parameters, but it may also give up some exploration too early.

The abnormal phenomena and possible causes:

- Severe fluctuations at the beginning: All methods have large fluctuations in the first few thousand rounds. This is because when the ϵ value is high, a higher probability of random exploration is given, and the returns are sparse due to the given reward setting, resulting in high noise.
- Q value fluctuations: Especially in *Monte Carlo* who get the returns from a single episode have a higher impact on Q value prediction, the MSE of Q values changes drastically due to the constant updates in the early episodes from the "guessed" initial Q table.
- Crest in time steps graph of *SARSA*: The crest is attributed to the "exploration" actions taken by the agent, deviating from the optimal path.

The results of *SARSA* using different ϵ are provided below as a demonstration to show the impact of varying parameters on performance in 10x10 environment, the plots for different algorithms using different parameters can be found in the `Results` folder.

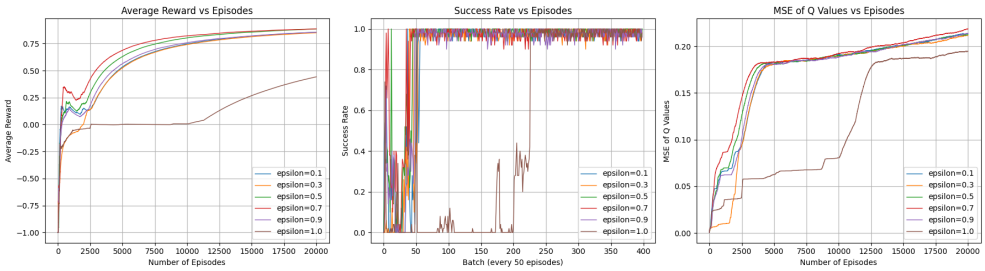


Figure 13: SARSA with Various ϵ

4 Problems, Possible Improvements and Significant Factors

Several issues occurred, possible improvements and important factors during implementation can be summarized below:

- Holes position matters and the Local Optimum: For certain configurations of the randomly generated holes (e.g., the starting point is surrounded by holes, only one spare space to move out.), the performance of the *Monte Carlo* and *SARSA* can perform badly when initially used the fixed value of ϵ, the agent will stick in a position or wandering between two states, thus unable to fully explore the environment before finding a path to the goal. This is because the location of holes limits the spaces for the agent to move which reduces the probability of the exploration of the feasible paths, and evironment rewards are sparse. This can be solved by finely tuning the ϵ and making it decay with episode.

- The settings of α, γ and ϵ: In all algorithms, the settings of the learning rate, discount factor, epsilon and its attenuation coefficient have a greater impact on the training results. If you set the epsilon high and its attenuation coefficient low, the *Q-learning* and *SARSA* can converge very slowly. Vice versa, if you set the epsilon low and its attenuation coefficient high, the performance of *Monte Carlo* will give bad results without an usable policy to allow the agent to find a route to the goal state. This is the ”trade-off” between exploitation and exploration. The gamma must be set to close to 1 to tackle the problem of sparse reward, make agent focus on long-term returns. The α should set to a low value to provide stable updates.This can be addressed to try out different combinations of parameters manually for different environment setups (e.g., position of holes) or plot a three-dimensional graph where the x and y-axis are the parameters, and the z-axis is the evaluation index of your choices to find a good combination of the parameters. This is because different environments (such as the location of holes) have different sensitivity to parameters, so there is no ”one-for-all” parameter combination that can be seen as omnipotent.

- Sample efficiency and state space scale: The scale of the environments for this project is only up to 10x10. The computation cost has already increased dramatically from 4x4 to 10x10, if the problem continues scaling up with larger action space, computing consumption grows exponentially, table-based methods (*Monte Carlo*, *SARSA*, and *Q-learning*) may require more sampling to achieve better performance and will face the problem of dimensional disasters. This can be mitigated by feature extraction to map high-dimensional data to lower-dimensional, etc. The Deep neural network (e.g., DQN) as predictor for Q values. Or using better representation technique of Q table and more efficient way to update Q values.

- Rewards function design: The original rewards are sparse, the problem of local optimum and ”difficult cases” of holes location can be solved by adding intermediate states rewards based on the Euclidean distance between agent and the goal, giving a small negative reward every step taken and the actions towards the map boundaries and giving higher penalties for falling into a dangerous state. In this way, the test results give a much faster convergence speed of all methods.

In conclusion, the frozen lake problem is simple and straight forward but useful in understanding core concepts of the Reinforcement Learning techniques.