Name: Hazem Taha ID: 400550393

ECE 718 - Course Project Report

I. Environment Design

In order to have full control over the slightest detail inside the environment to be able to effectively do a thorough comparison between the different RL algorithms, I started the project by designing my own maze environment shown in figure 1. Setting the map, defining the rules, rewarding function, and how the data is structured and updated. This gives more freedom to alter some parts to serve specific algorithms' needs.

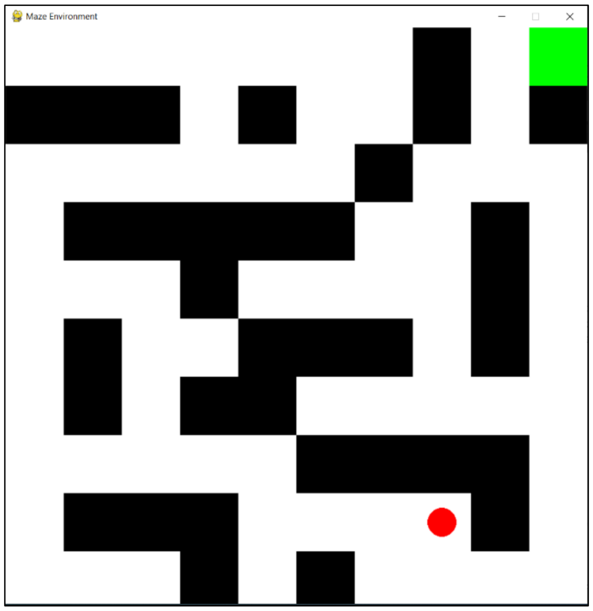


Figure 1: Live screenshot from the environment

Maze map:

The maze map is parameterizable and is initially set as a 10x10 matrix of cells. Each cell can take a value of three as follows:

{0 : Empty cell, 1 : Wall, 2 : Terminal}

Hidden variables:

- `agent_pos`: Defined as a 2D vector (y, x), this variable defines the current location in the grid the agent is located in. Upon reset, the agent is randomly initialized inside the maze.

Observation Space:

The observation space is a single discrete value ranging from 0 to 99. This represents a linear representation of the Agent Position in a 10x10 grid and is calculated as follows:

```
State = agent_pos.y * COLS + agent_pos.x
```

Action Space:

The action space is composed of 4 discrete actions (Up, Right, Down, left) each of them is integer encoded from 0 to 3. When an action is received, it's mapped using a lookup table into delta (change) in y and delta in x. if the new state is valid (not hitting a wall or going outside of the maze) the deltas are applied to the hidden variable `agent_pos` and the reward is calculated.

Rewarding function:

The rewards were designed to stimulate the agent to reach the terminal state in the least number of steps. Therefore, there is a negative penalty for each time step spent inside the maze. Furthermore, to discourage the agent from hitting a wall, an extra penalty was given when the movement is invalid.

- 10 : If reached a terminal
- -0.1: for each movement inside the maze
- -1: for invalid moves (hitting walls)

II. Optimal Policy and State-value Function (Value-Iteration Algorithm)

Before starting the RL algorithms comparisons, it was crucial to have a baseline (optimal answer) to compare with. In order to find the optimal policy $\boldsymbol{\pi}_*$ and optimal state-value function $\boldsymbol{V}_*$, a **Value-Iteration Algorithm** has been implemented and applied to the maze.

As the developed environment is a deterministic one where each state-action pair gives a deterministic reward and next state. I made a small adjustment to the original algorithm to make it more computationally efficient. When calculating

$$V(s) = \max_a(\mathbb{E}[R|s,a] + \gamma \sum_{s'} P(s'|a,s)V(s'))$$

We can simplify it to

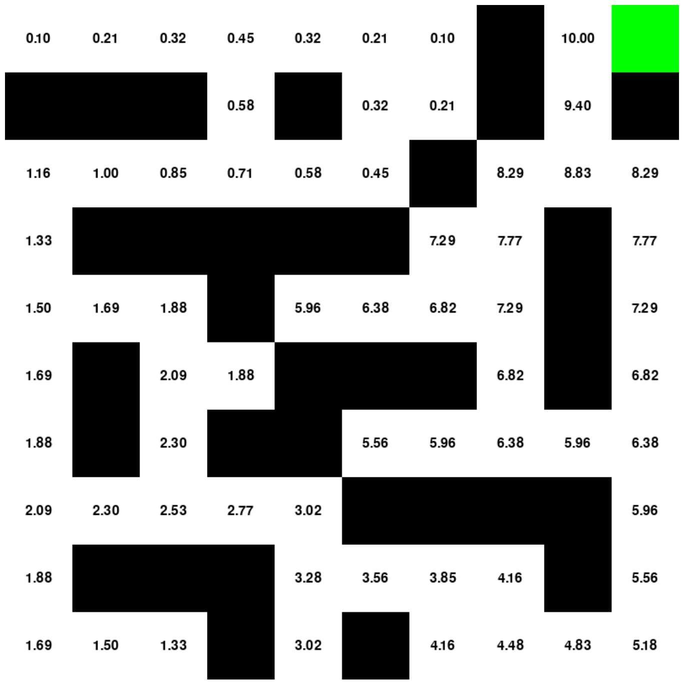
$$V(s) = \max_a(R + \gamma V(s'))$$

Where R and s' are computed by setting the environment state to s and taking action a and observing the outcomes. The same method is used in the policy extraction part of the algorithm.

To be able to easily visualize the outcome of the algorithm, I developed a class named `MazeVisualizer` that can be instantiated with certain policy $\pi(s)$ and state-value $V(s)$ and outputs gives 2 plots of the game environment showing the content at of both functions on each cell in the maze. This is shown below in figures (2) and (3).

As shown in figure 2 (b) the results show smooth and consistent flow from any state to the terminal. Figure 2(a) also gives reasonable results, for instance, it was anticipated that the state just beside the terminal would have a state value function of 10 because an agent there should definitely choose to go to the terminal, and he will receive a reward of 10 and the episode ends.

(a)



(b)

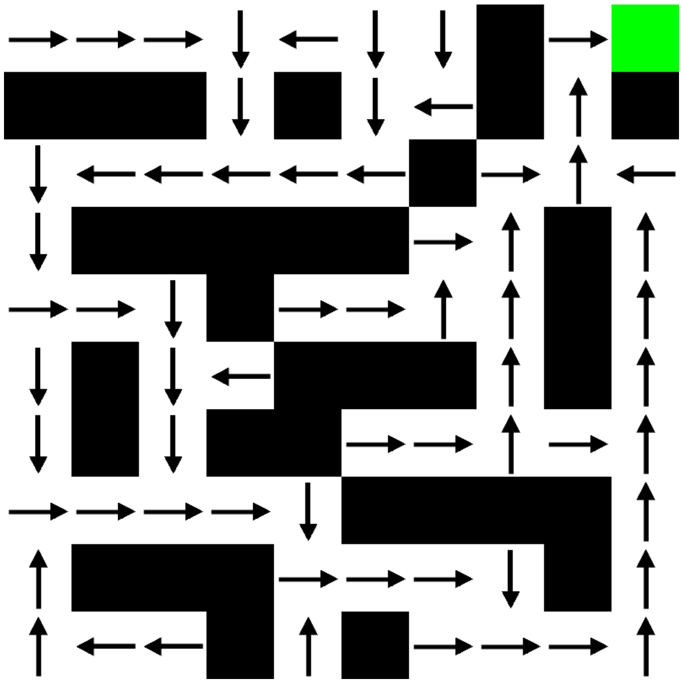


Figure 2: Visualization of the Value-Iteration Algorithm results
(a) State-value function - (b) Policy function.

III. State-Value and Policy Estimates (Various RL Algorithms)

1. Temporal Difference (TD) Learning Algorithms:

In this category of algorithms, I implemented the three required algorithms (SARSA, Expected SARSA and Q-learning) using a common class `TD_Agent`, which makes use of the same methods just with a couple of conditional statements to differentiate between some algorithms.

All three algorithms follow an epsilon-greedy strategy in choosing actions while training. The parameter **epsilon** is set to 1 at the starting episode to allow maximum exploration and is complemented by a **decay rate** to control the scheduled reduction of exploration and hence increase in exploitation. Notice that the decay rate is a fraction multiplied by epsilon and hence a higher rate leads to slower decay over time.

1.1 SARSA:

This is an on-policy algorithm which follows that same policy that gets updated. I used some if statements to make sure that the next state action is chosen using the current policy before updating it. I stared my analysis be setting the hyperparameters of learning rate (α) and decay rate (r) as follows:

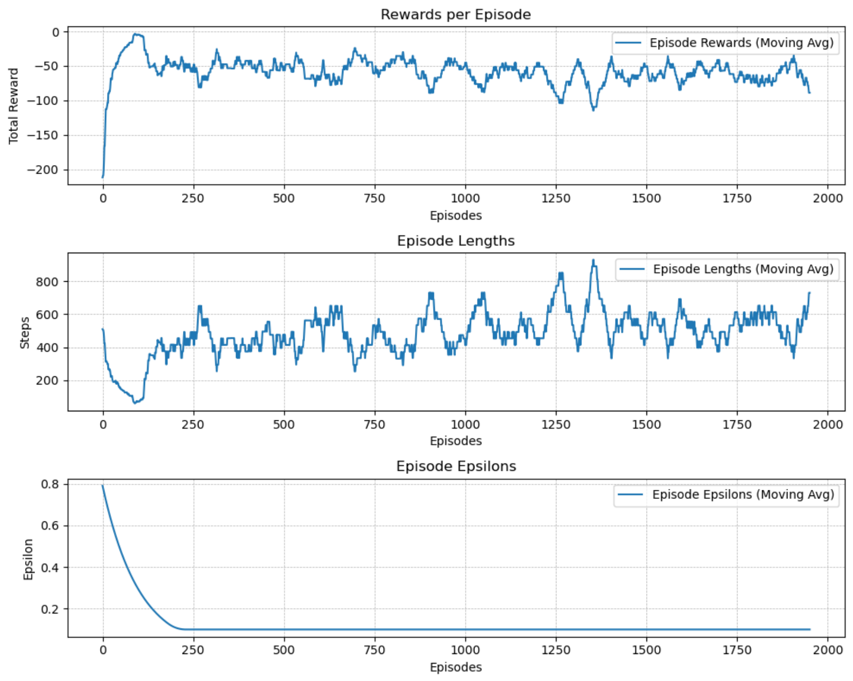


Figure 3: SARSA agent with **α = 0.1, r = 0.99**

As shown in figure (3) It was seen that neither the training cumulative rewards nor episode length coverage to a stable point. I will sweep over **r** and when I find a sweet spot I will sweep over **α.**

Name: Hazem Taha ID: 400550393

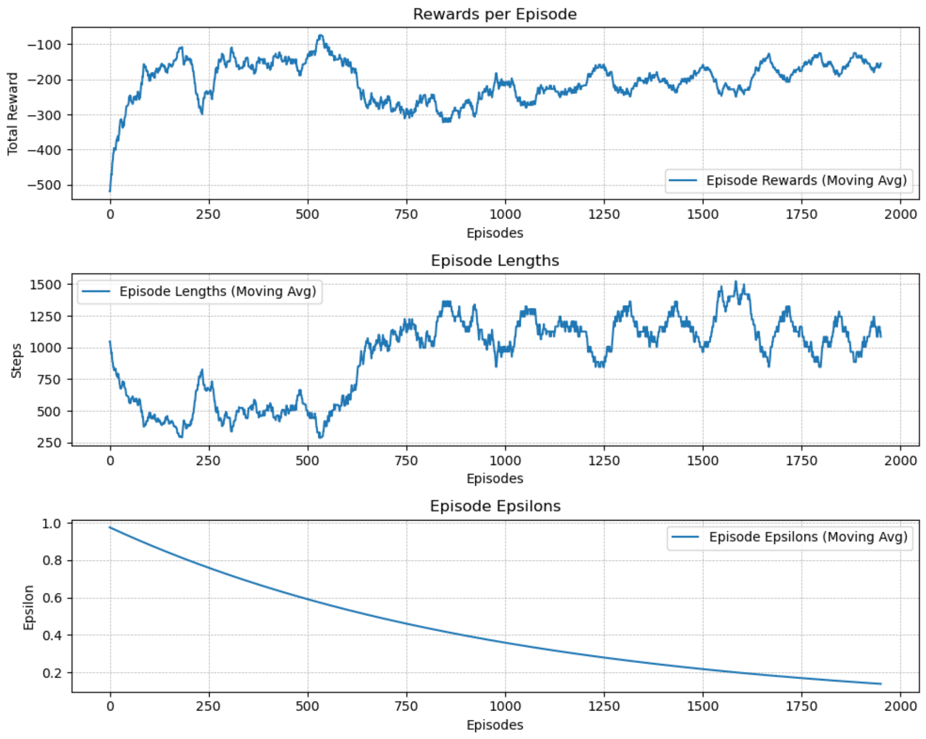


Figure 4: SARSA agent with **α = 0.1, r = 0.999**

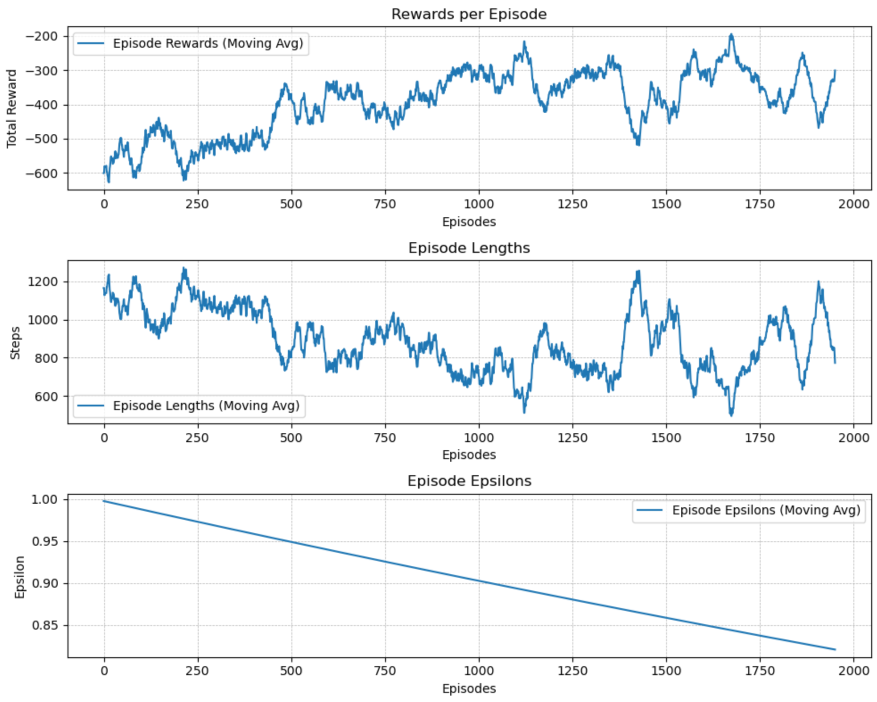


Figure 5: SARSA agent with **α = 0.1, r = 0.9999**

Sarsa α = 0.1, ε = 0.995

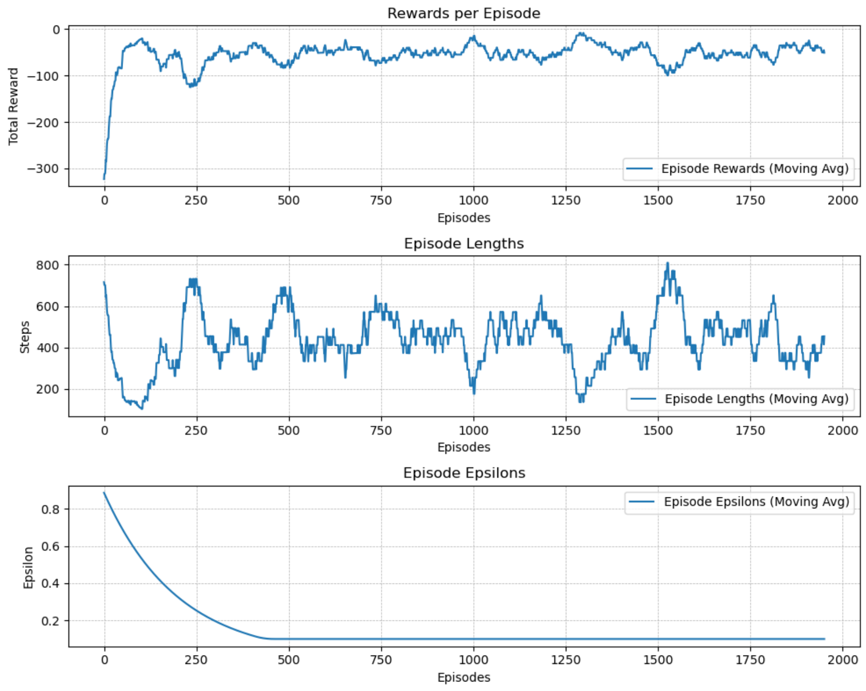


Figure 6: SARSA agent with **α = 0.1, r = 0.995**

Figure 6 shows the best-found decay rate so far (0.995), I will start tuning the learning rate **α.**

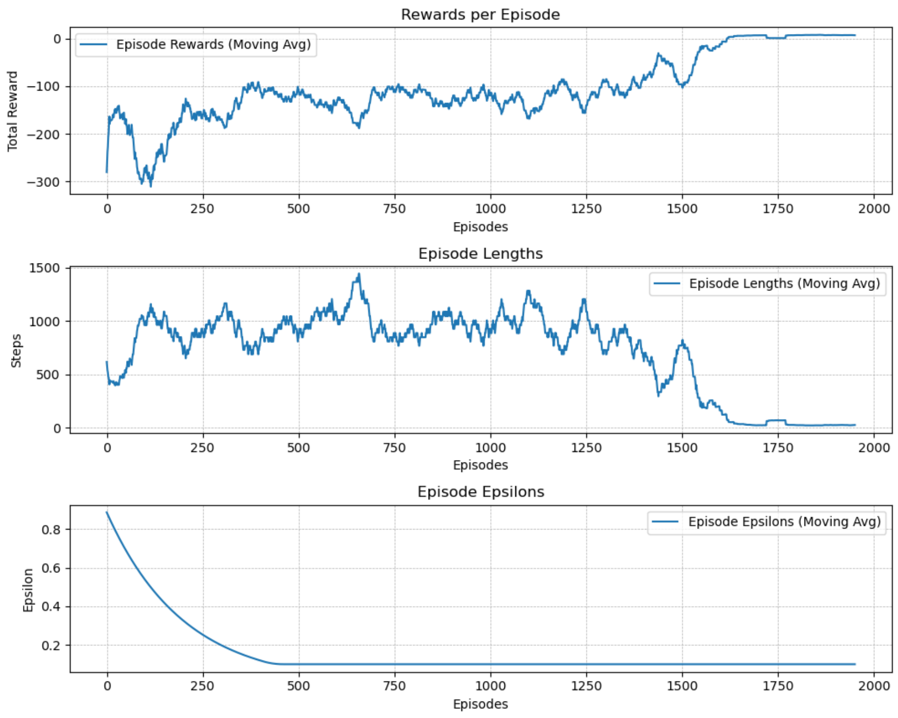


Figure 7: SARSA agent with **α = 0.4, r = 0.995**

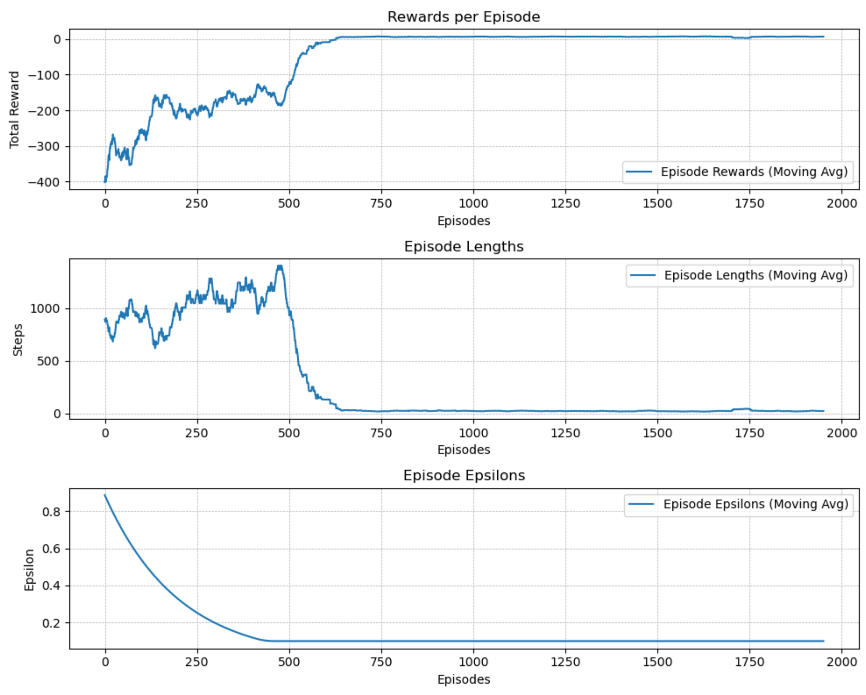


Figure 8: SARSA agent with **α = 0.6, r = 0.995**

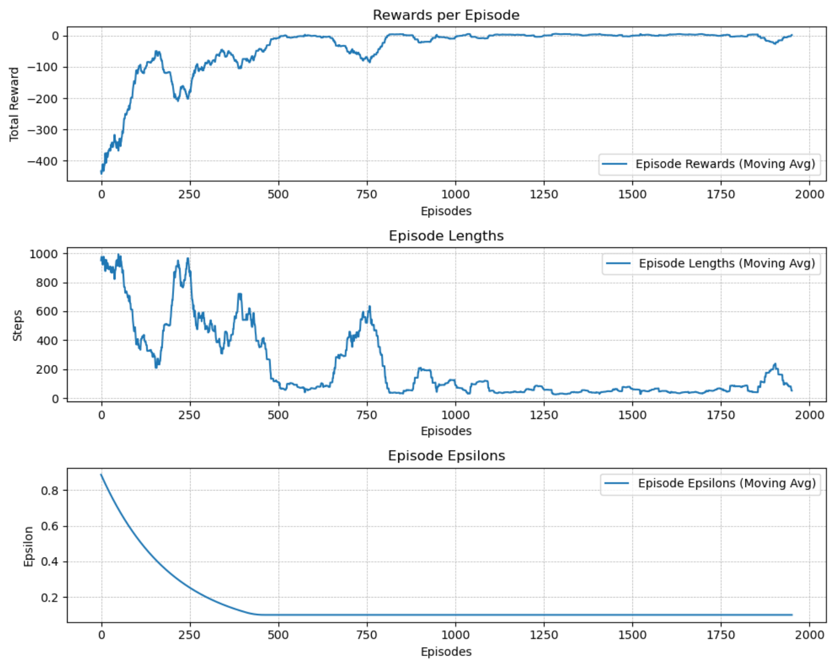


Figure 9: SARSA agent with **α = 0.8, r = 0.995**

Again, **α** between 0.6 and 0.8 gives the optimal training curves so far. I will set **α** to 0.7.

Name: Hazem Taha ID: 400550393

Final training and testing of SARSA:

Using the decided hyperparameters, an extended training for 10,000 episodes has been done followed by a live testing for another 10,000 episodes and the results are as shown in Figure 10. Furthermore, Figure 11 shows a visualization of the results on the maze.

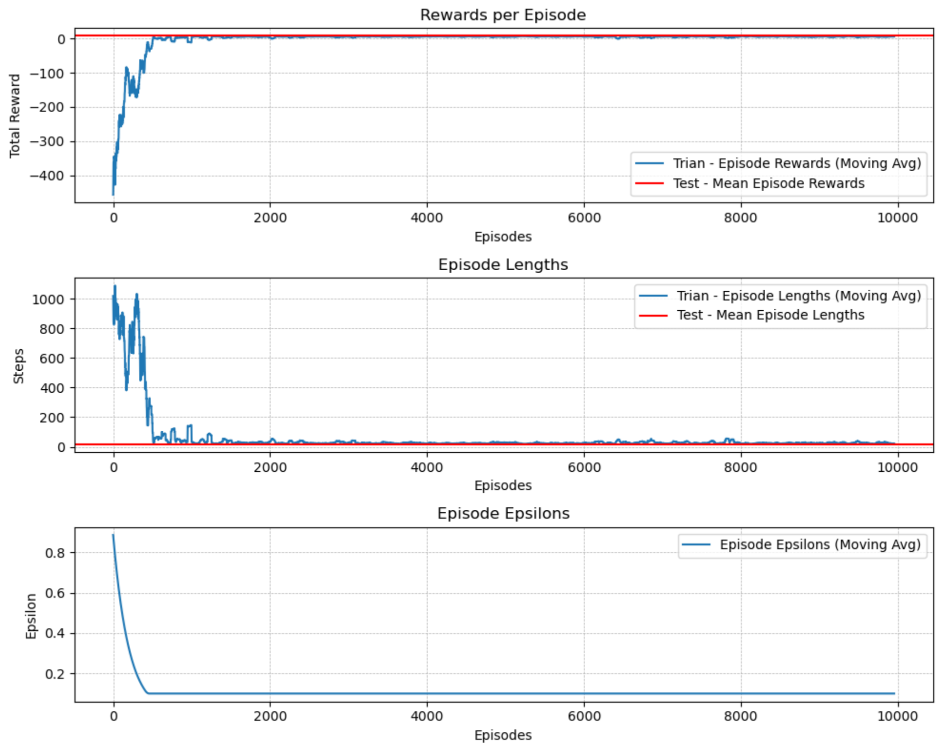


Figure 10: Final training and testing of SARSA.
α = 0.7, r = 0.995

Notice that testing was done by allowing the agent to exploit the trained policy without any exploration. It is the same function used across all RL algorithms later. The red line is a single value representing the mean of (cumulative rewards / episode lengths) that the trained agent got across all 10,000 testing episodes.

Numerical results:

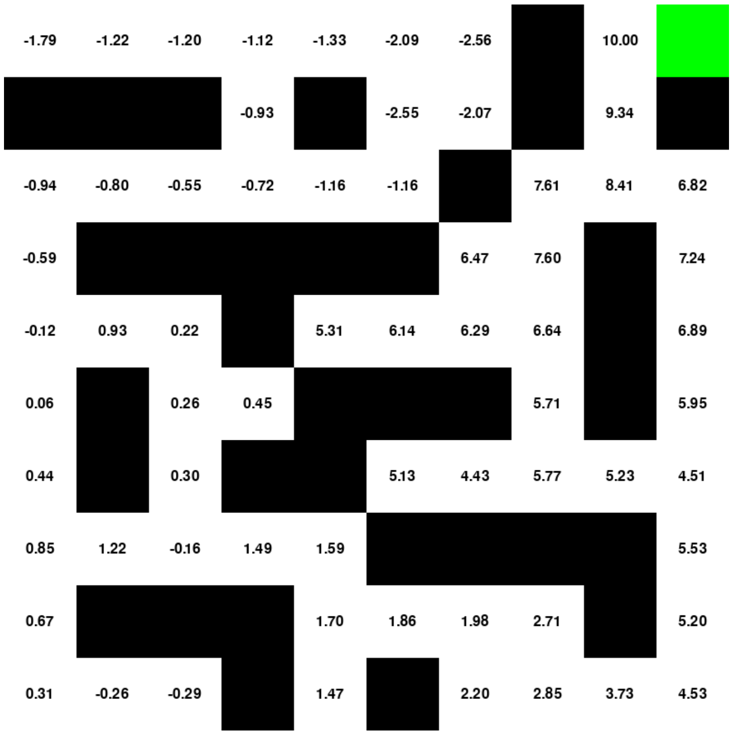
- Last rolling average of training episodes cumulative rewards: 6.042
- Last rolling average of training episodes lengths: 22.04
- Mean of testing episodes cumulative rewards: 8.263
- Mean of testing episodes lengths: 18.374

Comparison with optimal functions:

- Hamming Distance (policy difference): 8
- MSE (value function difference): 2.1680

Name: Hazem Taha ID: 400550393

(a)



(b)

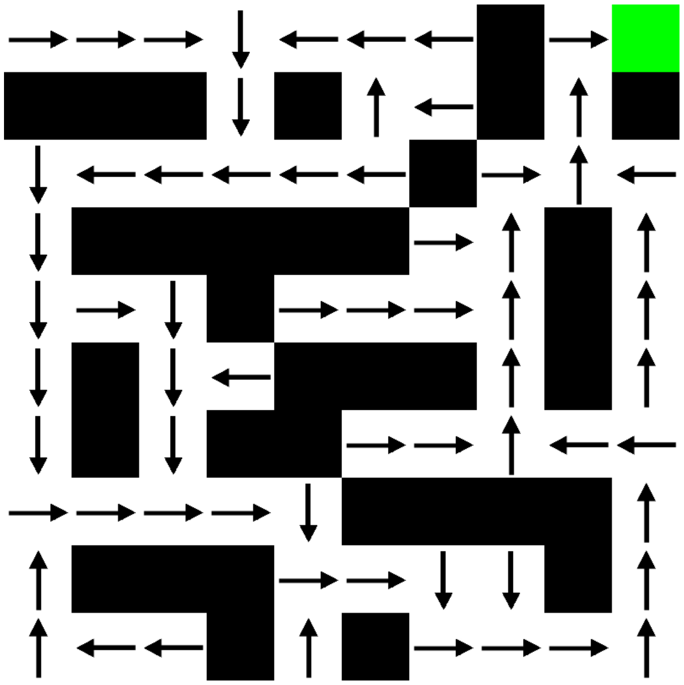


Figure 11: Visualization of the SARSA algorithm results
(a) State-value function - (b) Policy function.

1.2 Expected-SARSA:

Similarly, when training an expected SARSA agent, I did an extended training for 10,000 episodes followed by a live testing for another 10,000 episodes and the results are as shown in Figure 12. Furthermore, Figure 13 shows a visualization of the results on the maze.

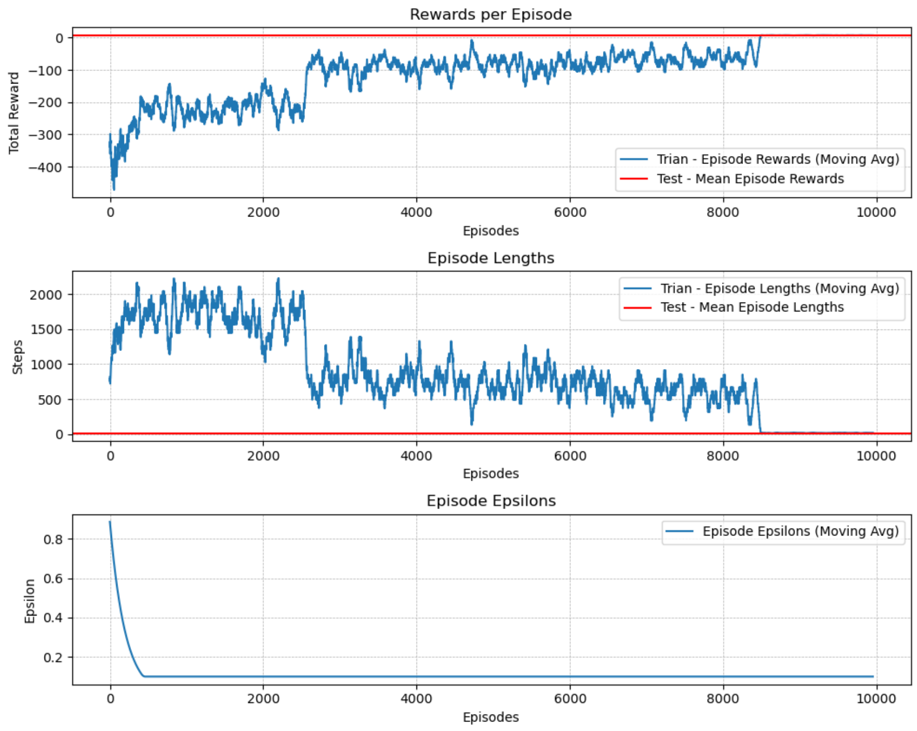


Figure 12: Final training and testing of Expected SARSA.
α = 0.7, r = 0.995

It can be noticed that expected SARSA takes much more time to converge than SARSA, converging at around 9,000 episodes.

Numerical results:

- Last rolling average of training episodes cumulative rewards: 7.136
- Last rolling average of training episodes lengths: 21.36
- Mean of testing episodes cumulative rewards: 8.286
- Mean of testing episodes lengths: 18.135

Comparison with optimal functions:

- Hamming Distance (policy difference): 5
- MSE (value function difference): 0.772

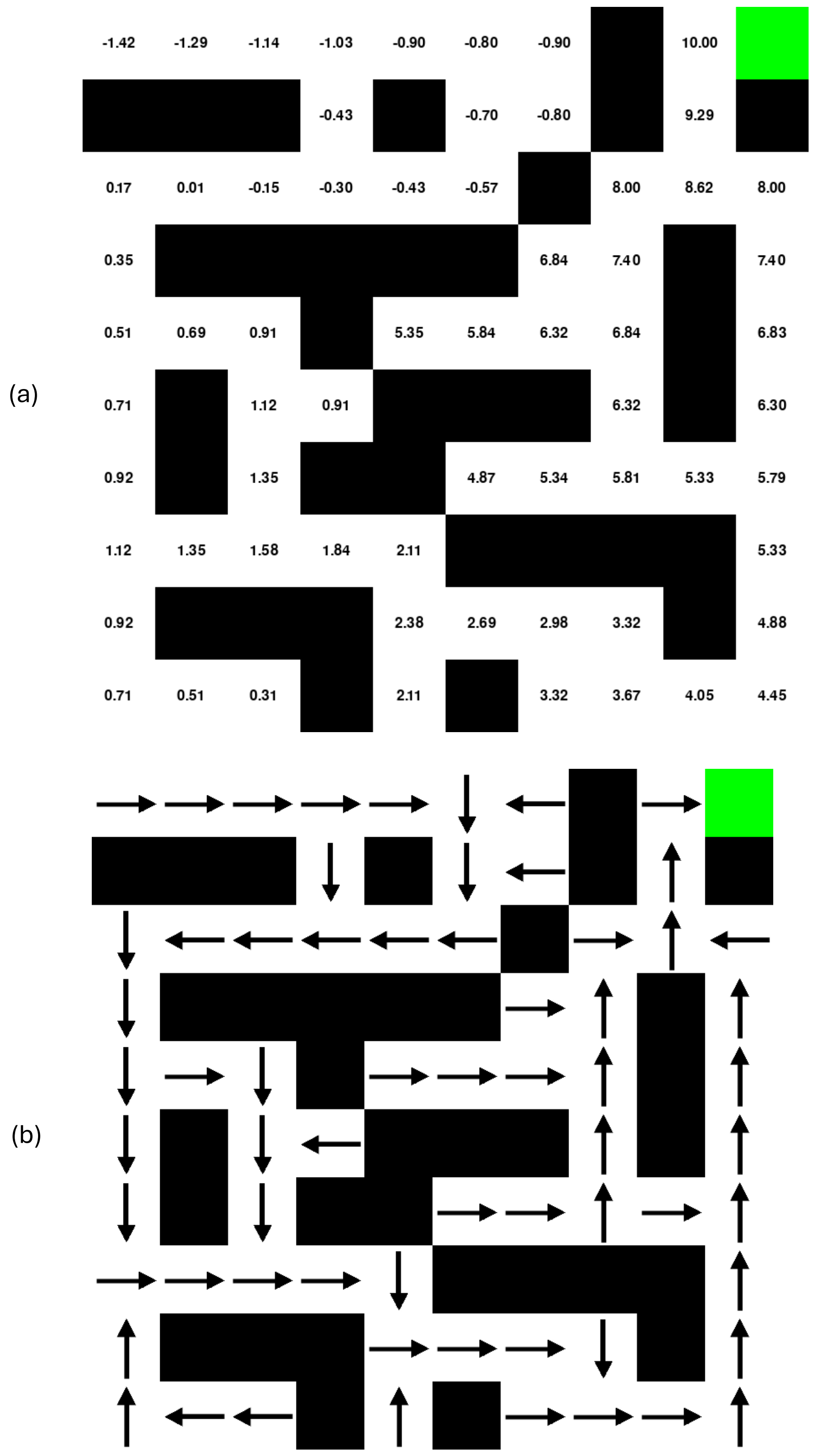


Figure 13: Visualization of the Expected SARSA algorithm results
(a) State-value function - (b) Policy function.

1.3 Q-Learning:

Similarly, when training a Q-Learning, I did an extended training for 10,000 episodes followed by a live testing for another 10,000 episodes and the results are as shown in Figure 14. Furthermore, Figure 15 shows a visualization of the results on the maze.

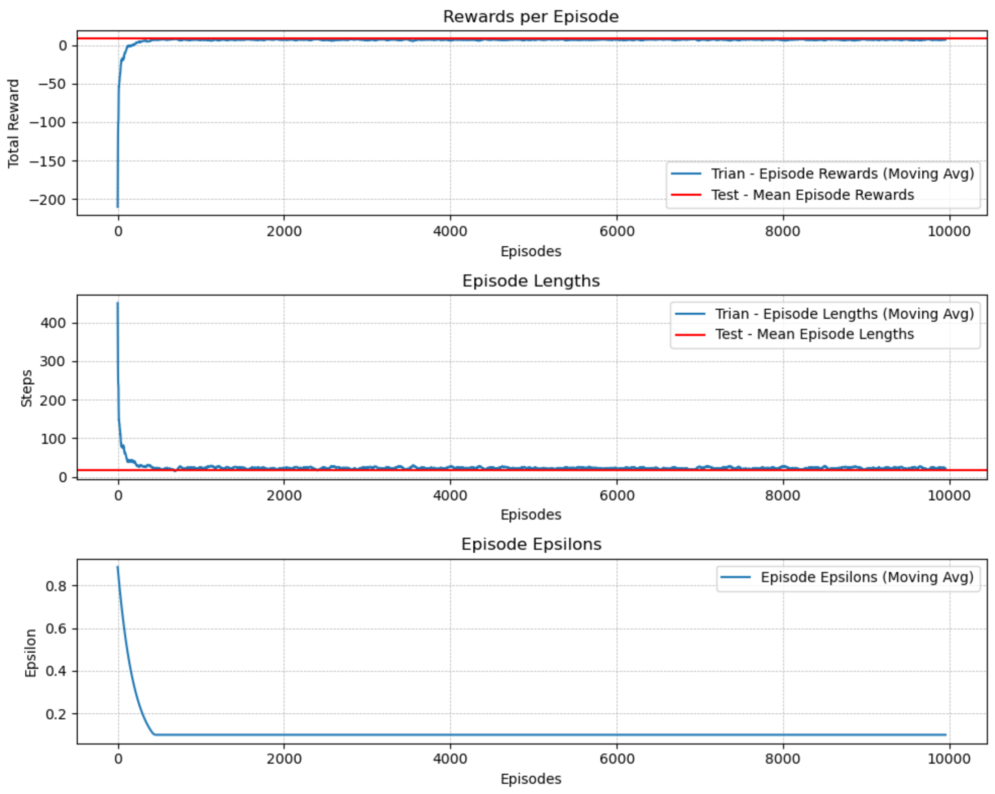


Figure 14: Final training and testing of Q-Learning.

α = 0.7, r = 0.995

It can be seen that training using Q-learning converged at around 1000 episode similar to SARSA.

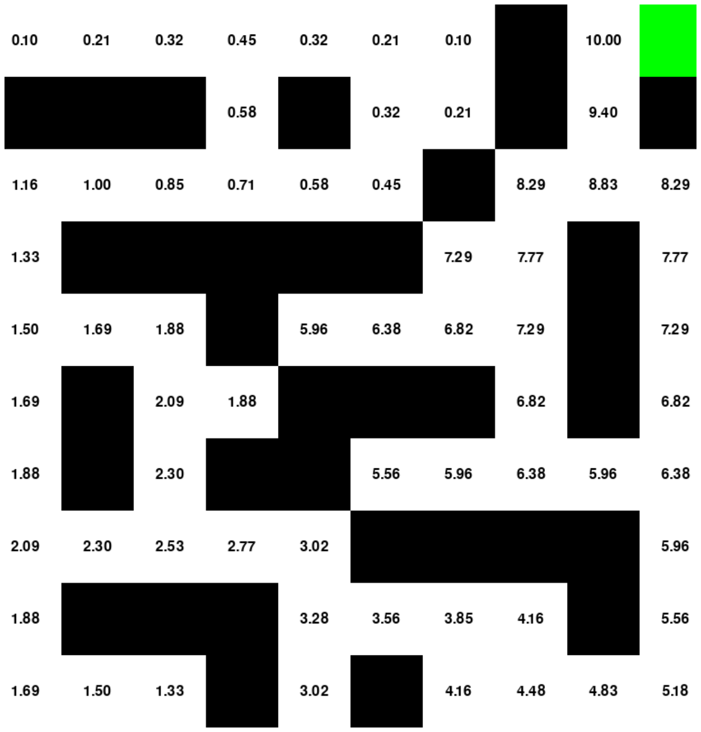
Numerical results:

- Last rolling average of training episodes cumulative rewards: 7.407
- Last rolling average of training episodes lengths: 20.08
- Mean of testing episodes cumulative rewards: 8.3
- Mean of testing episodes lengths: 17.997

Comparison with optimal functions:

- Hamming Distance (policy difference): 1
- MSE (value function difference): 0.0001

(a)



(b)

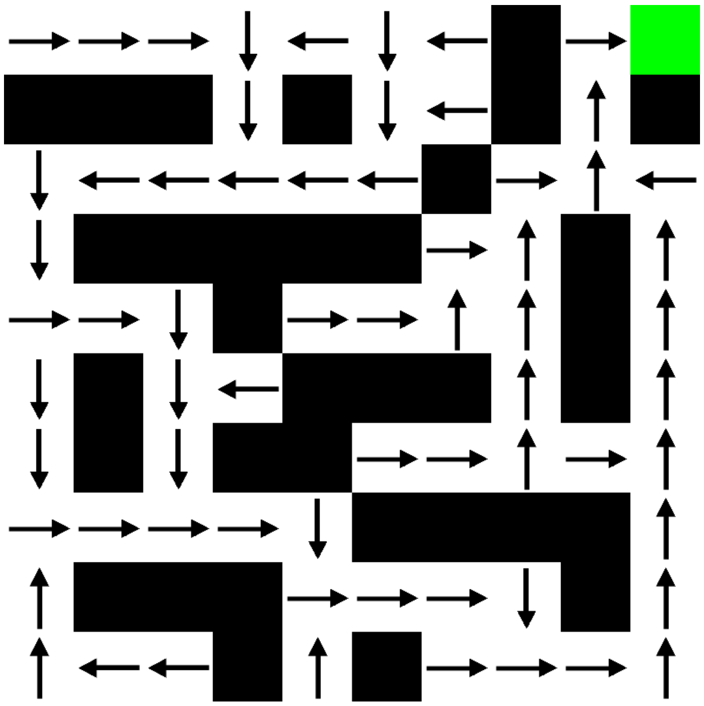


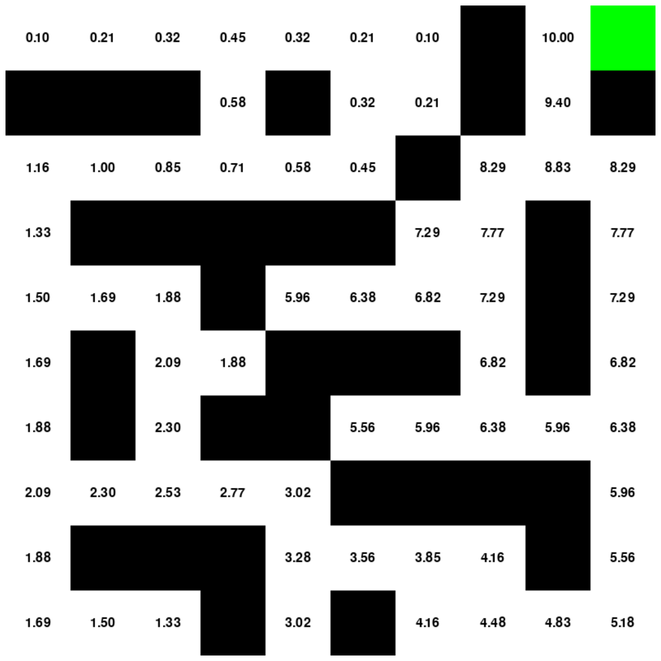
Figure 15: Visualization of the Q-Learning algorithm results
(a) State-value function - (b) Policy function.

2. Model-based Reinforcement Learning

1. Dyna-Q

For implementing the Dyna-Q algorithm, I did the environment model as a dictionary which is initialized empty and updated with every collected state and action. I then use this dictionary to do the simulated (indirect) RL for n=5 times. In each time, I sample a state and an action that was previously done by sampling from the dictionary (as it was filled by the encountered states and actions). Figure 16 shows a visualization of the results of training a Dyna-Q agent on the maze.

(a)



(b)

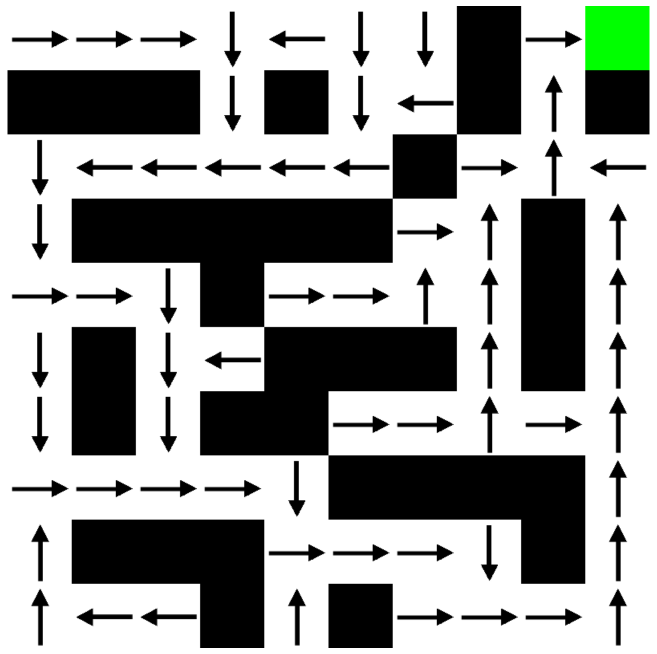


Figure 16: Visualization of the Dyna-Q algorithm results
(a) State-value function - (b) Policy function.

Numerical results:

- Last rolling average of training episodes cumulative rewards: 7.452
- Last rolling average of training episodes lengths: 18.92
- Mean of testing episodes cumulative rewards: 8.259
- Mean of testing episodes lengths: 18.417

Comparison with optimal functions:

- Hamming Distance (policy difference): 1
- MSE (value function difference): 0.0001

IV. Conclusion

Tabel 1 shows a summary of all results collected so far and can be used to compare how every RL algorithm behaved in comparison with the optimal policy π_* and optimal State-Value function V_*. Note that the absolute value of the MSE for a specific algorithm isn't an indicator of the performance of the algorithm as it's dependent on the scale of rewards in the environment. However, the relative difference between MSE for different algorithms can be used to compare performances.

Table 1: Summary of results of all RL implemented algorithms.

	Hamming Distance (Policy Difference with π_*)	MSE (State-Value function difference with V_*)	Mean Rewards in Testing	Mean episode length in Testing
SARSA	8	2.168	8.263	18.374
Expected SARSA	5	0.772	8.286	18.135
Q-Learning	1	0.0001	8.3	17.997
Dyna-Q	0	0	8.259	18.417

Looking at the MSE and hamming distance alone, it can be concluded that SARSA has the poorest performance of all implementations and Dyna-Q and Q-learning has the best performance. Notice that Q-learning nearly achieved optimal policy but because of a very small difference in one state it had one difference in the policy. If we are to consider the mean of rewards and episode length in testing the agents, Q-Learning slightly gives higher performance than Dyna-Q. Indicating that both algorithms. I can say that Q-Learning and Dyna-Q both achieved an optimal policy while SARSA and expected SARSA achieved an approximate optima policy.