Report Assignment 4 for APML24/25@UU

Rens van Moorsel, Ruben de Groot, Ewoud ter Wee, Lucas van der Jagt
Group 17

January 26, 2025

Abstract

In this report, two different Reinforcement Learning algorithms are trained and evaluated to find the shortest path in a maze. The algorithms are QLearning and SARSA. For QLearning and SARSA different hyperparameters like exploration rate, learning rate and discount factor are tested. Rewards and human interpretation is used to measure the quality of different hyperparameter configurations. The results are that Q-Learning and SARSA both perform well with an exploration rate of 0.1 and Q-Learning performs better using a high discount rate(0.99), while SARSA performs better with a lower discount rate(about 0.90). SARSA can be used for safe and low-risk pathfinding and Q-Learning can be used for aggressive high-risk path-finding.

1 Introduction

Reinforcement Learning algorithms are algorithms that use previous experience to learn about the problem it needs to solve. These algorithms are really useful for machine learning problems without labels, as they can learn based on interaction and rewards.

In this assignment two different algorithms of Reinforcement Learning are trained with different parameters. The goal is to find the optimal parameters and model for solving a maze. The objectives are: Formalizing a practical problem into a Markov Decision Process, gaining familiarity with the OpenAI Gym framework, applying SARSA and Q-Learning algorithms to solve the maze problem, evaluating the outcomes of the reinforcement learning process and interpreting findings and reflecting on the distinction between types of Reinforcement Learning algorithms.

The paper starts with an explanation about the Reinforcement Learning algorithms, followed by the methods and experimental setups of the experiment. After this, the results are visualised and discussed.

2 Data

- **Describing Markov Decision Process**

  Before we can provide an analysis of our Markov Decision Process (MDP) model, it is important to fully understand what an MDP entails. An MDP consists of several components:

  - **A finite set of states (S):** This represents the set of all possible states in which the agent can exist.
  - **A finite set of actions for each state (A):** This is the set of all possible actions the agent can perform.
  - **A transition probability:** This denotes the probability of transitioning to the next state, given that the agent takes a specific action. For instance, an agent in state S_0 may have a probability of 0.8 of moving to S_1 and a probability of 0.2 of remaining in S_0, depending on the chosen action.
  - **A reward function:** The reward received after performing a specific action in a particular state.
  - **A discount factor:** This is a value between 0 and 1 that weighs the importance of immediate rewards (a single beneficial action) against achieving the ultimate goal.

  A crucial aspect of MDPs is that the next state depends solely on the current state and the action chosen by the agent. This is referred to as the *Markov property.* As a result, the reward is based on the current state, the action taken, and the next state; previous events have no influence on the reward.

The action selected by the agent is determined by the *policy*. The policy defines which action should be taken in each state. This means that solving a MDP problem involves finding the optimal policy.

- **Exploring the MDP model**

  Now that we have an understanding of what an MDP entails, it is important to gain a clear idea of our specific MDP model. The problem under consideration is the MDP problem titled *"Robot in the Maze"*. The agent's task is to find the shortest possible route from its starting position to the goal at the end of the maze (the terminal state).
  The environment is represented as a maze in the form of a 2D grid. The grid consists of cells classified into four types:

  - **Free:** A cell accessible to the agent.
  - **Obstacle:** A cell that is inaccessible to the agent.
  - **Goal:** The terminal state or endpoint of the maze.
  - **Agent:** The current position of the agent within the maze.

  Considering how we have defined MDP above, it is interesting to analyze how it relates to this specific model.

  The *current state* of the agent is its current location in the maze. The *possible next states* are the free spaces directly adjacent to the current state.

  The actions the agent can perform are:

  - Move up
  - Move right
  - Move left
  - Move down

  The reward is calculated based on the agent's actions and the resulting state:

  - **+1:** For reaching the goal (*goal_reward*).
  - **<0:** For invalid actions, such as attempting to move into an obstacle or outside the grid (*invalid_reward*).
  - **<0:** For running out of time before reaching the goal (*timeout_reward*).
  - **<0:** For values on the path to the finish line, high values correspond to tiles that are closer to the finish.

  The agent can observe its current position within the maze. Additionally, it has full knowledge of the maze's layout. This means the agent knows exactly which cells in the maze are free spaces, obstacles, and the terminal state.

3 Methods and Experimental Setups

3.1 Model Selection

For this experiment, the three Reinforcement Learning models Random agent, Q-Learning and SARSA are used. Random agent is a RL algorithm choosing the next action randomly. It is a simple form of RL but it does not really learn anything and is inefficient.

Q-Learning is a bit complexer. The Q-Learning algorithm saves a table to save actions for certain states. The goal is to have a table with optimal actions for states. The table is updated based on new actions made. This algorithm is off-policy, which means that the algorithm learns about the optimal actions, and does not learn about the policy it is following.

The last model is SARSA. SARSA is an on-policy algorithm and stores information about the policy it is using, instead of saving the optimal actions. It looks a bit like Q-Learning but stores information about the policy instead of the optimal actions. This means, the values in the table are considering exploration strategies.

3.2 Implementation Details

For the implementation of the models, various libraries are used. To generate random numbers, the random library is used. Mazelab is used for the maze and plotting the mazes. To generate heatmaps, the Seaborn library is used. Numpy is used for calculations on numbers. For plots, the matplotlib library is used.

The three RL models are implemented by creating a class for each model. They all use an instance of TaskEnv to save the environment. This class is from the gymnasium library. Furthermore they have the parameters: exploration rate, learning rate and discount factor. The exploration rate determines how much the algorithm tries to find new solutions, instead of using already familiar options. The learning rate determines how much the model changes values with new information. The discount factor determines how much future rewards are worth in proportion to the current reward. All the models have a select action function, which selects the next action, and a learn function, which learns from the action.

The init function of the class sets the parameters, and the other functions are filled in based on how the algorithm of the model works.

The experiment is done by training the Q-Learning algorithm and the SARSA algorithm with different parameters. The resulting models are evaluated to determine which model and which parameters are the best.

3.3 Evaluation Metrics

To evaluate the resulting models, several methods are used. The first one is the total rewards a model gets. This value tells something about how good the model performed while solving the maze. A high value means a good solution. With graphs for each parameter the best values for parameters can be found.

Besides the total rewards, other evaluation methods are also used. For each model, an animation of the solving is made, so the process of the solving can be seen. Furthermore the agent brain is shown. This is a visual where can be seen what action the model would choose for each position in the maze. With this, it can be easy seen if the choises a model makes are good choices. Next to the brain, the highest state value for each position in the maze can be seen here. The state value is the expectation of the reward from the next action if it is in a position.

4 Results & Discussion

4.1 Random Agent

Before moving into the actual algorithms, as a point of reference let's first see what happens if we just let a robot randomly traverse the maze, which does not use any reinforcement learning. In figure 1 and 2 we see that there is no convergences in the Q-tables and hyperparameters have no influence whatsoever on anything, as no learning is actually happening and the agents just randomly go through the maze until they finish or hit a timelimit.

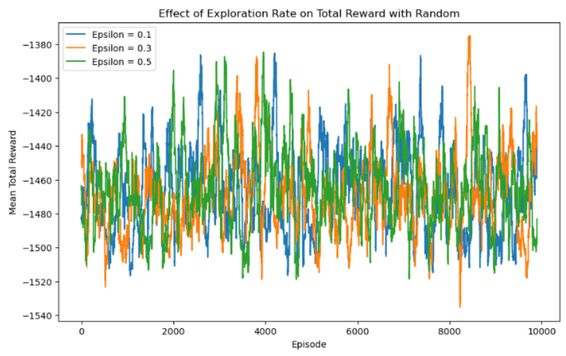

Figure 1: Episodes plotted against mean total reward per episode for different explorations values with a random agent

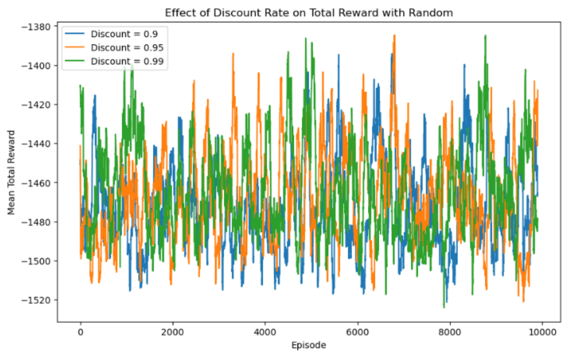

Figure 2: Episodes plotted against mean total reward per episode for different discountvalues witha random agent

4.2 Exploration Rate

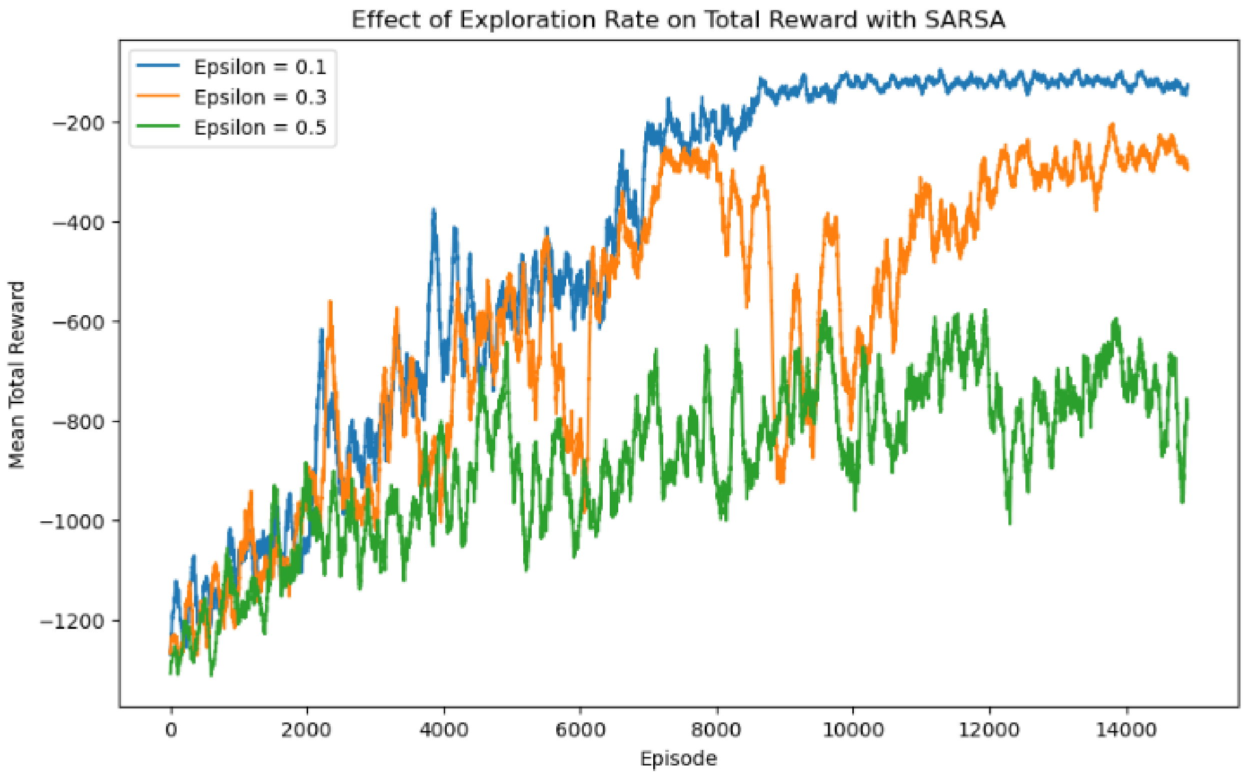

Figure 3: Episodes plotted against mean total reward per episode for different explorations values with SARSA

In figure 3 we can see the episodes plotted against the mean total rewards per episode for SARSA, averaged over 5 different agent trainings. For each graph, the exploration rate was varied, to see the impact the exploration rate has on the convergence of the Q-table. 3 explorations rates were tested: 0.1, 0.3 and 0.5, which corresponded to the different probabilities of exploring the maze, instead of exploiting. From the results in figure 3 we can see that epsilon=0.1 resulted in converging relatively quickly and in a stable matter and with quite an optimal Q-table. Epsilon=0.3 and epsilon=0.5 however resulted in poorer Q-tables with poorer rewards and also struggle to converge properly. For epsilon=0.3 the Q-table looked like it was converging around episode 7000, but the high exploration rate caused it to diverge away from the optimum. Later, around episode 11,000 it converges back again, but there's still a lot more fluctuations than with epsilon=0.1 and worse rewards as well. When looking at epsilon=0.5, the exploration rate is so high, you can barely see it converging as the fluctuations are so large, because we explore 50% of the time. For SARSA, epsilon=0.1 seems to be a good exploration rate.

Looking at figure 4, we can see again the episodes plotted again the mean total reward for different epsilons, but this time for the Q-Learning algorithm. Again epsilon=0.1 seems to perform the best in both Q-table convergence and maximum rewards. However we also see that these graphs look a lot more stable in its fluctuations. This is because it updates based on the highest possible reward, leading it to choose more high-reward paths. SARSA updates on the path actually chosen, which means it explores more paths that are safer and low-risk, leading in big fluctuations of rewards.

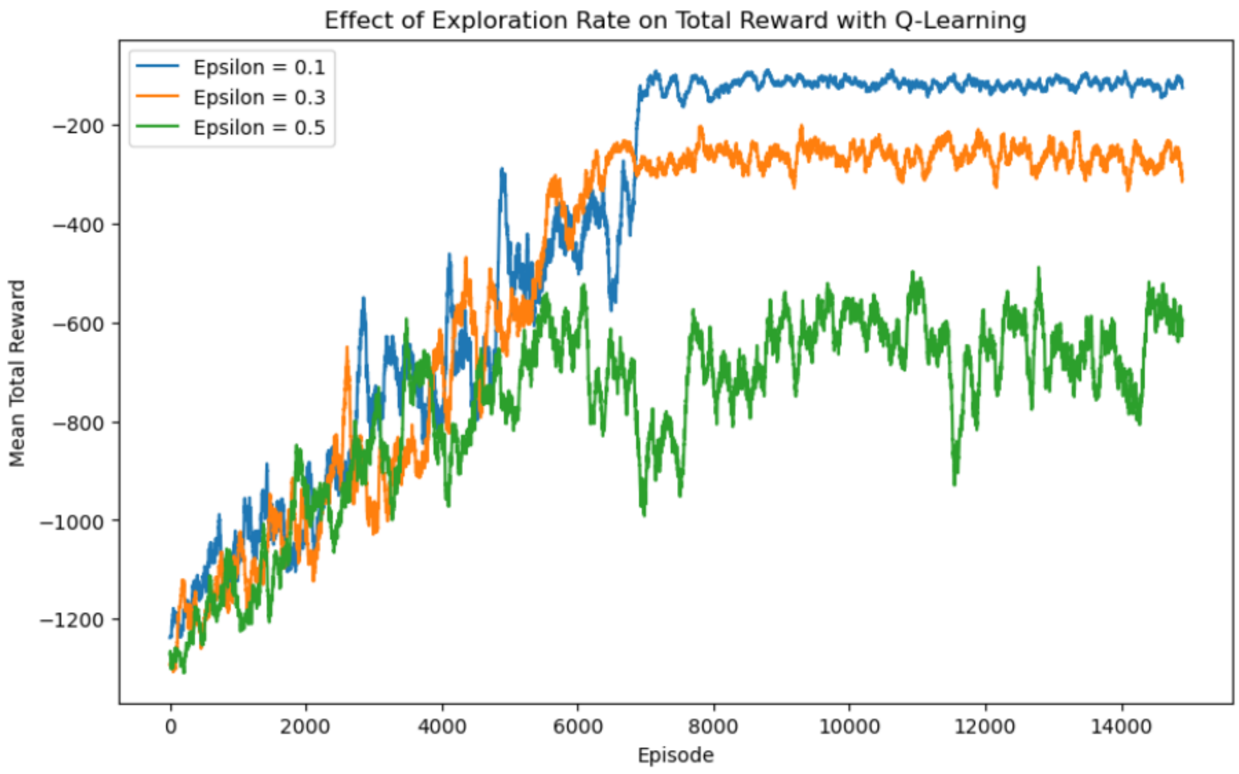

Figure 4: Episodes plotted against mean total reward per episode for different explorations values with Q-Learning

4.3 Discount Rate

In figure 5, we see the episodes plotted against the mean total reward per episode for different discount rate values with SARSA. In this figure we can see that SARSA converges faster with discount=0.9 compared to higher values. Given enough episodes however, the higher discount values also converge to the same optimum. The reason behind SARSA performing better with a lower discount value, is again because SARSA performs its updates on the action taken, which results in more exploratory moves. A high discount value means it can overvalue future rewards that can be uncertain.

In contrast, in figure 6 we see the same graph plotted for Q-Learning. Q-Learning actually performs slightly better with a high discount rate. Again, because Q-Learning bases its actions off of the maximum possible future reward, in combination with a higher discount rate it focuses even more on actions that have a maximum future reward. Making it a very aggressive combination of looking for the most optimal path, as the more optimal actions are counted even heavier.

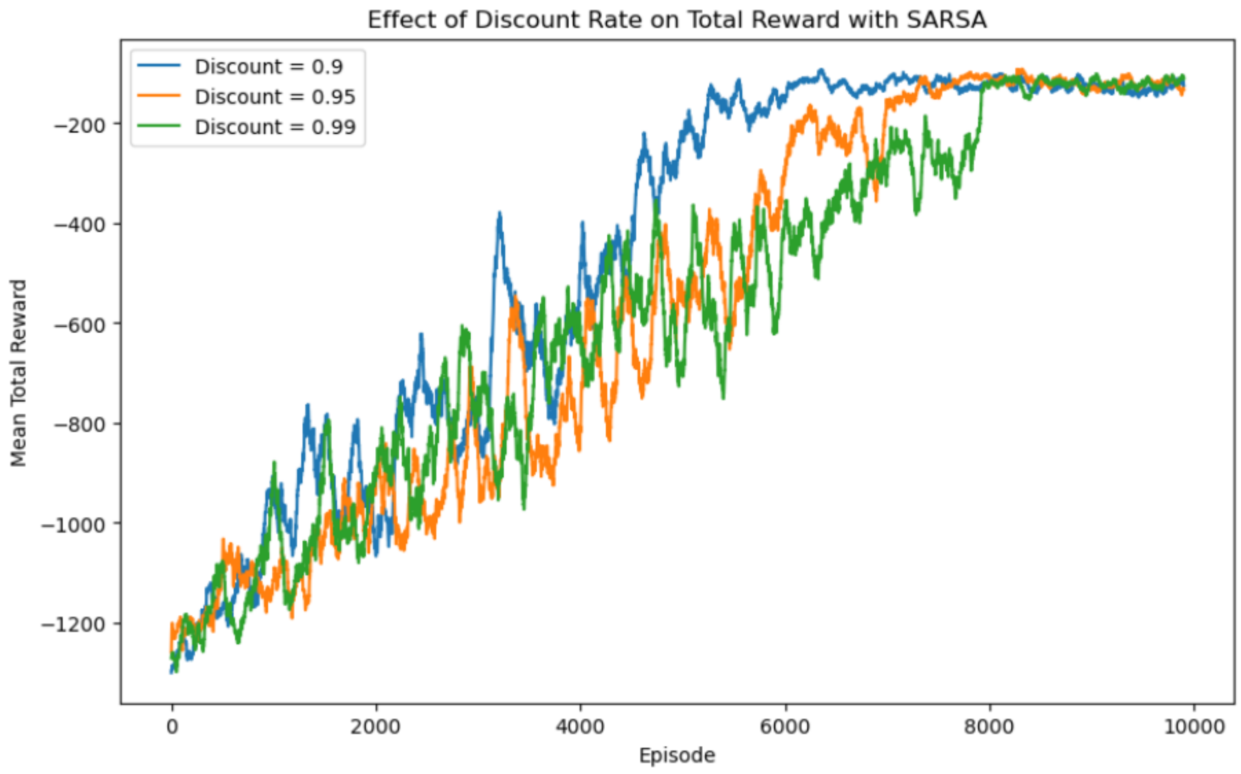

Figure 5: Episodes plotted against mean total reward per episode for different discount values with SARSA

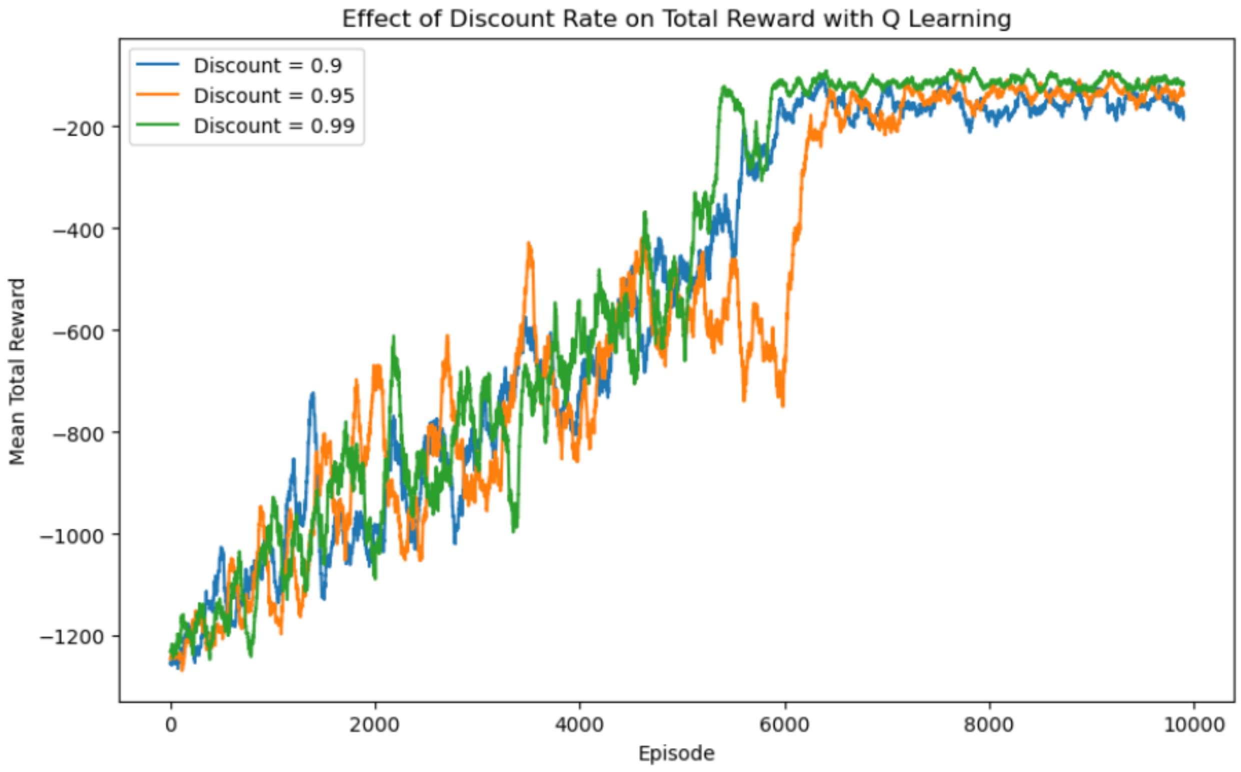

Figure 6: Episodes plotted against mean total reward per episode for different discount values with Q-Learning

4.4 Q-Tables

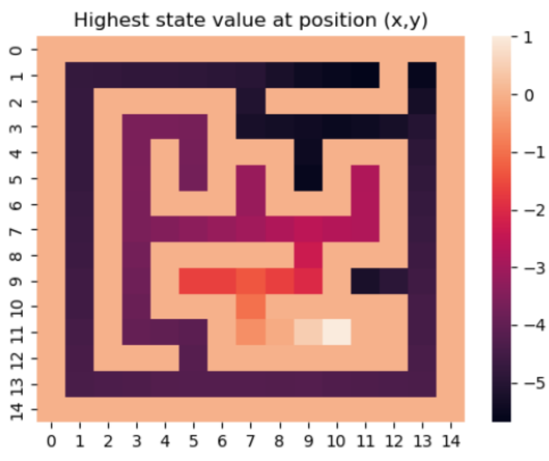

Figure 7: SARSA Q-Table

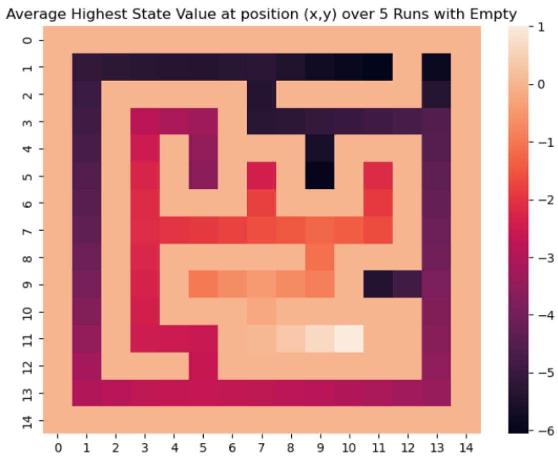

Figure 8: Q-Learning Q-Table

After setting setting the hyperparameters to the optimal found values within our maze, we can now compare the Q-Tables from both algorithms after learning. For both algorithms 5 agents were trained in the same maze and the average of their Q-tables was taken. In figure 7 we can see the Q-table that was learned by the SARSA algorithm and in figure 8 we can see the Q-table learned by the Q-Learning algorithm. These 2 visualizations really highlight the difference between on-policy and off-policy. For Q-Learning, which is an off-policy method, higher Q-values seem to extend further into the maze. Q-Learning is a lot more optimistic in its path-finding, updating on its maximum possible future reward instead of the actual action taken. If we look at the dead-ends in the top section of figure 8, we can see this visualized by the purple gradients still existing, meaning Q-Learning might still take some potential in those options. If we now compare this to SARSA's on-policy method, which updates the Q-table based on the actual actions taken instead of the most optimal one, the property of on-policy's risk aversion is highlighted. In the same dead-ends in the top section of figure 7, we no longer see any purple gradients and instead we see many more black gradients, meaning SARSA will really try to avoid these areas of the maze, as they are high-risk.

4.5 Sources of Error

A quick note to make before moving on to the conclusion, is that these graphs can be rather depending on the complexity of the maze that is chosen. I.e. in easier mazes the Q-tables differentiate less and the fluctuations within the graphs will also be less prominent with less complex mazes.

5 Conclusions

In conclusion, the Q-Learning and SARSA algorithms perform best with an exploration rate of around 0.1 from the exploration rates that were tested. Q-Learning goes well in combination with a high discount rate(0.99) for aggressive maze solving and SARSA performs a bit better with discount rate that are slightly lower(about 0.90), so it does not overvalue paths that lead nowhere. Q-Learning is more stable in its learning due to the properties of being on-policy and just looking for the best path as soon as possible, leading in less fluctuations during learning compared to SARSA. Q-Learning is a good algorithm for getting a high-risk optimal path and SARSA is a good algorithm to get a safer less optimal path, both have their own value.

One hyperparameter that was not tweaked was the learning rate. For future research, one can look at the influence of the learning rate on the model performance. Not only statically changing it, but adding a learning rate decay is also a very popular choice in machine learning algorithms[1]. Starting off with a very high learning rate, when we do not know anything yet and then during the process decreasing it, so the learned knowledge becomes less susceptible to change. In the same way, adding an exploration decay could also prove beneficial, we mainly want to explore much in the beginning, but less towards the end of learning. If fast hardware is present, it is also very possible to of course test out more hyperparameter configurations, with more agents at the same time, to get more averaged results.

References

[1] GeekForGeeks. Learning rate decay, 2023. URL: `https://www.geeksforgeeks.org/learning-rate-decay/`.

6 Contribution

Coding:

- 2.2: Lucas, Ruben
- 2.3: Lucas, Ruben
- 3: Ruben

Report:

- Abstract: Rens, Ruben
- Introduction: Rens
- Data: Ewoud
- Methods and Experimental Setups: Rens
- Results & Discussion: Ruben
- Conclusion: Ruben