RL Assignment

Group 7: Paul Bonnie, Nicolas Forcella, Zheyan Lin, Tijmen Westeneng

April 3, 2025

Contents

1 Introduction **2**

2 Dynamic Programming algorithms **2**
2.1 Policy Iteration 2
2.2 Value Iteration 3

3 Monte Carlo algorithms **4**

4 Temporal Difference algorithms **6**
4.1 SARSA 6
4.2 Q-Learning 7

5 Comparison and Discussion **9**
5.1 Mean Squared Error Analysis 9
5.2 Cumulative Rewards Analysis 10

6 Conclusion **11**

1 Introduction

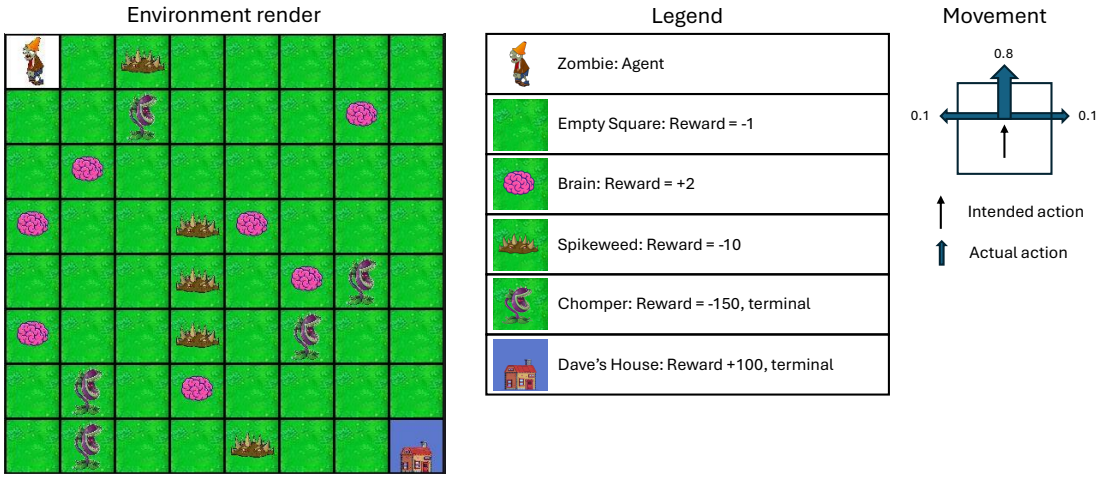


Figure 1: Render of the Grid Environment, Reward Structure, and Movement in the Environment

The aim of the present report is to describe and compare multiple common reinforcement learning algorithms. The performance of the algorithms will be compared in a custom environment inspired by the game *Plants vs Zombies*.

In the environment used in the current project, the zombie is the agent and wants to invade Dave's house while avoiding various obstacles. Figure 1 shows a render of the 8×8 grid environment. The zombie starts in the top left square and wants to reach Dave's house in the bottom right square. On its way to Dave's house, the zombie can consume brains, which provide a reward of $+2$. Furthermore, if the zombie steps on a Spikeweed, it suffers damage, which is associated with a reward of -10. If the zombie steps on a Chomper square, the Chomper will devour the zombie, resulting in a reward of -150 and terminating the game. Finally, if the zombie manages to reach Dave's house, it will receive a reward of $+100$, and the game will end. Additionally, the empty squares have a reward of -1 as the zombie wants to reach the house quickly.

In each non-terminal state, the zombie has four possible actions to choose from: going up, right, down, or left. The environment is stochastic, meaning that the zombie does not always perform its desired action. As shown in the right part of Figure 1, the probability of moving in the intended direction is 0.8 while the probabilities of moving in directions one off of the direction action are 0.1. For example, if the zombie wants to move up, it actually moves up with a probability of 0.8, while going to either left or right with a probability of 0.1 each.

In the following sections of the report, we will introduce several common reinforcement learning algorithms. These algorithms will be applied to the environment introduced in this section, and their performance will be evaluated. Finally, the strengths and limitations of the algorithms will be discussed, and the algorithms will be compared.

2 Dynamic Programming algorithms

2.1 Policy Iteration

The Policy Iteration algorithm consists of policy evaluation and policy improvement.

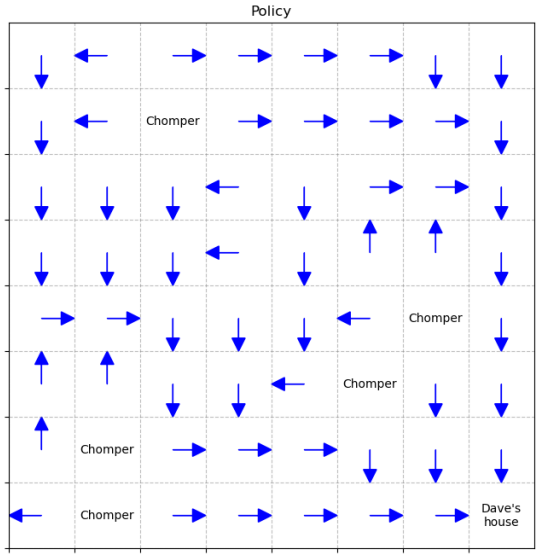
Policy evaluation updates the state value table iteratively using the Bellman equation until the maximum change (δ) is smaller than θ (a manually defined convergence threshold). Policy improvement refines the policy for each state by computing and selecting the action that maximizes the action-value function. The process repeats until the policy stabilizes.

2.2 Value Iteration

In the Value Iteration algorithm, a single iteration updates state values by computing state-action values using the Bellman equation and selecting the maximum between all actions. This process repeats until the difference between the updated and original value tables falls below θ. Once converged, the policy selects the action that has the highest value.

Policy Iteration alternates between policy evaluation and policy improvement. The number of steps of policy iteration needed is comparatively low. However, policy iteration requires full policy evaluation at each iteration step, so the computation could be expensive. Value Iteration updates state values directly in each iteration and extracts the policy only at the end. In each iteration, the computation is lighter, but it may take more iterations to achieve convergence.

However, both algorithms are able to solve the MDP and guarantee convergence to the real value table and policy. Therefore, in the following implementation of algorithms, we will use the value table and policy from policy iteration as the ground truth, testing the accuracy of other algorithms' training results.



(a) Policy and Value Iteration Policy

Value Function

19.06	16.58	-2.41	19.64	22.57	25.71	29.06	29.86
22.5	20.02	0.0	21.74	25.17	29.01	30.23	34.12
26.28	26.17	28.0	23.55	26.82	29.31	33.8	38.75
26.48	29.74	32.36	27.86	27.56	26.53	30.74	44.36
29.5	33.55	38.85	37.58	32.79	10.97	0.0	51.58
26.61	31.01	45.88	52.58	40.16	0.0	56.93	84.25
6.94	0.0	53.77	58.03	66.58	77.57	87.35	96.94
-2.17	0.0	53.0	59.24	75.58	86.37	96.94	0.0

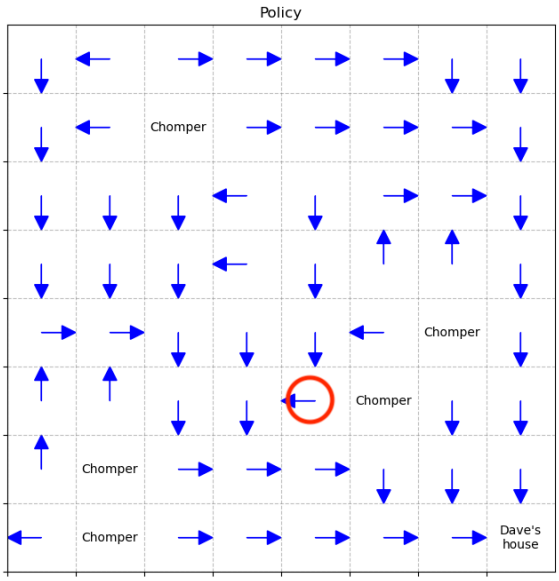
(b) Policy and Value Iteration Value Function

Figure 2: Policy and Value Function produced by Policy and Value Iteration

Figure 2a and Figure 2b show the optimal policy and value table produced by both Policy Iteration and Value Iteration.

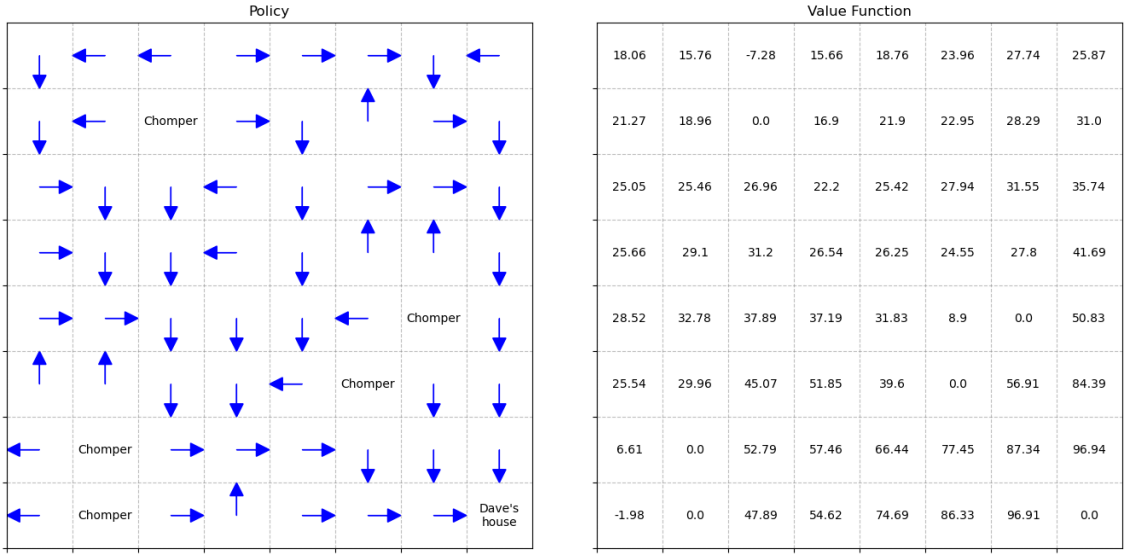
When observing the optimal policy produced by policy iteration, the action for some states may appear less reasonable at first sight. For example, for the square indicated by the red circle in Figure 3, we could expect that the agent should go down to reach the goal faster, while the optimal action is actually going left. This happens as the chompers have a large negative reward and terminate the episode, and the environment has stochastic movement. Since there is a 0.1 probability of going right when the action is down, the agent would rather take a few additional steps by going left than risk running into a chomper.

Figure 3: Example Policy Choice



3 Monte Carlo algorithms

Monte Carlo methods rely on sampled episodes of experience to learn the optimal behavior in a given environment. It does not require knowledge about the MDP that describes the environment. Monte Carlo algorithms alternate between evaluating the current policy and improving the policy based on the evaluation. Monte Carlo estimates the action values by sampling episodes from a policy π. In first visit Monte Carlo, $q_\pi(s, a)$ is estimated as the average of the returns following the first visit of the state-action pair (s, a) in each episode, while every visit Monte Carlo averages all visits. Monte Carlo control uses these estimates to approximate optimal policies by choosing a greedy policy based on the current action value function. If run infinitely, Monte Carlo converges to the optimal policy. However, a problem is that many state-action pairs are never visited if π is a deterministic policy, and therefore the value for these state-action pairs will not be updated. This problem can be solved by using exploring starts.



(a) Monte Carlo Policy

(b) Monte Carlo Value Function

Figure 4: Policy and Value Function produced by first visit Monte Carlo exploring starts

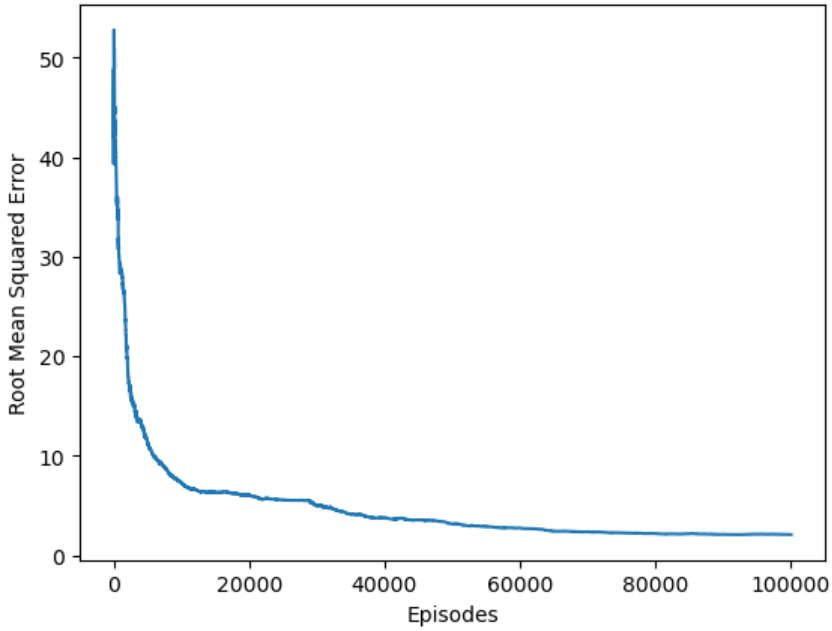


Figure 5: Root Mean Square Error (RMSE) of first visit Monte Carlo exploring starts

In the current project, we implemented first visit Monte Carlo with exploring starts. Figure 4a shows the policy produced by the algorithm after 100,000 episodes. Figure 4b shows the estimated value function of the Monte Carlo algorithm. The values from the Monte Carlo algorithm are within a small margin of the value function computed by the Policy and Value Iteration algorithms shown in Figure 2b. Note that there are minor differences between the policy and value functions produced by the Monte Carlo algorithm and the ground truth, which occur since only a limited number of episodes have been run. As the number of episodes goes to infinity, the policy and value function of the Monte Carlo algorithm approaches the optimal policy and value function. Figure 5 shows the root mean square error (RMSE) of the algorithm. It can be observed that the RMSE approaches zero as the number of episodes increases.

4 Temporal Difference algorithms

4.1 SARSA

SARSA algorithm updates a state-action value table after each step taken using the following update formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

The next action a'_{t+1} is selected based on an epsilon greedy policy. Since SARSA is an on policy method, we want it to converge to a deterministic policy, therefore, we need ϵ to go to 0 over time. For this we implemented an exponential decay function that decreases ϵ over the episodes and scales depending on the number of episodes to run.

$$\epsilon = e^{\frac{-10 decrease\_rate}{total\_episodes} episode\_number}$$

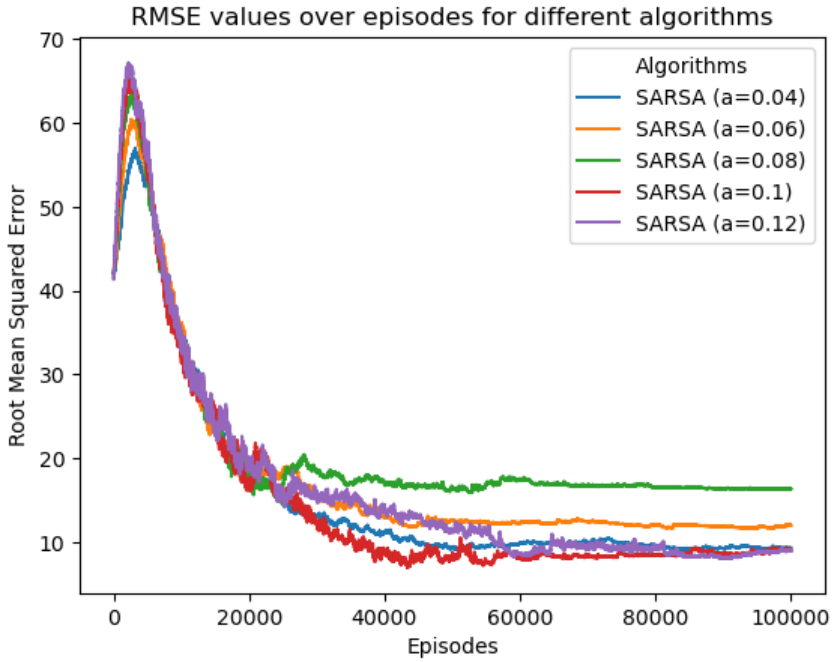
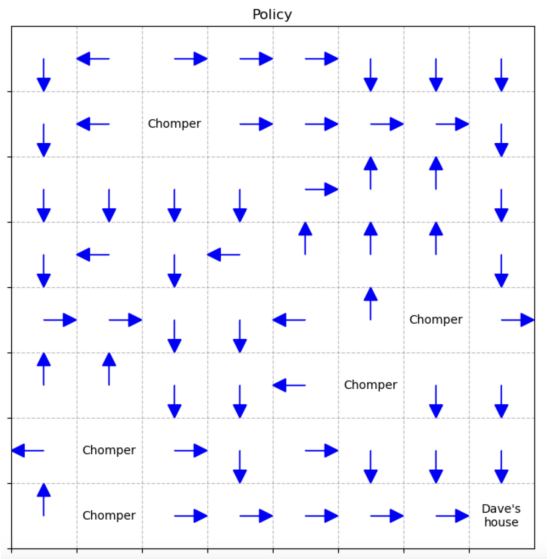


Figure 6: Comparison between different alpha values for the SARSA algorithm

We experimented with different α(s) for better convergence of the algorithm. Figure 6 shows the RMSE of each α after running 100000 episodes. From the result, we observed that although different α values do not differ greatly, $\alpha = 0.1$ performs best at RMSE convergence. So we choose $\alpha = 0.1$ for our final plotting.



(a) SARSA Policy

Value Function

15.7	13.57	-5.71	10.94	15.42	17.87	19.79	16.91
19.21	16.28	0.0	9.49	18.23	21.92	22.18	25.07
22.27	20.75	24.49	13.52	16.77	19.45	23.03	29.9
22.15	22.87	28.45	22.94	14.18	14.9	20.58	37.32
25.75	31.04	37.71	29.56	19.86	-7.17	0.0	39.19
23.4	26.8	44.48	45.9	33.7	0.0	73.14	89.24
-12.7	0.0	49.23	52.05	65.93	78.23	87.53	97.02
-32.57	0.0	52.73	57.51	75.12	86.18	98.35	0.0

(b) SARSA Value Function

Figure 7: Policy and Value Function produced by SARSA

In Figure 7, the policy matrix and value matrix for the SARSA algorithm after 100,000 episodes can be found.

4.2 Q-Learning

The Q-Learning algorithm is built on the same principles as the SARSA algorithm, but differs in that it's an off-policy method. This means that when estimating the value of the next state, it's not using the current policy for choosing the Q-value, but instead uses a greedy policy (which means it just takes the max Q-value of the next state). The update formula thus looks like this:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_a Q(s',a) - Q(s,a) \right]$$

This is off-policy, because it doesn't actually use this pure greedy policy to choose the action when it arrives at the next state. This allows for much more aggressive exploration, but might lead to risky paths, instead of the safer paths SARSA mostly finds.

The epsilon decreases linearly over the episodes from 1 to 0, which we have found to be the perfect balance between exploration and exploitation.

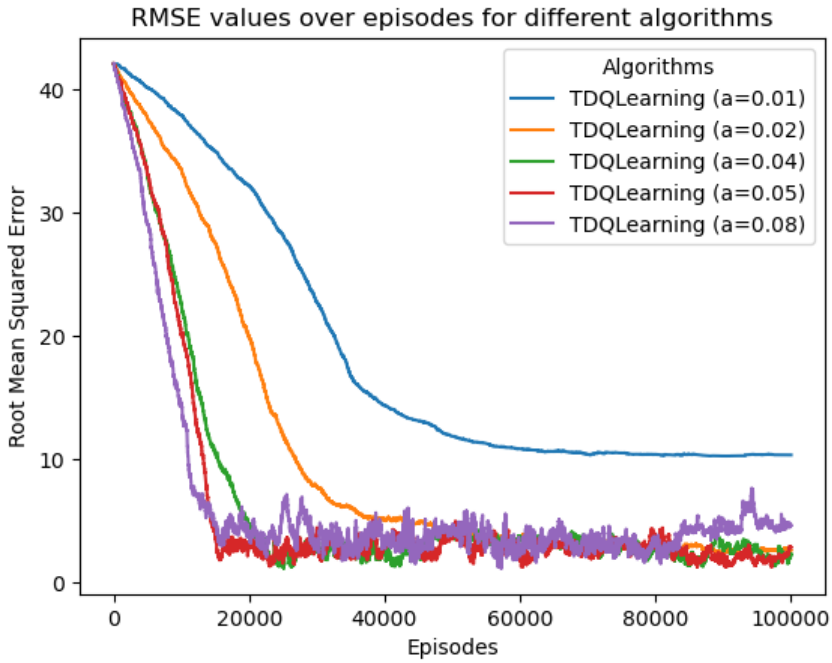
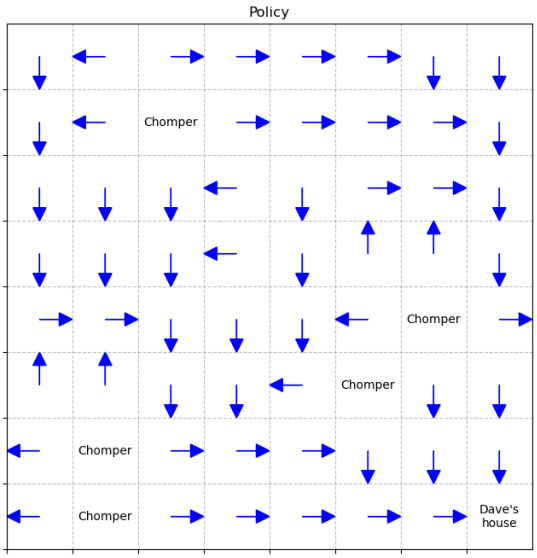


Figure 8: Comparison between different alpha values for the Q-Learning algorithm

The main factor influencing Q-learning performance is the learning rate α. That's why we tested different values for alpha and plotted the RMSE during training for these values in Figure 8. It can be clearly seen that an alpha of 0.01 is performing much worse than the other values we've tried. This can be explained by the fact that because the epsilon is decreasing over time, the algorithm will not explore many states outside of the general path when epsilon gets low enough. This means, to reach a low RMSE, it's important to have the values in these non-exploited states change quickly in the beginning and an alpha of 0.01 is not enough for this. Out of the other values we see a similar performance, although we chose 0.05 as having the best performance. Finally, Figure 9 shows the policy matrix and value matrix for the Q-learning algorithm after 100,000 episodes.



(a) Q-learning Policy

Value Function

19.48	17.05	-13.07	18.24	20.8	23.97	27.09	27.39
22.47	20.23	0.0	20.69	23.21	26.35	26.88	29.19
26.16	26.32	28.66	22.25	26.07	26.08	29.05	31.35
26.58	29.58	33.23	27.27	28.55	24.54	28.4	36.02
29.93	34.22	40.08	38.13	35.25	16.29	0.0	42.62
26.59	31.18	46.8	53.53	41.5	0.0	56.26	82.27
7.2	0.0	54.56	57.56	64.0	78.43	86.89	95.65
-2.14	0.0	53.87	57.77	72.38	87.37	98.25	0.0

(b) Q-learning Value Function

Figure 9: Policy and Value Function produced by Q-learning

5 Comparison and Discussion

In order to compare the performance of the different algorithms, we used two methods:

1. Once all of the algorithms are fine-tuned, we run a training instance using the best hyper-parameter found. We then plot the root mean squared error (RMSE) over the episodes on a single plot.

2. When training the algorithm, every 1000 episodes, we run the agent following current policy and we compute the cumulative reward obtained. We repeat this process 5 times and plot the average value obtained for each episode number.

5.1 Mean Squared Error Analysis

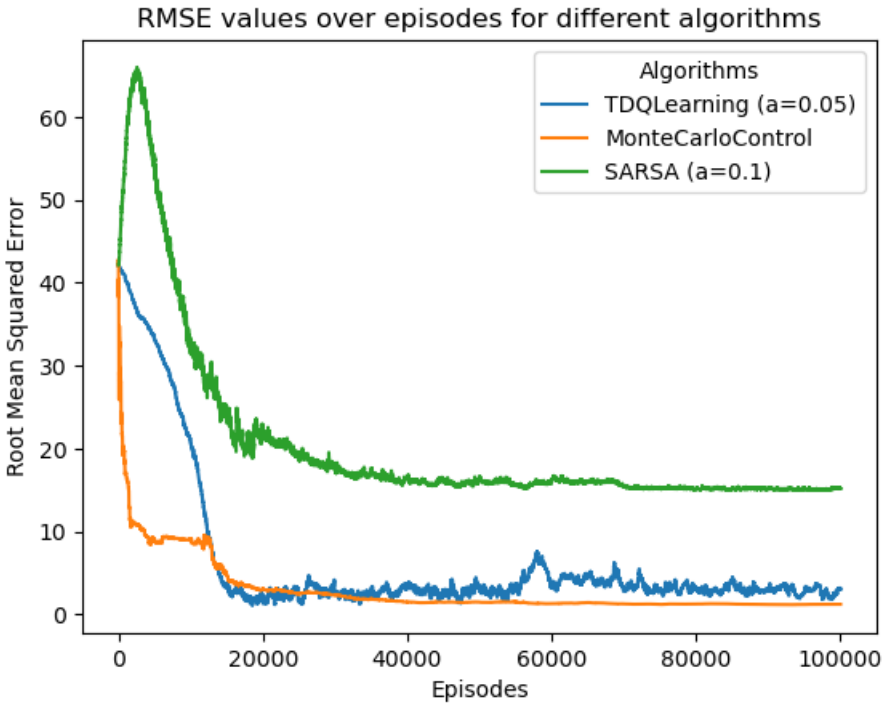


Figure 10: Comparison of RMSE between algorithms

Figure 10 shows that all three algorithms converge to a rather small RMSE after 100000 episodes. Monte Carlo and Q-Learning converge to a Root Mean Squared Error (RMSE) of less than 5 within 20,000 episodes. SARSA, the other TD algorithm, performs significantly worse with a minimum RMSE close to 18. Additionally, it can also be noted that SARSA shows more variability in the RMSE than the other algorithms.

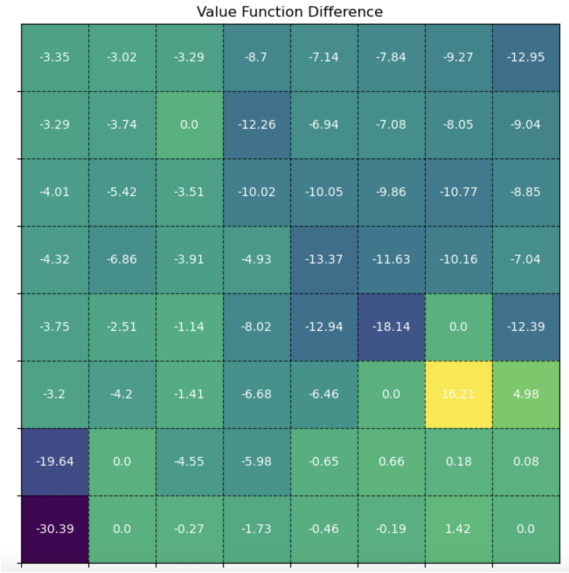


Figure 11: Value table difference between SARSA and Policy Iteration

We plotted the value differences between SARSA and our ground truth values and observed from figure 11 that SARSA's values tend to be much lower around cells of chompers. This explains the higher RMSE of SARSA. We believed that SARSA is behaving this way because of its updating rules. In the exploration phase, SARSA chooses all possible actions for the given state. So it's more likely that in this phase the agent steps into the chomper and gets the large negative reward, and the reward will be propagated to the Q_S_A table. Even though the updating rule averages out this negative reward over time, it does not eliminate it entirely.

5.2 Cumulative Rewards Analysis

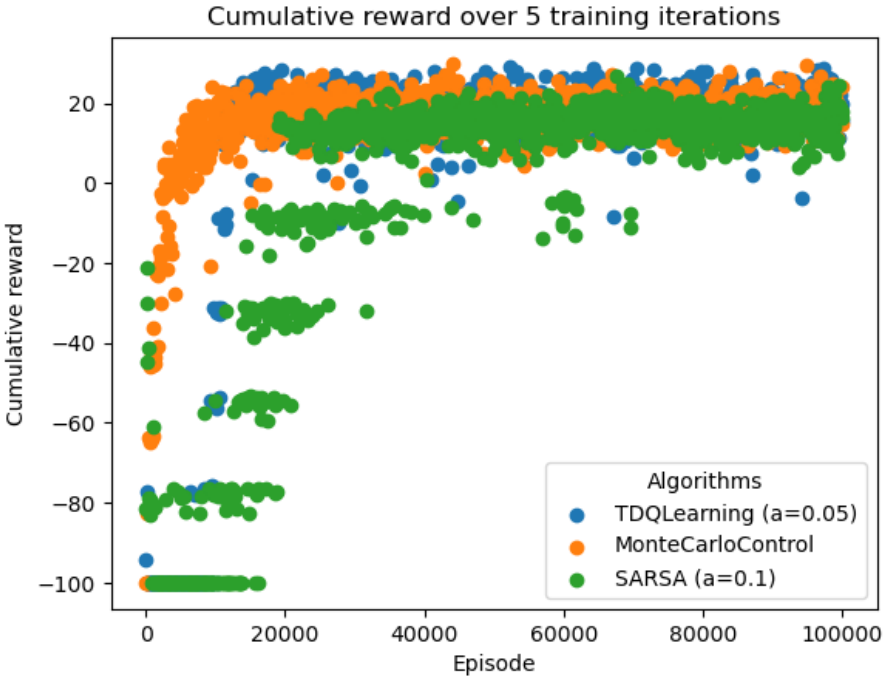


Figure 12: Comparison of cumulative reward between algorithms

In Figure 12, a comparison between the cumulative rewards over 5 training iterations for the different episodic algorithms can be found. There is clearly an upwards trend visible in all of the algorithms with respect to the episode numbers. This is what we expected, since the cumulative rewards should increase once better policies are found by the algorithms. What is further interesting to note is that although SARSA's rewards are increasing slower, they seem to converge to approximately the same value as the other two algorithms. As we've seen earlier, this is not the case for the RMSE value of SARSA. This indicates, that although the values of the states may be far off, the actual policy that is found by SARSA is of similar quality as the policies found by the other algorithms.

6 Conclusion

In this project, we set up a non-deterministic tabular environment with an associated agent to compare the performance of five different reinforcement learning algorithms divided over three categories: Dynamic Programming (DP), Monte Carlo Methods (MC) and Temporal Difference Learning (TD). Using the values calculated by the DP algorithms (Value Iteration and Policy Iteration) as ground truth, we can conclude that for our environment, MC is the best-performing algorithm, closely followed by Q-Learning, while SARSA performs slightly worse. However, all algorithms converge to a near-optimal policy that results in similar average cumulative rewards compared to the theoretical optimum value. In the future, it would be interesting to further investigate the interaction between different epsilon policies and alpha values on the performance of the algorithms.

In conclusion, all algorithms can be used to train our created agent to successfully navigate the environment, while following a near-optimum path.