INFOB3APML: Assignment 4

Group 2

Lorenzo Marogna Justin Nguyen Tomás Tavares Minna Vainikainen

January 2024

Abstract

This report intends to give account for the application of Reinforcement Learning (RL) algorithms to solve the problem of navigating a robot through a maze. The research involves formulating the problem as a Markov Decision Process (MDP) and using the OpenAI Gym framework to implement RL agents. SARSA, Q-learning and expected SARSA algorithms are used to solve the MDP problem. The results of the reinforcement learning process are evaluated and interpreted to reflect on the differences between the three RL algorithms. The results show that expected SARSA is the fastest learner among the agents. Overall, the results show that a lower discount value reduces the impact of the updates in the learning process, making it slower.

1 Introduction

The idea that we learn by interacting with our environment is relatable for everyone, and is the basis for how we think about the nature of learning. This is a well-explored area in the field of machine learning and this study will focus on the computational approach to learning from interaction. Reinforcement Learning (RL) is the subfield of machine learning that deals with this. To understand optimal behavior, the agent has to discover which actions yield the most reward by trying them several times. In that way, it maps situations to actions and strives to maximize a numerical reward signal. In order to apply RL, the problem needs to be formalized into a suitable model. Markov Decision Process (MDP) is a mathematical framework to model sequential decision-making tasks and is commonly used in formalizing RL problems [3]. The purpose of this assignment is to address the challenge of navigating a robot through a maze, by formalizing the problem into a MDP and applying RL algorithms.

2 Data

In order to address the challenge of navigating a robot through a maze, this task aims to delve into RL algorithms. This challenge is about formalization, where we model and conceptualize this real-world problem into an appropriate mathematical representation like MDP. In a MDP, two entities are the main protagonists: the learner and decision maker is called the ***agent***, which in our case represents the robot. The robot interacts with the ***environment***, the maze. These interact continually, with the robot selecting actions and the environment responding to these actions by giving it rewards in the form of specific numeric values. The final objective of a learning process is to find the so-called *optimal policy*, where a *policy* is a strategy that an agent uses in pursuit of goals. It dictates the actions that the agent takes as a function of the agent's state. An optimal policy tends to guide the agent to the goal of the maze by first trying to get closer and closer to it. For every position in the maze, there is a certain state value, which represents the *closeness* of the position to the goal of the maze. State values are an important tool for the agent to learn and improve its policy, together with the so-called state-action values, which specify how good an action is given a certain state. As state values and state-action values both depend on the current policy of the agent, it is important that policy improvement and state values evaluation are performed concurrently.

For an environment to be modeled as an MDP, the markovian assumption must hold. That is, the probability of each possible value for S_t and R_t depends **only** on the immediately preceding state and action, S_{t-1} and A_{t-1}. We will later analyze in depth the structure of our environment and try to determine whether it can be modeled efficiently as an MDP problem. For now, it is important to notice that solving a RL problem usually involves delayed rewards and a need to trade off immediate and delayed rewards, as a greedy approach would often lead to suboptimal solutions.

The "robot in a maze" challenge is usually suitable to be described as a MDP because there is a natural notion for final time steps, which is when the agent-environment interaction breaks naturally into subsequences, such as finding the way through the maze. However, this depends on how the environment is implemented. In

reinforcement learning, the purpose of an agent is to maximize the total amount of reward it receives on an episode, which is the attempt of a randomly spawned agent to find the shortest path to the goal of the maze. This means that if we want the robot to perform a specific task, we must provide rewards in such a way that the robot maximizes them, and by that, it will also achieve our goal. It is thus important that the reward we set up indicates what we want to accomplish.[3]

The environment is set up by a pre-implemented maze structure. In this assignment the OpenAI Gym API will be used. The core Gym interface is Env and serves as a unified environment interface. It provides methods for resetting the environment's state, returning observation, and moving the environment with a time step and returning an observation, reward, etc. Gymnasium is a widely used standard toolkit for developing and comparing reinforcement learning algorithms.[1] The state of our agent is defined by the coordinates of the agent in the maze. Independently of the state, the agent can always choose among 4 different actions: *north*, *south*, *east*, and *west*. Different outcomes are possible, depending on the state and the chosen action: if the action is valid (it doesn't point to a wall of the maze) the agent deterministically moves in that direction, otherwise no movement is performed. If the agent bumps into a wall, the reward is -1. This means that bumping into a wall is undesirable, and is thus punished. On the other hand, when the agent reaches the goal, it receives a positive reward of 1. Finally, given an action that doesn't bump into a wall and doesn't reach the goal, the reward for that action is always

$$R = -(timer * 0.01) \tag{1}$$

where *timer* is the amount of steps that have been performed since the start of the episode.

Given the composition of our maze, it is true that the probability of transition in a state only depends on the preceding state and action. Indeed, moving right from a certain position on the maze certainly brings the agent to its right, without any influence from the previous path of the agent that brought it to that point. This is not true for what's regarding the reward: as it depends on the number of steps that have been done since the start of the episode, the next reward also depends on actions that the agent has performed in the past and not only the current one. More specifically, given one state, and given two actions which will lead the agent back to the state it started in (e.g. left and then right), the reward for the state will be worse in the second visit compared to the first one.

Given the absence of the Markov assumption for the specific implementation that we were provided with, we can't properly formalize the problem as an MDP. Although Temporal Difference (TD) methods are designed for MDPs, we'll see that their performance are promising for our usecase. A possible reason is the similarity between TD methods and Monte Carlo (MC) methods, where MC is proven to be more effective in solving a general class of problems besides MDPs. This is further explained in section 3. On the other hand, the bootstrapping aspect of this method, coming from dynamic programming, actually applies to MDPs, and the missing assumption might lead to unexpected results. In section 4.4 we will see the results of applying algorithms designed for MDP problems outside their realm.

3 Model Selection

In the realm of Reinforcement Learning, different techniques are available for tackling a problem. For a complete understanding of the environment, and therefore an accurate estimate of state values, one of the first techniques is called Dynamic Programming, which aims at a precise convergence based on bootstrapping, policy evaluation and policy iteration. DP assumes the knowledge of the transition's probabilities of the environment and is generally pretty expensive in terms of computational time. Another famous class of methods are the Monte Carlo methods, who aim at making state values converge by continuous experimentation of actions until the end of an episode, then updating all previous values based on the actual observation. MC methods allow the agent to learn state values from the actual experience, without the need of knowing the transition's probabilities.

We will explore today one of the most famous class of methods for Reinforcement Learning, which tries to get the best from both DP and MC methods. Temporal Difference (TD) methods, this is what they're called, use bootstrapping techniques from DP combined with update rules that come from actual experience like in MC. In particular, they perform a certain amount of steps from a starting state and update state value functions based on their previous knowledge of those states and the observation of the environment after performing the steps. The simplest update for state values in TD(0) - meaning that only one step is performed ahead - is the following:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2}$$

where the quantity inside the square brackets represents the error between the current estimate of the state value and the one at time $t + 1$. It is interesting to observe that, if the number of performed steps goes to ∞, then the update rule is the same as an MC method.[3] Therefore, an advantage of TD methods lies in the fact that there is no need for an episode to necessarily come to the terminal state, as it is in MC methods.

In this section, we will introduce and motivate some of the RL algorithms, such as SARSA, Q-learning and Expected SARSA, that can be used to learn from the environment. Their main difference lies in their update rule for the action state value.

3.1 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy temporal-difference algorithm that learns a policy for an agent based on the expected future rewards for taking an action a in state s (action-value function $Q(s, a)$). For each state-action pair, the estimated action-value is stored in a structure called the Q-table. The Q-table is updated as the agent explores the environment using the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{3}$$

where:

α = learning rate
γ = discount factor
(s, a, r, s', a') = quintuple of state, action, reward, next state, and next action

The SARSA algorithm is an on-policy method, which means that it learns from the actions taken by the agent. Consequently, the next action a' in the above Formula 3 must be obtained based on the policy learned so far. SARSA uses an ϵ-greedy exploration strategy to derive its policy from the Q-table. This strategy selects a random action with probability ϵ and the greedy action (the one with the highest Q-value) with probability 1-ϵ. Thus, the next action a' is obtained using an ϵ-greedy strategy.

SARSA is well-suited for our problem as it can learn online from incomplete information and adjust the Q-values based on feedback from the environment. Additionally, SARSA can balance exploration and exploitation through the ϵ-greedy strategy, allowing the agent to discover new states and actions while exploiting the current knowledge.

However, one limitation of SARSA is its tendency to be overly cautious, accounting for the potential error in its policy. This cautiousness may lead to suboptimal solutions in some cases.

3.2 Q-Learning

Q-learning is an off-policy TD learning algorithm that estimates the action-value function $Q(s, a)$. However, unlike SARSA, Q-learning updates the Q-values of the Q-table using the following rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{4}$$

where:

α = learning rate
γ = discount factor
(s, a, r, s', a') = quintuple of state, action, reward, next state, and next action

The Q-learning algorithm is an off-policy method, which means that the agent's policy for learning (exploration) is distinct from its policy for acting (exploitation). During the exploitation phase, the agents uses the same strategy as the SARSA algorithm, the ϵ-greedy strategy. When learning, the algorithm selects the next action a' according to the greedy policy, which always selects the action with the highest Q-value.

Q-learning can achieve faster convergence than SARSA since it does not depend on the current policy and always updates the Q-values towards the optimal ones. However, it can suffer from overestimation bias, where Q-values may be inflated due to the max operator and noise in estimates. Moreover, Q-learning can be less efficient than SARSA in some cases, as it ignores the exploration strategy and may learn a policy that is not feasible to follow.

3.3 Expected SARSA

For the bonus assignment, we've implemented an Expected SARSA agent. This agent is strongly related to the SARSA agent in section 3.1. The main difference between the agents can be explained by looking at the Q-Value update rule of the agents (eq. 3, 5). In contrast to the SARSA agent, the expected SARSA agent does not use one specific action to calculate the Q-value of the next state. Instead, the expected SARSA agent uses all possible actions in the next state and calculates the expected value over all actions. This is done by multiplying the Q-value of the actions in the next state by the probability that the actions are selected. All

calculated values are then summed, forming the expected value of the next state. The expected SARSA agent then follows the same procedure as the SARSA agent to update its Q-table for the current state.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \sum_{a} \pi(a|s_{t+1})Q(s_{t+1},a) - Q(s_t,a_t)) \tag{5}$$

where:

s	= Current state
a	= Current action
α	= Learning rate
r	= Reward
γ	= Discount factor
$\pi(a\|s_{t+1})$	= probability of an action

As seen in (Seijen, H et al (2009)[2], the expected SARSA agent should perform better than the SARSA agent. This is because the expected SARSA agent eliminates the variance that the SARSA agent creates when stochastically selecting an action in its Q-value update rule. The downside of using the expected SARSA agent is that it is computationally more complex, and thus more demanding during the training phase.

4 Evaluation & Discussion

In this section, we will firstly explore the result after training each model for a total amount of 5000 episodes with different values for the ϵ value. Each model was trained on the same maze, so that it is possible to compare their state-value functions and dive into the differences between strategies. Additionally, every model is compared with the actual optimal solution, which is computed using the Dijkstra algorithm. The process involves the conversion of the maze in a graph, in which the SSSH (single source shortest path) problem is solved deterministically, with the goal being the source of the problem. Sequentially, we will compare different models with objective metrics, running simulations between different models and focusing on exploring the influence of hyper-parameters in the results.

4.1 SARSA Agent

As the SARSA Agent policy is strictly dependent on the value of the ϵ parameter, we are firstly analyzing its influence on the results and the learning process of the model.

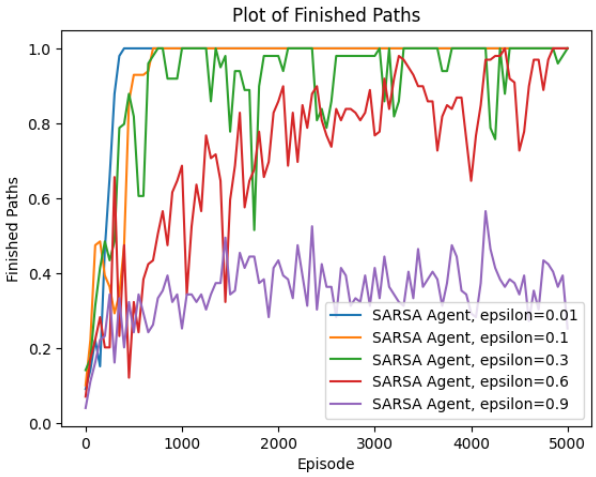

Figure 1: Percentage of starting positions whence the agent can find a path to the final goal within the time constraints

As we can see in figure 1, a lower ϵ value leads to stable results in a shorter time. This is coherent, since a highly randomly chosen ϵ often leads to choosing actions that the model already knows are bad, resulting in a slower process of understanding the value of 'actual' candidate actions. Furthermore, the closer ϵ is to 1, the more the model behaves as a random agent, which is surely performing poorly. On the other hand, we can observe that an agent with $\epsilon = 0.01$ always behaves with a greedy strategy, never looking for a better action than the actual one. Particularly, the figure shows the percentage of starting positions whence the agent successfully finds the goal within the time limit of 100 steps. Results show that a greedier agent is quicker in

finding a path (not necessarily optimal) to the goal. Given the simplicity of the maze, oftentimes the first path that is found by the agent is also the optimal one. Therefore, results for this model are close to optimal when comparing its rewards with Dijkstra's solver (figure 2).

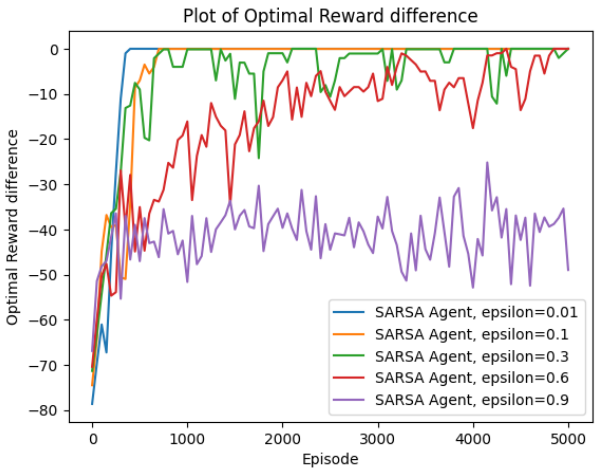

Figure 2: Difference between the average result of the agent and the optimal score

According to figure 2, it seems that the SARSA agent with $\epsilon = 0.1/0.01$ is equivalent to the Dijkstra solver after approximately 1000 episodes. This would be misleading in reality, as the difference between their result and the optimal result is close but not equal to zero. In fact, there are some positions in the maze in which the model has learned a way to the goal which is not optimal although efficient. This leads to a difference from the optimal reward which is slightly greater than zero, than averaged with tons of positions in which the model behaves optimally. We can see this aspect in the top left corner of the maze in figure 3(right), where the agents wants to come down from the left side of the board even when the path from the right side is found to be quicker by the solver. It is true though that the optimal agent's solution outperforms the agent's one by only a couple of steps, meaning that the difference in the state action values $q_*(s, a)$ of the two actions is actually very close and therefore needs a deeper convergence of the estimations $q(s, a)$ for a correct outcome. Indeed, state-action values for both options are evaluated positively by the agent, which might actually correct itself with a further training. For this reason, we believe that an ϵ value of 0.1 is the most suited for our task, guaranteeing some more exploration of the model at the cost of slower learning in the first iterations.

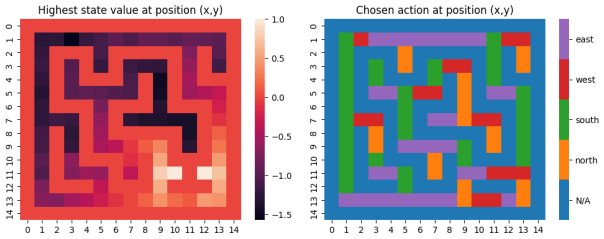

Figure 3: Visualization of SARSA Agent state value estimation(left) and best choice map (right)

Moreover, it is interesting to notice in figure 3 (left) how the state values vaguely reminds of a breath first visit of the maze, using the goal as a source. In particular, the state value $v(s)$ is proportioned to the opposite of the distance between the source and the state s. This observation further motivates the usage of Dijkstra's algorithm as a target result.

Overall, we can conclude that the agent is perfectly capable of learning the task of this specific maze. Given the intrinsic component of randomness during the training, the convergence to optimal results is sometime long to obtain in the furthest points from the goal of the maze, especially if multiple efficient options are available.

4.2 QLearning Agent

We initiated our analysis by determining the optimal epsilon value. It is crucial to note that the impact of epsilon on results is not as pronounced in QLearning as it is in SARSA. This distinction arises from QLearning being an off-policy algorithm, as explained in Section 3.2.

In SARSA, epsilon influences both the actions taken by the agent during exploration and the learning process. In contrast, in QLearning, it only affects the agent's exploration strategy in the environment. A low epsilon

corresponds to reduced exploration by the agent, while a higher epsilon leads to more extensive exploration.

This phenomenon is evident in the analysis of Figure 4, where both low epsilon (0.01) and high epsilon (0.9) can lead to learning the optimal solution. However, a lower epsilon promotes greater stability in the algorithm, as it is less prone to exploring suboptimal paths. To strike a balance between exploration and stability, we recommend using a more moderate epsilon value, such as 0.1.

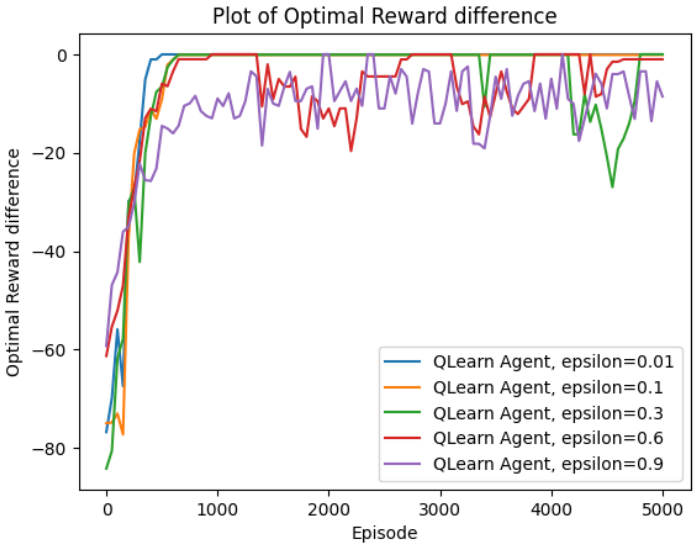

Figure 4: Ratio of optimal paths found, compared to the paths found by Dijkstra's algorithm

4.3 Expected SARSA Agent

Similar to section 4.1, we have evaluated several epsilon values on the expected SARSA agent. As seen in figure 5, an epsilon value in the range of 0 - 0.3 leads to a model where a path is found for all starting positions. The opposite is true for the models with a higher epsilon value. Just as we expect, due to the higher epsilon value, the agent will take more random actions during its training phase. Therefore it is less likely for the agent to find a path from all starting positions. As figure 5 shows, the epsilon value of 0.1 is optimal. In this case, the agent converges after $\pm$ 600 episodes. and is fully trained from this point onwards.

Similarly, while it takes more episodes, the agents with an epsilon value of 0.01 and 0.3 also end up converging towards a fully trained stable model. Additionally, as seen in figure 6 the finished paths found by the models with a relatively low epsilon value, are optimal or at least close to optimal. The paths found by the agent with a high epsilon score are usually suboptimal (compared to the paths found by the Dijkstra algorithm). This is again explained by the fact that the training uses more random actions, and thus is less likely to find optimal paths.

Compared to the SARSA agent, the expected SARSA agent has a similar performance. The only noticeable differences can be seen when comparing the agents with an epsilon value of 0.3. The expected SARSA agent has a significantly more stable training curve with this epsilon value. This might be explained by the fact that the expected SARSA agent eliminates the variance in the Q-value update rule. This could lead to more room for exploration, and thus a more stable training curve on slightly higher epsilon values. When comparing figure 3 and 7, the heatmap of the expected SARSA agent seems to look visually smoother. Additionally, the agent seems to slightly outperform the SARSA method. For example, when we take a look at state (2,1), the SARSA method takes a suboptimal path of 26 steps, towards the goal. In contrast, the expected SARSA takes an optimal path of 24 steps.

Whilst there are some minor differences between the agent's performances, the current maze is simply not complex enough for there to be a noticeable performance gain. In further evaluation, a bigger and more complex maze is preferred to get a better understanding of the performance difference between the two agents.

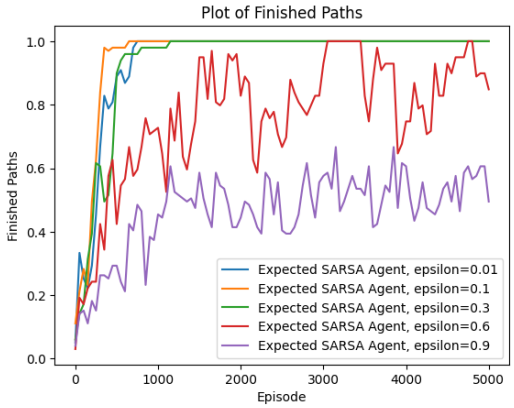

Figure 5: Ratio of completed paths within the time-limit, starting from all starting positions.

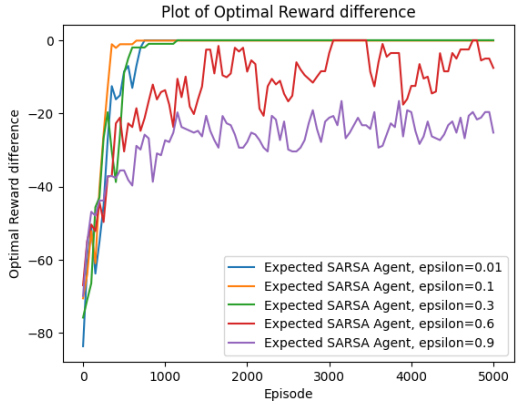

Figure 6: Ratio of optimal paths found, compared to the paths found by Dijkstra's algorithm

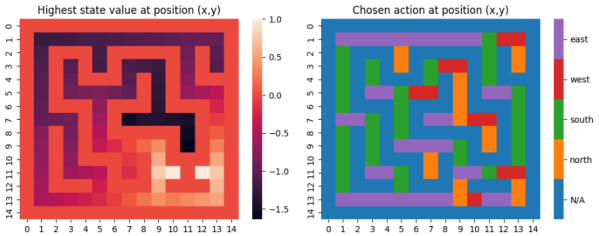

Figure 7: Heatmap of highest Q-value per state (left), map of the chosen action by the agent given a certain state (right)

4.4 Performance Comparison

In this section, we will compare the performances of agents with a special focus on analyzing a different value for the discount factor γ.

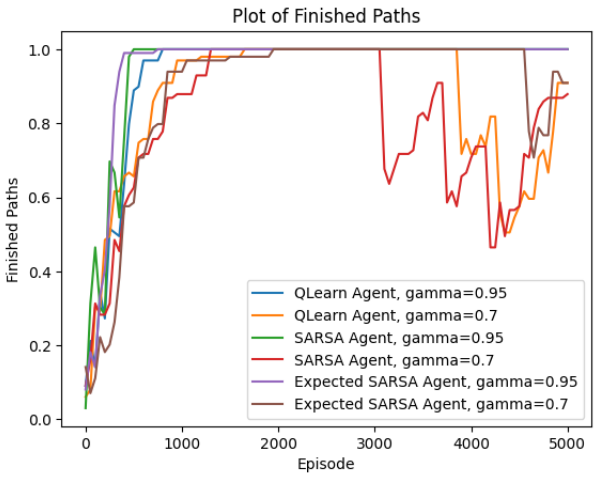

Figure 8: Percentage of starting positions whence the agent can find a path to the final goal within the time constraints

According to figure 8, the clearer distinction in an agent performance is determined more by the choice of its γ parameter than the algorithm itself. In fact, the worst agent with $\gamma = 0.95$ learned faster that the best agent with $\gamma = 0.7$. As we can see for example in 3, the value state update depends, given a fixed model and therefore its parameters α, ϵ, and γ, from the reward of the action and the state action value of the following state, discounted by the factor γ.

Therefore, given 2 valid actions that don't reach the goal and are performed at the same time, their reward is equal. It is now easier to comprehend that a lower discount diminishes the effect of the update that only depends on the value state of the neighboring position, making the learning process slower.

Among the agents with a good γ value, Expected SARSA is the fastest learner, even though normal SARSA reached 100% before. These results should not be taken too seriously, as they depend on the specific conformation of the maze and the randomness of the learning phase. After some testing, we observed that SARSA and Expected SARSA are usually interchangeable in the results, usually being slightly better than the Q Learn Agent. Running times are also the same, since the update laws for the Qtable of the agents have similar computational complexity.

An interesting aspect to notice in figure 8 is the presence of dips for agents with a $\gamma = 0.7$ after they have already found efficient solution, solving the maze in the totality of free positions. These events don't necessarily happen every time the simulation is run, and rarely involves the agents with the higher γ. There are a couple of possible reasons that might explain the presence of dips. First of all, they might in general be due to a poor choice of parameters: in our task the default value for the learning rate is 0.1 and was never changed, and together with ϵ and γ, a poorly chosen value could lead to unexpected behaviour. Additionally, it is important to remind that the environment can't be properly described as a Markov Decision Problem: as the reward is eventually defined by the time (equation 1), the Markov assumption is here missing. This might influence the learning dynamics, producing some problematic like the one in figure 9.

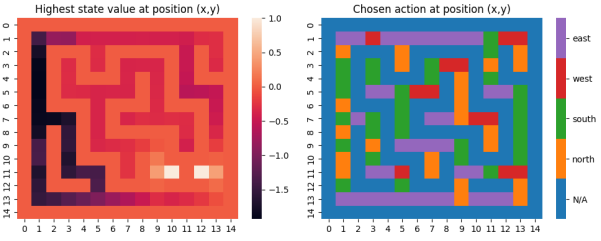

Figure 9: Visualization of an Agent state value estimation(left) and best choice map (right) during a dip phase

The image on the right shows the preferred choice for the agent on a certain position. As we can see, on the left side of the board there are some opposing cells that are creating an endless loop for the agent (going north and south endlessly), which will never reach the goal if spawned in one of these positions.

This process might be triggered from the attempt of the agent to switch from an suboptimal policy to an optimal one. This behaviour remains unexpected, as the theory on RL ([3]) states that a process of learning the optimal policy involves the following updating rule for convergence:

$$\pi \geq \pi' \Leftrightarrow v_\pi \geq v_{\pi'}, \forall s \in S \tag{6}$$

The reference book, on the other hand, relies on the Markov property validity for defining such constraint, so our observations are not opposing to the literature.

5 Conclusion

In this report, three TD methods have been explored for solving the problem of the 'robot in a maze' : SARSA, Q Learning and Expected SARSA. The results show that expected SARSA is the fastest learner among the agents with a good γ value. Usually, $\epsilon = 0.1$ is a good value for granting a good trade-off between exploration and exploitation. After some attempts, SARSA and expected SARSA were found to be interchangeable in the results and usually slightly better than Q Learn Agent. Overall, the results show that a lower discount value reduces the impact of the updates in the learning process, making it slower. However, the lack of the markovian assumption led to some unexpected behaviour in the results. In particular, it was found that some opposing cells were creating endless loops in which the agent will get stuck without reaching the goal. This process might be triggered from the attempt of the agent to switch from a suboptimal policy to an optimal one, although proving it can be challenging.

References

[1] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[2] Harm van Seijen et al. "A theoretical and empirical analysis of Expected Sarsa". In: *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. 2009, pp. 177–184. DOI: 10.1109/ADPRL.2009.4927542.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

Group Members & Contributions

- **Lorenzo Marogna:**
  - Code: SARSA Agent & Evaluation
  - Report: SARSA Agent evaluation 4.1, performance comparison4.4
- **Justin Nguyen:**
  - Code: Expected SARSA Agent & Evaluation
  - Report: Model Selection: Expected SARSA Agent 3.3, Expected SARSA Agent evaluation 4.3
- **Tomás Tavares:**
  - Code: Q Learning Agent & Evaluation
  - Report: Model Selection : SARSA & Q Learning 3.1 3.2, Q Learning Agent evaluation 4.2
- **Minna Vainikainen:**
  - Report: Abstract, Introduction1, Data section 2, conclusion5