Lab Assignment 3 Reinforcement Learning

Group 44

Jan Bryczkowski

Wiktor Grzybko

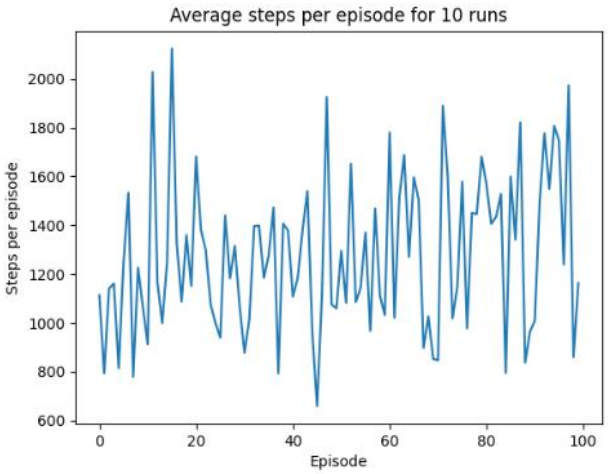
Rafał Owczarski

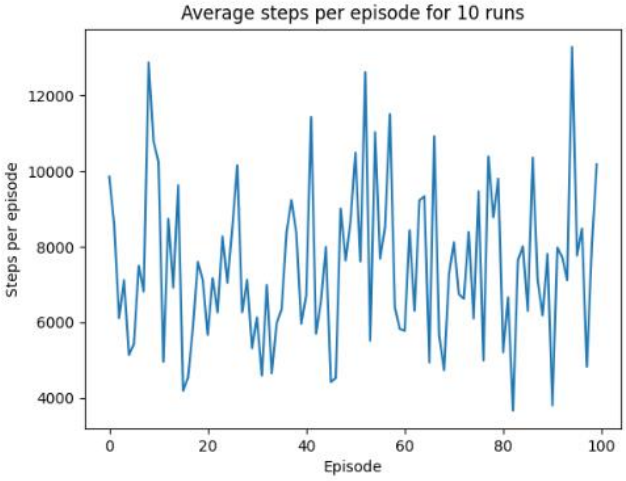
Zofia Rogacka-Trojak

1. To prevent our agent from consistently selecting the same action, we use the np.random.choice library function in our implementation of get_best_action(). This function selects random action with the highest value (from the list obtained from QLearn) to prevent selecting the same one every time when the agent has learned nothing yet. This ensures that our robot does not move in the direction of an obstacle.

2. In our cycle, we first obtain and save the current state and the possible actions, namely the directions in which we can move from the current state, without bumping into an obstacle. We select a new action using the "get_egreedy_action" method, which results in either the best direction or a randomly chosen one. Then, we calculate the state in which we end up after taking that direction. Then we get a reward based on the executed action. Next, we update our Q based on the states, possible actions, taken action, alpha, gamma, and reward. However, this update does not impact our method at this step as the function has not yet been implemented. It will be used in following exercises. Finally we check whether we have reached the goal. If we have, we reset our robot.

4. On both plots, it seems that our agent is not learning. The top plot shows the averages on the toy maze, while the right plot shows the averages on the easy maze. This lack of learning is indicated by the absence of a downward trend, which would suggest that our model is considering previous results and gradually taking fewer steps to reach the goal. Additionally, we can observe high variation and alternation in our results.



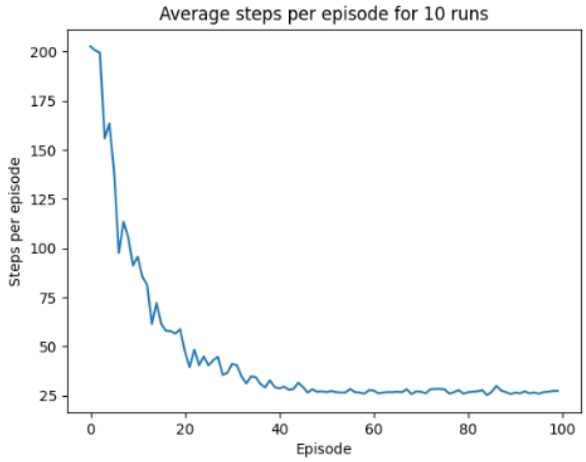


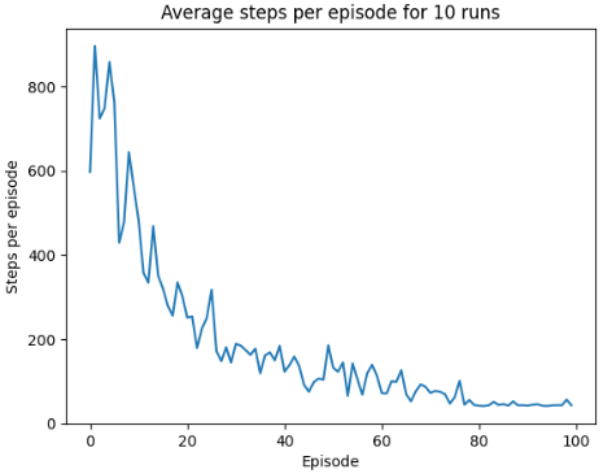
5. Our implementation of Q-Learning is based on the formula provided in the assignment. This formula is used to update the Q-value of the current state-action pair. We calculate the new Q value using the reward received, the Q value of the next state, and the learning rate (alpha) and discount factor (gamma) values. Once we have calculated the new Q value, we set it for our agent.

$$Q(s,a)_{new} = Q(s,a)_{old} + \alpha(r + \gamma Q(s', a_{max}) - Q(s,a)_{old}),$$
where $Q(s', a_{max})$ is the Q value of the best action so far in state s'.

6. On both the toy (upper) and easy (lower) mazes plots, we can observe that the values at the beginning are relatively high and then they are followed by a rapid drop (episodes 10-40). This is due to the fact that agent does not have any previous data to learn from and it selects actions randomly. Consequently, the first result is usually far from the optimal path. However, as the agent starts to learn from previous data, it improves its performance, and the subsequent results are closer to the optimal path. After the initial drop, the agent's performance usually stabilizes, and there are no significant changes in the results. This indicates that the agent has learned enough about the environment to make informed

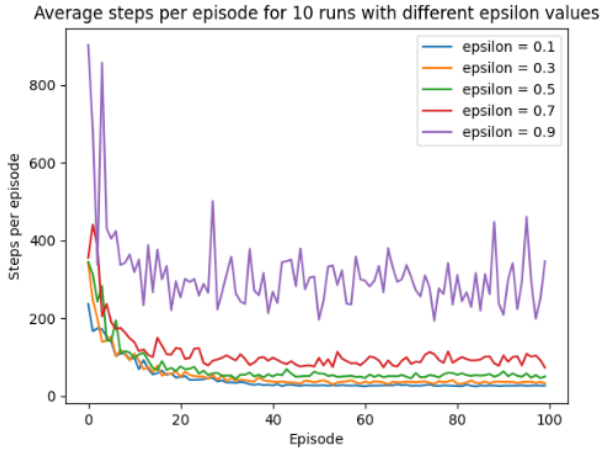
decisions and is following the optimal path. For toy maze the stabilization occurs earlier (40th episode onward), while for easy maze it can be noticed around 80th episode.

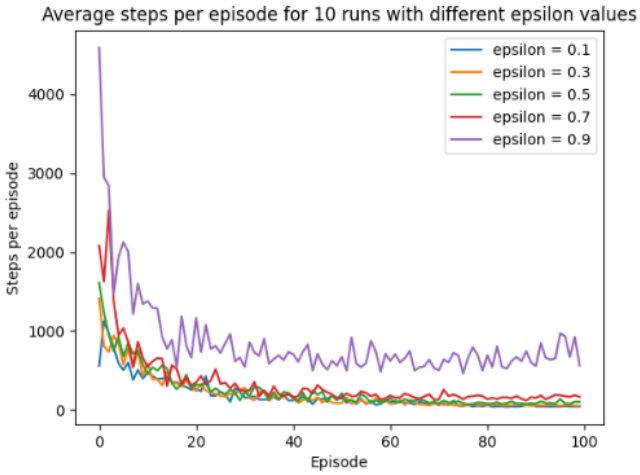




7. Below there are plots (upper– toy, bottom– easy) displaying the impact of four different values of ε {0.1, 0.3, 0.5, 0.7, 0.9} on the performance of the algorithm for the easy maze. The first noticeable difference is between the value of 0.9 and the other values. It is undeniably too high for our solution and results in suboptimal performance (the agent takes too many random actions). This suggests that the exploration rate of our model is too high and that we need to balance it more with the exploitation rate to obtain optimal solutions. Furthermore, we can also observe a slight difference with the value of 0.7 compared to the other values. It is important to note that values between 0.1 and 0.5 appear to be the best for our implementation, as they yield the most optimal solutions the only difference being the speed

of convergence and the value they converge to, with the epsilon value of 0.1 giving the best performance both convergent and optimal value wise.

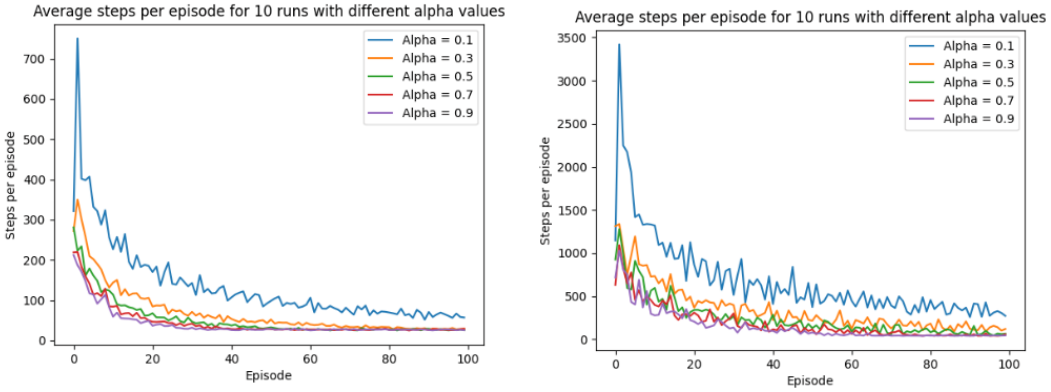




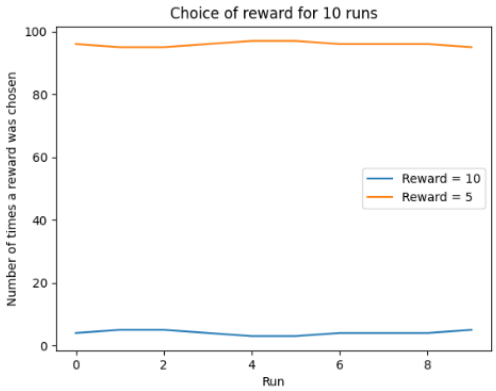
8. The value of ϵ plays a critical role in the algorithm's performance. It is responsible for two primary trade-offs that must be balanced: exploration vs. exploitation and convergence speed vs. optimal solution. A high value in the first one encourages the agent to explore more actions and environments, which may help find new and potentially better solutions, while a low value encourages the agent to exploit known actions and get the best out of them. Therefore, selecting an appropriate value of ϵ is critical to achieving the optimal balance between exploration and exploitation and convergence speed vs. optimal solution.

9. The learning rate has a significant impact on the algorithm, especially with regards to the exploration/exploitation balance. A high learning rate can favour exploration, as it may cause the algorithm to update the value function or policy based on the most recent rewards. A low

learning rate, on the other hand, can favour exploitation, as it may cause the algorithm to place more weight on past experiences and not rely too heavily on the most recent rewards. In the case of our problem, exploration is more beneficial than exploitation, as can be seen in the plot below. A learning rate of 0.1 results in the worst solutions, while a rate of 0.9 results in the best solutions from the beginning. Values between 0.3 and 0.7 are located between the two extremes, and are slightly closer to the better solutions. Therefore, this interval should be further explored to yield the best solution for the algorithm.



10. From the plot below, we can see that the reward at (9,0) is visited more frequently, usually around 10 times as much as the reward at (9,9). This phenomenon is due to the fact that although the reward smaller it is much closer to the starting point and with epsilon value being 0.1 it is discovered much quicker and with low exploration rate and learning rate of 0.7 the knowledge of the second point with higher reward is not retained an thus not used as often.

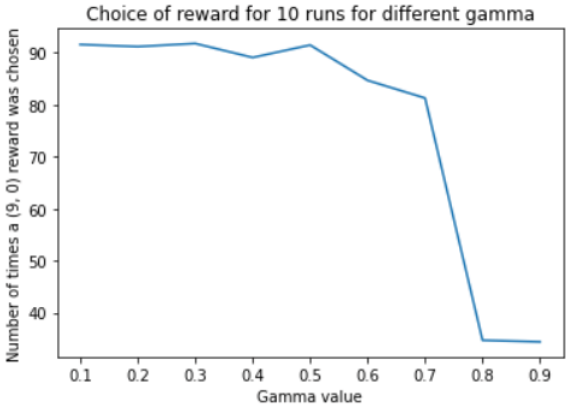


11. One way to mitigate the problem of the agent prioritizing the smaller reward and not fully exploring the environment is to dynamically adjust the value of epsilon (ϵ) during training

episodes. By modifying the epsilon value at each episode, we can control the agent's exploration behavior throughout the training process therefore adjusting how much we want the robot to explore new solutions as we progress our training. With that we encourage it to find the higher reward by setting initial epsilon to 1, this makes the robot able to discover further placed point giving the better reward, as we progress we want to use this knowledge rather than explore more so we decrease the epsilon linearly to exploit the knowledge of the second point. On the graphs one can see that during the training process at first (episodes 0 to 30) reward 5 is chosen more frequently, as it is closer and more likely to be discovered randomly, but as we progress and decrease epsilon further the reward 10 is starting to be chosen more often, which means the agent converges to the better reward. The code we implemented is when dynamically changing epsilon:

```
new_action = selection.get_egreedy_action(robot, maze, learn,
    ((num_of_episodes - episode) / num_of_episodes) * epsilon)
```

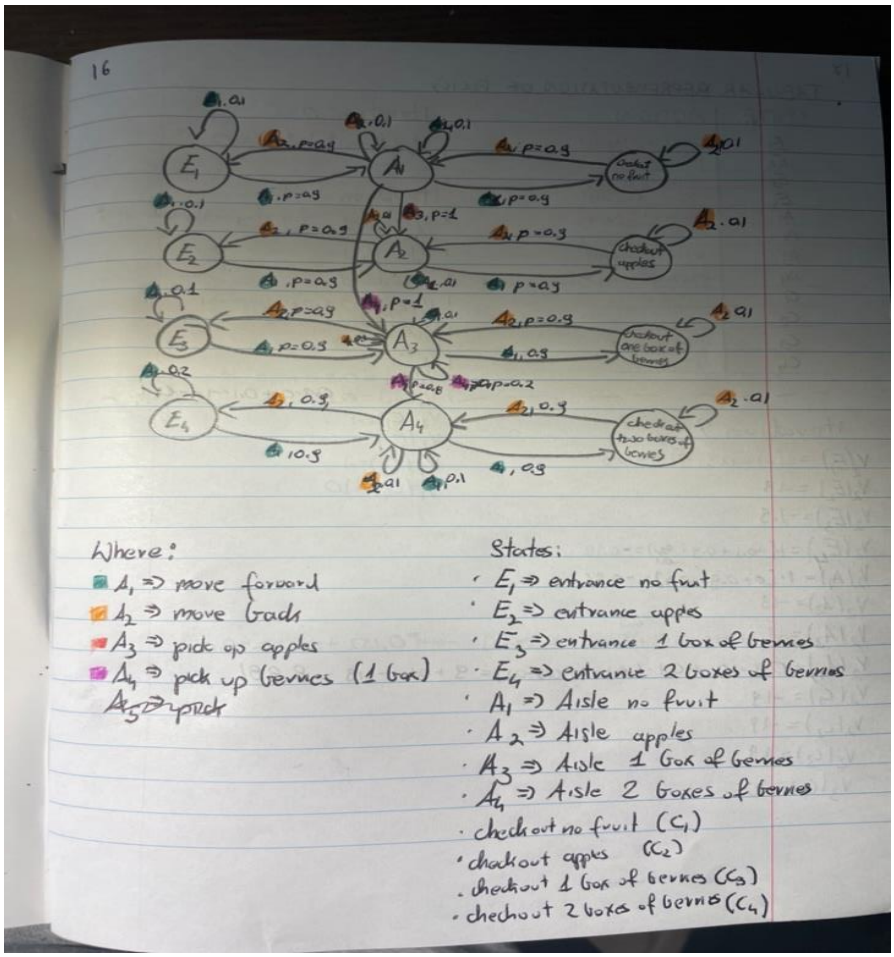
12. It is indeed possible to find a value of gamma for which our agent prioritizes the smaller reward over the bigger one. By selecting a gamma value that prioritizes short-term rewards over long-term ones, the agent will be more likely to approach the smaller reward at (9,0) instead of the reward at (9,9). This means that the agent will be willing to sacrifice the larger reward for the immediate satisfaction of the smaller one. For our problem, this value falls within the range of [0.1, 0.7], as shown in the plot below.



13. One possible downside of having an algorithm that is greedy with regard to its action selection is that it may lead to suboptimal outcomes in the long term or even get stuck in a local optimum. A greedy algorithm might focus on short-term gains and ignore long-term consequences, which can lead to a failure to explore new options that may lead to better outcomes.

14. Reward functions can cause problems in society when their purpose is ill-defined and thus does not fit with the values of society as a whole. For example, if a reward function directs healthcare AI to prioritize cost savings over patient outcomes, this can lead to unethical behavior that can cause a problem for the people affected.

15. It is very unlikely that these problems can be overcome for all reward functions, as there are ethical dilemmas that cannot be resolved without negatively affecting some issues. For example, when applying the reward function to the decision-making process for autonomous vehicles, what decision should be taken about the problem of whether to prioritize the safety of its passengers over the safety of other road users? However, we can partially reduce the impact of these issues by involving stakeholders in the design of the reward function and ensuring that it is in line with the goals of society as a whole.



17.

We start with writing down the policy	Now we start IPE:		
where: E => entrace, A => Aisle, C => Checkout (see graph notation)	**Iteration 0** (initialization)	**Iteration 1**	**Iteration 2**
π(E1) = pickup berries	v0(E1) = 0	v1(E1) = 1 [−1 + 0.9 · 0] = −1	v1(E1) = 1 [−1 + 0.9 · -1] = −1.9
π(E2) = pickup berries**	v0(E2) = 0	v1(E2) = 1 [−1 + 0.9 · 0] = −1*	v2(E2) = 1 [−1 + 0.9 · -1] = −1.9*
π(E3) = pickup berries	v0(E3) = 0	v1(E3) = 1 [−1 + 0.9 · 0] = −1	v2(E3) = 1 [−1 + 0.9 · -1] = −1.9
π(E4) = move forward	v0(E4) = 0	v1(E4) = 0.9[0 + 0.9 · 0] + 0.1[-1 + 0.9 · 0] = -0.1	v2(E4) = 0.9[0 + 0.9 · -0.1] + 0.1[-1 + 0.9 · -0.1] = -0.1
π(A1) = pickup berries	v0(A1) = 0	v1(A1) = 1[0 + 0.9 · 0] = 0	v2(A1) = 1 [0 + 0.9 · -0.4] = −0.36
π(A2) = pickup berries**	v0(A2) = 0	v1(A2) = 1 [−1 + 0.9 · 0] = −1*	v2(A2) = 1 [−1 + 0.9 · -1] = −1.9*
π(A3) = pickup berries	v0(A3) = 0	v1(A3) = 0.8[0 + 0.9 · 0] + 0.2[-2 + 0.9 · 0] = -0.4	v2(A3) = 0.8[0 + 0.9 · -0.1] + 0.2[-2 + 0.9 · -0.4] = -0.544
π(A4) = move forward	v0(A4) = 0	v1(A4) = 0.9[0 + 0.9 · 0] + 0.1[-1 + 0.9 · 0] = -0.1	v2(A4) = 0.9[0 + 0.9 · 10] + 0.1[-1 + 0.9 · -0.1] = 7,991
π(C1) = pickup berries	v0(C1) = 0	v1(C1) = 1 [−1 + 0.9 · 0] = −1	v2(C1) = 1 [−1 + 0.9 · -1] = −1.9
π(C2) = pickup berries**	v0(C2) = 0	v1(C2) = 1 [−1 + 0.9 · 0] = −1*	v2(C2) = 1 [−1 + 0.9 · -1] = −1.9*
π(C3) = pickup berries	v0(C3) = 0	v1(C3) = 1 [−1 + 0.9 · 0] = −1	v2(C3) = 1 [−1 + 0.9 · -1] = −1.9
π(C4) = pay*	v0(C4) = 0	v1(C4) = 1 [10 + 0.9 · 0] = 10	v2(C4) = 1 [10 + 0.9 · 0] = 10
* action not included on the graph (results on specific reward)		* according to the description of the task we cannot pick up berries when we already hold apples (so the result state of the action is not clear)	
**according to the description of the task we cannot pick up berries when we already hold apples (so the result state of the action is not clear)			

18. In order to determine the optimal policy for the robot we could apply SARSA algorithm. The algorithm will explore different actions and states of the robot, updating the estimated values of the Q-function (which gives the expected total reward for each state-action pair), and using these values to choose the action that maximizes the expected total reward at each state. After convergence, we have Qπ and can use policy improvement. Finally based on the determined policy we can determine whether the robot should get a bag of apples, one box of berries, or try to pick up two boxes of berries.

Sources:

- Slides from Reinforcement Learning lectures