Introduction to Robotics - Mini Project 3

Duarte Meneses Maria Duarte Vasco Silva

1 Introduction

We started by trying to understand what we were working with. For that we studied the references and tried to connect everything together.

After understanding the topics we searched and studied possible packages to solve our problem. In the end, we thought it was best to implement our own solution. The code for the package we developed for this project is on GitHub.

We have extra material regarding the experiments we did for this project in a Google Drive Folder with videos from the execution of the MDP, discretized maps and plots from our work with Reinforcement Learning.

2 Background

2.1 Markov Decision Processes

Sequential processes describe the evolution of a stochastic system over time. If the evolution of the system depends on the actions of an agent and the agent as no uncertainty when trying to determine the state of the system then we have Markov Decision Processes (MDPs).

To define an MDP we need: **States** - the possible configurations of the system. They are the main information each decision depends on; **Actions** - the actions the agent can take; **Dynamics** - how the actions of the agent make the system evolve; **Rewards** - they describe the "quality" of each state and ultimately they describe the goal of the agent.

For the remainder of this report we represent the set of all states as $\mathcal{X}$, the state at time t as x_t, the set of all actions as $\mathcal{U}$ and the action at time t as u_t. The dynamics are represented as probabilities that describe the probability of transitioning to a state x_{t+1} given the present state x_t and the action executed in that state u_t, $P(X_{t+1} = x_{t+1}|X_t = x_t, U_t = u_t)$. For

simplicity we will use $P(x'|x,u)$ to represent the same. The rewards depend on the state and action and we will use $r(x,u)$ to represent the reward given to the agent when it executes action u in state x.

MDPs satisfy the Markov Property, that is, the state at time $t+1$ depends only one the state of the system and the action executed at time t, formally:

$$\mathbb{P}(X_{t+1} = x_{t+1}|X_{0:t} = x_{0:t}, U_{0:t} = u_{0:t}) = \mathbb{P}(X_{t+1} = x_{t+1}|X_t = x_t, U_t = u_t) \tag{1}$$

,where $x_{0:t}$ represents the whole history of states and $u_{0:t}$ the whole history of actions.

The objective of the agent is to maximize the discounted cumulative reward in a given time horizon, T:

$$R_T = \sum_{t=0}^{T} \gamma^t r_t \tag{2}$$

,where γ is the discount factor and varies between 0 and 1 ($0 < \gamma \leq 1$).

The agent acts in the environment by following a policy π that maps states into distributions over actions.

To maximize rewards the agent follows the optimal policy, π^*. This policy has a higher expected reward than any other policy. The expected reward for a policy depends on the state x we start in and the considered time horizon and is defined as:

$$R_T^\pi(x) = \mathbb{E}_\pi\left[\sum_{t=0}^{T} \gamma^t r(x_t, u_t)\right] \tag{3}$$

We define now the value function V_T^π that maps each state into a value. This value is the expected discounted cumulative reward of starting to execute policy π in state x for T steps ($V_T^\pi(x) = R_T^\pi(x)$). We actually will solve MDPs with infinite horizon, so we will omit T from here on out to make the equations simpler.

In addition to the value function we can define a Q-function which maps pairs of states and actions to the expected discounted cumulative reward of executing a certain action in a certain state. Both these functions satisfy the Bellman equations:

$$V^{\pi}(x) = \sum_{u} \pi(u|x) \left[r(x,u) + \gamma \sum_{x'} P(x'|x,u) V^{\pi}(x') \right] \tag{4}$$

$$Q^{\pi}(x,u) = r(x,u) + \gamma \sum_{x'} P(x'|x,u) \sum_{u'} \pi(u'|x') Q^{\pi}(x',u') \tag{5}$$

In the case of the optimal policy π^*, we call its value function V^* and its Q-function Q^* and, in that particular case, we have:

$$V^*(x) = \max_{u} \left[r(x,u) + \gamma \sum_{x'} P(x'|x,u) V^*(x') \right] \tag{6}$$

$$Q^*(x,u) = r(x,u) + \gamma \sum_{x'} P(x'|x,u) \max_{u'} Q^*(x',u') \tag{7}$$

Because Equation 6 holds only for π^*, we can use it to solve the MDP. Using this recursion to solve the MDP is called Value Iteration. It is easier to implement the algorithm if we think about the problem in vector notation: $\boldsymbol{r_u}$ is a matrix with size $|\mathcal{X}| \times 1$ in which $[r_u]_x = r(x,u)$; $\boldsymbol{P_u}$ is a matrix with size $|\mathcal{X}| \times |\mathcal{X}|$ in which $[P_u]_{xx'} = P(x'|x,u)$; $\boldsymbol{V}$ is a matrix with size $|\mathcal{X}| \times 1$ in which $[V]_x = V(x)$.

Now our recursion becomes:

$$V = \max_{u} \left[r_u + \gamma P_u V \right] \tag{8}$$

Algorithm 1 shows the pseudo code for the Value Iteration algorithm which gives us the optimal policy.

2.2 Reinforcement Learning

The idea of Reinforcement Learning is to provide the agent with the appropriate tools to learn what it should do. The agent is deployed in the environment and every time it does something that we find desirable we give it a positive reinforcement and every time the agent does something we don't want it to do, we give it a negative reinforcement.

Algorithm 1 Value Iteration

Require: MDP; tolerance $\varepsilon > 0$

initialize $t = 0$, $V^{(0)} \equiv 0$

repeat

  $V^{(t+1)} \leftarrow \max_u \left[r_u + \gamma P_u V^{(t)} \right]$

  $t \leftarrow t + 1$

until $\left|\left| V^{(t-1)} - V^{(t)} \right|\right| < \varepsilon$

compute π^* from $V^{(t)}$

return π^*

There are three types of Reinforcement Learning, Model Based, Policy Based and Value Based. We are going to talk about only Value Based methods. In this learning methods the agent maintains an estimation of a Q-function, $\hat{Q}$. We are going to talk about two value based algorithms, Q-learning and SARSA.

2.2.1 Q-learning

Q-learning is an off-policy algorithm. This means that it learns the value of one policy while following another. In particular it learns the optimal Q-function. It relies on the following update:

$$\hat{Q}_{t+1}(x,u) \leftarrow \hat{Q}_t(x,u) + \alpha_t \left[r(x,u) + \gamma \max_{u'} \hat{Q}_t(x',u') - \hat{Q}_t(x,u) \right] \tag{9}$$

This update is correcting our previous estimate with an error. The first part of the error is actually very similar to Equation 7 (except for the probabilities) because we are trying to pull our estimation closer to the real value. The probabilities present in Equation 7 will appear naturally in our estimation as states whose transition probabilities are higher, will be updated more often. It all comes in place as the agent interacts with the environment and the following holds: $\lim_{t\to\infty} \hat{Q}_t(x,u) = Q^*(x,u)$. This algorithm is off-policy because of the $\max_{u'}$ in the update. That specific detail implies that the Q-learner always updates its estimate under the assumption that it is following the optimal policy even though it might not be. This also

means that the Q-learner can make an update as soon as it executes one action.

2.2.2 SARSA

SARSA is an on-policy algorithm. This means that it learns the value of the policy it is following. It relies on the following update:

$$\hat{Q}_{t+1}(x,u) \leftarrow \hat{Q}_t(x,u) + \alpha_t \Big[r(x,u) + \gamma \hat{Q}_t(x',u') - \hat{Q}_t(x,u) \Big] \tag{10}$$

The reasons why this update works are very similar to the reasons the Q-learning update works. The only difference is that we don't have the $\max_{u'}$ which explains why this algorithm is on policy. Now the agent can't do an update as soon as it executes an action, it as to wait until it executes the next one to make the update.

2.2.3 Exploration VS Exploitation

The Q-learning and SARSA agents have to choose actions to execute. Ideally they want to choose the ones with highest expected rewards, but in the beginning they don't know which are those. Because of this we have to come up with a balance between Exploration (randomly selecting actions) and Exploitation (choose the action with highest expected reward).

One of the ways to do this is called ε-greedy. There is a parameter ε that can vary with time and in each step, with probability ε the agent explores and with probability $1 - \varepsilon$ the agent exploits. Usually we want ε to decrease with time because we want to start exploring a lot to learn an accurate estimation, but as time goes by, we can start exploiting more and more.

Now that we know how the agents select actions we can attempt to compare Q-learning and SARSA. Because Q-learning operates under the assumption that the agent executes always the best action, it does not account for the fact that sometimes the agent will do exploratory actions, which means that the results of executing exploratory actions can result in very bad rewards. SARSA, on the other hand, takes into account the exploratory moves. Usually this means that the SARSA agent is more conservative, taking longer paths to reach the goal,

but, most of the times, collecting higher rewards than the Q-learning agent as the exploratory moves will not be so punishing.

3 Implementation

For this project we decided to implement our own solution. The package is on GitHub. We like to call our package **markov_discretization** because it essentially receives a map, discretizes it and runs MDPs on the resulting cells. It also is able to simulate Q-learning and SARSA.

The node starts by asking a map_server for a map. Then it discretizes the map into cells based on the **cell_size** parameter. The resulting MDP has a set of states that correspond to the resulting cells, each one identified by the real world coordinates of its center point. A real world space will only be included as a cell in the MDP if it is completely free. There are four possible actions, **Up**, **Down**, **Right** and **Left**. The transition probabilities are dependent on a parameter called **succ** (success), for simplicity we will refer to it as $\boldsymbol{\beta}$. As an example, if we execute the action Up, the robot will go up with probability β and go Down, Right or Left with probability $\frac{1-\beta}{3}$. If one of the actions is impossible to do because there are no adjacent cells in that direction than its probability will account for the probability of staying in the same cell. The rewards can also be adjusted. If the user does not specify anything about the rewards of a specific state then that state will have a **normal** reward (-1 by default). The user can specify one state as a goal in which case the state will have a **goal** reward (100 by default). And finally the user can specify undesired states that have a **deadly** reward (-100 by default). The user can also adjust the γ used in the MDP (0.9 by default) and the α used in Reinforcement Learning (0.3 by default). The Reinforcement Learning uses ε-greedy for action selection, it is also possible to adjust how ε evolves with time.

To execute a policy our packages sends a goal to move_base, corresponding to the center point of a cell, and waits until the action is completed. As soon as the goal is reached another action is chosen based on the policy and another goal is sent to move_base.

Besides solving MDPs and simulating Reinforcement Learning our package also produces images that represent the MDP and its solution. The size of the images can be controlled by

the parameter **scale**. It is possible to print the discretized map, the discretized map with colors symbolizing the rewards of each cell (green is goal, blue is normal and red is deadly), and the color coded discretized map with arrows indicating the action(s) prescribed by a policy. All the map images in this report were obtained with our package.

4 Results

4.1 Markov Decision Processes

We know that, in general, an agent following an MDP will try to reach goal states but we wanted to see the influence that the β and γ parameters had on the optimal policy. For that we thought the rewards used were a good idea because we get to see what the agent does when the goal state is very close to very bad states. Those rewards can be seen in the maps showing the optimal policies and they are color coded as described in Section 3 (Figure 1, for example).

With those rewards, the β, the γ, the actions (Up, Down, Right and Left) and with the knowledge that non black squares constitute one state each, we have the model completely defined.

First, we tried to fix γ and see the influences of β in the optimal policies. β can be seen as the confidence the agent has in its actions. The differences are well captured in Figure 1 and Figure 2. Because in 1 the agent is very confident in the outcomes of its actions, it passes very close to very bad states because it knows that the probabilities of ending up in those states are very small. On the other hand, in 2 the agent knows it is very probable that its actions will not have the desired effect so it tries to be as far from bad states as possible. In states next to deadly ones, the confident agent goes towards the goal while the not so confident agent invests some time in getting away from danger.

We then tried to fix β and see the influences of γ in the optimal policies. Higher γ means that the agent will put more weight on long term goals while lower γ means that the agent will put more weight on short term goals. This preference between short and long term goals can bee seen in Figure 3 and Figure 4. In 3 the agent is focused on long term goals and we

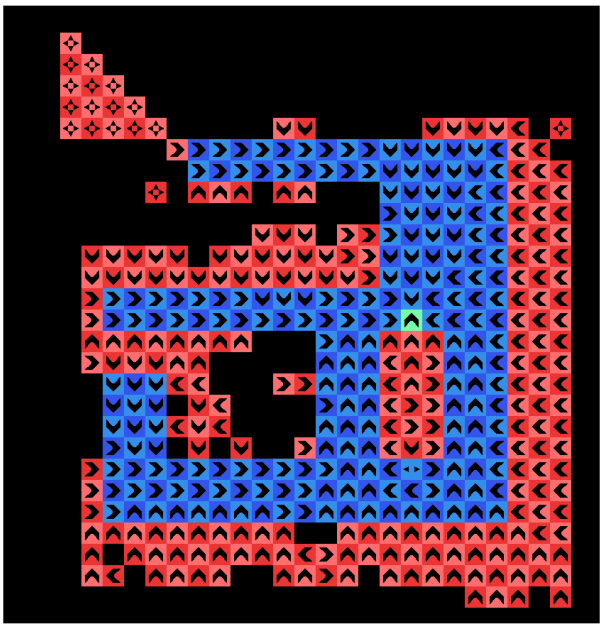

Figure 1: π^* for $\beta = 0.90$, $\gamma = 0.90$

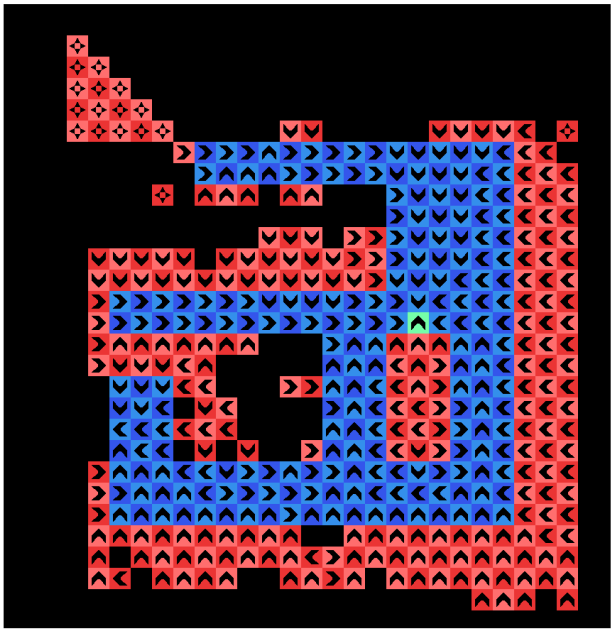

Figure 2: π^* for $\beta = 0.60$, $\gamma = 0.90$

can see that there is kind of a gradient in the map with a minimum in the goal because no matter how far the agent is, it always wants to get to that high reward state. In 4 however, in states distant from the goal, the agent is happy with not falling into a deadly state because it is more focused on short term goals, that is why in states distant from the goal the optimal policy prescribes so many actions. Figure 5 shows the paths followed by the robot in several runs, starting from two different positions. The true traversed paths can be seen in RVIZ in the videos in our Drive. There are more runs with different parameters.

4.2 Reinforcement Learning

In reinforcement learning we focused more on the trade off between exploration and exploitation. We used ε-greedy. As discussed in 2.2 we would like to have ε decreasing over time. We tried several functions for this, inclusively we tried constant ε just to see the results but the best results were with $\boldsymbol{\varepsilon(t) = e^{-\frac{2t}{10^6}}}$ as it decreases in a perfect rate while other decreasing functions we tried either started exploiting too soon or too late. The error is defined as $E(\hat{Q}) = \frac{||Q^* - \hat{Q}||}{||Q^*||}$ and it can be seen in function of t in Figure 6. It is also worth noting that

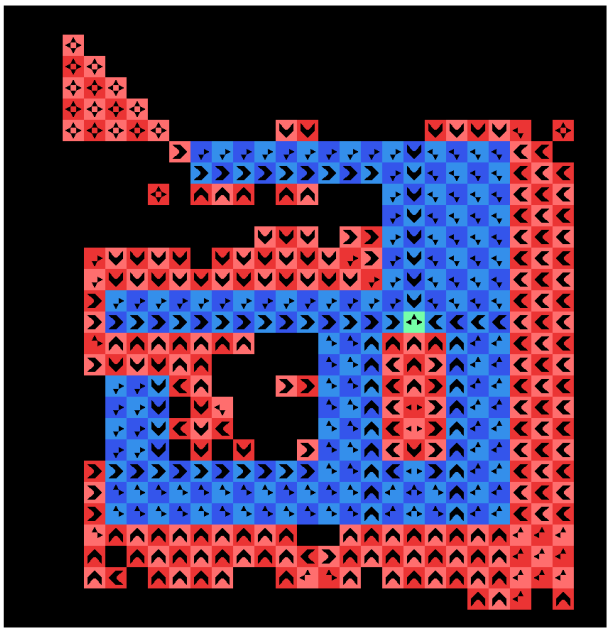

Figure 3: π^* for $\beta = 1.00$, $\gamma = 0.90$

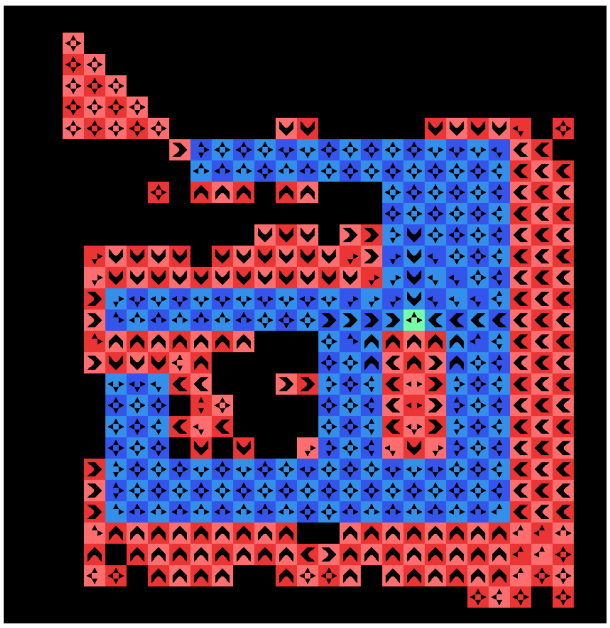

Figure 4: π^* for $\beta = 1.00$, $\gamma = 0.10$

SARSA only starts to converge to Q^* once the agent starts exploiting more. This happens because SARSA is on policy. We wanted to experiment with different α values because it is also an important parameter of SARSA and Q-learning. The results showed are for $\alpha = 0.3$. Ideally, we would like α to decrease over time because after a big number of steps we don't want the agent to override previously attained knowledge. Also α converging to zero is a necessary condition for the algorithm to converge under stochastic environments (which is our case). Extra plots for different ε over time can be seen in our Drive. The name of each image contains the function used.

5 Conclusions

In this project everything went well. The fact that we developed our own package really helped us understand the problem at hands. We were able to implement everything that was required of us and then we were able to try the influence of the parameters. The only thing we would have liked to explore more was the α in Reinforcement Learning.

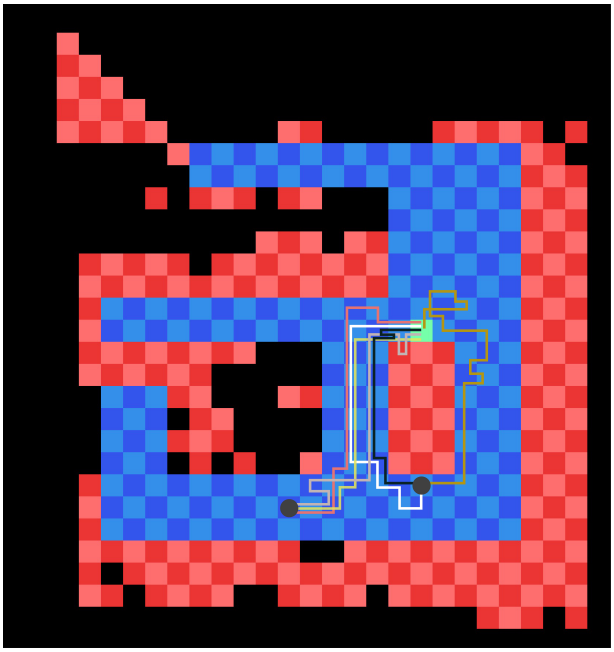

Figure 5: Paths with $\beta = 0.90$, $\gamma = 0.90$

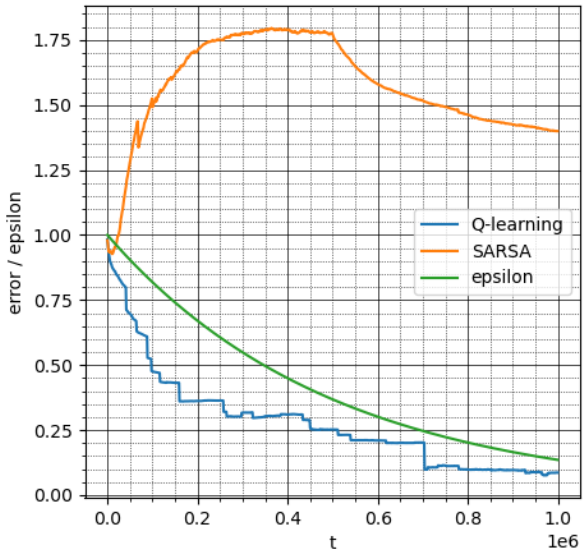

Figure 6: Error for estimated Q-functions over time and ε over time

References

[1] *ROS Robot Programming (English)*. ROBOTIS, Dec. 2017. ISBN: 9791196230715.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.

[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.