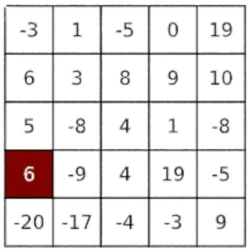
LIÈGE université
School of Engineering

INFO8003-1: Optimal Decision Making for Complex Problems



-3	1	-5	0	19
6	3	8	9	10
5	-8	4	1	-8
6	-9	4	19	-5
-20	-17	-4	-3	9

Assignment 1: Reinforcement Learning in a Discrete Domain

Staff :
ERNST Damien, *Teacher*
LOUETTE Arthur, *Teaching Assistant*
MIFTARI Bardhyl, *Teaching Assitant*

Group :
LAMBERMONT Romain
LOUIS Arthur

March 4, 2024

Contents

List of Figures

List of Tables

1 Implementation of the Domain

During this project, the domain is simply represented by a matrix of the rewards for the states visited. This matrix can be seen in the Figure on the front page. The behavior of the domain, either deterministic or stochastic is chosen by setting a boolean value to `False` or `True` accordingly. The rewards are stored inside a `numpy` array called `REWARDS`. The agent can take in each state one of the four actions `down up right left` stored in another array called `ALLOWED_ACTIONS` representing the values assessed to these actions. The dynamics of the domain are simply computed by computing the next state's tuple following:

$$\text{next\_state} = ((\min(\max(\text{state\_x} + \text{action\_x}, 0), 4), \min(\max(\text{state\_y} + \text{action\_y}, 0), 4)))$$

For the stochastic domain, at each step, the agent has a 50% probability of jumping directly to `(0,0)`. For both the deterministic and stochastic domains, we chose a straightforward rule-based policy. The agent simply chooses one of the four actions completely at random for each of the ten steps of the simulation. Starting from the position `(3,0)` we get as an example the following trajectories:

```
Deterministic domain:

((3, 0), (0, -1), 6, (3, 0))
((3, 0), (0, 1), -9, (3, 1))
((3, 1), (0, 1), 4, (3, 2))
((3, 2), (0, 1), 19, (3, 3))
((3, 3), (1, 0), -3, (4, 3))
((4, 3), (0, -1), -4, (4, 2))
((4, 2), (0, -1), -17, (4, 1))
((4, 1), (0, 1), -4, (4, 2))
((4, 2), (-1, 0), 4, (3, 2))
((3, 2), (1, 0), -4, (4, 2))
```

```
Stochastic domain:

((3, 0), (0, 1), -9, (3, 1))
((3, 1), (1, 0), -3, (0, 0))
((0, 0), (0, 1), 1, (0, 1))
((0, 1), (-1, 0), -3, (0, 0))
((0, 0), (-1, 0), -3, (0, 0))
((0, 0), (0, -1), -3, (0, 0))
((0, 0), (1, 0), -3, (0, 0))
((0, 0), (1, 0), 6, (1, 0))
((1, 0), (-1, 0), -3, (0, 0))
((0, 0), (0, 1), 1, (0, 1))
```

2 Expected Return of a Policy

In this section, we will compute the expected return of the rule based policy mentioned in the Section 1 for both the domains. Firstly, we need to decide for which horizon we need to compute this expected return by correctly choosing N. Indeed, the choice of the horizon N is crucial in computing the expected return of the rule-based policy. The horizon determines the number of time steps or periods over which the policy's decisions will be evaluated. If N is too small, J_N^μ will too far from the real value of J^μ and a too big will make the computations really long.

2.1 Choice of N

In this project, the maximum reward that can be obtained in one step B_r is +19 and the discount facor γ is equal to 0.99. We can thus choose our N by using:

$$\|J_N^\mu - J^\mu\|_\infty \leq \frac{\gamma^N}{1-\gamma} B_r$$

This relation tells us that for increasing values of N the difference between J_N^μ and J^μ will rapidly tend towards 0. Using the Table 1 we can select N equal to 10 000 as a good compromise between the small value of the divergence between J_N^μ and J^μ and the computation time which remains tolerable.

N	$\frac{\gamma^N}{1-\gamma}B_r$
10	1718.33
100	695.46
1000	0.082
10 000	4.27×10^{-41}
100 000	6.28×10^{-434}

Table 1: Upper bound of the divergence between J_N^μ and J^μ depending on the value of N. The value 10 000 seems to be a good compromise between precision and computational expenses

2.2 Computation of the Expected Return

With our correctly chosen value of N, we can now start to compute the expected return of the rule-based policy introduced in Section 1 using the recursive equation to compute the expected return of a stationary policy seen during the course:

$$J_N^\mu = \begin{cases} 0 \text{ if } N = 0 \\ E_{w\sim P_w(\cdot|s,a)}[r(s,\mu(s),w) + \gamma J_{N-1}^\mu(f(s,\mu(s),w))] & \forall N \geq 1, \text{ stochastic behavior} \\ r(s,\mu(s)) + \gamma J_{N-1}^\mu(f(s,\mu(s))) & \forall N \geq 1, \text{ deterministic behavior} \end{cases}$$

In the case of the stochastic behavior, the expected return can be decomposed in two parts with equal probabilities, one concerning the case where the agent succeeds to apply the action where it reaches s' and another where the agent is "teleported" to `(0,0)`:

$$E_{w\sim P_w(\cdot|s,a)}[r(s,\mu(s),w] = 0.5 \times (r(s') + r(0,0))$$

$$E_{w\sim P_w(\cdot|s,a)}[\gamma J_{N-1}^\mu(f(s,\mu(s),w))] = 0.5 \times \gamma \times (J_{N-1}^\mu(s') + J_{N-1}^\mu((0,0)))$$

The mean expected return of the rule-based policy over 10 runs on both the stochastic and deterministic domains can be seen in Figure 1. The standard deviation over those 10 runs can be seen in Figure 2. It can be easily seen that for the stochastic domain, the return seems to spread over all the states as the agent can always jump to `(0,0)` with a 50% probability as opposed to the deterministic domain where the closer to high rewards, the more a state has a greater expected return. When looking at the standard deviations, we can easily see that the deterministic domain has a much wider range of rewards compared to the stochastic which makes sense because the fact that the agent teleports 50% of the time to `(0,0)` obviously reduces the variance of the rewards.

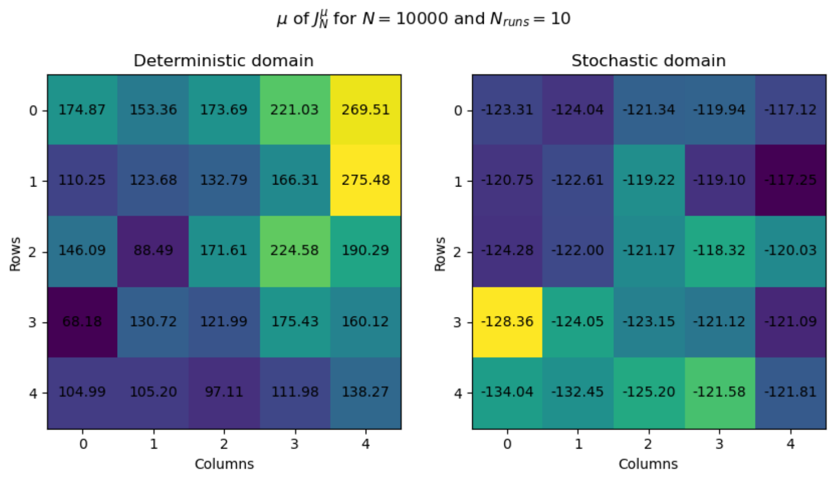


Figure 1: Mean expected return of the rule-based policy over 10 runs

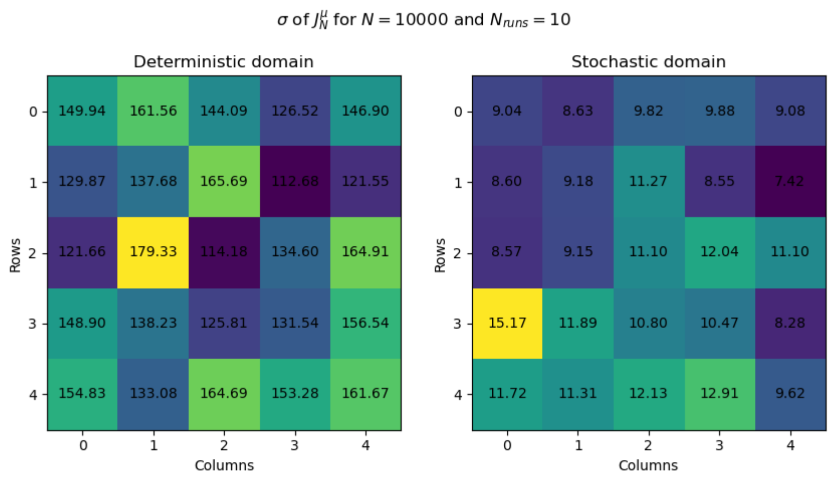


Figure 2: Standard deviation of the expected return of the rule-based policy over 10 runs

3 Optimal Policy

In this part of the project, we will try to find the best policy possible. To do so, we will need to compute the Q_N functions. For that, we should determine the behaviour of an equivalent MDP to our domain with the rewards and transition probabilities following:

$$\begin{aligned} r(s,a) &= E_{w\sim P_w(\cdot|s,a)}[r(f(s,a,w))] \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \\ p(s'|s,a) &= E_{w\sim P_w(\cdot|s,a)}[I_{s'=f(s,a,w)}] \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A} \end{aligned}$$

Once again we can rewrite these for the deterministic in a more simpler way and by dividing the stochastic behavior in two parts with equal probabilities. This gives for the deterministic domain:

$$\begin{aligned} r(s,a) &= r(f(s,a)) \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \\ p(s'|s,a) &= I_{s'=f(s,a)} \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A} \end{aligned}$$

These equations make sense because, in the deterministic domain, the reward always depends on the state reached in consequence to the action taken. As for the transition probability, since the state is reached accordingly to action with a 100% probability, it is normal to only check if s' is a consequence of the action a taken in state s. And for the stochastic domain:

$$\begin{aligned} r(s,a) &= 0.5 \times (r(f(s,a)) + r((0,0))) \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \\ p(s'|s,a) &= 0.5 \times (I_{s'=f(s,a)} + I_{s'=(0,0)}) \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A} \end{aligned}$$

Here we have to keep in mind that in 50% of the steps, the agent will teleport to the state `(0,0)`. The reward equation thus makes the mean between the reward of the state reached in a deterministic way and the one resulting in the teleportation to `(0,0)`. As for the probabilities they are all divided by 2 as in all cases the agent can teleport to `(0,0)` except for the actions `up` and `left` when taken in state `(0,0)` as in this case it stays in `(0,0)` with probability 1.

With all that in mind we can start implementing the Q_N using the recursive formula seen during the course:

$$Q_N(s,a) = \begin{cases} 0 \text{ if } N = 0 \\ r(s,a) + \gamma \times \sum_{s'\in\mathcal{S}} p(s'|s,a) \max_{a'\in\mathcal{A}} Q_{N-1}(s',a') \end{cases}$$

For the deterministic domain, the policy doesn't changes after $N = 7$ and stays the same after. We can then extract the optimal policy μ^* from the MDP and we get the policy located in Table 2. It is clear that the policy tries to reach the state with the most positive reward (upper right corner) and spam its reward by taking the action `up` to stay in the same state. The expected return of μ^* for the deterministic domain can be seen in Figure 3.

For the stochastic domain, the policy stops changing after $N = 10$. We extract μ^* once again and get the policy located in Table 3. The behavior of the optimal policy in this stochastic domain can be interpreted as trying to maximize the reward with a smaller horizon because the chances of being teleported to `(0,0)` are huge and planning something that may happen in 10 steps is improbable the agent would have to "win a coin flip" 10 times in a row to visit the state maximizing the reward. The stochasticity of the domain can be seen as a horizon reducer in the perspective of the agent.

↓	→	→	→	↑
→	→	→	→	↑
↑	→	↑	↑	↑
↑	→	→	↑	↑
↑	→	↑	↑	←

Table 2: Optimal policy μ^* for the deterministic domain

↓	↓	↓	→	↑
→	→	→	→	↑
↑	→	↑	↓	↑
←	→	→	←	←
↑	→	↑	↑	↓

Table 3: Optimal policy μ^* for the stochastic domain

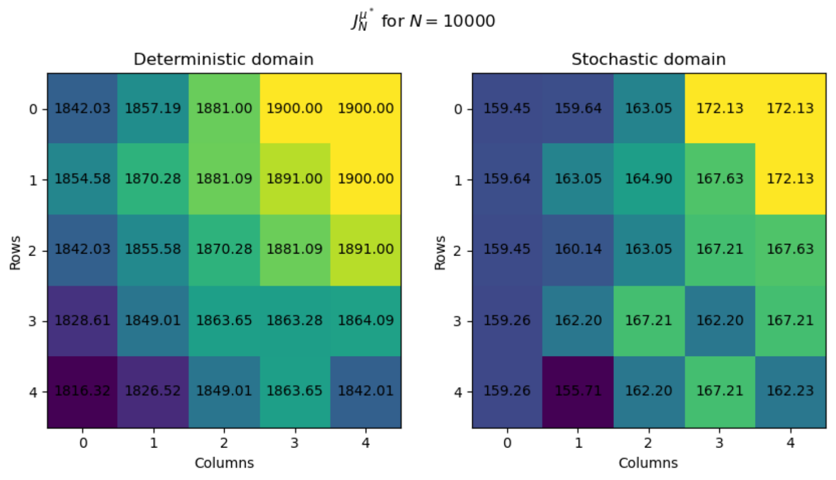


Figure 3: Expected return of the optimal policies for each domain behavior

4 System Identification

In this section, our objective is to approximate the p, r, and Q functions by employing a random trajectory. Subsequently, we calculate the infinite norm between the estimated r, p, and Q matrices and the ground truth, which is reused from the previous sections. This norm yields the maximum difference between two matrices.

To create a trajectory, we used the previous random policy and iterated N times to generate a list of N tuples of type $(\text{State}, \text{Action}, \text{Reward}, \text{Next State})$.

With these tuples, we fill two matrices $\hat{r}$ and $\hat{p}$. To fill the first one, we sum the total reward for a given tuple $(\text{state}, \text{action})$ from the trajectory and then divide by the number of occurrences. For the second one, we create a 3D matrix that allows us to have 25 possible states for a given current state and action. We count each time we reach a next state given the tuple state and action. At the end, we divide the 25 possibilities by the total number of times we encountered the tuple $(\text{state}, \text{action})$.

The next step is to compute the estimated Q-value function, denoted as $\hat{Q}$, using the estimated reward function $\hat{r}(s, a)$ and transition probabilities $\hat{p}(s'|s, a)$. For each state-action pair (s, a), we sum over all possible next states s' the product of the transition probability $\hat{p}(s'|s, a)$ and the sum of the reward $\hat{r}(s, a)$ and the discounted maximum Q-value over all actions at the next state s'.

Unfortunately, we have run out of time and cannot implement this part of the project properly at the moment. We plan to dedicate our efforts to address this aspect in preparation for the next deadline and ensure its successful implementation.

One of the significant challenges we encountered revolved around synthesizing information from the random trajectory. Initially, we attempted to use a dictionary, but the process became cumbersome, especially when dealing with lists stored within the dictionary. After extensive efforts with this approach, we made the decision to start afresh and leverage NumPy matrices for more efficient handling of the information.

Creating the matrix $\hat{r}$ posed a relatively straightforward task, as there is only one value for each pair $(\text{state}, \text{action})$. Thus, a 2D matrix proved to be sufficient.

However, constructing the matrix $\hat{p}$ presented a more intricate challenge. For every pair $(\text{state}, \text{action})$, we needed to represent a probability distribution with one value for each possible state, and these values must sum to 1.