Reinforcement learning - MVA 2017/2018
Homework 1

Rémi Lespinet

1 Dynamic Programming

• Q1: The transition table and the reward table for the graph of the exercice are presented in tables 1 and 2 respectively

	a0			**a1**		
	s0	s1	s2	s0	s1	s2
s0	0.45	0.00	0.55	0.00	0.00	1.00
s1	0.00	0.00	1.00	0.50	0.40	0.10
s2	0.60	0.00	0.40	0.00	0.90	0.10

Table 1 – Representation of the transition table corresponding to the graph

	a0	a1
s0	-0.40	0.00
s1	2.00	0.00
s2	-1.00	-0.50

Table 2 – Representation of the reward table corresponding to the graph

• Q2: The Policy evaluation and Value iteration algorithm are implemented in the file `lib_tp1_ex1 .py`. The figure 1 represents the distance to the optimal value $\|v^k - v^*\|$ as a function of the number of iteration k.

For the Policy evaluation, We have

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_y p(y|x, \pi(x)) V^\pi(y)$$

Which written in matrix form gives

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

With

$$P^\pi = \begin{bmatrix} p(x_1|x_1, \pi(x_1)) & \dots & p(x_N|x_1, \pi(x_1)) \\ \vdots & & \vdots \\ p(x_1|x_N, \pi(x_N)) & \dots & p(x_N|x_N, \pi(x_N)) \end{bmatrix}$$

Hence

$$(I - \gamma P^\pi)V^\pi = R^\pi$$

If we have $|X|$ states and $|A|$ actions, the complexity of evaluating a policy by this method is the cost of inverting a linear system $O(|X|^3)$ (the cost of evaluating P^π and R^π are negligeable, in fact we could directly do the calculation directly without calculating them). We could also compute it iteratively using Bellman equations, which would lead to the number of iterations K times the cost of multiplying a $N \times N$ matrix by a N vector, which would be $O\left(|X|^2 K\right)$ with $K = \dfrac{\log r_{max}/\epsilon}{\log 1/\gamma}$.

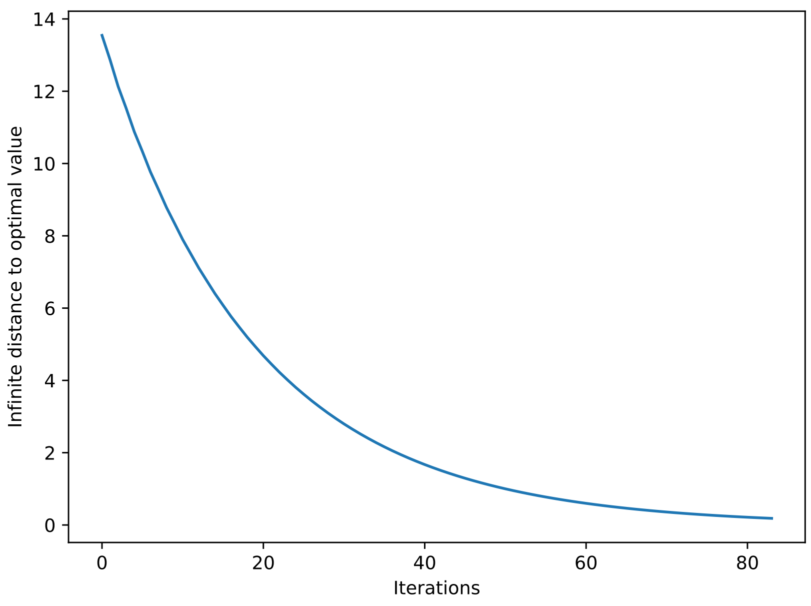

FIGURE 1 – Convergence of the value iteration algorithm

- Q3: The Policy Iteration (PI) algorithm is implemented in the file `lib_tp1_ex1.py`. In this case, the algorithm converges in 2 iterations.

At each iteration of the Policy Iteration algorithm, we need to compute a Policy evaluation, which has a cost of $O(|X|^3)$, and actually improve that policy, which has a cost of $O(|X|^2|A|)$.

The cost of the Value Iteration algorithm is K (number of iterations) times the cost of computing

$$V^{k+1}(x) = \max_a r(x,a) + \gamma \sum_y p(y|x,a)V^k(y)$$

which is $O(|X|^2|A|)$

Hence the costs are

— Value iteration : $O(|X|^2|A|\dfrac{\log \epsilon}{\log \gamma})$

— Policy iteration (evaluation in close form) : $O(L|X|^3)$

— Policy iteration (evaluation with Bellman equations) : $O(L|X|^2\dfrac{\log \epsilon}{\log \gamma})$

Where L is the number of iteration of the most outer loop of the Policy iteration algorithm.

We are doing mostly the same operations in the 2 algorithm, so the constants that stands before the big O are almost the same. Since here L is very small ($L = 2$ in our case), and given the fact that we don't need a very precise evaluation at each step, using the iterative version of the policy evaluation with a relatively large ϵ, should lead to a much lower constant, and hence a much faster convergence. In our case, because the number of states is small, solving the linear system is also very cheap which explains why the policy iteration is faster (In fact if it wouldn't, we would always use value iteration, which retrieves the policy for free, and policy iteration would be pointless).

2 Reinforcement Learning

- Q4:

Policy Evaluation

First, we define the policy that goes right if possible, otherwise up. The figure 2 represents the policy. (I only changed the colors so that we can see the missing cell, and a line in `gridrender.py` to export in postscript)

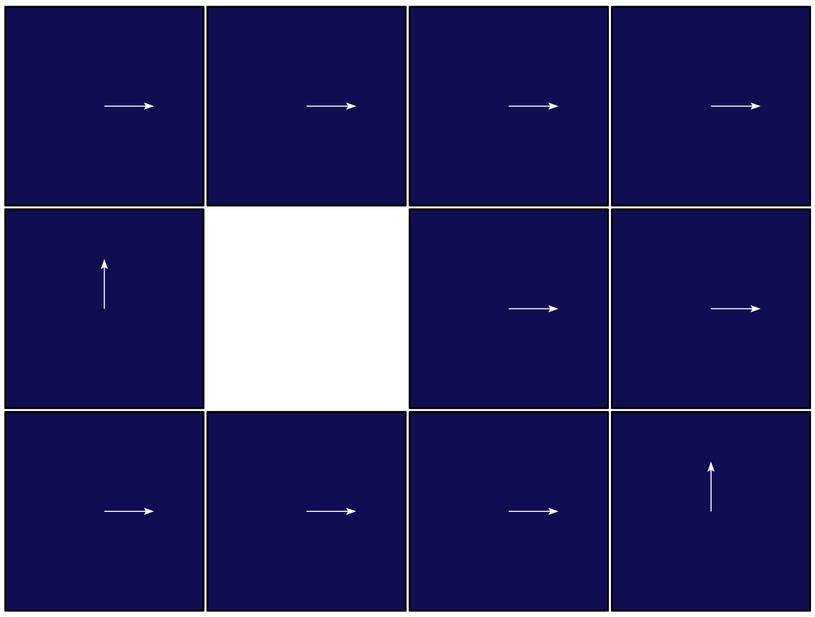

FIGURE 2 – Representation of the policy to evaluate

The estimated state value function obtained by a run of the Monte-Carlo algorithm (function **`EvaluatePolicy`** in the file `lib_tp1_ex1.py`) with 10000 iterations is given of figure 3 (to be compared with the real state value function for this policy given, given in figure 4

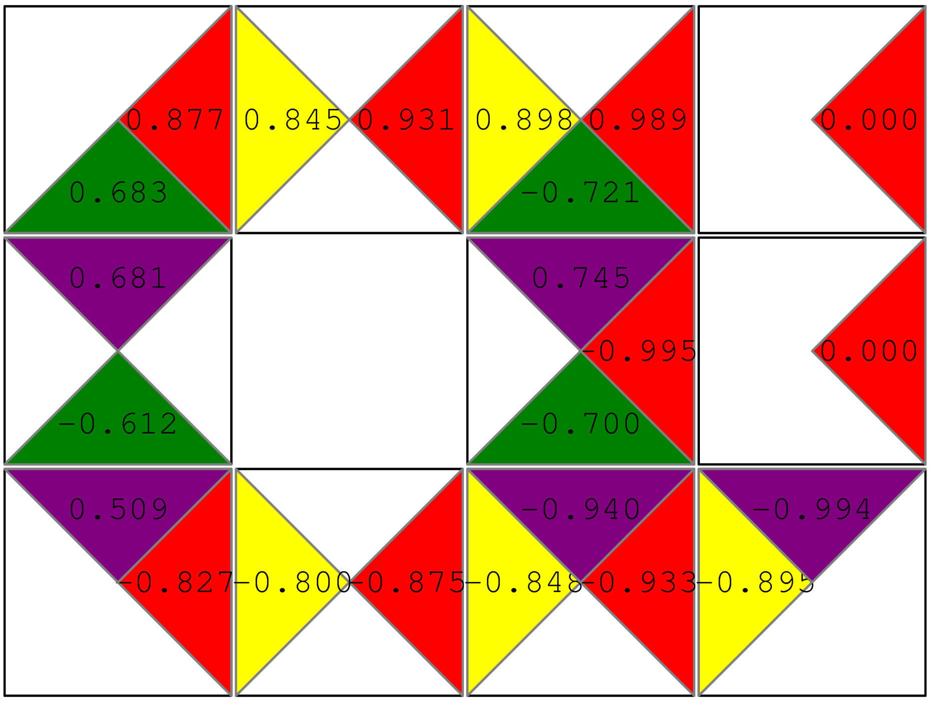

FIGURE 3 – Representation of the state value $\hat{Q}^\pi$ function obtained when evaluating the policy with the Monte-Carlo algorithm (10000 iterations)

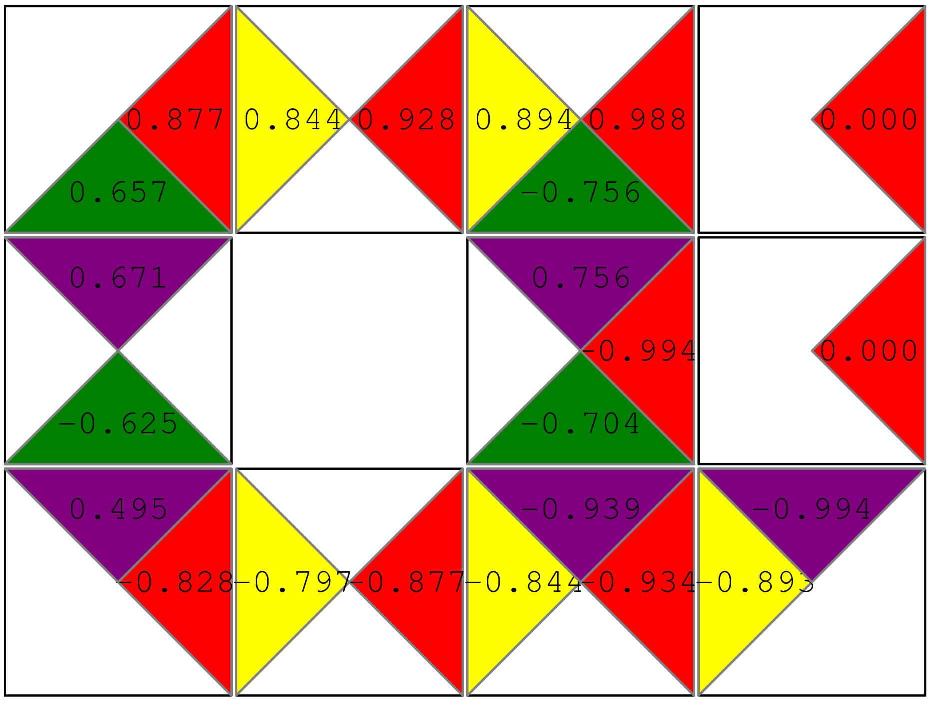

FIGURE 4 – Representation of the real state value function Q^π given

The results are good, since x_3 (in row 0 and column 3) is the good state (reward $+1$) and x_6 (in row 1 and column 3) is the bad state (reward -1). From the policy, we go right if we can, so basically, we need to be in one of the green cells (in figure 5) after step 1 if we want a positive reward, and that's exactly what we get when looking at the representation of Q.

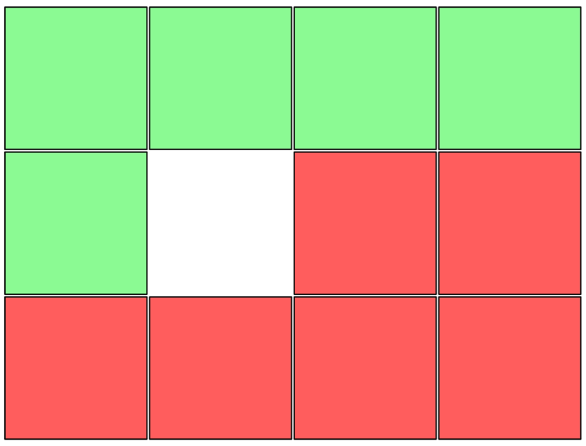

FIGURE 5 – Green cells (resp. red cells) represent the cells we have to be after the first step to obtain a positive (resp. negative) reward (for the policy)

The figure 6 represents the evolution of $J_k - J^\pi$ as a function of the number of iteration k

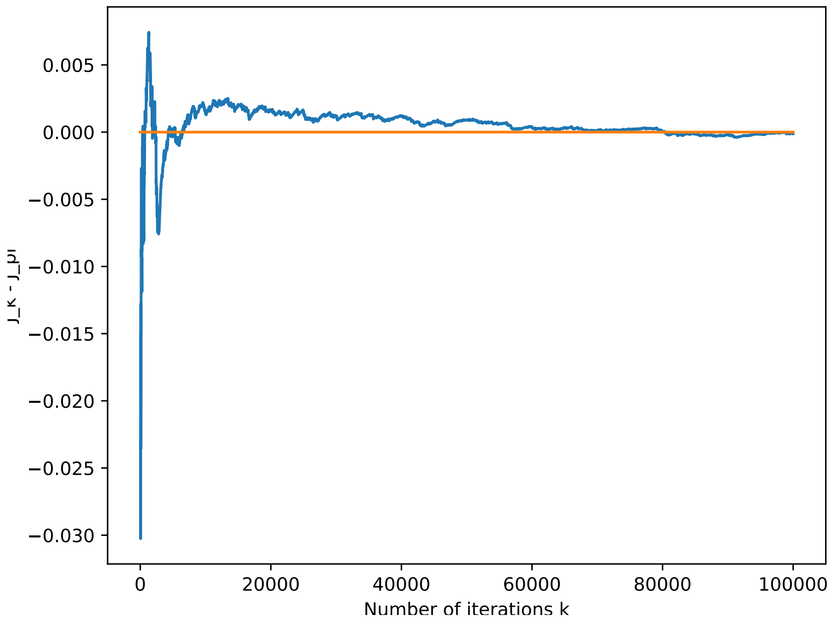

FIGURE 6 – Evolution of $J_k - J^\pi$ as a function of the number of iteration k

We can see that we have a good estimate of V^π really fast. In our example, there is a 10% probability of going the other direction when we do a step (by looking at the source code), this introduces a random in our simulation (if we had 100% probability of going in the direction we want, we would have basically estimated Q^π as soon as we would have tried all the pairs (state, action)).

Policy optimization

The Q-learning algorithm is implemented in the file `lib_tp1_ex1.py`.

- Q5: I've defined the step size to be the number of time we went through the same (state, action) Thus it satisfies the stochastic requirements.

$$\alpha_i(x,a) = \frac{1}{N_i(x,a)}$$

I've plotted $\|V_k - V^*\|_\infty$ as a function of the number of iteration k and the values of the reward cumulated over the episodes for different values of ϵ, the results are presented in figure 7 and 8

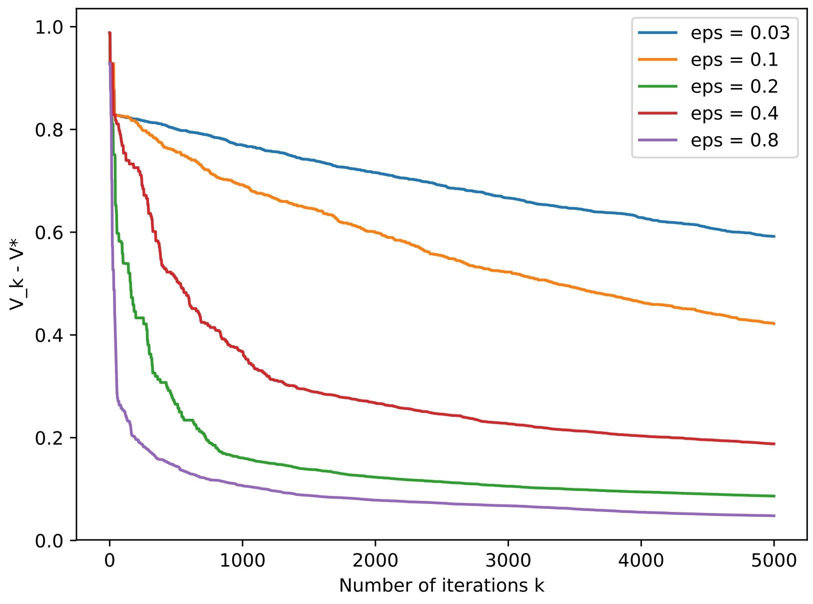

FIGURE 7 – Evolution of $\|V_k - V^*\|_\infty$ as a function of the number of iteration k for different values of ϵ

We can see that if we take a large ϵ, which means we are mainly exploring, the value decreases really fast compared to others, so this is a good strategy to have an accurate estimate of the statee value function, as expected.

The figure 8 shows the evolution of the reward cumulated over episodes when the number of episodes is low. We see that the curve with $\epsilon = 0.03$ achieves the worst performance when the number of iteration is low, which correspond to the time where it does not know anything. The algorithm will learn very slowly, but will then exploit its knowledge to get the highest possible reward (see 9)

This illustrates the exploration/exploitation tradeoff, High values of epsilon will explore very much, and hence produce accurate values of the state value function, at the cost of having a poor reward, as opposed to low values of epsilon, which would produce a high reward without exploring the problem very much.

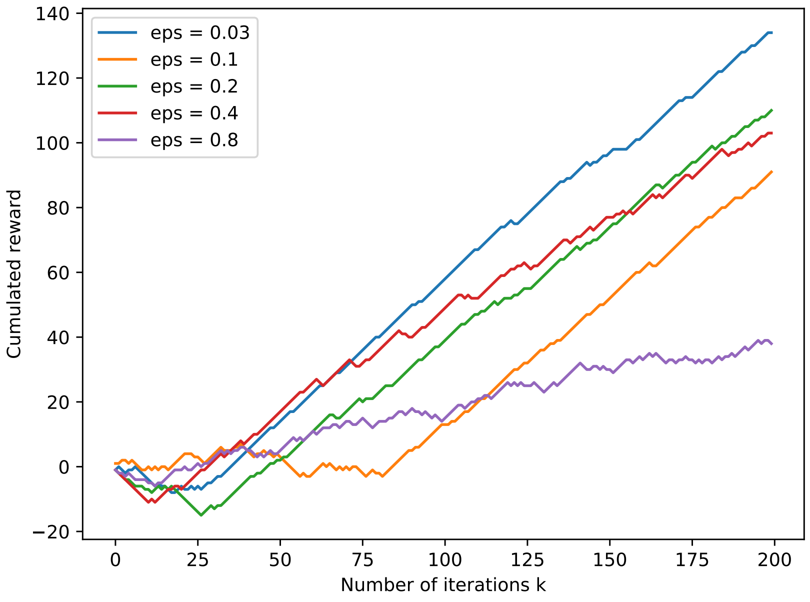

FIGURE 8 – Evolution of the reward cumulated over episodes as a function of the number of iteration k for different values of ϵ

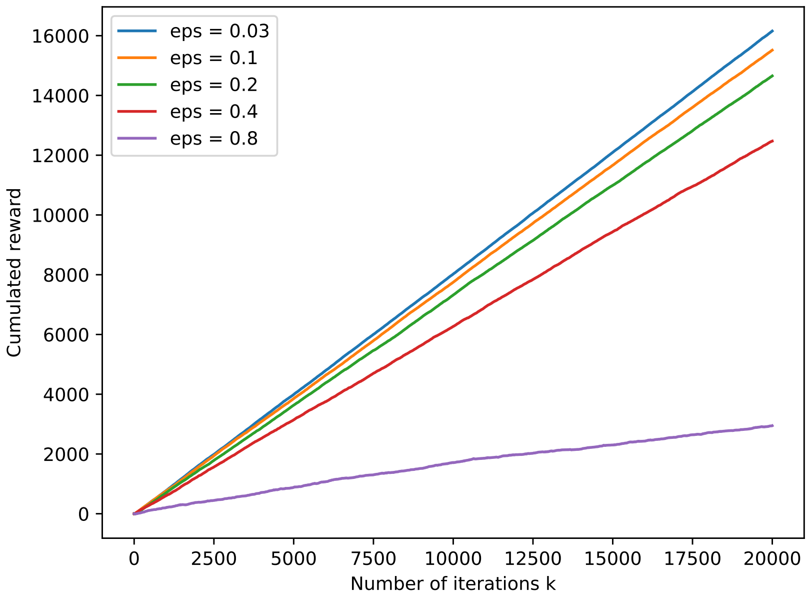

FIGURE 9 – Evolution of the reward cumulated over episodes as a function of the number of iteration k for different values of ϵ

- Q6: The initial distribution does not affect the optimal policy, because the optimal value achieves the highest value for all states. The final policy for the grid problem is presented in figure 10 In fact, we can verify it experimentally by changing the probability in the **reset** function. For example with the code proposed below, which makes the nodes with the higher indices more probables, we obtain the same estimated optimal policy and estimated optimal value function.

```python
    def reset(self):
        """
        Returns:
            An initial state randomly drawn from
            the initial distribution
        """
        states_prob = np.cumsum(np.arange(1, self.n_states + 1))
        p = np.random.randint(0, states_prob[-1])
        x_0 = np.argmax(states_prob > p)

        return x_0

```

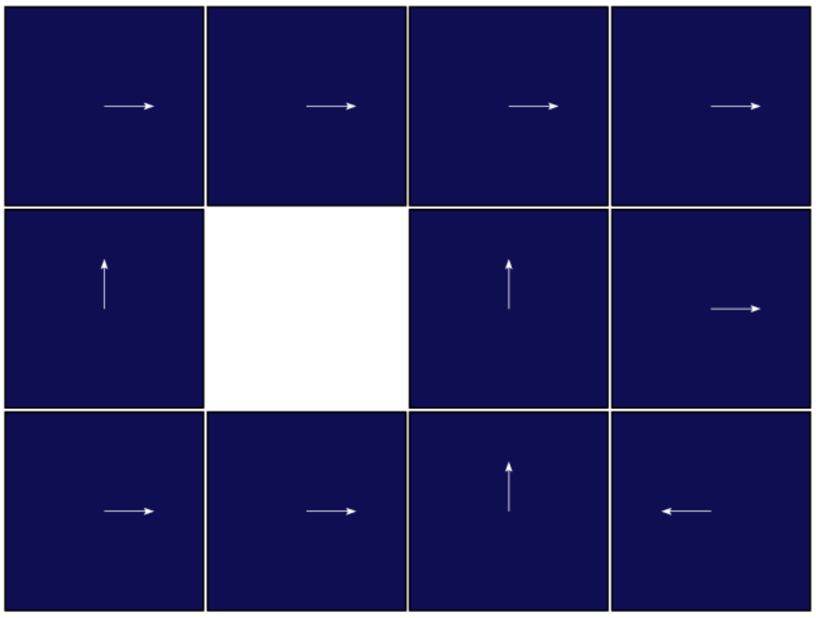

FIGURE 10 – Estimated optimal policy obtained by an execution of the Q-learning algorithm with $\epsilon = 0.2$ and $N = 20000$ episodes ($Tmax = 100$)