



MU4IN210 - ROBOTIQUE ET APPRENTISSAGE

---

## Lab : Tabular reinforcement learning

---

*Authors :*

Youssef KADHI

Vincent LIM

*Teacher :*

Olivier SIGAUD

# 1 MDPs and mazes

## Question 1

```
1 walls = [7, 8, 9, 10, 21, 27, 30, 31, 32, 33, 45, 46, 47]
2 height = 6
3 width = 9
4 m = build_maze(width, height, walls) # maze-like MDP definition
```

## 2 Dynamic Programming

### Question 3

```
1 def value_iteration_q(mdp, render=True):
2     q = np.zeros((mdp.nb_states, mdp.action_space.size)) # initial action
3     # values are set to 0
4     q_list = []
5     stop = False
6
7     if render:
8         mdp.new_render()
9
10    while not stop:
11        qold = q.copy()
12
13        if render:
14            mdp.render(q)
15
16        for x in range(mdp.nb_states):
17            for u in mdp.action_space.actions:
18                if x in mdp.terminal_states:
19                    # TODO: fill this
20                    q[x, :] = mdp.r[x, u]
21                else:
22                    # TODO: fill this
23                    summ = 0
24                    for y in range(mdp.nb_states):
25                        summ = summ + mdp.P[x, u, y] * np.max(qold[y, :])
26                    q[x, u] = mdp.r[x, u] + mdp.gamma * summ
27
28        if (np.linalg.norm(q - qold)) <= 0.01:
29            stop = True
30            q_list.append(np.linalg.norm(q))
31
32    if render:
33        mdp.render(q)
34    return q, q_list
```

### Question 4

```
1 def get_policy_from_q(q):
2     return np.argmax(q, axis=1)
```

### Question 5

```

1 def evaluate_one_step_q(mdp, q, policy):
2     # Outputs the state value function after one step of policy evaluation
3     q_new = np.zeros((mdp.nb_states, mdp.action_space.size))
4
5     for x in range(mdp.nb_states):
6         for u in mdp.action_space.actions:
7             if x in mdp.terminal_states:
8                 q_new[x, :] = mdp.r[x, u]
9             else:
10                summ = 0
11                for y in range(mdp.nb_states):
12                    summ += mdp.P[x, u, y] * q[y, policy[y]]
13                q_new[x, u] = mdp.r[x, u] + mdp.gamma * summ
14    return q_new

```

```

1 def evaluate_q(mdp, policy):
2     # Outputs the state value function of a policy
3     q = np.zeros((mdp.nb_states, mdp.action_space.size))
4     stop = False
5     while not stop:
6         qold = q.copy()
7         q = evaluate_one_step_q(mdp, qold, policy)
8
9         # Test if convergence has been reached
10        if (np.linalg.norm(q - qold)) < 0.01:
11            stop = True
12    return q

```

## Question 6

```

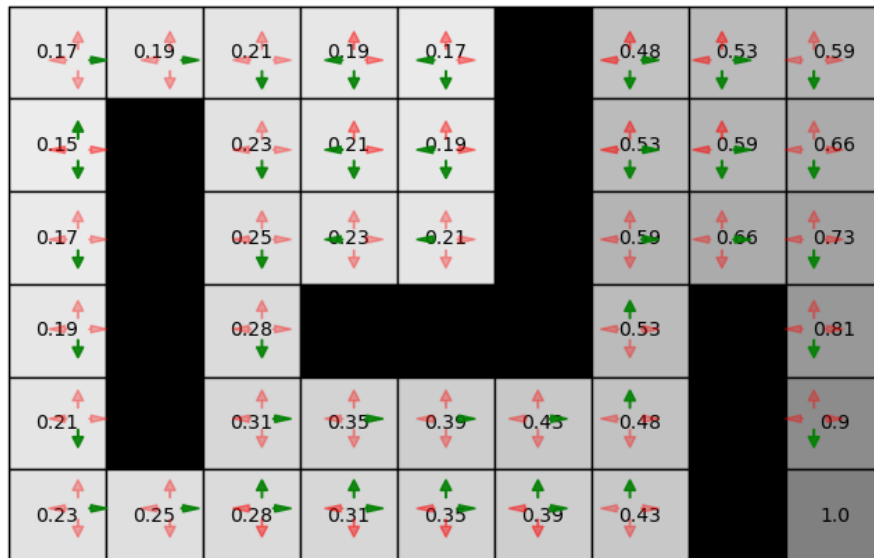
1 def policy_iteration_q(mdp, render=True): # policy iteration over the q
    function
2     q = np.zeros((mdp.nb_states, mdp.action_space.size)) # initial action
    values are set to 0
3     q_list = []
4     policy = random_policy(mdp)
5
6     stop = False
7
8     if render:
9         mdp.new_render()
10
11    while not stop:
12        qold = q.copy()
13
14        if render:
15            mdp.render(q)
16            mdp.plotter.render_pi(policy)

```

```

17
18     # Step 1 : Policy evaluation
19     q = evaluate_q(mdp, policy)
20
21     # Step 2 : Policy improvement
22     policy = get_policy_from_q(q)
23
24
25
26     # Check convergence
27     if (np.linalg.norm(q - qold)) <= 0.01:
28         stop = True
29         q_list.append(np.linalg.norm(q))
30
31     if render:
32         mdp.render(q, get_policy_from_q(q))
33     return q, q_list

```

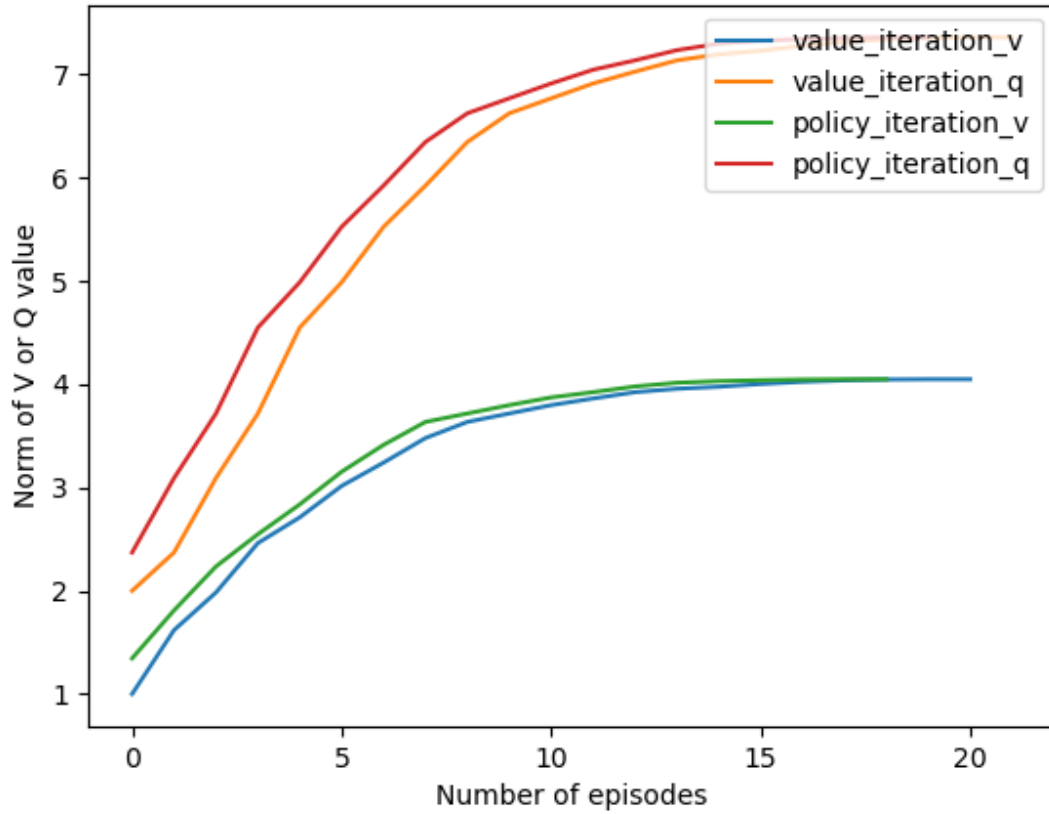


## Question 7

```
1 def policy_iteration_v(mdp, render=True):
2     # policy iteration over the v function
3     v = np.zeros(mdp.nb_states) # initial state values are set to 0
4     v_list = []
5     policy = random_policy(mdp)
6
7     stop = False
8
9     if render:
10         mdp.new_render()
11
12     while not stop:
13         vold = v.copy()
14         # Step 1 : Policy Evaluation
15         v = evaluate_v(mdp, policy)
16
17         if render:
18             mdp.render(v)
19             mdp.plotter.render_pi(policy)
20
21         # Step 2 : Policy Improvement
22         policy = get_policy_from_v(mdp, v)
23
24         # Check convergence
25         if (np.linalg.norm(v - vold)) < 0.01:
26             stop = True
27             v_list.append(np.linalg.norm(v))
28
29     if render:
30         mdp.render(v)
31         mdp.plotter.render_pi(policy)
32     return v, v_list
```

0.17 →	0.19 →	0.21 ↓	0.19 ↓	0.17 ↓		0.48 ↓	0.53 ↓	0.59 ↓
↑ 0.15		0.23 ↓	←0.21	←0.19		0.53 →	0.59 ↓	0.66 ↓
0.17 ↓		0.25 ↓	←0.23	←0.21		0.59 →	0.66 →	0.73 ↓
0.19 ↓		0.28 ↓				↑ 0.53		0.81 ↓
0.21 ↓		0.31 →	0.35 →	0.39 →	0.43 →	↑ 0.48		0.9 ↓
0.23 →	0.25 →	0.28 →	0.31 →	0.35 ↑	0.39 ↑	0.43 ↑		1.0

Question 8



function	number of iterations	number of elementary update	time
Value iteration v	21	505600	0s 210ms
Value iteration q	22	530880	1s 787ms
Policy iteration v	19	2186720	1s 113ms
Policy iteration q	20	6724480	3s 640ms

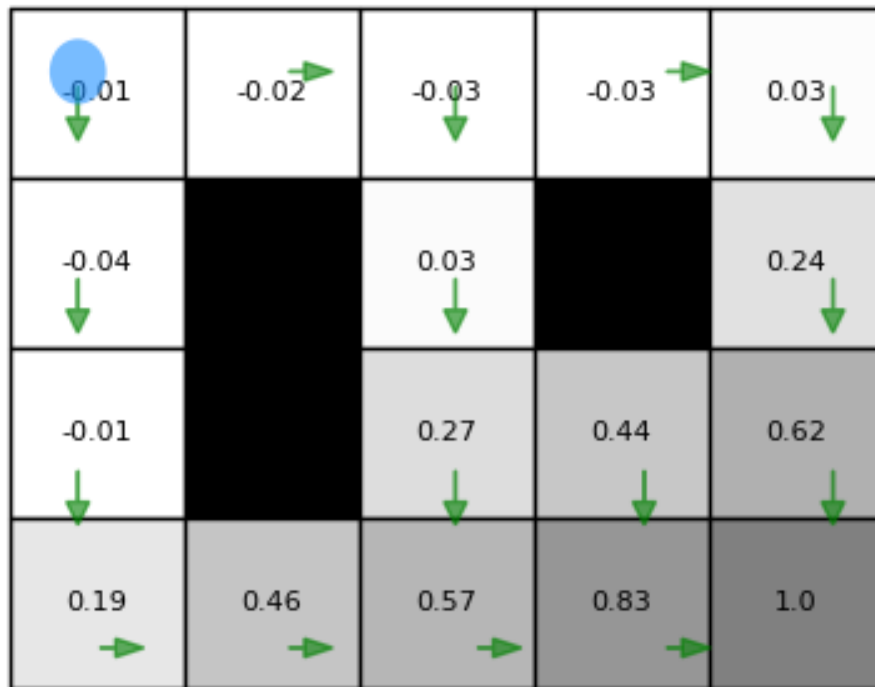
Policy iteration takes less iteration than value iteration but each one of his steps cost a lot of operation(almost as many as a complete value iteration) so it's slower.And we notice that the V function is quicker than the Q function probably due to the increased use of the max function in the Q function.



### 3 Reinforcement learning functions

#### Question 9

```
1 def temporal_difference(mdp, pol, nb_episodes=50, alpha=0.2, timeout=25,
2   render=True):
3     # alpha: learning rate
4     # timeout: timeout of an episode (maximum number of timesteps)
5     v = np.zeros(mdp.nb_states) # initial state value v
6     mdp.timeout = timeout
7
8     if render:
9         mdp.new_render()
10
11     for _ in range(nb_episodes): # for each episode
12
13         # Draw an initial state randomly (if uniform is set to False, the
14         # state is drawn according to the P0
15         #                                     distribution)
16         x = mdp.reset(uniform=True)
17         done = mdp.done()
18         while not done: # update episode at each timestep
19             # Show agent
20             if render:
21                 mdp.render(v, pol)
22
23             # Step forward following the MDP:
24             # x=current state,
25             # pol[i]=agent's action according to policy pol,
26             # r=reward gained after taking action pol[i],
27             # done=tells whether the episode ended,
28             # and info gives some info about the process
29             [y, r, done, _] = mdp.step(egreedy_loc(int(pol[x]), mdp.
30   action_space.size, epsilon=0.2))
31
32             # Update the state value of x
33             if x in mdp.terminal_states:
34                 v[x] = r
35             else:
36                 delta = r + mdp.gamma*v[y]-v[x]
37                 v[x] = v[x]+alpha*delta
38
39             # Update agent's position (state)
40             x = y
41
42     if render:
43         # Show the final policy
44         mdp.current_state = 0
45         mdp.render(v, pol)
46     return v
```



## Question 10

```

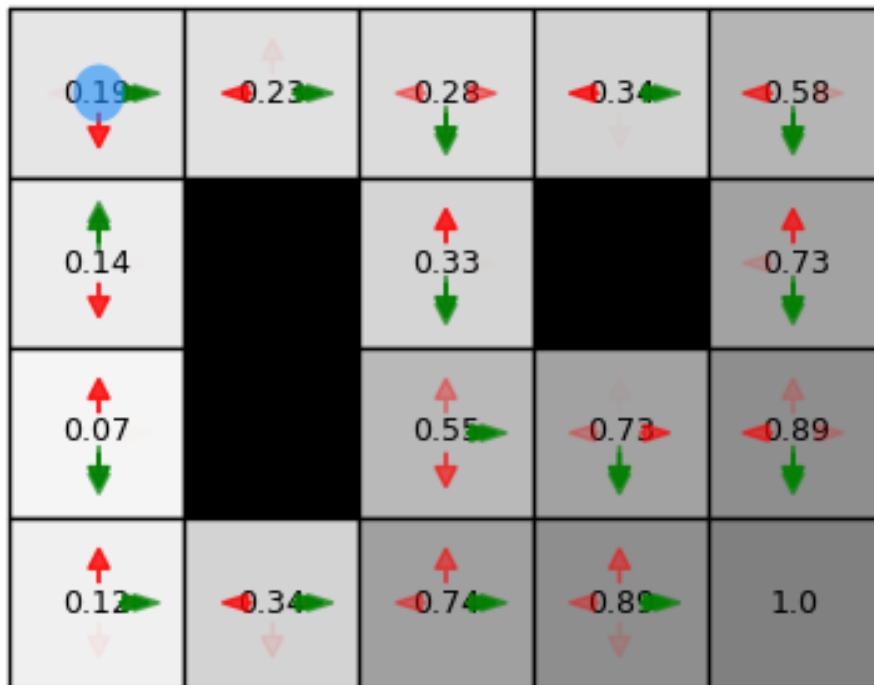
1 def q_learning_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5,
2   render=True):
3     # Initialize the state-action value function
4     # alpha is the learning rate
5     q = np.zeros((mdp.nb_states, mdp.action_space.size))
6     q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
7     q_list = []
8
9     # Run learning cycle
10    mdp.timeout = timeout # episode length
11
12    if render:
13        mdp.new_render()
14
15    for _ in range(nb_episodes):

```

```

15     # Draw the first state of episode i using a uniform distribution
over all the states
16     x = mdp.reset(uniform=True)
17     done = mdp.done()
18     while not done:
19         if render:
20             # Show the agent in the maze
21             mdp.render(q, q.argmax(axis=1))
22
23         # Draw an action using a soft-max policy
24         u = mdp.action_space.sample(prob_list=softmax(q, x, tau))
25
26         # Perform a step of the MDP
27         [y, r, done, _] = mdp.step(u)
28
29         # Update the state-action value function with q-Learning
30         if x in mdp.terminal_states:
31             q[x, u] = mdp.r[x, u] #TODO: fill this
32         else:
33             delta = mdp.r[x, u] + mdp.gamma * q[y, np.argmax(q[y])] -
q[x, u] #TODO: fill this
34             q[x, u] = q[x, u] + alpha*delta #TODO: fill this
35
36         # Update the agent position
37         x = y
38         q_list.append(np.linalg.norm(np.maximum(q, q_min)))
39
40     if render:
41         # Show the final policy
42         mdp.current_state = 0
43         mdp.render(q, get_policy_from_q(q))
44     return q, q_list

```



### Question 11

```

1 def q_learning_eps(mdp, epsilon, nb_episodes=20, timeout=50, alpha=0.5,
2   render=True):
3     # Initialize the state-action value function
4     # alpha is the learning rate
5     q = np.zeros((mdp.nb_states, mdp.action_space.size))
6     q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
7     q_list = []
8
9     # Run learning cycle
10    mdp.timeout = timeout # episode length
11
12    if render:
13        mdp.new_render()
14
15    for _ in range(nb_episodes):

```

```

15     # Draw the first state of episode i using a uniform distribution
    over all the states
16     x = mdp.reset(uniform=True)
17     done = mdp.done()
18     while not done:
19         if render:
20             # Show the agent in the maze
21             mdp.render(q, q.argmax(axis=1))
22
23         # Draw an action using a soft-max policy
24         u = egreedy(q, x, epsilon)
25
26         # Perform a step of the MDP
27         [y, r, done, _] = mdp.step(u)
28
29         # Update the state-action value function with q-Learning
30         if x in mdp.terminal_states:
31             q[x, u] = mdp.r[x, u] #TODO: fill this
32         else:
33             delta = mdp.r[x, u] + mdp.gamma * q[y, np.argmax(q[y])] -
q[x, u] #TODO: fill this
34             q[x, u] = q[x, u] + alpha*delta #TODO: fill this
35
36         # Update the agent position
37         x = y
38         q_list.append(np.linalg.norm(np.maximum(q, q_min)))
39
40     if render:
41         # Show the final policy
42         mdp.current_state = 0
43         mdp.render(q, get_policy_from_q(q))
44     return q, q_list

```

## Question 12

```

1 def sarsa_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5, render=
    True):
2
3     q = np.zeros((mdp.nb_states, mdp.action_space.size))
4     q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
5     q_list = []
6     predefined = False
7
8     mdp.timeout = timeout # episode length
9
10    if render:
11        mdp.new_render()
12
13    for _ in range(nb_episodes):

```

```

14     x = mdp.reset(uniform=True)
15     done = mdp.done()
16
17     while not done:
18         if render:
19             mdp.render(q, q.argmax(axis=1))
20
21             u = mdp.action_space.sample(prob_list=softmax(q, x, tau))
22
23             [y, r, done, _] = mdp.step(u)
24
25             y = discreteProb(mdp.P[x,u,:]) % mdp.nb_states
26             u1 = mdp.action_space.sample(prob_list=softmax(q, y, tau))
27
28             if x in mdp.terminal_states:
29                 q[x, u] = r
30             else:
31                 delta = r + mdp.gamma * q[y, u1] - q[x, u]
32                 q[x, u] = q[x, u] + alpha*delta
33
34             x = y
35             u = u1
36             q_list.append(np.linalg.norm(np.maximum(q, q_min)))
37
38     if render:
39         mdp.current_state = 0
40         mdp.render(q, get_policy_from_q(q))
41     return q, q_list

```

```

1 def sarsa_eps(mdp, epsilon, nb_episodes=20, timeout=50, alpha=0.5, render=
  True):
2     q = np.zeros((mdp.nb_states, mdp.action_space.size))
3     q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
4     q_list = []
5     predefined = False
6
7     mdp.timeout = timeout # episode length
8
9     if render:
10         mdp.new_render()
11
12     for _ in range(nb_episodes):
13         x = mdp.reset(uniform=True)
14         done = mdp.done()
15
16         while not done:
17             if render:
18                 mdp.render(q, q.argmax(axis=1))
19
20             u = egreedy(q, x, epsilon)
21

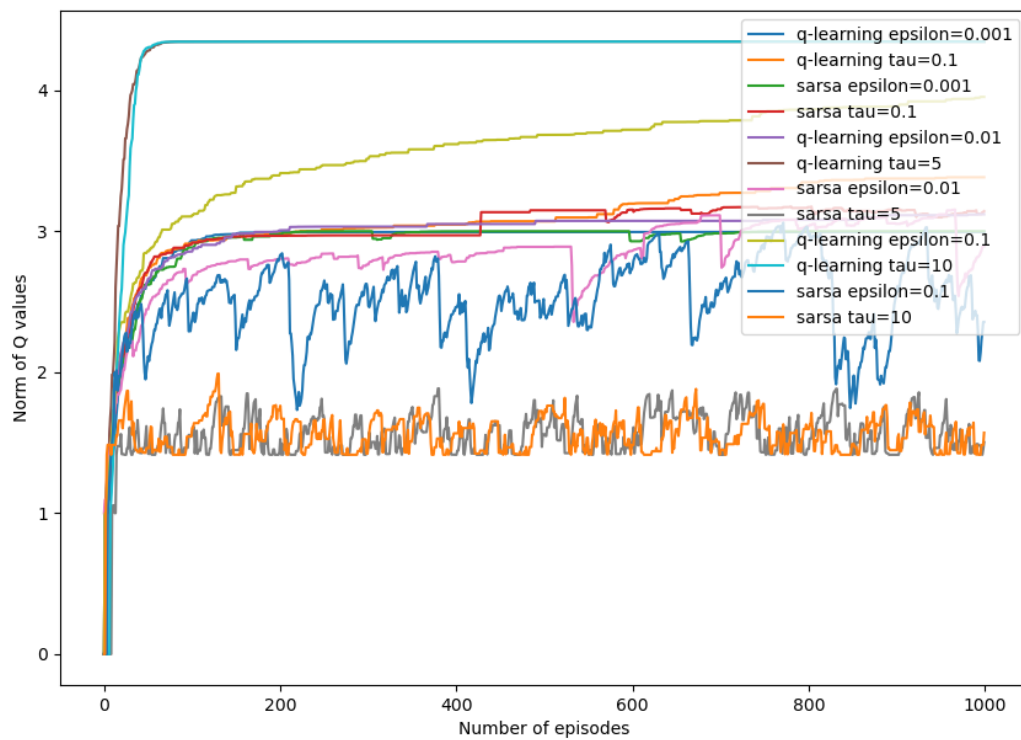
```

```

22     [y, r, done, _] = mdp.step(u)
23
24     y = discreteProb(mdp.P[x,u,:]) % mdp.nb_states
25     u1 = egreedy(q, y, epsilon)
26
27     if x in mdp.terminal_states:
28         q[x, u] = r
29     else:
30         delta = r + mdp.gamma * q[y, u1] - q[x, u]
31         q[x, u] = q[x, u] + alpha*delta
32
33     x = y
34     u = u1
35     q_list.append(np.linalg.norm(np.maximum(q, q_min)))
36
37 if render:
38     mdp.current_state = 0
39     mdp.render(q, get_policy_from_q(q))
40 return q, q_list

```

### Question 13



When we choose a higher epsilon/tau randomness increase :

For q learning who is off policy the effect is an higher Norm but it still converge and give an effective result. Sarsa is on policy, so with a lot of randomness he's not going to learn anything effective and doesn't converge.

#### **Question 14**

alpha is the learning rate, setting it too low and the agent learn nothing new, too high and the agent will not have a memory of old value.

gamma is the discount factor, close to 0 the agent will prefer small immediate reward, close to one and the agent will prefer large late rewards.