MU4IN210 - Robotique et Apprentissage

Lab : Tabular reinforcement learning

Authors :
Youssef KADHI
Vincent LIM

Teacher :
Olivier SIGAUD

1 MDPs and mazes

Question 1

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

2 Dynamic Programming

Question 3

```python
def value_iteration_q(mdp, render=True):
    q = np.zeros((mdp.nb_states, mdp.action_space.size))  # initial action values are set to 0
    q_list = []
    stop = False

    if render:
        mdp.new_render()

    while not stop:
        qold = q.copy()

        if render:
            mdp.render(q)

        for x in range(mdp.nb_states):
            for u in mdp.action_space.actions:
                if x in mdp.terminal_states:
                    # TODO: fill this
                    q[x, :] = mdp.r[x, u]
                else:
                    # TODO: fill this
                    summ = 0
                    for y in range(mdp.nb_states):
                        summ = summ + mdp.P[x, u, y] * np.max(qold[y, :])
                    q[x, u] = mdp.r[x, u]+mdp.gamma * summ

        if (np.linalg.norm(q - qold)) <= 0.01:
            stop = True
        q_list.append(np.linalg.norm(q))

    if render:
        mdp.render(q)
    return q, q_list
```

Question 4

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

Question 5

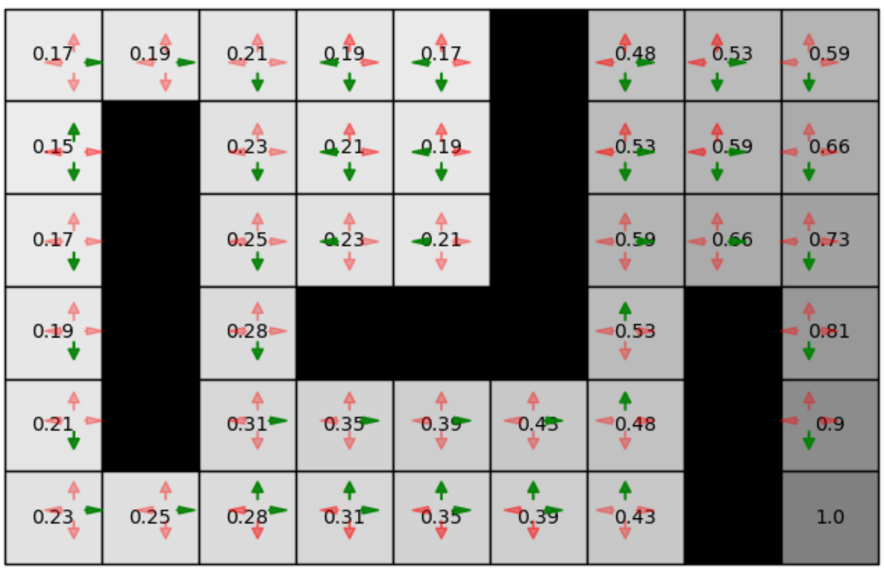
```python
def evaluate_one_step_q(mdp, q, policy):
    # Outputs the state value function after one step of policy evaluation
    q_new = np.zeros((mdp.nb_states, mdp.action_space.size))

    for x in range(mdp.nb_states):
        for u in mdp.action_space.actions:
            if x in mdp.terminal_states:
                q_new[x, :] = mdp.r[x, u]
            else:
                summ = 0
                for y in range(mdp.nb_states):
                    summ += mdp.P[x, u, y] * q[y, policy[y]]
                q_new[x, u] = mdp.r[x, u] + mdp.gamma * summ
    return q_new
```

```python
def evaluate_q(mdp, policy):
    # Outputs the state value function of a policy
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    stop = False
    while not stop:
        qold = q.copy()
        q = evaluate_one_step_q(mdp, qold, policy)

        # Test if convergence has been reached
        if (np.linalg.norm(q - qold)) < 0.01:
            stop = True
    return q
```

Question 6

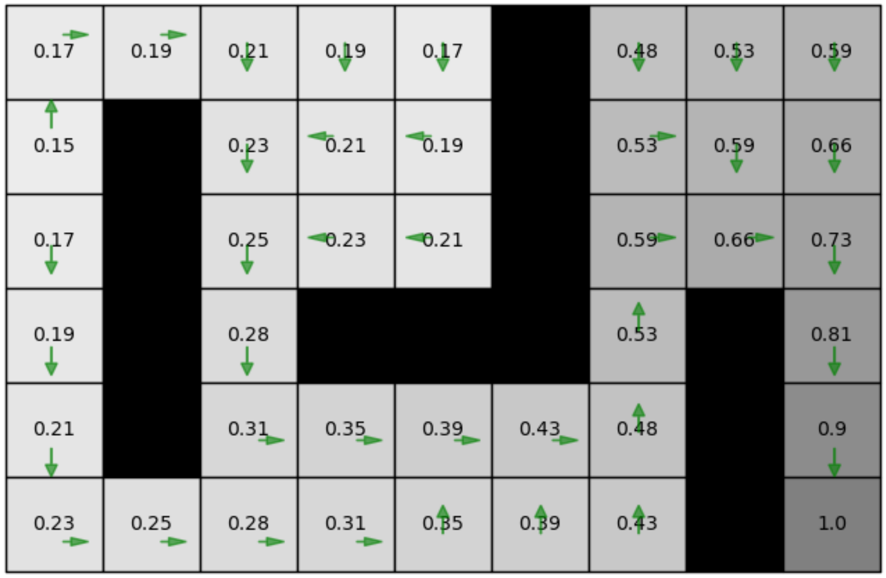
```python
def policy_iteration_q(mdp, render=True):  # policy iteration over the q function
    q = np.zeros((mdp.nb_states, mdp.action_space.size))  # initial action values are set to 0
    q_list = []
    policy = random_policy(mdp)

    stop = False

    if render:
        mdp.new_render()

    while not stop:
        qold = q.copy()

        if render:
            mdp.render(q)
            mdp.plotter.render_pi(policy)
```

```python

        # Step 1 : Policy evaluation
        q = evaluate_q(mdp, policy)

        # Step 2 : Policy improvement
        policy = get_policy_from_q(q)



        # Check convergence
        if (np.linalg.norm(q - qold)) <= 0.01:
            stop = True
        q_list.append(np.linalg.norm(q))

    if render:
        mdp.render(q, get_policy_from_q(q))
    return q, q_list
```

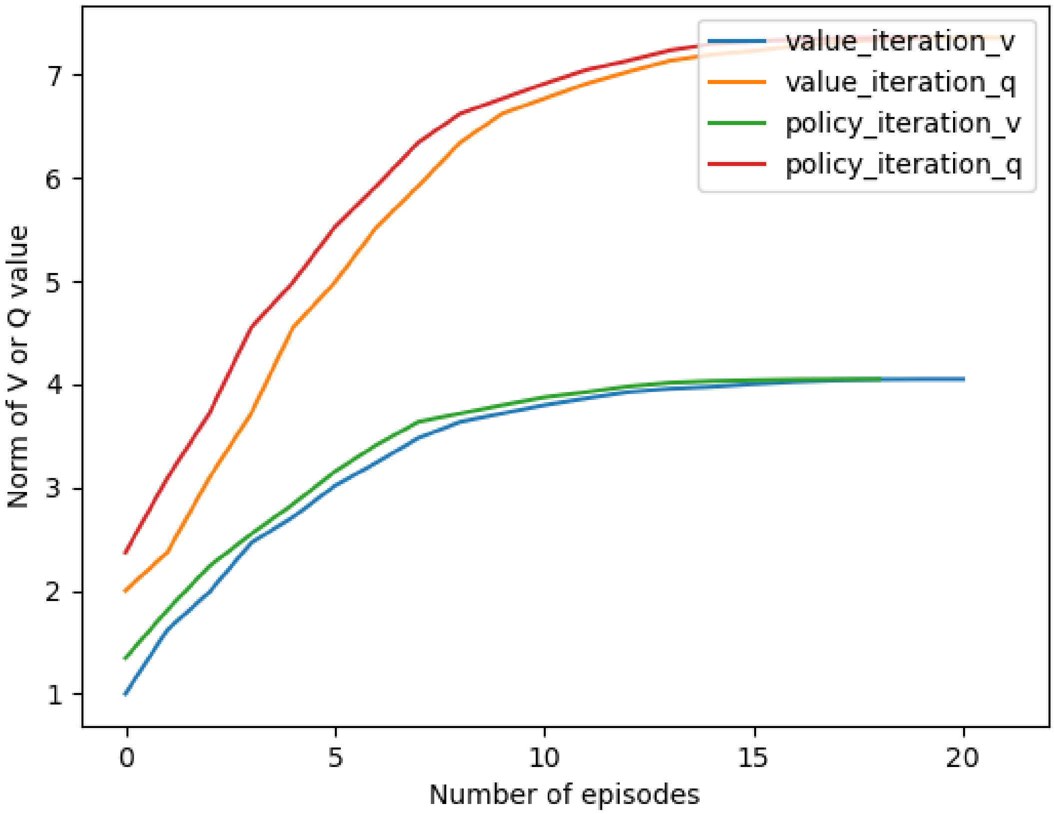


Question 7

```python
def policy_iteration_v(mdp, render=True):
    # policy iteration over the v function
    v = np.zeros(mdp.nb_states)  # initial state values are set to 0
    v_list = []
    policy = random_policy(mdp)

    stop = False

    if render:
        mdp.new_render()

    while not stop:
        vold = v.copy()
        # Step 1 : Policy Evaluation
        v = evaluate_v(mdp,policy)

        if render:
            mdp.render(v)
            mdp.plotter.render_pi(policy)

        # Step 2 : Policy Improvement
        policy = get_policy_from_v(mdp, v)

        # Check convergence
        if (np.linalg.norm(v - vold)) < 0.01:
            stop = True
        v_list.append(np.linalg.norm(v))

    if render:
        mdp.render(v)
        mdp.plotter.render_pi(policy)
    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



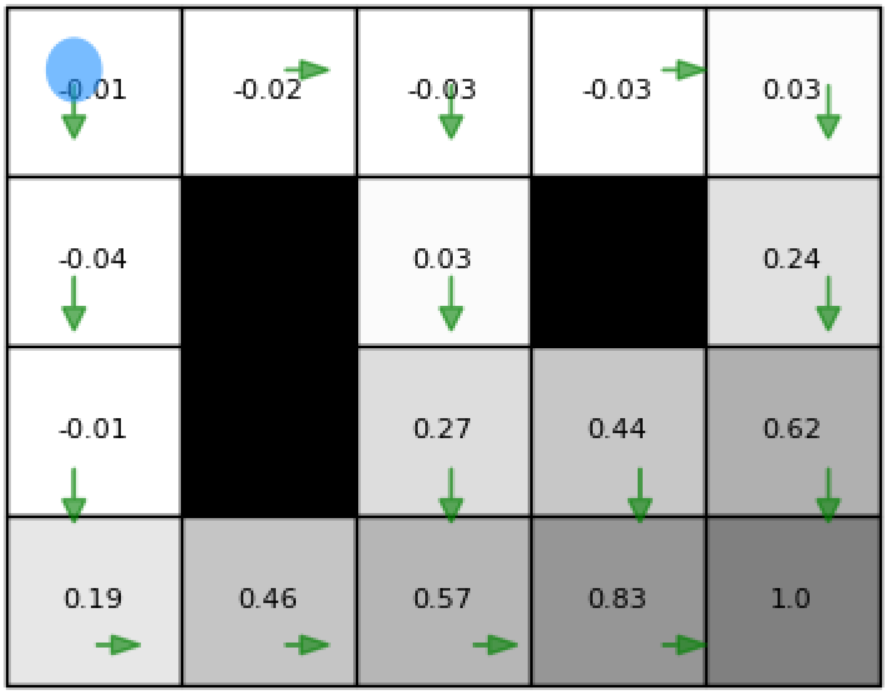
function	number of iterations	number of elementary update	time
Value iteration v	21	505600	0s 210ms
Value iteration q	22	530880	1s 787ms
Policy iteration v	19	2186720	1s 113ms
Policy iteration q	20	6724480	3s 640ms

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

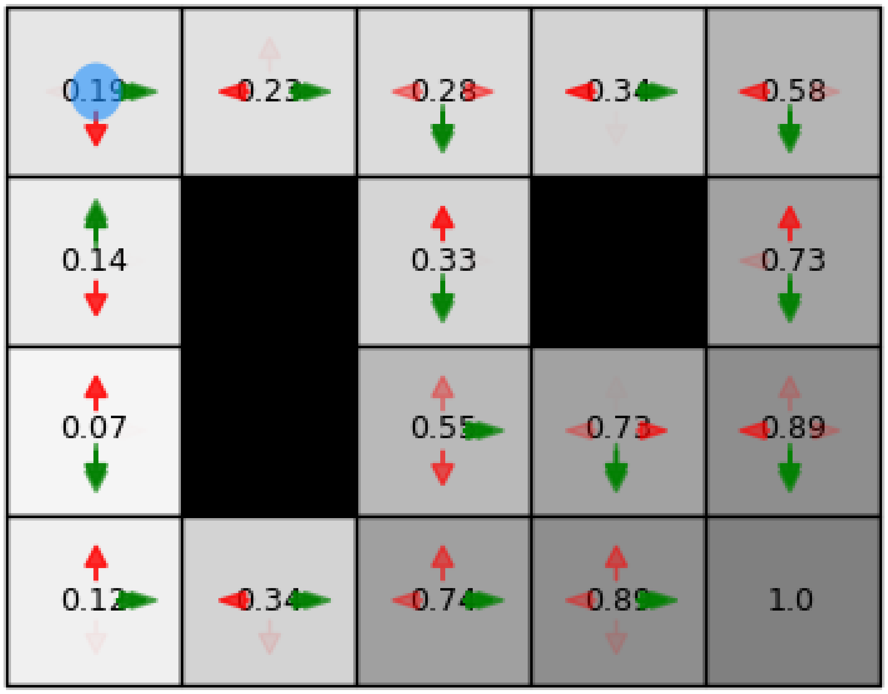
```python
def temporal_difference(mdp, pol, nb_episodes=50, alpha=0.2, timeout=25, render=True):
    # alpha: learning rate
    # timeout: timeout of an episode (maximum number of timesteps)
    v = np.zeros(mdp.nb_states)  # initial state value v
    mdp.timeout = timeout

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):  # for each episode

        # Draw an initial state randomly (if uniform is set to False, the state is drawn according to the P0
        #                                  distribution)
        x = mdp.reset(uniform=True)
        done = mdp.done()
        while not done:  # update episode at each timestep
            # Show agent
            if render:
                mdp.render(v, pol)

            # Step forward following the MDP:
            # x=current state,
            # pol[i]=agent's action according to policy pol,
            # r=reward gained after taking action pol[i],
            # done=tells whether the episode ended,
            # and info gives some info about the process
            [y, r, done, _] = mdp.step(egreedy_loc(int(pol[x]), mdp.action_space.size, epsilon=0.2))

            # Update the state value of x
            if x in mdp.terminal_states:
                v[x] = r
            else:
                delta = r + mdp.gamma*v[y]-v[x]
                v[x] = v[x]+alpha*delta

            # Update agent's position (state)
            x = y

    if render:
        # Show the final policy
        mdp.current_state = 0
        mdp.render(v, pol)
    return v
```



Question 10

```
def q_learning_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5, render=True):
    # Initialize the state-action value function
    # alpha is the learning rate
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list = []

    # Run learning cycle
    mdp.timeout = timeout  # episode length

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):
```

```
        # Draw the first state of episode i using a uniform distribution
    over all the states
        x = mdp.reset(uniform=True)
        done = mdp.done()
        while not done:
            if render:
                # Show the agent in the maze
                mdp.render(q, q.argmax(axis=1))

            # Draw an action using a soft-max policy
            u = mdp.action_space.sample(prob_list=softmax(q, x, tau))

            # Perform a step of the MDP
            [y, r, done, _] = mdp.step(u)

            # Update the state-action value function with q-Learning
            if x in mdp.terminal_states:
                q[x, u] = mdp.r[x, u] #TODO: fill this
            else:
                delta = mdp.r[x, u] + mdp.gamma * q[y, np.argmax(q[y])] -
    q[x, u] #TODO: fill this
                q[x, u] = q[x, u] + alpha*delta #TODO: fill this

            # Update the agent position
            x = y
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))

    if render:
        # Show the final policy
        mdp.current_state = 0
        mdp.render(q, get_policy_from_q(q))
    return q, q_list
```



Question 11

```python
def q_learning_eps(mdp, epsilon, nb_episodes=20, timeout=50, alpha=0.5,
    render=True):
    # Initialize the state-action value function
    # alpha is the learning rate
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list = []

    # Run learning cycle
    mdp.timeout = timeout  # episode length

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):
```

```
        # Draw the first state of episode i using a uniform distribution
    over all the states
        x = mdp.reset(uniform=True)
        done = mdp.done()
        while not done:
            if render:
                # Show the agent in the maze
                mdp.render(q, q.argmax(axis=1))

            # Draw an action using a soft-max policy
            u = egreedy(q, x, epsilon)

            # Perform a step of the MDP
            [y, r, done, _] = mdp.step(u)

            # Update the state-action value function with q-Learning
            if x in mdp.terminal_states:
                q[x, u] = mdp.r[x, u] #TODO: fill this
            else:
                delta = mdp.r[x, u] + mdp.gamma * q[y, np.argmax(q[y])] -
    q[x, u] #TODO: fill this
                q[x, u] = q[x, u] + alpha*delta #TODO: fill this

            # Update the agent position
            x = y
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))

    if render:
        # Show the final policy
        mdp.current_state = 0
        mdp.render(q, get_policy_from_q(q))
    return q, q_list
```

Question 12

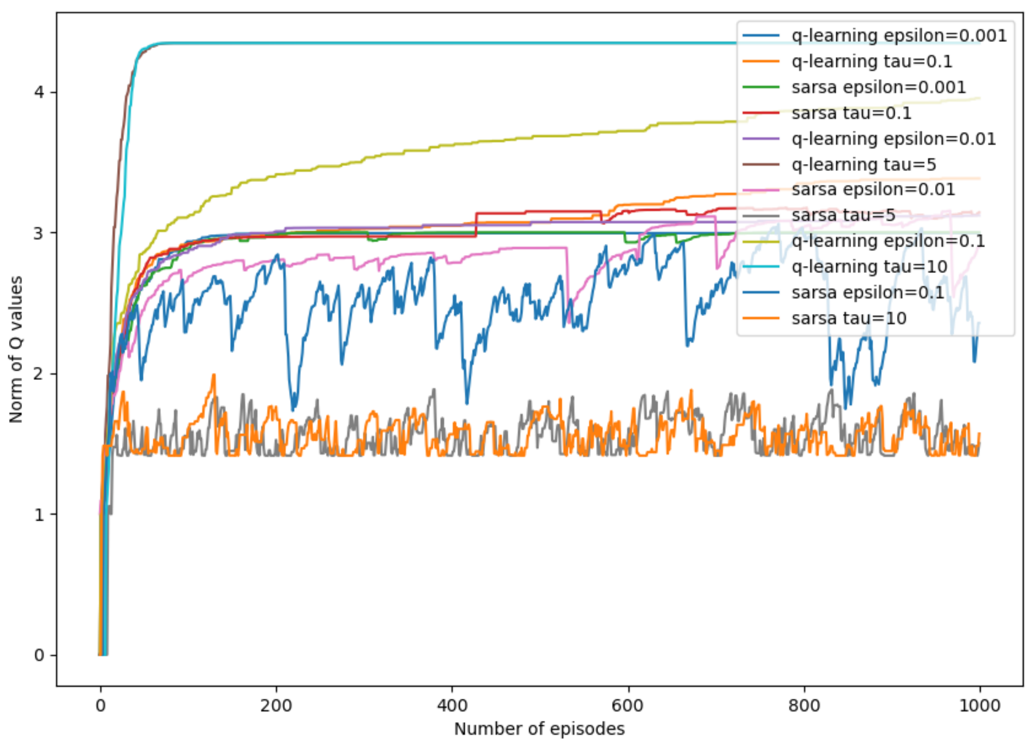
```
def sarsa_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5, render=
    True):

    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list = []
    predefined = False

    mdp.timeout = timeout  # episode length

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):
```

```
        x = mdp.reset(uniform=True)
        done = mdp.done()

        while not done:
            if render:
                mdp.render(q, q.argmax(axis=1))

            u = mdp.action_space.sample(prob_list=softmax(q, x, tau))

            [y, r, done, _] = mdp.step(u)

            y = discreteProb(mdp.P[x,u,:]) % mdp.nb_states
            u1 = mdp.action_space.sample(prob_list=softmax(q, y, tau))

            if x in mdp.terminal_states:
                q[x, u] = r
            else:
                delta = r + mdp.gamma * q[y, u1] - q[x, u]
                q[x, u] = q[x, u] + alpha*delta

            x = y
            u = u1
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))

    if render:
        mdp.current_state = 0
        mdp.render(q, get_policy_from_q(q))
    return q, q_list
```

```
def sarsa_eps(mdp, epsilon, nb_episodes=20, timeout=50, alpha=0.5, render=
    True):
  q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list = []
    predefined = False

    mdp.timeout = timeout  # episode length

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):
        x = mdp.reset(uniform=True)
        done = mdp.done()

        while not done:
            if render:
                mdp.render(q, q.argmax(axis=1))

            u = egreedy(q, x, epsilon)

```

```
            [y, r, done, _] = mdp.step(u)

            y = discreteProb(mdp.P[x,u,:]) % mdp.nb_states
            u1 = egreedy(q, y, epsilon)

            if x in mdp.terminal_states:
                q[x, u] = r
            else:
                delta = r + mdp.gamma * q[y, u1] - q[x, u]
                q[x, u] = q[x, u] + alpha*delta

            x = y
            u = u1
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))

    if render:
        mdp.current_state = 0
        mdp.render(q, get_policy_from_q(q))
    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 note 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.