Sorbonne Université

Rapport de TME

Reinforcement Learning

Zhuangzhuang YANG 28708351
XIN HE 28706290
Master 1 Semestre 2

Tuteur : M. Olivier Sigaud

Février 2021

2 MDPs and Mazes

2.2 Play with different MDPs

Code question 1

Here is the code which can build a maze like maze in figure 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
```

3 Dynamic Programming

Dynamic programming algorithms are used for planning and they require a full knowledge of the MDP from agent. They can find the best policy by computing a value function V or an action-value function Q over the stat space or state-action space of the given MDP.

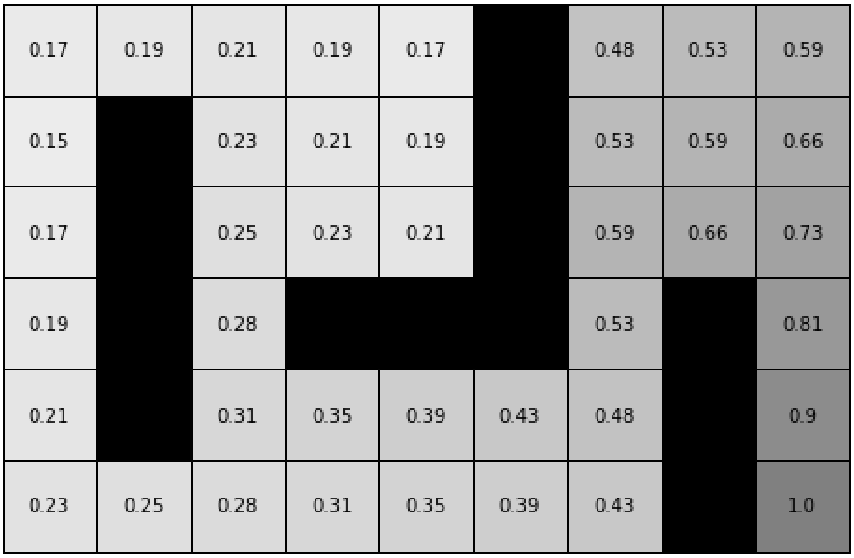
3.1 Value iteration

To calculate the values following policy π with Q function, we implemented the Bellman Optimality Equation for Q^*. And here is the missing piece of code of value_iteration_q(mdp) :

Code question 3

```
        for x in range(mdp.nb_states): #for each state x and action u
            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(q[y,:])
                    q[x, u] = mdp.r[x, u] + mdp.gamma * summ
```

The maze with the calculated state values :



3.2 Policy iteration

There are two steps of the Policy iteration :

1.evaluate policy π : compute V or Q based on the policy π

2.improve policy π : compute a better policy based on V or Q

We will use the Bellman Expectation with deterministic policy to find the optimal policy.

Code question 4

The function get_policy_from_q(q) :

```
def get_policy_from_q(mdp, q):
    # Outputs a policy given the state values
    policy = np.zeros(mdp.nb_states)  # initial state values are set to 0
    for x in range(mdp.nb_states):  # for each state x
        # Compute the value of the state x for each action u of the MDP action space
        q_temp = np.zeros((mdp.nb_states,mdp.action_space.size))
        for u in mdp.action_space.actions:
            if x not in mdp.terminal_states:
                # Process sum of the values of the neighbouring states
                summ = 0
                for y in range(mdp.nb_states):
                    summ = summ + mdp.P[x, u, y] * q[y,u]
                q_temp[x,u] = mdp.r[x, u] + mdp.gamma * summ
            else:  # if the state is final, then we only take the reward into account
                q_temp[x,:] = mdp.r[x, u]
        policy[x] = np.argmax(q_temp)
    return policy
```

Code question 5

The function evaluate_one_step_q(mdp,q,policy) :

```python
def evaluate_one_step_q(mdp, q, policy):
    # Outputs the state value function after one step of policy evaluation
    # Corresponds to one application of the Bellman Operator
    q_temp = np.zeros((mdp.nb_states, mdp.action_space.size))
    for x in range(mdp.nb_states):  # for each state x
        # Compute the value of the state x for each action u of the MDP action space
        for u in mdp.action_space.actions:
            if x not in mdp.terminal_states:
                # Process sum of the values of the neighbouring states
                summ = 0
                for y in range(mdp.nb_states):
                    summ = summ + mdp.P[x, u, y] * q[y,policy[y]]
                q_temp[x,u] = mdp.r[x, u] + mdp.gamma * summ
            else:  # if the state is final, then we only take the reward into account
                q_temp[x,:] = mdp.r[x, u]

        # q_new[x] = np.max(q_temp)
    return q_temp
```

The function evaluate_q(mdp,policy) :

```python
def evaluate_q(mdp, policy):
    # Outputs the state value function of a policy
    q = np.zeros((mdp.nb_states, mdp.action_space.size))  # initial state values are set to 0
    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
```

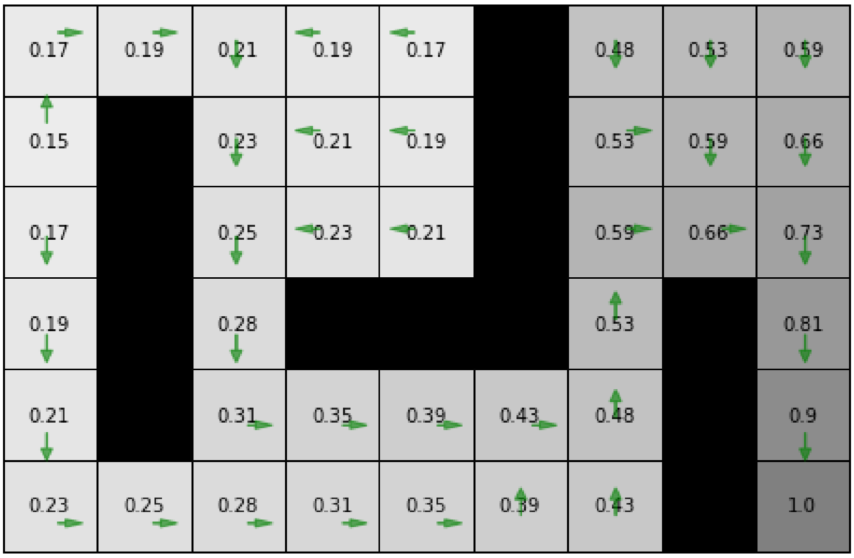
Code question 6

The missing code of the fonction policy_iteration_q(mdp) :

```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)

        # Step 1 : Policy evaluation
```

```
        # TODO: fill this
        q = evaluate_q(mdp,policy)

        # Step 2 : Policy improvement
        # TODO: fill this
        policy = improve_policy_from_q(mdp, q, policy)
        # 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
```

And the final configuration :



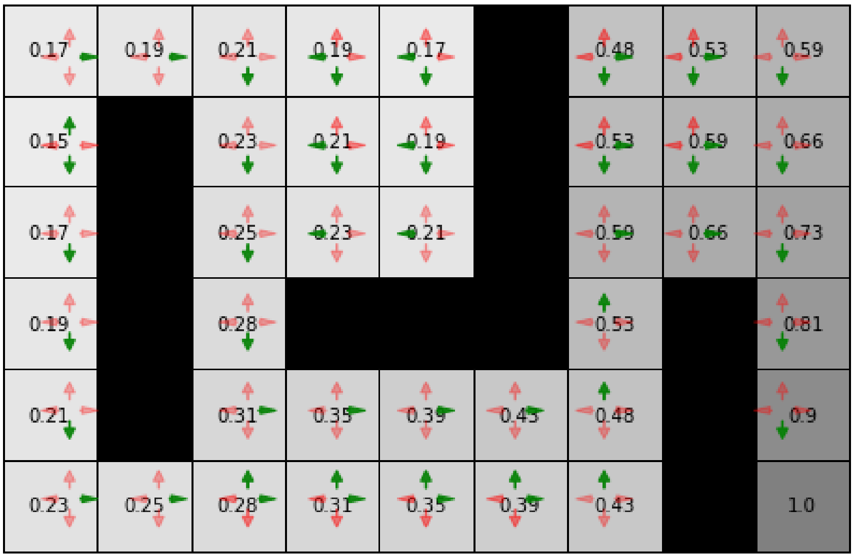
Code question 7

The missing code of the fonction policy_iteration_v(mdp) :

```
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)

```

```python
        # Step 2 : Policy Improvement
        # TODO: fill this
        policy = improve_policy_from_v(mdp,v,policy)

        # 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
```

And the final configuration :



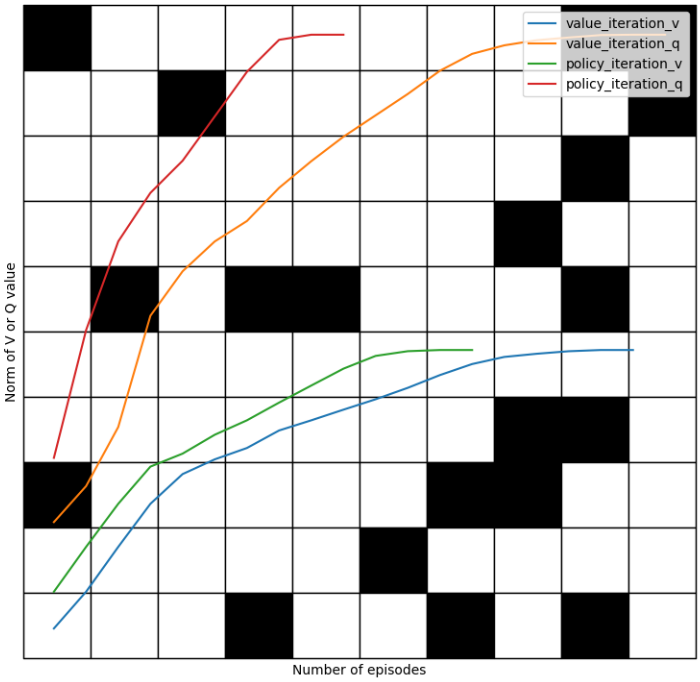
3.3 Comparisons

Study question 8

After add some code in dynamic programming functions, we have tested several different mazes, they have similar trends of results so that we select a represent one test :

method	nb_iterations	nb_elementary_updates	time_taken
value iteration V	480396	19	0s 380ms
value iteration Q	505680	20	2s 776ms
policy iteration V	13	155	1s 52ms
policy iteration Q	9	131	3s 53ms

And the image of convergence of those algorithms :



From the picture above, we can find the method policy_iteration_q has the fastest convergence, and the method policy_iteration_v has the second fastest convergence, but value_iteration_v and value_iteration_q have almost the same convergence. And we can find that value_iteration_v method take least time to get our final configuration, and it has less iterations and elementary than value_iteration_q, thus, we can drive the conclusion that value_iteration_v has better efficiency. For the two policy_iteration methods, policy_iteration_v runs faster but has a slower convergence, a more iterations of function and a more elementary updates. Thus policy_iteration_v is more efficient but not perfect. Generally, using a state value function is more efficient than a state-action value function.

4.1 TD learning

The TD-learning method is used for MDP which the agent doesn't know the transition and reward functions. It can compute the state value of the policy. And it's limitation : one cannot infer $\pi(s)$ from v(s) without knowing T, one must know which action lead to best $v^{'}$. And it's a On-policy method.

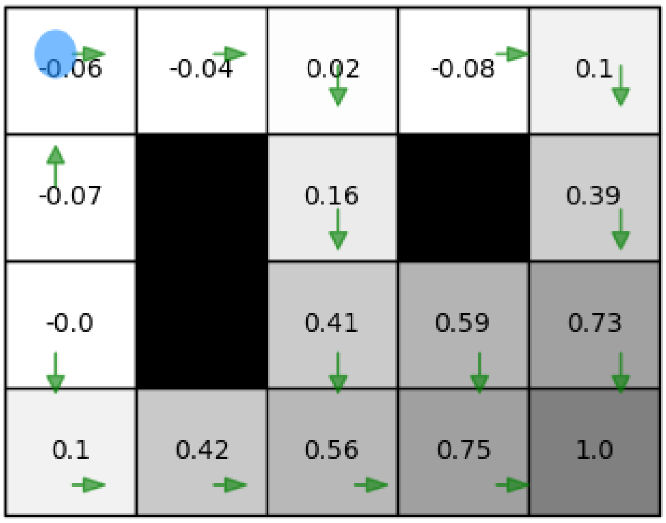
Code question 9

The corresponding code :

```
    # 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
```

And a picture of the final configuration :



4.2 Q-learning

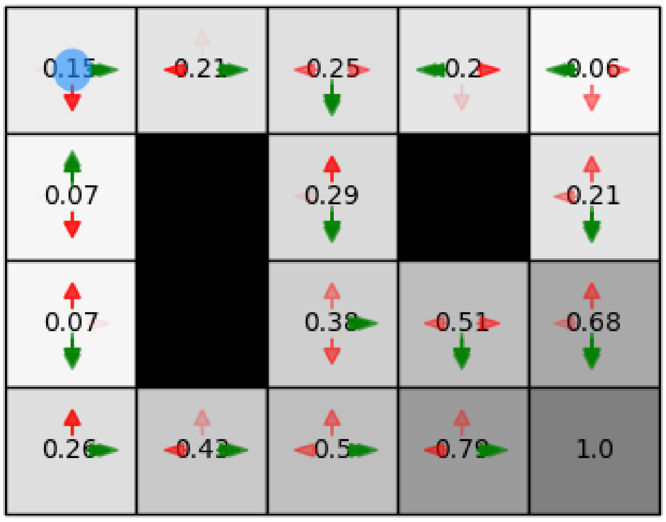
Q-learning is a method that can solve the limitation of TD(0), an agent exploring an MDP and updating at each step a model of the state-action-value. No more need to know a_{t+1} unlike the method TD(0), and it's a off-policy method.

Code question 10

The corresponding code :

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

And a picture of the final configuration :



Code question 11

The relevant piece of code :

```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 egreedy 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] = r
            else:
                delta = r + mdp.gamma * np.max(q[y]) - q[x,u]
                q[x, u] = q[x,u] + alpha*delta
            # Update the agent position
            x = y
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))
```

```python

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

4.3 SARSA

The difference between q_learning method is that Sarsa's update approach :

1. Sarsa is an update method of on-policy, and its action strategy and evaluation strategy are both ϵ-greedy strategies.

2. Sarsa is updated after first action, first execute the action through the ϵ-greedy strategy, and then update the value function according to the executed action.

Code question 12

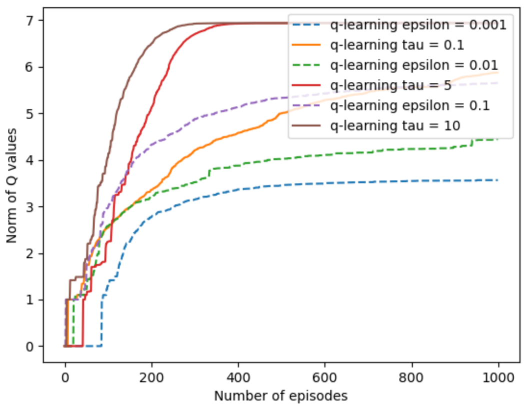
```python
#sarsa with softmax
def sarsa_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5, render=True):
    # Initialize the state-action value function
    # alpha is the learning rate
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))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

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

    if render:
        mdp.new_render()

    for i in range(nb_episodes):
        print(i)
        # Draw the first state of episode i using a uniform distribution over
    all the states
        x = mdp.reset(uniform=True)
        ux = 0
        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] = r
            else:
                uy = mdp.action_space.sample(prob_list=softmax(q, y, tau))
                delta = r + mdp.gamma * q[y,uy] - q[x,u]
                q[x, u] = q[x,u] + alpha*delta
            # Update the agent position
            x = y
        q_list.append(np.linalg.norm(np.maximum(q, q_min)))
```

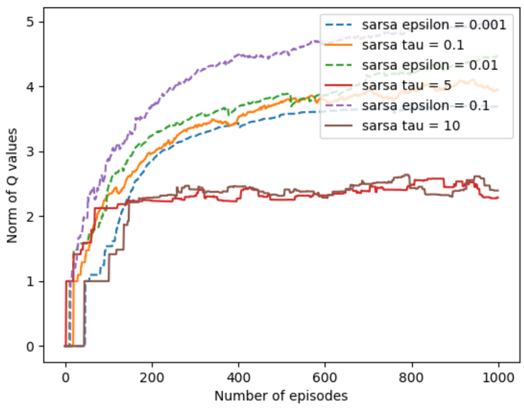
```
    if render:
        # Show the final policy
        mdp.current_state = 0
        mdp.render(q, get_policy_from_q(q))
    return q, q_list

#sarsa with egreedy
def sarsa_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 i in range(nb_episodes):
        print(i)
        # Draw the first state of episode i using a uniform distribution over all the states
        x = mdp.reset(uniform=True)
        ux = 0
        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 egreedy 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] = r
            else:
                uy = egreedy(q, y, epsilon)
                delta = r + mdp.gamma * q[y,uy] - q[x,u]
                q[x, u] = q[x,u] + alpha*delta
            # 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
```

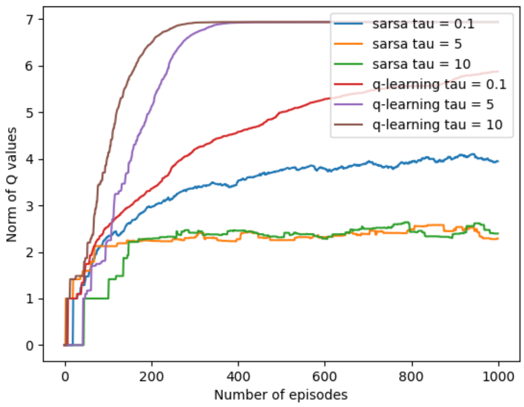
4.4 Comparisons and hyper-parameters

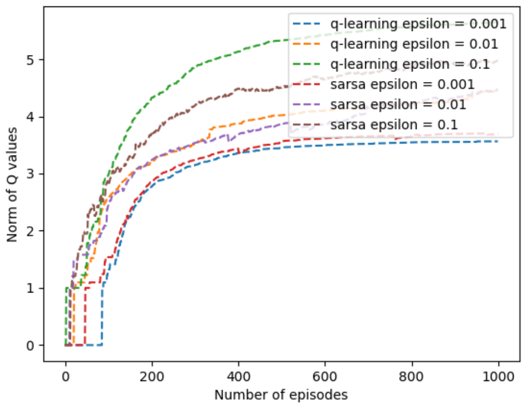
Study question 13

We used the function plot_ql_sarsa with various values of parameters epsilon and tau to get a image of curve below(the test come from the same maze) :









From the first image, we can see that the softmax method has faster convergence than ϵ-greedy, but if the epsilon=0.001, it has the same performance as softmax method. From the secode image, it's clear that softmax is better than ϵ-greedy because the red curve and brown curve have faster convergence, however the performance with epsilon=0.001 is better than with tau=0.1.And the third image, if tau=0.1, we can find that sarsa has faster convergence, and it has the same situation when tau=5 an tau=10. However, in the final image, we can't drive a conclusion because the performance are the same when the parameters of epsilon are the same. Thus, we can drive several conclusions as follows :

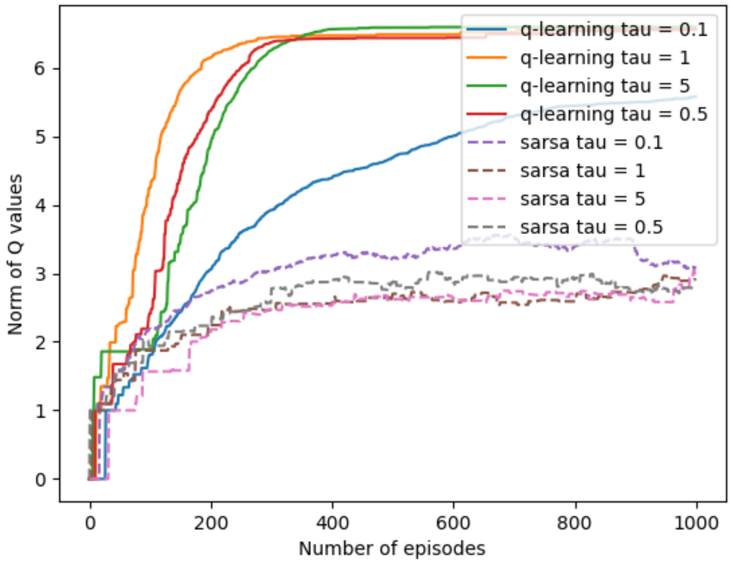
1. The method sarsa has a faster convergence if we use softmax method.

2. The smaller epsilon is, the faster convergence will be.

3. Once the method q-learning has convergence, it's norm of q values won't change unlike the method sarsa.

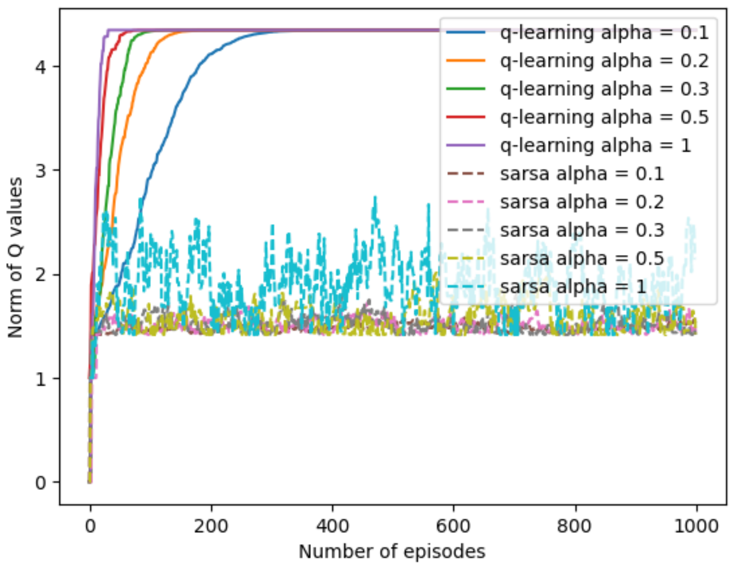
Study question 14

Here is the image of the relation of number of episode and Norm of Q value with parameter τ :



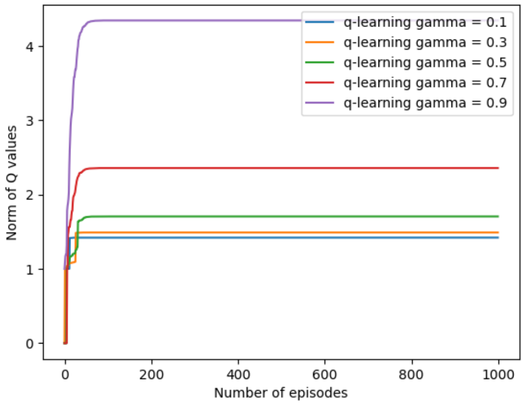
If tau is small, we observed the agent always move to the state where it can have the largest reward, so the agent can finish one episode rapidly, in this image, it represents the agent needs more episode to convergence like the blue curve.If the tau is large,the agent moved more randomly and passed by more state in one episode, so it represent the agent needs less episode to convergence, like the green curve in the image. However we observed that it runs faster if tau is smaller, so it is more efficient if the tau is smaller.

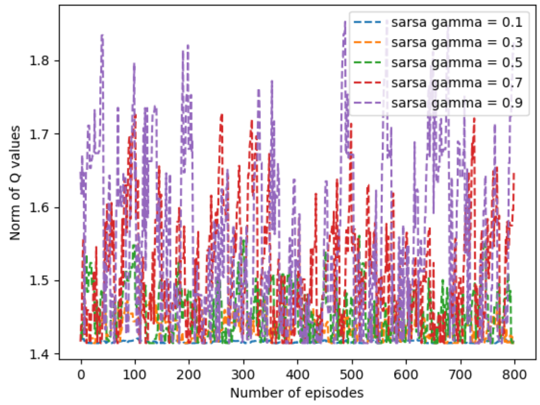
Here is the image of the relation of number of episode and Norm of Q value with parameter α :



If the value of alpha is smaller, the learning speed will be slower, we can see the blue curve has the slowest convergence with using q_learning method in this image. If the value of alpha is greater, the learning speed will be faster, we can see the purple one has the fastest convergence with using q_learning method in this image. Another influence : if the value of alpha is great, the state value changes evidently and the policy changes fast, sometimes we have already got the optimal policy, but it changed sooner, like those dotted curve in the image, their Q values changed fast, thus if the value of alpha is great, the policy is easier to oscillate.

Here is the image of the relation of number of episode and Norm of Q value with parameter γ :





The value of discount factor has an influence of MDP. To the q-learning method, we can see those curve take the same time to have convergence, the difference is that norm of q values is greater if discount factor is greater. And the oscillation is more strongly to the sarsa method if discount factor is greater. We have also taken an observation of the movement of agent, if the discount factor is closer to 0, the agent is more sensitive only to immediate reward, if the discount factor closer to 1, the future rewards are important as immediate reward.