LAB REPORT

MU4IN210 - Robotique et Apprentissage
Reinforcement Learning Labs

Master of Computer Science androide

First year

Academic year 2020 - 2021

Professor : Olivier Sigaud
Students : Vincent Fu
Yuhao Liu

1. Markov Decision Processes and mazes

Code question 1 :

To build the corresponding maze, we need to set up the parameters : height, width which are integers and walls which is a list of cell positions. So, we just need to give the correct positions of the walls, knowing that each cell is numbered from 0 to height $\times$ width - 1 by starting from the very top left position to the bottom of its column and then continuing the numbering to the next nearby column. However be careful, there are not corresponding states for cells which contain a wall.
The piece of code is the following :

```
walls = [7, 8, 9, 10, 21, 27, 30, 31, 32, 33, 45, 46, 47]
height = 6
width = 9
```

Code question 2

In the case of deterministic policy, we have a probability which equals one to reach the next corresponding state. That's why, for each action at each state, we obtain a vector which represents the probabilities to reach each particular state of the maze (so it's a vector of only one element which equals 1 and other are 0). So in the method `step` which takes a particular state and the deterministic action as parameters, we just need to combine all vectors of each action from this particular state into only one vector which contains `proba_success` for the next state provided by the deterministic action and $\frac{1 - \texttt{proba\_success}}{\text{number of possible actions} - 1}$ for the states provided by the remaining actions and 0 for other states. Then, we get a stochastic outcome just by sampling with this discrete distribution vector.
The piece of code is the following :

```
def step(self, u, proba_success, deviation=0):  # performs a step forward in the environment,
    # if you want to add some noise to the reward, give a value to the deviation param
    # which represents the mean μ of the normal distribution used to draw the noise

    noise = deviation*np.random.randn()  # generate noise, useful for RTDP

    # r is the reward of the transition, you can add some noise to it
    reward = self.r[self.current_state, u] + noise

    succ_ind = np.where(self.P[self.current_state, u, :] == 1)[0][0]
    actions = {k for k in self.action_space.actions if k != u}
    equi_inds = np.array([np.where(self.P[self.current_state, a, :] == 1)[0][0] for a in actions])
    equi_proba = (1 - proba_success) / len(equi_inds)
    p = np.zeros(self.nb_states)
    p[succ_ind] = proba_success
    p[equi_inds] = equi_proba
    state = discreteProb(p)

    self.timestep += 1

    info = {"State transition probabilities": p,
            "reward's noise value": noise}  # can be used when debugging

    self.current_state = state
    done = self.done()  # checks if the episode is over

    return [state, reward, done, info]
```

2. Dynamic Programming

Code question 3

By taking inspiration from the `value_iteration_v(mdp)`, the piece of code of `value_iteration_q(mdp)` is the following :

```
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 = 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
```

Code question 4 :

By using the updating policy formula at iteration t : $\pi^{(t+1)}(s) = \arg\max_a Q^{\pi^{(t)}}(s, a)$ where s as state, a as action and Q as Q-value function, we obtain this piece of code :

```
def get_policy_from_q(q):
    # Outputs a policy given the action values
    # TODO: fill this
    return np.array([np.argmax(q[x, :]) for x in range(len(q))])
```

Code question 5 :

By taking inspiration from the `evaluate_one_step_v(mdp, v, policy)` and `evaluate_v(mdp, policy)`, we obtain these pieces of code :

```python
def evaluate_one_step_q(mdp, q, policy):
    # Outputs the state value function after one step of policy evaluation
    # TODO: fill this
    q_new = np.zeros((mdp.nb_states, mdp.action_space.size))
    for x in range(mdp.nb_states):
        if x not in mdp.terminal_states:
            for a in range(mdp.action_space.size):
                summ = 0
                for y in range(mdp.nb_states):
                    summ = summ + mdp.P[x, a, y] * q[y, policy[y]]
                q_new[x, a] = mdp.r[x, a] + mdp.gamma * summ
        else:
            q_new[x, :] = mdp.r[x, :]
    return q_new
```

```python
def evaluate_q(mdp, policy):
    # Outputs the state value function of a policy
    # TODO: fill this
    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)

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

Code question 6 :

By using the concept of policy iteration : at each iteration, we compute the state-value function or the action-value function from the policy π then we improve the policy based on the last evaluation of the state-value function or the action-value function until convergence, we obtain this piece of code :

```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 = 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
```

The result of the final configuration is the following :

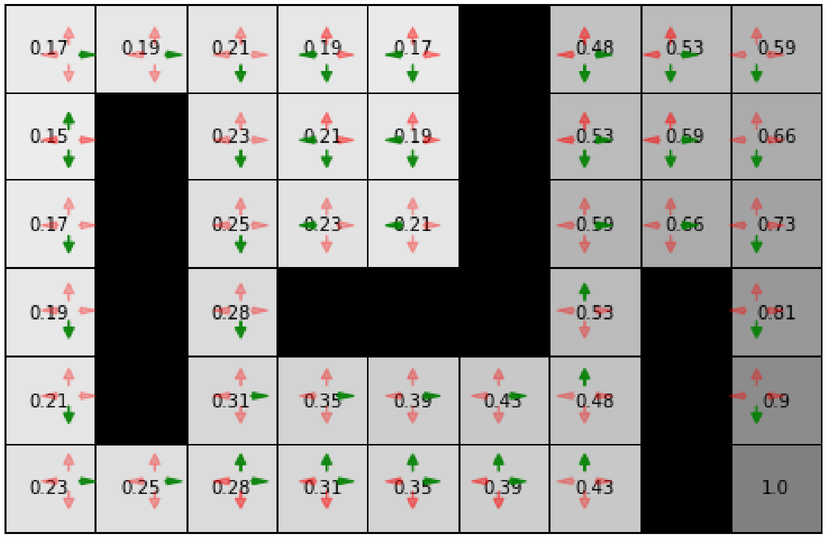


Figure 1: The final maze configuration obtained from policy iteration q

We observe that there is some path marked by the green arrows which is converging towards the higher value of the cells.

Code question 7 :

By using the concept of policy iteration : at each iteration, we compute the state-value function or the action-value function from the policy π then we improve the policy based on the last evaluation of the state-value function or the action-value function until convergence, we obtain this piece of code :

```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
        # TODO: fill this
        v = evaluate_v(mdp, policy)

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

        # 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
```

The result of the final configuration is the following :

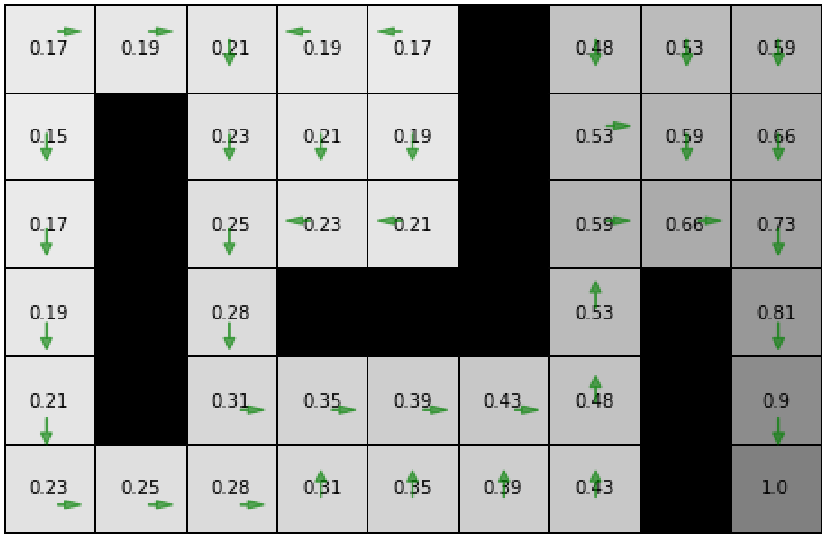


Figure 2: The final maze configuration obtained from policy iteration v

We observe that there is some path marked by the green arrows which is converging towards the higher value of the cells.

Study question 8 :

The results are the following :

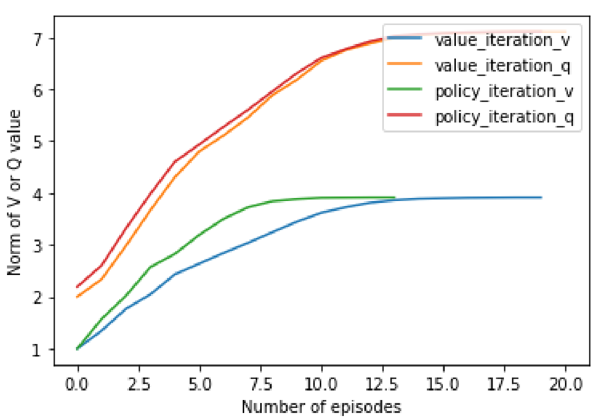


Figure 3: The graph of the Norms of V or Q value according to the number of episodes

```
value iteration V
Time : 5s 766ms
number of iterations = 20
number of Vs = 6400
------------------------
value iteration Q
Time : 10s 632ms
number of iterations = 21
number of Qs = 6720
------------------------
policy iteration Q
Time : 11s 970ms
number of iterations = 20
number of Qs = 82560
------------------------
policy iteration V
Time : 8s 850ms
number of iterations = 14
number of Vs = 14320
```

Figure 4: The Results regarding the numbers of iterations and V or Q values

First of all, we notice that using Q-value function gives a higher norm than using V-value function. It means that using Q-values function gives more information about the environment translated by a larger data storage (values regarding states combined with actions), whereas using V-value function gives less information. However most of the time, because of a large data storage for Q-value function, using V-value function enables a faster computation than using Q-value function due to the convergence comparison (We need to compute the difference between the old Q-value function and the new one. So, saving the current Q-value function in the old one costs a lot.). In terms of convergence, it seems that the value iteration V, the value iteration Q and the policy iteration Q are the same level (We obtain a stable norm at around 12 episodes.). Nevertheless in our case, the policy iteration V has a better convergence than other (convergence at around 7 episodes). That's why, according to what features would we like (fast convergence or fast computation or large information of learning process), some algorithms are preferred.

---

4. Reinforcement learning functions

Code question 9 :

By using the TD(0) equations for the state value function, we obtain this piece of code :

```
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(pol[x], mdp.action_space.size, epsilon=0.2))
            # Update the state value of x
            if x in mdp.terminal_states:
                v[x] = r #TODO: fill this
            else:
                delta = r + mdp.gamma * v[y] - v[x] #TODO: fill this
                v[x] = v[x] + alpha * delta#TODO: fill this

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

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

By using the initial policy given by the function `policy_iteration_q(mdp)`, the result is the following :

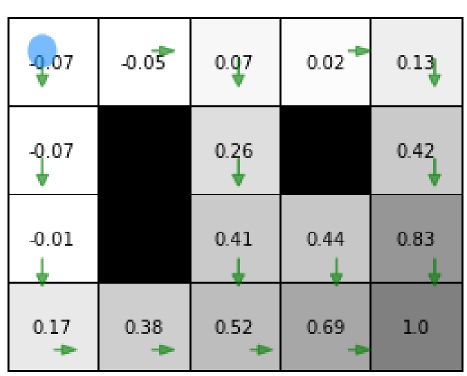


Figure 5: The final maze configuration obtained from temporal difference 0

Code question 10 :

By using the Q-learning equations for an agent exploring an MDP, we obtain this piece of code for action selection based on the softmax policy:

```python
def q_learning(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] = r #TODO: fill this
            else:
                delta = r + mdp.gamma * np.max(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
```

For `tau` $= 6$, the result is the following:

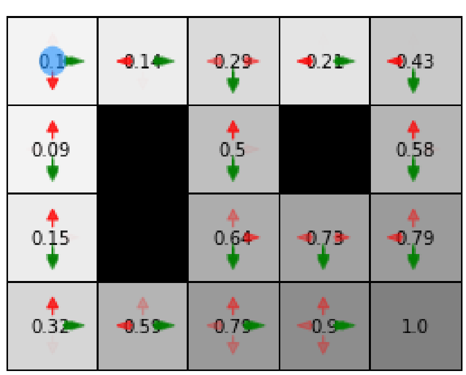


Figure 6: The final maze configuration obtained from q-learning with the softmax policy

Code question 11 :

By taking the same code of `q_learning_soft` and by replacing the softmax policy as an ϵ-greedy policy, we obtain this piece of code :

```
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))

            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 #TODO: fill this
            else:
                delta = r + mdp.gamma * np.max(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
```

Code question 12 :

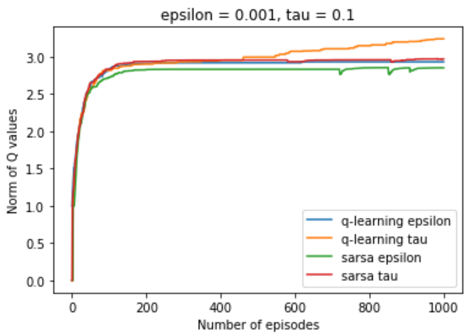
By taking inspiration from the Q-learning functions, we obtain these pieces of code for SARSA :

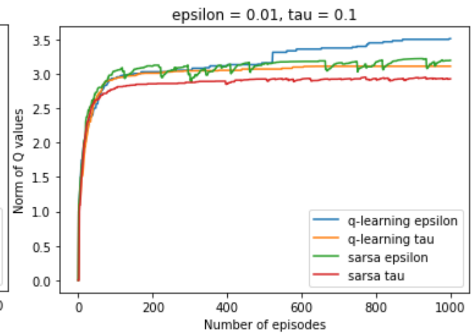
```python
def sarsa_soft(mdp, tau, nb_episodes=20, timeout=50, alpha=0.5, render=True):
    # Initialize the state-action value function
    # alpha is the learning rate
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list =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

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

    if render:
        mdp.new_render()

    for _ in range(nb_episodes):
        # Draw the first state of episode i using a uniform distribution over
        x = mdp.reset(uniform=True)
        done = mdp.done()
        activate = True
        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
            if activate :
                u = mdp.action_space.sample(prob_list=softmax(q, x, tau))
                activate = False

            # 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 #TODO: fill this
            else:
                us = mdp.action_space.sample(prob_list=softmax(q, y, tau))
                delta = r + mdp.gamma * q[y, us] - q[x, u] #TODO: fill this
                q[x, u] = q[x, u] + alpha * delta #TODO: fill this

            # Update the agent position
            x = y
            u = us
        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
```

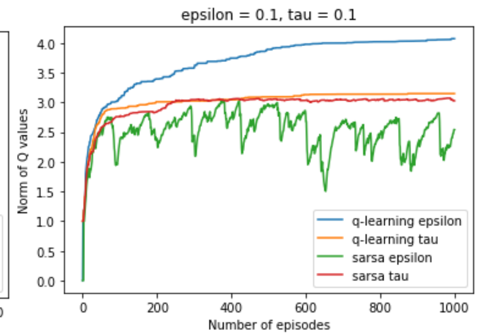
```python
def sarsa_eps(mdp, epsilon, nb_episodes=20, timeout=50, alpha=0.5, render=True):
    # Initialize the state-action value function
    # alpha is the learning rate
    q = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_min = np.zeros((mdp.nb_states, mdp.action_space.size))
    q_list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

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

    if render:
        mdp.new_render()

    for _ 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()
        activate = True
        while not done:
            if render:
                # Show the agent in the maze
                mdp.render(q, q.argmax(axis=1))

            if activate :
                u = egreedy(q, x, epsilon)
                activate = False

            # 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 #TODO: fill this
            else:
                us = egreedy(q, y, epsilon)
                delta = r + mdp.gamma * q[y, us] - q[x, u] #TODO: fill this
                q[x, u] = q[x, u] + alpha * delta #TODO: fill this

            # Update the agent position
            x = y
            u = us
        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
```

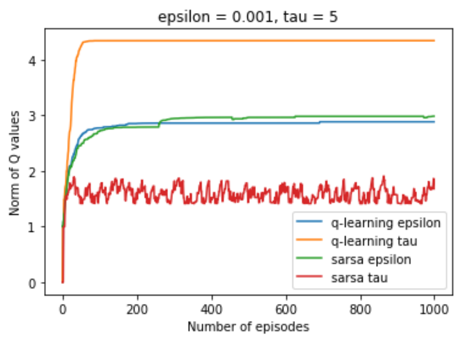
Study question 13 :

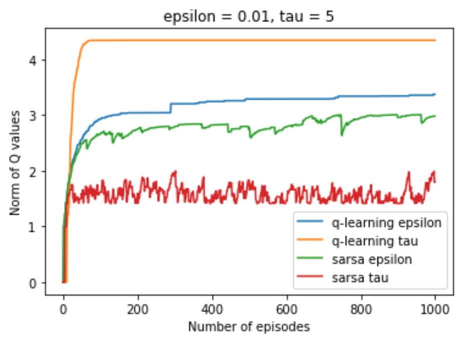
By testing different values for `epsilon` and `tau`, we obtain these results :

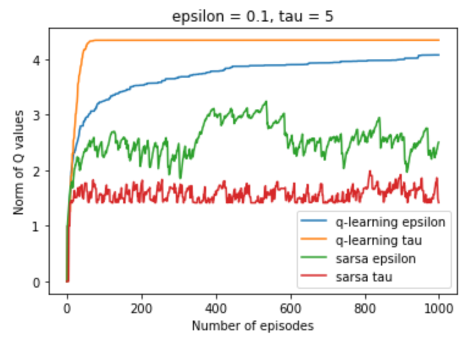


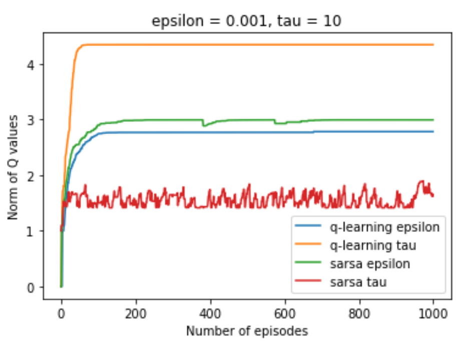


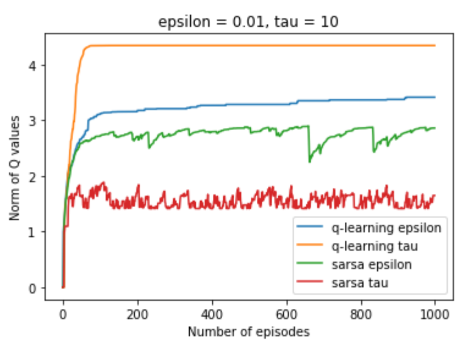


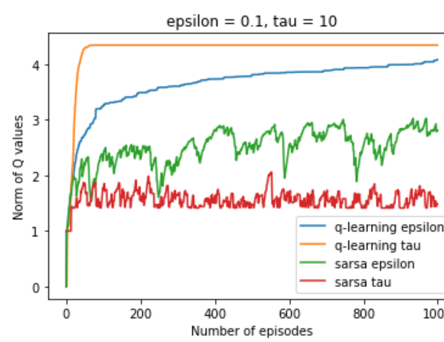








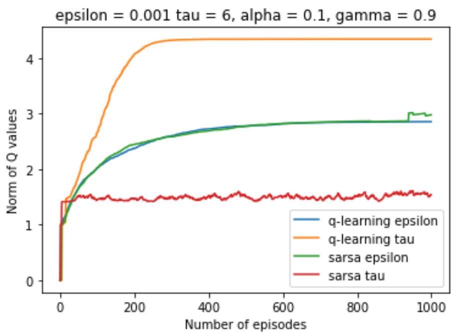


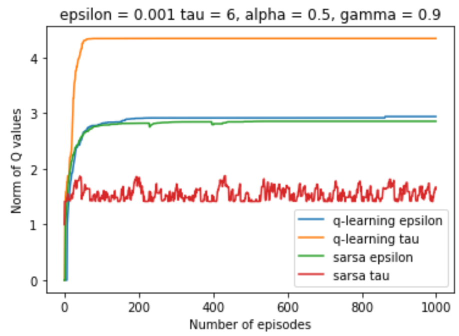


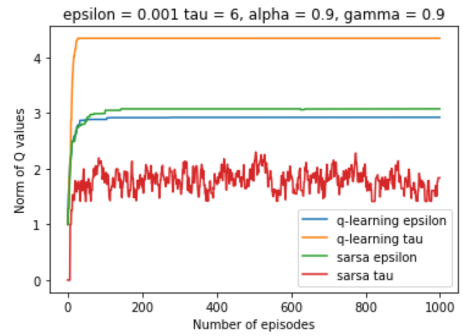
We observe that for $\epsilon = 0.001$ and $\tau = 0.1$, we have the same performance (for convergence and collecting information about the environment) for all algorithms. It means that when τ is low, we get a better discrimination between Q-values for selecting the maximum. As the difference between Q-learning and SARSA is in on-off policy and knowing that in a low τ value case, the selected actions for both algorithms will be the same most of the time, Q-learning and SARSA perform similarly. This is the same idea with a low ϵ. (They converge at around 100 episodes.) Then, when ϵ becomes higher (for example $\epsilon = 0.1$), we observe that Q-learning can learn a little bit more about its environment. So after 100 episodes, the norm of Q-learning is still increasing but slowly. So its convergence is not clear. However, SARSA becomes unstable with a high e-greedy value exploration because SARSA needs to predict the correct next action linked to its current policy to learn well. That's why selecting too many random actions will impact the SARSA's performance. Finaly, when τ becomes higher, Q-learning will learn more too but with convergence (It converges at around 50 episodes) because a higher τ means less discrimination between Q-values. So all actions will be considered more or less the same for rewarding. That's why, we get a random selection like among all good actions. (Less discrimination enables more considered actions during selection because of the softmax operator.) Then, knowing that Q-learning is off-policy so it is not policy dependant, Q-learning is still able to learn well whereas SARSA can not.

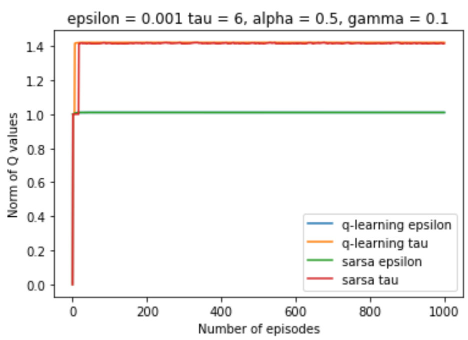
Study question 14 :

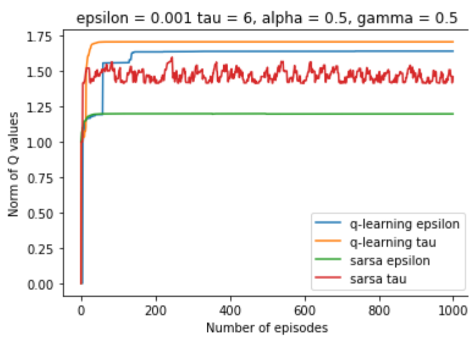
The influence of the parameter τ is said above. So now to analyze the influences of α and γ, we tested differents values for them and the results are the following :

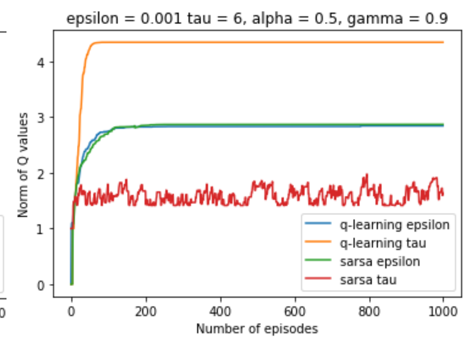












We observe that the learning rate influences on the convergence speed : if the learning rate is low, we need to more episodes to reach the convergence. However, in the case of non efficient learning (like SARSA with softmax policy), a high learning rate will increase noises (on the norm of Q-values) because it will propagate the errors. Then, the discount factor influences on the Q-value propagation efficiency. In fact, according to the importance that we give to the Q-value of the next state, it's important to tune γ in order to get a correct curvature.
