Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF BIOENGINEERING

BITS, BRAINS AND BEHAVIOUR

COURSEWORK 2

Author:
Jorge Jacobo Bennasar Vázquez
CID: 01809233

Question 1: Monte-Carlo Control in Easy21

a) Monte-Carlo algorithm

For the implementation of the Monte-Carlo control an on-policy ε-greedy first-visit algorithm was programmed. This algorithm runs through n episodes an averages the returns obtained for each state-action visited. Being first-visit, the algorithm only takes into account, for each episode, the return obtained from the first visit of each state-action. The state-action value function ($Q(s,a)$) updates following the formula shown below:

$$Q(s,a) \leftarrow \frac{Returns(s,a)}{N(s,a)}$$

$Returns(s,a)$ being the sum of first-visit returns for each state-action and $N(s,a)$ the counter of first-visits. For the Monte-Carlo algorithm, the chosen ε for the soft policy was:

$$\varepsilon = \frac{CE}{CE+N(s)} \quad \text{(GLIE convergence assurance)}$$

Being CE = 10 and $N(s)$ the amount of times that a state is visited. ε is close to one at the beginning, making the algorithm explore different possible paths. As the number of episodes increases, the algorithm turns greedier to obtain a more precise state value function for the observed optimal policy.

b) Learning curve

For twenty runs of n = 100000 episodes the learning curve (sum of rewards) of the Monte-Carlo algorithm with its standard deviation was (Figure 1):

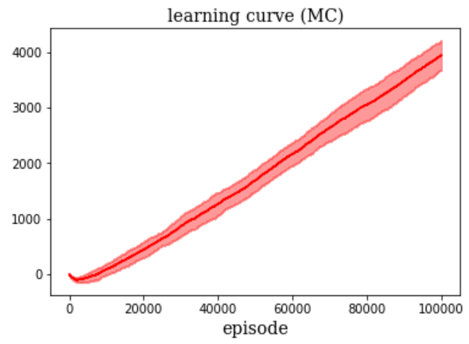


Figure 1. Learning curve (MC)

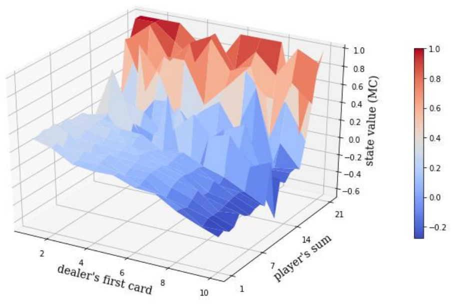


Figure 2. State-value plot (MC)

As it is visible in Figure 1, the algorithm starts having a negative sum of rewards, which means that it loses more than wins. As the episodes run, the algorithm learns how to beat the dealer. This is shown by the curve, which starts to have a positive slope. After that, the sum of rewards turns positive and grows continuously. This also happens with SARSA and Q-Learning, as it is shown in the following sections.

c) State value function

Considering that the states are two-dimensional (player's card sum and dealer's first card), the optimal state-value function ($V^*(s)$) obtained by the algorithm (n =

100000 episodes) can be shown in a 3D plot (see Figure 2, above). The obtained results are intuitive: the algorithm determines that the value of a state has a direct relation with its proximity to the player having a sum of 21.

Question 2: TD Learning in Easy21

a) SARSA algorithm

In the case of TD learning a SARSA ε-greedy algorithm was implemented. Contrary to the Monte-Carlo algorithm, SARSA does not need to retain information about previous episodes with the exception of the last calculated state-action value function ($Q(s,a)$), $N(s)$ and $N(s,a)$ (as defined above, for ε and α determination). For state-action value function determination, SARSA follows the next equation (α being an inertia factor, explained in more detail in section 2.b):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \times (r + Q(s',a') - Q(s,a))$$

Where r is the immediate reward obtained for carrying out action a departing from state s, s' is the next state and a' the next action chosen by an ε-greedy policy. In SARSA ε follows the same pattern as in Monte-Carlo (CE = 10).

b) Learning curve and α dependence

For twenty runs of n = 100000 episodes the learning curve (sum of rewards) of the SARSA algorithm with its standard deviation is shown below (Figure 3):

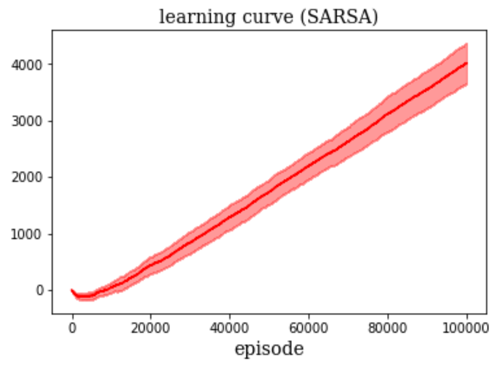


Figure 3. Learning curve (SARSA)

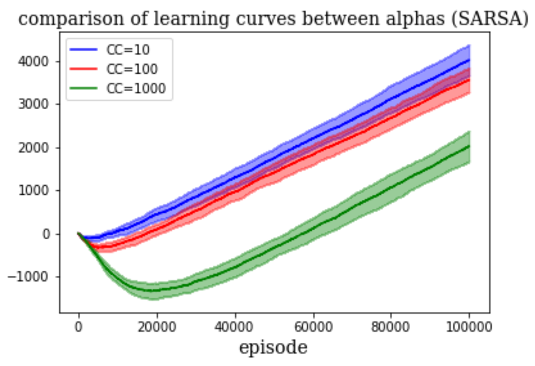


Figure 4. Comparison of learning curves between different α

As stated above, in the SARSA algorithm α acts as an inertia factor which determines how much importance we give to the previous state-action value function ($Q(s,a)$). At the beginning, the state-action value function is imprecise due to the low amount of episodes observed. Because of this, it is convenient to start with a high α (~ 1) and decrease it with the amount of times that we visit each state-action, as shown below:

$$\alpha = \frac{CA}{CA + N(s,a)^2} \quad \text{(Robbins-Monro requirements for convergence)}$$

The value of CA for the learning curve shown in Figure 3 was 10. Later, the effect of α over the results was studied. For that, the learning curves for different values of CA (10, 100 and 1000) were plotted together. As it is observable in Figure 4, CA = 10 and CA = 100 give similar results (having the first one a slight advantage). With CA

= 1000, however, the algorithm starts learning at a slower pace due to a high α, which undermines the importance of reliable values of $Q(s, a)$ during the first 20000 episodes. Considering the slope, *CA* = 1000 could end up giving better results than the other approaches for a greater number of episodes.

c) State value function

The optimal state-value function ($V^*(s)$) obtained by the SARSA algorithm (n = 100000 episodes) is shown in Figure 5. As it is visible, the results are similar to the ones obtained with Monte-Carlo.

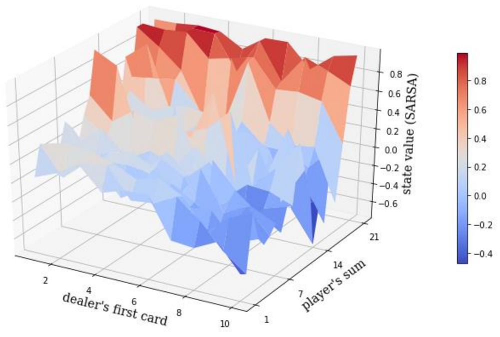


Figure 5. State-value plot (SARSA)

Question 3: Q-Learning in Easy21

a) Q-Learning algorithm

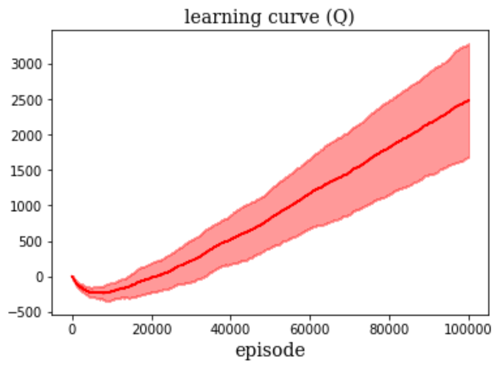
In Q-Learning the approach is similar to SARSA. However, in this case the method is off-policy, which means that the algorithm learns from a different policy than the one followed. This can be observed in the equation shown below:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \times (r + max_a Q(s', a) - Q(s, a))$$

As the formula shows, $Q(s, a)$ is updated without knowledge of the next action a'. In Q-Learning the chosen value for *CA* was always 10.

b) Learning curve and ε dependence

For twenty runs of n = 100000 episodes the learning curve (sum of rewards) of the Q-Learning algorithm with its standard deviation was (Figure 6):



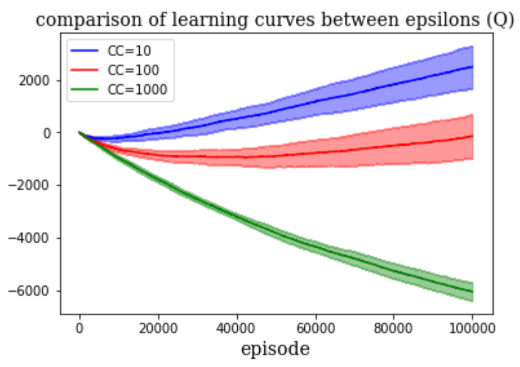


Figure 6. Learning curve (Q) **Figure 7.** Comparison of learning curves between different ε

In Figure 6 *CE* is 10 (as in Monte-Carlo and SARSA). To study the effect of ε over the results obtained in Q-Learning, various learning curves associated with different values of *CE* (10, 100 and 1000) were plotted in Figure 7. It is clear that *CE* = 10 gives the best results, followed by *CE* = 100. On the other hand, *CE* = 1000 gives terrible results. This is because CE exerts too much inertia, making ε too big. Therefore, the algorithm is not greedy enough, and it continuously explores new possibilities without exploiting the best one. When the algorithm turns greedy ($\varepsilon \sim 0$) α is too small, and the values of $Q(s, a)$ are not updated efficiently.

c) State value function

The optimal state-value function ($V^*(s)$) obtained by the Q-Learning algorithm (n = 100000 episodes) is shown in Figure 8. It is interesting that Q-Learning gives a greater value than the other two algorithms to the states in which the player's sum is 11 (see the peaks around the centre of the graph). This can be explained by saying that those states are the only ones in which, by hitting, the player cannot go busted.

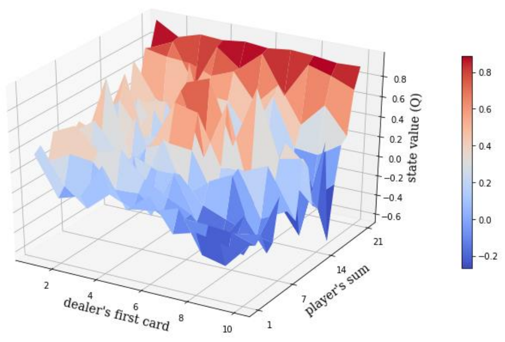


Figure 8. State-value plot (Q)

Question 4: Compare the algorithms

To compare the efficacy of the three algorithms, the learning curves of each approach (CA = CE = 10, n = 100000) were plotted together (Figure 9):

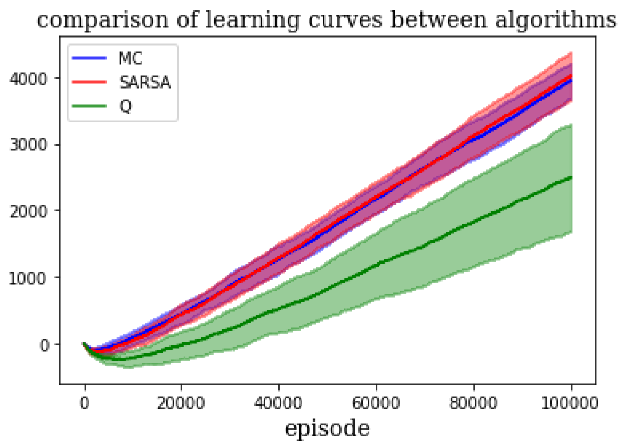


Figure 9. Comparison of learning curves between algorithms

Monte-Carlo and SARSA give similar results, while Q-Learning is the least efficient approach. This is because Q-Learning favours exploration by updating the state-action value function with non-behavioural (target) policies. This is good for open and complex environments. However, for a simple game like Easy 21 on-policy methods, as Monte-Carlo and SARSA, are more efficient.

Appendix: Python code

```
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n_episodes = 100000

font = {'family': 'serif',
        'color':  'black',
        'weight': 'normal',
        'size': 14,
        }

class Easy21:

    def reset(self):
        self.player_first_card_val = np.random.choice(10) + 1
        self.dealer_first_card_val = np.random.choice(10) + 1
        self.player_sum = self.player_first_card_val
        self.dealer_sum = self.dealer_first_card_val
        self.state = [self.dealer_first_card_val, self.player_sum]
        self.player_goes_bust = False
        self.dealer_goes_bust = False
        self.terminal = False

        return self.state

    def step(self, action):
        # action 1: hit   0: stick
        # color: 1: black   -1: red

        r = 0

        if action == 1:
            self.player_card_val = np.random.choice(10) + 1
            self.player_card_col = np.random.choice([-1, 1], p=[1./3., 2./3.])
            self.player_sum += (self.player_card_val * self.player_card_col)
            self.player_goes_bust = self.check_go_bust(self.player_sum)

            if self.player_goes_bust == 1:
                r = -1
                self.terminal = True

        if action == 0:
            self.terminal = True

            while self.dealer_sum < 17 and self.dealer_sum > 0:
                self.dealer_card_val = np.random.choice(10) + 1
                self.dealer_card_col = np.random.choice([-1, 1],
                                                        p=[1./3., 2./3.])
                self.dealer_sum += (self.dealer_card_val*self.dealer_card_col)
                self.dealer_goes_bust = self.check_go_bust(self.dealer_sum)

            if self.dealer_goes_bust == 1: r = 1
            else:
                if self.player_sum > self.dealer_sum: r = 1
                elif self.player_sum < self.dealer_sum: r = -1

        if self.terminal: return 'Terminal', r, self.terminal
        else:
            self.state[1] = self.player_sum
            return self.state, r, self.terminal

    def check_go_bust(self, Sum):
        return bool(Sum > 21 or Sum < 1)

## Monte Carlo -- one episode

def monte_carlo(Q, returns, count_state, count_state_action):
    actions = []
    s = env.reset()
    states = [s]

    while True:
        action_greedy = Q[s[0]-1, s[1]-1, :].argmax()
        count_state[s[0]-1, s[1]-1] += 1
        epsilon = count_eps / float(count_eps +
                                    count_state[s[0]-1, s[1]-1])
        action = np.random.choice([action_greedy, 1 - action_greedy],
                                  p=[1. - epsilon/2., epsilon/2.])
        actions.append(action)

        s, r, term = env.step(action=action)

        if term: break
        else: states.append(s)

    for t in range(len(states)):
        count_state_action[states[t][0]-1, states[t][1]-1, actions[t]] += 1
        returns[states[t][0]-1, states[t][1]-1, actions[t]] += r
        Q[states[t][0]-1, states[t][1]-1, actions[t]] =\
            returns[states[t][0]-1, states[t][1]-1, actions[t]] /\
            count_state_action[states[t][0]-1, states[t][1]-1, actions[t]]

    return Q, returns, count_state, count_state_action, r

## Monte-Carlo

returns_vector_MC = [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                     []]

for i in range(20):

    Q_MC = np.zeros([10, 21, 2]) # Q(s, a)
    returns = np.zeros([10, 21, 2]) # empirical first-visit returns
    count_state_action = np.zeros([10, 21, 2], dtype=int) # N(s, a)
    count_state = np.zeros([10, 21], dtype=int) # N(s)
    count_eps = 10

    env = Easy21()

    for i_epi in range(n_episodes):
        Q_MC, returns, count_state, count_state_action, r =\
            monte_carlo(Q_MC, returns, count_state, count_state_action)

        if i_epi == 0: returns_vector_MC[i].append(r)
        else: returns_vector_MC[i].append(r+returns_vector_MC[i][-1])

    V_MC = Q_MC.max(axis=2)

means_MC = []
stds_MC = []
episodes_MC = []

for i in range(n_episodes):
    x = 0
    y = 0

    for j in range(20):
        x += returns_vector_MC[j][i]
        y += returns_vector_MC[j][i]**2

    means_MC.append(x/20)
    stds_MC.append(np.sqrt((y/20)-(x/20)**2))
    episodes_MC.append(i+1)

min_MC = []
max_MC = []

for i in range(n_episodes):
    min_MC.append(means_MC[i] - stds_MC[i])
    max_MC.append(means_MC[i] + stds_MC[i])

## Monte-Carlo -- plot

plt.figure(1)
plt.plot(episodes_MC,means_MC,color='red')
plt.fill_between(episodes_MC,min_MC,max_MC,alpha=0.4,color='red')
plt.title('learning curve (MC)', fontdict=font)
plt.xlabel('episode', fontdict=font)

s1 = np.arange(10)+1
s2 = np.arange(21)+1
ss1, ss2 = np.meshgrid(s1, s2, indexing='ij')

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(ss1, ss2, V_MC, cmap=cm.coolwarm)

ax.set_xlabel("dealer's first card", fontdict=font)
ax.set_ylabel("player's sum", fontdict=font)
ax.set_zlabel("state value (MC)", fontdict=font)
plt.yticks([1, 5, 10])
plt.yticks([1, 7, 14, 21])
fig.colorbar(surf, shrink=0.6)
fig.tight_layout()

plt.show()

## SARSA -- one episode

def SARSA(Q, count_state, count_state_action, count_alpha):
    s = env.reset()

    count_state[s[0]-1, s[1]-1] += 1
    epsilon = count_eps / float(count_eps + count_state[s[0]-1, s[1]-1])
    action_greedy = Q[s[0]-1, s[1]-1, :].argmax()
    action = np.random.choice([action_greedy, 1 - action_greedy],
                          p=[1. - epsilon/2., epsilon/2.])

    while True:
        next_s, r, term = env.step(action=action)

        count_state_action[s[0]-1, s[1]-1, action] += 1
        alpha = count_alpha / float(count_alpha +
                                    count_state_action[s[0]-1, s[1]-1,
                                                       action]**2)

        if term:
            Q[s[0]-1, s[1]-1, action] += alpha * (r - Q[s[0]-1, s[1]-1,
              action])
            break

        count_state[next_s[0]-1, next_s[1]-1] += 1
        epsilon = count_eps / float(count_eps + count_state[next_s[0]-1,
                                                            next_s[1]-1])
        action_greedy = Q[next_s[0]-1, next_s[1]-1, :].argmax()
        next_action = np.random.choice([action_greedy, 1 - action_greedy],
                                       p=[1. - epsilon/2., epsilon/2.])

        Q[s[0]-1, s[1]-1, action] += alpha * (r +
         Q[next_s[0]-1, next_s[1]-1, next_action] - Q[s[0]-1, s[1]-1, action])

        s = next_s
        action = next_action

    return Q, count_state, count_state_action, r

## SARSA

returns_vector_TD = [[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                      []],[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                      [],[],[]],[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                      [],[],[],[],[]]]

for i in range(3):
    for j in range(20):
        Q_TD = np.zeros([10, 21, 2]) # Q(s, a)
        count_state_action = np.zeros([10, 21, 2], dtype=int) # N(s, a)
        count_state = np.zeros([10, 21], dtype=int) # N(s)
        count_eps = 10
        count_alpha = 10*(10**i)

        env = Easy21()

        for i_epi in range(n_episodes):
            Q_TD, count_state, count_state_action, r =\
                SARSA(Q_TD, count_state, count_state_action, count_alpha)

            if i_epi == 0: returns_vector_TD[i][j].append(r)
            else: returns_vector_TD[i][j].append(r+returns_vector_TD[i][j][-1])

        if i == 0 and j == 0: V_TD = Q_TD.max(axis=2)

means_TD = [[],[],[]]
stds_TD = [[],[],[]]

for i in range(3):
    episodes_TD = []

    for j in range(n_episodes):
        x = 0
        y = 0

        for k in range(20):
            x += returns_vector_TD[i][k][j]
            y += returns_vector_TD[i][k][j]**2

        means_TD[i].append(x/20)
        stds_TD[i].append(np.sqrt((y/20)-(x/20)**2))
        episodes_TD.append(j+1)

min_TD = [[],[],[]]
max_TD = [[],[],[]]

for i in range(3):
    for j in range(n_episodes):
        min_TD[i].append(means_TD[i][j] - stds_TD[i][j])
        max_TD[i].append(means_TD[i][j] + stds_TD[i][j])

## SARSA -- plot

plt.figure(3)
plt.plot(episodes_TD,means_TD[0],color='blue',label='CC=10')
plt.fill_between(episodes_TD,min_TD[0],max_TD[0],alpha=0.4,color='blue')
plt.plot(episodes_TD,means_TD[1],color='red',label='CC=100')
plt.fill_between(episodes_TD,min_TD[1],max_TD[1],alpha=0.4,color='red')
plt.plot(episodes_TD,means_TD[2],color='green',label='CC=1000')
plt.fill_between(episodes_TD,min_TD[2],max_TD[2],alpha=0.4,color='green')
plt.title('comparison of learning curves between alphas (SARSA)',
          fontdict=font)
plt.xlabel('episode', fontdict=font)
plt.legend()

plt.figure(4)
plt.plot(episodes_TD,means_TD[0],color='red')
plt.fill_between(episodes_TD,min_TD[0],max_TD[0],alpha=0.4,color='red')
plt.title('learning curve (SARSA)', fontdict=font)
plt.xlabel('episode', fontdict=font)

s1 = np.arange(10)+1
s2 = np.arange(21)+1
```

```
ss1, ss2 = np.meshgrid(s1, s2, indexing='ij')

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(ss1, ss2, V_TD, cmap=cm.coolwarm)

ax.set_xlabel("dealer's first card", fontdict=font)
ax.set_ylabel("player's sum", fontdict=font)
ax.set_zlabel("state value (SARSA)", fontdict=font)
plt.yticks([1, 5, 10])
plt.yticks([1, 7, 14, 21])
fig.colorbar(surf, shrink=0.6)
fig.tight_layout()

plt.show()

## Q-Learning -- one episode

def Q_Learning(Q, count_state, count_state_action, count_eps):
    s = env.reset()

    while True:
        action_greedy = Q[s[0]-1, s[1]-1, :].argmax()
        count_state[s[0]-1, s[1]-1] += 1
        epsilon = count_eps / float(count_eps + count_state[s[0]-1, s[1]-1])
        action = np.random.choice([action_greedy, 1 - action_greedy],
                                  p=[1. - epsilon/2., epsilon/2.])

        count_state_action[s[0]-1, s[1]-1, action] += 1
        alpha = count_alpha / float(count_alpha +
                                    count_state_action[s[0]-1, s[1]-1,
                                                       action]**2)

        next_s, r, term = env.step(action=action)

        if term:
            Q[s[0]-1, s[1]-1, action] += alpha * (r - Q[s[0]-1, s[1]-1,
              action])
            break

        Q[s[0]-1, s[1]-1, action] += alpha * (r + max(Q[next_s[0]-1,
              next_s[1]-1, :]) - Q[s[0]-1, s[1]-1, action])

        s = next_s

    return Q, count_state, count_state_action, r

## Q-Learning

returns_vector_Q = [[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                     []],[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                     [],[],[]],[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],
                     [],[],[],[],[]]]

for i in range(3):
    for j in range(20):
        Q_Q = np.zeros([10, 21, 2]) # Q(s, a)
        count_state_action = np.zeros([10, 21, 2], dtype=int) # N(s, a)
        count_state = np.zeros([10, 21], dtype=int) # N(s)
        count_eps = 10*(10**i)
        count_alpha = 10

        env = Easy21()

        for i_epi in range(n_episodes):
            Q_Q, count_state, count_state_action, r =\
                Q_Learning(Q_Q, count_state, count_state_action, count_eps)

            if i_epi == 0: returns_vector_Q[i][j].append(r)
            else: returns_vector_Q[i][j].append(r+returns_vector_Q[i][j][-1])
        if i == 0 and j == 0: V_Q = Q_Q.max(axis=2)

means_Q = [[],[],[]]
stds_Q = [[],[],[]]

for i in range(3):
    episodes_Q = []

    for j in range(n_episodes):
        x = 0
        y = 0

        for k in range(20):
            x += returns_vector_Q[i][k][j]
            y += returns_vector_Q[i][k][j]**2

        means_Q[i].append(x/20)
        stds_Q[i].append(np.sqrt((y/20)-(x/20)**2))
        episodes_Q.append(j+1)

min_Q = [[],[],[]]
max_Q = [[],[],[]]

for i in range(3):
    for j in range(n_episodes):
        min_Q[i].append(means_Q[i][j] - stds_Q[i][j])
        max_Q[i].append(means_Q[i][j] + stds_Q[i][j])

## Q-Learning -- plot

plt.figure(6)
plt.plot(episodes_Q,means_Q[0],color='blue',label='CC=10')
plt.fill_between(episodes_Q,min_Q[0],max_Q[0],alpha=0.4,color='blue')
plt.plot(episodes_Q,means_Q[1],color='red',label='CC=100')
plt.fill_between(episodes_Q,min_Q[1],max_Q[1],alpha=0.4,color='red')
plt.plot(episodes_Q,means_Q[2],color='green',label='CC=1000')
plt.fill_between(episodes_Q,min_Q[2],max_Q[2],alpha=0.4,color='green')
plt.title('comparison of learning curves between epsilons (Q)', fontdict=font)
plt.xlabel('episode', fontdict=font)
plt.legend()

plt.figure(7)
plt.plot(episodes_Q,means_Q[0],color='red')
plt.fill_between(episodes_Q,min_Q[0],max_Q[0],alpha=0.4,color='red')
plt.title('learning curve (Q)', fontdict=font)
plt.xlabel('episode', fontdict=font)

s1 = np.arange(10)+1
s2 = np.arange(21)+1
ss1, ss2 = np.meshgrid(s1, s2, indexing='ij')

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(ss1, ss2, V_Q, cmap=cm.coolwarm)

ax.set_xlabel("dealer's first card", fontdict=font)
ax.set_ylabel("player's sum", fontdict=font)
ax.set_zlabel("state value (Q)", fontdict=font)
plt.yticks([1, 5, 10])
plt.yticks([1, 7, 14, 21])
fig.colorbar(surf, shrink=0.6)
fig.tight_layout()

plt.show()

# Comparison Monte-Carlo - SARSA - Q-Learning

# For alpha = 10/(10+N(s,a)**2) and epsilon = 10/(10+N(s))

plt.figure(9)
plt.plot(episodes_MC,means_MC,color='blue',label='MC')
plt.fill_between(episodes_MC,min_MC,max_MC,alpha=0.4,color='blue')
plt.plot(episodes_TD,means_TD[0],color='red',label='SARSA')
plt.fill_between(episodes_TD,min_TD[0],max_TD[0],alpha=0.4,color='red')
plt.plot(episodes_Q,means_Q[0],color='green',label='Q')
plt.fill_between(episodes_Q,min_Q[0],max_Q[0],alpha=0.4,color='green')
plt.title('comparison of learning curves between algorithms', fontdict=font)
plt.xlabel('episode', fontdict=font)
plt.legend()
```