Защищено:
Гапанюк Ю.Е.

"__"_____2024 г.

Демонстрация:
Гапанюк Ю.Е.

"__"_____2024 г.

Отчет по лабораторной работе № 5 по курсу Методы машинного обучения

Тема работы: " Реализация алгоритма Policy Iteration "

10
(количество листов)
Вариант № **15**

ИСПОЛНИТЕЛЬ:

студент группы ИУ5-22М _____
(подпись)

Чиварзин А.Е.

"__"_____2024 г.

Москва, МГТУ - 2024

Задание

1. На основе рассмотренного на лекции примера реализуйте следующие алгоритмы:
    a. SARSA
    b. Q-обучение
    c. Двойное Q-обучение

    для любой среды обучения с подкреплением (кроме рассмотренной на лекции среды Toy Text / Frozen Lake) из библиотеки Gym (или аналогичной библиотеки).
2. Сформировать отчет и разместить его в своем репозитории на github.

Ход выполнения работы

Для реализации была выбрана среда Taxi-v3 из библиотеки Gym.

По документации: 500 состояний - 5*5 карта, 4 возможных локации точки выхода, 5 состояний пассажира (4 выхода и в такси).

6 действий - 4 движения и взять/высадить пассажира.

Из 500 состояний в рамках 1 итерации достижимо 400 - исключаются состояния где пассажир там же, где и здание.

Код программы:

```
import numpy as np
import matplotlib.pyplot as plt
import gym
from tqdm import tqdm

# ********************************** БАЗОВЫЙ АГЕНТ
**************************************

class BasicAgent:
    '''
    Базовый агент, от которого наследуются стратегии обучения
    '''

    # Наименование алгоритма
    ALGO_NAME = '---'

    def __init__(self, env, eps=0.1):
        # Среда
        self.env = env
        # Размерности Q-матрицы
        self.nA = env.action_space.n
        self.nS = env.observation_space.n
        #и сама матрица
        self.Q = np.zeros((self.nS, self.nA))
        # Значения коэффициентов
        # Порог выбора случайного действия
        self.eps=eps
```

```
        # Награды по эпизодам
        self.episodes_reward = []


    def print_q(self):
        print('Вывод Q-матрицы для алгоритма ', self.ALGO_NAME)
        print(self.Q)


    def get_state(self, state):
        '''
        Возвращает правильное начальное состояние
        '''
        if type(state) is tuple:
            # Если состояние вернулось с виде кортежа, то вернуть только номер состояния
            return state[0]
        else:
            return state


    def greedy(self, state):
        '''
        <<Жадное>> текущее действие
        Возвращает действие, соответствующее максимальному Q-значению
        для состояния state
        '''
        return np.argmax(self.Q[state])


    def make_action(self, state):
        '''
        Выбор действия агентом
        '''
        if np.random.uniform(0,1) < self.eps:

            # Если вероятность меньше eps
            # то выбирается случайное действие
            return self.env.action_space.sample()
        else:
            # иначе действие, соответствующее максимальному Q-значению
            return self.greedy(state)


    def draw_episodes_reward(self):
        # Построение графика наград по эпизодам
        fig, ax = plt.subplots(figsize = (15,10))
        y = self.episodes_reward
        x = list(range(1, len(y)+1))
        plt.plot(x, y, '-', linewidth=1, color='green')
        plt.title('Награды по эпизодам')
        plt.xlabel('Номер эпизода')
        plt.ylabel('Награда')
        plt.show()


    def learn():
        '''
        Реализация алгоритма обучения
        '''
        pass

# ********************************** SARSA ****************************************

class SARSA_Agent(BasicAgent):
    '''
    Реализация алгоритма SARSA
    '''
    # Наименование алгоритма
    ALGO_NAME = 'SARSA'
```

```
    def __init__(self, env, eps=0.4, lr=0.1, gamma=0.98, num_episodes=20000):
        # Вызов конструктора верхнего уровня
        super().__init__(env, eps)
        # Learning rate
        self.lr=lr
        # Коэффициент дисконтирования
        self.gamma = gamma
        # Количество эпизодов
        self.num_episodes=num_episodes
        # Постепенное уменьшение eps
        self.eps_decay=0.00005
        self.eps_threshold=0.01


    def learn(self):
        '''
        Обучение на основе алгоритма SARSA
        '''
        self.episodes_reward = []
        # Цикл по эпизодам
        for ep in tqdm(list(range(self.num_episodes))):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
            truncated = False
            # Суммарная награда по эпизоду
            tot_rew = 0

            # По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора
действия
            if self.eps > self.eps_threshold:
                self.eps -= self.eps_decay

            # Выбор действия
            action = self.make_action(state)

            # Проигрывание одного эпизода до финального состояния
            while not (done or truncated):

                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                # Выполняем следующее действие
                next_action = self.make_action(next_state)

                # Правило обновления Q для SARSA
                self.Q[state][action] = self.Q[state][action] + self.lr * \
                    (rew + self.gamma * self.Q[next_state][next_action] -
self.Q[state][action])

                # Следующее состояние считаем текущим
                state = next_state
                action = next_action
                # Суммарная награда за эпизод
                tot_rew += rew
                if (done or truncated):
                    self.episodes_reward.append(tot_rew)

# ********************************** Q-обучение
*************************************

class QLearning_Agent(BasicAgent):
    '''
    Реализация алгоритма Q-Learning
    '''
```

```
    # Наименование алгоритма
    ALGO_NAME = 'Q-обучение'


    def __init__(self, env, eps=0.4, lr=0.1, gamma=0.98, num_episodes=20000):
        # Вызов конструктора верхнего уровня
        super().__init__(env, eps)
        # Learning rate
        self.lr=lr
        # Коэффициент дисконтирования
        self.gamma = gamma
        # Количество эпизодов
        self.num_episodes=num_episodes
        # Постепенное уменьшение eps
        self.eps_decay=0.00005
        self.eps_threshold=0.01


    def learn(self):
        '''
        Обучение на основе алгоритма Q-Learning
        '''
        self.episodes_reward = []
        # Цикл по эпизодам
        for ep in tqdm(list(range(self.num_episodes))):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
            truncated = False
            # Суммарная награда по эпизоду
            tot_rew = 0

            # По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора
действия
            if self.eps > self.eps_threshold:
                self.eps -= self.eps_decay

            # Проигрывание одного эпизода до финального состояния
            while not (done or truncated):

                # Выбор действия
                # В SARSA следующее действие выбиралось после шага в среде
                action = self.make_action(state)

                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                # Правило обновления Q для SARSA (для сравнения)
                # self.Q[state][action] = self.Q[state][action] + self.lr * \
                #     (rew + self.gamma * self.Q[next_state][next_action] -
self.Q[state][action])

                # Правило обновления для Q-обучения
                self.Q[state][action] = self.Q[state][action] + self.lr * \
                    (rew + self.gamma * np.max(self.Q[next_state]) -
self.Q[state][action])

                # Следующее состояние считаем текущим
                state = next_state
                # Суммарная награда за эпизод
                tot_rew += rew
                if (done or truncated):
                    self.episodes_reward.append(tot_rew)

# ********************************** Двойное Q-обучение
*************************************
```

```
class DoubleQLearning_Agent(BasicAgent):
    '''
    Реализация алгоритма Double Q-Learning
    '''
    # Наименование алгоритма
    ALGO_NAME = 'Двойное Q-обучение'


    def __init__(self, env, eps=0.4, lr=0.1, gamma=0.98, num_episodes=20000):
        # Вызов конструктора верхнего уровня
        super().__init__(env, eps)
        # Вторая матрица
        self.Q2 = np.zeros((self.nS, self.nA))
        # Learning rate
        self.lr=lr
        # Коэффициент дисконтирования
        self.gamma = gamma
        # Количество эпизодов
        self.num_episodes=num_episodes
        # Постепенное уменьшение eps
        self.eps_decay=0.00005
        self.eps_threshold=0.01


    def greedy(self, state):
        '''
        <<Жадное>> текущее действие
        Возвращает действие, соответствующее максимальному Q-значению
        для состояния state
        '''
        temp_q = self.Q[state] + self.Q2[state]
        return np.argmax(temp_q)


    def print_q(self):
        print('Вывод Q-матриц для алгоритма ', self.ALGO_NAME)
        print('Q1')
        print(self.Q)
        print('Q2')
        print(self.Q2)


    def learn(self):
        '''
        Обучение на основе алгоритма Double Q-Learning
        '''
        self.episodes_reward = []
        # Цикл по эпизодам
        for ep in tqdm(list(range(self.num_episodes))):
            # Начальное состояние среды
            state = self.get_state(self.env.reset())
            # Флаг штатного завершения эпизода
            done = False
            # Флаг нештатного завершения эпизода
            truncated = False
            # Суммарная награда по эпизоду
            tot_rew = 0

            # По мере заполнения Q-матрицы уменьшаем вероятность случайного выбора
действия
            if self.eps > self.eps_threshold:
                self.eps -= self.eps_decay

            # Проигрывание одного эпизода до финального состояния
            while not (done or truncated):

                # Выбор действия
                # В SARSA следующее действие выбиралось после шага в среде
                action = self.make_action(state)
```

```
                # Выполняем шаг в среде
                next_state, rew, done, truncated, _ = self.env.step(action)

                if np.random.rand() < 0.5:
                    # Обновление первой таблицы
                    self.Q[state][action] = self.Q[state][action] + self.lr * \
                        (rew + self.gamma *
self.Q2[next_state][np.argmax(self.Q[next_state])] - self.Q[state][action])
                else:
                    # Обновление второй таблицы
                    self.Q2[state][action] = self.Q2[state][action] + self.lr * \
                        (rew + self.gamma *
self.Q[next_state][np.argmax(self.Q2[next_state])] - self.Q2[state][action])

                # Следующее состояние считаем текущим
                state = next_state
                # Суммарная награда за эпизод
                tot_rew += rew
                if (done or truncated):
                    self.episodes_reward.append(tot_rew)


def play_agent(agent):
    '''
    Проигрывание сессии для обученного агента
    '''
    env2 = gym.make('Taxi-v3',render_mode='human')
    state = env2.reset()[0]
    done = False
    while not done:
        action = agent.greedy(state)
        next_state, reward, terminated, truncated, _ = env2.step(action)
        env2.render()
        state = next_state
        if terminated or truncated:
            done = True


def run_sarsa():
    env = gym.make('Taxi-v3')
    agent = SARSA_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


def run_q_learning():
    env = gym.make('Taxi-v3')
    agent = QLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


def run_double_q_learning():
    env = gym.make('Taxi-v3')
    agent = DoubleQLearning_Agent(env)
    agent.learn()
    agent.print_q()
    agent.draw_episodes_reward()
    play_agent(agent)


def main():
    #run_sarsa()
    #run_q_learning()
```

```
    run_double_q_learning()

if __name__ == '__main__':
    main()
```

Вывод программы модифицирован для кодировки состояний: см. рис. 2.

Результаты выполнения:

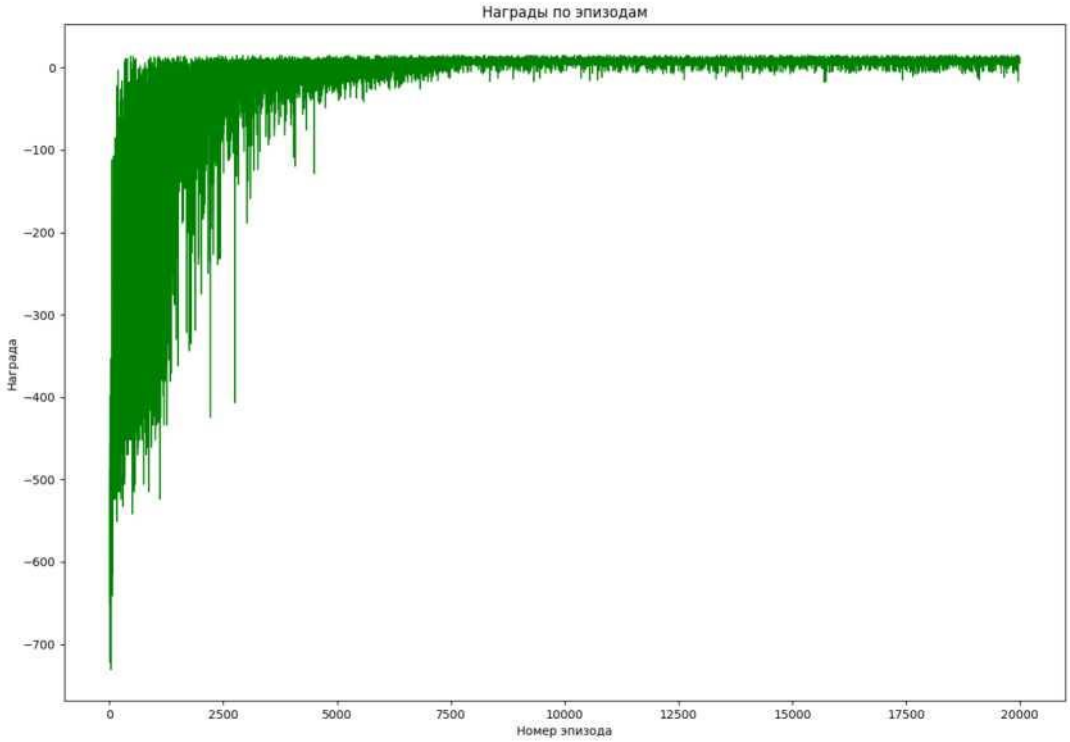

Рис. 1 - Награды по этапам SARSA.

```
Вывод Q-матрицы для алгоритма  SARSA
[[  0.          0.          0.          0.          0.
    0.        ]
 [ -6.59510594  -1.96007596  -6.99278993  -3.69658561   7.74183855
  -12.12066264]
 [  2.13952829   1.89041573   1.13819875   3.13294     13.03717386
   -4.96973917]
 ...
 [  4.91965448  14.53492432   4.65312868  -1.16372383  -2.90301868
   -5.06439372]
 [ -8.56123083  -4.78752598  -8.43412478  -8.38525565 -13.8541778
  -13.16174878]
```

Рис. 2 - Q-матрица SARSA.

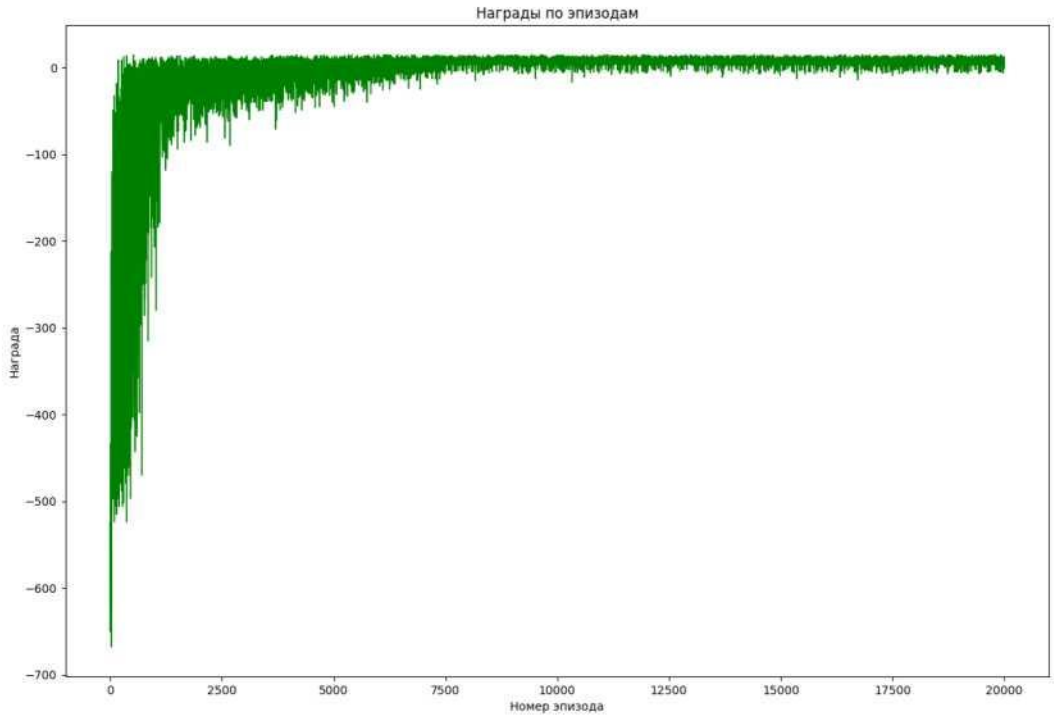

Рис. 3 - Награды по этапам Q-обучение.

```
Вывод Q-MaTpni+1 для алгоритма Q-обучение
[[ 0.         0.         0.         0.         0.         0.        ]
 [ 4.2B617674 5.03910366 3.46309001 6.65702923 0.36234335 -3.00320347] [10.17438811 10.11231643
 10.03753837 11.07922912 13.27445573 2.57430503]
 ...
 [-1.23264741 12.73513532 0.49390431 0.02753999 -4.90913303 -4.93051469] [-1.33069913 -0.36090479
 -2.5239913                          3.39620236 -6.34337216 -9.82047501]
 [ 6.69009335 1.40696151 5.34079492 13.59393093 4.19331129 1.4700502 ]] I
```

Рис. 4 - Q-матрица Q-обучения.

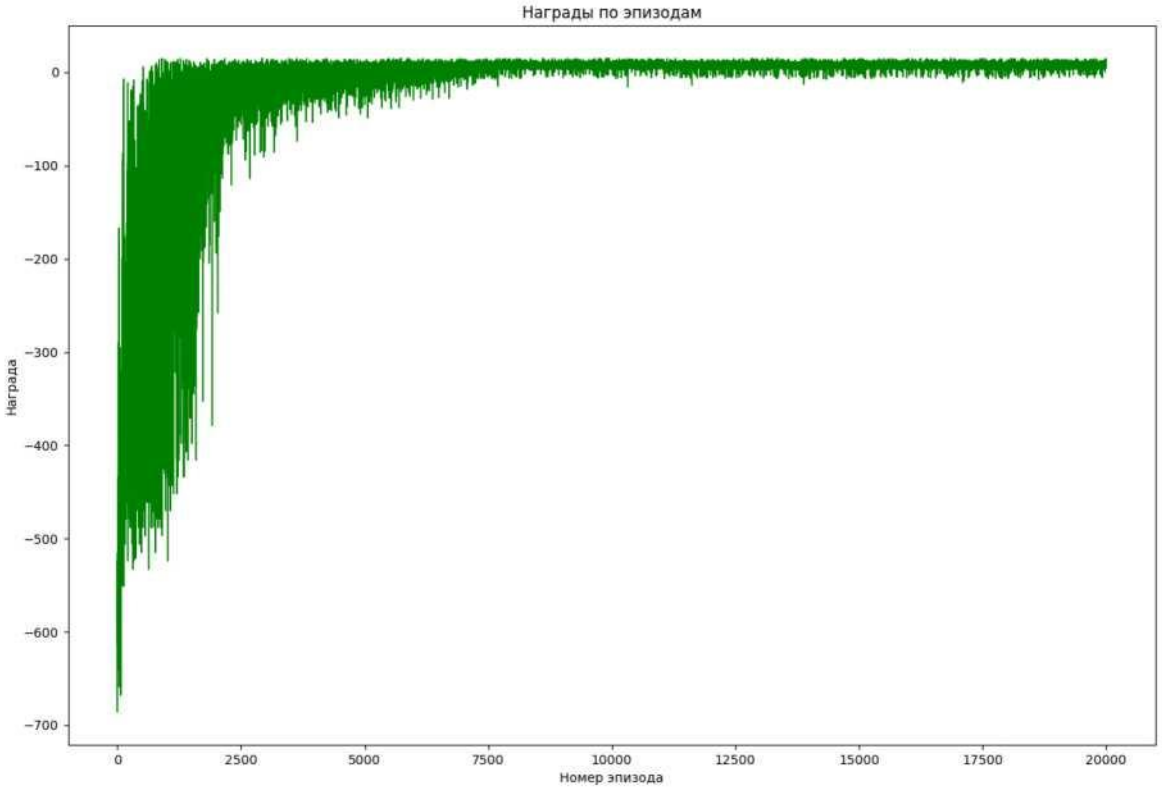

Рис. 5 - Награды по этапам dQ-обучение.

```
Вывод Q-матриц для алгоритма Двойное Q-обучение
Q1
[[ 0.          0.          0.          0.          0.
   0-       1
[  0.4956301   2.01616935  -0.37676916  -           3.36234335
  -4.40342241]                        1.03059973
[  5.58463476  7.83243378   1.08344404              13.27445578
  -2.14348329]                          6.9493195
                                      6
                                      -
[ -1.41111304 10.63493294  -1.7376361                -3.02222303
  -2.29353635]
[ -4.03353292 -4.33230483  -4.55711092   2.7093557 -10.15945533
  -9.65504553]
[ 1.40885897   1.98958961   3.3739806   18.4776031   0.65083876
   0.63652395]]
Q2
[[ 0.          0.          .                     .          0-       ]
[-0.3270464   3.49955587 -  .33440933   .05364569   36234335 -
[ 1.33739738 6.33973543 2.  .3069611    .13565323   27445573 -
                                    7 13 .          0 337340561
                            .22470337     33534724 -72093994 -3.7601113
[-1.49141743 7.32475969 -1.0.                     5.]
          [-4.57704201 -4.06435558 -4.26295557 -0.64435005 -9.27489379 -6.2435429 ]
[ 1.33162216 4.41925175 5.21988787 13.33312556 -0.67761377 0.77036741]]
          п
```

Рис. 6 - Q-матрицы dQ-обучение.

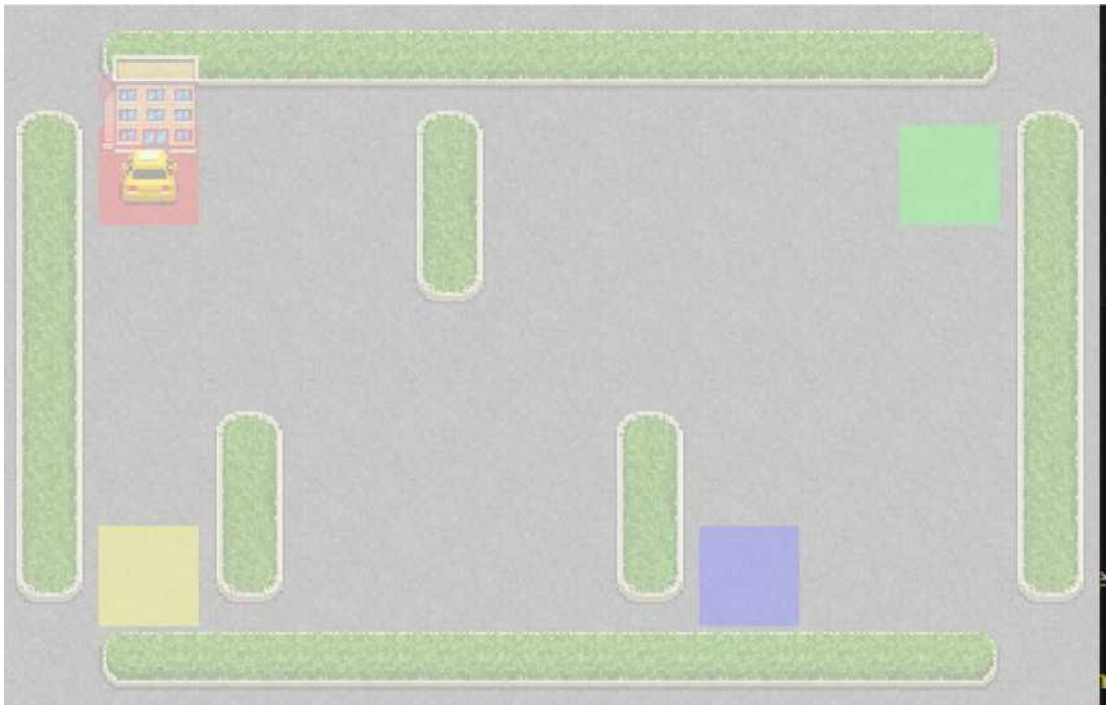

Рис. 7 - конечное состояние.

Вывод:

В ходе выполнения лабораторной работы мы ознакомились с базовыми методами обучения с подкреплением на основе временных различий.

Метод SARSA оказался самым быстрым, 2000 итераций в сек. Против 1900 и 1700 итераций в Q и dQ обучении. Вероятно, это связано с max() в Q и появлением промахов кэша в dQ.

Для настолько простой симуляции все алгоритмы имеют похожее время схождения, но Q и dQ имеют больший темп начального схождения.