ECSE 6440/MANE 6620: Optimal Control Theory

Final Project: Control of room cleaning robot using reinforcement learning

Authors:
Shaunak Basu
Talgat Alimbayev

Instructor:
Dr. Agung Julius

Spring 2020

Rensselaer Polytechnic Institute

1 Introduction

It is very common to use system modelling or system identification techniques to decide on the set of actions for a given system in an environment. However, when no model is available or it is hard to obtain, reinforcement learning techniques can be used to determine the optimal set of actions. In this project we focused on obtaining the optimal control policy for navigating the room cleaning robot in a world. The comparison will be made between SARSA and Q-learning algorithms

2 Literature review

For this project, we drew on the Bender example in the homework 4 [1] and famous travelling salesman problem [2]. In the homework, we were asked to find an optimal policy for a robot Bender to navigate on a 3×3 grid. Bender could find a treasure or meet a bandit and loose the treasure. Bender is rewarded every time it returns with the treasure to the base cell. The robot could move North, East, South and West. In the famous travelling salesman problem, a salesman has to visit every city from a set of cities by travelling the minimum distance. The cities cannot be revisited, not all of the cities are interconnected and the salesman has to return to the starting city [2]. Having described these two examples, we would like to move on to describing the room robot cleaning robot problem.

3 Problem Statement

A room cleaning robot starts from the cell 0 as shown in Figure 4.

0 start	1	2
3	4	5
6	7	8

Figure 1: Cells conventions with 0 as the starting cell

To complete cleaning a room the robot has to visit all of the cells. The order of visiting the cells is up to the algorithm to decide. Numbering convention for the movements is shown in Figure 2. The allowed movements for each cell and the cost of the associated movements is shown below in Figure 3.

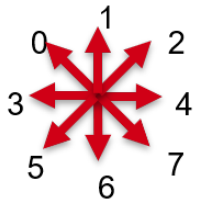

Figure 2: Allowed moves numbering

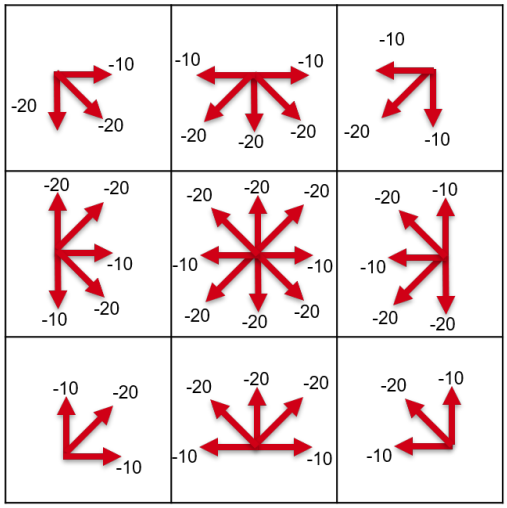

Figure 3: Allowed movements and their costs

The robot is rewarded for visiting unvisited cells and penalized for revisiting cells. The objective of the robot thus becomes to visit all of the cells with the minimum number of revisits.

In this project, we also want to compare SARSA and Q-learning algorithms.

4 Theory

The setup for learning can be visualized as follows:

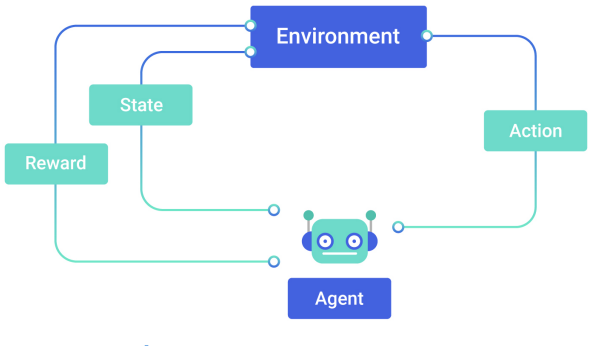

Figure 4: Setup for learning

An agent is placed in an environment with a pre-defined goal. The agent is allowed to perform a set of actions on the environment to reach its goal. The environment responds to the agent's actions and based on the feedback it receives from the environment, the agent is rewarded. The value of the reward received by the agent is dependent on the goal to be achieved. For our problem, the time space in which the agent performs its action is discrete, i.e., the agent performs an action at a time instant t, where $t = 0, 1, 2, ..., n_t$, where n_t is the final time.

Markov Decision Processes (MDPs) are used to formulate Reinforcement learning methods. As we have a finite set of states moving in a discrete time space, our problem is formulated using a Markov Chain [3]. As the name suggests, an MDP satisfies the Markov Property, i.e., the future states of the process is only dependent on the current state. Therefore information regarding the past states is stored in the current state. An MDP is formulated as a set of states(denoted as X), a set of possible actions(denoted as A), a real valued reward function(denoted as R) and a transition matrix(T) which denotes the probability of transitioning from one state to another, given a particular action at a state.

A policy π is a probability distribution of the actions taken by the agent, given the states. That is the likelihood of every action when an agent is in a particular state. This policy becomes the Markov Chain when implemented on an MDP [1]. We consider stationary policies for our problem, where the policy is independent of the time. The only dependency of the policy is on the state of the agent. In this case, the transition probability from a state $x \in X$, to a state $x^{'} \in X$ is given by:

$$P^{\pi}_{x,x'} = \sum_{u \in A} P(u|x)P(x'|u,x), \tag{1}$$

where $u \in A$, is the action taken at state x. Let us denote the reward obtained at a particular state as r. The reward at a particular state is dependent on the future rewards over the policy taken. At a time step t, the reward can be expressed as: The two reinforcement learning methods that we explore, SARSA and Q-learning are temporal difference learning methods. Temporal difference learning builds from the features of Monte Carlo methods and Dynamic Programming. Like Monte Carlo methods, it uses the function value calculated for each state in the previous episode, for the update in the current episode i.e.,

$$J_t = r_t + r_{t+1} + r_{t+2}..., \tag{2}$$

A discount factor, γ, is introduced to account for the relative importance of future rewards. We can reformulate the above equation as:

$$J_t = r_t + \gamma J_{t+1}, \tag{3}$$

The objective of learning is to maximize the expected reward over the policy taken. This can be expressed as:

$$J^{\pi} = \sum_{k=0}^{t} E_{\pi}(r(k))\gamma^k, \tag{4}$$

If we define the optimal reward as,

$$V_k(x) = max_{\pi} J^{\pi}_t(x), \tag{5}$$

we can formulate the Bellman Equation as :

$$V(x) = max_u \sum_{x' \in X} P^u_{x,x'}(r^u_{x,x'} + \gamma V(x')), \tag{6}$$

The two reinforcement learning methods that we explore, SARSA and Q-learning are temporal difference learning methods. Temporal difference learning builds from the features of Monte Carlo methods and Dynamic Programming. Like Monte Carlo methods, it uses the function value calculated for each state in the previous episode, for the update in the current episode i.e.,

$$V(x_t) \leftarrow V(x_t) + \alpha(J_t - V(x_t)), \tag{7}$$

but unlike Monte Carlo, the updates are made in the next time step, as the value of $V(x_t)$ is dependent on $V(x_{t+1})$. This feature is similar to Dynamic Programming where the update is made as follows:

$$V(x_t) \leftarrow V(x_t) + \alpha(r_{t+1} + \gamma V(x_{t+1}) - V(x_t)), \tag{8}$$

In the next section, the update step for each algorithm is presented.

5 Problem Formulation

In our problem, our state space is finite and we denote these set of states as X. For a grid(world) size of 3×3, each of the states in our state space can be denoted as $(cell\_number, visited)$, where $cell\_number = 1, 2, 3, ..., 9$ and $visited = 0, 1$. Hence if the 3^{rd} cell in the grid is visited, we can represent that state as $(3, 1)$ and if the cell is unvisited, we can represent that state as $(3, 0)$. We denote our set of possible actions as the set A. Therefore for our problem:

$$\begin{aligned} X &= \{(i, j)\} \qquad \text{where j = 0,1 and i =1,2,3,...,9,} \\ A &= \{\text{North, North-East, East, South-East,South,South-West, West, North-West, }\} \end{aligned} \tag{9}$$

5.1 SARSA

The SARSA algorithm is shown below. It is an on policy algorithm where the agent needs to evaluate the reward for the action chosen at the next state, in order to update the reward for the current state [4].

Algorithm 1: SARSA

Initialize the Q table:
Initialize $\boldsymbol{Q(x, u)}$,$\forall$ x $\in$ X, $u \in$ A(x) arbitrarily
for *each episode* **do**
  Initialize **x**;
  Choose **u** from A(**x**) using ϵ-greedy;
  for *each step in episode* **do**
    Take action **u**, observe **r** and **x**;
    Choose **u'** from **x'** using ϵ-greedy ;
    $\boldsymbol{Q(x, u) \leftarrow Q(x, u) + \alpha\Big[r + \gamma Q(x', u') - Q(x, u)\Big]}$;
    x $\leftarrow$ **x'**
    u $\leftarrow$ **u'**;
    until all states are visited;
  end
end

5.2 Q-learning

The algorithm for Q-Learning is shown below. It is an off policy algorithm where the agent chooses the greedy policy at the next state in order to update the reward for the current state.

Algorithm 2: Q-Learning

Initialize the Q table:
Initialize $\boldsymbol{Q(x, u)}$,$\forall x \in X, u \in A(x)$arbitrarily
for *each episode* **do**
  Initialize **x**;
  for *each step in episode* **do**
    Choose **u** from A(**x**) using ϵ-greedy;
    Take action **u**, observe **r** and **x'**;
    $\boldsymbol{Q(x, u) \leftarrow Q(x, u) + \alpha\Big[r + \gamma}\ max_{\boldsymbol{u}}\boldsymbol{Q(x', u) - Q(x, u)\Big]}$;
    x $\leftarrow$ **x'**;
    until all states are visited;
  end
end

6 Results

6.1 Predefined optimal control policy

The Figure 5 below shows the path obtained by each of the algorithms when the movement costs were as given in Figure 3. Starting ϵ_0, i.e. probability of choosing a movement different from the one dictated by the Q-table is $\epsilon_0 = 0.4$ and it is reduced by a factor of $\frac{1}{1.0009}$ with each episode/epoch.

For all experiments, the reward for visiting an unvisited cell is set to 10 and visiting a previously visited cell is set to -20.

The path is identical for both algorithms and indeed represents the optimal policy.

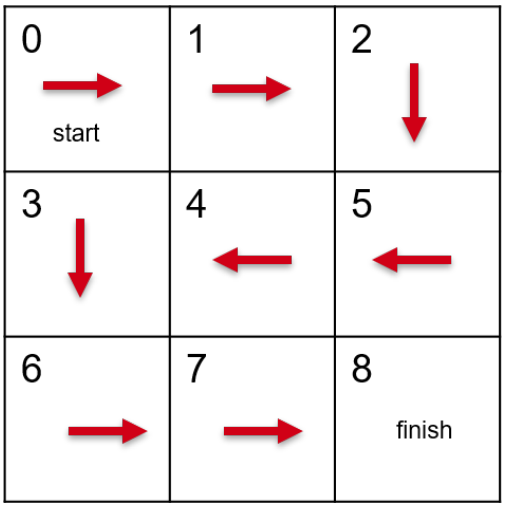

Figure 5: Optimal path as obtained by SARSA and Q-learning

The next two Figures below show the evolution of the optimal policy for SARSA (Figure 6) and Q-learning (Figure 7) algorithms. In these Figures y-axis represents an action taken.

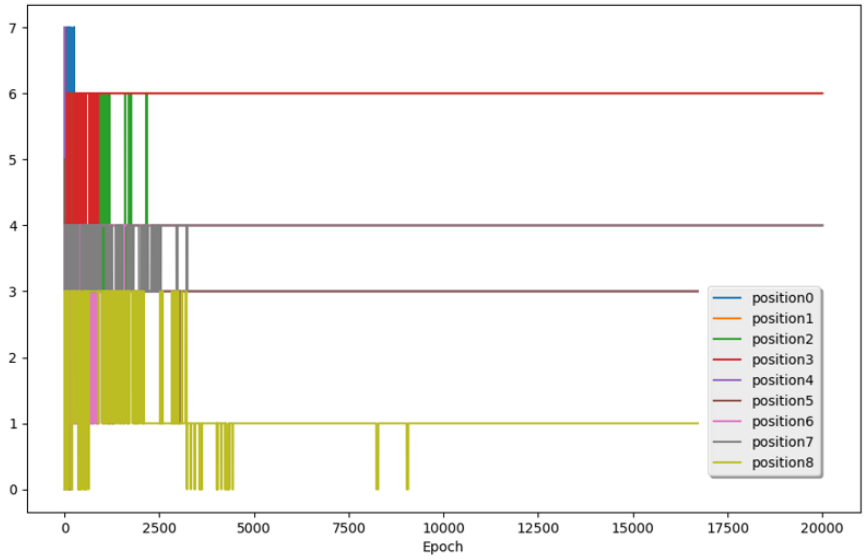

Figure 6: Optimal policy obtained by SARSA

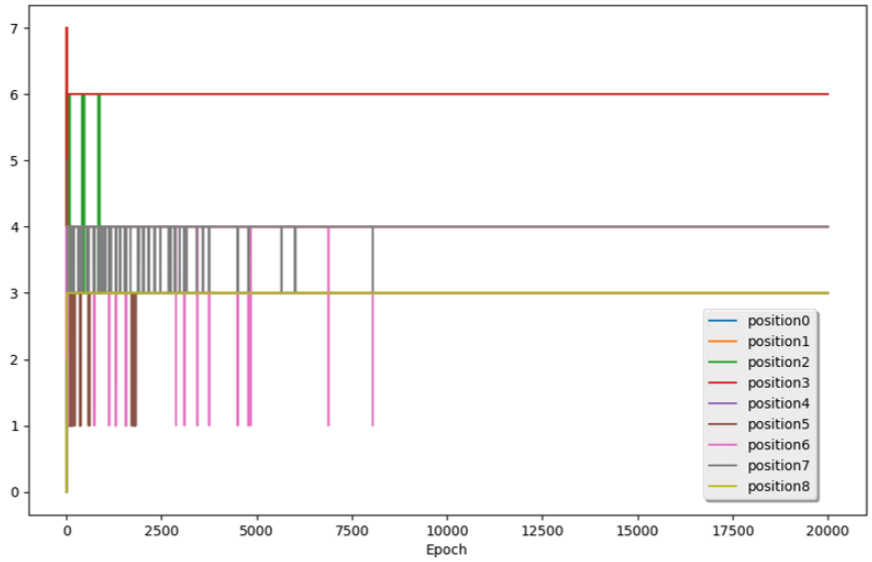

Figure 7: Optimal policy obtained by Q-learning

As can be seen, when action costs are as given in Figure 3, for most of the cells SARSA converged slightly faster than Q-learning algorithm. However, all states converged faster for

Q-learning algorithm. The difference is around 1000 epochs.

6.2 No predefined optimal control policy

The situation changes when action costs are the same as shown in Figure 8 below. The optimal policy is not predefined by the action costs and the algorithms have to truly learn the policy and not just find it.

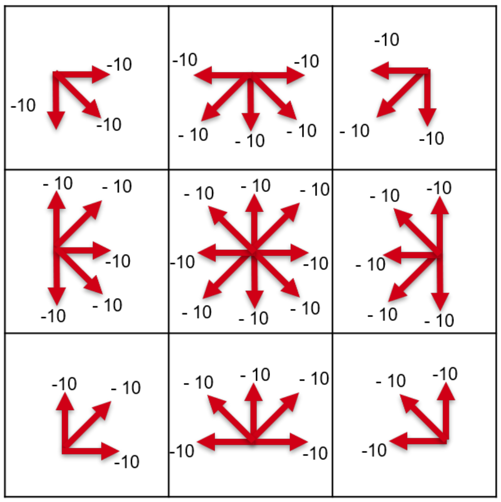

Figure 8: Allowed movements. Identical costs for each movement. No optimal policy is predefined.

In this case, SARSA and Q-learning algorithms find different optimal policies as shown in Figures 9 and 10. Both are correct as the movement costs are the same in every direction.

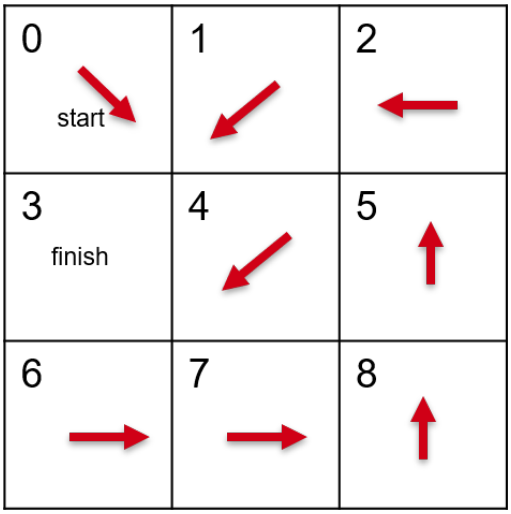

Figure 9: Optimal policy using SARSA for equal action costs

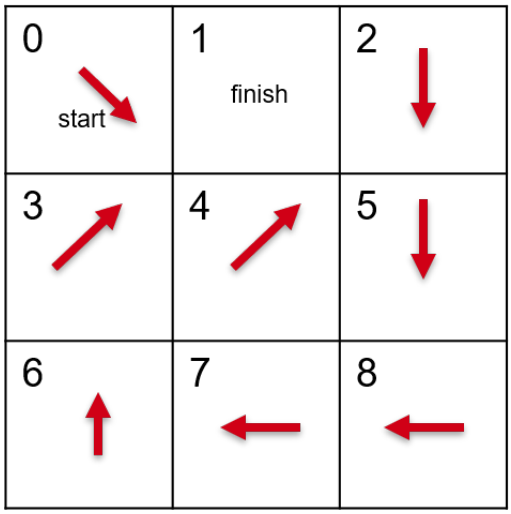

Figure 10: Optimal policy using Q-learning for equal action costs

As Figures 11 and 12 show, the difference in the speed of convergence is now more apparent than for the case with predefined optimal policy. In these Figures y-axis represents an action taken.

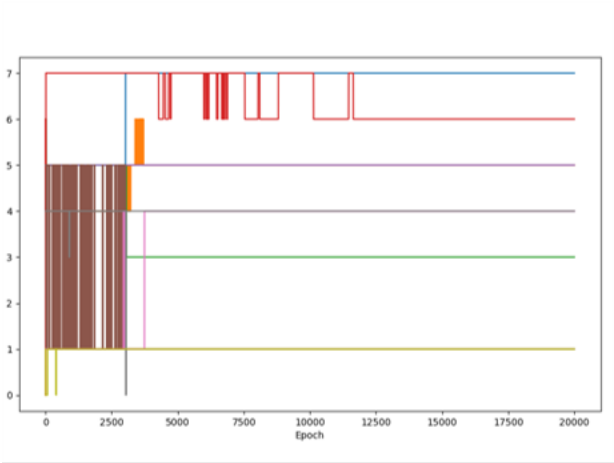

Figure 11: Optimal policy using SARSA for equal action costs

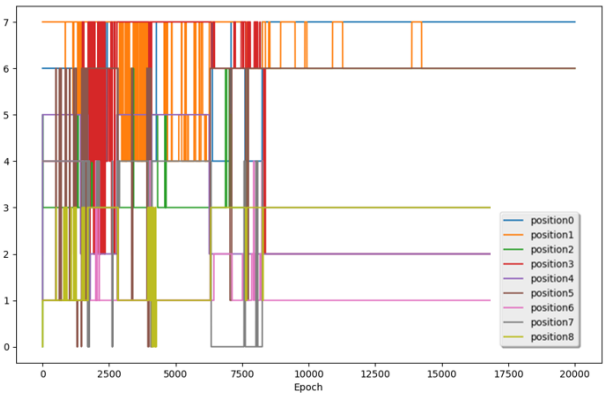

Figure 12: Optimal policy using Q-learning for equal action costs

The differences in the convergence speeds of the algorithms are much more profound than in the case of predefined optimal policy. When using SARSA, for most of the cells the optimal policy was found after around 4000 epochs with the policy for the last cell converging after 11000 epochs. From Figure 12, Q-learning converged for most of the cells after around 8000 epoch and the last cell converged after around 14000 epochs. From these findings we

can conclude that SARSA should be preferred over Q-learning algorithm as it converges faster or at roughly the same speed as Q-learning.

7 Learning experience, discussion and conclusion

In this project, we worked on further deepening our understanding of reinforcement learning concept. Optimal paths were constructed. SARSA and Q-learning algorithms were compared. SARSA is the preferred algorithm as it converges faster.
One big realization for us was that the same technique introduced above for 3×3 grid without revisits cannot be applied to 5×5 grid with revisits. This happens due to the fact, that revisits require taking different actions upon a visit and revisit. In order to make it work, we would need a bigger Q-table with the number of elements equal to $2^{48} \times 8$, where 48 is the number of possible states when starting at cell 0 and 8 is the number of possible actions. For the grid of size 3×3, it was taking roughly 10 minutes for Q-table to converge for both algorithms. With the Q-table of size $2^{48} \times 8$, this time would be much higher. A better approach might be to use neural networks.

References

[1] Dr. Agung Julius. Homework 4. Retrieved April 11, 2020 from https://www.piazza.com/.

[2] University of Crete Computer Science Department. The traveling salesman problem. Retrieved April 19, 2020 from https://www.piazza.com/.

[3] D. Revuz. *Markov Chains*. North-Holland Mathematical Library, 1984.

[4] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

Appendix. 3×3 SARSA

```
import numpy as np
import random as rd
import copy as cp
import test_plot as tp

def find_max_action(Q,directions,num,x,y,visited,R,dim):
       Q_copy = cp.deepcopy(Q[num,:])
```

```
        #print(Q_copy)

        for a in directions:
                n_s = find_next_state(x,y,a,dim)
                not_visited_reward = (1-visited[n_s])*R
                Q_copy[a] +=not_visited_reward

        max_Q = max(Q_copy[directions])

        action_greedy = [i for i,j in enumerate(Q_copy) if j == max_Q and i
            ↪ in directions]

        del Q_copy

        return action_greedy


def epsilon_greedy(epsilon,Q,directions,num,x,y,visited,R,dim):
        r = np.random.random_sample()

        if r>epsilon:
                action_greedy = find_max_action(Q,directions,num,x,y,visited,
                    ↪ R,dim)
                a = rd.choice(action_greedy)
        else:
                a = rd.choice(directions)
        return a


def available_actions(Q,C,x,y,n):
        directions = list()
        A = cp.deepcopy(C)
        if y==0:
                A[0] = 0
                A[3] = 0
                A[5] = 0
        if y==n-1:
```

```
                A[2] = 0
                A[4] = 0
                A[7] = 0
        if x==0:
                A[0] = 0
                A[1] = 0
                A[2] = 0
        if x==n-1:
                A[5] = 0
                A[6] = 0
                A[7] = 0

        #print(A)

        for i in range(0,np.size(A)):
                if not A[i]==0:
                        directions.append(i)

        del A
        return directions




def q_learn(epochs,alpha,gamma,epsilon,d_shape):
        Q = np.zeros(shape = (d_shape*d_shape,8)) #Cost matrix
        A = np.ones(8) #Array of possible directions
        visited = np.zeros(d_shape*d_shape) #array of visited states
        revisited = np.zeros(d_shape*d_shape) #array of states which are
            ↪ revisited

        R_visit = 10
        R_revisit = -20
        R_not_visited = 100
        '''R_movement = np.array([[0,-10,0,-20,-20,0,0,0,0],
                [-10,0,-10,-20,-20,-20,0,0,0],
```

```
        [0,-10,0,0,-20,-10,0,0,0],
        [-20,-20,0,0,-10,0,-10,-20,0],
        [-20,-20,-20,-10,0,-10,-20,-20,-20],
        [0,-20,-10,0,-10,0,0,-20,-20],
        [0,0,0,-10,-20,0,0,-10,0],
        [0,0,0,-20,-20,-20,-10,0,-10],
        [0,0,0,0,-20,-20,0,-10,0]])'''

R_movement = np.array([[0,-1,0,-1,-1,0,0,0,0],
        [-1,0,-1,-1,-1,-1,0,0,0],
        [0,-1,0,0,-1,-1,0,0,0],
        [-1,-1,0,0,-1,0,-1,-1,0],
        [-1,-1,-1,-1,0,-1,-1,-1,-1],
        [0,-1,-1,0,-1,0,0,-1,-1],
        [0,0,0,-1,-1,0,0,-1,0],
        [0,0,0,-1,-1,-1,-1,0,-1],
        [0,0,0,0,-1,-1,0,-1,0]])

Q_record = np.zeros(shape=(epochs,d_shape*d_shape,8))

action_record = np.zeros(shape =(epochs,d_shape*d_shape))

present_x_total = list()
next_x_total = list()
action_x_total = list()

for i in range(0,epochs):
        num = 0
        visited[num]=1
        present_x = list()
        next_x = list()
        action_x = list()
        x,y = calc_index(num,d_shape)
        present_x.append(num)
        directions = available_actions(Q,A,x,y,d_shape)
        #print(directions)
```

```
action = epsilon_greedy(epsilon,Q,directions,num,x,y,visited,
    ↪ R_not_visited,d_shape)
#print("Action taken",action)
action_x.append(action)
while not visited.all()==1:
        '''x,y = calc_index(num,d_shape)
        present_x.append(num)
        directions = available_actions(Q,A,x,y,d_shape)
        #print(directions)
        action = epsilon_greedy(epsilon,Q,directions,num,x,y,
            ↪ visited,R_not_visited,d_shape)
        #print("Action taken",action)
        action_x.append(action)'''
        next_state = find_next_state(x,y,action,d_shape)
        #print("Next State",next_state)
        next_x.append(next_state)

        x_next,y_next = calc_index(next_state,d_shape)
        next_state_directions = available_actions(Q,A,x_next,
            ↪ y_next,d_shape)
        next_state_action = epsilon_greedy(epsilon,Q,
            ↪ next_state_directions,next_state,x_next,y_next,
            ↪ visited,R_not_visited,d_shape)

        if visited[next_state]==1:
            revisited[next_state]==1
        reward = R_movement[num,next_state] + ((1-visited[
            ↪ next_state])*R_visit) + (revisited[next_state]*
            ↪ R_revisit)
        #print("Reward",reward)
        error = reward  + gamma*(Q[next_state,
            ↪ next_state_action])-Q[num,action]
        Q[num,action] = Q[num,action] + alpha*error
        visited[next_state] = 1
        num = next_state
        x,y = calc_index(num,d_shape)
        present_x.append(num)
```

```
                        directions = available_actions(Q,A,x,y,d_shape)
                        #print(directions)
                        action = epsilon_greedy(epsilon,Q,directions,num,x,y,
                            ↪ visited,R_not_visited,d_shape)
                        #print("Action taken",action)
                        action_x.append(action)

                #print(i)
                epsilon /= 1.0005
                Q_record[i,:,:] = Q
                present_x_total.append(present_x)
                next_x_total.append(next_x)
                action_x_total.append(action_x)
                visited[:] = np.zeros(d_shape*d_shape)
                del present_x
                del next_x
                del action_x
                max_val = final_action(Q)
                action_record[i] = max_val


        print("Present",present_x_total[-1])
        print("Aciton",action_x_total[-1])
        print("Next State",next_x_total[-1])
        print(Q_record[-1,:,:])
        final_output(Q)
        tp.action_plot(action_record)


def final_output(Q):
        ar = np.zeros(np.size(Q,0))

        for i in range(0,np.size(Q,0)):
                check = 1
                idx = 0
                for j in range(0,np.size(Q,1)):
                        if not Q[i,j]==0:
```

```
                        if check==1:
                                max = Q[i,j]
                                idx = j
                                check = 0
                        else:
                                if max<Q[i,j]:
                                        max = Q[i,j]
                                        idx = j
                print("Grid",i,"idx",idx)




def final_action(Q):
        ar = np.zeros(np.size(Q,0))

        for i in range(0,np.size(Q,0)):
                check = 1
                idx = 0
                for j in range(0,np.size(Q,1)):
                        if not Q[i,j]==0:
                                if check==1:
                                        max = Q[i,j]
                                        idx = j
                                        check = 0
                                else:
                                        if max<Q[i,j]:
                                                max = Q[i,j]
                                                idx = j
                ar[i] = idx
        return ar




def find_next_state(x,y,action,d_shape):
```

```
        if action==0:
                next = d_shape*(x-1)+(y-1)
        elif action==1:
                next = d_shape*(x-1)+y
        elif action==2:
                next = d_shape*(x-1)+(y+1)
        elif action==3:
                next = d_shape*x + (y-1)
        elif action==4:
                next = d_shape*x + (y+1)
        elif action==5:
                next = d_shape*(x+1) + (y-1)
        elif action==6:
                next = d_shape*(x+1) + y
        else:
                next = d_shape*(x+1) + (y+1)
        return next


def calc_index(num,d_shape):
        x = int(num/d_shape)
        y = num%d_shape
        return x,y

if __name__=="__main__":
        q_learn(20000,0.05,0.9,0.4,3)
```

Appendix. 3×3 Q-learning

```
import numpy as np
import pandas as pd
import random as rd
import copy as cp
import test_plot as tp
```

```
def find_max_action(Q,directions,num,x,y,visited,R,dim):
        Q_copy = cp.deepcopy(Q[num,:])
        #print(Q_copy)

        for a in directions:
                n_s = find_next_state(x,y,a,dim)
                not_visited_reward = (1-visited[n_s])*R
                Q_copy[a] +=not_visited_reward

        max_Q = max(Q_copy[directions])

        action_greedy = [i for i,j in enumerate(Q_copy) if j == max_Q and i
            ↪ in directions]

        del Q_copy

        return action_greedy


def epsilon_greedy(epsilon,Q,directions,num,x,y,visited,R,dim):
        r = np.random.random_sample()

        if r>epsilon:
                action_greedy = find_max_action(Q,directions,num,x,y,visited,
                    ↪ R,dim)
                a = rd.choice(action_greedy)
        else:
                a = rd.choice(directions)
        return a


def available_actions(Q,C,x,y,n):
        directions = list()
        A = cp.deepcopy(C)
        if y==0:
                A[0] = 0
                A[3] = 0
```

```
                A[5] = 0
        if y==n-1:
                A[2] = 0
                A[4] = 0
                A[7] = 0
        if x==0:
                A[0] = 0
                A[1] = 0
                A[2] = 0
        if x==n-1:
                A[5] = 0
                A[6] = 0
                A[7] = 0

        #print(A)

        for i in range(0,np.size(A)):
                if not A[i]==0:
                        directions.append(i)
        del A

        return directions




def q_learn(epochs,alpha,gamma,epsilon,d_shape):
        Q = np.zeros(shape = (d_shape*d_shape,8)) #Cost matrix
        A = np.ones(8) #Array of possible directions
        visited = np.zeros(d_shape*d_shape) #array of visited states
        revisited = np.zeros(d_shape*d_shape) #array of states which are
            ↪ revisited

        R_visit = 10
        R_revisit = -40
        R_not_visited = 100
```

```
'''R_movement = np.array([[0,-10,0,-20,-20,0,0,0,0],
        [-10,0,-10,-20,-20,-20,0,0,0],
        [0,-10,0,0,-20,-10,0,0,0],
        [-20,-20,0,0,-10,0,-10,-20,0],
        [-20,-20,-20,-10,0,-10,-20,-20,-20],
        [0,-20,-10,0,-10,0,0,-20,-20],
        [0,0,0,-10,-20,0,0,-10,0],
        [0,0,0,-20,-20,-20,-10,0,-10],
        [0,0,0,0,-20,-20,0,-10,0]])'''

R_movement = np.array([[0,-1,0,-1,-1,0,0,0,0],
        [-1,0,-1,-1,-1,-1,0,0,0],
        [0,-1,0,0,-1,-1,0,0,0],
        [-1,-1,0,0,-1,0,-1,-1,0],
        [-1,-1,-1,-1,0,-1,-1,-1,-1],
        [0,-1,-1,0,-1,0,0,-1,-1],
        [0,0,0,-1,-1,0,0,-1,0],
        [0,0,0,-1,-1,-1,-1,0,-1],
        [0,0,0,0,-1,-1,0,-1,0]])

'''R_movement = np.array
    ↪ ([[0,-1,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#0
        [-1,0,-1,0,0,-1,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#1
            ↪
        [0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#2
            ↪
        [0,0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#3
            ↪
        [0,0,0,-1,0,0,0,0,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#4
        [-1,-1,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0],#5
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#6
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#7
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#8
        [0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1,0,0,0,0,0,0,0,0,0],#9
        [0,0,0,0,0,-1,-100,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0,0],#10
            ↪
```

```
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#11
        [0,0,0,0,0,0,-100,-100,-100,0,0,-100,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0],
            ↪
        [0,0,0,0,0,0,0,-100,-100,-1,0,0,-1,0,-1,0,0,-100,-100,-1,0,0,0,0,0],#13
            ↪
        [0,0,0,0,0,0,0,0,-100,-1,0,0,0,-1,0,0,0,0,-100,-1,0,0,0,0,0],#14
            ↪
        [0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,-100,0,0,0,-1,-1,0,0,0],#15
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#16
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#17
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#18
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1],#19
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,-1,0,0,0],#20
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,-100,0,0,-1,0,-1,0,0],#21
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-100,0,0,-1,0,-1,0],#22
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-1,0,0,-1,0,-1],#23
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-1,0,0,0,-1,0],#24'''
            ↪


Q_record = np.zeros(shape=(epochs,d_shape*d_shape,8))

action_record = np.zeros(shape =(epochs,d_shape*d_shape))

present_x_total = list()
next_x_total = list()
action_x_total = list()

for i in range(0,epochs):
        num = 0
        visited[num]=1
        present_x = list()
```

```
next_x = list()
action_x = list()
while not visited.all()==1:
        x,y = calc_index(num,d_shape)
        present_x.append(num)
        directions = available_actions(Q,A,x,y,d_shape)
        #print(directions)
        action = epsilon_greedy(epsilon,Q,directions,num,x,y,
            ↪ visited,R_not_visited,d_shape)
        #print("Action taken",action)
        action_x.append(action)
        next_state = find_next_state(x,y,action,d_shape)
        #print("Next State",next_state)
        next_x.append(next_state)
        if visited[next_state]==1:
                revisited[next_state]==1
        reward = R_movement[num,next_state] + ((1-visited[
            ↪ next_state])*R_visit) + (revisited[next_state]*
            ↪ R_revisit)
        #print("Reward",reward)
        error = reward  + gamma*max(Q[num,:])-Q[num,action]
        Q[num,action] = Q[num,action] + alpha*error
        visited[next_state] = 1
        num = next_state

epsilon /= 1.0005
Q_record[i,:,:] = Q
present_x_total.append(present_x)
next_x_total.append(next_x)
action_x_total.append(action_x)
visited[:] = np.zeros(d_shape*d_shape)
#print("present_x:",present_x)
#print("action:",action_x)
del present_x
del next_x
del action_x
max_val = final_action(Q)
```

```
                action_record[i] = max_val


        print("Present",present_x_total[-1])
        print("Aciton",action_x_total[-1])
        print("Next State",next_x_total[-1])
        print(Q_record[-1,:,:])
        final_output(Q)
        #print("Action_record:",action_record)
        #tp.plot_Q_values(Q_record)
        tp.action_plot(action_record)



def final_output(Q):
        ar = np.zeros(np.size(Q,0))

        for i in range(0,np.size(Q,0)):
                check = 1
                idx = 0
                for j in range(0,np.size(Q,1)):
                        if not Q[i,j]==0:
                                if check==1:
                                        max = Q[i,j]
                                        idx = j
                                        check = 0
                                else:
                                        if max<Q[i,j]:
                                                max = Q[i,j]
                                                idx = j
                print("Grid",i,"idx",idx)




def final_action(Q):
        ar = np.zeros(np.size(Q,0))
```

```
	for i in range(0,np.size(Q,0)):
		check = 1
		idx = 0
		for j in range(0,np.size(Q,1)):
			if not Q[i,j]==0:
				if check==1:
					max = Q[i,j]
					idx = j
					check = 0
				else:
					if max<Q[i,j]:
						max = Q[i,j]
						idx = j
		ar[i] = idx
	return ar




def find_next_state(x,y,action,d_shape):
	if action==0:
		next = d_shape*(x-1)+(y-1)
	elif action==1:
		next = d_shape*(x-1)+y
	elif action==2:
		next = d_shape*(x-1)+(y+1)
	elif action==3:
		next = d_shape*x + (y-1)
	elif action==4:
		next = d_shape*x + (y+1)
	elif action==5:
		next = d_shape*(x+1) + (y-1)
	elif action==6:
		next = d_shape*(x+1) + y
	else:
```

```
                next = d_shape*(x+1) + (y+1)
        return next


def calc_index(num,d_shape):
        x = int(num/d_shape)
        y = num%d_shape
        return x,y

if __name__=="__main__":
        q_learn(20000,0.05,0.9,0.4,3)
```

Appendix. 5×5 SARSA

```
import numpy as np
import random as rd
import copy as cp
import operator


def find_max_action(Q,directions,num,x,y,visited,R,dim,not_allowed):
        Q_copy = cp.deepcopy(Q[num,:])
        #print(Q_copy)

        for a in directions:
                n_s = find_next_state(x,y,a,dim)
                if n_s not in not_allowed:
                        not_visited_reward = (1-visited[n_s])*R
                        Q_copy[a] +=not_visited_reward

        max_Q = max(Q_copy[directions])

        action_greedy = [i for i,j in enumerate(Q_copy) if j == max_Q and i
            ↪ in directions]

        del Q_copy
```

```
        return action_greedy


def epsilon_greedy(epsilon,Q,directions,num,x,y,visited,R,dim,not_allowed):
        r = np.random.random_sample()

        if r>epsilon:
                action_greedy = find_max_action(Q,directions,num,x,y,visited,
                    ↪ R,dim,not_allowed)
                a = rd.choice(action_greedy)
        else:
                a = rd.choice(directions)
        return a


def available_actions(Q,C,x,y,n):
        directions = list()
        A = cp.deepcopy(C)
        if y==0:
                A[0] = 0
                A[3] = 0
                A[5] = 0
        if y==n-1:
                A[2] = 0
                A[4] = 0
                A[7] = 0
        if x==0:
                A[0] = 0
                A[1] = 0
                A[2] = 0
        if x==n-1:
                A[5] = 0
                A[6] = 0
                A[7] = 0

        #print(A)
```

```
	for i in range(0,np.size(A)):
		if not A[i]==0:
			directions.append(i)
	del A

	return directions


def calc_go(l):
	prod = 1
	for i in l.values():
		prod *= i
	return prod



def q_learn(epochs,alpha,gamma,epsilon,d_shape):
	Q = np.zeros(shape = (d_shape*d_shape,8)) #Cost matrix
	A = np.ones(8) #Array of possible directions
	visited = dict() #array of visited states
	revisited = dict()#array of states which are revisited

	R_visit = 10
	R_revisit = -20
	R_not_visited = 30

	not_allowed = [6,7,8,11,16,17,18]

	for i in range(0,(d_shape*d_shape)):
		if i in not_allowed:
			continue
		else:
			visited[i] = 0
			revisited[i] = 0
```

```
R_movement = np.array
    ↪ ([[0,-1,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#0
        [-1,0,-1,0,0,-1,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#1
            ↪
        [0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#2
            ↪
        [0,0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#3
            ↪
        [0,0,0,-1,0,0,0,0,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#4
        [-1,-1,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0,0],#5
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#6
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#7
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#8
        [0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1,0,0,0,0,0,0,0,0,0],#9
        [0,0,0,0,0,0,-1,-100,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0],#10
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#11
        [0,0,0,0,0,0,0,-100,-100,-100,0,0,0,-100,0,0,-1,0,0,0,-100,-100,-100,0,0,0,0,0,0],
            ↪
        [0,0,0,0,0,0,0,0,-100,-100,-1,0,0,0,-1,0,0,-1,0,0,0,-100,-100,-1,0,0,0,0,0,0],#13
            ↪
        [0,0,0,0,0,0,0,0,0,-100,-1,0,0,0,0,-1,0,0,0,0,0,-100,-1,0,0,0,0,0,0],#14
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,0,-100,0,0,0,0,-1,-1,0,0,0],#15
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#16
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#17
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#18
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1],#19
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,-1,0,0,0],#20
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,-100,0,0,-1,0,-1,0,0],#21
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-100,0,0,-1,0,-1,0],#22
            ↪
```

```
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-1,0,0,-1,0,-1],#23
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-1,0,0,0,-1,0]])#24


Q_record = np.zeros(shape=(epochs,d_shape*d_shape,8))

present_x_total = list()
next_x_total = list()
action_x_total = list()

for j in range(0,epochs):
        num = 0
        visited[num]=1
        present_x = list()
        next_x = list()
        action_x = list()
        go = 0
        x,y = calc_index(num,d_shape)
        present_x.append(num)
        directions = available_actions(Q,A,x,y,d_shape)
        #print(directions)
        action = epsilon_greedy(epsilon,Q,directions,num,x,y,visited,
            ↪ R_not_visited,d_shape,not_allowed)
        #print("Action taken",action)
        action_x.append(action)
        while go==0:
                next_state = find_next_state(x,y,action,d_shape)
                #print("Next State",next_state)
                next_x.append(next_state)
                if next_state in not_allowed:
                        reward = R_movement[num,next_state]
                        error = reward  + gamma*max(Q[num,:])-Q[num,
                            ↪ action]
                        Q[num,action] = Q[num,action] + alpha*error
                        for i in range(0,(d_shape*d_shape)):
                                if i in not_allowed:
```

```
                                continue
                        else:
                                visited[i] = 0
                                revisited[i] = 0
                num = 0
                go = calc_go(visited)
                x,y = calc_index(num,d_shape)
                present_x.append(num)
                directions = available_actions(Q,A,x,y,d_shape)
                #print(directions)
                action = epsilon_greedy(epsilon,Q,directions,
                    ↪ num,x,y,visited,R_not_visited,d_shape,
                    ↪ not_allowed)
                #print("Action taken",action)
                action_x.append(action)
                present_x.clear()
                action_x.clear()
                next_x.clear()
                continue

        else:
                x_next,y_next = calc_index(next_state,d_shape)
                next_directions = available_actions(Q,A,x_next,
                    ↪ y_next,d_shape)
                next_action = epsilon_greedy(epsilon,Q,
                    ↪ next_directions,next_state,x_next,y_next
                    ↪ ,visited,R_not_visited,d_shape,
                    ↪ not_allowed)

                if visited[next_state]==1:
                        revisited[next_state]+=1
                reward = R_movement[num,next_state] + ((1-
                    ↪ visited[next_state])*R_visit)
                #print("Reward",reward)
                if not revisited[next_state]==0:
                        reward += + R_revisit
```

```
                    error = reward  + gamma*(Q[next_state,
                        ↪ next_action])-Q[num,action]
                    Q[num,action] = Q[num,action] + alpha*error
                    visited[next_state] = 1
                    num = next_state
                    go = calc_go(visited)
                    x,y = calc_index(num,d_shape)
                    present_x.append(num)
                    directions = available_actions(Q,A,x,y,d_shape)
                    #print(directions)
                    action = epsilon_greedy(epsilon,Q,directions,
                        ↪ num,x,y,visited,R_not_visited,d_shape,
                        ↪ not_allowed)
                    #print("Action taken",action)
                    action_x.append(action)

    if j%100==0:
            print(j)
    if not j==epochs-1:
            for i in range(0,(d_shape*d_shape)):
                    if i in not_allowed:
                            continue
                    else:
                            visited[i] = 0
                            revisited[i] = 0
    Q_record[j,:,:] = Q
    present_x_total.append(present_x)
    next_x_total.append(next_x)
    action_x_total.append(action_x)

    del present_x
    del next_x
    del action_x

    #epsilon /= 1.000
```

```
        print("Present",present_x_total[-1])
        print("Aciton",action_x_total[-1])
        print("Next State",next_x_total[-1])
        print(Q_record[-1,:,:])
        print("Revisited::",revisited)
        final_output(Q,revisited,not_allowed)



def final_output(Q,revisited,not_allowed):
        Q_copy = cp.deepcopy(Q)
        for k,v in revisited.items():
                if not k in not_allowed:
                        if not v==0:
                                ar = remove_zeros(Q_copy[k],0)
                                sorted_ar = sorted(ar.items(), key=operator.
                                    ↪ itemgetter(1),reverse = True)
                                while not (revisited[k]+1==0):
                                        if sorted_ar:
                                                print("Grid",k,"idx", sorted_ar
                                                    ↪ [0][0])
                                                del sorted_ar[0]
                                                revisited[k] -=1
                                        else:
                                                break
                        else:
                                ar = remove_zeros(Q_copy[k],0)
                                sorted_ar = sorted(ar.items(), key=operator.
                                    ↪ itemgetter(1),reverse = True)
                                print("Grid",k," idx",sorted_ar[0][0])


def remove_zeros(ar,val):
        map = dict()
        for i in range(0,len(ar)):
                if not ar[i]==val:
```

```
                        map[i] = ar[i]
        return map

def find_next_state(x,y,action,d_shape):
        if action==0:
                next = d_shape*(x-1)+(y-1)
        elif action==1:
                next = d_shape*(x-1)+y
        elif action==2:
                next = d_shape*(x-1)+(y+1)
        elif action==3:
                next = d_shape*x + (y-1)
        elif action==4:
                next = d_shape*x + (y+1)
        elif action==5:
                next = d_shape*(x+1) + (y-1)
        elif action==6:
                next = d_shape*(x+1) + y
        else:
                next = d_shape*(x+1) + (y+1)
        return next


def calc_index(num,d_shape):
        x = int(num/d_shape)
        y = num%d_shape
        return x,y

if __name__=="__main__":
        q_learn(20000,0.06,0.9,0.4,5)
```

Appendix. 5×5 Q-learning

```
import numpy as np
import pandas as pd
import random as rd
```

```
import copy as cp
import operator


def find_max_action(Q,directions,num,x,y,visited,R,dim,not_allowed):
        Q_copy = cp.deepcopy(Q[num,:])
        #print(Q_copy)

        for a in directions:
                n_s = find_next_state(x,y,a,dim)
                if n_s not in not_allowed:
                        not_visited_reward = (1-visited[n_s])*R
                        Q_copy[a] +=not_visited_reward

        max_Q = max(Q_copy[directions])

        action_greedy = [i for i,j in enumerate(Q_copy) if j == max_Q and i
            ↪ in directions]

        del Q_copy

        return action_greedy


def epsilon_greedy(epsilon,Q,directions,num,x,y,visited,R,dim,not_allowed):
        r = np.random.random_sample()

        if r>epsilon:
                action_greedy = find_max_action(Q,directions,num,x,y,visited,
                    ↪ R,dim,not_allowed)
                a = rd.choice(action_greedy)
        else:
                a = rd.choice(directions)
        return a


def available_actions(Q,C,x,y,n):
```

```
        directions = list()
        A = cp.deepcopy(C)
        if y==0:
                A[0] = 0
                A[3] = 0
                A[5] = 0
        if y==n-1:
                A[2] = 0
                A[4] = 0
                A[7] = 0
        if x==0:
                A[0] = 0
                A[1] = 0
                A[2] = 0
        if x==n-1:
                A[5] = 0
                A[6] = 0
                A[7] = 0

        #print(A)

        for i in range(0,np.size(A)):
                if not A[i]==0:
                        directions.append(i)
        del A

        return directions


def calc_go(l):
        prod = 1
        for i in l.values():
                prod *= i
        return prod
```

```
def q_learn(epochs,alpha,gamma,epsilon,d_shape):
        Q = np.zeros(shape = (d_shape*d_shape,8)) #Cost matrix
        A = np.ones(8) #Array of possible directions
        visited = dict() #array of visited states
        revisited = dict()#array of states which are revisited

        R_visit = 10
        R_revisit = -20
        R_not_visited = 30

        not_allowed = [6,7,8,11,16,17,18]

        for i in range(0,(d_shape*d_shape)):
                if i in not_allowed:
                        continue
                else:
                        visited[i] = 0
                        revisited[i] = 0

        R_movement = np.array
            ↪ ([[0,-1,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#0
                [-1,0,-1,0,0,-1,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#1
                    ↪
                [0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#2
                    ↪
                [0,0,-1,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#3
                    ↪
                [0,0,0,-1,0,0,0,0,-100,-100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#4
                [-1,-1,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0,0,0,0,0,0],#5
                    ↪
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#6
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#7
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#8
                [0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1,0,0,0,0,0,0,0,0,0,0],#9
                [0,0,0,0,0,-1,-100,0,0,0,0,0,-100,0,0,0,-1,-100,0,0,0,0,0,0,0],#10
                    ↪
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#11
```

```
        [0,0,0,0,0,0,0,-100,-100,-100,0,0,-100,0,-1,0,0,-100,-100,-100,0,0,0,0,0,0],
            ↪
        [0,0,0,0,0,0,0,0,-100,-100,-1,0,0,-1,0,-1,0,0,-100,-100,-1,0,0,0,0,0],#13
            ↪
        [0,0,0,0,0,0,0,0,0,-100,-1,0,0,0,-1,0,0,0,0,-100,-1,0,0,0,0,0],#14
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,-100,0,0,0,-1,-1,0,0,0],#15
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#16
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#17
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],#18
        [0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,0,0,0,-100,0,0,0,0,-1,-1],#19
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,0,0,0,0,-1,0,0,0],#20
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-100,-100,0,0,-1,0,-1,0,0],#21
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-100,0,0,-1,0,-1,0],#22
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-100,-1,0,0,-1,0,-1],#23
            ↪
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-100,-1,0,0,0,-1,0]])#24


Q_record = np.zeros(shape=(epochs,d_shape*d_shape,8))

present_x_total = list()
next_x_total = list()
action_x_total = list()

for j in range(0,epochs):
        num = 0
        visited[num]=1
        present_x = list()
        next_x = list()
        action_x = list()
        go = 0
        while go==0:
```

```
x,y = calc_index(num,d_shape)
present_x.append(num)
directions = available_actions(Q,A,x,y,d_shape)
#print(directions)
action = epsilon_greedy(epsilon,Q,directions,num,x,y,
    ↪ visited,R_not_visited,d_shape,not_allowed)
#print("Action taken",action)
action_x.append(action)
next_state = find_next_state(x,y,action,d_shape)
#print("Next State",next_state)
next_x.append(next_state)
if next_state in not_allowed:
        reward = R_movement[num,next_state]
        error = reward  + gamma*max(Q[num,:])-Q[num,
            ↪ action]
        Q[num,action] = Q[num,action] + alpha*error
        for i in range(0,(d_shape*d_shape)):
                if i in not_allowed:
                        continue
                else:
                        visited[i] = 0
                        revisited[i] = 0
        num = 0
        go = calc_go(visited)
        present_x.clear()
        action_x.clear()
        next_x.clear()
        continue

else:
        if visited[next_state]==1:
                revisited[next_state]+=1
        reward = R_movement[num,next_state] + ((1-
            ↪ visited[next_state])*R_visit)
        #print("Reward",reward)
        if not revisited[next_state]==0:
                reward += + R_revisit
```

```
                        error = reward  + gamma*max(Q[next_state,:])-Q[
                            ↪ num,action]
                        Q[num,action] = Q[num,action] + alpha*error
                        visited[next_state] = 1
                        num = next_state
                        go = calc_go(visited)

        if j%100==0:
                print(j)
        if not j==epochs-1:
                for i in range(0,(d_shape*d_shape)):
                        if i in not_allowed:
                                continue
                        else:
                                visited[i] = 0
                                revisited[i] = 0
        Q_record[j,:,:] = Q
        present_x_total.append(present_x)
        next_x_total.append(next_x)
        action_x_total.append(action_x)

        del present_x
        del next_x
        del action_x



print("Present",present_x_total[-1])
print("Aciton",action_x_total[-1])
print("Next State",next_x_total[-1])
print(Q_record[-1,:,:])
print("Revisited::",revisited)
final_output(Q,revisited,not_allowed)
```

```
def final_output(Q,revisited,not_allowed):
	Q_copy = cp.deepcopy(Q)
	for k,v in revisited.items():
		if not k in not_allowed:
			if not v==0:
				ar = remove_zeros(Q_copy[k],0)
				sorted_ar = sorted(ar.items(), key=operator.
				    ↪ itemgetter(1),reverse = True)
				while not (revisited[k]+1==0):
					if sorted_ar:
						print("Grid",k,"idx", sorted_ar
						    ↪ [0][0])
						del sorted_ar[0]
						revisited[k] -=1
					else:
						break
			else:
				ar = remove_zeros(Q_copy[k],0)
				sorted_ar = sorted(ar.items(), key=operator.
				    ↪ itemgetter(1),reverse = True)
				print("Grid",k," idx",sorted_ar[0][0])


def remove_zeros(ar,val):
	map = dict()
	for i in range(0,len(ar)):
		if not ar[i]==val:
			map[i] = ar[i]
	return map

def find_next_state(x,y,action,d_shape):
	if action==0:
		next = d_shape*(x-1)+(y-1)
	elif action==1:
		next = d_shape*(x-1)+y
	elif action==2:
		next = d_shape*(x-1)+(y+1)
```

```
        elif action==3:
                next = d_shape*x + (y-1)
        elif action==4:
                next = d_shape*x + (y+1)
        elif action==5:
                next = d_shape*(x+1) + (y-1)
        elif action==6:
                next = d_shape*(x+1) + y
        else:
                next = d_shape*(x+1) + (y+1)
        return next


def calc_index(num,d_shape):
        x = int(num/d_shape)
        y = num%d_shape
        return x,y

if __name__=="__main__":
        q_learn(5000,0.06,0.9,0.4,5)
```