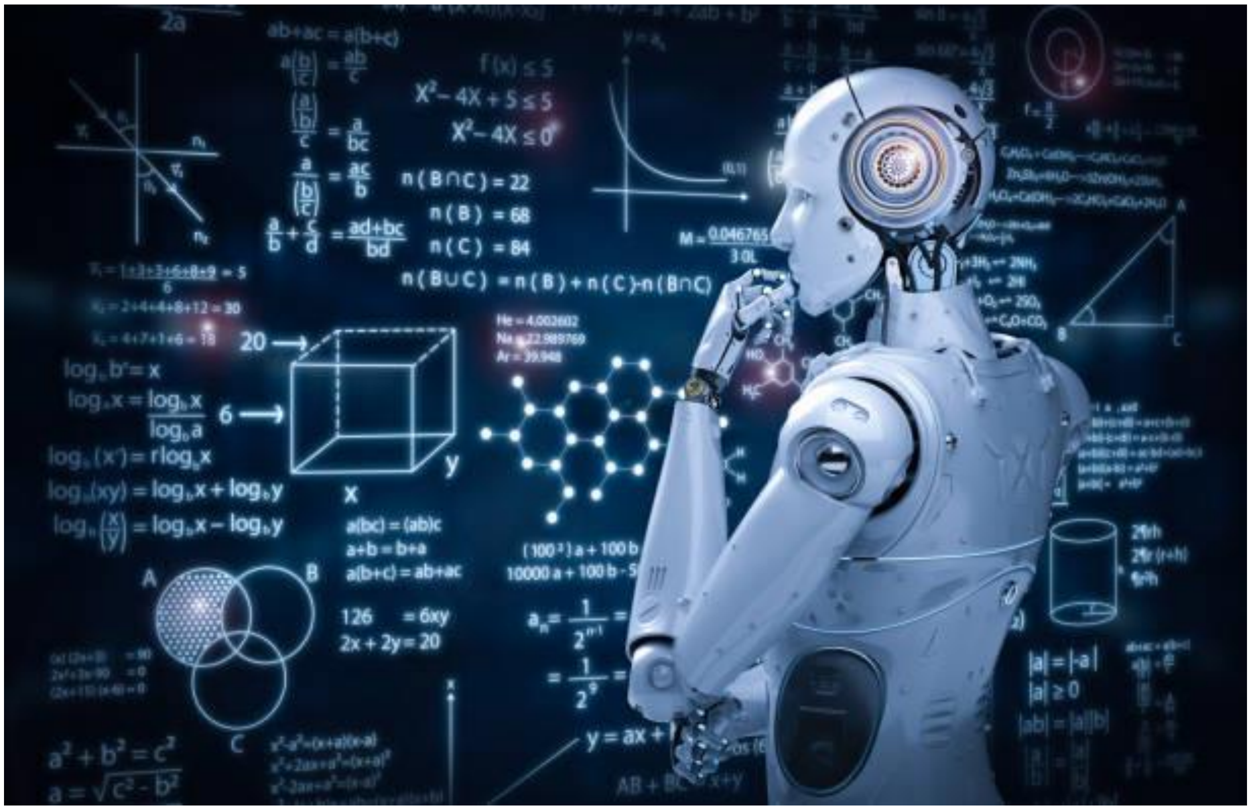
Reinforcement Learning: with Python

D.S.P.A.U.M.PEIRIS

S14490

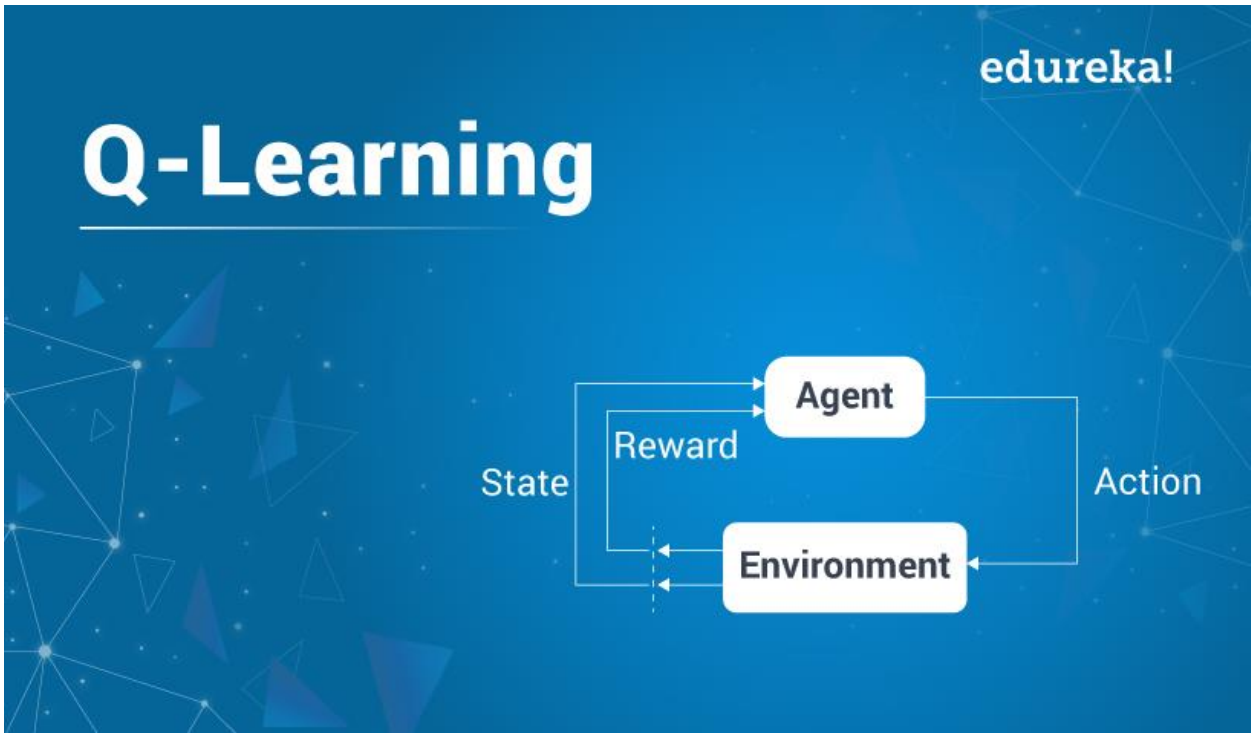


What is reinforcement learning?

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward.

Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.

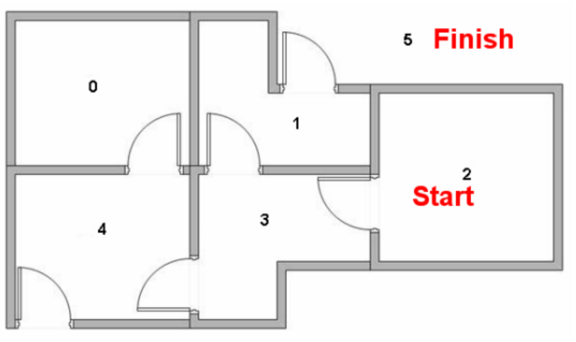
What is Q-learning?



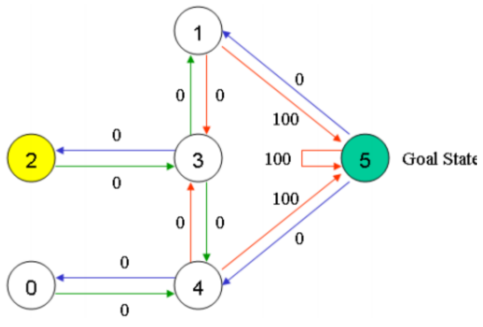
Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It’s considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn’t needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

Part 2

An agent has to travel from one room (start state) to another (goal state) in the following house



Representing the goal using immediate rewards

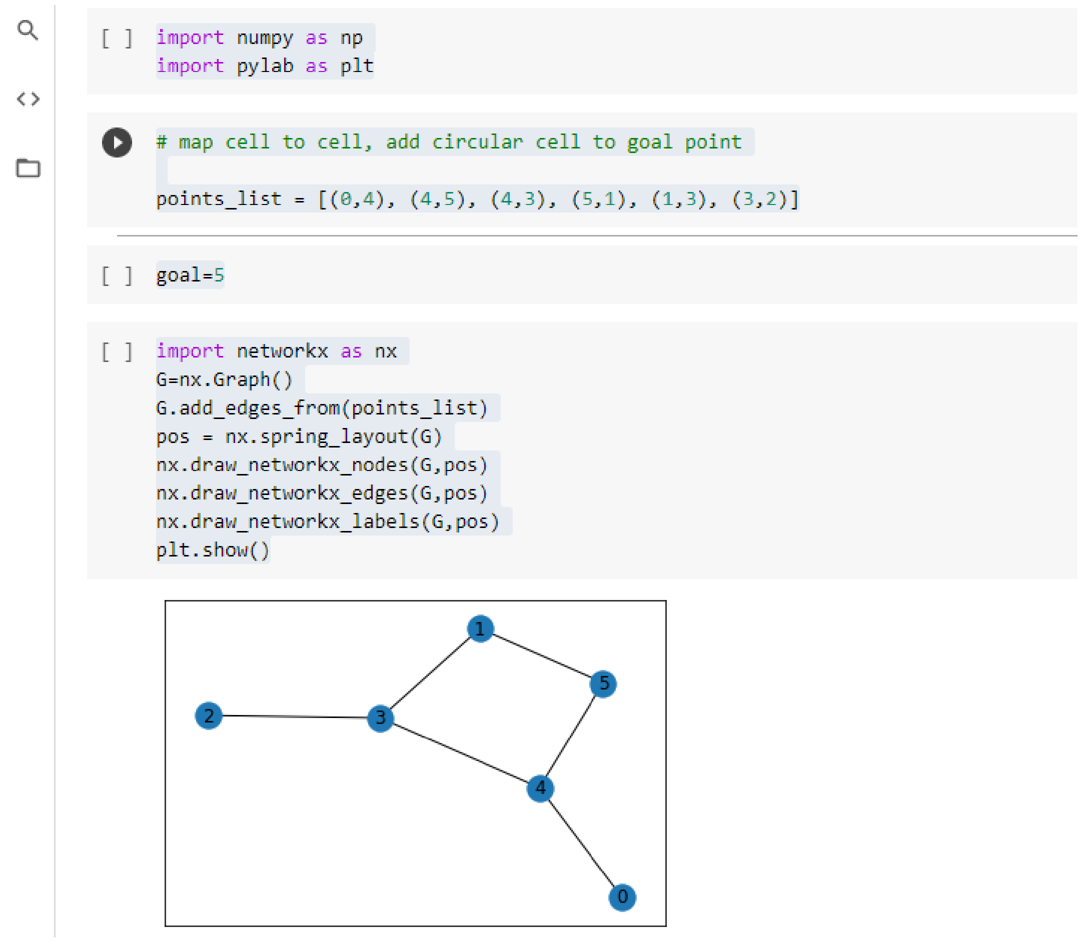


What paths are available?
What is the best path?

<u>Google colab Code:</u>

https://colab.research.google.com/drive/1AeCQADifRCXqG1Mrvve_p727TNQ1XL6e?usp=sharing

Creating Graph (python)

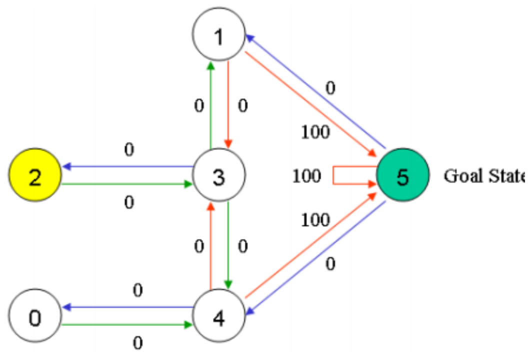


```
[ ] import numpy as np
    import pylab as plt
```

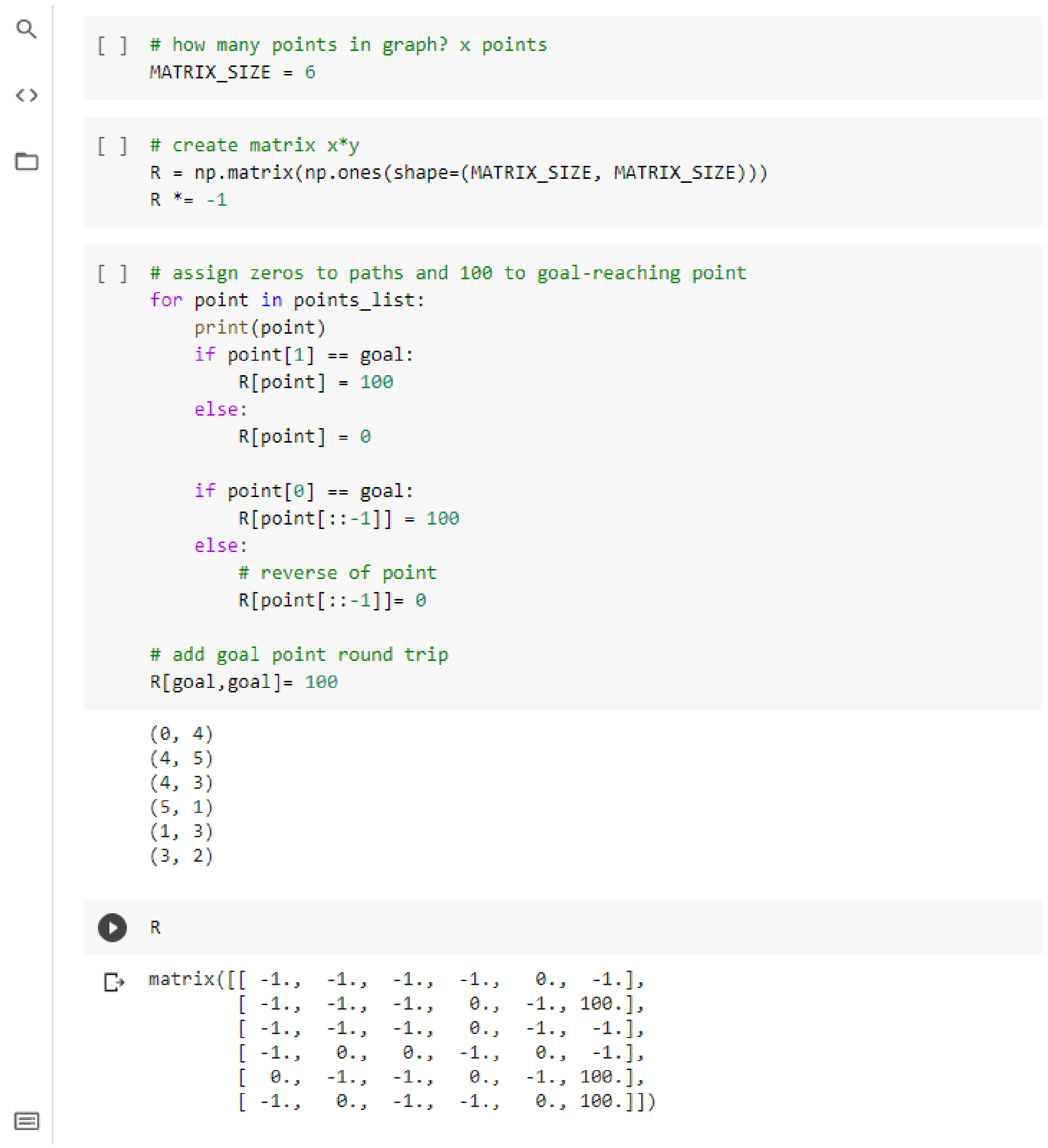
```
    # map cell to cell, add circular cell to goal point

    points_list = [(0,4), (4,5), (4,3), (5,1), (1,3), (3,2)]
```

```
[ ] goal=5
```

```
[ ] import networkx as nx
    G=nx.Graph()
    G.add_edges_from(points_list)
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G,pos)
    nx.draw_networkx_edges(G,pos)
    nx.draw_networkx_labels(G,pos)
    plt.show()
```



Creating Matrix (python)



```
[ ]  # how many points in graph? x points
     MATRIX_SIZE = 6
```

```
[ ]  # create matrix x*y
     R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))
     R *= -1
```

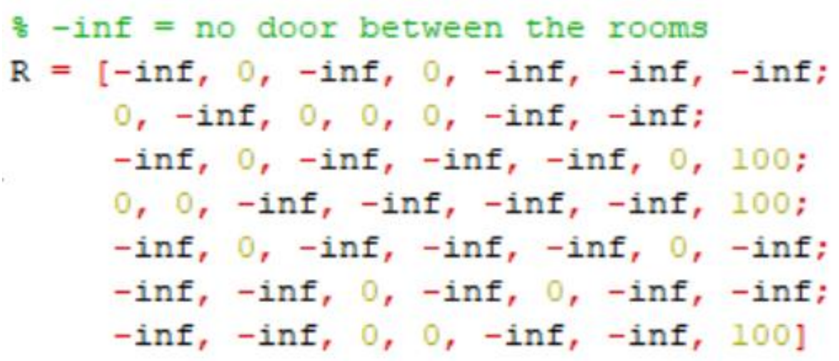
```
[ ]  # assign zeros to paths and 100 to goal-reaching point
     for point in points_list:
         print(point)
         if point[1] == goal:
             R[point] = 100
         else:
             R[point] = 0

         if point[0] == goal:
             R[point[::-1]] = 100
         else:
             # reverse of point
             R[point[::-1]]= 0

     # add goal point round trip
     R[goal,goal]= 100
```

```
(0, 4)
(4, 5)
(4, 3)
(5, 1)
(1, 3)
(3, 2)
```

```
R
```

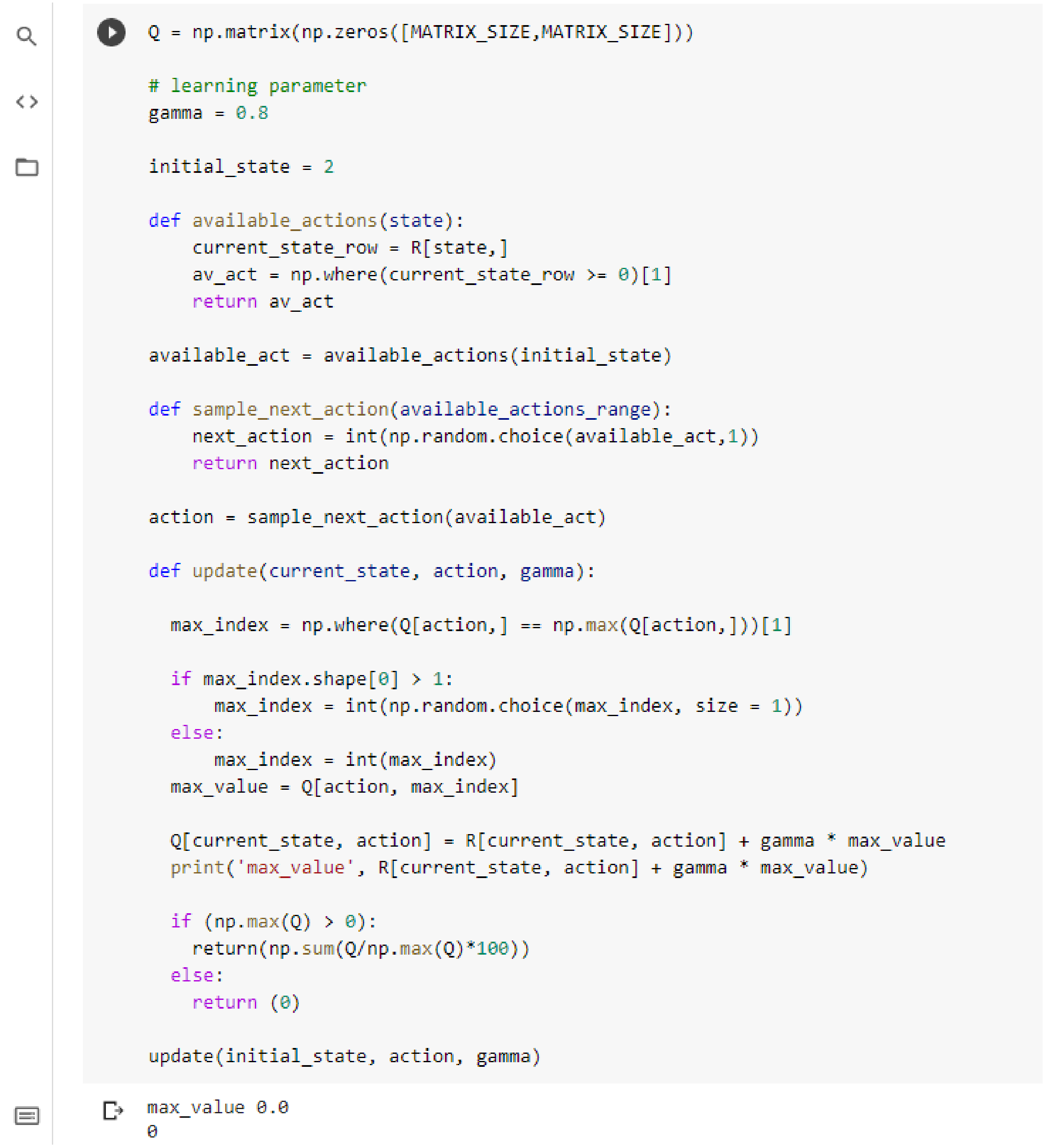
```
matrix([[ -1.,  -1.,  -1.,  -1.,   0.,  -1.],
        [ -1.,  -1.,  -1.,   0.,  -1., 100.],
        [ -1.,  -1.,  -1.,   0.,  -1.,  -1.],
        [ -1.,   0.,   0.,  -1.,   0.,  -1.],
        [  0.,  -1.,  -1.,   0.,  -1., 100.],
        [ -1.,   0.,  -1.,  -1.,   0., 100.]])
```

Octave (example not in above)



```
% -inf = no door between the rooms
R = [-inf, 0, -inf, 0, -inf, -inf, -inf;
     0, -inf, 0, 0, 0, -inf, -inf;
     -inf, 0, -inf, -inf, -inf, 0, 100;
     0, 0, -inf, -inf, -inf, -inf, 100;
     -inf, 0, -inf, -inf, -inf, 0, -inf;
     -inf, -inf, 0, -inf, 0, -inf, -inf;
     -inf, -inf, 0, 0, -inf, -inf, 100]
```

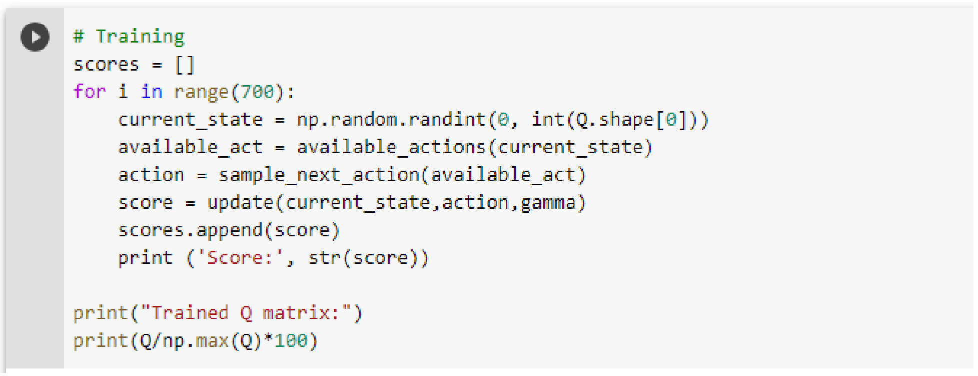
Exploration (python)



```
Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

# learning parameter
gamma = 0.8

initial_state = 2

def available_actions(state):
    current_state_row = R[state,]
    av_act = np.where(current_state_row >= 0)[1]
    return av_act

available_act = available_actions(initial_state)

def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_act,1))
    return next_action

action = sample_next_action(available_act)

def update(current_state, action, gamma):

  max_index = np.where(Q[action,] == np.max(Q[action,]))[1]

  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]

  Q[current_state, action] = R[current_state, action] + gamma * max_value
  print('max_value', R[current_state, action] + gamma * max_value)

  if (np.max(Q) > 0):
    return(np.sum(Q/np.max(Q)*100))
  else:
    return (0)

update(initial_state, action, gamma)
```

```
max_value 0.0
0
```

Exploiting (python)



```python
# Training
scores = []
for i in range(700):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_act = available_actions(current_state)
    action = sample_next_action(available_act)
    score = update(current_state,action,gamma)
    scores.append(score)
    print ('Score:', str(score))

print("Trained Q matrix:")
print(Q/np.max(Q)*100)
```

```
Trained Q matrix:
[[  0.           0.           0.           0.          79.9999946
    0.        ]
 [  0.           0.           0.          63.99999568   0.
  100.        ]
 [  0.           0.           0.          63.99999568   0.
    0.        ]
 [  0.          80.          51.19999222   0.          79.99998785
    0.        ]
 [ 63.99999568   0.           0.          64.           0.
  100.        ]
 [  0.          79.99993921   0.           0.          80.
  100.        ]]
```

Octave

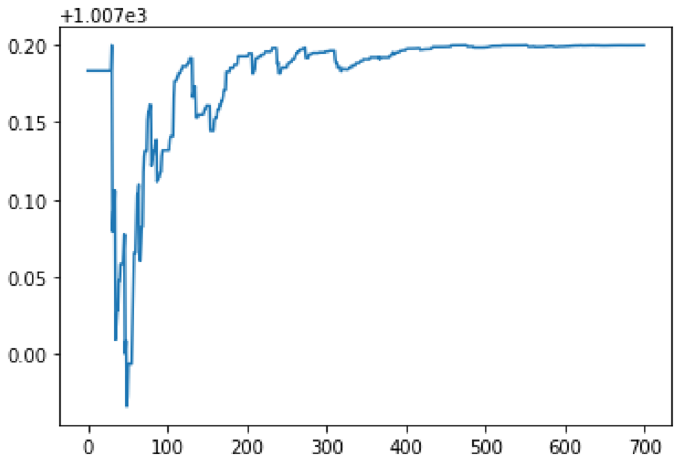
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Divided by the max

Testing

```
# Testing
current_state = 2
steps = [current_state]

while current_state != 5:

    next_step_index = np.where(Q[current_state,]
        == np.max(Q[current_state,]))[1]

    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))

    else:
        next_step_index = int(next_step_index)

    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

plt.plot(scores)
plt.show()
```

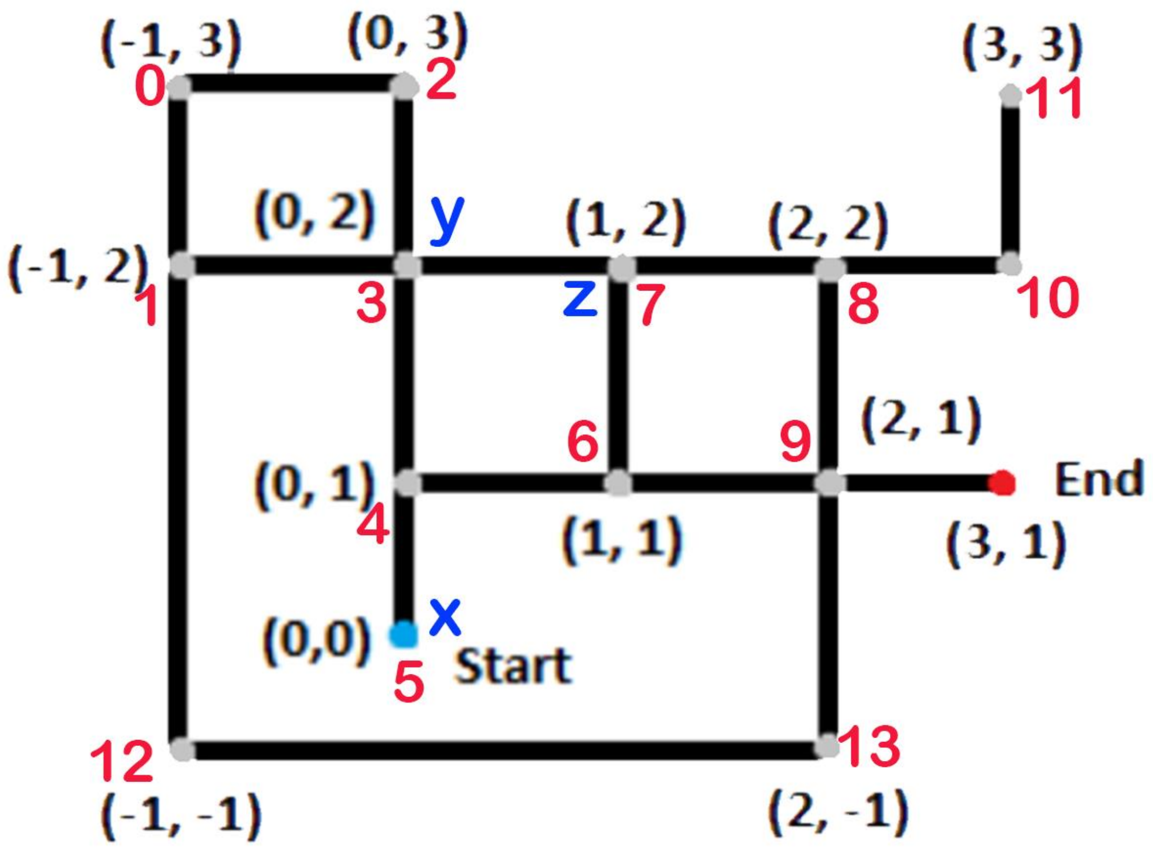
```
Most efficient path:
[2, 3, 1, 5]
```



When we compare result/output of Octave vs Python,

we get same output with both but it will be little bit different when considering the Q-table.

Part 3

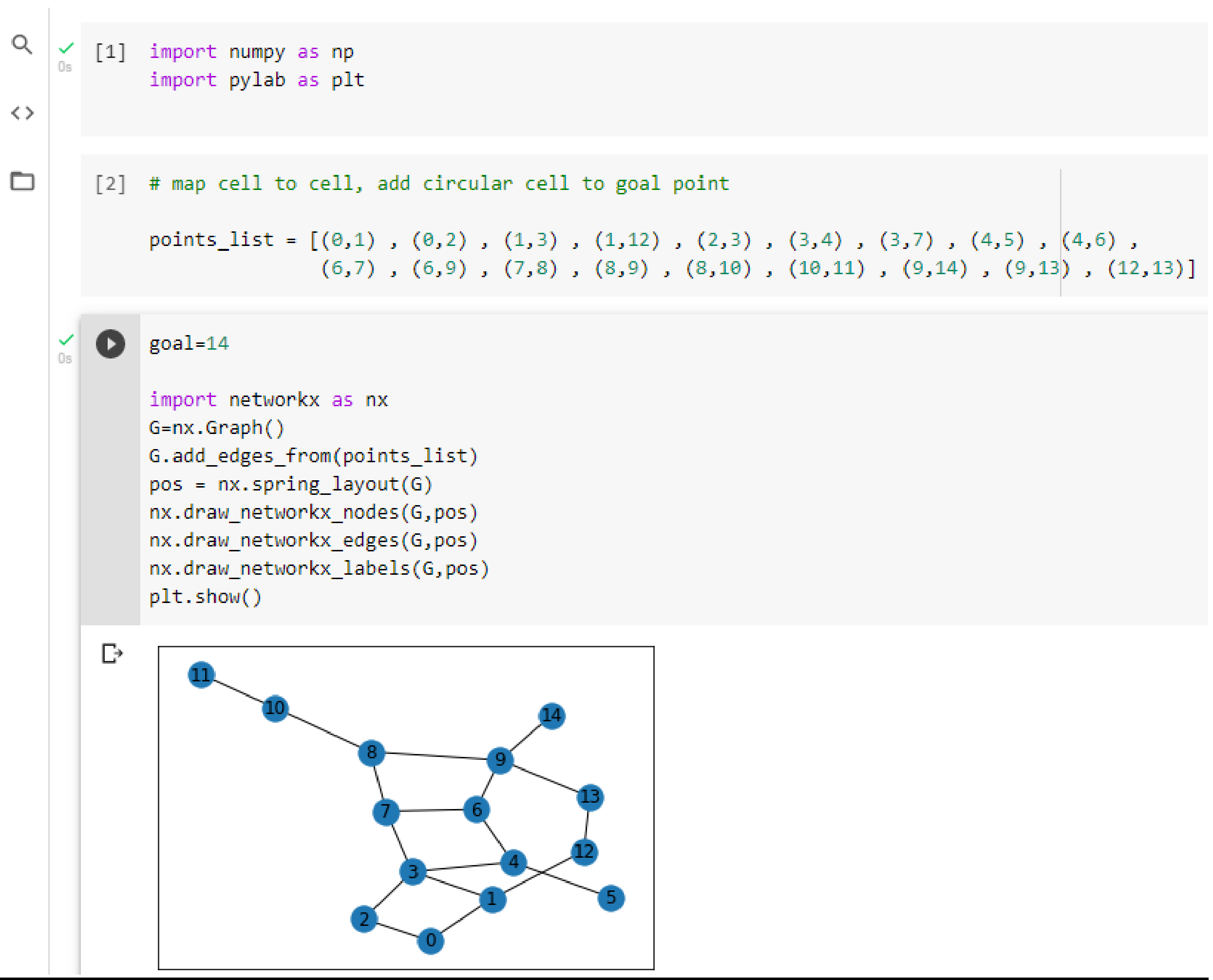


➢ **You are required to build a maze solving robot who should be able to reach the end point when its placed at any starting point. The layout of the maze (paths) is given below**

Google colab code:-

https://colab.research.google.com/drive/1uwMZz0jxH1pwxA6ZzwAl_ef3v0Sx9-oZ?usp=sharing

Graph

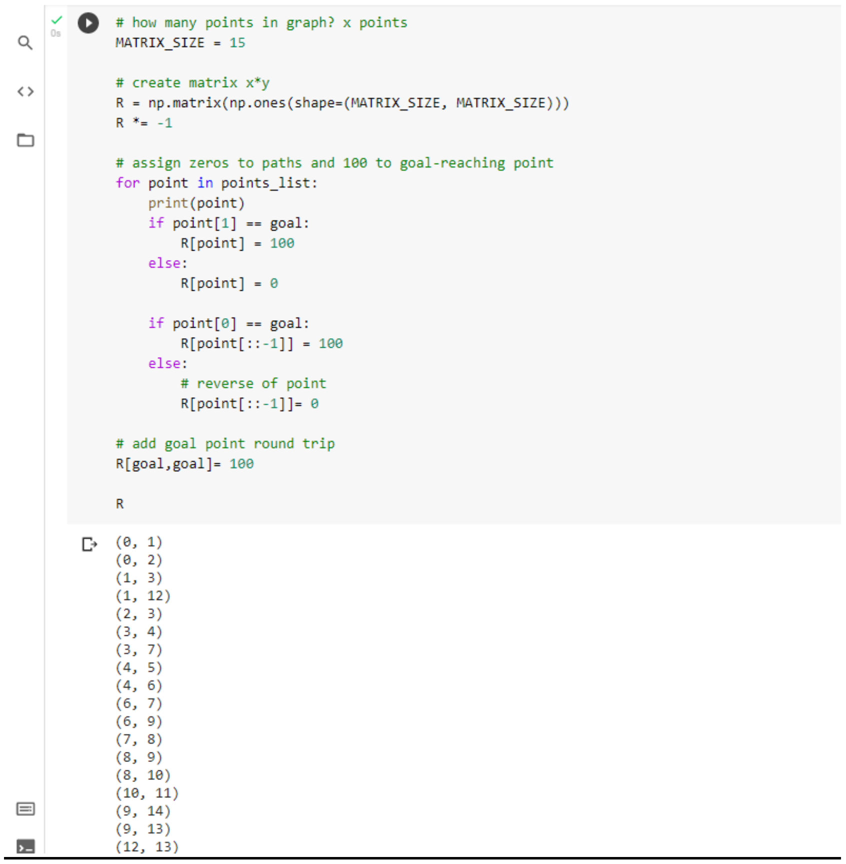


```
[1]  import numpy as np
     import pylab as plt
```

```
[2]  # map cell to cell, add circular cell to goal point

     points_list = [(0,1) , (0,2) , (1,3) , (1,12) , (2,3) , (3,4) , (3,7) , (4,5) , (4,6) ,
                    (6,7) , (6,9) , (7,8) , (8,9) , (8,10) , (10,11) , (9,14) , (9,13) , (12,13)]
```

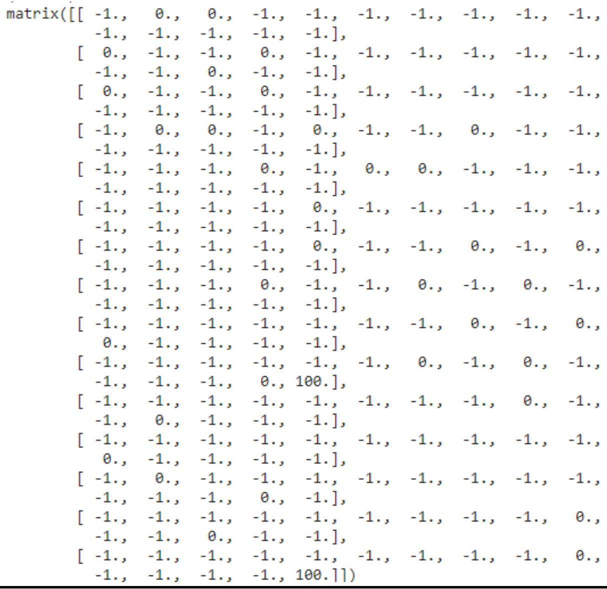
```
goal=14

import networkx as nx
G=nx.Graph()
G.add_edges_from(points_list)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G,pos)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
plt.show()
```

Making Matrix



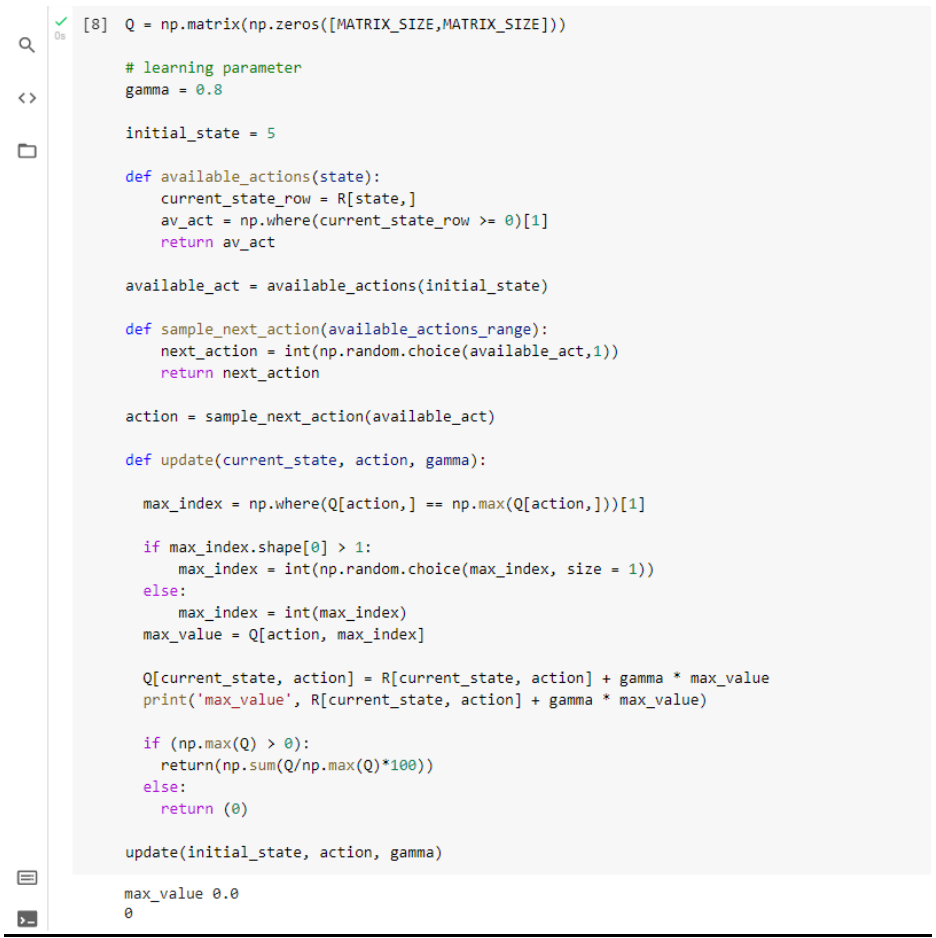
```
# how many points in graph? x points
MATRIX_SIZE = 15

# create matrix x*y
R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))
R *= -1

# assign zeros to paths and 100 to goal-reaching point
for point in points_list:
    print(point)
    if point[1] == goal:
        R[point] = 100
    else:
        R[point] = 0

    if point[0] == goal:
        R[point[::-1]] = 100
    else:
        # reverse of point
        R[point[::-1]]= 0

# add goal point round trip
R[goal,goal]= 100

R
```

```
(0, 1)
(0, 2)
(1, 3)
(1, 12)
(2, 3)
(3, 4)
(3, 7)
(4, 5)
(4, 6)
(6, 7)
(6, 9)
(7, 8)
(8, 9)
(8, 10)
(10, 11)
(9, 14)
(9, 13)
(12, 13)
```



```
matrix([[ -1.,   0.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [  0.,  -1.,  -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,
          -1.,  -1.,   0.,  -1.,  -1.],
        [  0.,  -1.,  -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,   0.,   0.,  -1.,   0.,  -1.,  -1.,   0.,  -1.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,   0.,  -1.,   0.,   0.,  -1.,  -1.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,   0.,  -1.,  -1.,   0.,  -1.,   0.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,   0.,  -1.,  -1.,   0.,  -1.,   0.,  -1.,
          -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,   0.,  -1.,   0.,
           0.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,   0.,  -1.,   0.,  -1.,
          -1.,  -1.,  -1.,   0., 100.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,   0.,  -1.,
          -1.,   0.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,
           0.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,
          -1.,  -1.,  -1.,   0.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,   0.,
          -1.,  -1.,   0.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,   0.,
          -1.,  -1.,  -1.,  -1., 100.]])
```

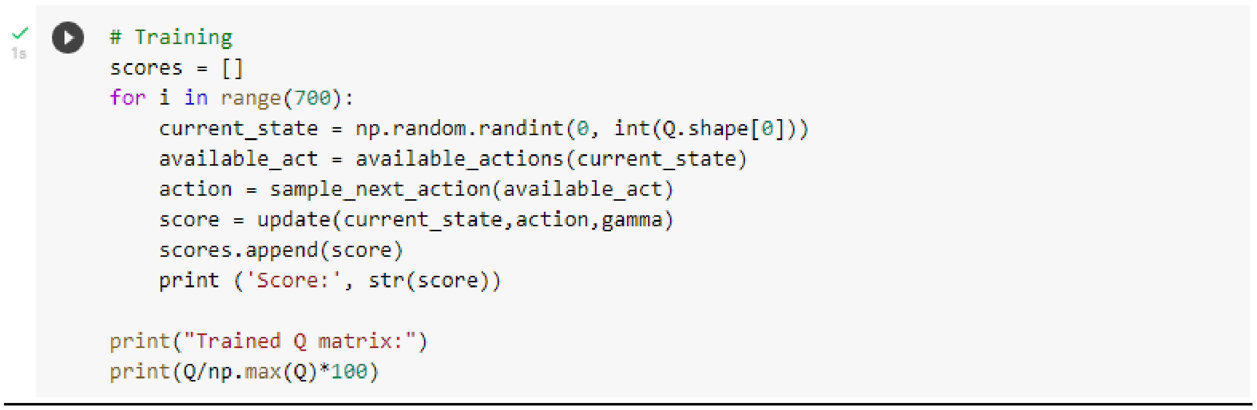
Exploration



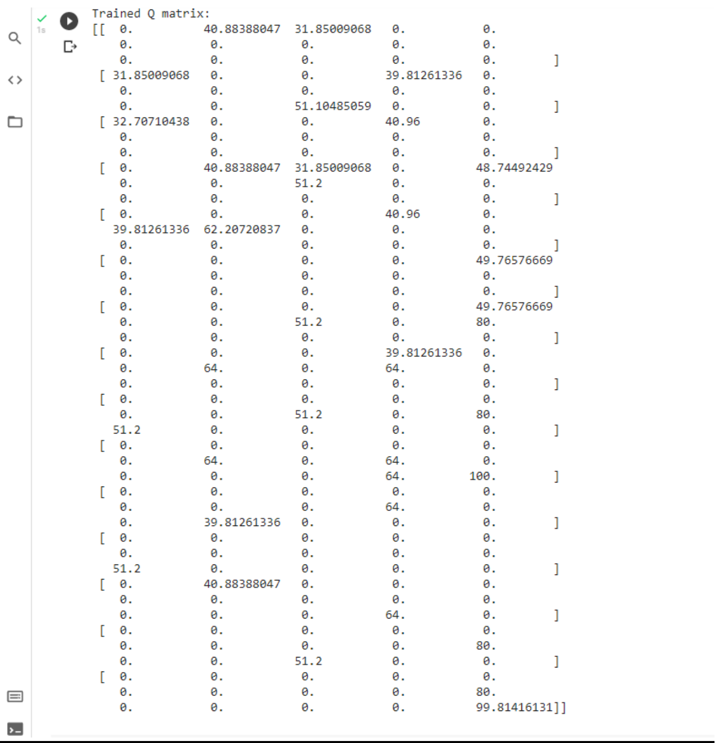
```
[8] Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

    # learning parameter
    gamma = 0.8

    initial_state = 5

    def available_actions(state):
        current_state_row = R[state,]
        av_act = np.where(current_state_row >= 0)[1]
        return av_act

    available_act = available_actions(initial_state)

    def sample_next_action(available_actions_range):
        next_action = int(np.random.choice(available_act,1))
        return next_action

    action = sample_next_action(available_act)

    def update(current_state, action, gamma):

      max_index = np.where(Q[action,] == np.max(Q[action,]))[1]

      if max_index.shape[0] > 1:
          max_index = int(np.random.choice(max_index, size = 1))
      else:
          max_index = int(max_index)
      max_value = Q[action, max_index]

      Q[current_state, action] = R[current_state, action] + gamma * max_value
      print('max_value', R[current_state, action] + gamma * max_value)

      if (np.max(Q) > 0):
        return(np.sum(Q/np.max(Q)*100))
      else:
        return (0)

    update(initial_state, action, gamma)
```

```
max_value 0.0
0
```

Exploiting

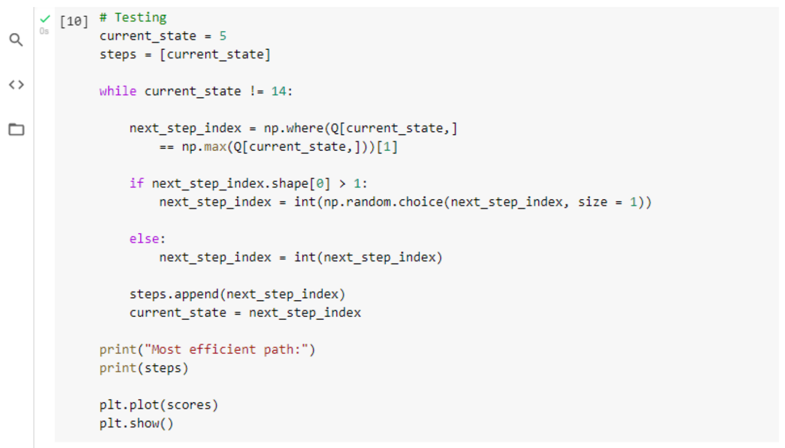


```
# Training
scores = []
for i in range(700):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_act = available_actions(current_state)
    action = sample_next_action(available_act)
    score = update(current_state,action,gamma)
    scores.append(score)
    print ('Score:', str(score))

print("Trained Q matrix:")
print(Q/np.max(Q)*100)
```



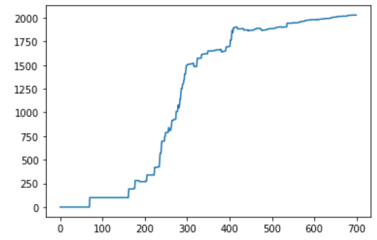
```
Trained Q matrix:
[[  0.          40.88388047  31.85009068   0.           0.
    0.           0.           0.           0.           0.
    0.           0.           0.           0.           0.        ]
 [ 31.85009068   0.           0.          39.81261336   0.
    0.           0.           0.           0.           0.
    0.           0.          51.10485059   0.           0.        ]
 [ 32.70710438   0.           0.          40.96         0.
    0.           0.           0.           0.           0.
    0.           0.           0.           0.           0.        ]
 [  0.          40.88388047  31.85009068   0.          48.74492429
    0.           0.          51.2          0.           0.
    0.           0.           0.           0.           0.        ]
 [  0.           0.           0.          40.96         0.
   39.81261336  62.20720837   0.           0.           0.
    0.           0.           0.           0.           0.        ]
 [  0.           0.           0.           0.          49.76576669
    0.           0.           0.           0.           0.
    0.           0.           0.           0.           0.        ]
 [  0.           0.           0.           0.          49.76576669
    0.           0.          51.2          0.          80.
    0.           0.           0.           0.           0.        ]
 [  0.           0.           0.          39.81261336   0.
    0.          64.           0.          64.           0.
    0.           0.           0.           0.           0.        ]
 [  0.           0.           0.           0.           0.
    0.           0.          51.2          0.          80.
   51.2          0.           0.           0.           0.        ]
 [  0.           0.           0.           0.           0.
    0.          64.           0.          64.           0.
    0.           0.           0.          64.         100.        ]
 [  0.           0.           0.           0.           0.
    0.           0.           0.          64.           0.
    0.          39.81261336   0.           0.           0.        ]
 [  0.           0.           0.           0.           0.
    0.           0.           0.           0.           0.
   51.2          0.           0.           0.           0.        ]
 [  0.          40.88388047   0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.           0.          64.           0.        ]
 [  0.           0.           0.           0.           0.
    0.           0.           0.           0.          80.
    0.           0.          51.2          0.           0.        ]
 [  0.           0.           0.           0.           0.
    0.           0.           0.           0.          80.
    0.           0.           0.           0.          99.81416131]]
```

Testing



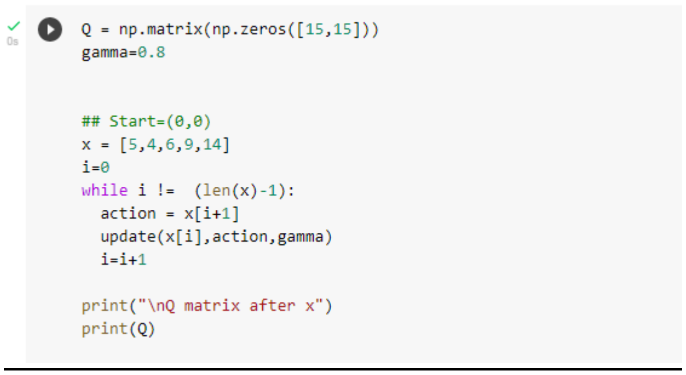
```
[10] # Testing
     current_state = 5
     steps = [current_state]

     while current_state != 14:

         next_step_index = np.where(Q[current_state,]
             == np.max(Q[current_state,]))[1]

         if next_step_index.shape[0] > 1:
             next_step_index = int(np.random.choice(next_step_index, size = 1))

         else:
             next_step_index = int(next_step_index)

         steps.append(next_step_index)
         current_state = next_step_index

     print("Most efficient path:")
     print(steps)

     plt.plot(scores)
     plt.show()
```

```
Most efficient path:
[5, 4, 6, 9, 14]
```

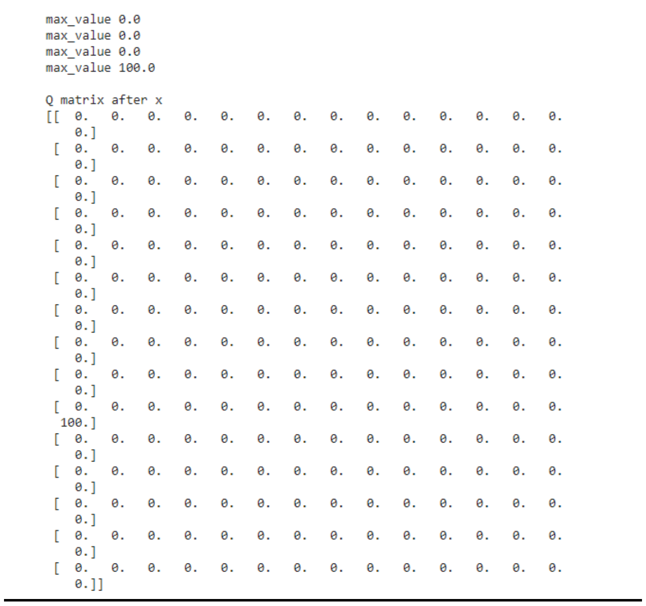


- **Give the resulting q-tables with calculations after the following state transitions during the exploration.**
  - **Start = (0, 0); Actions: UP, RIGHT, RIGHT, RIGHT**
  - **Start = (0, 2), Actions: DOWN, RIGHT, UP, RIGHT, DOWN, RIGHT**
  - **Start = (1, 2); Actions: DOWN, RIGHT, RIGHT**

1. Start = (0, 0); Actions: UP, RIGHT, RIGHT, RIGHT

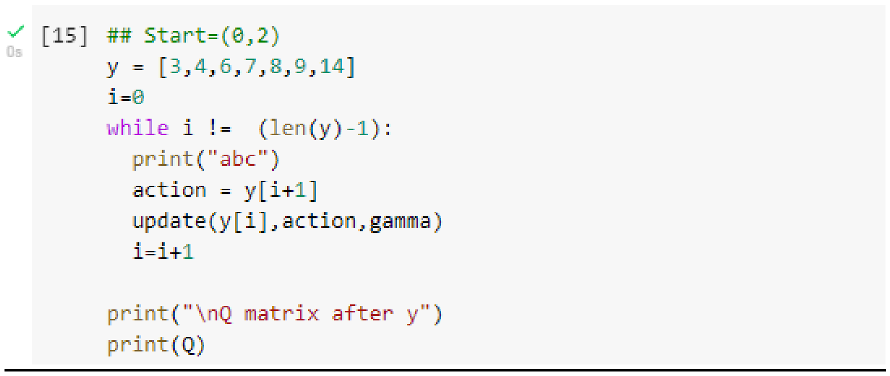


```
Q = np.matrix(np.zeros([15,15]))
gamma=0.8


## Start=(0,0)
x = [5,4,6,9,14]
i=0
while i !=  (len(x)-1):
  action = x[i+1]
  update(x[i],action,gamma)
  i=i+1

print("\nQ matrix after x")
print(Q)
```

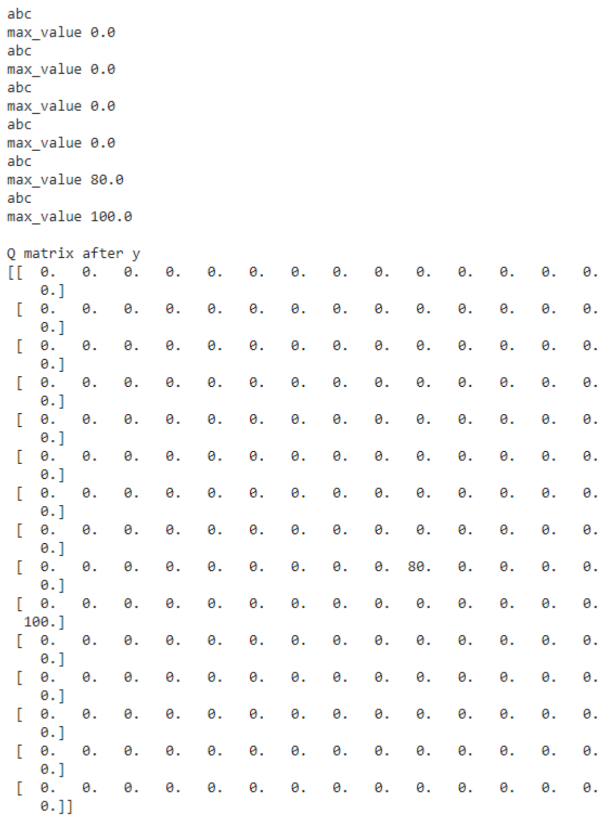


```
max_value 0.0
max_value 0.0
max_value 0.0
max_value 100.0

Q matrix after x
[[  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.   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.   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.   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.   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.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  100.]
 [  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.   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.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]]
```

2. Start = (0, 2), Actions: DOWN, RIGHT, UP, RIGHT, DOWN, RIGHT

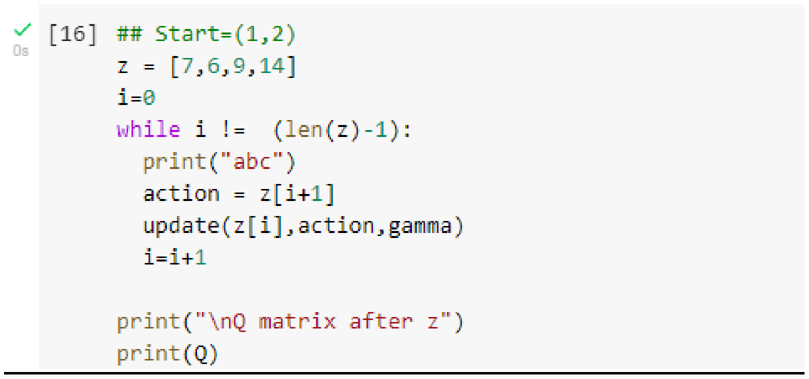


```
[15] ## Start=(0,2)
     y = [3,4,6,7,8,9,14]
     i=0
     while i !=  (len(y)-1):
       print("abc")
       action = y[i+1]
       update(y[i],action,gamma)
       i=i+1

     print("\nQ matrix after y")
     print(Q)
```

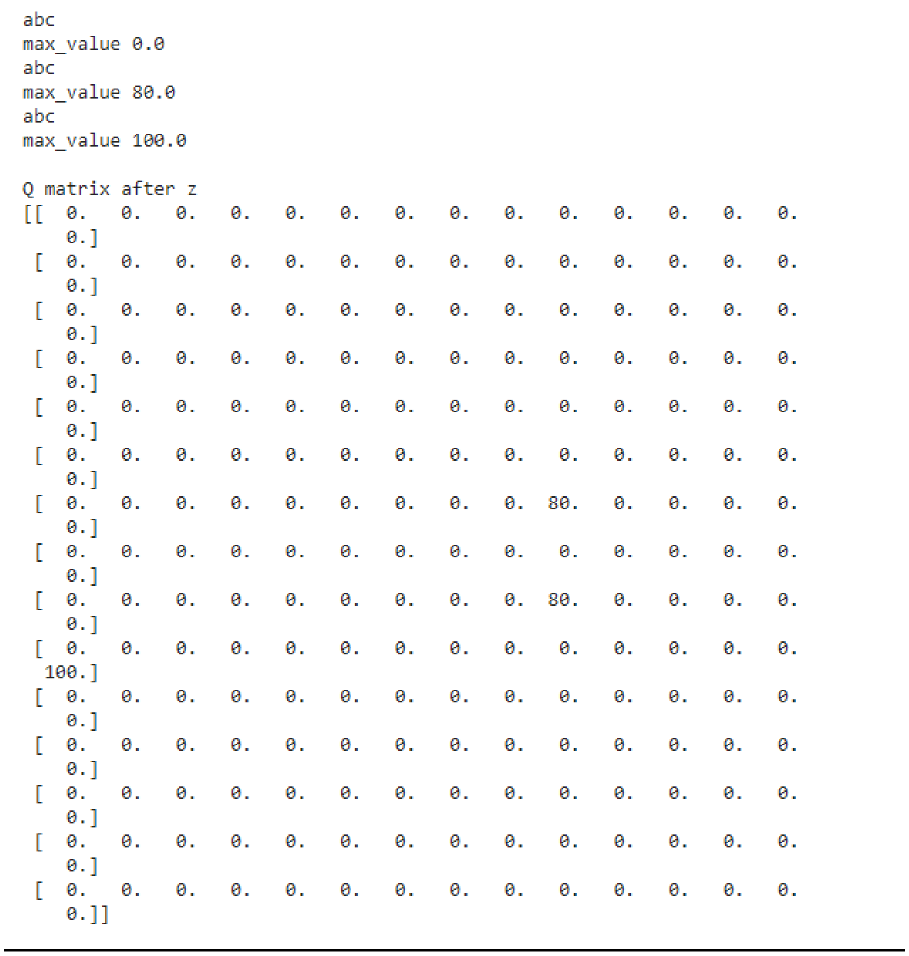


```
abc
max_value 0.0
abc
max_value 0.0
abc
max_value 0.0
abc
max_value 0.0
abc
max_value 80.0
abc
max_value 100.0

Q matrix after y
[[  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.   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.   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.   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.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.  80.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  100.]
 [  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.   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.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]]
```

3. Start = (1, 2); Actions: DOWN, RIGHT, RIGHT



```
[16] ## Start=(1,2)
     z = [7,6,9,14]
     i=0
     while i !=  (len(z)-1):
       print("abc")
       action = z[i+1]
       update(z[i],action,gamma)
       i=i+1

     print("\nQ matrix after z")
     print(Q)
```



```
abc
max_value 0.0
abc
max_value 80.0
abc
max_value 100.0

Q matrix after z
[[  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.   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.   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.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.  80.   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.   0.   0.  80.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  100.]
 [  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.   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.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]]
```

- **Now, what would be the sequence of states if the robot exploits to reach the goal starting from (0, 0)? Explain your answer**

When robot exploits it's Q matrix to reach the goal starting from (0,0) , robot will choose the path Action: UP , RIGHT , RIGHT , RIGHT [5 , 4 , 6 , 9 , 14] or [(0,0) ,(0,1) , (1,1) ,(2,1) , (3,1)]

because it is the most efficient path for it..

Reference:

- https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/
- https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
- https://www.edureka.co/blog/q-learning/
- GNU Octave Tutorial