Q Learning

In [1]:
```
#import libraries
import numpy as np
from tabulate import tabulate
```

In [2]:
```
#define the shape of the mini-world
rows = 4
columns = 6

#Create a 3D numpy array to hold the current Q-values for each state and action pair: Q(s, a)
#The array has 4 rows and 6 columns and each cell is a list of actions defined below
q_values = np.zeros((rows, columns, 4))
```

In [3]:
```
#define actions
#numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left
actions = ['up', 'right', 'down', 'left']
```

In [4]:
```
#Created a 2D numpy array to hold the rewards for each state.
#The array contains 4 rows and 6 columns
# each value is initialized to 0.
rewards = np.full((rows, columns), 0)


#set the rewards for all aisle locations (i.e., white squares)
for i in range(rows):
  for j in range(columns):
    rewards[i, j] = 0

#Set the Loss states (4,5) and (4,6)
rewards[3,4] = -1
rewards[3,5] = -1

#Set the win state (2,6)
rewards[1, 5] = 1

print("Initial set of Rewards before training the model")
print(tabulate(rewards, tablefmt='fancy_grid'))
```

Initial set of Rewards before training the model

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	0
0	0	0	0	-1	-1

In [5]:

```
#define a function that returns true or flase if a state is terminal state
def is_terminal_state(row_index, column_index):
  #if the reward for this location is 0, then it is not a terminal state, hence return false
  if rewards[row_index, column_index] == 0:
    return False
  else:
    return True

#define an epsilon greedy algorithm that will choose which action to take next
def get_next_action(current_row_index, current_column_index, epsilon):
  #if a randomly chosen value between 0 and 1 is less than epsilon, then choose the most promising value from the Q-table for
  if np.random.random() < epsilon:
    return np.argmax(q_values[current_row_index, current_column_index])
  else: #choose a random action
    return np.random.randint(4)

#define a function that will get the next location based on the chosen action
def get_next_location(current_row_index, current_column_index, action_index):
  new_row_index = current_row_index
  new_column_index = current_column_index
  if actions[action_index] == 'up' and current_row_index > 0:
    new_row_index -= 1
  elif actions[action_index] == 'right' and current_column_index < columns - 1:
    new_column_index += 1
  elif actions[action_index] == 'down' and current_row_index < rows - 1:
    new_row_index += 1
  elif actions[action_index] == 'left' and current_column_index > 0:
    new_column_index -= 1
  return new_row_index, new_column_index

#define a function that will choose a random, non-terminal starting location
def get_starting_location():
  #get a random row and column index
  row_index = np.random.randint(rows)
  column_index = np.random.randint(columns)
  return row_index, column_index


#Function that calcultes the shortest path
def get_shortest_path(start_row_index, start_column_index):
  # Base case as return immediately if this is an invalid starting location
  if is_terminal_state(start_row_index, start_column_index):
    return []
  else: #if valid starting location
    current_row_index, current_column_index = start_row_index, start_column_index
    shortest_path = []
    shortest_path.append([current_row_index, current_column_index])
    #continue moving along the path until we reach the goal (i.e., the item packaging location)
    while not is_terminal_state(current_row_index, current_column_index):
      #get the best action to take
      action_index = get_next_action(current_row_index, current_column_index, 1.)
      #move to the next location on the path, and add the new location to the list
      current_row_index, current_column_index = get_next_location(current_row_index, current_column_index, action_index)
      shortest_path.append([current_row_index, current_column_index])
    return shortest_path
```

In [20]:

```
#define training parameters
epsilon = 0.8 #the probability that it will take a best action instead of a random action
discount_factor = 0.9 #discount factor for future rewards
learning_rate = 0.9 #the rate at which the AI agent should learn

#The problem statement states to attemp for atleast 20 training paths, so lets train the model for 100
for episode in range(1000):
  #get the starting location for this episode
  row_index, column_index = get_starting_location()

  #continue taking actions (i.e., moving) until we reach a terminal state
  #(i.e., until we reach the item packaging area or crash into an item storage location)
  while not is_terminal_state(row_index, column_index):
    #choose which action to take (i.e., where to move next)
    action_index = get_next_action(row_index, column_index, epsilon)

    #perform the chosen action, and transition to the next state (i.e., move to the next location)
    old_row_index, old_column_index = row_index, column_index #store the old row and column indexes
    row_index, column_index = get_next_location(row_index, column_index, action_index)

    #receive the reward for moving to the new state, and calculate the temporal difference
    reward = rewards[row_index, column_index]
    old_q_value = q_values[old_row_index, old_column_index, action_index]

    #(R(s) + γ max Q(s0, a0) − Q(s, a))
    temporal_difference = reward + (discount_factor * np.max(q_values[row_index, column_index])) - old_q_value

    #Q(s, a) <- Q(s, a) + temporal_difference
    new_q_value = old_q_value + (learning_rate * temporal_difference)
    q_values[old_row_index, old_column_index, action_index] = new_q_value

print('Q Learning complete!')
```

```
Q Learning complete!
```

In [21]:

```
print(get_shortest_path(0, 0)) #starting at row 0, column 0
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [1, 5]]
```

In [22]:

```
print(get_shortest_path(2,1))
```

```
[[2, 1], [2, 2], [2, 3], [1, 3], [1, 4], [1, 5]]
```

In [23]:

```
print(q_values.shape)
```

```
(4, 6, 4)
```

In [28]:

```
print("The Utility Matrix is:")
print()
for i in range(4):
    for j in range(6):
        new_list = [round(item, 2) for item in q_values[i][j]]
        print( new_list, " | ", end=" ")

    print()
    print()
```

```
The Utility Matrix is:

[0.53, 0.59, 0.59, 0.53]  |  [0.59, 0.66, 0.66, 0.53]  |  [0.66, 0.73, 0.73, 0.59]  |  [0.73, 0.81, 0.81, 0.66]  |  [0.81,
0.9, 0.9, 0.73]  |  [0.9, 0.9, 1.0, 0.81]  |

[0.53, 0.66, 0.53, 0.48]  |  [0.59, 0.73, 0.59, 0.59]  |  [0.66, 0.81, 0.66, 0.66]  |  [0.73, 0.9, 0.73, 0.73]  |  [0.81, 1.
0, 0.81, 0.81]  |  [0.0, 0.0, 0.0, 0.0]  |

[0.59, 0.59, 0.0, 0.53]  |  [0.66, 0.66, 0.53, 0.53]  |  [0.73, 0.73, 0.59, 0.59]  |  [0.81, 0.81, 0.66, 0.66]  |  [0.9, 0.
9, -1.0, 0.73]  |  [1.0, 0.9, -1.0, 0.81]  |

[0.52, 0.53, 0.48, 0.0]  |  [0.59, 0.53, 0.53, 0.48]  |  [0.66, 0.63, 0.58, 0.0]  |  [0.73, -0.9, 0.66, 0.59]  |  [0.0, 0.0,
0.0, 0.0]  |  [0.0, 0.0, 0.0, 0.0]  |
```

In [25]:
```
arr = []

for i in range(4):
    for j in range(6):
        max = q_values[i][j][0]
        index = 0
        for k in range(4):
            if q_values[i][j][k] > max:
                max = q_values[i][j][k]
                index = k


        arr.append(index)

arrows_matrix = []

for i in arr:
    if(i == 0):
        arrows_matrix.append("↑")
    elif (i == 1):
        arrows_matrix.append("→")
    elif(i == 2):
        arrows_matrix.append("↓")
    elif(i == 3):
        arrows_matrix.append("←")

arrows_matrix[11] = 1
arrows_matrix[23] = -1
arrows_matrix[22] = -1

np_arrow = np.array(arrows_matrix).reshape(4,6)
```

In [26]:
```
print(tabulate(np_arrow, headers='Policy', tablefmt='fancy_grid'))
```

P	o	l	i	c	y
→	→	→	→	→	↓
→	→	→	→	→	1
→	→	→	↑	→	↑
→	↑	↑	↑	-1	-1

In [ ]:

ADP

In [122]:

```
import numpy as np

class Grid_Env:
    def __init__(self,width,height,start):
        self.width = width
        self.height = height
        self.i = start[0]
        self.j = start[1]

    def set(self,rewards, actions):
        self.rewards = rewards
        self.actions = actions

    def set_state(self,s):
        self.i = s[0]
        self.j = s[1]

    def current_state(self):
        return (self.i,self.j)

    def is_terminal(self,s):
        return s not in self.actions

    def move(self,action):
        '''
        checks if a action is possible, then moves in that direction
        '''
        if action in self.actions[self.i,self.j]:
            if action == 'U':
                self.i -= 1
            elif action == 'D':
                self.i += 1
            elif action == 'R':
                self.j += 1
            elif action == 'L':
                self.j -= 1
        #return reward if any
        return self.rewards.get((self.i,self.j),0)

    def all_states(self):
        return set(list(self.actions.keys()) + list(self.rewards.keys()))

# defining all possible actions that each state can perform
def grid():
    grd = Grid_Env(4, 6,(2,1))
    rewards = {(1, 5): 1, (3, 5): -1, (3,4): -1}
    actions = {
    (0, 0): ('D','R'),
    (0, 1): ('L','D', 'R'),
    (0, 2): ('L','D', 'R'),
    (0, 3): ('L','D', 'R'),
    (0, 4): ('L','D', 'R'),
    (0,5): ('L','D'),
    (1, 0): ('U', 'D','R'),
    (1, 1): ('U', 'D', 'R','L'),
    (1, 2): ('U', 'D', 'R','L'),
    (1, 3): ('U', 'D', 'R','L'),
    (1, 4): ('U', 'D', 'R','L'),
    (1, 5): ('U', 'D','L'),
    (2, 0): ('U', 'R','D'),
    (2, 1): ('U', 'D', 'R','L'),
    (2, 2): ('U', 'D', 'R','L'),
    (2, 3): ('U', 'D', 'R','L'),
    (2, 4): ('U', 'D', 'R','L'),
    (2, 5): ('U', 'D','L'),
    (3, 0): ('U', 'R'),
    (3, 1): ('L', 'R', 'U'),
    (3, 2): ('L', 'R', 'U'),
    (3, 3): ('L', 'R', 'U'),
    (3, 4): ('L', 'R', 'U'),
    (3, 5): ('L', 'U')
    }
    grd.set(rewards, actions)
    return grd
```

In [123]:

```
GAMMA = 0.9
ACTIONS = ('U','D','L','R')

SMALL_ENOUGH = 1e-4


if __name__ == '__main__':

    #we use the negative grid so we can make the agent as efficient as possible
    grid = grid()


    #state -> action
    #well randomly choose an action and update as we learn
    policy = {}
    for s in grid.actions.keys():
        policy[s] = np.random.choice(ACTIONS)

    #initialize V(s)
    V = {}
    states = grid.all_states()
    for s in states:
        if s in grid.actions:
            V[s] = np.random.random()
        else:
            #terminal state
            V[s] = 0

    #repeat until convergence
    #V[s] = max[a]{sum[s',r] {p(s',r|s,a)[r + GAMMA * V[s']] } }
    while True:
        biggest_change = 0
        for s in states:
            old_v = V[s]

            #V[s] only has value if not a terminal state
            if s in policy:
                new_v = float('-inf')

                for a in ACTIONS:
                    grid.set_state(s)
                    r = grid.move(a)
                    v = r + GAMMA * V[grid.current_state()]
                    if v > new_v:
                        new_v = v
                V[s] = new_v
                biggest_change = max(biggest_change, np.abs(old_v - V[s]))

        if biggest_change < SMALL_ENOUGH:
            break
    #find a policy that leads to optimal value function
    for s in policy.keys():
        best_a = None
        best_value = float('-inf')
        for a in ACTIONS:
            grid.set_state(s)
            r = grid.move(a)
            v = r + GAMMA * V[grid.current_state()]
            if v > best_value:
                 best_value = v
                 best_a = a
        policy[s] = best_a
```

In [124]:

```
l = list(policy.values())
l[11] = 1
l[22] = -1
l[23] = -1

for i in range(24):
    if(l[i] == 'R'):
        l[i] = '→'
    elif(l[i] == 'L'):
        l[i] = '←'
    elif(l[i] == 'U'):
        l[i] = '↑'
    elif(l[i] == 'D'):
        l[i] = '↓'

x = np.array(l)
x = x.reshape(4,6)
```

In [125]:
```python
from tabulate import tabulate
print(tabulate(x, headers='POLICY', tablefmt='fancy_grid'))
```

P	O	L	I	C	Y
↓	↓	↓	↓	↓	↓
→	→	→	→	→	1
↑	↑	↑	↑	↑	↑
↑	↑	↑	↑	-1	-1

In [126]:
```python
#Get shortest path
def get_shortest_path(i,j):
    start_point = (i,j)
    path = [start_point]
    while True:
        if x[i][j] == '1':
            break
        if x[i][j] == "↓":
            i = i+1
            path.append((i,j))
        elif x[i][j] == "↑":
            i = i-1
            path.append((i,j))
        elif x[i][j] == "→":
            j = j+1
            path.append((i,j))
        else:
            j = j-1
            path.append((i,j))
    return path
```

In [127]:
```python
print(get_shortest_path(0,0))
```

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

In [128]:
```python
print(get_shortest_path(2,1))
```

```
[(2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5)]
```

In [ ]:

The optimal path obtained in Q Learning and Adaptive Dynamic Programming are different. This is due to the following reasons:

- In case of passive ADP, the agent’s policy is fixed. In contrast to Q Learning, the agent needs to decide what’s the next step as there is no fixed policy. The goal of ADP is to execute the fixed policy while on the other hand Q Learning has to learn the optimal policy.
- Q-learning performs a simple update based on the observed transition only. It does not try to keep the Q values consistent between neighboring states.
- ADP tries to maintain the consistency in the utility values by adjusting them using the Bellman equations
- Thus, the optimal policies generated are different for Q learning and ADP which results in different shortest paths.