Untitled5

January 24, 2023

[15]:
```python
import numpy as np
# Define the states
location_to_state = {
    'L1' : 0,
    'L2' : 1,
    'L3' : 2,
    'L4' : 3,
    'L5' : 4,
    'L6' : 5,
    'L7' : 6,
    'L8' : 7,
    'L9' : 8
}
# Define the actions
actions = [0,1,2,3,4,5,6,7,8]
# Define the rewards
rewards = np.array([[0,1,0,0,0,0,0,0,0],
              [1,0,1,0,1,0,0,0,0],
              [0,1,0,0,0,1,0,0,0],
              [0,0,0,0,0,0,1,0,0],
              [0,1,0,0,0,0,0,1,0],
              [0,0,1,0,0,0,0,0,0],
              [0,0,0,1,0,0,0,1,0],
              [0,0,0,0,1,0,1,0,1],
              [0,0,0,0,0,0,0,1,0]])

# Maps indices to locations
state_to_location = dict((state,location) for location,state in␣
 ↪location_to_state.items())

# Initialize parameters
gamma = 0.7 # Discount factor
alpha = 0.9 # Learning rate

# Initializing Q-Values
Q = np.array(np.zeros([9,9]))
```

```
[12]: def get_optimal_route(start_location,end_location):
    # Copy the rewards matrix to new Matrix
    rewards_new = np.copy(rewards)

    # Get the ending state corresponding to the ending location as given
    ending_state = location_to_state[end_location]

    # With the above information automatically set the priority of
    # the given ending state to the highest one
    rewards_new[ending_state,ending_state] = 999

    # -----------Q-Learning algorithm-----------

    # Initializing Q-Values
    Q = np.array(np.zeros([9,9]))

    # Q-Learning process
    for i in range(1000):
        # Pick up a state randomly
        current_state = np.random.randint(0,9) # Python excludes the upper bound

        # For traversing through the neighbor locations in the maze
        playable_actions = []

        # Iterate through the new rewards matrix and get the actions > 0
        for j in range(9):
            if rewards_new[current_state,j] > 0:
                playable_actions.append(j)

        # Pick an action randomly from the list of playable actions
        # leading us to the next state
        next_state = np.random.choice(playable_actions)

        # Compute the temporal difference
        # The action here exactly refers to going to the next state
        TD = rewards_new[current_state,next_state] + gamma * Q[next_state,
↪                np.argmax(Q[next_state,])] - Q[current_state,next_state]

        # Update the Q-Value using the Bellman equation
        Q[current_state,next_state] += alpha * TD

    # Initialize the optimal route with the starting location
    route = [start_location]
    # We do not know about the next location yet, so initialize with the value
↪of
    # starting location
    next_location = start_location
```

```
    # We don't know about the exact number of iterations
    # needed to reach to the final location hence while loop will be a good␣
↪choice
    # for iteratiing

    while(next_location != end_location):
        # Fetch the starting state
        starting_state = location_to_state[start_location]

        # Fetch the highest value pertaining to starting state
        next_state = np.argmax(Q[starting_state,])

        # We got the index of the next state. But we need the corresponding␣
↪letter.
        next_location = state_to_location[next_state]
        route.append(next_location)

        # Update the starting location for the next iteration
        start_location = next_location

    return route
```

[13]:
```
print(get_optimal_route('L1', 'L9'))
```

```
['L1', 'L2', 'L5', 'L8', 'L9']
```

[16]:
```
def training(self, start_location, end_location, iterations):

    rewards_new = np.copy(self.rewards)

    ending_state = self.location_to_state[end_location]
    rewards_new[ending_state, ending_state] = 999

    for i in range(iterations):
        current_state = np.random.randint(0,9)
        playable_actions = []

        for j in range(9):
            if rewards_new[current_state,j] > 0:
                playable_actions.append(j)

        next_state = np.random.choice(playable_actions)
        TD = rewards_new[current_state,next_state] + self.gamma * self.
↪Q[next_state, np.argmax(self.Q[next_state,])]
        self.Q[current_state,next_state] += self.alpha * TD
```

```
    route = [start_location]
    next_location = start_location

    # Get the route
    self.get_optimal_route(start_location, end_location, next_location, route,␣
↪self.Q)
```

[17]:
```
def get_optimal_route(self, start_location, end_location, next_location, route,␣
↪Q):

    while(next_location != end_location):
        starting_state = self.location_to_state[start_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = self.state_to_location[next_state]
        route.append(next_location)
        start_location = next_location

    print(route)
```

[18]:
```
class QAgent():

    # Initialize alpha, gamma, states, actions, rewards, and Q-values
    def __init__(self, alpha, gamma, location_to_state, actions, rewards,␣
↪state_to_location, Q):

        self.gamma = gamma
        self.alpha = alpha

        self.location_to_state = location_to_state
        self.actions = actions
        self.rewards = rewards
        self.state_to_location = state_to_location

        self.Q = Q

    # Training the robot in the environment
    def training(self, start_location, end_location, iterations):

        rewards_new = np.copy(self.rewards)

        ending_state = self.location_to_state[end_location]
        rewards_new[ending_state, ending_state] = 999

        for i in range(iterations):
            current_state = np.random.randint(0,9)
            playable_actions = []
```

```
            for j in range(9):
                if rewards_new[current_state,j] > 0:
                    playable_actions.append(j)

            next_state = np.random.choice(playable_actions)
            TD = rewards_new[current_state,next_state] + \
                    self.gamma * self.Q[next_state, np.argmax(self.
↪Q[next_state,])] - self.Q[current_state,next_state]

            self.Q[current_state,next_state] += self.alpha * TD

        route = [start_location]
        next_location = start_location

        # Get the route
        self.get_optimal_route(start_location, end_location, next_location,␣
↪route, self.Q)

    # Get the optimal route
    def get_optimal_route(self, start_location, end_location, next_location,␣
↪route, Q):

        while(next_location != end_location):
            starting_state = self.location_to_state[start_location]
            next_state = np.argmax(Q[starting_state,])
            next_location = self.state_to_location[next_state]
            route.append(next_location)
            start_location = next_location

        print(route)
```

[19]:
```
qagent = QAgent(alpha, gamma, location_to_state, actions, rewards, ␣
↪state_to_location, Q)
qagent.training('L9', 'L1', 1000)
```

```
['L9', 'L8', 'L5', 'L2', 'L1']
```

[ ]:

[ ]:

[ ]: