Jackie Dutzy – Bailey Helfer

CIS 479

Program 3 – Reinforcement Learning

Screenshots:

Test 1 – Jackie Dutzy

Table of N(s, a):

```
Table of N(s, a)

  -50          -50          -50          -50          -50          -50          -50


               138          285          225          151
  -50       122   4957   266    258   223    239   5043   140       ####         -50
               150          9719         8724         145
                            421          641                       100
  -50          ####      396   15262  677    655       ####      109    102       -50
                            404          23985                     4056
                            346                                    245
  -50          ####      318   12212       +100        ####      229    225       -50
                            335                                    8317
               143          9181         15772        387          259
  -50       168   4972   297   1303   416    429   14138   391   9968   312       -50
               129          240          415          345          242


  -50          -50          -50          -50          -50          -50          -50
```

Table of Q(s, a):

```
Table of Q(s, a)

  -50          -50          -50          -50          -50          -50          -50


              -35.8        -34.1        -30.9        -27.8
  -50      -39.0   41.2   30.9   49.9   43.1   29.0   47.7   35.2     ####         -50
               27.4         60.6         69.9         42.5
                            53.7         61.4                       -1.1
  -50          ####       62.1   72.0   64.6   74.4     ####      6.3   -38.8      -50
                            71.5         85.1                        9.7
                            64.9                                    -0.2
  -50          ####       69.1   82.2      +100         ####      15.8   -33.3     -50
                            62.4                                    21.2
               28.0         66.9         79.8         46.3          12.8
  -50      -40.7   44.8   27.4   56.6   48.5   42.3   55.8   21.5   35.1   -41.4   -50
              -32.8        -26.8        -29.0        -25.9        -37.8


  -50          -50          -50          -50          -50          -50          -50
```

Optimal Policy:

-50	-50	-50	-50	-50	-50	-50
-50	>>>>	VVVV	VVVV	<<<<	####	-50
-50	####	>>>>	VVVV	####	VVVV	-50
-50	####	>>>>	+100	####	VVVV	-50
-50	>>>>	^^^^	^^^^	<<<<	<<<<	-50
-50	-50	-50	-50	-50	-50	-50

Test 2 – Bailey Helfer

Table of N(s, a):

```
Table of N(s, a)

    -50             -50             -50             -50             -50             -50             -50


                    134             280             245             133
    -50        128       4937  249        287   273        245  5129       124          ####            -50
                    115             9528            8775            152

                                    415             605                             135
    -50             ####       380       13324  613        574          ####       493       120        -50
                                    2264            22569                           3947

                                    403                                             217
    -50             ####       358       14538        +100              ####       356       248        -50
                                    402                                             8368

                    126            10628            14883           513             267
    -50        140       5130  278        287   395        416  14064      393   9898       231        -50
                    140             267             396             356             295


    -50             -50             -50             -50             -50             -50             -50
```

Table of Q(s, a):

```
Trial 50000
Table of Q(s, a)

    -50         -50         -50         -50         -50         -50         -50


             -31.7       -29.4       -28.6       -29.2

    -50    -36.4     40.2  27.4     49.6  42.1     36.3  48.5     30.7      ####        -50

              28.7        60.5        70.3        43.0

                          54.0        61.4                    -3.6

    -50       ####     62.1     72.1  65.2     74.4      ####     -6.3    -35.2      -50

                          71.4        85.2                    7.2

                          64.4                                -3.4

    -50       ####     69.2     82.2     +100        ####     7.3     -35.6      -50

                          62.4                                20.2

              32.6        67.1        80.0        35.6        9.0

    -50    -41.0     44.6  33.3     55.9  52.4     44.9  56.2     21.0  34.5    -39.9      -50

             -35.6       -25.7       -25.0       -26.3       -34.1


    -50         -50         -50         -50         -50         -50         -50
```

Optimal Policy:

```
-50     -50     -50     -50     -50     -50     -50

-50     >>>>    VVVV    VVVV    <<<<    ####    -50

-50     ####    >>>>    VVVV    ####    VVVV    -50

-50     ####    >>>>    +100    ####    VVVV    -50

-50     >>>>    ^^^^    ^^^^    <<<<    <<<<    -50

-50     -50     -50     -50     -50     -50     -50


C:\Users\bailley\source\repos\SCHOOL\CIS 479\P3>
```

Environment simulation was implemented with np.zeroes to create a 2D matrix to store the data to represent the maze. The terminal trap states in the array are defined with reward of -50, the goal terminal state in the array with reward of +100, and the obstacles of the maze are defined. The cost for moving southward is defined as 1, westward/eastward is 2, and northward is 3. The possibility to drift left or right of the desired action is defined as 0.1. If an obstacle is reached, the agent is returned to original position and is a terminal state is reached the agent is unable to move. Starting state, next state, and direction are all undefined.

The e-greedy algorithm was implemented by defining a random chance of action variable, and comparing this random chance of action to the epsilon value of 10% for a random action, and 90% for choosing the optimal action.

The Q-learning update was implemented by first incrementing the access frequency value and getting the current q-value, access frequency, and reward at each given state for the given action. If the next state is the goal state, the q value for the next state will be updated with goal reward (+100). If the next state is a trap state the q value for the next state is updated with the trap reward (-50). Otherwise the q value is updated with the maximum q value for the next state with the given action. The current q value data for each cell is then updated by following the equation updated q value = current q value + (1/ current access frequency) * (the reward state/action + (gamma * maximum q value of next state/action) – current q value.

Work Division:

We both worked on the environment simulation and defining the maze matrix, as well as generating the access frequencies. We also both worked on the e greedy algorithm. Bailey worked on most of the q-learning algorithm. Jackie worked on the output for the tables as well as the optimal policy.

Source:

```python
import numpy as np
import random as rnd

#---------------CONSTANT VARIABLES----------------#


#MAZE Initialization Properties
WIDTH = 7
HEIGHT = 6
GOAL_STATE = (3, 3)
GOAL_REWARD = 100
TRAP_STATES =  ((0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),
                (0,1),(0,2),(0,3),(0,4),(0,5),
                (6,1),(6,2),(6,3),(6,4),(6,5),
                (1,5),(2,5),(3,5),(4,5),(5,5))
TRAP_REWARD = -50
OBSTACLES = ((5,1),(1,2),(4,2),(1,3),(4,3))
OPEN_SPACES = [(x, y) for x in range(WIDTH) for y in range(HEIGHT) if (x, y) not in
OBSTACLES]
#Maze running sequence
MAX_TRIALS = 50000
MAX_STEPS = 100

#Logic for probability
# Discount factor
GAMMA = 0.9
# Chance of random action
EPSILON =  0.10
# Direction probabilies
PROB_STRAIGHT = 0.80
PROB_DRIFT = 0.1

#Formating for printing
SPACE_SIZE = 5
SPACE_OUTPUT = "     "
CELL_SPACE = "     "

# Define a random state to start from
def rnd_start_pos():

    rnd_pos = OPEN_SPACES[rnd.randrange(len(OPEN_SPACES))]

    # If terminal state is chosen
    while rnd_pos == GOAL_STATE or (rnd_pos in TRAP_STATES) :

        rnd_pos = OPEN_SPACES[rnd.randrange(len(OPEN_SPACES))]

    return rnd_pos

# Return reward for the current state with given action
def get_reward(state, direction):

    # South
    if direction == 3:
        return -1
    # West / East
```

```
    elif direction == 0 or direction == 2:
        return -2
    # North
    else:
        return -3

# Return the next state if action is completed without error
def get_new_state(present_state, direction):
    # West
    if (direction == 0):
        next_state = (present_state[0] - 1, present_state[1])
    # North
    elif (direction == 1):
        next_state = (present_state[0], present_state[1] - 1)
    # East
    elif (direction == 2):
        next_state = (present_state[0] + 1, present_state[1])
    # South
    elif (direction == 3):
        next_state = (present_state[0], present_state[1] + 1)

    # Condition if moves outside of maze or into an obstacle
    if (next_state in OBSTACLES or next_state[0] <= -1 or next_state[0] >= WIDTH or
next_state[1] <= -1 or next_state[1] >= HEIGHT):
        # Stay in present state
        return (present_state[0], present_state[1])
    else:
        return next_state

# Return the possible transitions at the state for the direction headed in
def get_poss_trans(state, direction):
    # Heading in intended direction
    state_straight = get_new_state(state, direction)

    # Drifting left of intended direction
    state_left = get_new_state(state, (direction - 1) % 4)

    # Drifting right of intended direction
    state_right = get_new_state(state, (direction + 1) % 4)

    return ((state_straight, PROB_STRAIGHT), (state_left, PROB_DRIFT), (state_right,
PROB_DRIFT))

# Return the new state given the direction headed in and possible drift
def get_poss_trans_drift(state, direction):

    # getting all transition possibility
    poss_trans = get_poss_trans(state, direction)

    # obtain random value
    random_val = rnd.random()

    for pt in poss_trans:

        # check if random value is within probability
        if random_val <= pt[1]:
            return pt[0]
        else:
```

```python
            random_val -= pt[1]

    print("Error with probability")

# Return all actions with the maximum q-value for the current state
def max_q_val(state, qvals):

    # Return q value for each action
    action_val = qvals[state[1], state[0]]

    # Return actions with the maximum q-value
    max_q_val = np.argwhere(action_val == np.amax(action_val))
    return max_q_val

# return action with the maximum q-value for the current state using tie breaker
def max_tie_break(state, qvals):

    action_val = max_q_val(state, qvals)

    # tie breaker
    tie_break = rnd.randrange(len(action_val))
    return action_val[tie_break]

# Possibility of action given random action chance and optimal action chance
def e_greedy(state, qvals):

    rnd_action_chance = rnd.random()

    # Chance to choose random action = 10%
    if rnd_action_chance <= EPSILON:
        return rnd.randrange(4)

    # Chance to choose optimal action = 90%
    else:
        return max_tie_break(state, qvals)

# Q-Learning
def q_learning(state, direction, next_state, access_frequency, qvals):

    # Increment the access frequency
    access_frequency[state[1], state[0]][direction] += 1

    # Get the q-value at the given state with given action
    curr_qval = qvals[state[1], state[0]][direction]

    # Get the access frequency at the given state with given action
    curr_freq = access_frequency[state[1], state[0]][direction]

    # Get the reward at the given state with given action
    reward = get_reward(state, direction)

    # If next state is goal state
    if next_state == GOAL_STATE:
        qval_next_state = GOAL_REWARD

    # If next state is trap state
    elif next_state in TRAP_STATES:
        qval_next_state = TRAP_REWARD
```

```
    else:
        # Get the maximum q-value at the next state with given action
        qval_next_state = qvals[next_state[1], next_state[0]][max_tie_break(next_state,
qvals)]

    # Get the new q-value for the next state with given action
    updated_qval = curr_qval + (1 / curr_freq) * (reward + (GAMMA * qval_next_state) -
curr_qval)
    qvals[state[1], state[0]][direction] = updated_qval

# output_table the data for each table
def output_table(data):

    for y, row in enumerate(data):

        for x, cell in enumerate(row):

            # Output North data
            if (x, y) in OBSTACLES or (x, y) == GOAL_STATE or (x,y) in TRAP_STATES:
                print(SPACE_OUTPUT + SPACE_OUTPUT + SPACE_OUTPUT + CELL_SPACE, end='')
            else:
                print(SPACE_OUTPUT, end='')
                print(str(round(cell[1], 1)).center(SPACE_SIZE), end='')
                print(SPACE_OUTPUT + CELL_SPACE, end='')

        print("\n")

        for x, cell in enumerate(row):

            # Output East/West data
            if (x, y) in OBSTACLES:
                print(SPACE_OUTPUT + "####".center(SPACE_SIZE) + SPACE_OUTPUT +
CELL_SPACE, end='')
            elif (x, y) == GOAL_STATE:
                print(SPACE_OUTPUT + ("+" + str(GOAL_REWARD)).center(SPACE_SIZE) +
SPACE_OUTPUT + CELL_SPACE, end='')
            elif (x,y) in TRAP_STATES:
                print(SPACE_OUTPUT + (str(TRAP_REWARD)).center(SPACE_SIZE) + SPACE_OUTPUT
+ CELL_SPACE, end='')
            else:
                print(str(round(cell[0], 1)).center(SPACE_SIZE), end='')
                print(SPACE_OUTPUT, end='')
                print(str(round(cell[2], 1)).center(SPACE_SIZE), end='')
                print(CELL_SPACE, end='')

        print("\n")

        for x, cell in enumerate(row):

            # Output South data
            if (x, y) in OBSTACLES or (x, y) == GOAL_STATE or (x,y) in TRAP_STATES:
                print(SPACE_OUTPUT + SPACE_OUTPUT + SPACE_OUTPUT + CELL_SPACE, end='')
            else:
                print(SPACE_OUTPUT, end='')
                print(str(round(cell[3], 1)).center(SPACE_SIZE), end='')
                print(SPACE_OUTPUT + CELL_SPACE, end='')
```

```
        print("\n")

# Output the optimal policy
def optimal_policy(data):

    for y, row in enumerate(data):

        for x, cell in enumerate(row):

            if (x, y) in OBSTACLES:
                # Obstacle
                print("####", end='')
            elif (x, y) == GOAL_STATE:
                # Terminal State
                print(("+"+str(GOAL_REWARD)).center(4), end='')
            elif (x,y) in TRAP_STATES:
                print((str(TRAP_REWARD)).center(4), end='')
            else:
                # Optimal Policy
                opt_pol = max_tie_break((x, y), data)
                if opt_pol==0:
                    print("<<<<", end='')
                elif opt_pol==1:
                    print("^^^^", end='')
                elif opt_pol==2:
                    print(">>>>", end='')
                elif opt_pol==3:
                    print("VVVV", end='')
            print("    ", end='')
        print("\n")

access_frequency = np.zeros((HEIGHT, WIDTH, 4), np.intc)
qvals = np.zeros((HEIGHT, WIDTH, 4), np.float64)
present_state = None
next_state = None
direction = None
steps = 0

# Run 50000 trials to generate data
for curr_trial in range(MAX_TRIALS):
    # Assign current state to random starting position (not the terminal state)
    present_state = rnd_start_pos()

    # Output the completed trial # by 5000
    if curr_trial % 5000 == 0:
        print('Trial ' + str(curr_trial + 1))

    # Complete trials until terminal state is reached
    while present_state != GOAL_STATE and (present_state not in TRAP_STATES) and steps <
MAX_STEPS:

        # Use e-greedy algorithm to determine the action at the present state
        direction = e_greedy(present_state, qvals)

        # Determine the next state with possibility of drifting
        next_state = get_poss_trans_drift(present_state, direction)

        # Update q-values
```

```
        q_learning(present_state, direction, next_state, access_frequency, qvals)

        # Increment steps count
        steps += 1

        # Continue to next state
        present_state = next_state

    # Reset step count
    steps = 0

# Output access frequency data
print("Table of N(s, a)")
output_table(access_frequency)
print()

# Output q-value data
print("Table of Q(s, a)")
output_table(qvals)
print()

# Output optimal policy
optimal_policy(qvals)
```