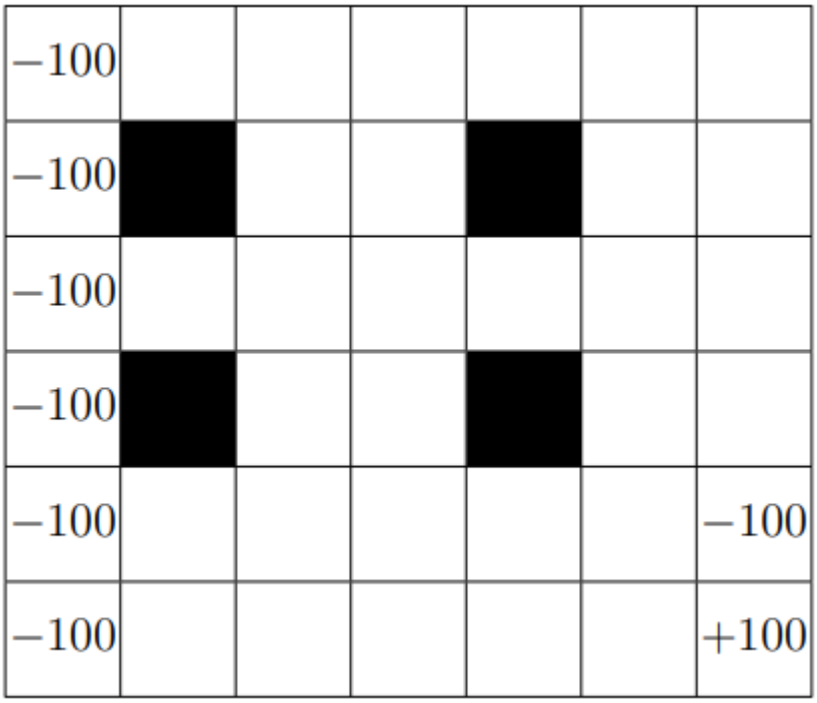
Nate Pierce
University of Michigan - Dearborn
Dr. Shenquan Wang
CIS479: AI
P3 - Reinforcement Learning

# Introduction

Goal & Environment: The primary goal of this project is to implement the Q-learning reinforcement learning algorithm.

The agent starts in some non-terminal, non-obstacle state within a 6 x 7 grid. Terminal states are defined by the -100, +100 blocks on the grid, and the black-filled blocks are obstacles, which the agent may not pass through. The agent may move west, north, east or south at any state *s*. These movements constitute the available agent actions *a*. Once reached, the agent cannot escape a terminal state.




Additionally, we consider that the maze is windy - that is, each action has an associated cost determined by the wind direction. The wind comes from the east. If the agent moves against the wind, it has an associated cost of 3. If the agent moves with the wind, the cost is 1, and if the agent moves perpendicular to the wind direction, the cost is 2. Once the agent has selected an action *a*, it may drift to the left or right each with probability 0.1. This simulates actuator error for a real-life agent. Note that if the agent moves into an obstacle or the outer wall of the maze, it simply gets bumped back and returned to its prior state.

Theory: In this scenario, the agent does not know the reward function nor the transition model. It must perform many trials to learn the optimal policy for each state. In such model-free cases, two methods are used primarily: Monte Carlo and Temporal Difference learning. Q-learning is one type of Temporal Difference learning (the other being SARSA).

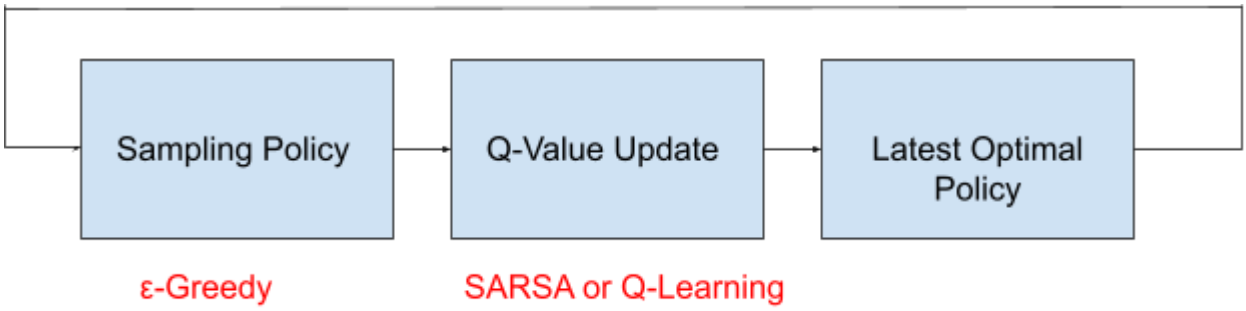
In the case where the reward function is not explicitly given for each state in the environment, two matrices must be initialized: the learning frequency matrix $N(s,a)$ and the Q-value matrix $Q(s,a)$. Typically, the elements of $N(s,a)$ are initialized to zero. Note that the elements of each matrix depend on both each possible state in the environment *s* and the associated agent action *a* for a given state. The formulae for both of these functions is given:

$$N(s,a) := N(s,a) + 1$$

$$Q(s,a) := Q(s,a) + \frac{1}{N(s,a)}(R(s,a) + \gamma max_{a'}Q(s',a') - Q(s,a))$$

Where γis some constant factor between 0 and 1, $R(s)$ is the reward function, and $max_{a'}Q(s',a')$ is the Q-Value associated with the next action in the trajectory with state $s'$ which returns the largest Q-Value. The trajectory is the agent's state/action sequence, and is represented by a list with tuple elements containing state, action, and reward information.

The workflow for learning is as follows: the trajectory must be initialized, along with the learning frequency matrix and Q-Value matrix. Then, the agent chooses a random direction based on the ε-greedy approach. Then, the Q-Value of the reached state must be updated. Finally, the optimal policy must be updated.




Finding the optimal policy is the ultimate goal of this approach to reinforcement learning. The optimal policy is defined by the function:

$$\pi^*(s) = argmax_a Q(s,a)$$

As can be seen, if$Q(s,a)$is known, the optimal policy is known. The task is then to find $Q(s,a)$ by

running many trials.

## Implementation

This implementation of Q-learning involves generating 50,000 trials starting from a random open square. We initialize the Q-values at any state-action as 0 except for one terminal state with +100 respectively. If the number of steps of a trial is more than 100, the trial is aborted and a new trial (trajectory) is started. This is to avoid getting stuck in loops. After 50,000 trials (including the aborted trials), the learning frequency, Q-Value, and optimal policy tables are reported.

Note that ε-greedy is chosen where $\varepsilon = 0.1$: the agent chooses the latest optimal action at each state with 90% probability and a random action with 10%. Also, the reward function is defined as the negative of the windy cost - so in this case, the reward function is a function of both state and action.

Environment Simulation: Drift probability (0.1 left/right, 0.8 forward) is accounted for in the move() functions using the random.choices() line. An action is passed to the function in the form of a list [0,0,0,0] which corresponds to [West, North, East, South] - except the zeroes list will have a 1 for whatever action is performed for that specific state in the trajectory. The location of the 1 is extracted using index(). The next block of code in move() checks the validity of the move. If the spot is open and within the bounds of the maze, then the move completes. Note this function ALLOWS movement into the terminal state. That condition is not checked here.

The windy condition simulation is built right into the lines of code that generate the rewards. This is done in the updateTrajectory() function. Using a similar method to the above, the action is inspected and assigned the corresponding reward.

Q-Learning and Epsilon-Greedy: Each item in the trajectory contains state, action, and reward information. The first element in every trajectory is randomized - the location in the matrix is random - the only requirement is that it is not a terminal state or a barrier. This is done in the initializeTrajectory() function. The action is also randomly generated.

Then, the trajectory is built using the updateTrajectory() function. This function also contains the Epsilon-greedy component - the random library is used to select the exploit (optimal) option with 90% weight or the explore (random) option with 10% weight.

Updating the Q-Matrix and N-Matrix are done updateQ() and updateN() functions, respectively. There's nothing special going on in these, just using the formula given in the handout. updateQ() does depend on the QPrime() function, which finds the Q(s',a') value needed.

Gamma is set as a global constant.

Output: All three outputs (Learning Frequency, Q-Value, and optimal policy) are in the form of

ROW x COLUMN matrices. The dashed line separates rows. Non-terminal, non-obstacle elements are in the form of a four value list, where each value represents a specific direction: [WEST, NORTH, EAST, SOUTH]. The barriers appear as #####.

Output:

```
In [14]: runfile('H:/Google Drive/Education/_UM Dearborn/Winter 2021/CIS479/Programming/P3/P3.py',
wdir='H:/Google Drive/Education/_UM Dearborn/Winter 2021/CIS479/Programming/P3')
Performing...
Learning Complete.
Learning Frequency Matrix

-------------------------------------------------------------------------------------------------------
-100    [485, 526, 3533, 539]    [615, 602, 5595, 602]    [612, 576, 2578, 5133]    [717, 692, 10417,
703]    [652, 685, 687, 9571]    [675, 660, 635, 8667]
-------------------------------------------------------------------------------------------------------
-100    #####    [755, 579, 4852, 1497]    [646, 648, 672, 8837]    #####    [696, 712, 669, 10675]
[750, 709, 639, 11161]
-------------------------------------------------------------------------------------------------------
-100    [505, 565, 4326, 549]    [742, 612, 606, 7941]    [758, 725, 738, 12568]    [918, 909, 18756,
879]    [1122, 1000, 1076, 23855]    [20021, 905, 989, 945]
-------------------------------------------------------------------------------------------------------
-100    #####    [697, 619, 8452, 718]    [917, 936, 938, 18838]    #####    [1346, 1311, 1368, 33277]
[17356, 869, 840, 860]
-------------------------------------------------------------------------------------------------------
-100    [490, 500, 3656, 481]    [765, 650, 10143, 662]    [1018, 993, 21966, 1038]    [934, 874, 18472,
956]    [1366, 1341, 1380, 34934]    -100
-------------------------------------------------------------------------------------------------------
-100    [543, 544, 3740, 494]    [630, 607, 6675, 588]    [878, 1172, 17811, 894]    [1051, 1110, 24244,
1121]    [1712, 1784, 49621, 1768]    100
-------------------------------------------------------------------------------------------------------
```

Q-Value Matrix

---

-100 [-91.0, 2.5177410171320616, 9.687657273684671, 2.118418080794595] [3.7666384121658876, 10.316138928686811, 18.60397091556093, 16.281024922785303] [10.606507923866445, 18.856948447550682, 22.31187354423089, 27.51207251572623] [19.40063345320927, 25.565133466356027, 33.212282778642695, 25.615663945650766] [25.787792796797728, 33.77103934942473, 26.657530178711948, 41.778007074810006] [34.55218779498821, 27.656405800251786, 26.79126286963938, 35.2897026556568]

---

-100 ##### [14.292596864164883, 10.610509893457039, 26.362803433436305, 21.92749190671749] [19.4847414647363, 19.404919695779594, 25.25080861469822, 33.61904859000514] ##### [41.6055092071302, 33.85959384940166, 34.21238929344124, 49.64947345452998] [41.470201004835864, 27.33847646387354, 34.261473728825294, 42.830471270048605]

---

-100 [-91.0, 15.073651200276803, 23.99120695737427, 15.427954054303624] [15.400430077836154, 18.538869978341136, 32.31136003322648, 32.43720950374757] [25.663244880324033, 26.148480999896837, 39.90787853138196, 40.683797286104195] [33.575970355615695, 40.892427031742436, 48.66904049229107, 40.88149583157864] [41.62149131085263, 41.75999611646558, 41.55004379425932, 57.83053707582118] [50.65739849707551, 35.199183829888725, 41.69327670206007, 50.30276735564789]

---

-100 ##### [28.562168622210958, 24.21635168319573, 39.57439024668962, 37.98433720489416] [33.117192982443655, 33.337031146041234, 39.50935338089572, 48.05180521288905] ##### [58.63843426708073, 49.59905159618075, 49.36972657680221, 66.63346339971869] [58.82919685283173, 42.644025353353044, 49.4133803780059, -92.0]

---

-100 [-91.0, 29.360164523106555, 38.42167064388471, 33.22773818620907] [29.71562189375273, 32.33763365326718, 46.99680244301908, 44.9650818800921] [39.94328184540366, 40.54390725576927, 55.94215995970594, 55.45411409981894] [48.63009622225668, 56.71800288320627, 65.62822252344901, 65.30169783313194] [57.789156264829316, 57.812657443785106, -93.0, 76.29551726112027] -100

---

-100 [-91.0, 29.891978816547297, 44.429912974527134, 32.49267615388429] [33.50691686195662, 39.16989794631146, 54.72566121889961, 44.3081249406808] [45.595599129141235, 46.81790147142597, 64.74033132596023, 55.77713061103053] [55.61316911315329, 56.54965042556415, 75.29677033492811, 65.23375020440719] [66.58043736293503, 66.58200465788312, 87.0, 76.3] 100

---

Optimal Action Matrix

---

-100 >>>>> >>>>> vvvvv >>>>> vvvvv vvvvv

---

-100 ##### >>>>> vvvvv ##### vvvvv vvvvv

---

-100 >>>>> vvvvv vvvvv >>>>> vvvvv <<<<<

---

-100 ##### >>>>> vvvvv ##### vvvvv <<<<<

---

-100 >>>>> >>>>> >>>>> >>>>> vvvvv -100

---

-100 >>>>> >>>>> >>>>> >>>>> >>>>> 100

---

Source Code (latest code in the submitted .py file):

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 26 12:37:01 2021

CIS479 - Program 3 - Reinforcement Learning
@author: Nate Pierce UMID 94712233
"""

import copy
import random


matrix = [[-100,0,0,0,0,0,0],
             [-100,'#',0,0,'#',0,0],
             [-100,0,0,0,0,0,0],
             [-100,'#',0,0,'#',0,0],
             [-100,0,0,0,0,0,-100],
             [-100,0,0,0,0,0,100]]

"""
0 is open space
# is an obstacle. Consider the walls of the maze to be an obstacle
-100, 100 are terminal states
"""

# Constants
GAMMA = 0.9
EPSILON = 0.1
ROWS = 6
COLS = 7
OBSTACLES = 4
GOAL_TERMINAL = 1
TRAP_TERMINAL = 7
TERMINAL = GOAL_TERMINAL + TRAP_TERMINAL
OPEN_SPOTS = (ROWS*COLS)-OBSTACLES - TERMINAL
SAMPLES = 50000 # number of trials, including aborted trials
```

```
# Initialize the learning frequency array as a 2D list where the elements
are dictionaries with the element ij as the key
# and the the list [west, north, east, south] as the value. The matrix that
is passed in to this function is directly modified.
def initializeN(matrix):

    arr = [0,0,0,0]
    for i in range(0, ROWS):
        for j in range(0, COLS):
            if matrix[i][j] != '#' and matrix[i][j] != -100 and
matrix[i][j] != 100: # check for valid state
                matrix[i][j] = arr


def updateN(traj, N_Matrix):
    s = traj[0] # current state in trajectory
    a = traj[1] # current action in trajectory

    i = s[0] # extract ith row from state
    j = s[1] # extract jth col from state

    sum_list = []
    for k in range(0, len(a)):
        t = N_Matrix[i][j][k] + a[k]
        sum_list.append(t)

    N_Matrix[i][j] = sum_list
    return sum(sum_list)

# The n passed in is already calculated for the current trajectory state
and action. Updates the Q_Matrix stored in memory
# (i.e. no need to return anything). The action associated with the current
item in the trajectory is the based on the
# epsilon greedy (90% it's optimal, 10% it's random) for the all trajectory
indices n > 2 (the first element in the
# trajectory is totally random - the location AND the action).
def updateQ(traj, Q_Matrix):
    s = traj[0] # extract state from trajectory
    a = traj[1] # extract action from trajectory
    r = traj[2] # extract reward function term from trajectory
```

```
    i = s[0]
    j = s[1]

    # Determine the Q-Value for the given state and the associated action
    q = copy.deepcopy(Q_Matrix[i][j])
    ind = a.index(1)

    n = N_Matrix[i][j][ind]


    # Apply the formula
    q_prime = QPrime(s, a, Q_Matrix)
    q_tot = q[ind] + (1/n)*(r + GAMMA*q_prime - q[ind])
    q[ind] = q_tot
    Q_Matrix[i][j] = q


# Given a Q_Matrix and some state and action, find the maximum Q-Value of
the adjacent state (i.e., the state we are moving
# in to). Returns the max value of that adjacent state.
def QPrime(s, a, Q_Matrix):
    i = s[0]
    j = s[1]

    if j - 1 > 0:
        if a == [1,0,0,0] and Q_Matrix[i][j-1] != '#':
            Q_Matrix[i][j] = Q_Matrix[i][j-1]
            j = j - 1
    if i - 1 > 0:
        if a == [0,1,0,0] and Q_Matrix[i-1][j] != '#':
            Q_Matrix[i][j] = Q_Matrix[i-1][j]
            i = i - 1
    if j + 1 < COLS :
        if a == [0,0,1,0] and Q_Matrix[i][j+1] != '#':
            Q_Matrix[i][j] = Q_Matrix[i][j+1]
            j = j + 1
    if i + 1 < ROWS :
        if a == [0,0,0,1] and Q_Matrix[i+1][j] != '#':
            Q_Matrix[i][j] = Q_Matrix[i+1][j]
            i = i + 1
```

```
    q = Q_Matrix[i][j] # q is a state, [i,j]
    loc = [i,j]

    if stateIsTerminal(loc) == True:
        return q
    else:
        Q_Prime  = max(q)
        return Q_Prime

def updatePolicy(Pi_Matrix, Q_Matrix, traj):
    s = traj[0]
    i = s[0]
    j = s[1]

    q = Q_Matrix[i][j]
    q_max = max(q)
    ind = q.index(q_max)

    if ind == 0:
        Pi_Matrix[i][j] = '<<<<<'
    if ind == 1:
        Pi_Matrix[i][j] = '^^^^^'
    if ind == 2:
        Pi_Matrix[i][j] = '>>>>>'
    if ind == 3:
        Pi_Matrix[i][j] = 'vvvvv'

# Location is guaranteed to not be an obstacle or a terminal - because the
location that is passed in here is generated
# in the rndLocation function which checks to see that a random position in
the matrix is NOT an obstacle or a
# terminal.
def initializeTrajectory(loc, traj):
    a = selectRndDir(matrix, loc)

    # Generate the reward based on the action. Reward is the negative of
the cost determined by windy condition
    x = a.index(1)

    if x == 0: r = -1 # action is move west
    if x == 1: r = -2 # action is move north
```

```
    if x == 2: r = -3 # action is move east
    if x == 3: r = -2 # action is move south
    update = [loc, a, r]
    traj.append(update)
    return traj

def selectRndDir(matrix, loc):
    a = random.randrange(0, 4)

    if a == 0: action = [1,0,0,0] # west
    if a == 1: action  = [0,1,0,0] # north
    if a == 2: action = [0,0,1,0] # east
    if a == 3: action = [0,0,0,1] # south

    return action


# State is the current element in the matrix.
def updateTrajectory(loc, traj, Q_Matrix):
    t = random.choices(['Optimal', 'Random'], weights=[0.9, 0.1])
    i = loc[0]
    j = loc[1]

    # generate the next action using epsilon greedy
    if t[0] == 'Random':
        a = selectRndDir(matrix, loc) # action is list in form [W, N, E,
S]. Only nonzero element is the move element.
    if t[0] == 'Optimal': # choose the optimal action for a given state
        q = Q_Matrix[i][j]
        q_max = max(q)
        ind = q.index(q_max)
        a = [0,0,0,0]
        a[ind] = 1

    x = a.index(1)

    # Generate the reward based on the action. Reward is the negative of
the cost determined by windy condition

    if x == 0: r = -1 # action is move west
    if x == 1: r = -2 # action is move north
```

```
    if x == 2: r = -3 # action is move east
    if x == 3: r = -2 # action is move south

    update = [loc, a, r]
    traj.append(update)

    return traj

def actionIsTerminal(a, loc):
    i = loc[0]
    j = loc[1]
    if a == [1,0,0,0] and j > 0: j = j-1
    if a == [0,1,0,0] and i > 0: i = i-1
    if a == [0,0,1,0] and j < COLS: j = j+1
    if a == [0,0,0,1] and i < ROWS: i = i+1

    if matrix[i][j] == -100 or matrix[i][j] == 100:
        return True # action is terminal

    else: return False # action does not result in terminal state

def stateIsTerminal(loc):
    i = loc[0]
    j = loc[1]

    if matrix[i][j] == -100 or matrix[i][j] == 100:
        return True # action is terminal

    else: return False # action does not result in terminal state


def rndLocation(matrix):
    # Generate random starting element in matrix
    g = False
    while g == False:
        i = random.randrange(0,ROWS)
        j = random.randrange(0,COLS)
        loc = [i,j] # location

        if matrix[i][j] != '#' and matrix[i][j] != -100 and matrix[i][j] !=
100:
```

```
            g = True # a valid location was randomly selected, so break the
loop

    return loc

def printM(matrix):
    for i in range(0, ROWS):
        print('\n','-'*100)
        for j in range(0, COLS):
            k = matrix[i][j]
            if k == '#': k = '#####'
            print(k, end=" "*4)
    print('\n','-'*100)

# Given a movement command [west, north, east, south] and current location,
move to that location and return the updated
# location.
def move(a, loc):
    x = a.index(1) # extract the move command
    i = loc[0]
    j = loc[1]
    t = []


    # movement command is west
    if x == 0:
        t = random.choices(['West', 'North', 'East', 'South'],
weights=(0.8, 0.1, 0.0, 0.1))

    # movement command is north
    if x == 1:
        t = random.choices(['West', 'North', 'East', 'South'],
weights=(0.1, 0.8, 0.1, 0.0))

    # movement command is east
    if x == 2:
        t = random.choices(['West', 'North', 'East', 'South'],
weights=(0.0, 0.1, 0.8, 0.1))

    # movement command is south
    if x == 3:
```

```
        t = random.choices(['West', 'North', 'East', 'South'],
weights=(0.1, 0.0, 0.1, 0.8))

    # evaluate the selected drift probability
    if j > 0:
        if t[0] == 'West' and matrix[i][j-1] != '#':
            j = j - 1
    if i > 0:
        if t[0] == 'North' and matrix[i-1][j] != '#':
            i = i - 1
    if j < COLS - 1:
        if t[0] == 'East' and matrix[i][j+1] != '#':
            j = j + 1
    if i < ROWS - 1:
        if t[0] == 'South' and matrix[i+1][j] != '#':
            i = i + 1

    # Assign the new location based on drift probability
    loc = [i, j]
    return loc

if __name__ == '__main__':

    initializeN(matrix)
    N_Matrix = copy.deepcopy(matrix) # initialize learning freq matrix all
elements are [0,0,0,0]
    Q_Matrix = copy.deepcopy(matrix)  # initialize Q-value matrix all
elements are [0,0,0,0]
    Pi_Matrix = copy.deepcopy(matrix) # initialize optimal action matrix
all elements are [0,0,0,0]
    loc = rndLocation(matrix)

    traj = []
    traj = initializeTrajectory(loc, traj) # generate first element in
trajectory (consists of a state, action, and reward)

    sample_count = 0
    q = 0 # counter used for trajectory indexing

    print('Performing...')
```

```
    while sample_count < SAMPLES:
        a = traj[q][1] # note that traj[q] is the current (state, action,
reward), and traj[q][1] gives the action
        n = updateN(traj[q],  N_Matrix)
        updateQ(traj[q], Q_Matrix)
        updatePolicy(Pi_Matrix, Q_Matrix, traj[q])


        TERMINAL = stateIsTerminal(loc) # boolean value - if True, the move
function has placed us in a terminal state

        # terminal state reached - end the current trajectory and start a
new one
        if TERMINAL == True:
            traj = []
            loc = rndLocation(matrix)
            initializeTrajectory(loc, traj)
            q = 0
            sample_count += 1
            continue

        traj = updateTrajectory(loc, traj, Q_Matrix) # update the
trajectory with (state, action, reward)
        loc = move(a, loc) # move to the element in the matrix determined
by action a

        q += 1


        # avoid getting bogged down in a loop
        if q > 100:
            q = 0
            sample_count += 1
            traj = []
            loc = rndLocation(matrix)
            initializeTrajectory(loc, traj) # end the current trajectory
and start a new one


    print("Learning Complete.")
```

```
    print('Learning Frequency Matrix')
    printM(N_Matrix)
    print('Q-Value Matrix')
    printM(Q_Matrix)
    print('Optimal Action Matrix')
    printM(Pi_Matrix)
```