INDIVIDUAL ASSIGNMENT MAS 6

Matteo De Rizzo s2749303

0_ Neccesary libraries

```
import numpy as np
import math
import ast
import itertools
import Levenshtein as lev
import random
import matplotlib.pyplot as plt
import seaborn
```

1 _ Monte Carlo estimation of Shapley value

Consider n agents (A,B,C ...N) boarding a (sufficiently large) taxi at location 0 on the real line. They share the taxi to return home. Agent A lives at distance 1, B at distance 2, etc. In general, for agent ai the distance to home equals i. The taxi driver agrees that they only need to pay the fare to the most remote destination (which is at location n).

Question 1

Compute (using the theory) the Shapley values for this problem when n is small, e.g. n = 4 or 5. This will allow you to generalise to arbitrary values of n.

```
def setting_number(num):

    ch = {}
    ch["[]"] = 0
    aid = [[]]
    set_agents = list(range(1, num+1))

    for i in range(0,num):
        x = []
        comb = itertools.combinations(set_agents,i+1)
        c = list(comb)

        for j in range(len(c)):
            fill = list(c[j])
            x.append(fill)

        for y in x:
            ch[str(y)] = y[-1]
            aid.append(y)

    return set_agents, aid, ch
```

For a small n i decided to use a value of 4

```
set_agents, aid, ch = setting_number(4)
```

```
def compute_shapley(num, set_agents, ch, aid):

    dict_shapley_vals = {}

    for agent in set_agents:
        answers = []

        for s in range(0,num):
            value_piece = []
            fact = ((math.factorial(s)*math.factorial(num-1-
s))/math.factorial(num-1))

            for i in aid:

                if len(i) == s and agent not in i:
                    new_list = i+[agent]
                    new_list.sort()
                    value = ch.get(str(new_list)) - ch.get(str(i))
                    value_piece.append(value*fact)

            answers.append(sum(value_piece))

        shapley_val = (1/num) * sum(answers)
        dict_shapley_vals[str(agent)] = shapley_val

    return dict_shapley_vals
```

Now compute the Shapley value for $n = 4$

```
compute_shapley(4, set_agents, ch, aid)
```

```
{'1': 0.25,
 '2': 0.5833333333333334,
 '3': 1.0833333333333333,
 '4': 2.083333333333333}
```

Question 2

Now set n to a large value, e.g n = 50 or n = 100. From the above you are able to guess what the Shapley values will be. However, use Monte Carlo sampling to find an approximate value for the Shapley values in this case. Discuss how effective Monte Carlo sampling is for this problem.

```
case = np.random.exponential(1, 100)
case_sorted = sorted(case)

print("Sorted Shapley values for n=100:")
```

```
print(case_sorted)

print("\nSummed values:")
print(sum(case))
```

```
Sorted Shapley values for n=100:
[0.004147991215923633, 0.008802842720003937, 0.01006493715480812,
0.012875982257934801, 0.020573144223864068, 0.03366965403490127,
0.04509122666713881, 0.04794922148530166, 0.05322342702995231,
0.059315953098062396, 0.06223259756132771, 0.07060824126362904,
0.0853674960547845, 0.09201439838275413, 0.09491770940383346,
0.09587013405345761, 0.09868685270407944, 0.10806686931406985,
0.10885497399374104, 0.12301768206575574, 0.1620947685703198,
0.16422981354450455, 0.17228707316233155, 0.17751637932870665,
0.1856744567013671, 0.18831780793914063, 0.1932037749092824,
0.2216228554608436, 0.2356523999822336, 0.23941641367523533,
0.25231928564421663, 0.27508699099636164, 0.27567655103701705,
0.28311025703288434, 0.301203752740155, 0.30914243039418277,
0.3541270386290673, 0.3628835153938474, 0.3732677056847099,
0.39165055206365007, 0.40785642582318654, 0.41182662121186436,
0.4183172687096122, 0.42038979999540055, 0.43170397239502584,
0.45939243587170214, 0.4762463482731426, 0.5024161787048388,
0.5174932984990562, 0.5355556696781538, 0.5398091419584632,
0.5583929245510625, 0.5847369431660742, 0.7164946747682923,
0.731007346644905, 0.733026517507286, 0.7334260251703422,
0.7338666673941009, 0.767137377001967, 0.7943149856345508,
0.8204302407147219, 0.83763455354971, 0.8471807668370309,
0.8854559414855968, 0.9008549554652935, 0.9122830323263293,
0.9295380178414855, 0.9343626630746271, 1.0559135893888971,
1.0803588828937, 1.0942581620950504, 1.1000681516316528,
1.110712102185659, 1.1339808595325156, 1.1491894431654401,
1.2392891548895457, 1.3640148159703949, 1.4216190749474875,
1.4810326966353216, 1.5591619669270234, 1.5798203726640092,
1.6491830365475655, 1.7758927727233567, 1.7760420840933069,
1.8398076220178483, 1.969905589754994, 1.9754155279367576,
2.063578582881693, 2.139809937974386, 2.1821221443953855,
2.210669473329026, 2.397631642487265, 2.4102398690740854,
2.5192485865898266, 2.821166821615393, 2.8644980489167122,
3.710042161952239, 3.823219244641706, 4.451945874441391,
4.808812731671765]

Summed values:
90.65363697780063
```

The application of the Monte Carlo Sampling approach can be used to calculate an approximation of any probability distribution, and it is much more efficient than the computation of Shapley values (SV). During some tests, I tried to compute SV for a n=20, and the necessary time was vastly superior to the one using the MC approach. Therefore, I believe that the Monte Carlo Technique is very effective for this problem, especially on larger n values

2 _ Monte Carlo Tree Search (MCTS)

Construct binary tree

Construct a binary tree of depth d = 20 (or more – if you're feeling lucky). Since the tree is binary, there are two branches (aka. edges, directions, decisions, etc) emanating from each node, each branch (call them L(eft) and R(ight)) pointing to a unique child node (except, of course, for the leaf nodes – see Fig 1). We can therefore assign to each node a unique "address" (A) that captures the route down the tree to reach that node (e.g. A = LLRL – for an example, again see Fig 1). Finally, pick a random leaf-node as your target node and let's denote its address as At.

Creating the Node class:

```
class Node():
    def __init__(self, parent, children, key, value, is_leaf):
        self.parent = parent
        self.children = children
        self.is_leaf = is_leaf
        self.value = value
        self.key = key
        self.num_visits = 0
```

In the following block I implemented a Binary Tree constructor, with a default depth of 20, as requested by the assignment. Furthermore, to each leaf node a value was assigned, based on edit distance with the following formula:

$$x_i = Be^{-d_i/\tau} + \varepsilon_i$$

```
class Tree():

    def __init__(self, depth=20):
        self.depth = depth
        self.root = Node(None, [], '', 0, False)
        self.leaves = []
        self.goal = None
        self.set_up()
        self.assign_val_to_leafs()

    def set_up(self):
        parents = [self.root]
        counter = 0
        num_nodes = 0
        while True:
            new_generation = []
            for parent in parents:
                num_nodes += 1
                if counter == self.depth - 1:
```

```
                L_child = parent.key + 'L'
                Lc = Node(parent, None, L_child, 0, True)

                R_child = parent.key + 'R'
                Rc = Node(parent, None, R_child, 0, True)

                self.leaves.append(Lc)
                self.leaves.append(Rc)
                num_nodes += 2
            else:
                L_child = parent.key + 'L'
                Lc = Node(parent, [], L_child, 0, False)

                R_child = parent.key + 'R'
                Rc = Node(parent, [], R_child, 0, False)

            parent.children.append(Lc)
            parent.children.append(Rc)

            new_generation += parent.children

        counter += 1

        if counter == self.depth:
            break

        parents = new_generation

    return num_nodes

def assign_val_to_leafs(self, B=10, tau=2):
    self.goal = np.random.randint(0, len(self.leaves))
    goal = self.leaves[self.goal]


    for leaf in self.leaves:
        distance = lev.distance(leaf.key, goal.key)
        eps = random.gauss(0,1)
        leaf.value = (B * np.exp(-distance / tau)) + eps

def nodes(self):
    print()

tree = Tree()
```

The tree created by the above code resulted in 2097151 nodes

QUESTION 1 and 2

-Implement the MCTS algorithm and apply it to the above tree to search for the optimal (i.e. highest) value.

-Collect statistics on the performance and discuss the role of the hyperparameter c in the UCB-score.

The Monte Carlo Tree Search algorithm was implemented.

```
class MCTS:
    def __init__(self, C):
        self.C = C


    def roll_out(self, start_node):
        while start_node.is_leaf == False:
            rand = np.random.randint(0, 2)
            start_node = start_node.children[rand]
        return start_node


    def backup(self, start_node):
        start_value = start_node.value
        start_node.num_visits += 1


        while start_node.parent != None:
            start_node = start_node.parent
            start_node.num_visits += 1
            start_node.value += start_value

        return start_node


    def ucb(self, node):
        return (node.value / node.num_visits) + self.C * 
np.sqrt(np.log(node.parent.num_visits) / node.num_visits)


    def policy(self, start_node):
        while start_node.is_leaf == False:
            ucbs = []
            not_explored = []
            for child in start_node.children:
                if child.num_visits == 0:
                    not_explored.append(child)
                else:
                    ucbs.append(self.ucb(child))
            if len(not_explored) > 0:
```

```
                return np.random.choice(not_explored)
            start_node = start_node.children[np.argmax(ucbs)]
        return start_node


    def run(self, start_node, episodes=1000):
        for episode in range(episodes):
            roll_out_node = self.policy(start_node)
            leaf_node = self.roll_out(roll_out_node)
            root_node = self.backup(leaf_node)
        print('The MCTS has finished.')
        return roll_out_node
```

When looking for the optimal value I used the Upper Confidence Bound (UCB), which has as hyperparameter C. In order to discuss the role of the hyperparameter C in the we need to look at the formula for UCB:

$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

In the experiments I have looked at different values of C, to determine its role. The value ranged from 0.5 to 5.

```
def find_optimal(C):
    print("Looking for optima with C=", str(C))
    mcts = MCTS(C)
    optimal = mcts.run(tree.root, episodes=10000)
    x = []
    for leaf in tree.leaves:
        x.append(leaf.num_visits)
    print("key: ", optimal.key)
    print("value: ",optimal.value)
    print("goal leaf: ",tree.leaves[tree.goal].key)
    print("argmax: ",tree.leaves[np.argmax(x)].key)

find_optimal(0.5)
print("")
find_optimal(1)
print("")
find_optimal(1.5)
print("")
find_optimal(2.5)
```

```
print("")
find_optimal(5)
```

```
Looking for optima with C= 0.5
The MCTS has finished.
key:  RRRLRLLLLRRRRRLLLRRR
value:  10.528089022665002
goal leaf:  RRRLRLLLLRRRRRLLLRRR
argmax:  RRRLRLLLLRRRRRLLLRRR

Looking for optima with C= 1
The MCTS has finished.
key:  RRRLRLLLLRRRRRLLLRRL
value:  5.101778143932502
goal leaf:  RRRLRLLLLRRRRRLLLRRR
argmax:  RRRLRLLLLRRRRRLLLRRR

Looking for optima with C= 1.5
The MCTS has finished.
key:  RRRLRLLLLRRRRRLLLRRL
value:  5.101778143932502
goal leaf:  RRRLRLLLLRRRRRLLLRRR
argmax:  RRRLRLLLLRRRRRLLLRRR

Looking for optima with C= 2.5
The MCTS has finished.
key:  RRRLRLLLLRRRRRLLLRRR
value:  10.528089022665002
goal leaf:  RRRLRLLLLRRRRRLLLRRR
argmax:  RRRLRLLLLRRRRRLLLRRR

Looking for optima with C= 5
The MCTS has finished.
key:  RRRLRLLLLRRRRRLLLRRL
value:  5.101778143932502
goal leaf:  RRRLRLLLLRRRRRLLLRRR
argmax:  RRRLRLLLLRRRRRLLLRRR
```

3 _ Reinforcement Learning: SARSA and Q-Learning for Gridworld

Consider the 9 ×9 gridworld example depicted in the figure 2 below. The blue gridcells represent walls that cannot be traversed. The green cell represent a treasure and transition to this cell yields a reward of +50 whereupon the episode is terminated (i.e. absorbing state). The red cell represents the snakepit: this state is also absorbing and entering it yields a negative reward of −50. All other cells represent regular states that are accessible to the agent. In each cell, the agent can take four actions: move north, east, south or wes (not moving is NOT a valid action). These actions result in a deterministic transition to the corresponding neighbouring cell. An action that makes the agent bump into a wall or the grid-borders, leaves its state unchanged. All non-terminal transitions (including running into walls or grid borders) incur a *negative reward ("cost") of −1. For the questions below, we assume that the agent is not aware of all the above information and needs to discover it by interacting with the environment (i.e. model-free setting).*

```
class Rules:
    def __init__(
        #Define all necessary inputs
        self,
        list_walls: np.array,
        list_pitfalls: np.array,
        dimensions: tuple = (9, 9),
        in_tile: tuple = (0, 0),
        out_tile: tuple = (8, 8),
    ):
        self.walls = list_walls
        self.pitfalls = list_pitfalls
        self.dimensions = dimensions
        self.in_tile = in_tile
        self.out_tile = out_tile
        self.to_in_tile()

    def to_in_tile(self):
        self.condition = list(self.in_tile)
        self.done = False

    def step(self, step: str):
        assert step in [ "left", "right","up", "down"], "The step is
not possible"
        goal = self.condition[:]
        if step == "left" and self.condition[1] > 0:
            goal[1] -= 1
        elif step == "right" and self.condition[1] <
self.dimensions[1] - 1:
            goal[1] += 1
        elif step == "up" and self.condition[0] > 0:
            goal[0] -= 1
        elif step == "down" and self.condition[0] < self.dimensions[0]
- 1:
            goal[0] += 1

        else:
            return tuple(self.condition), 0, False
        if tuple(goal) == self.out_tile:
            self.done = True
            return tuple(self.condition), 50, True
        elif tuple(goal) in self.pitfalls:
            self.done = True
            return tuple(self.condition), -50, True
        elif tuple(goal) in self.walls:
            return tuple(self.condition), 0, False
```

```
        else:
            self.condition = goal
            return tuple(self.condition), 0, False


def simulation(policy, environment, walls, pitfalls):
    possible_steps = {0: "left", 1: "right", 2: "up", 3: "down"}
    environment.to_in_tile()
    num_trials = 500
    vals = np.zeros((9, 9))
    for x in range(9):
        for y in range(9):
            state_val = 0
            print(f"State being simulated: {x},{y}")
            if (x, y) not in walls:
                for i in range(num_trials):
                    val = 0
                    end = False
                    environment.to_in_tile()
                    environment.condition = [x, y]
                    pos = [x, y]
                    steps = 0
                    while not end and steps < 50:
                        steps += 1
                        if not ((x, y) == (8, 8) or (x, y) in
pitfalls):
                            prev_pos = pos
                            pos, reward, end = environment.step(
                                possible_steps[policy[tuple(pos)]]
                            )
                            if prev_pos == pos:
                                break
                            val += reward
                        elif (x, y) == (8, 8):
                            val += 50
                            end = True
                        else:
                            val -= 50
                            end = True
                    state_val += val
                vals[x, y] = state_val / num_trials
```

Question 1

Use SARSA in combination with greedification to search for an optimal policy.

```
list_walls = [(1, 2),(1, 3),(1, 4),(1, 5),(1, 6),(2, 6),(3, 6),(4, 6),
(5, 6),(7, 1),(7, 2),(7, 3),(7, 4)]
list_pitfalls = [(6, 5)]
num_steps = 600000
environment = Rules(
```

```python
    list_walls=list_walls,
    list_pitfalls=list_pitfalls,
)

#Define parameters
alpha = 0.2
gamma = 0.8
eps = 0.5

found_policy = np.full((*environment.dimensions,), "down")
possible_steps = ["left", "right","up", "down"]
Qvs = np.zeros((*environment.dimensions, 4))
Log = np.zeros((num_steps, *Qvs.shape))
for x in range(num_steps):
    if random.random() < eps:
        direction = np.random.randint(0, 4)
    else:
        direction = np.argmax(Qvs[tuple(environment.condition)])
    old_pos = tuple(environment.condition[:])

    next_state, reward, end =
environment.step(possible_steps[direction])
    if random.random() < eps:
        next_direction = np.random.randint(0, 4)
    else:
        next_direction = np.argmax(Qvs[tuple(next_state)])

    refresh = alpha * (
        reward + gamma * Qvs[next_state][next_direction] -
Qvs[old_pos][direction]
    )
    Qvs[old_pos][direction] += refresh
    Log[x] = Qvs
diff = np.zeros(num_steps)
for x in range(num_steps):
    diff[x] = np.sum(np.abs(Log[x] - Qvs))
found_policy = np.argmax(Qvs, axis=2)

print("Start of simulation")
simulation(found_policy, environment, list_walls, list_pitfalls)
print("End of simulation")
```

```
Start of simulation
State being simulated: 0,0
State being simulated: 0,1
State being simulated: 0,2
State being simulated: 0,3
State being simulated: 0,4
State being simulated: 0,5
```

```
State being simulated: 0,6
State being simulated: 0,7
State being simulated: 0,8
State being simulated: 1,0
State being simulated: 1,1
State being simulated: 1,2
State being simulated: 1,3
State being simulated: 1,4
State being simulated: 1,5
State being simulated: 1,6
State being simulated: 1,7
State being simulated: 1,8
State being simulated: 2,0
State being simulated: 2,1
State being simulated: 2,2
State being simulated: 2,3
State being simulated: 2,4
State being simulated: 2,5
State being simulated: 2,6
State being simulated: 2,7
State being simulated: 2,8
State being simulated: 3,0
State being simulated: 3,1
State being simulated: 3,2
State being simulated: 3,3
State being simulated: 3,4
State being simulated: 3,5
State being simulated: 3,6
State being simulated: 3,7
State being simulated: 3,8
State being simulated: 4,0
State being simulated: 4,1
State being simulated: 4,2
State being simulated: 4,3
State being simulated: 4,4
State being simulated: 4,5
State being simulated: 4,6
State being simulated: 4,7
State being simulated: 4,8
State being simulated: 5,0
State being simulated: 5,1
State being simulated: 5,2
State being simulated: 5,3
State being simulated: 5,4
State being simulated: 5,5
State being simulated: 5,6
State being simulated: 5,7
State being simulated: 5,8
State being simulated: 6,0
State being simulated: 6,1
```

```
State being simulated: 6,2
State being simulated: 6,3
State being simulated: 6,4
State being simulated: 6,5
State being simulated: 6,6
State being simulated: 6,7
State being simulated: 6,8
State being simulated: 7,0
State being simulated: 7,1
State being simulated: 7,2
State being simulated: 7,3
State being simulated: 7,4
State being simulated: 7,5
State being simulated: 7,6
State being simulated: 7,7
State being simulated: 7,8
State being simulated: 8,0
State being simulated: 8,1
State being simulated: 8,2
State being simulated: 8,3
State being simulated: 8,4
State being simulated: 8,5
State being simulated: 8,6
State being simulated: 8,7
State being simulated: 8,8
End of simulation
```

Question 2

Use Q-learning to search for an optimal policy. Implement two different update strategies:

A _ Direct updates: Update the Q-table while rolling out each sample path;

```
#Define parameters

alpha = 0.1
gamma = 0.8
eps = 0.8


Qvs = np.zeros((*environment.dimensions, 4))
policy = np.full((*environment.dimensions,), "down")
possible_steps = ["left", "right", "up", "down"]
Log = np.zeros((num_steps, *Qvs.shape))
for x in range(num_steps):
    policy_check = False
    if random.random() < eps:
        direction = np.random.randint(0, 4)
    else:
        direction = np.argmax(Qvs[tuple(environment.condition)])
    old_pos = tuple(environment.condition[:])
```

```
    next_state, reward, done =
environment.step(possible_steps[direction])
    next_direction = np.argmax(Qvs[tuple(next_state)])
    refresh = alpha * (
        reward + gamma * Qvs[next_state][next_direction] -
Qvs[old_pos][direction]
    )
    Qvs[old_pos][direction] += refresh
    Log[x] = Qvs
diff = np.zeros(num_steps)
for x in range(num_steps):
    diff[x] = np.sum(np.abs(Log[x] - Qvs))

plt.plot(range(num_steps), diff)
plt.xlabel("number of iterations")
plt.ylabel("difference from last Q as absovule value")
print(diff[0])
plt.show()
```

```
11379.541261422664
```