Lukas Jigberg 18h

Question 1

Since nothing has been stated do we assume that there is a 100% success chance when choosing a path/action.

a

Given that we didn't get a gamma will we assume that it is set to 1, no discount in points / points remain the same. Also, all the actions give the same/or no amounts of points, no action is better than any other.

We can theoretically create infinite point grabbing paths as F itself doesn't give us any points and the positions adjacent to F are also not lucrative. Doing loops above F at the 1 will give us a infinite string ex: "EE SWNE SWNE SW..". This would technically give us the most points, never ending as that wouldn't be lucrative.

Giving ourselves the rules;

- "We can't step on a Position we already stepped on"
- "We have to reach F sometime"

Will allow us to create a "functioning" string path.\ "SENESSS" scoring us 4 points.

b

Since there is no mention of any fail % do we assume that there is 100% probability of moving from (1,0) to (2,0) when using action E. Out of all the directions we can take at (1,0) is EAST still the most lucrative, scoring us the highest. Without a failure rate will we 100% go EAST if we were to pick.

c

Moving to the right would 'increase the score' by 1 and be lucrative, but the action itself 'EAST' would not score us any points.

Question 2

Using the starting matrixes already have from the exerice lab and

In [ ]:

```
#Creates more iterations until the difference is less then epsilon
def main(gridList,eps):
  iter=2
  while compare(gridList,eps):
    iter +=1
    gridList.append(createNewIteration(gridList))
  print("Iterations:",iter)
  print(gridList[-1])

#Compares the values of 2 matrixes
def compare(gridList,eps):
  done = False
  xl=[0,1,2]
  yl=[0,1,2]
  for x in xl:
    for y in yl:
      if ( ((gridList[-1])[x][y] - (gridList[-2])[x][y]) > eps ):
        done = True
  return done

#Creates the new iteration
#Adds the biggest value each position can achive by using a valid action
def createNewIteration(gridList):
  emptyGrid = []
  i = len(gridList)-1
  x = 0
  y = 0

  for row in gridList[1]:
    rowList = []
    for spot in row:
      direction = [(x+1,y),(x-1,y),(x,y+1),(x,y-1)]
      direction = list(filter(lambda x: x[0]>(-1), direction))
      direction = list(filter(lambda y: y[1]>(-1), direction))
      direction = list(filter(lambda x: x[0]<(3), direction))
      direction = list(filter(lambda y: y[1]<(3), direction))
      rowList.append( getMax(i,gridList,(x,y),direction) )
      x+=1
    emptyGrid.append(rowList)
    x=0
    y+=1
  return emptyGrid

#Returns the biggest achivable value of x actions
def getMax(i,gridList,spot,sList):
  valueList = []
  for s in sList:
    ans = 0.8*( reward(s)+ 0.9 *pointInGrid(gridList,i,s) ) + (1-0.8)*(reward(spot) +
0.9 *pointInGrid(gridList,i,spot))
    valueList.append(ans)
  return max(valueList)

def pointInGrid(gridList, i, spot):
  return ((gridList[i])[spot[0]][spot[1]])

#Returns the reward of position in matrix
def reward(spot):
  if (spot == (1,1)):
    return 10
```

```
  else: return 0

grid =([[0,0,0],
        [0,0,0],
        [0,0,0]])

grid1 =([[0,0,0],
        [0,10,0],
        [0,0,0]])

grid2 = ([[0,8,0],
        [8,2,8],
        [0,8,0]])


main([grid,grid1],0.0001)
```

```
Iterations: 105
[[45.61205231501365, 51.94718060136145, 45.61205231501365], [51.9471806013
6145, 48.05107670525755, 51.94718060136145], [45.61205231501365, 51.947180
60136145, 45.61205231501365]]
```

The converging optimal values are roughly:

[45.6128 51.9480 45.6128]\ [51.9480 48.0519 51.9480]\ [45.6128 51.9480 45.6128]

Question 3

In [ ]:

```
import gym
import numpy as np
import random
import math

#Set Variables

learningRate = 0.1
gamma = 0.95
num_episodes = 15000
eps = 0.5
enviroment = gym.make('NChain-v0')

Q = np.zeros([5,2])

for _ in range(num_episodes):
        state = enviroment.reset()
        done = False
        while done == False:
        # First we select an action:
                if random.uniform(0, 1) < eps: # Flip a skewed coin
                        action = enviroment.action_space.sample() # Explore action spac
e
                else:
                        action = np.argmax(Q[state,:]) # Exploit learned values
        # Then we perform the action and receive the feedback from the environment
                new_state, reward, done, info = enviroment.step(action)
        # Finally we learn from the experience by updating the Q-value of the selected
 action
                update = reward + (gamma*np.max(Q[new_state,:])) - Q[state, action]
                Q[state,action] += learningRate*update
                state = new_state

print("  Forwards   | Backwards")
print(Q)
```

```
  Forwards   | Backwards
[[74.31252641 74.00320776]
 [78.08017234 74.21034529]
 [82.53616686 74.5219749 ]
 [87.59753086 80.4080218 ]
 [95.15327428 79.49028234]]
```

Question 4

a

MDP for the 'Chain' Problem.\ Since no instructions were given how MDP where suppose to interact this time did I decide to use the same logic as before. 0.8 and 0.2 is the chance of going in the intended/opposite direction.

In [ ]:

```
gamma = 0.95
stay = 0.5
go = 0.5
misstep = 0.8
startChain = [0,0,0,0,0]
eps = 0.00001


#Creates more iterations until the difference is less then epsilon
def main_(gridList):
  iter = 0
  while compare_(gridList):
    gridList.append(newIteration(gridList))
    iter+=1
  print(gridList[-1])
  print("iterations;",iter)
  return gridList[-1]

#Compares all the values in two lists
def compare_(gridList):
  done = False
  for i in range(0,len(gridList[-1])-1):
    if len(gridList) ==1:
      return True
    if (gridList[-1])[i] - (gridList[-2])[i] > eps:
      done = True
  return done

def newIteration(gridList):
  newGrid = []
  for i in range(0,(len(gridList[-1]))):
    newGrid.append( getMax_(i,gridList[-1]) )
    i+=1
  return newGrid

#Returns the biggest achivable value of x actions
def getMax_(i,chain):
  if i==len(chain)-1:
    forward = misstep*(reward_(i)+gamma*chain[i]) + (1-misstep)*(2+gamma*chain[0])  #Ac
tion a
    toStart = misstep*(2+gamma*chain[0] ) + (1-misstep)*(reward_(i)+gamma*chain[i]) #Ac
tion b
  else:
    forward = misstep*(reward_(i)+gamma*chain[i+1]) + (1-misstep)*(2+gamma*chain[0])  #
Action a
    toStart = misstep*(2+gamma*chain[0] ) + (1-misstep)*(reward_(i)+gamma*chain[i+1]) #
Action b
  if forward > toStart:
    return forward
  else:
    return toStart

#Returns the reward of a given position
def reward_(i):
  if i == len(startChain)-1:
    return 10
  else: return 0
```

```
V = main_([startChain])
eList = []
for i in range(0,len(V)-1):
  eList.append( (V[i]-Q[i][0]) )

print()

print("Q:",Q[:,0])
print("V:",V)
print("Difference:",eList)
```

```
[61.37929954792862, 64.89110754792861, 69.51190754792862, 75.5919075479286
3, 83.59190754792863]
iterations; 251

Q: [74.31252641 78.08017234 82.53616686 87.59753086 95.15327428]
V: [61.37929954792862, 64.89110754792861, 69.51190754792862, 75.5919075479
2863, 83.59190754792863]
Difference: [-12.933226864354559, -13.189064787209674, -13.02425931019836,
-12.005623307627047]
```

It seems that Q differs when running the program with a pretty big margin. So the values that appear are different from mine. The Q that I tested for was actually quite close to V, only having an offset of 0 - 4.

b

Exploration is hugely important when optimizing a task, be it for an algorithm or 'if a bear is looking for food'. Two completely separate task but the point remains the same. If we only exploit, will we only go for the currently best option every time.\ The chain is a good example, option B will always be better than taking option A at first, but in the end will you find a even greater point gathering option rewarding you 10 points. Had we only went for the exploit would we receive more points instantly yes, but we would increase our points gathering ability 5 times if we just looked.\ Same could be said about the bear looking for food, or any animal for that matter. By exploring can the bear if lucky find an even greater source of food, optimizing its bear life. With its new food source can it exploit more food, but it will still be looking for / exploring for more food in case there is an even greater option, trying to (maybe unknowingly) optimizing its life and species in the process.

Question 5

The Multi-Armed Bandit Problem is a problem where we are looking to gather the most points / achieve the best result on a task where we don't know anything about the points achieving system when starting.

Example:\ We want our evening in town for the next 8 months to be awesome, there are 3 restaurants where we can spend the evening and we don't know anything about the enjoyment we get from them yet, they only way is to try. The restaurants also do not give exact enjoyment but deliver x enjoyment from their own enjoyment margin. The enjoyment from the same restaurant will thereby vary day to day. We now want to figure out which restaurant we should go to maximise our 8 months, what tactic do we use?

The task/problem to solve is to find a tactic/strategy that secures the most points/enjoyment.

Obviously we can choose to only exploit 1 for the 8 months, hoping that we pick the best restaurant by chance, which in this case there is 1/3 of doing, however it’s much harder if we increase the number of restaurants/K.

Exploring every day will give us the best margin of all the places and is probably the safest bet, but not the best.

They key is to find the best distribution of exploring and exploiting. Maybe we try a new restaurant at random every 5 days, seeing if there maybe is a better option and if there are we might switch to it when we exploit. This strategy is called Epsilon Greedy.

That is the basis of the Multi-Armed Bandit problem, finding the best distribution of Explore and Exploit to score the best on a problem/situation we know nothing about.