**Ministerul Educaţiei Tineretului şi Sportului al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**SARCINA INDIVIDUALĂ LA MATEMATICA DISCRETĂ**

Au efectuat Studenții grupei \_SI-212\_\_

\_\_Sandu Boboc și\_Vozian Vladimir\_

A verificat \_\_\_Lisnic Inga\_

**Chisinău 2022**

1. **Sarcina**

Într-un arbore decizional perfect (n-ary, fiecare nod parinte are acelasi numar N de noduri copii cu exceptia nodului radacina) format din actiuni, stari si ponderea pentru fiecare relatie actiune-starea.

Se cere sa se calculeze suma tuturor ponderilor pentru fiecare drum de la radacina spre vârful arborelui (noduri care nu au copii) si să se afiseze drumul care are cea mai mare suma.

**Cerinte**:

Algoritmul trebuie sa fie optimizat astfel incat sa permite parcurgerea arborilor care au mai multe niveluri, de exemplu 30 de niveluri (30 generatii parinte-copil). (Parcurgerea ar putea lucra mai rapid prin utilizarea matricilor cu index in loc de obiecte iar calcularea drumurilor ar putea fi mai rapida daca se anuleaza drumurile fara perspectiva)

*Exemplu de cod dat de profesoara ca reper*

**import numpy as np**

**import itertools**

**class Node(object):**

**def \_\_init\_\_(self, id, level, action, state, value, reward, parent, children=None):**

**self.id = id**

**self.level = level**

**self.parent = parent**

**self.action = action**

**self.state = state**

**self.value = value**

**self.children = list() if children is None else children**

**def dfs(node):**

**path = list()**

**def recurse(n):**

**path.append(n)**

**if not n.children:**

**yield path**

**for child in n.children:**

**for x in recurse(child):**

**yield x**

**path.pop()**

**for v in recurse(node):**

**yield path**

**def Add\_Nodes(level, actions):**

**global root**

**global idk**

**global childs**

**childs\_new = []**

**for child in childs:**

**for a in range(6):**

**idk += 1**

**v = np.random.randint(0,100)**

**s = np.random.randint(0,100)**

**child.children.append( Node(idk, level+1, a, s, v, child, None) )**

**childs\_new += child.children**

**childs = childs\_new**

**idk = 0**

**childs = []**

**root = None**

**actions = 6**

**levels = 6**

**root = Node(idk, 0, 0, 0, 0, None, None)**

**childs.append(root)**

**for i in range(levels):**

**Add\_Nodes(i, actions)**

**max\_path\_value = 0**

**maxpath = None**

**for path in dfs(root):**

**path\_str = ""**

**for n in path:**

**path\_str +=  "{}({}), ".format(n.id, n.value)**

**path\_value = sum(n.value for n in path)**

**print("{} : {}".format(path\_value, path\_str))**

**if max\_path\_value < path\_value:**

**max\_path\_value = path\_value**

**maxpath = "{} : {}".format(path\_value, path\_str)**

**print("Maximum path:\n", maxpath, "\n")**

1. **Cercetarea**

**Prima încercare**

Inițial am schimbat codul exemplu și anume sa scos secvența unde se țineau minte toate drumurile parcurse și se afișau la ecran Notă: initial la primirea datelor ca 6 noduri copii cu 6 nivele programul se efectua în aproximativ 10 secunde, în final sa obținut codul următor, care la rândul său deja sa optimizat destul de mult functionând cu aceleași date ca fiind date 6 noduri copii cu 6 nivele, și programul se finisa in deja mai puțin de 1 secundă :

import numpy as np

import itertools

class Node(object):

    def \_\_init\_\_(self, id, level, action, state, value, reward, parent, children=None):

        self.id = id

        self.level = level

        self.parent = parent

        self.action = action

        self.state = state

        self.value = value

        self.children = list() if children is None else children

def dfs(node):

    path = list()

    def recurse(n):

        path.append(n)

        if not n.children:

            yield path

        for child in n.children:

            for x in recurse(child):

                yield x

        path.pop()

    for v in recurse(node):

        yield path

def Add\_Nodes(level, actions):

    global root

    global idk

    global childs

    childs\_new = []

    for child in childs:

      for a in range(6):

          idk += 1

          v = np.random.randint(0,100)

          s = np.random.randint(0,100)

          child.children.append( Node(idk, level+1, a, s, v, child, None) )

      childs\_new += child.children

    childs = childs\_new

idk = 0

childs = []

root = None

actions = 6

levels = 6

root = Node(idk, 0, 0, 0, 0, None, None)

childs.append(root)

for i in range(levels):

  Add\_Nodes(i, actions)

max\_path\_value = 0

maxpath = None

for path in dfs(root):

    path\_str = ""

    for n in path:

          path\_str +=  "{}({}), ".format(n.id, n.value)

    path\_value = sum(n.value for n in path)

    print("{} : {}".format(path\_value, path\_str))

    if max\_path\_value < path\_value:

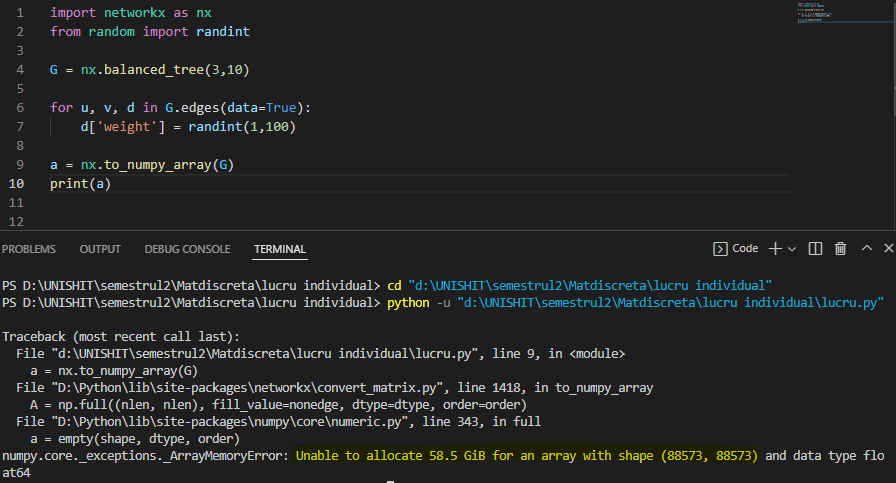
      max\_path\_value = path\_value

      maxpath = "{} : {}".format(path\_value, path\_str)

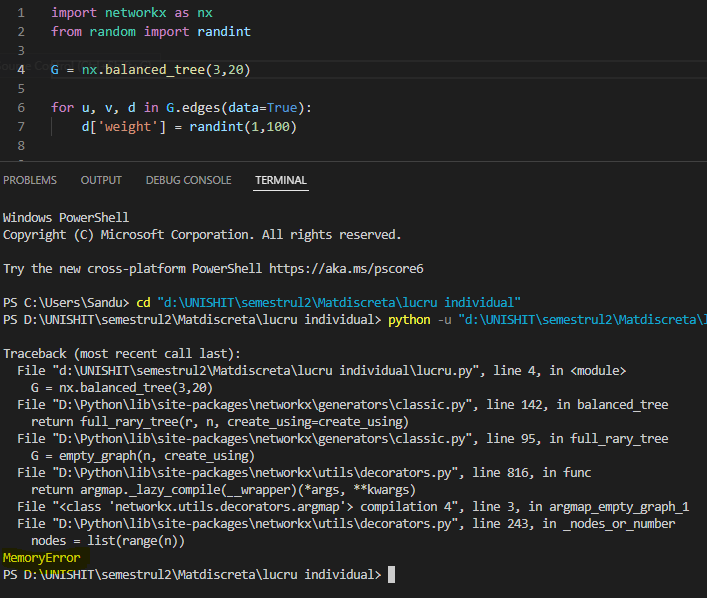
print("Maximum path:\n", maxpath, "\n")

**A doua încercare:**

Prin modulul din python networkx am încercat să generăm un arbore și sa punem informația într-o matrice prin modulul numpy însă la alocarea memoriei se producea o eroare chiar și pentru un arbore de nivel 10 cu 3 descendenți(imaginea 1). Numere sunt foarte mari și din cauza asta memoria alocata pentru această matrice este la fel de mare și operațiunea cu matrice nu putea sa se primească. Iar dacă folosim la generare doar networkx la nivelul 20 cu 3 descendenți chiar și modulul da o eroare de memorie(imaginea 2)



Imaginea 1



Imaginea 2

Modulul sau matricea nu ne era de ajutor din cauza alocarii vaste de memorie.

**Rezultatul final:**

Am implementat un algoritm recursiv cu backtracking care mergea de la început pană la sfârșit pe un drum și salva valorile inițiale precum ponderea pe acel drum și drumul propriu zis apoi prin backtracking el genera alt drum și compara valorile de drumul maxim în ambele , dacă primul drum era mai mic ca al doilea primul drum se ștergea din memorie și în locul valorilor sale erau deja valorile din al doilea drum și asta se repeata pe parcursul la tot arborele, ce ne permitea sa salvam memorie RAM pentru că algoritmul nu ținea minte fiecare drum generat și fiecare pondere pe ele ci le genera și le compara în același timp înlocuindu-le acolo unde drumul successor avea o valoare mai mare de pondere. Deși este un algoritm simplu rezultatul pentru 30 de nivele cu 3 descendenți după aproximari va avea o durata de 18 ore de lucru, pentru că algoritmul are de lucrat în totalmente cu 102945566047324 de noduri. La nivelul 20 cu 3 descendenți sunt deja 1117 secunde (18 minute) de lucru iar la nivelul 30 cu 2 descendenți rezultatul este de 716 secunde (12 minute). Acesta a fost cel mai optim algoritm care lam putut implementa pentru sarcina data:

import time

from random import randint

maxN = [[]]

sumMax = 0

currNode = 1

def calc(n, currNivel, nivel, pondereSum, tempMax):

    global sumMax

    global maxN

    global currNode

    if currNivel >= nivel:

        if pondereSum >= sumMax:

            sumMax = pondereSum

            maxN = tempMax.copy()

        return

    for i in range(1,n+1):

        pond = randint(0,100)

        currNode = currNode + 1

        tempMax.append([currNode,pond])

        calc(n, currNivel+1, nivel, pondereSum + pond, tempMax)

        tempMax.pop(len(tempMax) - 1)

start = time.time()

calc(2, 1, 20, 0, [[currNode,0]])

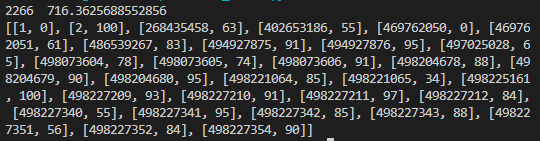
end = time.time()

print(sumMax,"",end-start)

print(maxN)

Rezultatele despre care sa menționat mai sus:

N30 cu 2 descendenti:



N20 cu 3 descendenti:

