Avioane/var Rucsac Discret

import numpy as np  
import matplotlib.pyplot as grafic  
from FunctiiCrossoverIndivizi import crossover\_unipunct  
from FunctiiMutatieIndivizi import m\_binar,m\_ra  
from FunctiiSelectii import turneu,elitism,SUS  
#functia obiectiv  
def avioane(x):  
 #calculam fitness-ul  
 autonomie=(6000\*x[0]+4200\*x[1]+2800\*x[2])/(x[0]+x[1]+x[2])  
  
 #restrictia pentru suma disponibila  
 cost=100\*x[0]+60\*x[1]+50\*x[2];  
  
 #restrictia pentru vizibilitatea meteorologica  
 vizibilitate=(1500\*x[0]+2400\*x[1]+1600\*x[2])/(x[0]+x[1]+x[2])  
  
 return cost <=5000 and vizibilitate>2000, autonomie;  
  
#generarea populatiei  
def generare(dim):  
 n=3  
 pop = []  
 pop\_fara\_fitness=[]  
 for i in range(dim):  
 gata = False # flag  
 while gata == False:  
 x=[];  
 x1=np.random.randint(0,51);  
 x2=np.random.randint(0,84);  
 x3=np.random.randint(0,101);  
 x=[x1,x2,x3];  
 gata, val = avioane(x);  
  
 x = list(x)  
  
 pop\_fara\_fitness=pop\_fara\_fitness+[x]  
  
 x = x + [val]  
  
 pop = pop + [x]  
 pop = np.asarray(pop)  
 return pop,dim,n,pop\_fara\_fitness  
  
#selectie parinti  
  
def SelectieParinti(pop\_initiala, dim, n):  
 #pop\_initiala = gen(dim)  
 lista\_gene = []  
 lista\_fitness = []  
 for i in range(dim):  
 individ = pop\_initiala[i][:n]  
 lista\_gene = lista\_gene + [individ]  
 lista\_fitness = lista\_fitness + [pop\_initiala[i][n]]  
 lista\_gene\_parinti, fitness\_parinti = turneu(lista\_gene, lista\_fitness, dim,2)  
 parinti = []  
 for i in range(dim):  
 x = [None]\*(n+1)  
 x[:n] = lista\_gene\_parinti[i].copy()  
 x[n] = fitness\_parinti[i]  
 parinti = parinti + [x]  
 #pop\_initiala = np.asarray(pop\_initiala)  
 parinti = np.asarray(parinti)  
 lista\_gene\_parinti=np.asarray(lista\_gene\_parinti)  
 return parinti, lista\_gene\_parinti  
  
  
  
  
#mutatia  
def mutatie\_populatie(pop, dim,n,pm):  
 # copiem populatia curenta in rezultatul pop\_m  
 pop\_m = pop.copy()  
 pop\_fitness=[]  
 for i in range(dim): # pentru fiecare individ  
 # copiem in x individul i  
 x = pop[i][:n].copy()  
 for j in range(3): # pentru fiecare gena  
 # genereaza aleator daca se face mutatie in individul i gena j  
 r = np.random.uniform(0, 1)  
 if r <= pm:  
 # mutatie  
 x[j] = m\_ra(0,101)  
  
 x=[x[0],x[1],x[2]];  
  
 fez, val = avioane(x) # recalculez daca individul mutant este in continuare fezabil dodv al costului maxim  
 if fez: # daca este fezabil va fi inlocuit individul curent cu cel mutant; daca nu este fezabil va ramane in populatia mutanta individul initial(nemutant)  
  
 x = list(x)  
 #x = x + [val]  
 pop\_fitness=pop\_fitness+[val]  
 pop\_m[i] = x.copy()  
 pop\_m = np.asarray(pop\_m)  
 return pop\_m,pop\_fitness  
  
  
#generarea copiilor  
def crossover\_populatie(populatie, dim,n, pc):  
 populatie\_copii = populatie.copy()  
 # populatia este parcursa astfel incat sunt selectati indivizii 0,1 apoi 2,3 s.a.m.d  
 for i in range(0, dim - 1, 2):  
 # selecteaza parintii  
 x = populatie[i].copy()  
 y = populatie[i + 1].copy()  
 r = np.random.uniform(0, 1)  
 if r <= pc:  
 c1, c2 = crossover\_unipunct(x, y, n)  
 fez, val = avioane(c1)  
 if fez:  
 c1 = c1 + [val]  
 populatie\_copii[i] = c1.copy()  
 fez, val = avioane(c2)  
 if fez:  
 c2 = c2 + [val]  
 populatie\_copii[i + 1] = c2.copy()  
 return populatie\_copii  
  
  
def arata(sol, v):  
 # vizualizare rezultate Rucsac 0-1  
 t = len(v)  
 val = max(v)  
 print("Cea mai buna valoare calculată: ", val)  
 print("Alegerea corespunzatoare este: ", sol)  
 fig = grafic.figure()  
 x = [i for i in range(t)]  
 y = [v[i] for i in range(t)]  
 grafic.plot(x, y, 'ro-')  
 grafic.ylabel("Valoarea")  
 grafic.xlabel("Generația")  
 grafic.title("Evoluția calității celui mai bun individ din fiecare generație")  
 fig.show()  
  
#generatia urmatoare  
  
def Generatia\_urmatoare(populatie\_initiala,populatie\_copii,dim,n):  
 populatie\_initiala\_gene=[]  
 populatie\_initiala\_fitness=[]  
  
 for i in range(dim):  
 individ=populatie\_initiala[i][:n]  
 populatie\_initiala\_gene = populatie\_initiala\_gene + [individ]  
 populatie\_initiala\_fitness = populatie\_initiala\_fitness + [populatie\_initiala[i]]  
  
 copii\_mutanti\_gene = []  
 # lista calitatilor  
 copii\_mutanti\_fitness = []  
 for i in range(dim):  
 individ = populatie\_copii[i][:n]  
 copii\_mutanti\_gene = copii\_mutanti\_gene + [individ]  
 copii\_mutanti\_fitness = copii\_mutanti\_fitness + [populatie\_copii[i][:n]]  
  
 pop\_urmatoare\_gene, pop\_urmatoare\_fitness = elitism(populatie\_initiala\_gene, populatie\_initiala\_fitness, copii\_mutanti\_gene,  
 copii\_mutanti\_fitness, dim)  
  
  
 return pop\_urmatoare\_gene, pop\_urmatoare\_fitness  
#Apel  
#import avioane as a  
#pop,pop\_f=a.generare(10)  
#pop\_parinti,gene\_parinti=a.SelectieParinti(pop,10,3)  
#populatie\_copii=a.crossover\_populatie(pop\_parinti,10,3,0.7)  
#populatie\_mutanta=a.mutatie\_populatie(populatie\_copii,10,3,0.1)  
#pop\_urmatoare,pop\_urmatoare\_fitness=a.Generatia\_urmatoare(pop,populatie\_mutanta,10,3)  
  
#GA  
def GA(dim,NMAX,pc,pm):  
 pop\_initiala,dim,n,pop\_fara\_fitness=generare(10)  
 pop\_initiala\_gene = []  
 # lista calitatilor  
 pop\_initiala\_fitness = []  
 for i in range(dim):  
 individ = pop\_initiala[i][:n]  
 pop\_initiala\_gene = pop\_initiala\_gene + [individ]  
 pop\_initiala\_fitness = pop\_initiala\_fitness + [pop\_initiala[i][n]]  
 # in istoric\_v pastram cel mai bun cost din populatia curenta, la fiecare moment al evolutiei  
 istoric\_v = [np.max(pop\_initiala\_fitness)]  
 # evolutia - cat timp  
 # - nu am depasit NMAX si  
 # - populatia are macar 2 indivizi cu calitati diferite si  
 # - in ultimele NMAX/4 iteratii s-a schimbat macar o data calitatea cea mai buna  
  
 # initializari pentru GA  
 it = 0 # contor pentru numarul de iteratii (generatii) parcurse  
 gata = False # flag pentru eventuala oprire a algoritmului daca este atinsa una dintre conditiile de oprire  
 nrm = 1 # contor pentru numarul maxim de iteratii consecutive fara imbunatatirea celui mai bun individ  
  
 while it < NMAX and not gata:  
 parinti,lista\_gene\_parinti=SelectieParinti(pop\_initiala,dim,n)  
 pop\_copii=crossover\_populatie(lista\_gene\_parinti,dim,n,pc)  
 pop\_copii\_mutanti,pop\_copii\_mutanti\_fitness= mutatie\_populatie(pop\_copii,dim,n,pm)  
 pop\_urmatoare\_gene, pop\_urmatoare\_fitness=Generatia\_urmatoare(pop\_initiala,pop\_copii\_mutanti,dim,n)  
 minim = np.min(pop\_urmatoare\_fitness)  
 maxim = np.max(pop\_urmatoare\_fitness)  
 # daca cel maik bun individ de acum e egal cu ultimul adaugat in lista cu cei mai buni  
 if maxim == istoric\_v[it]:  
 nrm = nrm + 1 # inseamna ca nu s-a imbunatatit in generatia curenta, incrementam nr de iteratii consecutive fara imbunatatire  
 else:  
 nrm = 0 # la orice imbunatatire resetam contorul la zero  
  
 # daca max=min (adica toti rindivizii sunt identici calitativ) sau daca in ultimele nmax/4 iteratii consecutive nu s-a imbunatatit calitatea  
 if maxim == minim or nrm == int(NMAX / 4):  
 gata = True # opresc algoritmul la aceasta generatie  
 else:  
 it = it + 1 # altfel, incrementez contorul de generatii pentru ca voi trece la urmatoarea daca nu am ajuns la ultima  
  
 # salvez cel mai bun individ in istoric  
 istoric\_v.append(np.max(pop\_urmatoare\_fitness))  
  
 # initializez populatia initiala de la urmatorul pas cu populatia urmatoare de la generatia curenta (gene si fitness)  
 pop\_initiala\_gene = pop\_urmatoare\_gene.copy()  
 pop\_initiala\_fitness = pop\_urmatoare\_fitness.copy()  
 # de aici algoritmul se reia de sus, de la while, pentru (eventuala) urmatoare generatie  
  
 # la acest moment s-a iesit din while, deci algoritmul a luat sfarsit, ramane sa procesam datele obtinute  
 # transformarea din lista in vector pentru a aplica functia where corect  
 pop\_initiala\_fitness = np.array(pop\_initiala\_fitness)  
 poz\_max = np.where(pop\_initiala\_fitness == maxim)  
 individ\_max\_gene = pop\_initiala\_gene[poz\_max[0][0]]  
 individ\_max\_fitness = maxim  
 arata(individ\_max\_gene, istoric\_v)  
 return individ\_max\_gene, individ\_max\_fitness

Functia obiectiv si generare la permutare circulara

Într-un fișier text sunt păstrate n numere naturale, n > 15. Utilizați un algoritm genetic

#pentru a determina o aranjare circulară a acestor numere astfel încât suma produselor oricăror

#două numere din poziții pare consecutive să fie maximă.

def fob(x):

lista=[3,4,12,9,6,6,5,2,1,1]

n=10

suma=0

for i in range(0,n-2,2):

suma=suma+lista[x[i]]\*lista[x[i+2]]

suma=suma+lista[x[n-2]]\*lista[x[0]]

return suma

def gen(dim):

pop=np.zeros((dim,10),dtype=int)

val=np.zeros((dim,1), dtype=float)

for i in range(dim):

pop[i]=np.random.permutation(10)

val[i]=fob(pop[i])

return [pop, val]

Diferite probleme facut cu Rucsac Discret/Continuu

#pb cu investiile  
  
def functieInvestii(x):  
 #functia fitness reprezinta profitul  
 val=(x[0]\*0.04+x[1]\*0.07+x[2]\*0.06)  
  
 #valoare investita < 100.000  
 suma\_investita=x[0]+x[1]+x[2]  
  
 #factorul de risc < 2.1  
 risc\_mediu=(x[0]+3\*x[1]+2\*x[2])/(x[0]+x[1]+x[2]);  
  
  
  
  
 #if suma\_investita<100000 and risc\_mediu<2.1:  
 #return true,val  
  
 return x[0]<x[1] and x[0]<x[2] and suma\_investita<100000 and risc\_mediu<2.1, val  
  
def genInvestitie(dim):  
  
 # n=dimensiunea problemei  
  
 # lucreaza cu populatia ca lista de dim elemente - liste cu cate n+1 indivizi  
 pop = []  
 for i in range(dim):  
 gata = False # flag  
 while gata == False:  
 # genereaza candidatul x cu elemente pe [0,1]  
  
 x=[]  
 t2=np.random.uniform(1,100000);  
 t3=np.random.uniform(1,100000-t2);  
 t1=np.random.uniform(1,100000-t2-t3);  
  
 x=[t1,t2,t3];  
  
 gata, val =functieInvestii(x)  
 # am gasit o solutie candidat fezabila, in data de tip ndarray (vector) x  
 # x este transformat in lista  
 x = list(x)  
 # adauga valoarea  
 x = x + [val]  
 # adauga la populatie noul individ cu valoarea f. obiectiv - adauga inca o lista cu n+1 elemente ca element  
 # al listei pop  
 pop = pop + [x]  
 pop = np.asarray(pop)  
 return pop  
  
#problema cu tunuri  
def functieTunuri(x):  
 suma=0  
 val=0  
 #sa nu se depaseasca suma de 50 000  
  
 suma=x[0]\*1000+x[1]\*500+x[2]\*750  
  
 #distanta medie  
 distanta\_medie=(500\*x[0]+300\*x[1]+800\*x[2])/(x[0]+x[1]+x[2])  
  
 #fitness-ul  
  
 numar\_ostasi=(7\*x[0]+5\*x[1]+10\*x[2])/(x[0]+x[1]+x[2])  
  
 #ptr val de minim se ia 1/fitness, pentru maxim se ia fitness  
  
 return suma<=50000 and distanta\_medie>=520, 1/numar\_ostasi  
  
  
def generareTunuri(dim):  
  
 pop = []  
 for i in range(dim):  
 gata = False # flag  
 while gata == False:  
  
 x=[]  
 tun1=np.random.randint(0,51)  
 tun2=np.random.randint(0,101)  
 tun3=np.random.randint(0,67)  
  
 x=[tun1,tun2,tun3]  
  
 gata, val = functieTunuri(x)  
 # am gasit o solutie candidat fezabila, in data de tip ndarray (vector) x  
 # x este transformat in lista  
 x = list(x)  
 # adauga valoarea  
 x = x + [val]  
 # adauga la populatie noul individ cu valoarea f. obiectiv - adauga inca o lista cu n+1 elemente ca element  
 # al listei pop  
 pop = pop + [x]  
 pop = np.asarray(pop)  
 return pop

UPU/ Problema de stari

import numpy as np

import matplotlib.pyplot as grafic

from FunctiiMutatieIndivizi import m\_ra, m\_fluaj

from FunctiiSelectii import elitism, ruleta

# functia obiectiv

def foUPU(p, fr, n):

val = 0

for i in range(n):

for j in range(n):

val += fr[i, j] \* (1.7 + 3.4 \* np.sqrt(np.square(p[0] - i) + np.square(p[1] - j)))

return 1 / val

# genereaza populatia initiala

# I:

# ff - numele fisierului frecventelor

# dim - numarul de indivizi din populatie

# E: pop, qual - populatia initiala si vectorul calitatilor

# fr, n:matricea frecventelor si dimeniunea problemei - pentru transferuri in alte functii

def gen(ff, dim):

# citeste datele din fisier

fr = np.genfromtxt(ff)

# n=dimensiunea problemei

n = len(fr)

# defineste o variabila ndarray vida

pop = np.zeros([dim, 2], dtype='int')

qual = np.zeros(dim)

for i in range(dim):

pop[i] = np.random.randint(0, n, 2)

qual[i] = foUPU(pop[i], fr, n)

return pop, qual, n, fr

# crossover pe populatia de parinti pop, cu vectorul de calitati qual

# I: pop,qual,dim,n - ca mai sus

# fr - datele problemei

# pc- probabilitatea de crossover

# E: po,qo - populatia copiilor, calitati qo

# este implementata recombinarea asexuata

def crossover\_populatie(pop, qual, dim, n, fr, pc):

po = pop.copy()

qo = qual.copy()

# populatia este parcursa astfel incat sunt selectati indivizii dupa un amestec

poz = np.random.permutation(dim)

for i in range(0, dim - 1, 2):

# selecteaza parintii

i1 = poz[i]

i2 = poz[i + 1]

x = pop[i1].copy()

y = pop[i2].copy()

r = np.random.uniform(0, 1)

if r <= pc:

# crossover x cu y - unipunct pentru o pozitie

po[i1][0] = x[0]

po[i1][1] = y[1]

po[i2][0] = y[0]

po[i2][1] = x[1]

qo[i1] = foUPU(po[i1], fr, n)

qo[i2] = foUPU(po[i2], fr, n)

return po, qo

# mutatie asupra populatiei de copii

# I:pop,qual,dim,n - populatia, calitatile, dimensiunea matricei fr

# pm - probabilitatea de mutatie

# E: - mpo,mqo - populatia mutata, vectorul calitatilor

def mutatie\_populatie(pop, qual, dim, n, fr, pm):

mpo = pop.copy()

mqo = qual.copy()

x = np.zeros(2, dtype='int')

for i in range(dim):

x[:] = pop[i]

for j in range(2):

# genereaza aleator daca se face mutatie in individul i gena j

r = np.random.uniform(0, 1)

if r <= pm:

# mutatie prin resetare aleatoare

x[j] = m\_ra(0, n)

mpo[i] = x.copy()

mqo[i] = foUPU(mpo[i], fr, n)

return mpo, mqo

def arata(sol, v):

# vizualizare rezultate

n = len(sol)

t = len(v)

val = max(v)

print("Cea mai buna valoare calculată: ", 1 / val)

print("Alegerea corespunzatoare este: ", sol)

fig = grafic.figure()

x = [i for i in range(t)]

y = [1 / v[i] for i in range(t)]

grafic.plot(x, y, 'ro-')

grafic.ylabel("Valoarea")

grafic.xlabel("Generația")

grafic.title("Evoluția calității celui mai bun individ din fiecare generație")

fig.show()

##ALGORITMUL GENETIC PENTRU REZOLVAREA PROBLEMEI PLASARII UPU

# I: ff - fisierul cu frecvente

# dim - dimensiunea unei populatii

# NMAX - numarul maxim de simulari ale unei evolutii

# pc - probabilitatea de crossover

# pm - probabilitatea de mutatie

#

# E: sol - solutia calculata de GA

# val - 1/maximul functiei fitness - cost\_minim

def GA(ff, dim, NMAX, pc, pm):

# generarea populatiei la momentul initial

pop, qual, n, fr = gen(ff, dim)

# initializari pentru GA

it = 0

gata = False

# in istoric\_v pastram cel mai bun cost din populatia curenta, la fiecare moment al evolutiei

istoric\_v = [np.max(qual)]

# evolutia - cat timp

# - nu am depasit NMAX si

# - populatia are macar 2 indivizi cu calitati diferite si

# - in ultimele NMAX/3 iteratii s-a schimbat macar o data calitatea cea mai buna

nrm = 1

while it < NMAX and not gata:

spop, sval = ruleta(pop, qual, dim, 2)

po, qo = crossover\_populatie(spop, sval, dim, n, fr, pc)

mpo, mqo = mutatie\_populatie(po, qo, dim, n, fr, pm)

newpop, newval = elitism(pop, qual, mpo, mqo, dim)

minim = np.min(newval)

maxim = np.max(newval)

if maxim == istoric\_v[it]:

nrm = nrm + 1

else:

nrm = 0

if maxim == minim or nrm == int(NMAX / 2):

gata = True

else:

it = it + 1

istoric\_v.append(np.max(newval))

pop = newpop.copy()

qual = newval.copy()

i\_sol = np.where(qual == maxim)

sol = pop[i\_sol[0][0]]

val = maxim

arata(sol, istoric\_v)

return sol, 1 / val

# import PlasareUPU\_test as pu

# sol,val=pu.GA("frecventa2.txt",30,20,0.8,0.2)

LABIRINT

import numpy

import time

from Labirint import ind2sub, sub2ind

from Selectii import s\_ruleta\_SUS, s\_superelitista

from Recombinari import r\_unipunct

from Mutatii import m\_int\_ra

import matplotlib.pyplot as plt

import matplotlib.colors as mpcolors

import datetime

ZID=1

LIBER=3

VIZITAT=2

BONUS=10000

def GA\_Labirint(fl, sx, sy, fx, fy, dim, nmax, pr, pm):

# gasirea unui traseu in labirint de la (sx,sy) la (fx,fy); miscari N=0, E=1, S=2, W=3, nimic=-1

# I: fl - fisier .npy cu matricea care descrie labirintul: 1=zid, 3=liber

# se creaza cu functia create\_maze din Labirint sau Labirint2 (mici diferente doar la animatie)

# sx, sy - coordonate celula start / intrare (uzual coltul stinga sus: 2,2)

# fx, fy - coordonate celula finala / iesire (uzual coltul dreapta jos: 2\*dim,2\*dim)

# dim - dimensiunea populatiei de lucru

# nmax - numar maxim de generatii

# pr, pm - probabilitati de recombianre si mutatie

# E: sol - cea mai buna solutie gasita (set miscari)

# vmax - calitatea solutiei gasite (numar de miscari, bonus pentru gasire iesire)

# v - vector, evolutia calitatii celui mai bun individ din fiecare generatie

# Exemple de apel

# import GA\_Labirint as GL

# s,vmx,v=GL.GA\_Labirint('L\_15x15\_1.npy',2,2,30,30,30,100,0.8,0.2)

# Evoluție în 100 generații

# Soluție= [2 2 2 1 2 2 1 1 1 1 1 1 1 2 2 2 1 1 2 1 2 1 2 2 1 1 2 2]

# Soluție= S S S E S S E E E E E E E S S S E E S E S E S S E E S S

# Soluția găsită ajunge la destinație în 28 mișcări

# Am găsit 1 drumuri distincte

# Timp consumat de algoritmul genetic: 0:00:11.776645

# s,vmx,v=GL.GA\_Labirint('L\_20x20\_1.npy',2,2,40,40,30,100,0.8,0.2)

# Evoluție în 100 generații

# Soluție= [2 1 1 1 2 1 2 2 1 2 2 2 2 2 1 2 2 1 2 1 2 2 1 2 2 1 1 1 2 2 2 1 0 1 1 1 1 1 2 1]

# Soluție= S E E E S E S S E S S S S S E S S E S E S S E S S E E E S S S E N E E E E E S E

# Soluția găsită ajunge la destinație în 40 mișcări

# Am găsit 3 drumuri distincte

# Timp consumat de algoritmul genetic: 0:00:33.544874

# initializari

timp\_inceput=datetime.datetime.now()

L = numpy.load(fl)

v = numpy.zeros(nmax)

\_, n = numpy.shape(L)

m = (n - 3) // 2

c = [sx, sy, fx, fy] # capetele traseului, pentru a usura scrierea la transmiterea lor ca parametri

# generare populatie initiala

pop = gen\_ini(dim, L, c)

# bucla GA

print("Muncesc ...")

i=0

gata=False

while i<nmax and not gata:

if i / 10 == i // 10:

print("Generația ", i)

# sortare populatie

pop = pop[pop[:, -1].argsort()]

# selectie parinti

parinti = s\_ruleta\_SUS(pop)

# recombinare

desc = recombinare(parinti, pr, L, c)

# mutatie

descm = mutatie(desc, pm, L, c)

# selectie generatie urmatoare

pop = s\_superelitista(pop, descm)

# alte operatii

vmax = max(pop[:, -1])

v[i] = vmax

vmin=min(pop[:,-1])

gata= vmin==vmax

i+=1

# calcul timp consumat

durata=datetime.datetime.now()-timp\_inceput

# cea mai buna solutie

poz = numpy.argmax(pop[:, -1])

sol = pop[poz, :-1]

# aplica solutia pe labirint si afiseaza

sol, \_ = fix\_ind(sol, L, c)

print("Evoluție în",i,"generații")

# inlocuire valori numerice cu N E S W

miscari=['N', 'E', 'S', 'W']

sol\_m=[miscari[sol[i]] for i in range(len(sol))]

print("Soluție=", sol)

print("Soluție=", ' '.join(sol\_m))

ok = test\_sol(L, sol, c)

if ok:

print("Soluția găsită ajunge la destinație în",BONUS-vmax,"mișcări")

# solutii care ajung la destinatie si au calitatea cea mai buna

drumuri = pop[pop[:, -1] == vmax]

# cite din ele sint distincte

cite, \_ = numpy.shape(numpy.unique(drumuri, axis=0))

print("Am găsit", cite, "astfel de drumuri distincte")

# numara cite din drumurile gasite ajung la destinatie si sint distincte

# solutii care ajung la destinatie

drumuri=pop[pop[:,-1]>BONUS\*0.8] # ajunge la destinatie in maxim 199 miscari. trebuie modificat pentru labirint mai mare

# cite din ele sint distincte

cite,\_=numpy.shape(numpy.unique(drumuri,axis=0))

print("Am găsit in total",cite,"drumuri distincte care ajung la destinație")

else:

print("Soluția găsită nu ajunge la destinație")

print("Timp consumat de algoritmul genetic:",durata)

fig2 = plt.figure()

plt.plot(v)

return sol, vmax, v

def gen\_ini(dim, L, c):

# generare populatie initiala

# I: dim - dimensiune populatie

# L - labirintul pe care se lucreaza

# c - lista cu coordonatele capetelor labirintului (sx,sy,fx,fy)

# E: pop - populatia generata

\_, n = numpy.shape(L)

m = (n - 3) // 2

pop = numpy.zeros((dim, m \* m + 1), dtype=int)

for i in range(dim):

x = gen\_drum(L, m \* m, c[0], c[1])

pop[i, :-1] = x

pop[i, -1] = f\_obiectiv(x, L, c)

return pop

def gen\_drum(L, n, sx, sy):

# genereaza un drum aleator in labirint, de lungime n

# I: L - labirint (matrice)

# n - lungime drum

# sx, sy - punct de pornire

# E: x - drum generat (secventa de miscari N E S W

m, \_ = numpy.shape(L)

offsets=[-m, 1, m, -1] # ordinea n e s w

crtcell = sub2ind(sx, sy,(m,m))

x = numpy.random.randint(0, 4, n)

for i in range(n):

x[i] = numpy.random.randint(4)

gata = 0

while not gata:

zid = crtcell + offsets[x[i]] # ce e intre celula curenta si urmatoarea, conform miscarii alese

if L[ind2sub(zid,(m,m))] != ZID: # daca nu e miscare in zid

crtcell = crtcell + 2 \* offsets[x[i]]

gata = 1

else:

x[i] = numpy.random.randint(4)

return x

def f\_obiectiv(x, L, c):

# evaluare drum in labirint:

# I: L - labirint

# c - coordonate punct start si punct final (sx,sy,fx,fy)

# E: cal - calitate calculata (negata, pentru a avea functie de maximizat)

# = penalizare pentru neatingerea punctului final (dist.vert. + dist. oriz.)

# + penalizare pentru nr. pasi facuti (cit mai putini cu atit mai bine)

# + penalizare pentru miscari "in perete" (echivalent cu ne-miscare)

# - bonus pentru atingere punct final

# totul negat pentru transformare in functie de maximizat

\_, m = numpy.shape(L)

offsets = [-m, 1, m, -1] # ordinea n e s w

n = len(x)

p = 0 # penalizare pentru miscari "in perete"

crtcell = sub2ind(c[0], c[1], (m, m))

dest = 0 # bonus pentru atingere destinatie

pasi = n # numar pasi in solutie

i = 0

while i < n and dest == 0: # parcurge cromozom, oprire daca atinge punctul final

if x[i] != -1: # daca nu e ne-miscare

zid = crtcell + offsets[x[i]]

if L[ind2sub(zid, (m, m))] != ZID: # daca nu e miscare in zid

crtcell = crtcell + 2 \* offsets[x[i]]

else:

p = p + 1 # aduna penalizare pentru o miscare in zid

cx, cy = ind2sub(crtcell, (m, m))

if cx == c[2] and cy == c[3]: # ajunge la destinatie?

dest = -BONUS

x[i + 1:n] = -1 # restul miscarilor nu mai sint necesare

p = 0

y, \_ = fix\_ind(x, L, c) # "aranjeaza" cromozom, elimina cicluri

l = len(y)

x = -numpy.ones(n) # umple individul cu ne-miscari

x[:l] = y[:l] # si copiaza doar miscarile ramase

pasi = l

i += 1

# calitate = -cost drum

cal = -(abs(cx - c[2]) + abs(cy - c[3]) + p + dest + pasi)

return cal

def fix\_ind(x, L, c):

# "aranjeaza" un cromozom prin eliminarea ciclurilor

# I: x - cromozom

# L - labirint

# c - coordonate punct start si punct final (sx, sy, fx, fy)

# E: Y - cromozom curatat de cicluri (lista de miscari)

# cells - lista celule prin care trece drumul (ranguri in matricea liniarizata)

\_, m = numpy.shape(L)

mm = (m - 3) // 2

offsets = [-m, 1, m, -1] # ordinea n e s w

crtcell = sub2ind(c[0], c[1], (m, m))

cells = numpy.zeros(mm \* mm + 1)

cells[0] = crtcell

i = 0

ok = 0

while i < mm \* mm and ok == 0:

# calculeaza urmatoarea celula

if x[i] != -1: # daca nu e ne-miscare

zid = crtcell + offsets[x[i]]

if L[ind2sub(zid, (m, m))] != ZID: # daca nu e miscare in zid

crtcell = crtcell + 2 \* offsets[x[i]]

# altfel, ramin in celula curenta

cells[i + 1] = crtcell

# daca am ajuns la punctul final sterge urmatoarele miscari (inlocuieste cu -1)

cx, cy = ind2sub(crtcell, (m, m))

if cx == c[2] and cy == c[3]:

ok = 1

x = x[:i + 1]

cells = cells[:i + 2]

i += 1

# elimina cicluri

l = len(cells)

for i in range(1, l):

# celula curenta a mai fost parcursa pina acum?

p = numpy.where(cells[:i] == cells[i]) # aici am inlocuit :i+1 cu :i

if len(p[0]) != 0:

x[p[0][0]:i] = -1

cells[p[0][0] + 1:i+1] = -1 # aici am inlocuit i cu i+1

# elimina ne-miscarile dn cromozom

p = numpy.where(x != -1)

y = x[p]

return y, cells

def recombinare(parinti, pr, L, c):

# etapa de recombinare

# I: parinti - multisetul de parinti

# pr - probabilitate de recombinare

# L - labirint

# c - coordonate puncte de start si final (sx,sy,fx,fy)

# E: desc - descendenti generati

desc = parinti.copy()

dim, n = numpy.shape(parinti)

perechi = numpy.random.permutation(dim)

for i in range(0, dim, 2):

x = parinti[perechi[i], :n - 1]

y = parinti[perechi[i + 1], :n - 1]

d1, d2 = r\_unipunct(x, y, pr)

desc[i, :n - 1] = d1

desc[i][n - 1] = f\_obiectiv(d1, L, c)

desc[i + 1, :n - 1] = d2

desc[i + 1][n - 1] = f\_obiectiv(d2, L, c)

return desc

def mutatie(desc, pm, L, c):

# operatia de mutatie a descendentilor obtinuti din recombinare

# I: desc - multisetul de indivizi obtinuti din recombinare

# pm - probabilitate de mutatie

# L - labirint

# c - coordonate puncte de start si final (sx,sy,fx,fy)

# E: descm - matricea indivizilor obtinuti

dim, n = numpy.shape(desc)

descm = desc.copy()

for i in range(dim):

x = descm[i, :n - 1]

y = m\_int\_ra(x, pm, 0, 3)

descm[i, :n - 1] = y

descm[i][n - 1] = f\_obiectiv(y, L, c)

return descm

def test\_sol(L, x, c):

# verifica solutie labirint si ajusteaza

# I: L - labirint

# x - solutia testata

# c - coordonate puncte de start si final

# E: ok - 1 daca ajunge la punctul final, 0 altfel

m, \_ = numpy.shape(L)

cm2 = mpcolors.ListedColormap([(0, 0, 0), (0, 1, 0), (1, 1, 1)])

offsets = [-m, 1, m, -1] # ordinea N E S W

crtcell = sub2ind(c[0], c[1], (m, m))

L[c[0], c[1]] = VIZITAT

i = 0

ok = 0

l = len(x)

while i < l and ok == 0:

if x[i] != -1: # nu e ne-miscare

zid = crtcell + offsets[x[i]]

if L[ind2sub(zid, (m, m))] != ZID: # nu e miscare in zid

L[ind2sub(zid, (m, m))] = VIZITAT

crtcell += 2 \* offsets[x[i]]

L[ind2sub(crtcell, (m, m))] = VIZITAT

# aici afisare succesiva miscari???

# am ajuns la final?

cx, cy = ind2sub(crtcell, (m, m))

if cx == c[2] and cy == c[3]:

ok = 1

i += 1

plt.imshow(L, cm2, aspect='equal')

plt.xticks([]), plt.yticks([])

return ok

Transport

import numpy

import matplotlib.pyplot as grafic

from Selectii import s\_ruleta\_SUS, s\_elitista

from Recombinari import r\_CX

from Mutatii import m\_perm\_schimb

def GA\_Transport(fo,fc,fcost,dim,nmax,pr,pm):

# rezolvarea problemei generale de transport

# fenotip: matricea de transport

# genotip: ordinea in care se aloca elementele din matricea de transport <=> permutare, ind. mereu fezabili

# I: fo - fisier oferta (text, 1xm)

# fc - fisier cerere (text, 1 x n)

# fcost - fisier costuri (text, m x n)

# dim - dimensiune populatie

# nmax - numar maxim de iteratii

# pr - probabilitate de recombinare

# pm - probabilitate de mutatie

# E: sol - solutia de transport gasita

# cost - cost de transport calculat

# Exemple de apel:

# import GA\_Transport as GTR

# s,c=GTR.GA\_Transport('T\_oferta3.txt','T\_cerere3.txt','T\_costuri3.txt',10,50,0.8,0.1)

# Cel mai mic cost gasit: 2316

# Solutia de transport:

# 6 0 9 0 0

# 2 0 0 10 8

# 3 12 0 0 0

#

# folosing CX si interschimbare:

# s,c=GTR.GA\_Transport('T\_oferta.txt','T\_cerere.txt','T\_Costuri.txt',150,300,0.8,0.1)

# Cel mai mic cost gasit: 39736.1

# Solutia de transport:

# 0 40 0 0 0 0 0 0 0 60

# 60 0 0 90 0 0 0 0 0 0

# 0 0 0 0 0 20 0 0 60 0

# 0 0 10 0 0 10 70 0 0 0

# 0 0 0 0 0 10 0 0 0 0

# 0 0 0 0 0 0 10 50 0 0

# 10 0 0 0 0 70 0 0 0 0

# 0 0 0 0 20 0 30 0 0 0

# 0 0 20 0 0 0 0 0 0 0

# 0 0 0 0 0 70 0 0 0 0

# initializari

oferta=numpy.genfromtxt(fo)

cerere=numpy.genfromtxt(fc)

costuri=numpy.genfromtxt(fcost)

# generare populatie initiala

pop=gen\_pop(dim,oferta,cerere,costuri)

v=[min(1000./pop[:,-1])]

# bucla GA

ok=True

t=0

while t<nmax and ok:

# selectie parinti

parinti = s\_ruleta\_SUS(pop)

# recombinare

desc = recombinare(parinti, pr, oferta,cerere,costuri)

# mutatie

descm = mutatie(desc, pm, oferta,cerere,costuri)

# selectie generatie urmatoare

pop = s\_elitista(pop, descm)

# alte operatii

# retine cea mai buna solutie

vmax = min(1000./pop[:,-1])

i = numpy.argmin(pop[:, -1])

best = pop[i][:-1]

v.append(vmax)

t+=1

ok=max(pop[:,-1])!=min(pop[:,-1])

print("Cel mai bun cost găsit: ",vmax)

print("Soluția de transport:")

sol=gen\_alocare(best,oferta,cerere)

print(sol)

fig=grafic.figure()

grafic.plot(v)

verificare(sol,oferta,cerere)

return (sol,vmax)

def f\_obiectiv(x,oferta,cerere,costuri):

# functia obiectiv pentru problema de transport

# I: x - cromozom evaluat

# oferta, cerere - restrictii

# costuri - matricea costurilor de transport

# E: c - calitate (1000/cost transport)

a=gen\_alocare(x,oferta,cerere)

c=1000./numpy.sum(a\*costuri)

return c

def gen\_alocare(permutare,oferta,cerere):

# conversie genotip->fenotip = decodificare (alocare resurse cu respectarea restrictiilor)

# I: permutare - genotip, ordinea de alocare

# oferta, cerere - restrictii

# E: x - alocarea obtinuta, sub forma de matrice

m=len(oferta)

n=len(cerere)

x=numpy.zeros((m,n))

i=0

OR=sum(oferta) #oferta totala ramasa

#v2

o\_r=oferta.copy() #oferta ramasa

c\_r=cerere.copy() #cerere ramasa

#end v2

while OR>0:

lin,col=numpy.unravel\_index(int(permutare[i]),(m,n))

#v1

#o\_r=oferta(lin)-sum(x[lin,:])

#c\_r=cerere(col)-sum(x[:,col])

#a=min([o\_r,c\_r])

#x[lin,col]=a #aloca maximul posibil - min dintre oferta si cerere

#OR-=a

#end v1

#v2

x[lin,col]=min([o\_r[lin],c\_r[col]])

o\_r[lin]-=x[lin,col]

c\_r[col]-=x[lin,col]

OR-=x[lin,col]

#end v2

i+=1

return x

def gen\_pop(dim,oferta,cerere,costuri):

# generare populatie initiala pentru problema de transport

# I: dim - dimensiune populatie

# oferta - oferta de transport (1xm)

# cerere = cerere de transport (1xn)

# costuri - costuri de transport (mxn)

# E: pop - populatia generata

m=len(oferta)

n=len(cerere)

pop=numpy.zeros((dim,m\*n+1))

for i in range(dim):

x=numpy.random.permutation(m\*n)

pop[i,:-1]=x

pop[i,-1]=f\_obiectiv(x,oferta,cerere,costuri)

return(pop)

def recombinare(parinti,pr,oferta,cerere,costuri):

# etapa de recombinare

# I: parinti - multiseutl parintilor

# pr - probabilitatea de recombianre

# oferta, cerere, costuri - paramatri ai problemei

# E: desc - descendentii creati

dim,n=numpy.shape(parinti)

desc=numpy.zeros((dim,n))

#selectia aleatoare a perechilor de parinti

perechi=numpy.random.permutation(dim)

for i in range(0,dim,2):

x = parinti[perechi[i], :n - 1]

y = parinti[perechi[i + 1], :n - 1]

#d1, d2 = r\_OCX(x, y, pr)

d1, d2 = r\_CX(x, y, pr)

desc[i, :n - 1] = d1

desc[i][n - 1] = f\_obiectiv(d1, oferta,cerere,costuri)

desc[i + 1, :n - 1] = d2

desc[i + 1][n - 1] = f\_obiectiv(d2, oferta,cerere,costuri)

return desc

def mutatie(desc,pm,oferta,cerere,costuri):

# etapa de mutatie

# I: desc - descendentii obtinuti in etapa de recombianre

# pm - probabilitatea de mutatie

# oferta,cerere,costuri - restrictiile problemei

# E: descm - descendenti modificati

dim,n=numpy.shape(desc)

descm=desc.copy()

for i in range(dim):

x=descm[i,:n-1]

y=m\_perm\_schimb(x,pm)

descm[i,:n-1]=y

descm[i,n-1]=f\_obiectiv(y,oferta,cerere,costuri)

return descm

def verificare(sol,oferta,cerere):

# verificare solutie

# I: sol - solutia problemei de transport

# oferta,cerere - restrictiile problemei

# E: -

o\_r=oferta-numpy.sum(sol,axis=1)

c\_r=cerere-numpy.sum(sol,axis=0)

mino=min(o\_r); maxo=max(o\_r)

minc=min(c\_r); maxc=max(c\_r)

print("Oferta rămasă:", o\_r)

if mino<0:

print("Eroare la ofertă: se consumă mai mult de cît e disponibil")

if maxo>0:

print("Eroare la ofertă: se consumă mai puțin de cît e disponibil")

if mino==0 and maxo==0:

print("Oferta e consumată perfect")

print("Cerere rămasă:", c\_r)

if minc<0:

print("Eroare la cerere: se transportă mai mult de cît se cere")

if maxc>0:

print("Eroare la cerere: se transportă mai putin de cît se cere")

if minc==0 and maxc==0:

print("Cererea acoperită perfect")

Grecia

import numpy as np

import matplotlib.pyplot as grafic

# Multimea de candidati este alcatuita din totalitatea permutarilor de N componente.

# Scaunele de la masa circulara sunt numerotate de la 0 la N-1, iar valoarea x a genei corespunzatoare scaunului este reprezentantul orasului x

def fps(qual, dim):

fps = np.zeros(dim)

suma = np.sum(qual)

for i in range(dim):

fps[i] = qual[i] / suma

qfps = fps.copy()

for i in range(1, dim):

qfps[i] = qfps[i - 1] + fps[i]

return qfps

def sigmafps(qual, dim):

med = np.mean(qual)

var = np.std(qual)

newq = [max(0, qual[i] - (med - 2 \* var)) for i in range(dim)]

if np.sum(newq) == 0:

qfps = fps(qual, dim)

else:

qfps = fps(newq, dim)

return qfps

def ruleta(pop, qual, dim, n):

spop = pop.copy()

squal = np.zeros(dim)

qfps = sigmafps(qual, dim)

for it in range(dim):

r = np.random.uniform(0, 1)

poz = np.where(qfps >= r)

isel = poz[0][0]

spop[it][:] = pop[isel][:]

squal[it] = qual[isel]

return spop, squal

def crossover\_PMX(x, y, n):

# generarea secventei de crossover

poz = np.random.randint(0, n, 2)

while poz[0] == poz[1]:

poz = np.random.randint(0, n, 2)

p1 = np.min(poz)

p2 = np.max(poz)

c1 = PMX(x, y, n, p1, p2)

c2 = PMX(y, x, n, p1, p2)

return c1, c2

def PMX(x, y, n, p1, p2):

# initializare copil - un vector cu toate elementele -1 - valori care s=sa nu fie in 0,...,n-1

c = -np.ones(n, dtype=int)

# copiaza secventa comuna in copilul c

c[p1:p2 + 1] = x[p1:p2 + 1]

# analiza secventei comune - in permutarea y

for i in range(p1, p2 + 1):

# plasarea alelei a

a = y[i]

if a not in c:

curent = i

plasat = False

while not plasat:

b = x[curent]

# poz=pozitia in care se afla b in y

[poz] = [j for j in range(n) if y[j] == b]

if c[poz] == -1:

c[poz] = a

plasat = True

else:

curent = poz

# z= vectorul alelelor din y inca necopiate in c

z = [y[i] for i in range(n) if y[i] not in c]

# poz - vectorul pozitiilor libere in c - cele cu vaori -1

poz = [i for i in range(n) if c[i] == -1]

# copierea alelelor inca necopiate din y in c

m = len(poz)

for i in range(m):

c[poz[i]] = z[i]

return c

def crossover\_populatie(pop, dim, n, pc, riv):

# initializeaza populatia de copii, po, cu populatia parintilr

po = pop.copy()

# populatia este parcursa astfel incat sunt selectati indivizii 0,1 apoi 2,3 s.a.m.d

for i in range(0, dim - 1, 2):

# selecteaza parintii

x = pop[i]

y = pop[i + 1]

r = np.random.uniform(0, 1)

if r <= pc:

# crossover x cu y - OCX - potrivit pentru NRegine

c1, c2 = crossover\_PMX(x, y, n)

val1 = calitate(c1, n, riv)

val2 = calitate(c2, n, riv)

po[i][:n] = c1.copy()

po[i][n] = val1

po[i + 1][:n] = c2.copy()

po[i + 1][n] = val2

return po

def m\_perm\_inserare(x, n):

# generarea pozitiilor pentru inversiune

poz = np.random.randint(0, n, 2)

while poz[0] == poz[1]:

poz = np.random.randint(0, n, 2)

p1 = np.min(poz)

p2 = np.max(poz)

y = x.copy()

y[p1 + 1] = x[p2]

if p1 < n - 2:

y[p1 + 2:n] = np.array([x[i] for i in range(p1 + 1, n) if i != p2])

return y

def mutatie\_populatie(pop, dim, n, pm, riv):

mpop = pop.copy()

for i in range(dim):

# genereaza aleator daca se face mutatie

r = np.random.uniform(0, 1)

if r <= pm:

# mutatie in individul i - prin interschimbare

# x=m\_perm\_interschimbare(mpop[i,:n],n)

# mutatie in individul i - prin inserare

x = m\_perm\_inserare(mpop[i, :n], n)

mpop[i, :n] = x.copy()

mpop[i, n] = calitate(x, n, riv)

return mpop

def elitism(pop\_c, qual\_c, pop\_mo, qual\_mo, dim):

pop = np.copy(pop\_mo)

qual = np.copy(qual\_mo)

max\_c = np.max(qual\_c)

max\_mo = np.max(qual\_mo)

if max\_c > max\_mo:

p1 = np.where(qual\_c == max\_c)

# imax = prima pozitie in qual\_c in care este atinsa valoarea maxima

imax = p1[0][0]

ir = np.random.randint(dim)

# inlocuirea

pop[ir] = pop\_c[imax].copy()

qual[ir] = max\_c

return pop, qual

def arata(sol, v):

# vizualizare asezare regine pe tabla de sah

# I: sol - permutarea care defineste asezarea

# v - vectorul cu cea mai buna calitate din fiecare generatie

# E: -

n = len(sol)

t = len(v)

minim = min(v)

t1 = "Evoluția calității (cel mai bun individ din fiecare generație).\nRezultatul "

t2 = "Cea mai bună așezare a reginelor găsită.\nRezultatul "

t4 = "este optim (" + str(minim) + " vs 0" + ")"

if minim > 0:

t3 = "nu "

else:

t3 = ""

titlu1 = f'{t1}{t3}{t4}'

titlu2 = t2 + t3 + t4

fig1 = grafic.figure()

x = [i for i in range(t)]

grafic.plot(x, v, 'ro-')

grafic.ylabel("Calitate")

grafic.xlabel("Generația")

grafic.title(titlu1)

def citesteMatrice(fis):

riv = np.genfromtxt(fis)

return riv # citim matricea de rivalitati

def calitate(x, N, riv):

# cel mai rau caz este cel in care toti de la masa sunt rivali

# unul cu altul. Deci, in cel mai rau caz, cal = dim mesei

cal = N

for i in range(N - 1):

if riv[x[i]][x[i + 1]] == 1:

cal = cal - 1

if riv[x[0]][x[N - 1]] == 1:

cal = cal - 1

return cal

# Construiesc generatia initiala

def gen(N, dim, riv):

pop = np.zeros((dim, N + 1), dtype=int)

for i in range(dim):

pop[i, :N] = np.random.permutation(N)

pop[i, N] = calitate(pop[i, :N], N, riv)

return pop

def Lideri(n, dimPop, fisier, NMAX, pc, pm):

riv = citesteMatrice(fisier)

pop = gen(n, dimPop, riv)

it = 0

gata = False

maxim = np.max(pop[:, n])

print(pop)

print(maxim)

istoric\_v = [n - maxim]

while it < NMAX and not gata and maxim < n:

# selectez parintii prin mecanismul ruleta cu un singur brat

spop, sval = ruleta(pop[:, :n], pop[:, n], dimPop, n)

pop\_s = np.zeros([dimPop, n + 1])

pop\_s[:, :n] = spop.copy()

pop\_s[:, n] = sval.copy()

# recombinarea perechilor de parinti

pop\_o = crossover\_populatie(pop\_s, dimPop, n, pc, riv)

pop\_o = pop\_o.astype(int)

# mutatia populatiei obtinute

pop\_mo = mutatie\_populatie(pop\_o, dimPop, n, pm, riv)

# selectia generatiei urmatoare

newpop, newval = elitism(pop[:, :n], pop[:, n], pop\_mo[:, :n], pop\_mo[:, n], dimPop)

minim = np.min(newval)

maxim = np.max(newval)

# opreste evolutia la populatia cu toti indivizii de aceeasi calitate

if maxim == minim:

gata = True

else:

it = it + 1

istoric\_v.append(n - np.max(newval))

pop[:, :n] = newpop.copy()

pop[:, n] = newval.copy()

i\_sol = np.where(pop[:, n] == maxim)

sol = pop[i\_sol[0][0]][:n]

val = n - maxim

arata(sol, istoric\_v)

return sol, val

Lideri(10, 100, 'rivalitati1.txt', 200, 0.8, 0.2)

#Lideri(20, 200, 'rivalitati2.txt', 300, 0.8, 0.3)

#Lideri(20, 250, 'rivalitati2.txt', 300, 0.8, 0.3)

#Lideri(20, 100, 'rivalitati3.txt', 300, 0.8, 0.3)

JSS

import numpy as np

import matplotlib.pyplot as grafic

from FunctiiCrossoverIndivizi import crossover\_PMX, crossover\_OCX, crossover\_CX

from FunctiiMutatieIndivizi import m\_perm\_inversiune, m\_perm\_interschimbare, m\_perm\_inserare

from FunctiiSelectii import elitism, ruleta

#####

#PLANIFICAREA UNEI ACTIVITATI NEPLANIFICATA INCA - activitatile sunt considerat in ordinea: prima asignata fiecarei masini, a doua asignata fiecarei masini,...,conform cromozomului x

# CROSSOVER - pe o pereche de permutari selectata aleator din perechea de parinti

#####

#citeste datele din fisierele

#I:

# masini\_op - distributia operatiilor fiecarei sarcini pe cate o masina; primul element- nr. de masini

# durata\_op - durata fiecarei operatii

#E:

# nrm - numarul masinilor

# nrs - numarul sarcinilor; - numarul operatiilor=nrm x nrs

# m\_op - vectorul distributiei operatiilor pe masini - (i,m\_op[i]) - operatia i+1 pe masina m\_op[i]

# d\_op - durata fiecarei operatii

def citeste(masini\_op,durata\_op):

a = np.genfromtxt(masini\_op,'int')

nrm=a[0]

nro=len(a)-1

m\_op=np.zeros(nro,'int')

m\_op[:]=a[1:nro+1]

d\_op=np.genfromtxt(durata\_op)

nrs=int(len(d\_op)/nrm)

return nrm,nrs,m\_op,d\_op

#calculul predecesorilor si succesorilor unei operatii

def pred\_succ(op,nrm):

cat,rest=np.divmod(op,nrm)

if rest:

pred=[cat\*nrm+i for i in range(1,rest)]

succ=[cat\*nrm+i for i in range(rest+1,nrm+1)]

else:

pred=[(cat-1)\*nrm+i for i in range(1,nrm)]

succ=[]

return pred,succ

# initializeaza masivul 3D cu planificarile pe fiecare masina, plan\_m si vectorul plan

#I:

# nrm, nrs, m\_op - ca in citire

#E:

# i=0,...,nrm-1:

# plan\_m[i] - matrice cu nrs linii si 3 coloane- plan\_m[i][j]=(operatie, start,stop), pentru j=0,...,nrs-1

# initial plan\_m[i][j]=(operatie,0,0)

# i =0,nro-1:

# plan[i]=0 - timpul de inceput al operatiei i

# planificat[i]=0 - daca operatia i este sau nu planificata

def init\_plan(nrm,nrs,m\_op):

nro=nrm\*nrs

plan\_m = np.zeros([nrm, nrs, 3])

plan=np.zeros(nro)

planificat = np.zeros(nro)

for i in range(nro):

# plaseaza operatia i pe masina corespunzatoare, cu start, stop=0

m = m\_op[i] - 1

p = np.where(plan\_m[m])

nr = int(np.size(p) / 2)

plan\_m[m][nr][0] = i + 1

# de exemplu, pentru masina i cu activitatile alocate ei a1, a2,...,anrs din sarcinile 1,2,..., nrs rezulta

#plan\_m[i]=[a1 0 0

# a2 0 0

#...

#anrs 0 0], deci conform permutarii identice

return plan,planificat,plan\_m

#calculul planificarii corespunzatoare unui cromozom si a costului acesteia

#cromozomul x=[p1,p2,...,pnrm], pj este o permutare care da ordinea in care analizam cele nrs activitati care trebuie

#programate pe masina j, cate una din fiecare sarcina si identificate cu indicii sarcinilor, 0,1,...,nrs-1

def calcul\_plan(x,nrm,nrs,m\_op,d\_op):

plan,planificat,plan\_m=init\_plan(nrm,nrs,m\_op)

# modificam ordinea analizei planificarii fiecarei activitati pe fiecare masina conform cromozomului x

plan\_masina=np.copy(plan\_m)

for i in range(nrm):

#permutarea pe masina i

px=np.zeros(nrs,'int')

px[:]=x[nrs\*i:nrs\*(i+1)]

for k in range(nrs):

#initial masina i are ca activitati asociate pj, conform x

#plan\_masina[i]=[pj(0) 0 0

# pj(1) 0 0

#...

#pj(nrs-1) 0 0], deci conform permutarii identice

plan\_masina[i][k][0]=plan\_m[i][px[k]][0]

gata=0

while not gata:

# pentru toate activitatile neplanificate, in ordinea sarcinilor: prima asignata fiecarei masini, a doua asignata fiecarei masini,...,conform cromozomului x

for k in range(nrs):

for j in range(nrm):

a=int(plan\_masina[j][k][0])

if not planificat[a-1]:

# planificarea activitatii a

plan\_masina,planificat,plan=programare(j,k,plan\_masina,planificat,plan,nrm,nrs,m\_op,d\_op)

ex=np.where(planificat==0)

if np.size(ex)==0:

gata=1

#ordoneaza planificarile pe masini crescator dupa timpul de sfarsit, adica dupa ultima coloana

for i in range(nrm):

aux = plan\_masina[i]

aux = aux[aux[:, 2].argsort()]

plan\_masina[i] = aux

#costul asociat

cost = max([plan\_masina[i][nrs - 1][2] for i in range(nrm)])

return plan\_masina,planificat,plan, cost

#programarea activitatii plan\_masina[j][k][0]

def programare(j,k,plan\_masina,planificat, plan,nrm,nrs,m\_op,d\_op):

a=int(plan\_masina[j][k][0])

plan[a-1]=0

#calculeaza operatiile succesor si predecesor

pred,succ=pred\_succ(a,nrm)

s=np.size(pred)

#pentru toti predecesorii planificati ai lui a, programeaza a dupa acestia

for i in range(s):

b=pred[i]

if planificat[b-1]:

#masina pe care e planificat b

jp=m\_op[b-1]-1

#a cata linie ii corespunde lui b <->al catelea element din prima coloana a lui plan\_masina[jp]este

c = np.where(np.transpose(plan\_masina[jp])[0] == b)

# al catelea element este b pe jp

kp=c[0][0]

#seteaza primul timp liber pe masina j, pentru planificarea lui a dupa b

# modifica plan[a-1] dupa ce toti pred. planificati au fost executati, daca este nevoie

if plan[a-1]<plan\_masina[jp][kp][2]:

plan[a-1]=plan\_masina[jp][kp][2]

#la acest moment a este programat dupa toti predecesorii deja planificati

# replanifica a dupa plan[a-1] daca planificarea intra in conflict cu cele deja efectuate pe masina j

gata=0

while not gata:

gata=1

for i in range(nrs):

b=int(plan\_masina[j][i][0])

#modifica planificarea lui a pe j daca se ajunge la conflict cu b

if (planificat[b-1]) and (plan\_masina[j][i][2]>plan[a-1]) and (plan[a-1]+d\_op[a-1]>plan\_masina[j][i][1]):

plan[a-1]=plan\_masina[j][i][2]

gata=0

break

#planifica a

plan\_masina[j][k][1]=plan[a-1]

plan\_masina[j][k][2]=plan[a-1]+d\_op[a-1]

planificat[a-1]=1

#pentru toti succesorii lui a

s=np.size(succ)

for i in range(s):

# toti succesorii lui a care la aceste moment se executa inaintea lui a sunt deplanificati

b=succ[i]

if planificat[b - 1]:

#masina pe care este planificat b

js=m\_op[b-1]-1

c = np.where(np.transpose(plan\_masina[js])[0] == b)

#al catelea element este b pe js

ks=c[0][0]

#deplanifica succesorul b, daca timpul de start este inaintea finalului lui a

if plan\_masina[js][ks][1]<plan\_masina[j][k][2]:

planificat[b-1]=0

plan[b-1]=0

return plan\_masina,planificat,plan

#COMPONENTELE GA

#POPULATIA INITIALA

#genereaza populatia initiala

#I: nrm,nrs,m\_op,d\_op - ca mai sus

#E: [pop,val] - populatia initiala si vectorul valorilor

def gen(nrm,nrs,m\_op,d\_op,dim):

#numarul operatiilor n

n=nrm\*nrs

#defineste o variabila ndarray dimx(n+1) cu toate elementele 0

pop=np.zeros((dim,n),dtype=int)

val=np.zeros(dim,dtype=float)

for i in range(dim):

#genereaza candidatul a

a=np.zeros(n)

for j in range(nrm):

px = np.random.permutation(nrs)

a[nrs \* j:nrs \* (j + 1)] = px[:]

pop[i] = a

# evalueaza candidat

plan\_masina, planificat, plan, cost = calcul\_plan(a, nrm, nrs, m\_op, d\_op)

val[i] = 1/cost

return [pop, val]

#CROSSOVER

#crossover pe populatia de parinti pop, de dimensiune dimxn

# I: - l=[pop,valori], ca in functia de generare

# nrm, nrs, m\_op, d\_op - ca mai sus

# pc- probabilitatea de crossover

#E: [po,val] - populatia copiilor, insotita de calitati

# este implementata recombinarea asexuata

def crossover(l,dim,nrm, nrs, m\_op, d\_op,pc):

pop=l[0]

valori=l[1]

n=nrm\*nrs

# initializeaza populatia de copii, po, cu matricea cu elementele 0

po=np.zeros((dim,n),dtype=int)

# initializeaza valorile populatiei de copii, val, cu matricea cu elementele 0

val=np.zeros(dim,dtype=float)

#populatia este parcursa astfel incat sunt selectati aleator cate 2 indivizii - matricea este accesata dupa o permutare a multimii de linii 0,2,...,dim-1

poz=np.random.permutation(dim)

for i in range(0,dim-1,2):

#selecteaza parintii

x = pop[poz[i]]

y = pop[poz[i+1]]

r = np.random.uniform(0,1)

c1 = x.copy()

c2 = y.copy()

if r<=pc:

# crossover x cu y

# selecteaza aleator a cata permutare din cromozom intra in recombinare

xx=np.zeros(nrs)

yy=np.zeros(nrs)

k=np.random.randint(0,nrm)

xx[:]=x[k\*nrs:(k+1)\*nrs]

yy[:] = y[k \* nrs:(k + 1) \* nrs]

#cc1, cc2 = crossover\_CX(xx, yy, nrs)

cc1,cc2 = crossover\_PMX(xx,yy,nrs)

#cc1, cc2 = crossover\_OCX(xx, yy, nrs)

c1[k\*nrs:(k+1)\*nrs]=cc1[:]

c2[k \* nrs:(k + 1) \* nrs] = cc2[:]

plan\_masina, planificat, plan, cost = calcul\_plan(c1, nrm, nrs, m\_op, d\_op)

v1=1/cost

plan\_masina, planificat, plan, cost = calcul\_plan(c2, nrm, nrs, m\_op, d\_op)

v2 = 1 / cost

else:

# recombinare asexuata

v1=valori[poz[i]]

v2=valori[poz[i+1]]

#copiaza rezultatul in populatia urmasilor

po[i] = np.copy(c1)

po[i+1] = np.copy(c2)

val[i]=v1

val[i+1]=v2

return [po, val]

#MUTATIE

# operatia de mutatie a descendentilor obtinuti din recombinare

# I: desc - [po,vo] - populatia copiilor insoltita de vectorul calitatilor

# dim - dimensiunea

# nrm, nrs, m\_op, d\_op - ca mai sus

# pm - probabilitatea de mutatie

# E: descm - [mpo,mvo] - indivizii obtinuti

def mutatie(desc,dim,nrm, nrs, m\_op, d\_op, pm):

po=desc[0]

vo=desc[1]

mpo=po.copy()

mvo=vo.copy()

for i in range(dim):

r=np.random.uniform(0,1)

if r<=pm:

x=mpo[i]

# selecteaza aleator a cata permutare din cromozom intra in recombinare

xx = np.zeros(nrs)

k = np.random.randint(0, nrm)

xx[:] = x[k \* nrs:(k + 1) \* nrs]

yy=m\_perm\_inserare(xx,nrs)

#yy = m\_perm\_interschimbare(xx, nrs)

mpo[i][k \* nrs:(k + 1) \* nrs]=yy[:]

plan\_masina, planificat, plan, cost = calcul\_plan(mpo[i], nrm, nrs, m\_op, d\_op)

mvo[i]=1/cost

return [mpo,mvo]

def arata(sol,v,plan\_masina, plan):

# vizualizare rezultate

# I: x - permutarea care defineste asezarea

# E: -

n=len(sol)

t=len(v)

cost=min(v)

print("Cea mai buna solutie calculată are costul: ",cost)

print("Planificarea este: ",plan)

fig=grafic.figure()

x=[i for i in range(t)]

y=[v[i] for i in range(t)]

grafic.plot(x,y,'ro-')

grafic.ylabel("Costul")

grafic.xlabel("Generația")

grafic.title("Evoluția calității celui mai bun individ din fiecare generație")

fig.show()

##ALGORITMUL GENETIC PENTRU REZOLVAREA JSS

#I: masini\_op,durata\_op: - fisierele cu datele de intrare

# dim - dimensiunea unei populatii

# NMAX - numarul maxim de simulari ale unei evolutii

# pc - probabilitatea de crossover

# pm - probabilitatea de mutatie

#

#E: sol - solutia calculata de GA

# val - 1/maximul functiei fitness - cost\_minim

# plan,plan\_masina - planificarea cea mai buna

def GA(masini\_op,durata\_op,dim,NMAX,pc,pm):

#generarea populatiei la momentul initial

nrm, nrs, m\_op, d\_op = citeste(masini\_op,durata\_op)

list=gen(nrm,nrs,m\_op,d\_op,dim)

pop=list[0]

qual=list[1]

n=nrm\*nrs

#initializari pentru GA

it=0

gata=False

#in istoric\_v pastram cel mai bun cost din populatia curenta, la fiecare moment al evolutiei

istoric\_v=[1/np.max(qual)]

# evolutia - cat timp

# - nu am depasit NMAX si

# - populatia are macar 2 indivizi cu calitati diferite si

while it<NMAX and not gata:

spop,sval=ruleta(pop,qual,dim,n)

lpop\_o=crossover([spop,sval],dim, nrm, nrs, m\_op, d\_op, pc)

lpop\_mo=mutatie(lpop\_o,dim,nrm, nrs, m\_op, d\_op, pm)

newpop,newval=elitism(pop,qual,lpop\_mo[0],lpop\_mo[1],dim)

minim=np.min(newval)

maxim=np.max(newval)

if maxim==minim:

gata=True

else:

it=it+1

istoric\_v.append(1/np.max(newval))

pop=newpop.copy()

qual=newval.copy()

i\_sol=np.where(qual==maxim)

sol=pop[i\_sol[0][0]]

plan\_masina, planificat, plan, cost = calcul\_plan(sol, nrm, nrs, m\_op, d\_op)

val=maxim

arata(sol,istoric\_v,plan\_masina,plan)

return sol,1/val, plan\_masina, plan

# import JSS\_FINAL as t

#sol,cost, plan\_masina, plan=t.GA('masini-operatii.txt','operatii-durata.txt',200,50,0.8,3/200)

#sol,cost, plan\_masina, plan=t.GA('masini-operatii11.txt','operatii-durata11.txt',500,50,0.8,4/500)

#sol,cost, plan\_masina, plan=t.GA('masini-operatii2.txt','operatii-durata2.txt',700,80,0.8,6/700)

#sol,cost, plan\_masina, plan=t.GA('masini-operatii3.txt','operatii-durata3.txt',700,50,0.8,6/700)

#sol,cost, plan\_masina, plan=t.GA('masinioperatii.txt','operatiidurate.txt',1000,70,0.8,80/1000)