Enunț:

Aranjați n câini și m (m>1) pisici în cerc astfel încât între 2 câini (aranjați consecutiv) fie să nu se afle nicio pisică, fie să existe cel puțin două.

* Reprezentare:
* Genotip 🡪 vector binar cu m+n elemente
  + 0 🡪 câine
  + 1🡪 pisică
* Spațiul soluțiilor 🡪 🡪 numărul elementelor lui S
* Problema este modelată fără constrângeri
* Funcție fitness:
  + X genotip
  + Cost(X) = numărul aranjărilor greșite:
    - poziții i 🡪 X(i)=X(i+2)=0, X(i+1)=1, i=0,…,m+n-3
    - eventualele erori în șirul X(n+m-2), X(n+m-1), X(0), X(1)

🡪 X(n+m-2)=X(0)=0, X(n+m-1)=1; X(n+m-1)=X(1)=0, X(0)=1

* + - 🡪 cost(X)+=
  + 🡪 cu valori pozitive
* Modelul de populație 🡪 generațional (problema NP-completă, funcția fitness ușor de evaluat), populații cu dimensiune constantă, Dim
* Mutația:
  + Problemă fără constrângeri 🡪 schema generală de mutație la nivel de populație: fără constrângeri
  + Mutația este la nivel de poziție (genă)
  + reprezentare binară 🡪 bitflip
* Crossover:
  + Problemă fără constrângeri 🡪 schema generală de crossover la nivel de populație: fără constrângeri
  + Crossover la nivel de pereche de indivizi
  + Reprezentare binară 🡪 unipunct/multipunct/uniform
  + Model generaționl 🡪 include recombinare asexuată (nr. copii=Dim)
* Selecția părinților
  + Model generațional 🡪 din dim indivizi în populația curentă selectăm dim părinți
  + Este utilizat mecanismul SUS cu distribuția de probabilitate FPS cu sigma-scalare
* Selecția generației următoare
  + Model generațional 🡪 elitist
* Algoritmul este încheiat
  + după un număr maxim de iterații, sau
  + dacă în populația curentă toți indivizii au aceeași calitate, sau
  + dacă a fost calculat un individ cu fitness maxim (egal cu 1)
* Soluția calculată 🡪 cel mai bun individ din populația finală (datorită utilizării elitismului pentru selecția supraviețuitorilor)
* Reprezentare – varianta 2 🡪 reprezentare prin permutări
* Genotip 🡪 permutare cu m+n elemente
  + 0..m-1 🡪 pisici
  + m...n+m-1🡪 câini
* Spațiul soluțiilor 🡪 🡪 numărul elementelor lui S
* Problema este modelată fără constrângeri
* Cate 3 elemente 🡪
  + gresit: X(i)>=m, X(i+1)<m, X(i+2)>=m + situațiile din margine: X(n+m-2),X(n+m-1),X(0), X(1)
  + Numărăm erorile – contor
  + 🡪 cu valori pozitive
* #codific cainii cu 1 si pisicile cu 0  
  #consider n = 5, m = 3 (5 caini, 3 pisici)  
  # x = [0 0 0 0 0 1 1 1]  
  # fitness: numar (minimizez) secventele 0 1 0 gasite  
  # adica caine - pisica - caine  
    
  import numpy  
    
  def fitness(x):  
   contor = 0  
   for i in range(len(x) - 2):  
   if(x[i] == 0 and x[i+1] == 1 and x[i+2] == 0):  
   contor = contor + 1  
    
   #verificam secventa ultimul element - primul element - al doilea element  
   if (x[len(x) - 1] == 0 and x[0] == 1 and x[1] == 0):  
   contor = contor + 1  
    
   if (x[len(x) - 2] == 0 and x[len(x) - 1] == 1 and x[0] == 0):  
   contor = contor + 1  
   return 1 / (1 + contor) #problema de maxim; individ ideal are contor 0 si fitness 1  
    
  def gen(dim):  
   pop = []  
   for i in range(dim):  
   x = [0, 0, 0, 0, 0, 1, 1, 1]  
   numpy.random.shuffle(x) #creez un nou individ prim amestecarea elem.  
   val = fitness(x) #calculez fitness-ul noului individ creat  
   x = list(x)  
   x = x + [val]  
   pop = pop + [x]  
   return numpy.asarray(pop)  
    
  #varianta rezolvare cu permutari:  
  # valorile 0, 1, 2, 3, 4 - caini  
  # valorile 5, 6, 7 - pisici  
    
  def fitness2(x):  
   contor = 0  
   for i in range(len(x) - 2):  
   if(x[i] < 5 and x[i+1] >= 5 and x[i+2] < 5):  
   contor = contor + 1  
   if(x[len(x) - 1] < 5 and x[0] >= 5 and x[1] < 5): #ultimul cu primele doua  
   contor = contor + 1  
   if(x[len(x) - 2] < 5 and x[len(x) - 1] >= 5 and x[0] < 5): #ultimele doua cu primul  
   contor = contor + 1  
   return 1 / (1 + contor)  
    
  def gen2(dim):  
   pop = []  
   for i in range(dim):  
   x = numpy.random.permutation(8)  
   val = fitness2(x)  
   x = list(x)  
   x = x + [val]  
   pop = pop + [x]  
   return numpy.asarray(pop)

Problema Domino

**Proiectare problema:**

M-am gandit sa rezolv problema privind paritatea pieselor de domino folosind permutari.

**Genele individului** sunt reprezentate prin permutari care ne arata ordinea de verificare a conditiei de paritate a pieselor de domino.

Lungimea invidului( **numarul de gene**) este egala cu lungimea listei de liste. ( Cate piese de domino avem)

*Exemplu:*

ld=[[9,1],[4,4],[3,1],[3,5],[7,8],[2,4]]  
" 0 1 2 3 4 5"

*Secventa de mai sus are o lungime de 6, asta inseamna ca avem 6 piese de domino. Cele 2 valori din fiecare sublista reprezinta valorile de pe piesa de domino.*

**Spațiul soluțiilor** este reprezentat de mulțimea tuturor permutărilor posibile ale punctelor din lista de piese de domino. Adica**, elementele permutarilor va lua valori intre 0 si lungimea listei de piese de domino.**

**Functia fitness:**

Initializam un contor cu valoarea 0 pe care il incrementam de fiecare data cand gasim 2 piese alaturate au aceeasi paritate.

O piesa de domino este formata din 2 valori. Pentru prima piesa, verificam daca a doua valoare inscrisa pe piesa are acceasi paritate cu prima valoare de pe piesa urmatoare. Daca este adevarat, incrementam **contorul pe care il returnam** la finalul functiei fitness.

În cazul de față, **funcția "fitness" este o funcție de maximizare**, deoarece se dorește găsirea unei soluții care să maximizeze numărul de piese de domino care respectă condiția dată. **Cu cât mai mare este valoarea returnată de funcția "fitness", cu atât soluția este considerată mai bună.**

**Populatia:**

Alegem ca dimensiunea poulatiei sa o dam ca parametru in functia de generare. Pentru cazul meu, o sa aleg o dimensiune de 20 de indivizi.

**Selectia parintilor**:

Selectam parinti din populatia actuala prin mecanismul ruleta, pentru ca acordam o probabilitate de selectie mai mare indivizilor cu scor fitness mai bun (fps). Vom forma perechile de parinti luand la rand parintii selectati doi cate doi.

**Recombinare: (Probabilitate 0.7)**

Folosesc operatorul de recombinare prin permutari **OCX.** Acesta presupune alegerea aleatoare a doua puncte p1 si p2. Copiem in primul copil valorile dintre p1 si p2 din primul parinte. Dupa asta, copiem valorile neutilizate in primul copil, din cel de-al doilea parinte incepand de la p2, in ordinea aparitiei lor.

**Mutatie:(Probabilitate: 0.1)**

Alegem **interschimbarea** ca operator de mutatie specific permutarilor. Aceasta consta in alegerea a doua pozitii din permutare ale caror valori se interschimba.

**Selectia indivizilor care raman / sunt transmisi mai departe in urmatoarea generatie:**

Selectia indivizilor din toti cromozomii rezultati se face pe baza mecanismului de **selectie elitism**, pentru a transmite in generatia urmatoare doar cei mai buni indivizi.

Elitismul presupune compararea populatiei de copii cu populatia initiala prin prisma celui mai bun individ din fiecare dintre cele doua populatii.

**Conditii de terminare ale algortimului genetic:**

1. (NMAX) Daca este atins numarul maxim de generatii (Eu am ales 20)
2. Daca maximul din generatia actuala nu se imbunatateste pe parcursul NMAX/4
3. Daca cel mai bun individ din generatia actuala are scorul fitness egal cu cel mai slab individ din generatia actuala (toti indivizii din populatia mea au acelasi scor fitness)
4. Conditie de terminare cand se gaseste un individ cu fitness maxim, adica cu valoarea 5..

O imagine care conține captură de ecran, afișaj, software, Software multimedia

Descriere generată automat

Problema UPU

* Într-un oraș se dorește plasare unei unități medicale de urgență care să asigure un serviciu optim din punctul de vedere al timpului de răspuns. În acest scop este realizată o hartă a frecvențelor apelurilor efectuate din diverse puncte ale orașului.
* Harta corespunde unei matrice , conform împărțirii suprafeței într-o rețea de zone.
* Pentru orice punct de apel din rețea, timpul de răspuns din poziția unde este plasată unitatea de urgență este estimat la

**Proiectare problema:**

Rezolvam problema folosind formula data in enunt, referitoare la cost.

**Genele individului** sunt reprezentate de un vector de 2 elemente, numere intregi, care va lua valori in intervalul 0 si lungimea matricei de frecvente. ( In cazul nostru, intre 0 si 9, deoarece matricea are 10 elemente ). Aceste doua elemente ale vectorului reprezinta pozitia (linie, coloana) a orasului din matricea de frecvente.

Lungimea individului(**numarul de gene**) este egala cu 2, adica i si j intr-o matrice(i,j).

**Spatiul solutiilor** cromozomului este reprezentat de valori, **numere intregi, cuprinse intre 0 si n,** unde *n este dimensiunea matricei de frecvente.*

**Functia fitness:**

Ne folosim de functia data in enunt, unde vom calcula costul plasarii unei unitati UPU in fiecare zona de coordonate (i,j) salvate intr-un vector pe care il dam ca parametru functiei de fitness.

**Functia noastra va returna costul, aceasta fiind o functie de minimizare**.

**Populatia:**

Alegem ca dimensiunea poulatiei sa o dam ca parametru in functia de generare. Pentru cazul meu, o sa aleg o dimensiune de 20 de indivizi.

**Selectia parintilor**:

Selectam parinti din populatia actuala prin mecanismul ruleta, pentru ca acordam o probabilitate de selectie mai mare indivizilor cu scor fitness mai bun (fps). Vom forma perechile de parinti luand la rand parintii selectati doi cate doi.

**Recombinare: (Probabilitate: 0.7)**

La nivel de pereche de indivizi, este aplicată recombinarea uniformă, identică în acest caz cu o recombinare unipunct dupa prima poziție. (probabilitate de recombinare=1)

**Mutatie:(Probabilitate: 0.1)**

La nivel de genă, este aplicată resetarea aleatoare pentru că problema este cu date cardinale. **Resetarea aleatoare** - cu probabilitatea 16%, valoarea fiecărei gene este modificată prin generarea aleatoare a unei valori din mulţimea valorilor admisibile pentru gena respectivă.

**Selectia indivizilor care raman / sunt transmisi mai departe in urmatoarea generatie:**

Selectia indivizilor din toti cromozomii rezultati se face pe baza mecanismului de **selectie elitism**, pentru a transmite in generatia urmatoare doar cei mai buni indivizi.

Elitismul presupune compararea populatiei de copii cu populatia initiala prin prisma celui mai bun individ din fiecare dintre cele doua populatii.

**Conditii de terminare ale algortimului genetic:**

1. (NMAX) Daca este atins numarul maxim de generatii (Eu am ales 20)
2. Daca maximul din generatia actuala nu se imbunatateste pe parcursul NMAX/4
3. Daca cel mai bun individ din generatia actuala are scorul fitness egal cu cel mai slab individ din generatia actuala (toti indivizii din populatia mea au acelasi scor fitness)
4. Cand se gaseste individul cu fitness maxim

* Soluția calculată 🡪 cel mai bun individ(costul cel mai mic) din populația finală

import numpy as np  
  
  
def fitness(x, fr, n):  
 val = 0  
 for i in range(n):  
 for j in range(n):  
 val += fr[i, j] \* (1.7 + 3.4 \* np.sqrt((i - x[0]) \*\* 2 + (j - x[1]) \*\* 2))  
 return 1 / val # transformare in problema de maxim, normal era <return val>  
  
  
def gen\_pop(ff, dim):  
 fr = np.genfromtxt(ff)  
 n = len(fr)  
 pop = []  
 for i in range(dim):  
 x = np.random.randint(0, n, 2)  
 val = fitness(x, fr, n)  
 x = list(x)  
 x += [val]  
 pop += [x]  
 return pop

Problema Orașe Grecia

. Cele n orașe stat din Grecia antică se luptă între ele pentru dominație, dar în fața unei amenințări externe hotărăsc să se unească. Pentru a stabili planul de apărare, delegații orașelor urmează să se întîlnească, fiecare oraș desemnîndu-și un singur reprezentant. Cunoscînd rivalitățile istorice dintre orașe, folosiți un algoritm genetic pentru a găsi o modalitate de așezare a delegaților la masa (rotundă) tratativelor astfel încît delegații din orașe rivale să nu fie vecini (se presupune că acest lucru este posibil). Harta orașelor stat între care există animozități este exprimată printr-o matricea pătratică de ordin n, numită CONFLICT: 𝐶𝑂𝑁𝐹𝐿𝐼𝐶𝑇(𝑖,𝑗) = { 0, 𝑑𝑎𝑐ă 𝑖 ș𝑖 𝑗 𝑛𝑢 𝑠𝑒 𝑎𝑓𝑙ă î𝑛 𝑠𝑡𝑎𝑟𝑒 𝑑𝑒 𝑐𝑜𝑛𝑓𝑙𝑖𝑐𝑡 𝑠𝑎𝑢 𝑖 = 𝑗 1, 𝑑𝑎𝑐ă 𝑜𝑟𝑎ș𝑢𝑙 𝑠𝑡𝑎𝑡 𝑖 𝑒𝑠𝑡𝑒 î𝑛 𝑐𝑜𝑛𝑓𝑙𝑖𝑐𝑡 𝑐𝑢 𝑜𝑟𝑎ș𝑢𝑙 𝑠𝑡𝑎𝑡 �

Proiectare problemă:

Orașele din Grecia le voi gândi ca fiind denumite drept linia corespunzătoare a matricei CONFLICTE. Un individ este reprezentant de **ordinea așezării** la masă a delegaților orașelor respective. De exemplu, pentru 10 orașe, o posibilă ordine de plasare a delegaților la masă ar fi [0, 5, 2, 3, 6, 9, 8, 7, 4, 1].

**Spațiul soluțiilor** este reprezentat de mulțimea permutărilor de ordin **len(CONFLICTE)**.

**Funcția fitness:** Funcția fitness va fi o funcție de minimizare și va returna 1/(1+nr\_orașe\_aflate\_în\_conflict). Așadar, știm că valoare optimă pentru această funcție este 1. **nr\_orașe\_aflate\_în\_conflict** este un contor ce se inițializează cu 0 la intrarea în funcție și se incrementează de oricâte ori delegații orașelor reprezentați în genele individuluiaflă în conflict. Adică, simplu spus, de fiecare dată când CONFLICTE[individ[i]][individ[j]] == 1, i=[0, len(CONFLICTE)-1]; j = [i+1, len(CONFLICTE)].

Nu există restricții pentru această problemă, orice individ din spațiul soluțiilor fiind fezabil.

**Populația inițială: DIM** permutări de ordin **len(CONFLICTE)** + calitatea fiecărui individ pe ultima poziție a listei.

**Selecția părinților:** Selectăm populația actuală prin mecanismul ruletă pentru a asigura o șansă mai mare indivizilor cu un fitness mai bun. Formăm **DIM/2** perechi de părinți ce vor da naștere, fiecare, la câte doi copii.

**Recombinare:** Pentru recombinare voi folosi operatorul **OCX** cu o probabilitate de 0,7.

**Mutație**: Pentru mutație alegem interschimbarea a două gene dintr-un individ cu o probabilitate de 0,1. Această interschimbare ne asigură că rămânem tot în spațiul soluțiilor.

**Selecția supraviețuitorilor:** Această selecție se face pe baza **elitismului**. Dacă în populația de copii\_mutanți nu există niciun individ cu un fitness mai bun decât populația de părinți, se elimină aleator unul dintre copiii mutanți și se înlocuiește cu cel mai bun individ din populația de părinți.

**Condiții de terminare ale GA:**

1. Dacă este atins numărul maxim de iterații
2. Dacă fitness-ul maxim al populației nu se îmbunătățește pe parcursul a (numărului maxim de iterații)/4
3. Dacă cel mai bun individ din generația actuală are scorul fitness egal cu cel mai slab individ din generația actuală (toți indivizii au același scor fitness)
4. *"""CERINTA: Cele n orașe stat din Grecia antică se luptă între ele pentru dominație, dar în fața unei amenințări externe  
   hotărăsc să se unească. Pentru a stabili planul de apărare, delegații orașelor urmează să se întîlnească, fiecare  
   oraș desemnîndu-și un singur reprezentant. Cunoscînd rivalitățile istorice dintre orașe, folosiți un algoritm  
   genetic pentru a găsi o modalitate de așezare a delegaților la masa (rotundă) tratativelor astfel încît delegații  
   din orașe rivale să nu fie vecini (se presupune că acest lucru este posibil).  
   Harta orașelor stat între care există animozități este exprimată printr-o matricea pătratică de ordin n, numită  
   CONFLICT:  
   𝐶𝑂𝑁𝐹𝐿𝐼𝐶𝑇(𝑖, 𝑗) = {  
   0, 𝑑𝑎𝑐ă 𝑖 ș𝑖 𝑗 𝑛𝑢 𝑠𝑒 𝑎𝑓𝑙ă î𝑛 𝑠𝑡𝑎𝑟𝑒 𝑑𝑒 𝑐𝑜𝑛𝑓𝑙𝑖𝑐𝑡 𝑠𝑎𝑢 𝑖 = 𝑗  
   1, 𝑑𝑎𝑐ă 𝑜𝑟𝑎ș𝑢𝑙 𝑠𝑡𝑎𝑡 𝑖 𝑒𝑠𝑡𝑒 î𝑛 𝑐𝑜𝑛𝑓𝑙𝑖𝑐𝑡 𝑐𝑢 𝑜𝑟𝑎ș𝑢𝑙 𝑠𝑡𝑎𝑡 𝑗"""*import numpy as np  
     
   CONFLICTE = np.genfromtxt('CONFLICT.txt', dtype=int)  
     
     
   # Individul este o permutare a numerelor de la 0 la len(CONFLICTE)-1 inclusiv (reprezentand delegatii oraselor)  
   # Functia fitness va fi mai mare cu cat numarul de conflicte dintre delegati este mai mic  
   # ex individ = [0, 5, 2, 3, 6, 9, 8, 7, 4, 1]  
   def fitness(individ):  
    *"""Functia de fitness"""* fit = 0  
    for i in range(len(individ) - 1):  
    for j in range(i + 1, len(individ)):  
    if CONFLICTE[individ[i]][individ[j]] == 1:  
    fit += 1  
    if CONFLICTE[individ[0]][individ[-1]] == 1:  
    fit += 1  
    return 1 / (fit + 1)  
     
     
   def gen\_pop(dim):  
    *"""Generarea unei populatii de dimensiune dim"""* pop = []  
    for i in range(dim):  
    individ = np.random.permutation(len(CONFLICTE))  
    pop.append(list(individ) + [fitness(individ)])  
    return np.asarray(pop)  
     
     
   if \_\_name\_\_ == '\_\_main\_\_':  
    NR\_ITERATII = 500  
    NR\_INDIVIZI = 100  
    i = 1  
    gata = False  
    pop = gen\_pop(NR\_INDIVIZI)  
    max\_qual = max(pop[:, -1])  
    nrm = 1  
    while i<=500 and not gata:  
    # selectia parintilor prin metoda ruletei  
    spop = ruleta(pop, NR\_INDIVIZI, len(CONFLICTE))  
    # recombinarea parintilor  
    copii = OCX(spop, NR\_INDIVIZI, len(CONFLICTE), prob\_rec=0.7)  
    # mutatia copiilor  
    copii\_m = interschimbare(copii, len(CONFLICTE), prob\_mut=0.1)  
    # selectia supravietuitorilor  
    pop = elitism(pop, copii\_m, NR\_INDIVIZI).copy()  
    minim = min(pop[:, -1])  
    maxim = max(pop[:, -1])  
    if maxim == max\_qual:  
    nrm += 1  
    else:  
    max\_qual = maxim  
    nrm = 1  
    if maxim == minim or nrm == int(NR\_ITERATII/10):  
    gata = True  
    i += 1  
    max\_qual = maxim  
    print("Solutia optima este: ", pop[np.argmax(pop[:, -1])])

PERIMETRU MAX TRIUNGHI

import numpy as np  
import math  
  
  
# inputul este o matrice cu n linii si 2 coloane (coloana 1 - coordonata lui x, coloana 2- coordonata lui y)  
  
# f. obiectiv  
def PerMax(x, n):  
 # functia obiectiv pentru problema perimetrului maxim  
  
 # I: x - individul (permutarea) evaluat(a), n-dimensiunea problemei  
 # E: per- calitate  
 perimetru = 0  
 latura1 = 0  
 latura2 = 0  
 latura3 = 0  
 for i in range(n + 1):  
 latura1 = math.sqrt(math.pow(x[i][0] - x[i + 1][0], 2) + math.pow(x[i][1] - x[i + 1][1], 2))  
 latura2 = math.sqrt(math.pow(x[i + 1][0] - x[i + 2][0], 2) + math.pow(x[i + 1][1] - x[i + 2][1], 2))  
 latura3 = math.sqrt(math.pow(x[i][0] - x[i + 2][0], 2) + math.pow(x[i][1] - x[i + 2][1], 2))  
 perimetru = latura1 + latura2 + latura3  
 return perimetru  
# genereaza populatia initiala  
# I:  
# n - dimensiunea prolemei  
# dim - numarul de indivizi din populatie  
# E: pop - populatia initiala  
def gen(n, dim):  
 n = 6  
 # defineste o variabila ndarray cu toate elementelo nule  
 pop = np.zeros((dim, n + 1), dtype=int)  
 for i in range(dim):  
 # genereaza candidatul permutare cu n elemente  
 pop[i, :n] = np.random.permutation(n)  
 pop[i, n] = PerMax(pop[i, :n], n)  
 return pop

PROBLEMA 32 ECHIPE

from numpy import random  
  
v = [[random.randint(10) + 1 for i in range(33)] for j in range(3)]  
for i in range(1, 33):  
 v[1][i] = i  
print("Echipe si valori generate:")  
for i in range(1, 3):  
 for j in range(1, 33):  
 print(v[i][j], end=' ')  
 print()  
  
  
def fitness(v):  
 f = 0  
 i = 0  
 while i < 30:  
 f = f + abs(v[2][1 + i] - v[2][2 + i])  
 f = f + abs(v[2][1 + i] - v[2][3 + i])  
 f = f + abs(v[2][1 + i] - v[2][4 + i])  
 f = f + abs(v[2][2 + i] - v[2][3 + i])  
 f = f + abs(v[2][2 + i] - v[2][4 + i])  
 f = f + abs(v[2][3 + i] - v[2][4 + i])  
 i = i + 4  
 return f  
  
  
v1 = [[0 for i in range(33)] for j in range(3)]  
v2 = [[0 for i in range(33)] for j in range(3)]  
  
for i in range(1, 33):  
 v1[1][i] = v[1][i]  
 v1[2][i] = v[2][i]  
 v2[1][i] = v[1][i]  
 v2[2][i] = v[2][i]  
  
contor = 0  
fitnessanterior = 999999  
while contor < 1000:  
  
 if fitness(v1) < fitness(v2):  
 for i in range(1, 3):  
 for j in range(1, 33):  
 v2[i][j] = v1[i][j]  
 if fitness(v1) == fitnessanterior:  
 contor = contor + 1  
 else:  
 contor = 0  
 fitnessanterior = fitness(v1)  
 else:  
 for i in range(1, 3):  
 for j in range(1, 33):  
 v1[i][j] = v2[i][j]  
 if fitness(v2) == fitnessanterior:  
 contor = contor + 1  
 else:  
 contor = 0  
 fitnessanterior = fitness(v2)  
  
 if random.randint(2) == 0:  
 fvechi = fitness(v1)  
 p1 = random.randint(32) + 1  
 p2 = random.randint(32) + 1  
 aux = v1[1][p1]  
 v1[1][p1] = v1[1][p2]  
 v1[1][p2] = aux  
 aux = v1[2][p1]  
 v1[2][p1] = v1[2][p2]  
 v1[2][p2] = aux  
 if fvechi < fitness(v1):  
 aux = v1[1][p1]  
 v1[1][p1] = v1[1][p2]  
 v1[1][p2] = aux  
 aux = v1[2][p1]  
 v1[2][p1] = v1[2][p2]  
 v1[2][p2] = aux  
 if random.randint(2) == 0:  
 fvechi = fitness(v2)  
 p1 = random.randint(32) + 1  
 p2 = random.randint(32) + 1  
 aux = v2[1][p1]  
 v2[1][p1] = v2[1][p2]  
 v2[1][p2] = aux  
 aux = v2[2][p1]  
 v2[2][p1] = v2[2][p2]  
 v2[2][p2] = aux  
 if fvechi < fitness(v2):  
 aux = v2[1][p1]  
 v2[1][p1] = v2[1][p2]  
 v2[1][p2] = aux  
 aux = v2[2][p1]  
 v2[2][p1] = v2[2][p2]  
 v2[2][p2] = aux  
  
print("Grupele de echipe sunt (o grupa pe linie): ")  
for i in range(1, 33):  
 print(v1[1][i], ' ', end='')  
 if i % 4 == 0:  
 print()

**SAU ECHIPE**

import random as rp

import numpy as np

import math as mt

def echipe():

x=[i+1 for i in range(32)]

val=[rp.randint(1,10) for j in range(32)]

return x,val

def ok(n,dim,val2):

sum=np.zeros(8,dtype=int)

j=0

i=0

while i < n-3:

sum[j]=val2[i]+val2[i+1]+val2[i+2]+val2[i+3]

j=j+1

i=i+4

dif=np.abs((sum[0]-sum[1])+(sum[1]-sum[2])+(sum[2]-sum[3])+(sum[3]-sum[4])+

(sum[4]-sum[5])+(sum[5]-sum[6])+(sum[6]-sum[7]))

return -dif

# am generat echipele si valorile pentru fiecare echipa

def gen(x,val,n,dim):

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

#genereaza candidatul permutare cu n elemente

for i in range(dim):

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

for j in range(n):

pop[i,j]=pop[i,j]+1

val2=np.zeros((n),dtype=int)

for k in range(32):

val2[k]=val[pop[i,k]-1]

pop[i,n]=ok(n,dim,val2)

return pop,val2,x

Problema Avioane

2. O companie aeriană dorește să achiziționeze 3 tipuri de aeronave, avînd un buget de achiziții de 5.000 de unități. Fiecare tip de avion, notat de la a. la c. are următoarele caracteristici: a. costă 100 de unități, autonomie 6.000 km și raza de detecție TCAS 30 km; b. costă 60 de unități, autonomie 4.200 km și raza de detecție TCAS 48 km; c. costă 50 de unități, autonomie 2.800 km și raza de detecție TCAS 32 km. Calculați cîte aeronave din fiecare tip trebuie cumpărate astfel încît - să nu fie depășită suma disponibilă; - autonomia medie să fie maximă; - valoarea medie razei de detecție TCAS să fie cel puțin 40 km. Observație. Dacă numărul avioanelor din fiecare tip este a, b, respectiv c, atunci autonomia medie este 6000∙𝑎+4200∙𝑏+2800∙𝑐 𝑎+𝑏+𝑐 și valoarea medie a razei de detecție TCAS este 30∙𝑎+48∙𝑏+32∙𝑐 𝑎+𝑏+𝑐 .

Proiectare problemă:

**Reprezentare:** Un individ va fi reprezentat de 3 gene, reprezentând numărul de avioane achiziționate din fiecare categorie.

**Spațiul soluțiilor** este reprezentat de intervalul maxim de avioane achiziționabile din fiecare categorie. Adică din prima categorie putem achiziționa maxim 5.000/100=50 avioane, din a doua maxim 5.000/60 = 83,33 = 83 avioane, din a treia 5.000/50 = 100 avioane. Deci spațiul soluțiilor va fi [0,50] x [0,83] x [0,100]. Constrângerile de fezabilitate presupun respectarea condițiilor din enunț, și anume: să nu fie depășită suma disponibilă; valoarea medie razei de detecție TCAS să fie cel puțin 40 km.

**Populația inițială** va fi constituită dintr-o matrice de **DIM** indivizi cu 4 coloane reprezentând cele 3 gene menționate anterior și calitatea fiecărui individ. Fiecare genă a individului va fi aleasă aleator din spațiul soluțiilor respectând constrângerile amintite anterior.

**Funcția fitness** presupune evaluarea autonomiei pe baza formulei .

**Selecția părinților:** Selectăm populația actuală prin mecanismul de ruletă standard cu FPS cu sigma scalare.

**Recombinare:** Schema generală de recombinare pentru probleme cu constrângeri, reprezentare cu numere. Ca mecanism, voi folosi recombinarea uniformă cu probabilitate **0,8**.

**Mutație:** Schema generală de mutație pentru probleme cu constrângeri, reprezentare cu numere. Ca mecanism, voi folosi operatorul de mutație cu fluaj cu probabilitate **0,1** și limita de fluaj 2.

**Selecția generației următoare:** Voi aplica regula ca fiecare individ să aibă o **limită de vârstă 1**, iar ca mecanism voi folosi **elitismul**.

Drept **condiții de terminare a algoritmului genetic**:

1. Numărul maxim de iterații este atins
2. Dacă fitness-ul maxim al populației nu se îmbunătățește pentru un număr de iterații.
3. import numpy as np  
     
   COSTURI = [100, 60, 50]  
   AUTONOMII = [6000, 4200, 2800]  
   RAZE = [30, 48, 32]  
     
   def fitness(x):  
    return (x[0]\*AUTONOMII[0]+x[1]\*AUTONOMII[1]+x[2]\*AUTONOMII[2])/np.sum(x)  
     
   def gen\_pop(dim):  
    pop = []  
    for i in range(dim):  
    individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
    gata = False  
    while not gata:  
    if COSTURI[0] \* individ[0] + COSTURI[1] \* individ[1] + COSTURI[2] \* individ[2] > 5000:  
    individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
    continue  
    if (RAZE[0] \* individ[0] + RAZE[1] \* individ[1] + RAZE[2] \* individ[2]) < 40:  
    individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
    continue  
    gata = True  
    individ += [fitness(individ)]  
    pop.append(individ)  
    return pop  
     
   if \_\_name\_\_ == '\_\_main\_\_':  
    print('Populatia initiala:')  
    NMAX = 400  
    NR\_INDIVIZI = 100  
    PC = 0.8  
    PM = 0.1  
    pop = gen\_pop(NR\_INDIVIZI)  
    for line in sorted(pop, key=lambda x: x[-1]):  
    print(line)  
    i = 1  
    gata = False  
    max\_qual = max(pop[:, -1])  
    nrm = 1  
    while i<=NMAX and not gata:  
    spop = ruleta\_fps\_sigma\_scalare(pop, NR\_INDIVIZI)  
    copii = incrucisare(spop, NR\_INDIVIZI, PC)  
    copii\_m = mutatie(copii, PM)  
    pop = elitism(pop, copii\_m, NR\_INDIVIZI)  
    max\_qual\_nou = max(pop[:, -1])  
    if max\_qual\_nou == max\_qual:  
    nrm += 1  
    else:  
    nrm = 1  
    max\_qual = max\_qual\_nou  
    if nrm == NMAX//4:  
    gata = True  
    i += 1  
    print('Populatia finala:')  
    for line in sorted(pop, key=lambda x: x[-1]):  
    print(line)

import numpy  
import matplotlib.pyplot as grafic  
from FunctiiCrossoverIndivizi import crossover\_uniform  
from FunctiiMutatieIndivizi import m\_fluaj  
from FunctiiSelectii import elitism, ruleta  
  
# functia obiectiv  
def fAvioane(x, autonomie):  
 val = numpy.dot(x, autonomie)  
 nr\_av = numpy.sum(x)  
 if (nr\_av):  
 val = val / nr\_av  
 return val  
  
  
# verificarea admisibilitatii unei solutii  
def admisibil(x, vizibilitate, cost, costmax, vmin):  
 vmed = numpy.dot(x, vizibilitate)  
 cost = numpy.dot(x, cost)  
 nr\_av = sum(x)  
 if (nr\_av):  
 vmed = vmed / nr\_av  
 return cost <= costmax and vmed > vmin  
  
  
# genereaza populatia initiala  
# I:  
# dim - numarul de indivizi din populatie  
# c,v,a, s - fisierele din care citim datele  
# costmax - costul maxim  
# vmin - vizibilitatea minima  
# E: pop, qual - populatia initiala si vectorul calitatilor  
def gen(c, v, a, dim, costmax, vmin):  
 # citeste datele din fisiere  
 cost = numpy.genfromtxt(c)  
 vizibilitate = numpy.genfromtxt(v)  
 autonomie = numpy.genfromtxt(a)  
 n = len(cost)  
 # numarul maxim de avioane achizitionate din fiecare tip  
 variante = [int(costmax / cost[i]) for i in range(n)]  
 pop = numpy.zeros([dim, n], dtype='int')  
 qual = numpy.zeros(dim)  
 i = 0  
 while i < dim:  
 x = numpy.zeros(n)  
 for k in range(n):  
 x[k] = numpy.random.randint(0, variante[k] + 1)  
 if admisibil(x, vizibilitate, cost, costmax, vmin):  
 pop[i] = x.copy()  
 qual[i] = fAvioane(pop[i], autonomie)  
 i += 1  
 return pop,qual,n,cost,vizibilitate,autonomie,variante  
  
  
#crossover pe populatia de parinti pop, cu vectorul de calitati qual  
# I: pop,qual,dim,n - ca mai sus  
# cost,vizibilitate,autonomie,variante,costmax,vmin: datele problemei  
# pc- probabilitatea de crossover  
#E: po,qo - populatia copiilor, calitati qo  
# este implementata recombinarea asexuata  
def crossover\_populatie(pop,qual,dim,n,cost,vizibilitate,autonomie,variante,costmax,vmin,pc):  
 po=pop.copy()  
 qo=qual.copy()  
 #populatia este parcursa astfel incat sunt selectati indivizii dupa un amestec  
 poz=numpy.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 = numpy.random.uniform(0,1)  
 if r<=pc:  
 # crossover x cu y - uniform  
 c1,c2=crossover\_uniform(x, y, n)  
 if admisibil(c1,vizibilitate,cost,costmax,vmin):  
 po[i1]=c1.copy()  
 qo[i1]=fAvioane(c1,autonomie)  
 if admisibil(c2,vizibilitate,cost,costmax,vmin):  
 po[i2]=c2.copy()  
 qo[i2]=fAvioane(c2,autonomie)  
 return po,qo  
  
  
  
#mutatie asupra populatiei de copii  
# I:pop,qual,dim,n - populatia, calitatile  
# cost,vizibilitate,autonomie,variante,costmax,vmin: datele problemei  
# pm - probabilitatea de mutatie  
#E: - mpo,mqo - populatia mutata, vectorul calitatilor  
def mutatie\_populatie(pop,qual,dim,n,cost,vizibilitate,autonomie,variante,costmax,vmin,pm):  
 mpo=pop.copy()  
 mqo=qual.copy()  
 x=numpy.zeros(n,dtype='int')  
 for i in range(dim):  
 x[:]=pop[i]  
 for j in range(n):  
 #genereaza aleator daca se face mutatie in individul i gena j  
 r=numpy.random.uniform(0,1)  
 if r<=pm:  
 #mutatie fluaj  
 x[j]=m\_fluaj(x[j],0,variante[j])  
 if admisibil(x,vizibilitate,cost,costmax,vmin):  
 mpo[i]=x.copy()  
 mqo[i]=fAvioane(x,autonomie)  
 return mpo,mqo  
  
  
def arata(sol,v):  
 # vizualizare rezultate  
 n=len(sol)  
 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()  
  
  
  
##ALGORITMUL GENETIC PENTRU REZOLVAREA PROBLEMEI  
#I: c,v,a, s - fisierele din care citim datele  
# costmax,vmin - datele problemei  
# 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 - maximul functiei fitness  
def GA(c,v,a,costmax,vmin,dim,NMAX,pc,pm):  
 #generarea populatiei la momentul initial  
 pop, qual, n, cost, vizibilitate, autonomie, variante = gen(c,v,a,dim, costmax, vmin)  
 #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=[numpy.max(qual)]  
 # evolutia - cat timp  
 # - nu am depasit NMAX si  
 # - populatia are macar 2 indivizi cu calitati diferite si  
 # - in ultimele NMAX/2 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, n)  
 po,qo=crossover\_populatie(spop,sval,dim,n,cost,vizibilitate,autonomie,variante,costmax,vmin,pc)  
 mpo,mqo = mutatie\_populatie(po,qo,dim,n,cost,vizibilitate,autonomie,variante,costmax,vmin,pm)  
 newpop, newval = elitism(pop, qual, mpo, mqo, dim)  
 minim=numpy.min(newval)  
 maxim=numpy.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(numpy.max(newval))  
 pop =newpop.copy()  
 qual =newval.copy()  
 i\_sol=numpy.argmax(qual)  
 sol=pop[i\_sol]  
 val=maxim  
 arata(sol,istoric\_v)  
 return sol,val  
  
#import Avioane\_test  
#sol,val=Avioane\_test.GA('cost.txt','vizibilitate.txt','autonomie.txt',5000,2000,120,100,0.8,0.2)

Problema Cursuri

. La începutul anului universitar un student trebuie să aleagă cursurile pe care dorește să le urmeze. Bugetul disponibil este de 10.000 lei și pot fi alese trei tipuri de cursuri: a. Curs cu costul 1000 lei, acordă 5 puncte de credit și necesită 80 ore de studiu individual. b. Curs cu costul 800 lei, acordă 3 puncte de credit și necesită 40 ore de studiu individual. c. Curs cu costul 1500 lei, acordă 8 puncte de credit și necesită 100 ore de studiu individual. Folosiți un algoritm genetic pentru a ajuta studentul să aleagă o combinație de cursuri (cîte din fiecare tip) astfel încît să nu depășească bugetul disponibil, numărul mediu de credite să fie cît mai mare, timpul mediu de studiu individual să nu depășească 70 de ore. Notă: numărul mediu de credite și numărul mediu de ore de studiu individual se calculează ca medie ponderată.

Proiectare problemă:

**Reprezentare:** Un individ va fi reprezentat de 3 gene, reprezentând numărul de cursuri achiziționate din fiecare categorie.

**Spațiul soluțiilor** este reprezentat de intervalul maxim de cursuri achiziționabile din fiecare categorie. Adică din prima categorie putem achiziționa maxim 10.000/1.000=10 cursuri, din a doua maxim 10.000/800 = 12,5 = 12 cursuri, din a treia 10.000/1.500 = 6,66 = 6 cursuri. Deci spațiul soluțiilor va fi [0,10] x [0,12] x [0,6]. Constrângerile de fezabilitate presupun respectarea condițiilor din enunț, și anume: să nu fie depășit bugetul disponibil; timpul mediu de studiu individual să nu depășească 70 de ore.

**Populația inițială** va fi constituită dintr-o matrice de **DIM** indivizi cu 4 coloane reprezentând cele 3 gene menționate anterior și calitatea fiecărui individ. Fiecare genă a individului va fi aleasă aleator din spațiul soluțiilor respectând constrângerile amintite anterior.

**Funcția fitness** presupune evaluarea numărului de credite mediu pe baza formulei .

**Selecția părinților:** Selectăm populația actuală prin mecanismul de ruletă standard cu FPS cu sigma scalare.

**Recombinare:** Schema generală de recombinare pentru probleme cu constrângeri, reprezentare cu numere. Ca mecanism, voi folosi recombinarea uniformă cu probabilitate **0,8**.

**Mutație:** Schema generală de mutație pentru probleme cu constrângeri, reprezentare cu numere. Ca mecanism, voi folosi operatorul de mutație cu fluaj cu probabilitate **0,1** și limita de fluaj 2.

**Selecția generației următoare:** Voi aplica regula ca fiecare individ să aibă o **limită de vârstă 1**, iar ca mecanism voi folosi **elitismul**.

Drept **condiții de terminare a algoritmului genetic**:

1. Numărul maxim de iterații este atins
2. Dacă fitness-ul maxim al populației nu se îmbunătățește pentru un număr de iterații.

import numpy as np  
  
COSTURI = [1000, 800, 1500]  
CREDITE = [5, 3, 8]  
ORE\_STUDIU = [80,40,100]  
  
def fitness(x):  
 return (x[0]\*CREDITE[0]+x[1]\*CREDITE[1]+x[2]\*CREDITE[2])/np.sum(x)  
  
def gen\_pop(dim):  
 pop = []  
 for i in range(dim):  
 individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
 gata = False  
 while not gata:  
 if COSTURI[0] \* individ[0] + COSTURI[1] \* individ[1] + COSTURI[2] \* individ[2] > 5000:  
 individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
 continue  
 if (ORE\_STUDIU[0] \* individ[0] + ORE\_STUDIU[1] \* individ[1] + ORE\_STUDIU[2] \* individ[2]) < 40:  
 individ = [np.random.randint(low=0, high=51), np.random.randint(low=0, high=84), np.random.randint(low=0, high=100)]  
 continue  
 gata = True  
 individ += [fitness(individ)]  
 pop.append(individ)  
 return pop  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('Populatia initiala:')  
 NMAX = 400  
 NR\_INDIVIZI = 100  
 PC = 0.8  
 PM = 0.1  
 pop = gen\_pop(NR\_INDIVIZI)  
 for line in sorted(pop, key=lambda x: x[-1]):  
 print(line)  
 i = 1  
 gata = False  
 max\_qual = max(pop[:, -1])  
 nrm = 1  
 while i<=NMAX and not gata:  
 spop = ruleta\_fps\_sigma\_scalare(pop, NR\_INDIVIZI)  
 copii = incrucisare(spop, NR\_INDIVIZI, PC)  
 copii\_m = mutatie(copii, PM)  
 pop = elitism(pop, copii\_m, NR\_INDIVIZI)  
 max\_qual\_nou = max(pop[:, -1])  
 if max\_qual\_nou == max\_qual:  
 nrm += 1  
 else:  
 nrm = 1  
 max\_qual = max\_qual\_nou  
 if nrm == NMAX//4:  
 gata = True  
 i += 1  
 print('Populatia finala:')  
 for line in sorted(pop, key=lambda x: x[-1]):  
 print(line)

4. Un produs P este obţinut în două fabrici situate în două orașe (București și Craiova) și este stocat pentru desfacere în trei depozite, unul situat în Ploiești, unul în Pitești și unul la Cluj. Fabrica din București poate produce săptămînal 120 de tone din produsul P, iar fabrica din Craiova poate produce P în cantitate de 140 tone pe săptămînă. Pentru desfacerea produsului, necesarul săptămînal este: pentru depozitul din Ploiești 100 de tone, pentru depozitul din Pitești, 60 de tone, respectiv pentru depozitul din Cluj 80 de tone. În tabelul de mai jos sunt prezentate costurile de transport pe tona de produs. Ploiești Pitești Cluj București 50 70 90 Craiova 60 70 100 Problema de rezolvat: calculul numărului de tone din produsul P care trebuie furnizate de cele două fabrici fiecărui depozit astfel încît costul de transport să fie minim și astfel încît să fie respectate condițiile enunțate mai sus

Problema Transport

Proiectare problemă:

**Reprezentare:** Un individ va fi reprezentat de o permutare de ordin **m x n** reprezentând ordinea de alocare a resurselor către depozite. Astfel, poziția **k** din ordinea de alocare va semnifica alocarea elementului xi,j (), unde pk = (i-1)\*n+j. Pe baza acestei permutări se va alcătui un plan de transport astfel: se inițializează oferta totală (OR) cu suma tuturor ofertelor celor două depozite, vectorul de oferte, vectorul de cerere, k=1 și matricea alocării, x. Cât timp OR este strict pozitiv, se prelucrează elementul curent din permutare, pk astfel: , se alocă elementul corespunzător, x[lin][col] = minim(o\_r[lin], c\_r[col]), se actualizează o\_r și c\_r scăzându-se cantitatea aleasă din fiecare (ori o\_r[lin] ori c\_r[col] va fi 0), se actualizează oferta totală rămasă scăzându-se din aceasta valoarea aleasă anterior și, în final, se incrementează k-ul.

**Spațiul soluțiilor** este reprezentat orice permutare de ordin **m x n** respectă restricțiile, deci avem o problemă fără constrângeri.

**Populația inițială** va fi reprezentată din **dim** permutări generate aleator de dimensiune **n x m**, fără restricții.

**Funcția fitness** presupune evaluarea costului de transport astfel: se decodifică permutarea după metoda de mai sus și se calculează costul planului de transport. Acesta trebuie să fie minim, deci vom întoarce

**Selecția părinților:** Selectăm populația actuală prin mecanismul de ruletă SUS cu FPS standard deoarece dimensiunile problemei sunt reduse (avem puține fabrici și depozite).

**Recombinare:** Schema generală de recombinare pentru probleme fără constrângeri, reprezentare prin permutări. Ca mecanism, voi folosi recombinarea **CX** cu probabilitate **0,8**.

**Mutație:** Schema generală de mutație pentru probleme fără constrângeri, reprezentare cu permutări. Ca mecanism, voi folosi operatorul de mutație prin interschimbare cu probabilitate **0,1**.

**Selecția generației următoare:** Voi aplica regula ca fiecare individ să aibă o **limită de vârstă 1**, iar ca mecanism voi folosi **elitismul**, alegând cei mai buni indivizi, adică cele mai bune planuri de transport. (va fi necesară decodificarea)

Drept **condiții de terminare a algoritmului genetic**:

1. Numărul maxim de iterații este atins
2. Diversitate genetică (să avem minim 2 indivizi cu fitness diferit)

import numpy as np  
  
COSTURI = np.genfromtxt('COSTURI.txt')  
CERERE = np.genfromtxt('CERERE.txt')  
OFERTA = np.genfromtxt('OFERTA.txt')  
  
m = len(OFERTA)  
n = len(CERERE)  
  
def fitness(individ):  
 x = np.zeros((m,n))  
 for i in individ:  
 lin, col = np.unravel\_index(i, (m,n))  
 x[lin][col] = 1  
 return 1000/np.sum(x\*COSTURI)  
  
def gen\_pop(dim):  
 pop = []  
 for i in range(dim):  
 individ = np.random.permutation(m\*n).tolist()  
 individ += [fitness(individ)]  
 pop.append(individ)  
 return pop  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 NR\_INDIVIZI = 100  
 NMAX = 500  
 PC = 0.8  
 PM = 0.1  
 pop = gen\_pop(NR\_INDIVIZI)  
 print('Populatia initiala:')  
 for line in sorted(pop, key=lambda x: x[-1]):  
 print(line)  
 i=1  
 gata = False  
 while i<=NMAX and not gata:  
 spop = SUS\_cu\_FPS\_standard(pop, NR\_INDIVIZI)  
 copii = incrucisare(spop, NR\_INDIVIZI, PC)  
 copii\_m = mutatie(copii, PM)  
 pop = elitism(pop, copii\_m, NR\_INDIVIZI)  
 if len(set(pop[:,-1])) == 1:  
 gata = True  
 i += 1  
 print('Populatia finala:')  
 for line in sorted(pop, key=lambda x: x[-1]):  
 print(line)

1. Fie un set de n puncte în planul 𝑆, fiecare punct 𝐴 ∈ 𝑆 fiind reprezentat prin coordonatele carteziene (xA, yA), Proiectați și implementați un algoritm genetic pentru a determina 3 puncte coliniare din setul dat, dacă astfel de puncte există. Punctele 𝐴, 𝐵, 𝐶 ∈ 𝑆 sînt coliniare dacă una din următoarele proprietăți este adevărată: - |𝐴𝐵| = |𝐴𝐶| + |𝐶𝐵| - |𝐴𝐶| = |𝐴𝐵| + |𝐵𝐶| - |𝐶𝐵| = |𝐶𝐴| + |𝐴𝐵| unde |𝐴𝐵| = √(xA − xB) 2 + (yA − yB)

Proiectare problemă:

**Reprezentare**:

Fiecare punct are două coordonate pe cele două axe, Ox și Oy. Așadar, un individ va fi format din 6 gene, reprezentând abscisa și ordonata fiecăruia dintre cele 3 puncte.

**Spațiul soluțiilor** este reprezentat de ℝ6.

**Funcția fitness:** Funcția fitness va fi o funcție de maximizare a uneia dintre valorile formulelor , sau .

Singura restricție care se aplică este ca .

**Populația inițială: O matrice** de **dim** linii și 7 coloane, reprezentând coordonatele fiecărui punct și funcția fitness pe ultima coloană, generată aleator, din spațiul soluțiilor.

**Selecția părinților:** Folosim mecanismul ruletei SUS cu FPS sigma scalare pentru a păstra cât mai bine probabilitatea naturală de selecție.

**Recombinare:** Pentru recombinare voi folosi schema recombinării generale a problemelor cu constrângeri, reprezentare cu șiruri de numere reale. Drept mecanism voi folosi **recombinarea uniformă** cu probabilitate **0,8**.

**Mutație**: Pentru mutație voi respecta schema de mutație generală a problemelor cu constrângeri, reprezentarea prin șiruri de numere reale. Drept mecanism voi folosi **resetarea aleatorie** cu o probabilitate de **0,1**.

**Selecția supraviețuitorilor:** Această selecție se face pe baza **elitismului**. Dacă în populația de copii mutanți nu există niciun individ cu un fitness mai bun decât populația de părinți, se elimină aleator unul dintre copiii mutanți și se înlocuiește cu cel mai bun individ din populația de părinți.

**Condiții de terminare ale GA:**

1. Dacă este atins numărul maxim de iterații
2. Dacă oricare doi indivizi din populație au calități diferite
3. Dacă nivelul calității maxime este suficient de aproape de 1

"Gasiti 3 puncte coliniare din spatiul solutiilor [-10;10] NUMERE INTREGI"

"6 gene: x=[x1,y1,x2,y2,x3,y3]"

" 0 1 2 3 4 5 "

"Fitness: (y3-y1)/(y2-y1)=(x3-x1)/(x2-x1) =>daca e true return 1 else return 0"

import numpy

def fitness(x):

if(x[0]==x[2]==x[4] or x[1]==x[3]==x[5]):

return 1

if(x[0]==x[2] or x[1]==x[3]):

return 0

s=(x[5]-x[1])/(x[3]-x[1])

d=(x[4]-x[0])/(x[2]-x[0])

if s==d:

return 1

else:

return 0

def gen(dim):

pop=[]

for i in range(dim):

x=numpy.random.randint(-10,11,6)

val=fitness(x)

x=list(x)

x=x+[val]

pop=pop+[x]

return numpy.asarray(pop)

O imagine care conține text, electronice, Dispozitiv de ieșire, afișaj

Descriere generată automat

**PROIECTARE-PROBLEMA TERRA**

**Reprezentarea unui candidat:** Fiecare candidat va fi reprezentat de un vector de m numere intregi, fiecare genă fiind corespondentă containerului ales, valoare din acea celulă (alela) reprezentând numărul vagonului în care este alocat containerul. |\_3\_|\_2\_|\_3\_|\_1\_|\_1\_|\_1\_|

**Functia fitness:** Pentru evaluarea unui candidat individual, se va calcula scorul fitness adunând greutatea fiecărui container la greutatea alocată pentru fiecare vagon în funcție de containerele prezente în acesta, și se vor aduna diferențele în modul dintre greutățile vagoanelor individuale și media greutăților acestora. Scorul fitness final va fi (1/1+scorul menționat în enunțurile anterioare).

**Generarea populatiei initiale:** Candidatii la solutie vor fi generati in mod aleator conform reprezentarii de un vector de valori. Alelele vor fi alocate in mod random un numar aleator de la 0 la (n-1), reprezentand numarul vagonului in care containerul este alocat. Vagoanele sunt numerotate de la 0 la (n-1). Functia de randomizare folosita va fi np.random.randint(0,n,m).

**Selectia parintilor:** Vom folosi mecanismul de selectie ruleta, intrucat indivizii cu scoruri fitness mai ridicate vor avea o probabilitate mai mare de selectie (cu metoda de descriere/desenare/alocare a probabilitatilor conform metodei sigmafps). Parintii vor fi apoi recombinati doi cate doi, parcurgandu-i conform vectorului selectie\_parinti in care pastram ordinea in care parintii selectati au fost alesi de mecanismul de selectie. Vom selecta *dim* parinti.

**Metoda de crossover/recombinare:** Folosim metoda de crossover uniform pentru a recombina genele parintilor, intrucat alocam o sansa egala fiecarei gene de la fiecare parinte de a fi transmisa mai departe. Probabilitatea de recombinare este setata la ~80%, iar daca aceasta probabilitate nu este indeplinita, atunci vom recurge la recombinare asexuata in care se vor prelua genele de la un singur parinte in mod direct. Vom genera *dim* copii.

**Metoda de mutatie**: Folosim metoda de mutatie fluaj in care vom opera o mutatie pe genele la care este indeplinita probabilitatea de mutatie de ~5%, altfel individul/copilul va ramane in varianta lui fara mutatii suferite. Capetele limita ale intervalului pe care metoda de fluaj poate efectua schimbari este intre 0 si n-1, astfel incat sa nu se aloce un vagon neexistent vreunui container.

**Selectia generatiei urmatore:** Vom folosi mecanismul de selectie numit elitism, in care vom compara populatia initiala de parinti cu populatia nou rezultata de copii, si vom pastra doar *dim* cei mai buni indivizi pentru generatia urmatoare, punandu-i intr-o variabila separata specifica populatiei noi.

**Conditiile de terminare ale algoritmului genetic:** Algoritmul genetic va cauta un individ cu o configuratie optima de-a lungul NMAX generatii. Algoritmul genetic se va opri la solutia cea mai buna din populatia actuala, chiar daca aceasta nu este optima la nivel global, in urmatoarele conditii:

1. Numarul de generatii create a atins numarul maxim de generatii alocat ca parametru de intrare (NMAX). *Iteratie = NMAX*
2. In caz ca scorul fitness maxim (1) a fost atins, atunci algoritmul genetic se opreste.
3. Daca maximul functiei fitness de-a lungul a NMAX/3 generatii nu se imbunatateste, atunci algoritmul genetic este oprit.

**Solutia algoritmului:** Este individul reprezentat conform celor specificate mai sus in forma de vector cu numere intregi, cu fiecare gena reprezentand un anumit container, si fiecare alela reprezentand numarul vagonului in care este alocat containerul. Se va returna individul cu scorul fitness cel mai bun din populatia actuala, indiferent daca algoritmul se opreste inainte de a gasi o solutie optima, si scorul fitness al acestuia.

def fitness(bv, individ,n):  
 m = len(individ)  
 v = np.zeros(n)  
 v=list(v)  
 for i in range(m):  
 v[individ[i]]=v[individ[i]]+bv[i]  
 scor2 = 0  
 media=0  
 for i in range(n):  
 media=media+v[i]  
 media=media/n  
 for i in range(n):  
 scor2=scor2+abs(v[i]-media)  
 scor=1/(1+scor2)  
 return scor  
  
def gen(fg, n, dim):  
 b = np.genfromtxt(fg)  
 b[1]=list(b[1])  
 individ=[]  
 pop=[]  
 scoruri\_pop=[]  
 m = len(b[0])  
 for i in range(dim):  
 individ=np.random.randint(0,n,m)  
 individ=list(individ)  
 pop=pop+[individ]  
 scoruri\_pop=scoruri\_pop+[fitness(b[1], individ, n)]  
 return pop, scoruri\_pop

**CONTINUT FISIER TEXT**

**1 2 3 4 5 6**

**50 10 40 20 70 10**

O imagine care conține text, document, captură de ecran

Descriere generată automat

Pentru rezolvarea problemei :

Reprezentare: -> prin permutari

Fenotip(solutii posibile): Vector cu n elemente (z1,z2,z3,...,zn), ce reprezinta o rearanjare a vectorului Y

Solutia problemei va fi reprezentata de un vector cu n elemente (z1,z2,z3,...,zn), ce reprezinta o rearanjare a vectorului Y care este un multiplu al vectorului X de forma zi=a\*xi+b (genotipul). Fiind vorba de o reprezentare prin intermediul permutarilor, problema nu are constrangeri, deoarece orice permutare generata poate fi o solutie.

Functia obiectiv (fitness):

Functia fitness sta la baza procesului de selectie. Aceasta calculeaza calitatea individului evaluat. Ea va primi ca parametrii individul(fenotipul/permutarea), vectorul x si numerele naturale a si b date. Calitatea maxima a functiei este n, si anume numarul de valori ale vectorului z, zi care sunt egale cu a\*xi+b. Pentru fiecare valoare care nu respecta aceasta cerinta, din calitatea maxima se va scadea 1.

Populatia initiala:

Aceasta va consta in generarea aleatoare de fenotipuri si anume de rearanjari ale vectorului Y dat la inceput.

Recombinare:

Pentru recombinare am ales OCX deoarece pare sa fie cel care se potriveste cel mai bine in contextul problemei, chiar daca setul de permutari nu semnifica ordinea de aparitie a unor evenimente, dar acestea au legatura cu ordinea elementelor din vectorul X

Muatia:

Pentru mutatie am ales interschimbarea deoarece prin acest operator avem cele mai mari sanse de a nu denatura un individ care are deja gene care respecta conditia initiala

Selectia parintilor:

Pentru selectia parintilor am ales mecanismul SUS (ruleta multibrat) si am calculat probabilitatea de recombinare folosind FPS(Fitness Proportional Selection) cu sigma scalare deoarece este o versiune imbunatatita a FPS standard .

Conditia de oprire a algoritmului:

Algoritmul se va opri atunci cand calitatea populatiei nu mai este imbunatatita sau este atinsa calitatea maxima intrucat aceasta este deja cunoscuta.

O imagine care conține text, captură de ecran, Font, document

Descriere generată automat

Reprezentare: prin permutari

Fenotip(solutii posibile) : Vector cu n elemente ce reprezinta o aranjare circulara a numerelor date

Solutia problemei este reprezentata de un vector cu n elemente (x1,x2,...,xn) ce reprezinta o aranjare a celor n numere date astfel incat suma produselor oricaror doua numere alaturate sa fie maxima(genotipul).

Fiind vorba de o reprezentare prin intermediul permutarilor, problema nu are constrangeri deoarece orice permutare generata poate fi o solutie.

Functia fitness(obiectiv):

Functia fitness sta la baza procesului de selectie. Aceasta calculeaza calitatea individului evaluat. Ea va primi ca parametrii individul (permutarea). Calitatea maxima a functiei este n, si anume numarul de perechi de numere alaturate a caror suma este maxima. Pentru fiecare valoare care nu respecta aceasta cerinta, din calitatea maxima se va scadea 1.

Populatia initiala:

Aceasta va consta in generarea aleatoare de fenotipuri si anume de permutari ale celor n numere primite la inceput.

Recombinare:

Pentru recombinare am ales CX deoarece pare sa fie cel care se potriveste cel mai bine in contextul problemei, fiecare element apare in pereche cu un alt element al aceluiasi ciclu in perechea de cromozomi parinte.

Muatia:

Pentru mutatie am ales interschimbarea deoarece prin acest operator avem cele mai mari sanse de a nu denatura un individ care are deja gene care respecta conditia initiala

Selectia parintilor:

Pentru selectia parintilor am ales mecanismul SUS (ruleta multibrat) si am calculat probabilitatea de recombinare folosind FPS(Fitness Proportional Selection) cu sigma scalare deoarece este o versiune imbunatatita a FPS standard, iar pentru selectia generatiei urmatoare am ales selectia elitista pentru a nu pierde cel mai bun individ.

Conditia de oprire a algoritmului:

Algoritmul se va opri atunci cand calitatea populatiei nu mai este imbunatatita sau este atinsa calitatea maxima intrucat aceasta este deja cunoscuta.

POATE TURNIR ca sunt valori multe

Nume:

Grupa:

Test P.E.A.G

Enunt: Pregătind apărarea țării, un conducător din perioada medievală creează un sistem de fortificații la granițe pe care le înzestrează cu 3 tipuri de tunuri din bronz, având la dispoziție 20.000 de unități monetare.

Fiecare tip de tun, numerotat de la I la III are următoarele caracteristici:

I.                   costă 500 de unități, poate să arunce o ghiulea la 500 de metri și are timpul de încărcare de 4 minute;

II.                costă 400 de unități, poate să arunce o ghiulea la 300 de metri și are timpul de încărcare de 2 minute;

III.              costă 750 de unități, poate să arunce o ghiulea la 800 de metri și are timpul de încărcare de 5 minute.

Problema este de a calcula câte tunuri din fiecare tip trebuie cumpărate astfel încât

-                      să nu fie depășită suma disponibilă;

-                      distanța medie la care poate fi aruncată ghiuleaua să fie maximă;

-                      timpul mediu de încărcare să nu depășească 3.5 minute.

            Observație. Dacă numărul tunurilor din fiecare tip este a, b, respectiv c, atunci distanța medie este (500 \* a + 300 \* b + 800 \* c) / (a + b + c) și timpul mediu de încărcare este

(4 \* a + 2 \* b + 5 \* c) / (a + b + c)

Reprezentare: siruri de numere intregi

Fenotip(solutii posibile) : Cantitatile care trebuiesc cumparate din fiecare dintre cele 3 tipuri de tunuri de bronz pentru a maximiza distanta medie la care poate fi aruncata ghiuleaua, a avea un timp mediu de incarcare care sa nu depaseasca 3,5 minute, dar in acelasi timp incadrandu-se in bugetul de 20 000 de unitati monetare.

Genotip: Sir de 3 numere intregi ce reprezinta cantitatea ce trebuie achizitionata din fiecare tip de tun. Vector cu 3 elemente (a,b,c) ce reprezinta combinatia de tunuri din fiecare tip, unde a=numarul de tunuri care costa 500 unitati monetare, pot arunca o ghiulea la 500 de metri si au un timp de incarcare de 4 min; b= numarul de tunuri care costa 400 unitati monetare, pot arunca o ghiulea la 300 de metri si au un timp de incarcare de 2 min; c= numarul de tunuri care costa 750 unitati monetare, pot arunca o ghiulea la 800 de metri si au un timp de incarcare de 5 min.

Constrangerile problemei sunt bugetul de 20 000 de unitati monetare, timp mediu de incarcare de maxim 3,5 minute.

Sub forma matematica problema ar arata astfel:

[max] (500\*a+300\*b+800\*c)/(a+b+c)

500\*a+400\*b+750\*b<=20 000

(4\*a+2\*b+5\*c)/(a+b+c)<=3,5

Functia fitness(obiectiv): Functia fitness este de maximizare deoarece se doreste sa se obtina distanta maxima la care poate fi aruncata ghiuleaua. Se va folosi ca si la celelalte probleme de optim produsul scalar pentru a realza o medie ponderata.

Populatia initiala: Populatia initiala este generata pe baza functiei obiectiv, verificand constrangerile problemei – limita bugetului si timpul mediu maxim de incarcare

Recombinare: Am ales operatorul de incrucisare uniforma deoarece vectorul solutie are doar 3 elemente si poate fi recombinat prin aceasta metoda, astfel recombinarea va fi semnificativa.

Mutatia: Am ales operatorul de resetare aleatoare care modifica valoarea unei alele cu o valoare aleatoare deoarece se potriveste sirului de numere intregi si deoarece intervalul de valori este mare si este mai usor de controlat prin aceasta metoda decat prin mecanismul de fluaj.

Selectia parintilor: Pentru selectia parintilor am ales mecanismul SUS (ruleta multibrat) si am calculat probabilitatea de recombinare folosind FPS(Fitness Proportional Selection) cu sigma scalare deoarece este o versiune imbunatatita a FPS standard .

Conditia de oprire a algoritmului: Algoritmul se opreste atunci cand se atinge un numar maxim de iteratii ( generatii ) care este dat de parametrul NRMAX. Valoarea optima nu este cunoscuta, din aceasta cauza nu putem utiliza aceasta conditie de oprire.

**StuDocu nu este sponsorizat sau avizat de nicio universitate**

**PEAG project - CSIE, An 2, Sem. 2, Informatica economica**

**Proiect PEAG seminar, 50% nota finala.**

**Programare evolutiva si algoritmi genetici (Academia de Studii Economice din**

**București)**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**DOCUMENTAȚIE**

**PROIECT**

**PEAG**

**1**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**30.Elaborați un proiect pentru rezolvarea genetică a următoarei probleme de tip amestec. O**

**fabrică de produs şi ambalat cafea utilizează două tipuri de materii prime, cafea braziliană şi**

**cafea columbiană şi produce două mărci de cafea, Super şi Lux. Fiecare kilogram de cafea Super**

**conţine cafea braziliană şi cafea columbiană în cantităţi egale, iar un kilogram de cafea din**

**marca Lux conţine 25% cafea braziliană şi 75% cafea columbiană. Resursele disponibile unei**

**linii de producţie sunt de 120 kg. cafea braziliană şi 160 kg. cafea columbiană. Dacă profitul**

**obţinut din vânzarea unui kg. din marca Super este de 20 de unităţi şi profitul rezultat prin**

**vânzarea unui kg. de cafea din marca Lux este de 30 de unităţi, problema este de a determina**

**numărul de kg. din fiecare marcă de cafea care trebuie produse pentru a obţine profit maxim.**

**1 kg SUPER =50%braziliana +50%columbiana**

**1 kg LUX=25%braziliana + 75%columbiana**

**X kg SUPER = 0.5 X braziliana+ 0.5 X columbiana**

**Y kg LUX = 0.25 Ybraziliana +0.75Y columbiana**

**Xkg SUPER +y kg LUX = (0.5X +0.25Y) braziliana +(0.5X +0.75Y)columbiana**

**Dar -braziliana max 120kg**

**-columbiana max 160kg**

**=> 0.5X +0.25Y <=120**

**0.5X+0.75Y <=160**

**Componentele unui Algoitm Genetic (AG) :**

**2**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

** Reprezentarea (definirea membrilor populaţiei)**

** Funcţia de evaluare (fitness)**

** Populaţia**

** Mecanismul de selectare a părinţilor**

** Operatorii de variaţie (recombinarea şi mutaţia)**

** Mecanismul de selectare a membrilor generaţiei următoare**

** Definirea modulului de iniţializare (determinarea populaţiei iniţiale)**

** Definirea condiţiei terminale.**

**Structura generala a unui AG , respecta structura unui algoritm evolutiv ( EA) :**

**1.Modelul de populatie**

**= populatie cu dimensiune fixa dim , deoarece nu se aduc imbunatatiri daca modificam dimensiunea.**

**2.Asocierea genotip-fenotip**

**\*Fenotip: combinatie de 2 cantitati de marci de cafea x->marca Super , y->marca Lux**

**\*Genotip: vector cu 2 elemente numere intregi v=(x,y), X-Columbiana; Y-Braziliana**

**\*Restrictii:**

**0.5X +0.25Y <=120**

**3**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**0.5X+0.75Y <=160**

**X [0,240];**

**Y[0,213];**

**\*Reprezentare populatie: matrice dimx3, in care fiecare linie k e formata din:**

**->reprezentarea unui cromozom v -> pop(k,1:2)=v;**

**->valoarea functiei obiectiv -> pop(k,3)=F(x,y)**

**3.Generarea populatiei**

**=aleator, cu respectarea restrictiilor**

**\*conditie de terminare: numar limitat de generatii**

**4.Selectia parintilor**

**\*Functia obiectiv are valori positive => distributie de probabilitate de selectie FPS cu sigma scalare**

**\*Ruleta de tip SUS**

**5.Recombinarea**

**\*Operatorul de recombinare incrucisata uniforma cu probabilitate pc**

**\*Perechile de parinti sunt unice, indivizii parinti sunt folositi doar o data**

**\*Descendenti sunt mereu viabili datorita acestui tip de recombinare.**

**6.Mutatie**

**\*Resetare aleatoare cu probabilitatea pm.**

**\*Trebuie verificata viabilitatea descendentilor, conform restrictiilor**

**0.5X +0.25Y <=120**

**0.5X+0.75Y <=160**

**X [0,240];**

**Y[0,213];**

**7.Selectie generatia urmatoare**

**4**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**\*Inlocuire generatie cu varianta aleasa cu algoritmul genitor2.**

**PARAMETRII:**

**\*dim->dimensiune populatie**

**\*maxim->numar iteratii maxime**

**\*pm->probabilitate de mutatie**

**\*pc->probabilitate de combinare**

**\*P->vector de profituri din fiecare generatie**

**\*S->vector de solutii din fiecare generatie**

**Codul sursa**

**-----------------------------------------algoritmul genetic----------------------------------------------**

**function[]=cafea(dim,pm,pc,maxim)**

**Pop=gen\_pop(dim);**

**P=zeros(1,maxim); %multimea profiturilor corespunzatoare fiecarei**

**generatii , obtinute de-a lungul evolutiei**

**S=zeros(maxim,3); %multimea solutiilor**

**%cele mai bune din ambele vor fi ultimele si ele vor fii**

**%profitul si solutia cautata**

**[val,i]=max(Pop(:,3));**

**P(1)=val;**

**5**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**[~,poz]=ismember(val,Pop(:,3));**

**S(1,:)=Pop(poz,:);**

**t=1;**

**while t<maxim**

**disp('Parinti');**

**Parinti=SUS(Pop);**

**disp(Parinti);**

**disp('Copii recombinare');**

**pp=unifrnd(0,1);**

**Copii=recombinare\_incrucisare\_uniforma(Parinti,pc,pp);**

**disp(Copii);**

**disp('Copii mutatie');**

**a=[0,0];**

**b=[240,213]**

**Copii2=mutatie(Copii,a,b,pm);**

**disp(Copii2);**

**disp('Generatia urmatoare');**

**Urmasi=genitor2(Pop,Copii2);**

**disp(Urmasi);**

**t=t+1;**

**[val,poz]=max(Urmasi(:,3));**

**P(t)=val;**

**disp(val);disp(poz);**

**S(t,:)=Urmasi(poz,:);**

**Pop=Urmasi;**

**end;**

**disp('Profituri');disp(P);**

**disp('Solutii');disp(S);**

**disp('Profitul cel mai mare:'); disp(P(t));**

**disp('Cantitatile sunt');disp(S(t,1)); disp(S(t,2));**

**end**

**----------------------------functia obiectiv-----------------------------**

**function[val]=functie\_obiectiv(x,y)**

**val=20\*x+30\*y;**

**end**

**-----------------------generarea populatiei------------------**

**6**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**function[pop]=gen\_pop(dim)**

**pop=zeros(dim,3);**

**for i=1:dim**

**x=unidrnd(241)-1;**

**y=unidrnd(214)-1;**

**while( (0.5\*x +0.25\*y > 120) || (0.5\*x +0.75\*y >160) )**

**x=unidrnd(241)-1;**

**y=unidrnd(214)-1;**

**end;**

**pop(i,1)=x;**

**pop(i,2)=y;**

**pop(i,3)=functie\_obiectiv(x,y);**

**end;**

**disp(pop);**

**end**

**------------------------------- SUS-----------------------------------**

**function[Pop]=SUS(pop)**

**[dim,n]=size(pop);**

**Pop=zeros(dim,n);**

**c=2;**

**media=mean(pop(:,n));**

**sigma=std(pop(:,n));**

**val=media-c\*sigma;**

**p=zeros(1,dim);**

**for i=1:dim**

**p(i)=max([pop(i,n)-val 0]);**

**end;**

**suma=sum(p);**

**g=p/suma;**

**q=zeros(1,dim);**

**for i=1:dim**

**q(i)=sum(g(1:i));**

**end;**

**i=1 ; j=1;**

**r= unifrnd(0,1/dim);**

**while (i<=dim)**

**while(r<=q(j))**

**Pop(i,:)=pop(j,:);**

**i=i+1;**

**r=r+1/dim;**

**end;**

**j=j+1;**

**end;**

**end**

**7**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**-------------------------recombinarea ------------------------**

**function[Pop]=recombinare(pop,pc,p)**

**Pop=pop;**

**[dim,n]=size(pop);**

**for i=1:2:dim**

**x=pop(i,:);**

**y=pop(i+1,:);**

**r=unifrnd(0,1);**

**if(r<pc)**

**for k=1:n-1**

**q=unifrnd(0,1);**

**if(q<p)**

**Pop(i,k)=pop(i,k);**

**Pop(i+1,k)=pop(i+1,k);**

**else**

**Pop(i,k)=pop(i+1,k);**

**Pop(i+1,k)=pop(i,k);**

**end ;**

**end;**

**Pop(i,n)=functie\_obiectiv(Pop(i,1),Pop(i,2));**

**end;**

**end;**

**end**

**-------------------------- mutatia ---------------------------**

**function[Pop]=mutatie(pop,a,b,pm)**

**Pop=pop;**

**[dim,n]=size(pop);**

**for i=1:dim**

**gata=0;**

**r=unifrnd(0,1);**

**if r<pm**

**Pop(i,1)=unidrnd(b(1)-a(1)+1)+a(2)-1;**

**gata=gata+1;**

**end;**

**r2=unifrnd(0,1);**

**if r2<pm**

**Pop(i,2)=unidrnd(b(2)-a(2)+1)+a(2)-1;**

**gata=gata+1;**

**end;**

**if gata~=0**

**while( (0.5\*Pop(i,1) +0.25\*Pop(i,2) > 120) || (0.5\*Pop(i,1) +0.75\*Pop(i,2)**

**>160) )**

**8**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**Pop(i,1)=unidrnd(b(1)-a(1)+1)+a(2)-1;**

**Pop(i,2)=unidrnd(b(2)-a(2)+1)+a(2)-1;**

**end;**

**end;**

**if gata>0**

**Pop(i,3)=functie\_obiectiv( Pop(i,1),Pop(i,2));**

**end;**

**end;**

**end**

**-------------------------- genitor2---------------------------**

**function[Pop]=genitor2(parinti,copii)**

**[dim,n]=size(parinti);**

**pop1=sortrows(parinti,3);**

**pop2=sortrows(copii,3);**

**Pop=pop1;**

**i=fix(dim/2);**

**Pop(1:i,:)=pop2(dim:-1:dim-i+1,:);**

**end**

**-------------------exemplu de rulare ---------------------**

**function []=rulare()**

**cafea(6,0.8,0.8,6);**

**end**

**9**

**Desc?rcat de Laura Huluba (hulubalaura31@gmail.com)**

**lOMoARcPSD|5146640**

**PROBLEMA CAFEA**

Problema alegerii cursurilor (ex 3)

**Cerinta:**

Utilizați un algoritm genetic pentru a calcula, dacă există, o modalitate corectă de aranjare a n piese de sah pe o tabla n\*n

**Proiectul de rezolvare:**

1. **Populatia initiala:**

Forma de reprezentare: permutari in care ordinea evenimentelor este importanta

Problema fara constrangeri : problema cu paritatea am integrat-o in functia obiectiv

Cromozomi: permutari de la 0 la n-1 care contin pozitia coloanei in care se afla piesa de sah (pozitia liniei fiind pozitia din vector) si calitatea indivizilor . Exemplu: [ 5, 4, 1, 6, 0, 2, 3, 7, 24]

Calitatea inseamna numarul de piese pozitionate corect pe tabla

Functia obiectiv: maximizarea numarului de piese corect pozitionate

Domeniu de definitie: [0,n\*(n-1)/2].

Generare populatie: aleator

Dimensiune populatie: 3

Probabilitate de recombinare: 0.7 / 0.8

Probabilitate de mutatie: 0.1 / 0.2

Iteratii: 100

1. **Conditii de determinare posibile:**

Cand s-a atins numarul de iteratii sau calitatea este maxima (s-a gasit individul cautat)

1. **Selectia parintilor**

Distributia de selectie: FPS cu sigma scalare deoarece aceasta depinde de calitatea individului

Selectia: tip SUS (multibrat) pt a evita procesul de convergenta aparut la FPS cu un singur brat in urma faptului ca nu exista aproape nicio constrangere de selectie

1. **Recombinare**

Varianta fara constrangeri

Operator recombinare: OCX intrucat tinem cont de ordinea evenimentelor

1. **Mutatie**

Operatorul de mutatie prin interschimbare pentru permutari

1. **Selectie generatie urmatoare**

Operator: selectie elitistica – deoarece urmarim evolutia indivizilor

1. **Rulare**

Exemple:

Intrare:

import TEST as t

s,v=t.TEST(8,50,0.8,0.1,100)

Iesire:

S= array([3., 5., 0., 1., 6., 4., 2., 7.])

V=[23.0, 23.0, 23.0, 21.0, 18.0, 18.0, 19.0, 20.0, 14.0, 13.0, 17.0, 15.0, 19.0, 22.0, 14.0, 15.0, 15.0, 14.0, 22.0, 22.0, 22.0, 22.0, 21.0, 21.0, 21.0, 20.0, 20.0, 22.0, 21.0, 21.0, 20.0, 21.0, 14.0, 12.0, 17.0, 21.0, 21.0, 15.0, 14.0, 14.0, 14.0, 23.0, 23.0, 14.0, 17.0, 23.0, 24.0, 23.0, 23.0, 23.0, 17.0, 22.0, 23.0, 17.0, 17.0, 17.0, 17.0, 18.0, 18.0, 22.0, 17.0, 14.0, 22.0, 22.0, 22.0, 21.0, 21.0, 21.0, 21.0, 16.0, 16.0, 17.0, 22.0, 22.0, 17.0, 24.0, 22.0, 16.0, 22.0, 22.0, 22.0, 22.0, 14.0, 19.0, 19.0, 24.0, 15.0, 21.0, 22.0, 22.0, 17.0, 22.0, 17.0, 13.0, 17.0, 14.0, 17.0, 14.0, 13.0, 15.0]

PIESE DE SAH

def f\_obiectiv(x):

# functia obiectiv

# I: x - individul (permutarea) evaluat(a)

# E: c - calitate (numarul de perechi de piese corect pozitionate)

n=len(x)

c=n\*(n-1)/2

for i in range(n):

for j in range(i+1,n):

if abs(i-j)==abs(x[i]-x[j]):

c=c-1

elif (abs(i-j))%2!=abs((x[i]-x[j]))%2:

c=c-1

return c

def gen\_pop\_perm(dim, n):

# generare populatie de permutari, varianta cu constrangeri

# I: dim - dimensiune populatie (nr. de indivizi)

# n - dimensiune individ (numar de gene)

# E: pop - populatia aleatoare generata, cu calitatea fiecarui individ pe ultima coloana

pop=numpy.zeros((dim,n+1),dtype=int)

for i in range(dim):

pop[i,:n]=numpy.random.permutation(n)

pop[i,n]=f\_obiectiv(pop[i,:n])

    return pop

O imagine care conține text, captură de ecran, Font, document

Descriere generată automat

Reprezentare: prin permutari

Fenotip(solutii posibile) : Vector cu n elemente ce reprezinta o aranjare circulara a numerelor date

Solutia problemei este reprezentata de un vector cu n elemente (x1,x2,...,xn) ce reprezinta o aranjare a celor n numere date astfel incat suma produselor oricaror doua numere alaturate sa fie maxima(genotipul).

Fiind vorba de o reprezentare prin intermediul permutarilor, problema nu are constrangeri deoarece orice permutare generata poate fi o solutie.

Functia fitness(obiectiv):

Functia fitness sta la baza procesului de selectie. Aceasta calculeaza calitatea individului evaluat. Ea va primi ca parametrii individul (permutarea). Calitatea maxima a functiei este n, si anume numarul de perechi de numere alaturate a caror suma este maxima. Pentru fiecare valoare care nu respecta aceasta cerinta, din calitatea maxima se va scadea 1.

Populatia initiala:

Aceasta va consta in generarea aleatoare de fenotipuri si anume de permutari ale celor n numere primite la inceput.

Recombinare:

Pentru recombinare am ales CX deoarece pare sa fie cel care se potriveste cel mai bine in contextul problemei, fiecare element apare in pereche cu un alt element al aceluiasi ciclu in perechea de cromozomi parinte.

Muatia:

Pentru mutatie am ales interschimbarea deoarece prin acest operator avem cele mai mari sanse de a nu denatura un individ care are deja gene care respecta conditia initiala

Selectia parintilor:

Pentru selectia parintilor am ales mecanismul SUS (ruleta multibrat) si am calculat probabilitatea de recombinare folosind FPS(Fitness Proportional Selection) cu sigma scalare deoarece este o versiune imbunatatita a FPS standard, iar pentru selectia generatiei urmatoare am ales selectia elitista pentru a nu pierde cel mai bun individ.

Conditia de oprire a algoritmului:

Algoritmul se va opri atunci cand calitatea populatiei nu mai este imbunatatita sau este atinsa calitatea maxima intrucat aceasta este deja cunoscuta.

POATE TURNIR ca sunt valori multe

**def f\_obiectiv(x,vector):**

**n=len(x)**

**vector\_sortat=numpy.zeros(n-1,dtype=int)**

**for i in range(n-1):**

**vector\_sortat[i] = vector[x[i]]**

**y=vector\_sortat[:n-1]**

**c=numpy.dot(vector\_sortat[:n-2],y)**

**return c**

**def gen\_pop(dim,n,vector):**

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

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

**pop[i, :n] = numpy.random.permutation(n)**

**pop[i, n] = f\_obiectiv(pop[i, :n],vector)**

**return pop**

**PERIMETRU MAXIM LA CV>?**

import numpy as np  
import math  
import random as rp  
  
# inputul este o matrice cu n linii si 2 coloane (coloana 1 - coordonata lui x, coloana 2- coordonata lui y)  
  
# f. obiectiv  
def PerMax(x, n):  
 # functia obiectiv pentru problema perimetrului maxim  
  
 # I: x - individul (permutarea) evaluat(a), n-dimensiunea problemei  
 # E: per- calitate  
 #perimetru = 0  
 latura1 = math.sqrt(pow(x[2] - x[0], 2) + pow(x[1] - x[3], 2))  
 latura2 = math.sqrt(pow(x[2]- x[4], 2) + pow(x[3]- x[5], 2))  
 latura3 = math.sqrt(pow(x[0] - x[4], 2) + pow(x[1] - x[5], 2))  
 perimetru = latura1 + latura2 + latura3  
 return perimetru  
  
  
  
   
  
  
#genereaza populatia initiala  
#I:  
# fc, fv - numele fisierelor cost, valoare  
# max - capacitatea maxima  
# dim - numarul de indivizi din populatie  
#E: pop - populatia initiala  
# dim,nc,v,max:datele problemei - pentru transferuri in alte functii  
def gen(n,dim):  
   
 #lucreaza cu populatia ca lista de dim elemente - liste cu cate n+1 indivizi  
 pop=[]  
 for i in range(dim):  
 x=np.zeros(n)  
 for i in range(n):  
 #am gasit o solutie candidat fezabila, in data de tip ndarray (vector) x  
 x[i]=rp.randint(-10,10)  
 val=PerMax(x,n)  
 # 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]  
   
 return pop,n