**Algoritm evolutiv**

1. Generare populație inițială aleatorie (populație P).
2. Evaluarea populației pentru a obține best, avg și worst
3. Selecție părinți.
4. Formarea copiilor prin încrucișare (C)
5. Transformarea copiilor print mutație (M)
6. Selecție supraviețuitori: cei mai buni mărime(P) din P+M. Populația nouă se reține în P
7. Evaluarea poluției noi pentru a obține best, avg și worst
8. Cât timp număr de generației e mai mic decât numărul de generații totale se execută totul începând de la pasul 3
9. Se returnează cel mai bun individ(best) din ultima generație

**Parametri algoritm: mărimea populației, numărul de generații, rata de încrucișare, rata de mutație.**

func algoritmEvolutiv(mărimePopulație, ratăMutație, ratăÎncrucișare, generații):P = inițializarePopulație(mărimePopulație)  
 b, a, w = evaluare(P) # best, avg, worst  
 nrÎncrucișări = int((mărimePopulație \* ratăÎncrucișare / 100) / 2)  
 pentru i între (0, generații - 1):

părinți = selecție(P)  
 C = încrucișare(nrÎncrucișări, părinți)  
 M = mutații(C)  
 P = populațieNouă(P, M)  
 b, a, w = evaluarePopulație(P)  
 returnează b

**Explicații algoritmi și pseudo cod problema 1**

**Problema rucsacului**

* n obiecte, fiecare obiect are o valoare (v) și o greutate (w)
* Obiectiv: puneți în rucsac valoarea maximă fără a depăși greutatea maximă admisă W
* 𝑥𝑖=1 înseamnă obiectul i este pus în rucsac
* 𝑥𝑖=0 înseamnă obiectul i nu este pus în rucsac

****

**Pentru încrucișare**

Se folosește încrucișarea printr-un punct de tăietură pentru reprezentare binară.

1. Se stabilește un punct de tăietură aleatoriu k în intervalul (0, lungime soluție)
2. Se formează copiii(C1 și C2) din părinți(P1 și P2) astfel

C1 = P1[:k) + P2(k:]

C2 = P2[:k) + P1(k:]

func încrucișarePunctTăietură(P1, P2):  
 k = randomInt(1, problem\_size - 1) # punctul de tăietură  
 C1 = P1[:k] + P2[k:]  
 C2 = P2[:k] + P1[k:]  
 returnează C1, C2

Se folosește încrucișarea prin două puncte de tăietură pentru reprezentare binară.

1. Se stabilesc două puncte de tăietură aleatoriu i, j în intervalul (0, lungime soluție)
2. Se formează copiii(C1 și C2) din părinți(P1 și P2) astfel

C1 = P2[:i) + P1[i:j] + P2(j:]

C2 = P1[:i) + P2[i:j] + P1(j :]

func încrucișarePunctTăietură(P1, P2):  
 i, j = random2Point(1, problem\_size - 1)  
 C1 = P2[:i] + P1[i:j+1] + P2[j+1:]  
 C2 = P1[:i] + P2[i:j+1] + P1[j+1:]  
 returnează C1, C2

**Pentru mutație**

Se folosește mutație tare.

1. Pentru un membru din populația de copii și pentru fiecare poziție din acel mebru al populație se execută:

1.1 Se generează un număr aleatoriu în intervalul (0, 1)

1.2 Dacă q < rm atunci se transformă valoarea lui c(i, j) în 1 dacă era 0 și în 0 dacă era 1

1.3 Altfel nu se întâmplă nimic

func mutație(C, rm):

M = C  
 pentru c în M:  
 pentru i între (0, lungime(c)):  
 dacă random(0, 1) < rm:  
 c[i] = 1 - c[i] # Bit flip  
 returnează M

Se folosește mutație slabă.

1. Pentru un membru din populația de copii și pentru fiecare poziție din acel mebru al populație se execută:

1.1 Se generează un număr aleatoriu în intervalul (0, 1)

1.2 Dacă q < rm atunci se transformă valoarea lui c(i, j) într-o valoare întâmplătoare, mai precis 1 sau 0

1.3 Altfel nu se întâmplă nimic

func mutație(C, rm):

M = C  
 pentru c în M:  
 pentru i între (0, lungime(c)):  
 dacă random(0, 1) < rm:  
 c[i] = random([0,1])  
 returnează M

**Tabele de date**

**Testele pentru algoritm au fost făcute începând de la un caz de bază pentru care am tot modificat parametri.**

**Pentru compararea cu alte variante ale algoritmului au fost schimbate doar încrucișarea și mutația pentru cazul de bază.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Problema | Nr rulări | Medie best | Medie average | Medie worst | Cel mai bun best | Mărime populație | Rata de încrucișare | Rata de mutație | Număr generații | Operator de încrucișare | Operator de mutație | Timpul mediu de execuție |
| Rucsac-20.txt | 10 | 699.9 | 642.98 | 622.2 | 716 | 50 | 60 | 0.4 | 200 | Un punct de tăietură | Tare | 1.42 |
| 667.1 | 637.31 | 621.9 | 685 | 10 | 60 | 0.4 | 200 | 0.25 |
| 688.9 | 632.74 | 611.2 | 706 | 50 | 40 | 0.4 | 200 | 0.99 |
| 726.0 | 726.0 | 726.0 | 726 | 50 | 60 | 0.1 | 200 | 1.24 |
| 671.9 | 602.8 | 577.0 | 694 | 50 | 60 | 0.4 | 50 | 0.34 |
| 724.4 | 704.48 | 692.9 | 726 | 50 | 60 | 0.4 | 200 | Slabă | 1.22 |
| 691.6 | 642.09 | 621.6 | 710 | 50 | 60 | 0.4 | 200 | Două puncte de tăietură | Tare | 1.46 |
| 724.4 | 705.05 | 693.8 | 726 | 50 | 60 | 0.4 | 200 | Slabă | 1.62 |
| Rucsac-200.txt | 10 | 133372.1 | 132715.83 | 132486.8 | 133679 | 50 | 60 | 0.4 | 200 | Un punct de tăietură | Tare | 11.25 |
| 133123.0 | 132773.81 | 132551.9 | 133517 | 10 | 60 | 0.4 | 200 | 2.48 |
| 133302.6 | 132609.06 | 132358.2 | 133933 | 50 | 40 | 0.4 | 200 | 9.23 |
| 134585.4 | 134078.83 | 133899.1 | 135002 | 50 | 60 | 0.1 | 200 | 11.95 |
| 133141.3 | 132259.90 | 131958.0 | 133427 | 50 | 60 | 0.4 | 50 | 2.79 |
| 133917.7 | 133411.62 | 133197.5 | 134169 | 50 | 60 | 0.4 | 200 | Slabă | 11.79 |
| 133350.0 | 132714.38 | 132475.7 | 133798 | 50 | 60 | 0.4 | 200 | Două puncte de tăietură | Tare | 12.55 |
| 133997.3 | 133370.55 | 133155.5 | 134379 | 50 | 60 | 0.4 | 200 | Slabă | 11.95 |

**Reprezentări grafice**

**Chart

Description automatically generated**

Problema: rucsac-20.txt

Mărime populație: 50

Rată de încrucișare: 60

Rată de mutație: 0.4

Număr generații: 150

Operator încrucișare: încrucișare prin două tăieturi

Operator mutație: mutație slabă

Best: 726

**A picture containing chart

Description automatically generated**

Problema: rucsac-200.txt

Mărime populație: 560

Rată de încrucișare: 60

Rată de mutație: 0.3

Număr generații: 200

Operator încrucișare: încrucișare prin o tăietură

Operator mutație: mutație tare

Best: 133504

**Observații**

Toate dintre următoarele observații sunt făcute având în vedere atât rezultatele obținute anterior cât și rezultatele obținute cu diferiți parametri pentru acest algoritm. Observațiile sunt:

* Timpul mediu de execuție a crescut tot mai mult cu cât oricare dintre parametri a fost mai mare;
* Pentru probleme foarte mici se poate ajunge foarte repede la rezultate bune(a se vedea ultima și prima instanță pentru problema rucsacului) și nu este necesar tot timpul un număr mare de generații;
* În cazul în care singura diferență între execuțiile algoritmului a fost numărul de generații, se poate observa că pentru un număr suficient de mare de generații se poate ajunge la rezultate mai bune;
* În cazul în care singura diferență a fost rata de mutație, se poate observa că se obțin rezultate cu mult mai bune pentru o rată de mutație mai mică
* În cazul în care singura diferență a fost rata de încrucișare, se poate observa că se obțin rezultate mai bune pentru o rată de încrucișare mai mare;
* În comparație cu algoritmi folosiți precedent, se poate observa că acest algoritm dă rezultate mult mai bune. Drept exemplu considerăm instanța problemei numită rucsac-200.txt, unde pentru TS cel mai bun rezultat a fost 133844, pentru RS a fost 133164 și pentru RHC a fost 133342, care sunt mult mai mici decât cel mai bun rezultat obținut aici, 135002;
* Pentru o rată de mutație prea mică valorile de best average și worst pot ajunge egale din cauză că nu se obțin rezultate destul de diversificate, dar se obțin rezultate mai bune. Pentru instanța rucsac-200.txt nu face a dat cel mai bun rezultat;
* S-au obținut rezultatele cele mai bune, în funcție de mediile obținute, pentru încrucișarea prin două tăieturi și mutația slabă pentru ambele instanțe ale problemei. Rezultatele sunt destul de apropiate față de cele pentru încrucișarea printr-o tăietură și mutație slabă;
* În reprezentarea grafică se poate observa cum se maximizează soluția obținută și cum după un anumit număr de generații valoarea cea mai bună nu se mai modifică.

**Concluzie**

Acest algoritm a dat rezultate mult mai bune decât algoritmi utilizați precedent, pentru un timp mult mai scurt de execuție. Cele mai bune rezultate au fost obținute pentru populații mai mari, cu un număr mai mare de generații, cu o rată de încrucișare mai mare și o rată de mutație mai mică. Se poate observa că în funcție de fiecare operator de mutație și încrucișare se obțin rezultate foarte diferite, însă nu se poate considera că au fost testate toate cazurile posibile.

**Explicații algoritmi și pseudo cod problema 2**

**TSP**

* n orașe reprezentate prin coordonatele lor;
* Există un drum între fiecare două orașe;
* Distanța dintre orașe este distanța euclidiană;
* Se poate trece o singură dată printr-un oraș;
* Trebuie să se ajungă înapoi la punctul de start;
* Soluțiile sunt reprezentate drept permutări de n elemente;
* Valoarea unei soluții este dată de către distanța parcursă.

Aceasta este o problemă de minimizare.

**Pentru încrucișare**

Încrucișare ciclică(CX).

1. Construirea unui ciclu de alele din P1 astfel:
   1. Prima poziție–prima poziție din P1.
   2. Merge la aceeași poziție din P2.
   3. Merge la poziția cu aceeași valoare din P1.
   4. Adaugă această poziție în ciclu.
   5. Repetă pașii 1.2-1.4 până când ajunge din nou la prima poziție din P1.
2. Copiază valorile din pozițiile din primul ciclu folosind primul părinte.
3. Următorul ciclu din al doilea părinte

func CX(P1, P2):C = [-1] \* lungime(P1)  
 index = 0  
 cât timp C[index] == -1:  
 C[index] = P1[index]  
 v = P2[index]  
 index = parent1.index(v)  
 index = 0  
 pentru i între (0, lungime(P2)):  
 Dacă P2[i] nu e în C:  
 cât timp C[index] != -1 și index < lungime(C):  
 index += 1  
 C[index] = P2[i]  
 returnează C

Încrucișare ordonată(OX).

1. Alege aleatoriu o parte ( i ...j) din primul părinte

1. Pentru primul descendent

2.1. Copiază partea (i ... j)

2.2. Selectează celelalte poziții astfel:

• Începând de la poziția imediat următoare lui j

• Folosind ordinea din al doilea părinte

• Continuând circular pana la poziția dinaintea lui i

1. Al doilea descendent se creează similar cu primul dar cu rolurile părinților schimbate

func OX(P1, P2, i, j):C = [-1] \* lungime(P1)  
 C[i:j] = P1[i:j]   
 index = 0  
 pentru i între (0, lungime(P2)):  
 Dacă P2[i] nu e în C:  
 cât timp C[index] != -1 și index < lungime(C):  
 index += 1  
 C[index] = P2[i]  
 returnează C

**Pentru mutație**

Aici și folosește permutarea prin inversiune și rata de mutație reprezintă șansele ca un membru din populația de copii să sufere o mutație.

1. Selectează aleatoriu 2 poziții
2. Inversează ordinea pozițiilor în segmentul astfel determinat

func mutație(C, rm):M = C  
 for c in M:  
 dacă random(0, 1) < rm:  
 i, j = random.2Point(lungime(c))  
 dacă i > j:  
 i, j = i, j  
 c[i : j] = c[j : i]  
 returnează M

Aici și folosește permutarea prin interschimbare și rata de mutație reprezintă șansele ca un membru din populația de copii să sufere o mutație.

1. Selectează aleatoriu 2 poziții
2. Inversează ordinea elementelor de pe acele pozi’ii

func mutație(C, rm):M = C  
 for c in M:  
 dacă random(0, 1) < rm:  
 i, j = random.2Point(lungime(c))   
 c[i], c[j] = c[j], c[i]  
 returnează M

**Tabele de date**

**Testele pentru algoritm au fost făcute începând de la un caz de bază pentru care am tot modificat parametri.**

**Pentru compararea cu alte variante ale algoritmului au fost schimbate doar încrucișarea și mutația pentru cazul de bază.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Problema | Nr rulări | Medie best | Medie average | Medie worst | Cel mai bun best | Mărime populație | Rata de încrucișare | Rata de mutație | Număr generații | Operator de încrucișare | Operator de mutație | Timpul mediu de execuție |
| kroC100.txp | 5 | 22536.0 | 22556.37 | 22564.6 | 21974 | 80 | 60 | 0.5 | 2000 | CX | Inversiune | 99.89 |
| 27024.0 | 27031.56 | 27034.8 | 25766 | 20 | 60 | 0.5 | 2000 | 24.87 |
| 30399.0 | 30533.78 | 30648.0 | 29696 | 80 | 20 | 0.5 | 2000 | 46.62 |
| 28540.6 | 28540.6 | 28540.6 | 28065 | 80 | 60 | 0.1 | 2000 | 99.21 |
| 30020.0 | 30311.28 | 30386.4 | 28368 | 80 | 60 | 0.5 | 800 | 39.92 |
| 44660.0 | 44700.57 | 44701.4 | 41230 | 80 | 60 | 0.5 | 2000 | Interschimbare | 100.35 |
| 22710.4 | 22809.02 | 22819.0 | 21532 | 80 | 60 | 0.5 | 2000 | OX | Inversiune | 82.53 |
| 36977.2 | 37001.22 | 37008.8 | 34116 | 80 | 60 | 0.5 | 2000 | Interschimbare | 82.63 |

**Reprezentări grafice**

**Chart

Description automatically generated**

Mărime populație: 200

Rată de încrucișare: 70

Rată de mutație: 0.2

Număr generații: 800

Operator încrucișare: încrucișare prin ciclare(cx)

Operator mutație: mutație prin interschimbare

Best: 47997

**Chart, histogram

Description automatically generated**

Mărime populație: 500

Rată de încrucișare: 70

Rată de mutație: 0.4

Număr generații: 100

Operator încrucișare: încrucișare prin ordonare(ox)

Operator mutație: mutație prin inversiune

Best: 73581

**Observații**

Toate dintre următoarele observații sunt făcute având în vedere atât rezultatele obținute anterior, rezultatul optim pentru instanța problemei și rezultatele obținute cu diferiți parametri pentru acest algoritm. Observațiile sunt:

* Timpul mediu de execuție a crescut tot mai mult cu cât oricare dintre parametri a fost mai mare;
* În cazul în care singura diferență între execuțiile algoritmului a fost numărul de generații, se poate observa că pentru un număr suficient de mare de generații se poate ajunge la rezultate mai bune;
* În cazul în care singura diferență a fost rata de mutație, se poate observa că se obțin rezultate cu mult mai bune pentru o rată de mutație mai mare
* În cazul în care singura diferență a fost rata de încrucișare, se poate observa că se obțin rezultate mai bune pentru o rată de încrucișare mai mare;
* Cele mai bune rezultate obținute au fost cele în care toți parametri au avut valori mai mari
* Comparabil cu algoritmul folosit precedent, cel mai bun rezultat obținut(21532) e mai mic decât cel obținut prin SA(20749, soluția optimă). Cu toate acestea, timpul de execuție este cu mult mai mic și rezultatele se pot datora unor parametrilor problemei;
* Pentru un număr prea de generații se poate ajunge la punctul în care algoritmul se blochează într-o anumită valoare și sunt șanse ca algoritmul să ajungă cu valori egale pentru best, avg și worst;
* Pentru o rată de mutație prea mică valorile de best average și worst ajung egale din cauză că nu se obțin rezultate destul de diversificate și rezultatele sunt mai slabe;
* S-au obținut rezultatele cele mai bune, în funcție de mediile obținute, pentru încrucișarea prin ciclare și mutația prin inversiune;
* Cel mai bun rezultat obținut a fost pentru încrucișarea prin ordonare și mutația prin inversiune;
* Mutația prin interschimbare a dat tot timpul rezultate mai slabe;
* Din reprezentările grafice se poate observa cum minimizarea soluției și cum valorile pentru best, avg și worst ajung să fie din ce în ce mai apropiate pentru un număr mai mare de generații(prima figură).

**Concluzie**

Acest algoritm a dat rezultate mult mai bune în general decât algoritmul utilizat precedent, pentru un timp mult mai scurt de execuție. Cele mai bune rezultate au fost obținute pentru populații mai mari, cu un număr mai mare de generații, cu o rată de încrucișare mai mare și o rată de mutație mai mare. Se poate observa că în funcție de fiecare operator de mutație și încrucișare se obțin rezultate foarte diferite, însă nu se poate considera că au fost testate toate cazurile posibile.