**Problema rucsacului**

**Prezentare cod( Random search + Steepest Ascent Hill Climbing ):**

Problema a fost rezolvata utilizand principiile OOP. Algoritmul este format din 3 clase, Backpack si Object, pentru a reduce codul duplicat, si clasa Service in care are loc toata logica aplicatiei.

* Primele 2 metode ale clasei au rolul de a citi si afisa continutul unui fisier





* Urmatoarele doua metode a acestei clase creeaza o solutie random, respectiv, o solutie random valida, constand intr-o lista binara

A blue and white text

Description automatically generated



Ambele avand aceeasi functionalitate, singura diferenta fiind aceea ca una dintre ele verifica validitatea solutiei(este folosita in cazul NAHC) cu ajutorul urmatoarei metode(aceasta metoda returneaza True daca solutia este valida si False in caz contrar):



* Aceasta metoda se foloseste de urm. metoda:



Care returneaza un numar intreg reprezentand greutatea obiectelor unei solutii

* Urmatoarea metoda este



Si determina fitness-ul unei solutii

* Urmatoarea metoda determina cea mai buna solutie in functie de fitness-ul maxim



* Urmatoarele doua metode





Determina fitness-ul mediu si cel mai rau fitness al vreunei solutii valide, in functie de valoarea obiectelor acelei solutii

* Urmatoarele 2 metode reprezinta inima aplicatiei, aici fiind apelate metodele mai sus mentionate





In aceste metode au loc: generari de solutii random, validarea solutiilor, determinarea calitatii solutiilor, cat si determinarea: celei mai bune solutii, solutia medie si cea mai rea solutie, tot acest proces care se afla in interiorul metodelor fiind cronometrat:

A screen shot of a computer code

Description automatically generated

Metodele pot fi apelate utilizand meniul pentru random search:

A blue background with white text

Description automatically generated

Si alegand tasta 1 pentru cautarea cu date de la tastatura, tasta 2 pentru citirea datelor din fisier, respectiv tasta 3 pentru a ne intoarce la meniul principal:

A blue background with white text

Description automatically generated

* Urmatoarele 2 metode deservesc cautarii cu ajutorul metodei de hill-climbing



Care genereaza o lista cu toti vecinii unei solutii date ca parametru. Aceasta generare are loc prin shiftarea pe rand a bitilor solutiei. Aceasta shiftare are loc cu ajutorul metodei:



* Urmatoarea metoda contine implementarea algoritmului pentru steepest hill-climbing: 

In aceasta metoda au loc urmatorii pasi:

1. Se selecteaza un punct aleator c in spatial de cautare
2. Se considera pe rand vecinii punctului c
3. Daca fitness-ul unui vecin este mai bun decat fitness-ul lui c, atunci c devine acel vecin si nu se mai continua cu restul vecinilor
4. Daca niciun vecin x al punctului c nu duce la o evaluare mai buna, se salveaza c si se continua procesul de la punctul a)
5. Dupa ce s-au epuizat evaluarile, se returneaza cel mai bun c

Generarea acestor solutii este posibila atat adaugand date manual, cat si adaugand date din fisier. Pentru a apela metodele utilizati meniul pentru SAHC din meniul principal, apand tasta 2. Urmatorul meniu vi se va deschide:

A blue background with white text

Description automatically generated

Apasand tasta 1 are cautarea cu date de la tastatura, apasand tasta 2 are loc citirea datelor din fisier, respective apasand tasta 3 pentru a ne intoarce la meniul principal.

**Rezultate algoritm:**

1. **Căutare aleatoare**

Pe setul de date din fișierul ***„rucsac-20.txt”*** s-au obtinut urmatoarele valori:

* **k** reprezintă numărul de solutii aleatoare generate.
* **Best** reprezintă fitness-ul celei mai bune soluții valide
* **Average** reprezintă fitness-ul mediu a soluțiilor valide generate
* **Worst** reprezintăfitness-ul cel mai mică dintre soluțiile valide generate
* **Timp executie** timpul necesar algoritmului pentru generării celorsoluțiilor și determinării: celei mai bune soluții, solutia medie, cea mai rea soluție
* Am ales ca valori pentru k: 10, 500 și 10000; iar pentru fiecare valoare am rulat programul de 10 ori.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare **k** | Best | Average | Worst | Timp executie(secunde) |
| 10 | 596 | 566 | 522 | 0.19178056716918945 |
|  |  |  |  |  |
| 500 | 672 | 666 | 617 | 2.861255645751953 |
|  |  |  |  |  |
| 10000 | 836 | 798 | 495 | 141.2061848640442 |

* Analizând datele din tabelul de mai sus ajungem la concluzia că diferența dintre timpul de execuție, dupa 10 rulari, pentru generarea a 10 soluții si timpul de execuție pentru generarea a 500 de soluții este de cateva secunde, într-adevăr din 500 de încercări obținem soluții mai calitative. Diferenta de timp intre 500 si 10000 de solutii generate este extrem de mare (aprox. 2 minute-pentru 10 rulari), insa diferenta dintre fitness-uri este mai mica, aproape inexistenta in cazul calitatilor medii. Acest lucru este evident datorita numarul mai mare de solutii.
* Următoarele date au fost obținute în urma rulării algoritmului pe setul de date din fișierul ***„rucsac-200.txt”.***
* Reptam pasii de mai sus, pentru aceleași valori ale lui k, avand 200 de obiecte.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare k | Best | Average | Worst | Timp executie(secunde) |
| 10 | 132306 | 132178 | 131987 | 0.260648250579834 |
|  |  |  |  |  |
| 500 | 132443 | 132238 | 131946 | 9.84912919998169 |
|  |  |  |  |  |
| 10000 | 133737 | 133542 | 132824 | 245.44471836090088 |

* Diferențele între calități sunt sesizabile odata ce numarul de repetari este mai mare, se observa ca pentru 10000 de solutii generate, calitatile sunt mult mai calitative decat calitatile obtinute pentru 10 solutii generate.
* Legat de timpul de executie, daca pentru setul de date din fisierul rucsac-20.txt diferenta era de aprox. 2 minute intre timpul pentru generarea a 10 solutii si timpul pentru generarea a 10000 de solutii, analizand setul de date din rucsac-200.txt acest lucru se schimba considerabil; diferenta de data aceasta fiind enorma, dubla fata de diferenta din cazul fisierului precedent, precizia fiind, mult mai exacta.

In continuare voi analiza un set de date introdus de la tastatura

* 5
* 6 30
* 8 45
* 5 60
* 20 50
* 15 120
* 230
* Voi analiza acest set de date pentru 5 valori diferite ale lui k.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare **k** | Best | Average | Worst | Timp executie(secunde) |
| 5 | 42 | 41 | 40 | 0.10061758358085938 |
|  |  |  |  |  |
| 25 | 42 | 42 | 41 | 0.3773672580718994 |
|  |  |  |  |  |
| 500 | 43 | 43 | 42 | 9.007820129394531 |
|  |  |  |  |  |
| 1000 | 43 | 43 | 43 | 18.66882848739624 |
|  |  |  |  |  |
| 10000 | 43 | 43 | 43 | 216.434586361132529 |

* Am ales un numar relative mic de obiecte (5) pentru a putea observa ca pentru un numar mic de obiecte se poate ajunge la o calitate aproape de cea optima cu un numar mic de solutii aleatoare generate. (cu 5 solutii generate am obtinut best 42, cu 10000 am obtinut best 43, diferenta foarte mica.)( toti timpii de executie sunt rezultatul a 10 rulari)

Next Ascent Hill-Climbing

Următoarele date au fost obținute în urma rulării algoritmului pe setul de date din fișierul ***„rucsac-20.txt”.***

Ca numar de evaluari am ales 10, 1000 si 10000 si am rulat programul de 10 ori pentru fiecare numar de evaluari ales.

|  |  |  |
| --- | --- | --- |
| Numar evaluari | Calitate solutie | Timp executie |
| 10 | 770 | 1.0093717575073242 |
|  |  |  |
| 1000 | 1003 | 5.535881757736206 |
|  |  |  |
| 10000 | 505 | 2.4636166095733643 |

Analizand datele din tabelul de mai sus ajungem la concluzia ca acest algoritm, *steepest ascent hill climbing,* depinde intr-o mare masura de solutia initiala de la care se porneste. Observam ca pentru acelasi numar de evaluari, dar pornind de la solutii diferite, diferentele intre rezulate pot fi destul de mari. (ex, 505 si 1003).

Acest algoritm ofera o solutie buna, dar se poate ca aceasta sa fie doar un maxim local, nu maximul global. In cazul in care niciun vecin direct al solutiei curente nu are o calitate mai buna decat solutia respectiva, solutia va fi returnata, chiar daca exista una mai calitativa.

Următoarele date au fost obținute în urma rulării algoritmului pe setul de date din fișierul ***„rucsac-200.txt”.***

Facem același lucru ca mai sus, pentru aceleași numar de evaluri ales, dar numărul de obiecte este mai mare de data aceasta (200).

|  |  |  |
| --- | --- | --- |
| Numar evaluari | Calitate solutie | Timp executie(secunde) |
| 10 | 36492 | 1.1418335437774658 |
|  |  |  |
| 1000 | 30856 | 2.129286766052246 |
|  |  |  |
| 10000 | 56905 | 4.411120891571045 |

Daca comparam calitatile solutiilor observam diferente mici, chiar daca numarul de evaluari este de 100x mai mare, in schimb timp de executie este de aprox 1 secunda pt. 10 evaluari, iar pentru 10000 este 4. Aceasta comparatie scoate din nou in evidenta una dintre posibilele problema pe care le putem intampina in reolvarea unei probleme de optimizare cu ajutorul algoritmului *hill climb* si anume, problema maximului local(toti timpii de executie sunt rezultatul a 10 rulari).

In continuare voi analiza un set de date introdus de mine de la tastatura:

5

6 40

7 50

5 30

12 70

20 140

210

Voi analiza acest set de date pentru 10 numere de evaluari diferite.

|  |  |  |
| --- | --- | --- |
| Numar evaluari | Calitate solutie | Timp executie(secunde) |
| 3 | 25 | 4.157868146896362 |
|  |  |  |
| 25 | 50 | 1.6199150085449219 |
|  |  |  |
| 100 | 50 | 3.0295095443725586 |
|  |  |  |
| 500 | 50 | 2.831127405166626 |
|  |  |  |
| 1000 | 50 | 3.0059735774993896 |
|  |  |  |
| 5000 | 50 | 3.1213927268981934 |
|  |  |  |
| 10000 | 50 | 2.98281192779541 |

Am ales un numar relative mic de obiecte (5) pentru a putea observa ca pentru un numar mic de obiecte se poate ajunge la o calitate aproape de cea optima cu un numar mic de evaluari. (cu 3 evaluari am obtinut 25, cu 10000 am obtinut 50, diferenta de doar 25 de unitati.)