**Problema rucsacului**

**(Tabu search)**

**Prezentare cod:**

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.

* Prima metoda a clasei are rolul de a citi 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 SAHC) 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

A group of text on a dark background

Description automatically generated

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

* Urmatoarele 2 metode deservesc cautarii celui mai bun vecin non tabu



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



* Urmatoarea metoda contine implementarea algoritmului tabu search:



In aceasta metoda au loc urmatorii pasi:

1. Generarea random a unei solutii valide
2. Cautarea celui mai bun vecin non tabu, cu ajutorul urmatoarei metode:



1. Actualizarea solutiei cu cel mai bun vecin non tabu daca acesta este mai bun decat solutia curenta
2. Se repeta acesti pasi pana un numar de iteratii este atins
3. La final returnam cea mai buna solutie

Generarea acestor solutii este posibila atat adaugand date manual(in cazul nostru aceste date sunt adaugate deja intr-o lista in interiorul aplicatiei), cat si adaugand date din fisier folosind urmatorul meniu:

A blue screen with white text

Description automatically generated

**Rezultate algoritm:**

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

* **k** reprezintă numărul de solutii generate de tabu search.
* **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. Numarul de iteratii tabu a fost 5 pentru k = 10 si 10 pentru k = 500 si k = 10000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare **k** | Best | Average | Worst | Timp executie(secunde) |
| 10 | 634 | 498 | 369 | 0.039197444915771484 |
|  |  |  |  |  |
| 500 | 674 | 512 | 402 | 6.166177749633789 |
|  |  |  |  |  |
| 10000 | 681 | 531 | 353 | 104.42734122276306 |

Analizând datele din tabelul de mai sus ajungem la concluzia că diferența dintre timpul de execuție pentru generarea a 10 soluții si timpul de execuție pentru generarea a 500 de soluții este de câteva secunde, diferența de timp între 500 și 10000 de soluții generate este de aproape 2 minute,insa ce observam este ca, desi timpul de executie pentru generarea a 10000 de solutii este mai mare am obtinut un best nu cu mult mai bun, average-ul este intr-adevar mai bun.

Daca comparam aceste rezultate cu cele obtinute la cautarea aleatoare, observam ca obtinem solutii mai calitative folosind tabu search dar timpul de executie este mai mare.

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

Repetam pasii de mai sus, pentru aceleași valori ale lui k, avand 200 de obiecte.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare k | Best | Average | Worst | Timp executie(secunde) |
| 10 | 131313 | 130442 | 128490 | 1. 91295862197875 |
|  |  |  |  |  |
| 50 | 131007 | 129374 | 126759 | 11.008623123168945 |
|  |  |  |  |  |
| 100 | 131547 | 129978 | 128151 | 25.70676350593567 |

Diferențele între calitățile soluțiilor sunt destul de mici. Doar dacă comparăm best-ul lui 100 cu best-ul lui 50 observam o solutie nu cu mult mai calitativa pentru 2500.

Legat de timpul de execuție, dacă pentru setul de date din rucsac-20.txt diferenta era de aprox. 100 secunde 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.

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

10

7 20

15 50

20 140

14 81

5 50

13 100

11 25

20 45

21 73

1 10

489

Voi analiza acest set de date pentru 5 valori diferite ale lui k. De fiecare dată numărul de iterații tabu a fost 5. Pentru fiecare valoare a lui k, algoritmul a fost rulat de 10 ori.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valoare **k** | Best | Average | Worst | Timp executie(secunde) |
| 5 | 106 | 98 | 89 | 0.005010843276977539 |
|  |  |  |  |  |
| 25 | 108 | 103 | 97 | 0.05089378356933594 |
|  |  |  |  |  |
| 500 | 109 | 106 | 100 | 0.7785868644714355 |
|  |  |  |  |  |
| 1000 | 109 | 105 | 100 | 1.589717149734497 |
|  |  |  |  |  |
| 10000 | 109 | 107 | 102 | 15.270273923873901 |

Am ales un numar relativ mic de obiecte (10) pentru a putea observa ca pentru un numar mic de obiecte se poate ajunge la calitatea optima cu un numar mic de solutii generate. (cu 10 ca numar maxim de evaluari, am obtinut acelasi best ca la 10000 de evaluari)

Același lucru s-a întâmplat în acest caz, cu un numar relativ mic de obiecte și la căutarea aleatoare. Cu un numar mic de iteratii am ajuns la aproape acelasi best ca si cu un numar mare de iteratii.