Problema rucsacului

Berar George 30641

Descrierea problemei

Se da o multime formata din N obiecte, fiecare fiind caracterizat de o greutate si o valoare, si un rucsac avand o greutate maxima admisa G. Sa se gaseasca o submultime de obiecte astfel incat valoarea totala sa fie maxima, iar suma greutatilor lor sa nu depaseasca greutatea rucsacului.

Prezentarea algoritmilor

1. **Metoda exhaustiva**

Aceasta metoda reprezinta o abordare directa a problemei in care se incearca toate combinatiile posibile din spatiul de cautare pentru a gasi cea mai buna solutie. Deoarece problema rucsacului are n obiecte, iar un obiect poate fi ales (valoare 1) sau nu (valoare 0), spatiul de cautare rezultat este 2^n. Pentru 20 de obiecte spatiul de cautare rezultat este 2^20, ceea ce implica verificarea multor posibilitati valide.

**ALGORITHM BruteForce(items[1..N], G)**

// items – array containing the item objects with associated weight and value

// G – the knapsack capacity

**A = [1..N]**

**bestValue = 0**

**bestChoice = []**

**for i = 0 to 2^n do**

**j = n**

**cumulatedWeight = 0**

**cumulatedValue = 0**

**while( A[j] != 0 and j > 0 )**

**A[j] = 0**

**j= j – 1**

**A[j] = 1**

**for k = 1 to n do**

**if( A[k] = 1 ) then**

**cumulatedWeight = cumulatedWeight + items[k].weight**

**cumulatedValue = cumulatedValue + items[k].value**

**if( (cumulatedValue > bestValue) AND (cumulatedWeight <= G)) then**

**bestValue = cumulatedValue**

**bestWeight = cumulatedWeight**

**bestChoice = A**

Parametri de intrare ai algoritmului ii reprezinta lista cu obiectele care trebuie puse in rucsac si greutatea rucsacului admisa. ***A***reprezinta un vector avand dimensiunea egala cu numarul de obiecte, si are pe fiecare pozitie valoarea 0, pentru inceput. Acest vector va fi folosit pentru a genera toate combinatiile, sub forma binara, 1 – obiect ales, 0 – obiectul nu este ales. ***bestValue***si ***bestChoice***reprezinta variabilele, care, in final, vor reprezenta cea mai buna solutie si anume, cea mai buna valoarea totala a solutiei obtinute si reprezentarea binara a acesteia.

Deoarece spatiul de cautare are valoarea 2^n , algoritmul impune parcurgerea tuturor combinatiilor, si anume verificarea tuturor 2^n solutii posibile. Am precizat mai sus ca vectorul ***A*** va fi folosit pentru generearea fiecarei solutii. In continuare vom prezenta, pentru 5 obiecte, cum se intampla acest lucru.

Initial, vectorul A are pe toate pozitiile valoarea 0.

A = [0,0,0,0,0]

Generarea fiecarei solutii in parte presupune doar schimbarea valorii pozitiilor din 1 in 0 sau invers.

Se verifica tot timpul daca ultima pozitie a vectorului are valoarea 1.

* Daca da, atunci se schimba valoarea in 0 si se decrementeaza pozitia din vector la n – 1. Se verifica din nou daca pozitia curenta (n-1) are valoarea 1 si se repeta aceeasi operatie pana cand conditia ca valoarea pozitiei curenta sa fie 1 nu mai este indeplinita.
* Daca nu, se initializeaza aceasta pozitie cu valoarea 1

Ex.

A = [0 0 0 0 0]

A[5] == 0 ceea ce determina initializarea A[5] = 1

Acuma A = [0 0 0 0 1]

La pasul urmator , A[5] == 1 deci suntem in primul caz

A[5] = 0

A[4] == 0 deci parasim primul caz

A[4] = 1

Acum A = [0 0 0 1 0]

La pasul urmator A[5] == 0 deci A[5] = 1

Acum A = [0 0 0 1 1]

La pasul urmator A[5] == 1 deci suntem in primul caz

A[5] = 0

A[4] == 1 deci A[4] = 0

A[3] == 0 parasim primul caz

A[3] = 1

Acum A = [0 0 1 0 0 ]

La urmatoarea A va fi [0 0 1 0 1] si tot asa pana cand se ajunge la [1 1 1 1 1]

1. **Metoda Random-Search**

Aceasta metoda se bazeaza pe generarea unei solutii binare aleatoare, valida. Daca la algoritmul precedent se cautau solutiile in tot spatiul de cautare, aici nu se mai intampla acest lucru. Se va genera doar un numar de K solutii valide. Este mai rapid, dar cea mai buna solutie obtinuta nu reprezinta intotdeauna solutia globala cea mai buna.

**ALGORITHM RandomSearch(items[1..N], G, K)**

// items - array containing the item objects with associated weight and value

// G – the knapsack capacity

// K – the number of iterations

**randomSolutionValue = 0**

**generatedRandomArray = [1..N]** // initial only 0s

**bestSolutionValue = 0**

**bestSolution = [1..N]** // initial only 0s

**for i to K do**

**generatedRandomArray = generateRandomSolution(G, items)**

**randomSolutionValue = evaluateValueOf(generatedRandomArray)**

**if( randomSolutionValue > bestSolutionValue ) then**

**bestSolutionValue = randomSolutionValue**

**bestSolution = generatedRandomArray**

Parametri de intrare ai algoritmului ii reprezinta lista cu obiectele care trebuie puse in rucsac, greutatea rucsacului admisa si numarul de iteratii reprezentand numarul de solutii valide care trebuie generat. ***generatedRandomArray*** reprezinta un vector cu doar 2 valori posibile, 1 – obiect ales, 0 – obiectul nu este ales. La fiecare iteratie se genereaza o combinatie de 1 si 0 a acestui vector. De exemplu , pentru 5 obiecte, vectorul aleator generat ar putea fi 0 1 1 0 0. Desigur, acest vector trebuie sa respecte restrictia impusa de greutatea rucsacului, iar greutatea totala a obiectelor de pe pozitiile aferente care contin valoarea 1 sa nu depaseasca aceasta greutate admisa.

Calitatea celei mai bune solutii gasite depinde in mare masura de numarul de iteratii. Am determinat ca un numar optim de iteratii, pentru care raportul (timp de calcul/valoare obtinuta) reprezinta o imbunatatire, este 10000.

1. **Metoda Greedy**

Aceasta metoda este folosita pentru optimizare. Foloseste un set de euristici pentru a genera o solutie care ar putea reprezenta solutia optima. Tipurile de euristici posibile aplicabile in contextul problemei rucsacului sunt urmatoarele:

* Alege de fiecare data, dintre obiectele ramase, obiectul cu valoarea cea mai mare; aceasta euristica urmareste maximizarea valorii totale a obiectelor din rucsac
* Alege de fiecare data, dintre obiectele ramase, obiectul care are greutatea cea mai mica; aceasta euristica urmareste adaugarea a cat mai multe obiecte in rucsac
* Alege de fiecare data, dintre obiectele ramase, obiectul care are cel mai bun raport intre greutate si valoare

1. Highest Value

**Sort the items in non-increasing order of the value**

**for all sorted items do**

**if the current item on the list fits into the knapsack then**

**place it in the knapsack**

**else**

**proceed to the next one**

1. Min Weight

**Sort the items in increasing order of the weight**

**for all sorted items do**

**if the current item on the list fits into the knapsack then**

**place it in the knapsack**

**else**

**proceed to the next one**

1. Value to Weight Ratio

**Compute the value-to-weight ratios ri = vi / wi , i = 1, …, N, for the items given**

**Sort the items in non-increasing order of the value-to-weight ratios**

**for all sorted items do**

**if the current item on the list fits into the knapsack then**

**place it in the knapsack**

**else**

**proceed to the next one**

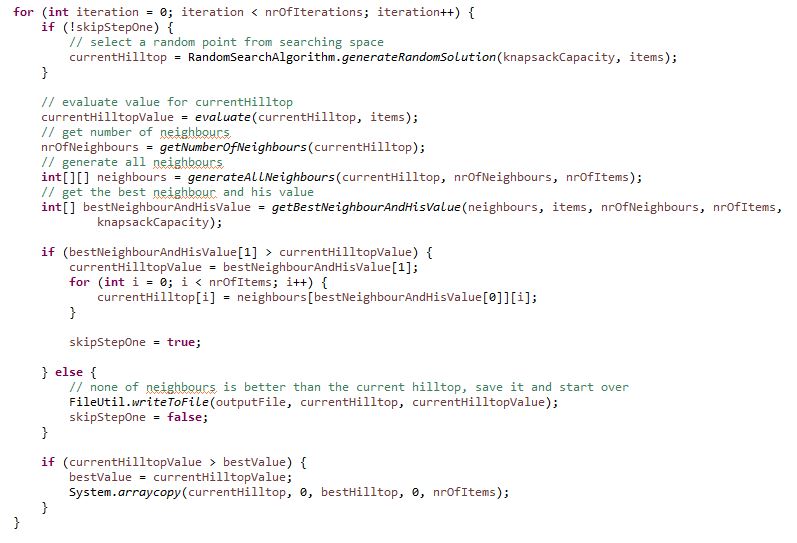
1. **Metoda Steepest Ascent Hill-Climbing**

Aceasta metoda reprezinta o varianta de implementare a algoritmului Hill-Climbing, alaturi de Random Hill-Climbing si Next Ascent Hill-Climbing. Metoda porneste cu o solutie valida aleatoare dupa care incearca sa gaseasca urmatoarea solutie valida, aflata in vecinatate, care este mai buna decat cea curenta. In acest fel, incearca sa ajunga la o solutie globala, la fel ca un alpinist care incearca sa ajunga la varf, reprezentand maximul global. Problema la aceasta abordare o reprezinta faptul ca algoritmul nu gaseste intotdeauna maximul global, ci mai degraba un maxim local, deoarece punctul de start este unul aleator, si se calculeaza pe parcurs urmatoarea valoare din vecinatate.

In continuare prezentam pasii algoritmului:

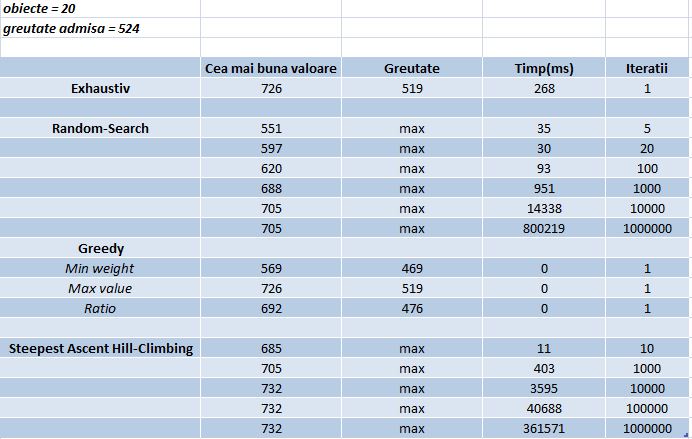
1. Se selecteaza un punct aleator **c** (o solutie valida) in spatiul de cautare
2. Se determina toate punctele din vecinatatea acestui punct
3. Daca oricare vecin **x** are o valoare mai buna decat **c** atunci **c = x**
4. Daca nici un vecin nu are o valoare mai buna, atunci se salvaza **c** si se trece la pasul 1
5. Dupa un numar maxim de evaluari, se returneaza cel mai bun punct **c**

La fel ca si la Random Search, calitatea celei mai bune solutii gasite depinde in mare masura de numarul de iteratii sau evaluari cum este descris in pasul 5. Am determinat ca un numar optim de iteratii, pentru care raportul (timp de calcul/valoare obtinuta) reprezinta o imbunatatire, este 10000. Mai jos este prezentata portiunea de cod, scrisa in Java, care realizeaza toti pasii de mai sus pentru a gasi solutia.

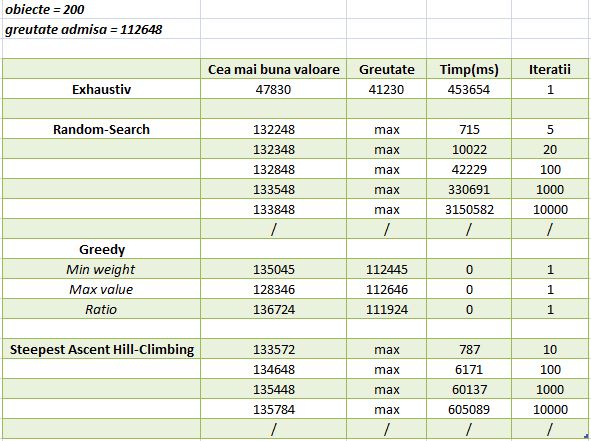


Rezultate experimentale

Tabel 1 - Rezultate experimentale pentru 20 de obiecte

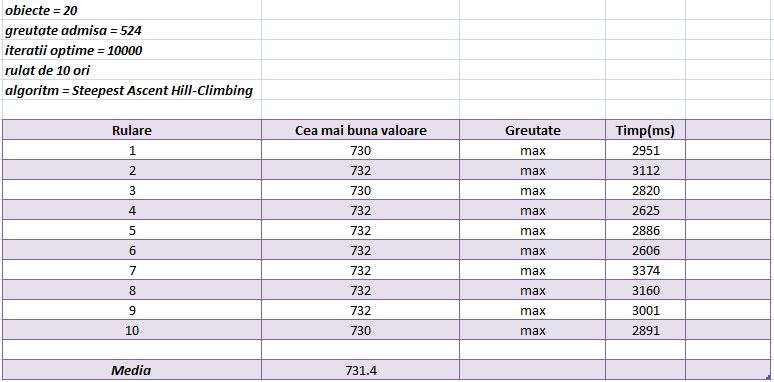


Tabel 2 - Rezultate experimentale pentru 200 de obiecte





Tabel 3 - Rezultate experimentale pentru 20 de obiecte cu numar optim de iteratii



Tabel 4- REZULTATE EXPERIMENTALE PENTRU 20 DE OBIECTE CU NUMAR OPTIM DE ITERATII