Problema rucsacului

Prepared for: Camelia Chira, PhD
Prepared by: Razvan-Stefan Grecu
28 February 2016
Group: TI 30643

REFERAT

Descrierea problemei

Problema rucsacului este o problema de optimizare. Avem un rucsac si o colectie de obiecte ce dorim sa le punem in rucsac. Obiectele au urmatoarele proprietati: o greutate, pe care o vom nota cu w_i si o valoare, notata cu v_i. Rucsacul are o capacitate maxima W care nu trebuie depasita. Problema cere gasirea unei combinatii de obiecte care pot fi puse in rucsac, avand valoarea maxima intre combinatiile candidate.

Pentru rezolvarea acestei probleme, vom presupune ca atat valoarea, dar mai ales greutatea sunt numere intregi si ca nu vom putea pune parti din obiecte ci doar in intregime.

Prezentarea algoritmului

Exista mai multe metode de rezolvare a acestei probleme, de la reprezentarea sub forma de graf, la programare dinamica si altele. In contextul acestui laborator, au fost cerute trei metode de rezolvare a problemei, si acestea sunt: cautare exhaustiva/enumerativa, cautare aleatorie si cautare greedy.

Incepem cu primul algoritm, si anume cel de **cautare enumerativa** in spatiul solutiilor. Acesta este algoritmul naiv, cu timp de rulare exponential, dar garanteaza solutia optima. Dupa nume, acest algoritm va incerca fiecare combinatie posibila de obiecte, le va valida, apoi va alege cel mai bun rezultat. Pentru a face asta, trebuie generata fiecare secventa posibila de obiecte. Vom crea urmatoarea notatie: x_i cu urmatoarea semnificatie: daca obiectul i este pus in rucsac atunci x_i va fi 1, altfel va fi 0. Apoi, trebuie sa pastram constrangerea problemei si anume ca suma greutatilor alese sa fie mai mica sau egala decat capacitatea rucsacului. In final, sortam solutiile valide dupa suma valorilor obiectelor si alegem rezultatul cu valoarea maxima. Pseudocod:

```
var candidateSolutions = generateBinaryStrings(items.count)
var knapsacks = createNKnapsacks(candidateSolutions.count)
for (candidateSolution in candidateSolutions) DO
        for (currentElement in candidateSolution) DO
                if (currentElement == “1”) DO
                        Item item = items.getAtIndexOf(currentElement)
                        knapsack.getAtIndexOf(candidateSolution).addItem(item)
knapsacks.sort()
return knapsacks.getAllValid().first()
```

Functia *generateBinaryStrings(int number)* genereaza toate sirurile de caractere binare de lungime *number*. Aceasta este functia cu cel mai mare consum de resurse, din cauza numerelor uriase si a volumului de informatii ce trebuie servit. *Knapsackul* este modelul utilizat pentru reprezentarea unui rucsac cu obiecte; acesta are cateva caracteristici, ce ne ajuta sa validam o solutie - greutate maxima, greutate curenta si valoare totala. Astfel, intre pseudocod si implementare, nu exista o diferenta sesizabila, daca excludem unele caracteristici ale limbajelor utilizate. Nu este recomandata utilizarea acestei solutii pentru numere mai mari de 23-24, timpul de executie fiind mare.

Sortarea se face in functie de valoare, in mod descrescator, iar validarea se face prin verificarea acestei egalitati: `knapsack.currentWeight <= knapsack.maxWeight`

Pentru aceasta solutie, nu avem nevoie de parametri speciali, in afara de greutatea rucsacului.

A doua metoda ceruta de rezolvare a acestei este **cautarea aleatorie**. Aceasta este o metoda de cautare stohastica, ce nu ofera mereu solutia optima. Acest algoritm va genera o secventa binara aleatorie de lungimea numarului obiectelor. Apoi acesta va genera rucsacul cu obiectele “alese”. Aceste operatii trebuie rulate de un numar de ori, specificat de un parametru. In abordarea mea, am ales ca si acel parametru sa fie generat de alt parametru, dat la rulare. Parametrul este de tip double, cu valori acceptate intre [0,1], ce reprezinta gradul de exactitate, sub forma 1 (garantat sa fie corect) - epsilon (valoarea parametrului). Formula ce genereaza numarul de generari aleatorii este dat de urmatoarea formula:

$$| (n^2 - random(n^2 - n - c1, n^2 - n + c2) * (1 - epsilon)) | / epsilon$$

unde n este numarul de obiecte, iar c1 si c2 sunt constante arbitrare. Astfel, se garanteaza mai multe rulari daca epsilon este mic, si mai putine daca epsilon este mare. Cazurile speciale sunt 0 si 1, la 0 realizandu-se 2^n rulari, oferind o probabilitate destul de mare pentru obtinerea rezultatului optim. La valoarea 1 se genereaza un singur sir si acela este rezultatul. Pseudocod:

```
var numberOfIterations = getNumberOfIterations(epsilon)
var samples = { }
for (i in 1 to numberOfIterations) DO
    var candidate = generateRandomSequence(items.count)
    var sample = createSampleKnapsackFromCandidate(candidate)
    samples.add(sample)
return getBestSamples(samples)
```

Functia *getNumberOfIterations* utilizeaza formula de mai sus pentru a genera numarul de rulari in functie de parametrul epsilon. Functia *generateRandomSequence* este similara celei de la cautarea exhaustiva,

dar in acest caz va genera doar o singura secventa. Functia *createKnapsackFromCandidate* este, din nou, similara celei de la cautare exhaustiva, convertind sirul binar intr-un rucsac. Functia *getBestSamples* face partea de validare, iar in cazul valorilor intermediare ale lui epsilon, ofera, pe langa cel mai bun candidat din piscina de valori, cel mai des aparut candidat din executie. Parametrii ce trebuie luati in considerare aici sunt greutatea rucsacului si valoarea lui epsilon.

Solutia numarul 3 reprezinta o **cautare greedy.** Nici aceasta solutie nu este mereu optima, dar beneficiile sale sunt viteza de executie, si rezultatele sale, preponderent optime. Trebuie gasita o metrica buna pentru stabilirea caror obiecte vor fi puse in ghiozdan, astfel incat sa fie respectate constrangerile problemelor.

Orice metrica am alege, aceasta nu ne va da o garantie de optimalitate mereu - diferite metrici fiind mai bune sau mai rele pentru diferite instante de date. Pentru aceasta problema, am ales varianta clasica de raport intre valoare si greutate. Astfel, solutia consta in sortarea descrescatoare a elementelor dupa raportul dintre valoarea lor si greutatea lor. Pasul doi consta in umplerea rucsacului cu elemente, pana cand acestea nu mai incap. Ultimul pas este unul decizional: daca exista un element singular cu valoare mai mare decat suma valorilor primelor k elemente, atunci il vom pune pe acela in ghiozdan, altfel raman primele k elemente. Acest pas asigura o corectitudine cel putin egala cu 50% in toate cazurile. Complexitatea va varia in functie de algoritmul de sortare folosit, in general fiind **O(n*ln n)**. Pseudocodul este:

```
var knapsack = new Knapsack()
var sortedItems = sortItemsByValueAndWeight(items)
for (item in sortedItems) DO
    if (item.weight < knapsack.maxWeight - knapsack.currentWeight) DO
        knapsack.add(item)
var maxValuedItem = getMaxValuedItem(items)
if (knapsack.totalValue < maxValuedItem.value && maxValuedItem.weight < knapsack.weight) DO
    clearKnapsack(knapsack)
    knapsack.add(maxValuedItem)
```

Singurul parametru important este greutatea rucsacului. Algoritmul de sortare este cel implicit folosit in java, asigura **O(n*ln n).**

Rezultate exprimentale

Solutiile au fost rulate pe instante avand 10,15,20,30,50,100,200, 1000 si respectiv 50000 de elemente. Cautarea exhaustiva a fost folosita doar pe primele trei din cauza timpului de rulare necesar. Cautarea aleatorie si cea greedy au fost utilizate pe toate instantele.

Pentru primul fisier, parametrii sunt acestia: greutatea maxima admisa de rucsac - 100, greutatea maxima a unui obiect este 50 si valoarea maxima, 30. Pentru epsilonul cautarii aleatoare am dat valoarea 0.1, asigurand 90% (nu chiar garantat, prin numarul de incercari) corectitudine.
Datele sunt urmatoarele:

Item number	Item weight	Item value
1	43	20
2	49	28
3	26	27
4	37	9
5	49	2
6	28	15
7	41	22
8	13	26
9	10	6
10	29	4

In aceasta situatie, toate trei cautarile au oferit acelasi raspuns, prin urmare, cel optim:

Total Value	Weight	Max weight
87	98	100

In cazul cautarii aleatorii, am generat si rezultatul cu cea mai deasa aparitie:

Total Value	Weight	Max weight
68	97	100

Considerand cantitatea obiectelor din fisierele urmatoare, se recomanda consultarea fisierelor propriu-zise, in acest referat fiind in continuare trecute doar rezultatele si valorile parametrilor utilizati.

Cazul cu 20 de obiecte, are capacitatea rucsacului ca 125, greutatea maxima a unui obiect 50 si valoarea maxima 25. Pentru cautarea aleatorie, am folosit 0.9 pentru epsilon. Atat cautarea enumerativa cat si cautarea greedy au dat acelasi rezultat:

Total value	Weight	Max weight
139	121	125

Pe cand, cautarea aleatorie a oferit un raspuns mai putin optim (acesta fiind si cel mai des in piscina de valori) :

Total value	Weight	Max weight
107	124	125

Pentru cazul de test cu 100 de obiecte, capacitatea rucsacului este de 750, greutatea maxima a unui obiect este 150 si valoarea maxima 250. Pentru epsilon am folosit 0.5. Cautarea greedy ofera urmatorul rezultat:

Total value	Weight	Max weight
5303	734	750

Cautarea aleatorie nu a produs nici un rezultat valid, la cateva incercari. Marind capacitatea ghiozdanului la 2500, am obtinut insa urmatorul rezultat:

Total value	Weight	Max weight
6309	2477	2500

Cel mai des aparut element este:

Total value	Weight	Max weight
5574	2471	2500

Pentru ultimul test, cu 50000 de elemente, am avut 15000 ca si capacitatea rucsacului, greutatea maxima a unui obiect 250 si valoarea maxima 175. Epsilon a fost 0.8, insa timpul de executie a fost extrem de lung. Rezultatele cautarii greedy sunt urmatoarele - considerand cantitatea de obiecte, o cautare enumerativa ar fi durat mult mai mult timp, iar rezultatul obtinut este satisfacator:

Total value	Weight	Max weight
257776	14999	15000

Comparatii

In mod evident, cand avem o problema, dorim solutia optima - cea mai buna; insa, de multe ori, din cauza cantitatii datelor, cea mai buna solutie ofera cele mai putin practice rezultate, din punct de vedere al timpului sau al resurselor folosite. Astfel, recurgem la euristici sau la modificari ale modelului matematic dat pentru relaxarea problemei, astfel incat sa putem obtine rezultate foarte similare cu cele oferite de o solutie optima, daca nu identice, intr-un timp mult mai mic.

Solutia prin cautarea enumerativa functioneaza si returneaza solutia optima, dar, intr-un timp rezonabil doar pana la 24-25 de obiecte. Solutiile obtinue prin cautare aleatorie pot de multe ori sa ofere o solutie mai putin satisfacatoare, la un numar mare de obiecte, daca numarul de rulari nu este potrivit. Solutia greedy propusa functioneaza in cel putin 50% din cazuri, intr-un timp rezonabil, chiar si la un numar mare de obiecte. Acest procentaj, chiar daca pare mic, este foarte mare, iar pentru multe intrari este foarte aproape de solutia optima (100%).

Ca solutii alternative, se pot aduce in discutie metode folosind programare dinamica, folosind o metrica similara celei utilizate la cautarea aleatorie, oferind 1-epsilon siguranta ca raspunsul este cel optim, sau grafuri, nodurile continand obiectul si greutatea curenta, iar muchiile fiind notate cu valoarea obiectului.