**Problema rucsacului 1/0. Rezolvare prin metoda branch and bound**

Fiind date n obiecte fiecare cu o greutate si o valoare cunoscuta si un rucsac de capacitate W determinati setul de obiecte care pot fi incarcate in rucasac astfel incat sa se maximizeze valoarea lor.

Datele initiale ale problemei:

N = 4, W = 16

|  |  |  |  |
| --- | --- | --- | --- |
| i | value[i] | weight[i] | Value[i]/weight[i] |
| 1 | 40 | 2 | 20 |
| 2 | 30 | 5 | 6 |
| 3 | 50 | 10 | 5 |
| 4 | 10 | 5 | 2 |

Obiectele au fost ordonate descrescator dupa rapotul value[i]/weight[i] (valoarea pe unitate de greutate).

Pentru a determina daca un nod din arborele decizional este promitator vom initializa variabilele *totalweight* si *bound* cugreutatea si valoarea obiectelor deja incluse in rucasac si vom adauga apoi obiecte pana cand dam de un obiect a carui greutate ar face ca *totalweight* sa depaseasca W.

Pentru a calcula limita maxima a valorii luam o fractiune din acest obiect, atat cat permite spatiul ramas in rucsac si adaugam valoarea acestei fractiuni la valoarea variabilei *bound*. In acest fel *bound* devine o limita maxima a valorii ce poate fi obtinuta expandand acest nod.

Daca nodul pe care il expandam este la nivelul *i* iar nodul care ar duce la depasirea greutatii W este la nivelul *k* atunci avem:

Si

Un nod nu este promitator daca aceasta limita este mai mica decat *maxvalue*, adica valoarea celei mai bune solutii gasita pana la acest punct.

Vom adopta o strategie de tipul Best-First de vizitare a nodurilor arborelui de decizie. Dupa vizitarea tuturor copiilor unui nod, dintre nodurile promitatoare vom vizita nodul cu cea mai mare limita a valorii. In acest fel putem ajunge la solutia optimala mai repede decat daca am vizita toate nodurile promitatoare intr-o ordine oarecare.

In figura de mai jos avem reprezentat arborele de decizie iar pentru fiecare nod avem specificate *value*, *weight* si *bound*. Ne vom referi la un nod prin nivelul lui (level) si pozitia sa (din stanga) in arbore.

(0,0)

0

0

115

Valoarea asociata nodului

Greutatea asociata nodului

Limita (bound) asociata nodului

Nivel (index)

Numarul nodului pe un nivel

(0,0)

0

0

115

(1,2)

0

0

82

(1,1)

40

2

115

(2,1)

70

7

115

(2,2)

40

2

98

(3,1)

120

17

0

(3,2)

70

7

80

(3,3)

90

12

98

(3,4)

40

2

50

(4,1)

100

17

0



(4,1)

90

12

90

Obiectul1(40,2)

Obiectul2(30,5)

Obiectul3(50,10)

Obiectul4(10,5)

Pasii algoritmului:

1. Vizitam nodul (0, 0) (radacina).
2. Setam *value* si *weight* la 0 and 0.
3. Calculam *bound* si obtinem 115 ( = 40 + 30 + 5 \* (16-7).
4. Setam maxvalue la 0.
5. Vizitam nodul (1, 1).
6. Caculam *value* si *weight* : 40 respectiv 2.
7. Deoarece 2 < 16 iar 40 > 0 setam maxvalue = 40.
8. Calculam *bound* = 115.
9. Vizitam nodul (1, 2).
10. Calculam *value* = 0 si *weight* =0.
11. Calculam *bound* = 82 (=30 + 50 + 2 \* (16 – 15)).
12. Determinam nodul promitator cu cel mai mare *bound*
13. Alegem nodul (1, 1) deoarece 115 > 82 si il vom expanda la pasul urmator.
14. Vizitam nodul (2, 1).
15. Calculam *value* = 70, *weight* = 7.
16. Deaorece 7 < 16, si 70 > 40 setam maxvalue = 70.
17. Calculam *bound* = 115.
18. Vizitam nodul (2, 2).
19. Calculam *value* = 40, *weight* = 2.
20. Calculam *bound* = 98 ( = 40 + 50 + (16-2)\*2).
21. Alegem nodul promitator cu cel mai mare *bound*.
22. Acesta este nodul (2, 1). Il vom expanda la pasul urmator.
23. Vizitam nodul (3, 1).
24. Calculam value=120, weight=17.
25. Acesta nu este admisibil deoarece 17 > 16 Setam *bound* =0.
26. Vizitam nodul (3, 2).
27. Calculam *value*=70, *weight*=7.
28. Calculam *bound* = 80.
29. Determinam nodul promitator cu cel mai mare *bound*.
30. Acesta este nodul (2, 2). Il vom expanda la pasul urmator.
31. Vizitam nodul (3, 3).
32. Calculam *value* = 90, *weight* = 12.
33. Deoarece 12 < 16 si 90 > 70 , actualizam maxvalue = 90.
34. In acest moment nodurile (1, 2) si (3, 2) devin nepromitatoare deoarece valoarea *bound* (82 respectiv 80) este mai mica decat 90.
35. Calculam *bound* = 98 (pentru nodul (3,3).
36. Vizitam nodul (3, 4).
37. Calculam *value* = 40, *weight* = 2.
38. Calculam *bound* = 50.
39. Determinam ca nu este nod promitator deoarece 50 < 90 (valaorea actuala a lui maxvalue).
40. Determinam nodul promitator cu cea mai mare valoarea a lui *bound*.
41. Acesta este nodul (3, 3). Il vom expanda la pasul urmator.
42. Vizitam nodul (4, 1).
43. Calculam *value* = 100, *weight* = 17.
44. Determinam ca nu este nod promitator deoarece 17 > 16. Setam *bound* = 0.
45. Vizitam nodul (4, 2).
46. Calculam *value* = 90, *weight* =12.
47. Calculam *bound* = 90.
48. Determinam ca nu este nod promitator deoarece 90 <= 90 (valoarea curenta a lui maxvalue).

Observatie: chiar daca la fiecare pas expandam nodul cu cea mai mare valoare a lui *bound*, nu avem totusi nici o garantie ca acesta va conduce la solutia optimala. In exemplul nostru nodul (2,1) pare a fi mai bun decat nodul (2,2) dar totusi nodul (2,2) a condus la solutia optima.

Mai jos este prezentat pseudocodul pentru algoritmul branch and bound aplicat in cazul problemei rucsacului. Vom mentine lista nodurilor promitatoare intr-o structura de tip coada. Pentru a extrage din aceasta coada nodul cu valoarea bound cea mai mare (strategia best-first), elementele din coada vor fi sortate in ordine descrescatoare dupa valoarea bound.

void best\_first\_branch\_and\_bound (state\_space\_tree T, number& best) {

queue Q;

node u, v;

initialize (Q); // Initializam Q (coada goala)

v = root of T;

best = value(v);

insert (Q, v);

while (! isEmpty (Q)) {

remove (Q, v); // extragem nodul cu cea mai mare valoare bound

if (bound (v) is better than best) // Verificam daca nodul este promitator

for (each child u of v) {

if (value(u) is better than best)

best = value (u);

if (bound (u) is better than best)

insert (Q, u);

}

}

}

Un nod in arborele solutiilor va avea structura:

struct node {

int level;

int value;

int weight;

float bound;

};

Aici, *level* este nivelul nodului in arbore (totodata, indexul nodului in vectorul cu obiecte), *value* este valoarea totala a obiectelor incarcate in rucsac inclusiv obiectul din nodul curent, *weight* greutatea totala a obiectelor incarcate in rucsac inclusiv obiectul din nodul curent iar *bound* este limita maxima a valorii ce se poate obtine expandand nodul curent.

Functia *knapsack* de mai jos implementeaza algoritmul prezentat.

void knapsack(int n, int value[], int weight[],int W) {

Queue Q;

int maxvalue;

node u, v;

Initialize(Q);

v.level = 0;

v.value = 0;

v.weight = 0;

v.bound = bound(v); // nodul v va fi radacina arborelui

maxvalue = 0;

insert(Q, v);

while (! isEmpty(Q)){

remove(Q, v); //extrag nodul cu cel mai mare bound

if (v.bound > maxvalue) { // verific daca este nod promitator

u.level = v.level + 1; // nodul u va fi nodul copil care include

// urmatorul obiect

u.weight = v.weight + w[u.level];

u.value = v.value + value[u.level];

if (u.weight <= W && u.value > maxvalue)

maxvalue = u.value;

u.bound = bound(u);

if (u.bound > maxvalue)

insert(Q, u);

u.weight = v.weight; // nodul u va fi nodul copil care NU include

// urmatorul obiect

u.value = v.value;

u.bound = bound(u);

if (u.bound > maxvalue)

insert(Q, u);

}

}

return maxvalue;

}

Functia *bound(node u)* care calculeaza limita maxima a valorii ce se poate obtine plecand de la nodul u este:

float bound (node u){

index j, k;

int totweight;

float result;

if (u.weight >= W)

return 0;

else {

result = u.value;

j = u.level + 1;

totweight = u.weight;

while (j <= n && totweight + weight[j] <= W){

totweight = totweight + w[j];

result = result + value[j];

j++;

}

k = j;

if (k <=n)

result = result + (W - totweight) \* value[k] /weight[k];

return result;

}

}