Capitolul 7

Tehnica alegerii local optimale

O clasă de probleme frecvent întâlnită în practică este cea a problemelor de *optimizare* de tipul:

> *Să se determine $x \in X$ astfel încât: (i) x satisface anumite condiţii (restricţii); (ii) x optimizează (minimizează sau maximizează) un criteriu.*

O subclasă importantă în informatică o reprezintă cea în care X este o mulţime finită, problema de optimizare fiind numită în acest caz *discretă* sau *combinatorială*. La o primă vedere problema pare foarte simplă întrucât este suficient să se parcurgă X şi să se aleagă elementul care satisface restricţiile şi optimizează criteriul. O astfel de abordare bazată pe metoda forţei brute devine ineficientă (sau chiar impracticabilă) atunci când numărul de elemente din X este foarte mare.

Exemplul 7.1 (*problema submulţimii de sumă dată şi cardinal minim*) Se consideră mulţimea $A = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{R}$ şi valoarea $C \leq \sum_{i=1}^{n} a_i$. Se cere să se determine (dacă există) o submulţime având cât mai puţine elemente şi a căror sumă să fie egală cu C.

În acest caz X reprezintă ansamblul submulţimilor lui A. O abordare prin metoda forţei brute a unei astfel de probleme necesită generarea tuturor submulţimilor lui A, reţinerea celor care satisfac restricţia (suma elementelor egală cu C) şi alegerea celei (celor) ce au cele mai puţine elemente. Un astfel de algoritm ar avea ordinul de complexitate $\mathcal{O}(2^n)$.

Pentru a rezolva mai eficient probleme de acest tip au fost dezvoltate tehnici specifice. Două dintre acestea sunt tehnica alegerii local optimale ("greedy" sau alegere lacomă) şi cea a programării dinamice.

7.1 Principiul tehnicii

Să considerăm problema de optimizare formulată într-o variantă uşor diferită:

> *Fie $A = (a_1, a_2, \ldots, a_n)$ un set finit de elemente (nu neapărat distincte). Să se determine $S = (s_1, s_2, \ldots s_k) \subset A$ astfel încât S să satisfacă anumite restricţii şi să optimizeze un criteriu.*

În continuare prin set vom înţelege o mulţime în sens general (elementele sale nu sunt neapărat distincte). O astfel de entitate este uneori numită *multiset*. Principiul metodei constă în a construi succesiv soluţia S pornind de la setul vid şi adăugând la fiecare etapă un nou element, şi anume acela care pare "optim" la momentul respectiv. Selecţia elementului care va fi eventual adăugat urmăreşte satisfacerea restricţiilor şi apropierea de optim. Insă alegerea care pare optimă la etapa curentă s-ar putea să nu fie cea mai bună şi din punct de vedere global astfel că e posibil ca o astfel de strategie "lacomă" să nu conducă la o soluţie finală optimă. Totuşi, o dată efectuată o alegere nu se mai revine asupra ei, aceasta fiind irevocabilă. Datorită modului în care se alege următoarea componentă a soluţiei această tehnică este cunoscută şi ca tehnica *alegerii lacome* sau pur şi simplu *greedy*. Trebuie menţionat şi faptul că prin strategia greedy se obţine o singură soluţie chiar dacă problema are mai multe.

Structura generală a algoritmului este descrisă în continuare.

```
greedy1(A)
S ← ∅
while "S nu este soluţie" and "există elemente neselectate în A" do
  selectează a din A
  if S ∪ {a} satisface restricţiile then
    se adaugă a la S
  end if
end while
return S
```

Etapa cea mai importantă este cea a selectării unui nou element. Pentru anumite probleme este posibil ca setul iniţial, A, să fie ordonat după un criteriu determinat de criteriul de optim astfel că elementele vor fi alese chiar în ordinea în care apar în A. Criteriul de sortare depinde de problema concretă care se rezolvă. În acest caz algoritmul poate fi descris ceva mai detaliat astfel:

```
greedy2(A[1..n])
A[1..n] ← sort(A[1..n]) // sortare după un criteriu corelat cu cel de optimizat
i ← 1 // contor de parcurgere a lui A
k ← 0 // contor utilizat în construirea lui S
while "S[1..k] nu este soluţie" and i ≤ n do
  if "(S[1], ..., S[k], A[i]) satisface restricţiile" then
    k ← k + 1; S[k] ← A[i] // se transferă elementul curent din A în S
  end if
```

$i \leftarrow i + 1$ // se trece la următorul element din A
end while
return $S[1..k]$

În aplicaţii contează principiul metodei, algoritmii construiţi pe baza lui putând fi destul de diferiţi de structurile generale `greedy1` respectiv `greedy2`.

Exemplul 7.2 (*Problema submulţimii de cardinal dat şi sumă maximă.*) Se cere determinarea unui subset cu m elemente, S, dintr-un set A având $n > m$ elemente astfel încât suma elementelor selectate să fie cât mai mare.

În acest caz este suficient să ordonăm descrescător elementele din A şi să reţinem primele m elemente:

`submulţime`($A[1..n]$,m)
$A[1..n] \leftarrow$ sortare_descrescătoare($A[1..n]$)
for $i \leftarrow 1, m$ **do**
  $S[i] \leftarrow A[i]$
end for
return $S[1..m]$

Pentru această problemă se va demonstra în secţiunea următoare că strategia greedy produce întotdeauna o soluţie optimă.

Exemplul 7.3 (*Problema monedelor.*) Presupunem că o sumă de bani C trebuie acoperită folosind cât mai puţine monede. Dacă valorile monedelor sunt $\{v_1, v_2, \ldots, v_n\}$ atunci a rezolva problema înseamnă a determina $(k_1, k_2, \ldots, k_n)$ (k_i reprezintă numărul de monede de valoare v_i şi poate fi egal cu 0) astfel încât să fie satisfăcută restricţia $\sum_{i=1}^{n} k_i v_i = C$ iar $\sum_{i=1}^{n} k_i$ să fie cât mai mică. Se observă că această problemă este similară cu problema submulţimii de sumă dată şi cardinal minim cu observaţia că setul A este infinit: $A = \{v_1, v_1, \ldots, v_2, v_2, \ldots, \ldots, v_n, v_n, \ldots)$ (se presupune că pentru fiecare valoare v_i există un număr nelimitat de elemente).

O abordare de tip greedy ar conduce (în cazul în care numărul de monede de fiecare tip este nelimitat) la un algoritm de forma:

`monede`($v[1..n]$,C)
$v[1..n] \leftarrow$ sortare_descrescătoare($v[1..n]$)
for $i \leftarrow 1, n$ **do**
  $S[i] \leftarrow 0$
end for
$i \leftarrow 1$
while $C > 0$ **and** $i \leq n$ **do**
  $S[i] \leftarrow C$ DIV $v[i]$
  $C \leftarrow C$ MOD $v[i]$
  $i \leftarrow i + 1$
end while
if $C = 0$ **then**

```
    return S[1..n]
  else
    "nu s-a gasit solutie"
  end if
```

Observaţie. Tehnica greedy nu conduce pentru orice instanţă a problemei monedelor la soluţia optimă. De exemplu pentru cazul monedelor cu valorile $(25, 20, 10, 5, 1)$ şi a valorii $C = 40$ strategia greedy ar conduce la soluţia $(1, 0, 1, 1, 0)$ în timp ce varianta $(0, 2, 0, 0, 0)$ este optimă. Pe de altă parte există situaţii în care problema nu are soluţii. De exemplu pentru monede având valorile $(20, 10, 5)$ şi $C = 17$ problema nu are soluţie. Se observă uşor că dacă există monedă cu valoarea 1 atunci pentru orice sumă C se poate determina o soluţie. Aceasta nu este însă neapărat optimală. Tehnica greedy conduce la o soluţie optimă pentru problema monedelor doar dacă valorile acestora satisfac anumite restricţii. Un exemplu de valori pentru care se poate demonstra optimalitatea soluţiei este: v_{i-1} este multiplu al lui v_i pentru fiecare $i \geq 2$ (în condiţiile în care $v[1..n]$ este ordonat descrescător).

7.2 Verificarea corectitudinii şi analiza complexităţii

Tehnica alegerii local optimale este o tehnică intuitivă şi eficientă, dar nu garantează obţinerea soluţiei optime decât pentru anumite cazuri particulare de probleme.

7.2.1 Verificarea corectitudinii

Intrucât tehnica greedy nu asigură implicit optimalitatea soluţiei, aceasta trebuie demonstrată pentru cazul particular al problemei tratate. In general, problemele pentru care tehnica greedy poate fi aplicată cu succes au următoarele proprietăţi:

- *Proprietatea de alegere lacomă.* Inseamnă că se poate ajunge la soluţia optimă global efectuând alegeri local optimale. Pentru a demonstra că o problemă are această proprietate se porneşte de la o soluţie optimă global şi se arată că poate fi modificată astfel încât prima componentă să fie obţinută efectuând o alegere local optimală. Se analizează în continuare restul soluţiei (problema se reduce la una similară, dar de dimensiune mai mică). Aplicând principiul inducţiei matematice se arată că la fiecare pas poate fi efectuată o alegere de tip greedy. Această etapă se reduce de fapt la a arăta că problema are proprietatea de *substructură optimă.*

- *Proprietatea de substructură optimă.* Se consideră că o problemă are proprietatea de substructură optimă dacă o soluţie optimă a problemei

conţine soluţii optime ale subproblemelor. Pentru a demonstra acest lucru se poate folosi tehnica reducerii la absurd. Pornind de la o soluţie optimă $S(n) = (s_1, s_2, \ldots, s_n)$ se arată că $S(n-1) = (s_2, \ldots, s_n)$ este soluţie optimă pentru subproblema de dimensiune $n-1$. Se presupune că $S(n-1)$ nu este optimă. Atunci ar exista $S'(n-1) = (s'_2, \ldots, s'_n)$ o soluţie mai bună. Aceasta conduce de regulă la faptul că $S'(n) = (s_1, s'_2, \ldots, s'_n)$ este mai bună decât $S(n)$ ceea ce contrazice ipoteza că $S(n)$ este optimă.

Exemplul 7.2 Demonstrăm că algoritmul `submulţime` generează soluţia optimă.
Proprietatea de alegere lacomă. Fie $O = (o_1, o_2, \ldots, o_m)$ o soluţie optimă a problemei. Cum nu contează ordinea elementelor în O putem presupune că $o_1 > o_2 > \ldots > o_m$. Cum elementele lui A sunt ordonate descrescător $(a_1 > a_2 > \ldots > a_n)$ prin alegere de tip greedy prima componentă ar trebui să fie a_1. Vom arăta că $o_1 = a_1$ prin metoda reducerii la absurd. Presupunem că $o_1 \neq a_1$. Rezultă că $a_1 > o_1$ iar soluţia $O' = (a_1, o_2, \ldots, o_m)$ are proprietatea că $\sum_{o \in O'} o - \sum_{o \in O} o = a_1 - o_1 > 0$ adică este mai bună decât O. Aceasta însă contrazice ipoteza că O este soluţie optimă. Prin urmare $o_1 = a_1$ adică primul element al soluţiei este ales conform strategiei greedy. Este suficient acum să demonstrăm că problema posedă proprietatea de substructură optimă.
Proprietatea de substructură optimă. Fie $O = (o_1, o_2, \ldots, o_m)$ o soluţie optimă a problemei iniţiale. Presupunem că elementele lui A sunt ordonate descrescător $(a_1 > a_2 > \ldots > a_n)$. Arătăm că $O_{(1)} = (o_2, \ldots, o_m)$ este soluţie optimă a problemei similare pentru $A_{(2)} = (a_2, \ldots, a_n)$ prin reducere la absurd. Dacă nu ar fi soluţie optimă înseamnă că ar exista o altă soluţie mai bună: $(o'_2, \ldots, o'_m)$ care completată pe prima poziţie cu o_1 ar conduce la o soluţie globală mai bună decât O. S-a ajuns la o contradicţie, deci problema posedă proprietatea substructurii optime.

Exemplul 7.3 Presupunem că valorile din *problema monedelor* satisfac proprietăţile: $v_1 > v_2 > \ldots > v_n = 1$ şi $v_{i-1} = d_{i-1} v_i$ (v_{i-1} este multiplu al lui v_i) pentru $i = \overline{2, n}$ şi demonstrăm că în acest caz tehnica greedy conduce la o soluţie optimă.

Proprietatea alegerii greedy. Fie $O = (o_1, \ldots, o_n)$ o soluţie optimă ($\sum_{i=1}^{n} o_i v_i = C$ şi $\sum_{i=1}^{n} o_i = K$ este minimă). Arătăm că prima componentă a lui O satisface proprietatea alegerii greedy, adică $o_1 = \lfloor C/v_1 \rfloor$. Fie $C_2 = C - o_1 v_1$ suma ce rămâne după ce s-au dat o_1 monede de valoare v_1. Analizăm două situaţii:

(i) $C_2 < v_1$. În acest caz $C_2 = C \bmod v_1 = C - \lfloor C/v_1 \rfloor v_1$ deci $o_1 = \lfloor C/v_1 \rfloor$.

(ii) $C_2 \geq v_1$. Pornind de la O construim o nouă soluţie O' caracterizată prin $o'_1 = o_1 + \lfloor C_2/v_1 \rfloor$. Suma ce rămâne de acoperit după ce s-au dat monedele de valoare v_1 este de această dată $C'_2 = C - o'_1 v_1 = C_2 - \lfloor C_2/v_1 \rfloor v_1 < C_2$. Diferenţa $C_2 - C'_2$ era acoperită în soluţia O din monedele de valoare mai

mică. Să considerăm cazul particular în care $\lfloor C_2/v_1 \rfloor v_1 \leq o_2 v_2$ (diferenţa era acoperită doar cu monede de valoare v_2). Noua soluţie va fi atunci: $O' = (o'_1, o'_2, o_3, \ldots, o_n)$ cu $o'_1 = o_1 + \lfloor C_2/v_1 \rfloor$ şi $o'_2 = o_2 - \lfloor C_2/v_1 \rfloor d_1$. Aceasta acoperă valoarea:

$$o'_1 v_1 + o'_2 v_2 + o_3 v_3 + \ldots + o_n v_n = (o_1 + \lfloor C_2/v_1 \rfloor) v_1 + (o_2 - \lfloor C_2/v_1 \rfloor d_1) v_2 + \sum_{i=3}^{n} o_i v_i.$$

Cum $v_1 = d_1 v_2$ rezultă că suma este chiar C. În ceea ce priveşte numărul de monede folosite, acesta este:

$$o'_1 + o'_2 + o_3 + \ldots + o_n = o_1 + \lfloor C_2/v_1 \rfloor + o_2 - \lfloor C_2/v_1 \rfloor d_1 + \sum_{i=3}^{n} o_i =$$

$$= \sum_{i=1}^{n} o_i + \lfloor C_2/v_1 \rfloor (1 - d_1) < K$$

adică alegerea de tip greedy conduce la utilizarea a mai puţine monede. Aceasta contrazice ipoteza că O este soluţie optimă. Prin urmare primul element se alege după tehnica greedy. Acelaşi raţionament se aplică şi dacă $\lfloor C_2/v_1 \rfloor v_1 > o_2 v_2$ doar că suma se acoperă din contribuţia mai multor monede de valoare mai mică, fapt posibil datorită relaţiei de divizibilitate existente între valori).

Proprietatea de substructură optimă. Fie $O = (o_1, \ldots, o_n)$ o soluţie optimă. Arătăm că $O_{(2)} = (o_2, \ldots, o_n)$ e soluţie optimă pentru problema acoperirii sumei $C_2 = C - o_1 v_1$ cu monede de valori $v_2, v_3, \ldots, v_n$. Presupunem că $O_{(2)}$ nu este optimă, adică există $O'_{(2)} = (o'_2, \ldots, o'_n)$ cu proprietăţile $\sum_{i=2}^{n} o'_i v_i = \sum_{i=2}^{n} o_i v_i = C_2$ şi $\sum_{i=2}^{n} o'_i < \sum_{i=2}^{n} o_i$. Atunci $O' = (o_1, o'_2, \ldots, o'_n)$ verifică $o_1 v_1 + \sum_{i=2}^{n} o'_i v_i = C$ şi $o_1 + \sum_{i=2}^{n} o'_i < o_1 + \sum_{i=2}^{n} o_i$, adică O' este mai bună decât O, ceea ce contrazice ipoteza că O este optimă.

7.2.2 Analiza complexităţii

Strategia greedy conduce în general la algoritmi eficienţi. Din structura generală (`greedy1`) se observă că prelucrarea care determină ordinul de complexitate este cea de selecţie. Dacă aceasta este simplă se pot obţine chiar algoritmi de complexitate liniară. În general însă aplicarea strategiei necesită o sortare prealabilă a elementelor lui A astfel că se ajunge la complexitate de tipul $O(n^2)$ sau $O(n \lg n)$ (dacă se foloseşte un algoritm eficient de sortare).

În cazul algoritmului `submulţime` dacă m este mic în raport cu n atunci nu este justificat să se ordoneze întregul set A ci să se alegă la fiecare etapă valoarea maximă din cele ce nu au fost încă selectate. Se ajunge astfel la un ordin de complexitate $O(mn)$. În algoritmul `monede` sortarea domină celelalte prelucrări astfel că aceasta impune ordinul de complexitate ($\mathcal{O}(n^2)$ sau $\mathcal{O}(n \lg n)$).

7.3 Aplicaţii

7.3.1 Varianta continuă a problemei rucsacului

Este o problemă clasică ce are multe aplicaţii şi diverse variante. Se consideră un set A de obiecte caracterizate fiecare prin profitul (p) şi dimensiunea lor (d): $A = ((p_1, d_1), \ldots, (p_n, d_n))$. Se mai consideră un rucsac de capacitate maximă C şi se pune problema selectării unui subset de obiecte, $((p_{i_1}, d_{i_1}), \ldots, (p_{i_k}, d_{i_k}))$ care să nu depăşească capacitatea rucsacului ($\sum_{j=1}^{k} d_{i_j} \leq C$) şi să asigure un profit maxim ($\sum_{j=1}^{k} p_{i_j}$ este maximă). Două dintre cele mai cunoscute variante ale problemei sunt:

- *Varianta discretă (0-1).* Obiectele nu pot fi divizate: un obiect este fie preluat în întregime fie nu este preluat.
- *Varianta continuă (fracţionară).* Este posibil să fie transferate şi fracţiuni din obiecte, profitul asigurat fiind proporţional cu fracţiunea.

Doar pentru varianta fracţionară se poate obţine o soluţie optimă aplicând strategia greedy. În acest caz alegerea local optimală este cea care corespunde *profitului relativ* maxim (profitul relativ al obiectului i este p_i/d_i). Aplicând strategia greedy se ajunge la a efectua următoarele prelucrări:

(i) se ordonează A descrescător după profitul relativ;

(ii) se transferă obiectele în ordinea în care apar în A, în întregime cu excepţia ultimului obiect din care se ia doar o fracţiune, atât cât să se umple rucsacul.

Pentru a descrie algoritmul facem următoarele convenţii: $A[i].p$ şi $A[i].d$ reprezintă profitul respectiv dimensiunea obiectului i. Soluţia va fi de forma: $S = (S[1], \ldots, S[n])$ cu $S[i] \in [0, 1]$, iar elementele sale vor fi interpretate astfel: $S[i] = 0$ dacă obiectul i nu este selectat, $S[i] = 1$ dacă obiectul i este selectat în întregime, $S[i] \in (0, 1)$ - este selectată doar o fracţiune $s[i]$ din obiectul i. Cu aceste convenţii algoritmul este descris în 7.1.

Soluţia obţinută aplicând acest algoritm va fi de forma: $S = (1, 1, \ldots, 1, f, 0, \ldots, 0)$. În plus $\sum_{i=1}^{n} s_i d_i = C$, astfel că în continuare considerăm că restricţia problemei este de tip egalitate. Demonstrăm că `rucsac_fracţionar` generează o soluţie optimă arătând că orice soluţie optimă trebuie să respecte criteriul alegerii greedy.

Considerăm că obiectele sunt ordonate descrescător după valoarea profitului relativ: $p_1/d_1 > p_2/d_2 > \ldots > p_n/d_n$ (în cazul când inegalitatea nu este strictă alegerea de tip greedy nu este unică şi complică puţin demonstraţia). Considerăm $O = (o_1, o_2, \ldots, o_n)$ o soluţie optimă şi demonstrăm că este de tip greedy prin reducere la absurd. Presupunem că O nu este de tip greedy şi considerăm că $O' = (o'_1, \ldots, o'_n)$ este generată aplicând tehnica greedy. Fie

Algoritmul 7.1 Varianta fracţionară a problemei rucsacului

```
rucsac_fracţionar(A[1..n],C)
A[1..n] ← sortare_descrescătoare(A[1..n]) // sortare după profitul relativ
for i ← 1, n do
   S[i] ← 0
end for
i ← 1;
while C > 0 and i ≤ n do
   if S[i].d ≤ C then
      S[i] ← 1; C ← C − A[i].d
   else
      S[i] ← C/A[i].d; C ← 0
   end if
   i ← i + 1
end while
return S[1..n]
```

$B_+ = \{i | o'_i \geq o_i\}$, $B_- = \{i | o'_i < o_i\}$ iar $k = \min B_-$ (cel mai mic indice pentru care $o'_i < o_i$). Din structura unei soluţii de tip greedy rezultă că dacă $i \in B_+$ şi $j \in B_-$ atunci $i < j$. Din restricţia problemei rezultă $\sum_{i=1}^{n} o_i d_i = \sum_{i=1}^{n} o'_i d_i$ adică $\sum_{i \in B_+} (o'_i - o_i) d_i = \sum_{i \in B_-} (o_i - o'_i) d_i$. Calculăm profiturile corespunzătoare celor două soluţii: $P = \sum_{i=1}^{n} o_i p_i$ şi $P' = \sum_{i=1}^{n} o'_i p_i$. Diferenţa lor este:

$$P' - P = \sum_{i=1}^{n} (o'_i - o_i) p_i = \sum_{i \in B_+} (o'_i - o_i) d_i \frac{p_i}{d_i} - \sum_{i \in B_-} (o_i - o'_i) d_i \frac{p_i}{d_i}.$$

Se observă că $p_i/d_i > p_k/d_k$ pentru orice $i \in B_+$ iar $p_i/d_i \leq p_k/d_k$ pentru orice $i \in B_-$. Prin urmare

$$P' - P > \frac{p_k}{d_k} \sum_{i \in B_+} (o'_i - o_i) d_i - \frac{p_k}{d_k} \sum_{i \in B_-} (o_i - o'_i) d_i = 0$$

Deci $P' > P$ ceea ce contrazice ipoteza că O este optimă. Deci O are o structură de tip greedy. Proprietatea de substructură optimă rezultă imediat prin reducere la absurd.

Observaţie. Pentru a ilustra faptul că pentru varianta discretă această strategie nu conduce la soluţia optimă considerăm exemplul: $A = ((60, 10), (100, 20), (120, 30))$, $C = 50$. Profiturile relative sunt: 6, 5, 4, iar A este deja ordonat descrescător după acest criteriu. Aplicând strategia greedy s-ar transfera primele două obiecte ducând la un profit egal cu 160. Dacă s-ar transfera al doilea şi al treilea obiect s-ar ajunge la un profit mai mare, 220.

Deşi nu conduce la soluţia optimă, strategia greedy poate fi aplicată şi pentru varianta discretă conducând la o soluţie sub-optimală (se poate demonstra că soluţia obţinută aplicând tehnica greedy satisface: $\sum_{i=1}^{k} s_i p_i \geq P_{opt} - \max_i p_i$, P_{opt} fiind profitul corespunzător soluţiei optime). În schimb se poate obţine o soluţie optimă aplicând tehnica programării dinamice.

7.3.2 Problema selectării activităţilor

Se consideră un set de activităţi care au nevoie de o anumită resursă şi la un moment dat o singură activitate poate beneficia de resursa respectivă (de exemplu activitatea poate fi un examen, iar resursa o sală de examen). Presupunem că pentru fiecare activitate, A_i, se cunoaşte momentul de începere, p_i şi cel de finalizare t_i ($t_i > p_i$). Presupunem că activitatea se desfăşoară în intervalul $[p_i, t_i)$. Două activităţi A_i şi A_j se consideră *compatibile* dacă intervalele asociate sunt disjuncte. Se cere să se selecteze un număr cât mai mare de activităţi compatibile.

O soluţie a acestei probleme constă într-un subset de activităţi $S = (a_{i_1}, \ldots, a_{i_m})$ care satisfac $[p_{i_j}, t_{i_j}) \cap [p_{i_k}, t_{i_k}) = \emptyset$ pentru orice $j \neq k$.

Criteriul de selecţie ar putea fi: (i) cea mai mică durată; (ii) cel mai mic moment de începere; (iii) cel mai mic moment de sfârşit; (iv) intervalul care se intersectează cu cele mai puţine alte intervale. Dintre aceste variante cea pentru care se poate demonstra că asigură obţinerea soluţiei optime este a treia: la fiecare etapă se alege activitatea care se termină cel mai devreme.

Notând cu $A[i].p$, $A[i].t$ momentul de începere respectiv cel de finalizare a activităţii $A[i]$ algoritmul bazat pe strategia greedy este descris în 7.2.

Algoritmul 7.2 Problema selectării activităţilor

```
selecţie_activităţi(A[1..n])
A[1..n] ← sortare_crescătoare(A[1..n]) // sortare după timpul de finalizare
S[1] ← A[1]
i ← 2; k ← 1
while i ≤ n do
  if S[k].t ≤ A[i].p then
    k ← k + 1; S[k] ← A[i];
  end if
  i ← i + 1
end while
return S[1..k]
```

Demonstrăm că algoritmul de mai sus conduce la soluţia optimă. După ordonarea crescătoare după timpul de finalizare avem $t_1 \leq t_2 \leq \ldots \leq t_n$. Considerăm o soluţie optimă $O = ((p_{i_1}, t_{i_1}), \ldots, (p_{i_m}, t_{i_m})$. Evident $t_{i_1} \geq t_1$ prin urmare înlocuind A_{i_1} cu A_1 în O, se obţine o soluţie în care există acelaşi

număr de activităţi compatibile dar prima activitate este aleasă conform strategiei greedy. Deci problema satisface proprietate alegerii local optimale. O dată aleasă prima activitate, problema se reduce la planificarea optimă a activităţilor compatibile cu prima. Dacă O este soluţia optimă a problemei iniţiale atunci $O' = ((p_{i_2}, t_{i_2}), \ldots, (p_{i_m}, t_{i_m})$ este soluţie optimă a subproblemei la care se ajunge după stabilirea primei activităţi. Astfel problema satisface şi proprietatea substructurii optime.

Să considerăm cazul $A = ((1,8),(2,5),(6,8),(5,6))$. Sortând A în ordinea crescătoare a duratei activităţilor se obţine ca soluţie: $S = ((5,6),(6,8),(2,5))$. Prin selecţie după momentul de începere a activităţilor se obţine $((1,8))$, în schimb prin selecţie după momentul de finalizare se obţine o soluţie, $S = ((2,5),(5,6),(6,8))$, în care activităţile sunt deja ordonate în ordinea în care vor fi efectuate.

7.3.3 Problema planificării activităţilor

Se consideră un set de n prelucrări ce trebuie executate de către un procesor. Durata execuţiei prelucrărilor este aceeaşi (se consideră egală cu 1). Fiecare prelucrare i are asociat un termen final de execuţie, $t_i \leq n+1$ şi un profit, p_i. Profitul unei prelucrări intervine în calculul profitului total doar dacă prelucrarea este executată (dacă prelucrarea nu poate fi planificată înainte de termenul final de execuţie profitul este nul). Se cere să se planifice activităţile astfel încât să fie maximizat profitul total (acesta este corelat cu numărul de prelucrări planificate).

O soluţie constă în stabilirea unui "orar" de execuţie a prelucrărilor $S = (s_1, s_2, \ldots, s_n)$, $s_i \in \{1, \ldots, n\}$ fiind indicele prelucrării planificate la momentul i. Pentru rezolvarea problemei se poate aplica o tehnică de tip greedy caracterizată prin:

- se sortează activităţile în ordinea descrescătoare a profitului;
- fiecare activitate se planifică într-un interval liber cât mai apropiat de termenul final de execuţie.

Algoritmul este descris în 7.3. Se observă că dacă pentru fiecare $i \in \{1, \ldots, n\}$ numărul activităţilor care au termenul final cel mult i este cel mult egal cu i ($\text{card}\{j | t_j \leq i\} \leq i-1$) atunci toate activităţile vor fi planificate, altfel vor exista activităţi ce nu pot fi planificate.

Să considerăm $n = 4$ activităţi având termenele finale: $(2,3,4,2)$ şi profiturile corespunzătoare $(4,3,2,1)$. Atunci aplicând tehnica greedy se obţine soluţia $(4,1,2,3)$. Dacă însă termenele de execuţie sunt: $(2,4,1,2)$ şi aceleaşi profituri se obţine soluţia $(3,1,0,2)$ iar activitatea 4 nu este planificată.

Algoritmul 7.3 Problema planificării activităţilor

```
planificare(A[1..n])
A[1..n] ← sortare_descrescătoare(A[1..n]) // sortare după profit (A[i].p)
for i ← 1, n do
   S[i] ← 0
end for
for i ← 1, n do
   poz ← A[i].t − 1 // termenul final de execuţie
   while S[poz] ≠ 0 and poz > 1 do
      poz ← poz − 1
   end while
   if S[poz] = 0 then
      S[poz] ← i // se planifica activitatea
   else
      "activitatea i nu poate fi planificată"
   end if
end for
return S[1..n]
```

7.3.4 Problema împachetării

Se consideră un set de numere $a_1, a_2, \ldots, a_n$ cu proprietatea că $a_i \in (0, C]$. Se cere să se grupeze în cât mai puţine subseturi (k) astfel încât suma elementelor din fiecare subset să nu depăşească valoarea C. Este cunoscută şi sub numele de "bin-packing problem". Există mai multe variante ce folosesc principiul alegerii greedy însă toate sunt *sub-optimale* (nu garantează obţinerea optimului ci doar a unei soluţii suficient de apropiate de cea optimă).

Varianta 1. Se construiesc succesiv submulţimile: se transferă (în ordinea în care se află în set) în primul subset atâtea elemente cât este posibil, din elementele rămase se transferă elemente în al doilea subset etc. (nu e garantată optimalitatea soluţiei însă s-a demonstrat că $k < 2k_{opt}$, k fiind numărul de subseturi generat prin strategia greedy de mai sus, iar k_{opt} este numărul optim de subseturi).

Varianta 2. Se iniţializează n subseturi vide. Se parcurge setul de numere şi fiecare număr se transferă în primul subset în care "încape". Numărul de subseturi nevide astfel constituite k, are proprietatea $k < 1.7k_{opt}$.

Varianta 3. Dacă se ordonează descrescător setul iniţial şi se aplică varianta 2 se obţine o îmbunătăţire: $k < 11/9k_{opt} + 4$.

Considerăm că soluţia va fi reprezentată printr-o matrice $R[1..n, 1..n]$ în care linia i corespunde subsetului i iar $R[i, j]$ conţine fie 0 fie indicele unui element din a care trebuie plasat în subsetul i. Pentru a controla modul de completare a subseturilor se pot folosi tablourile $K[1..n]$ ($K[i]$ specifică indicele ultimului element completat în linia i) şi $S[1..n]$ care conţine suma curentă a

elementelor de pe linia i. Algoritmul este descris în 7.4.

Algoritmul 7.4 Problema împachetării

```
impachetare(integer a[1..n])
R[1..n, 1..n] ← 0; K[1..n] ← 0; S[1..n] ← 0;
a[1..n] ←sortare_descrescatoare(a[1..n])
for i ← 1, n do
  j ← 1
  while S[j] + a[i] > C do
    j ← j + 1
  end while
  K[j] ← K[j] + 1; R[j, K[j]] ← a[i]; S[j] ← S[j] + a[i]
end for
return R[1..n, 1..n]
```

7.4 Probleme

Problema 7.1 Se consideră un rucsac de capacitate C şi un set de obiecte având toate aceeaşi dimensiune, d, însă valori diferite. Să se determine un subset de obiecte care să încapă în rucsac şi a căror valoare totală să fie maximă.

Indicaţie. Se selectează C/d obiecte în ordinea descrescătoare a valorilor.

Problema 7.2 Se consideră un rucsac de capacitate C şi un set de obiecte având toate aceeaşi valoare, v, însă dimensiuni diferite. Să se determine un subset de obiecte care să încapă în rucsac şi a căror valoare totală să fie maximă.

Indicaţie. Se selectează obiecte în ordinea crescătoare a dimensiunilor până se depăşeşte capacitatea rucsacului (ultimul obiect selectat este ignorat).

Problema 7.3 (*Stocare optimală pe suport cu acces secvenţial*) Se consideră un set de n fişiere de dimensiuni d_1,d_2,...,d_n. Se pune problema stocării acestora pe un suport cu acces secvenţial astfel încât timpul total de regăsire a fişierelor să fie cât mai mic. Procesul de regăsire satisface următoarele restricţii: (i) fişierele trebuie regăsite în ordinea inversă a stocării lor; (ii) timpul necesar regăsirii fişierului k este proporţional cu suma dimensiunilor tuturor fişierelor stocate înainte de acest fişier la care se adaugă dimensiunea fişierului (d_k).

Indicaţie. Să considerăm că permutarea $(p(1), p(2), \ldots, p(n))$ indică ordinea în care sunt stocate fişierele. Atunci timpul necesar accesării fişierului k este $\tau(k) = d_{p(1)} + \ldots + d_{p(k)}$. Prin urmare timpul de accesare al tuturor fişierelor este:

$$T(n) = \sum_{k=1}^{n} \tau(k) = nd_{p(1)} + (n-1)d_{p(2)} + \ldots + 2d_{p(k-1)} + d_{p(k)}$$