Metoda Greedy

Metoda Greedy (*greedy*=lacom) este aplicabilă problemelor de optim.

Considerăm mulţimea finită $A=\{a_1,\ldots,a_n\}$ şi o proprietate p definită pe mulţimea submulţimilor lui A:

$p: P(A)\to\{0,1\}$ cu $\begin{cases} p(\phi)=1 \\ p(X)\Rightarrow p(Y),\ \forall Y\subset X \end{cases}$

O submulţime $S\subset A$ se numeşte *soluţie* dacă $p(S)=1$.

Dintre soluţii aleg una care optimizează o funcţie $f: P(A)\to\mathbf{R}$ dată.

Metoda urmăreşte evitarea parcurgerii tuturor submulţimilor (ceea ce ar necesita un timp de calcul exponenţial), mergându-se "direct" spre soluţia optimă. Nu este însă garantată obţinerea unei soluţii optime; de aceea aplicarea metodei Greedy trebuie însoţită neapărat de o demonstraţie.

Distingem două variante generale de aplicare a metodei Greedy:

```
S←φ
for i=1,n
   x←alege(A); A←A\{x}
   if p(S∪{x})=1
   then S←S∪{x}
```

```
prel(A)
S←φ
for i=1,n
   if p(S∪{ai})
   then S←S∪{ai}
```

Observaţii:

- în algoritm nu apare funcţia `f` !!
- timpul de calcul este liniar (exceptând prelucrările efectuate de procedurile `prel` şi `alege`);
- dificultatea constă în a concepe procedurile `alege`, respectiv `prel`, în care este "ascunsă" funcţia `f`.

Exemplul 1. Se consideră mulţimea de valori reale $A=\{a_1,\ldots,a_n\}$. Se caută submulţimea a cărei sumă a elementelor este maximă.

Vom parcurge mulţimea şi vom selecta numai elementele pozitive.

```
k←0
for i=1,n
   if ai>0
   then k←k+1; sk←ai
write(s)
```

Exemplul 2. Caut cel mai lung şir strict crescător format cu elemente din mulţimea $\{a_1,\ldots,a_n\}$.

Începem prin a ordona crescător elementele mulţimii (corespunzător procedurii `prel`). Apoi eliminăm dublurile.

Astfel, dacă în urma ordonării a rezultat $a=(1,1,2,3,4,4,5,6,7,8,8\}$, vom obţine succesiv:

$s=(1,2,3,4,5,6,7,8)$

(Contra)exemplul 3. Fie mulţimea $A=\{a_1,\ldots,a_n\}$ cu elemente pozitive. Caut submulţimea de sumă maximă, dar cel mult egală cu M dat.

Dacă procedez ca în exemplul 1, pentru A=(6,3,4,2) şi M=7 obţin {6}. Dar soluţia optimă este {3,4} cu suma egală cu 7.

Continuăm cu prezentarea unor exemple clasice.

- **Memorarea textelor pe bandă**

Textele cu lungimile L(1),...,L(n) urmează a fi aşezate pe o bandă. Pentru a citi textul de pe poziţia k, trebuie citite textele de pe poziţiile 1,2,...,k (conform specificului accesului secvenţial pe bandă).

O *soluţie* înseamnă o permutare $p \in S_n$.

Pentru o astfel de permutare (ordine de aşezare a textelor pe bandă), timpul pentru a citi textul de pe poziţie k este: $T_p(k)=L(p_1)+...+L(p_k)$. Presupunând textele egal probabile, trebuie minimizată valoarea $T(p)=\frac{1}{n}\sum_{k=1}^{n} T_p(k)$.

Să observăm că funcţia T se mai poate scrie: $T(p)=\frac{1}{n}\sum_{k=1}^{n}(n-k+1)L(p_k)$

(textul de pe poziţia k este citit dacă vrem să citim unul dintre textele de pe poziţiile k,...,n).

Conform strategiei Greedy, începem prin a ordona crescător vectorul L. Rezultă că în continuare L(i)<L(j), $\forall i<j$.

Demonstrăm că în acest mod am obţinut modalitatea optimă, adică permutarea identică minimizează funcţia de cost T.

Fie $p \in S_n$ optimă, adică p minimizează funcţia T. Dacă p este diferită de permutarea identică $\Rightarrow \exists i<j$ cu $L(p_i)>L(p_j)$:

p=(|_____|p_i|_____|p_j|_____|)

Considerăm permutarea p' în care am interschimbat elementele de pe poziţiile i şi j:

p'=(|_____|p_j|_____|p_i|_____|)

Atunci $n[T(p)-T(p')] = (n-i+1)L(p_i) + (n-j+1)L(p_j) - (n-i+1)L(p_j) - (n-j+1)L(p_i) = (j-i)L(p_i)+(i-j)L(p_j) = (j-i)[L(p_i)-L(p_j)]>0$, ambii factori fiind pozitivi.

Rezultă că $T(p')<T(p)$. Contradicţie.

- **Problema continuă a rucsacului**

Se consideră un rucsac de capacitate (greutate) maximă G şi n obiecte caracterizate prin:

- greutăţile lor $g_1,...,g_n$;
- câştigurile $c_1,...,c_n$ obţinute la încărcarea lor în totalitate în rucsac.

Din fiecare obiect poate fi încărcată orice fracţiune a sa.

Se cere o modalitate de încărcare de (fracţiuni de) obiecte în rucsac, astfel încât câştigul total să fie maxim.

Prin *soluţie* înţelegem un vector $x=(x_1,\ldots,x_n)$ cu $\begin{cases} x_i \in [0,1], \forall i \\ \sum_{i=1}^{n} g_i x_i \le G \end{cases}$

O *soluţie optimă* este soluţie care maximizează funcţia $f(x)=\sum_{i=1}^{n} c_i x_i$.

Dacă suma greutăţilor obiectelor este mai mică decât G, atunci încarc toate obiectele: $x=(1,\ldots,1)$. De aceea presupunem în continuare că $g_1+\ldots+g_n>G$.

O primă ideea pentru aplicarea metodei Greedy constă în a ordona obiectele descrescător după cost, întrucât doresc un maxim. Dar pentru exemplul:

G=5	i=1	i=2	i=3
c_i	6	4	2.5
g_i	5	3	2
x_i	1	0	0

am obţine maximul egal cu 5 (cu primul obiect am umplut rucsacul), dar varianta cu al 2-lea şi al 3-lea obiect conuce la câştigul 6.5.

În strategiei Greedy care urmează, ordonez obiectele descrescător după câştigul la unitatea de greutate, deci lucrăm în situaţia:

$$\frac{c_1}{g_1} \ge \frac{c_2}{g_2} \ge \ldots \ge \frac{c_n}{g_n} \qquad (*)$$

Exemplu

G=5	i=1	i=2	i=3
c_i	6	4	2.5
g_i	4	3	2
x_i	1	1/3	0

Algoritmul constă în încărcarea în această ordine a obiectelor, atâta timp cât nu se depăşeşte greutatea G (ultimul obiect pote fi eventual încărcat parţial):

```
disp ← G { disp reprezintă greutatea disponibilă }
for i=1,n
   if gi≤disp then xi←1; disp←disp-gi
              else xi←disp/gi;
                   for j=i+1,n
                    xj ← 0
                   stop
write(x)
```

Pentru $G=5$ vom avea x de mai sus cu câştigul total 6+4/3 = 7.33

Am obţinut deci $x=(1,\ldots,1,x_j,0,\ldots,0)$ cu $x_j \in [0,1)$.

Arătăm că soluţia astfel obţinută este optimă.

Fie y soluţia optimă: $y=(\ldots,y_k,\ldots)$ cu $\begin{cases}\sum_{i=1}^{n} g_i y_i = G \\ \sum_{i=1}^{n} c_i y_i \text{ maxim}\end{cases}$

Dacă $y \neq x$, fie k *prima* poziţie pe care $y_k \neq x_k$.

Observaţii:

- $k \leq j$: pentru $k>j$ se depăşeşte G.
- $y_k < x_k$:
  - pentru $k<j$: evident, deoarece $x_k=1$;
  - pentru $k=j$: dacă $y_k > x_k$ se depăşeşte G.

Considerăm soluţia: $y'=(y_1,\ldots,y_{k-1},x_k,\alpha y_{k+1},\ldots,\alpha y_n)$ cu $\alpha<1$ (primele $k-1$ componente coincid cu cele din x). Păstrăm greutatea totală G, deci: $g_k x_k+\alpha(g_{k+1}y_{k+1}+\ldots+g_n y_n)=g_k y_k+g_{k+1}y_{k+1}+\ldots+g_n y_n$. Rezultă:

$$g_k(x_k-y_k)=(1-\alpha)(g_k+1y_{k+1}+\ldots+g_n y_n) \qquad (**)$$

Compar performanţa lui y' cu cea a lui y:

$$\begin{aligned} f(y')-f(y) &= c_k x_k+\alpha c_{k+1}y_{k+1}+\ldots+\alpha c_n y_n-(c_k y_k+c_{k+1}y_{k+1}+\ldots+c_n y_n) = \\ &= c_k(x_k-y_k)+(\alpha-1)(c_{k+1}y_{k+1}+\ldots+c_n y_n) = \\ &= c_k/g_k[g_k(x_k-y_k)+(\alpha-1)(g_k/c_k c_{k+1}y_{k+1}+\ldots+g_k/c_k c_n y_n)] \end{aligned}$$

Dar $\alpha-1<0$ şi $g_k/c_k \leq g_s/c_s$, $\forall s>k$.

Atunci $f(y')-f(y)>c_k/g_k[g_k(x_k-y_k)+(\alpha-1)(g_{k+1}y_{k+1}+\ldots+g_n y_n)]=0$, deci $f(y')>f(y)$. Contradicţie.

Problema discretă a rucsacului diferă de cea continuă prin faptul că fiecare obiect poate fi încărcat numai în întregime în rucsac.

Să observăm că aplicarea metodei Greedy eşuează în acest caz. Într-adevăr, aplicarea ei pentru ultimul exemplu considerat:

$n=3$ $c=(6,4,2.5)$ şi $g=(4,3,2)$, $G=5$

are ca rezultat încărcarea primul obiect; câştigul obţinut este 6. Dar încărcarea ultimelor două obiecte conduce la câştigul superior 6.5.

• Problema arborelui parţial de cost minim.

Fie $G=(V,M)$ un graf neorientat cu muchiile etichetate cu costuri strict pozitive. *Se cere determinarea unui graf parţial de cost minim*.

Ca exemplificare, să considerăm n oraşe iniţial nelegate între ele. Pentru fiecare două oraşe se cunoaşte costul conectării lor directe (considerăm acest cost egal cu $+\infty$ dacă nu este posibilă conectarea lor). Constructorul trebuie să conecteze oraşele astfel încât din oricare oraş să se poată ajunge în oricare altul. Ce legături directe trebuie să aleagă constructorul astfel încât costul total al lucrării să fie minim?

Este evident că graful parţial căutat este un arbore (dacă ar exista un ciclu, am putea îndepărta o muchie din el, cu păstrarea conexităţii şi micşorarea costului total).

Vom presupune în continuare că muchiile au costuri diferite. Pentru muchiile de acelaşi cost, introducem mici perturbaţii în costurile lor. Dacă perturbaţiile sunt suficient de mici:
- două muchii de costuri diferite în graful iniţial vor avea în continuare costuri diferite;
- un arbore de cost minim pentru noul graf va fi un arbore de cost minim şi pentru graful iniţial.

Cele de mai sus arată că presupunerea făcută nu constituie o restricţie.

Lemă. Fie `S` o submulţime de vârfuri care nu este nici vidă şi nici nu coincide cu `V`. Fie `m=(i,j)` muchia de cost minim cu `i∈S` şi `j∈V\S`. Atunci orice arbore parţial de cost minim va conţine muchia `m`.

Presupunem prin absurd că există un arbore parţial de cost minim `A` în care nu apare `m`. Considerăm drumul din arbore de la `i` la `j`. Fie `m'=(i',j')` o muchie din acest drum cu `i'∈S` şi `j'∈V\S`. Fie `A'=A∪{m}\{m'}`.

`A'` este conex, deoarece orice drum ce conţinea pe `m'` poate fi "rerutat" prin `m`.

`A'` este şi aciclic, deoarece singurul ciclu din `A∪{m}` este cel format din drumul considerat şi închis cu `m`; cum am eliminat o muchie din acest ciclu (şi anume pe `m'`), rezultă că acest unic ciclu nu mai este prezent.

În plus costul noului arbore este mai mic, datorită proprietăţii de minimalitate a lui `m`. Contradicţie.

○ *Algoritmul lui Kruskal*

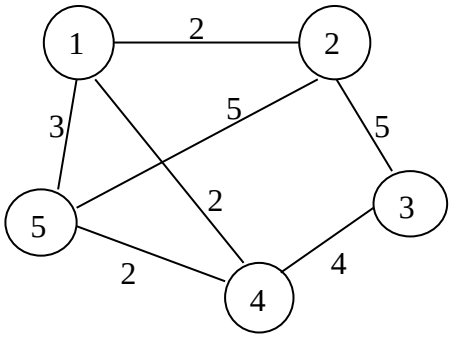
Vom aplica metoda Greedy: *adaug mereu o muchie de cost minim dintre cele nealese şi care nu formează un ciclu cu precedentele muchii alese*.

Ca de obicei, fie `|V|=n`, `|M|=m`. Vor fi alese deci `n-1` muchii.

Construim o matrice `mat` cu `m` linii şi `n` coloane. Pe fiecare linie apar extremităţile `i` şi `j` ale unei muchii, precum şi costul `c(i,j)` al acestei muchii.

Începem prin a ordona liniile matricii crescător după ultima coloană (a costurilor muchiilor). Muchiile alese vor fi marcate în prezentare cu bold.

Exemplu. Considerăm graful de mai jos şi matricea `mat` ataşată.



1	2	2
1	4	2
4	5	2
1	5	3
3	4	4
2	5	5
2	3	5

Conform algoritmului lui Kruskal, vor fi alese în ordine muchiile:
(1,2), (1,4), (4,5), (3,4)
cu costul total egal cu 10. Muchia (1,5) nu a fost aleasă deoarece formează cu precedentele un ciclu.

Dificultatea principală constă în verificarea faptului că o muchie formează sau nu cu precedentele un ciclu. Descriem în continuare o modalitate (nu cea optimă!) de a rezolva problema.

Plecând de la observaţia că orice soluţie parţială este o pădure, vom asocia fiecărui vârf `i` un reprezentant r_i care identifică componenta conexă (arborele) din care face parte vârful în soluţia parţială. Atunci:

- o muchie `(i,j)` va forma un ciclu cu precedentele $\Leftrightarrow r_i=r_j$;
- la alegerea (adăugarea) unei muchii `(i,j)` vom reuni arborii cu rădăcinile r_i şi r_j (pentru ca vârfurile noului arbore să aibă acelaşi reprezentant).

În algoritmul care urmează metoda descrisă, `k` este numărul liniei curente din matricea `mat`, `nm` este numărul de muchii alese, iar `cost` este costul muchiilor alese:

```
r_i ← i, ∀i=1,n
k ← 1; nm ← 0; cost ← 0
while k≤m & nm<n-1
   i1 ← mat(k,1); i2 ← mat(k,2)
   r1 ← r_i1; r2 ← r_i2
   if r1≠r2
   then nm ← nm+1; cost ← cost+mat(k,3);
        write(i1,i2);
        //reun(r1,r2)
        for x=1,n
          if r_x=r1 then
   k ← k+1
if nm<n-1 then write('Graf neconex')
          else write('Graf conex. Costul=',cost)
```

Demonstrăm în continuare corectitudinea algoritmului lui Kruskal.

Să considerăm o muchie oarecare `m=(i,j)` adăugată de algoritmul lui Kruskal şi fie `S` mulţimea vârfurilor arborelui din care face parte la momentul curent `i`. Vârful `j` nu aparţine lui `S` deoarece altfel s-ar închide un ciclu. Rezultă că muchia `m` este prima (şi, implicit, de cost minim) care leagă un vârf din `S` cu unul din `V\S` deci, conform lemei, face parte din *orice arbore de cost minim*. Cum algoritmul lui Kruskal construieşte un arbore parţial, rezultă că acesta este de cost minim.

- ***Problema apartenenţei şi reuninunii (Find and Union)***

Fie $A=\{a_1,\ldots,a_n\}$ o mulţime finită. Vom lucra cu partiţii ale sale.

Iniţial $A=\{a_1\}\cup\ldots\cup\{a_n\}$.

Asupra partiţiei curente se fac, într-o ordine oarecare, următoarele operaţii:

- *apartenenţă*: determinarea submulţimii căreia îi aparţine un element oarecare al lui A;
- *reuniune*: înlocuirea a două submulţimi din partiţie cu reuniunea lor.

Pentru gestionarea celor două operaţii cu timp de executare mai mic decât liniar, fiecare submulţime va fi memorată sub forma unui arbore în care rădăcina va deveni reprezentantul submulţimii. Pentru fiecare vârf `x` al unui arbore vom memora informaţia `tata(x)` cu următoarea semnificaţie:

- dacă x este diferit de rădăcină, tata(x) este tatăl său;
- dacă x este rădăcina unui arbore, tata(x) este -(numărul de elemente din arbore).

Exemplu. O reprezentare a partiţiei:
{1,2,3,4,5,6}={1,2,3}∪{4}∪{5,6}
este:

cu tata=(-3,1,1,-1,-2,5).

Reprezentantul r al submulţimii căruia îi aparţine un element x al mulţimii se determină în mod evident astfel:

```
r ← x
while tata(r)>0
   r ← tata(r)
return r
```

Cum timpul necesar apartenenţei depinde de înălţimea arborilor, vom căuta ca la reuniune să obţinem un arbore de înălţime cât mai mică. Fie r1 şi r2 rădăcinile a doi arbori (reprezentanţii a două submulţimi). Nu vom lucra direct cu înălţimile arborilor, ci vom ataşa arborelui cu mai multe elemente pe cel cu mai puţine elemente:

```
k ← tata(r1)+ tata(r2)
if tata(r1)>tata(r2)  //
then tata(r1) ← r2; tata(r2) ← k
else tata(r2) ← r1; tata(r1) ← k
```

Pentru exemplul de mai sus, reuniunea arborilor de rădăcini 1 şi 5 constă în a ataşa 5 ca fiu al lui 1, cu actualizările de rigoare.

Evident, timpul cerut de reuniune este constant.

Propoziţie. Pentru un arbore cu n vârfuri construit prin regulile de mai sus pentru reuniune, înălţimea sa este cel mult $[\log_2 n]$.

Facem demonstraţia prin inducţie după n.

Pentru n=1 propoziţia este evident verificată.

Presupunem propoziţia adevărată pentru toţi arborii cu cel mult n-1 vârfuri şi considerăm un arbore cu n vârfuri. El a luat naştere prin reuniunea a doi arbori A_1 şi A_2 având respectiv n_1 şi n_2 vârfuri, cu $n_1+n_2=n$. Vom presupune că $n_1 \le n/2$, deci $n_1 \le n_2$.

Fie h, h_1, h_2 înălţimile arborelui considerat şi cele ale celor două subarbori din care a rezultat.

Dacă $n_1<n_2$, atunci $n_1<n/2<n_2<n$, $h_1 \le [\log_2(n/2)]$, $h_2 \le [\log_2 n]$. Cum A_1 devine "subarbore" al lui A_2 :

$h = \max\{h_1+1, h_2\} \le \max\{[\log_2(n/2)]+1, [\log_2 n]\} = [\log_2 n]$.

Dacă $n_1=n_2$, atunci $n_1=n_2=n/2$ şi $h_1, h_2 \le [\log_2(n/2)]$. Cum A_2 devine "subarbore" al lui A_1 :

$h = \max\{h_1, h_2+1\} \le [\log_2(n/2)]+1 = [\log_2 n]$.

Observaţie. Dacă folosim tehnica *Find and Union* pentru păstrarea partiţiei vârfurilor în arbori (cu legătura `tata`), timpul va fi de ordinul $O(m \cdot \log n)$ deoarece:
- arborii din orice partiţie a lui V au cel mult n vârfuri, deci înălţimea lor este cel mult egală cu $[\log_2 n]$;
- sunt analizate cel mult m muchii, iar la analiza fiecăreia operaţiile de determinare a reprezentanţilor capetelor muchiei curente necesită cel mult $[\log_2 n]$ paşi.

o ***Algoritmul lui Prim***

Acest algoritm este tot o ilustrare a metodei Greedy şi constă în următoarele:
- se începe prin selectarea unui vârf;
- la fiecare pas aleg o muchie (i,j) de lungime minimă cu i selectat, dar j neselectat.

De această dată, la fiecare pas se obţine un arbore; după $n-1$ paşi se va obţine un arbore parţial de cost minim.

Conform lemei, corectitudinea algoritmului lui Prim este evidentă: *la fiecare pas se adaugă o muchie ce face parte din <u>orice</u> arbore parţial de cost minim*.

Pentru a obţine şi pentru algoritmul lui Prim un timp de ordinul $O(m \cdot \log n)$, putem folosi cozi cu prioritate (ansamble).

Fie v_0 primul vârf ales şi S mulţimea vârfurilor din arborele curent; iniţial $S=\{v_0\}$.

Pentru orice $v \in V \setminus S$ definim $d(v)=\{c(u,v) \mid u \in S\}$. Iniţial $d(v)$ este: $c(v_0,v)$ pentru $(v_0,v) \in M$, respectiv $+\infty$ în caz contrar.

Orice element din ansamblu este un triplet $(v,d(v),u)$, unde u este acel vârf din S pentru care se realizează minimul, adică $c(u,v)=d(v)$; ansamblul este realizat pe baza cheilor $d(v)$.

Mai folosim tabloul `poz`, unde $poz(v)$ este poziţia în ansamblu a vârfului v; reamintim că un vârf poate fi retrogradat sau promovat în cursul operaţiilor asupra ansamblului.

La operaţiile cu ansamble, deja cunoscute şi necesitând un timp de ordinul cel mult $O(\log m)$, adăugăm următoarele care necesită tot timp logaritmic:

- `extrag_rad`, prin care este extrasă rădăcina; ea este înlocuită cu ultimul element şi apoi este refăcut ansamblul, cu actualizarea tabloului `poz`;
- `înlocuiesc`, care realizează actualizarea informaţiei $d(v)$ a unui vârf v astfel:

a) identifică elementul din ansamblu pe baza tabloului `poz` (mereu actualizat);

b) înlocuieşte pe $d(v)$, după care reface ansamblul prin promovarea elementului (dacă este mai mic decât tatăl său) sau prin retrogradare (dacă este mai mare decât cel puţin unul dintre fiii săi); în acelaşi timp este actualizat tabloul `poz`.

Algoritmul devine acum următorul:

$S \leftarrow \{v_0\}$; cost $\leftarrow 0$;
este creat ansamblul;
de $n-1$ ori:

```
   extrag(v,d(v),u) din ansamblu (coada cu priorităţi)
   write(u,v); cost ← cost + c(u,v); S ← S ∪ {v}
   for toţi w∈Lv
      if c(v,w)<d(w)
      then înlocuiesc(w,d(w),u) cu (w,c(v,w),u)
write(cost)
```

Timpul necesitat de algoritm este acum $O(m \cdot \log n)$, deoarece crearea ansamblului necesită acest timp, iar operaţia de înlocuire poate avea loc cel mult o dată pentru fiecare muchie.