TEHNICA DE PROGRAMARE "GREEDY"

1. Prezentare generală

Tehnica de programare Greedy este utilizată, de obicei, pentru rezolvarea problemelor de optimizare, adică a acelor probleme în care se cere determinarea unei submulţimi a unei mulţimi date pentru care se minimizează sau se maximizează valoarea unei funcţii obiectiv. Formal, o problemă de optimizare poate fi enunţată astfel: "*Fie A o mulţime nevidă şi $f: \mathcal{P}(A) \to \mathbb{R}$ o funcţie obiectiv asociată mulţimii A, unde prin $\mathcal{P}(A)$ am notat mulţimea tuturor submulţimilor mulţimii A. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie minimă/maximă pe S (i.e., pentru orice altă submulţime $T \subseteq A, T \neq S$, valoarea funcţiei obiectiv f va fi cel puţin /cel mult egală cu valoarea funcţiei obiectiv f pe submulţimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulţime nevidă de numere întregi. Să se determine o submulţime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcţia obiectiv nu este dată în formă matematică şi nu se precizează explicit faptul că suma elementelor submulţimii S trebuie să fie maximă *în raport cu suma oricărei alte submulţimi*, acest lucru subînţelegându-se. Formal, problema poate fi enunţată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ şi $f: \mathcal{P}(A) \to \mathbb{R}$, $f(S) = \sum_{x \in S} x$. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie maximă pe S, i.e. $\forall T \subseteq A, T \neq S \Rightarrow f(T) \leq f(S)$ sau, echivalent, $\forall T \subseteq A, T \neq S \Rightarrow \sum_{x \in T} x \leq \sum_{x \in S} x$."*

Evident, orice problemă de acest tip poate fi rezolvată prin *metoda forţei-brute*, astfel: se generează, pe rând, toate submulţimile S ale mulţimii A şi pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obţinută este mai mică/mai mare decât minimul/maximul obţinut până în acel moment, atunci se actualizează minimul/maximul şi se reţine submulţimea S. Deşi aceasta rezolvare este corectă, ea are o complexitate exponenţială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulţimea S, la fiecare pas, cel mai bun element disponibil din mulţimea A din punct de vedere al optimizării funcţiei obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcţiei obiectiv combinând optimele sale locale. Totuşi, prin combinarea unor optime locale nu se obţine întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre Bucureşti şi Arad (un optim local), precum şi cel mai scurt drum posibil dintre Arad şi Ploieşti (alt optim local). Combinând cele două optime locale nu vom obţine un optim global, deoarece, evident, cel mai scurt drum de la Bucureşti la Ploieşti nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însoţită de o demonstraţie a corectitudinii (optimalităţii) criteriului de selecţie pe care trebuie să-l îndeplinească un element al mulţimii A pentru a fi adăugat în soluţia S.

De exemplu, să considerăm *problema plății unei sume folosind un număr minim de monede*. O rezolvare de tip Greedy a acestei probleme ar putea consta în utilizarea, la fiecare pas, a unui număr maxim de monede cu cea mai mare valoare admisibilă. Astfel, pentru monede cu valorile de 8$, 7$ și 5 $, o sumă de 23$ va fi plătită în următorul mod: 23$ = 2*8$ + 7$ = 2*8$ + 1*7$, deci se vor utiliza 3 monede, ceea ce reprezintă o soluție optimă. Dacă vom considera monede cu valorile de 8$, 7$ și 1 $, o sumă de 14$ va fi plătită în următorul mod: 14$ = 1*8$ + 6$ = 1*8$ + 6*1$, deci se vor utiliza 7 monede, ceea ce nu reprezintă o soluție optimă (i.e., 2*7$). Mai mult, pentru monede cu 8$, 7$ și 5 $, o sumă de 14$ nu va putea fi plătită deloc: 14$ = 1*8$ + 6$ = 1*8$ + 1*5$ + 1$, deoarece restul rămas, de 1$, nu mai poate fi plătit (evident, soluția optimă este 2*7$). În concluzie, acest algoritm de tip Greedy, numit *algoritmul casierului*, nu furnizează întotdeauna o soluție optimă pentru plata unei sume folosind un număr minim de monede. Totuși, pentru anumite valori ale monedelor, el poate furniza o soluție optimă pentru orice sumă dată (de exemplu, pentru monedele din Statele Unite ale Americii: https://personal.utdallas.edu/~sxb027100/cs6363/coin.pdf)

Revenind la problema determinării unei submulțimi S cu sumă maximă, observăm faptul că aceasta trebuie să conțină toate elementele pozitive din mulțimea A, deci criteriul de selecție este ca elementul curent din A să fie pozitiv (demonstrația optimalității este banală). Dacă mulțimea A nu conține niciun număr pozitiv, care va fi soluția problemei?

În anumite probleme, criteriul de selecție poate fi aplicat mai eficient dacă se realizează o prelucrare inițială a elementelor mulțimii A – de obicei, o sortare a lor. De exemplu, să considerăm următoarea problemă: "*Fie A o mulțime nevidă formată din n numere întregi. Să se determine o submulțime $S \subseteq A$ având exact k elemente ($k \leq n$) cu proprietatea că suma elementelor sale este maximă.*". Evident, submulțimea S trebuie să conțină cele mai mari k elemente ale mulțimii A, iar acestea pot fi selectate în două moduri:

- de k ori se selectează maximul din mulțimea A și se elimină (sau doar se marchează – important este ca, la fiecare pas, să nu mai luăm în considerare maximul determinat anterior), deci această soluție va avea complexitatea $\mathcal{O}(kn)$, care oscilează între $\mathcal{O}(n)$ pentru valori ale lui k mult mai mici decât n și $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n;
- sortăm crescător elementele mulțimii A și apoi selectăm ultimele k elemente, deci această soluție va avea complexitatea $\mathcal{O}(k + n \log_2 n) \approx \mathcal{O}(n \log_2 n)$, care nu depinde de valoarea k.

În plus, a doua varianta de implementare are avantajul unei implementări mai simple decât prima.

În concluzie, pentru o mulțime A cu n elemente, putem considera următoarea formă generală a unui algoritm de tip Greedy:

```
prelucrarea inițială a elementelor mulțimii A
S = []
for x in A:
   if elementul x verifică criteriul de selecție:
        S.append(x)
afișarea elementelor mulțimii S
```

Se observă faptul că, de obicei, un algoritm de acest tip are o complexitate relativ mică, de tipul $\mathcal{O}(n \log_2 n)$, dacă prin sortarea (prelucrarea) elementelor mulțimii A cu complexitatea $\mathcal{O}(n \log_2 n)$ se poate ulterior testa criteriul de selecție în $\mathcal{O}(1)$. Dacă nu se realizează prelucrarea inițială a elementelor mulțimii A, atunci algoritmul (care trebuie puțin adaptat) va avea complexități de tipul $\mathcal{O}(n)$ sau $\mathcal{O}(n^2)$, induse de complexitatea verificării criteriului de selecție. Evident, acestea nu sunt toate complexitățile posibile pentru un algoritm de tip Greedy, ci doar sunt cele mai des întâlnite!

2. Minimizarea timpului mediu de așteptare

La un ghișeu stau la coadă n persoane $p_1, p_2, \dots, p_n$ și pentru fiecare persoană p_i se cunoaște timpul său de servire t_i. Să se determine o modalitate de reașezare a celor n persoane la coadă, astfel încât timpul mediu de așteptare să fie minim.

De exemplu, să considerăm faptul că la ghișeu stau la coadă $n = 6$ persoane, având timpii de servire $t_1 = 7,\ t_2 = 6,\ t_3 = 5,\ t_4 = 10,\ t_5 = 6$ și $t_6 = 4$. Evident, pentru ca o persoană să fie servită, aceasta trebuie să aștepte ca toate persoanele aflate înaintea sa la coadă să fie servite, deci timpii de așteptare ai celor 6 persoane vor fi următorii:

Persoana	**Timpul de servire (t_i)**	**Timp de așteptare (a_i)**
p_1	7	7
p_2	6	7 + 6 = 13
p_3	3	13 + 3 = 16
p_4	10	16 + 10 = 26
p_5	6	26 + 6 = 32
p_6	3	32 + 3 = 35
Timpul mediu de așteptare (M):		$\frac{7 + 13 + 16 + 26 + 32 + 35}{6} = \frac{129}{6} = 21.5$

Deoarece timpul de servire al unei persoane influențează timpii de așteptare ai tuturor persoanelor aflate după ea la coadă, se poate intui foarte ușor faptul că minimizarea

timpului mediu de aşteptare se obţine rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	**Timpul de servire** (t_i)	**Timp de aşteptare** (a_i)
p_3	3	3
p_6	3	3 + 3 = 6
p_2	6	6 + 6 = 12
p_5	6	12 + 6 = 18
p_1	7	18 + 7 = 25
p_4	10	25 + 10 = 35
Timpul mediu de aşteptare (*M*):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

Practic, minimizarea timpului mediu de aşteptare este echivalentă cu minimizarea timpului de aşteptare al fiecărei persoane, iar minimizarea timpului de aşteptare al unei persoane se obţine minimizând timpii de servire ai persoanelor aflate înaintea sa!

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \dots, p_i, \dots, p_j, \dots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \le t_2 \le \cdots \le t_i \le \cdots \le t_j \le \cdots \le t_n$. De asemenea, vom presupune faptul că timpii individuali de servire $t_1, t_2, \dots, t_n$ nu sunt toţi egali între ei (în acest caz, problema ar fi trivială), deci există $i < j$ astfel încât $t_i < t_j$. În continuare, presupunem faptul că această modalitate P_1 de aranjare a persoanelor la coadă (o permutare, de fapt) nu este optimă, deci există o altă modalitate optimă P_2 de aranjare $p_1, p_2, \dots, p_j, \dots, p_i, \dots p_n$ diferită de cea iniţială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i şi p_j din varianta iniţială, adică persoana p_j se află acum pe poziţia i în coadă, iar persoana p_i se află acum pe poziţia j, unde $i < j$).

În cazul primei modalităţi de aranjare P_1, timpul mediu de servire M_1 este egal cu:

$$M_1 = \frac{t_1 + (t_1 + t_2) + \cdots + (t_1 + \cdots + t_i) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_i + \cdots + (n-j+1)t_j + \cdots + 2t_{n-1} + t_n}{n}$$

În cazul celei de-a doua modalităţi de aranjare P_2, timpul mediu de servire M_2 este egal cu:

$$M_2 = \frac{t_1 + (t_1 + t_2) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_i) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_j + \cdots + (n-j+1)t_i + \cdots + 2t_{n-1} + t_n}{n}$$

Comparăm acum M_1 cu M_2, calculând diferența dintre ele:

$$M_1 - M_2 = \frac{(n-i+1)t_i + (n-j+1)t_j - (n-i+1)t_j - (n-j+1)t_i}{n} =$$
$$= \frac{t_i(n-i+1-n+j-1) + t_j(n-j+1-n+i-1)}{n} =$$
$$= \frac{t_i(-i+j) + t_j(-j+i)}{n} = \frac{-t_i(i-j) + t_j(i-j)}{n} = \frac{(t_j - t_i)(i-j)}{n}$$

Deoarece $i < j$ și $t_j > t_i$, obținem faptul că $M_1 - M_2 = \frac{(t_j-t_i)(i-j)}{n} < 0$ (evident, $n \geq 1$), ceea ce implică $M_1 < M_2$. Acest fapt contrazice optimalitatea modalității de aranjare P_2, deci presupunerea că modalitatea de aranjare P_1 (în ordinea crescătoare a timpilor de servire) nu ar fi optimă este falsă!

Atenție, soluția acestei probleme constă într-o rearanjare a persoanelor $p_1, p_2, \dots, p_n$, deci în implementarea acestui algoritm nu este suficient să sortăm crescător timpii de servire, ci trebuie să memorăm perechi de forma (p_i, t_i), folosind, de exemplu, un tuplu, iar apoi să le sortăm crescător după componenta t_i.

```
# functie folosita pentru sortarea crescătoare a persoanelor
# în raport de timpii de servire (cheia)
def cheieTimpServire(t):
    return t[1]

# funcția afișează, într-un format tabelar, timpii de servire
# și timpii de așteptare ai persoanelor
# ts = o listă cu timpii individuali de serivre
def afisareTimpi(ts):
    print("Persoana\tTimp de servire\tTimp de asteptare")
    # timpul de așteptare al persoanei curente
    tcrt = 0
    # timpul total de așteptare
    ttotal = 0
    for t in ts:
        tcrt = tcrt + t[1]
        ttotal = ttotal + tcrt
        print(str(t[0]).center(len("Persoana")),
              str(t[1]).center(len("Timp de serivire")),
              str(tcrt).center(len("Timp de așteptare")), sep="\t")
    print("Timpul mediu de așteptare:", round(ttotal/len(ts), 2))


# timpii de servire ai persoanelor se citesc de la tastatură
aux = [int(x) for x in input("Timpii de servire: ").split()]
# asociem fiecărui timp de servire numărul de ordine al persoanei
tis = [(i+1, aux[i]) for i in range(len(aux))]
```

```
print("Varianta inițială:")
afisareTimpi(tis)

# sortăm persoanele în ordinea crescătoare a timpilor de servire
tis.sort(key=cheieTimpServire)

print("\nVarianta optimă:")
afisareTimpi(tis)
```

Evident, complexitatea algoritmului este dată de complexitatea operației de sortare utilizate, deci complexitatea sa, optimă, este $\mathcal{O}(n \log_2 n)$.

Încheiem prezentarea acestei probleme precizând faptul că este o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră n activități cu duratele $t_1, t_2, \dots, t_n$ care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă* (*respectiv, la un moment dat, resursa comună poate fi alocată unei singure activități*)*, să se determine o modalitate de planificare a activităților astfel încât timpul mediu de așteptare să fie minim.*".