Laborator 02 : Greedy

Responsabili:

- Alex Rotaru [mailto:rotarualexandruandrei94@gmail.com]
- Darius Neațu [mailto:neatudarius@gmail.com]
- Radu Vișan [mailto:visanr95@gmail.com]
- Cristian Banu [mailto:cristb@gmail.com]

Obiective laborator

- Înțelegerea noțiunilor de bază legate de tehnica greedy
- Însușirea abilităților de implementare a algoritmilor bazați pe greedy

Precizari initiale

Toate exemplele de cod se gasesc in arhiva demo-lab02.zip.

Acestea apar incorporate si in textul laboratorului pentru a facilita parcurgerea cursiva a laboratorului.

- Toate bucatile de cod prezentate in partea introductiva a laboratorului (inainte de exercitii) au fost testate. Cu toate acestea, este posibil ca din cauza mai multor factori (formatare, caractere invizibile puse de browser etc) un simplu copy-paste sa nu fie de ajuns pentru a compila codul.
- Va rugam sa incercati si codul din arhiva **demo-lab02.zip**, inainte de a raporta ca ceva nu merge. :D
- Pentru orice problema legata de continutul acestei pagini, va rugam sa dati email unuia dintre responsabili.

Importanță – aplicații practice

În general tehnicile de tip Greedy sau Programare Dinamică (lab04) sunt folosite pentru rezolvarea problemelor de optimizare. Acestea pot adresa probleme în sine sau pot fi subprobleme dintr-un algoritm mai mare. De exemplu, algoritmul Dijkstra pentru determinarea drumului minim pe un graf alege la fiecare pas un nod nou urmărind algoritmul greedy.

Exista însă probleme care ne pot induce în eroare. Astfel, există probleme în care urmărind criteriul Greedy nu ajungem la soluția optimă. Este foarte important să identificăm cazurile când se poate aplica Greedy și cazurile când este nevoie de altceva. Alteori această soluție neoptimă este o aproximare suficientă pentru ce avem nevoie. Problemele NP-complete necesita multă putere de calcul pentru a găsi optimul absolut. Pentru a optimiza aceste calcule mulți algoritmi folosesc decizii Greedy și găsesc un optim foarte aproape de cel absolut.

Greedy

“greedy” = “lacom”. Algoritmii de tip greedy vor să construiască într-un mod cât mai rapid soluția unei probleme. Ei se caracterizează prin luarea unor decizii rapide care duc la găsirea unei soluții potențiale a problemei. Nu întotdeauna asemenea decizii rapide duc la o soluție optimă; astfel ne vom concentra atenția pe identificarea acelor anumite tipuri de probleme pentru care se pot obține soluții optime. În general exista mai multe soluții posibile ale problemei. Dintre acestea se pot selecta doar anumite soluții optime, conform unor anumite criterii. Algoritmii greedy se numără printre cei mai direcți algoritmi posibili. Ideea de bază este simplă: având o problema de optimizare, de calcul al unui cost minim sau maxim, se va alege la fiecare pas decizia cea mai favorabilă, fără a evalua global eficiența soluţiei. Scopul este de a găsi una dintre acestea sau dacă nu este posibil, atunci o soluție cât mai apropiată, conform criteriului optimal impus.

Trebuie înțeles faptul ca rezultatul obținut este optim doar dacă un optim local conduce la un optim global. În cazul în care deciziile de la un pas influențează lista de decizii de la pasul următor, este posibila obținerea unei valori neoptimale. În astfel de cazuri, pentru găsirea unui optim absolut se ajunge la soluții supra-polinomiale. De aceea, dacă se optează pentru o astfel de soluție, algoritmul trebuie însoțit de o demonstrație de corectitudine. Descrierea formală a unui algoritm greedy este următoarea:

```
// C este mulțimea candidaților
function greedy(C) {
    S ← ∅ // în S construim soluția

    while !solutie(C) and C ≠ ∅
        x ← un element din C care minimizează/maximizează select(x)
        C ← C \ {x}
        if fezabil( S ∪ {x}) then S ← Su{x}

    return S
}
```

Este ușor de înțeles acum de ce acest algoritm se numește ”greedy”: la fiecare pas se alege cel mai bun candidat de la momentul respectiv, fără a studia alternativele disponibile în moment respectiv şi viabilitatea acestora în timp.

Dacă un candidat este inclus în soluție, rămâne acolo, fără a putea fi modificat, iar dacă este exclus din soluție, nu va mai putea fi niciodată selectat drept un potențial candidat.

Exemple

Simple task

Enunt

Fie un șir de N numere pentru care se cere determinarea unui subșir de numere cu suma maximă. Un subșir al unui șir este format din elemente (nu neapărat consecutive) ale șirului respectiv, în ordinea în care acestea apar în șir.

Pentru numerele $1, -5, 6, 2, -2, 4$ răspunsul este $1, 6, 2, 4$ (suma 13).

Solutie

Se observa ca tot ce avem de făcut este sa verificam fiecare număr dacă este pozitiv sau nu. În cazul pozitiv, îl introducem în subșirul soluție.

Daca toate numerele sunt negative, solutia este data de cel mai mare numar negativ (cel mai mic in modul).

Problema spectacolelor

Enunt

Se dau mai multe spectacole, prin timpii de start şi timpii de final. Se cere o planificare astfel încât o persoană să poată vedea cât mai multe spectacole.

Solutie

Rezolvarea constă în **sortarea** spectacolelor crescător după timpii de final, apoi la fiecare pas se alege **primul spectacol** care are timpul de start mai mare decât ultimul timp de final. Timpul inițial de final este inițializat la $-\inf$ (spectacolul care se termină cel mai devreme va fi mereu selectat, având timp de start mai mare decât timpul inițial).

Implementare

```
bool end_hour_comp (pair<int, int>& e1, pair<int, int>& e2) {
    // comparam doar dupa ora de sfarsit
    return (e1.second < e2.second);
}

vector<pair<int, int>> plan(vector<pair<int, int> >& intervals) {
    vector<pair<int, int>> plan;
    // se sorteaza intervalele pe baza orei de sfarsit a spectacolelor
    sort(intervals.begin(), intervals.end(), end_hour_comp);
```

```
    // se ia ultimul spectacol ca terminat la -oo pt a putea incepe cu
    // cel mai devreme
    int last_end = INT_MIN; // -oo a.k.a -infinit
    for (auto interval : intervals) {
        // daca inceputul intervalului curent este dupa sfarsitul ultimului
        // spectacol (last_end) il adaugam in lista de spectacole la care
        // se participa
        if (interval.first >= last_end)
        {
            plan.push_back(interval);
            // dupa ce am adaugat un spectacol, updatam ultimul sfarsit de spectacol
            last_end = interval.second;
        }
    }
    return plan;
}
```

Complexitate

Solutia va avea urmatoarele complexitati:

- **complexitate temporala** : $T(n) = O(n * log(n))$
  - explicatie
    - sortarea are $O(n * log(n))$
    - facem inca o parcurgere in $O(n)$
- **complexitate spatiala** : depinde de algoritmul de sortare folosit.

Problema florarului

Enunt

Se da un grup de k oameni care vor sa cumpere impreuna n flori. Fiecare floare are un pret de baza, insa pretul cu care este cumparata variaza in functie de numarul de flori cumparate anterior de persoana respectiva. De exemplu daca George a cumparat 3 flori (diferite) si vrea sa cumpere o floare cu pretul 2, el va plati $(3 + 1) * 2 = 8$. Practic el va plati un pret proportional cu numarul flori cumparate pana atunci tot de el.

Cerinta: Se cere pentru un numar k de oameni si n flori se cere sa se deterimne care este costul minim cu care grupul poate sa achizitioneze toate cele n flori o singura data.

Observatie: Un tip de floare se cumpara o singura data. O persoana poate cumpara mai multe tipuri de flori. In final in grup va exista un singur exemplar din fiecare tip de floare.

Formal avem k numar de oameni, n numar de flori, $c[i]$ = pretul florii de tip i, costul de cumparare i va fi $(x + 1) * c[i]$, unde x este numarul de flori cumparate anterior de persoana respectiva.

n=3 k=3 c=[2 5 6]

Cost minim = 13

Explicatie: Fiecare individ cumpara cate o floare si deci acestea se cumpara la pretul nominal.

Solutie

Se observa ca pretul efectiv de cumpare va fi mai mare cu cat cumparam acea floare mai tarziu. Daca consideram cazul in care avem o singura persoana in grup observam ca are sens sa cumparam obiectele in ordine descrescatoare(deoarece vrem sa minimizam costul fiecarui tip de flori si aceste creste cu cat cumparam floarea mai tarziu).

De aici, gandindu-ne la versiunea cu k persoane, observam ca ar fi mai ieftin daca am repartiza urmatoarea cea mai scumpa floare la alt individ. Deci impartim florile sortate descrescator dupa pret in grupuri de cate k, fiecare individ luand o floare din acest grup si ne asiguram ca pretul va creste doar in functie de numarul de grupuri anterioare.

Implementare

```
struct greater_comparator
{
    template<class T>
    bool operator()(T const &a, T const &b) const { return a > b; }
};

int minimum_cost(int k, vector<int>& costs) {
    // sortam vectorul de preturi in ordine descrescatoare
    sort(costs.begin(), costs.end(), greater_comparator());

    // numarul de flori cumparate de fiecare individ din grup la un moment dat
    int x = 0;
    // o varianta mai putin eficienta spatial ar fi fost sa retinem pt fiecare
    // individ din grup numarul de flori cumparate intr-un hashmap
    // costul total
    int total_cost = 0;

    // parcurgem fiecare pret de floare si o "asignam" unui individ din grup
    // pretul acesteia fiind proportional cu numarul de achizitii facut pana acum
    // de acesta (x)
    for (int idx = 0; idx < costs.size(); idx++) {
        int customer_idx = idx % k;
        total_cost += (x + 1) * costs[idx];
        // in momentul in care ultimul individ a cumparat o floare din grupul curent
        // incrementam numarul de flori achizitionate de fiecare
        if (customer_idx == k - 1) {
            x += 1;
        }
    }
    return total_cost;
}
```

Complexitate

Solutia va avea urmatoarele complexitati:

- **complexitate temporala** : $T(n) = O(n * log(n))$
  - explicatie
    - sortarea are $O(n * log(n))$
    - facem inca o parcurgere in $O(n)$
- **complexitate spatiala** : depinde de algoritmul de sortare folosit. Fara partea de sortare, spatiul este constant (nu se ia in considerare vectorul de elemente).

Problema cuielor

Enunt

Fie N scânduri de lemn, descrise ca niște **intervale închise** cu capete reale. Găsiți o mulțime **minimă** de cuie astfel încât fiecare scândură să fie bătută de cel puțin un cui. Se cere poziția cuielor.

Formulat matematic: găsiți o mulțime de puncte de cardinal **minim** M astfel încât pentru orice interval $[a_i, b_i]$ din cele N, să existe un punct x din M care să aparțină intervalului $[a_i, b_i]$.

- intrare: N = 5, intervalele: [0, 2], [1, 7], [2, 6], [5, 14], [8, 16]
- ieșire: M = {2, 14}
- explicație: punctul 2 se afla în primele 3 intervale, iar punctul 14 în ultimele 2

Solutie

Se observa că dacă x este un punct din M care nu este capăt dreapta al nici unui interval, o translație a lui x la dreapta care îl duce în capătul dreapta cel mai apropiat nu va schimba intervalele care conțin punctul. Prin urmare, exista o mulțime de cardinal minim M pentru care toate punctele x sunt capete dreapta.

Astfel, vom crea mulțimea M folosind numai capete dreapta în felul următor:

- sortăm intervalele dupa capatul dreapta

- iterăm prin fiecare interval și dacă intervalul curent nu conține ultimul punct introdus în mulțime atunci îl adăugam pe acesta la mulțime

Implementare

```
bool point_in_interval(const pair<int, int>&interval, int point) {
    return point >= interval.first && point <= interval.second;
}

bool right_edge_comparator (pair<int, int>& e1, pair<int, int>& e2) {
    // comparam scandurile dupa capatul drepata
    return (e1.second < e2.second);
}

vector<int> cover_intervals_greedy(vector<pair<int, int>>& intervals) {
    vector<int> nails; // pozitiile cuielor, a.k.a multimea M
    // ultimul punct inserat
    int last_point = INT_MIN;

    //sortam invervalele dupa capatul drepata
    sort(intervals.begin(), intervals.end(), right_edge_comparator);

    for (auto interval : intervals) {
        // daca intervalul nu contine ultimul punct adaugat
        if (!point_in_interval(interval, last_point)) {
            // il adaugam in multimea M
            nails.push_back(interval.second);
            // updatam ultimul punt inserat
            last_point = interval.second;
        }
    }

    return nails;
}
```

Complexitate

Soluția va avea următoarele complexități:

- **complexitate temporala** : $T(n) = O(n * log(n))$
    - explicație
        - sortare: $O(n * logn)$
        - parcurgerea intervalelor: $O(n)$
- **complexitate spațială** : depinde de algoritmul de sortare folosit.

Concluzii şi observații

Aspectul cel mai important de reținut este că soluțiile găsite trebuie să reprezinte optimul global și nu doar local. Se pot confunda ușor problemele care se rezolvă cu Greedy cu cele care se rezolvă prin Programare Dinamică (vom vedea saptamana viitoare).

Exercitii

In acest laborator vom folosi scheletul de laborator din arhiva skel-lab02.zip.

ATENTIE! Au aparut modificari minore la checker. Rulati comanda `./check.sh` si recititi documentatia afisata.

Rucsac

Fie un set cu n obiecte (care pot fi **taiate** - varianta continua a problemei). Fiecare obiect i are asociata o pereche (w_i, p_i) cu semnificatia:

- w_i = $weight_i$ = greutatea obiectului cu numarul i

- p_i = $price_i$ = pretul obiectului cu numarul i
    - $w_i >= 0$ si $p_i > 0$

Gigel are la dispozitie un rucsac de **volum infinit**, dar care suporta o **greutate maxima** (notata cu W - weight knapsack).

El vrea sa gaseasca **o submultime de obiecte** (nu neaparat intregi) pe care sa le bage in rucsac, astfel incat suma profiturilor sa fie **maxima**.

Daca Gigel baga in rucsac obiectul i, caracterizat de (w_i, p_i), atunci profitul adus de obiect este p_i (presupunem ca il vinde cu cat valoareaza).

In aceasta varianta a problemei, Gigel poate taia oricare dintre obiecte, obtinand o proportie din acesta. Daca Gigel alege alege doar x din greutatea w_i a obiectului i, atunci el castiga doar $\frac{x}{w_i} * p_i$.

Task-uri:

- Sa se determine profitul maxim pentru Gigel.
- Care este complexitatea solutiei (timp + spatiu)? De ce?

obiecte:

index	0	1	2
greutate	60	100	120
valoare	10	20	30

greutate = 50

Output: 12.5 Explicatie: avem 50 capacitate si toate obiectele au o greutatate mai mare, decidem sa luam cat putem din produsul cu raportul valoare / greutate cel mai mare. profitu = 30 / 120 * 50 = 12.5

obiecte:

index	0	1	2
greutate	20	50	30
valoare	60	100	120

greutate = 50

Output: 180 Explicatie: Sortam obiectele dupa raportul valoare profit si avem in ordine: {30, 120}, {20, 60}, {50, 100} Introducem obiecte pana cand umplem sacul ⇒ intra primele 2 obiecte. Calculam profitul 120 + 60 = 180

Distante

Consideram 2 localitati A si B aflate la distanta D. Intre cele 2 localitati avem un numar de n benzinarii, date prin distanta fata de localitatea A. Masina cu care se efectueaza deplasarea intre cele 2 localitati poate parcurge maxim m kilometri avand rezervorul plin la inceput. Se doreste parcurgerea drumului cu un numar minim de opriri la benzinarii pentru realimentare (dupa fiecare oprire la o benzinarie, masina pleaca cu rezervorul plin).

Distantele catre benzinarii se reprezinta printr-o lista de forma $0 < d1 < d2 < \ldots < dn$, unde di ($1 <= i <= n$) reprezinta distanta de la A la benzinaria i. Pentru simplitate, se considera ca localitatea A se afla la 0, iar $dn = D$ (localitatea B se afla in acelasi loc cu ultima benzinarie).

Se garanteaza ca exista o planificare valida a opririlor astfel incat sa se poata ajunge la localitatea B.

[Greedy] Se alimentează doar dacă nu se poate ajunge la benzinăria următoare.

$n = 5$

$m = 10$

$d = (2, 8, 15, 25, 30)$

Raspunsul este 3, efectuand 3 opriri la a 2-a, a 3-a, respectiv a 4-a benzinarie.

Teme la ACS

Pe parcursul unui semestru, un student are de rezolvat n teme (nimic nou pana aici…). Se cunosc enunțurile tuturor celor n teme de la **începutul semestrului**.

Timpul de rezolvare pentru oricare dintre teme este de **o săptămână** și **nu** se poate lucra la mai multe teme în același timp. Pentru fiecare tema se cunoaște un termen limita $d[i]$ (exprimat în săptămâni - deadline pentru tema i) și un punctaj $p[i]$.

Nicio fracțiune din punctaj nu se mai poate obține după expirarea termenului limită.

Task-uri:

- Să se definească o planificare de realizare a temelor, în așa fel încât punctajul obținut să fie **maxim**.
- Care este complexitatea solutiei (timp + spatiu)? De ce?

index	0	1	2	3	4
deadline	6	6	2	7	7
punctaj	5	4	1	5	8

Output: $1 + 4 + 5 + 5 + 8 = 23$

Explicatie: Putem face toate temele deoarece pana ajungem la deadline-urile lor avem suficiente unitati de timp.

index	0	1	2	3	4	5	6	7
deadline	3	3	3	3	9	11	11	11
punctaj	4	9	6	5	10	4	2	6

Output: $5 + 6 + 9 + 10 + 2 + 4 + 6 = 42$

Explicatie: Pana in deadline 3 avem la dispozitie 3 unitati de timp si 4 teme. Deci sortam dupa punctaj si le includem pe cele mai valoroase: 5, 6 ,9. Pana la deadline 9 avem la dispoztie 6 unitati de timp si 4 teme. Le includem pe toate.

BONUS

Rezolvati problema Dishonest Sellers [http://codeforces.com/problemset/problem/779/C].

Hint: aici [http://codeforces.com/blog/entry/50724].

Extra

Incercati problema MaxSum [https://www.hackerrank.com/contests/test-practic-pa-2017-v1-plumbus/challenges/1-1-usoare] de la test PA 2017.

Problema 1 de la tema PA 2017. Puteti descarca enuntul si checkerul de aici [https://ocw.cs.pub.ro/courses/_media/pa/teme/pa2017_tema1.zip].

Problema 3 de la tema PA 2017. Puteti descarca enuntul si checkerul de aici [https://ocw.cs.pub.ro/courses/_media/pa/teme/pa2017_tema1.zip].

Referințe

[0] Capitolul **Greedy Algorithms** din **Introductions to Algorithms** de către T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein

[1] http://en.wikipedia.org/wiki/Greedy_algorithm [http://en.wikipedia.org/wiki/Greedy_algorithm]

[2] http://ww3.algorithmdesign.net/handouts/Greedy.pdf [http://ww3.algorithmdesign.net/handouts/Greedy.pdf]