Lucrarea 7 – Greedy.

Introducere tehnica Greedy

Este greu de spus forma generală a unei probleme rezolvabile cu tehnică Greedy. Algoritmii *greedy* (greedy = lacom) sunt în general simpli şi sunt folosiți la probleme de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum într-un graf etc. In cele mai multe situații de acest fel avem:

- o mulțime de *elemente* (lucrări de executat, vârfuri ale grafului etc)
- o funcție care verifică dacă o anumită mulțime de candidați constituie o *soluție posibilă*, nu neapărat optimă, a problemei
- o funcție care verifică dacă pentru o mulțime de candidați este posibil să o completăm astfel încât să obținem o soluție posibilă, nu neapărat optimă, a problemei
- o *funcție de selecție* care alege la orice moment cel mai potrivit element, încă nefolosit
- o *funcție soluție* care spune când s-a ajuns la soluție.

Pentru a rezolva problema o anumită problemă, un algoritm greedy construiește soluția pas cu pas. Inițial, mulțimea candidaților selectați este vidă. La fiecare pas, încercăm să adăugăm acestei mulțimi cel mai potrivit element, conform funcției de selecție. Dacă, după o astfel de adăugare, mulțimea de elemente obținută nu mai poate duce la soluție, se elimină ultimul element adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulțimea de elemente selectate poate duce la soluție, ultimul element adăugat va rămâne de acum încolo în ea. De fiecare dată când lărgim mulțimea elementelor selectate, verificăm dacă această mulțime nu constituie o soluție posibilă a problemei noastre.

Dacă algoritmul Greedy funcționează corect, prima soluție găsită va fi totodată o soluție optimă a problemei.

O întrebare firească este aceea dacă algoritmul Greedy duce totdeauna la soluție optimă? Evident că nu Sunt situații când soluția găsită nu este optimă. Mai mult, pentru cele mai multe din probleme nu se cunosc algoritmi Greedy de rezolvare. Spre deosebire de Backtracking, algoritmul Greedy nu permite atunci când s-a oservat că nu se poate ajunge la soluție pentru o anumită secvență de elemente, revenirea înapoi, pe nivelele anterioare.

Pentru problemele care nu duc la soluția optimă este necesar să se caute soluții, chiar dacă nu optime, dar cât mai apropiate de acestea.

Descrierea în pseudocod al algoritmului de rezolvare este următoarea:

```
function greedy(C)
    {C este multimea candidatilor}
    S ← ∅    {S este multimea in care construim solutia}
    while not solutie(S) and C ≠ ∅ do
        x ← un element din C care maximizeaza/minimizeaza
             select(x)
        C ← C \ {x}
        if corect(S ∪ {x})
                        then S ← S ∪ {x}
   if  solutie(S)
```

```
                    then   return S
            else   return "nu exista solutie"
```

La fiecare pas, procedura alege cel mai bun candidat la momentul respectiv, fără să-i pese de viitor şi fără să se răzgândească. Daca un candidat este inclus în soluție, el rămâne acolo;

dacă un candidat este exclus din soluție, el nu va mai fi niciodată reconsiderat. Asemenea unui întreprinzător rudimentar care urmărește câştigul imediat în dauna celui de perspectivă, un algoritm Greedy acționează simplist. Totuşi, ca şi în afaceri, o astfel de metodă poate da rezultate foarte bune tocmai datorită simplității ei.

Între metoda Greedy şi metoda Backtracking putem enumera următoarele diferențe:

- ambele tehnici oferă soluții sub formă de vector
- tehnica Backtracking oferă toate soluțiile problemei, în timp ce *Greedy oferă o singură soluție*
- tehnica Greedy nu dispune de mecanismul întoarcerii, specific tehnici Backtracking

Exemplu de problemă rezolvată prin metoda Greedy

Să presupunem că deținem un magazin de soft şi dorim să dăm restul unui client care ne-a dat o sumă oarecare de bani, folosind un număr cât mai mic de monezi. Presupunem că avem următoare listă de monede disponibile: 100lei, 50lei, 25lei, 1leu. Moneda unitate (1 leu) este obligatoriu să se regăsească printre ele. De asemenea din fiecare tip de monedă avem un număr nelimitat de bucăți.

```
        void function Greedy(int suma)
//se primeşte ca parametru suma pe care trebuie să o dea rest
        {
          int nr_monezi;
//initial dam cat se poate din bani în moneda cea mai mare (de 100)
  Deoarece atat suma cat si 100 sunt numere întregi, suma/100 va avea
  ca rezultat un numar întreg, partea de după virgulă ignorându-se
          nr_monezi = suma/100;
          printf("s-au plătit %d monede de 100",nr_monezi);
//restul care mai este de achitat este de fapt restul împărțirii
  suma, la 100
          suma = suma%100;
//se plăteşte cât este posibil în moneda următoare
          nr_monezi = suma/50;
          printf("s-au plătit %d monede de 50",nr_monezi);
          suma = suma%50

          nr_monezi = suma/25;
          printf("s-au plătit %d monede de 100",nr_monezi);
          suma = suma%25;

//Deoarece nu mai avem la dispoziție decât moneda unitate, acum se va
  plăti restul care mai este de dat în monede de 1 leu
          printf("s-au plătit %d monede de 1",suma);
        }
```

Se poate demonstra că algoritmul Greedy va găsi în acest caz mereu soluția optimă (restul cu un număr minim de monezi).

Dacă în schimb nu am avea moneda unitate, nu s-ar mai putea o obține soluție optimă. Este posibil ca după ce am plătit tot ce se poate în monedele disponibile, folosind algoritmul de mai sus, să ne rămână o sumă mică (mai mică decât cea mai mică monedă) pe care nu avem cu ce să o restituim. Cum algoritmul Greedy nu dispune de mecanismul întoarcerii pentru a încerca altă combinație de monede pentru restituirea restului, acea sumă…. ne va rămâne ca bacşiş.

Probleme propuse

1. să se rescrie problema casierului în condiţiile în care am avea un număr citit de la tastatură de tipuri de monede. (de asemenea trebuie citit pentru fiecare tip în parte valoarea acelei monede)

2. Într-o sală de teatru, într-o anumită zi trebuie planificate n spectacole. Pentru fiecare spectacol se cunoaşte intervalul în care se desfăşoară [incep…termin]. Se cere să se planifice un număr maxim de spectacole astfel încât să nu se suprapună.

Indicație: se folosește un vector bidimensional spectacol[i][j], unde i reprezintă ora începerii spectacolului, iar j ora terminării lui. Se sortează spectacolele după ora terminării lui. Primul spectacol este programat cel care se termină cel mai devreme. Mai departe alegem primul spectacol care îndeplineşte condiția că începe după ce s-a terminat cel anterior lui.

Acest algoritm găseşte soluția optimă?

3. O persoană are un rucsac cu care poate transporta o greutate maximă G. Persoana are la dispoziție n obiecte şi cunoaşte pentru fiecare obiect greutatea şi valoarea sa. Se pune problema ce obiecte trebuie să transport persoana în aşa fel încât câştigul să fie maxim.

Modificați rezolvarea astfel încât să corespundă situației când persoana nu este obligată să ia un obiect întreg, ci poate lua doar un fragment din el.

4. Se dau n numere întregi nenule $b_1, b_2 \dots b_n$ şi m numere întregi nenule $a_1, a_2 \dots a_m$. Să se determine o submulțime a mulțimii $B= \{b_1, b_2 \dots b_n\}$ care să maximizeze valorile expresiei:

$$E = a_1x_1 + a_2x_2 + \dots + a_mx_m$$

unde $n > m$ şi $x_i \in \{b_1, b_2 \dots b_n\}$

5. Un comis-voiajor pleacă dintr-un oraş, trebuie să viziteze un număr de n oraşe şi să se întoarcă în oraşul de plecare cu efort minim. Se da matricea de vecinătăți a celor n oraşe, precizându-se şi distanțele dintre ele.

6. Se dă o tablă de şah n×n. Să se arate punctele prin care trece un cal care porneşte din punctul 1,1 în încercarea de a acoperi cât mai multe puncte posibile.

Indicație: la fiecare pas se alege acea mutare care plasează calul într-o poziție din care la pasul următor există cât mai puține posibilități de a muta din nou