Curs 13 - Technici de programare

- **Greedy**
- **Programare dinamică**

Curs 12 – Technici de programare

- **Divide-et-impera (divide and conquer)**
- **Backtracking**

Metoda Greedy

- o strategie de rezolvare pentru probleme de optimizare
- aplicabil unde optimul global se poate afla selectând succesiv optime locale
- permite rezolvarea problemei fara a reveni asupra alegerilor facute pe parcurs
- folosit în multe probleme practice care necesite selecția unei mulțimi de elemente care satisfac anumite condiții și realizează un optim

Probleme

Problema rucsacului

Se dă un set de obiecte, caracterizate prin greutate și utilitate, și un rucsac care are capacitatea totala W. Se caută o submultime de obiecte astfel încât greutatea totală este mai mică decât W și suma utilității obiectelor este maximal.

Monede

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un numar nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Forma generală a unei probleme Greedy-like

Avănd un set de obiecte C candidați pentru a face parte din soluție, se cere să se găsească un subset B (B $\subseteq$ C) care indeplinește anumite condiții (condiții interne) și maximizează (minimizează) o funcție de obiectiv.

- Dacă un subset *X* îndeplinește condițiile interne atunci subsetul X este *acceptabil (*posibil)
- Unele probleme pot avea mai multe soluții acceptabile, caz în care se caută o soluție cât mai bună, daca se poate solutia ceea mai bună (cel care realizeaza maximul pentru o funcție obiectiv).

Pentru a putea rezolva o problema folosind metoda Greedy, problema trebuie să satisfacă proprietatea:

- dacă *B* este o soluție acceptabilă și $X \subset B$ atunci și *X* este o soluție acceptabilă

Algoritmul Greedy

Algoritmul Greedy găsește soluția incremetal, construind soluții acceptabile care se tot extind pe parcurs. La fiecare pas soluția este exstinsă cu cel mai bun candidat dintre candidații rămași la un moment dat. (Strategie greedy - lacom)

Principiu (strategia) Greedy :

- adaugă succesiv la rezultat elementul care realizează optimul local
- o decizie luată pe parcurs nu se mai modifică ulterior

Algoritmul Greedy

- Poate furniza soluția optimă (doar pentru anumite probleme)
  - alegerea optimului local nu garanteaza tot timpul optimul global
  - solutie optimă - dacă se găsește o modalitate de a alege (optimul local) astfel încat se ajunge la solutie optimă
  - în unele situații se preferă o soluție, chiar și suboptimă, dar obținut în timp rezonabil
- Construiește treptat soluția (fără reveniri ca în cazul backtracking)
- Furnizează o singură soluție
- Timp de lucru polinomial

Greedy – python

```
def greedy(c):
    """
       Greedy algorithm
       c - a list of candidates
       return a list (B) the solution found (if exists) using the greedy
strategy, None if the algorithm
       selectMostPromissing - a function that return the most promising
candidate
       acceptable - a function that returns True if a candidate solution can be
extended to a solution
       solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Algoritm Greedy - elemente

1. **Mulțimea candidat** (***candidate set)*** **–** de unde se aleg elementele soluției
2. **Funcție de selecție** (***selection function)*** **–** alege cel mai bun candidat pentru a fi adăugat la soluție;
3. **Acceptabil** (***feasibility function)*** **–** folosit pentru a determina dacă un candidat poate contribui la soluție
4. **Funcţie obiectiv** (***objective function)*** – o valoare pentru soluţie şi pentru orice soluţie parţială
5. **Soluţie (*solution function)***, - indică dacă am ajuns la soluţie

Exemplu

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un numar nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Set Candidat:

Lista de monede - COINS = {1, 5, 25, 50}

Soluţie Candidat:

o listă de monede - $X=(X_0, X_1, .., X_k)$ **unde** $X_i \in COINS$ **– monedă**

Funcția de selecție:

candidate solution: $X=(X_0, X_1, .., X_k)$

alege moneda cea mai mare care e mai mic decăt ce mai e de plătit din sumă

Acceptabil (*feasibility function)*:

Soluție candidat: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i \leq M$

suma monedelor din soluția candidat nu depășește suma cerută

Soluție:

Soluție candidat: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i = M$

suma monedelor din soluția candidat este egal cu suma cerută

Monede – cod python

```
#Let us consider that we have a sum M of money and coins of 1, 5, 25 units (an unlimited number of coins).
#The problem is to establish a modality to pay the sum M using a minimum number of coins.
```

```
def selectMostPromissing(c):
    """
      select the largest coin from the remaining
      c - candidate coins
      return a coin
    """
    return max(c)
```

```
def acceptable(b):
    """
   verify if a candidate solution is valid
   basically verify if we are not over the sum M
    """
    sum = _computeSum(b)
    return sum<=SUM
```

```
def solution(b):
    """
   verify if a candidate solution is an actual solution
   basically verify if the coins conduct to the sum M
   b – candidate solution
    """
    sum = _computeSum(b)
    return sum==SUM

def _computeSum(b):
    """
   compute the payed amount with the current candidate
    return int, the payment
   b – candidate solution
    """
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        #if this is in a candidate solution we need to
use at least 1 coin
        if nrCoins==0: nrCoins =1
        sum += nrCoins*coin
    return sum
```

```
def printSol(b):
 """
   Print the solution: NrCoinns1 * Coin1 +  NrCoinns2 *
Coin2 +...
 """
    solStr = ""
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        solStr+=str(nrCoins)+"*"+str(coin)
        sum += nrCoins*coin
        if SUM-sum>0:solStr+=" + "
    print solStr
```

Greedy

1. Algoritmul Greedy are complexitate polinomială - $O(n^2)$ unde n este numărul de elemente din lista candidat C
2. Înainte de a aplica Greedy este nevoie să demonstrăm că metoda găsește soluția optimă. De multe ori demonstrația este netrivială
3. Există o mulțime de probleme ce se pot rezolva cu greedy. Ex: Algoritmul Kruskal – determinarea arborelui de acoperire, Dijkstra, Bellman-Kalaba – drum minim întrun graf neorientat
4. Există probleme pentru care Greedy nu găsește soluția optimă. În unele cazuri se preferă o soluție obținut în timp rezonabil (polinomial) care e aprope de soluția optimă, în loc de soluția optimă obținută în timp exponențial (*heuristics algorithms)*.

Programare Dinamică

Se poate folosi pentru a rezolva probleme de optimizare, unde:

- soluția este rezultatul unui șir de decizii, $d_1, d_2, \ldots, d_n$,
- *principiul optimalității* este satisfăcut.
- în general timp polinomial de execuție
- tot timpul găsește soluția optimă.
- Rezolvă problema combinân sub soluții de la subprobleme, calculează subsoluția doar o singură data (salvând rezultatul pentru a fi refolosit mai târziu).

Fie stăritle $s_0, s_1, \ldots, s_{n-1}, s_n$, unde s_0 este starea inițială, s_n este starea finală, prin aplicariea succesivă a deciziilor $d_1, d_2, \ldots, d_n$ se ajunge la starea finală (decizia d_i duce din starea s_{i-1} în starea s_i, pentru i=1,n):

$$s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \rightarrow \overset{\ldots}{\ldots} \rightarrow s_{n-1} \xrightarrow{d_n} s_n$$

Programare Dinamică

Caracteristici:

- principiul optimalității;
- probleme suprapuse (overlapping sub problems);
- *memoization*.

Principiul optimalității

- *optimul general* implică *optimul parțial*
  - la greedy aveam *optimul local* implică *optimul global*
- într-o secvență de decizii optime, fiecare decizie este optimă.
- Principiul nu este satisfăcut pentru orice fel de problemă. In general nu e adevărat în cazul in care subsecvențele de decizii nu sunt independente și optimizarea uneia este în conflict cu optimizarea de la alta secvența de decizii.

Principiul optimalității

Dacă $d_1, d_2, ..., d_n$ este o secvență de decizii care conduce optim sistemul din starea inițială s_0 în starea finală s_n, atunci una din utmătoarele sunt satisfăcute:

1). $d_k, d_{k+1}, ..., d_n$ este o secvență optimă de decizii care conduce sistemul din starea s_{k-1} în starea s_n, $\forall k, 1 \le k \le n$. **(*forward* variant –** decizia d_k depinde de deciziile $d_{k+1}..d_n$**)**

2). $d_1, d_2, ..., d_k$ este o secvență optimă de decizii care conduce sistemul din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*backward* variant)**

3). $d_{k+1}, d_{k+2}, ..., d_n$ și $d_1, d_2, ..., d_k$ sunt secvențe optime de decizii care conduc sistemul din starea s_k în starea s_n, respectiv, din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*mixed* variant)**

Sub-Probleme suprapuse (Overlapping Sub-problems)

O problema are sub-probleme suprapuse daca poate fi inpărțit în subprobleme care se refolosesc de mai multe ori

Memorizare (Memorization)
salvarea rezultatelor de la o subproblemă pentru a fi refolosit

Cum aplicăm programare dinamică

- Principiul optimalității (oricare variantă: forward, backward or mixed) este demonstrat.
- Se definește structura solutiei optime.
- Bazat pe principiul optimalității, valoarea soluției optime se definește recursiv. Se definește o recurență care indică modalitatea prin care se opține optimul general din optime parțiale.
- Soluția optimă se calculează in manieră bottom-up, începem cu subproblema cea mai simplă pentru care soluția este cunoscută.

Cea mai lungă sublistă crescătoare

Se dă o listă $a_1, a_2, ..., a_n$. *Determinați cea mai lungă sublistă crescătoare* $a_{i_1}, a_{i_2}, ..., a_{i_s}$ *a listei date.*

Soluție:

- *Pricipiul optimalității*
  - varianta înainte *forward*
- *The structure of the optimal solution*
  - Construim două șiruri: $l =< l_1.l_2,...l_n >$ și $p =< p_1.p_2,...p_n >$.
    - l_k lungime sublistei care incepe cu elementul a_k.
    - p_k indexul elementului a care urmează după elementul a_k în sublista cea mai lungă care incepe cu a_k.
- *Definiția recursivă*

– $l_n = 1,\ p_n = 0$

– $l_k = \max\{1 + l_i \mid a_i \ge a_k,\ k+1 \le i \le n\}, \quad \forall k = n-1, n-2, ...1$

– $p_k = \arg\max\{1 + l_i \mid a_i \ge a_k,\ k+1 \le i \le n\}, \quad \forall k = n-1, n-2, ..., 1$

Cea mai lungă sublistă crescătoare– python

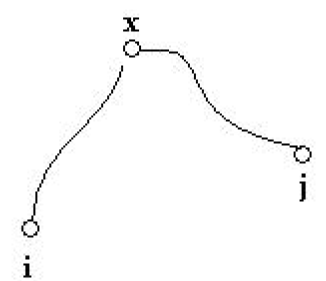
```
def longestSublist(a):
    """
      Determines the longest increasing sub-list
      a -  a list of element
      return sublist of x, the longest increasing sublist

    """
    #initialise l and p
    l = [0]*len(a)
    p = [0]*len(a)
    l[lg-1] = 1
    p[lg-1]=-1
    for k in range(lg-2, -1, -1):
        print p, l
        p[k] = -1
        l[k]=1
        for i in range(k+1, lg):
            if a[i]>=a[k] and l[k]<l[i]+1:
                l[k] = l[i]+1
                p[k] = i

    #identify the longest sublist
    #find the maximum length
    j = 0
    for i in range(0, lg):
        if l[i]>l[j]:
            j=i
    #collect the results using the position list
    rez = []
    while j!=-1:
        rez = rez+[a[j]]
        j = p[j]
    return rez
```

Dynamic programming vs. Greedy

- ambele se aplică în probleme de optimizare
- Greedy : *optimumul general* se opține din optime *partiale* (*locale*)
- DP *optimumul general* implică optimul *partial*.

Determinați drumul cel mai scurt între nodul **i** și **j** într-un graf:

- *Principiul optimalității* este verificat: daca drumul de la **i** la **j** este optimal și trece prin nodul **x**, atunci drumul de la **i** la **x**, și de la **x** la **j** este optimal.



- DP poate fi aplicat pentru a rezolva problema drumului minim
- Greedy nu se poate aplica fiindcă: dacă drumul de la **i** la **x**, și de la **x** la **j** este optim, nu există garanții că drumul cel mai scurt de la **i** la **j** trece prin **x**.