Cursul 1 – Divide et Impera

Scheme de algoritmi = tipare commune pe care le putem aplica in rezolvarea unor probleme similar

⇨ Cunoașterea schemelor determină o rezolvare mai rapidă și mai eficientă a problemelor.

Divide et Impera:

⇨ o clasă de algoritmi care au ca principale caracteristici faptul că împart problema în subprobleme similare cu problema inițială dar mai mici ca dimensiune, rezolvă problemele recursiv și apoi combină soluțiile pentru a crea o soluție pentru problema originală.
⇨ Schema Divide et impera consta in 3 pasi la fiecare nivel al recurentei:
  - Divide problema dată într-un număr de subprobleme
  - Impera (cucerește) – subproblemele sunt rezolvate recursiv. Dacă subproblemele sunt suficient de mici ca date de intrare se rezolvă direct (ieșirea din recurență)
  - Combină – soluțiile subproblemelor sunt combinate pentru a obține soluția problemei inițiale.
⇨ Avantaje:
  - Produce algoritmi eficienti
  - Descompunerea problemei în subprobleme facilitează paralelizarea algoritmului în vederea execuției sale pe mai multe procesoare.
⇨ Dezavantaje:
  - Se adaugă un overhead datorat recursivității (reținerea pe stivă a apelurilor funcțiilor).

⇨ Exemple: **Merge Sort** – exemplu clasic de rezolvare cu D&I
  - **Divide**: Divide cele n elemente ce trebuie sortate în 2 secvențe de lungime n/2.
  - **Impera**: Sortează secvențele recursiv folosind merge sort.
  - **Combina**: Secvențele sortate sunt asamblate pentru a obține vectorul sortat.
  - Recurența se oprește când secvența ce trebuie sortată are lungimea 1 (un vector cu un singur element este întotdeauna sortat ☺) .

```
MERGE(A, p, q, r) // p si r sunt capetele intervalului, q este "mijlocul"
1       n1 ← q - p + 1 // numărul de elemente din partea stânga
2       n2 ← r – q  // numărul de elemente din partea dreapta
3       creează vectorii S[1 -> n1 + 1] și D[1 -> n2 + 1]
4       Pentru i de la 1 la n1
5               S[i] ← A[p + i - 1] // se copiază partea stânga în S
6       Pentru j de la 1 la n2
7               D[j] ← A[q + j] // si partea dreapta în D
8       S[n1 + 1] ← ∞
9       D[n2 + 1] ← ∞
10      i ← 1
11      j ← 1
12      Pentru k de la p la r  // se copiază înapoi în vectorul de
13              Dacă S[i] ≤ D[j]        // sortat elementul mai mic din cei
14                      Atunci A[k] ← S[i]   // doi vectori sortați deja
15                              i ← i + 1
16                      Altfel A[k] ← D[j]
17                              j ← j + 1
```

  - **Complexitate**: T(n) = 2T(n/2) + O(n) => T(n) = O(n*log n).

- ⇨ Exemplu 2: **Calculul puterii unui numar**:
  - Dacă n este par:
    Atunci intoarce x^(n/2) * x^(n/2);
    Altfel (n este impar) intoarce x * x^((n-1)/2) * x^((n-1)/2)
  - **Complexitate**: T(n) = T(n/2) + O(1) = O(log n)

- ⇨ Exemplu 3: **Calcului celei mai scurte distante intre 2 puncte din plan**:
  - Sortează punctele în ordinea crescătoare a coordonatei x (Θ(nlog n));
  - Împărțim setul de puncte în 2 seturi de dimensiune egală și calculăm recursiv distanța minimă în fiecare set (l = linia ce împarte cele 2 seturi, d = distanța minimă calculată în cele 2 seturi);
  - Elimină punctele care sunt plasate la distanță de l > d;
  - Sortează punctele rămase după coordonata y;
  - Calculează distanțele de la fiecare punct rămas la cei 6 vecini (nu pot fi mai mulți);
  - Dacă găsește o distantă < d, atunci actualizează d.

Cursul 2 – Greedy

- ⇨ Metodă de rezolvare eficientă a unor probleme de optimizare.
- ⇨ Soluția trebuie să satisfacă un criteriu de optim global (greu de verificat) -> optim local mai ușor de verificat.
- ⇨ Se aleg soluții parțiale ce sunt îmbunătățite repetat pe baza criteriului de optim local până ce se obțin soluții finale.
- ⇨ Soluțiile parțiale ce nu pot fi îmbunătățite sunt abandonate proces de rezolvare irevocabil (fără reveniri)!

```
1. sol_parțiale = sol_inițiale(Problemă); // determinarea soluțiilor parțiale
2. sol_fin = Φ;
3. Cât timp (sol_parțiale ≠ Φ)
4.         Pentru fiecare (s în sol_parțiale)
5.                 Dacă (s este o soluție a problemei) { // dacă e soluție
6.                         sol_fin = sol_fin U {s}; // finală se salvează
7.                         sol_parțiale = sol_parțiale \ {s};
8.                 } Altfel // se poate optimiza?
9.                 Dacă (optimizare_posibilă (s, Crit_optim, Problemă))
10.                        sol_parțiale = sol_parțiale \ {s} U      // da
                                  optimizare(s, Crit_optim, Problemă)
11.                Altfel sol_parțiale = sol_parțiale \ {s}; // nu
12. Întoarce sol_fin;
```

- ⇨ **Comparatie D&I si Greedy**
  - Tip de abordare: D&I: topdown ; Greedy: top-down
  - Criteriu de optim: D&I: nu; Greedy: da
- ⇨ Cand functioneaza algoritmii Greedy?
  - Problema are proprietatea de substructură optima: Soluţia problemei conţine soluţiile subproblemelor.
  - Problema are proprietatea alegerii locale: Alegând soluţia optimă local se ajunge la soluţia optimă global.
  - Teoremă. Dacă (E, I) este matroid atunci există un algoritm Greedy care rezolvă problema de optimizare. Un sistem accesibil este un matroid dacă satisface

proprietatea de interschimbare: oricare X, Y apartine I şi |X| < |Y| => exista e apartine Y \ X a. î. X U {e} apartine I.

-

⇨ Exemplu 1: **Arbori Huffman**

- Construcția unui astfel de arbore se realizează printrun algoritm Greedy.
- 1. Pentru fiecare k din K se construiește un arbore cu un singur nod care conține cheia k si este caracterizat de ponderea w = p(k). Subarborii construiți formează o mulțime numită Arb.
- 2. Se aleg doi subarbori a şi b din Arb astfel încât a şi b au pondere minimă.
- 3. Se construiește un arbore binar cu o rădăcina r care nu conține nici o cheie și cu descendenții a și b. Ponderea arborelui este definită ca w(r) = w(a) + w(b).
- 4. Arborii a și b sunt eliminați din Arb iar r este inserat în Arb.
- 5. Se repetă procesul de construcție descris de pașii 2-4 până când mulțimea Arb conține un singur arbore – Arborele Huffman pentru cheile K.

```
Huffman(K,p){
1. Arb = {frunză(k, p(k)) | k ∈ K};
2. Cât timp (card (Arb) > 1) // am mai mulți subarbori
3.          Fie a1 și a2 arbori din Arb a.i. ∀a ∈ Arb a ≠ a1 si a ≠ a2, avem
                w(a1) ≤ w(a) și w(a2) ≤ w(a));     // practic se extrage
                // de două ori minimul și se salvează în a1 și a2
4.          Arb = Arb \ {a1, a2} U nod_intern(a1, a2, w(a1) + w(a2));
5. Dacă (Arb = Φ)
6.          Întoarce arb_vid;
6. Altfel
7.          fie A singurul arbore din mulțimea Arb;
8.          Întoarce A;
```

```
Decodificare (in, out)
   A = restaurare_arbore (in)        // reconstruiesc arborele
  Cât timp (! terminare_cod(in)) // mai am caractere de citit
      nod = A                          // pornesc din rădăcină
      Cât timp (! frunză(nod))     // cât timp nu am determinat caracterul
          Dacă (bit(in) = 1) nod = dreapta(nod)      // avansez în arbore
              Altfel nod = stânga(nod)
      Scrie (out, cheie(nod))       // am determinat caracterul și îl scriu la
                                                  // ieșire
```

⇨ Exemplu 2: **Problema rucsacului**

Varianta 1: Algoritm Greedy

- sortăm obiectele după raportul v_i/m_i;
- adăugăm fracţiuni din obiectul cu cea mai mare valoare per kg până epuizăm stocul şi apoi adăugăm fracţiuni din obiectul cu valoarea următoare.
- Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4, m_4 = 5 kg, v_4 = 10
- Soluție: (m_2 , v_2) 8kg şi 2kg din (m_1,v_1) sau din (m_4,v_4) – valoarea totală: 19 + 2 * 10 / 5 = 23

Varianta 2: Algoritmul Greedy nu funcţionează => contraexemplu

- Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4, m_4 = 5 kg, v_4 = 10
- Rezultat corect – 2 obiecte: (m_1,v_1) și (m_4,v_4) – valoarea totală: 20
- Rezultat algoritm Greedy – 1 obiect (m_2,v_2) – valoarea totală: 19

Fie I = mulțimea subsoluțiilor și X = {m_2} și Y = {m_1, m_4} → $m_1 \in Y \setminus X$ dar $X \cup \{m_1\} \notin I$ → problema nu respectă proprietatea de matroid → problema nu se poate rezolva folosind tehnica Greedy!

Cursul 3 – Programare dinamica

- ⇨ Descriere generala: Soluții optime construite iterativ asamblând soluții optime ale unor probleme similare de dimensiuni mai mici.
- ⇨ Algortmi clasici:
  - Înmulţirea unui şir de matrici
  - AOC
  - Algoritmul Floyd-Warshall care determină drumurile de cost minim dintre toate perechile de noduri ale unui graf.
  - Numere catalane
  - Viterbi
- ⇨ Algoritm generic:

Programare dinamică (crit_optim, problema)

- // fie $problema_0$ $problema_1$ … $problema_n$ astfel încât
- // $problema_n$ = problema; $problema_i$ mai simplă decât $problema_{i+1}$
- 1. Sol = soluții_inițiale(crit_optim, $problema_0$);
- 2. **Pentru** *i* **de la** 1 **la** *n* // construcție soluții pentru
  // $problema_i$ folosind soluțiile problemelor precedente
- 3. Sol_i = calcul_soluții(Sol, crit_optim, $Problema_i$);
  // determin soluția $problemei_i$
- 4. Sol = Sol U Sol_i;
  // noile soluții se adaugă pentru a fi refolosite pe viitor
- 5. s = soluție_pentru_$problema_n$(Sol);
  // selecție / construcție soluție finală
- 6. **Întoarce** s;

⇨ Caracteritici:
- O soluție optimă a unei probleme conține soluții optime ale subproblemelor.
- Decompozabilitatea recursivă a problemei P în subprobleme similare P = Pn, Pn-1, … P0 care acceptă soluții din ce în ce mai simple.
- Suprapunerea problemelor (soluția unei probleme Pi participă în procesul de construcție a soluțiilor mai multor probleme Pk de talie mai mare k > i) – memoizare (se foloseşte un tablou pentru salvarea soluţiilor subproblemelor cu scopul de a nu le recalcula)
- În general se folosește o abordare bottom-up, de la subprobleme la probleme.

⇨ Diferente: **Greedy – Programare dinamica**

Greedy	**Programare dinamica**
Sunt mentinute doar solutiile partiale curente din care evolueaza solutiile partiale urmatoare	La constructia unei solutii noi poate contribui orice alta solutie partial generata anterior
Solutiile partiale anterioare sunt eliminate	Se pastreaza toate solutiile partiale
Se poate obtine o solutie neoptima	Se obtine solutia optima

⇨ Diferente: **D&I – Programare dinamica**

Divide et impera	**Programare dinamica**
Abordare top-down – problema este descompusă în subprobleme care sunt rezolvate independent	In general abordare bottom-up - se porneşte de la sub-soluţii elementare şi se combină sub-soluţiile mai simple în sub-soluţii mai complicate, pe baza criteriului de optim
Putem rezolva aceeaşi problemă de mai multe ori (dezavantaj potenţial foarte mare)	Se evită calculul repetat al aceleiaşi subprobleme prin memorarea rezultatelor intermediare (memoizare)

⇨ Exemplu 1: **Parantezarea matricilor**
- Se dă un şir de matrice: A1, A2, ..., An.
- Care este numărul minim de înmulţiri de scalari pentru a calcula produsul: A1 x A2 x ... x An?
- Să se determine una dintre parantezările care minimizează numărul de înmulţiri de scalari.
- Descompunerea in subprobleme:
  - Încercăm să definim subprobleme identice cu problema originală, dar de dimensiune mai mică.
  - Oricare $1 \le i \le j \le n$:
    - Notăm Ai, j = Ai x … x Aj. Ai,j are pi-1 linii şi pj coloane: Ai,j(pi-1, pj)
    - m[i, j] = numărul optim de înmulţiri pentru a rezolva subproblema Ai,j
    - s[i, j] = poziţia primei paranteze pentru subproblema Ai,j.
  - Problema iniţială: A1,n
  - Pentru a rezolva Ai,j
    - Trebuie găsit acel indice $i \le k < j$ care asigură parantezarea optimă: Ai, j = (Ai x…x Ak) x (Ak+1 x…x Aj)

Ai, j = Ai, k x Ak+1, j

- Alegerea optimala:
  - Căutăm optimul dintre toate variantele posibile de alegere (i ≤ k < j)
  - Pentru aceasta, trebuie însă ca şi subproblemele folosite să aibă soluţie optimală (adică Ai, k şi Ak+1, j să aibă soluție optimă).
- Substructura optimala:
  - Dacă ştim că alegerea optimală a soluţiei pentru problema Ai, j implică folosirea subproblemelor (Ai, k şi Ak+1, j) şi soluţia pentru Ai, j este optimală, atunci şi soluţiile subproblemelor Ai, k şi Ak+1, j trebuie să fie optimale!
- Definirea recursiva
  - $$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & i < j \end{cases}$$
  - Cazurile de bază sunt m[i, i]
  - Noi vrem să calculăm m[1, n]
  - Cum alegem s[i, j] ?: Bottom-up de la cele mai mici subprobleme la cea inițială.

$$m[1,2], m[2,3], m[3,4], \ldots, m[n-3,n-2], m[n-2,n-1], m[n-1,n]$$
$$m[1,3], m[2,4], m[3,5], \ldots, m[n-3,n-1], m[n-2,n]$$
$$m[1,4], m[2,5], m[3,6], \ldots, m[n-3,n]$$
$$\vdots$$
$$m[1,n-1], m[2,n]$$
$$m[1,n]$$

- Pseudocod:

```
Înmulțire_matrici (p, n)
  Pentru i de la 1 la n // inițializare
    m[i, i] = s[i, i] = 0
  Pentru l de la 1 la n - 1 // dimensiune problema
    Pentru i de la 1 la n – l  // indice stânga
      j = i + l // indice dreapta
      m[i, j] = ∞ // pentru determinare minim
      Pentru k de la i la j – 1
        q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]
        Dacă q < m[i, j]
              m[i, j] = q
              s[i, j] = k
  Întoarce m și s
```

- Complexitate:
  - Spatiala: O(n^2)
  - Temporala: O(n^3) (O(n^2) = numarul total de subprobleme; O(n) = numarul total de alegeri la fiecare pas)

- ⇨ Exemplu 2: **Arbori binari de cautare**
  - Probleme:
    - Costul căutării depinde de frecvenţa cu care este căutat fiecare termen. -> Ne dorim ca termenii cei mai des căutaţi să fie cât mai aproape de vârful arborelui pentru a micşora numărul de apeluri recursive.
    - Dacă arborele nu este construit prin sosirea aleatorie a cheilor putem ajunge la o simplă listă cu n elemente.
  - Un arbore de căutare probabilistică având cost minim este un arbore optim la căutare (AOC).
  - AOC(x, p, q, n)
    Pentru *i* **de la** 0 **la** *n*
    $\{C_{i,i} = 0, R_{i,i} = 0, w_{i,i} = q_i\}$ // inițializare costuri AOC vid $A_{i,i}$
    Pentru *d* **de la** 1 **la** *n*
    Pentru *i* **de la** 0 **la** *n-d* // calcul indice rădăcină și cost pentru $A_{i,i+d}$
    $j = i + d,\ C_{i,j} = \infty,\ w_{i,j} = w_{i,j-1} + p_j + q_j$
    Pentru *α* **de la** *i + 1* **la** *j* // ciclul critic – operații intensive
    Dacă $(C_{i,\alpha-1} + C_{\alpha,j} < C_{i,j})$ // cost mai mic?
    $\{ C_{i,j} = C_{i,\alpha-1} + C_{\alpha,j} ; R_{i,j} = \alpha\}$ // update
    $C_{i,j} = C_{i,j} + w_{i,j}$ // update
    Întoarce gen_AOC(C, R, x, 0, n) // construcție efectivă arbore $A_{0,n}$
    // cunoscând indicii
  - Complexitate: O(n^3) ; optimizare-knuth: O(n^2)

Curs 4 – Backtracking

- ⇨ Cronologic – orb:
  - Construieşte soluţiile iterativ.
  - Menţine evidenţa alegerilor făcute.
  - În momentul în care se ajunge la o contradicţie se revine la ultima decizie luată şi se încearcă alegerea unei alte variante.
  - **Schema Backtracking**
    - Soluție-parțială ← INIT // inițializez
    - EȘEC-DEFINITIV ← fals // nu am ajuns (încă) la eșec
    - **Cât timp** Soluție-parțială nu este soluție finală **și nu avem** EȘEC-DEFINITIV
      - Soluție-parțială ← AVANS (Soluție-parțială) // avansez
      - **Dacă** EȘEC (Soluție-parțială) // nu mai pot avansa
        - **Atunci** REVENIRE (Soluție-parțială) // mă întorc
    - **Dacă** EȘEC-DEFINITIV
      - **Atunci** Întoarce EȘEC // nu s-a găsit nicio soluție
      - **Altfel** Întoarce SUCCES // am ajuns la soluția problemei
    - **Sfârșit.**

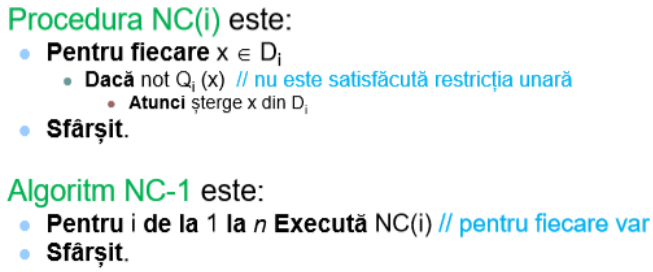
    Procedura AVANS (Soluție-parțială)
    - **Dacă** există alternativă de extindere // pot avansa?
      - **Atunci** Soluție-parțială ← Soluție-parțială ∪ alternativă de extindere // avansez
      - **Altfel Dacă** Soluție-parțială este INIT
        - **Atunci** EȘEC-DEFINITIV ← adevărat // nu s-au găsit soluții pentru problemă
        - **Altfel** EȘEC (Soluție-parțială) // ramura curentă a dus la eșec

- Optimizari posibile:
  - Alegerea variabilelor în altă ordine.
  - Îmbunătăţirea revenirilor. -> Necesită detectarea cauzei producerii erorii.
  - Evitarea redundanţelor în spaţiul de căutare (îmbunătăţirea avansului). -> Evitarea repetării unei căutări care ştim că va duce la un rezultat greşit.

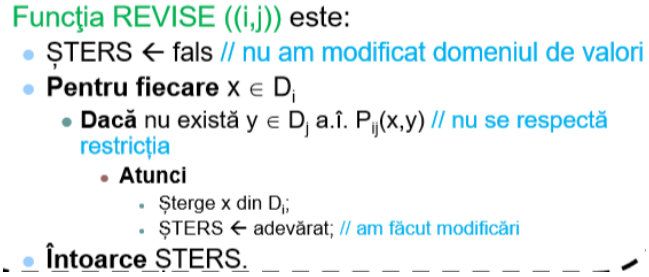
- O restricţie c este o relaţie între una sau mai multe variabile v1, …, vm, (denumite nodurile sau celulele restricţiei). Fiecare variabila vi poate lua valori într-o anumită mulţime Di, denumită domeniul ei (ce poate fi finit sau nu, numeric sau nu).
  - Unare: specificarea domeniului variabile
  - Binare: intre 2 variabile $V1j \neq V1k$ orice $j \neq k$, $0 < j, k < 10$
  - N-are: intre n variabile, Regula triunghiului: $L1 + L2 > L3$
  - O problemă de satisfacere a restricţiilor (PSR) este un triplet <V,D,C>, format din:
    - o mulţime V formată din n variabile V = {v1, …, vn};
    - mulţimea D a domeniilor de valori corespunzătoare acestor variabile: D = {D1, … Dn} ;
    - o mulţime C de restricţii C = {c1, …, cp} peste submulţimi ale V (ci (vi1, . . ., vij) ⊆ Di1 x Di2 x ... x Dij).
  - O restricţie ci (vi1, ..., vij), este o submulţime a produsului cartezian Di1 x Di2 x ... x Dij, constând din toate tuplurile de valori considerate că satisfac restricţia pentru (vi1, . . ., vij).
  - O soluţie a unei PSR <V,D,C> este un tuplu de valori <x1, … , xn> care conține toate variabilele V, din domeniile corespunzătoare D, astfel încât toate restricţiile din C să fie satisfăcute.
  - PSR binară este o PSR ce conţine doar restricţii unare şi binare.
  - Reprezentarea folosită: graf
    - Nodurile: variabilele restricţiei
    - Arcele: restricţiile problemei
    - Valabilă doar pentru PSR binare
  - Altă reprezentare posibilă: graf
    - Nodurile: restricţiile problemei
    - Arcele: variabilele restricţiei
    - Valabilă pentru orice tip de PSR

- Algoritmi de rezolvare a PSR: - Propagarea restrictiilor
  - Rezolvă PSR binare;
  - Variabilele au domenii finite de valori;
  - Prin propagarea restricţiilor se filtrează mulţimile de valori (se elimină elementele din domeniu conform unui criteriu dat);
  - Procesul de propagare se opreşte când:
    - O mulţime de valori este vidă -> EȘEC;
    - Nu se mai modifică domeniul vreunei mulţimi

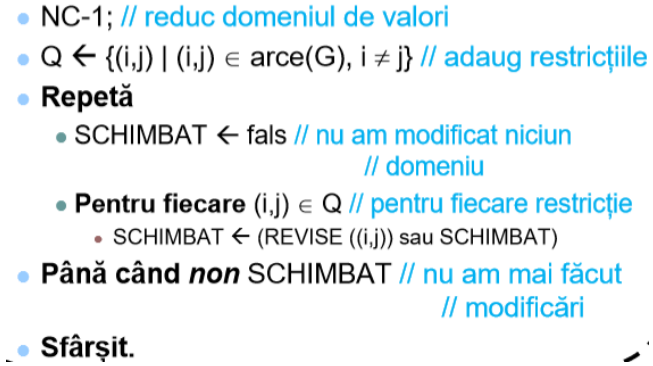
- Algoritm de consitenta nodurilor – restrictii unare



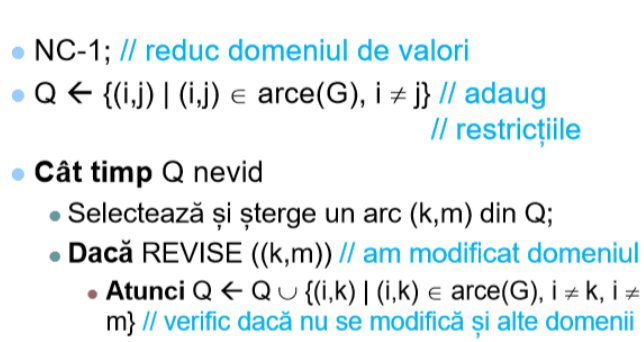
- Algoritm de consistenta a arcelor
  - Inlatura toate inconsistentele submultimilor de 2 elemente ale retelei de restrictii



  - Funcţia Revise este apelată pentru un arc al grafului de restricţii (binare) și șterge acele valori din domeniul de definiţie al unei variabile pentru care nu este satisfăcută restricţia pentru nici o valoare corespunzătoare celeilalte variabile a restricţiei.
  - Compexitate: O(a^2)
- AC-1 (Arc Consistency - 1)
  -



  - Se aplică algoritmul de consistenţa nodurilor şi apoi se aplică REVISE până nu se mai realizează nici o schimbare
  - Complexitate: O(na * 2r * a^2) (na = La fiecare iteraţie (din Repetă) eliminăm o singură valoare (şi avem maxim na valori posibile) ; 2r =Numărul de apelări al Revise într-o iteraţie (din Repetă); a^2 = Complexitate Revise).

- AC-3 (Arc Consistency - 3)
  - 

  - Se elimină pe rând arcele (constrângerile).
  - Dacă o constrângere aduce modificări în reţea adăugăm pentru reverificare nodurile care punctează către nodul de plecare al restricţiei verificate. -> Scopul: Reverificarea nodurilor direct implicate de o constrângere din reţea.
  - Avantaj: Se fac mult mai puţine apeluri ale funcţiei REVISE.
  - Complexitate: O(a^3*r)

⇨ Backtracking+ Propagarea restrictiilor
- În general, propagarea restricţiilor nu poate rezolva complet problema dată.
- Metoda ajută la limitarea spaţiului de căutare (foarte importantă în condiţiile în care backtracking-ul are complexitate exponenţială!).
- În cazul în care propagarea restricţiilor nu rezolvă problema se foloseşte:
  - Backtracking pentru a genera soluţii parţiale;
  - Propagarea restricţiilor după fiecare pas de backtracking pentru a limita spaţiul de căutare (şi eventual a găsi că soluţia nu este validă).

Curs 5- Minimax

⇨ Metoda Minimax:
- 2 jucători: Max și Min care mută pe rând (Max mută primul).
- Max urmărește să-și maximizeze câștigul.
- Min urmărește să-și minimizeze pierderea.
- Se construiește un arbore de tip AND-OR:
- Nivelurile pare -> mutările jucătorului Max.
- Nivelurile impare -> mutările jucătorului Min.
- Frunzele desemnează câștigul/pierderea lui Max.
- Arcele reprezintă mutările propriu-zise.

⇨ Functionare Minimax:
- 1) Se generează întregul arbore;
- 2) Se evaluează frunzele și li se asociază valori;
- 3) Se propagă rezultatele dinspre frunze spre rădăcină astfel:
  - Nivelul MIN alege cea mai mică valoare dintre cele ale copiilor.
  - Nivelul MAX alege cea mai mare valoare dintre cele ale copiilor.

⇨ Probleme:
- Dimensiunea arborelui pentru “X și 0” e ≤ 9!
- Pentru Șah fiecare nod are în medie 35 copii!
- Pentru Go ramificarea este de cca. 150 – 250!
- Complexitatea arborelui: pentru Șah – 10123 noduri; pentru Go – 10360 noduri.

- Limitări:
  - Nu putem să construim întregul arbore
  - Nu putem ajunge de fiecare dată la stările finale pentru a le putea evalua.

⇨ Optimizari:

- Limitarea adâncimii căutării
  - Trebuie să construim o funcție euristică care să estimeze șansele de câștig pentru o poziție dată.
    - Ex. pentru șah: Regină:10p; Turn: 5p; Cal, Nebun: 3p; Pion: 1p;
    - Ex: Funcție de evaluare a poziției = suma pieselor proprii – suma pieselor adversarului.
  - Oprirea căutării:
    - Limitare statică: după un număr maxim de nivele/interval de timp.
    - Limitare dinamică: când profitul obținut din continuarea căutării devine foarte mic (scade sub o valoare fixată).
  - Se estimează valoarea funcției de evaluare la nivelul respectiv.
  - Apoi propagăm valorile conform principiului enunțat anterior.

⇨ Functii de evaluare:

- Funcția euristică trebuie să cuantifice ”poziția”. -> Chiar în dauna avantajului material.
- Trebuie să ia în calcul potențialele amenințări!

⇨ Algoritm:

MINIMAX_limitat (n, nivel_limită)
- **Pentru fiecare** n' ∈ succs(n) // pentru toate mutările
  - Fie m = mutarea corespunzătoare arcului (n, n')
  - VAL(m) = W(n', nivel_limită, 1) // determin valoarea mutării
- **Întoarce** m a.î. VAL(m) = max {VAL(x)| x ∈ mutări(n)}

W(n, limită, nivel)
- **Dacă** n este frunză **Întoarce** cost(n)
- **Dacă** nivel ≥ limită **Întoarce** euristică(n)
- **Dacă** jucătorul MAX este la mutare **Întoarce**
  - max {w(n', limită, nivel + 1) | n' ∈ succs(n)}
- **Dacă** jucătorul MIN este la mutare **Întoarce**
  - min {w(n', limită, nivel + 1) | n' ∈ succs(n)}

⇨ Taieri α-β

- Încercăm să limităm spațiul de căutare prin eliminarea variantelor ce nu au cum să fie alese.
- Idee: Dacă V21<V1 toată ramura V2 poate fi Ignorată.
- α = max dintre valorile găsite pentru un nod MAX.
- β = min dintre valorile găsite pentru un nod MIN
- Tăiem o ramură dacă:
  - am găsit un nod pe nivelul MAX cu valoare β <= oricare din valorile α calculate anterior;
  - am găsit un nod pe nivelul MIN cu valoare α >= oricare din valorile β calculate anterior.
- Teorema α-β. Fie J un nod din arborele MINIMAX explorat. Daca α(J) ≥ β(J), atunci explorarea nodului J nu este necesară.

- Algoritm

α-β(n, limită)
- w = eval_max(n, -∞, ∞, 0, limită)
- **Întoarce** m ∈ mutări(n) a.i. VAL(m) = w

eval_max(n, α, β, nivel, limită)
- **Dacă** n este frunză **Întoarce** cost(n)
- **Dacă** (nivel ≥ limita) **Întoarce** euristică(n) // sunt limitat
- a = -∞ // valoarea curenta a nodului de tip Max
- **Pentru fiecare** (n' ∈ succs(n))
  - a = max(a, eval_min(n', max(α,a), β, nivel+1, limită)); // propag
  - **Dacă** (a ≥ β) **oprire**;
- **Întoarce** a

similar eval_min

  - Reduce complexitatea minimax în cazul ideal de la: Număr_ramificări^(număr_nivele) la Număr_ramificări^(număr_nivele/2)
  - Contează foarte mult ordinea în care analizăm mutările! -> Sortarea mutărilor după un criteriu dat nu este costisitoare comparativ cu costul exponențial al algoritmului.
  - Se folosesc euristici pentru a alege mutările examinate mai întâi.
- ⇨ Observatii :
  - Algoritmi de căutare în adâncime
  - Pot cauza probleme când avem un timp limită. -> Soluție posibilă IDDFS (căutare în adâncime mărind iterativ adâncimea maximă până la care căutăm).
  - Algoritmi cu complexitate foarte mare.
  - Soluții euristice pentru limitarea complexității.
  - Recomandabil să se combine cu alte strategii – baze de date cu poziții, pattern-matching.

Cursul 6 – Introducere in grafuri

- ⇨ Tipuri de grafuri:
  - Orientate: noduri (A-L) + arce (AB, BC, CD…).
  - Neorientate: noduri (A-L) + muchii (AB, BC, CD…).
- ⇨ Utilizari practice:
  - Hărți, rețele (calculatoare, instalații, etc.), rețele sociale, analiza fluxurilor (semaforizare, proiectarea dimensiunii țevilor de apă).
- ⇨ Culoarea nodului – specifică starea nodului la un anumit moment al parcurgerii:
  - Alb – nedescoperit;
  - Gri – descoperit, în curs de prelucrare;
  - Negru – descoperit și terminat (cu semnificații diferite pentru BFS si DFS).

⇨ BFS – Parcurgere in latime

```
BFS(s,G)
  Pentru fiecare nod u (u ∈ V)
    p(u) = null; dist(s,u) = inf; c(u) = alb; // inițializări
  Q = (); // se folosește o coadă în care reținem nodurile de prelucrat

  dist(s,s) = 0; // actualizări: distanța de la sursă până la sursă este 0
  Q ← Q + s; // adăugăm sursa în coadă → începem prelucrarea lui s
  c(s) = gri; // și atunci culoarea lui devine gri

  Cât timp (!empty(Q)) // cât timp mai am noduri de prelucrat
    u = top(Q); // se determină nodul din vârful cozii
    Pentru fiecare nod v (v ∈ succs(u)) // pentru toți vecinii
      Dacă c(v) este alb // nodul nu a mai fost găsit, nu e în coadă
        Atunci { dist(s,v) = dist(s,u) + 1; p(v) = u; c(v) = gri; Q ← Q + v;}
                  // actualizăm structura date
    c(u) = negru; // am terminat de prelucrat nodul curent
    Q ← Q - u; // nodul este eliminat din coadă
```

- Proprietati:
  - În cursul execuției BFS(s,G) v∈Q ⇔ v∈R(s). => ∈ Q => v ∈ R(s): Q conține exclusiv noduri din R(s) (BFS parcurge toate nodurile ce pot fi atinse din s); Toate nodurile ce pot fi atinse din s vor fi introduse cândva în coadă. (drum s = v0v1…vp = v) .
  - Oricare (u,v) ∈ E, δ(s,v) ≤ δ(s,u) + 1 : δ(s,v) ≤ δ(s,u) + 1 în general este egalitate când v este descoperit din u; < când v deja a fost descoperit înainte să se ajungă în u.
  - La terminarea BFS(s,G) există proprietatea dist(s,u) ≥ δ(s,u).
  - După orice execuție a ciclului principal al BFS, Q conține v1, v2, …, vp ai: Prop(Q) = dist(s,v1) ≤ dist(s,v2) ≤ ... ≤ dist(s,vp) ≤ dist(s,v1) + 1 => la un moment dat in coadă sunt elemente de pe același nivel din arborele generat de BFS (sau maxim 1 nivel diferență).
  - d(u) = momentul în care nodul u este inserat în coada Q. Atunci: d(u) < d(v) => dist(s,u) ≤ dist(s,v).
  - BFS este corect și după terminare δ(s,u) = dist(s,u), oricare u din V.

- Complexitate: O(n+m)
- Optimalitate: DA
- Parcurge tot graful? NU

⇨ DFS – parcurgerea in adancime

```
DFS(G)
  V = noduri(G)
  Pentru fiecare nod u (u ∈ V)
    c(u) = alb; p(u) = null; // inițializare structură date
  timp = 0; // reține distanța de la rădăcina arborelui DFS până la nodul curent
  Pentru fiecare nod u (u ∈ V)
    Dacă c(u) este alb
      Atunci explorare(u); // explorez nodul

explorare(u)
  d(u) = ++ timp; // timpul de descoperire al nodului u
  c(u) = gri; // nod în curs de explorare
  Pentru fiecare nod v (v ∈ succs(u)) // încerc sa prelucrez vecinii
    Dacă c(v) este alb
      Atunci {p(v) = u; explorare(v);} // dacă nu au fost prelucrați deja
  c(u) = negru; // am terminat de explorat nodul u
  f(u) = ++ timp; // timpul de finalizare al nodului u
```

- Nu mai avem nod de start, nodurile fiind parcurse în ordine.
- Proprietati:
  - G = (V,E); u ∈ V; pentru fiecare v descoperit de DFS pornind din u este construită o cale v, p(v), p(p(v)),…, u.
  - G = (V,E); DFS(G) sparge graful G într-o pădure de arbori Arb(G) = { Arb(u); p(u) = null } unde Arb(u) = (V(u),E(u));
    - V(u) = { v | d(u) < d(v) < f(u) } + {u};
    - E(u) = { (v, z) | v, z ∈ V(u) && p(z) = v }.
  - Dacă DFS(G) generează 1 singur arbore => G este conex. Graf orientat conex înseamnă că prin transformarea arcelor în muchii se obține un graf neorientat conex.
  - Teorema parantezelor: Oricare u, v avem I(u) ∩ I(v) = 0 sau I(u) inclus I(v) sau I(v) inclus I(u).
  - Oricare u, v apartine V, atunci v apartine V(u) ⇔ I(v) inclus I(u).
  - Teorema drumurilor albe: G = (V,E); Arb(u); v este descendent al lui u in Arb(u) ⇔ la momentul d(u) există o cale numai cu noduri albe u..v.
  - Într-un graf neorientat, DFS poate descoperi doar muchii directe și inverse.
  - G = graf orientat; G ciclic ⇔ în timpul execuției DFS găsim arce inverse.

- Tipuri de arce:
  - Arc direct(de arbore) : intre nod gri si nod alb
  - Arc invers (de ciclu) : intre nod gri si nod gri
  - Arc inainte: nod gri si nod negru si d(u) < d(v)
  - Arc transversal: nod gri si nod negru si d(u) > d(v)

- Complexitate: O(n+m)
- Optimalitate: NU
- Parcurge tot graful? DA

Cursul 7 – Sortare Topologica, Componente Tare Conexe

⇨ Sortare Topologica:
- Se folosește la sortarea unei mulțimi parțial ordonate (nu orice pereche de elemente pot fi comparate).
- Fie A o mulțime parțial ordonată față de o relație de ordine ∝ (∝ ⊆ A*A) atunci ∃ e1 și e2 astfel încât e1, e2 nu pot fi comparate.
- O sortare topologică a lui A este o listă L = <e1,e2…en>, cu proprietatea că ⍰⍰ i, j, dacă ei ∝ ej, atunci i < j.
- G = (V,E) orientat, aciclic.
- VS – secvența de noduri a.î. oricare (u,v) apartine E, avem index(u) < index(v).
- Scop: Sortare_topologică(G) => VS.
- Idee bazată pe DFS: G = (V,E) orientat, aciclic; la sfârșitul DFS avem oricare(u,v) apartine E, f(v) < f(u) => colectăm în VS vârfurile în ordinea descrescătoare a timpilor f

- Sortare_topologică (G)
  - **Pentru fiecare** nod u (u ∈ V) {c(u) = alb;} // inițializări
  - $V_S = \varnothing$;
  - **Pentru fiecare** nod u (u ∈ V) // pentru fiecare componentă conexă
    - **Dacă** c(u) este alb
      - V_S = Explorează (u, V_S) // prelucrez componenta conexă
  - **Întoarce** V_S

  Explorează (u, V_S)
  - c(u) = gri // prelucrez nodul, deci îi actualizez culoarea
  - **Pentru fiecare** nod v (v ∈ succs(u))
    - **Dacă** c(v) este alb **atunci** V_S = Explorează (v, V_S) // recursivitate
    - **Dacă** c(v) este gri **atunci Întoarce** Eroare: graf ciclic
  - c(u) = negru // am terminat prelucrarea nodului
  - **Întoarce** cons(u, V_S) // inserează nodul u la începutul lui V_S

- Complexitate: O(n+m)

⇨ Componente tari conexe

- Fie G = (V,E) un graf orientat. G este tareconex ⇔ oricare u,v apartine V exista o cale u..v și o cale v..u (u apartine R(v) și v apartine R(u)).
- G = (V,E) graf orientat. G' = (V',E'), V'⊆V, E'⊆E. G' este o CTC a lui G ⇔ G' e tare-conex (oricare u,v apartine V', u apartine R(v) și v apartine R(u)) și G' este maximal (ca număr de noduri).
- Lema : G = (V,E) graf orientat, G' CTC => oricare u,v apartine V' avem oricare u..v din G are noduri exclusiv in V'
- G = (V,E) orientat, G' = (V',E') o CTC a lui G. Toate nodurile v apartine V' sunt grupate in același Arb(u) construit de DFS(G), unde u este primul nod descoperit al componentei.
- G = (V,E) orientat, u apartine V. Φ (u) = strămoș DFS al lui u determinat în cursul DFS(G) dacă: Φ (u) apartine R(u) ; f(Φ (u)) = max{f(v) | v apartine R(u)} ; Φ(u) = primul nod din CTC descoperit de DFS(G).
- Teorema: Φ (u) satisface următoarele proprietăți:
  - 1. f(u)≤f(Φ (u)) când e egalitate? U este primul nod din CTC
  - 2. Oricare v apartine R(u), f(Φ (v)) ≤ f(Φ (u)) ce înseamnă ca e egalitate? U si v sunt in aceeasi CTC
  - 3. Φ (Φ (u)) = Φ (u)
- Φ(u) apartine R(u) și f(Φ(u)) = max{f(v) | v apartine R(u)}.
- G = (V,E) orientat, oricare u apartine V, u este descendent al lui Φ(u) în Arb(Φ (u)) construit de DFS
- G = (V,E) orientat, oricare u,v apartine V; u și v aparțin aceleiași CTC ⇔ Φ(u) = Φ (v).
- Probleme:
  - vrem ca fiecare arbore construit să conțină o CTC.
  - trebuie să eliminăm nodurile care nu sunt în componenta conexă.
  - Idee: eliminăm nodurile ce nu aparțin CTC! Dacă aparțin Arb(u) și nu CTC => exista u..v și ∄ v..u. -> DFS pe graful transpus, în ordinea descrescătoare a timpilor de finalizare obținuți din DFS pe graful normal!
- Observatii:
  - Înlocuind componentele tare conexe cu noduri obținem un graf aciclic. (Pentru că altfel am avea o singură CTC!)

- Prima parcurgere DFS este o sortare topologică. (Pentru că sortează nodurile în ordinea inversă a timpilor de finalizare a strămoșilor fiecărei CTC!)

- Algoritmul lui Kosaraju:

  CTC(G)
  - DFS(G)
  - G^T = transpune(G)
  - DFS(G^T) (în bucla principală se tratează nodurile în ordinea descrescătoare a timpilor de finalizare de la primul DFS)

  - Complexitate: O(n+m)
  - Algoritmul CTC calculează corect componentele tare conexe ale unui graf G = (V,E).

Cursul 8 – Puncte de articulatie, Punti, Drumuri minime

⇨ Puncte de articulatie:

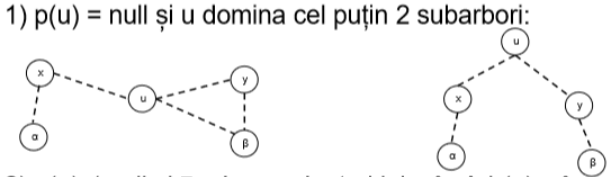
- Def: G = (V,E) graf neorientat, u apartine V. U este punct de articulație dacă oricare x,y apartine V, x ≠ y, x ≠ u, y ≠ u, a.î. oricare x..y în G trece prin u.
- Algoritm naiv:

  Elimină fiecare nod și verifică conectivitatea grafului rezultat:
  - Graf conex → nodul nu e punct de articulație.
  - Altfel → punct de articulație.

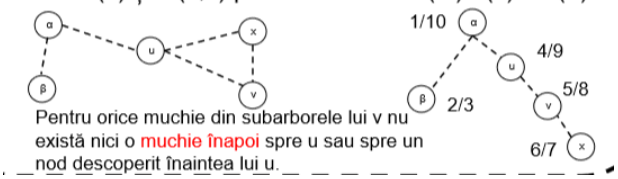
  Complexitate?
  O(V(V+E))

- Teorema: G = (V,E), graf neorientat și u apartine V. U este punct de articulație în G ⇔ în urma DFS în G una din proprietățile de mai jos este satisfăcută:
  - p(u) = null și u domină cel puțin 2 subarbori;
  - p(u) ≠ null și exista v descendent al lui u în Arb(u) a.î. oricare x apartine Arb(v) și oricare (x,z) parcursă de DFS(G) avem d(z) ≥ d(u).

  1) p(u) = null și u domina cel puțin 2 subarbori:

  2) p(u) ≠ null și ∃v descendent al lui u în Arb(u) a.î. ∀x∈Arb(v) și ∀(x,z) parcursă de DFS(G) d(z) ≥ d(u):

  Pentru orice muchie din subarborele lui v nu există nici o muchie înapoi spre u sau spre un nod descoperit înaintea lui u.

- Algoritm Tarjan:
  - Se aplică DFS și se salvează pentru fiecare nod până unde merge înapoi (low): low[u] = min {d(u), d(v) pentru toate muchiile înapoi (u,v), low(w) pentru toți fiii w ai lui u}.

- Pentru eficiență, trebuie ca fiii să se parcurgă înaintea părinților -> ordinea inversă a d(u)
-

Articulații (G)

```
V = noduri(G) // inițializări
Timp = 0;
Pentru fiecare (u ∈ V)
    c(u) = alb;
    d(u) = 0;
    p(u) = null;
    low(u) = 0;
    subarb(u) = 0; // reține numărul de subarbori dominați de u
    art(u) = 0; // reține punctele de articulație
Pentru fiecare (u∈V)
    Dacă c(u) e alb
        Exploreaza(u);
        Dacă (subarb(u) > 1) // cazul în care u este rădăcina în arborele
            art(u) = 1       // DFS și are mai mulți subarbori → cazul
                             // 1 al teoremei
```

Explorează(u)

```
d(u) = low(u) = timp++; // inițializări
c(u) = gri;
Pentru fiecare nod v ∈ succs(u)
    Dacă (c(v) e alb)
        p(v) = u; subarb(u)++; // actualizare nr subarbori
                               // dominați de u
        Explorează(v);
        low(u) = min{low(u), low(v)} // actualizare low
        Dacă (p(u) != null && low(v) ≥ d(u)) art(u) = 1;
                               // cazul 2 al teoremei
    Altfel low(u) = min{low(u), d(v)} // actualizare low
```

- Algoritm Tarjan pentru CTC

```
index = 0  // nivelul pe care este nodul în arborele DFS
S = empty // se folosește o stivă care se inițializează cu Ø
Pentru fiecare v din V
    Dacă (v.index e nedefinit) atunci // se pornește DFS din fiecare nod pe care
        Tarjan(v)                     // nu l-am vizitat încă

Tarjan(v)
    v.index = index // se setează nivelul nodului v
    v.lowlink = index // reține strămoșul nodului v
    index = index + 1  // incrementez nivelul
    S.push(v) // introduc v în stivă
    Pentru fiecare (v, v') din E // se prelucrează succesorii lui v
        Dacă (v'.index e nedefinit sau v' e în S) atunci // CTC deja identificate sunt ignorate
            Dacă (v'.index e nedefinit) atunci Tarjan(v') // dacă nu a fost vizitat v' intru în  recursivitate
            v.lowlink = min(v.lowlink, v'.lowlink) //actualizez strămoșul
    Dacă (v.lowlink == v.index) atunci // printez CTC începând de la coadă spre rădăcină
        print "CTC:"
        Repetă
            v' = S.pop // extrag nodul din stiva și îl printez
            print v'
        Până când (v' == v) // până când extrag rădăcina
```

⇨ Punti

- Def: G = (V,E), graf neorientat și (u,v) ∈ E. (u,v) este punte în G ⇔ exista x,y ∈ V, x ≠ y, a.î. oricare x..y conține muchia (u,v).

- Algoritm:

Punți(G)

```
V = noduri(G) // inițializări
Timp = 0;
Pentru fiecare nod u (u ∈ V)
    c(u) = alb;
    d(u) = 0;
    p(u) = null;
    low(u) = 0;
    punte(u) =0; // înlocuiește: subarb(u) = 0; art(u) = 0;
Pentru fiecare nod u (u ∈ V)
    Dacă c(u) e alb
        Explorează(u)
```

Explorează(u)

```
d(u) = low(u) = timp++; // inițializări
c(u) = gri;
Pentru fiecare nod v (v ∈ succs(u))
    Dacă c(v) e alb
        p(v) = u; // se elimină: subarb(u)++;
        Explorează(v);
        low(u) = min{low(u), low(v)} // actualizare low
        Dacă (low(v) > d(u)) punte(v) = 1;
        // în loc de: Dacă(p(u) != null && low(v) >= d(u))
    Altfel
        Dacă (p(u) ≠ v) low(u) = min{low(u), d(v)} // actualizare low
```

⇨ Drumuri de cost minim

- Variante de calcul:
  - Drumuri punct – multipunct: pentru un nod dat s ∈ V, să se găsească un drum de cost minim de la s la oricare u ∈ V; **Dijkstra, Bellman-Ford**
  - Drumuri multipunct – punct: pentru un nod dat e ∈ V, să se găsească un drum de cost minim de la oricare u ∈ V la e; **G^T si apoi 1**
  - Drumuri punct – punct: pentru două noduri date u și v ∈ V, să se găsească un drum u..v de cost minim; **Folosind 1**
  - Drumuri multipunct – multipunct: ∀u, v ∈ V, să se găsească un drum u..v de cost minim. **Floyd-Warshall**
- Subdrumurile unui drum minim sunt drumuri optimale: G = (V,E), w : E -> R funcție de cost asociată. Fie p = v1v2...vk un drum optim de la v1 la vk. Atunci pentru orice i și j cu 1≤ i ≤ j ≤ k, subdrumul lui p de la vi la vj este un drum minim.
- G = (V,E), w : E -> R funcție de cost asociată. Fie p = s..uv un drum optim de la s la v. Atunci costul optim al acestui drum poate fi scris ca δ(s,v) = δ(s,u) + w(u,v)
- G = (V,E), w : E -> R funcție de cost asociată. ∀ (u,v) ∈ E avem δ(s,v) ≤ δ(s,u) + w(u,v).
- Drumuri minime de sursa unica:
  - Pentru grafuri orientate
  - Bazati pe Greedy
  - Se pornește de la nodul de start și pe baza unui optim local, drumurile sunt extinse și optimizate până la soluția finală.
  - Relaxarea arcelor -> dacă d[v] > d[u] + w(u,v), atunci actualizează d[v].
  - **Dijkstra, Bellman-Ford**
  - G = (V,E), w : E -> R funcție de cost asociată. ∀ v ∈ V, d[v] obținut de algoritmul lui Dijkstra respectă d[v] ≥ δ(s,v). În plus, odată atinsă valoarea δ(s,v), ea nu se mai modifică.
  - G = (V,E), w : E -> R funcție de cost asociată. Fie p = s..uv un drum optim de la s la v. Dacă d[u] = δ(s,u) la un moment dat, atunci începând cu momentul imediat următor relaxării arcului (u,v) avem d[v] = δ(s,v)

⇨ **DIJKSTRA**

- Folosește o coadă de priorități în care se adaugă nodurile în funcție de distanța cunoscută în momentul respectiv de la s până la nod.
- Se folosește NUMAI pentru costuri pozitive (w(u,v) > 0, orice u,v apartine V).
-

Dijkstra_generic (G,s)

- V = nodurile lui G
- **Cât timp** (V != ∅)
  - u = nod din V cu d[u] min
  - V = V - {u}
  - **Pentru fiecare** (v ∈ succesorii lui u) relaxare_arc(u,v)

// optimizare drum s..v pentru v ∈ succesorilor lui u

- Dijkstra(G,s)
  - **Pentru fiecare** (u ∈ V) // inițializări
    - d[u] = ∞; p[u] = null;
  - d[s] = 0;
  - Q = construiește_coada(V) // coadă cu priorități
  - **Cât timp** (Q != ∅)
    - u = ExtrageMin(Q); // extrage din V elementul cu d[u] minim
    - // Q = Q - {u} – se execută în cadrul lui ExtrageMin
    - **Pentru fiecare** (v ∈ Q și v din succesorii lui u)
      - **Dacă** (d[v] > d[u] + w(u,v))
        - d[v] = d[u] + w(u,v) // actualizez distanța
        - p[v] = u // și părintele

- Complexitate:
  - Operații ce trebuie realizate pe coadă + frecvența lor:
    - insert – V;
    - delete – V;
    - conține? – E;
    - micșorează_val – E;
    - este_vidă? – V.
- Complexitate:
  - Vector:
    - insert – 1 * V = V;
    - delete – V * V = V2 (necesită căutarea minimului);
    - conține? – 1 * E = E;
    - micșorează_val – 1 * E = E;
    - este_vidă? – 1 * V = V;
  - Heap binar:
    - structură de date de tip arbore binar + 2 constrângeri:
    - Fiecare nivel este complet; ultimul se umple de la stânga la dreapta; oricare u ∈ Heap; u ≥ răd(st(u)) și u ≥ răd(dr(u)) (u este ≥ decât ambii copii ai săi) unde ≥ este o relație de ordine pe mulțimea pe care sunt definite elementele heapului.
    - insert – logV * V = VlogV;
    - delete – logV * V = VlogV;
    - conține? – 1 * E = E;
    - micșorează_val – logV * E = ElogV;
    - este_vidă? – 1 * V = V.
    - Eficient dacă graful are arce puține comparativ cu numărul de noduri.
  - Heap Fibonacci
    - Poate fi format din mai mulți arbori.
    - Cheia unui părinte ≤ cheia oricărui copil
    - Fiind dat un nod u și un heap H:
      - p(u) – părintele lui u;
      - copil(u) – legătura către unul din copiii lui u;
      - st(u), dr(u) – legătura la frații din stânga și din dreapta (cei de pe primul nivel sunt legați între ei astfel);

        - grad(u) – numărul de copii ai lui u;
        - min(H) – cel mai mic nod din H;
        - n(H) – numărul de noduri din H.
    - insert – 1 * V = V;
    - delete – logV * V = VlogV(amortizat!);
    - micșorează_val – 1 * E = E;
    - este_vidă? – 1 * V = V.
    - Cea mai rapidă structură dpdv teoretic.
- Implementarea trebuie realizată în funcție de tipul grafului pe care lucrăm:
    - vectori pentru grafuri “dese” - O(V2);
    - heap pentru grafuri “rare”: HB - O(E logV), HF - O(V log V+E)
- Heapul Fibonacci este mai eficient decât heapul binar dar mai dificil de implementat.

Curs 9 – Drumuri de cost minim

⇨ **DIJKSTRA**

- G = (V,E), w : E -> R funcție de cost asociată nenegativă. La terminarea aplicării algoritmului Dijkstra pe acest graf plecând din sursa s vom avea d[v] = δ(s,v) pentru oricare v apartine V.
- Algoritmul nu funcționează pentru grafuri ce conțin arce de cost negativ!

⇨ **BELLMAN-FORD**

- Cicluri de cost negativ
-

```
BellmanFord(G,s)   // G=(V,E),s=sursa
  Pentru fiecare nod v (v ∈ V) // inițializări
    d[v] = ∞;
    p[v] = null;
  d[s] = 0; // actualizare distanță de la s la s
  Pentru i de la 1 la |V| -1 // pentru fiecare pas pornind din s
  // spre restul nodurilor se încearcă construcția unor drumuri
  // optime de dimensiune i
    Pentru fiecare (u,v) din E
    // pentru arcele ce pleacă de la nodurile deja considerate
      Dacă d[v] > d[u] + w(u,v) atunci // se relaxează arcele corespunzătoare
        d[v] = d[u] + w(u,v);
        p[v] = u;
  Pentru fiecare (u,v) din E
    Dacă d[v] > d[u] + w(u,v) atunci
      Eroare ("ciclu negativ");
```

```
BellmanFordOpt2(G,s)
  Pentru fiecare nod v (v ∈ V)
    d[v] = ∞;
    p[v] = null;
    marcat[v] = false; // marcăm nodurile pentru care am făcut relaxare
  Q = ∅; // coadă cu priorități
  d[s] = 0; marcat[s] = true; Introdu(Q,s);
  Cât timp (Q != ∅)
    u = ExtrageMin(Q);   marcat[u] = false; // extrag minimul
    Pentru fiecare nod v (v ∈ succesorilor lui u)
      Dacă d[v] > d[u] + w(u,v) atunci // relaxez arcele ce pleacă din u
        d[v] = d[u] + w(u,v);
        p[v] = u;
        Dacă (marcat[v] == false) {marcat[v] = true; Introdu(Q,v);}
```

- Complexitate:
    - Caz defavorabil:

Pentru i de la 1 la |V| - 1 -> V *
Pentru fiecare (u,v) din E -> E
Dacă d[v] > d[u] + w(u,v) atunci
d[v] = d[u] + w(u,v);
p[v] = u; => **O(V*E)**

- G = (V,E), w : E -> R funcție de cost asociată; dacă G nu conține ciclu de cost negativ atunci după |V| - 1 iterații ale relaxării fiecărui arc avem d[v] = δ(s,v) pentru oricare v apartine R(s).
- G = (V,E), w : E -> R funcție de cost asociată. Algoritmul Bellman-Ford aplicat acestui graf plecând din sursa s nu returnează EROARE dacă G nu conține cicluri negative, iar la terminare d[v] = δ(s,v) pentru oricare v apartine V. Dacă G conține cel puțin un ciclu negativ accesibil din s, atunci algoritmul întoarce EROARE.
- Observatii:
  - Dacă d[v] nu se modifică la pasul i atunci nu trebuie sa relaxăm niciunul din arcele care pleacă din v la pasul i + 1. => păstrăm o coadă cu vârfurile modificate (o singură copie).

⇨ **FLOYD-WARSHALL**

- Algoritm prin care se calculează distanțele minime între oricare 2 noduri dintr-un graf (drumuri optime multipunct-multipunct).
- Exemplu clasic de programare dinamică.
- Idee: la pasul k se calculează cel mai bun cost între u și v folosind cel mai bun cost u..k și cel mai bun cost k..v calculat până în momentul respectiv.
- Se aplică pentru grafuri ce nu conțin cicluri de cost negativ.
- Fie formulele de mai jos pentru calculul valorii dk(i,j), 0 ≤ k ≤ n:

  d0(i,j) = w(i,j);

  dk(i,j) = min{dk-1(i,j), dk-1(i,k) + dk-1(k,j)}, pentru 0 < k ≤ n;

  Atunci dn(i,j) = δ(i,j), pentru ∀ i, j ∈ V

- Algoritm:

```
Floyd-Warshall2(G)
• Pentru i de la 1 la n
  • Pentru j de la 1 la n // inițializări
    • d(i,j) = w(i,j)
    • Dacă (w(i,j) == ∞)
      • p(i,j) = null;
    • Altfel p(i,j) = i;
• Pentru k de la 1 la n
  • Pentru i de la 1 la n
    • Pentru j de la 1 la n
      • Dacă (d(i,j) > d(i,k) + d(k,j)) // determinăm minimul
        . d(i,j) = d(i,k) + d(k,j)
        . p(i,j) = p(k,j); // și actualizăm părintele
```

Complexitate? $O(V^3)$

Complexitate spațială? $O(V^2)$

⇨ Inchiderea tranzitiva

- Fie G = (V,E). Închiderea tranzitivă a lui E e un G* = (V,E*), unde

  E* (I,j) = 1, dacă ∃ i..j 0, dacă ∄ i..j
- Poate fi determinată prin modificarea algoritmului Floyd-Warshall:
  - min ⟹ operatorul boolean sau (ᵛ)
  - + ⟹ operatorul boolean și (^)

Închidere_tranzitivă(G)

- **Pentru** i **de la** 1 **la** n
  - **Pentru** j **de la** 1 **la** n
    - $E^*(i,j) = (((i,j) \in E) \vee (i = j))$ // inițializări
- **Pentru** k **de la** 1 **la** n
  - **Pentru** i **de la** 1 **la** n
    - **Pentru** j **de la** 1 **la** n
      - $E^*(i,j) = E^*(i,j) \vee (E^*(i,k) \wedge E^*(k,j))$

Complexitate? $O(V^3)$ Complexitate spațială? $O(V^2)$

⇨ **JOHNSON**

- Pentru grafuri rare.
- Folosește liste de adiacență.
- Bazat pe Dijkstra și Bellman-Ford.
- Complexitate: O(V^2*logV + VE) => Mai bună decât Floyd-Warshall pentru grafuri rare.
- Idee algoritm:
  - Dacă graful are numai arce pozitive: se aplică Dijkstra pentru fiecare nod ⟹ cost V^2*logV.
  - Altfel se calculează costuri pozitive pentru fiecare arc menținând proprietățile:
    - w1(u,v) ≥ 0, oricare (u,v) apartine E;
    - p este drum minim utilizând w ⟺ p este drum minim utilizând w1.
- Johnson(G)
  - // Construim G' = (V',E');
  - V' = V ∪ {s}; // adăugăm nodul s
  - E' = E ∪ (s,u), ∀u∈V; w(s,u) = 0; // și îl legăm de toate nodurile
  - **Dacă** BF(G',s) e fals // aplic BF pe G'
    - **Eroare** "ciclu negativ"
  - **Altfel**
    - **Pentru fiecare** v ∈ V
      - h(v) = δ(s,v); // calculat prin BF
    - **Pentru fiecare** (u,v) ∈ E
      - $w_1(u,v) = w(u,v) + h(u) - h(v)$ // calculez noile costuri pozitive
    - **Pentru fiecare** (u ∈ V)
      - Dijkstra(G,w_1,u) // aplic Dijkstra pentru fiecare nod
      - **Pentru fiecare** (v ∈ V)
        - $d(u,v) = \delta_1(u,v) + h(v) - h(u)$ // calculez costurile pe graful inițial

⇨ Concluzii **FLOYD-WARSHALL & JOHNSON**

- Algoritmi ce găsesc drumurile minime între oricare 2 noduri din graf.
- Funcționează pe grafuri cu arce ce au costuri negative (dar care nu au cicluri de cost negativ).
- Floyd-Warshall e optim pentru grafuri dese.
- Johnson e mai bun pentru grafuri rare.

Cursul 10 – Arbori minimi de acoperire

⇨ Definitii:

- Un arbore liber al lui G este un graf neorientat conex și aciclic Arb = (V',E'); V' ⊆ V, E' ⊆ E. Costul arborelui este: C(Arb) = Σ w(e), e apartine E'.
- Un arbore liber se numește arbore de acoperire dacă V' = V.
- Un arbore de acoperire (Arb) se numește arbore minim de acoperire (notăm AMA) dacă Arb apartine ARB(G) a.î. C(Arb) = min{C(Arb') | Arb' ∄ ARB(G)}.
- Fie A ⊆ E o mulțime de muchii ale unui graf G = (V,E) și (S,V-S) o partiționare a lui V. Partiționarea respectă mulțimea A dacă ∄ e apartine A care taie frontiera dintre S și V-S (∀ (u,v) apartine A → u,v apartine S sau u,v apartine V-S).
- Fie A ⊆ E' o mulțime de muchii ale unui AMA parțial Arb = (V,E') al grafului G = (V,E), iar e apartine E o muchie oarecare din G. Muchia e este sigură în raport cu A dacă mulțimea A U {e} face parte dintr-un AMA al lui G.
- Fie A o mulțime de muchii ale unui AMA parțial al grafului G = (V,E). Fie (S,V-S) o partiționare care respectă A, iar (u,v) apartine E o muchie care taie frontiera dintre S și V-S a.î. w(u,v) = min{w(x,y) | (x,y) apartine E și (x apartine S, y apartine V-S) sau (x apartine V-S, y apartine S)} Muchia (u,v) este sigură in raport cu A.

⇨ Utilizari:
- Proiectarea retelelor: electrice, calculatoare, drumuri
- Clustering
- Algoritmi de aproximare pentru probleme NP-complete.

⇨ Proprietati:
- G = (V,E), C = (V',E') – ciclu în G; e apartine E' a.î. w(e) = max {w(e') | e' apartine E'} => e !apartine Arb(G) unde Arb(G) = AMA în G.
- G = (V,E), S = (V',E') un AMA parțial al lui G, V' ⊆ V; e = (u,v) a.î. e !apartine E' și (u apartine V' și v !apartine V') sau (u !apartine V' și v apartine V') cu proprietatea că: w(u,v) = min{w(u',v')| (u' apartine V' și v' !apartine V') sau (u' !apartine V' și v apartine V')} => (u,v) apartine AMA.

⇨ Algoritmi de tip Greedy:
- Prim: se pornește cu un nod și se extinde pe rând cu muchiile cele mai ieftine care au un singur capăt în mulțimea de muchii deja formată (Proprietatea 2). Algoritmul este asemănător algoritmului Dijkstra.
- Kruskal: inițial toate nodurile formează câte o mulțime și la fiecare pas se reunesc 2 mulțimi printr-o muchie. Muchiile sunt considerate în ordinea costurilor și sunt adăugate în arbore doar dacă nu creează ciclu (Proprietatea 1).
- **Prim:**
  - Prim(G,w,s) **Complexitate?**
    - A = ∅ // AMA
    - **Pentru fiecare** (u ∈ V)
      - d[u] = ∞; p[u] = null // inițializăm distanța și părintele
    - d[s] = 0; // nodul de start are distanța 0
    - Q = constrQ(V, d); // ordonată după costul muchiei // care unește nodul de AMA deja creat
    - **Cât timp** (Q != ∅) // cât timp mai sunt noduri neadăugate
      - u = ExtrageMin(Q); // extrag nodul aflat cel mai aproape
      - A = A ∪ {(u,p[u])}; // adaug muchia în AMA
      - **Pentru fiecare** (v ∈ succs(u))
        - **Dacă** d[v] > w(u,v) **atunci**
          - d[v] = w(u,v); //+ d[u] // actualizăm distanțele și părinții nodurilor
          - p[v] = u; // adiacente care nu sunt în AMA încă
    - **Întoarce** A - {(s,p(s))} // prima muchie adăugată
  - Complexitate:

- Matrice de adiacență O(V^2)
- Heap binar O(E^logV)
- Heap Fibonacci O(V*logV + E)

- Observatii:
  - Grafuri dese -> Matrice de adiacență preferată
  - Grafuri rare -> Heap binar sau Fibonacci

- **Kruskal:**
  -

Kruskal(G,w)
- A = ∅; // AMA
- **Pentru fiecare** (v ∈ V)
  - Constr_Arb(v) // creează o mulțime formată din nodul respectiv // (un arbore cu un singur nod)
- Sortează_asc(E,w) // se sortează muchiile în funcție de // costul lor
- **Pentru fiecare** ((u,v) ∈ E) // muchiile se extrag în ordinea // costului
  - **Dacă** Arb(u) != Arb(v) **atunci** // verificăm dacă se creează ciclu
    - Arb(u) = Arb(u) ∪ Arb(v) // se reunesc mulțimile de noduri (arborii)
    - A = A ∪ {(u,v)} // se adaugă muchia sigură în AMA
- **Întoarce** A

Complexitate?

  - Complexitate:
    - Sortarea muchiilor: O(ElogE) ≈ O(ElogV)
    - Arb(u) = Arb(v) – compararea a 2 mulțimi disjuncte {1,2,3} {4,5,6} – mai precis trebuie identificat dacă 2 elemente sunt în aceeași mulțime
    - Arb(u) U Arb(v) – reuniunea a 2 mulțimi disjuncte într-una singură

  - Implementare rapida multimi disjuncte
    -

Comparare (u, v)
- Verifică dacă 2 noduri au aceeași rădăcină;
- Implică identificarea rădăcinii:

Arb(u) // identificarea rădăcinii unei componente
- **Cât timp** (i != id[i]) i = id[i];
- **Întoarce** i

Comparare (u, v)
- **Întoarce** Arb(u) != Arb(v)

Complexitate?

Reuniune (u,v) // implică identificarea rădăcinii
- v = Arb(v)
- id[v] = u;

    - Complexitate:
      - Compararea – O(V)
      - Reuniunea – O(V)

- Optimizare-reuniune rapida balansata
    - Se menține numărul de noduri din fiecare subarbore
    - Se adaugă arborele mic la cel mare pentru a face mai puține căutări → înălțimea arborelui e mai mică și numărul de căutări scade de la V la lg V.
    - Complexitate:
        - Compararea – O(lg V)
        - Reuniune – O(lg V)
- Complexitate dupa optimizare:
    - Orice secvență de E operații de căutare și reuniune asupra unui graf cu V noduri consumă O(V + E*α(V,E)).
    - α – de câte ori trebuie aplicat lg pentru a ajunge la 1. (in practica ≤ 5)
    - In practica: O(E)

▪ Complexitate dupa optimizari:
- Max (complexitate sortare, complexitate operații mulțimi) = max(O(ElogV), O(E)) = O(ElogV)
- Complexitatea algoritmului Kruskal este dată de complexitatea sortării costurilor muchiilor.

o Aplicatie practica- K-clustering
▪ Împărțirea unui set de obiecte în k grupuri astfel încât obiectele din cadrul unui grup să fie “apropiate” considerând o “distanță” dată.
▪ Utilizat în clasificare, căutare (web search de exemplu).
▪ Dându-se un întreg K și un grup de obiecte, se cere să se împartă grupul de obiecte în K grupuri astfel încât spațiul dintre grupuri să fie maximizat.
▪ Algoritm:
- Se formează V clustere (un cluster per obiect).
- Găsește cele mai apropiate 2 obiecte din clustere diferite și unește cele 2 clustere.
- Se oprește când au mai rămas k clustere.
- => Kruskal

Cursul 11 – Retele de flux. Flux maxim

⇨ Flux
o Proprietati:
▪ Oricare u, v apartine V
▪ f(u,v) ≤ c(u,v) (fluxul printr-un arc este mai mic sau egal cu capacitatea arcului) – respectarea capacității arcelor;
▪ Oricare u, v apartine V, f(u,v) = -f(v,u) – simetria fluxului;
▪ Σf(u,v) = 0 pentru oricare u apartine V \ {s,t} – conservarea fluxului.
o Notatii:
▪ fi(u) = Σf(v,u) – fluxul total care intra în nodul u;
▪ fo(u) = Σf(u,v) – fluxul total care iese din nodul u;
▪ Valoarea totală a fluxului:
- |f| = Σf(s,v) = fo(s);

- |f| = fluxul ce părăsește sursa;
- |f| = Σf(s,v) = Σf(v,t) = fi(t).

- Se adaugă o sursă unică cu arce de capacitate infinită spre sursele s1..sn și flux egal cu fluxul generat de sursele respective;
- Se adaugă o destinație unică t și arce de capacitate infinită între t1..tm și t și flux egal cu fluxul ce intră în destinațiile respective.

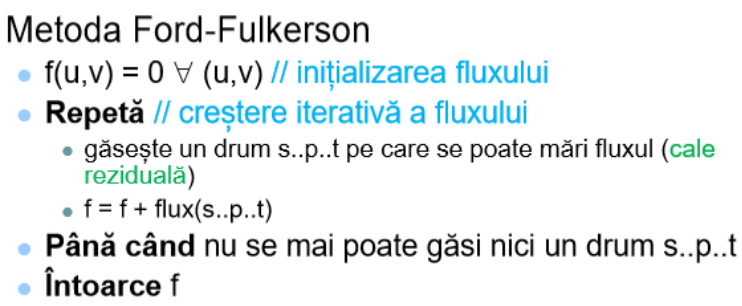
⇨ Arc rezidual
- Un arc (u,v) pentru care f(u,v) < c(u,v) se numește arc rezidual. => Fluxul pe acest arc se poate mări.

⇨ Capacitate reziduala
- Cantitatea cu care se poate mări fluxul pe arcul (u,v) se numește capacitatea reziduală a arcului (u,v) (cf(u,v)): cf(u,v) = c(u,v) - f(u,v)

⇨ Retea reziduala:
- Rețeaua reziduală (Gf = (V,Ef)) este o rețea de flux formată din arcele ce admit creșterea fluxului: Ef = {(u,v) ∈ V x V | cf(u,v) > 0}.
- Ef ⊄ E
- Fie G = (V,E) rețea de flux, f fluxul în G, Gf rețeaua reziduală a lui G. Fie f' un flux prin Gf și f+f' o funcție definită astfel: f+f' (u,v) = f(u,v) + f'(u,v). Atunci f+f' reprezintă un flux în G și |f+f'| = |f| + |f'| => cum putem mări fluxul printr-o rețea de flux.
- G – rețea de flux, f flux în G, p = s..t – cale reziduală în Gf, fp:V x V-> R se definește ca fiind:
  - fp(u,v) = cf(p), dacă (u,v) apartine p; -cf(p), dacă (v,u) apartine p 0; dacă (u,v) și (v,u) !apartine p

  fp = flux în Gf; |fp| = cf(p)
- f' = f + fp = flux în G, astfel încât |f'| = |f| + |fp| > |f|. => cum se defineste fluxul printr-o retea reziduala

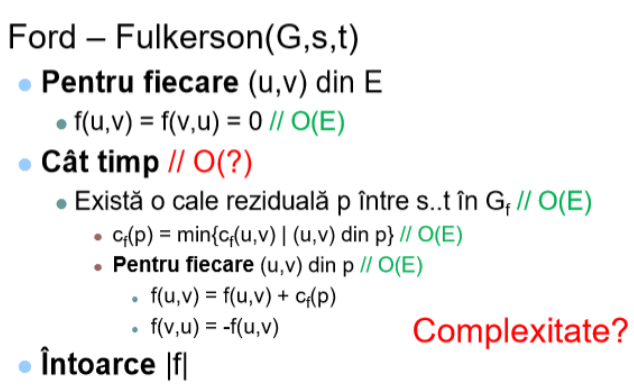
⇨ Cale reziduala:
- Cale reziduală e un drum s..t ⊆ Gf, unde cf(u,v) este capacitatea reziduală a arcului (u,v).
- Capacitatea reziduală a căii = capacitatea reziduală minimă de pe calea s..t descoperită.

⇨ Calcului fluxului maxim: **FORD-FULKERSON**
- 

⇨ Taieturi in retele de flux:
- O tăietură (S,T) a unei rețele de flux G = partiționare a nodurilor în 2 mulțimi disjuncte S și T = V \ S astfel încât s apartine S și t apartine T.
  - f(S,T) = ΣxΣy f(x,y) (x apartine S, y apartine T) – fluxul prin tăietura
  - c(S,T) = ΣxΣy c(x,y) (x apartine S, y apartine T) – capacitatea tăieturii
- Fluxul prin tăietură = fluxul prin rețea – f(S,T) = |f|

- S, T – tăietură oarecare – fluxul maxim este limitat superior de capacitatea tăieturii $|f| \leq c(S,T)$

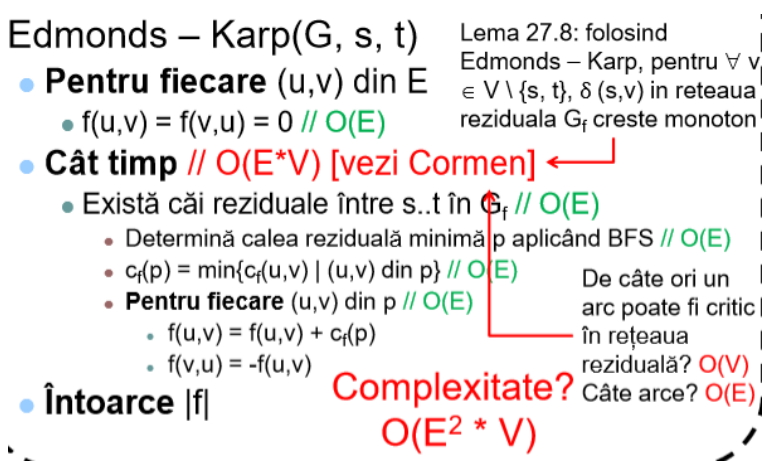
⇨ Flux maxim – taietura minima

- G = (V,E) rețea de flux – următoarele afirmații sunt echivalente:
  - f este o funcție de flux în G astfel încât |f| este flux maxim total în G;
  - rețeaua reziduală Gf nu are căi reziduale;
  - există o tăietură (S,T) astfel încât |f| = c(S,T).
- 

- Complexitate: O(E*fmax) ; fmax = fluxul maxim
- Probleme:
  - Se folosesc căi cu capacitate mică;
  - Se pun fluxuri pe mai multe arce decât este nevoie.

- Imbunatatiri:
  - Se aleg căile reziduale cu capacitate maximă – complexitatea va depinde în continuare de fmax și de valoarea capacităților;
  - Se aleg căile reziduale cele mai scurte → în acest caz complexitatea nu mai depinde de fmax ci numai de numărul de arce (ex. Edmonds-Karp: identificarea căilor reziduale minime prin aplicarea unui BFS)

⇨ **EDMONDS-KARP**

- 

⇨ Pompare preflux

- Simularea curgerii lichidelor într-un sistem de conducte ce leagă noduri aflate la diverse înălțimi;
- Sursa – înălțime |V|
- Inițial toate nodurile exceptând sursa sunt la înălțime 0

- Destinația rămâne în permanență la înălțimea 0
- Există un preflux inițial în rețea obținut prin încărcarea la capacitate maximă a tuturor conductelor ce pleacă din s;
- Excesul de flux dintr-un nod poate fi stocat întrun rezervor al nodului (Notat e(u));
- Când un nod u are flux disponibil în rezervor și o conductă spre un alt nod v nu este încărcată complet → înălțimea lui u este crescută pentru a permite curgerea din u în v.
- Definitii
  - Preflux = f: V x V → R astfel încât să fie satisfăcute restricțiile:
    - (u,v) ≤ c(u,v) (fluxul printr-un arc este mai mic sau egal cu capacitatea arcului) – respectarea capacității arcelor;
    - Oricare u, v apartine V, f(u,v) = -f(v,u) – simetria fluxului;
  - Supraîncărcare a unui nod: $e(u) = f(V,u) \geq 0$, oricare u apartine $V \setminus \{s\}$.
  - O funcție h: V → N este o funcție de înălțime dacă îndeplinește restricțiile:
    - h(s) = |V| – fixă;
    - h(t) = 0 – fixă;
    - h(u) ≤ h(v) + 1 pentru orice arc rezidual (u,v) apartine Gf – variabilă.
  - G – rețea de flux, h: V → N este o funcție de înălțime. Dacă oricare u, v apartine V, h(u) > h(v) + 1 atunci arcul (u,v) nu este arc rezidual.

- Algoritm:

Init_preflux(G, s, t)
- **Pentru fiecare** (u ∈ V)
  - e(u) = 0 // inițializare exces flux în nodul u
  - h(u) = 0 // inițializare înălțime nod u
- **Pentru fiecare** ((u,v) ∈ E) // inițializare fluxuri
  - f(u,v) = 0
  - f(v,u) = 0
- h(s) = |V| // inițializare înălțime sursă
- **Pentru fiecare** (u ∈ succs(s) \ {s})
  - f(s,u) = c(s,u); // actualizare flux
  - f(u,s) = -c(s,u);
  - e(u) = c(s,u) // actualizare exces

Complexitate?

Pompare_preflux(G, s, t)
- Init_preflux(G, s, t) // inițializarea prefluxului
- **Cât timp** (1) // cât timp pot face pompări sau înălțări
  - **Dacă** (∃u ∈ V \ {s, t}, v ∈ V | e(u) > 0 **și** c_f(u,v) > 0 **și** h(u) = h(v) + 1) // încerc să pompez
    - Pompare(u,v); **continuă;**
  - **Dacă** (∃u ∈ V \ {s, t}, v ∈ V | e(u) > 0 **și** ∀(u,v) ∈ E_f, h(u) ≤ h(v))
    - Înălțare(u); **continuă;** // încerc să înalț
  - **Întrerupe;** // nu mai pot face nimic → am ajuns la flux max
- **Întoarce** e(t) // e(t) = |f| = fluxul total în rețea

- Complexitate:
  - Init_preflux: O(E)
  - Pompare(u,v): O(1)
  - Înălțare(u): O(V) – implică găsirea minimului dintre nodurile succesoare
  - Cât timp: [vezi Cormen]
    - Câte înălțări?
      - Care e înălțimea maximă? 2 |V| - 1 – drum rezidual de lungime maximă
      - Care este numărul maxim total de înălțări? (2 |V| - 1) (|V| - 2)
    - Câte pompări?
      - Pompări saturate: 2 |V| |E| - de câte ori un arc poate fi saturat? (în funcție de suma h(u) + h(v))

- Pompări nesaturate: 4 |V|2 (|V| + |E|) – sumă înălțimi noduri excedentare
- Complexitate totală: O(V2 * E)

Cursul 12 - Algoritmi euristici de explorare

⇨ Explorarea spațiului stărilor problemei
- Stare a problemei = abstractizare a unei configurații valide a universului problemei, configurație ce determină univoc comportarea locală a fenomenului descris de problemă.
- Spațiul stărilor = graf în care nodurile corespund stărilor problemei, iar arcele desemnează tranzițiile valide între stări.
  - Caracteristică importantă: nu este cunoscut apriori, ci este descoperit pe măsura explorării!
  - Descriere:
    - Nodul de start (starea inițială);
    - Funcție de expandare a nodurilor (produce lista nodurilor asociate stărilor valide în care se poate ajunge din starea curentă);
    - Predicat de testare dacă un nod corespunde unei stări soluție.

- Obiectivele navigarii:
  - Cartografierea sistematică a spațiului stărilor.
  - Asamblarea soluțiilor parțiale care în final conduc la soluția finală. Această soluție finală poate fi:
    - Identificarea stărilor soluție (poziționarea a n regine pe tabla de șah fără să se atace);
    - Drumul străbătut de la starea inițială spre o stare soluție (acoperirea tablei de șah cu un cal);
    - Strategia de rezolvare = arbore multicăi în care rădăcina este starea inițială, iar frunzele sunt stări soluție. În acest arbore, unele noduri corespund unor evenimente neprevăzute care influențează calea de urmat în rezolvare (identificarea monedei false dintr-un grup de 3 monede).

- Algoritmi tentativi/irevocabili:
  - Dacă explorarea se bazează pe informația acumulată în cursul explorării, informație prelucrată euristic (costuri)→ algoritm informat.
  - Dacă explorarea este 'la întâmplare' → algoritm neinformat.
  - Dacă algoritmul de explorare are posibilitatea să abandoneze calea curentă de rezolvare și să revină la o cale anterioară → algoritmi tentativi.
  - Altfel (algoritmul avansează pe o singură direcție) → algoritmi irevocabili.

- Cautari informate vs neinformate
  - Căutările informate beneficiază de informații suplimentare pe care le colectează și le utilizează în încercarea de a ghici direcția în care trebuie explorat spațiul stărilor pentru a găsi soluția.
  - Aceste informații sunt stocate:

- În nodurile din spațiul stărilor:
  - Starea problemei reprezentată de nod;
  - Părintele nodului curent;
  - Copii nodului curent (obținuți prin expandarea acestuia);
  - Costul asociat nodului curent care estimează calitatea nodului f(n);
  - Adâncimea de explorare.
- În structuri auxiliare pentru diferențierea nodurilor în raport cu gradul de prelucrare:
  - Expandat (închis) – toți succesorii nodului sunt cunoscuți;
  - Explorat (deschis) – nodul e cunoscut, dar succesorii săi nu;
  - Neexplorat – nodul nu e cunoscut.

- Listele Closed si Open
  - OPEN = mulțimea (lista) nodurilor explorate (frontiera dintre zona cunoscută și cea necunoscută).
  - CLOSED = mulțimea (lista) nodurilor expandate (regiunea cunoscută în totalitate).
  - Explorarea zonelor necunoscute se face prin alegerea și expandarea unui nod din OPEN. După expandare, nodul respectiv e trecut în CLOSED.
  - Majoritatea algoritmilor tentativi folosesc lista OPEN, dar doar o parte folosesc lista CLOSED.

- Completitudine si optimalitate
  - Algoritm complet = algoritm de explorare care garantează descoperirea unei soluții, dacă problema acceptă soluție.
    - Algoritmii irevocabili sunt mai rapizi și consumă mai puține resurse decât cei tentativi, dar nu sunt compleți pentru că pierd informație.
  - Algoritm optimal = algoritm de explorare care descoperă soluția optimă a problemei.

- Algoritm generic de explorare:
  -
```
Explorare(StInit, test_sol)
  OPEN = {constr_nod(StInit)}; // starea inițială

  Cât timp (OPEN ≠ Ø)
                    // mai am noduri de prelucrat
    nod = selecție_nod(OPEN); // aleg un nod
    Dacă (test_sol(nod)) Întoarce nod;
                              // am găsit o soluție
    OPEN = OPEN \ {nod} U expandare{nod};
                              // extind căutarea
  Întoarce insucces; // nu s-a găsit nicio soluție
```
  - Dacă selecție_nod se realizează independent de costul nodurilor din graful stărilor → căutare neinformată:
    - Dacă e de tip “random” → algoritm aleator - ex: RandomBounce
    - Dacă e de tip “primul venit, primul servit” → OPEN e coadă → BFS – ex: Breadth-first

- Dacă e de tip “ultimul venit, primul servit” → OPEN e stivă → DFS – ex: Depth-first limitat / IDDFS

- Dacă selecție_nod se bazează pe un cost exact sau estimat (euristic) al stărilor problemei → căutare informată:
  - Estimarea costului și folosirea sa în procesul de selecție → esențiale pentru completitudinea, optimalitatea și complexitatea algoritmilor de explorare!

⇨ Explorare informata irevocabila

- Algoritmul alpinistului = algoritmul gradientului maxim
- Fiecărui nod i se asociază o valoare f(nod) ≥ 0 → calitatea soluției parțiale din care face parte nodul.
- Se păstrează doar cel cu valoare maximă → OPEN are un singur element!
- Gradientul maxim:
  -

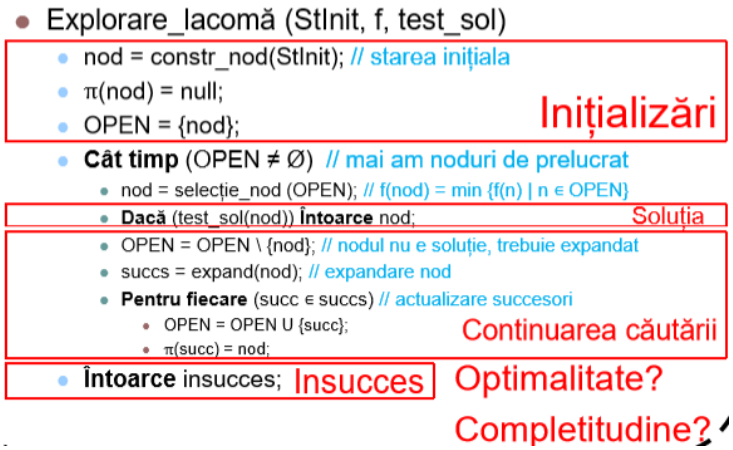
```
Gradient_maxim(StInit, f, test_sol)
  nod = constr_nod(StInit); // starea inițial      Inițializări
  π(nod) = null;

  Cât timp (!test_sol(nod))                        Testez soluția
    succs = expandare(nod); // nodurile au o valoare estimata
                            // prin f
    Dacă (succs = Ø) Întoarce insucces;            Insucces
                            // nu mai am noduri de prelucrat
    succ = selecție_nod(succs); // f(succs) = max {f(n) | n ∈ succs}
    π(succ) = nod;
    nod = succ;                                    Găsesc calea de continuat
  Întoarce nod; // am ajuns la soluție             Soluția
```

- Discutie
  - Algoritmul nu e complet și nu e optimal!
  - Complexitate scăzută: O(bd) - b = branching factor, iar d = depth!
  - Performanțele algoritmului depind foarte tare de forma teritoriului explorat și de euristica folosită (de dorit să existe puține optime locale și o euristică de evaluare cât mai bună).
  - Pseudo-soluție eliminare optim local: se lansează algoritmul de mai multe ori plecând din stări inițiale diferite și se alege cea mai bună soluție obținută.

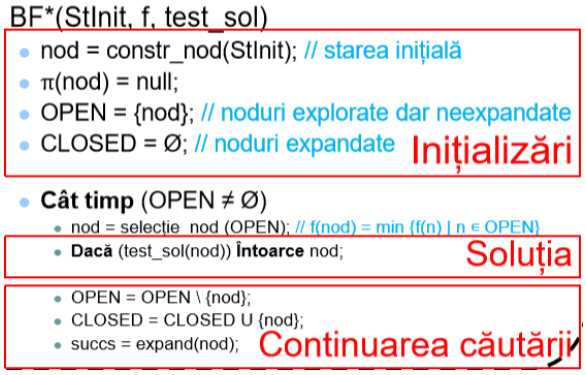
⇨ Explorări tentative informate

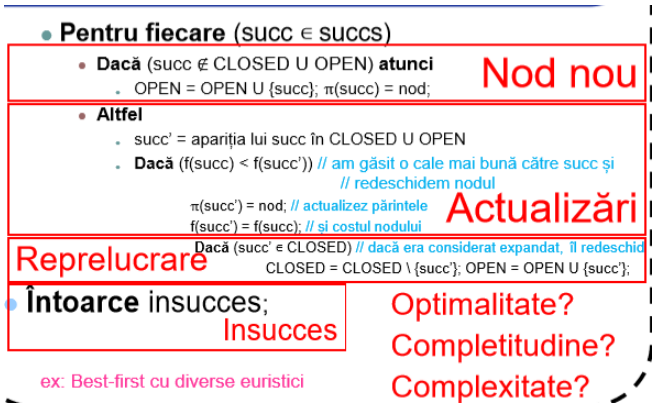
- Păstrează toate nodurile de pe frontieră (OPEN), unii păstrând și nodurile expandate (CLOSED).
- Fiecare nod are un cost asociat f(n) ≥ 0 care estimează calitatea nodului (distanța de la nodul respectiv până la un nod soluție).
- Cu cât f(n) este mai mic, cu atât nodul este mai bun.
- Greedy:

-



  - Explorarea lacomă nu e completă → trebuie să se rețină teritoriul deja parcurs ca să se evite ciclurile!
- Best first*
  -





  - Păstrează întreg teritoriul explorat:
    - OPEN – nodurile de pe frontieră;
    - CLOSED – nodurile expandate (unele noduri pot fi redeschise) → se evită ciclurile.
  - Algoritmul este complet dar nu este optim → optimalitatea depinde de euristica f!
  - Complexitate: O(b^(d+1))

- A*
  - Varianta a BF*
  - Nu poate fi aplicat mereu → trebuie demonstrat că păstrează ordinea soluțiilor unde soluțiile problemelor sunt drumuri în spațiul stărilor!
  - Costul unui drum este aditiv (= suma costurilor arcelor) și crescător în lungul drumului.
  - Folosește două funcții de cost:
    - h(n) - distanța estimată de la nodul curent până la nodul țintă;
    - g(n) - distanța parcursă de la nodul inițial până la nodul curent;
    - f(n) = g(n) + h(n).

- Notatii:

S = (V,E) – graful asociat spațiului stărilor problemei;

n_0 – nodul de start asociat stării inițiale a problemei;

$\Gamma \subseteq V$ – mulțimea nodurilor soluție. Un nod soluție se notează γ;

c(n,n') > 0 – costul arcului (n,n');

π(n) – părintele lui n;

g(n) – costul drumului n_0..n descoperit de algoritm la momentul curent de timp;

$g_p(n)$ – costul exact al porțiunii n_0..n din lungul unei căi date P;

$g^*(n)$ – costul exact al unui drum optim n_0..n;

h(n) ≥ 0 – costul estimat al drumului optim de la nodul n la cel mai favorabil nod soluție $\gamma \in \Gamma$. În plus $h(\gamma) = 0$, pentru orice $\gamma \in \Gamma$;

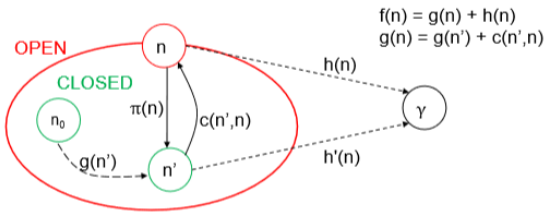
$h^*(n)$ – costul exact al porțiunii de drum n.. γ, pentru cel mai favorabil nod $\gamma \in \Gamma$ ($h^*(n) = \min \{cost(n..\gamma) | \gamma \in \Gamma\}$);

f(n) = g(n) + h(n) – costul estimat al întregului drum n_0..n.. γ, pentru cel mai favorabil nod $\gamma \in \Gamma$, unde porțiunea de drum n_0..n este cea descoperită de algoritm la momentul curent de timp în cursul execuției;

$f^*(n) = g^*(n) + h^*(n)$ – costul exact al unui drum optim n_0..n.. γ, pentru cel mai favorabil nod $\gamma \in \Gamma$;

$C = \min\{f^*(\gamma) | \gamma \in \Gamma\}$ – costul exact al unui drum optim n_0.. γ, $\gamma \in \Gamma$. (C = costul soluției optime);

- Functia de evaluare A*



- Algoritm:

A*(StInit, h, test_sol)

- n_0 = constr_nod(StInit); // starea inițială

Inițializări

- f(n_0) = h(n_0); g(n_0) = 0; π(n_0) = null; // euristici
- OPEN = {n_0}; CLOSED = Ø; // și cozi
- **Cât timp** (OPEN ≠ Ø) // mai am noduri de prelucrat
  - nod = selecție nod (OPEN); // f(nod) = min {f(n) | n ∈ OPEN}
  - **Dacă** (test_sol(nod)) **întoarce** nod;

Soluția

Continuarea căutării

- OPEN = OPEN \ {nod}; // updatez OPEN
- CLOSED = CLOSED U {nod}; // și CLOSE
- succs = expand(nod); // determin nodurile succesoare

- **Pentru fiecare** (succ ∈ succs) { // prelucrare succs

Prelucrare succesori

  - g_succ = g(nod) + c(nod, succ); // calculez g
  - f_succ = g_succ + h(succ); // calculez f = g + h
  - **Dacă** (succ ∉ CLOSED U OPEN) **atunci** // nod nou descoperit →
    - OPEN = OPEN U {succ}; // îl bag în OPEN
      g(succ)= g_succ; f(succ)= f_succ; π(succ) = nod;

Nod nou

  - **altfel** // a mai fost prelucrat
    - **Dacă** (g_succ < g(succ)) // verific dacă noul g este mai mic decât // anteriorul
      g(succ)= g_succ; f(succ)= f_succ; π(succ) = nod; // cale mai bună

Actualizări

Reprelucrare **Dacă** (succ ∈ CLOSED) // dacă era considerat expandat, îl redeschid
CLOSED = CLOSED \ {succ'}; OPEN = OPEN U {succ'};

- **Întoarce** Insucces;

Insucces

ex: A* cu diverse euristici

- Completitudine si optimalitate:
  - Algoritmul A* este complet chiar dacă graful explorat nu este finit.
  - Fie P = n0,n1,...,nm un drum oarecare în graful explorat de A*, astfel încât la un moment T al explorării toate nodurile din P sunt în CLOSED. Atunci, la orice moment de timp egal sau superior lui T, există inegalitatea g(ni) ≤ gp(ni), i = 0,m:
    - Costul nodurilor din CLOSED poate să scadă, dar de fiecare dată când acest lucru se întâmplă, se pierde timp → scoaterea nodului din CLOSED, punerea în OPEN, prelucrarea acestuia încă o dată → trebuiesc evitate aceste

situații → alegerea unei euristici cât mai bune care să minimizeze numărul acestor actualizări!

- Funcția euristică h este admisibilă dacă pentru orice nod n din spațiul stărilor $h(n) \leq h^*(n)$. Cu alte cuvinte, o euristică admisibilă h este optimistă și $h(\gamma) = 0$ pentru orice nod $\gamma \in \Gamma$.
- Algoritmul A* ghidat printr-o euristică admisibilă descoperă soluția optimă dacă există soluții.
- Complexitate:
  - Liniară dacă $|h^*(n) - h(n)| \leq \delta$, unde $\delta \geq 0$ este o constantă.
  - Subexponențială, dacă $|h^*(n) - h(n)| \leq O(\log(h^*(n)))$.
  - Exponențială, altfel, (dar mult mai bună decât a căutărilor neinformate).

- Euristici: consistenta si monotonie
  - O euristică h este consistentă dacă pentru oricare două noduri n și n' ale grafului explorat, astfel încât n' este accesibil din n, există inegalitatea: $h(n) \leq h(n') + k(n,n')$, unde k(n,n') este costul unui drum optim de la n la n'.
  - O euristică h este monotonă dacă pentru oricare două noduri n și n' ale grafului explorat, astfel încât n' este succesorul lui n, există inegalitatea $h(n) \leq h(n') + c(n,n')$, unde c(n,n') este costul arcului (n,n').
  - O euristică este consistentă ⇔ este monotonă.
  - O euristică consistentă este admisibilă.
  - O euristică monotonă este admisibilă.

- Dominanta
  - Fie h1 și h2 două euristici admisibile. Spunem că h1 este mai informată decât h2 dacă $h2(n) < h1(n)$ pentru orice nod $n \notin \Gamma$ din graful spațiului de stare explorat.
  - Un algoritm A1* domină un algoritm A2* dacă orice nod expandat de A1* este expandat și de A2*. (eventual, A2* expandează noduri suplimentare față de A1*, deci A1* poate fi mai rapid ca A2*.)
  - Dacă o euristică monotonă h1 este mai informată decât o euristică monotonă h2, atunci un algoritm A1* condus de h1 domină un algoritm A2* condus de h2.

Cursul 13 – Algoritmi aleatori

⇨ Algorimi aleatori
- Micșorăm timpul de rezolvare a problemei relaxând restricțiile impuse soluțiilor.
- Determinarea soluției optime (Las Vegas): Renunțăm la optimalitate (soluția suboptimală are o marjă de eroare garantată prin calcul probabilistic).
- Găsirea unei singure soluții (Monte Carlo): Găsim o soluție ce se apropie cu o probabilitate măsurabilă de soluția exactă.

⇨ **LAS VEGAS**
- Găsesc soluția corectă a problemei, însă timpul de rezolvare nu poate fi determinat cu exactitate.

- Creșterea timpului de rezolvare -> creșterea probabilității de terminare a algoritmului (găsirea soluției optime).
- Timp = ∞ -> algoritmul se termină sigur (soluția e optimă).
- Probabilitatea de găsire a soluției crește extrem de repede astfel încât să se determine soluția optimă într-un timp suficient de scurt.
- Complexitate:
  - Algoritmii Las Vegas au complexitatea f(n) = O(g(n)) dacă exista c > 0 și n0 > 0 a.î. oricare n ≥ n0 avem 0 < f(n) < c α g(n) cu o probabilitate de cel puțin 1 – n^(-α) pentru α > 0 fixat și suficient de mare.

⇨ **MONTE CARLO**

- Găsesc o soluție a problemei care nu e garantat corectă (soluție aproximativă).
- Timp = ∞ -> soluția corectă a problemei.
- Probabilitatea ca soluția să fie corectă crește o dată cu timpul de rezolvare.
- Soluția găsită într-un timp acceptabil este aproape sigur corectă.
- Complexitate:
  - Algoritmii Monte Carlo au complexitatea f(n) = O(g(n)) dacă exist c > 0 și n0 > 0 astfel încât:
    - Oricare n ≥ n0, 0 < f(n) ≤ c α g(n) cu o probabilitate de cel puțin 1 – n^(-α) pentru α > 0 fixat si suficient de mare;
    - Probabilitatea ca soluția determinată de algoritm să fie corectă este cel puțin 1 - n^(-α).