

Parcurgerea în lătime

- Graf orientat sau neorientat, conex, ponderat sau neponderat
- Determinarea arborelui parțial de cost minim pentru grafuri ponderate
- Aplicații: determinarea componentelor conexe, drumuri minime

Pseudocod:

```
pentru fiecare  $x \in V$  execută    cat timp  $C \neq \emptyset$  execută
    viz[x] = 0                                i = extrage(C);
    tata[x] = 0                                afiseaza(i);
    d[x] =  $\infty$                                 pentru j vecin al lui i ( $ij \in E$ )
                                                daca viz[j]==0 atunci
                                                adauga(j, C)
                                                viz[j] = 1
                                                tata[j] = i
                                                d[j] = d[i]+1
BFS(s)
coada C =  $\emptyset$ 
adauga(s, C)
viz[s] = 1; d[s] = 0
```

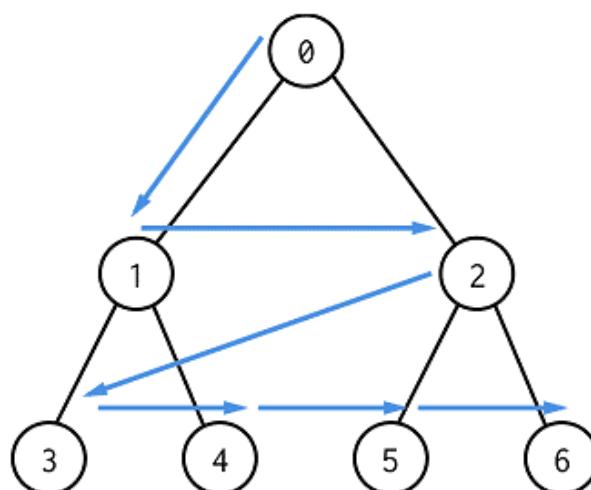
Pentru a parcurge toate vârfurile grafului se reia apelul subprogramului BFS pentru vârfuri rămase nevizitate:

```
pentru fiecare  $x \in V$  execută
    daca viz[x] == 0 atunci
        BFS(x)
```

Complexitate:

- Matrice de adiacență: $O(n^2)$
- Listă de adiacență: $O(n + m)$

Exemplu:



Parcursarea în adâncime

- Graf orientat sau neorientat, conex, ponderat sau neponderat
- Determinarea arborelui parțial de cost minim pentru grafuri ponderate
- Aplicații: determinarea componentelor conexe, arbore parțial, cicluri și circuite și componente tare conexe

Pseudocod:

```
DFS(x)
    culoare[x] = gri
    timp = timp + 1
    desc[x] = timp; //incepe explorarea varfului x
    pentru fiecare xy ∈ E //y vecin al lui x
        daca culoare[y]==alb atunci
            tata[y] = x
            d[y] = d[x]+1 //nivel, nu distanta
            DFS(y)
    culoare[x] = negru
    timp = timp + 1
    fin[x] = timp //s-a finalizat explorarea varfului x
```

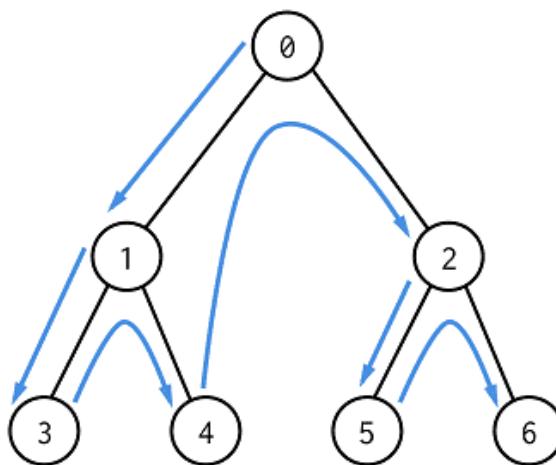
Apel:

```
pentru fiecare x in V execută
    culoare[x] = alb
    timp = 0
    pentru fiecare x in V execută
        daca culoare[x] == alb
            DFS(x)
```

Complexitate:

- Matrice de adiacență: $O(n^2)$
- Lista de adiacență: $O(n + m)$
- Traversare: $O(n)$

Exemplu:



Algoritmul lui Kosaraju

- Determinarea componentelor tare conexe dintr-un graf orientat

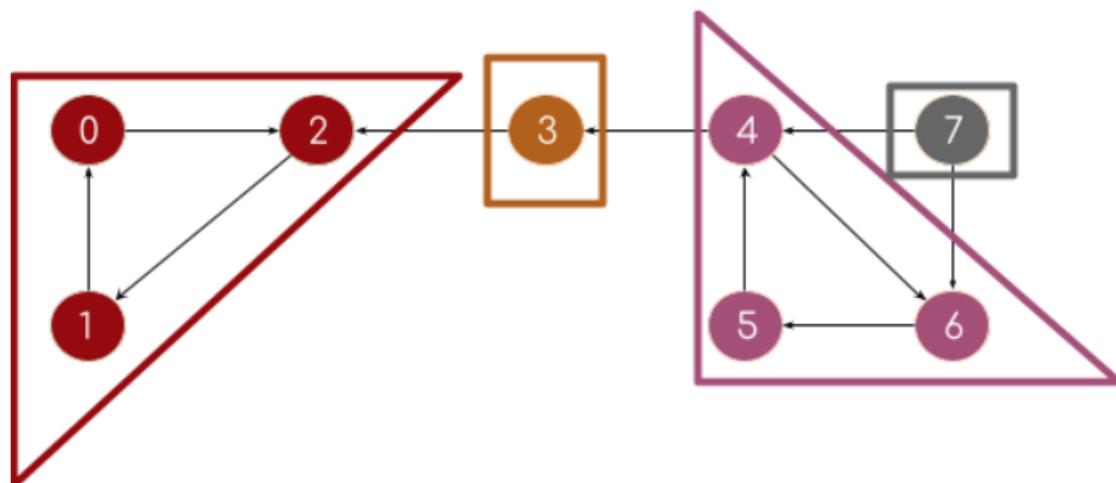
Algoritm:

1. Parcugerea grafului cu DFS și introducerea vârfurilor într-o stivă
2. Inversarea sensurilor din graf și parcugerea nodurilor cu DFS având ca nod de start elemetele din stivă (descrescător după timpul de finalizare)
3. Vârfurile vizitate în fiecare apel al DFS-ului în graful invers reprezintă o componentă tare conexă

Complexitate:

- Două parcugeri și construirea grafului invers: $O(n + m)$

Exemplu:



Componente tare conexe:

- Componenta 1: 0, 2, 1
- Componenta 2: 3
- Componenta 3: 4, 6, 5
- Componenta 4: 7

Sortarea topologică

- Graf orientat, aciclic, ponderat sau neponderat
- Ordonarea vârfurilor astfel încât dacă $uv \in E$ atunci u se află înaintea lui v în ordonare
- Aplicații: ordinea de calcul unde intervin dependențe, detectia de deadlock, determinarea drumurilor critice

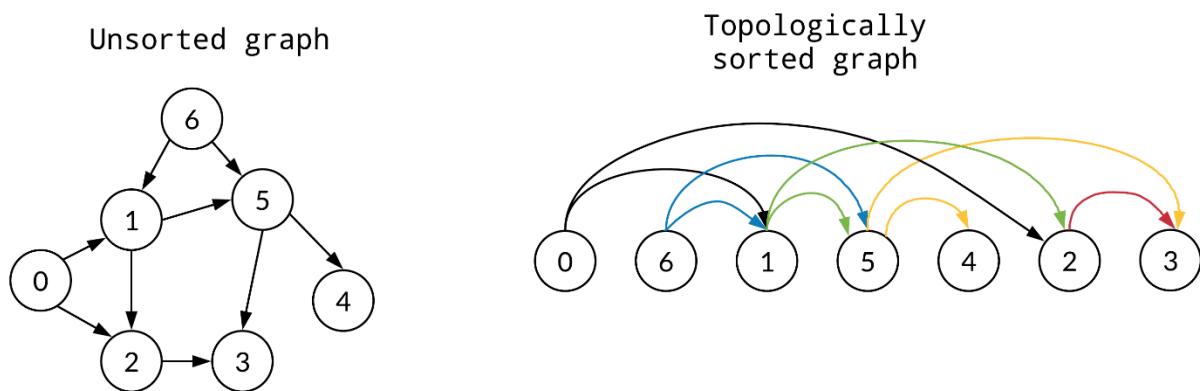
Algoritm:

1. Adăugăm toate vârfurile cu grad intern 0 într-o coadă
2. Extragem un vârf din coadă și îl eliminăm din graf (nu din coadă) și scădem gradele interne ale vecinilor
3. Repetăm pasul 1 și 2 până când nu mai avem noduri rămase

Pseudocod:

```
coada C = ∅;  
adauga in C toate vârfurile v cu  $d^-[v]=0$   
  
cat timp  $C \neq \emptyset$  executa  
    i ← extrage(C);  
    adauga i in sortare  
    pentru  $ij \in E$  executa  
         $d^-[j] = d^-[j] - 1$   
        daca  $d^-[j]==0$  atunci  
            adauga(j, C)
```

Complexitate: $O(n + m)$



Algoritmul lui Kruskal

- Graf conex, ponderat, neorientat
- Determinarea arborelui parțial de cost minim: graf conex fără cicluri
- Aplicații: proiectarea de rețele, clustering, protocoale de rutare

Algoritm:

1. Sortăm muchiile crescător după cost (algoritm Greedy)
2. Selectăm muchia de cost minim care nu formează cicluri cu muchiile deja selectate (care unește două componente conexe)
3. Pentru a testa dacă se formează un ciclu vom marca două noduri unite cu aceeași culoare și vom unii două noduri doar dacă au culori diferite
4. Oprim algoritmul când toate nodurile au aceeași culoare

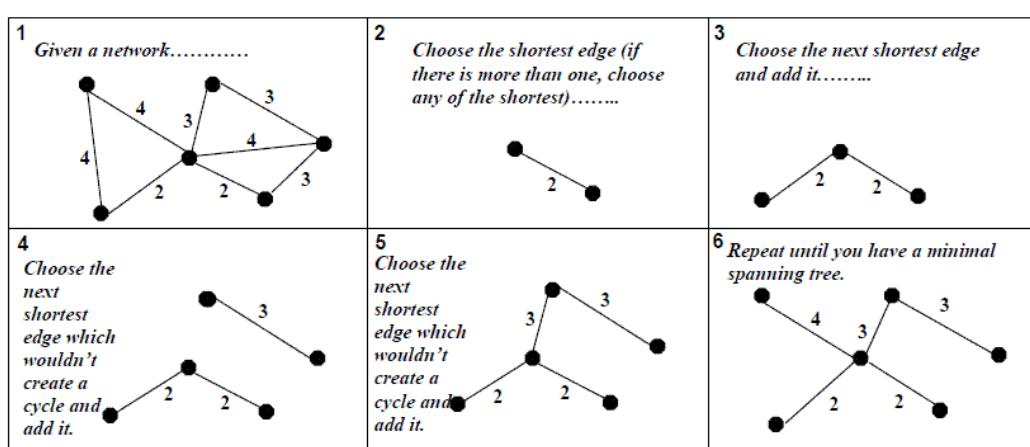
Pseudocod:

```

sorteaza(E)
for(v=1;v<=n;v++)
    Initializare(v) ;
    nrmsele=0
    for(uv ∈ E)
        if(Reprez(u) !=Reprez(v) )
        {
            E(T) = E(T) ∪ {uv} ;
            Reuneste(u,v) ;
            nrmsele=nrmsele+1;
            if(nrmsele==n-1)
                STOP ;
        }
    }

```

Complexitate: $O(m \log n)$



Algoritmul lui Prim

- Graf conex, ponderat, neorientat
- Determinarea arborelui parțial de cost minim: graf conex fără cicluri
- Aplicații: proiectarea de rețele, clustering, protocoale de rutare

Algoritm:

1. Selectăm o muchie de cost minim de la un vârf deja adăugat în arbore la unul neadăugat
2. Pentru fiecare vârf neselectat memorăm doar muchia de cost minim care îl unește de un vârf deja adăugat în arbore

Pseudocod:

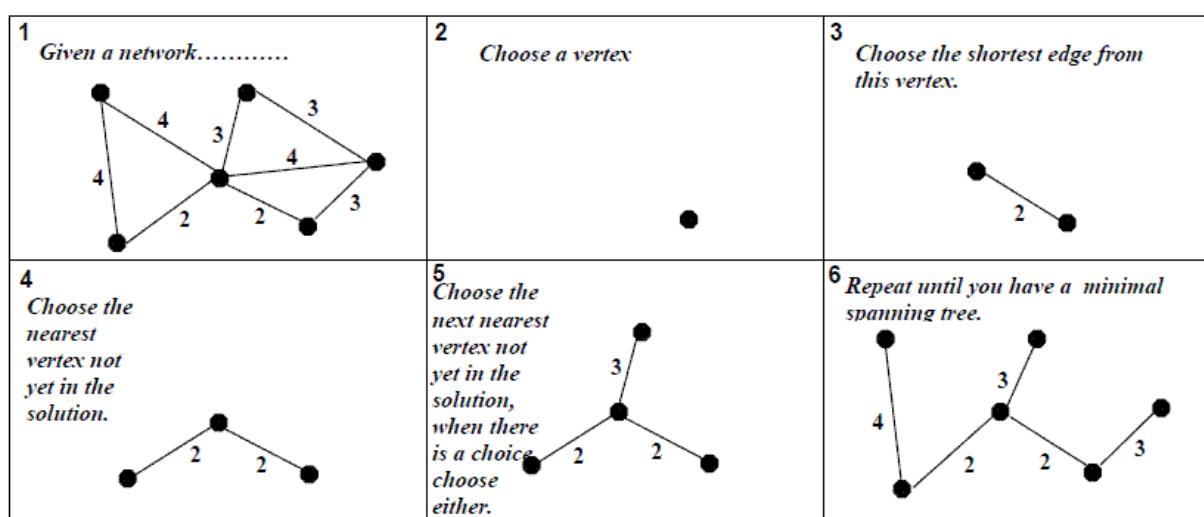
```

s - vârful de start
initializează Q cu V
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
     $d[s] = 0$ 
cat timp  $Q \neq \emptyset$  execută
    extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă
    pentru fiecare  $uv \in E$  execută
        daca  $v \in Q$  si  $w(u,v) < d[v]$  atunci
             $d[v] = w(u,v)$ 
             $tata[v] = u$ 
    scrie  $(u, tata[u])$ , pentru  $u \neq s$ 
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
    initializează Q cu V
    cat timp  $Q \neq \emptyset$  execută
         $u =$  extrage vârf cu eticheta  $d$  minimă din Q
        pentru fiecare  $v$  adjacent cu  $u$  execută
            daca  $v \in Q$  si  $w(u,v) < d[v]$  atunci
                 $d[v] = w(u,v)$ 
                 $tata[v] = u$ 
            //actualizează Q - pentru Q heap
        scrie  $(u, tata[u])$ , pentru  $u \neq s$ 

```

Complexitate:

- Implementare cu coadă: $O(n^2)$
- Implementare cu min-heap: $O(m \log n)$



Directed Acyclic Graph

- Graf orientat, aciclic, ponderat și poate avea ponderi negative
- Determinarea drumului minim de la o sursă unică

Algoritm:

1. Considerăm vârfurile în ordinea dată de sortarea topologică
2. Pentru fiecare vârf relaxăm arcele către vecinii săi pentru a găsi posibilele drumuri noi minime către aceștia

Pseudocod:

`s - vârful de start`

```
//initializam distante - ca la Dijkstra
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0

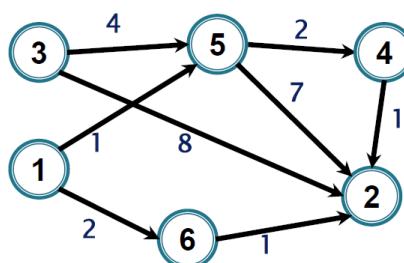
//determinăm o sortare topologică a vârfurilor
SortTop = sortare_topologica(G)

pentru fiecare u ∈ SortTop
    pentru fiecare uv∈E executa
        daca d[u]+w(u,v)< d[v] atunci //relaxam uv
            d[v] = d[u]+w(u,v)
            tata[v] = u

scrie d, tata
```

Complexitate: $O(m + n)$

- Inițializare: $O(n)$
- Sortare: $O(m + n)$
- Relaxări: $O(m)$



Sortare topologică

1, 3, 6, 5, 4, 2

`s=3` - vârf de start

Ordine de calcul distanțe:

1, 3, 6, 5, 4, 2

d/tata	1	2	3	4	5	6
$u = 1:$	$\infty/0,$	$\infty/0,$	$0/0,$	$\infty/0,$	$\infty/0,$	$\infty/0$
$u = 3:$	$\infty/0,$	$8/3,$	$0/0,$	$\infty/0,$	$4/3,$	$\infty/0$
$u = 6:$	$\infty/0,$	$8/3,$	$0/0,$	$\infty/0,$	$4/3,$	$\infty/0$
$u = 5:$	$\infty/0,$	$8/3,$	$0/0,$	$6/5,$	$4/3,$	$\infty/0$
$u = 4:$	$\infty/0,$	$7/4,$	$0/0,$	$6/5,$	$4/3,$	$\infty/0$
$u = 2:$	$\infty/0,$	$7/4,$	$0/0,$	$6/5,$	$4/3,$	$\infty/0$

Dijkstra

- Graf orientat, poate fi ciclic, ponderat cu valori pozitive
- Determinarea drumului minim de la o sursă unică

Algoritm:

1. Selectăm o vârf care are asociat costul minim cost minim
2. Pentru fiecare vârf relaxăm arcele către vecinii săi pentru a găsi posibilele drumuri noi minime către aceștia

Pseudocod:

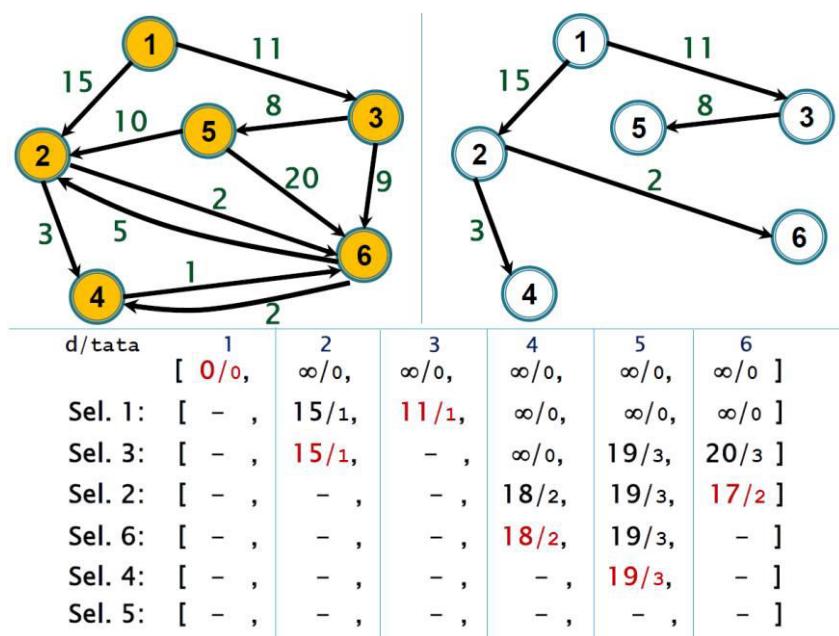
```

initializează multimea vârfurilor neselectate Q cu V pentru fiecare u∈V execută
pentru fiecare u∈V execută
    d[u] = ∞; tata[u]=0
    d[s] = 0
Q = V //creare heap cu cheile din d
cat timp Q ≠ ∅ execută
    u = extrage_min(Q)
    pentru fiecare uv∈E execută
        daca d[u]+w(u,v)< d[v] atunci
            d[v] = d[u]+w(u,v)
            tata[v] = u
scrie d, tata
//scrie drum minim de la s la t un varf t dat folosind tata
    
```

pentru fiecare u∈V execută
 d[u] = ∞; tata[u]=0
 d[s] = 0
 Q = V //creare heap cu cheile din d
 cat timp Q ≠ ∅ execută
 u = extrage_min(Q)
 pentru fiecare uv∈E execută
 daca d[u]+w(u,v)< d[v] atunci
 d[v] = d[u]+w(u,v)
 repara(Q,v)
 tata[v] = u
 scrie d, tata
 //scrie drum minim de la s la t un varf t dat

Complexitate:

- Implementare cu coadă: $O(n^2)$
- Implementare cu min-heap: $O(m \log n)$



Bellman Ford

- Graf orientat, ponderat și ponderile pot fi negative, fără cicluri negative
- În cazul în care există cicluri negative acestea pot fi detectate
- Determinarea drumului minim de la o sursă unică

Algoritm:

1. Relaxăm toate arcele din graf
2. Repetăm primul pas până când nu mai apar îmbunătățiri la distanțe:
vom avea cel mult $n-1$ repetări
3. În cazul în care există un circuit negativ se va ajunge la pasul n

Pseudocod:

```
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

pentru  $i = 1, n-1$  executa
    pentru fiecare  $uv \in E$  executa
        daca  $d[u] + w(u, v) < d[v]$  atunci
             $d[v] = d[u] + w(u, v)$ 
             $tata[v] = u$ 
```

Complexitate: $O(nm)$

Floyd-Warshall

- Graf orientat, ponderat și ponderile pot fi negative, fără cicluri negative
- Drumuri minime între toate perechile de noduri

Algoritm:

1. Inițializare:

- Se construiește o matrice D în care se rețin distanțele inițiale între noduri.
- Dacă există o muchie între nodurile i și j , atunci $D[i][j]$ este setat la ponderile acelei muchii. Dacă nu există o muchie directă, $D[i][j]$ este setat la un infinit mare.
- Pentru fiecare nod i , se setează $D[i][i]$ la 0.

2. Actualizare Matrice D :

- Se parcurg toți nodurile k posibile și se verifică dacă există un drum mai scurt de la i la j trecând prin nodul k .
- Pentru fiecare pereche de noduri i și j , se verifică dacă distanța de la i la j printr-un nod intermediu k este mai mică decât distanța curentă $D[i][j]$.
- Dacă este adevărat, se actualizează $D[i][j]$ cu noua distanță găsită.

3. Verificare Cicluri Negative:

- Se parcurg toate nodurile i și se verifică dacă $D[i][i]$ este negativ. Dacă acest lucru este adevărat pentru cel puțin un nod, atunci există un ciclu negativ în graf.

Pseudocod:

$d[i][j] = w(i,j) - \text{costul arcului } (i,j)$

$$p[i][j] = \begin{cases} i, & \text{daca } ij \in E \\ 0, & \text{altfel} \end{cases}$$

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) {
        d[i][j]=w[i][j];
        if(w[i][j]== ∞)
            p[i][j]=0;
        else
            p[i][j]=i;
    }
    for(k=1;k<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                if(d[i][j]>d[i][k]+d[k][j]){
                    d[i][j]=d[i][k]+d[k][j];
                    p[i][j]=p[k][j];
                }
            }
```

Complexitate: $O(n^3)$

Ford-Fulkerson & Edmonds-Karp

Fie $N = (G, \{s\}, \{t\}, I, c)$ o rețea

- Un lanț este o succesiune de vârfuri distincte și arce din G

$$P = [v_0, e_1, v_1, \dots, v_{k-1}, e_k, v_k]$$

unde arcul e_i este fie $v_{i-1}v_i$, fie v_iv_{i-1}

(P este lanț elementar în graful neorientat asociat lui G)

Dacă

- $e_i = v_{i-1}v_i \in E(G)$, e_i s.n arc direct (înainte) în P
- $e_i = v_iv_{i-1} \in E(G)$, e_i s.n arc invers (înapoi) în P



- Fie N rețea, f flux în N , P un lanț

- Asociem fiecărui arc e din P o pondere, numită **capacitate reziduală** în P :

$$i_P(e) = \begin{cases} c(e) - f(e), & \text{dacă } e \text{ este arc direct în } P \\ f(e), & \text{dacă } e \text{ este arc invers în } P \end{cases}$$

= cu cât mai poate fi modificat fluxul pe arcul e , de-a lungul lanțului P

- ▶ Capacitatea reziduală a unui lanț P este

$$i(P) = \min\{i_p(e) \mid e \in E(P)\}$$

= cu cât mai poate fi modificat fluxul de-a lungul lanțului P

- ▶ Convenție: Dacă $E(P) = \emptyset$, $i(P) = \infty$

- ▶ Un s-t lanț P se numește

- **f-nesaturat (f-drum de creștere)** augmenting path dacă $i(P) \neq 0$
- **f-saturat** dacă $i(P) = 0$

Fluxuri în rețele de transport

- ▶ Fie N- rețea, f flux în N, P un s-t lanț **f-nesaturat**.
- ▶ Fluxul revizuit de-a lungul lanțului P se definește ca fiind $f_P : E \rightarrow \mathbb{N}$,

$$f_P(e) = \begin{cases} f(e) + i(P), & \text{dacă } e \text{ este arc direct în } P \\ f(e) - i(P), & \text{dacă } e \text{ este arc invers în } P \\ f(e), & \text{altfel} \end{cases}$$

Algoritm generic de determinare a unui flux maxim – algoritmul FORD – FULKERSON

- Fie f un flux în N (de exemplu $f \equiv 0$ fluxul vid:
 $f(e) = 0, \forall e \in E$)
- Cât timp există un s-t lanț f -nesaturat P în G
 - determină un astfel de lanț P
 - revizuește fluxul f de-a lungul lanțului P
- returnează f

Algoritm FORD – FULKERSON

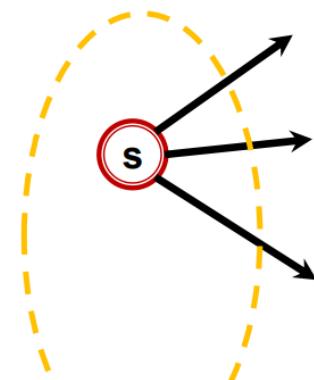
Complexitate

- $O(mL)$, unde L este capacitatea minimă a unei tăieturi

Avem $L \leq c^+(s) \leq n \cdot C$, unde $C = \max\{c(e) | e \in E(G)\} \Rightarrow$

- $O(nmC)$ unde

$$C = \max\{c(e) | e \in E(G)\}$$



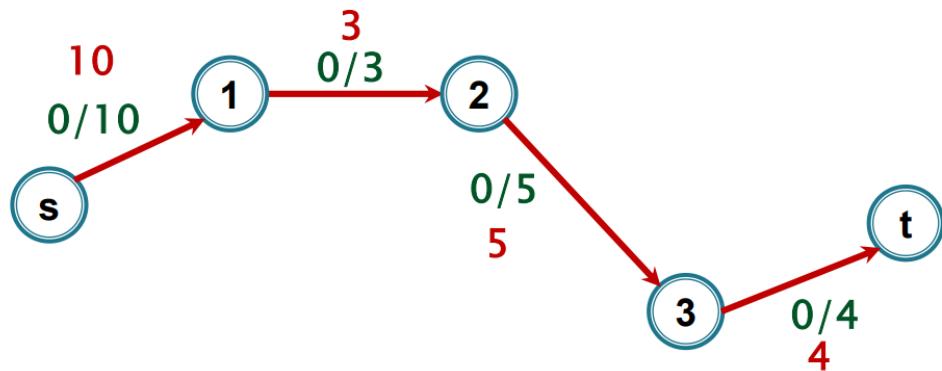
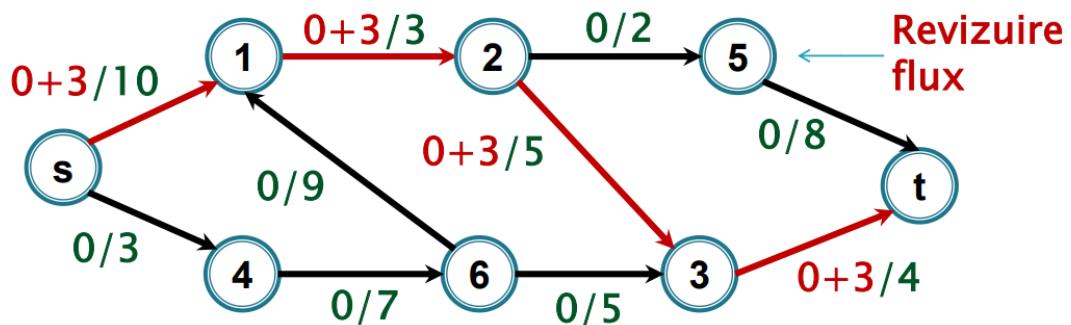
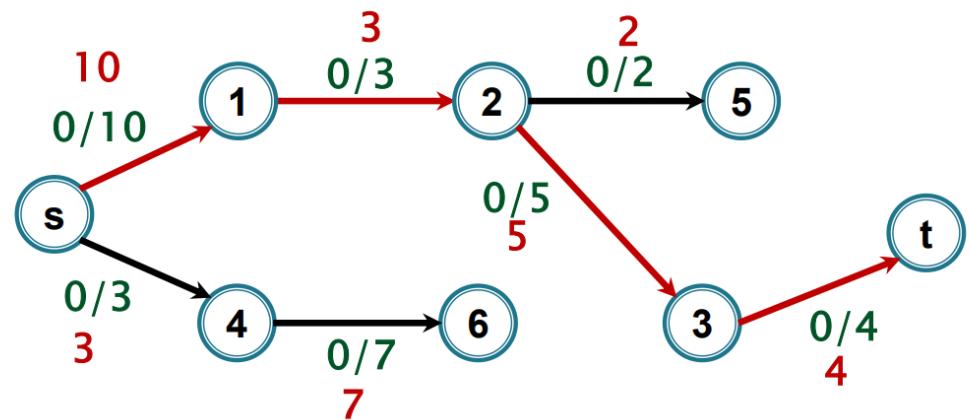
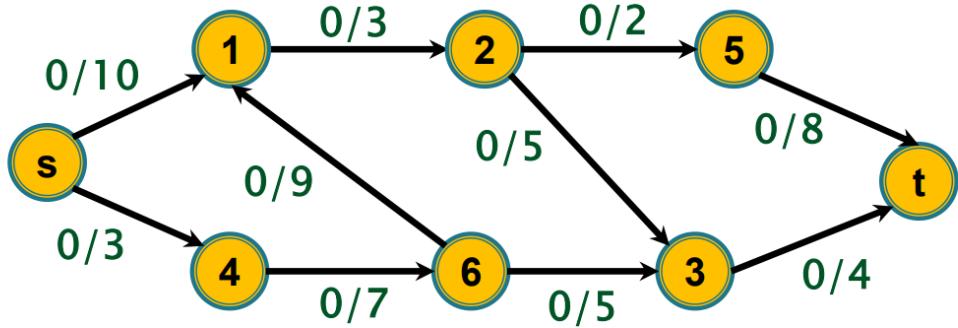
Implementare. Algoritmul Edmonds-Karp

construieste_s-t_lant_nesat_BF() – construiește un s–t lanț nesaturat prin parcurgerea BF din s

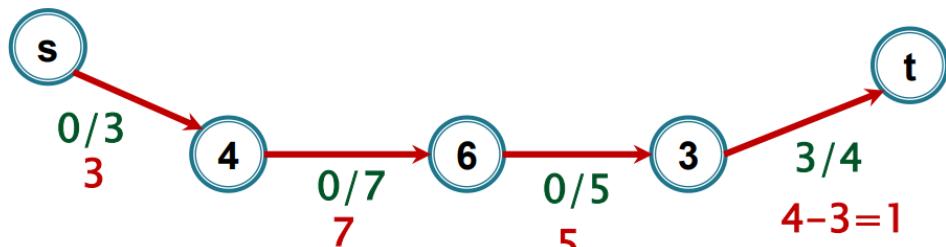
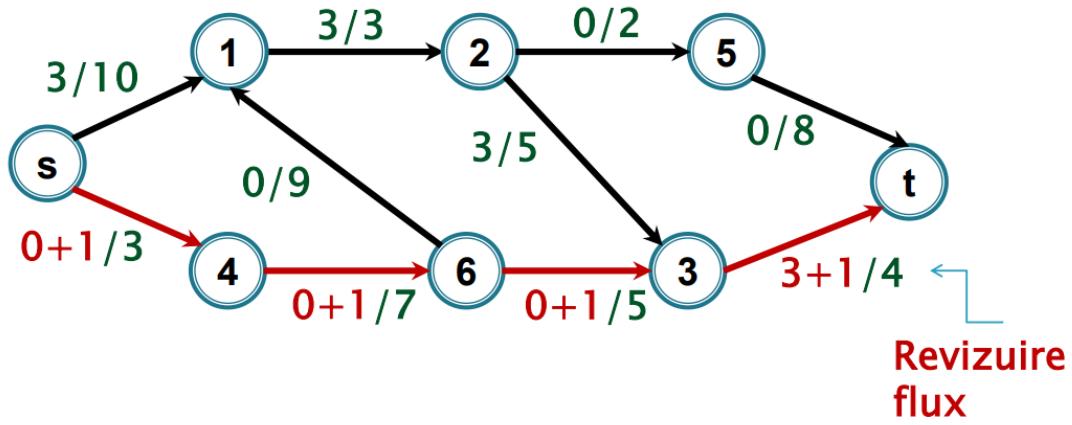
- sunt considerate în parcursare doar arce pe care se poate modifica fluxul, adică având capacitate reziduală pozitivă
- Returnează **false** dacă un astfel de lanț nu există
(și **true** dacă l-a putut construi)

revizuieste_flux_lant()

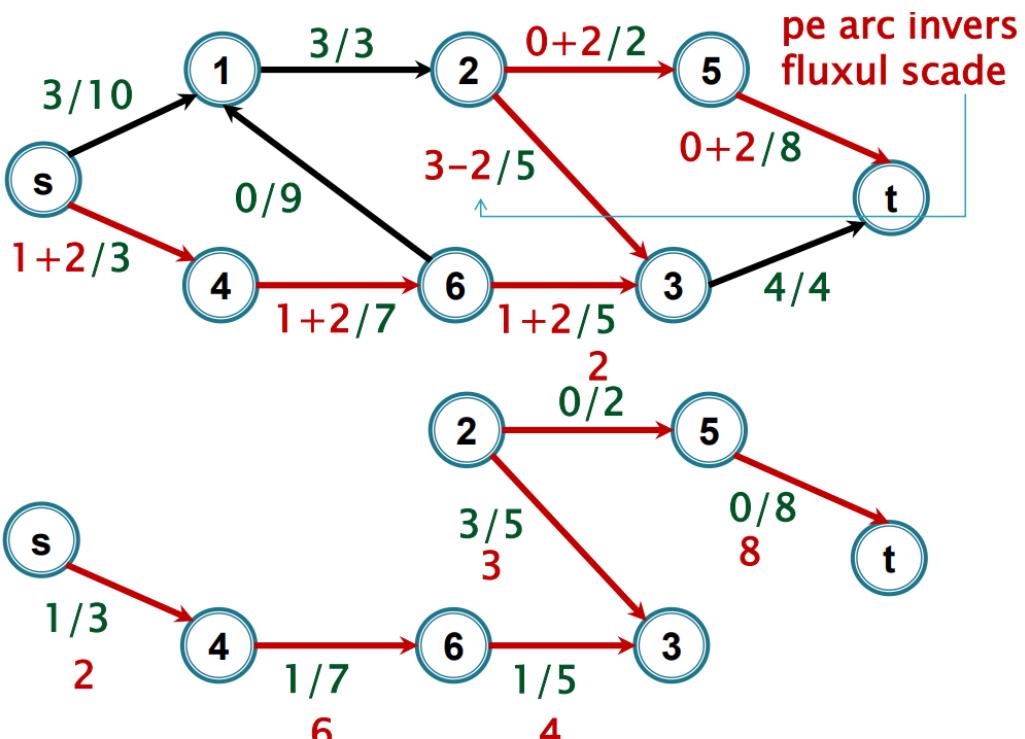
- fie P s–t lanțul găsit în **construieste_s-t_lant_nesat_BF()**
- calculăm $i(P)$
- pentru fiecare arc e al lanțului P
 - creștem cu $i(P)$ fluxul pe e dacă este arc direct
 - scădem cu $i(P)$ fluxul pe e dacă este arc invers



$$i(P) = \min \{10, 3, 5, 4\} = 3$$



$$i(P) = \min\{ 3, 7, 5, 1 \} = 1$$



$$i(P) = \min\{ 2, 6, 4, 3, 2, 8 \} = 2$$

Pseudocod EDMONDS-KARP:

```
construieste_s-t_lant_nesat_BF()
    pentru(v∈V) executa tata[v] ←0; viz[v] ←0
    coada C ← ∅
    adauga(s, C)
    viz[s]← 1
    cat timp C ≠ ∅ executa
        i ← extrage(C)
        pentru (ij ∈ E) executa      arc direct
            dacă (viz[j]=0 și c(ij)-f(ij)>0) atunci
                adauga(j, C)
                viz[j] ← 1; tata[j] ← i
                daca (j=t) atunci STOP și returnează true(1)
        pentru (ji ∈ E) executa      arc invers
            daca (viz[j]=0 și f(ji)>0) atunci
                adauga(j, C)
                viz[j] ← 1; tata[j] ← -i
                daca (j=t) atunci STOP și returnează true(1)
    returnează false(0)
```

► Complexitate

- Algoritm generic Ford Fulkerson O(mL)/O(nmC)
- Implementare Edmonds Karp O(nm²)

Grafuri Euleriene

Lemă

Fie $G=(V,E)$ un graf neorientat, conex, cu **toate vârfurile de grad par** și $E \neq \emptyset$.

Atunci pentru orice $x \in V$ există un ciclu C în G cu $x \in V(C)$
(ciclu care conține x , nu neapărat eulerian, nici neapărat elementar)

Teorema lui Euler

Fie $G=(V, E)$ un **(multi)graf** neorientat, conex, cu $E \neq \emptyset$.

Atunci

G este eulerian \Leftrightarrow orice vârf din G are grad par

Teorema lui Euler

Fie $G=(V, E)$ un graf neorientat, conex, cu $E \neq \emptyset$.

Atunci

G are un lanț eulerian $\Leftrightarrow G$ are cel mult două vârfuri de grad impar

Teorema lui Euler

Fie $G=(V, E)$ un graf orientat, conex (= graful neorientat asociat este conex), cu $E \neq \emptyset$.

Atunci

$$G \text{ este eulerian} \Leftrightarrow \forall v \in V \quad d_G^-(v) = d_G^+(v)$$

Teorema lui Euler

Fie $G=(V, E)$ un (multi)graf orientat, conex, cu $E \neq \emptyset$.

Atunci

G are un drum eulerian \Leftrightarrow

$$(\forall v \in V \quad d_G^-(v) = d_G^+(v)) \text{ sau}$$

$$(\exists x \in V \text{ cu } d_G^-(x) = d_G^+(x) - 1,$$

$$\exists y \in V, y \neq x \text{ cu } d_G^-(y) = d_G^+(y) + 1,$$

$$\forall v \in V - \{x, y\} \quad d_G^-(v) = d_G^+(v))$$

Teoremă – Descompunere euleriană

Fie $G=(V, E)$ un graf orientat, conex (= graful neorientat asociat este conex), cu **exact $2k$ vârfuri de grad impar** ($k > 0$).

Atunci există o k -descompunere euleriană a lui G și k este cel mai mic cu această proprietate.

Colorări în Grafuri

Aplicații p -colorări

De câte săli este nevoie minim pentru programarea într-o zi a n conferințe cu intervale de desfășurare date?

Conf. 1: interval (1,4)

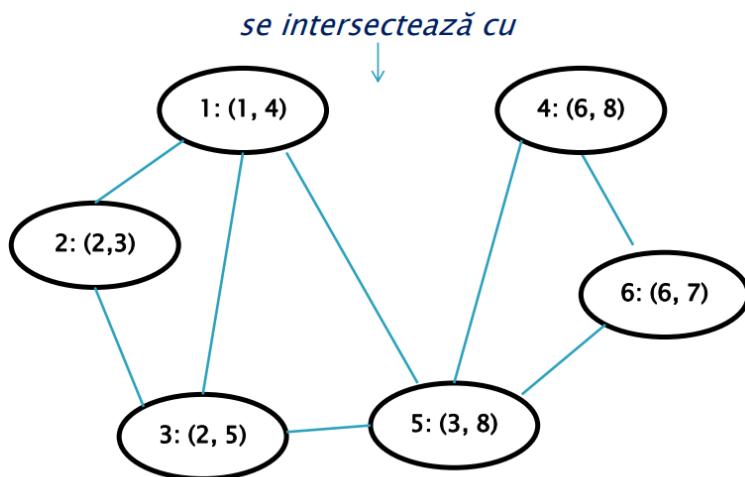
Conf. 2: interval (2,3)

Conf. 3: interval (2,5)

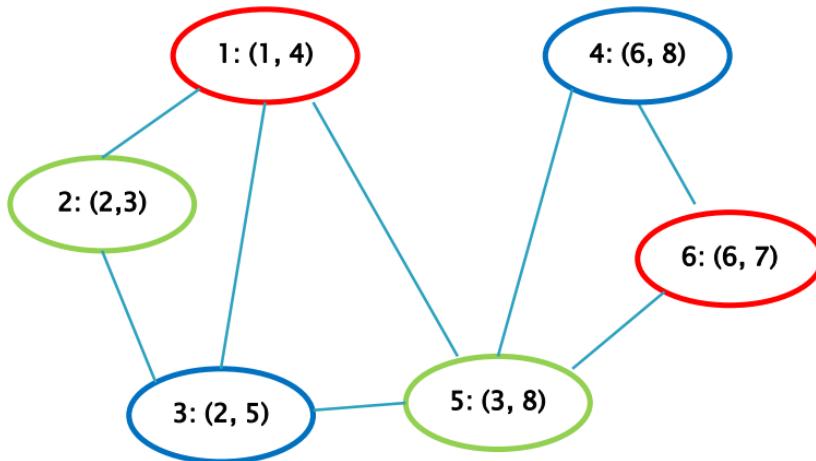
Conf. 4: interval (6,8)

Conf. 5: interval (3,8)

Conf. 6: interval (6,7)



Graful intersecției intervalelor este 3-colorabil:



Sunt necesare minim 3 săli (corespunzătoare celor 3 culori):

Sala 1: (1,4), (6,7)

Sala 2: (2,3), (3,8)

Sala 3: (2,5), (6,8)



Graf Bipartit

► Teorema König – Caracterizarea grafurilor bipartite

Fie $G = (V, E)$ un graf cu $n \geq 2$ vârfuri.

Avem

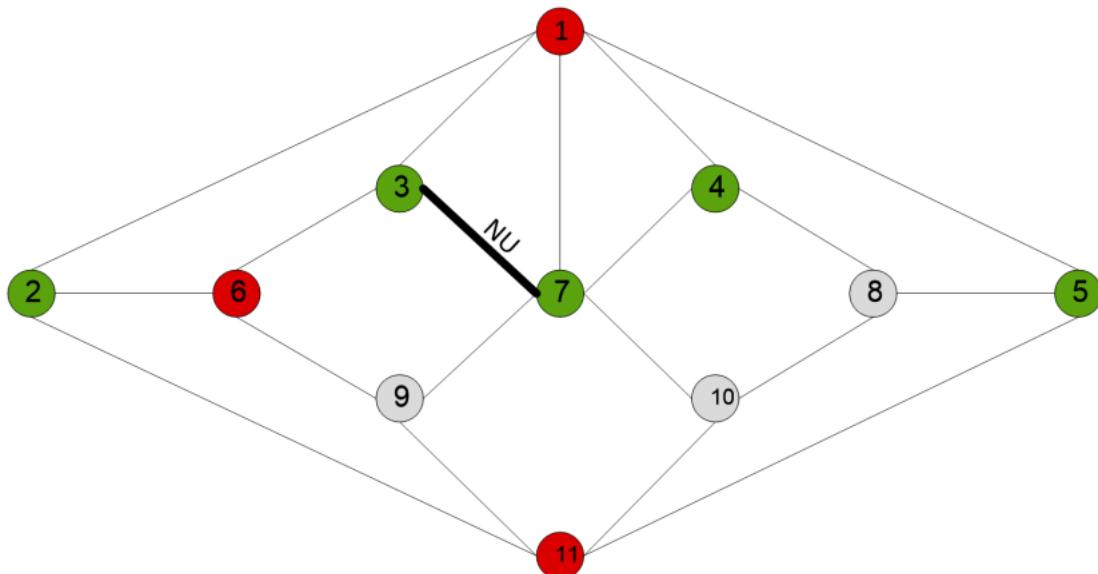
G este bipartit \Leftrightarrow toate ciclurile elementare

din G sunt pare

► Teorema König \Rightarrow Algoritm pentru a testa dacă un graf conex este bipartit

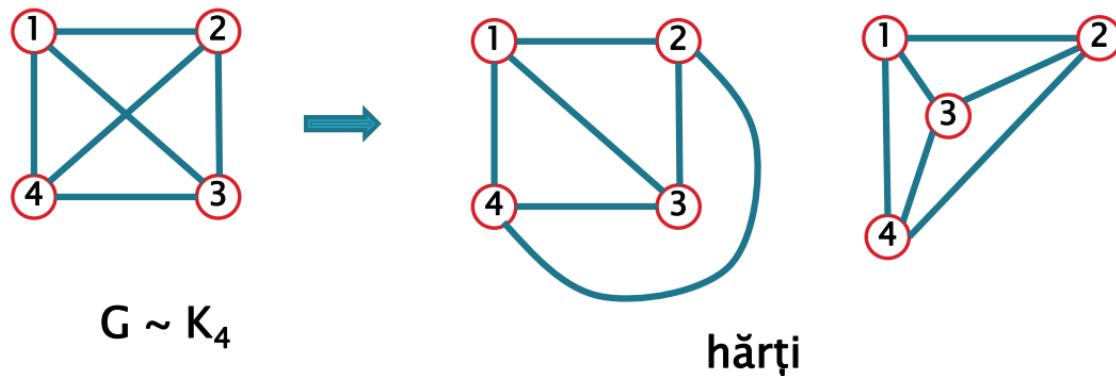
- Colorăm cu (cel mult) 2 culori un arbore parțial al său printr-o parcurgere (colorăm orice vecin j nevizitat al vârfului curent i cu culoarea diferită de cea a lui i)
- Testăm dacă celelalte muchii – de la i la vecini j deja vizitați (colorați) au extremitățile i și j colorate diferit

Dacă graful nu este conex, testăm fiecare componentă conexă

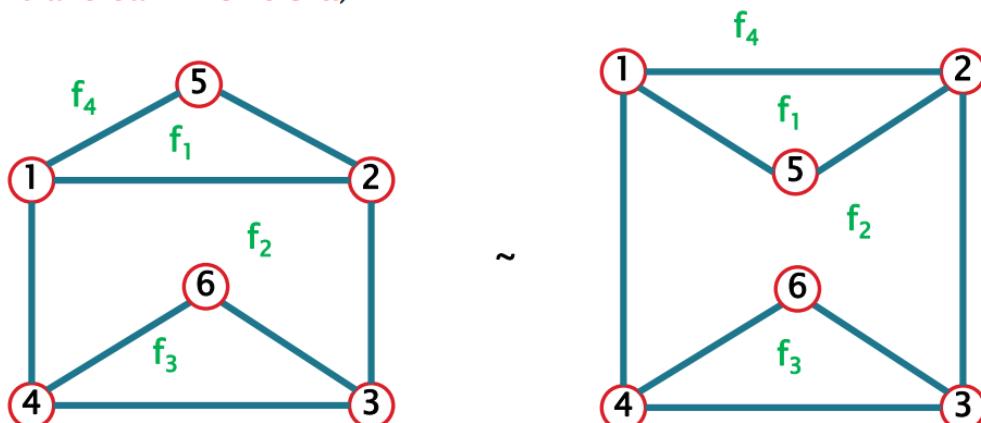


Grafuri Planare

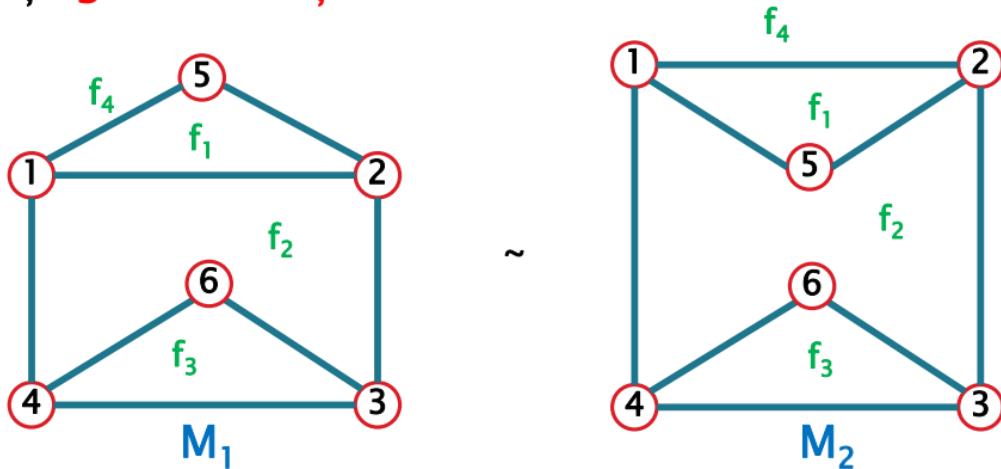
- ▶ $G = (V, E)$ graf neorientat s.n. planar \Leftrightarrow admite o reprezentare în plan a.î. muchiilor le corespund segmente de curbe continue care nu se intersectează în interior unele pe altele
- ▶ O astfel de reprezentare s.n. hartă a lui G



- ▶ $M = (V, E, F)$ hartă
- ▶ Pentru o față $f \in F$ definim
 - $d_M(f) =$ gradul feței $f =$ numărul muchiilor lanțului închis (**frontierei**) care delimitizează f (*câte muchii sunt parcuse atunci când traversăm frontiera*)



Observație: Hărți diferite ale aceluiași graf pot avea secvența gradelor fețelor diferită



$$d_{M_1}(f_1) = 3$$

$$d_{M_1}(f_2) = 5$$

$$d_{M_1}(f_3) = 3$$

$$d_{M_1}(f_4) = 5$$

$$d_{M_2}(f_1) = 3$$

$$d_{M_2}(f_2) = 6$$

$$d_{M_2}(f_3) = 3$$

$$d_{M_2}(f_4) = 4$$

- ▶ $M = (V, E, F)$ hartă

- Avem

$$\sum_{f \in F} d_M(f) = 2 |E|$$

(deoarece o muchie este incidentă cu două fețe)

► Teorema poliedrală a lui EULER

Fie $G=(V, E)$ un graf planar conex și $M = (V, E, F)$ o hartă a lui. Are loc relația

$$|V| - |E| + |F| = 2$$

► Consecință

Orice hartă M a lui G are $2 - |V| + |E|$ fețe

► Proprietăți

Fie $G=(V, E)$ un graf planar conex cu $n=|V|>2$ și $m=|E|$.

Atunci:

- a) $m \leq 3n - 6$
- b) $\exists x \in V$ cu $d(x) \leq 5$.

► Consecință

K_5 nu este graf planar

► Proprietăți (temă)

Fie $G=(V, E)$ un graf planar conex bipartit cu $n=|V|>2$ și $m=|E|$. Atunci:

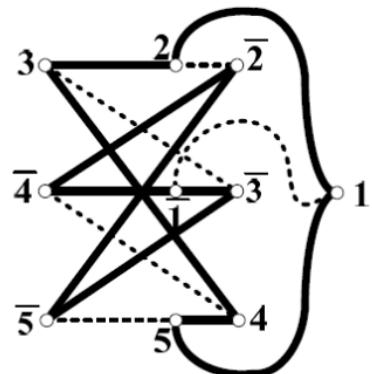
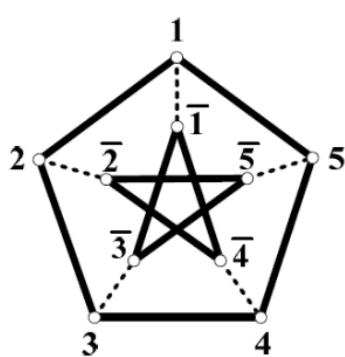
- a) $m \leq 2n - 4$
- b) $\exists x \in V$ cu $d(x) \leq 3$.

► Consecință

$K_{3,3}$ nu este graf planar

► Teorema lui Kuratowski

G este graf planar \Leftrightarrow nu conține subdiviziuni ale lui $K_{3,3}$ și ale lui K_5 .



Graful lui Petersen

► Teorema celor 6 culori

Orice graf planar conex este 6 -colorabil.

► Algoritm de colorare a unui graf planar cu 6 culori

colorare(G)

daca $|V(G)| \leq 6$ atunci coloreaza varfurile cu culori distincte din $\{1, \dots, 6\}$

altfel

alege x cu $d(x) \leq 5$

colorare($G - x$)

colorează x cu o culoare din $\{1, \dots, 6\}$

diferită de culorile vecinilor deja

colorați (!se poate, x are cel mult 5

vecini din $G - x$)

► Algoritm de colorare a unui graf planar cu 6 culori

Sugestie implementare nerecursivă

1. Determinarea iterativă a ordinii v_1, \dots, v_n în care sunt colorate vârfurile

- de la ultimul la primul astfel:

- v_n – un varf de grad ≤ 5 în G
- v_{n-1} – un varf de grad ≤ 5 în $G - v_n$
- v_{n-2} – un varf de grad ≤ 5 în $G - \{v_n, v_{n-1}\}$
- ...
- v_i – un varf de grad ≤ 5 în $G - \{v_n, v_{n-1}, \dots, v_{i+1}\}$
- ...
- v_1 – un varf de grad ≤ 5 în $G - \{v_n, v_{n-1}, \dots, v_2\}$

2. Colorăm pe rând vârfurile v_1, \dots, v_n cu o culoare din $\{1, \dots, 6\}$

diferită de culorile vecinilor deja colorați

Determinarea iterativă a ordinii în care sunt coloante vârfurile la pasul 1 se poate face similar cu determinarea unei ordonări topologice:

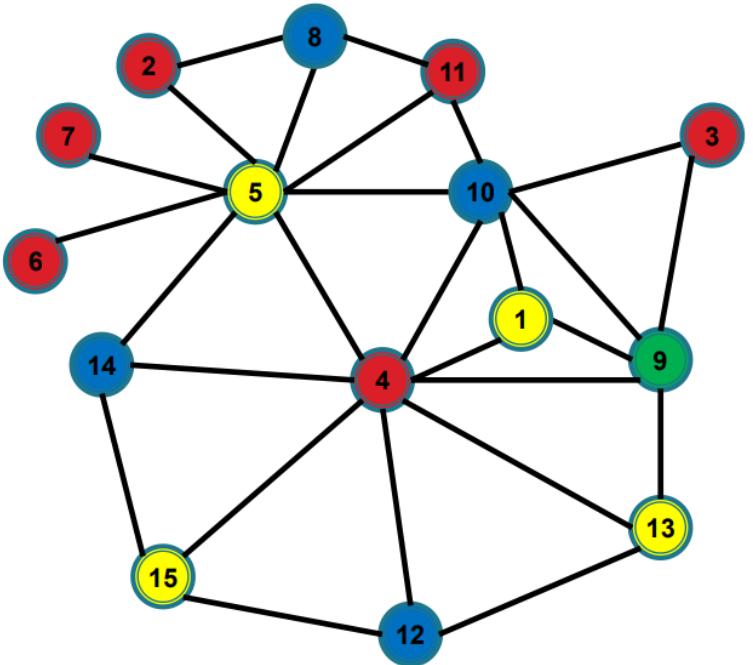
```
coada C = Ø;
adauga in C toate vârfurile v cu d[v] ≤ 5 și
marcheaza-le ca vizitate

stiva S = Ø

cat timp C ≠ Ø executa
    i ← extrage(C);
    adauga i in S

    pentru ij ∈ E execută
        d[j] = d[j] - 1
        dacă d[j] ≤ 5 și j este nevizitat atunci
            adauga(j, C)

cat timp S este nevida execută
    u = pop(S)
    adauga u in sortare
```



- Culoarea 1 –
- Culoarea 2 –
- Culoarea 3 –
- Culoarea 4 –
- Culoarea 5 –
- Culoarea 6 –

Ordinea în care se colorează vârfurile

4, 5, 10, 15, 14, 13, 12, 11, 9, 8, 7, 6, 3, 2, 1

- cu prima culoare disponibilă din cele 6 (nefolosită de un vecin)

► Teorema celor 5 culori

Orice graf planar conex este 5 -colorabil.

Algoritm Greedy de coloare

Ordonări ale vârfurilor – strategii generale

- **SL Smallest Last [Matula et al]:** v_1, \dots, v_n astfel încât v_i este vârful de grad minim din $G - v_n - \dots - v_{i+1}$
 - folosește cel mult 6 culori pentru grafuri planare

Distanța de Editare LEVENSHTEIN

Distanțe de editare

Subprobleme:

$c[i][j]$ = numărul minim de operații de inserare, ștergere, modificare pentru a transforma $x_1 \dots x_i$ în $y_1 \dots y_j$

Relații de recurență – corespund cazurilor:

- ▶ $x_i = y_j$: $(x_1 \dots x_{i-1} \Rightarrow y_1 \dots y_{j-1}) + \text{pastrăm } x_i$
- ▶ $(x_1 \dots x_{i-1} \Rightarrow y_1 \dots y_j) + \text{ștergem } x_i$
- ▶ $(x_1 \dots x_{i-1} \Rightarrow y_1 \dots y_{j-1}) + \text{modificăm } x_i \leftrightarrow y_j$
- ▶ $(x_1 \dots x_i \Rightarrow y_1 \dots y_{j-1}) + \text{inserăm } y_j$

$$c[i][j] = \begin{cases} c[i-1][j-1], & \text{dacă } x_i = y_j \\ 1 + \min \{c[i-1][j], c[i-1][j-1], c[i][j-1]\}, & \text{altfel} \end{cases}$$

ștergem x_i $x_i \leftrightarrow y_j$ inserăm y_j

Soluția $c[n][m]$

Ce valori din c știm direct:

- ▶ $c[0][0] = 0$ (ambele cuvinte sunt vide)
- ▶ pentru $i=0$ sau $j=0$ (unul dintre cuvinte este vid):
 - $x_1 \dots x_{i-1} \Rightarrow$ secvență vidă prin i stergeri succesive
 - $secvență vidă \Rightarrow y_1 \dots y_j$ prin j inserări succesive

$$c[0][0] = 0$$

$$c[i][0] = 1 + c[i-1][0] = i, \text{ pentru } i = 1, \dots, n$$

$$c[0][j] = 1 + c[0][j-1] = j, \text{ pentru } j = 1, \dots, m$$

Ordine de calcul a matricei: $i = 0, \dots, n; j = 0, \dots, m$

- ▶ Exemplu care \Rightarrow antet

		0	1	2	3	4	5
		a	n	t	e	t	
c:	0	0	1	2	3	4	5
	1 c	1	2	3	4	5	
	2 a	2	1	2	3	4	5
	3 r	3	2	2	3	4	5
	4 e	4	3	3	3	3	4

ștergem c
păstrăm a
modificăm r \leftrightarrow n
inserăm t, pastram e
inserăm t