

BFS – PAG 2

DFS – PAG 3

COMPONENTE TARE CONEXE(KOSARAJU) – PAG 5

SORTARE TOPOLOGICA – PAG 7

MUCHII CRITICE – PAG 8

PUNCTE CRITICE – PAG 10

COMPONENTE BICONEXE – PAG 11

GRAFURI BIPARTITE – PAG 11

APCM - PAG 12

ALGORITMUL LUI KRUSKAL – PAG 13

ALGORITMUL LUI PRIM – PAG 14

CLUSTERING – PAG 15

DIJKSTRA – PAG 16

ALGORITMUL LUI BELLMAN FORD – PAG 19

FLUX MAX – PAG 20

TAIETURA MINIMA – PAG 23

CAPACITATE REZIDUALA – PAG 24

Algoritmul FORD – FULKERSON / EDMONDS KARP – PAG 26

GRAF REZIDUAL – PAG 27

CUPLAJ MAXIM IN GRAF BIPARTIT – PAG 28

CONSTRUIREA UNUI GRAF ORIENTAT DIN SECVENTELE DE GRADE – PAG 30

S-T DRUMURI ARC DISJUNCTE – PAG 31

GRAF EULERIAN – PAG 32

COLORARI IN GRAFURI – PAG 33

GRAFURI PLANARE – PAG 34

BFS

1. se adaugă în coadă vârful de start (nevizitat) și se marchează ca fiind vizitat
2. cât timp mai sunt vârfuri în coadă
 - se scoate din coadă un vârf
 - se pun în coadă toți vecinii nevizitați ai acestuia și se marchează

BFS (s)

```
coada C = Ø
adauga(s, C)
viz[s] = 1; d[s] = 0
cat timp C ≠ Ø executa
    i = extrage(C);
    afiseaza(i);

    pentru j vecin al lui i (ij ∈ E)

        daca viz[j]==0 atunci
            adauga(j, C) → s devine gri
            viz[j] = 1
            tata[j] = i
            d[j] = d[i]+1

    → j devine gri

    i devine negru
    (s-a finalizat explorarea sa)
```

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

```
pentru fiecare x ∈ V executa
    daca viz[x] == 0 atunci
        BFS(x)
```

Toate componentele conexe – reluăm BFS din vârfuri nevizitate
 => **pădure BF**, cu arbori parțiali pentru fiecare componentă

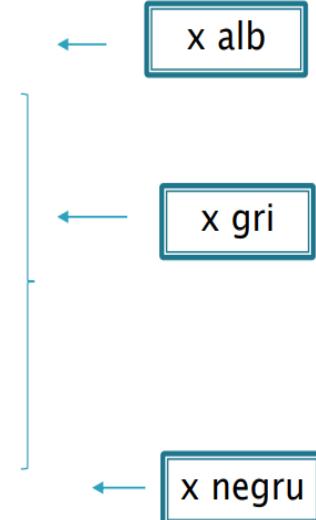
```
nr_componente = 0
pentru fiecare x ∈ V executa
    daca viz[x] == 0 atunci
        BFS(x)
        nr_componente += 1
```

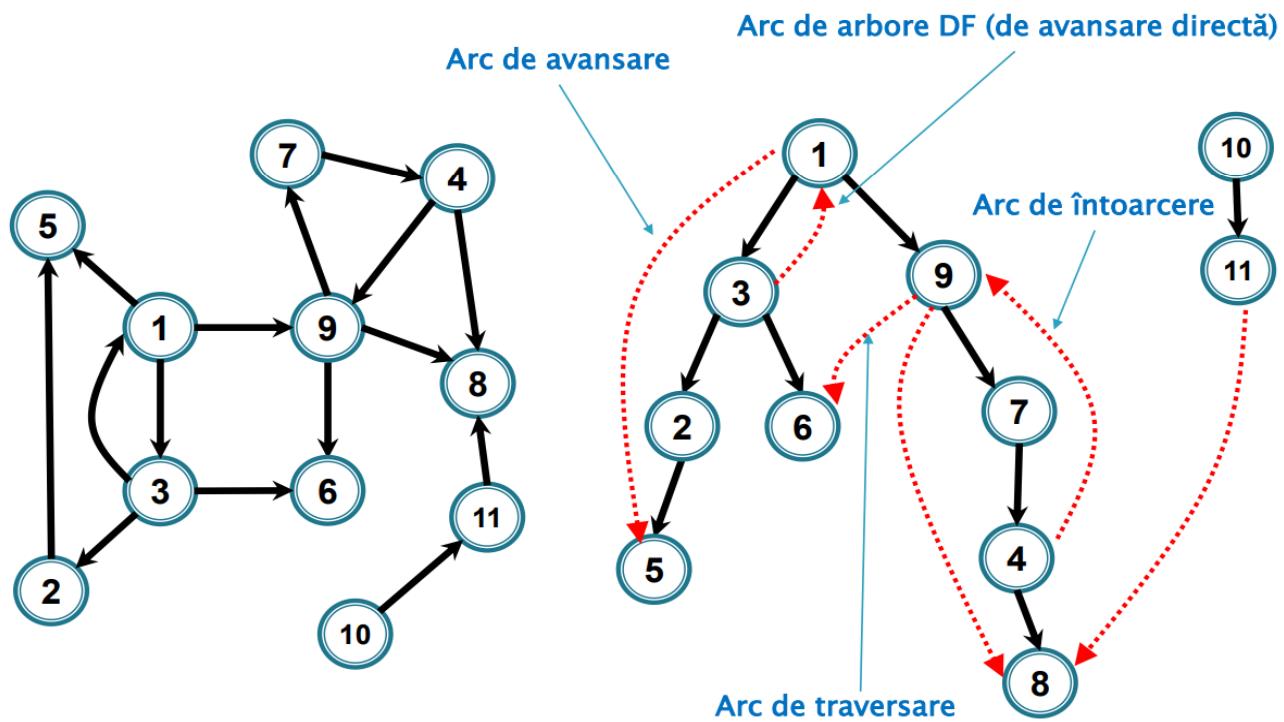
DFS

- Inițial: vârful de start s – devine vârf curent
- La un pas:
 - se trece la primul vecin nevizitat al vârfului curent, dacă există
 - altfel
 - se merge **înapoi** pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați (dacă nu mai există un astfel de vârf parcurgerea de oprește)
 - se trece la **primul** dintre aceștia și se reia procesul

DFS(x)

```
//incepe explorarea varfului x
viz[x] = 1
pentru fiecare xy ∈ E //y vecin al lui x
    daca viz[y]==0 atunci
        tata[y] = x
        d[y] = d[x]+1 //nivel, nu distanta
        DFS(y)
    //s-a finalizat explorarea varfului x
```





În raport cu pădurea DF muchiile/arcele se împart în 4 categorii:

- **de arbore DF (de avansare directă/ de arbore)**: din pădurea DF
(u, v cu v fiu al lui u (v a fost descoperit din u))
- **de întoarcere (înapoi)** :
(u, v cu v strămoș al lui u (și care nu este de arbore))
- **de avansare (de avansare înainte)** :
(u, v cu v descendant al lui u care nu este însă fiu al lui u)
- **de traversare (transversale)** :
toate celelalte muchii;
pot fi între vârfuri din același arbore din pădurea DF cu condiția
ca unul să nu fie strămoșul celuilalt sau pot uni vârfuri din arbori
diferiți

COMPONENTE TARE CONEXE(KOSARAJU)

Există drum între oricare 2 noduri.

- G este **tare conex** dacă între oricare două vârfuri există un drum
- O **componentă tare conexă** a lui G = subgraf induș al lui G tare conex, maximal

KOSARAJU:

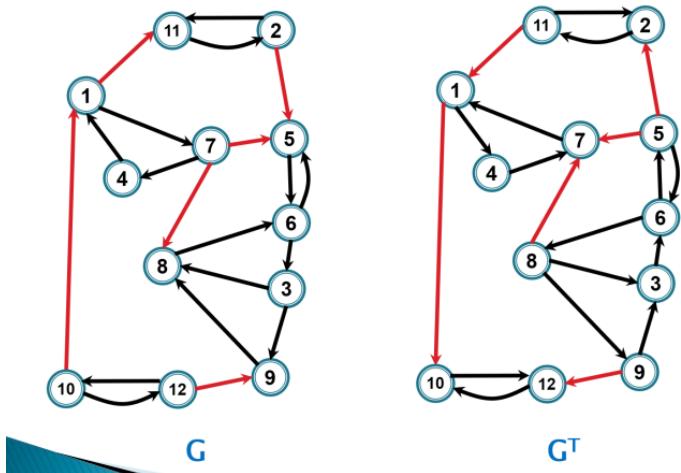
- › **Pasul 1. Parcurgem DFS graful G** +
introducem într-o stivă S fiecare varf la momentul la care este finalizat
(pentru a obține o ordonare descrescătoare a vîrfurilor după timpul de finalizare)
- › **Pasul 2. Parcurgem DFS graful G^T** considerând vâfurile în ordinea în care sunt extrase din S (descrescătoare după timpul de finalizare de la Pasul 1):

Componenta tare conexă a vârfului 1 =

mulțimea vârfurilor accesibile din 1 în G (vizitate în $\text{DFS}(1,G)$)

↪ intersectată cu

mulțimea vârfurilor accesibile din 1 în G^T (vizitate în $\text{DFS}(1,G^T)$)



$$\text{DFS}(1,G) \Rightarrow 1,7,4,5,6,3,9,8,11,2$$

$$\text{DFS}(1,G^T) \Rightarrow 1,4,7,10,12$$

Intersecția vârfurilor vizitate =>

$$\text{Componenta}(1) = 1,4,7$$

- ▶ **Pasul 1. Parcurgem DFS graful G^T** +
 - introducem într-o stivă S fiecare varf la momentul la care este finalizat**
 - (pentru a obține o ordonare descrescătoare a varfurilor după timpul de finalizare)

```

stack S

DFS(G, i)
  viz[i] = 1
  pentru ij ∈ E(G)
    daca viz[j]==0 atunci
      DFS(j)
    push(S, i) //i este finalizat

pentru x ∈ V execută
  daca viz[x]==0 atunci
    DFS(G, x)
  
```

- ▶ **Pasul 2. Parcurgem DFS graful G^T considerând vârfurile în ordinea în care sunt extrase din S** (descrescătoare după timpul de finalizare de la Pasul 1):

```

marcăm toate vârfurile ca fiind nevizitate
cat timp S este nevida
  x = pop(S)
  daca x este nevizitat atunci
    DFS( $G^T$  , x)
    afiseaza componenta tare conexă (formată cu
    vârfurile vizitate în  $DFS(G^T, x)$ )
  
```

SORTARE TOPOLOGICA

- ▶ Sortare topologică a lui G =
ordonare a vârfurilor astfel încât dacă $uv \in E$ atunci u se află înaintea lui v în ordonare
- ▶ Propoziție. Dacă G este aciclic atunci G are o sortare topologică

▶ Implementare – similar BF

- Pornim cu toate vârfurile cu grad intern 0 și le adăugăm într-o coadă
- Repetăm:
 - extragem un vârf din coadă
 - îl eliminăm din graf (= scădem gradele interne ale vecinilor, nu îl eliminăm din reprezentare)
 - adăugăm în coadă vecinii al căror grad intern devine 0

Sortare topologică a lui G = E atunci $u \in$ ordonare a vârfurilor astfel încât dacă $uv \in E$ se află înaintea lui v în ordonare.

```
Stack S

DFS(i)
    viz[i] = 1
    pentru ij ∈ E
        daca viz[j]==0 atunci
            DFS(j)
    //i este finalizat
    push(S, i)

pentru i ∈ V execută
    daca viz[i]==0 atunci
        DFS(i)

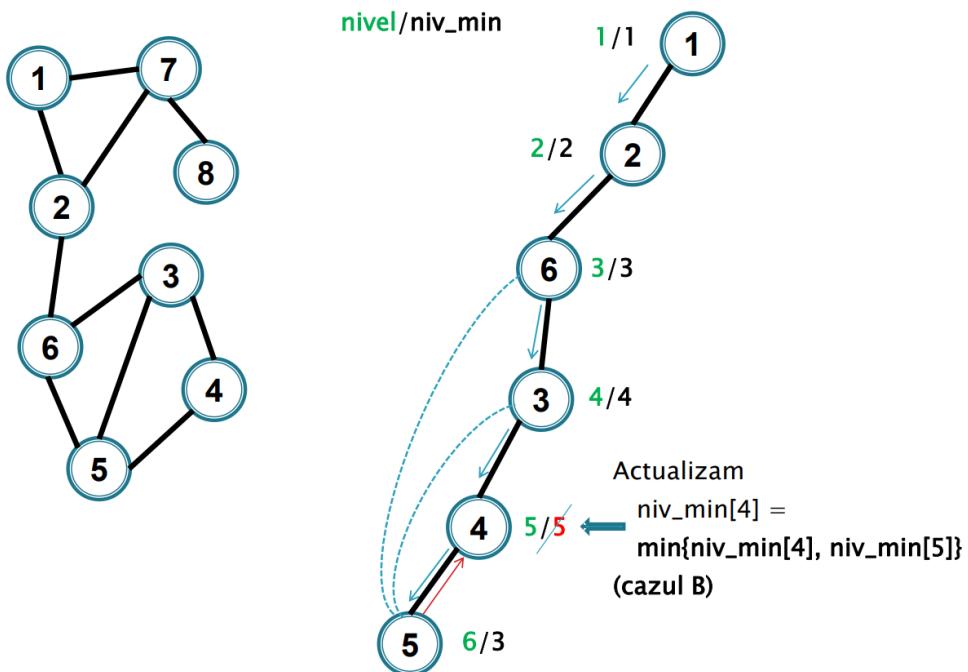
cat timp S este nevida execută
    u = pop(S)
    adauga u în sortare
```

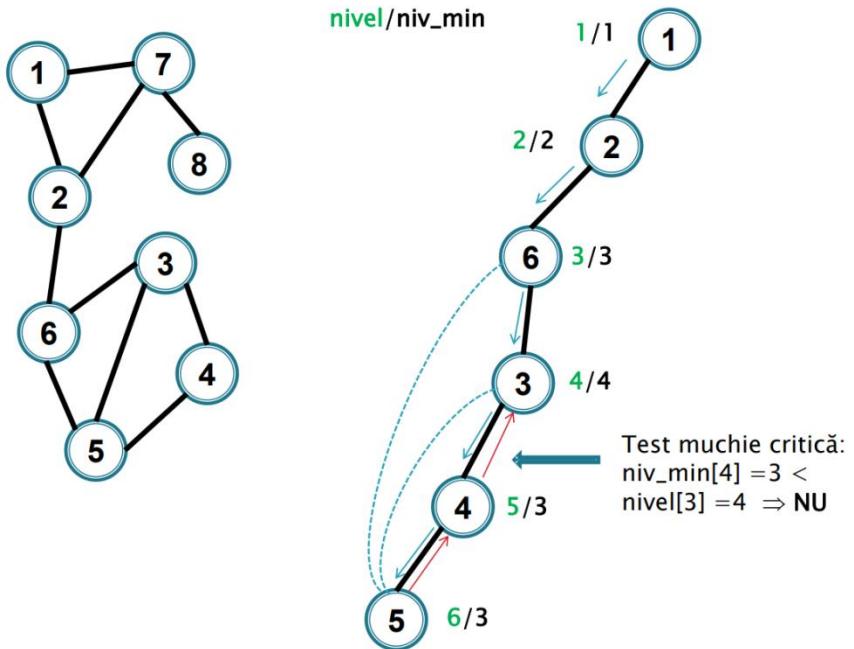
MUCHII CRITICE

- › G – graf neorientat
- › $e \in E(G)$ critică (punkte, muchie de articulație/ *bridge*) = prin eliminarea ei crește numărul de componente conexe ale grafului
- › O muchie este critică \Leftrightarrow nu este conținută într-un ciclu

Memorăm pentru fiecare vârf i :

`niv_min[i]` = nivelul minim al unui vârf care este extremitatea unei muchii de întoarcere din i sau dintr-un descendent al lui i
 = nivelul minim la care se închide un ciclu elementar care conține vârful i printr-o muchie de întoarcere





```

void df(int i){
    viz[i] = 1;
    niv_min[i] = nível[i];
    for(j vecin al lui i)
        if(viz[j]==0){ //ij muchie de avansare
            nível[j] = nível[i]+1;
            df(j);

            //actualizare niv_min[i]- formula B
            niv_min[i] = min{niv_min[i], niv_min[j] }

            //test ij este muchie critica
            if (niv_min[j]>nível[i]) scrie muchia ij
        }
        else
            if(nível[j]<nível[i]-1) //ij muchie de intoarcere
                //actualizare niv_min[i]- formula A
                niv_min[i] = min{niv_min[i], nível[j] }
}

```

PUNCTE CRITICE

- Un vârf v este punct critic \Leftrightarrow

există două vârfuri $x, y \neq v$ astfel

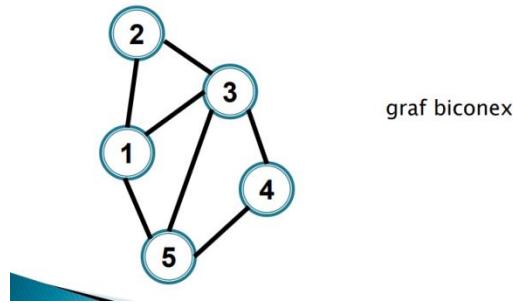
încât v aparține oricărui x, y -lanț

```
void df(int i){  
    viz[i] = 1;  
    niv_min[i] = nivel[i];  
    for(j vecin al lui i)  
        if(viz[j]==0){ //ij muchie de avansare  
            nivel[j] = nivel[i]+1;  
            adauga(S,ij)  
            df(j);  
            niv_min[i] = min{niv_min[i], niv_min[j] }  
            if (niv_min[j]>= nivel[i])  
                elimina din S toate muchiile pana la ij  
        }  
    else  
        if(nivel[j]<nivel[i]-1){ //ij muchie de intoarcere  
            //actualizare niv_min[i]- formula A  
            niv_min[i] = min{niv_min[i], nivel[j]}  
            adauga(S,ij)  
        }  
}
```

COMPONENTE BICONEXE

Componente biconexe

- ▶ G – graf neorientat
- ▶ $G = (V, M)$ biconex = nu are puncte critice (de articulație).
- ▶ Componentă biconexă (bloc) a lui G = subgraf biconex maximal.



▶ Observații

Într-un graf neorientat:

- Componentele biconexe sunt muchie-disjuncte, dar nu neapărat vârf-disjuncte (pot avea în comun un vârf care era punct critic în graful inițial)
- Orice vârf aparține unei componente biconexe
- Orice muchie aparține unei componente biconexe

GRAFURI BIPARTITE

Un arbore este graf bipartite.

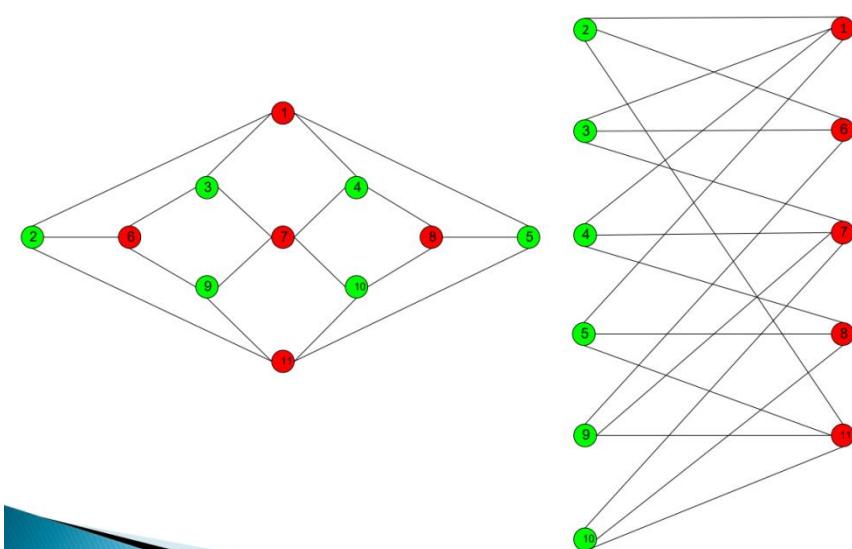
- ▶ Un graf neorientat $G = (V, E)$ se numește **bipartit** \Leftrightarrow există o partiție a lui V în două submulțimi V_1, V_2 (**bipartiție**):

$$V = V_1 \cup V_2$$

$$V_1 \cap V_2 = \emptyset$$

astfel încât orice muchie $e \in E$ are o extremitate în V_1 și cealaltă în V_2 :

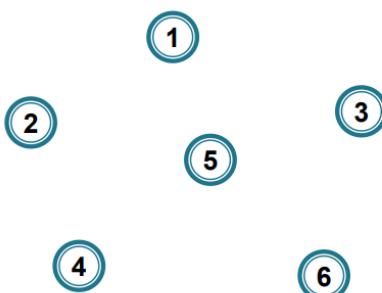
$$|e \cap V_1| = |e \cap V_2| = 1$$



ARBORI PARTIALI DE COST MINIM

Kruskal

- **Inițial:** cele n vârfuri sunt izolate, fiecare formând o componentă conexă



- Se încearcă unirea acestor componente prin muchii de cost minim

Prim

- **Inițial:** se pornește de la un vârf de start

1

- Se adăugă pe rând câte un vârf la arborele deja construit, folosind muchii de cost minim

ALGORITMUL LUI KRUSKAL

- La un pas este selectată o muchie de cost minim din G care nu formează cicluri cu muchiile deja selectate (care unește două componente conexe din graful deja construit)

Pentru a selecta ușor o muchie de cost minim cu proprietatea dorită ordonăm crescător muchiile după cost și considerăm muchiile în această ordine

```

sorteaza(E)
for (v=1;v<=n;v++)
    Initializare(v) ;
nrmse1=0
for(uv ∈ E)
    if(Reprez(u) !=Reprez(v) )
    {
        E(T) = E(T) ∪ {uv};
        Reuneste(u,v);
        nrmse1=nrmse1+1;
        if(nrmse1==n-1)
            STOP; //break;
    }
}

```

Reprez(u) Optimizare – compresie de cale

– tatăl vârfurilor de pe lanțul de la u la rădăcină se va seta ca fiind rădăcina

(vârfurile de pe acest lanț, parcurs pentru a găsi reprezentantul lui u , vor deveni fii ai rădăcinii, pentru că reprezentantul lor să fie găsit mai ușor în căutările ulterioare)

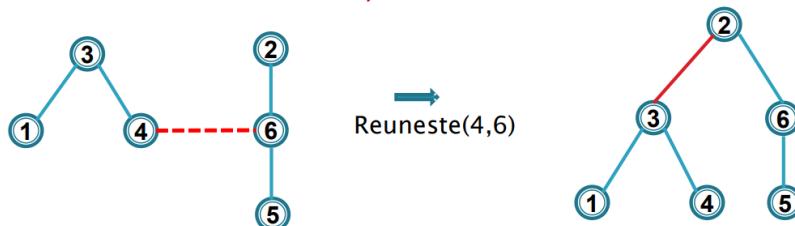
```

int Reprez(int u){
    if (tata[u]==0)
        return u;
    tata[u]=Reprez(tata[u]);
    return tata[u];
}

```

- Reuniunea se va face în funcție de înălțimea/dimensiunea arborilor (reuniune ponderată), pentru a obține arbori de înălțime mică

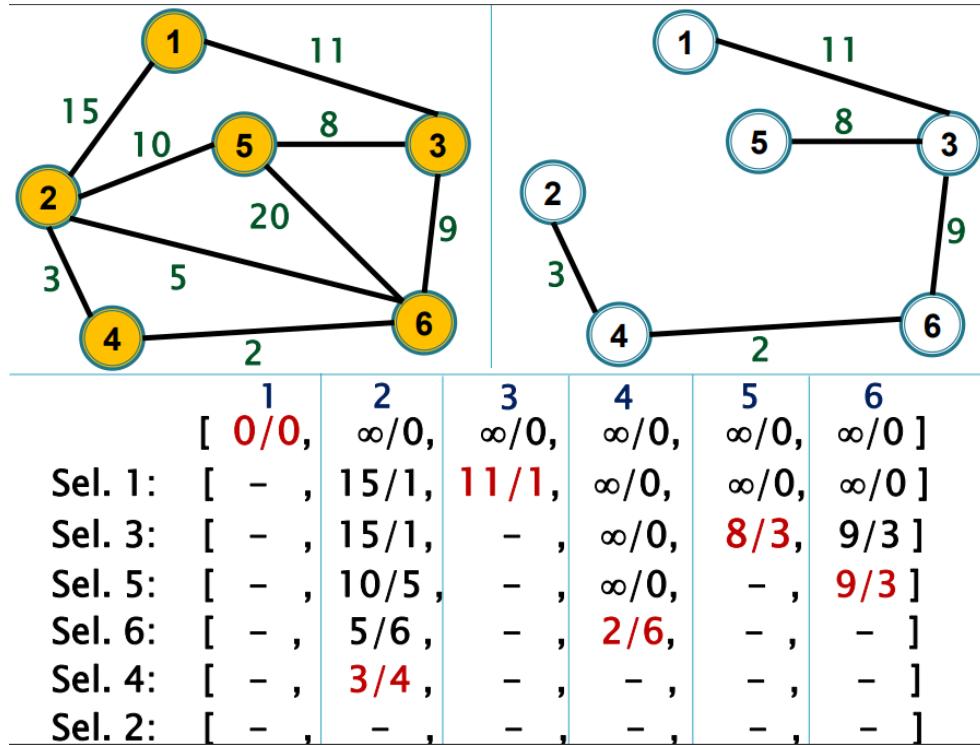
⇒ arbori de înălțime logaritmică



- arborele cu înălțimea mai mică devine subarbore al rădăcinii celuilalt arbore

ALGORITMUL LUI PRIM

- › Se pornește de la un vârf (care formează arborele inițial)
- › La un pas este selectată o muchie de cost minim de la un vârf deja adăugat la arbore la unul neadăugat



- s – vârful de start
- initializează Q cu V
- pentru fiecare $u \in V$ executa
 $d[u] = \infty$; $tata[u] = 0$
 $d[s] = 0$
- cat timp $Q \neq \emptyset$ executa
 extractează un vârf $u \in Q$ cu eticheta $d[u]$ minimă
 pentru fiecare $uv \in E$ executa
 daca $v \in Q$ si $w(u,v) < d[v]$ atunci
 $d[v] = w(u,v)$
 $tata[v] = u$

CLUSTERING

Distanțe de editare – numărul minim de operații (inserări, modificări, ștergeri etc) de caractere necesar pentru transforma prima secvență în cea de a doua

Distanța de editare Levenshtein – sunt permise operații de inserare, modificare și stergere

Exemplu: Distanța de la **care** la **antet** este 4

care → are → ane → ante → antet
 stergem c modificăm r <-> n inseram t inseram t

- ▶ Un **k-clustering** a lui **S** = o **partiționare** a lui **S** în **k** submulțimi nevide (numite **clase** sau **clustere**)

$$\mathcal{C} = (C_1, \dots, C_k)$$

- ▶ **Gradul de separare** a lui \mathcal{C}

= distanța minimă dintre două obiecte aflate în clase diferite
 = distanța minimă dintre două clase ale lui \mathcal{C}

Idee

- Inițial fiecare obiect (cuvânt) formează o clasă
- La un pas determinăm **cele mai asemănătoare (apropiate) două obiecte** aflate în clase diferite (cu distanța cea mai mică între ele) și unim clasele lor
- Repetăm până obținem **k** clase $\Rightarrow n - k$ pași

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

↑ ↑ ↑
 stergem x_i $x_i \leftrightarrow y_j$ inserăm y_j

► Exemplu care => antet

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

Soluția: $c[4][5] = 4$

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

ștergem c

păstrăm a

modificăm r ↔ n

inserăm t, pastram e

inserăm t

DIJKSTRA – cost > 0

► La un pas

- este selectat un vîrf u (neselectat) care “pare” cel mai apropiat de s ⇔ are eticheta d minimă
- Se actualizează etichetele $d[v]$ ale vecinilor lui u – extinzând drumul minim deja găsit de la s la u cu arcul uv
- Relaxarea unui arc (u, v) = a verifica dacă $d[v]$ poate fi îmbunătățit extinzând drumul minim găsit de la s la u cu arcul uv

► Algoritmi – $G=(V, E)$ graf orientat

G – neponderat

Parcursere lățime BF

BF(s)

```
coada C ← Ø;
adauga(s, C)
```

```
pentru fiecare u∈V
d[u]=∞; tata[u]=viz[u]=0
```

```
viz[s]← 1; d[s] ← 0
```

```
cat timp C ≠ Ø
u ← extrage(C);
```

```
pentru fiecare u∈E
daca viz[v]=0
d[v] ← d[u]+1
tata[v] ← u
adauga(v, C)
viz[v] ← 1
```

scrie d, tata

$O(n+m)$

G – ponderat, ponderi >0

Algoritmul lui Dijkstra

Dijkstra(s)

```
Q ← V
{se putea incepe doar cu Q ← {s}
+vector viz; v∈Q ⇔ v nevizitat}
```

```
pentru fiecare u∈V
d[u] = ∞; tata[u]=0
```

```
d[s] = 0
```

```
cat timp Q ≠ Ø
u = extrage(Q) vîrf cu eticheta
d minimă
```

```
pentru fiecare u∈E
daca v∈Q si d[u]+w(u,v)< d[v]
d[v] = d[u]+w(u,v)
tata[v] = u
repara(v,Q)
```

scrie d, tata

$O(m \ log(n)) / O(n^2)$

G – ponderat fără circuite

DAGS(s)

```
SortTop ← sortare_topologica(G)
```

```
pentru fiecare u∈V
d[u] = ∞; tata[u]=0
```

```
d[s] = 0
```

pentru fiecare u ∈ SortTop

```
pentru fiecare u∈E
daca d[u]+w(u,v)< d[v]
d[v] = d[u]+w(u,v)
tata[v] = u
```

scrie d, tata

$O(n+m)$

Drumuri minime din s \Rightarrow arbore de drumuri minime (distanțe) din s

\neq arbore parțial de cost minim – minimizează costul total

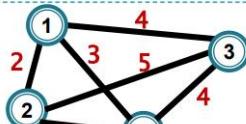
G-(ne)orientat ponderat, ponderi >0

Drumuri minime din s

Algoritmul lui Dijkstra

```
Dijkstra(s)
(min-heap) Q ← V
pentru fiecare u∈V
d[u] = ∞; tata[u]=0
d[s] = 0
cat timp Q ≠ Ø
u = extrage(Q) vîrf cu eticheta
d minimă
pentru fiecare u∈E
daca v∈Q si d[u]+w(u,v)< d[v]
d[v] = d[u]+w(u,v)
tata[v] = u
repara(v,Q)
```

scrie d, tata



arbore al drumurilor minime față de 1

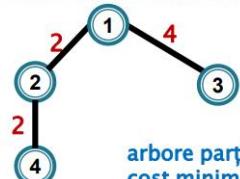
G- neorientat ponderat ponderi reale

Arbore parțial de cost minim

Algoritmul lui Prim

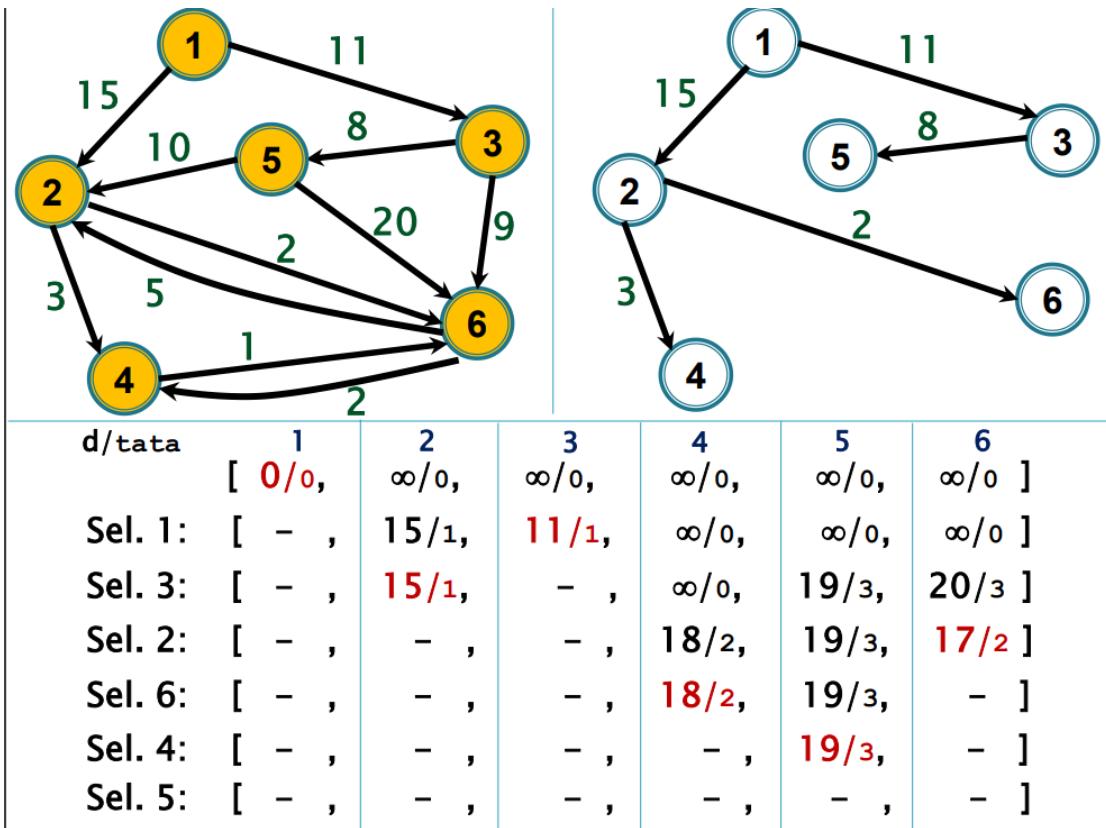
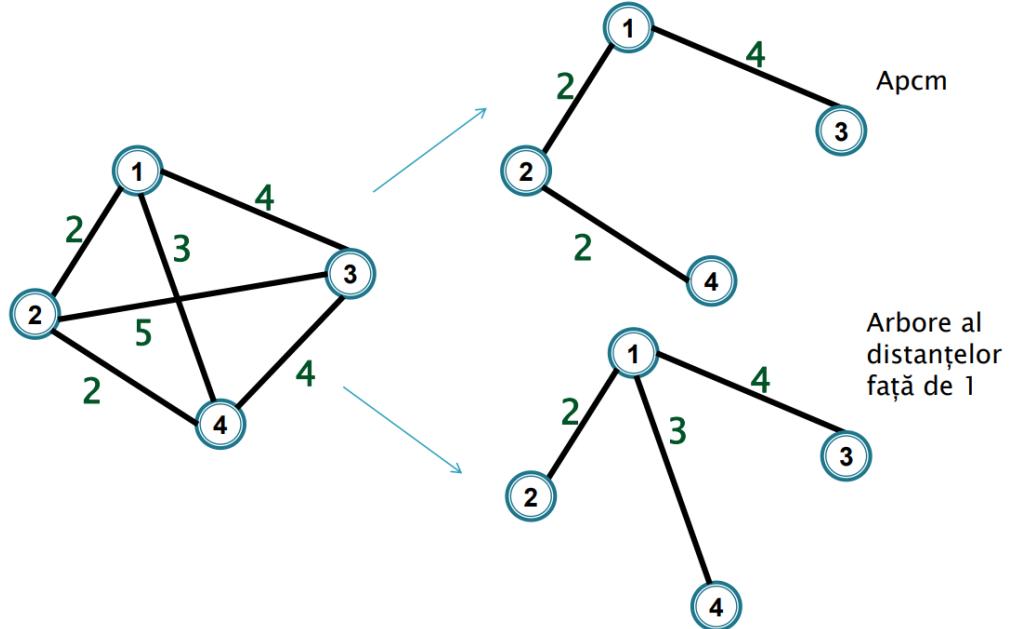
```
Prim(s)
(min-heap) Q ← V
pentru fiecare u∈V
d[u] = ∞; tata[u]=0
d[s] = 0
cat timp Q ≠ Ø
u = extrage(Q) vîrf cu
eticheta d minimă
pentru fiecare u∈E
daca v∈Q si w(u,v)< d[v]
d[v] = w(u,v)
tata[v] = u
repara(v,Q)
```

scrie (u, tata[u]), pentru u≠s



arbore parțial de cost minim

Un arbore parțial de cost minim nu este neapărat un arbore de distanțe minime



ALGORITMUL LUI BELLMAN FORD

FACE DIJKSTRA DE N-1 ORI SI MERGE SI PT COSTURI NEGATIVE.

NU MERGE PENTRU CIRCUITE DE COST NEGATIV. (daca suma circuitului este negativ)

► Idee: La un pas relaxăm **toate arcele**

(nu relaxăm arcele dintr-un vîrf selectat u, ci **din toate vîrfurile**)

► **n-1 etape** – după k etape $d[u] \leq$ costul minim al unui drum de la s la u cu cel mult k arce (**programare dinamică**)

=> după k etape sunt corect calculate etichetele $d[u]$ pentru acele vîrfuri u pentru care **există un s-u drum minim cu cel mult k arce**

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

    pentru  $k = 1, n-1$  execută
        pentru fiecare  $u \in E$  execută
            dacă  $d[u] + w(u, v) < d[v]$  atunci
                 $d[v] = d[u] + w(u, v)$ 
                 $tata[v] = u$ 

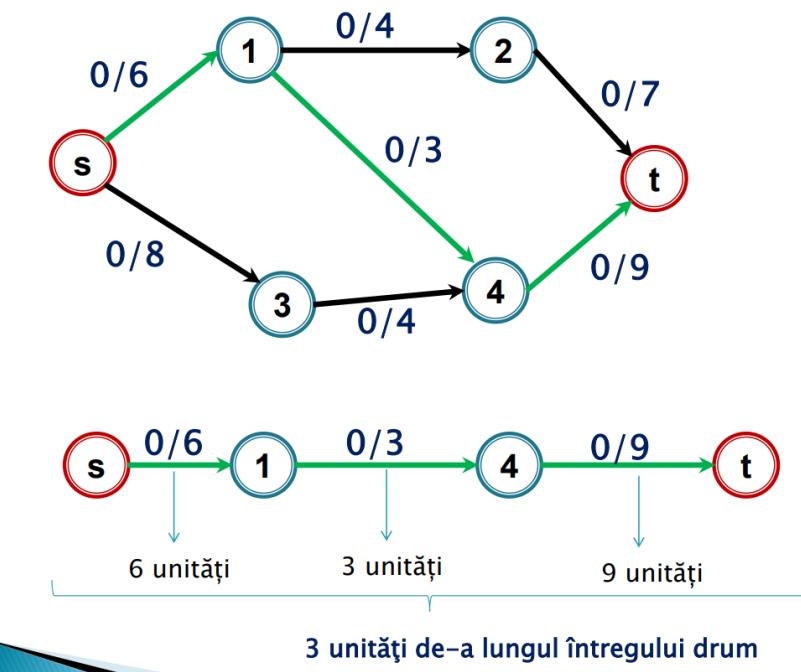
        pentru fiecare  $u \in E$  execută
            dacă  $d[u] + w(u, v) < d[v]$  atunci
                 $d[v] = d[u] + w(u, v)$ 
                 $tata[v] = u$ 
            STOP, există circuit negativ ACCESIBIL DIN s
```

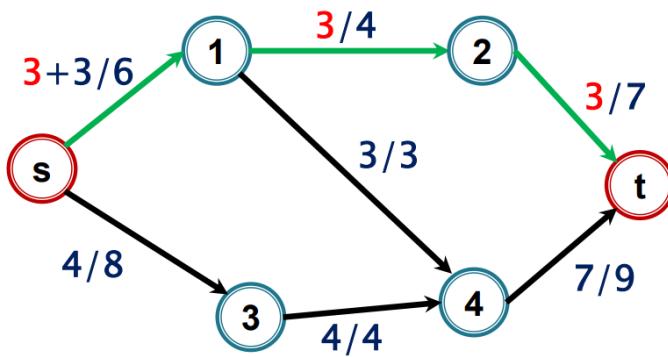
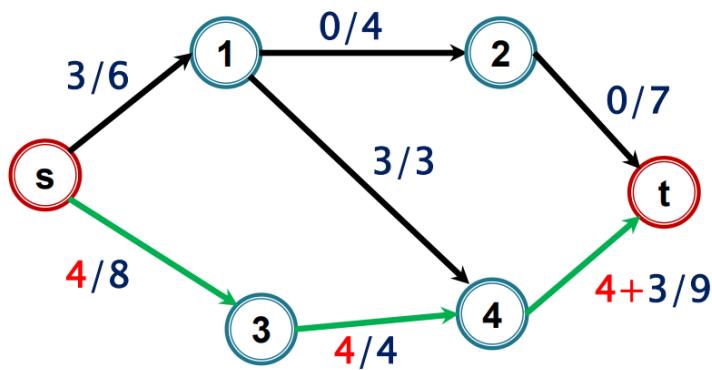
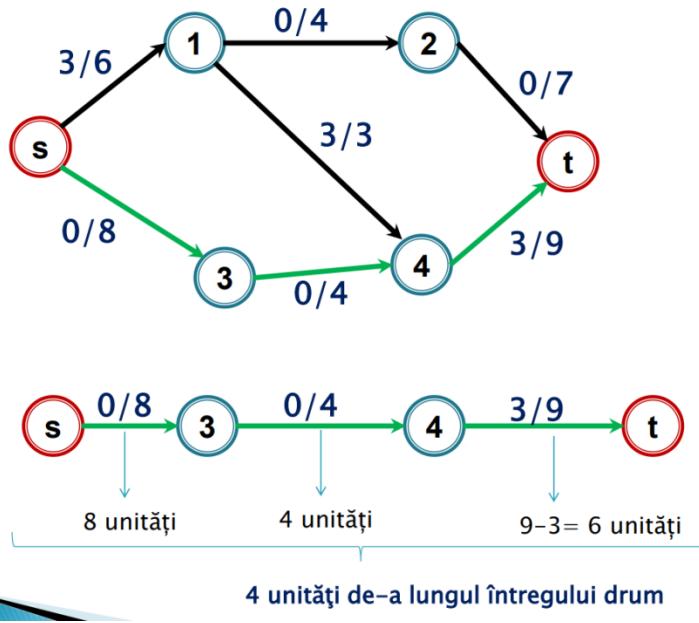
FLUX MAX

Încercăm să trimitem marfă (flux) de la vârful sursă s la destinația t

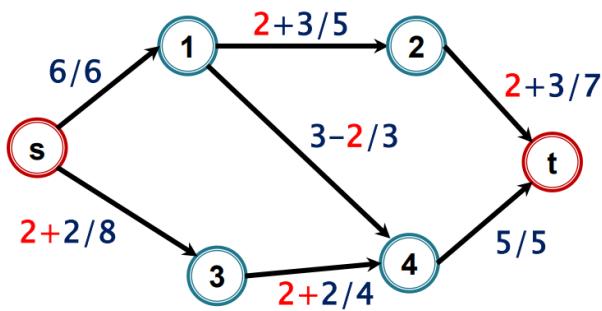
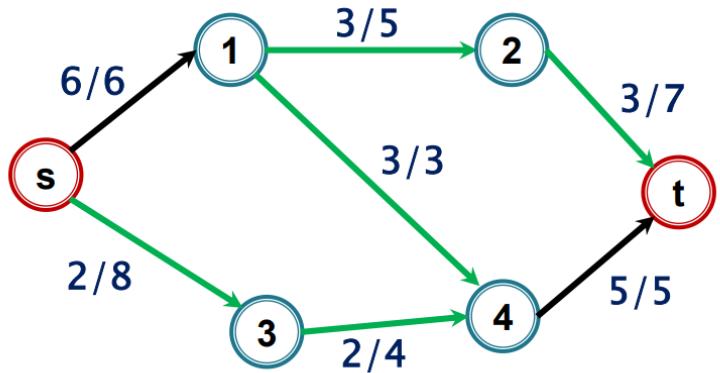
Condiția de conservare a fluxului devine:

$$f^-(v) = f^+(v), \forall v \in I$$





ALT EXEMPLU

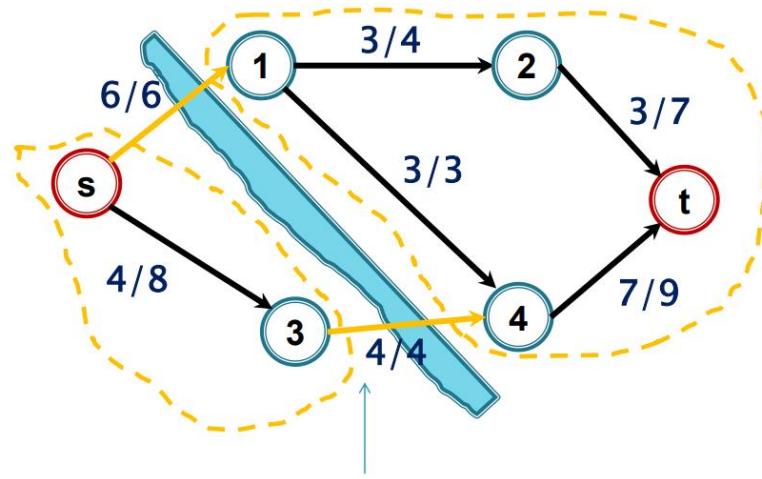


TAIETURA MINIMA

- ▶ O s-t tăietură $K = (X, Y)$ în rețea este o (bi)partiție (X, Y) a mulțimii vârfurilor V astfel încât $s \in X$ și $t \in Y$

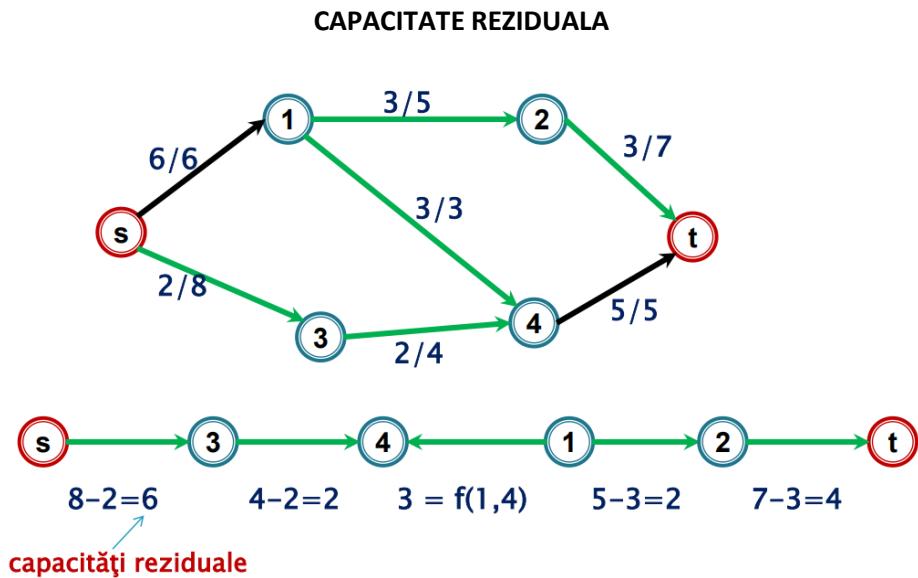
Capacitatea tăieturii = suma capacitatilor arcelor din tăietura

O tăietură s.n. o tăietură minima dacă capacitatea tăieturii este minima și = flux max.



- singurele arce (“poduri”) care trec din regiunea lui s în cea a lui t nu mai pot fi folosite pentru a trimite flux (au fluxul = capacitatea) \Rightarrow fluxul este maxim
- **s-t tăietură**

Determinarea unui flux maxim \Rightarrow determinarea unei tăieturi minime



- ▶ 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

c- capacitatea muchiei

f – fluxul muchiei in acel moment

► Capacitatea reziduală a lanțului P



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

► Un s-t lanț P se numește

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

► 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 $f_P : E \rightarrow \mathbb{N}$,

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

Algoritmul FORD – FULKERSON / EDMONDS KARP

Algoritmul EDMONDS-KARP = Ford-Fulkerson

în care lanțul P ales la un pas are lungime minimă

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

sunt considerate în parcurgere doar arce pe care se poate modifica fluxul, adică având capacitate reziduală pozitivă

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

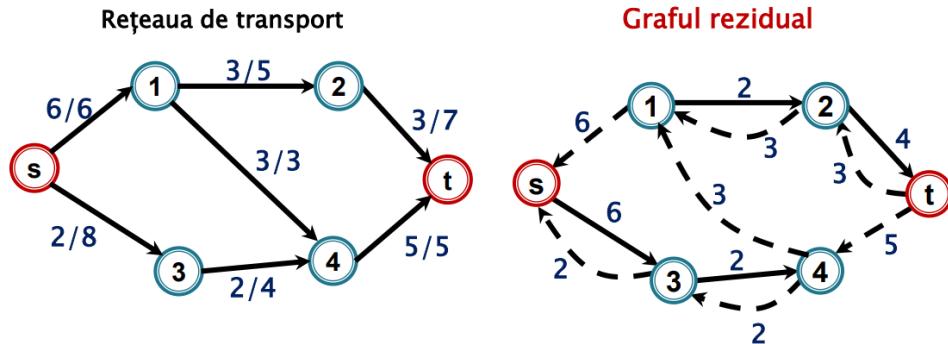
tăietura determinată de vârfurile accesibile din s la ultimul pas prin lanțuri f -nesaturate este tăietură minimă (= din vârfurile vizitate la ultimul pas)

GRAF REZIDUAL

a determină un s-t lanț nesaturat folosind BF în G

\Leftrightarrow

a determină un s-t drum folosind BF în graful rezidual G_f



(Multi)graful rezidual G_f asociat fluxului f = graf orientat ponderat, cu funcția de pondere (de capacitate) c_f construit astfel:

- $V(G_f) = V(G)$
- Pentru fiecare arc $e = uv$ cu $c(e) - f(e) > 0$, adăugăm arcul e la G_f cu ponderea $c_f(e) = c(e) - f(e)$
- Pentru fiecare arc $e = uv$ cu $f(e) > 0$, adăugăm la G_f arcul $e^{-1} = vu$ (inversul arcului e) cu ponderea asociată $c_f(e^{-1}) = f(e)$.

IMPLEMENTARE EDMINDS KARP CU GRAF REZIDUAL

1. Setăm $f := 0$ fluxul vid ($f(e) = 0, \forall e \in E$)
2. Construim $G_f :=$ graful rezidual pentru f
3. Cât timp există un s-t drum în G_f
 - determină P un s-t drum minim în G_f folosind BF (pentru arcele cu $c_f(e) > 0$)
 - actualizează G_f

pentru $e \in E(P) \subseteq E(G_f)$

$$c_f(e) \leftarrow c_f(e) - c_f P;$$

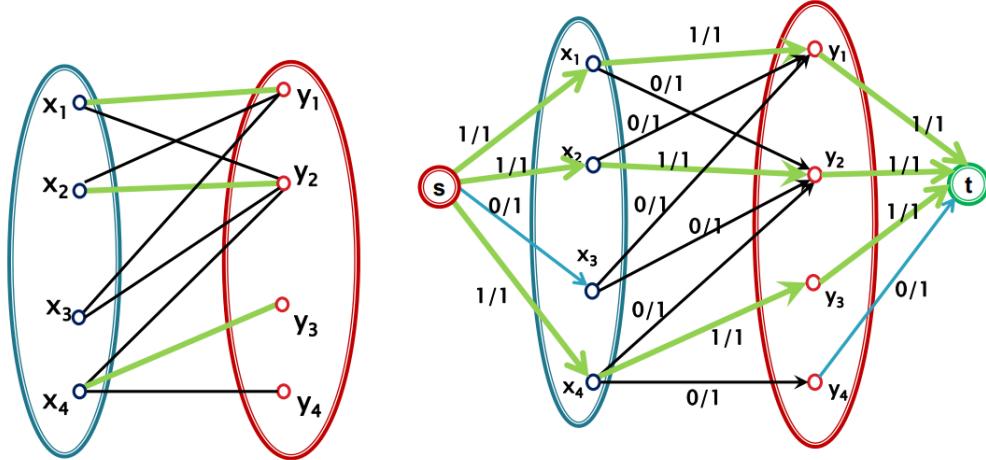
$$c_f(e^{-1}) \leftarrow c_f(e^{-1}) + c_f P$$
4. returnează f

CUPLAJ MAXIM IN GRAF BIPARTIT

- ▶ M s.n **cuplaj** dacă orice două muchii din M sunt neadiacente
 - ▶ $G = (V, E)$ graf neorientat s.n. **bipartit** \Leftrightarrow există o partiție a lui V în două submulțimi V_1, V_2 (**bipartiție**):
$$V = V_1 \cup V_2$$
$$V_1 \cap V_2 = \emptyset$$
astfel încât orice muchie $e \in E$ are o extremitate în V_1 și cealaltă în V_2
 - ▶ Notăm $G = (V_1 \cup V_2, E)$

- ▶ **Algoritm de determinare a unui cuplaj maxim într-un graf bipartit**
 - Reducem problema determinării unui cuplaj maxim într-un cuplaj bipartit G la determinarea unui flux maxim într-o rețea de transport asociată lui G
 - Construim rețeaua de transport N_G asociată lui G astfel:

- ▶ Cuplaj M în $G \Leftrightarrow$ flux f în rețea
cu $|M| = \text{val}(f)$



A determină un **cuplaj maxim** într-un graf bipartit \Leftrightarrow
a determină un **flux maxim** în rețeaua asociată

Algoritm de determinare a unui cuplaj maxim în $G=(X \cup Y, E)$:

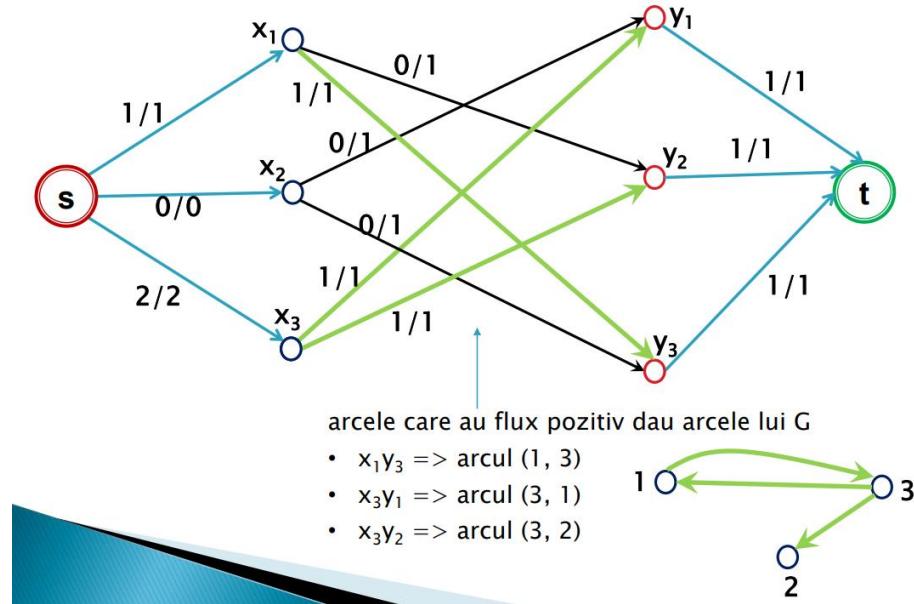
1. Construim N rețeaua de transport asociată
2. Determinăm f^* flux maxim în N
3. Considerăm $M = \{xy \mid f^*(xy) = 1, x \in X, y \in Y, xy \in N\}$

(pentru fiecare arc cu flux nenul xy din N care nu este incident în s sau t , muchia xy corespunzătoare din G se adaugă la M)

4. return M

CONSTRUIREA UNUI GRAF ORIENTAT DIN SECVENTELE DE GRADE

- $s_0^+ = \{1, 0, 2\}$
- $s_0^- = \{1, 1, 1\}$



Merge și reciproc

Algoritm de determinare a unui graf orientat G cu

$$s^+(G) = s_0^+ \text{ și } s^-(G) = s_0^-$$

1. Construim N rețeaua de transport asociată

2. Determinăm f^* flux maxim în N

3. Dacă $\text{val}(f^*) < d_1^+ + \dots + d_n^+$ atunci

Nu există G . STOP

4. $V(G) = \{1, \dots, n\}$

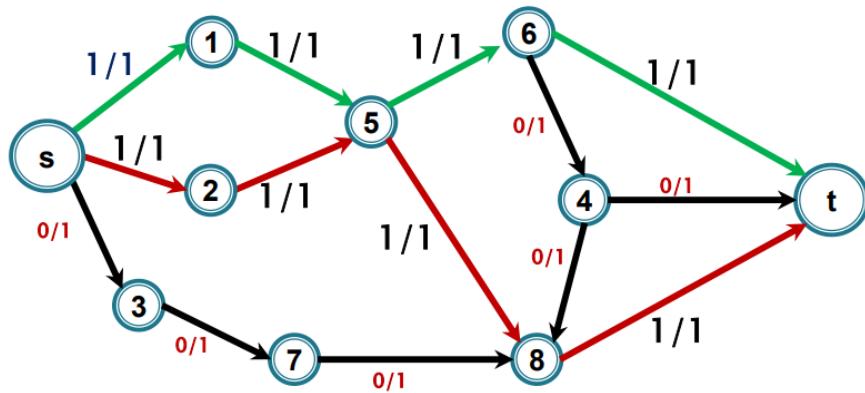
$$E(G) = \{(i, j) \mid x_i y_j \in N \text{ cu } f^*(x_i y_j) = 1\}$$

S-T DRUMURI ARC DISJUNCTE

Să se determine numărul maxim k de s-t drumuri elementare arc-disjuncte ($+ k$ astfel de drumuri).

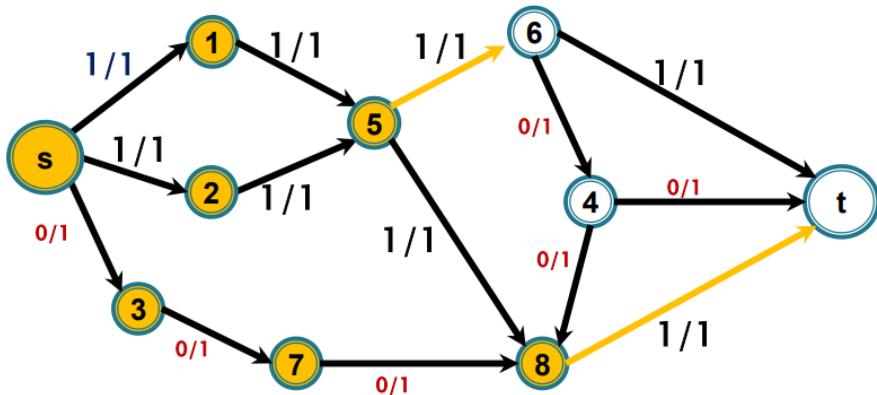
Două drumuri P_1, P_2 s.n. **arc-disjuncte** dacă $E(P_1) \cap E(P_2) = \emptyset$

- Asociem fiecărui arc capacitatea 1
- Fluxul maxim: $f(e) \in \{0, 1\}$
- Un drum de la s la t = **traseul parcurs de o unitate de flux de la s la t**
- Numărul de s-t drumuri arc-disjuncte= valoarea fluxului maxim



S-t tajetura

Mulțimea vârfurilor accesibile din s prin lanțuri f-nesaturate



- ▶ **Muchie-conectivitatea lui G** $k'(G)$ = cardinalul minim al unei mulțimi de muchii $F \subseteq E$ cu proprietatea că
 $G - F$ nu mai este conex
- ▶ Dacă $k'(G) \geq t$, G se numește **t-muchie conex**

GRAF EULERIAN

Ciclu eulerian – traseu închis care trece o singură dată prin toate muchiile

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

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

G este bipartit \Leftrightarrow toate ciclurile elementare
din G sunt pare

Fie G graf neorientat

► **Ciclu eulerian** al lui G = ciclu C în G cu

$$E(C) = E(G)$$

► **G eulerian** = conține un **ciclu** eulerian

► **Lanț eulerian** al lui G = lanț simplu P în G cu

$$E(P) = E(G)$$

Algoritmul lui Hierholzer

► **Pasul 0** – verificare condiții (conex+vf. izolate, grade pare)

► **Pasul 1:**

- alege $v \in V$ arbitrar
- construiește C un ciclu în G care începe cu v (cu algoritmul din Lema)

► cât timp $|E(C)| < |E(G)|$ execută

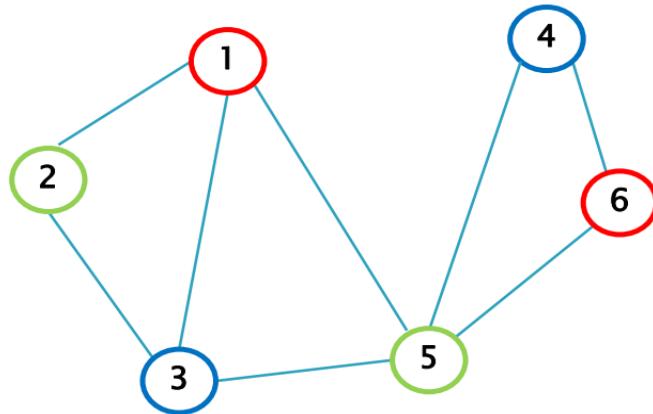
- selectează $v \in V(C)$ cu $d_{G-E(C)}(v) > 0$ (în care sunt incidente muchii care nu aparțin lui C)
- construiește C' un ciclu în $G - E(C)$ care începe cu v
- $C =$ ciclul obținut prin fuziunea ciclurilor C și C' în v

► scrie C

COLORARI IN GRAFURI

- ▶ $G = (V, E)$ graf neorientat
 - $c : V \rightarrow \{1, 2, \dots, p\}$ s.n **p-colorare** a lui G
 - $c : V \rightarrow \{1, 2, \dots, p\}$ cu $c(x) \neq c(y) \forall xy \in E$ s.n **p-colorare proprie** a lui G
 - G s.n **p-colorabil** dacă admite o p-colorare proprie

Valoarea p minimă pentru care G este p-colorabil se numește **numărul cromatic**

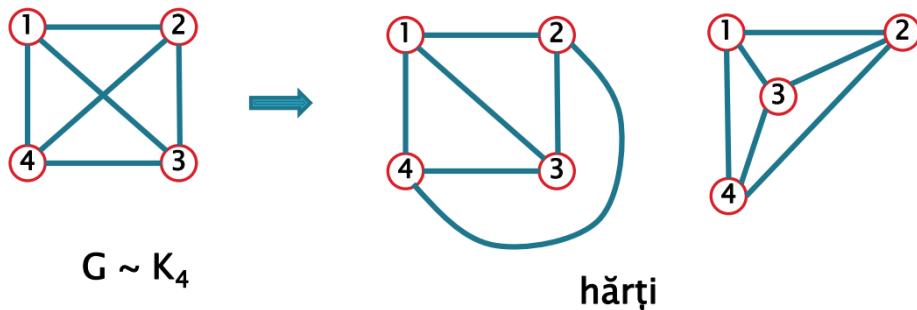


3-colorabil, dar nu și 2-colorabil (!)
=> are numărul cromatic 3

- ▶ $G = (V, E)$ bipartit \Leftrightarrow
există o 2-colorare proprie a vârfurilor (**bicolorare**)

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

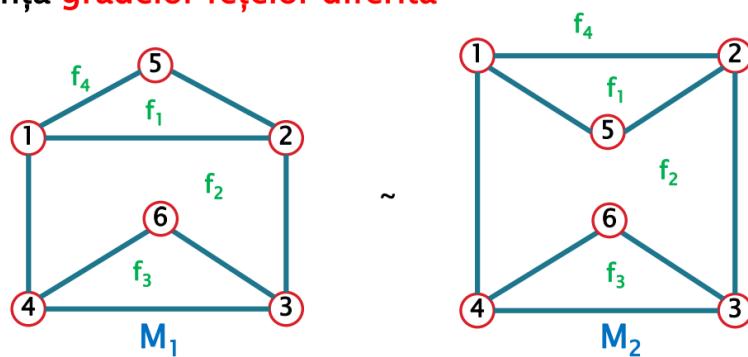


M induce o împărțire a planului într-o mulțime F de părți convexe numite **fețe**

Una dintre acestea este **fața infinită (exterioară)**

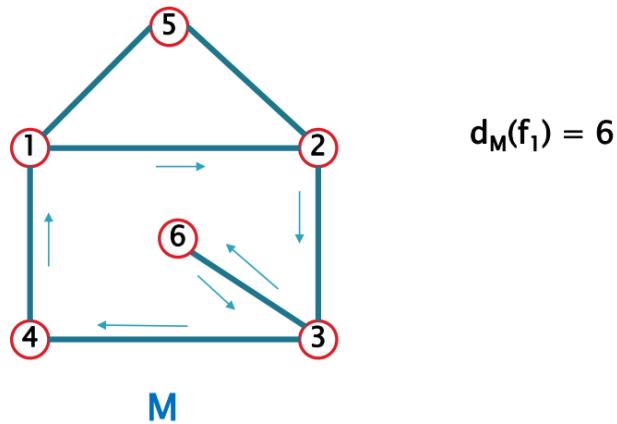
$d_M(f) = \text{gradul feței } f = \text{numărul muchiilor lanțului închis (frontierei) care delimită } f$ (*câte muchii sunt parcuse atunci când traversăm frontiera*)

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



$$\begin{aligned} d_{M_1}(f_1) &= 3 \\ d_{M_1}(f_2) &= 5 \\ d_{M_1}(f_3) &= 3 \\ d_{M_1}(f_4) &= 5 \end{aligned}$$

$$\begin{aligned} d_{M_2}(f_1) &= 3 \\ d_{M_2}(f_2) &= 6 \\ d_{M_2}(f_3) &= 3 \\ d_{M_2}(f_4) &= 4 \end{aligned}$$



► 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 (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 $d(x) \leq 3$.

► 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 $d(x) \leq 5$.

► 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 $G - x$)

coada $C = \emptyset$;

adauga in C toate vârfurile v cu $d[v] \leq 5$ și marcheaza-le ca vizitate

stiva $S = \emptyset$

cat timp $C \neq \emptyset$ executa

$i \leftarrow$ extrage(C);

adauga i in S

pentru $ij \in E$ executa

$d[j] = d[j] - 1$

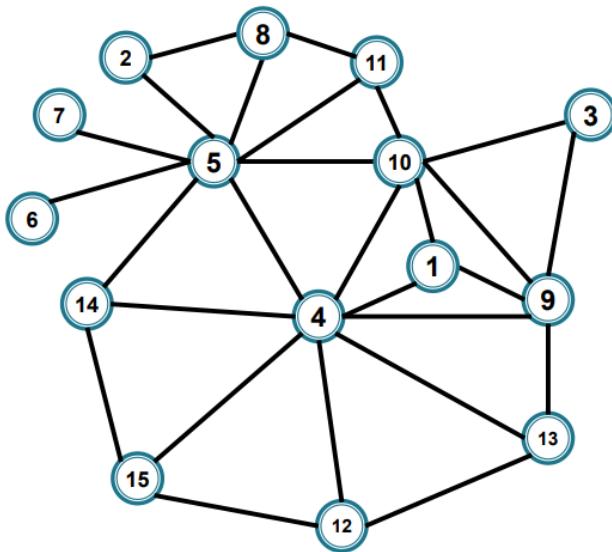
daca $d[j] \leq 5$ si j este nevizitat atunci

adauga(j, C)

cat timp S este nevida executa

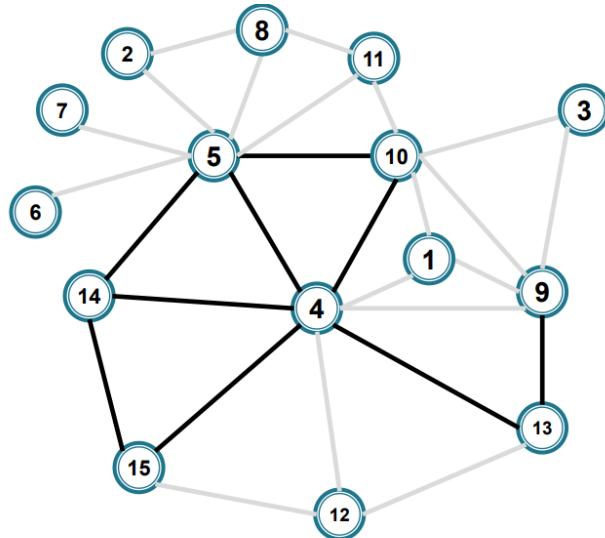
$u = \text{pop}(S)$

adauga u in sortare



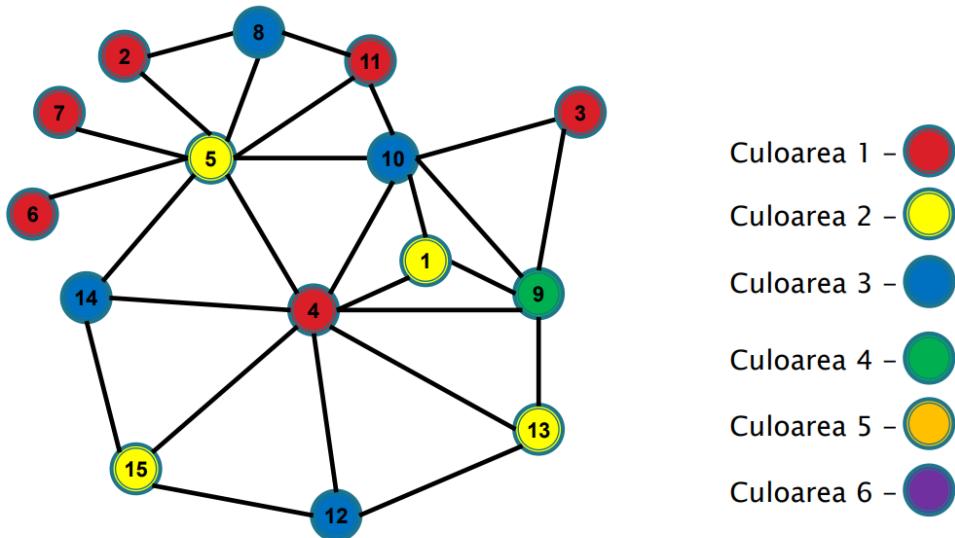
1, 2, 3, 6, 7, 8, 9, 11, 12, 13, 14, 15

grad ≤ 5



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

Ordinea în care se colorează vîrfurile



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)

Algoritm Greedy de coloare

- ▶ Repetarea algoritmului pe ordonări diferite:

repetă în timpul avut la dispoziție:

- generează aleator o ordonare a vârfurilor
- colorează G folosind algoritmul Greedy pentru această ordonare
- dacă colorarea obținută folosește un număr mai mic de culori decât cea mai bună găsită până acum, memorează această colorare ca fiind cea mai bună

FORD-FLUKERSON

Algoritmul se bazeaza pe BFS.

PENTRU FLUX MAX: (f/c)

1) MUCHIE DIRECTA

- o selectam daca are ce sa mai adauge ($f < c$)
- o selectam si daca $f = 0$
- cand calculam pt lant adaugam la f

2) MUCHIE INVERSA

- o selectam daca am ce lua din ea (si daca $f = c$)
- nu o selectam cand $f = 0$
- cand calculam pt lant scadem din f

Cand calculam minimul lantului:

- pt muchii directe luam $c - f$
- si pt muchii inverse luam f

Bugatti

