

# Reprezentări ale grafurilor. Parcurgeri în grafuri



# Reprezentări



# Reprezentări

Ce reprezentări ale grafurilor cunoașteți?

# Reprezentări

Ce reprezentări ale grafurilor cunoașteți?

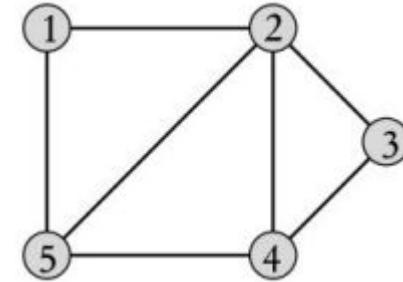
- matrice de adiacență
- liste de adiacență
- lista de muchii

# Reprezentări

Ce reprezentări ale grafurilor cunoașteți?

- matrice de adiacență
  - într-un graf **orientat**, matricea în general nu este simetrică

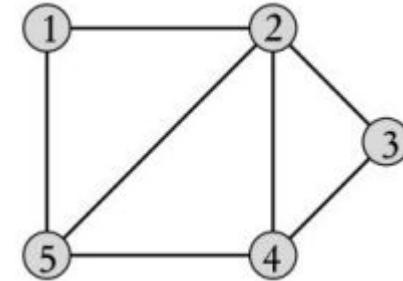
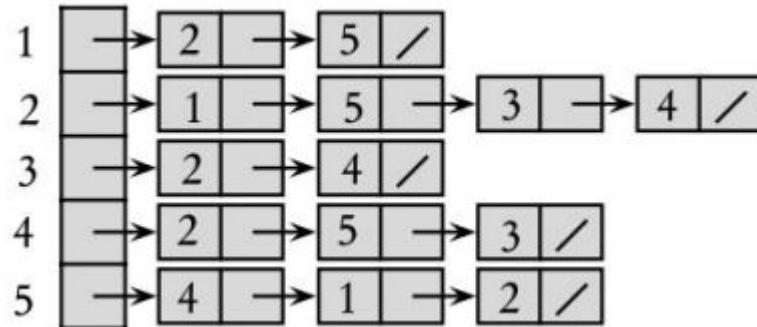
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



# Reprezentări

Ce reprezentări ale grafurilor cunoașteți?

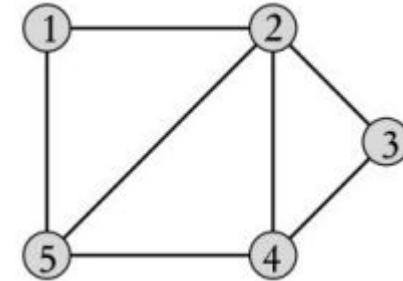
- liste de adiacență



# Reprezentări

Ce reprezentări ale grafurilor cunoașteți?

- liste de muchii
  - $[(1,2), (1,5), (2,5), (2,4), (2,3), (3,4), (4,5)]$



# Reprezentări

De ce avem mai multe reprezentări?

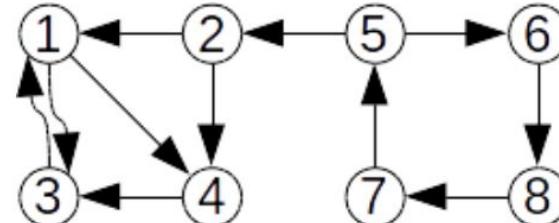
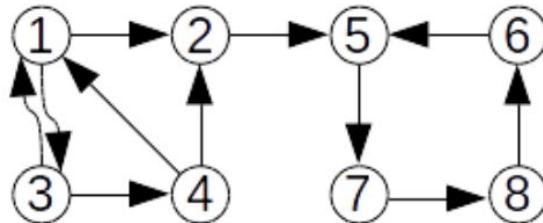
- Modul cum reprezentăm influențează complexitatea timp a algoritmilor implementați
  - Exemplu DF:
    - $O(n^2)$  matrice de adiacență
    - $O(n+m)$  lista de vecini
    - $O(n*m)$  lista de muchii
- Student:
  - În funcție de graful reprezentat, unele reprezentări consumă mai multă, respectiv mai puțină memorie
  - Memorie consumată:
    - Matrice de adiacență  $O(n^2)$
    - Lista de vecini:  $O(n+m)$
    - Lista de muchii:  $O(m)$
  - Dacă avem graf dens (aproape complet), matricea de adiacență nu va mai fi rea ... pentru că  $m \sim n^2$

# Reprezentări

## Graful Transpus

Fie  $G = (V, E)$  un graf **orientat**. Se numește **graf transpus** al lui  $G$  graful orientat  $G^T = (V, E^T)$  având aceeași multime de noduri, iar  $E^T = \{(y, x) \mid (x, y) \in E\}$ .

Practic, sensul arcelor se schimbă, pentru toate arcele din graful  $G$ .



# Parcurgeri



# Parcurgerea în lățime (BFS)

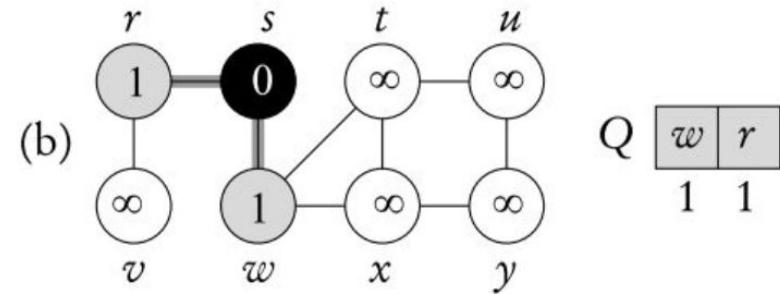
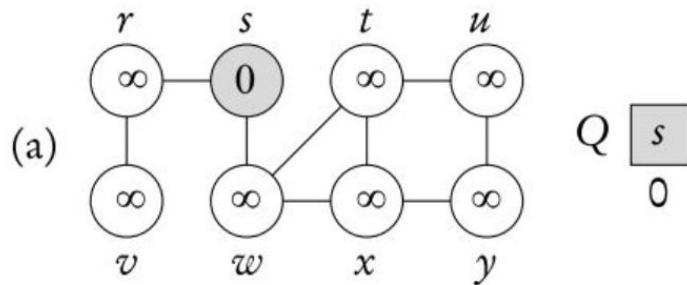
Dat fiind un graf  $G = (V, E)$  și un nod sursă  $s$ , **căutarea în lățime** explorează sistematic muchiile lui  $G$  pentru a “descoperi” fiecare nod care este accesibil din  $s$ .

De asemenea, algoritmul găsește distanța minimă de la sursa  $s$  la toate nodurile din graf.

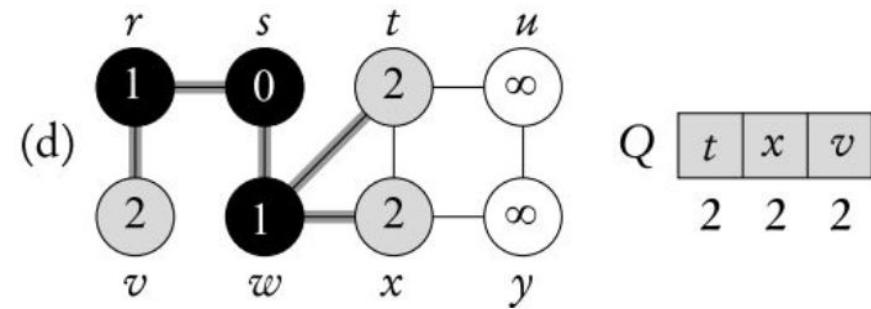
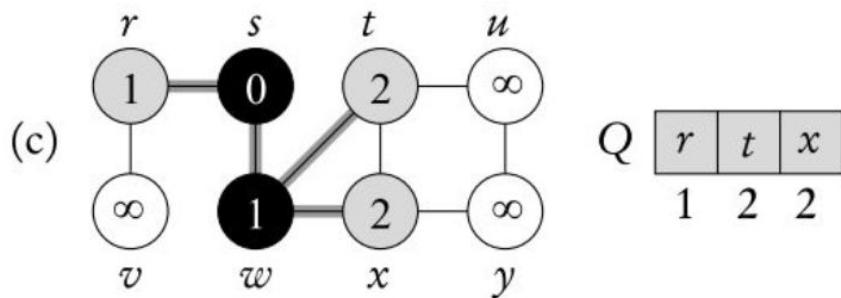
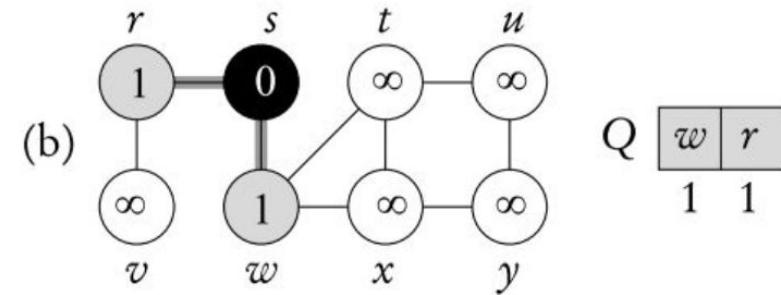
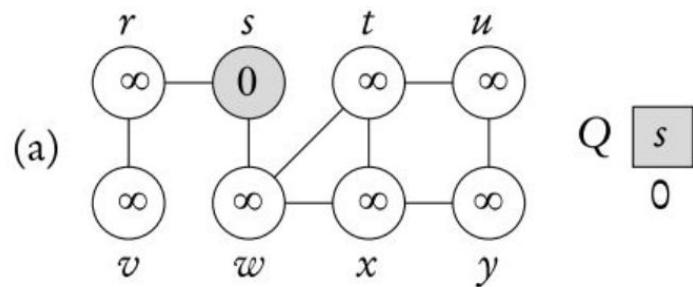
# Parcurgerea în lățime (BFS)

1. Se începe explorarea dintr-un nod nevizitat, care se adaugă într-o coadă
2. Cât timp există elemente în coadă
  - a. Se scoate din coadă
  - b. Se pun în coadă toți vecinii nevizitați

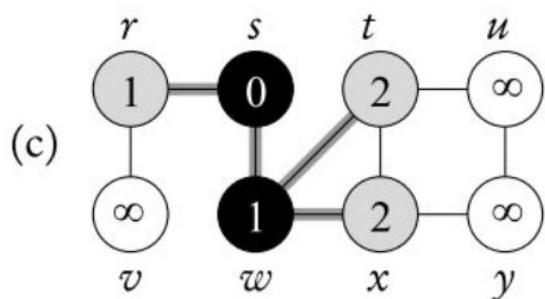
# Parcurgerea în lățime - Exemplu



# Parcurgerea în lățime - Exemplu

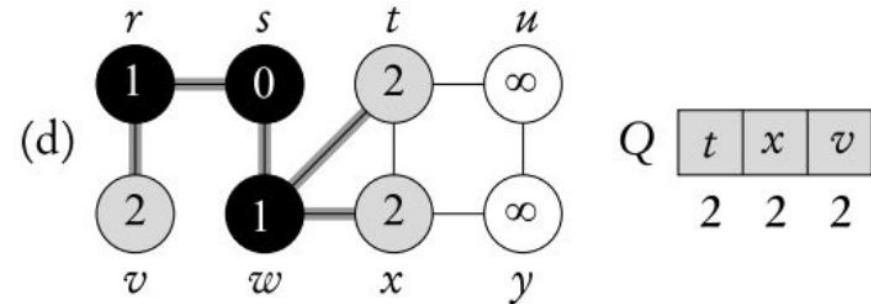


# Parcurgerea în lățime - Exemplu



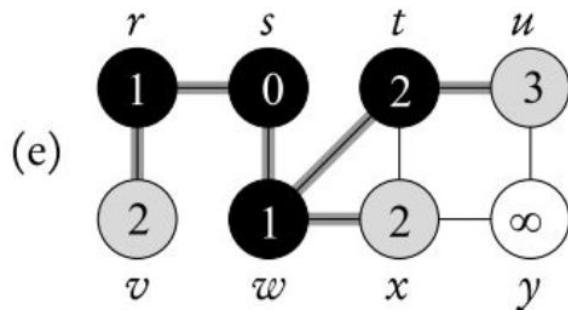
$Q$

$r$	$t$	$x$
1	2	2



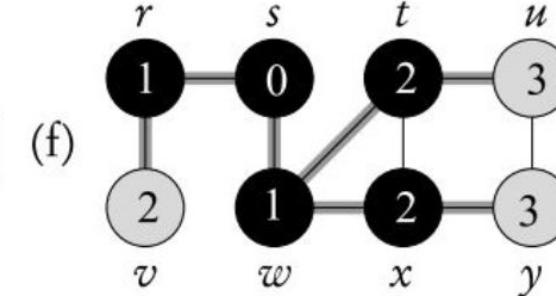
$Q$

$t$	$x$	$v$
2	2	2



$Q$

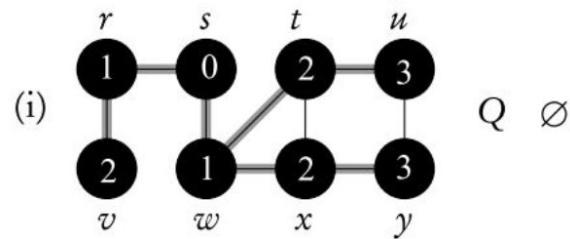
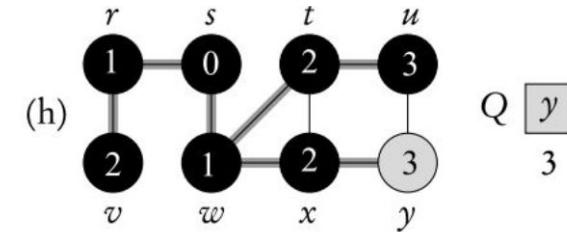
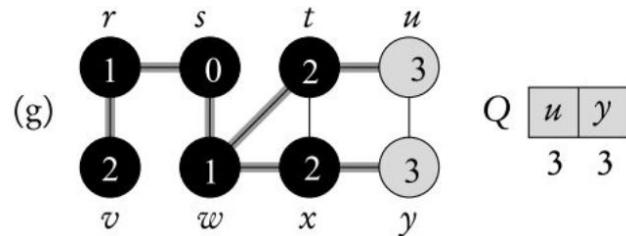
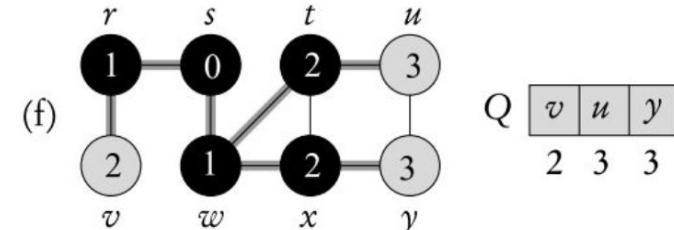
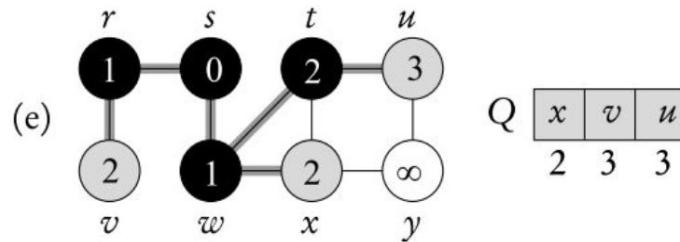
$x$	$v$	$u$
2	3	3



$Q$

$v$	$u$	$y$
2	3	3

# Parcurgerea în lățime - Exemplu



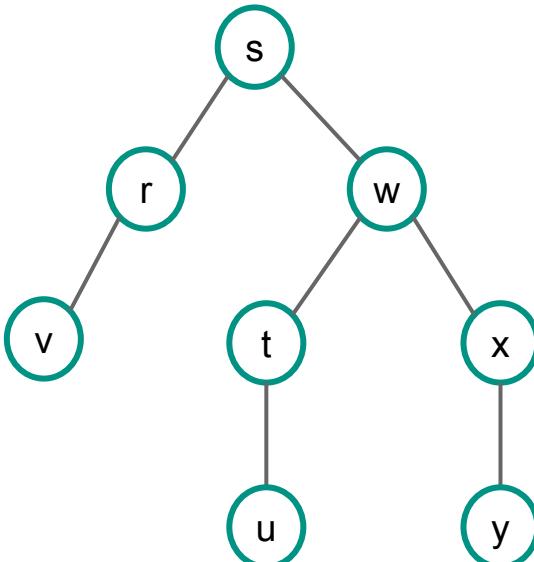
# Parcurgerea în lățime - Algoritm

Să scriem împreună pseudocodul:

# Parcurgerea în lățime - Algoritm

Algoritm Cormen:

Pi → tata



$CL(G, s)$

```
1: pentru fiecare vârf  $u \in V[G] - \{s\}$  execută
2:    $color[u] \leftarrow$  ALB
3:    $d[u] \leftarrow \infty$ 
4:    $\pi[u] \leftarrow$  NIL
5:  $color[s] \leftarrow$  GRI
6:  $d[s] \leftarrow 0$ 
7:  $\pi[s] \leftarrow$  NIL
8:  $Q \leftarrow \{s\}$ 
9: cât timp  $Q \neq \emptyset$  execută
10:    $u \leftarrow cap[Q]$ 
11:   pentru fiecare vârf  $v \in Adj[u]$  execută
12:     dacă  $color[v] =$  ALB atunci
13:        $color[v] \leftarrow$  GRI
14:        $d[v] \leftarrow d[u] + 1$ 
15:        $\pi[v] \leftarrow u$ 
16:       PUNE-ÎN-COADĂ( $Q, v$ )
17:     SCOATE-DIN-COADĂ( $Q$ )
18:    $color[u] \leftarrow$  NEGRU
```

# Parcurgerea în lățime - Complexitate

Care este complexitatea algoritmului?

# Parcurgerea în lățime - Complexitate

Care este complexitatea algoritmului?

- O( $V+E$ ) sau O( $n+m$ )
  - unde
    - $V = n$  = numărul de noduri
    - $E = m$  = numărul de muchii

# Parcurgerea în lățime - Aplicații

## Ieșirea din labirint într-un număr minim de pași

0 0 → punct de pornire

0 7 → punct de ieșire

0 0 0 0 0 0 -1 0

0 -1 -1 -1 -1 -1 -1 0

0 0 0 0 0 0 -1 0

-1 -1 -1 -1 -1 0 -1 0

0 0 0 0 -1 0 0 0

# Parcurgerea în lățime - Aplicații

## Ieșirea din labirint într-un număr minim de pași

0 0 → punct de pornire

0 7 → punct de ieșire

0 1 2 3 4 5 -1 15

1 -1 -1 -1 -1 -1 -1 14

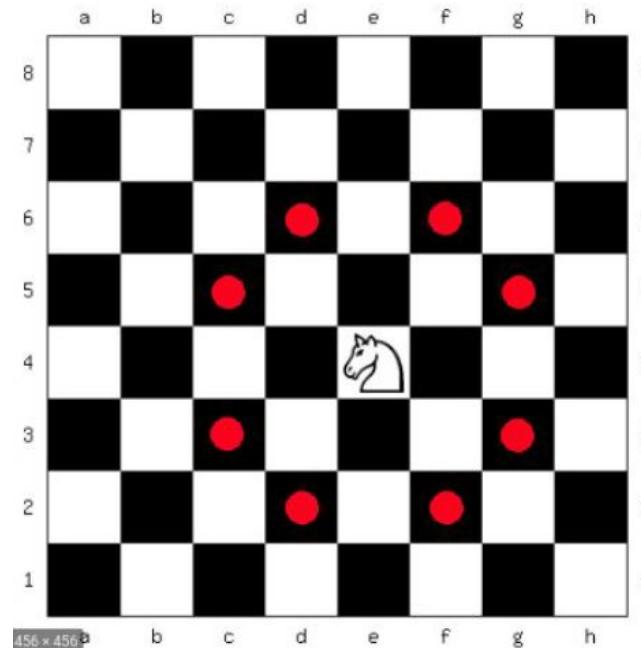
2 3 4 5 6 7 -1 13

-1 -1 -1 -1 -1 8 -1 12

0 0 0 0 -1 9 10 11

# Parcurgerea în lățime - Aplicații

Drum de lungime minimă a calului pe tabla de șah



# Parcurgerea în adâncime (DFS)

1. Se începe explorarea dintr-un nod nevizitat
2. Se caută un vecin nevizitat, care devine nodul curent.  
Cât timp nodul curent are un vecin nevizitat, repetăm pasul 2 (intrăm în adâncime)
3. Când nodul curent nu are niciun vecin nevizitat, ne întoarcem la strămoșii lui (tatăl, bunicul etc), până găsim un nod care are vecini nevizitați și reluăm pasul 2
4. Dacă există noduri nevizitate, reluăm pasul 1

Când se întâmplă pasul 4?



# Parcurgerea în adâncime (DFS)

1. Se începe explorarea dintr-un nod nevizitat
2. Se caută un vecin nevizitat, care devine nodul curent.  
Cât timp nodul curent are un vecin nevizitat, repetăm pasul 2 (intrăm în adâncime)
3. Când nodul curent nu are niciun vecin nevizitat, ne întoarcem la strămoșii lui (tatăl, bunicul etc), până găsim un nod care are vecini nevizitați și reluăm pasul 2
4. Dacă există noduri nevizitate, reluăm pasul 1

Când se întâmplă pasul 4?

- Când graful are mai multe componente conexe

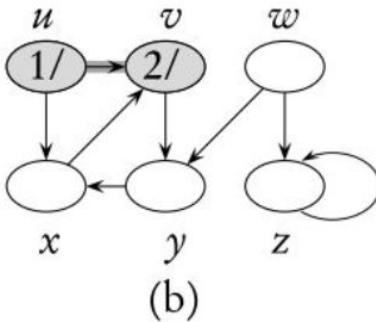
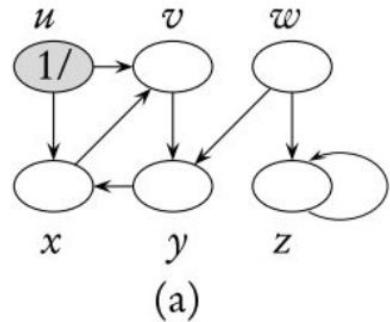
# Parcuregerea în adâncime - Cu timpi de intrare și ieșire

Pentru o parcuregere în adâncime, este uneori util să ținem minte cronologia parcurgerii.

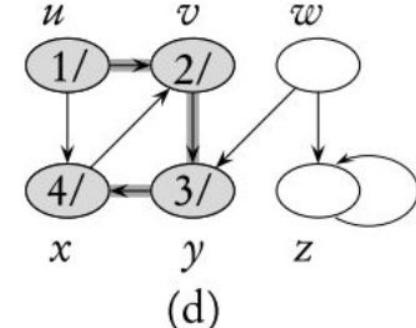
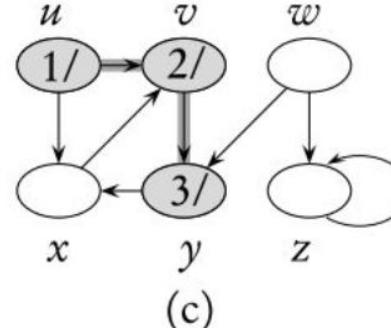
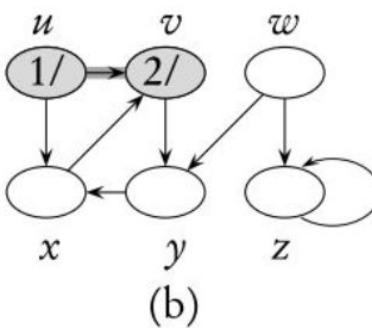
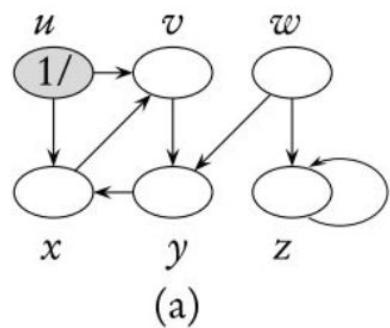
- De fiecare dată când ajungem într-un nod sau când terminăm de vizitat toți vecinii, incrementăm un contor și ținem minte aceste informații

**Observație:** O să existe situații în care cronologia va fi un pic diferită, cum s-a întâmplat la RMQ → LCA

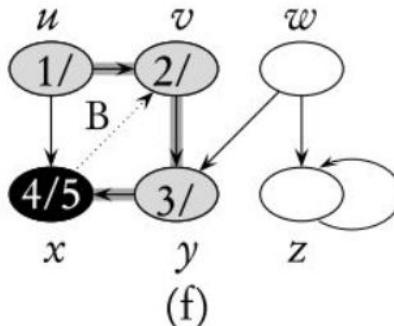
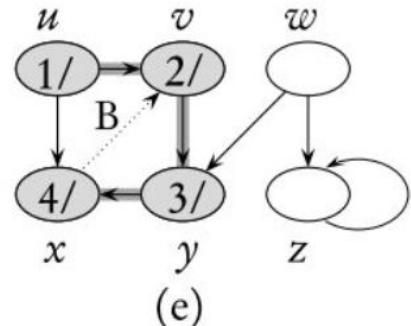
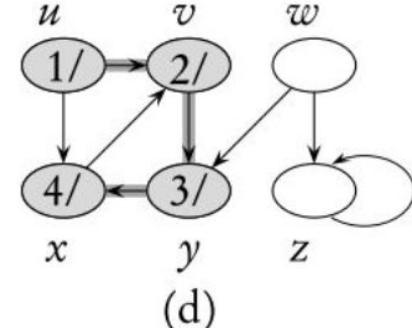
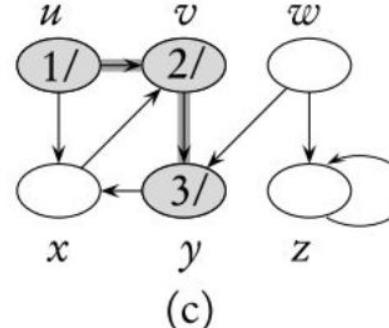
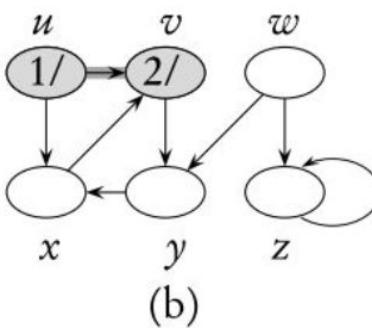
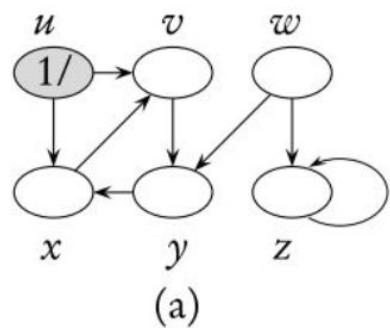
# Parcurea în adâncime - Exemplu



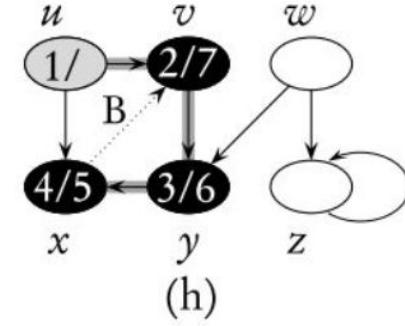
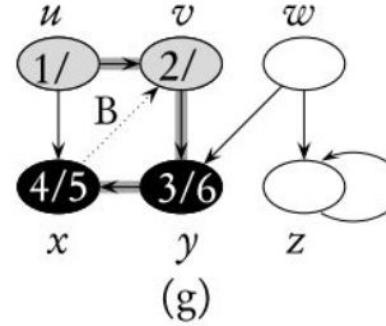
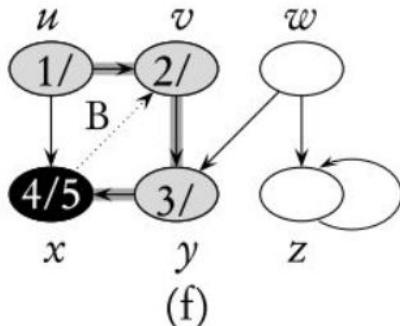
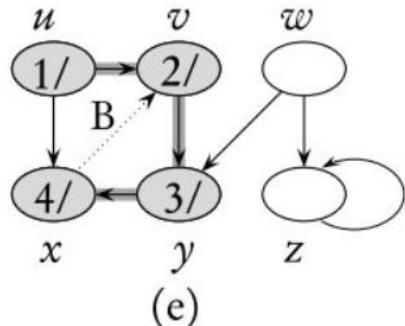
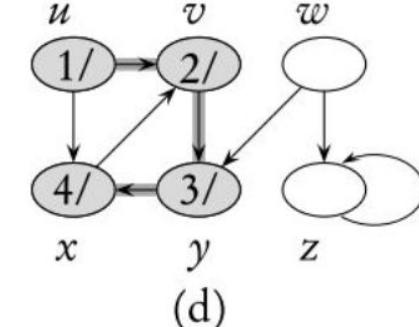
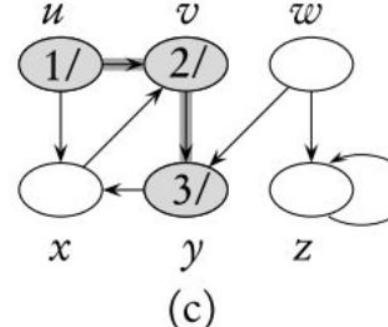
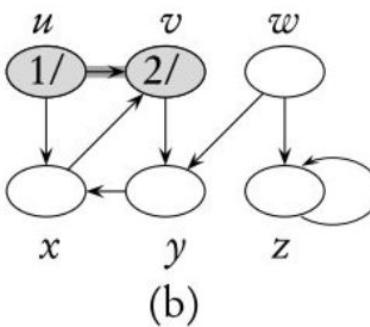
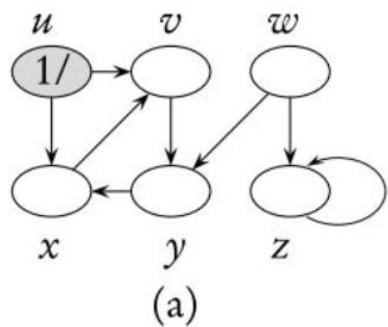
# Parcurea în adâncime - Exemplu



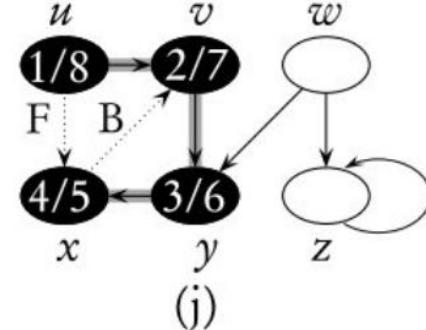
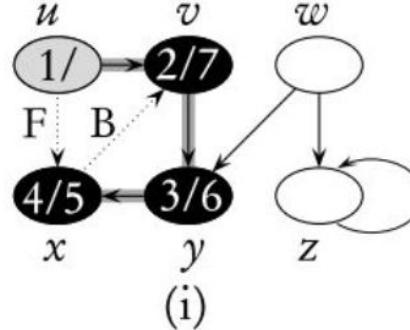
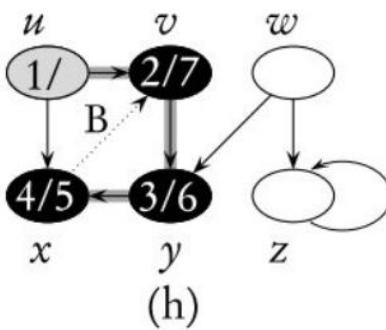
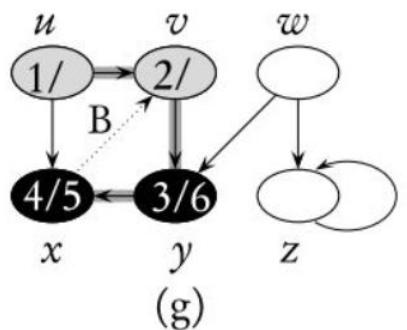
# Parcurgerea în adâncime - Exemplu



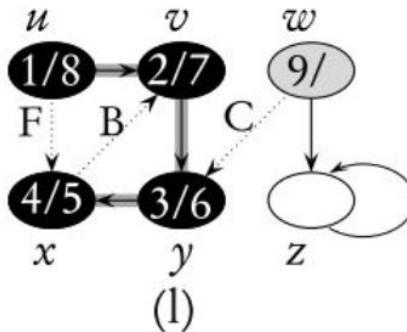
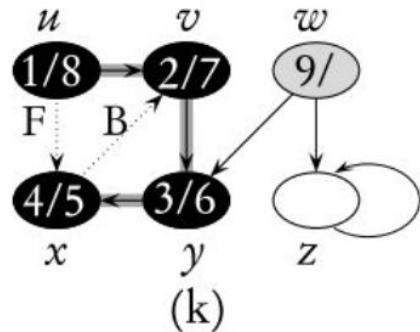
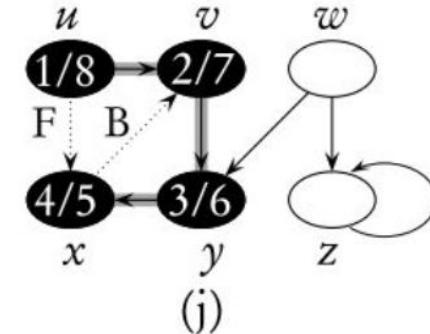
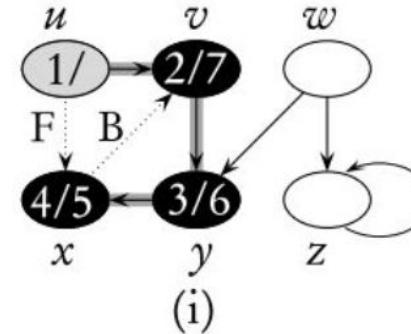
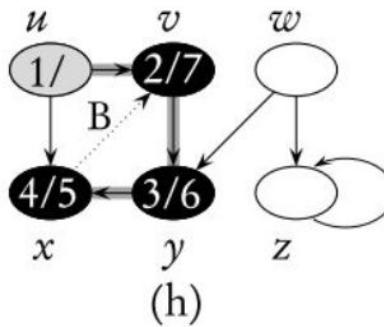
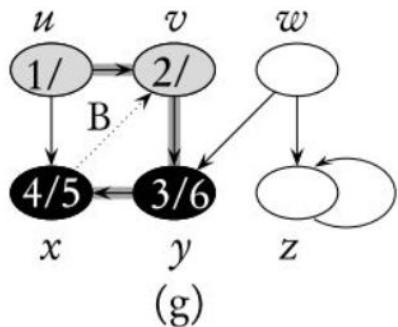
# Parcurgerea în adâncime - Exemplu



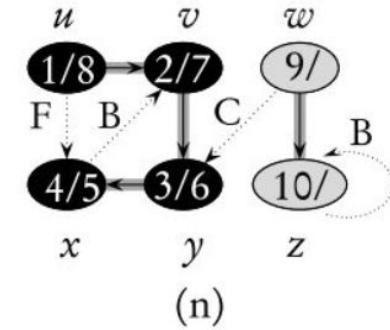
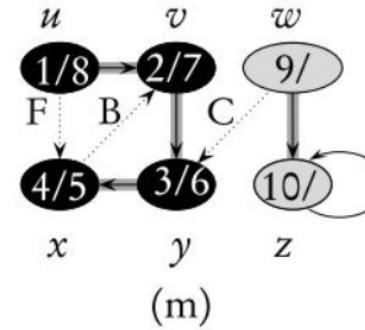
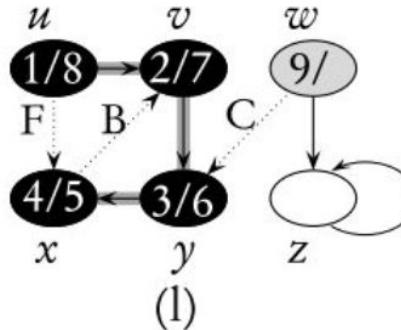
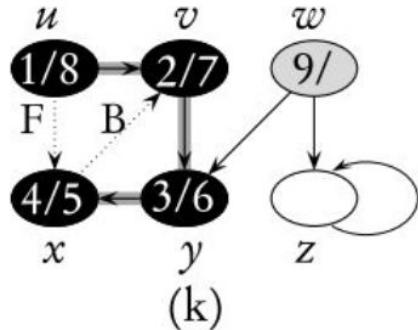
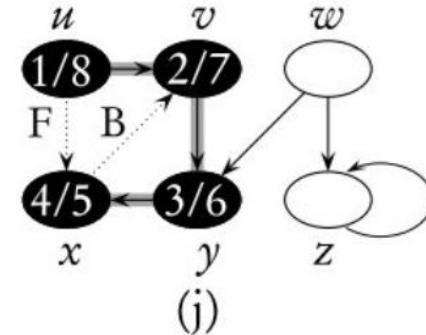
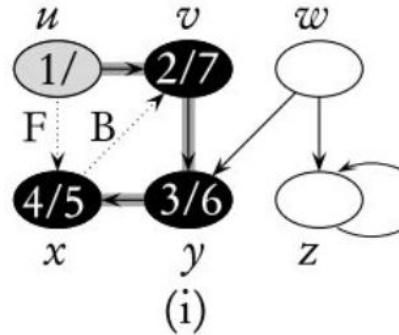
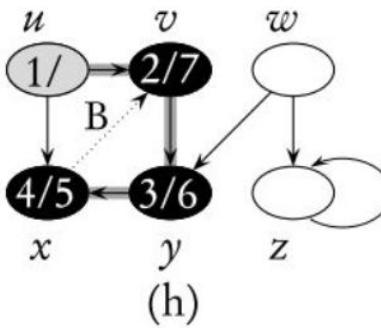
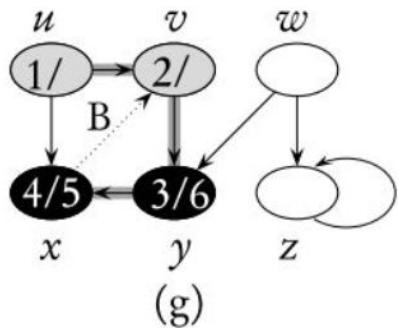
# Parcurgerea în adâncime - Exemplu



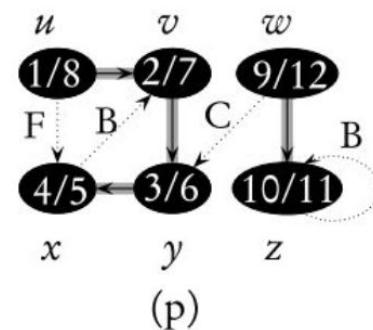
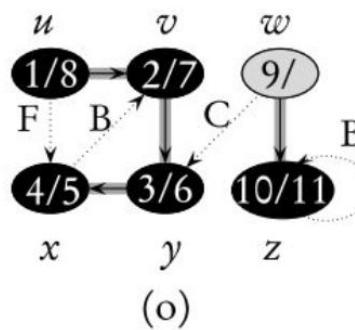
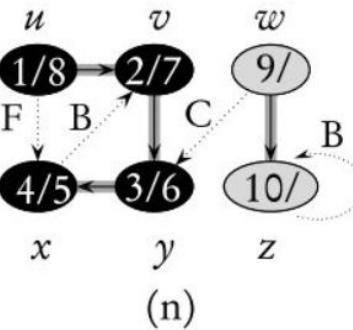
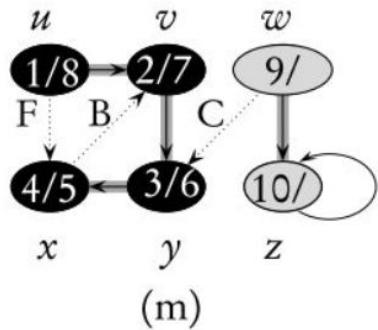
# Parcurgerea în adâncime - Exemplu



# Parcurgerea în adâncime - Exemplu



# Parcurgerea în adâncime - Exemplu



# Parcurea în adâncime - Algoritm

Să scriem împreună pseudocodul:

# Parcurea în adâncime – Algoritm

## Algoritm Cormen:

CA( $G$ )

- 1: **pentru** fiecare vârf  $u \in V[G]$  **execută**
  - 2:     $culoare[u] \leftarrow \text{ALB}$
  - 3:     $\pi[u] \leftarrow \text{NIL}$
  - 4:     $temp \leftarrow 0$
  - 5: **pentru** fiecare vârf  $u \in V[G]$  **execută**
  - 6:    **dacă**  $culoare[u] = \text{ALB}$  **atunci**
  - 7:     CA-VIZITĂ( $u$ )

### CA-VIZITĂ( $u$ )

- 1:  $cupoare[u] \leftarrow \text{GRI}$  ▷ Vârful alb  $u$  tocmai a fost descoperit.  
 2:  $d[u] \leftarrow \text{temp} \leftarrow \text{temp} + 1$   
 3: **pentru** fiecare  $v \in \text{Adj}[u]$  **execută** ▷ Explorează muchia  $(u, v)$ .  
 4:   **dacă**  $cupoare[v] = \text{ALB}$  **atunci**  
 5:      $\pi[v] \leftarrow u$   
 6:     CA-VIZITĂ( $v$ )  
 7:  $cupoare[u] \leftarrow \text{NEGRU}$  ▷ Vârful  $u$  este colorat în negru. El este terminat.  
 8:  $f[u] \leftarrow \text{temp} \leftarrow \text{temp} + 1$

# Parcurgerea în adâncime - Complexitate

Care este complexitatea algoritmului?

# Parcurgerea în adâncime - Complexitate

Care este complexitatea algoritmului?

- O( $V+E$ ) sau O( $n+m$ )
  - unde
    - $V = n$  = numărul de noduri
    - $E = m$  = numărul de muchii

# Parcurea în adâncime - Teorema parantezelor

## Teorema parantezelor

În orice căutare în adâncime a unui graf (orientat sau neorientat)  $G = (V, E)$ , pentru oricare două vârfuri  $u$  și  $v$ , exact una din următoarele trei condiții este adevărată:

- intervalele  $[d[u], f[u]]$  și  $[d[v], f[v]]$  sunt total disjuncte
- intervalul  $[d[u], f[u]]$  este conținut, în întregime, în intervalul  $[d[v], f[v]]$ , iar  $u$  este un descendent al lui  $v$  în arborele de adâncime
- intervalul  $[d[v], f[v]]$  este conținut, în întregime, în intervalul  $[d[u], f[u]]$ , iar  $v$  este un descendent al lui  $u$  în arborele de adâncime

# Parcurea în adâncime - Teorema parantezelor

## Teorema parantezelor: Demonstrație

Începem cu cazul în care  $d[u] < d[v]$ .

În funcție de valoarea de adevăr a inegalității  $d[v] < f[u]$ , există două subcazuri care trebuie considerate.

1. **În primul subcaz:**  $d[v] < f[u]$ , deci v a fost descoperit, în timp ce u era încă gri. Aceasta implică faptul că v este un descendant al lui u. Mai mult, deoarece v a fost descoperit înaintea lui u, toate muchiile care pleacă din el sunt explorate, iar v este terminat, înainte ca algoritmul să revină pentru a-l termina pe u. De aceea, în acest caz, intervalul  $[d[v], f[v]]$  este conținut în întregime în intervalul  $[d[u], f[u]]$ .
2. **În celălalt subcaz:**  $f[u] < d[v]$  și din inegalitatea  $d[u] < f[u]$  implică faptul că intervalele  $[d[u], f[u]]$  și  $[d[v], f[v]]$  sunt disjuncte.

Cazul în care  $d[v] < d[u]$  este similar, inversând rolurile lui u și v în argumentația de mai sus.

# Parcurea în adâncime - Teorema parantezelor

## Corolarul 23.7 (Interclasarea intervalorilor descendenților)

Vârful  $v$  este un descendant al lui  $u$  în pădurea de adâncime pentru un graf  $G$  orientat sau neorientat dacă și numai dacă  $d[u] < d[v] < f[v] < f[u]$ .

# Parcugerea în adâncime - Teorema parantezelor

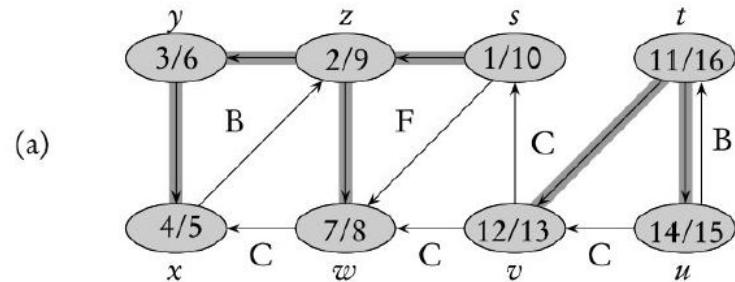
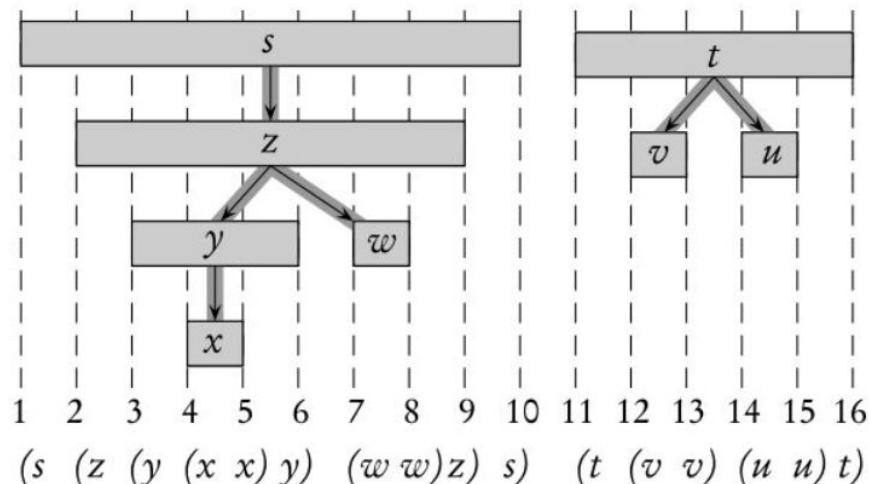


Diagrama b exemplifică foarte bine teorema anterioară.

(b)



# Parcurea în adâncime - Teorema parantezelor

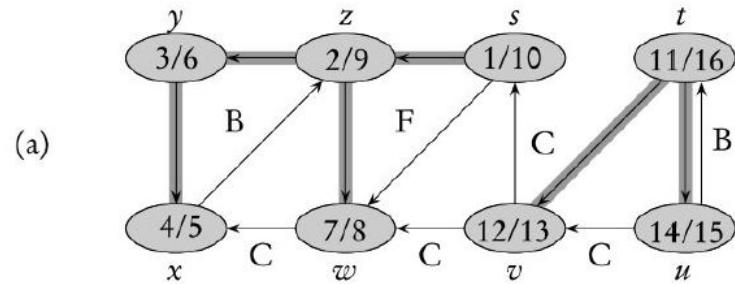
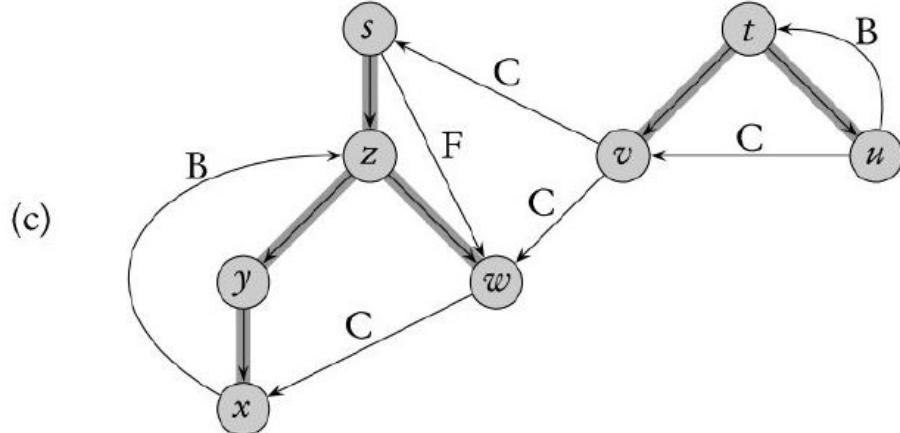


Diagrama c exemplifică muchiile de întoarcere despre care vom vorbi imediat.



# Parcurea în adâncime - Clasificarea muchiilor

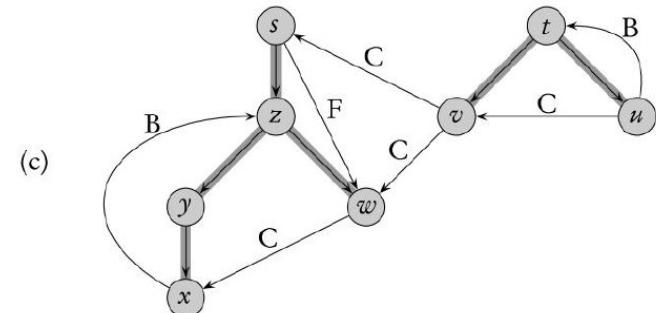
## Clasificarea muchiilor

1. **Muchiile de arbore** sunt muchii din pădurea de adâncime  $G\pi$ . Muchia  $(u, v)$  este o muchie de arbore dacă  $v$  a fost descoperit explorând muchia  $(u, v)$ .
2. **Muchiile înapoi** sunt acele muchii  $(u, v)$  care unesc un vârf  $u$  cu un strămoș  $v$  într-un arbore de adâncime. Bucile (muchii de la un vârf la el însuși) care pot apărea într-un graf orientat sunt considerate muchii înapoi.
3. **Muchiile înainte** sunt acele muchii  $(u, v)$  ce nu sunt muchii de arbore și conectează un vârf  $u$  cu un descendant  $v$  într-un arbore de adâncime.
4. **Muchiile transversale** sunt toate celelalte muchii. Ele pot uni vârfuri din același arbore de adâncime, cu condiția ca unul să nu fie strămoșul celuilalt, sau pot uni vârfuri din arbori de adâncime diferiți.

# Parcurea în adâncime - Clasificarea muchiilor

## Clasificarea muchiilor

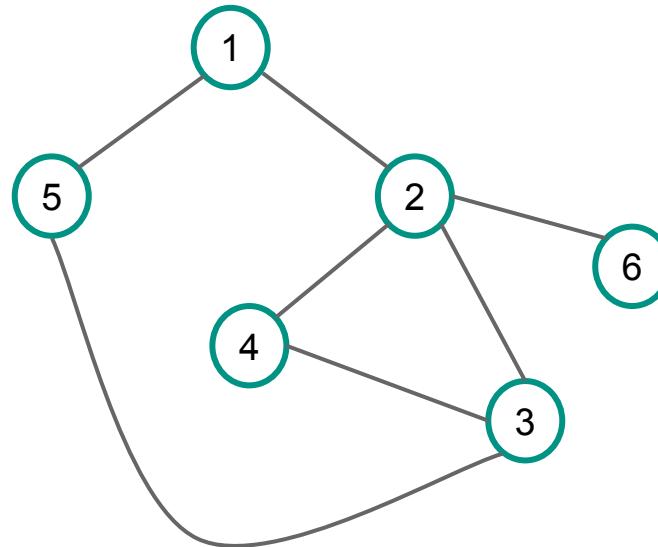
1. **Muchiile de arbore** sunt muchii din pădurea de adâncime  $G\pi$ . Muchia  $(u, v)$  este o muchie de arbore dacă  $v$  a fost descoperit explorând muchia  $(u, v)$ .
2. **Muchiile înapoi** sunt acele muchii  $(u, v)$  care unesc un vârf  $u$  cu un strămoș  $v$  într-un arbore de adâncime. Bucile (muchii de la un vârf la el însuși) care pot apărea într-un graf orientat sunt considerate muchii înapoi.
3. **Muchiile înainte** sunt acele muchii  $(u, v)$  ce nu sunt muchii de arbore și conectează un vârf  $u$  cu un descendant  $v$  într-un arbore de adâncime.
4. **Muchiile transversale** sunt toate celelalte muchii. Ele pot uni vârfuri din același arbore de adâncime, cu condiția ca unul să nu fie strămoșul celuilalt, sau pot uni vârfuri din arbori de adâncime diferiți.



# Parcurea în adâncime - Clasificarea muchiilor

Într-un graf neorientat nu vom avea toate cele 4 categorii de muchii.

Ce categorii vom avea?

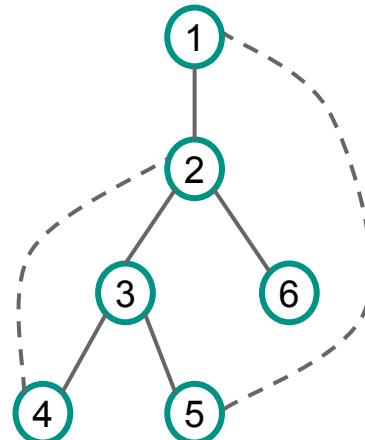
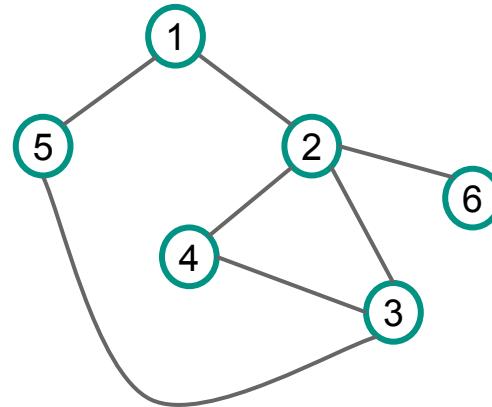


# Parcurea în adâncime - Clasificarea muchiilor

Într-un graf neorientat nu vom avea toate cele 4 categorii de muchii.

**Ce categorii vom avea?**

- doar primele două categorii (muchii de arbore și muchii înapoi)



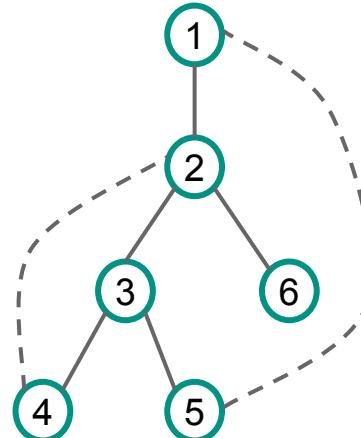
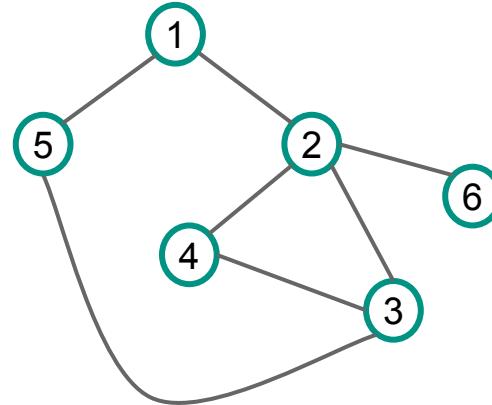
# Parcurea în adâncime - Clasificarea muchiilor

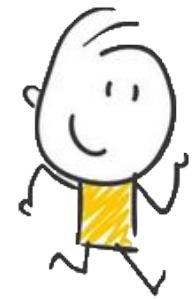
Într-un graf neorientat vom avea un ciclu dacă găsim ce fel de muchie?

# Parcurea în adâncime - Clasificarea muchiilor

Într-un graf neorientat vom avea un ciclu dacă găsim ce fel de muchie?

- muchie înapoi





# Sortarea topologică



# Sortarea topologică

Fie  $G = (V, E)$  un **graf orientat**.

**Sortarea topologică a lui  $G$  =**

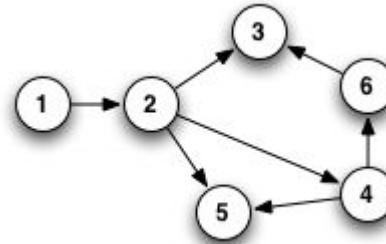
ordonarea vârfurilor astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$  în ordonare.

**Nu este neapărat unică!**

# Sortarea topologică - Aplicații

- Ordinea de calcul în proiecte în care intervin relații de dependență / precedentă  
**(exemplu:** calcul de formule, ordinea de compilare când clasele/pachetele depind unele de altele)
- Detectie de deadlock
- Determinarea de drumuri critice

Activitatea 5 depinde de activitatea 4,  
deci trebuie să se desfășoare după ea



În ce ordine trebuie executate activitățile?

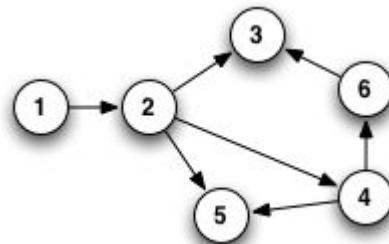
Formulele din celulele B2..D2

	A	B	C	D
1				
2				
3				
4		"=B1+D2"	"=2*B2"	"=2*C1+C2"

În ce ordine se evaluatează formulele?  
**Probleme: dacă există dependențe circulare**

# Sortarea topologică - Aplicații

- planificarea de proiecte, ordinea de execuție a unor operații: compilarea pachetelor, ordinea de calcul a formulelor din excel etc
- determinarea de drumuri minime în grafuri fără circuite



În ce ordine trebuie executate activitățile?

	A	B	C	D
1				
2		3	2	
3		3	6	0
4	=B1+D2"	"=2*B2"	"=2*C1+C2"	

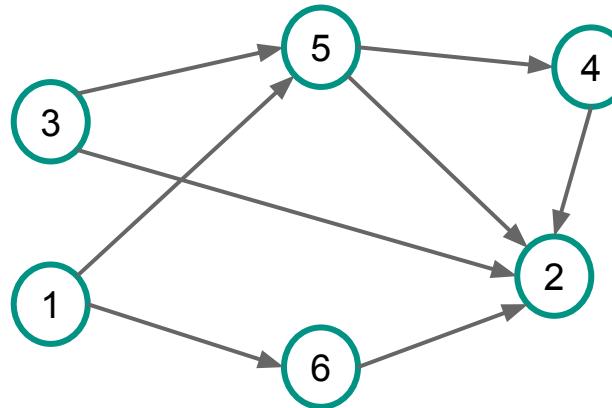
În ce ordine se evaluatează formulele?  
Probleme: dacă există dependențe circulare

# Sortarea topologică

Fie  $G = (V, E)$  un **graf orientat**.

**Sortarea topologică a lui  $G$**  =

ordonarea vârfurilor astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$  în ordonare.



# Sortarea topologică

Fie  $G = (V, E)$  un **graf orientat**.

**Sortarea topologică a lui G =**

ordonarea 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.

# Sortarea topologică

Fie  $G = (V, E)$  un **graf orientat**.

**Sortarea topologică a lui G =**

ordonarea 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.

- Demonstrație  $\Rightarrow$  Algoritm?



# Sortarea topologică

Fie  $G = (V, E)$  un **graf orientat**.

**Sortarea topologică a lui G =**

ordonarea 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.

- Demonstrație  $\Rightarrow$  Algoritm?
  
- Care poate fi primul nod în sortarea topologică?



# Sortarea topologică

Fie  $G = (V, E)$  **graf orientat**.

**Lemă.** Dacă  $G$  este aciclic, atunci  $G$  are cel puțin un vârf  $v$  cu gradul intern 0 ( $d^-(v) = 0$ ).

# Pseudocod



# Sortarea topologică - Pseudocod

cât timp  $|V(G)| > 0$  execută

    alege  $v$  cu  $d^-(v) = 0$

    adauga  $v$  in ordonare

$G \leftarrow G - v$

**Corectitudine:** rezultă din **Lemă** + inducție

# Sortarea topologică - Pseudocod

cât timp  $|V(G)| > 0$  execută

    alege  $v$  cu  $d^-(v) = 0$

    adauga  $v$  in ordonare

$G \leftarrow G - v$



**Implementare? Complexitate?**

# Sortarea topologică - Pseudocod

cât timp  $|V(G)| > 0$  execută

alege  $v$  cu  $d^-(v) = 0$

adauga  $v$  in ordonare

$G \leftarrow G - v$

## Implementare - similar BF

- pornim cu toate vârfurile cu grad intern 0 și le adăugăm într-o coadă

# Sortarea topologică - Pseudocod

cât timp  $|V(G)| > 0$  execută

alege  $v$  cu  $d^-(v) = 0$

adauga  $v$  in ordonare

$G \leftarrow G - v$

## 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**)

# Sortarea topologică - Pseudocod

cât timp  $|V(G)| > 0$  execută

alege  $v$  cu  $d^-(v) = 0$

adauga  $v$  in ordonare

$G \leftarrow G - v$

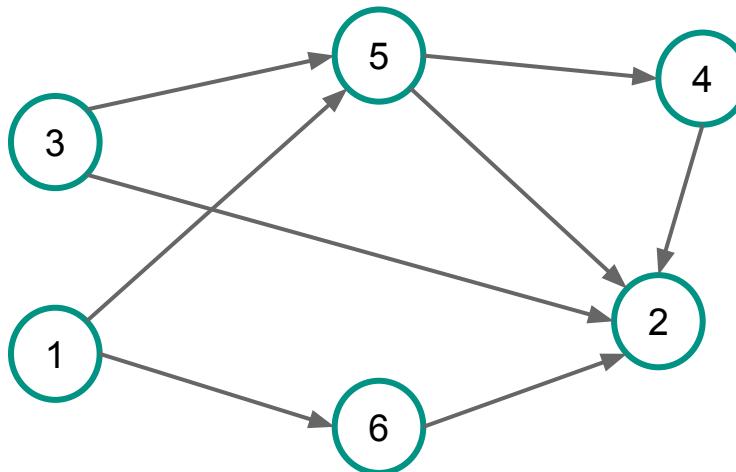
## Implementare - similar BF

- 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

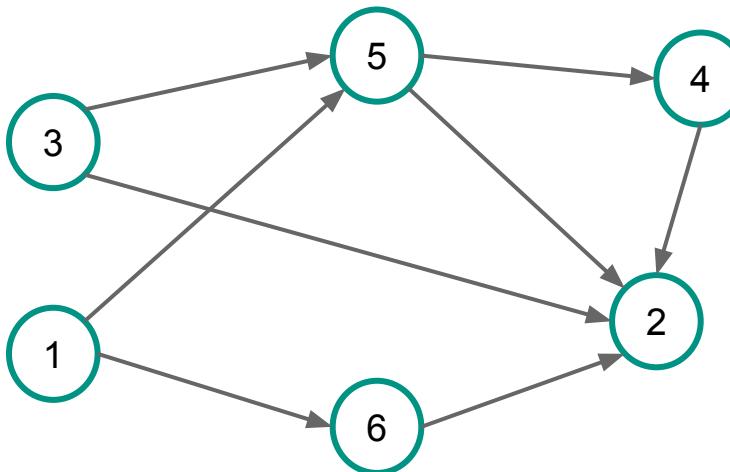
# Exemplu



# Sortare topologică - Exemplu

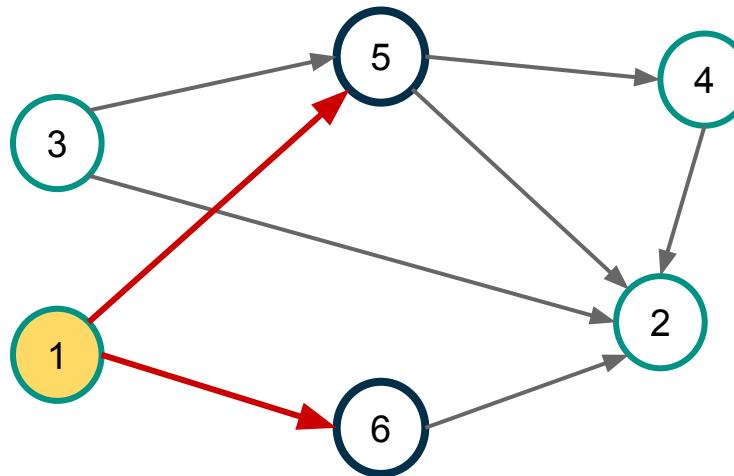


# Sortare topologică - Exemplu



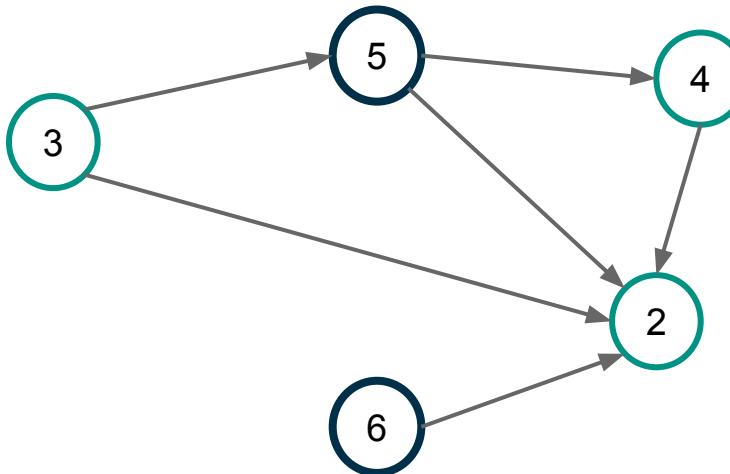
C: 1 3

# Sortare topologică - Exemplu



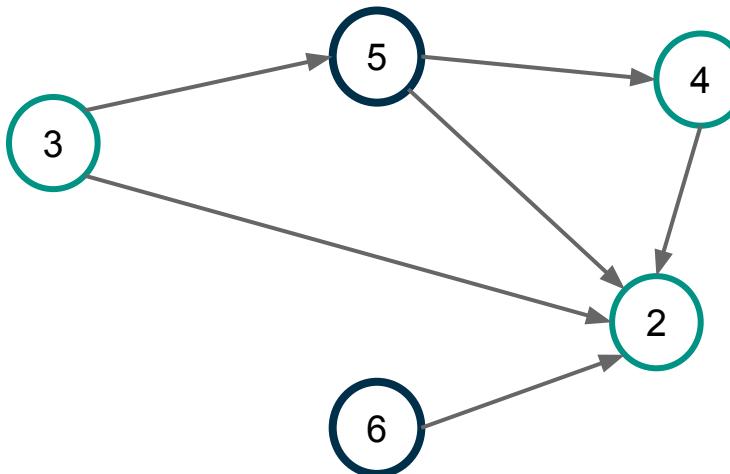
C: **1** 3

# Sortare topologică - Exemplu



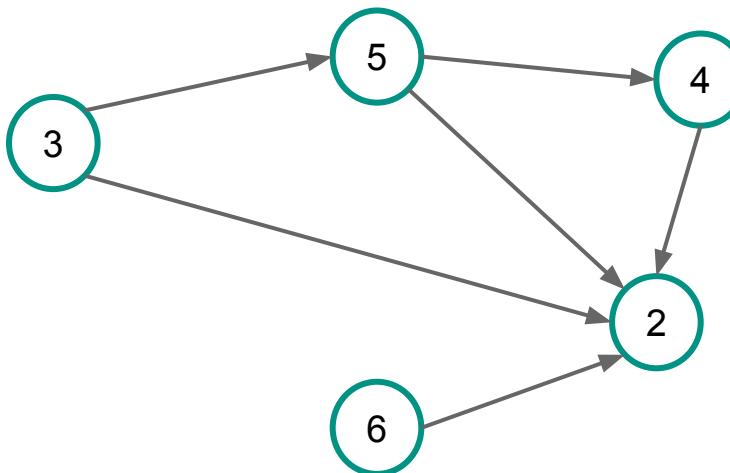
C: **1** 3

# Sortare topologică - Exemplu



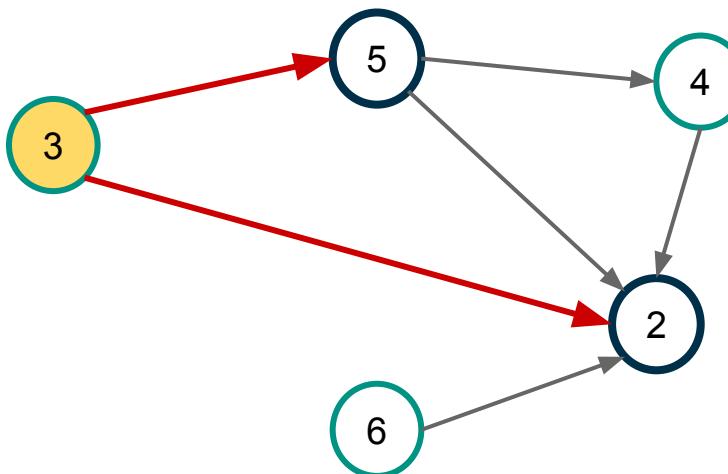
C: **1** 3 6

# Sortare topologică - Exemplu



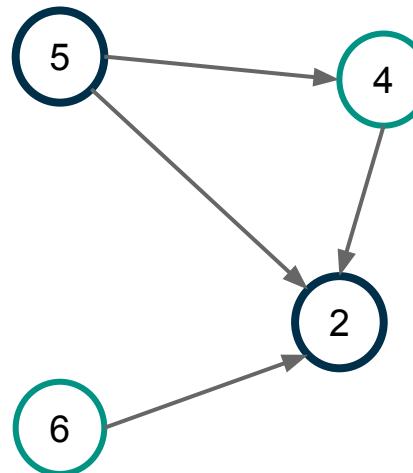
C: **1** 3 6

# Sortare topologică - Exemplu



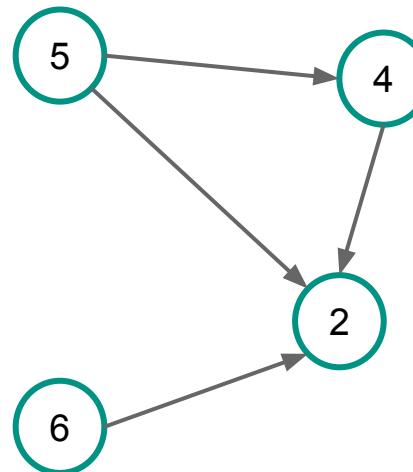
C: **1 3 6**

# Sortare topologică - Exemplu



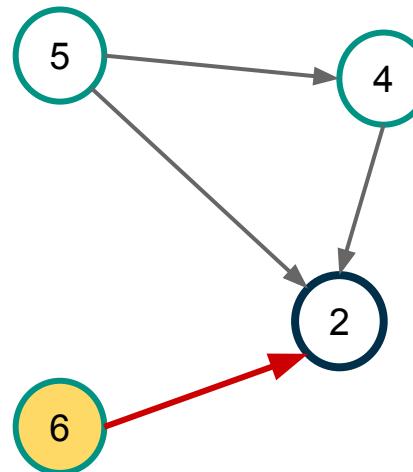
C: **1 3 6**

# Sortare topologică - Exemplu



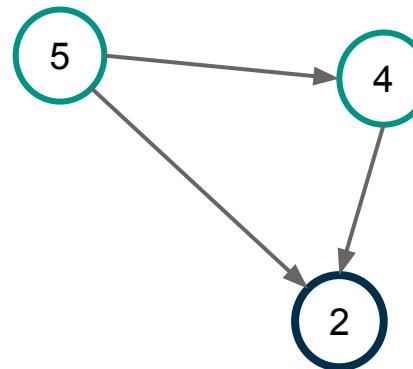
C: **1 3 6 5**

# Sortare topologică - Exemplu



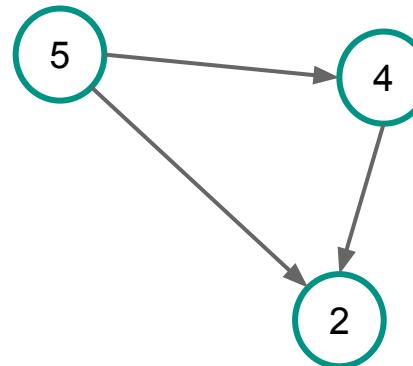
C: **1 3 6 5**

# Sortare topologică - Exemplu



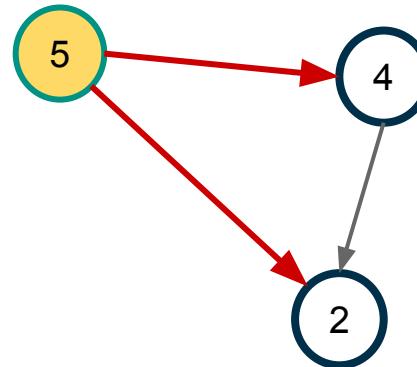
c: **1 3 6 5**

# Sortare topologică - Exemplu



c: **1 3 6 5**

# Sortare topologică - Exemplu



c: 1 3 6 5

# Sortare topologică - Exemplu



c: **1 3 6 5**

# Sortare topologică - Exemplu



C: **1 3 6 5 4**

# Sortare topologică - Exemplu



c: 1 3 6 5 4

# Sortare topologică - Exemplu

2

c: 1 3 6 5 4

# Sortare topologică - Exemplu

2

C: 1 3 6 5 4 2

# Sortare topologică - Exemplu

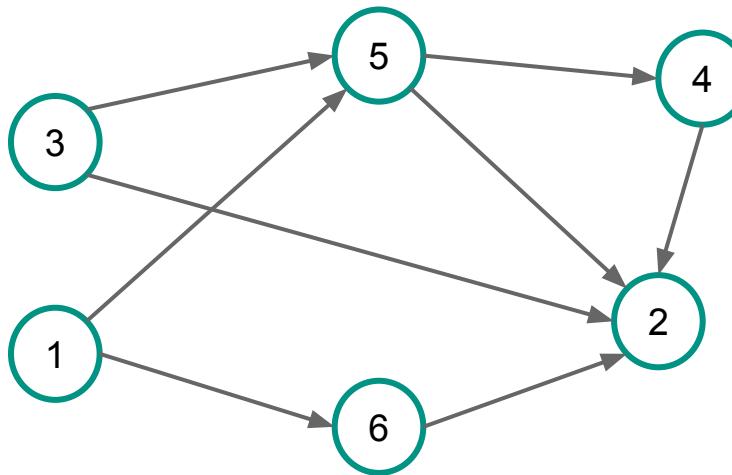


c: 1 3 6 5 4 2

# Sortare topologică - Exemplu

C: 1 3 6 5 4 2

# Sortare topologică - Exemplu



SORTARE TOPOLOGICĂ: 1 3 6 5 4 2

# Algorithm



# Sortare topologică - Algoritm

coada  $C \leftarrow \emptyset$

adauga in  $C$  toate varfurile  $v$  cu  $d^-[v]=0$

# Sortare topologică - Algoritm

**coada C**  $\leftarrow \emptyset$

**adauga** in C toate varfurile v cu  $d^-[v]=0$

**cat timp** C  $\neq \emptyset$  **executa**

    i  $\leftarrow$  **extrage**(C)

**adauga** i in sortare

**pentru** ij  $\in E$  **executa**

# Sortare topologică - Algoritm

**coada C**  $\leftarrow \emptyset$

**adauga** in C toate varfurile v cu  $d^-[v]=0$

**cat timp** C  $\neq \emptyset$  **executa**

    i  $\leftarrow$  **extrage**(C)

**adauga** i in sortare

**pentru** ij  $\in E$  **executa**

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

# Sortare topologică - Algoritm

coada  $C \leftarrow \emptyset$

adauga in  $C$  toate varfurile  $v$  cu  $d^-[v]=0$

**cat timp**  $C \neq \emptyset$  **executa**

$i \leftarrow \text{extrage}(C)$

    adauga  $i$  in sortare

**pentru**  $ij \in E$  **executa**

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

**daca**  $d^-[j] = 0$  **atunci**

            adauga( $j$ ,  $C$ )

# Sortare topologică - Algoritm



**Ce se întâmplă dacă graful conține, totuși, circuite?**

**Cum detectăm acest lucru pe parcursul algoritmului?**

# Alt algoritm



# Sortare topologică - Alt algoritm

## Suplimentar

Există un algoritm bazat pe DF, pornind de la următoarea **observație**:

- dacă **final[u]** = momentul la care a fost finalizat vârful u în parcurgerea DF, avem:
  - $uv \in E \Rightarrow f[u] > f[v]$

# Sortare topologică - Alt algoritm

## Suplimentar

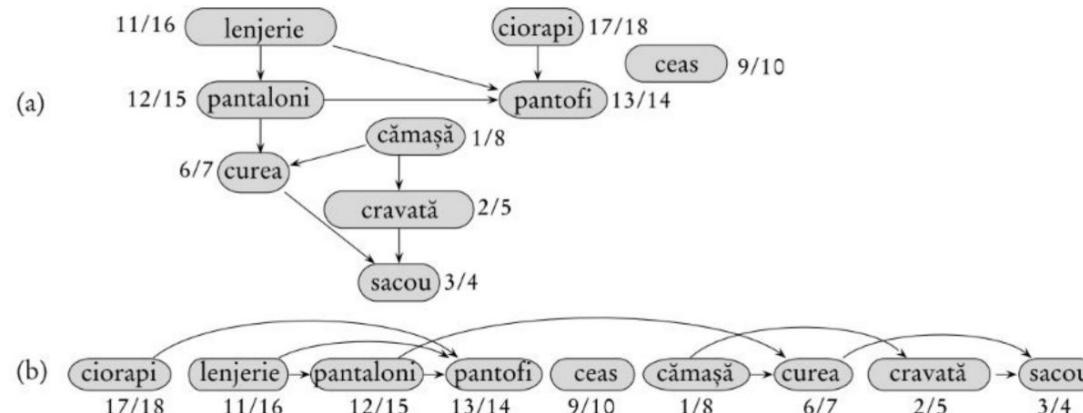
Există un algoritm bazat pe DF, pornind de la următoarea **observație**:

- dacă **final[u]** = momentul la care a fost finalizat vârful u în parcurgerea DF, avem:
  - $uv \in E \Rightarrow f[u] > f[v]$
- atunci sortarea topologică = sortare descrescătoare în raport cu **final**

# Sortare topologică - Alt algoritm

Dacă  $\text{final}[u] = \text{momentul la care a fost finalizat vârful } u \text{ în parcurgerea DF}$ , avem:  $uv \in E \Rightarrow f[u] > f[v]$

- atunci sortarea topologică = sortare descrescătoare în raport cu  $\text{final}$



**Figura 23.7** (a) Profesorul Bumstead își sortează topologic îmbrăcămintea când se îmbrăcă. Fiecare muchie  $(u, v)$  înseamnă că articolul  $u$  trebuie îmbrăcat înaintea articolului  $v$ . Timpii de descoperire și de terminare dintr-o căutare în adâncime sunt prezențați alături de fiecare vârf. (b) Același graf sortat topologic. Vâfurile lui sunt aranjate de la stânga la dreapta în ordinea descrescătoare a timpului de terminare. Se observă că toate muchiile orientate merg de la stânga la dreapta.

# Sortare topologică - Alt algoritm

Dacă  $\text{final}[u] = \text{momentul la care a fost finalizat vârful } u \text{ în parcurgerea DF}$ , avem:  $uv \in E \Rightarrow f[u] > f[v]$

- atunci sortarea topologică = sortare descrescătoare în raport cu **final**

## Sortare\_Topologică(G)

1. apelează CA(G) pentru a calcula timpii de terminare  $f[v]$  pentru fiecare vârf  $v$
2. pe măsură ce fiecare vârf este terminat, inserează în capul unei liste înlănțuite
3. returnează lista înlănțuită de vârfuri



# Construcția de grafuri cu secvență gradelor dată



# Secvențe de grade



Dată o formulă chimică, există un compus chimic care are această formulă?

Dar unul aciclic?

Ce structuri poate avea un astfel de compus?

- $C_mH_n$  - poate există moleculă **aciclică** cu această formulă?

# Secvențe de grade



Din studii empirice, chestionare, analize  $\Rightarrow$  informații despre numărul de interacțiuni ale unui nod.

Este realizabilă o rețea de legături între noduri care să respecte numărul de legături?

Dacă da, să se construiască un model de rețea.

# Secvențe de grade



Din studii empirice, chestionare, analize  $\Rightarrow$  informații despre numărul de interacțiuni ale unui nod.

Este realizabilă o rețea de legături între noduri care să respecte numărul de legături?

Dacă da, să se construiască un model de rețea.

**Exemplu:** Într-o grupă de studenți, fiecare student este întrebat cu câți colegi a colaborat în timpul anilor de studii. Este realizabilă o rețea de colaborări care să corespundă răspunsurilor lor (sau este posibil ca informațiile adunate să fie incorecte)?

- Studentul 1 - cu 3
- Studentul 2 - cu 3
- Studentul 3 - cu 2
- Studentul 4 - cu 3
- Studentul 5 - cu 2

# Secvențe de grade



Dată o secvență de numere  $s$ , se poate construi un graf neorientat având secvența gradelor  $s$ ?

Dar un multigraf neorientat?

Dar un arbore?

- Condiții necesare
- Condiții suficiente

# Secvențe de grade

## Construcția de grafuri cu secvența gradelor dată

### Aplicații:

- chimie** – studiul structurii posibile a unor compuși cu formula chimică dată
- proiectare de rețele**
- biologie** – rețelele metabolice, de interacțiuni între gene/proteine
- studii epidemiologice** – în care, prin chestionare anonime, persoanele declară numărul de persoane cu care au interacționat
- studii bazate pe simulări de rețele**

# Construcția de grafuri neorientate cu secvența gradelor dată

# Algoritmul Havel-Hakimi

# Construcția de grafuri cu secvență gradelor dată

## Problemă

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de numere naturale.

Să se construiască, dacă se poate, un graf neorientat  $G$  cu  $s(G) = s_0$ .

# Construcția de grafuri cu secvență gradelor dată

## Problemă

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de numere naturale.

Să se construiască, dacă se poate, un graf neorientat  $G$  cu  $s(G) = s_0$ .

**Condiții necesare pentru existența lui  $G$ :**

- $d_1 + \dots + d_n$  - număr par
- $d_i \leq n - 1, \forall i$

# Construcția de grafuri cu secvență gradelor dată

## Problemă

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de numere naturale.

Să se construiască, dacă se poate, un graf neorientat  $G$  cu  $s(G) = s_0$ .

**Condiții necesare pentru existența lui  $G$ :**

- $d_1 + \dots + d_n$  - număr par
- $d_i \leq n - 1, \forall i$



Pentru  $s_0 = \{3,3,1,1\}$  - nu există  $G$   
**⇒ condițiile nu sunt suficiente**  
Totuși, putem crea un multigraf

# Construcția de grafuri cu secvență gradelor dată



Idee de algoritm de construcție a unui graf  $G$  cu  $s(G) = s_0$

1. Începem construcția de la vârful cu gradul cel mai mare
2. îi alegem ca vecini vârfurile cu gradele cele mai mari

# Construcția de grafuri cu secvență gradelor dată

## Exemplu

$$s_0 = \{3, 4, 2, 1, 3, 4, 2, 1\}$$

etichete vârfuri

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8$

## Pasul 1

- construim muchii pentru vârful de gradul maxim
- alegem ca vecini următoarele vârfuri cu cele mai mari grade

# Construcția de grafuri cu secvența gradelor dată



Idee de algoritm de construcție a unui graf  $G$  cu  $s(G) = s_0$

1. începem construcția de la vârful cu gradul cel mai mare
2. îi alegem ca vecini vârfurile cu gradele cele mai mari
3. actualizăm secvența  $s_0$  și reluăm până când
  - secvența conține doar 0  $\Rightarrow G$
  - secvența conține numere negative  $\Rightarrow$

# Construcția de grafuri cu secvența gradelor dată

Idee de algoritm de construcție a unui graf  $G$  cu  $s(G) = s_0$

1. începem construcția de la vârful cu gradul cel mai mare
2. îi alegem ca vecini vârfurile cu gradele cele mai mari
3. actualizăm secvența  $s_0$  și reluăm până când
  - secvența conține doar 0  $\Rightarrow G$
  - secvența conține numere negative  $\Rightarrow G$  nu se poate construi prin acest procedeu



Se poate construi  $G$  altfel?

# Construcția de grafuri cu secvența gradelor dată

Idee de algoritm de construcție a unui graf  $G$  cu  $s(G) = s_0$

1. începem construcția de la vârful cu gradul cel mai mare
2. îi alegem ca vecini vârfurile cu gradele cele mai mari
3. actualizăm secvența  $s_0$  și reluăm până când
  - secvența conține doar 0  $\Rightarrow G$
  - secvența conține numere negative  $\Rightarrow G$  nu se poate construi prin acest procedeu



**Teorema Havel-Hakimi  $\Rightarrow$  NU**

$\Rightarrow$  Algoritmul anterior = **Algoritmul Havel-Hakimi**

# Exemplu algoritm Havel-Hakimi

$$s_0 = \{3, 4, 2, 1, 3, 4, 2, 1\}$$

etichete vârfuri

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8$

## Pasul 1

- construim muchii pentru vârful de gradul maxim =  $x_2$
- alegem ca vecini următoarele vârfuri cu cele mai mari grade

# Exemplu algoritm Havel-Hakimi

$$s_0 = \{3, 4, 2, 1, 3, 4, 2, 1\}$$

etichete vârfuri

$$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8$$

## Pasul 1

- construim muchii pentru vârful de gradul maxim =  $x_2$
- alegem ca vecini următoarele vârfuri cu cele mai mari grade  
⇒ ar fi utilă sortarea descrescătoare a elementelor lui  $s_0$

$$s_0 = \{4, 4, 3, 3, 2, 2, 1, 1\}$$

etichete vârfuri

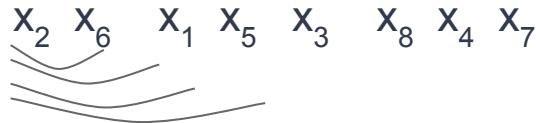
$$x_2 \ x_6 \ x_1 \ x_5 \ x_3 \ x_8 \ x_4 \ x_7$$

# Exemplu algoritm Havel-Hakimi

Pasul 1

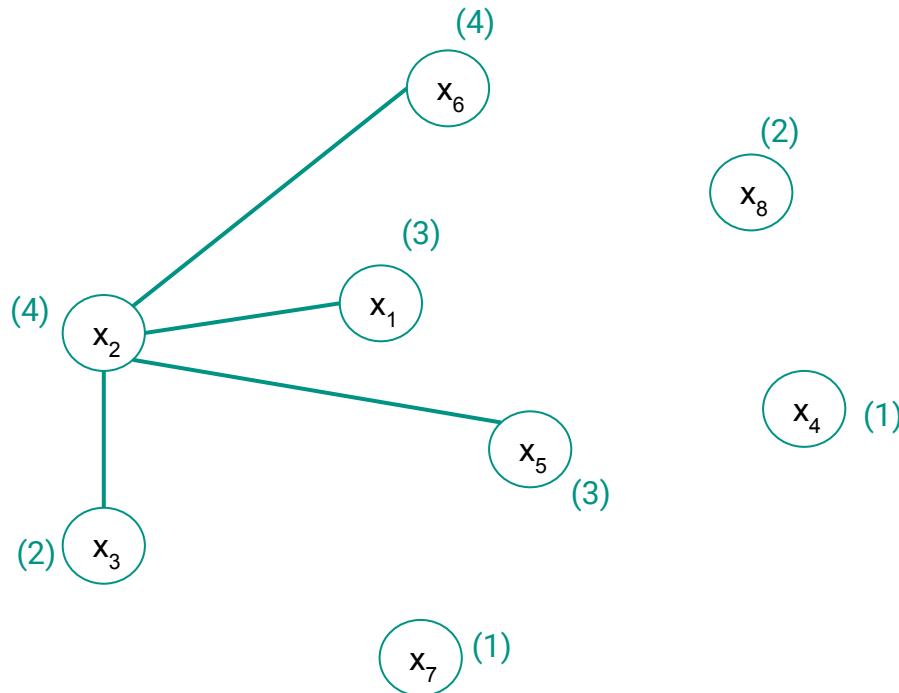
$$s_0 = \{ 4, 4, 3, 3, 2, 2, 1, 1 \}$$

etichete vârfuri



**Muchii construite:**  $x_2x_6, x_2x_1, x_2x_5, x_2x_3$

# Exemplu algoritm Havel-Hakimi

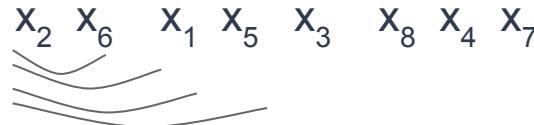


# Exemplu algoritm Havel-Hakimi

Pasul 1

$$s_0 = \{ 4, 4, 3, 3, 2, 2, 1, 1 \}$$

etichete vârfuri



Muchii construite:  $x_2x_6, x_2x_1, x_2x_5, x_2x_3$

Secvența rămasă:

$$s'_0 = \{ \textcolor{red}{3}, \textcolor{red}{2}, \textcolor{red}{2}, \textcolor{red}{1}, 2, 1, 1 \}$$

etichete vârfuri

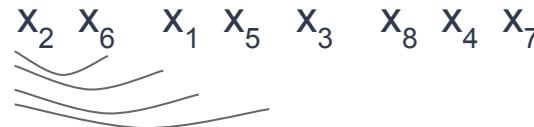


# Exemplu algoritm Havel-Hakimi

Pasul 1

$$s_0 = \{ 4, 4, 3, 3, 2, 2, 1, 1 \}$$

etichete vârfuri



Muchii construite:  $x_2x_6, x_2x_1, x_2x_5, x_2x_3$

Secvența rămasă:

$$s'_0 = \{ 3, 2, 2, 1, 2, 1, 1 \}$$

etichete vârfuri



Secvența rămasă ordonată descrescător:

$$s'_0 = \{ 3, 2, 2, 2, 1, 1, 1 \}$$

etichete vârfuri

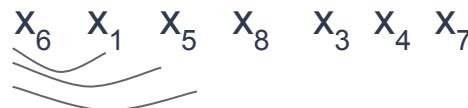


# Exemplu algoritm Havel-Hakimi

Pasul 2

$$s'_0 = \{ \quad \textcolor{red}{3}, \quad \textcolor{red}{2}, \quad \textcolor{red}{2}, \quad \textcolor{red}{2}, \quad 1, \quad 1, \quad 1 \}$$

etichete vârfuri



Muchii construite:  $x_6x_1, x_6x_5, x_6x_8$

Secvența rămasă:

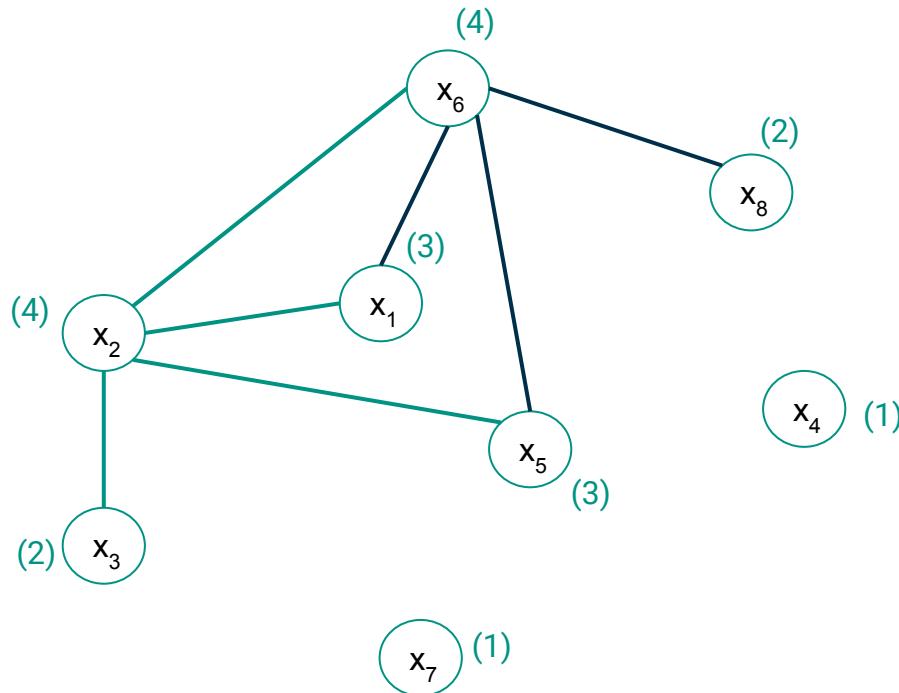
$$s''_0 = \{ \quad \textcolor{red}{1}, \quad \textcolor{red}{1}, \quad \textcolor{red}{1}, \quad 1, \quad 1, \quad 1 \}$$

etichete vârfuri



(este ordonată descrescător)

# Exemplu algoritm Havel-Hakimi

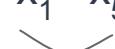


# Exemplu algoritm Havel-Hakimi

Pasul 3

$$s''_0 = \{ 1, 1, 1, 1, 1, 1 \}$$

etichete vârfuri

$x_1 \quad x_5 \quad x_8 \quad x_3 \quad x_4 \quad x_7$   


Muchii construite:  $x_1x_5$

Secvența rămasă:

$$s'''_0 = \{ 0, 1, 1, 1, 1 \}$$

etichete vârfuri

$x_5 \quad x_8 \quad x_3 \quad x_4 \quad x_7$

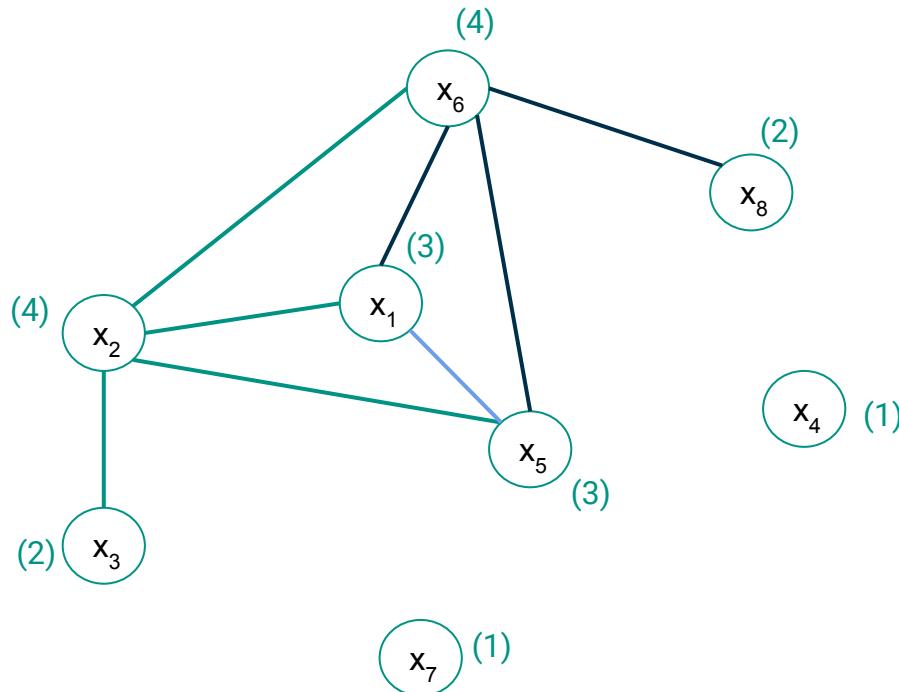
Secvența rămasă ordonată descrescător:

$$s'''_0 = \{ 1, 1, 1, 1, 0 \}$$

etichete vârfuri

$x_7 \quad x_3 \quad x_4 \quad x_8 \quad x_5$

# Exemplu algoritm Havel-Hakimi



# Exemplu algoritm Havel-Hakimi

Pasul 4

$$s'''_0 = \{1, 1, 1, 1, 0\}$$

etichete vârfuri

$$\begin{matrix} x_7 & x_3 & x_4 & x_8 & x_5 \\ \underbrace{\phantom{x_7}} & \phantom{x_3} & \phantom{x_4} & \phantom{x_8} & \phantom{x_5} \end{matrix}$$

Muchii construite:  $x_7x_3$

Secvența rămasă:

$$s''''_0 = \{0, 1, 1, 0\}$$

etichete vârfuri

$$\begin{matrix} x_3 & x_4 & x_8 & x_5 \end{matrix}$$

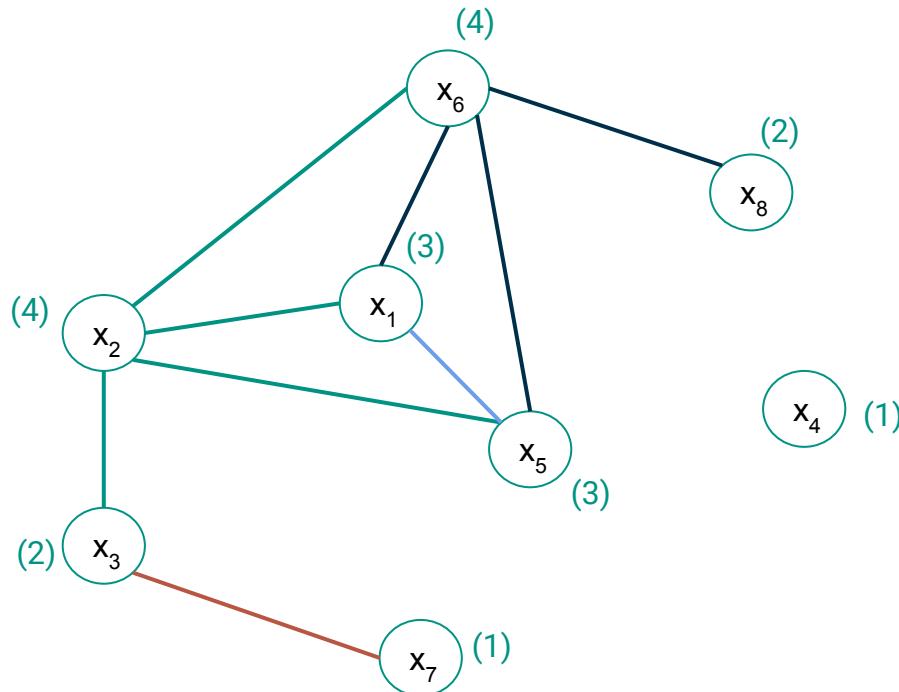
Secvența rămasă ordonată descrescător:

$$s''''_0 = \{1, 1, 0, 0\}$$

etichete vârfuri

$$\begin{matrix} x_4 & x_8 & x_3 & x_5 \end{matrix}$$

# Exemplu algoritm Havel-Hakimi

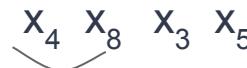


# Exemplu algoritm Havel-Hakimi

Pasul 5

$$s''''_0 = \{1, 1, 0, 0\}$$

etichete vârfuri



Muchii construite:  $x_4x_8$

Secvența rămasă:

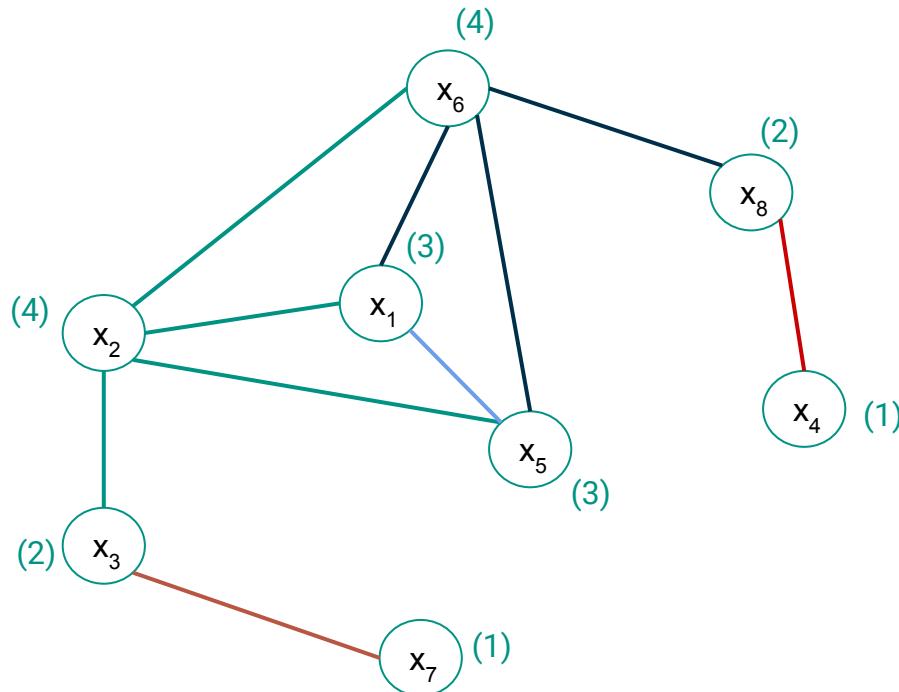
$$s''''_0 = \{0, 0, 0\}$$

etichete vârfuri

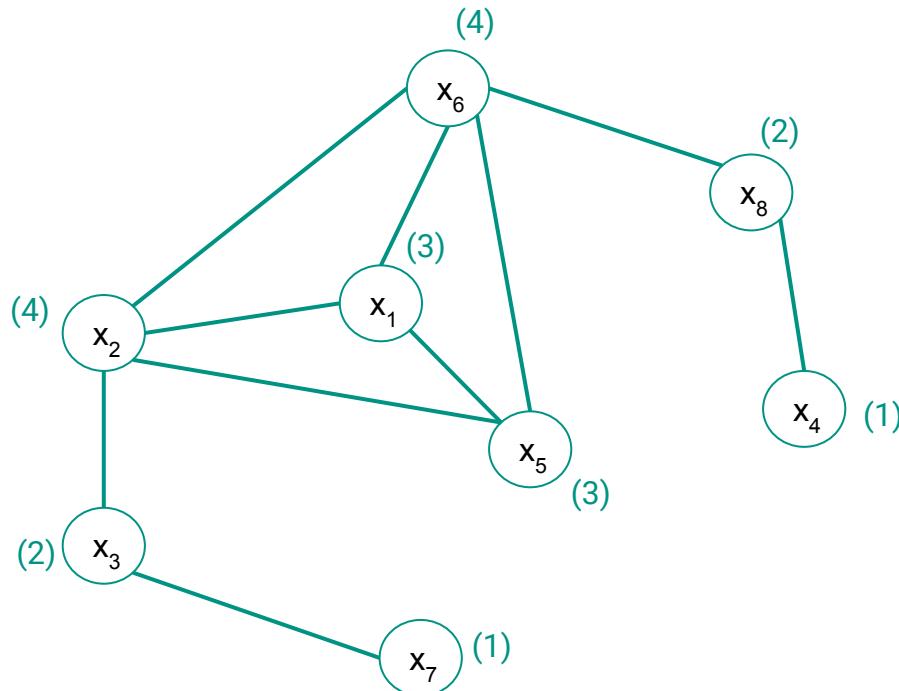
$x_8 \ x_3 \ x_5$

**STOP**

# Exemplu algoritm Havel-Hakimi



# Exemplu algoritm Havel-Hakimi



# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci  
scrive NU, STOP

# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci  
scrie NU, STOP
2. Cât timp  $s_0$  conține valori nenule, execută  
alege  $d_k$  cel mai mare număr din secvența  $s_0$   
elimină  $d_k$  din  $s_0$   
fie  $d_{i1}, \dots, d_{ik}$  cele mai mari  $d_k$  numere din  $s_0$

# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci  
scrie NU, STOP
2. Cât timp  $s_0$  conține valori nenule, execută  
alege  $d_k$  cel mai mare număr din secvența  $s_0$   
elimină  $d_k$  din  $s_0$   
fie  $d_{i_1}, \dots, d_{i_{dk}}$  cele mai mari  $d_k$  numere din  $s_0$   
pentru  $j \in \{i_1, \dots, i_{dk}\}$  execută

# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci scrie NU, STOP
2. Cât timp  $s_0$  conține valori nenule, execută
  - alege  $d_k$  cel mai mare număr din secvența  $s_0$
  - elimină  $d_k$  din  $s_0$
  - fie  $d_{i_1}, \dots, d_{i_{dk}}$  cele mai mari  $d_k$  numere din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{dk}\}$  execută
    - adaugă muchia  $x_k x_j$  la  $G$
    - înlocuiește  $d_j$  în secvența  $s_0$  cu  $d_j - 1$
    - dacă  $d_j - 1 < 0$ , atunci scrie NU, STOP

# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci scrie NU, STOP
2. Cât timp  $s_0$  conține valori nenule, execută
  - alege  $d_k$  cel mai mare număr din secvența  $s_0$
  - elimină  $d_k$  din  $s_0$
  - fie  $d_{i_1}, \dots, d_{i_{dk}}$  cele mai mari  $d_k$  numere din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{dk}\}$  execută
    - adaugă muchia  $x_k x_j$  la  $G$
    - înlocuiește  $d_j$  în secvența  $s_0$  cu  $d_j - 1$
    - dacă  $d_j - 1 < 0$ , atunci scrie NU, STOP

**Observație:** Pentru a determina ușor care este cel mai mare număr din secvență și care sunt cele mai mari valori care îi urmează, este util ca pe parcursul algoritmului secvența  $s_0$  să fie ordonată descrescător.

# Algoritm Havel-Hakimi

1. Dacă  $d_1 + \dots + d_n$  este impar sau există în  $s_0$  un  $d_i > n-1$ , atunci scrie NU, STOP
2. Cât timp  $s_0$  conține valori nenule, execută
  - alege  $d_k$  cel mai mare număr din secvența  $s_0$
  - elimină  $d_k$  din  $s_0$
  - fie  $d_{i_1}, \dots, d_{i_{dk}}$  cele mai mari  $d_k$  numere din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{dk}\}$  execută
    - adaugă muchia  $x_k x_j$  la  $G$
    - înlocuiește  $d_j$  în secvența  $s_0$  cu  $d_j - 1$
    - dacă  $d_j - 1 < 0$ , atunci scrie NU, STOP

**Observație:** Pentru a determina ușor care este cel mai mare număr din secvență și care sunt cele mai mari valori care îi urmează, este util ca pe parcursul algoritmului secvența  $s_0$  să fie ordonată descrescător.

Complexitate?

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi

O secvență de  $n \geq 2$  numere naturale

$$s_0 = \{d_1 \geq \dots \geq d_n\}$$

cu  $d_1 \leq n-1$  este secvența gradelor unui graf neorientat (cu  $n$  vârfuri)  $\Leftrightarrow$

secvența

$$s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

este secvența gradelor unui graf neorientat (cu  $n-1$  vârfuri).

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi

O secvență de  $n \geq 2$  numere naturale

$$s_0 = \{d_1 \geq \dots \geq d_n\}$$

cu  $d_1 \leq n-1$  este secvența gradelor unui graf neorientat (cu  $n$  vârfuri)  $\Leftrightarrow$

secvența

$$s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

este secvența gradelor unui graf neorientat (cu  $n-1$  vârfuri).

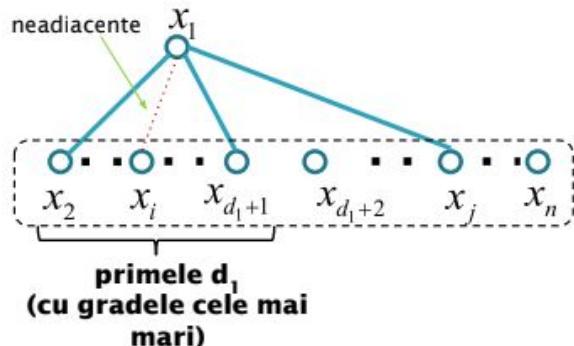
**Observatie:** Secvența  $s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$  se obține din  $s_0$  **eliminând primul element ( $d_1$ ) și scăzând 1 din primele  $d_1$  elemente rămase – acestea au indicii 2, 3, ...,  $d_1+1$ .**

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

$$s_0 = \{d_1 \geq \dots \geq d_n\} \quad \Rightarrow \quad s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

$$G, s(G) = s_0$$

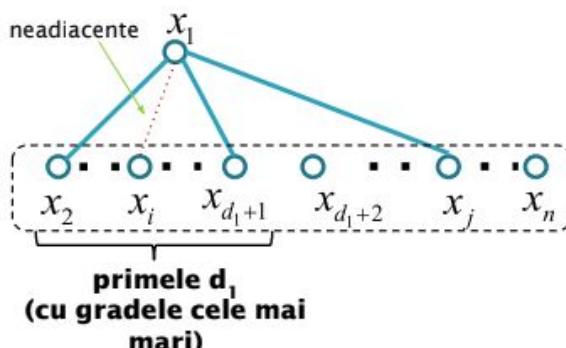


# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

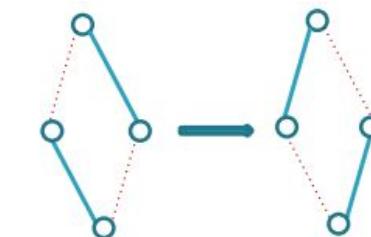
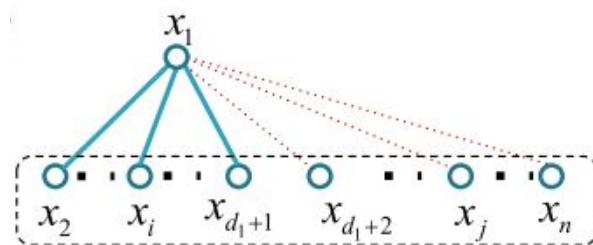
$$s_0 = \{d_1 \geq \dots \geq d_n\} \quad \Rightarrow \quad s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

$$G, s(G) = s_0$$



transformare  
t pe pătrat

$$G^*, s(G^*) = s_0$$
$$N_{G^*}(x_1) = \{x_2, \dots, x_{d_1+1}\}$$

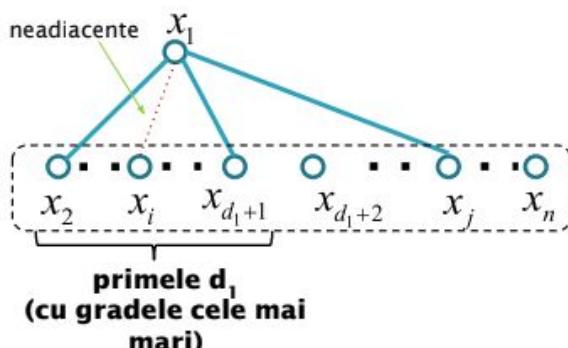


# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

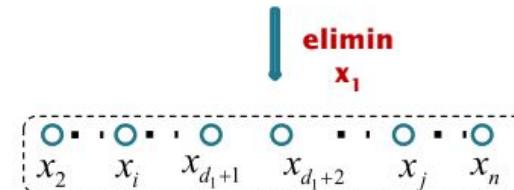
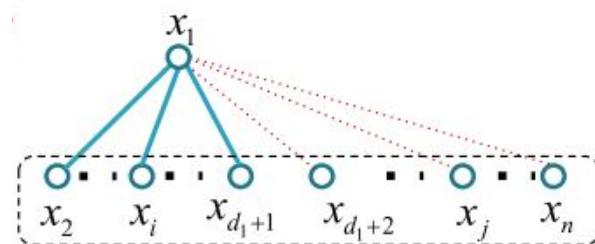
$$s_0 = \{d_1 \geq \dots \geq d_n\} \Rightarrow s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

$$G, s(G) = s_0$$



transformare  
t pe pătrat

$$G^*, s(G^*) = s_0$$
$$N_{G^*}(x_1) = \{x_2, \dots, x_{d_1+1}\}$$



$$G' = G^* - x_1, s(G') = s'_0$$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

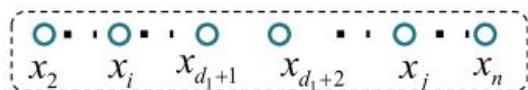
$$s_0 = \{d_1 \geq \dots \geq d_n\} \quad \Leftarrow \quad s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

$$s_0 = \{d_1 \geq \dots \geq d_n\} \quad \Leftarrow \quad s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

Fie  $G'$  cu  $s(G') = s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$



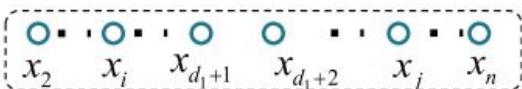
# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi - Demonstrație

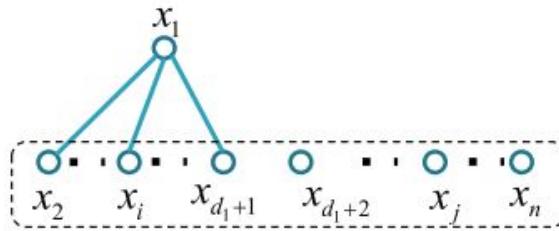
$$s_0 = \{d_1 \geq \dots \geq d_n\} \Leftarrow s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

Fie  $G'$  cu  $s(G') = s'_0$

$$\begin{aligned} G: V(G) &= V(G') \cup \{x_1\} \\ E(G) &= E(G') \cup \{x_1x_2, \dots, x_1x_{d_1+1}\} \end{aligned}$$



adăugăm un  
vârf  $x_1$  pe  
care îl unim  
cu  $x_2, \dots,$   
 $x_{d_1+1}$



Avem  $s(G) = s_0$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de  $n \geq 2$  numere naturale mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat.

Fie secvența obținută din  $s_0$  astfel:

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de  $n \geq 2$  numere naturale mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat.

Fie secvența obținută din  $s_0$  astfel:

eliminăm elementul  $d_i$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de  $n \geq 2$  numere naturale mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat.

Fie secvența obținută din  $s_0$  astfel:

eliminăm elementul  $d_i$

scădem o unitate din primele  $d_i$  componente, în ordine descrescătoare a secvenței rămase

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

Fie  $s_0 = \{d_1, \dots, d_n\}$  o secvență de  $n \geq 2$  numere naturale mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat.

Fie secvența obținută din  $s_0$  astfel:

eliminăm elementul  $d_i$

scădem o unitate din primele  $d_i$  componente, în ordine descrescătoare a secvenței rămase

## Are loc echivalența:

$s_0$  este secvența gradelor unui graf neorientat  $\Leftrightarrow$

$s_0^{(i)}$  este secvența gradelor unui graf neorientat

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi



Unde intervine în demonstrație faptul că  $d_1$  este maxim?

**Se poate renunța la această ipoteză  $\Rightarrow$**

## Extindere a teoremei Havel-Hakimi

**La un pas, vârful poate fi ales arbitrar (nu neapărat cel corespunzător elementului maxim).**

Se păstrează, însă, criteriul de alegere al vecinilor (cu gradele cele mai mari).

# Construcția de grafuri cu secvența gradelor dată

Cu ajutorul transformării  $t$  pe pătrat, putem obține, pornind de la un graf  $G$ , toate grafurile cu secvența gradelor  $s(G)$  (și mulțimea vârfurilor  $V(G)$ ).



# Construcția de grafuri cu secvența gradelor dată

Cu ajutorul transformării  $t$  pe pătrat, putem obține, pornind de la un graf  $G$ , toate grafurile cu secvența gradelor  $s(G)$  (și mulțimea vârfurilor  $V(G)$ ).

Mai exact, are loc următorul rezultat (exercițiu):

Fie  $G_1$  și  $G_2$  două grafuri neorientate cu mulțimea vârfurilor  $V = \{1, \dots, n\}$ .

Atunci  $s(G_1) = s(G_2) \Leftrightarrow$  există un sir de transformări  $t$  de interschimbare pe pătrat, prin care se poate obține graful  $G_2$  din  $G_1$ .



# Construcția de grafuri cu secvență gradelor dată

## Teorema Erdős-Gallai (suplimentar)

O secvență de  $n \geq 2$  numere naturale  $s_0 = \{d_1 \geq \dots \geq d_n\}$  este secvența gradelor unui graf neorientat  $\Leftrightarrow$

- $d_1 + \dots + d_n$  par și
- $d_1 + \dots + d_k \leq k(k-1) + \sum_{i=k+1}^n \min\{d_i, k\}, \forall 1 \leq k \leq n$

# Muchii critice



# Muchii critice

O muchie este **critică**  $\Leftrightarrow$  nu este conținută într-un ciclu.

Găsirea unui ciclu - parcurgere DF

- muchii de avansare** - ale arborelui DF (memorat cu un vector de tată), prin care se descoperă vârfuri noi
- muchii de întoarcere** - închid ciclu, nu pot fi critice

# Muchii critice



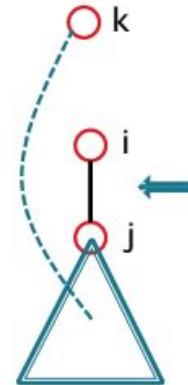
**Cum testăm dacă o muchie de avansare ( $i, j$ ) este critică?**

# Muchii critice



Cum testăm dacă o muchie de avansare  $(i, j)$  este critică?

- nu este conținută într-un ciclu închis de o muchie de întoarcere



# Muchii critice

O muchie de avansare  $(i, j)$  este critică

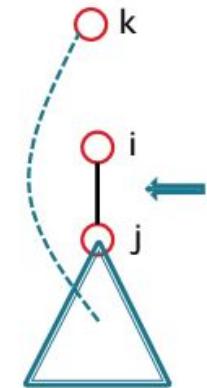
$\Leftrightarrow$

nu este conținută într-un ciclu închis de o muchie de întoarcere

$\Leftrightarrow$

nu există nicio muchie de întoarcere cu

- o extremitate în  $j$  sau într-un descendenter al lui  $j$
- cealaltă extremitate în  $i$  sau într-un ascendent al lui  $i$  (într-un vârf de pe un nivel mai mic sau egal cu nivelul lui  $i$ )



# Muchii critice

Memorăm, pentru fiecare vârf i:

**niv\_min[i]** = nivelul minim al unui vârf care este extremitate a unei muchii de înțoarcere din i sau dintr-un descendant al lui i

= nivelul minim la care se închide un ciclu elementar care conține vârful i (printr-o muchie de înțoarcere)



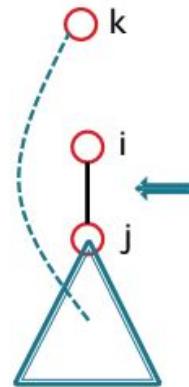
# Muchii critice

- **nivel[i]** = nivelul lui  $i$  în arborele DF
- **niv\_min[i]** =  $\min \{ \text{nivel}[i], A, B \}$ 
  - $A = \min \{ \text{nivel}[k] \mid ik \text{ muchie de întoarcere} \}$
  - $B = \min \{ \text{nivel}[k] \mid j \text{ descendent al lui } i, jk \text{ muchie de întoarcere} \}$



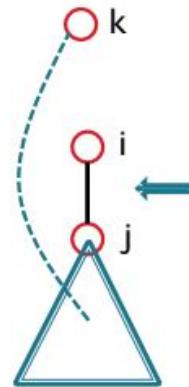
# Muchii critice

O muchie de avansare  $ij$  este critică



# Muchii critice

O muchie de avansare  $ij$  este critică  $\Leftrightarrow niv\_min[j] > nivel[i]$



# Muchii critice



Cum calculăm eficient `niv_min[i]`?

- $\text{niv\_min}[i] = \min \{ \text{nivel}[i], A, B \}$ 
  - $A = \min \{ \text{nivel}[k] \mid i \text{ k muchie de întoarcere} \}$
  - $B = \min \{ \text{nivel}[k] \mid j \text{ } \underline{\text{descendent}} \text{ al lui } i, j \text{ k muchie de întoarcere} \}$

# Muchii critice

Cum calculăm eficient `niv_min[i]`?

- $\text{niv\_min}[i] = \min \{ \text{nivel}[i], A, B \}$ 
  - $A = \min \{ \text{nivel}[k] \mid ik \text{ muchie de întoarcere} \}$
  - $B = \min \{ \text{nivel}[k] \mid j \text{ } \underline{\text{descendent}} \text{ al lui } i, jk \text{ muchie de întoarcere} \}$



**B se poate calcula recursiv**

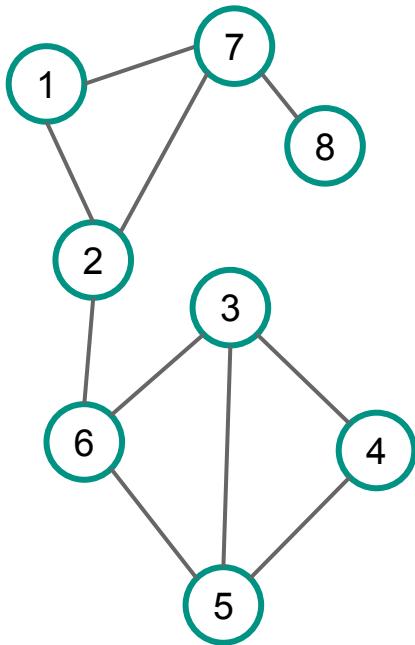
# Muchii critice

Cum calculăm eficient  $niv\_min[i]$ ?

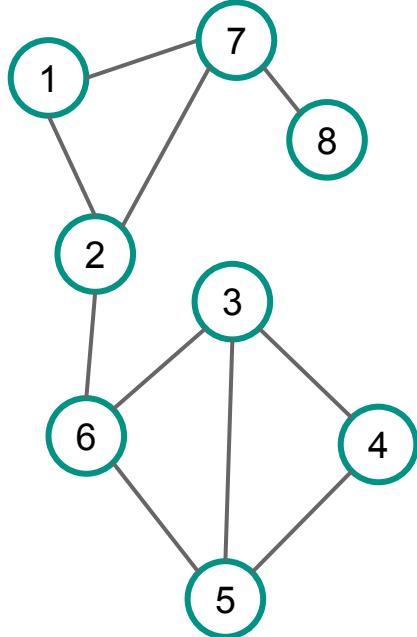
- $niv\_min[i] = \min \{ nivel[i], A, B \}$ 
  - $A = \min \{ nivel[k] \mid ik \text{ muchie de întoarcere} \}$
  - $B = \min \{ nivel[k] \mid j \text{ } \underline{\text{descendent}} \text{ al lui } i, jk \text{ muchie de întoarcere} \}$

**B = min { niv\_min[j] | j fiu al lui i }**

# Muchii critice - Exemplu



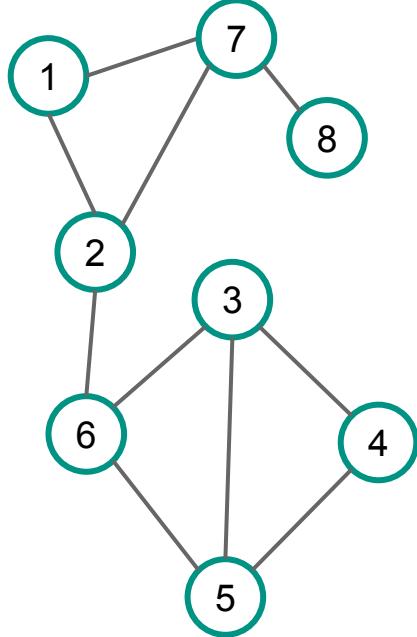
# Muchii critice - Exemplu



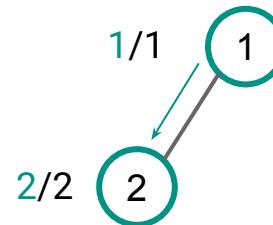
nivel/niv\_min

1/1  
1

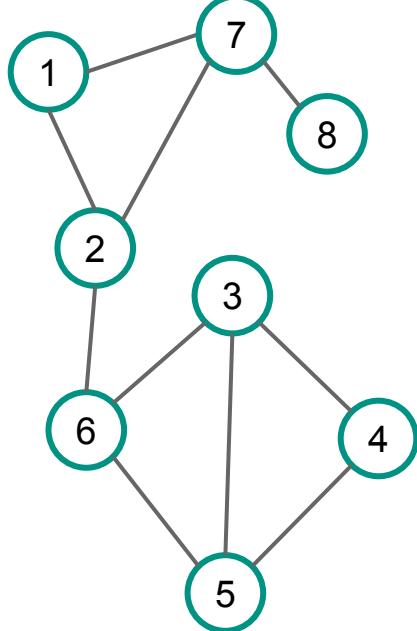
# Muchii critice - Exemplu



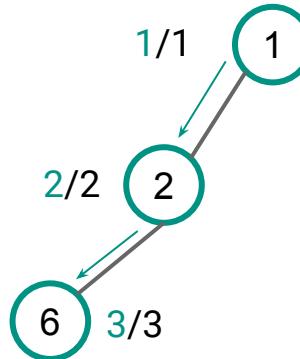
nivel/niv\_min



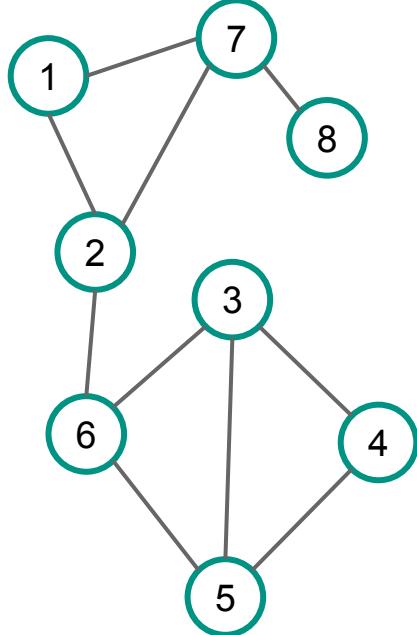
# Muchii critice - Exemplu



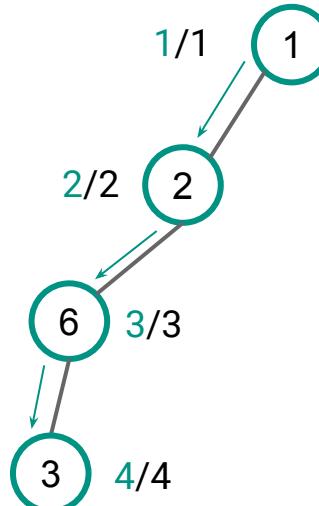
nivel/niv\_min



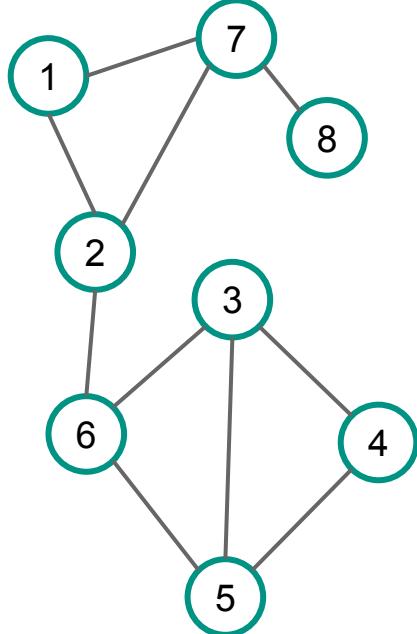
# Muchii critice - Exemplu



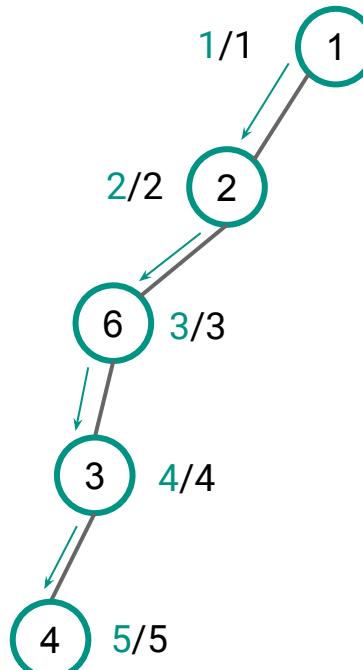
nivel/niv\_min



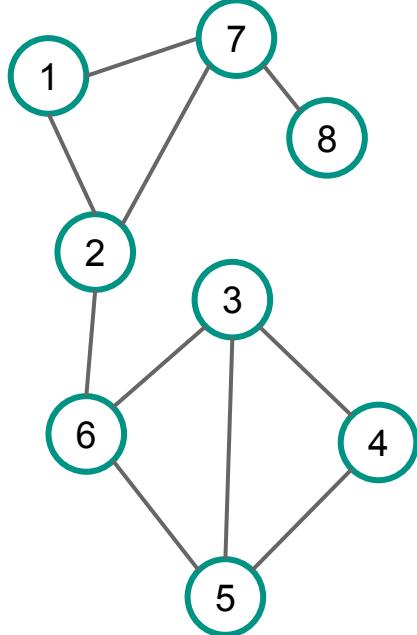
# Muchii critice - Exemplu



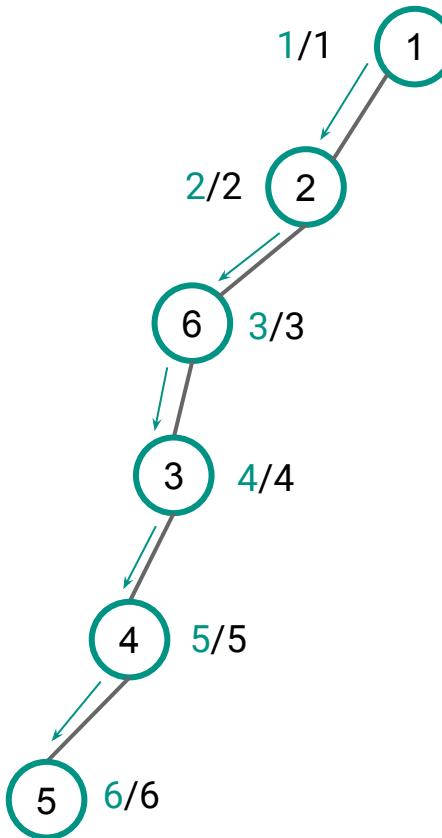
nivel/niv\_min



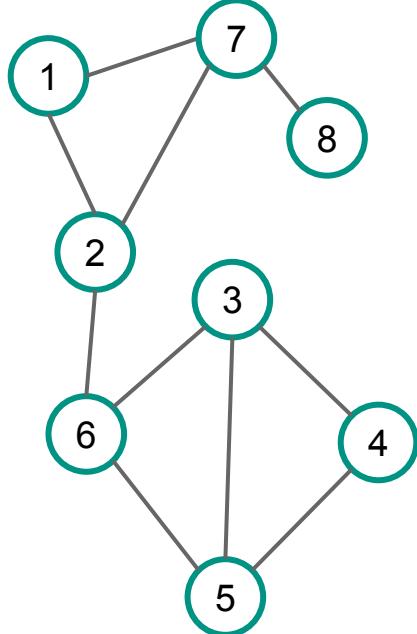
# Muchii critice - Exemplu



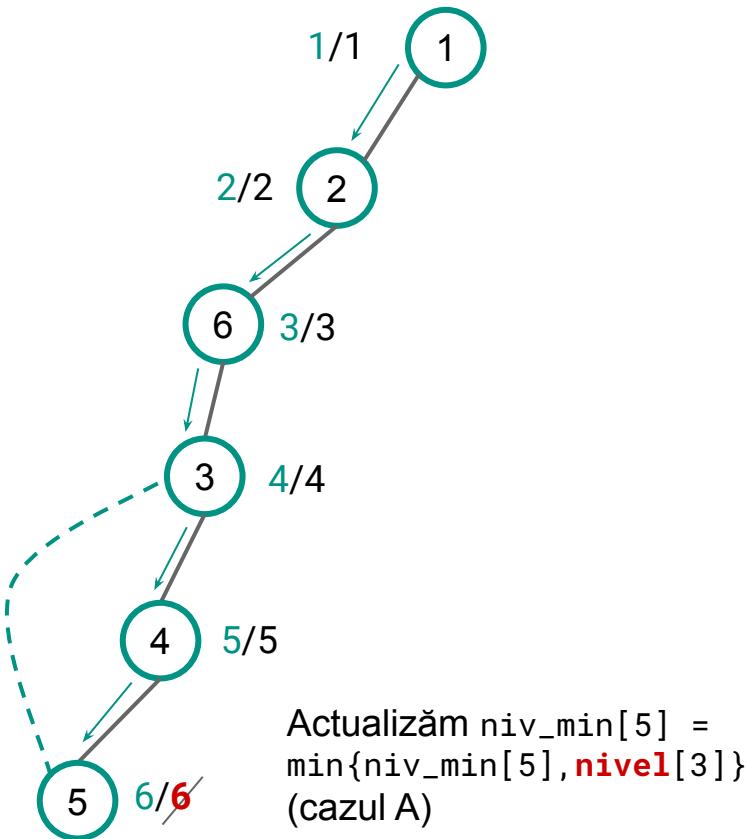
nivel/niv\_min



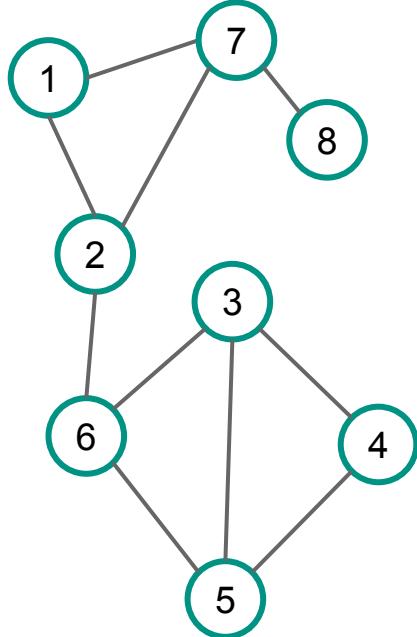
# Muchii critice - Exemplu



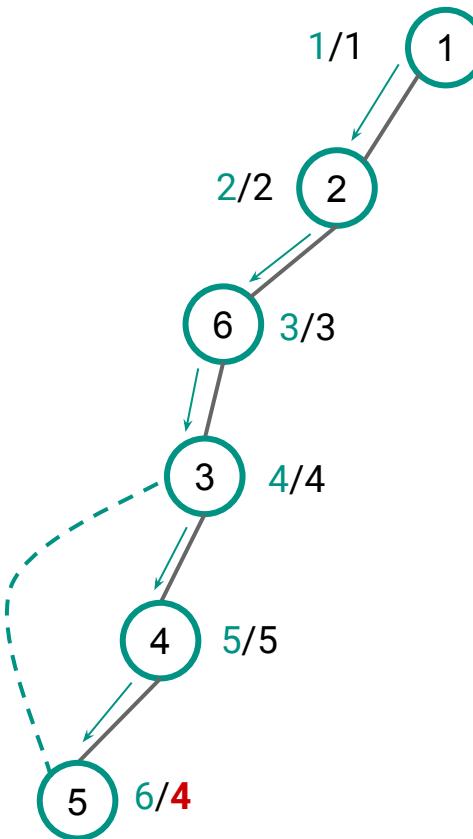
nivel/niv\_min



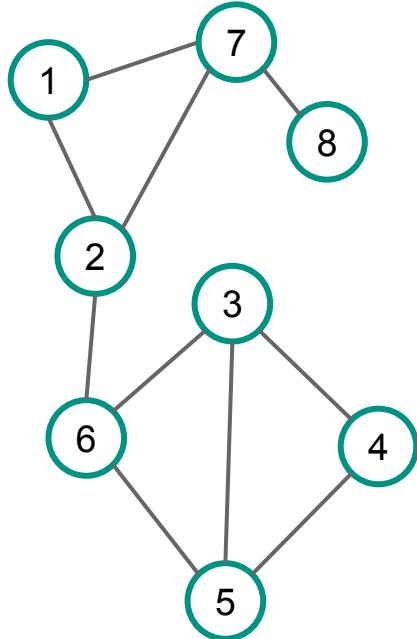
# Muchii critice - Exemplu



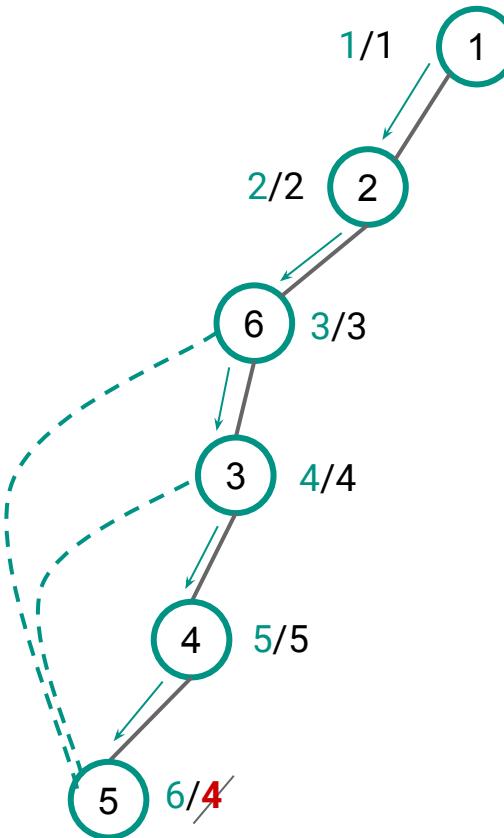
nivel/niv\_min



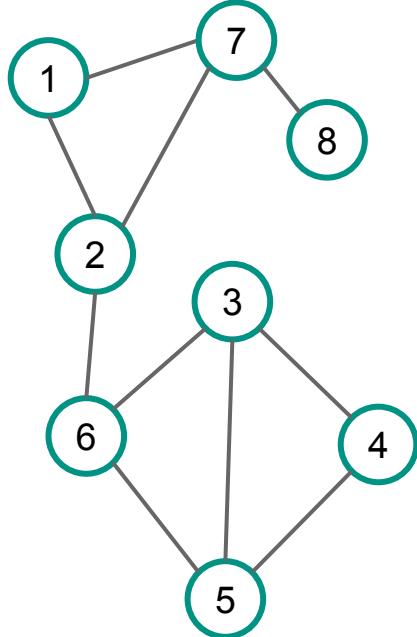
# Muchii critice - Exemplu



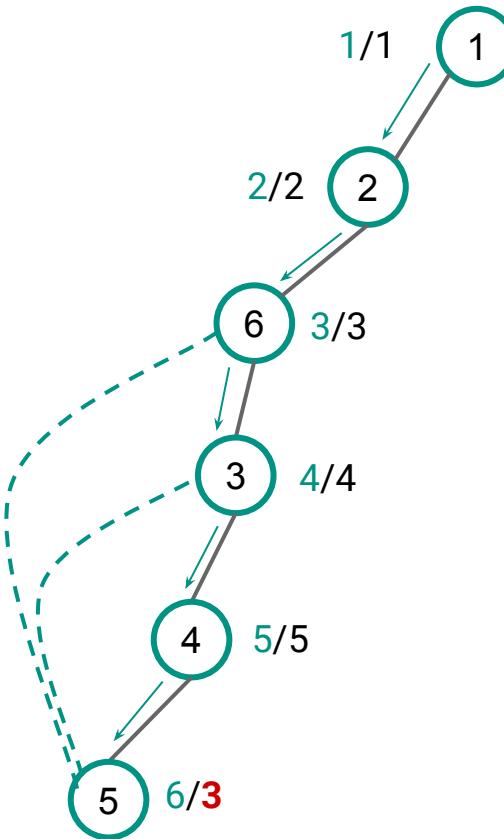
nivel/niv\_min



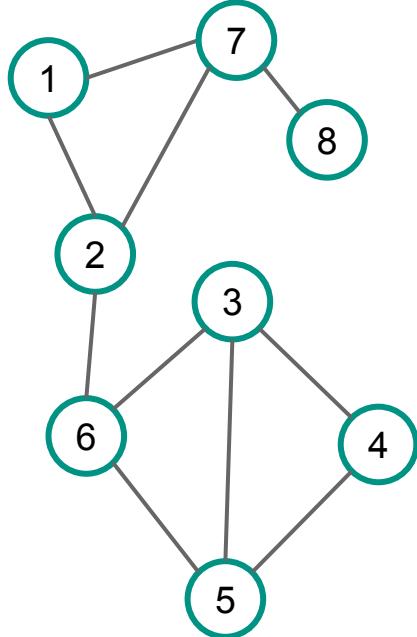
# Muchii critice - Exemplu



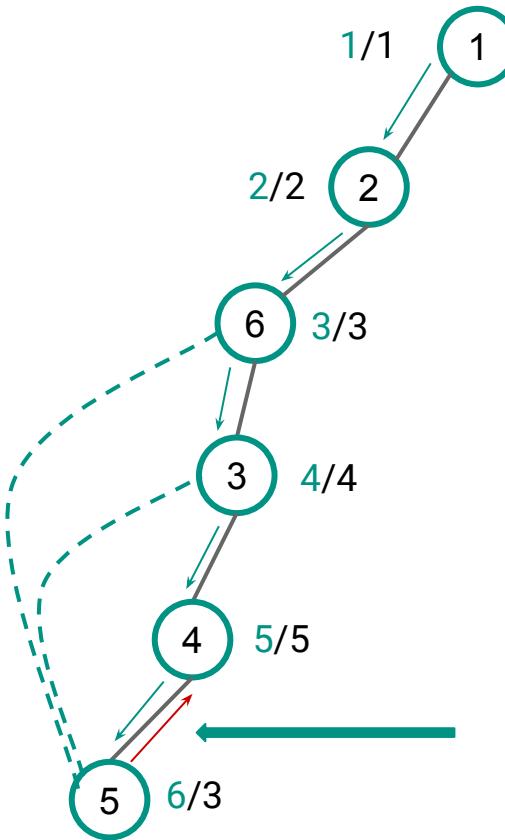
nivel/niv\_min



# Muchii critice - Exemplu

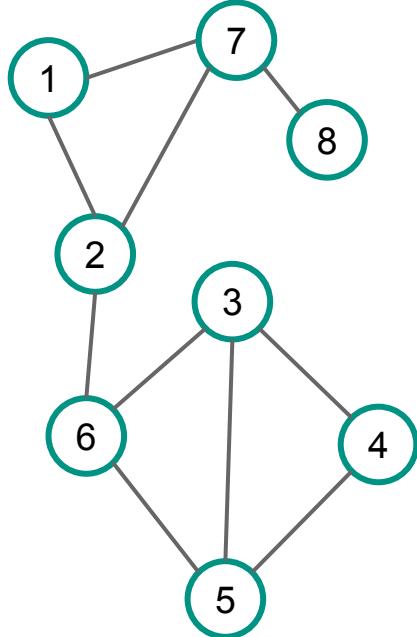


nivel/niv\_min

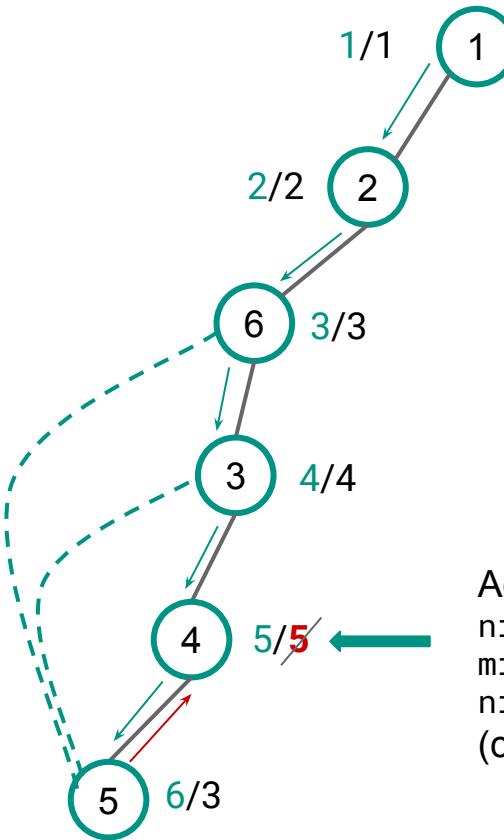


Test muchie critică:  
 $niv\_min[5] = 3 < niv[4] = 5$   
 $\Rightarrow \text{NU}$

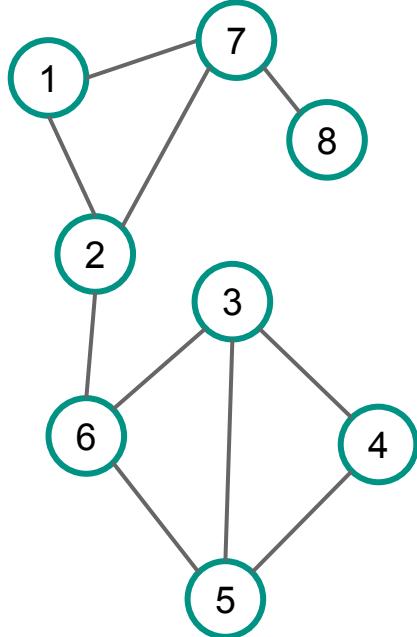
# Muchii critice - Exemplu



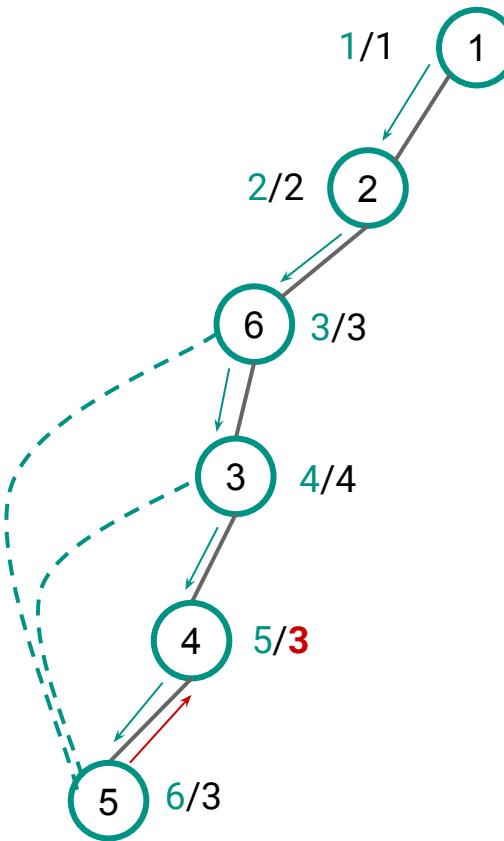
nivel/niv\_min



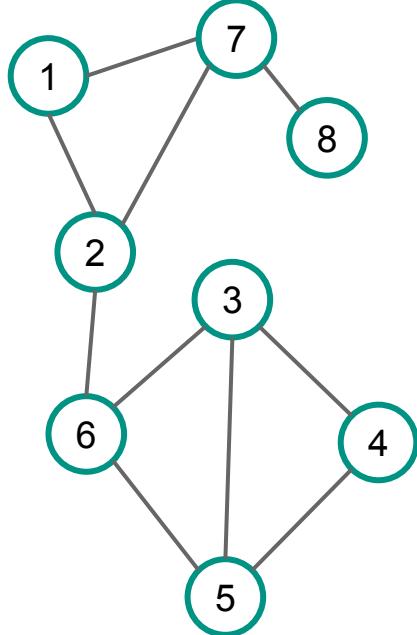
# Muchii critice - Exemplu



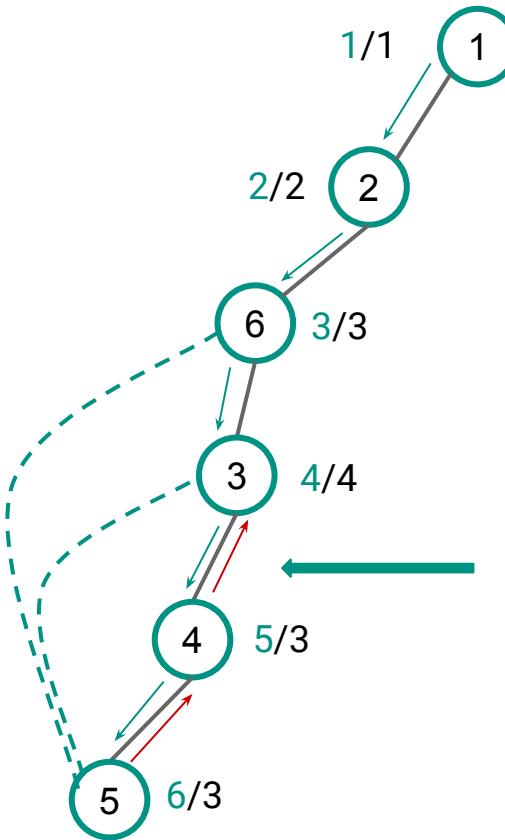
nivel/niv\_min



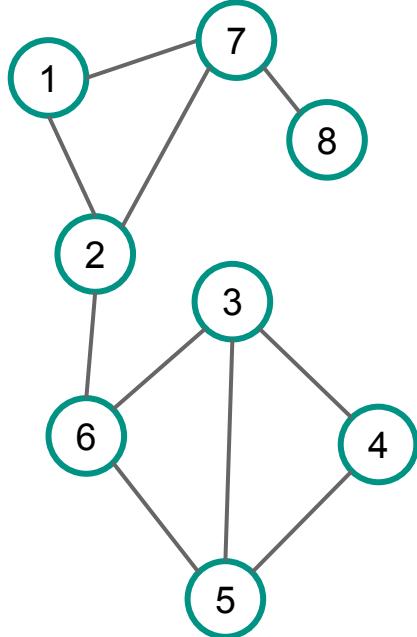
# Muchii critice - Exemplu



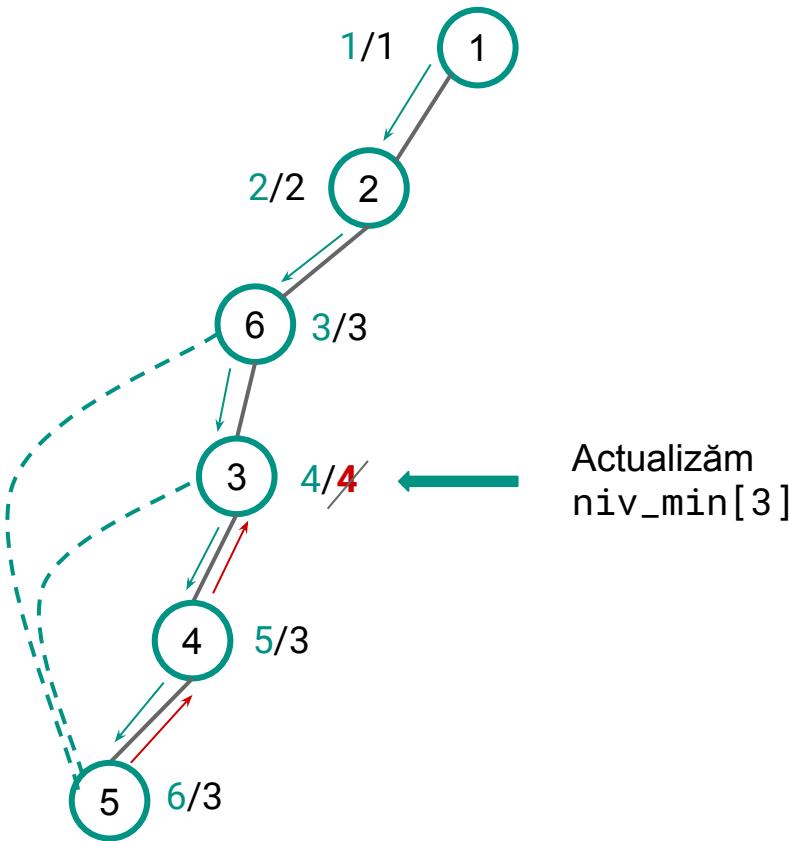
nivel/niv\_min



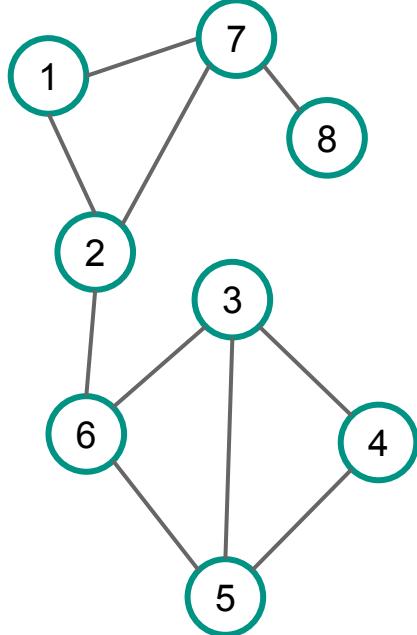
# Muchii critice - Exemplu



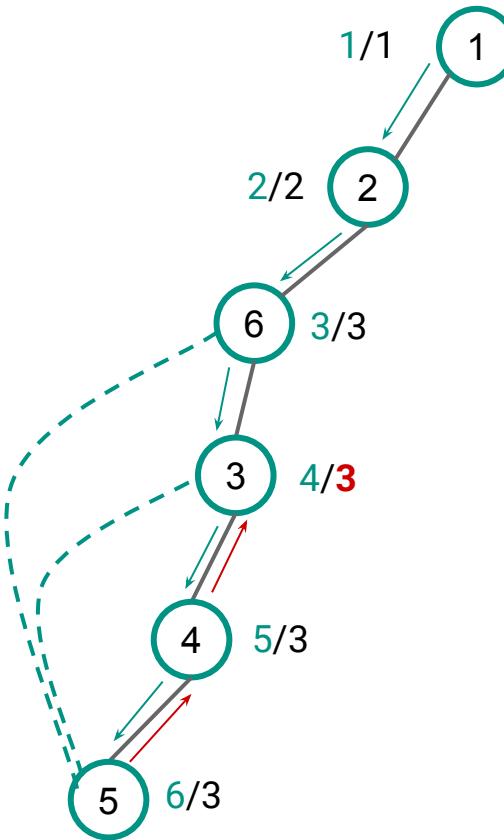
nivel/niv\_min



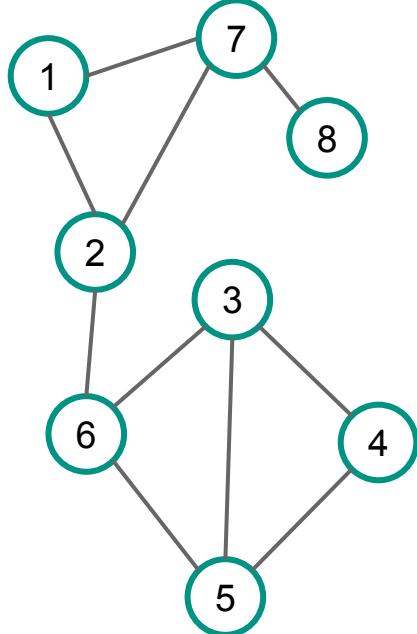
# Muchii critice - Exemplu



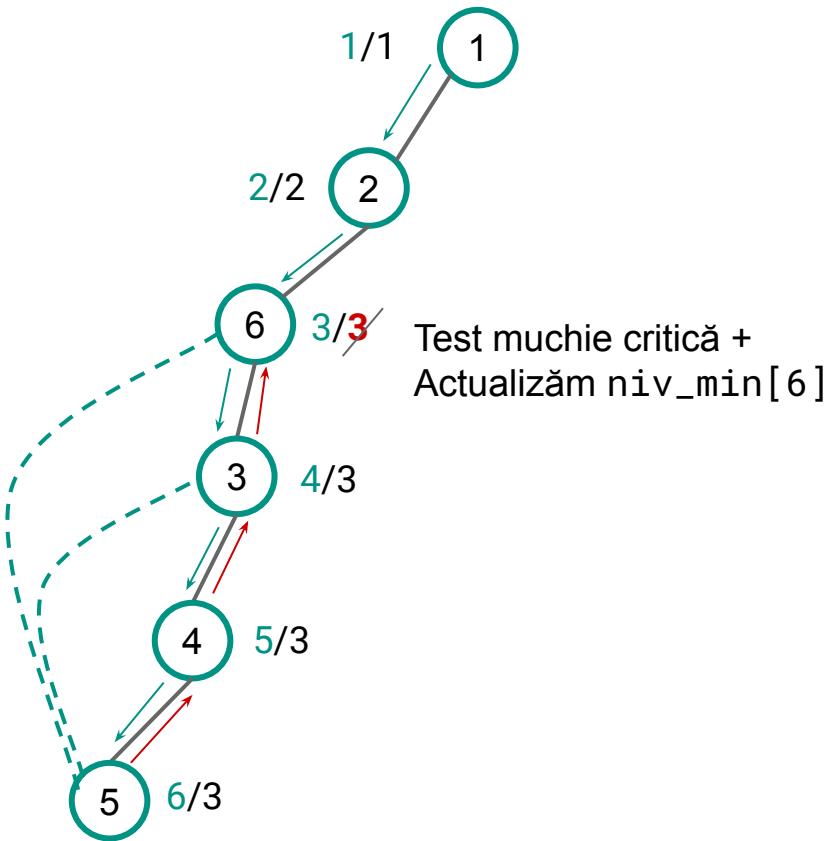
nivel/niv\_min



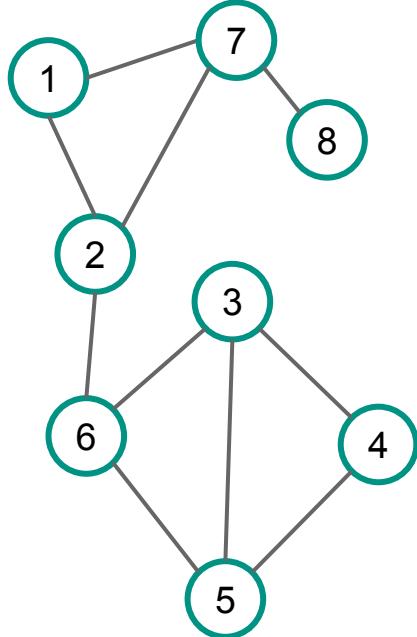
# Muchii critice - Exemplu



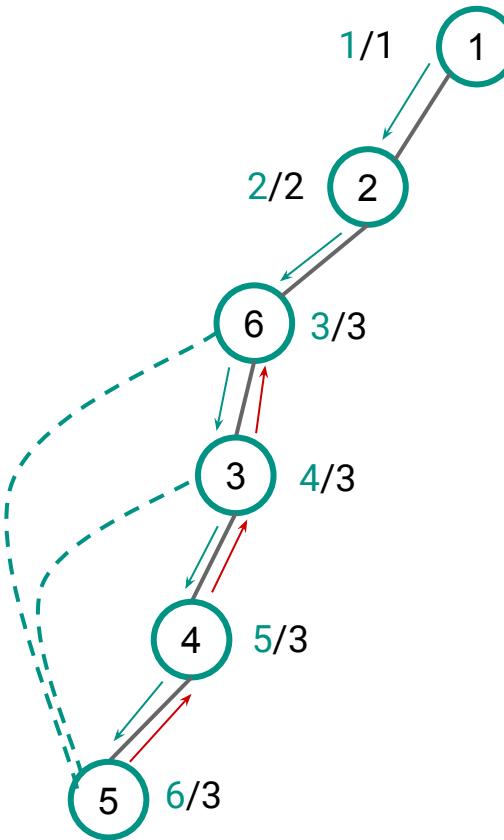
nivel/niv\_min



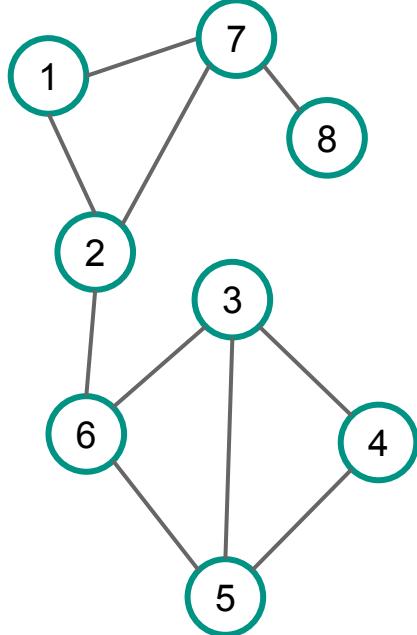
# Muchii critice - Exemplu



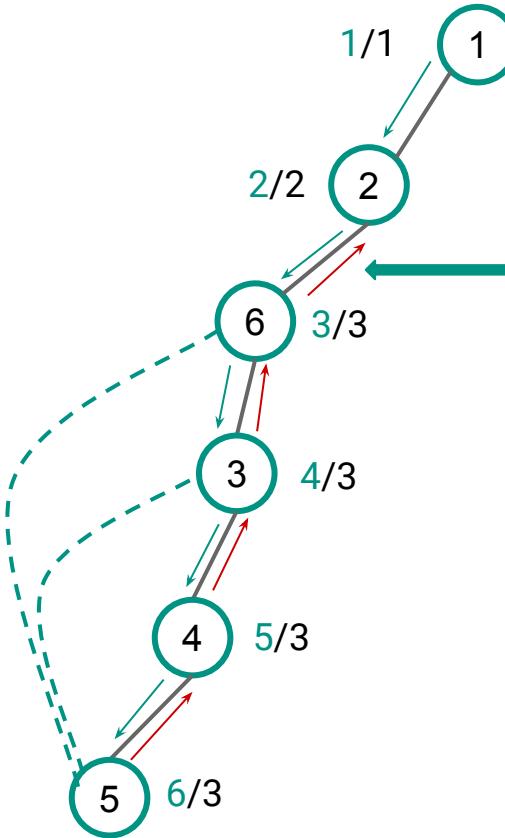
nivel/niv\_min



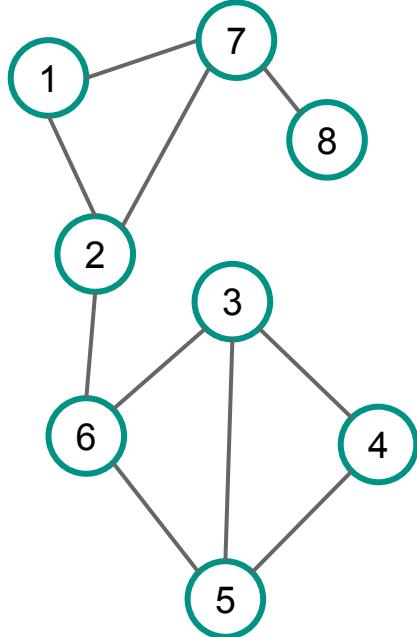
# Muchii critice - Exemplu



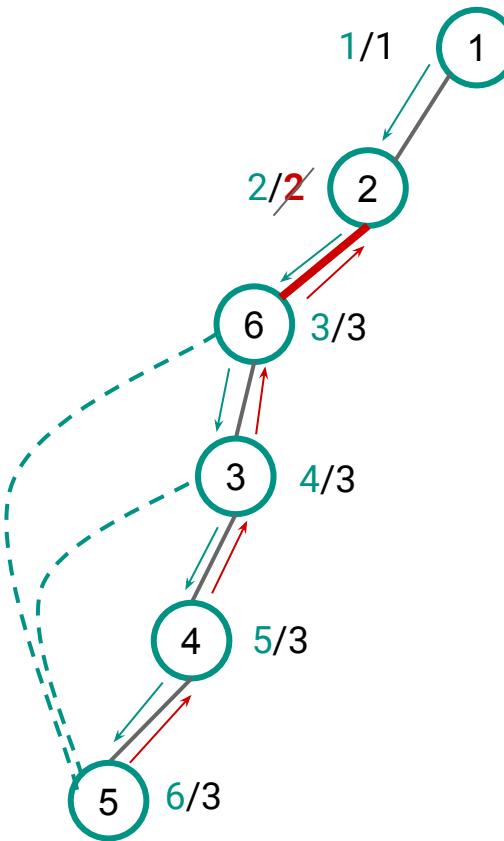
nivel/niv\_min



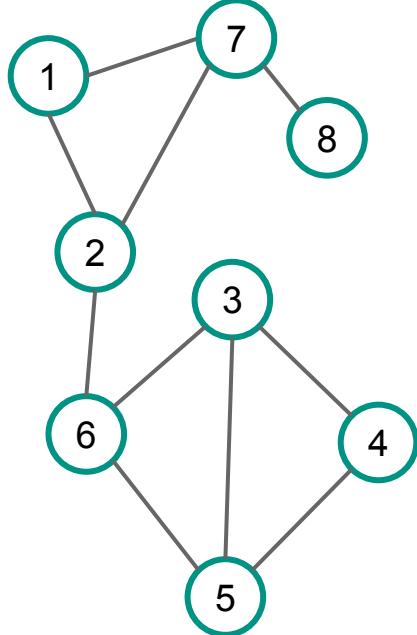
# Muchii critice - Exemplu



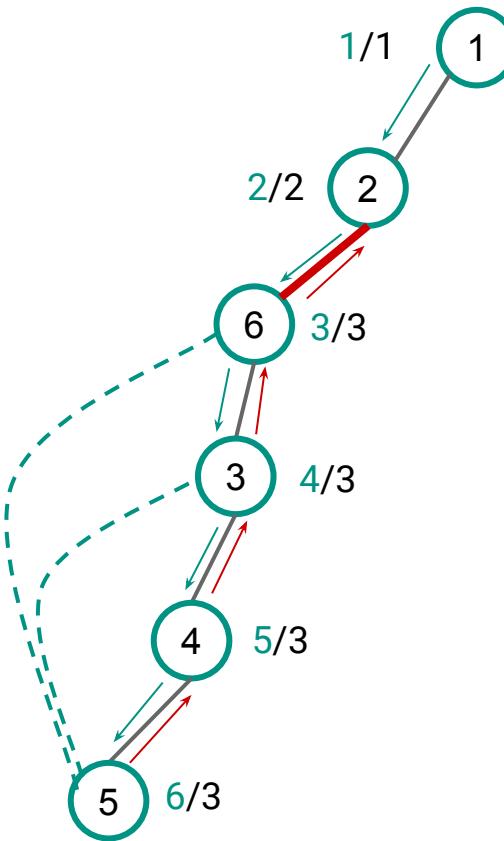
nivel/niv\_min



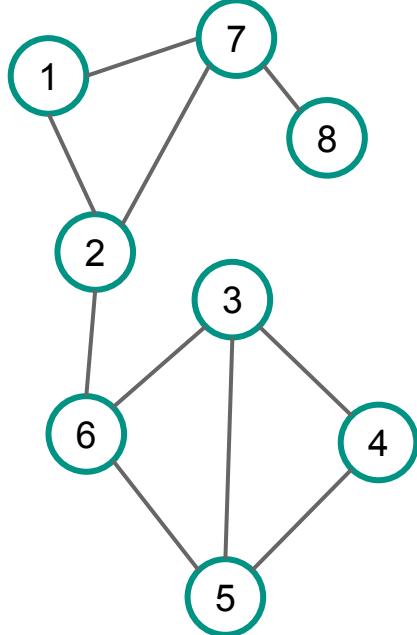
# Muchii critice - Exemplu



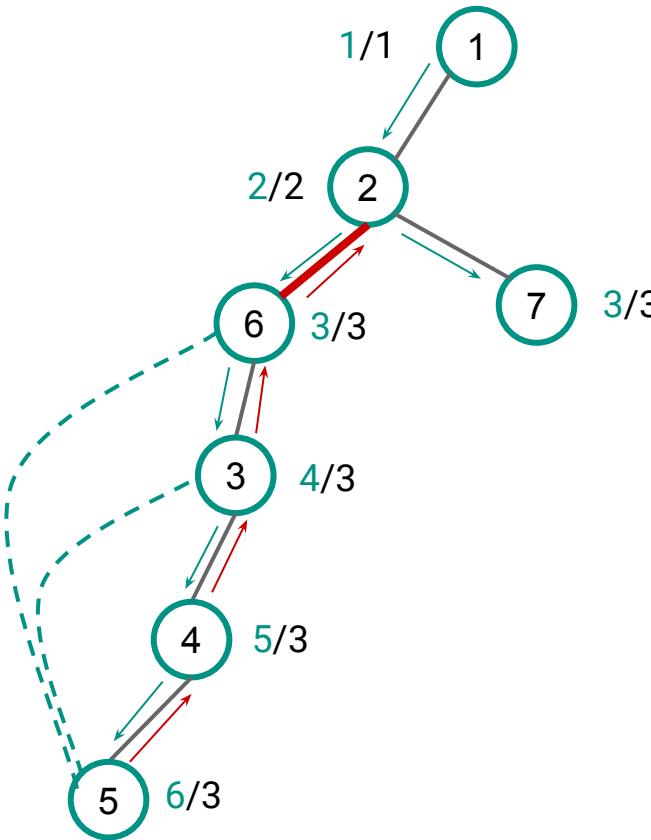
nivel/niv\_min



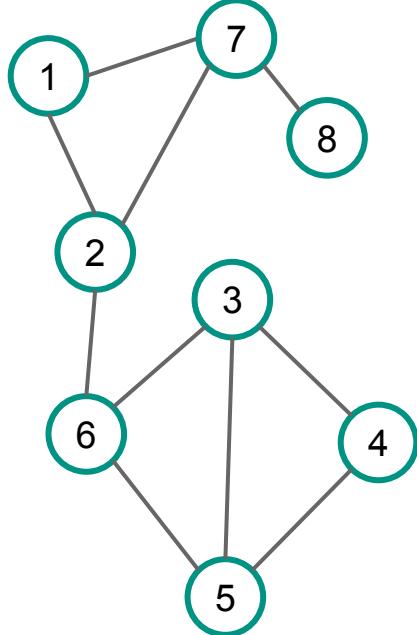
# Muchii critice - Exemplu



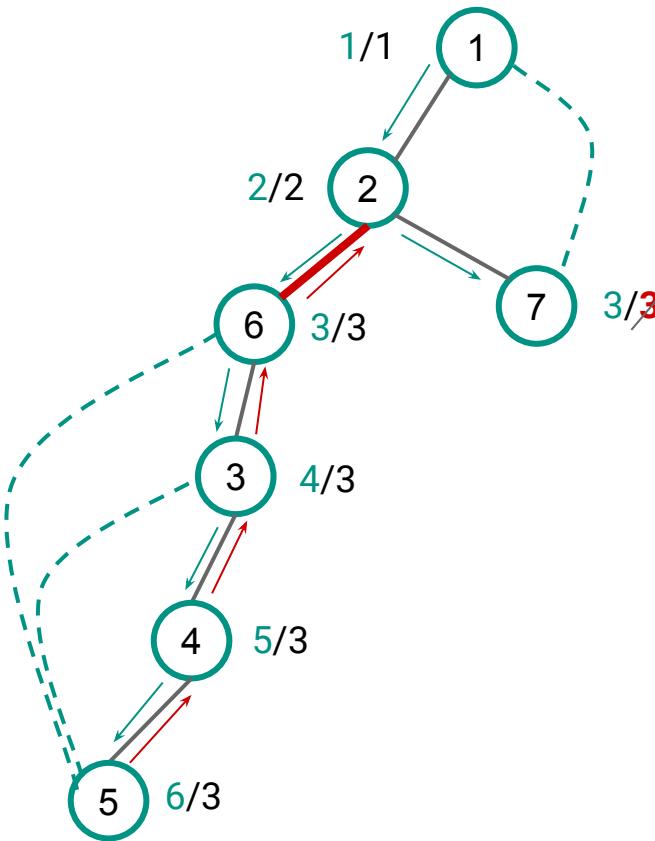
nivel/niv\_min



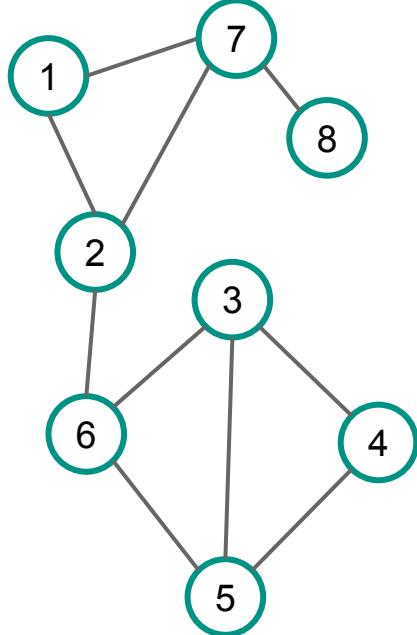
# Muchii critice - Exemplu



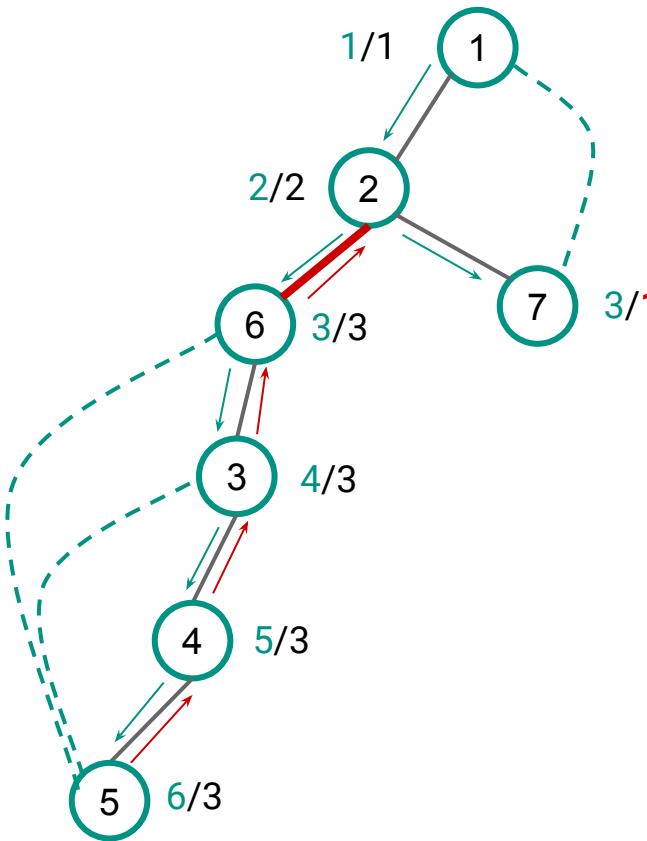
nivel/niv\_min



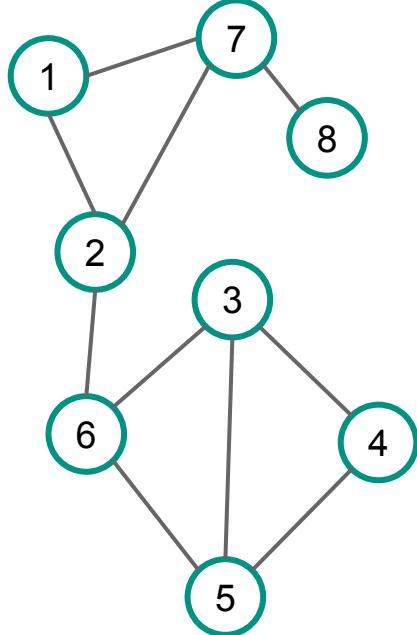
# Muchii critice - Exemplu



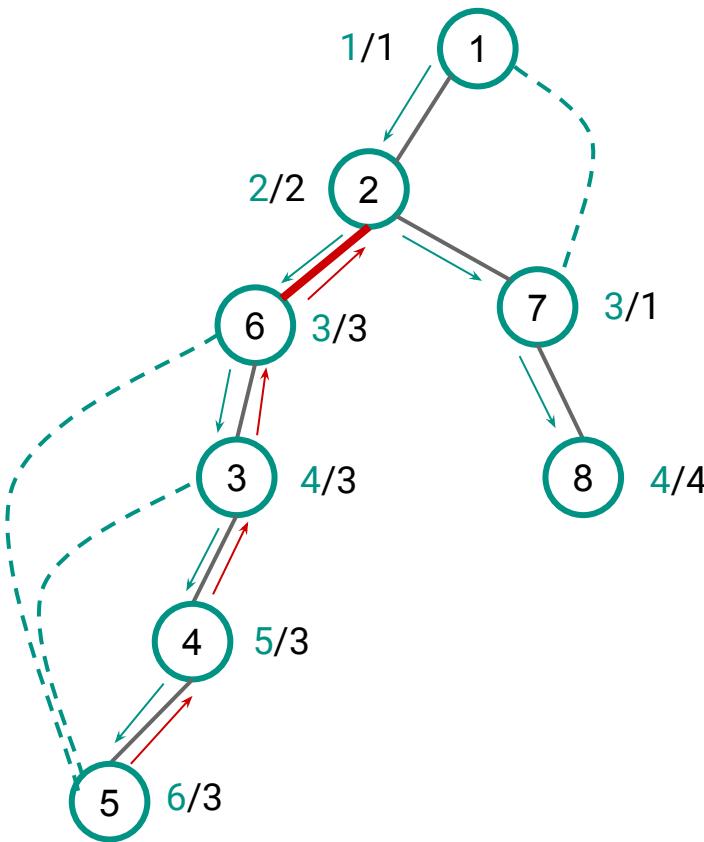
nivel/niv\_min



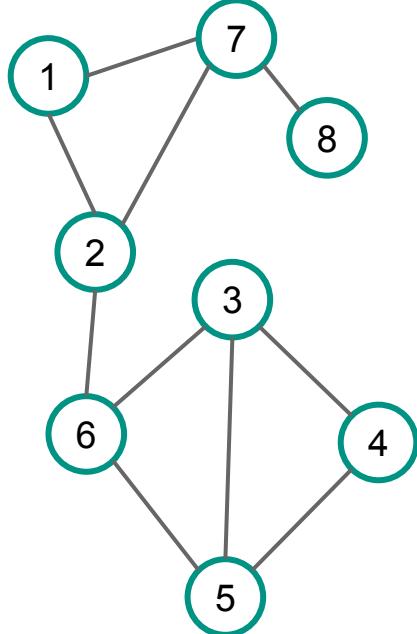
# Muchii critice - Exemplu



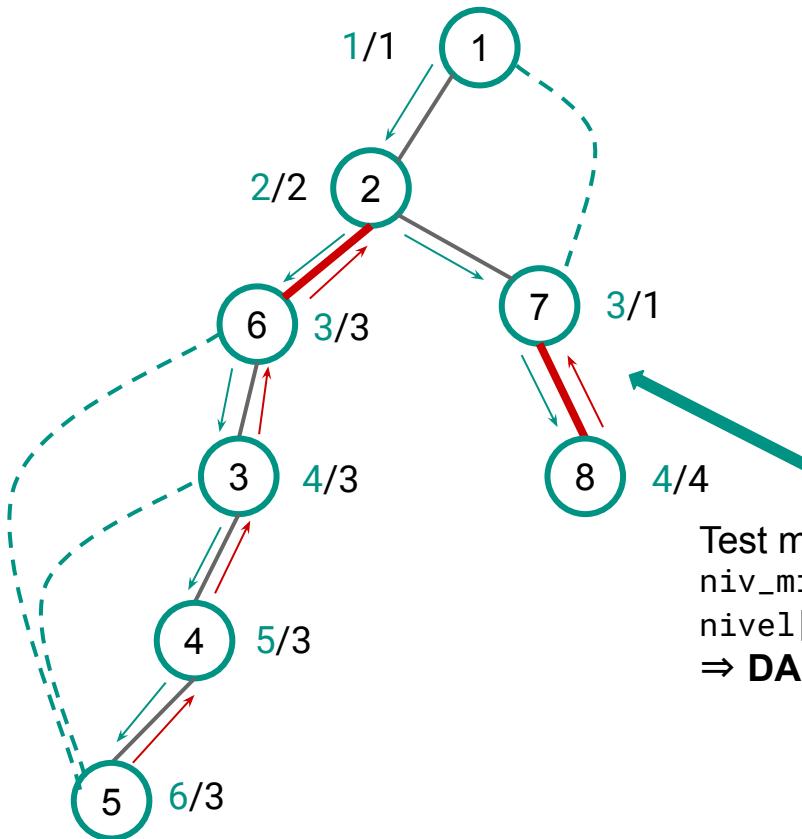
nivel/niv\_min



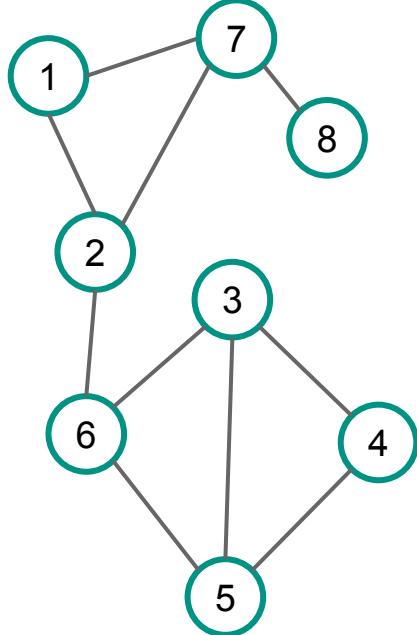
# Muchii critice - Exemplu



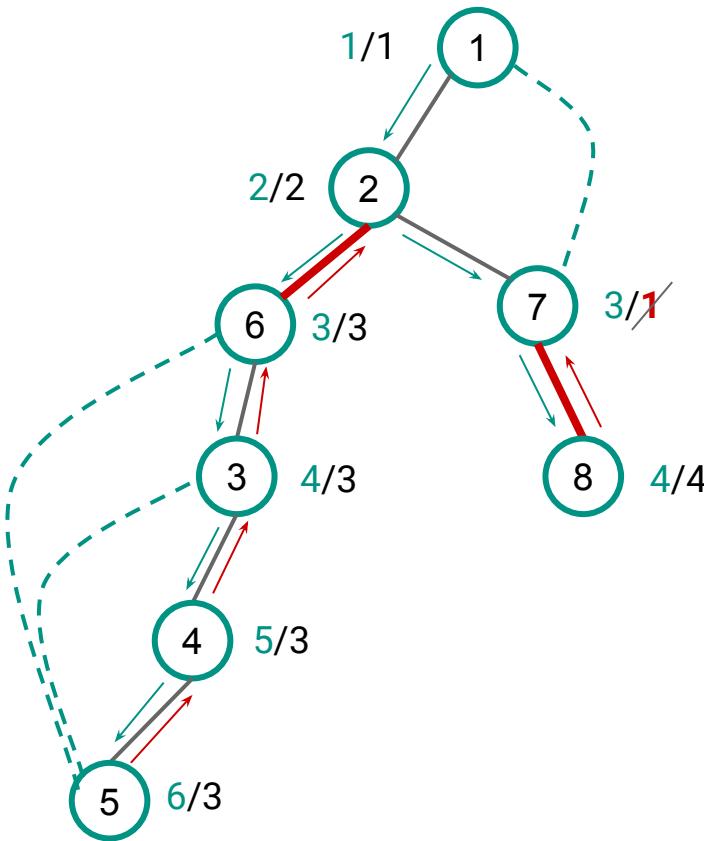
nivel/niv\_min



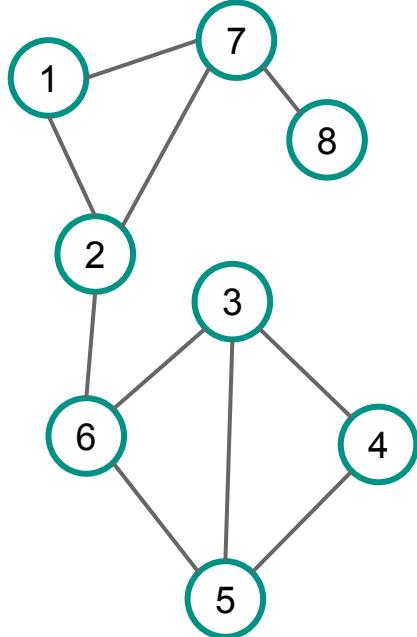
# Muchii critice - Exemplu



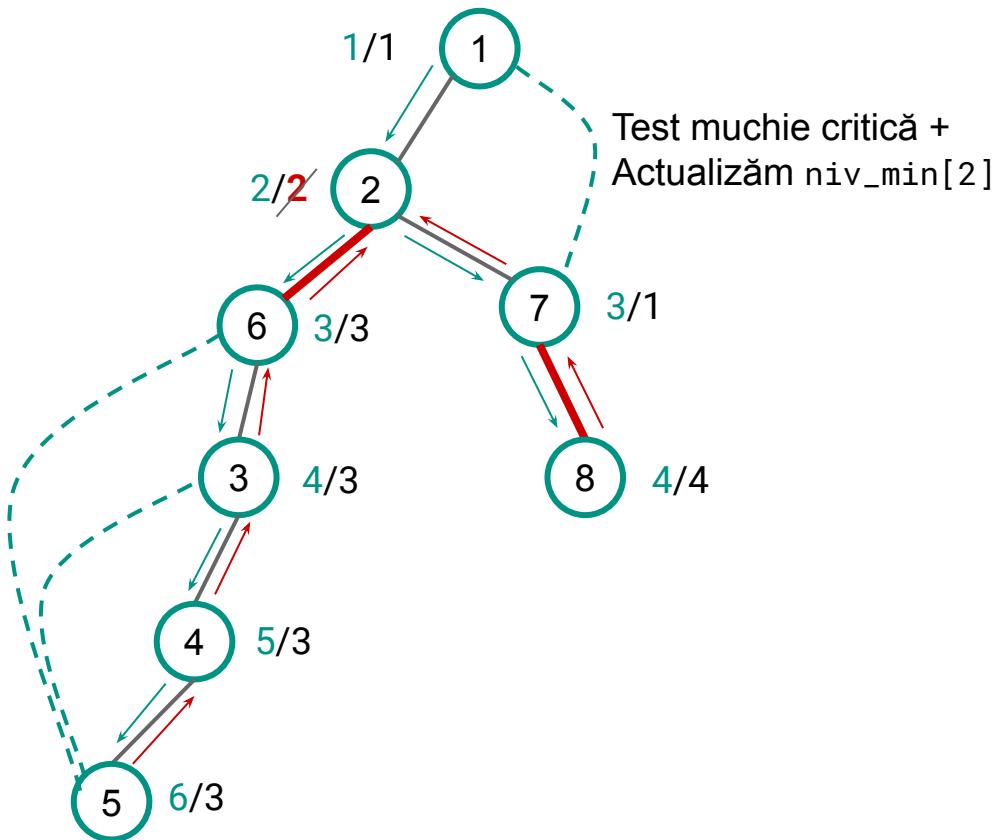
nivel/niv\_min



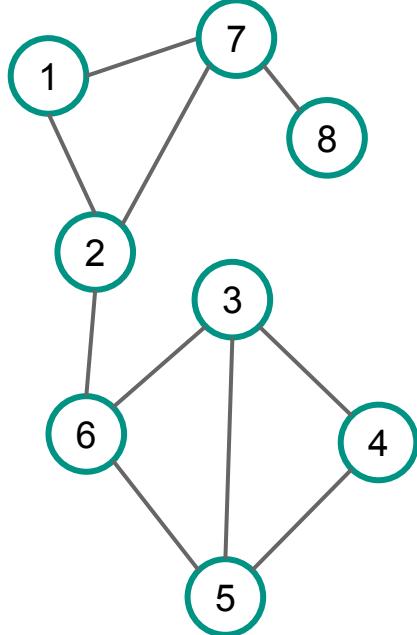
# Muchii critice - Exemplu



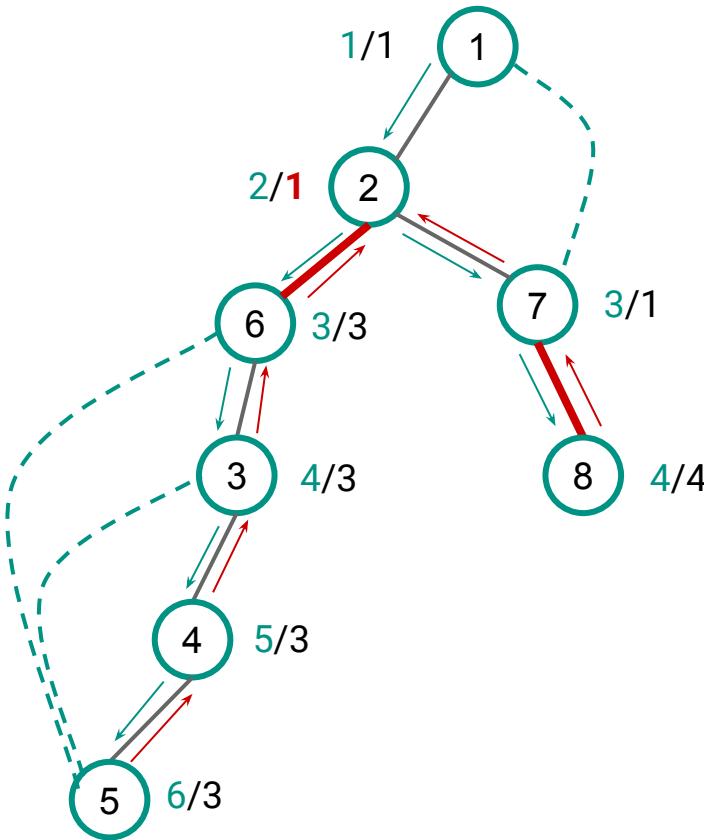
nivel/niv\_min



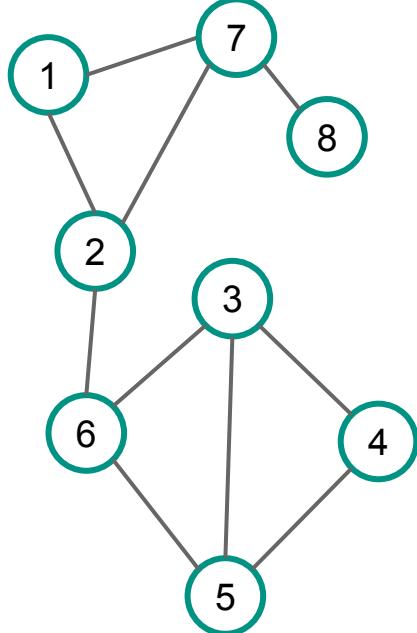
# Muchii critice - Exemplu



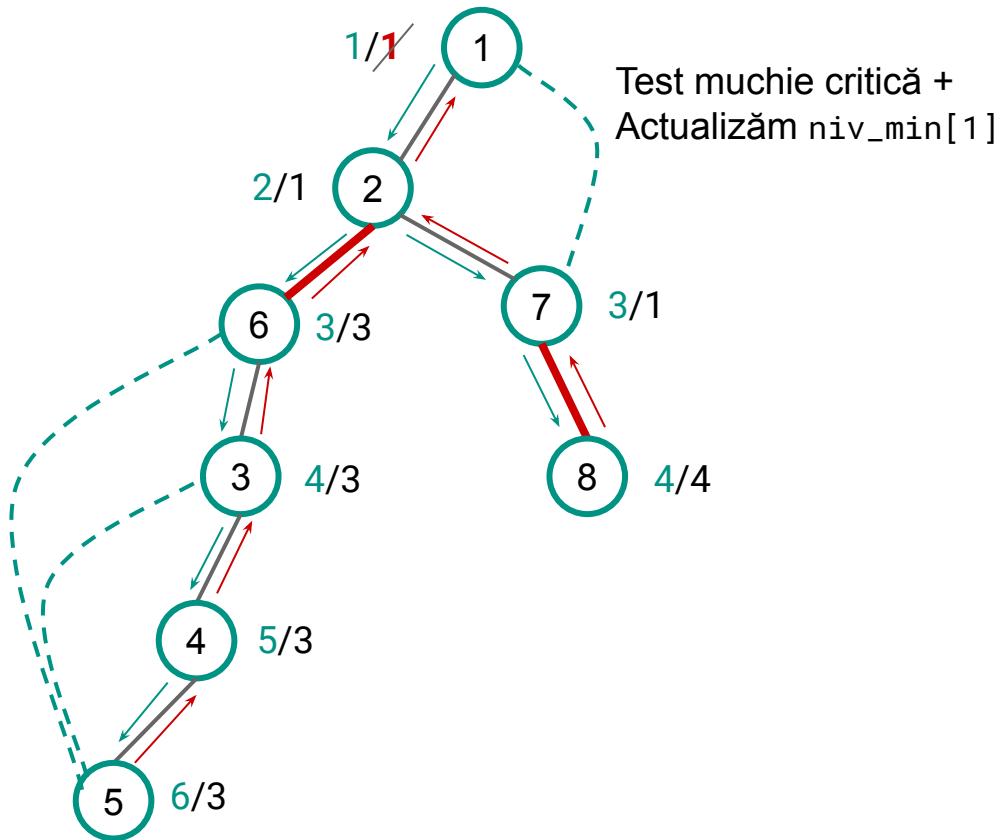
nivel/niv\_min



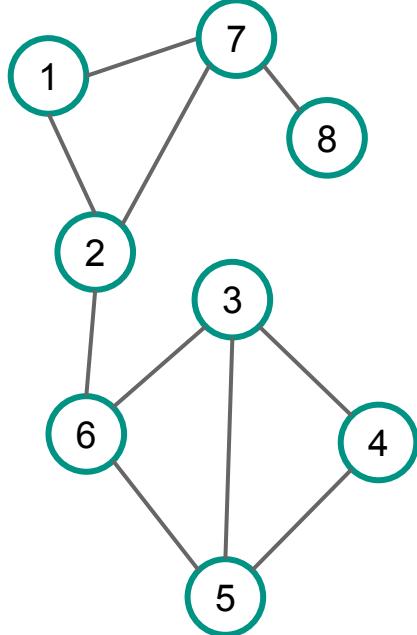
# Muchii critice - Exemplu



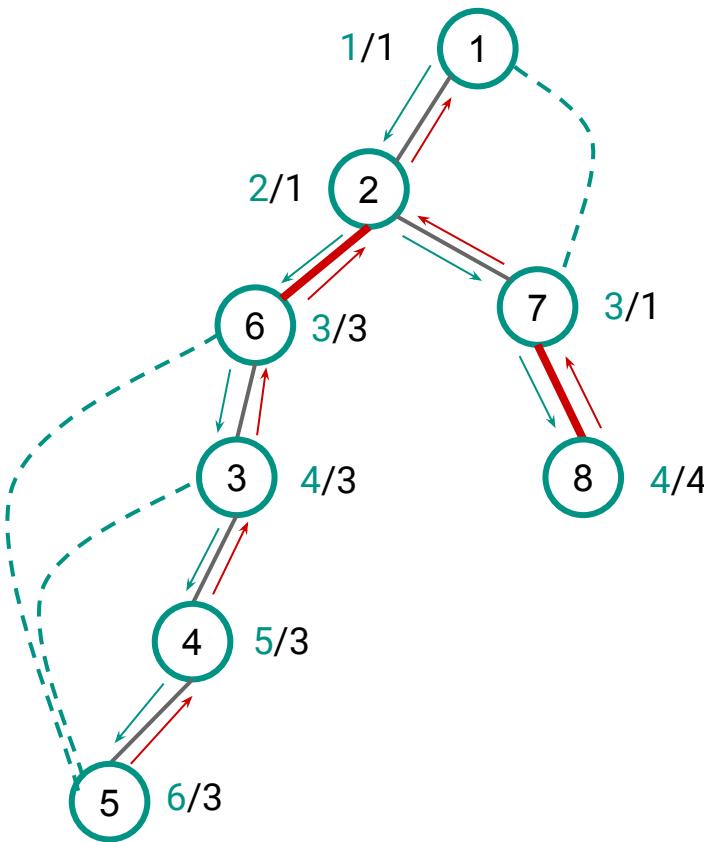
nivel/niv\_min



# Muchii critice - Exemplu



nivel/niv\_min



# Indicații de implementare

```
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;
            df(j);
            //actualizare niv_min[i] - formula B
            niv_min[i] = min{ niv_min[i], niv_min[j] }
            //test ij este muchie critica
            ...
        }
    else
        if(nivel[j] < nivel[i] - 1) //ij muchie de intoarcere
            //actualizare niv_min[i] - formula A
            ...
}
```

# Indicații de implementare

```
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;
            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]> nivel[i]) scrie muchia 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], niv[j] }
}
```

# Puncte critice



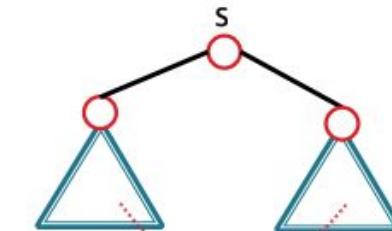
# Puncte critice

Un vârf este **punct critic**  $\Leftrightarrow$  există două vârfuri  $x,y \neq v$  astfel încât aparține oricărui  $x,y$ -lanț.

Arborele DF

**rădăcina** este punct critic

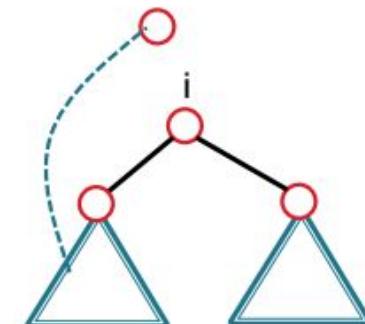
$\Leftrightarrow$



nu există muchii între subarbore (de traversare)

**un alt vârf i** din arbore este critic

$\Leftrightarrow$

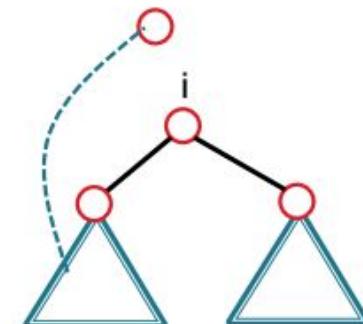
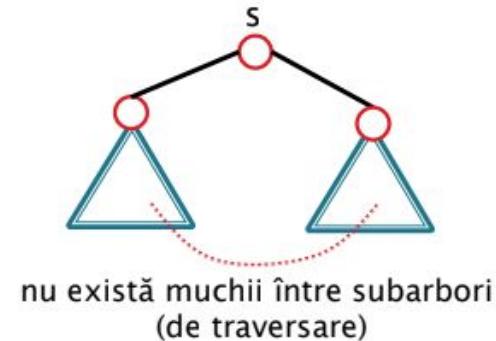


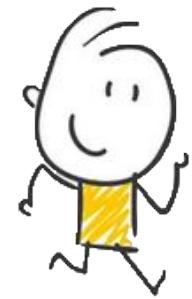
# Puncte critice

Un vârf este **punct critic**  $\Leftrightarrow$  există două vârfuri  $x, y \neq v$  astfel încât aparține oricărui  $x, y$ -lanț.

Arborele DF

- rădăcina este punct critic**  $\Leftrightarrow$   
**are cel puțin 2 fii în arborele DF**
  
- un alt vârf  $i$  din arbore este critic**  $\Leftrightarrow$   
**are cel puțin un fiu  $j$  cu  $niv\_min[j] \geq nivel[i]$**





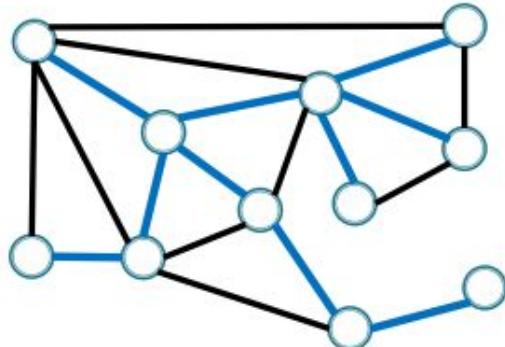
# Arborei parziali



# Arbore parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial (un graf parțial care este arbore).



# Arbore parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial.

**Demonstrație:** două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex  $G=(V, E)$

Prin adăugare de muchii (bottom-up)	Prin eliminare de muchii (cut-down)

# Arbore parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial.

**Demonstrație:** două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex  $G=(V, E)$

Prin adăugare de muchii (bottom-up)	Prin eliminare de muchii (cut-down)
$T \leftarrow (V, \emptyset)$ cât timp $T$ nu este conex execută <ul style="list-style-type: none"><li>• alege <math>e \in E(G) - E(T)</math> care unește două componente conexe din <math>T</math> (nu formează cicluri cu muchiile din <math>T</math>)</li><li>• <math>E(T) \leftarrow E(T) \cup \{e\}</math></li></ul>	
returnează $T$	

# Arborei parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial.

**Demonstrație:** două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex  $G=(V, E)$

Prin adăugare de muchii (bottom-up)	Prin eliminare de muchii (cut-down)
$T \leftarrow (V, \emptyset)$ cât timp $T$ nu este conex execută <ul style="list-style-type: none"><li>• alege <math>e \in E(G) - E(T)</math> care unește două componente conexe din <math>T</math> (nu formează cicluri cu muchiile din <math>T</math>)</li><li>• <math>E(T) \leftarrow E(T) \cup \{e\}</math></li></ul>	
returnează $T$	
În final, $T$ este conex și aciclic, deci arbore	

# Arborei parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial.

**Demonstrație:** două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex  $G=(V, E)$

Prin adăugare de muchii ( <b>bottom-up</b> )	Prin eliminare de muchii ( <b>cut-down</b> )
$T \leftarrow (V, \emptyset)$ cât timp $T$ <b>nu este conex</b> execută <ul style="list-style-type: none"><li>• alege <math>e \in E(G) - E(T)</math> care <b>unește două componente conexe</b> din <math>T</math> (nu formează cicluri cu muchiile din <math>T</math>)</li><li>• <math>E(T) \leftarrow E(T) \cup \{e\}</math></li></ul>	$T \leftarrow (V, E)$ cât timp $T$ <b>conține cicluri</b> execută <ul style="list-style-type: none"><li>• alege <math>e \in E(T)</math> o <b>muchie dintr-un ciclu</b></li><li>• <math>E(T) \leftarrow E(T) - \{e\}</math></li></ul> returnează $T$
returnează $T$	
În final, $T$ este conex și aciclic, deci arbore	

# Arborei parțiali

## Proprietate

Orice graf neorientat conex conține un arbore parțial.

**Demonstrație:** două tipuri de algoritmi de construcție a unui arbore parțial al unui graf conex  $G=(V, E)$

Prin adăugare de muchii ( <b>bottom-up</b> )	Prin eliminare de muchii ( <b>cut-down</b> )
$T \leftarrow (V, \emptyset)$ cât timp $T$ <b>nu este conex</b> execută <ul style="list-style-type: none"><li>• alege <math>e \in E(G) - E(T)</math> care <b>unește două componente conexe</b> din <math>T</math> (nu formează cicluri cu muchiile din <math>T</math>)</li><li>• <math>E(T) \leftarrow E(T) \cup \{e\}</math></li></ul> returnează $T$	$T \leftarrow (V, E)$ cât timp $T$ <b>conține cicluri</b> execută <ul style="list-style-type: none"><li>• alege <math>e \in E(T)</math> o <b>muchie dintr-un ciclu</b></li><li>• <math>E(T) \leftarrow E(T) - \{e\}</math></li></ul> returnează $T$
În final, $T$ este conex și aciclic, deci arbore	În final, $T$ este aciclic și conex (s-au eliminat doar muchii din ciclu), deci arbore

# Arbore parțiali

Algoritmi de determinare a unui arbore parțial al unui graf conex



**Complexitate algoritm?**

# Arbore parțiali

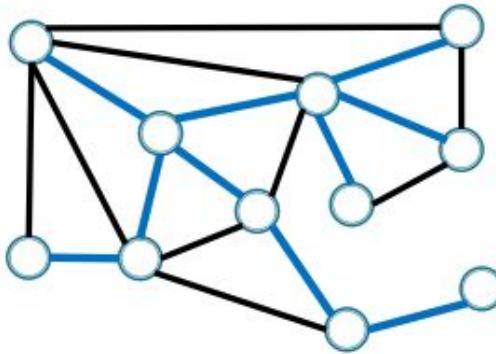
Algoritmi de determinare a unui arbore parțial al unui graf conex

Complexitate algoritm?



arbore asociat unei parcurgeri este arbore parțial ⇒  
determinăm un arbore parțial printr-o parcuregere

# Arborei parțiali



- "Scheletul" grafului
- Transmiterea de mesaje în rețea astfel încât mesajul să ajungă o singură dată în fiecare vârf
- Conectare fără redundanță + cu cost minim

# Arbore parțiali de cost minim

# Arborei parțiali de cost minim



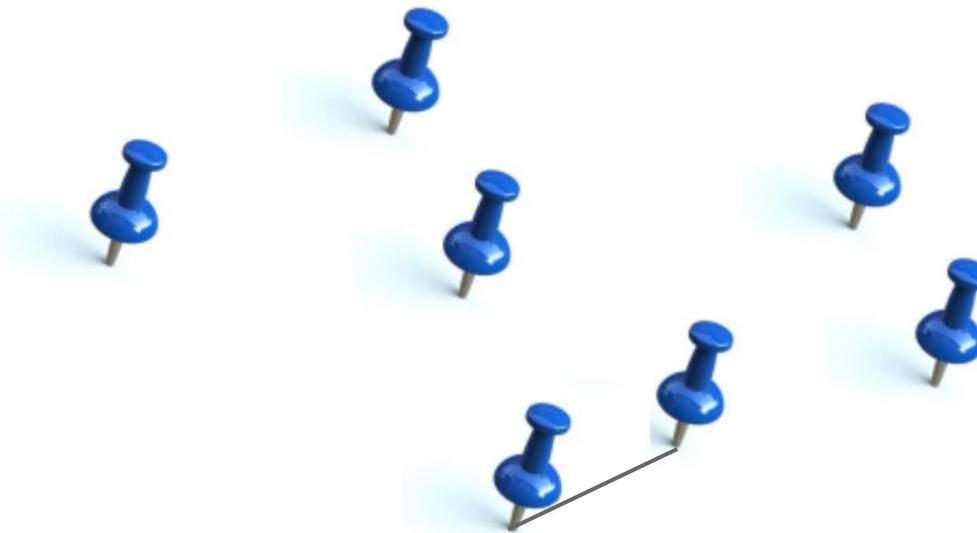
Conectați pinii astfel încât să folosiți cât mai puțin cablu.

# Arbore parțiali de cost minim

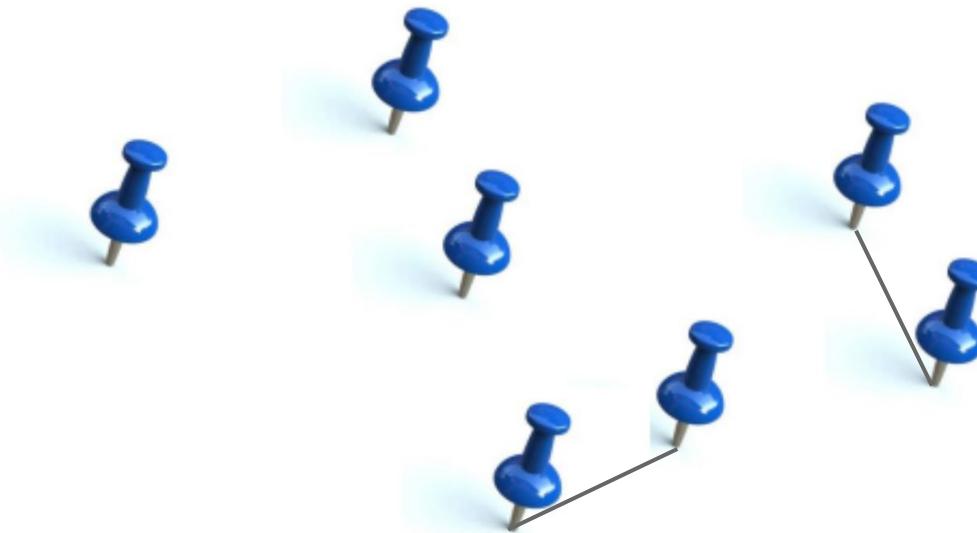


- Legăm pini apropiati
- Nu închidem cicluri

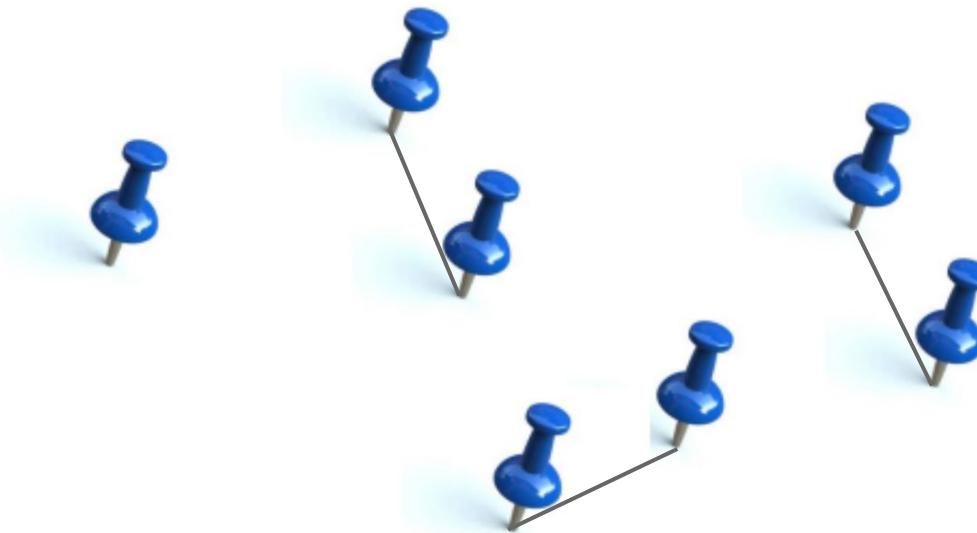
# Arbore parțiali de cost minim



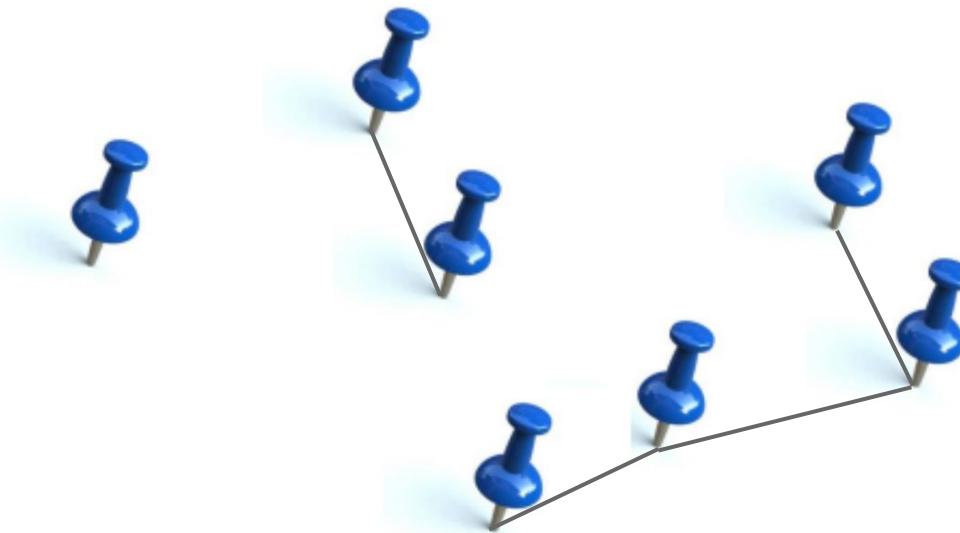
# Arbore parțiali de cost minim



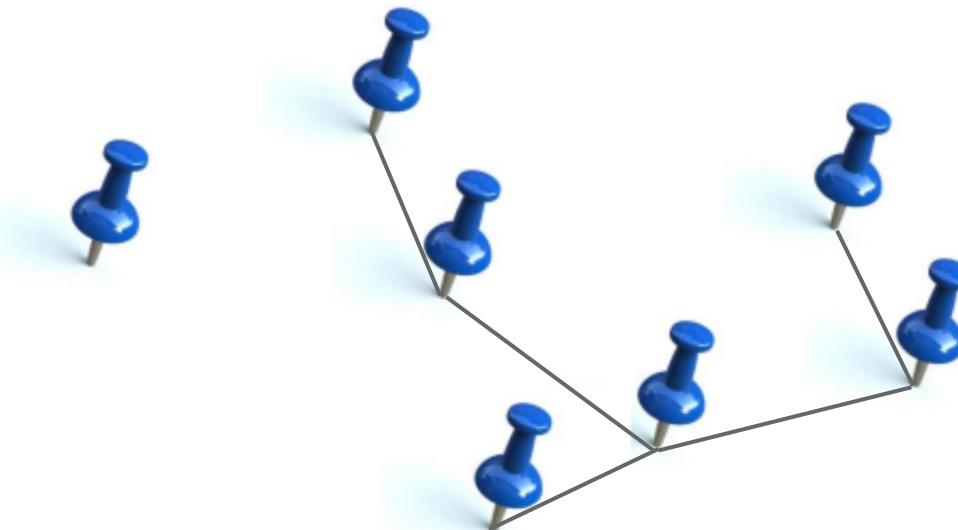
# Arbore parțiali de cost minim



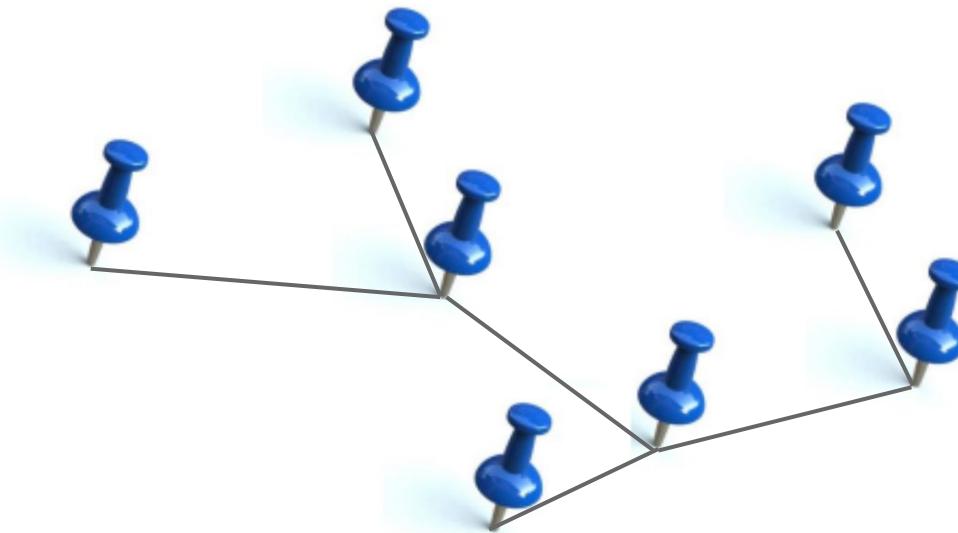
# Arbore parțiali de cost minim



# Arbore parțiali de cost minim



# Arbore parțiali de cost minim



# Arborei parțiali de cost minim



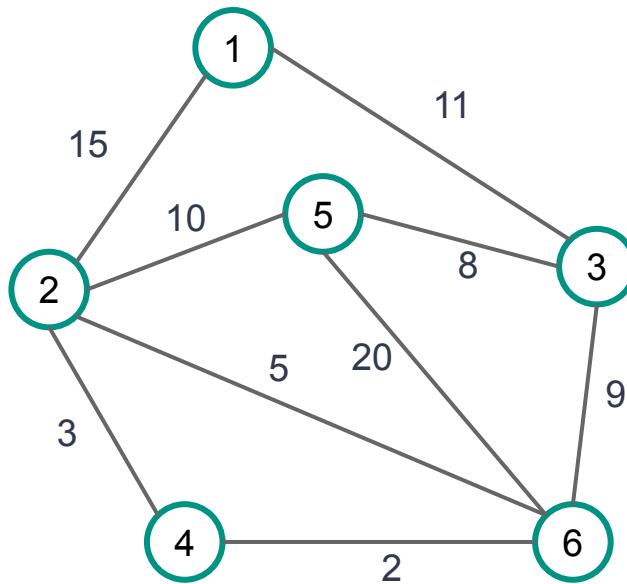
**conectare cu cost minim  $\Rightarrow$  evităm ciclurile**

Deci, trebuie să construim

**graf conex + fără cicluri  $\Rightarrow$  arbore**

**cu suma costurilor muchiilor minimă**

# Grafuri ponderate



# Grafuri ponderate

**$G = (V, E)$  ponderat =**

- $w : E \rightarrow \mathbb{R}$  funcție **pondere (cost)**

Notăm  $G = (V, E, w)$

# Grafuri ponderate

**G = (V, E) ponderat =**

- $w : E \rightarrow \mathbb{R}$  funcție **pondere (cost)**

Pentru  $A \subseteq E$

$$w(A) = \sum_{e \in A} w(e)$$

# Grafuri ponderate

**G = (V, E) ponderat =**

- $w : E \rightarrow \mathbb{R}$  funcție **pondere (cost)**

Pentru  $A \subseteq E$

$$w(A) = \sum_{e \in A} w(e)$$

Pentru T subgraf al lui G

$$w(T) = \sum_{e \in E(T)} w(e)$$

# Grafuri ponderate

**Reprezentarea grafurilor ponderate**

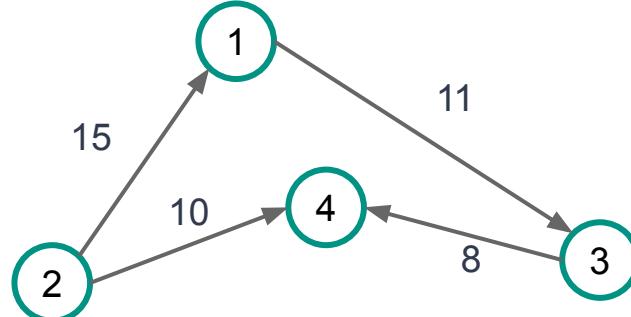
# Grafuri ponderate

## Reprezentarea grafurilor ponderate

### Matrice de costuri (ponderi)

$$W = (w_{ij})_{i,j=1,\dots,n}$$

$$w_{ij} = \begin{cases} 0, & \text{daca } i = j \\ w(i,j), & \text{daca } ij \in E \\ \infty, & \text{daca } ij \notin E \end{cases}$$

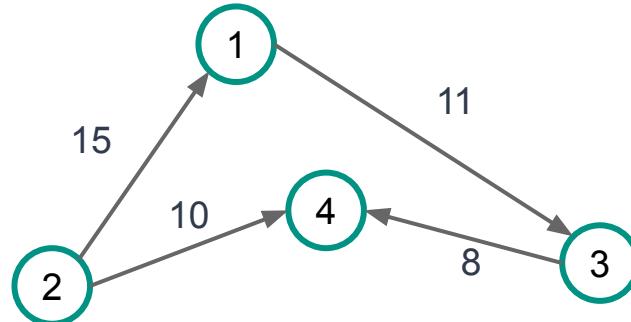


0	$\infty$	11	$\infty$
15	0	$\infty$	10
$\infty$	$\infty$	0	8
$\infty$	$\infty$	$\infty$	0

# Grafuri ponderate

## Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
- Liste de adiacență

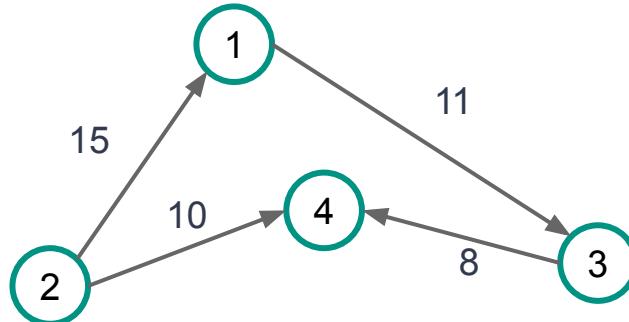


- |    |                  |
|----|------------------|
| 1: | $3 / 11$         |
| 2: | $1 / 15, 4 / 10$ |
| 3: | $4 / 8$          |
| 4: |                  |

# Grafuri ponderate

## Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)
- Liste de adiacență
- Liste de muchii / arce



1 3 11

2 1 15

2 4 10

3 4 8

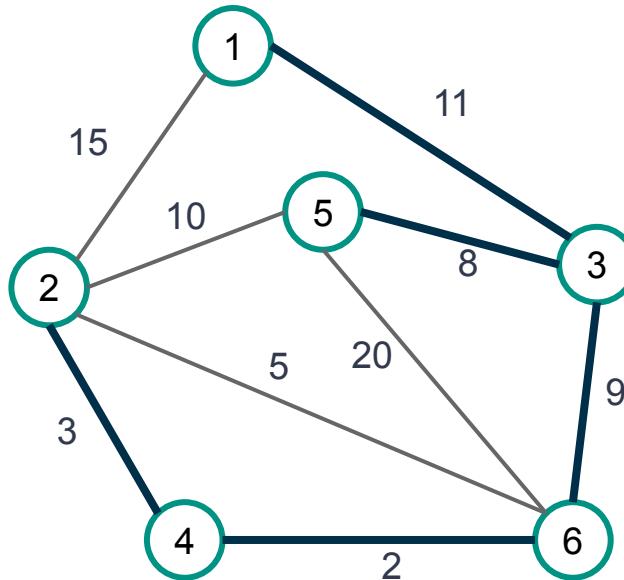
# Arbore parțiali de cost minim (APCM)

$G = (V, E, w)$  **conex ponderat**

**Arbore parțial de cost minim** al lui  $G$  = un arbore parțial  $T_{\min}$  al lui  $G$  cu

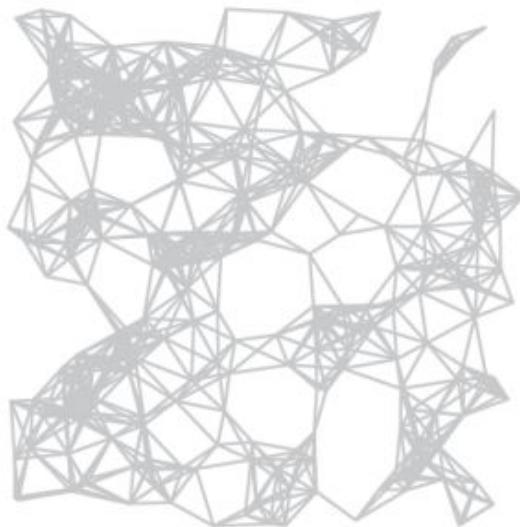
$$w(T_{\min}) = \min \{ w(T) \mid T \text{ arbore parțial al lui } G \}$$

# Arbore parțiali de cost minim (APCM)



# Arbore parțiali de cost minim (APCM)

graf 250 noduri



apcm



Imagine din

R. Sedgewick, K. Wayne – **Algorithms**, 4th edition, Pearson Education, 2011

# APCM - Aplicații

- **Construcția / renovarea unui sistem de căi ferate astfel încât:**
  - oricare două stații să fie conectate (prin căi renovate)
  - sistem economic (costul total minim)
- **Proiectarea de rețele, circuite electronice**
  - conectarea pinilor cu cost minim / fără cicluri
- **Clustering**
- **Subroutines în alți algoritmi (trasee hamiltoniene)**
- ...

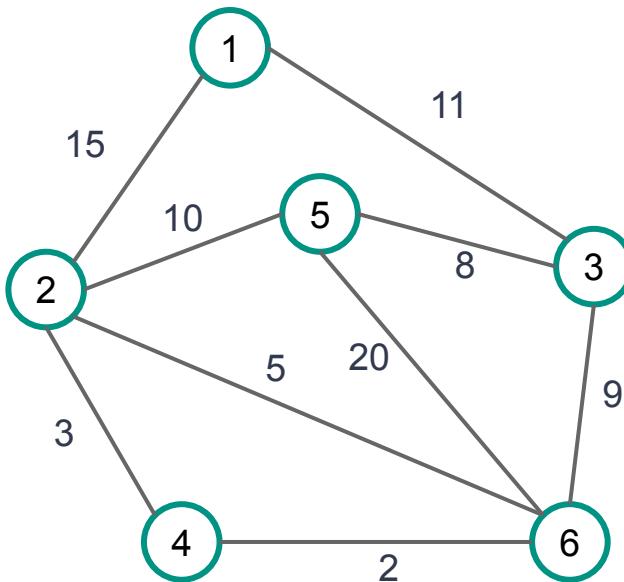
# Algoritmi de determinare a unui arbore parțial de cost minim

# Arbore parțiali de cost minim



**Cum determinăm un arbore parțial de cost minim al unui graf conex ponderat?**

# Arbore parțiali de cost minim



# Arborei parțiali de cost minim



Idee: Prin **adăugare** succesivă de muchii, astfel încât mulțimea de muchii selectate:

- să aibă costul cât mai mic
- să fie submulțime a mulțimii muchiilor unui arbore parțial de cost minim (apcm)

# Arbore parțiali de cost minim



După ce criteriu selectăm muchiile?

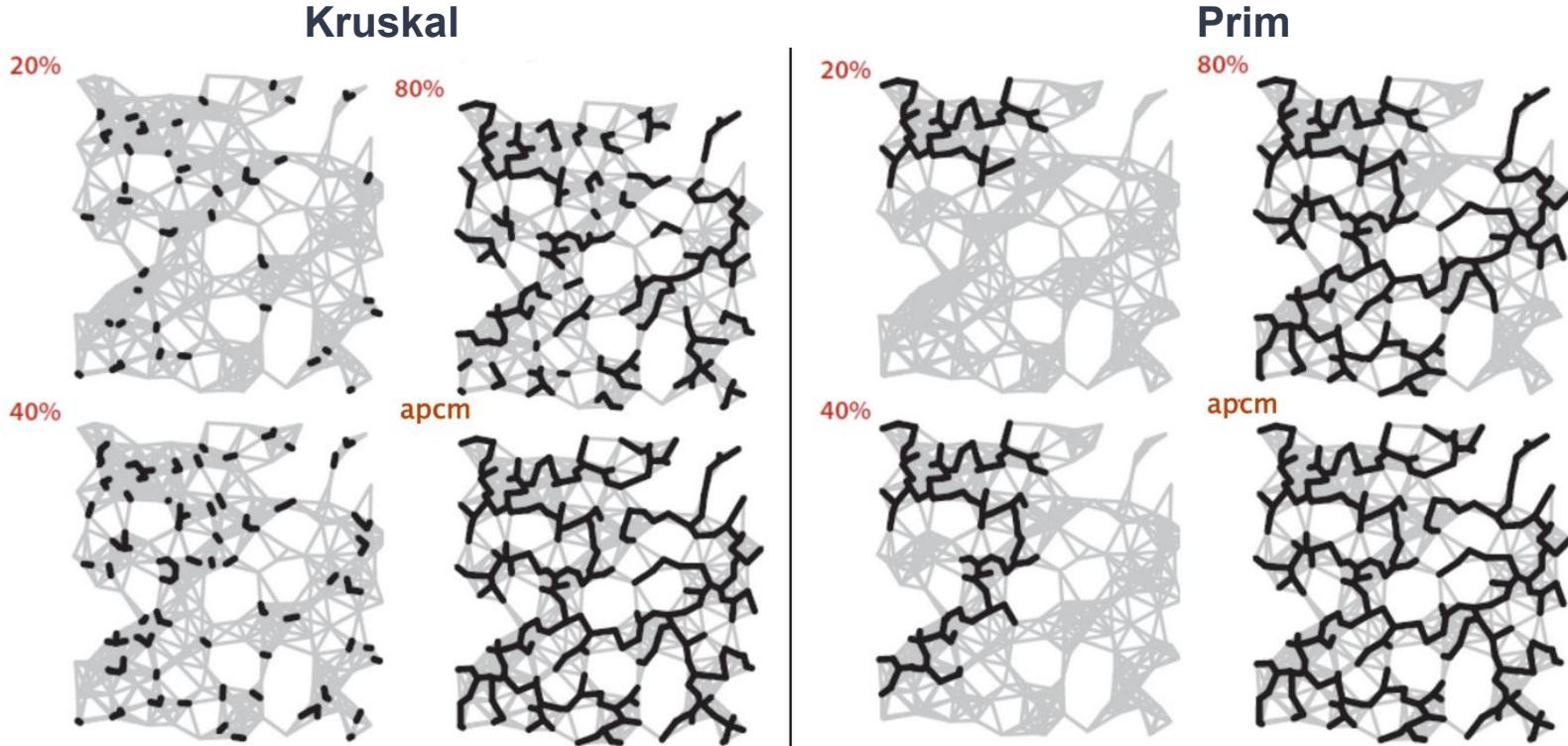
# Arbore parțiali de cost minim



După ce criteriu selectăm muchiile?

⇒ diversi algoritmi

# Arbore parțiali de cost minim



Imagine din

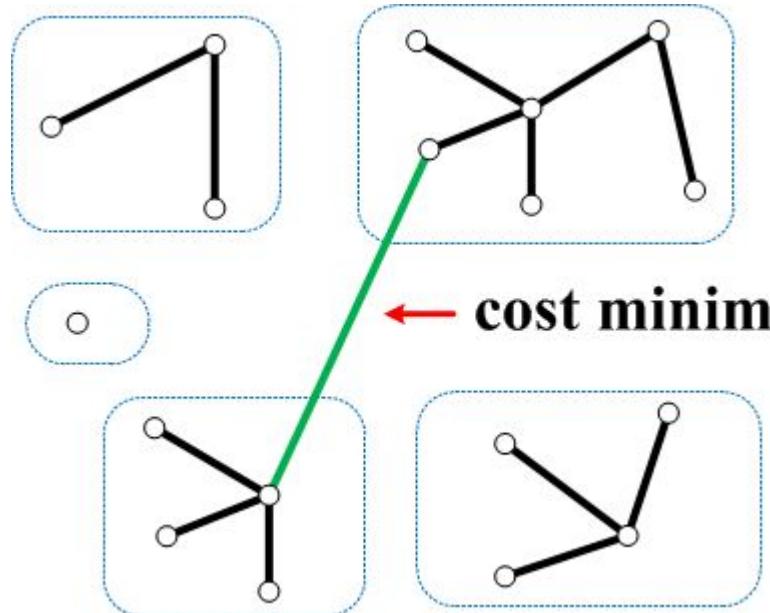
R. Sedgewick, K. Wayne – **Algorithms**, 4th edition, Pearson Education, 2011

# Algoritmul lui Kruskal



# 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).

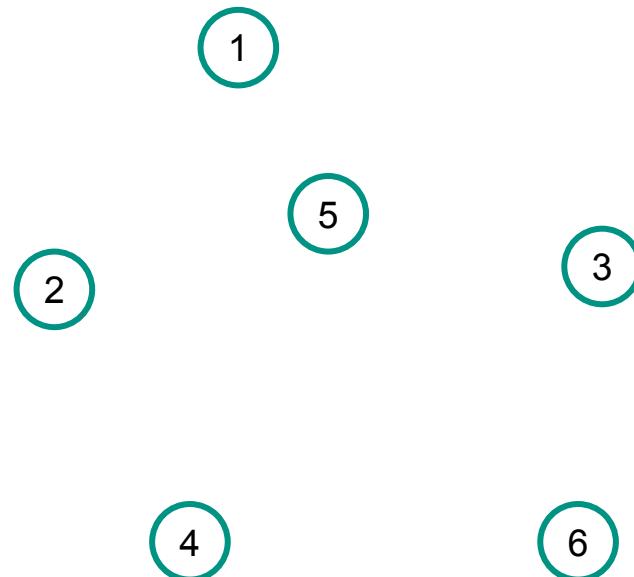


# O primă formă a algoritmului

- **Initial:**  $T = (V, \emptyset)$
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** din **G** a.î.  $u,v$  sunt în **componente conexe diferite** ( $T + uv$  aciclic)
  - $E(T) = E(T) \cup \{uv\}$

# O primă formă a algoritmului

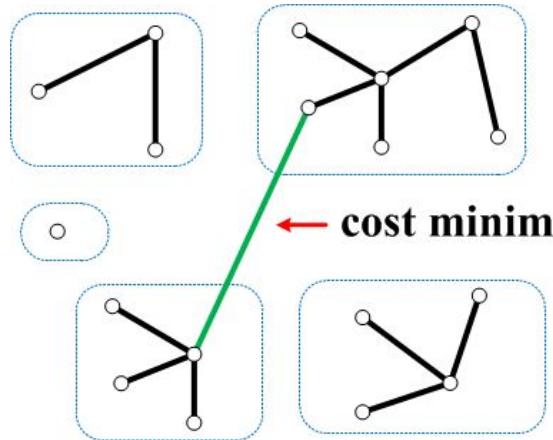
- **Initial:** cele  $n$  vârfuri sunt **izolate**, fiecare formând o componentă conexă



# O primă formă a algoritmului

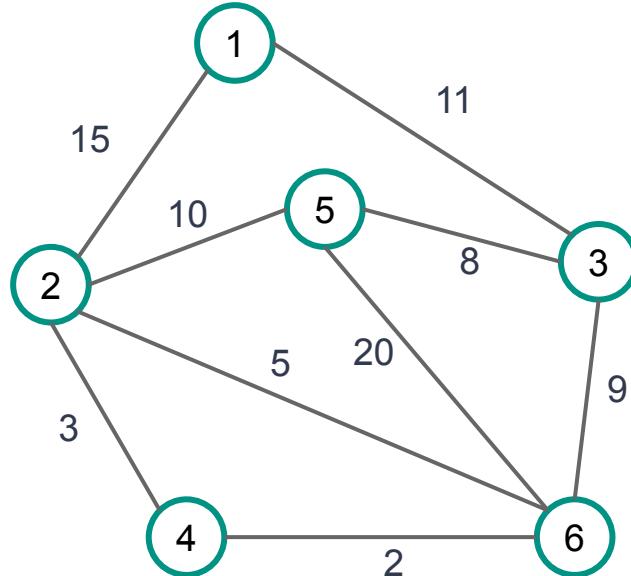
## □ La un pas:

Muchiile selectate formează o **pădure**.

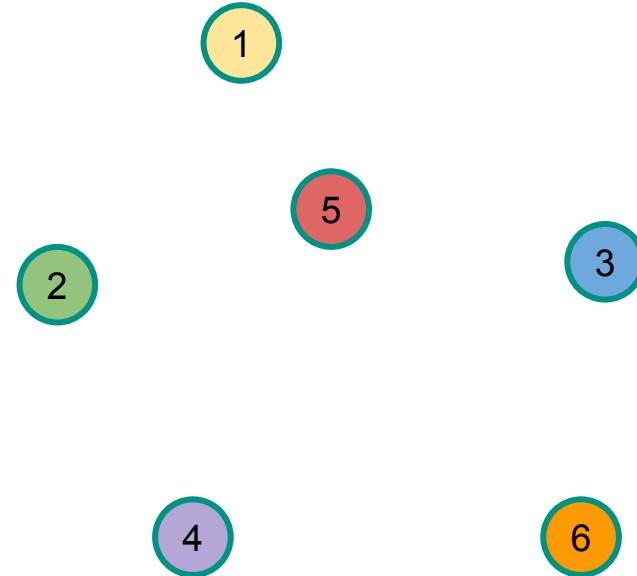
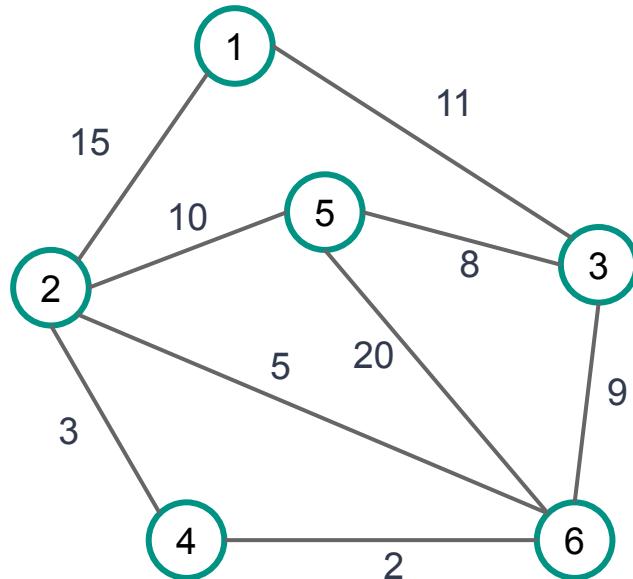


Este selectată o muchie de cost minim, care unește doi arbori din pădurea curentă (două componente conexe).

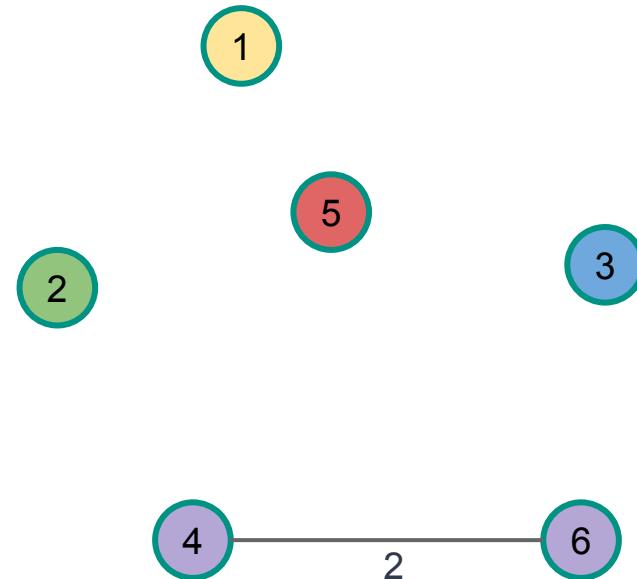
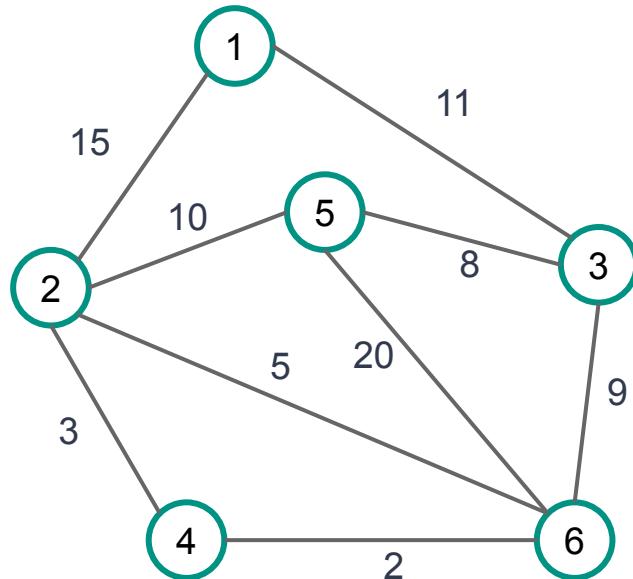
# Exemplu



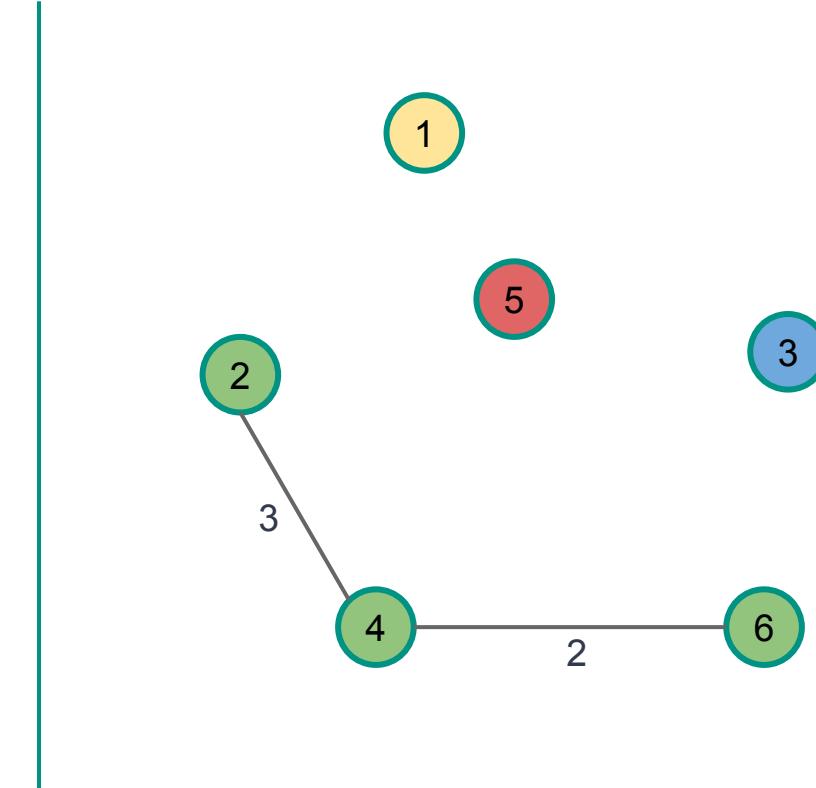
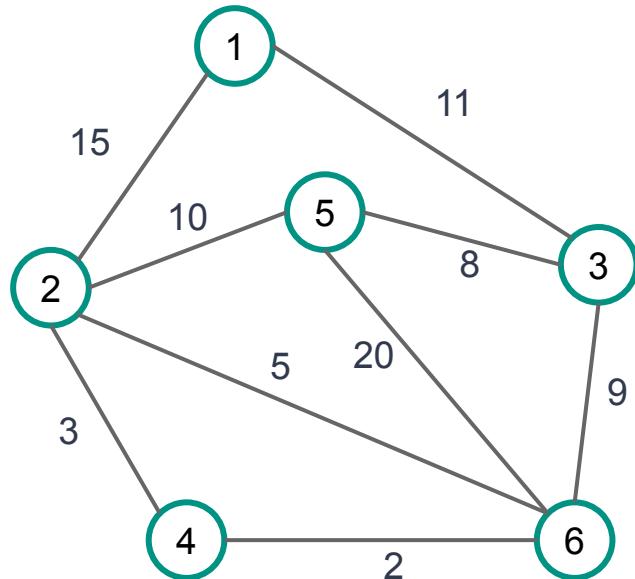
# Exemplu



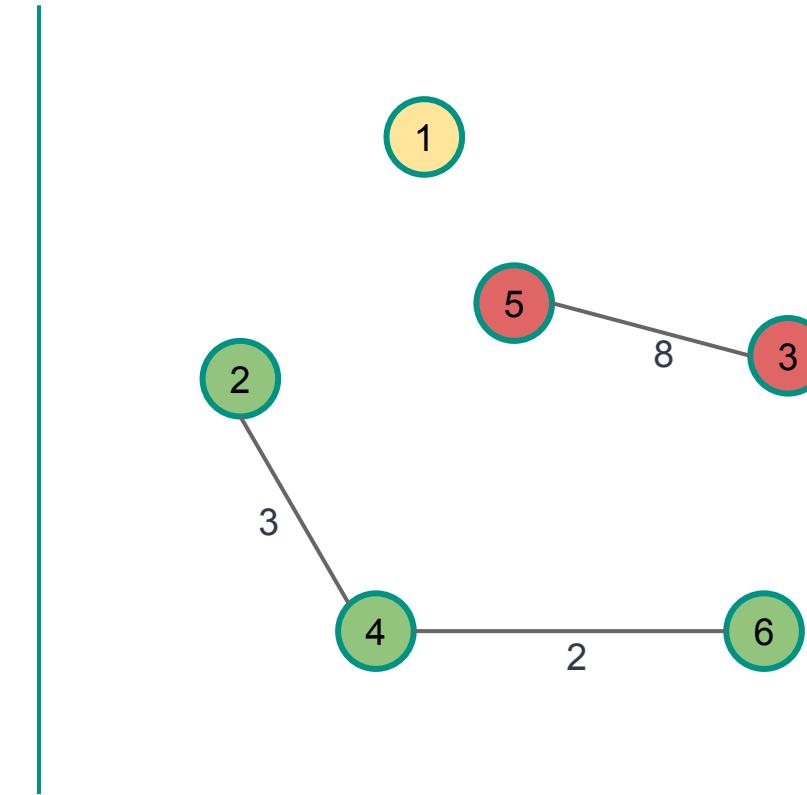
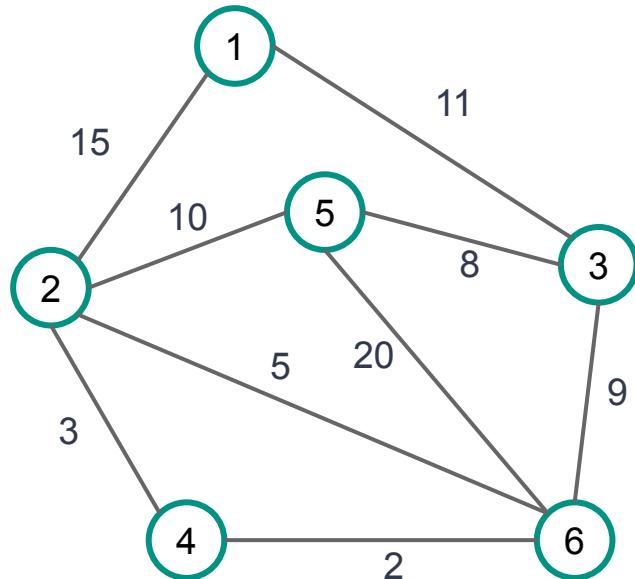
# Exemplu



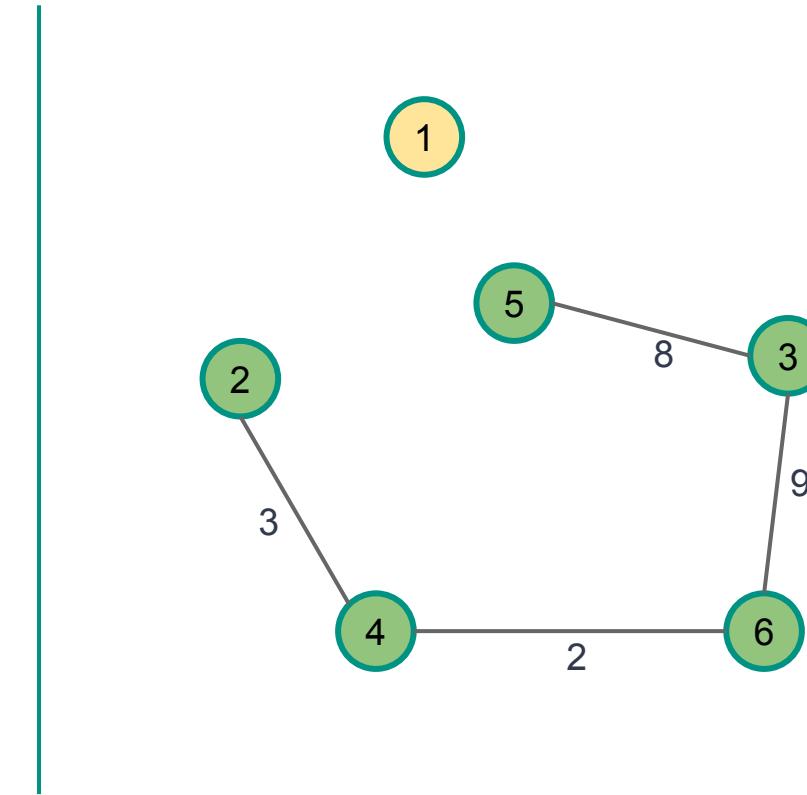
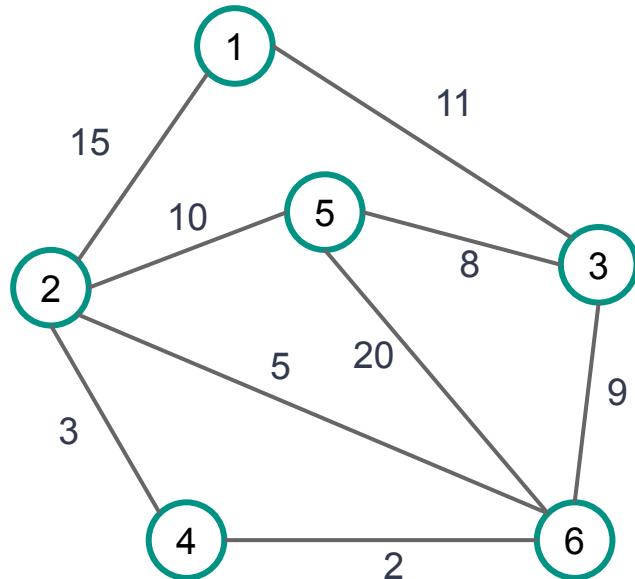
# Exemplu



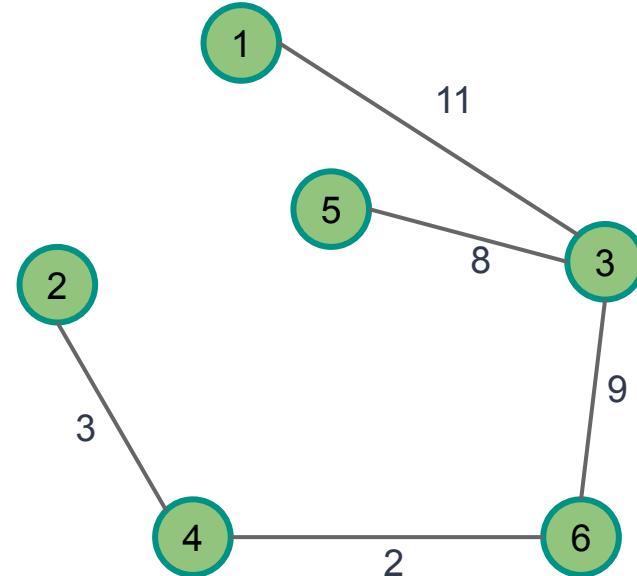
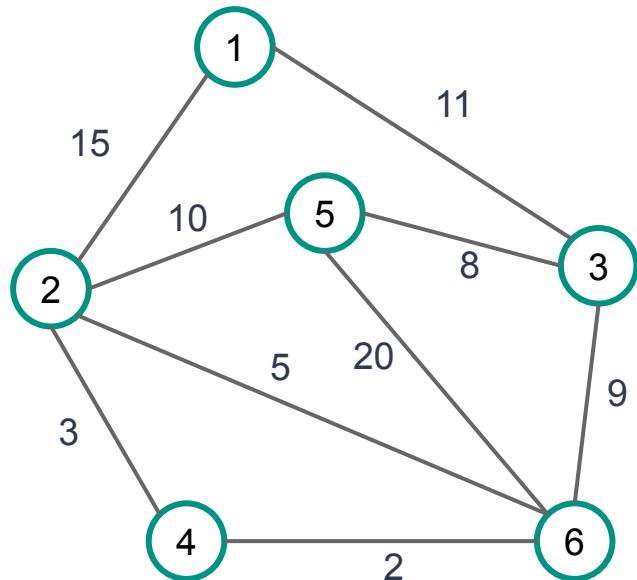
# Exemplu



# Exemplu



# Exemplu



# Kruskal - Implementare



- Cum reprezentăm graful în memorie?**
- Cum selectăm ușor o muchie:**
  - de cost minim
  - care unește două componente (nu formează cicluri cu muchiile deja selectate)

# Kruskal - Implementare



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

# Kruskal - Implementare



## Reprezentarea grafului ponderat

- **Listă de muchii:** memorăm, pentru fiecare muchie, extremitățile și costul

# Kruskal - Implementare



- Cum testăm dacă muchia curentă unește două componente ( $\Leftrightarrow$  nu formează cicluri cu muchiile deja selectate)?

# Kruskal - Implementare



- Cum testăm dacă muchia curentă unește două componente ( $\Leftrightarrow$  nu formează cicluri cu muchiile deja selectate)?



verificăm printr-o parcurgere dacă extremitățile muchiei sunt deja unite printr-un lanț

# Kruskal - Implementare



- Cum testăm dacă muchia curentă unește două componente ( $\Leftrightarrow$  nu formează cicluri cu muchiile deja selectate)?



verificăm printr-o parcurgere dacă extremitățile muchiei sunt deja unite printr-un lanț

⇒  $O(mn)$  - ineficient



# Kruskal - Implementare



Componentele sunt **mulțimi disjuncte** din **V** (partiție a lui **V**)

⇒ **structuri pentru mulțimi disjuncte**

- asociem fiecărei componente un reprezentant (o culoare)

# Kruskal - Implementare

## Operații necesare:

- Inițializare( $u$ )** -
- Reprez( $u$ )** -
- Reunește( $u,v$ )** -

# Kruskal - Implementare

## Operații necesare:

- **Inițializare( $u$ )** - creează o componentă cu un singur vârf,  $u$
- **Reprez( $u$ )** -
- **Reunește( $u,v$ )** -

# Kruskal - Implementare

## Operații necesare:

- **Inițializare( $u$ )** - creează o componentă cu un singur vârf,  $u$
- **Reprez( $u$ )** - returnează reprezentantul (culoarea) componentei care conține pe  $u$
- **Reunește( $u,v$ )** -

# Kruskal - Implementare

## Operații necesare:

- **Inițializare( $u$ )** - creează o componentă cu un singur vârf,  $u$
- **Reprez( $u$ )** - returnează reprezentantul (culoarea) componentei care conține pe  $u$
- **Reunește( $u,v$ )** - unește componenta care conține  $u$  cu cea care conține  $v$

# Kruskal - Implementare

O muchie uv unește două componente dacă și numai dacă

# Kruskal - Implementare

O muchie uv unește două componente dacă și numai dacă

$$\text{Reprez}(u) \neq \text{Reprez}(v)$$

# Kruskal

**sorteaza**(E)

```
for (v=1; v <= n; v++)
    Initializare(v);
```

# Kruskal

**sorteaza**(E)

**for** (v=1; v <= n; v++)

**Initializare**(v);

nrmse1=0

**for** (uv ∈ E)

**if** (**Reprez**(u) != **Reprez**(v)) {

}

# Kruskal

**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};

}

# Kruskal

**sorteaza**(E)

**for** (v=1; v <= n; v++)

**Initializare**(v);

nrm sel=0

**for** (uv ∈ E)

**if** (**Reprez**(u) != **Reprez**(v)) {

        E(T) = E(T) ∪ {uv};

**Reuneste**(u,v);

        nrm sel = nrm sel + 1;

**if** (nrm sel == n-1)

**STOP**; // break

}

# Kruskal

## Complexitate



De câte ori se execută fiecare operație?

# Kruskal

## Complexitate

- **Sortare** →  $O(m \log m) = O(m \log n)$
- **$n * \text{Initializare}$**
- **$2m * \text{Reprez}$**
- **$(n-1) * \text{Reuneste}$**

**Depinde de modalitatea de memorare a componentelor conexe**

# Kruskal



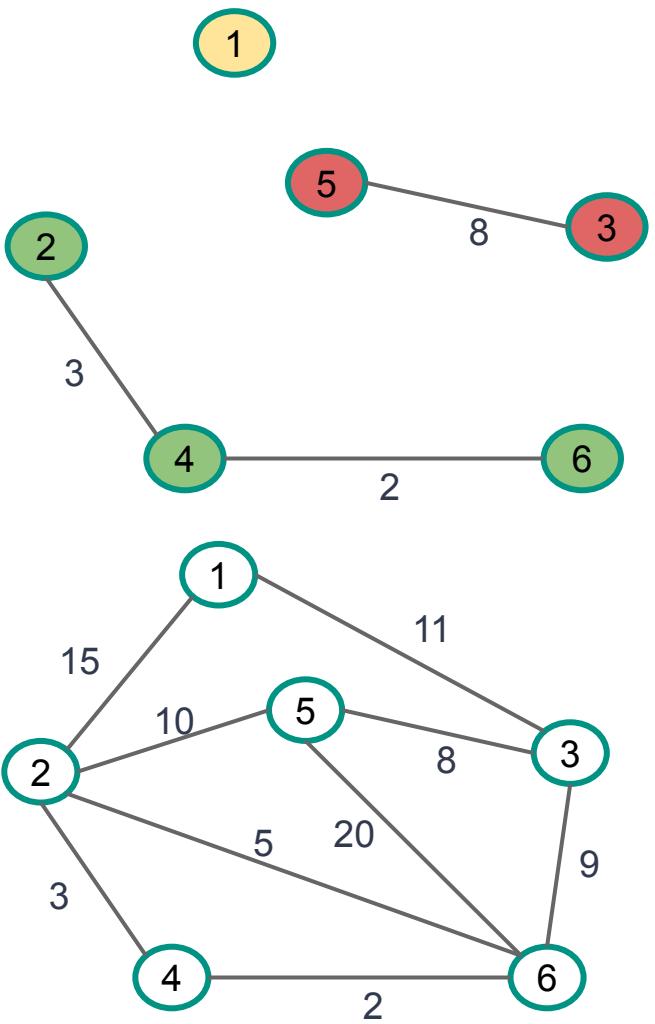
Cum memorăm componentele + reprezentantul / culoarea componentei în care se află un vârf?

# Kruskal



**Varianta 1** - memorăm într-un vector, pentru fiecare vârf, reprezentantul / culoarea componentei din care face parte

$r[u]$  = culoarea (reprezentantul) componentei care conține vârful  $u$



$$r = [1, 2, 3, 2, 3, 2]$$

# Kruskal

**Initializare**

**Reprez**

**Reuneste**

# Kruskal

- **Initializare - O(1)**

```
void Initializare(int u) {  
    r[u] = u;  
}
```

- **Reprez**

- **Reuneste**

# Kruskal

- **Initializare - O(1)**

```
void Initializare(int u) {  
    r[u] = u;  
}
```

- **Reprez - O(1)**

```
int Reprez(int u) {  
    return r[u];  
}
```

- **Reuneste**

# Kruskal

- **Initializare - O(1)**

```
void Initializare(int u) {  
    r[u] = u;  
}
```

- **Reprez - O(1)**

```
int Reprez(int u) {  
    return r[u];  
}
```

```
void Reuneste(int u, int v) {  
    r1 = Reprez(u); //r1=r[u]  
    r2 = Reprez(v); //r2=r[v]  
    for(k=1; k<=n; k++)  
        if(r[k] == r2)  
            r[k] = r1;  
}
```

- **Reuneste - O(n)**

# Kruskal

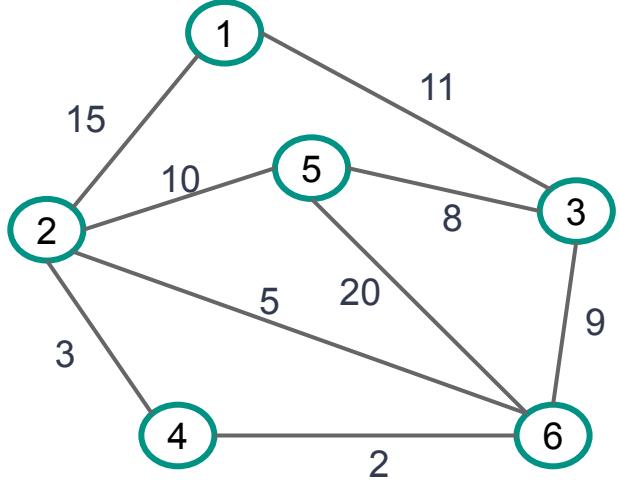
## Complexitate

**Varianta 1** - dacă folosim vector de reprezentanți

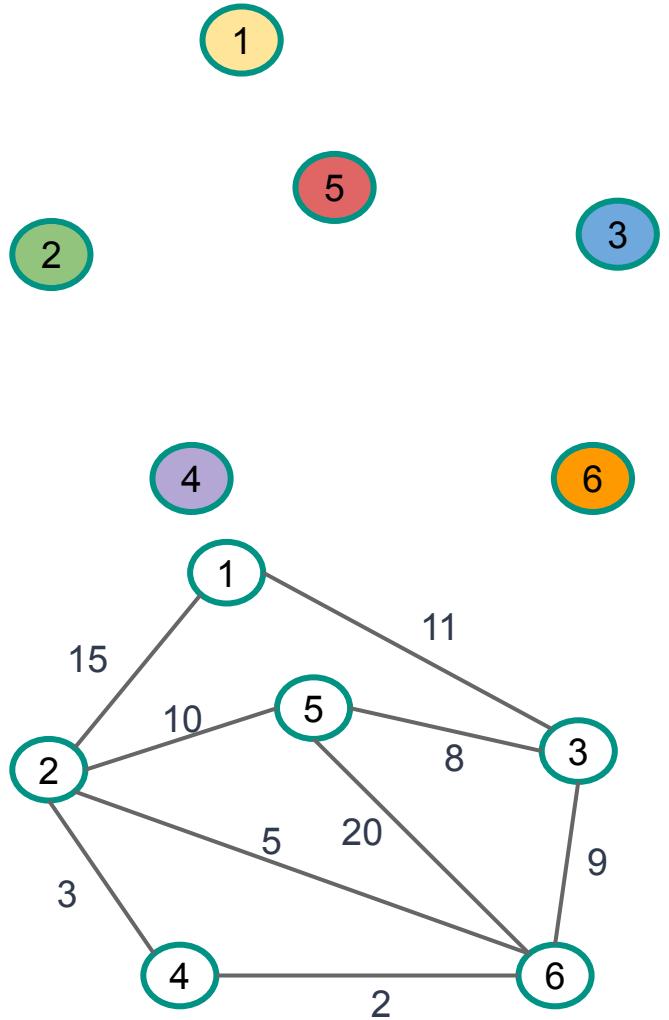
- Sortare** →  $O(m \log m) = O(m \log n)$
- $n * \text{Initializare}$**  →  $O(n)$
- $2m * \text{Reprez}$**  →  $O(m)$
- $(n-1) * \text{Reuneste}$**  →  $O(n^2)$

---

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



(4, 6)  
(2, 4)  
(2, 6)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)



$$r = [1, 2, 3, 4, 5, 6]$$

(4, 6)

(2, 4)

(2, 6)

(3, 5)

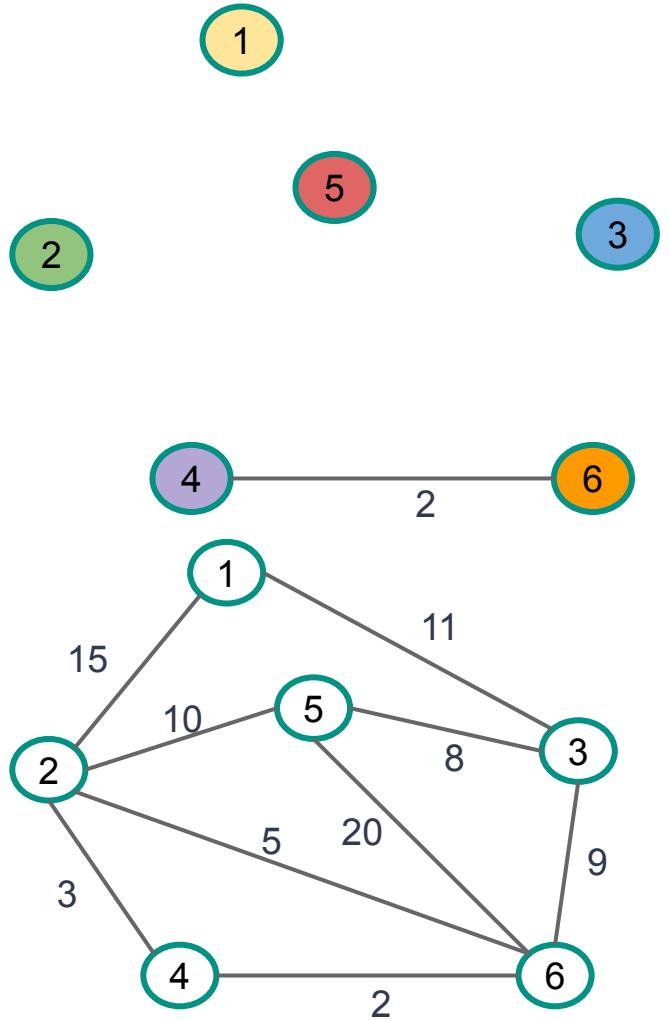
(3, 6)

(2, 5)

(1, 3)

(1, 2)

(5, 6)



**(4, 6)**

(2, 4)

(2, 6)

(3, 5)

(3, 6)

(2, 5)

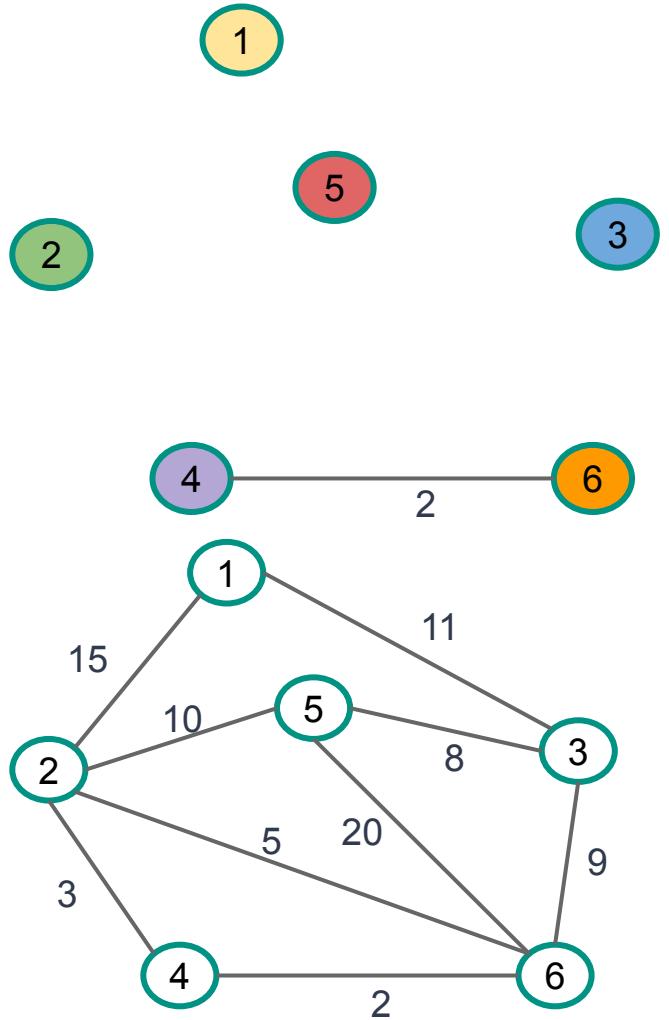
(1, 3)

(1, 2)

(5, 6)

$r = [1, 2, 3, \underline{4}, 5, 6]$

$r(4) \neq r(6)$



(4, 6)

(2, 4)

(2, 6)

(3, 5)

(3, 6)

(2, 5)

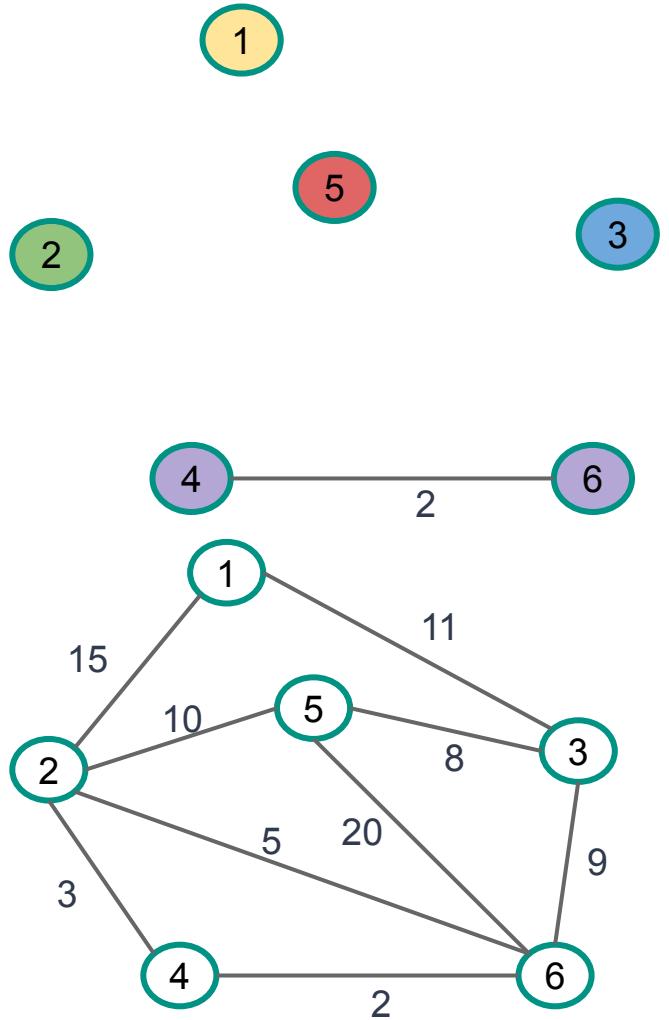
(1, 3)

(1, 2)

(5, 6)

$r = [1, 2, 3, 4, 5, 6]$

Reuneste(4, 6)



**(4, 6)**

(2, 4)

(2, 6)

(3, 5)

(3, 6)

(2, 5)

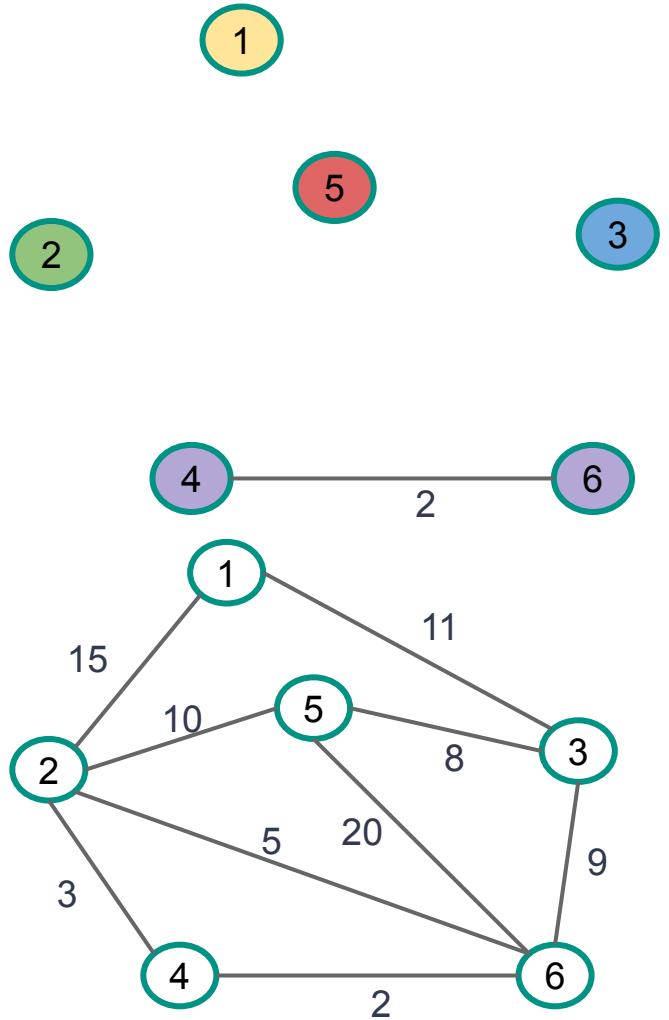
(1, 3)

(1, 2)

(5, 6)

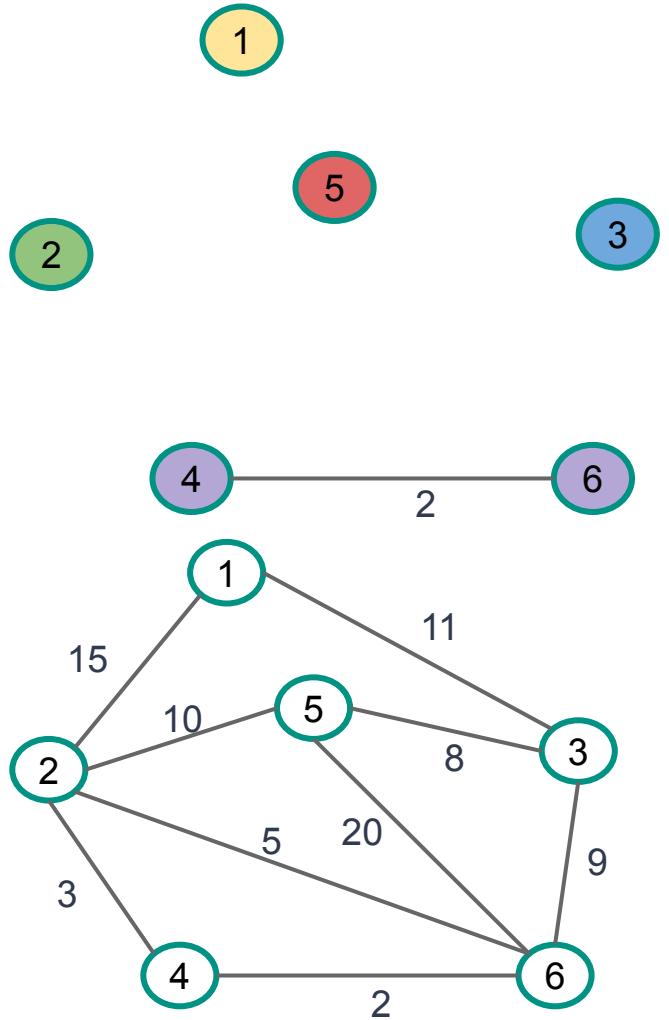
$r = [1, 2, 3, 4, 5, 6]$

$r = [1, 2, 3, 4, 5, 4]$



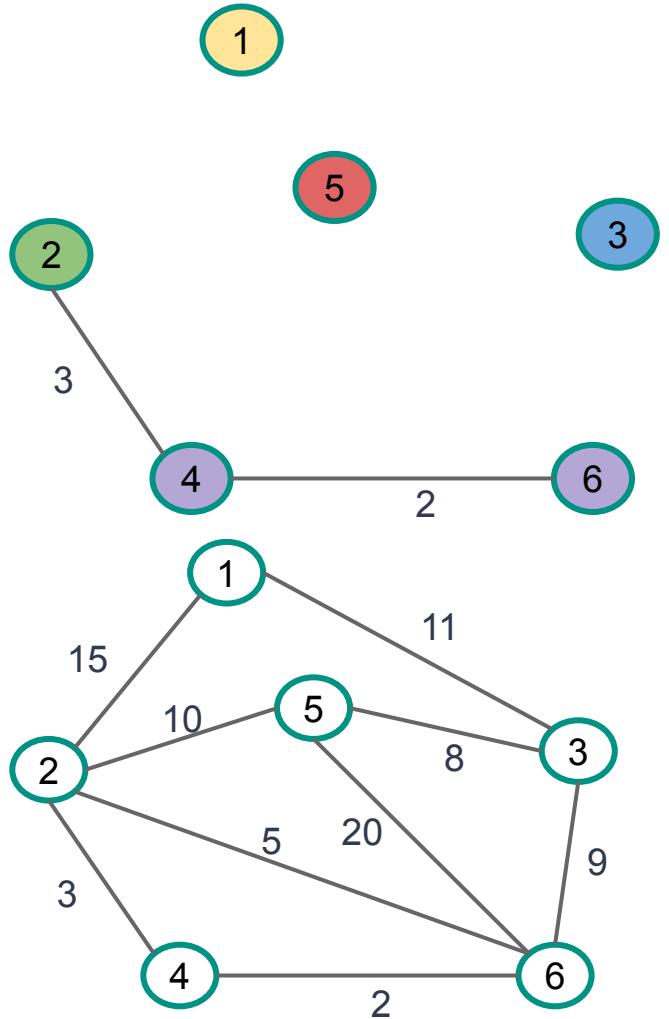
$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, 2, 3, 4, 5, 4]$

(4, 6)  
**(2, 4)**  
(2, 6)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)



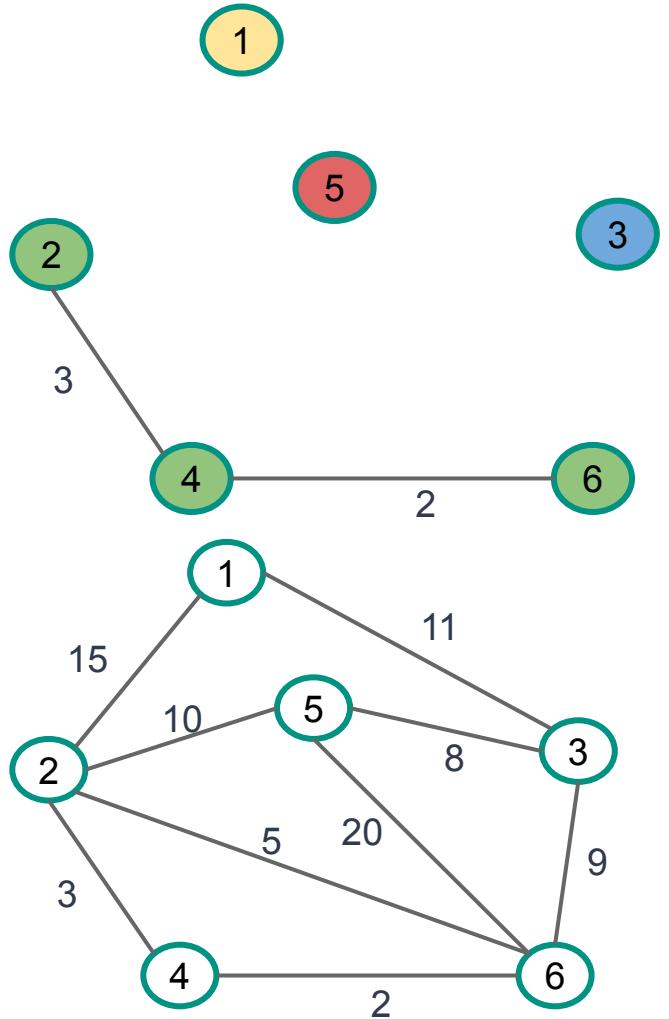
$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, \underline{2}, 3, \underline{4}, 5, 4]$   
 $r(2) \neq r(4)$

(4, 6)  
**(2, 4)**  
(2, 6)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)



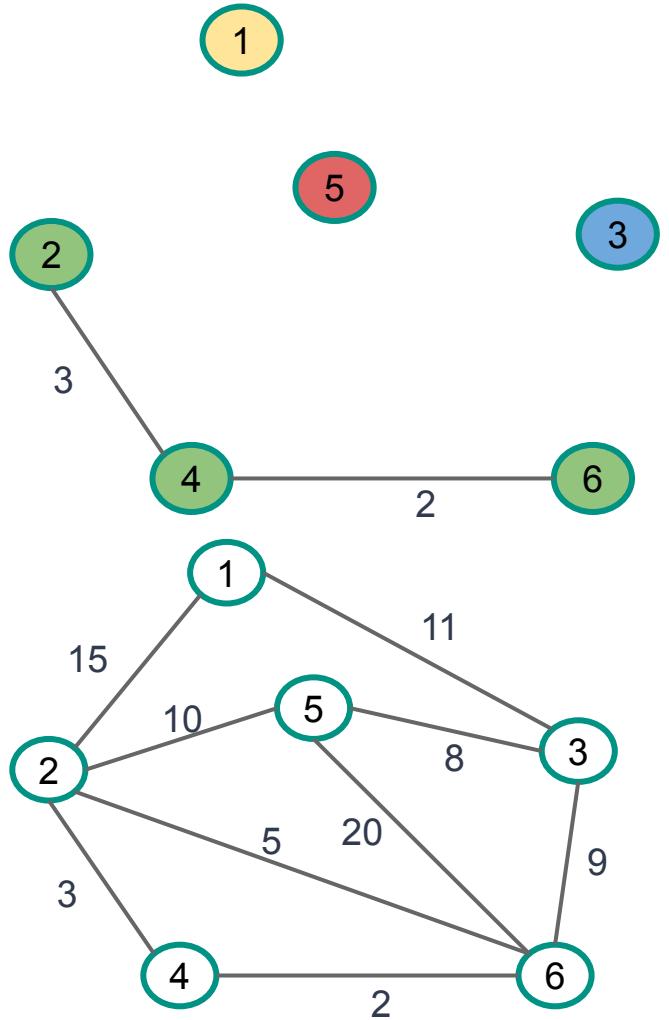
$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, \underline{2}, 3, \underline{4}, 5, 4]$   
 $r(2) \neq r(4)$

(4, 6)  
**(2, 4)**  
(2, 6)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)



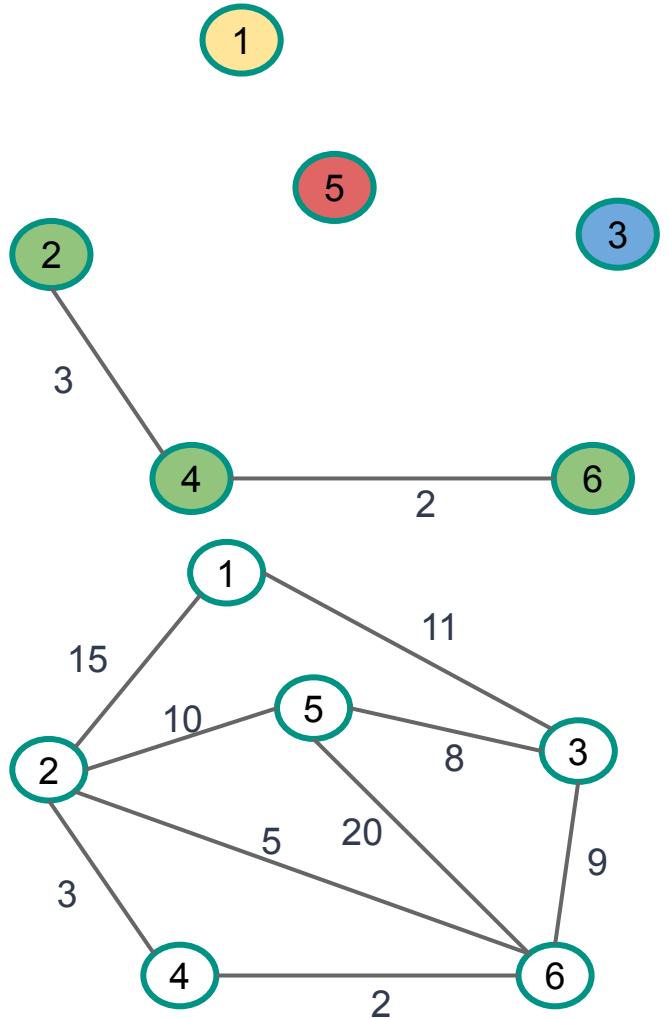
$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, 2, 3, 4, 5, 4]$   
 $r = [1, 2, 3, 2, 5, 2]$

(4, 6)  
**(2, 4)**  
(2, 6)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)

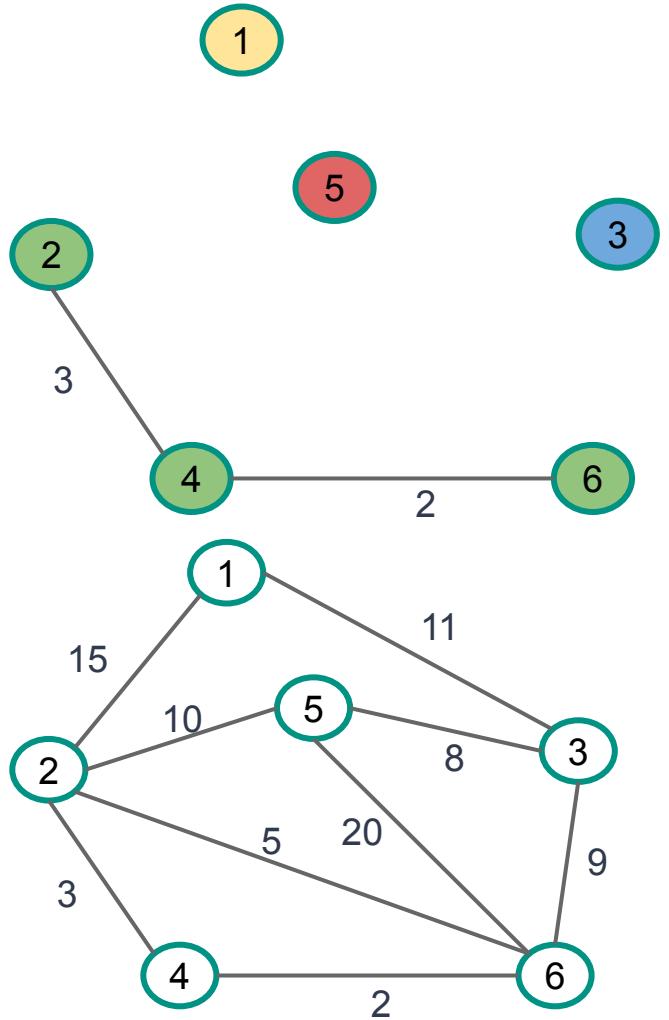


$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, 2, 3, 4, 5, 4]$   
 $r = [1, 2, 3, 2, 5, 2]$

**(2, 6)**  
(4, 6)  
(2, 4)  
(3, 5)  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)

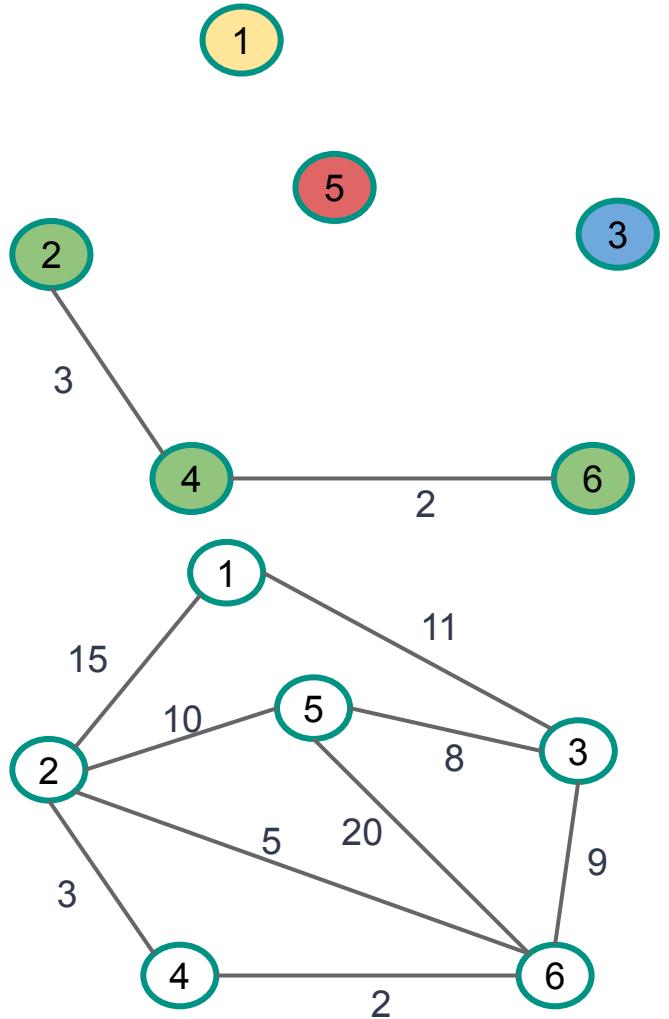


- |  |  |
|--|--|
| $r = [1, 2, 3, 4, 5, 6]$                         | $\begin{array}{c} (4, 6) \\ (2, 4) \\ \textbf{(2, 6)} \\ (3, 5) \\ (3, 6) \\ (2, 5) \\ (1, 3) \\ (1, 2) \\ (5, 6) \end{array}$ |
| $r = [1, 2, 3, 4, 5, 4]$                         |  |
| $r = [1, \underline{2}, 3, 2, 5, \underline{2}]$ |  |
| $r(2) = r(6) \rightarrow \mathbf{NU}$            |  |

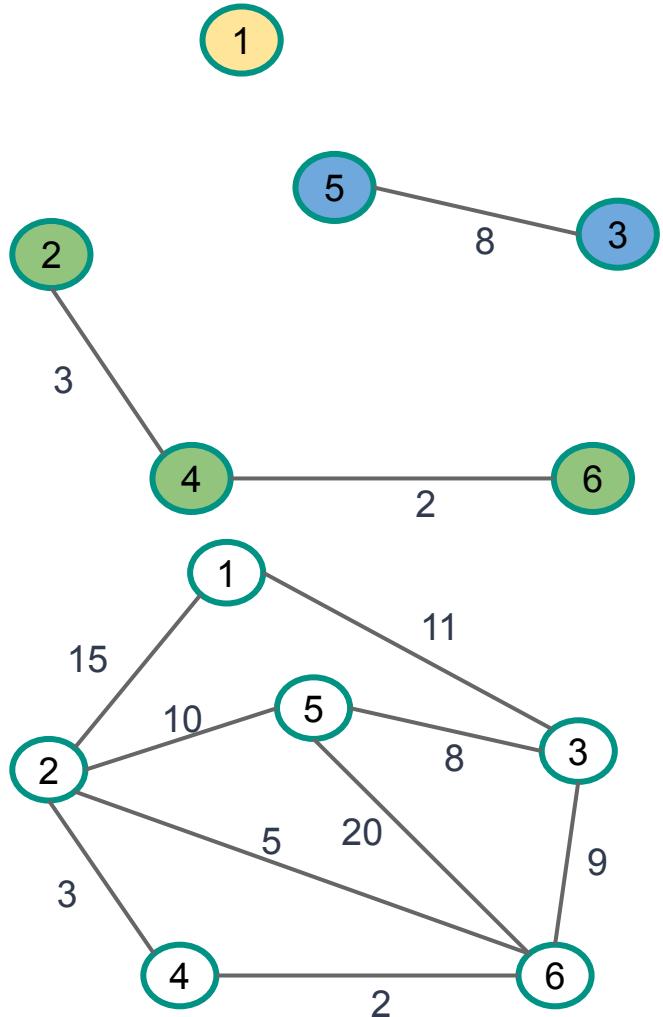


$r = [1, 2, 3, 4, 5, 6]$   
 $r = [1, 2, 3, 4, 5, 4]$   
 $r = [1, 2, 3, 2, 5, 2]$   
 $r(2) = r(6) \rightarrow \mathbf{NU}$

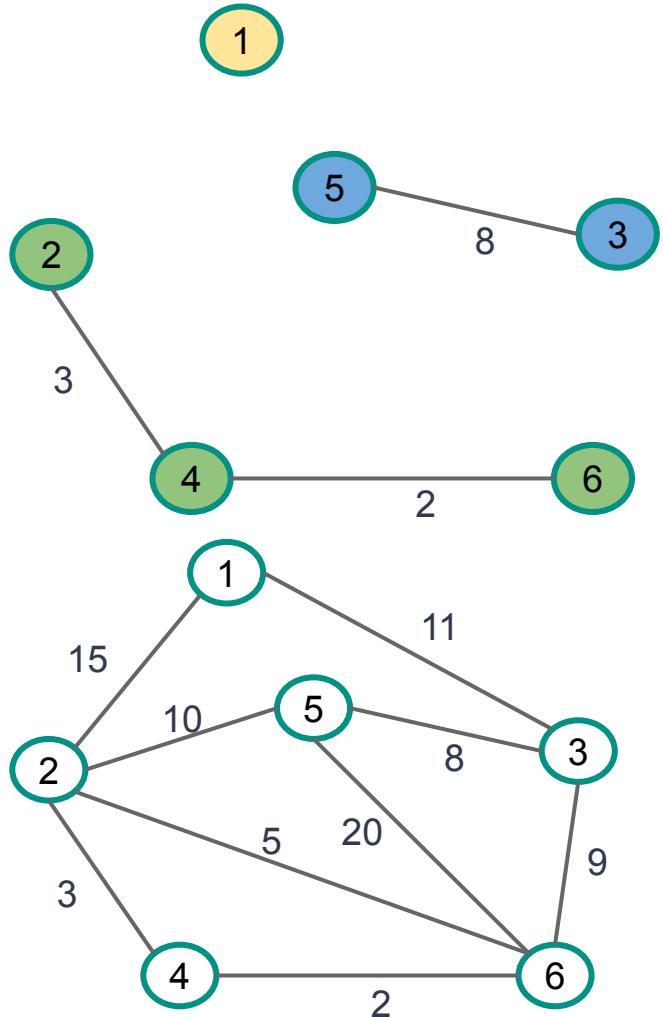
(4, 6)  
(2, 4)  
(2, 6)  
**(3, 5)**  
(3, 6)  
(2, 5)  
(1, 3)  
(1, 2)  
(5, 6)



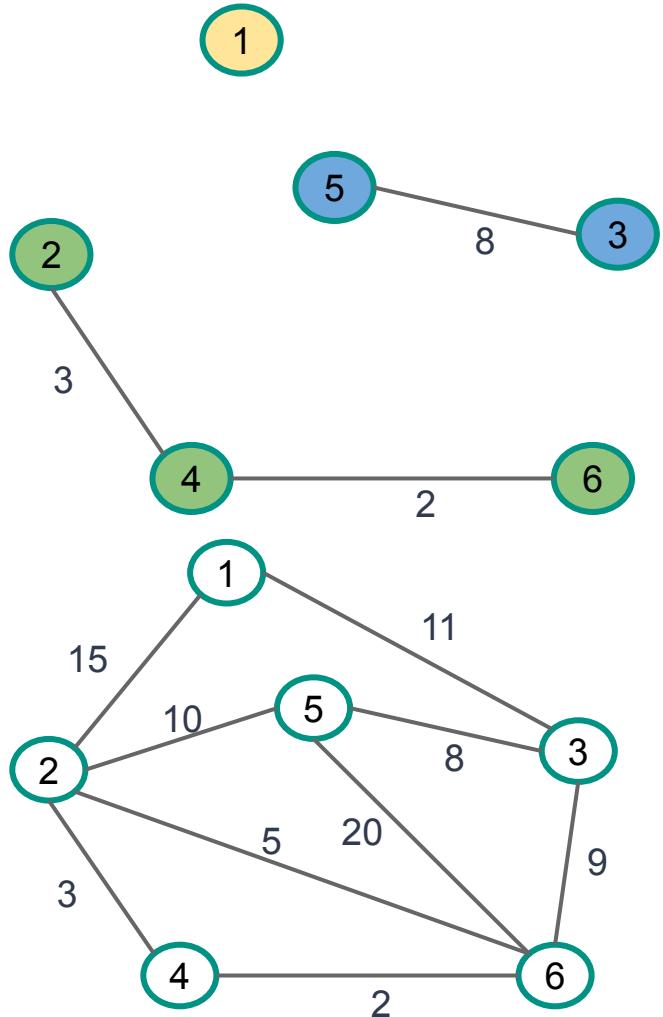
- |  |  |
|--|--|
| $r = [1, 2, 3, 4, 5, 6]$                         |  |
| $r = [1, 2, 3, 4, 5, 4]$                         |  |
| $r = [1, 2, \underline{3}, 2, \underline{5}, 2]$ |  |
| $r(2) = r(6) \rightarrow \mathbf{NU}$            |  |
| $r(3) \neq r(5)$                                 |  |
| $(4, 6)$   |  |
| $(2, 4)$   |  |
| $(2, 6)$   |  |
| $(3, 5)$   |  |
| $(3, 6)$   |  |
| $(2, 5)$   |  |
| $(1, 3)$   |  |
| $(1, 2)$   |  |
| $(5, 6)$   |  |



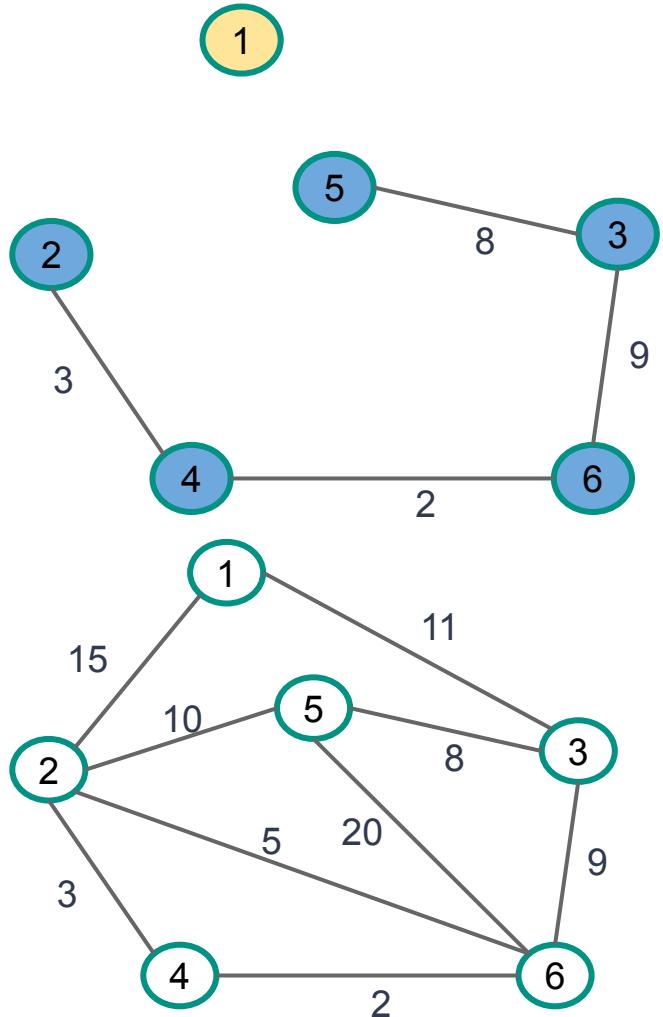
- |                           |                                       |
|---------------------------|---------------------------------------|
| $r = [1, 2, 3, 4, 5, 6]$  |                                       |
| $r = [1, 2, 3, 4, 5, 4]$  |                                       |
| $r = [1, 2, 3, 2, 5, 2]$  |                                       |
| $\textcolor{red}{(2, 6)}$ | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 5)$                  | $r = [1, 2, 3, 2, 3, 2]$              |
| $(3, 6)$                  |                                       |
| $(2, 5)$                  |                                       |
| $(1, 3)$                  |                                       |
| $(1, 2)$                  |                                       |
| $(5, 6)$                  |                                       |



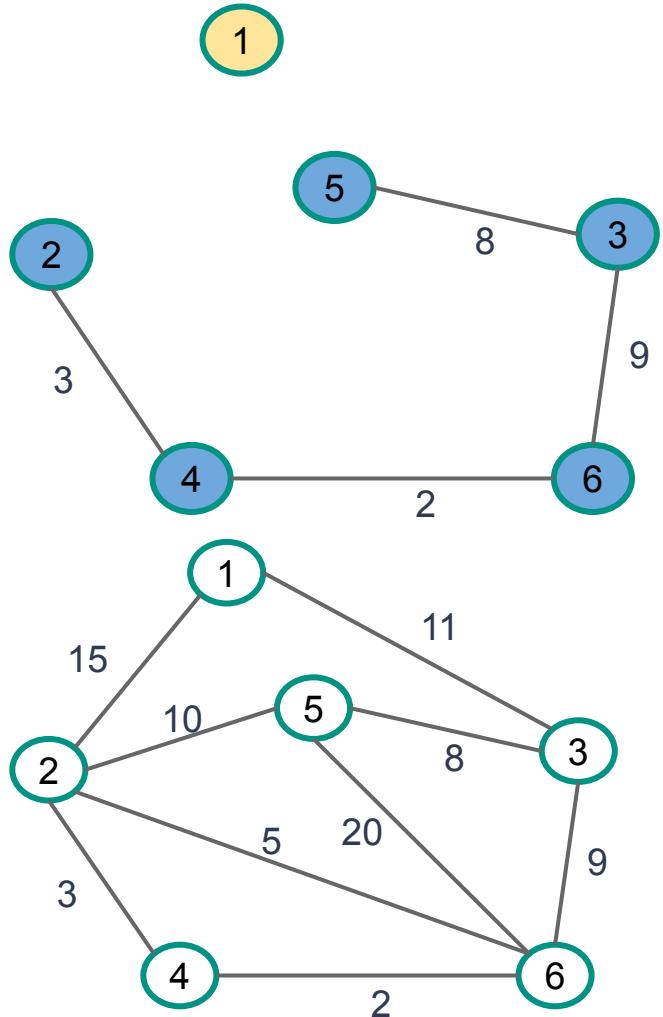
- |                            |                                       |
|----------------------------|---------------------------------------|
| $r = [1, 2, 3, 4, 5, 6]$   |                                       |
| $r = [1, 2, 3, 4, 5, 4]$   |                                       |
| $r = [1, 2, 3, 2, 5, 2]$   |                                       |
| $(2, 6)$                   | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 5)$                   | $r = [1, 2, 3, 2, 3, 2]$              |
| <b><math>(3, 6)</math></b> |                                       |
| $(2, 5)$                   |                                       |
| $(1, 3)$                   |                                       |
| $(1, 2)$                   |                                       |
| $(5, 6)$                   |                                       |



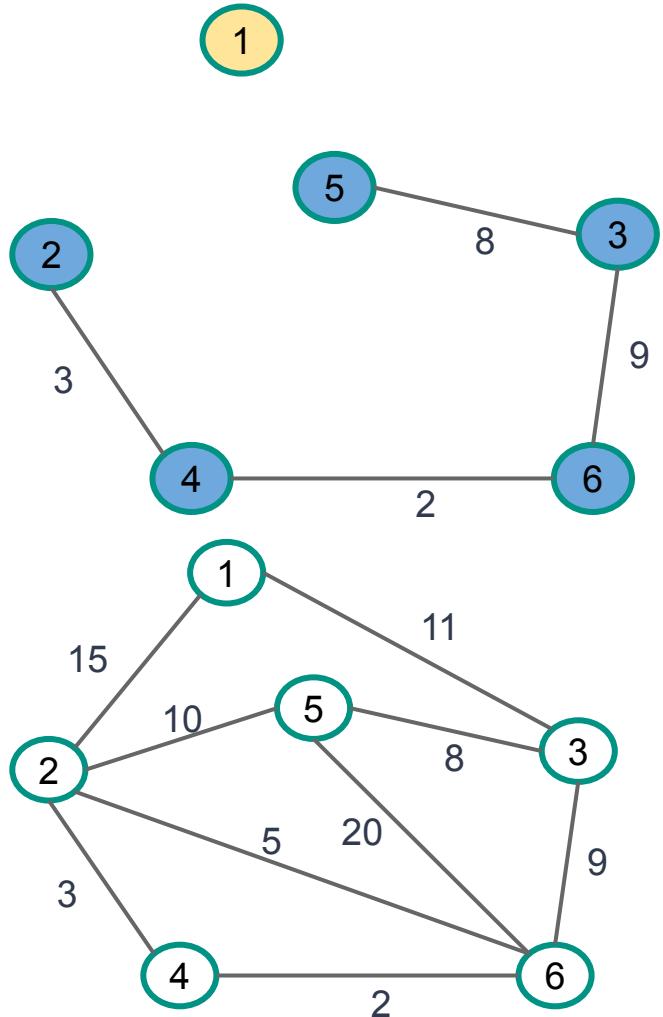
- |          |  |
|----------|--|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$                         |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$                         |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$                         |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \text{NU}$              |
| $(3, 6)$ | $r = [1, 2, \underline{3}, 2, 3, \underline{2}]$ |
| $(2, 5)$ | $r(3) \neq r(6)$                                 |
| $(1, 3)$ |  |
| $(1, 2)$ |  |
| $(5, 6)$ |  |



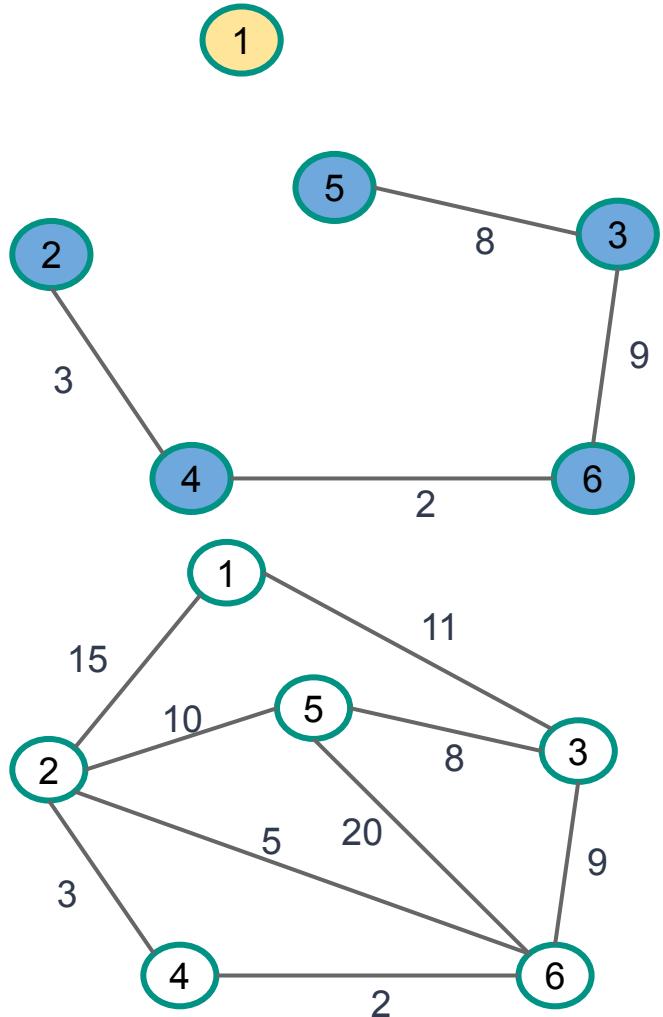
- |                          |                                     |
|--------------------------|-------------------------------------|
| $r = [1, 2, 3, 4, 5, 6]$ |                                     |
| $r = [1, 2, 3, 4, 5, 4]$ |                                     |
| $r = [1, 2, 3, 2, 5, 2]$ |                                     |
| $(2, 6)$                 | $r(2) = r(6) \rightarrow \text{NU}$ |
| $(3, 5)$                 | $r = [1, 2, 3, 2, 3, 2]$            |
| $(3, 6)$                 | $r = [1, 3, 3, 3, 3, 3]$            |
| $(2, 5)$                 |                                     |
| $(1, 3)$                 |                                     |
| $(1, 2)$                 |                                     |
| $(5, 6)$                 |                                     |



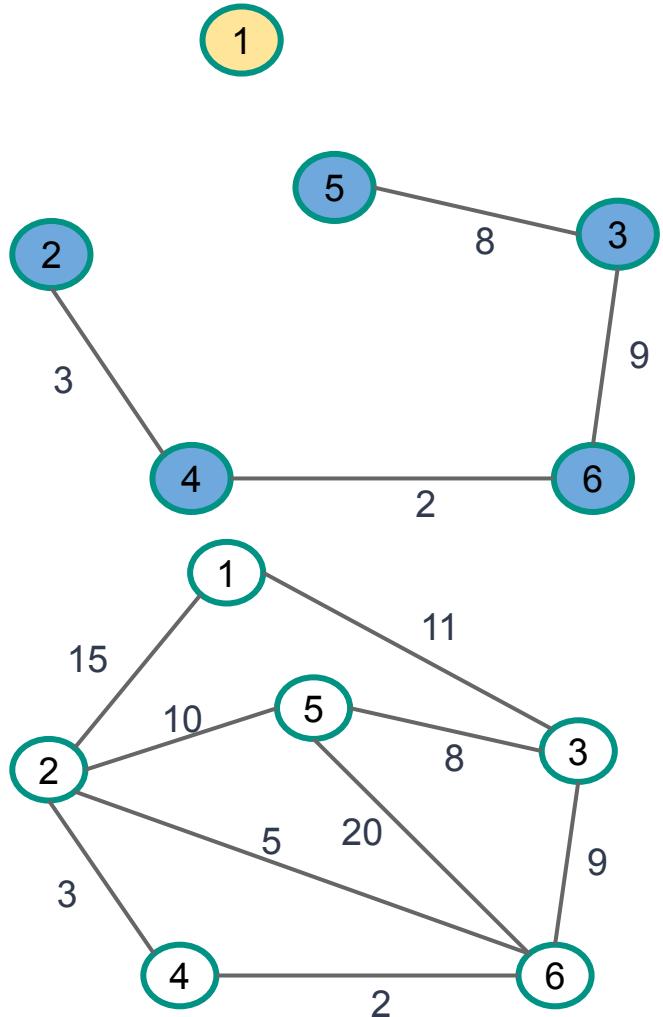
- |          |                                       |
|----------|---------------------------------------|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$              |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$              |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$              |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 6)$ | $r = [1, 2, 3, 2, 3, 2]$              |
| $(2, 5)$ | $r = [1, 3, 3, 3, 3, 3]$              |
| $(1, 3)$ |                                       |
| $(1, 2)$ |                                       |
| $(5, 6)$ |                                       |



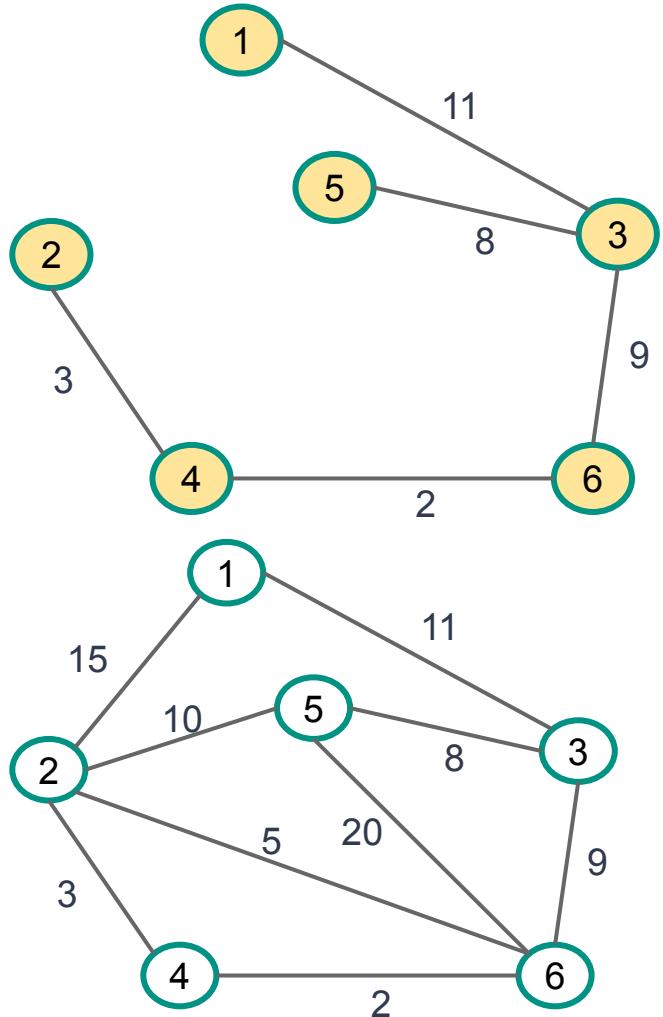
- |          |  |
|----------|--|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$                         |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$                         |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$                         |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \mathbf{NU}$            |
| $(3, 6)$ | $r = [1, 2, 3, 2, 3, 2]$                         |
| $(2, 5)$ | $r = [1, \underline{3}, 3, 3, \underline{3}, 3]$ |
| $(1, 3)$ | $r(2) = r(5) \rightarrow \mathbf{NU}$            |
| $(1, 2)$ |  |
| $(5, 6)$ |  |



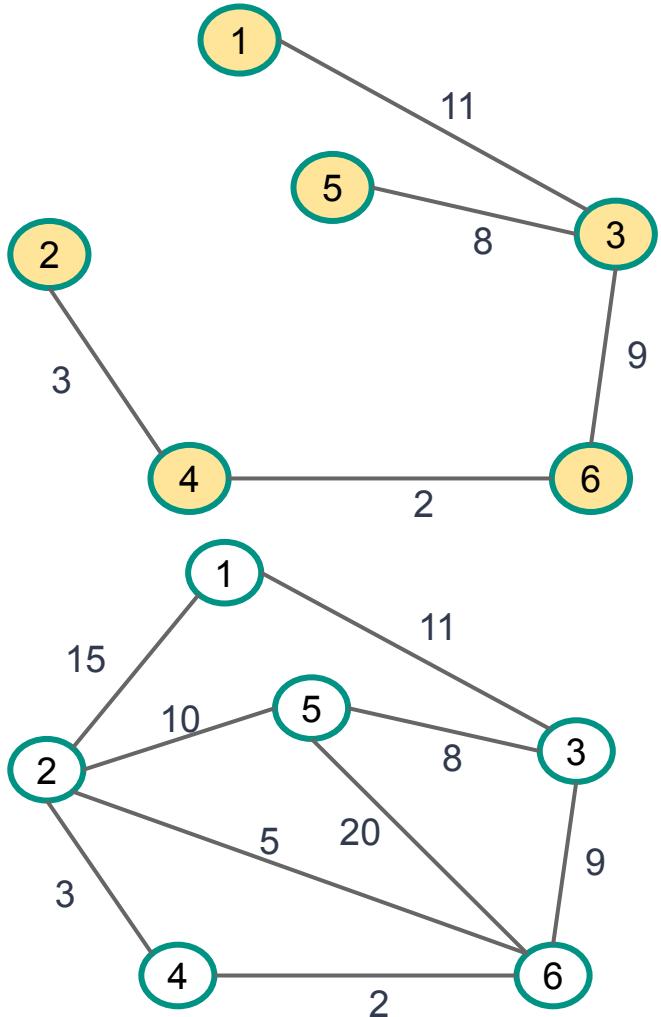
- |          |                                       |
|----------|---------------------------------------|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$              |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$              |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$              |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 6)$ | $r = [1, 2, 3, 2, 3, 2]$              |
| $(2, 5)$ | $r = [1, 3, 3, 3, 3, 3]$              |
| $(1, 3)$ | $r(2) = r(5) \rightarrow \mathbf{NU}$ |
| $(1, 2)$ |                                       |
| $(5, 6)$ |                                       |



- |          |                                       |
|----------|---------------------------------------|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$              |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$              |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$              |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 6)$ | $r = [1, 2, 3, 2, 3, 2]$              |
| $(2, 5)$ | $r = [1, 3, \underline{3}, 3, 3, 3]$  |
| $(1, 3)$ | $r(2) = r(5) \rightarrow \mathbf{NU}$ |
| $(1, 2)$ | $r(1) \neq r(3)$                      |
| $(5, 6)$ |                                       |



- |          |                                       |
|----------|---------------------------------------|
| $(4, 6)$ | $r = [1, 2, 3, 4, 5, 6]$              |
| $(2, 4)$ | $r = [1, 2, 3, 4, 5, 4]$              |
| $(2, 6)$ | $r = [1, 2, 3, 2, 5, 2]$              |
| $(3, 5)$ | $r(2) = r(6) \rightarrow \mathbf{NU}$ |
| $(3, 6)$ | $r = [1, 2, 3, 2, 3, 2]$              |
| $(2, 5)$ | $r = [1, 3, 3, 3, 3, 3]$              |
| $(1, 3)$ | $r(2) = r(5) \rightarrow \mathbf{NU}$ |
| $(1, 2)$ | $r = [1, 1, 1, 1, 1, 1]$              |
| $(5, 6)$ |                                       |



$(4, 6)$	$r = [1,2,3,4,5,6]$
$(2, 4)$	$r = [1,2,3,4,5,4]$
$(2, 6)$	$r = [1,2,3,2,5,2]$
$r(2) = r(6) \rightarrow \mathbf{NU}$	$r(2) = r(6) \rightarrow \mathbf{NU}$
$(3, 5)$	$r = [1,2,3,2,3,2]$
$(3, 6)$	$r = [1,3,3,3,3,3]$
$(2, 5)$	$r(2) = r(5) \rightarrow \mathbf{NU}$
$(1, 3)$	$r = [1,1,1,1,1,1]$
<b><u>STOP</u></b>	
$(1, 2)$	
$(5, 6)$	

# Kruskal



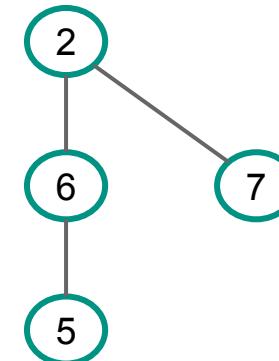
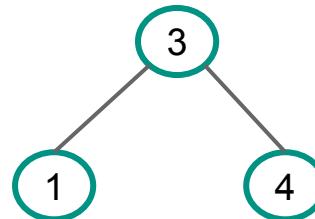
**Varianta 2 - Structuri pentru multimi disjuncte Union / Find**

# Kruskal



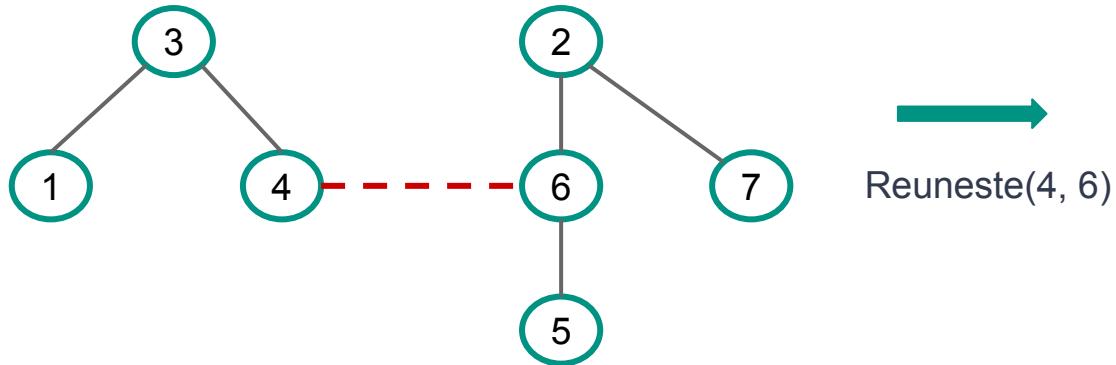
## Varianta 2 - Structuri pentru multimi disjuncte Union / Find - arbori

- memorăm componentele conexe ca arbori, folosind **vectorul tata**
- reprezentantul componentei va fi rădăcina arborelui**



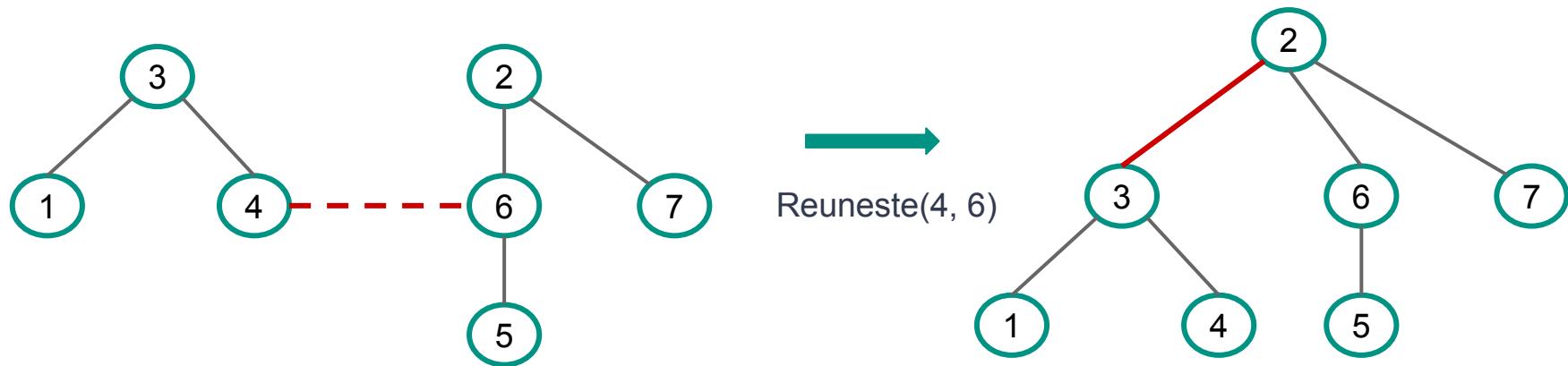
# Kruskal

- **Reuniunea** a doi arbori  $\Rightarrow$  rădăcina unui arbore devine fiu al rădăcinii celuilalt arbore



# Kruskal

- **Reuniunea** se va face în funcție de înălțimea arborilor (reuniune ponderată)



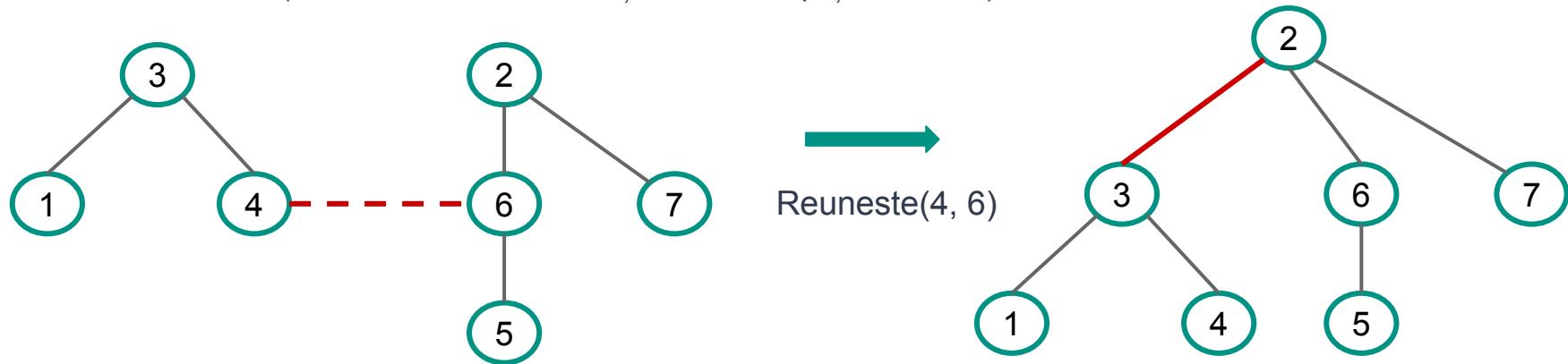
- arborele cu înălțimea mai mică devine subarbore al rădăcinii celuilalt arbore

# Kruskal

- Reuniunea se va face în funcție de înălțimea arborilor (reuniune ponderată)

⇒ **arbori de înălțime logaritmică**

(Inductiv: un arbore de înălțime  $h$  are cel puțin  $2^h$  vârfuri)



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

# Kruskal

**Detalii de implementare operații cu structuri Union / Find pentru multimi disjuncte**

- **Initializare**
- **Reprez(u)** ⇒ determinarea rădăcinii arborelui care conține u
- **Reuneste(u)** ⇒ reuniune ponderată

**ASD + laborator AG**

# Kruskal

```
void Initializare(int u) {
```

```
}
```

```
int Reprez(int u) {
```

```
}
```

# Kruskal

```
void Initializare(int u) {  
    tata[u] = h[u] = 0;  
}
```

```
int Reprez(int u) {
```

```
}
```

# Kruskal

```
void Initializare(int u) {  
    tata[u] = h[u] = 0;  
}  
  
int Reprez(int u) {  
    while (tata[u] != 0)  
        u = tata[u];  
    return u;  
}
```

# Kruskal

```
void Initializare(int u) {  
    tata[u] = h[u] = 0;  
}
```

```
int Reprez(int u) {  
    while (tata[u] != 0)  
        u = tata[u];  
    return u;  
}
```

```
void Reuneste(int u, int v) {
```

```
}
```

# Kruskal

```
void Initializare(int u) {  
    tata[u] = h[u] = 0;  
}  
  
int Reprez(int u) {  
    while (tata[u] != 0)  
        u = tata[u];  
    return u;  
}
```

```
void Reuneste(int u, int v) {  
    int ru, rv;  
    ru = Reprez(u);  
    rv = Reprez(v);  
    if (h[ru] > h[rv])  
        ru = rv;  
    h[ru] += h[rv];  
    tata[rv] = ru;  
}  
}
```

# Kruskal

```
void Initializare(int u) {  
    tata[u] = h[u] = 0;  
}  
  
int Reprez(int u) {  
    while (tata[u] != 0)  
        u = tata[u];  
    return u;  
}
```

```
void Reuneste(int u, int v) {  
    int ru, rv;  
    ru = Reprez(u);  
    rv = Reprez(v);  
    if (h[ru] > h[rv])  
        tata[rv] = ru;  
    else {  
        tata[ru] = rv;  
    }  
}
```

# Kruskal

```
void Initializare(int u) {
    tata[u] = h[u] = 0;
}

int Reprez(int u) {
    while (tata[u] != 0)
        u = tata[u];
    return u;
}
```

```
void Reuneste(int u, int v) {
    int ru, rv;
    ru = Reprez(u);
    rv = Reprez(v);
    if (h[ru] > h[rv])
        tata[rv] = ru;
    else {
        tata[ru] = rv;
        if (h[ru] == h[rv])
            h[rv] = h[rv] + 1;
    }
}
```

# Kruskal

## Complexitate

**Varianta 2** - dacă folosim arbori Union / Find

- Sortare** →  $O(m \log m) = O(m \log n)$
  - $n * \text{Initializare}$**  →  $O(n)$
  - $2m * \text{Reprez}$**  →
  - $(n-1) * \text{Reuneste}$**  →
-

# Kruskal

## Complexitate

**Varianta 2** - dacă folosim arbori Union / Find

- Sortare** →  $O(m \log m) = O(m \log n)$
  - $n * \text{Initializare}$**  →  $O(n)$
  - $2m * \text{Reprez}$**  →  $O(m \log n)$
  - $(n-1) * \text{Reuneste}$**  →  $O(n \log n)$
-

# Kruskal

## Complexitate

**Varianta 2** - dacă folosim arbori Union / Find

- Sortare** →  $O(m \log m) = O(m \log n)$
- $n * \text{Initializare}$**  →  $O(n)$
- $2m * \text{Reprez}$**  →  $O(m \log n)$
- $(n-1) * \text{Reuneste}$**  →  $O(n \log n)$

---

**$O(m \log n)$**

# Kruskal

**Concluzii complexitate -  $O(m \log n)$**

# Aplicații - Clustering

**Gruparea unor obiecte în k clase cât mai *bine separate* (k dat)**

- obiecte din clase diferite să *fie cât mai diferite*

# Aplicații - Clustering

**Gruparea unor obiecte în k clase cât mai bine separate (k dat)**

- obiecte din clase diferite să fie cât mai diferențiate

**Exemplu:** k = 3, multime de cuvinte:

sinonim, ana, apa, care, martian, este, case, partial, arbore, minim

⇒ 3 clase



# Aplicații - Clustering

**Gruparea unor obiecte în k clase cât mai bine separate (k dat)**

- obiecte din clase diferite să fie cât mai diferențiate

**Exemplu:** k = 3, multime de cuvinte:

sinonim, ana, apa, care, martian, este, case, partial, arbore, minim

⇒ 3 clase



**Sunt necesare (se dau):**

- Criteriu de "asemănare" între 2 obiecte ⇒ o distanță
- Măsură a gradului de separare a claselor

# Clustering

## Cadru formal

Se dau:

- O mulțime de **n obiecte**  $S = \{o_1, \dots, o_n\}$ 
  - cuvinte, imagini, fișiere, specii de animale etc
- O funcție de **distanță**  $d : S \times S \rightarrow \mathbb{R}_+$ 
  - $d(o_i, o_j) = \text{gradul de asemănare între } o_i \text{ și } o_j$
- $k$  - un număr natural
  - $k = \text{numărul de clase}$

# Clustering

## Definiții

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

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

# Clustering

## Definiții

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

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

**Gradul de separare** a lui  $\mathcal{C}$

= distanță minimă dintre două obiecte aflate în clase diferite

= distanță minimă dintre două clase ale lui  $\mathcal{C}$

**sep( $\mathcal{C}$ )** =  $\min \{ d(o, o') \mid o, o' \in S, o \text{ și } o' \text{ sunt în clase diferite ale lui } \mathcal{C} \}$

=  $\min \{ d(C_i, C_j) \mid i \neq j \in \{1, \dots, k\} \}$

# Clustering

- **obiecte = cuvinte**
- **d = distanță de editare**       $d(\text{ana}, \text{care}) = 3$ :  $\text{ana} \rightarrow \text{cana} \rightarrow \text{cara} \rightarrow \text{care}$
- **k = 3**

A 3x3 grid of words used for clustering:

este	martian	care
ana	apa	sinonim
minim	partial	case

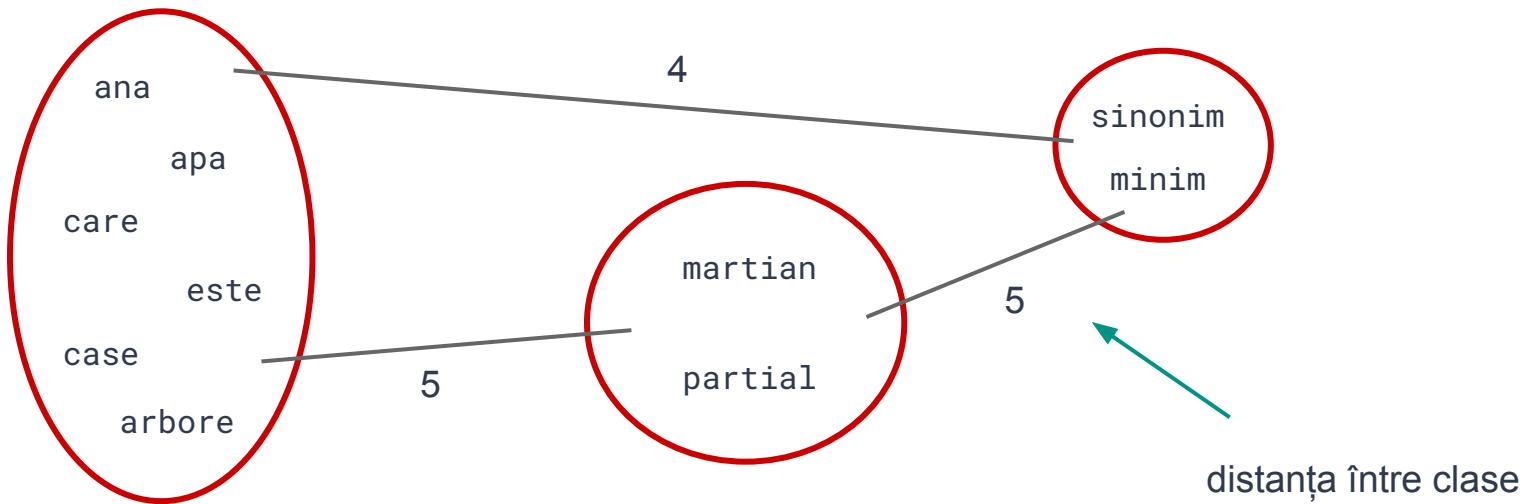
The words are arranged as follows:

- Row 1: este, martian, care
- Row 2: ana, apa, sinonim
- Row 3: minim, partial, case

Arbore is located at the bottom center of the grid.

# Clustering

- ❑ obiecte = cuvinte
- ❑  $d$  = distanță de editare
- ❑  $k = 3$



**3-clustering cu gradul de separare = 4**

# Clustering

**Problemă de Clustering:**

**Date  $S$ ,  $d$  și  $k$ , să se determine un  $k$ -clustering cu **grad de separare maxim**.**

# Clustering



Idee:

este

martian

care

ana

apa

sinonim

minim

partial

case

arbore

# Clustering



## 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

# Clustering



## 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

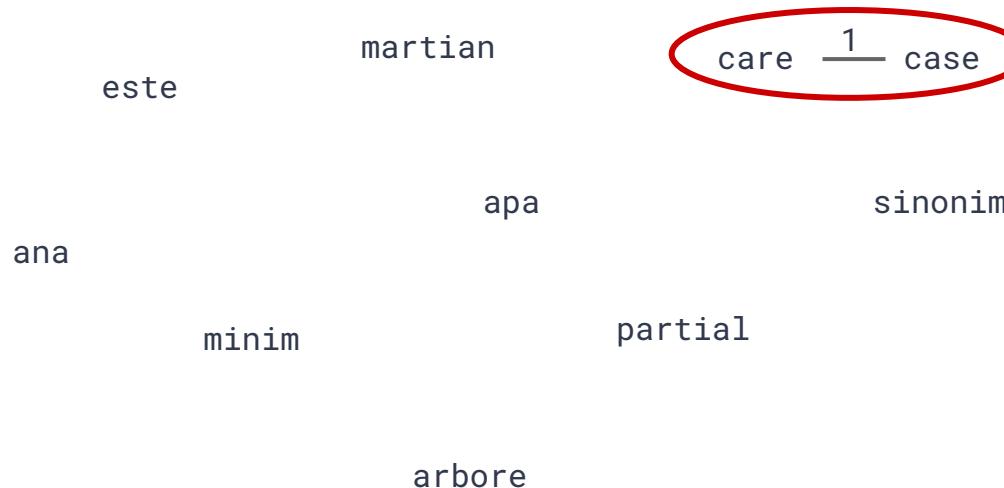
# Clustering

## Cuvinte - distanță de editare

**k = 3 clustere**

# Clustering

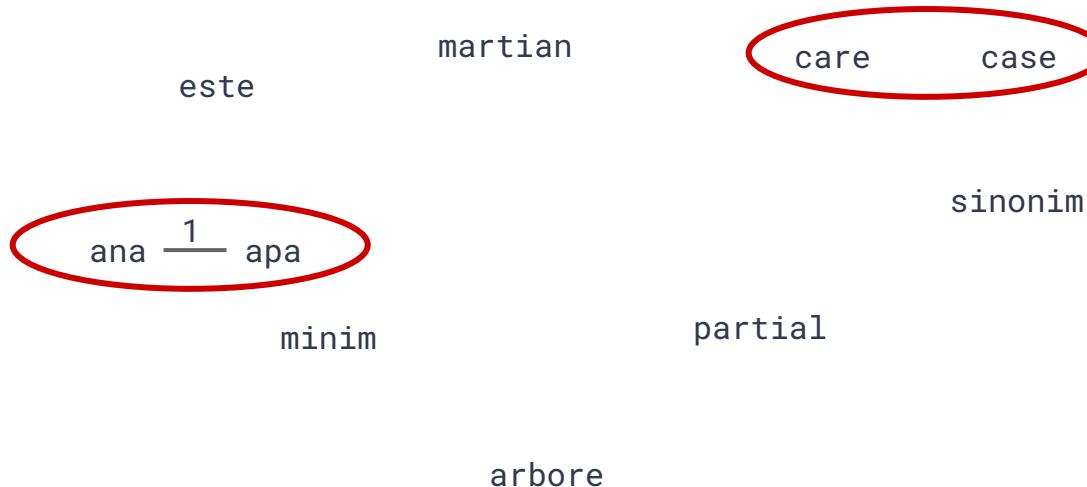
## Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

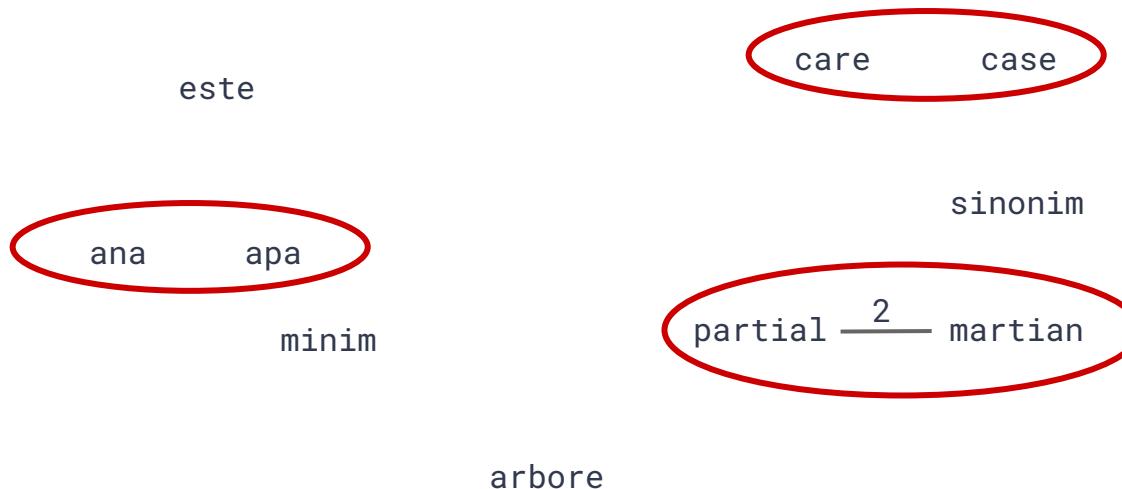
## Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

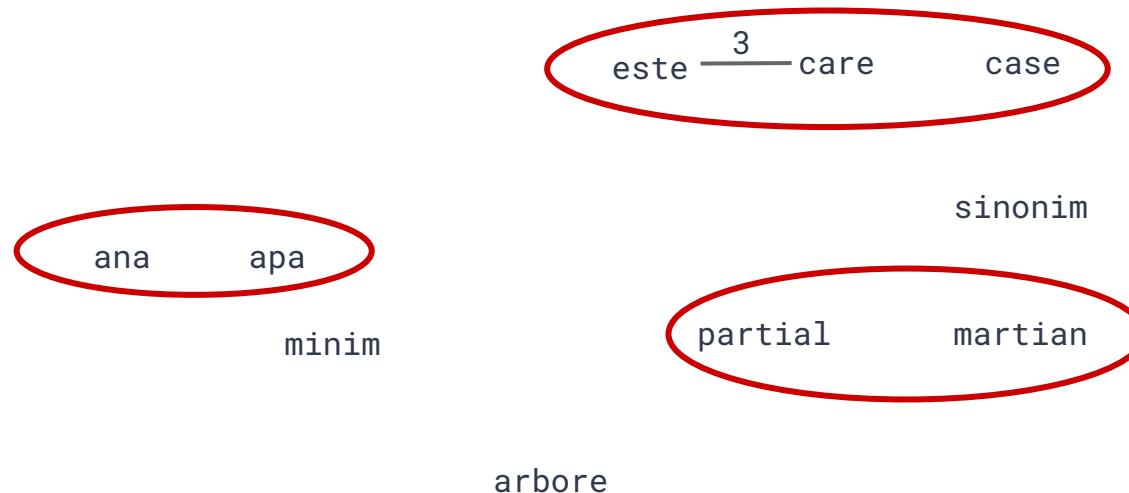
## Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

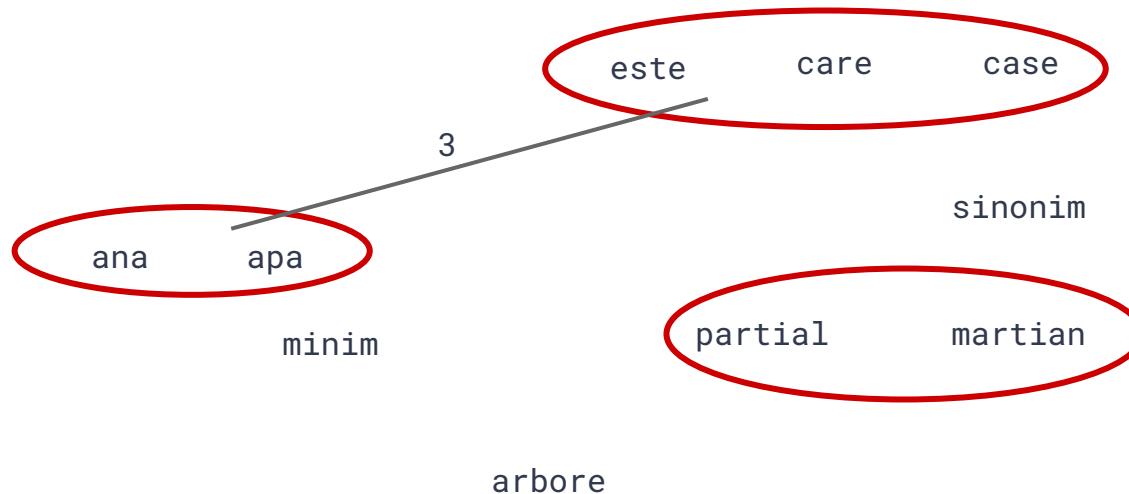
Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

## Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

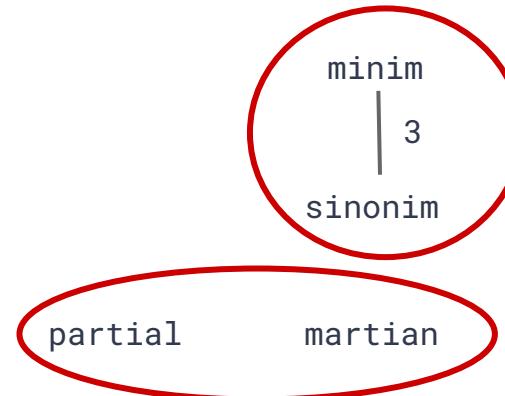
Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

Cuvinte - distanță de editare

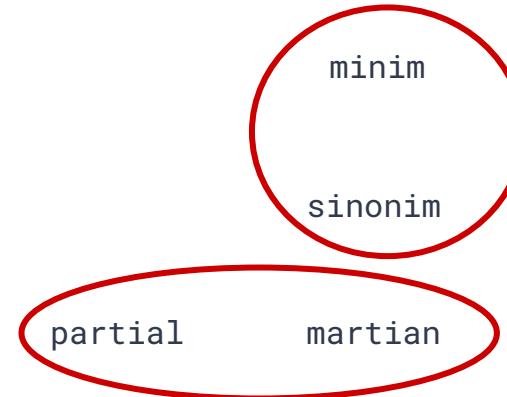


arbore

**k = 3 clustere**

# Clustering

Cuvinte - distanță de editare

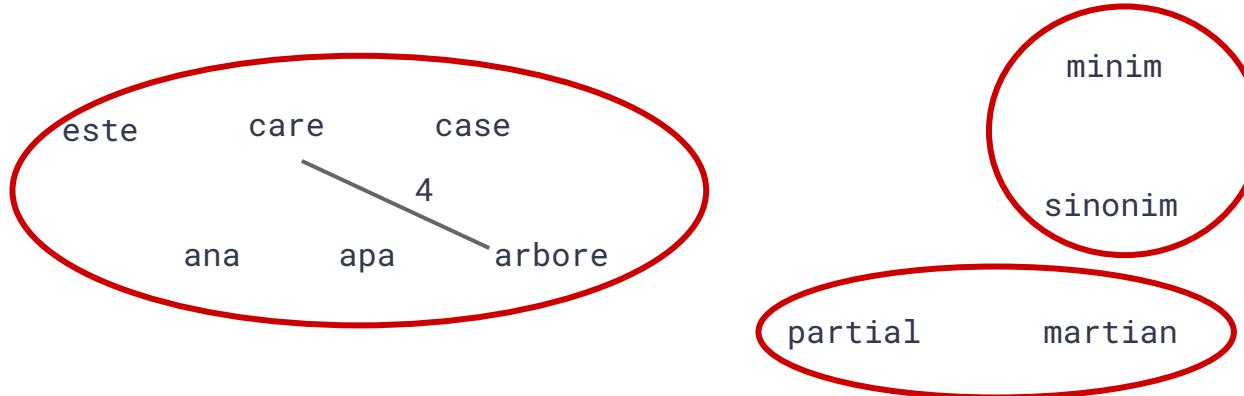


arbore

**k = 3 clustere**

# Clustering

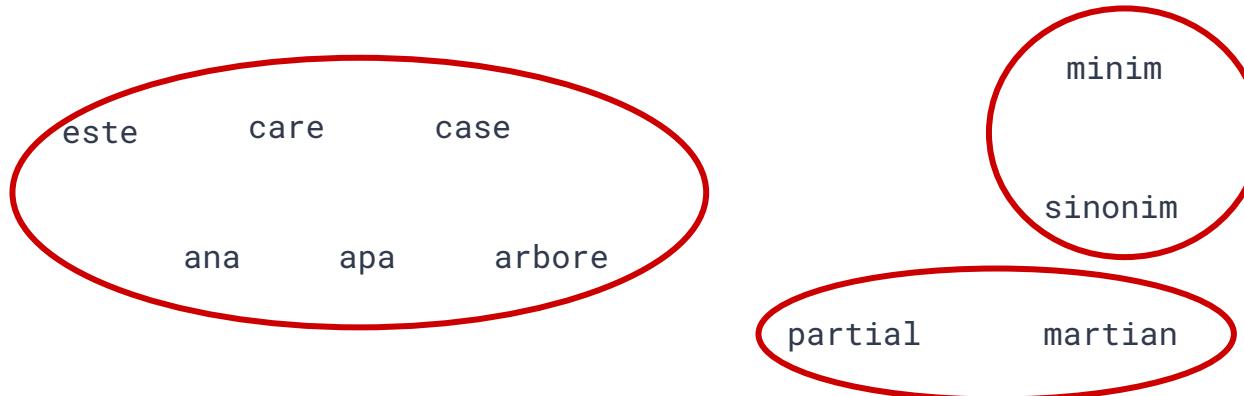
Cuvinte - distanță de editare



**k = 3 clustere**

# Clustering

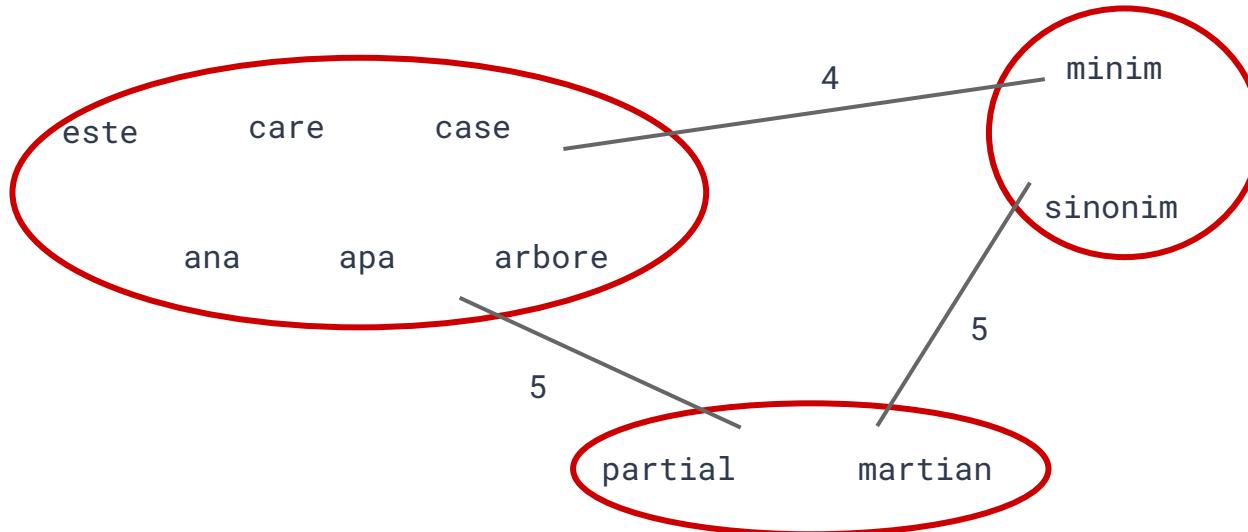
## Cuvinte - distanță de editare



Soluția cu  $k = 3$  clustere

# Clustering

## Cuvinte - distanță de editare



**Grad de separare = 4**

# Clustering

## Pseudocod

- Inițial, fiecare obiect formează o clasă
- pentru  $i = 1, n-k$ 
  - alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_t, o_r)$  minimă
  - reunește (clasa lui  $o_r, o_t$ )
- afișează cele k clase obținute

# Clustering

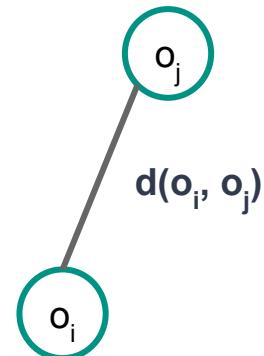
## Pseudocod

- Inițial, fiecare obiect formează o clasă
- pentru  $i = 1, n-k$ 
  - alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_t, o_r)$  minimă
  - reunește (clasa lui  $o_r, o_t$ )
- afișează cele k clase obținute



**Modelare cu graf ponderat (complet)**

**⇒  $n - k$  pași din algoritmul lui Kruskal**



# Clustering

## Pseudocod:

Inițial, fiecare obiect (cuvânt) formează o clasă

pentru  $i = 1, n-k$

- alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_r, o_t)$  minimă
- reunește clasa lui  $o_r$  și clasa lui  $o_t$

returneză cele k clase obținute

## Pseudocod - modelare cu graf complet G:

$$V = \{o_1, \dots, o_n\}, \quad w(o_i o_j) = d(o_i, o_j)$$

# Clustering

## Pseudocod:

Inițial, fiecare obiect (cuvânt) formează o clasă

pentru  $i = 1, n-k$

- alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_r, o_t)$  minimă
- reunește clasa lui  $o_r$  și clasa lui  $o_t$

returneză cele k clase obținute

## Pseudocod - modelare cu graf complet G:

$$V = \{o_1, \dots, o_n\}, \quad w(o_i, o_j) = d(o_i, o_j)$$

Inițial, fiecare vîrf formează o componentă conexă (clasă):  $T' = (V, \emptyset)$

# Clustering

## Pseudocod:

Inițial, fiecare obiect (cuvânt) formează o clasă

pentru  $i = 1, n-k$

- alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_r, o_t)$  minimă
- reunește clasa lui  $o_r$  și clasa lui  $o_t$

returneză cele k clase obținute

## Pseudocod - modelare cu graf complet G:

$$V = \{o_1, \dots, o_n\}, \quad w(o_i, o_j) = d(o_i, o_j)$$

Inițial, fiecare vîrf formează o componentă conexă (clasă):  $T' = (V, \emptyset)$

pentru  $i = 1, n-k$

- alege o muchie  $e_i=uv$  de **cost minim** din G astfel încât **u și v sunt în componente conexe diferite ale lui T'**
- reunește componenta lui u și componenta lui v:  
 $E(T') = E(T') \cup \{uv\}$

# Clustering

## Pseudocod:

Inițial, fiecare obiect (cuvânt) formează o clasă

pentru  $i = 1, n-k$

- alege două obiecte  $o_r, o_t$  din clase diferite, cu  $d(o_r, o_t)$  minimă
- reunește clasa lui  $o_r$  și clasa lui  $o_t$

returneză cele k clase obținute

## Pseudocod - modelare cu graf complet G:

$$V = \{o_1, \dots, o_n\}, \quad w(o_i, o_j) = d(o_i, o_j)$$

Inițial, fiecare vîrf formează o componentă conexă (clasă):  $T' = (V, \emptyset)$

pentru  $i = 1, n-k$

- alege o muchie  $e_i=uv$  de **cost minim** din G astfel încât **u și v sunt în componente conexe diferite** ale lui  $T'$
- reunește componenta lui u și componenta lui v:  
 $E(T') = E(T') \cup \{uv\}$

returnează cele k mulțimi formate cu vârfurile celor k componente conexe ale lui  $T'$

# Clustering

**Observație:** Algoritmul este echivalent cu următorul mai general

- **determinăm un apcm**  $T$  al grafului complet  $G$

# Clustering

**Observație:** Algoritmul este echivalent cu următorul mai general

- determinăm un apcm**  $T$  al grafului complet  $G$
- considerăm multimea  $\{e_{n-k+1}, \dots, e_{n-1}\}$  formată cu  $k-1$  muchii cu **cele mai mari ponderi** în  $T$
- fie pădurea  $T' = T - \{e_{n-k+1}, \dots, e_{n-1}\}$

# Clustering

**Observație:** Algoritmul este echivalent cu următorul mai general

- determinăm un apcm  $T$  al grafului complet  $G$
- considerăm multimea  $\{e_{n-k+1}, \dots, e_{n-1}\}$  formată cu  $k-1$  muchii cu **cele mai mari ponderi** în  $T$
- fie pădurea  $T' = T - \{e_{n-k+1}, \dots, e_{n-1}\}$
- definește clasele  $k$ -clustering-ului  $\mathcal{C}$  ca fiind mulțimile vârfurilor celor  $k$  componente conexe ale pădurii astfel obținute

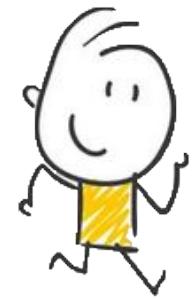
# Clustering

## Corectitudine - v. curs

- k-clustering-ul obținut are grad de separare maxim

Jon Kleinberg, Éva Tardos , Algorithm Design , Addison Wesley 2005, **Secțiunea 4.7**

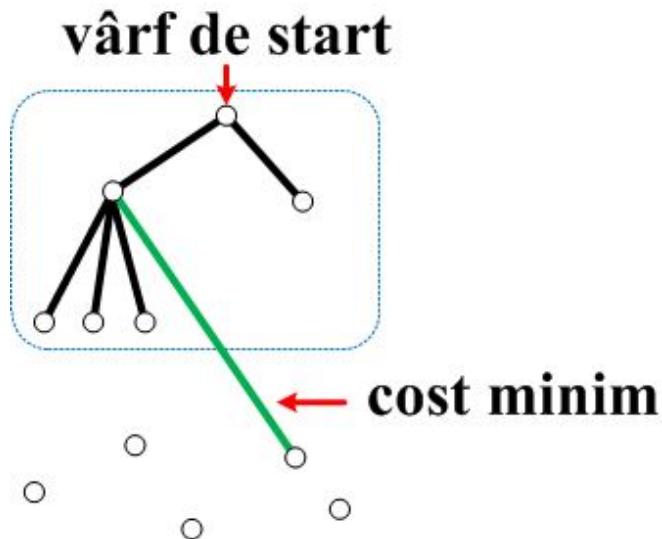
<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsII-2x2.pdf>



# Algoritmul lui Prim

# 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 în arbore, la un vârf neadăugat



# O primă formă a algoritmului

## KRUSKAL

- Inițial:**  $T = (V, \emptyset)$
- pentru**  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim din  $G$**  a.î.  $u, v$  sunt în **componente conexe diferite** ( $T + uv$  aciclic)
  - $E(T) = E(T) \cup \{uv\}$

## PRIM

- s - vârful de start**
- Inițial,  $T = (\{s\}, \emptyset)$**

# O primă formă a algoritmului

## KRUSKAL

- Inițial:**  $T = (V, \emptyset)$
- pentru**  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim din  $G$**  a.î.  $u, v$  sunt în **componente conexe diferite** ( $T + uv$  aciclic)
  - $E(T) = E(T) \cup \{uv\}$

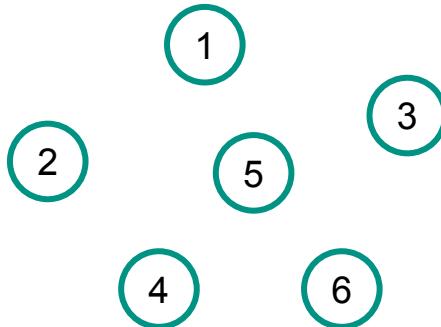
## PRIM

- s - vârful de start**
- Inițial,  $T = (\{s\}, \emptyset)$**
- pentru**  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim din  $G$**  a.î.  $u \in V(T)$  și  $v \notin V(T)$
  - $V(T) = V(T) \cup \{v\}$
  - $E(T) = E(T) \cup uv$

# O primă formă a algoritmului

## KRUSKAL

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



- Se încearcă unirea acestor componente prin muchii de cost minim

## PRIM

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



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

# O primă formă a algoritmului

## KRUSKAL

**La un pas:**

- muchiile selectate formează o **pădure**

## PRIM

**La un pas:**

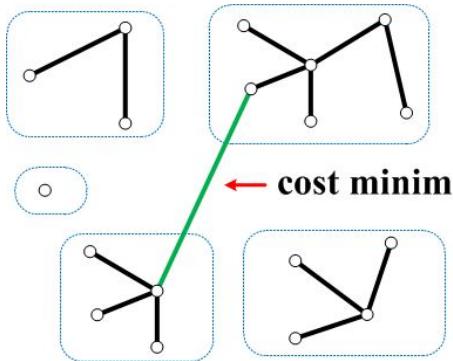
- muchiile selectate formează un **arbore**

# O primă formă a algoritmului

## KRUSKAL

### La un pas:

- muchiile selectate formează o pădure

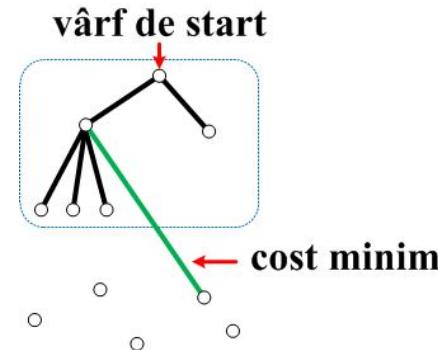


Este selectată o muchie de cost minim, care unește doi arbori din pădurea curentă (două componente conexe).

## PRIM

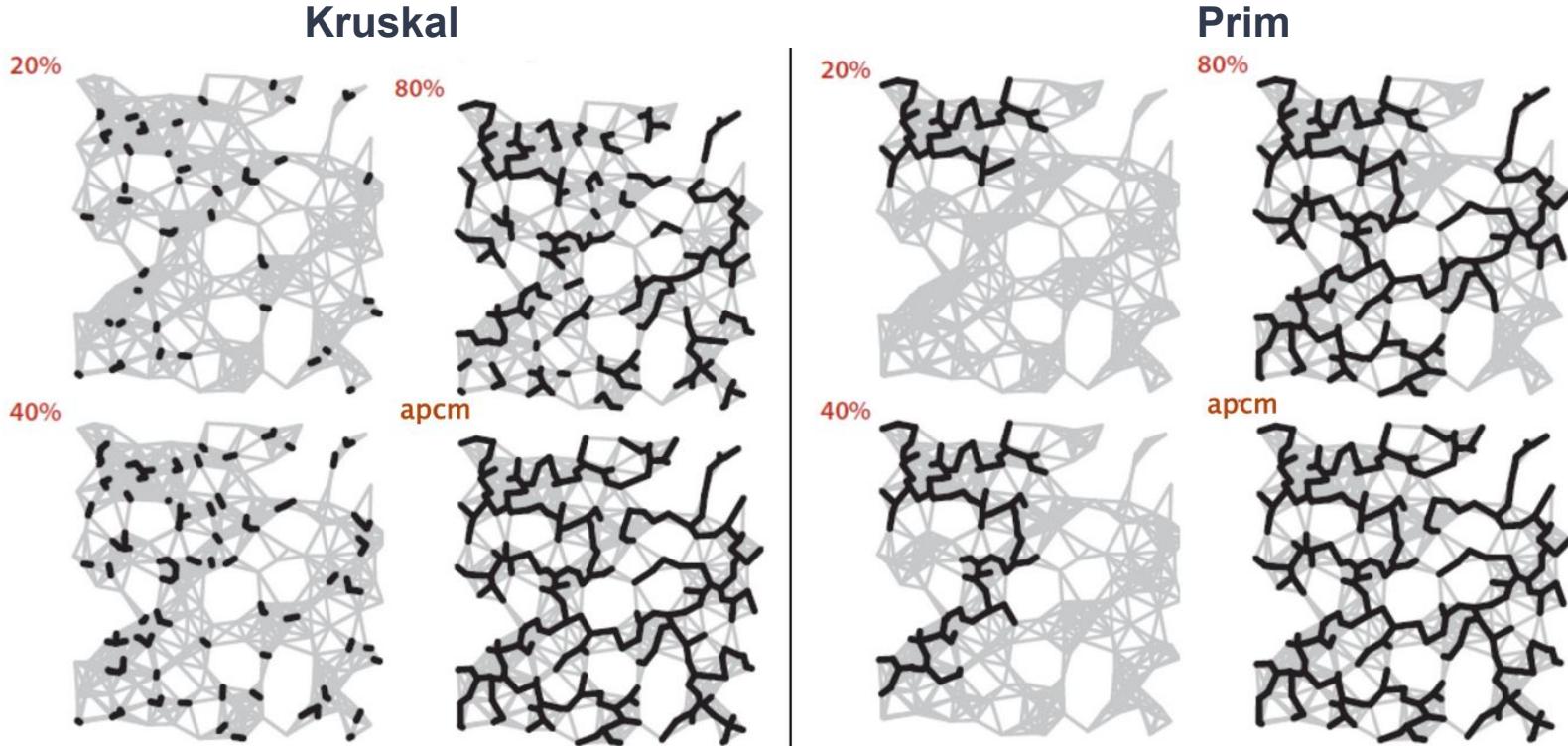
### La un pas:

- muchiile selectate formează un arbore



Este selectată o muchie de cost minim, care unește un vârf din arbore cu unul care nu este în arbore (neselectat).

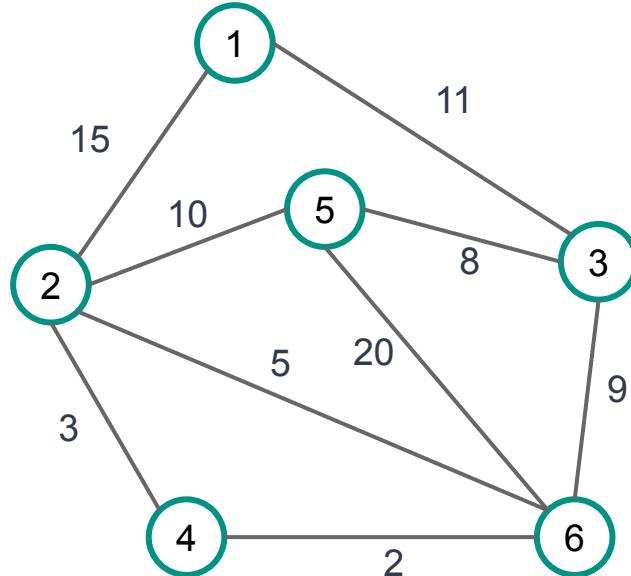
# Arbore parțiali de cost minim



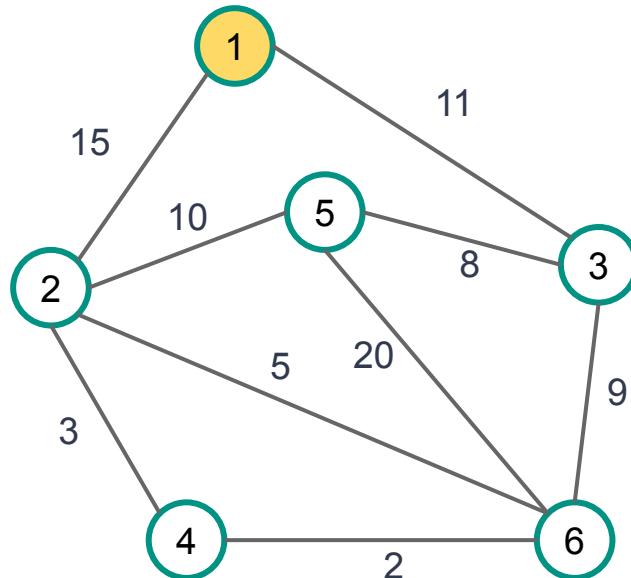
Imagine din

R. Sedgewick, K. Wayne – **Algorithms**, 4th edition, Pearson Education, 2011

# Exemplu

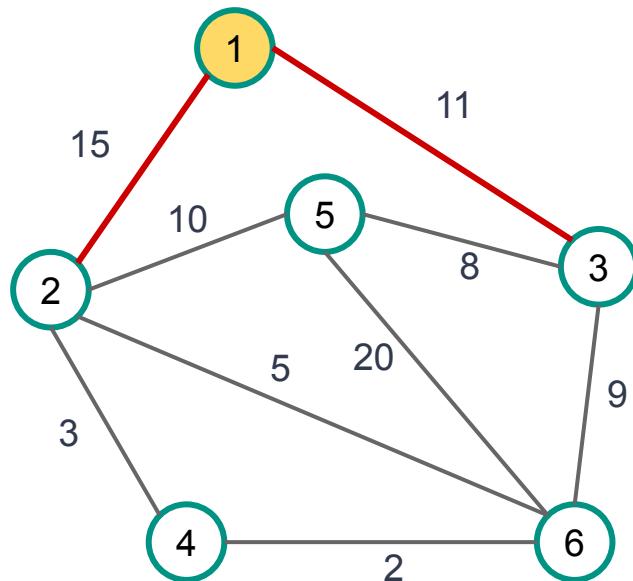


# Exemplu

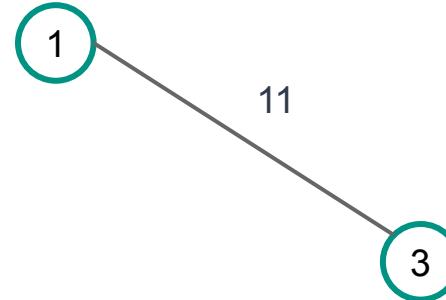
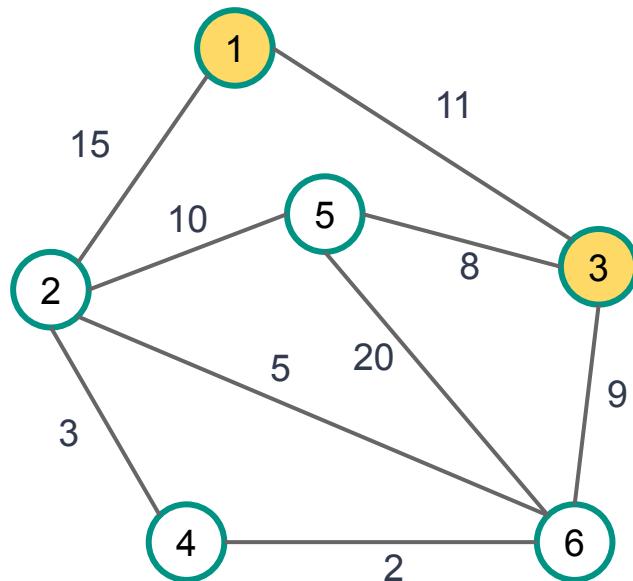


$s =$  1

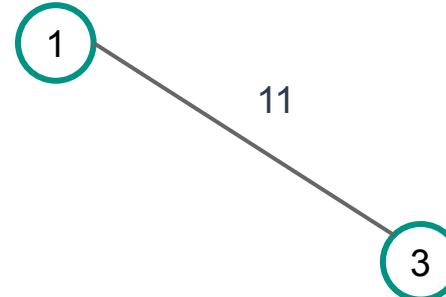
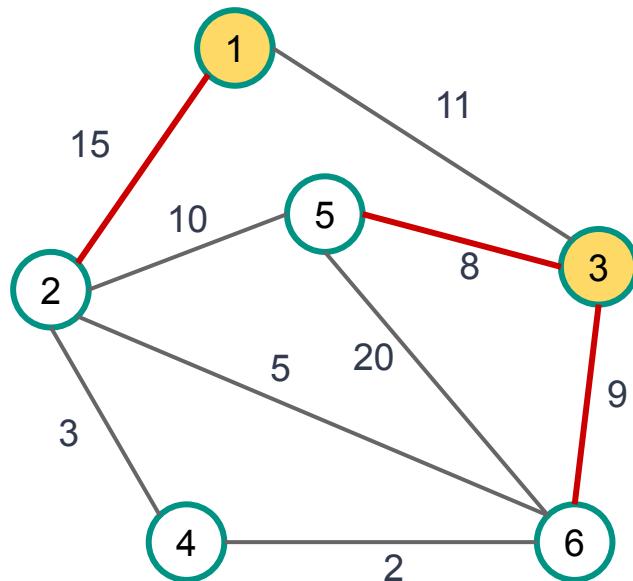
# Exemplu



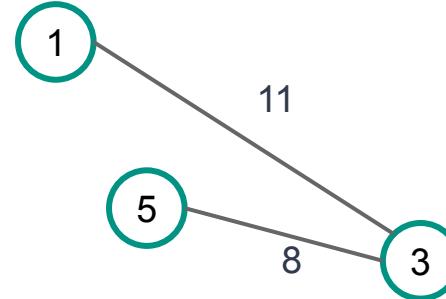
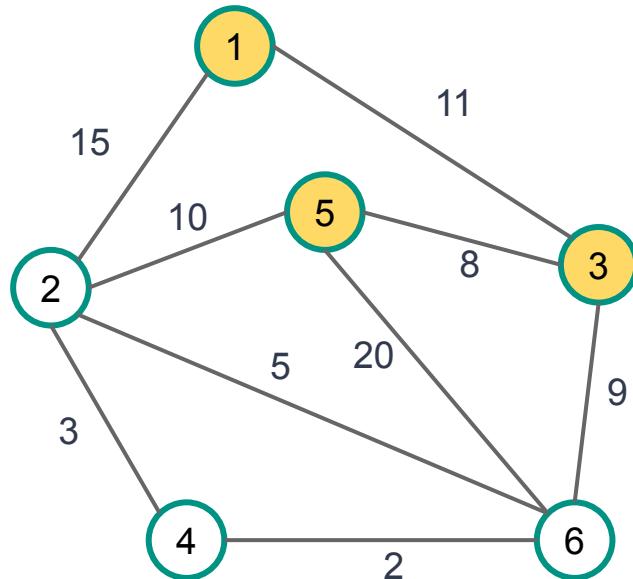
# Exemplu



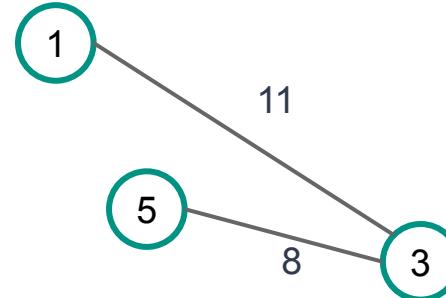
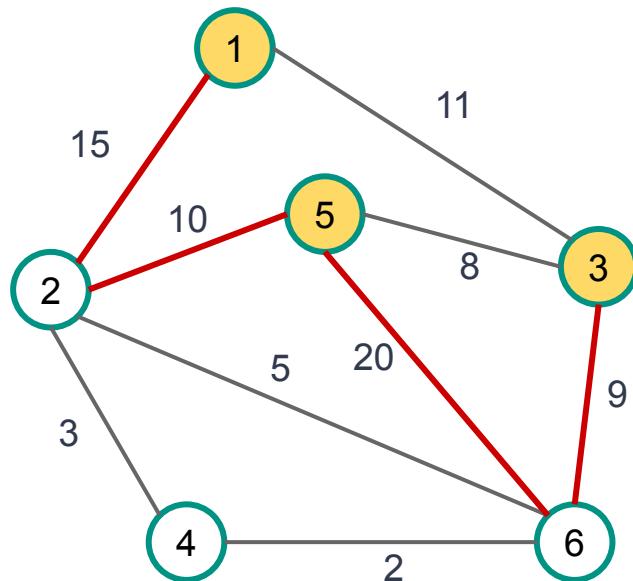
# Exemplu



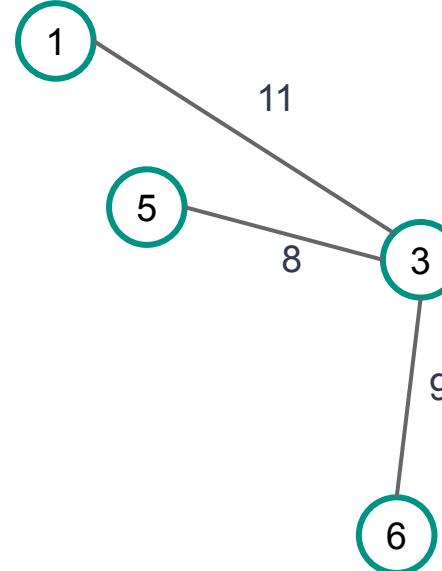
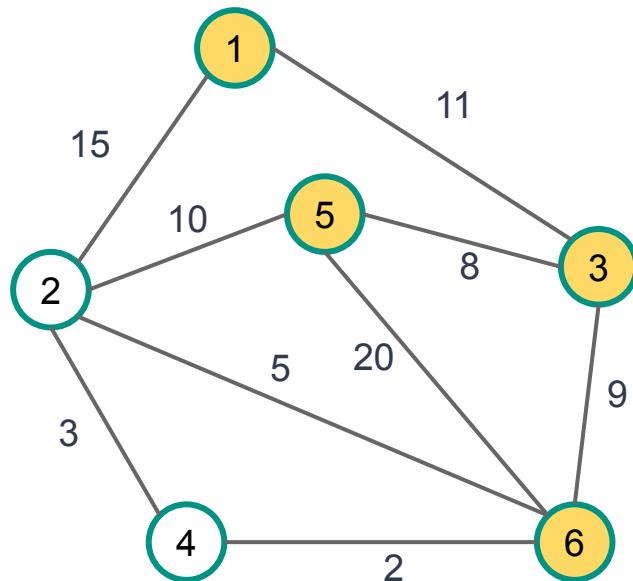
# Exemplu



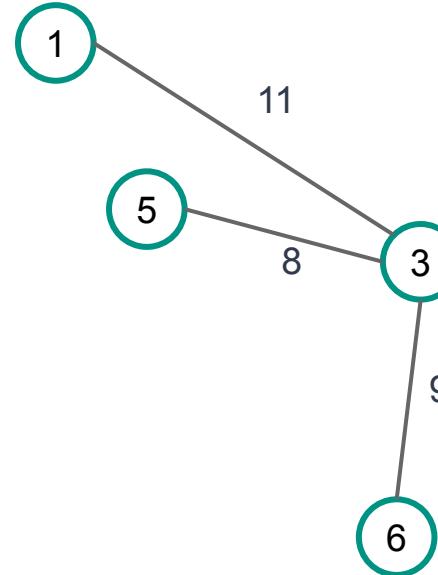
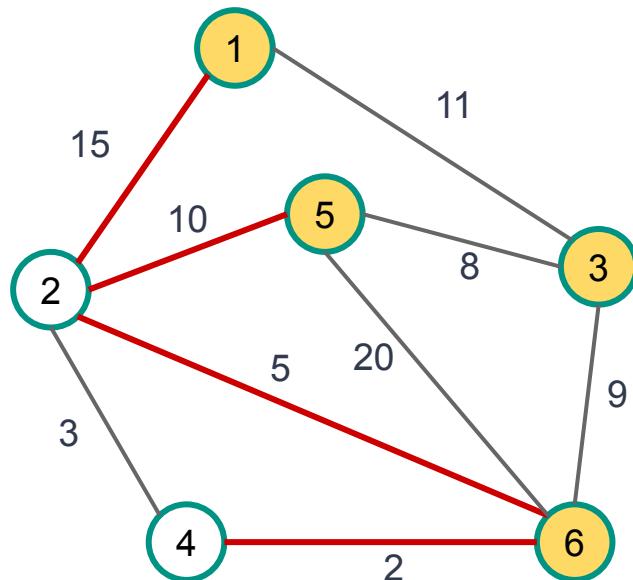
# Exemplu



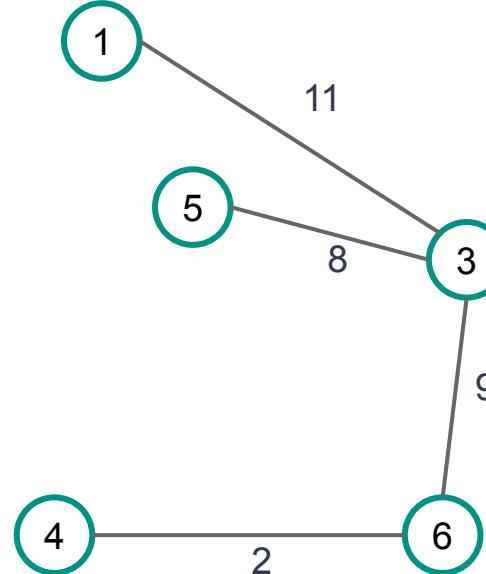
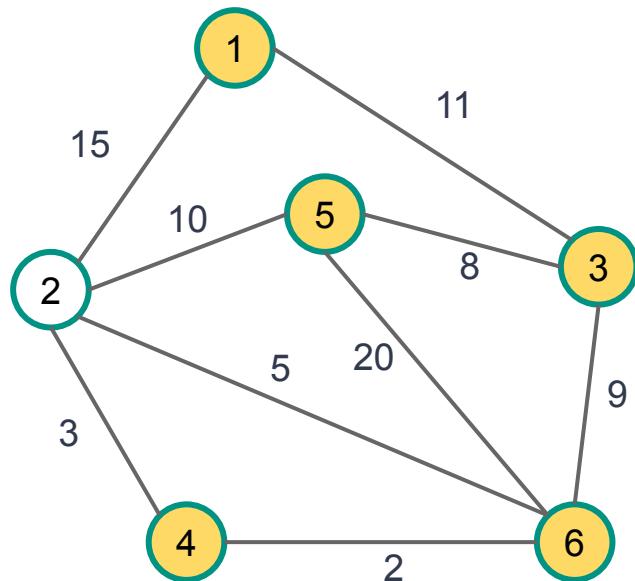
# Exemplu



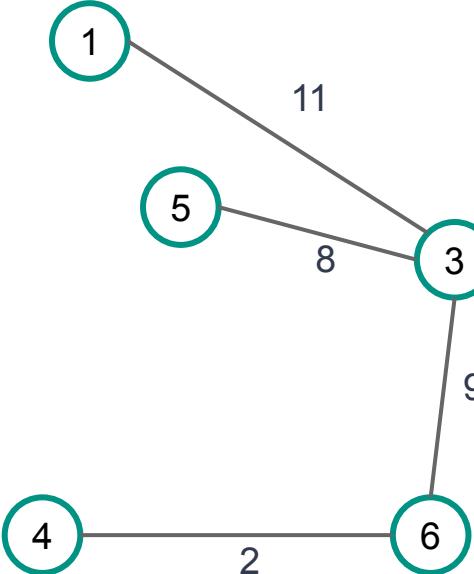
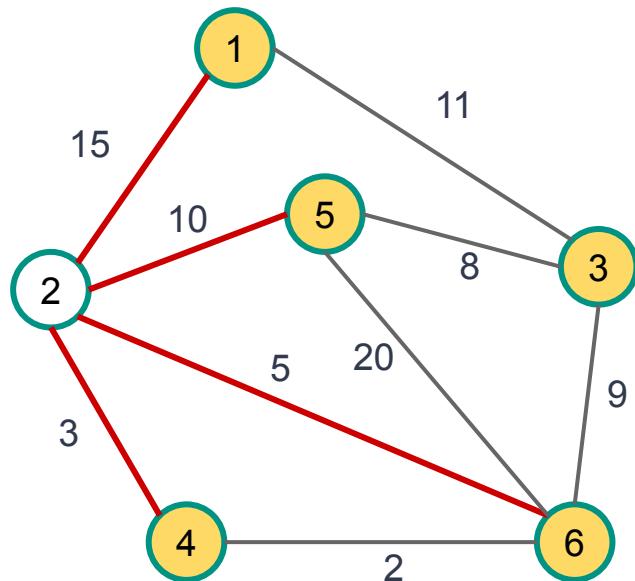
# Exemplu



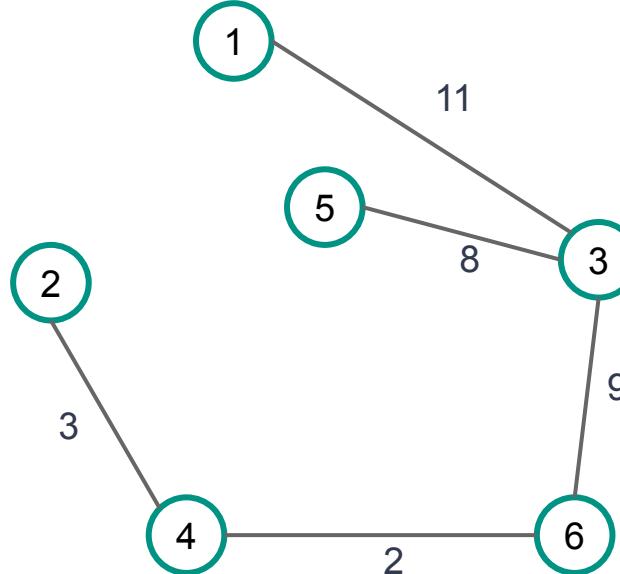
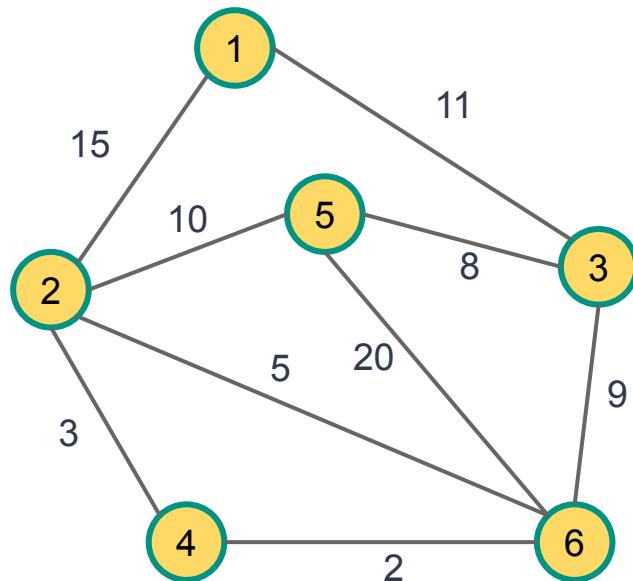
# Exemplu



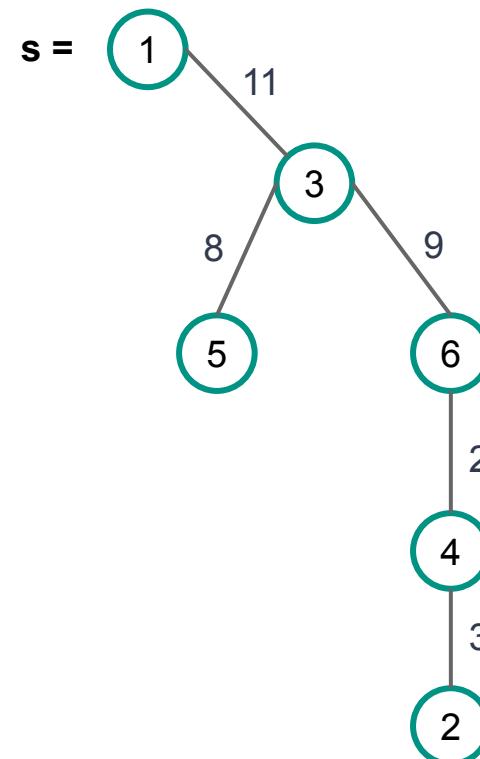
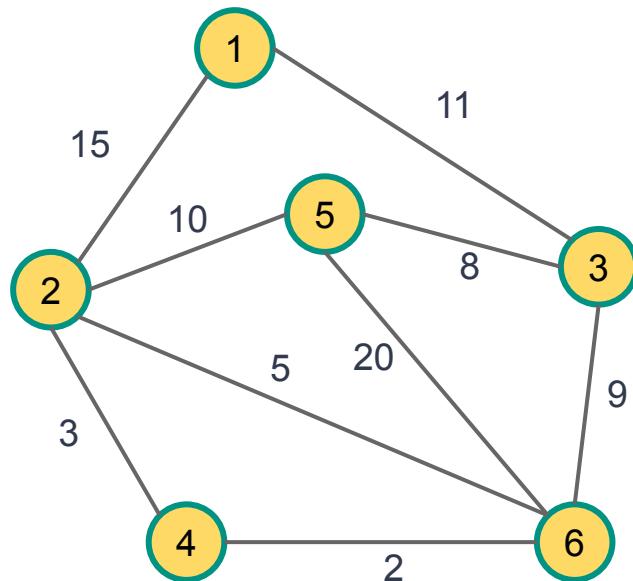
# Exemplu



# Exemplu



# Exemplu



# Prim - Implementare



- Cum alegem eficient o muchie de cost minim cu o extremitate selectată (deja în arbore) și cealaltă nu?

# Prim - Implementare



La fiecare pas, parcurgem toate muchiile și o alegem pe cea de cost minim cu o extremitate selectată și una neselectată.

# Prim - Implementare



La fiecare pas, parcurgem toate muchiile și o alegem pe cea de cost minim cu o extremitate selectată și una neselectată.

⇒ **O( $mn$ ) - ineficient**



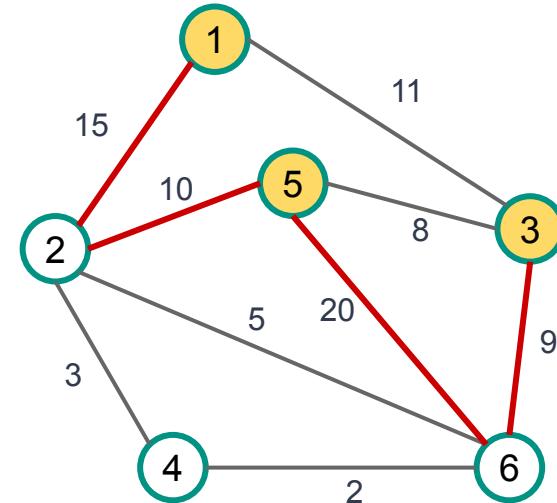
# Prim - Implementare



- Cum evităm să comparăm de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu?

**Exemplu:**

După ce vârfurile 1 și 5 au fost adăugate în arbore, muchiile **(2, 1)** și **(2, 5)** sunt comparate la fiecare pas, deși  $w(2, 1) > w(2, 5)$ , deci **(2, 1)** nu va fi selectată niciodată.



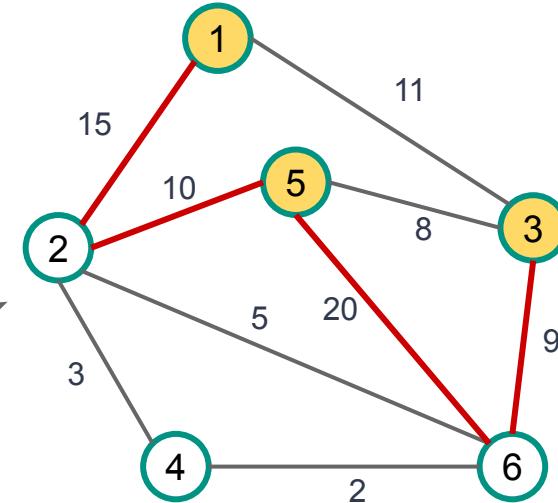
# Prim - Implementare



- Cum evităm să comparăm de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu?

Pentru fiecare vârf (neselectat), memorăm **doar muchia de cost minim** care îl unește cu un vârf din arbore (selectat).

pentru vârful 2, va fi memorată, la acest pas,  
muchia (2, 5)



# Prim - Complexitate

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

- memorăm, la fiecare pas, pentru fiecare vârf, muchia de cost minim care îl unește de un vârf care este deja în arbore

sau

- heap de muchii  
  
(v. și laborator + seminar + alg. Dijkstra)

# Detalii de implementare

## Algoritmul lui Prim

# Prim - Implementare

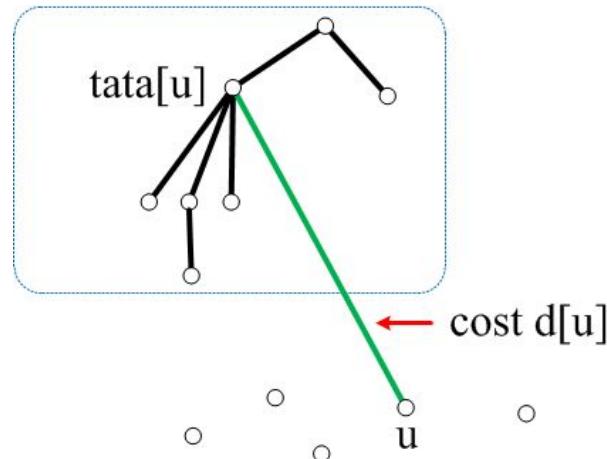
Asociem fiecărui vârf **u** următoarele informații (etichete) - pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore:**



# Prim - Implementare

Asociem fiecărui vârf  $u$  următoarele informații (etichete) - pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore**:

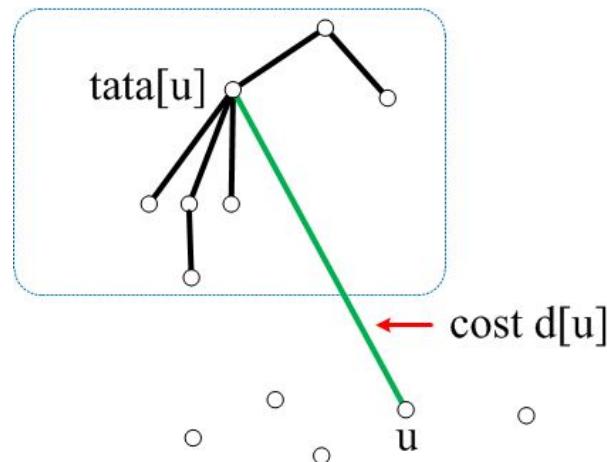
- $d[u]$  = costul minim al unei muchii de la  $u$  la un vârf deja selectat în arbore
- $tata[u]$  = acel vârf din arbore pentru care se realizează minimul



# Prim - Implementare

Avem:

- $(u, \text{tata}[u])$  este muchia de cost minim de la  $u$  la un vârf din arbore
- $d[u] = w(u, \text{tata}[u])$



# Prim - Implementare

Algoritmul se modifică astfel:

La un pas:

- se alege **un vârf**  $u$  cu **eticheta  $d$  minimă**, care nu este încă în arbore și se adaugă la arbore muchia **(tata[ $u$ ],  $u$ )**
  - **aceasta este muchia de cost minim care unește un vârf neselectat de un vârf din arbore**

# Prim - Implementare

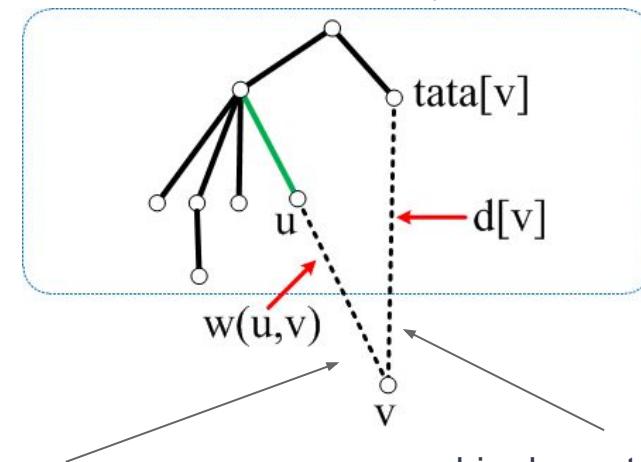
Algoritmul se modifică astfel:

La un pas:

- se alege **un vârf**  $u$  cu **eticheta  $d$  minimă**, care nu este încă în arbore și se adaugă la arbore muchia **(tata[u], u)**

- se actualizează etichetele vârfurilor  $v \notin V(T)$  vecine cu  $u$  astfel:

dacă  $w(u, v) < d[v]$  atunci  
 $d[v] = w(u, v)$   
 $tata[v] = u$



noua muchie:  $vu$

muchia de cost minim determinată până acum:  
 $(v, tata[v])$

# Prim - Implementare

Muchiile arborelui vor fi, în final:

$$(u, \text{tata}[u]), \quad u \neq s$$

# Prim

Notăm  $Q = V(G) - V(T) = \text{mulțimea vârfurilor neselectate încă în arbore}$

# Prim - Algoritm

- $s$  - vârful de start
- initializează  $Q$  cu  $V$
- pentru fiecare  $u \in V$  execută
  - $d[u] = \infty$
  - $tata[u] = 0$
- $d[s] = 0$

# Prim - Algoritm

- **s** - vârful de start
- **initializează Q cu V**
- **pentru fiecare  $u \in V$  execută**
  - $d[u] = \infty$**
  - $tata[u] = 0$**
- **$d[s] = 0$**
- **cât timp  $Q \neq \emptyset$  execută // pentru  $i=1, n$  (suficient  $n-1$ )**

# Prim - Algoritm

- **s** - vârful de start
- **initializează** Q cu V
- **pentru** fiecare  $u \in V$  **execută**
  - $d[u] = \infty$
  - $tata[u] = 0$
- **$d[s] = 0$**
- **cât timp**  $Q \neq \emptyset$  **execută** // pentru  $i=1, n$  (suficient  $n-1$ )
  - extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă**

# Prim - Algoritm

- **s** - vârful de start
- **initializează** Q cu V
- **pentru** fiecare  $u \in V$  **execută**
  - $d[u] = \infty$
  - $tata[u] = 0$
- **$d[s] = 0$**
- **cât timp**  $Q \neq \emptyset$  **execută** // pentru  $i=1, n$  (suficient  $n-1$ )
  - extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă
  - pentru** fiecare  $uv \in E$  **execută**

# Prim - Algoritm

- $s$  - vârful de start
- initializează  $Q$  cu  $V$
- pentru fiecare  $u \in V$  execută
  - $d[u] = \infty$
  - $tata[u] = 0$
- $d[s] = 0$
- cât timp  $Q \neq \emptyset$  execută // pentru  $i=1, n$  (suficient  $n-1$ )
  - extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă
  - pentru fiecare  $uv \in E$  execută
    - dacă  $v \in Q$  și  $w(u, v) < d[v]$  atunci
      - $d[v] = w(u, v)$
      - $tata[v] = u$

# Prim - Algoritm

- $s$  - vârful de start
- initializează  $Q$  cu  $V$
- pentru fiecare  $u \in V$  execută
  - $d[u] = \infty$
  - $tata[u] = 0$
- $d[s] = 0$
- cât timp  $Q \neq \emptyset$  execută // pentru  $i=1, n$  (suficient  $n-1$ )
  - extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă
  - pentru fiecare  $uv \in E$  execută
    - dacă  $v \in Q$  și  $w(u, v) < d[v]$  atunci
      - $d[v] = w(u, v)$
      - $tata[v] = u$
- scrie  $(u, tata[u])$  pentru  $u \neq s$

# Prim - Complexitate

- Inițializări** →
- $n *$  extragere vârf minim** →
- actualizare etichete vecini** →

# Prim - Complexitate



- Cum putem memora  $Q$  pentru a determina eficient vârful  $u \in Q$  cu eticheta minimă?

# Prim - Complexitate



- Cum putem memora  $Q$  pentru a determina eficient vârful  $u \in Q$  cu eticheta minimă?
  - vector?
  - heap?

# Prim - Complexitate

**Varianta 1 - Folosim vector de vizitat**

$$Q[u] = 1, \text{ dacă } u \in Q$$

0, altfel

# Prim - Complexitate

**Varianta 1** - cu vector de vizitat

**Inițializări** →

**n \* extragere vârf minim** →

**actualizare etichete vecini** →

---

# Prim - Complexitate

**Varianta 1** - cu vector de vizitat

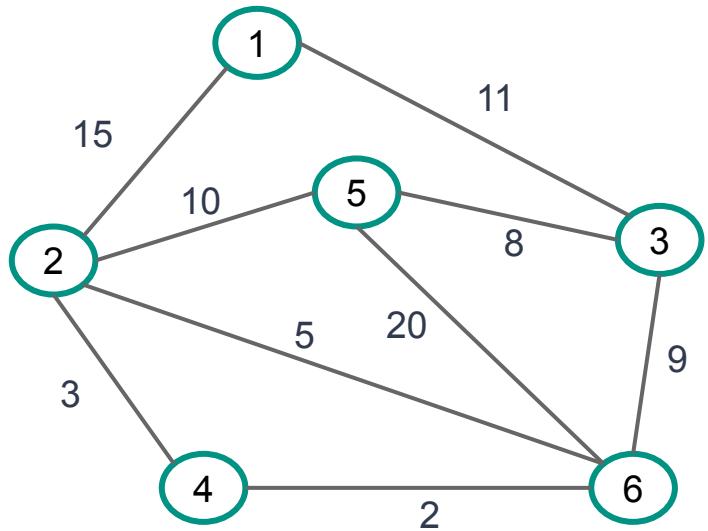
**Inițializări** →  $O(n)$

**$n * \text{extragere vârf minim}$**  →  $O(n^2)$

**actualizare etichete vecini** →  $O(m)$

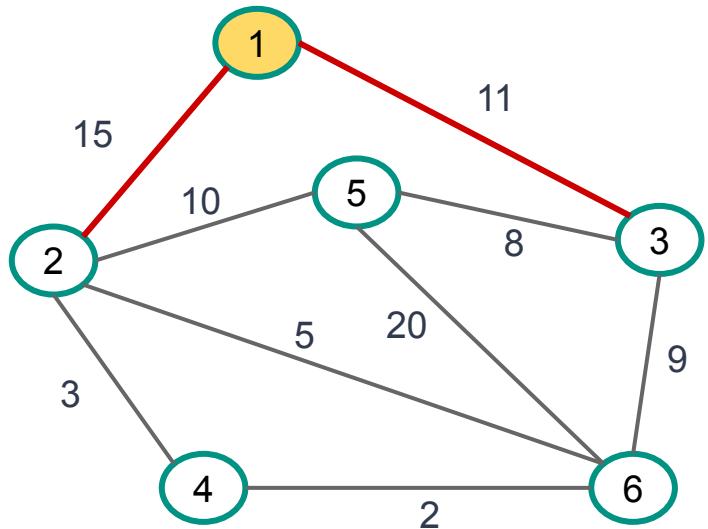
---

$O(n^2)$



---

$d/tata = [ 0/0, \infty/0, \infty/0, 2, 1, 0/0, \infty/0, \infty/0, 6 ]$



d/tata

1  
[ 0/0,

2  
 $\infty/0$ ,

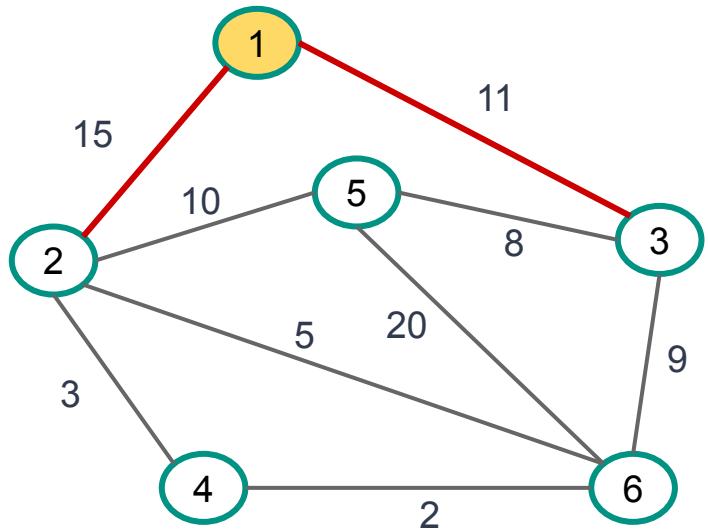
3  
 $\infty/0$ ,

4  
 $\infty/0$ ,

5  
 $\infty/0$ ,

6  
 $\infty/0$  ]

Selectăm 1



d/tata

[ 0/0,

$\infty/0$ ,

$\infty/0$ ,

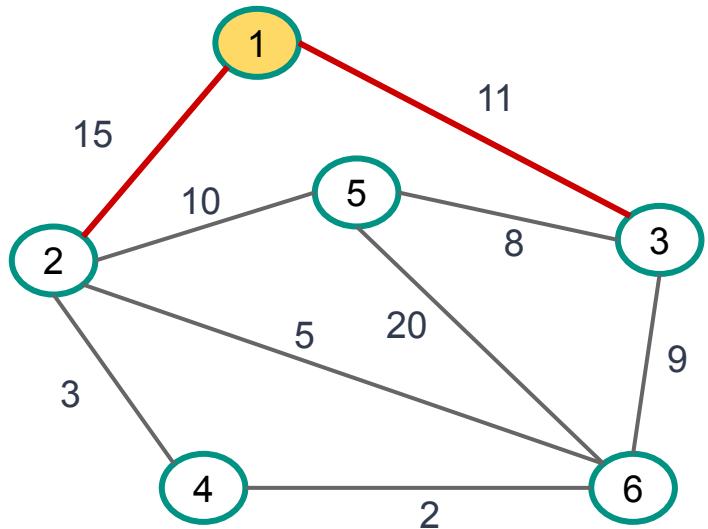
$\infty/0$ ,

$\infty/0$ ,

$\infty/0$  ]

Selectăm 1

viz[1] = 1 (vom reprezenta prin - )



d/tata

**[ 0/0,**

$\infty/0,$

$\infty/0,$

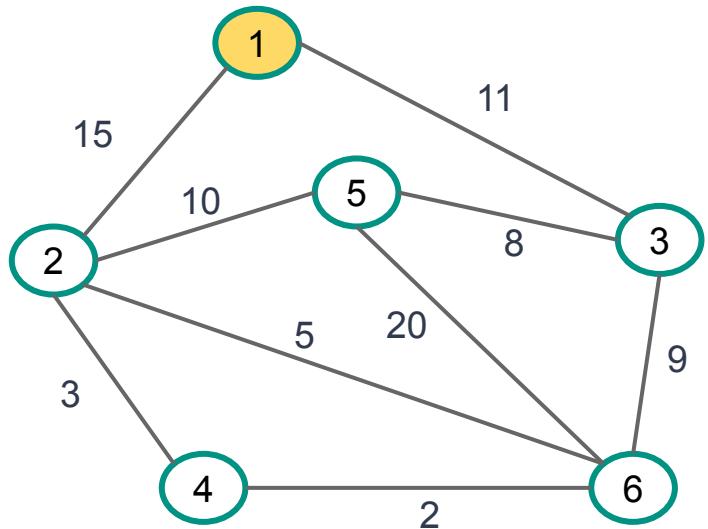
$\infty/0,$

$\infty/0,$

$\infty/0 ]$

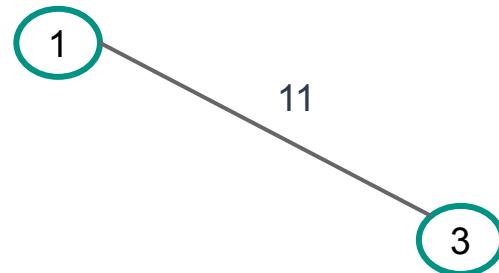
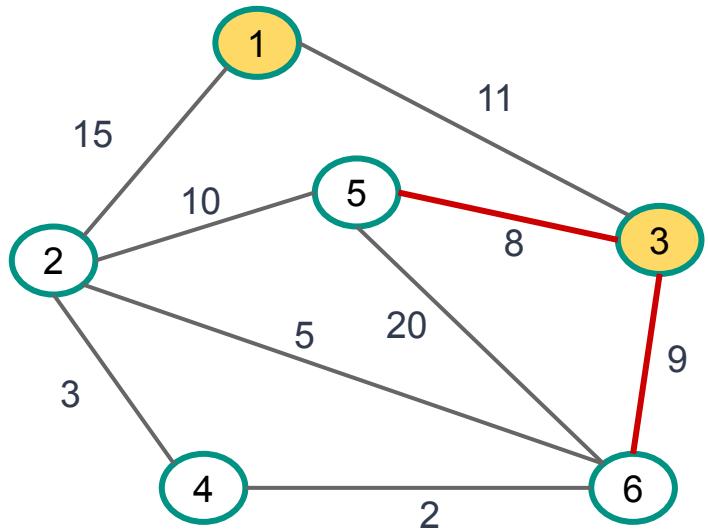
Selectăm 1

actualizăm etichetele vecinilor



d/tata	1	2	3	4	5	6
Selectăm 1	[ <b>0/0</b> , - ]	<b>15/1</b>	<b>11/1</b>	<b><math>\infty/0</math></b>	<b><math>\infty/0</math></b>	<b><math>\infty/0</math></b>

Pentru vârful **2**, este memorată muchia **(2,1)** de cost **15**, mai exact costul muchiei în **d[2]** și cealaltă extremitate în **tata[2]**



d/tata

[ 0/0,

$\infty/0$ ,

$\infty/0$ ,

$\infty/0$ ,

$\infty/0$ ,

$\infty/0$  ]

Selectăm 1

[ - ,

15/1,

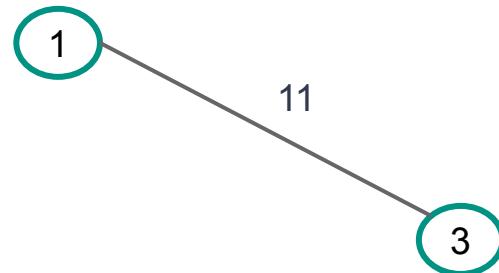
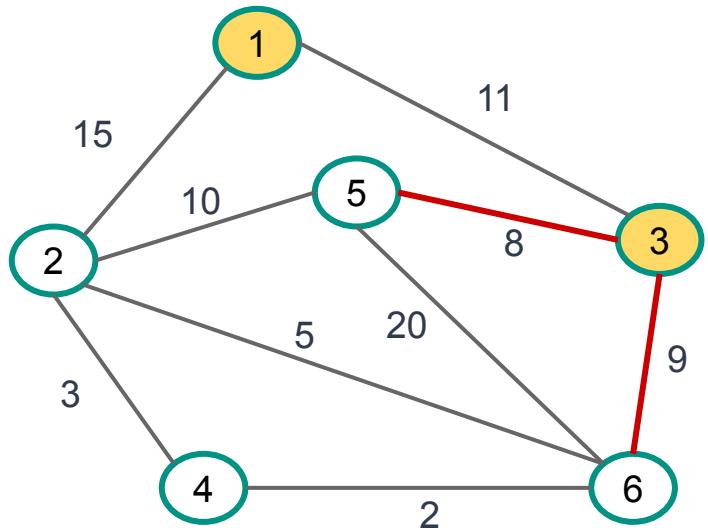
11/1,

$\infty/0$ ,

$\infty/0$ ,

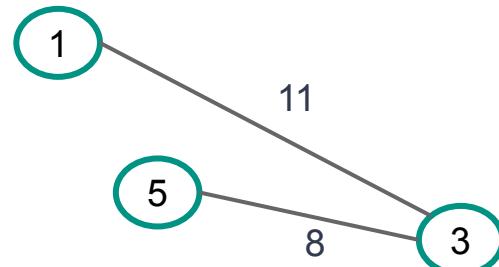
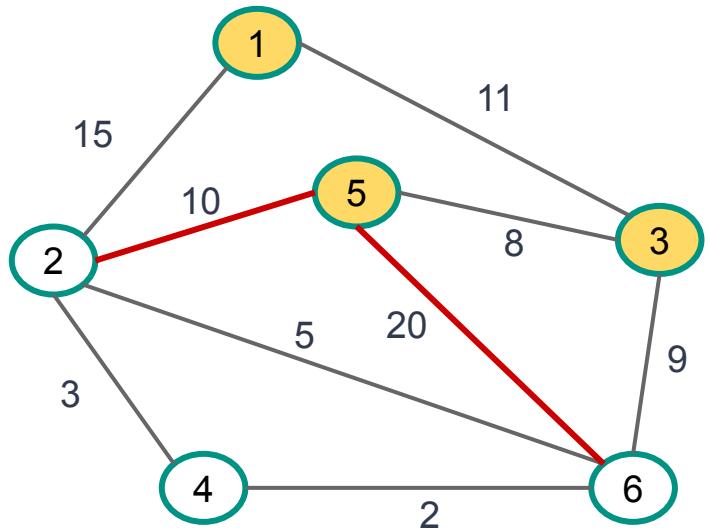
$\infty/0$  ]

Selectăm 3



d/tata	1	2	3	4	5	6
	$0/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$

Selectám 1	[ - ,	$15/1$ ,	$11/1$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
Selectám 3	[ - ,	$15/1$ ,	- ,	$\infty/0$ ,	$8/3$ ,	$9/3$ ]



d/tata	1	2	3	4	5	6
	$0/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$

Selectám 1	$[ - , \quad 15/1, \quad \textcolor{red}{11/1}, \quad \infty/0, \quad \infty/0, \quad \infty/0 ]$
------------	---

Selectám 3	$[ - , \quad 15/1, \quad - , \quad \infty/0, \quad \textcolor{red}{8/3}, \quad 9/3 ]$
------------	---

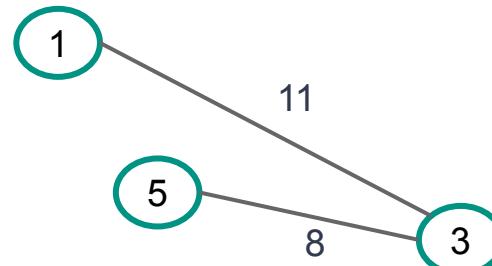
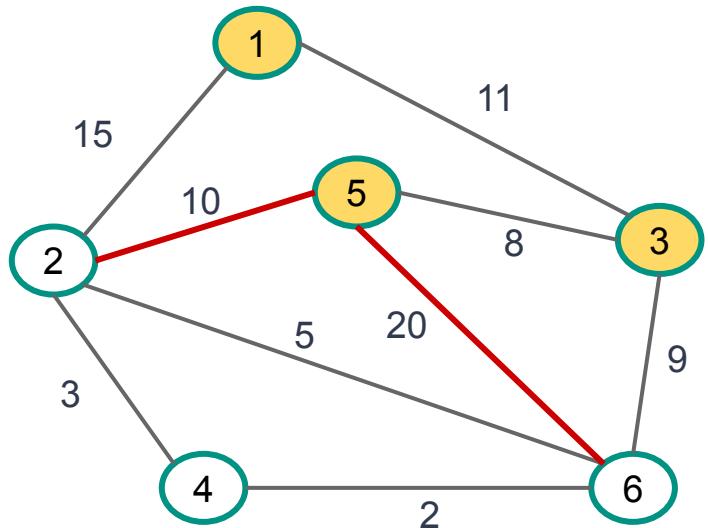
Selectám 5



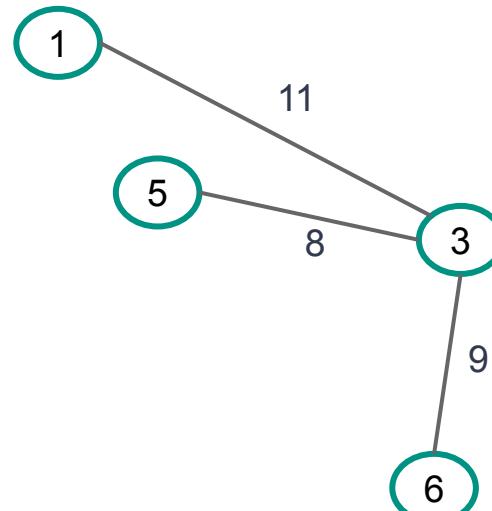
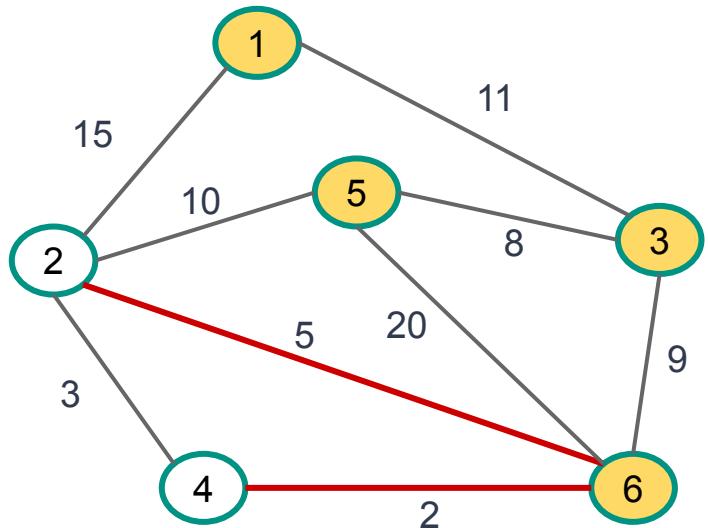
$$d[2] = \min \{ d[2], w(2, 5) \}$$



$$d[6] = \min \{ d[6], w(6, 5) \}$$



d/tata	1	2	3	4	5	6
Selectám 1	[ $0/0$ , - , $15/1$ , $11/1$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]					
Selectám 3	[ - , $15/1$ , - , $\infty/0$ , $8/3$ , $9/3$ ]					
Selectám 5	[ - , $10/5$ , - , $\infty/0$ , - , $9/3$ ]					



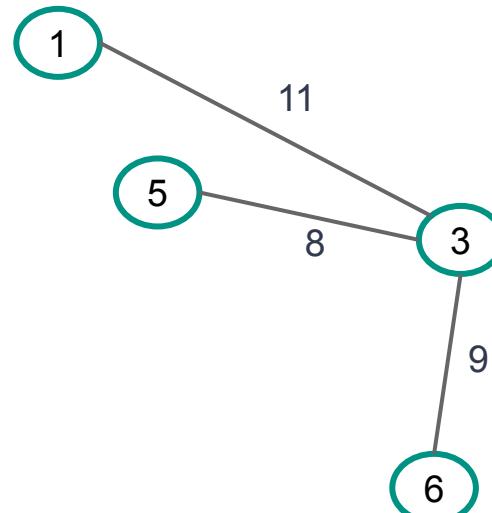
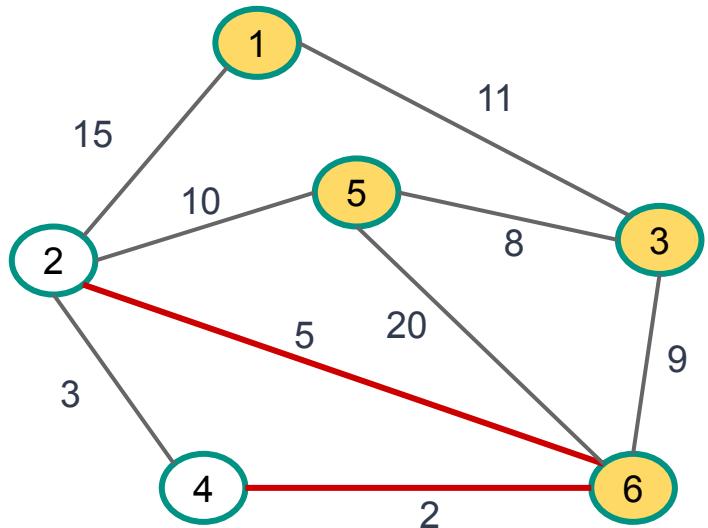
d/tata	1	2	3	4	5	6
	$0/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$	$\infty/0$

Selectám 1 [ - ,  $15/1$ ,  $11/1$ ,  $\infty/0$ ,  $\infty/0$ ,  $\infty/0$  ]

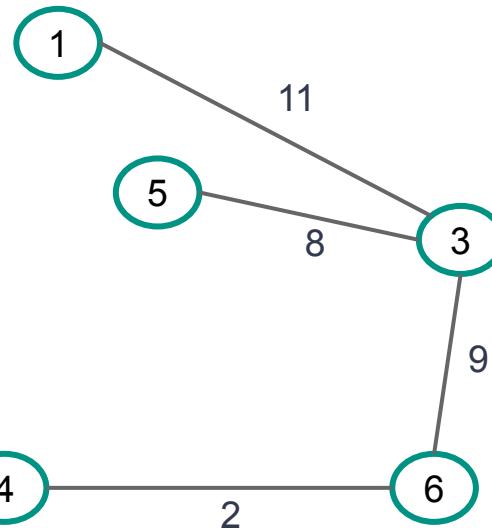
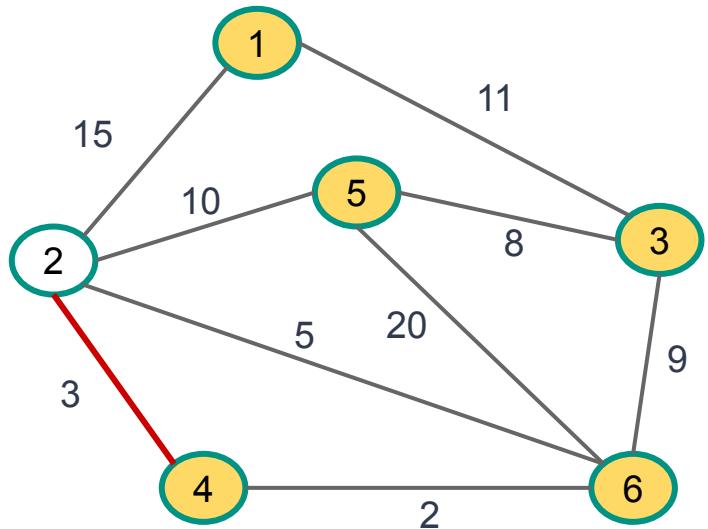
Selectám 3 [ - ,  $15/1$ , - ,  $\infty/0$ ,  $8/3$ ,  $9/3$  ]

Selectám 5 [ - ,  $10/5$ , - ,  $\infty/0$ , - ,  $9/3$  ]

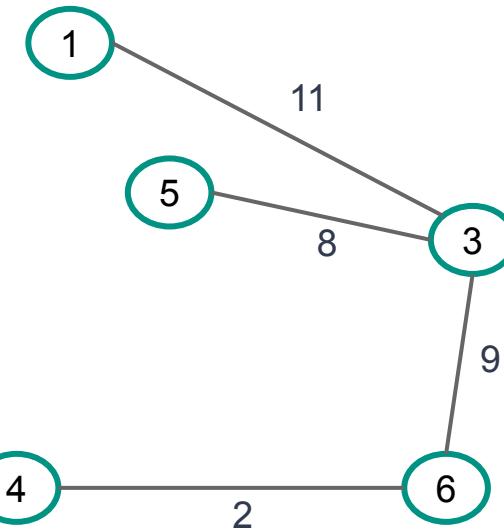
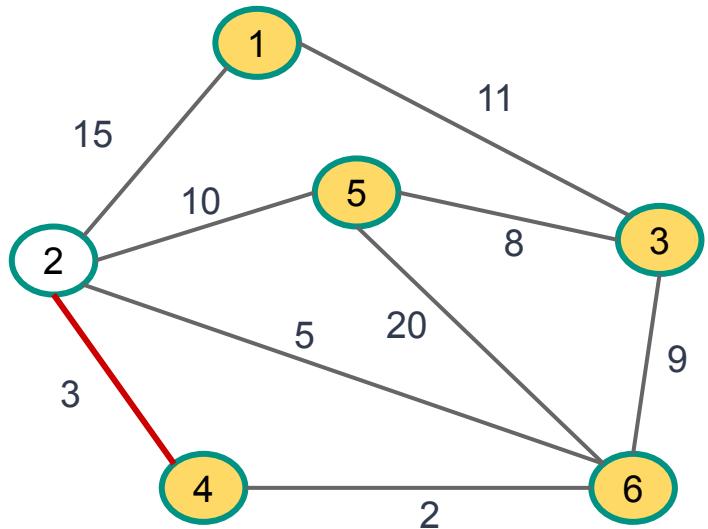
Selectám 6



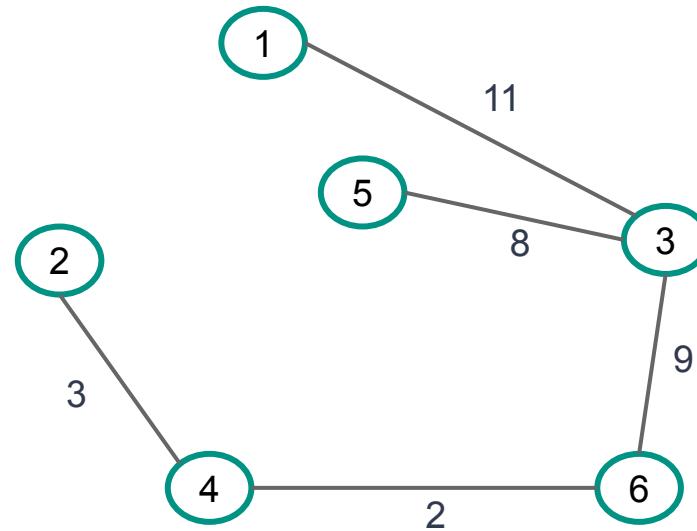
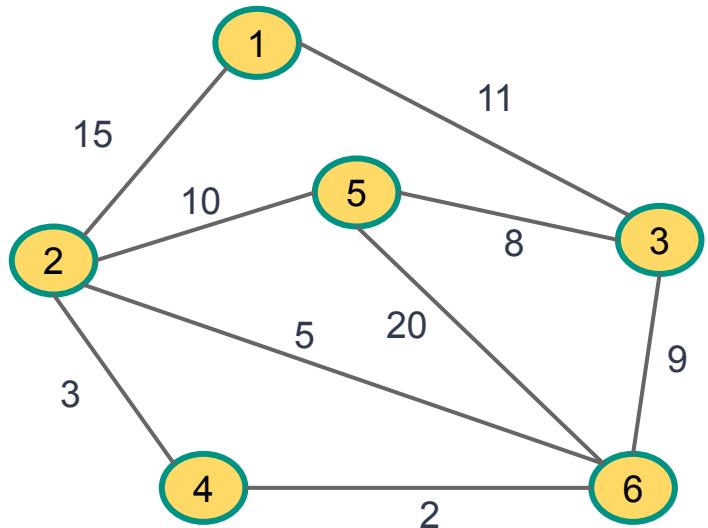
d/tata	1	2	3	4	5	6
Selectám 1	[ <b>0/0</b> ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
Selectám 3	[ - ,	$15/1$ ,	<b><math>11/1</math></b> ,	$\infty/0$ ,	$\infty/0$ ,	$9/3$ ]
Selectám 5	[ - ,	$10/5$ ,	- ,	$\infty/0$ ,	- ,	<b><math>9/3</math></b> ]
Selectám 6	[ - ,	<b><math>5/6</math></b> ,	- ,	$2/6$ ,	- ,	- ]



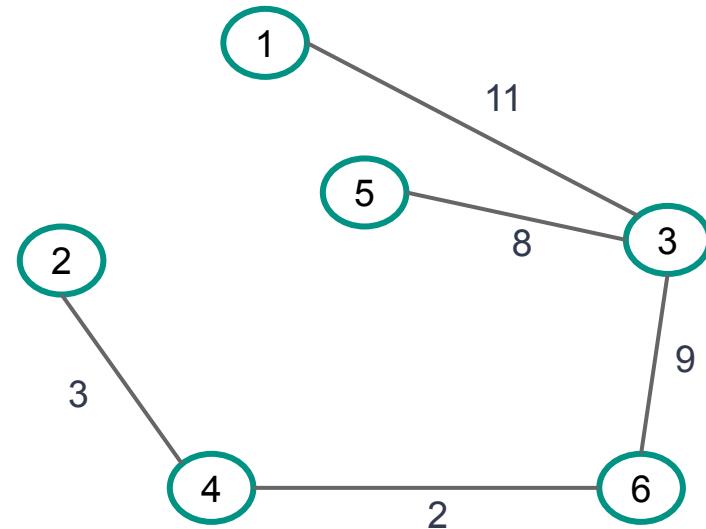
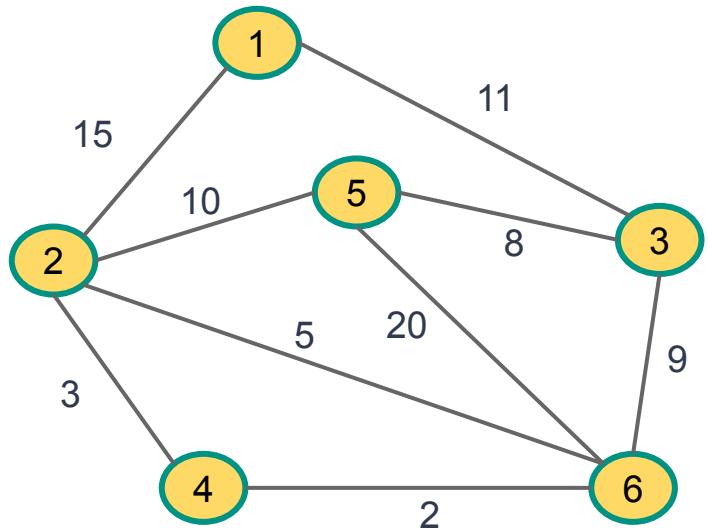
d/tata	1	2	3	4	5	6
Selectám 1	[ <b>0/0</b> ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
Selectám 3	[ - ,	$15/1$ ,	<b><math>11/1</math></b> ,	$\infty/0$ ,	$\infty/0$ ,	$9/3$ ]
Selectám 5	[ - ,	$10/5$ ,	- ,	$\infty/0$ ,	- ,	<b><math>9/3</math></b> ]
Selectám 6	[ - ,	$5/6$ ,	- ,	<b><math>2/6</math></b> ,	- ,	- ]
Selectám 4						



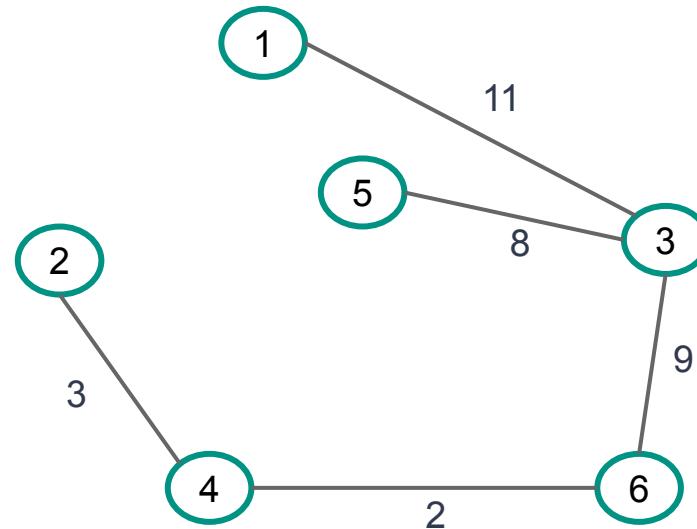
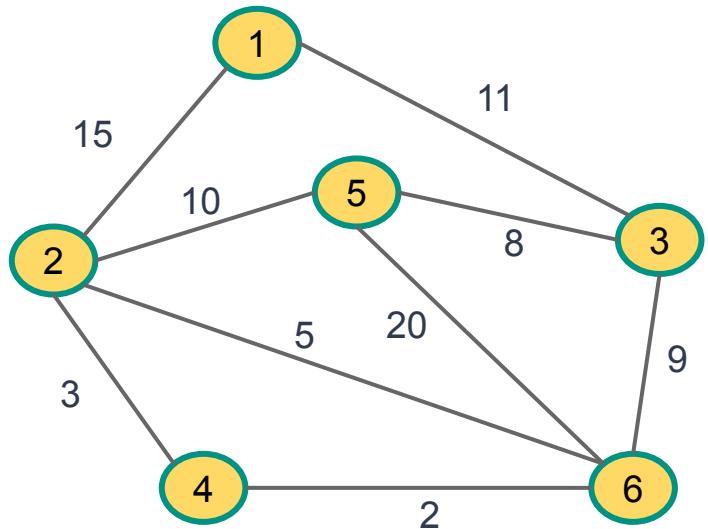
d/tata	1	2	3	4	5	6
Selectám 1	[ - , <b>15/1</b> , <b>11/1</b> , $\infty/0$ , $\infty/0$ , $\infty/0$ ]					
Selectám 3	[ - , <b>15/1</b> , - , $\infty/0$ , <b>8/3</b> , <b>9/3</b> ]					
Selectám 5	[ - , <b>10/5</b> , - , $\infty/0$ , - , <b>9/3</b> ]					
Selectám 6	[ - , <b>5/6</b> , - , <b>2/6</b> , - , - ]					
Selectám 4	[ - , <b>3/4</b> , - , - , - , - ]					



d/tata	1	2	3	4	5	6
Selectám 1	[ 0/0,	15/1,	11/1,	∞/0,	∞/0,	∞/0 ]
Selectám 3	[ - ,	15/1,	- ,	∞/0,	8/3,	9/3 ]
Selectám 5	[ - ,	10/5,	- ,	∞/0,	- ,	9/3 ]
Selectám 6	[ - ,	5/6,	- ,	2/6,	- ,	- ]
Selectám 4	[ - ,	3/4,	- ,	- ,	- ,	- ]
Selectám 2						



d/tata	1	2	3	4	5	6
Selectám 1	[ - , <b>15/1</b> , <b>11/1</b> , $\infty/0$ , $\infty/0$ , $\infty/0$ ]					
Selectám 3	[ - , <b>15/1</b> , - , $\infty/0$ , <b>8/3</b> , <b>9/3</b> ]					
Selectám 5	[ - , <b>10/5</b> , - , $\infty/0$ , - , <b>9/3</b> ]					
Selectám 6	[ - , <b>5/6</b> , - , <b>2/6</b> , - , - ]					
Selectám 4	[ - , <b>3/4</b> , - , - , - ]					
Selectám 2	[ - , - , - , - , - ]					



d/tata  
FINAL

[ 0/0,

3/4,

11/1,

2/6,

8/3,

9/3 ]

**Vectorul tata  $\Rightarrow$  muchiile arborelui ( $u$ , tata[u]),  $u \neq s$**

# Prim - Complexitate

**Varianta 2** - cu memorarea vârfurilor într-un min-heap Q (min-ansamblu)

**Inițializare Q** →

**n \* extragere vârf minim** →

**actualizare etichete vecini** →

---

# Prim - Algoritm

Prim(G, w, s)

```
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
initializeaza Q cu V
cat timp  $Q \neq \emptyset$  executa
     $u =$  extrage varf cu eticheta minima din Q
    pentru fiecare  $v$  adiacent cu  $u$  executa
        daca  $v \in Q$  si  $w(u, v) < d[v]$  atunci
             $d[v] = w(u, v)$ 
             $tata[v] = u$ 
```

???

scrive  $(u, tata[u])$ , pentru  $u \neq s$

# Prim - Algoritm

Prim(G, w, s)

```
pentru fiecare u ∈ V executa
    d[u] = ∞; tata[u] = 0
d[s] = 0
initializeaza Q cu V
cat timp Q ≠ ∅ executa
    u = extrage varf cu eticheta minima din Q
    pentru fiecare v adiacent cu u executa
        daca v ∈ Q si w(u, v) < d[v] atunci
            d[v] = w(u, v)
            tata[v] = u
        // actualizeaza Q - pentru Q heap
```

scrie (u, tata[u]), pentru u ≠ s

# Prim - Complexitate

**Varianta 2** - cu memorarea vârfurilor într-un min-heap Q (min-ansamblu)

- Inițializare Q** →  $O(n)$
- n \* extragere vârf minim** →  $O(n \log n)$
- actualizare etichete vecini** →  $O(m \log n)$

---

$O(m \log n)$

# Prim - Complexitate

**Observație** - Dacă graful este complet (spre exemplu, dacă toate punctele se pot conecta și distanța dintre puncte este distanța euclidiană), atunci  $m = n(n-1) / 2$  este de ordin  $n^2$

⇒  $O(n^2)$  mai eficient

# Algoritmi bazați pe eliminare de muchii



**Temă:** Care dintre următorii algoritmi determină corect un arbore parțial de cost minim? Justificați. Pentru fiecare algoritm corect, precizați ce complexitate are.

1.  $T \leftarrow G$   
**cât timp  $T$  conține cicluri execută**  
alege  $e$  o muchie de cost minim care este conținută într-un ciclu din  $T$   
 $T \leftarrow T - e$
  
2.  $T \leftarrow G$   
**cât timp  $T$  conține cicluri execută**  
alege  $C$  un ciclu oarecare din  $T$  și fie  $e$  muchia de cost maxim din  $C$   
 $T \leftarrow T - e$

# Corectitudine Algoritmii Kruskal + Prim

# Corectitudine Kruskal + Prim



- Cei doi algoritmi determină corect un apcm?  
**Chiar dacă muchiile au și costuri negative?**
- Costul arborelui obținut de algoritmul lui Prim nu depinde de vârful de start?

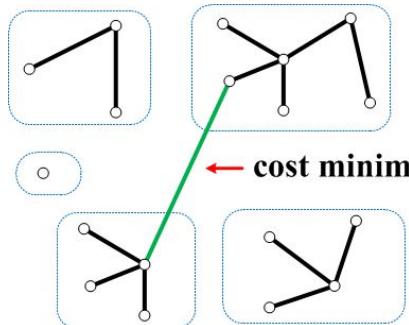
# Corectitudine Kruskal + Prim

- Fie  $A \subseteq E$  o mulțime de muchii
- Notăm  $A \subseteq apcm \Leftrightarrow \exists T$  un apcm astfel încât  $A \subseteq E(T)$

# Amintim

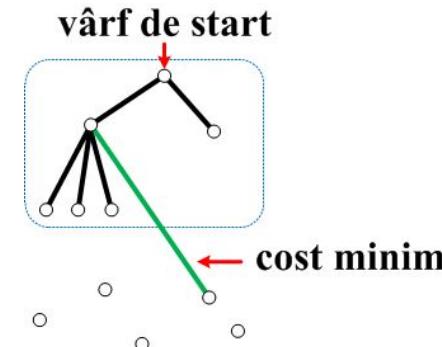
## KRUSKAL

- **Inițial:**  $T = (V, \emptyset)$
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** din **G** a.î.  $u, v$  sunt în **componente conexe diferite** ( $T + uv$  aciclic)
  - $E(T) = E(T) \cup \{uv\}$



## PRIM

- **s - vârful de start**
- **Inițial,  $T = (\{s\}, \emptyset)$**
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** din **G** a.î.  $u \in V(T)$  și  $v \notin V(T)$
  - $V(T) = V(T) \cup \{v\}$
  - $E(T) = E(T) \cup uv$



# Corectitudine Kruskal + Prim

Atât algoritmul lui Kruskal, cât și cel al lui Prim, funcționează după următoarea schemă:

- $A = \emptyset$  (multimea muchiilor selectate în arborele construit)
- pentru  $i = 1, n-1$  execută
  - alege o muchie  $e$  astfel încât  $A \cup \{e\} \subseteq \text{apcm}$
  - $A = A \cup \{e\}$
- returnează  $T = (V, A)$

# Corectitudine Kruskal + Prim

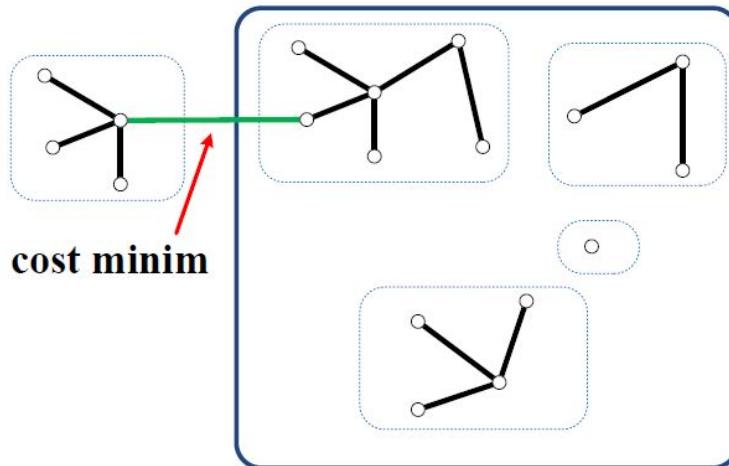
Vom demonstra cu **criteriu de alegere a muchiei e** la un pas astfel încât:

$$A \subseteq \text{apcm} \Rightarrow A \cup \{e\} \subseteq \text{apcm}$$

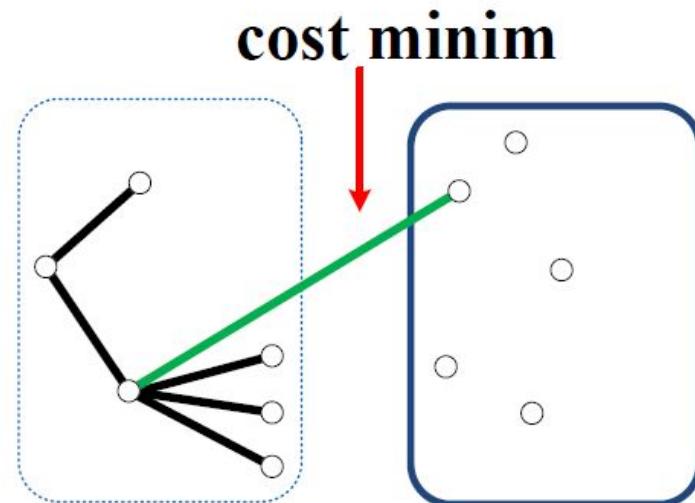
Vom demonstra că algoritmii lui Kruskal și Prim aplică acest criteriu.

# Corectitudine Kruskal + Prim

KRUSKAL

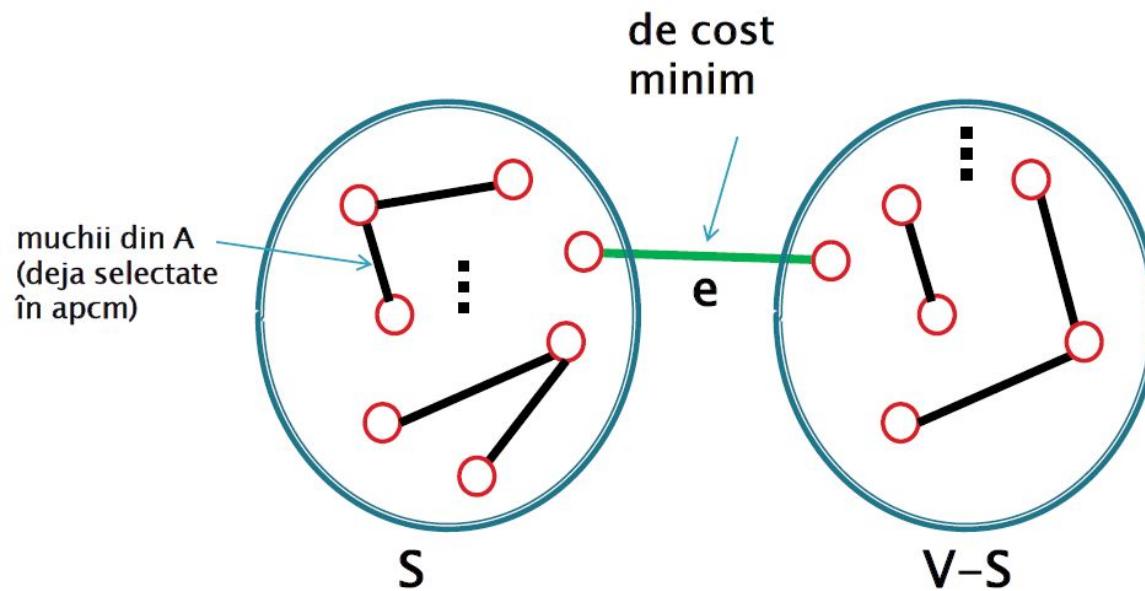


PRIM



**cost minim**

# Corectitudine Kruskal + Prim



$$A \subseteq \text{apcm} \Rightarrow A \cup \{e\} \subseteq \text{apcm}$$

# Corectitudine Kruskal + Prim

Fie  $G = (V, E, w)$  un **graf conex ponderat**.

**Propoziție.** Algoritmul lui Kruskal determină un apcm.

**Propoziție.** Algoritmul lui Prim determină un apcm.



# Structuri pentru mulțimi disjuncte



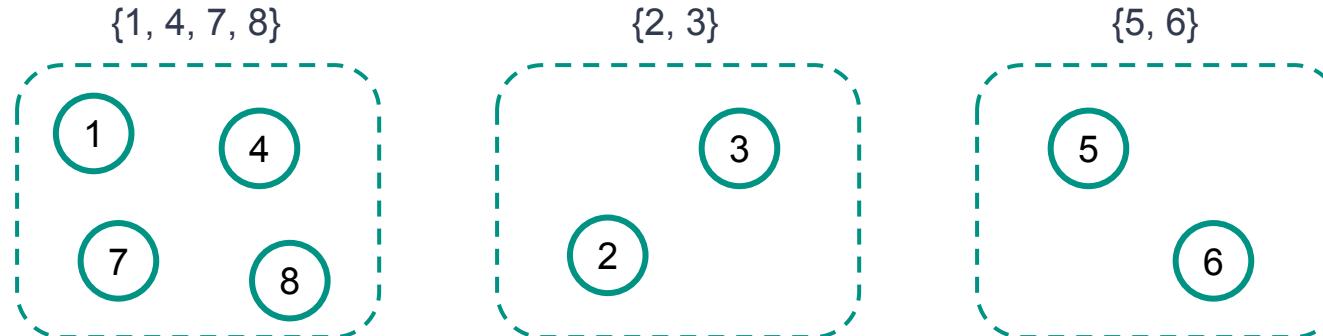
# Operații cu multimi disjuncte

## Problemă

Asupra unei partiții ale mulțimii  $\{1, 2, \dots, n\}$  (în submulțimi disjuncte) se efectuează o succesiune de operații de tip

- reuniune
- test de apartenență

Cum putem memora eficient submulțimile, astfel încât operațiile să se efectueze "eficient"?



# Operații cu multimi disjuncte

## Soluții

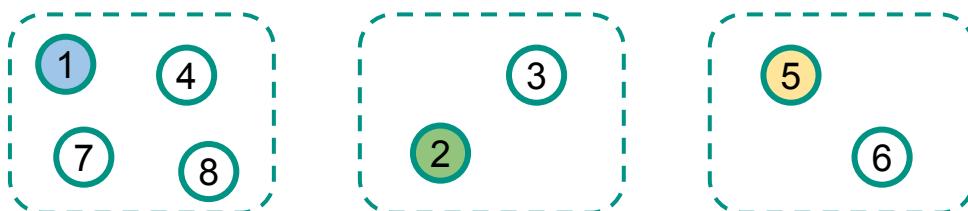
Asociem fiecărei submultimi un reprezentant (o culoare).

Notăm operațiile:

- Inițializare( $u$ )** - creează o mulțime cu un singur element  $u$
- Reprez( $u$ )** - returnează reprezentantul mulțimii care conține pe  $u$
- Reunește( $u, v$ )** - unește mulțimea care conține  $u$  și cea care conține  $v$

# Vector de reprezentanți

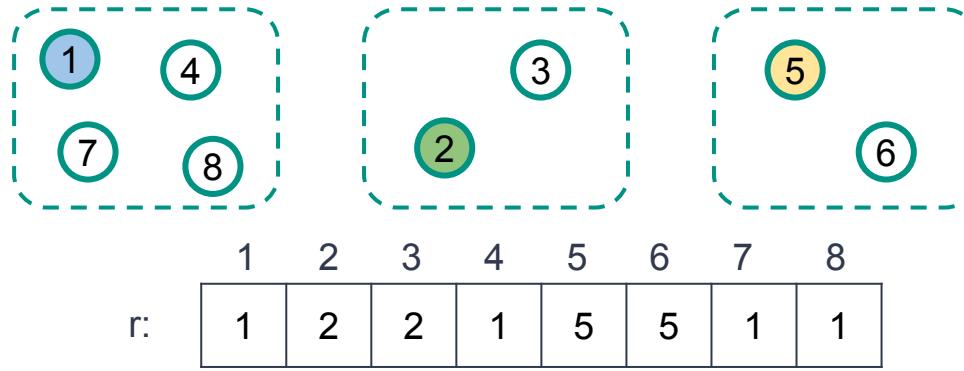
**Varianta 1** - memorăm într-un vector  $r$ , pentru fiecare element  $x$ , reprezentantul mulțimii  $r[x]$  (v. Kruskal curs)



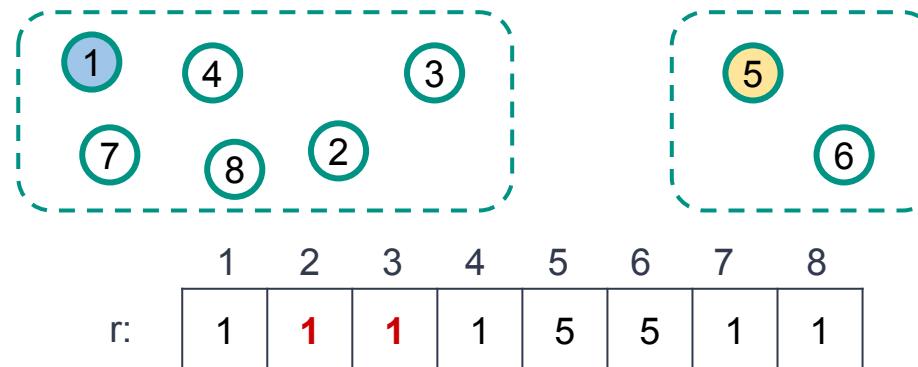
r:	1	2	3	4	5	6	7	8
	1	2	2	1	5	5	1	1

- Inițializare( $u$ )** -  $O(1)$       void **Initializare**(int  $u$ ) {  $r[u] = u;$  }
- Reprez( $u$ )** -  $O(1)$       int **Reprez**(int  $u$ ) { return  $r[u];$  }
- Reuneste( $u, v$ )** -  $O(n)$       void **Reuneste**(int  $u, v$ ) {  
         $r1 = \text{Reprez}(u);$     //  $r1 = r[u]$   
         $r2 = \text{Reprez}(v);$     //  $r2 = r[v]$   
        for ( $k=1; k \leq n; k++$ )  
            if ( $r[k] == r2$ )  
                 $r[k] = r1;$   
    }

# Vector de reprezentanți - Exemplu

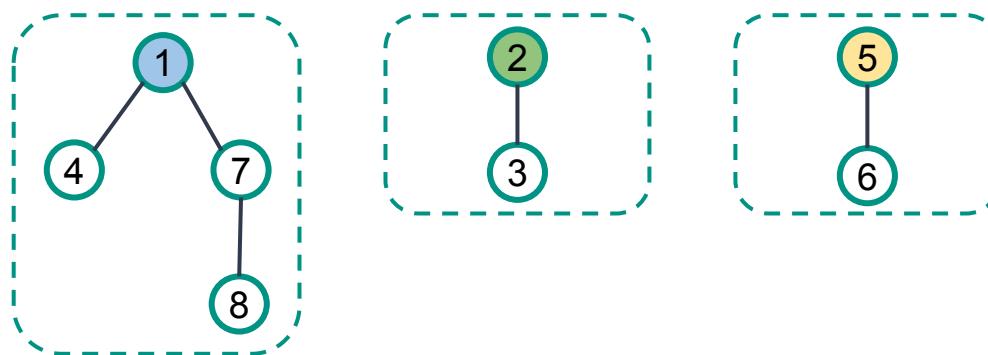


Reuneste(4, 3)  $\Rightarrow$



# Operații cu multimi disjuncte

**Varianta 2** - memorăm vârfurile fiecărei multimi ca un arbore (memorat cu tata), având ca reprezentant rădăcina



tata:

1	2	3	4	5	6	7	8
0	0	2	1	0	5	1	7

# Păduri de mulțimi disjuncte

**Varianta 2** - memorăm vârfurile fiecărei mulțimi ca un arbore (memorat cu tata), având ca reprezentant rădăcina

- **Inițializare(u)** - O(1)

```
void Initializare(int u) { tata[u] = h[u] = 0; }
```

- **Reprez(u)**

- determinarea rădăcinii arborelui care conține u
- liniar în înălțimea arborelui

```
int Reprez(int u) {
    while (tata[u] != 0)
        u = tata[u];
    return u;
}
```

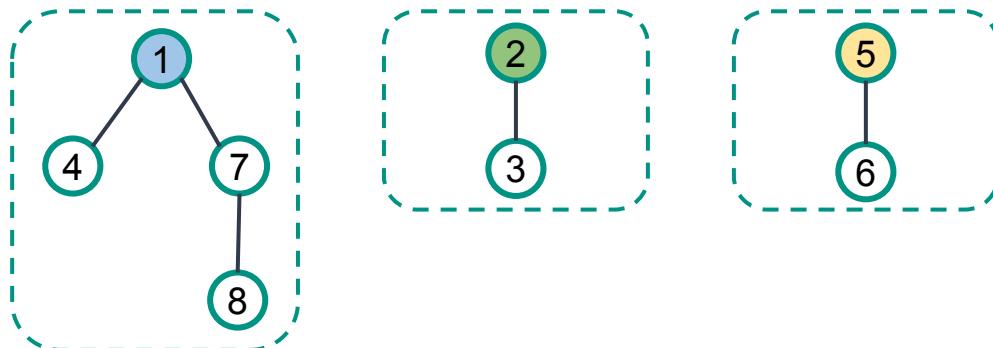
- **Reunește(u, v)**

- reuniune ponderată
- **în funcție de înălțimea arborilor**
- O(1) după determinarea reprezentanților lui u și v

```
void Reuneste(int u, int v) {
    int ru=Reprez(u), rv=Reprez(v);
    if (h[ru] > h[rv])
        tata[rv] = ru;
    else {
        tata[ru] = rv;
        if (h[ru] == h[rv])
            h[rv]++;
    }
}
```

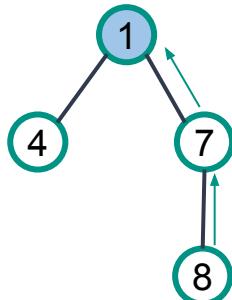
- **Arbore de înălțime logaritmică**

# Păduri de mulțimi disjuncte - Exemplu



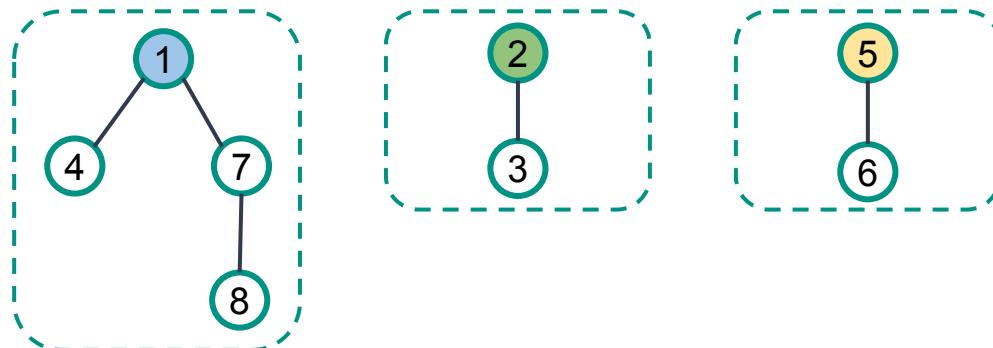
1	2	3	4	5	6	7	8
0	0	2	1	0	5	1	7
2	1	0	0	1	0	1	0

Reprez(8)  $\Rightarrow$  returnează 1



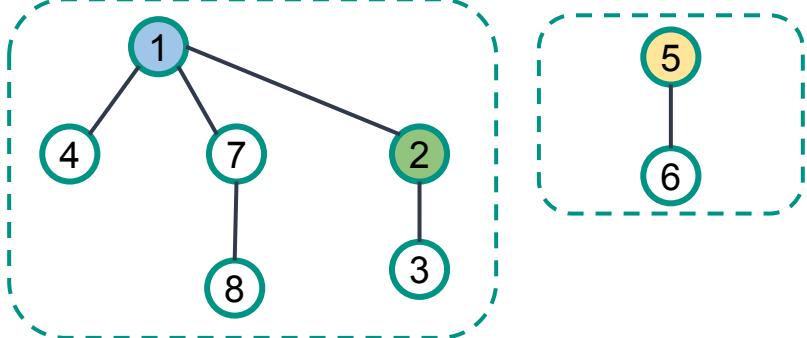
tata[8] = 7, tata[7] = 1, tata[1] = 0

# Păduri de mulțimi disjuncte - Exemplu



1	2	3	4	5	6	7	8	
tata:	0	0	2	1	0	5	1	7
h:	2	1	0	0	1	0	1	0

Reuneste(4, 3)  $\Rightarrow$  deoarece  $h[1] > h[2]$ , se va seta tata[2] = 1 (h nu se modifică)



1	2	3	4	5	6	7	8	
tata:	0	1	2	1	0	5	1	7

# Păduri de multimi disjuncte

## Reprez(u) - Optimizare - compresie de cale

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

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

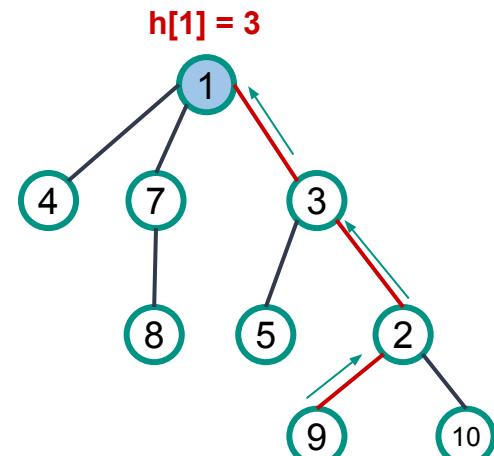
!! h nu se actualizează

De exemplu, după apelul **Reprez(9)** pentru arborele din dreapta,



rezultatul va fi 1, iar arborele devine

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



# Păduri de multimi disjuncte

## Reprez(u) - Optimizare - compresie de cale

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

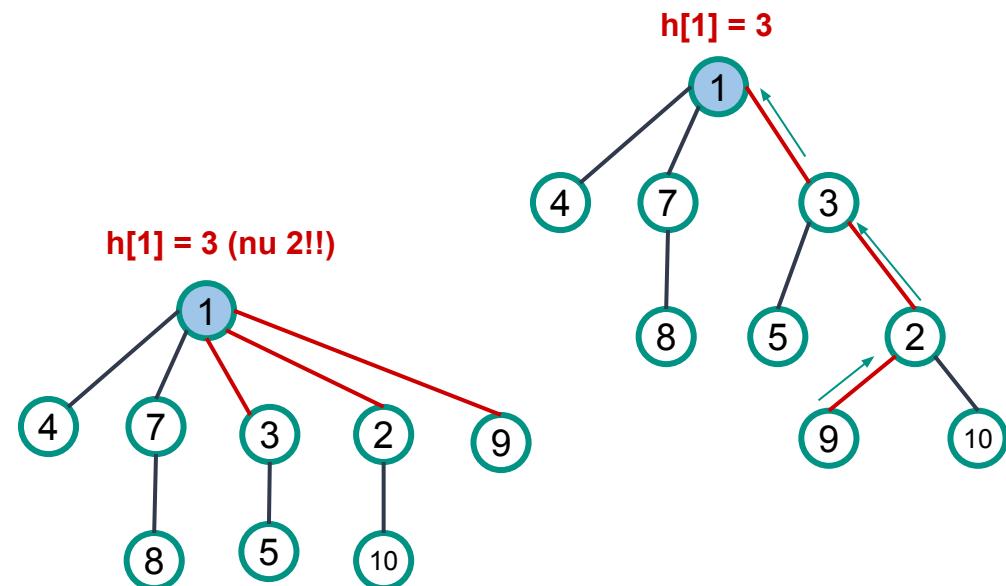
(vârfurile de pe acest lanț, parcurs pentru a găsi reprezentantul lui u, vor deveni fii ai rădăcinii, pentru ca reprezentantul lor să fie găsit mai ușor în căutările ulterioare)

!! h nu se actualizează

De exemplu, după apelul **Reprez(9)** pentru arborele din dreapta,

rezultatul va fi 1, iar arborele devine

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



# Algoritmul lui Kruskal

Implementare cu păduri disjuncte

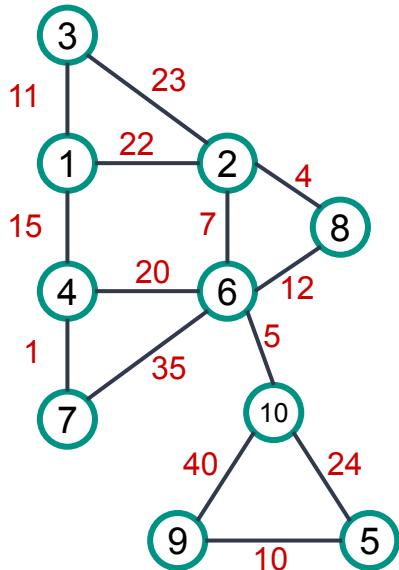
# Kruskal - Pseudocod

**sorteaza**(E)

```
for(v=1; v<=n; v++)
    Initializare(v)
```

nrm sel=0

```
for(uv ∈ E)
    if (Reprez(u) != Reprez(v)) {
        E(T) = E(T) ∪ {uv}
        Reuneste(u,v)
        nrm sel = nrm sel +1
        if (nrm sel == n-1)
            STOP // break
    }
```



## Ordine muchii

(4, 7) (4, 6)

(2, 8) (1, 2)

$$(6, 10) \quad (2, 3)$$

(2, 6) (5, 10)

(5, 9) (6, 7)

(1, 3) (9 ,10)

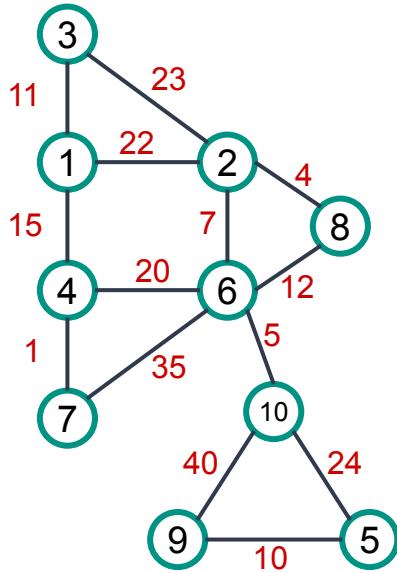
(6, 8)

(1, 4)

Pădurea de multimi disjuncte la pasul curent



1 2 3 4 5 6 7 8 9 10



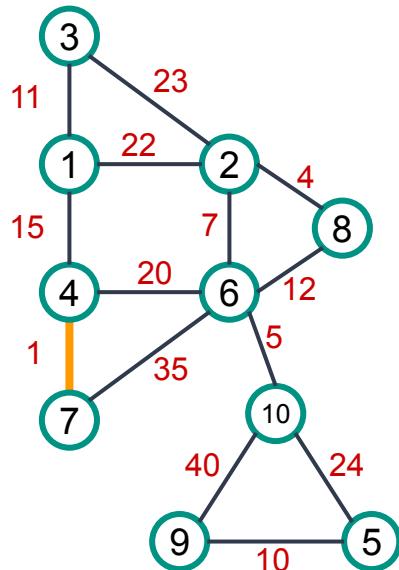
Ordine muchii

- |               |         |
|---------------|---------|
| <b>(4, 7)</b> | (4, 6)  |
| (2, 8)        | (1, 2)  |
| (6, 10)       | (2, 3)  |
| (2, 6)        | (5, 10) |
| (5, 9)        | (6, 7)  |
| (1, 3)        | (9, 10) |
| (6, 8)        |         |
| (1, 4)        |         |

Muchia curentă  
**(4, 7):**

Pădurea de multimi disjuncte la pasul curent





Ordine muchii

- |               |         |
|---------------|---------|
| <b>(4, 7)</b> | (4, 6)  |
| (2, 8)        | (1, 2)  |
| (6, 10)       | (2, 3)  |
| (2, 6)        | (5, 10) |
| (5, 9)        | (6, 7)  |
| (1, 3)        | (9, 10) |
| (6, 8)        |         |
| (1, 4)        |         |

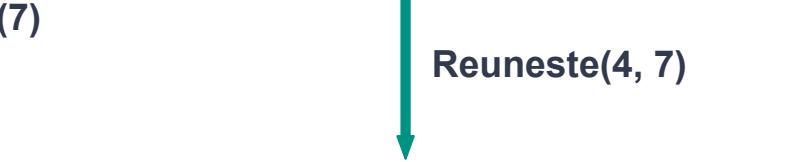
Pădurea de multimi disjuncte la pasul curent



Muchia curentă

**(4, 7):**

**Reprez(4) ≠ Reprez(7)**

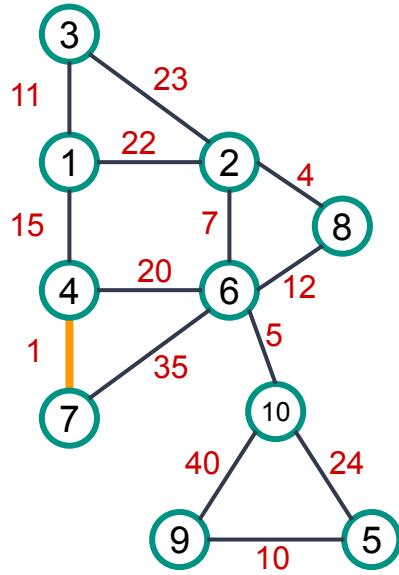


**Reuneste(4, 7)**



1 2 3 4 5 6 7 8 9 10

tata	0	0	0	<b>7</b>	0	0	0	0	0
h	0	0	0	0	0	0	<b>1</b>	0	0

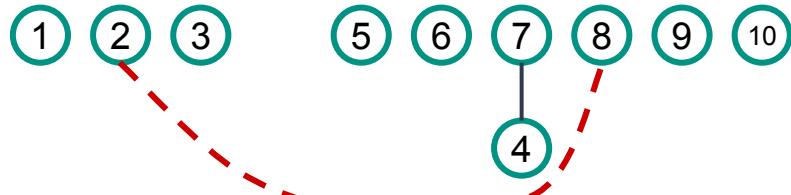


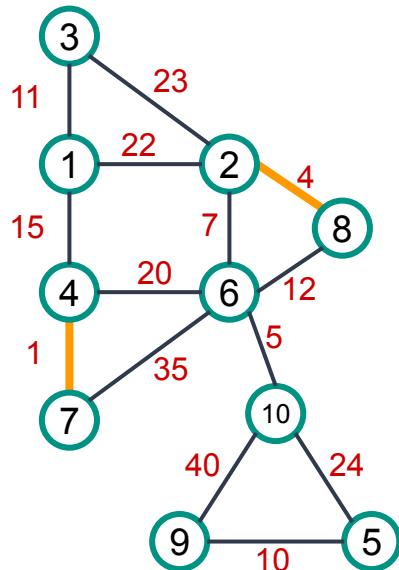
Ordine muchii

- |               |         |
|---------------|---------|
| (4, 7)        | (4, 6)  |
| <b>(2, 8)</b> | (1, 2)  |
| (6, 10)       | (2, 3)  |
| (2, 6)        | (5, 10) |
| (5, 9)        | (6, 7)  |
| (1, 3)        | (9, 10) |
| (6, 8)        |         |
| (1, 4)        |         |

Muchia curentă  
**(2, 8):**

Pădurea de multimi disjuncte la pasul curent





Ordine muchii

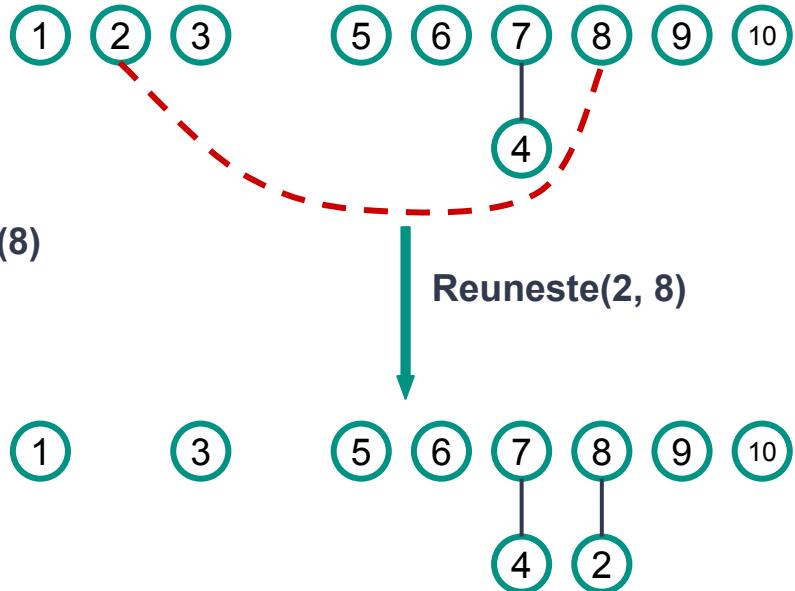
(4, 7)	(4, 6)
<b>(2, 8)</b>	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Muchia curentă

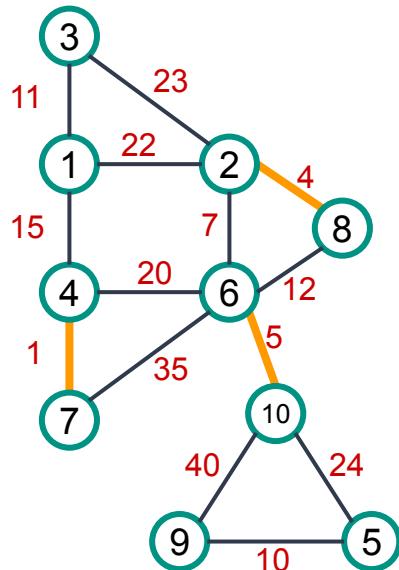
**(2, 8):**

**Reprez(2)  $\neq$  Reprez(8)**

Pădurea de multimi disjuncte la pasul curent



	1	2	3	4	5	6	7	8	9	10
tata	0	<b>8</b>	0	7	0	0	0	0	0	0
h	0	0	0	0	0	0	1	<b>1</b>	0	0



Ordine muchii

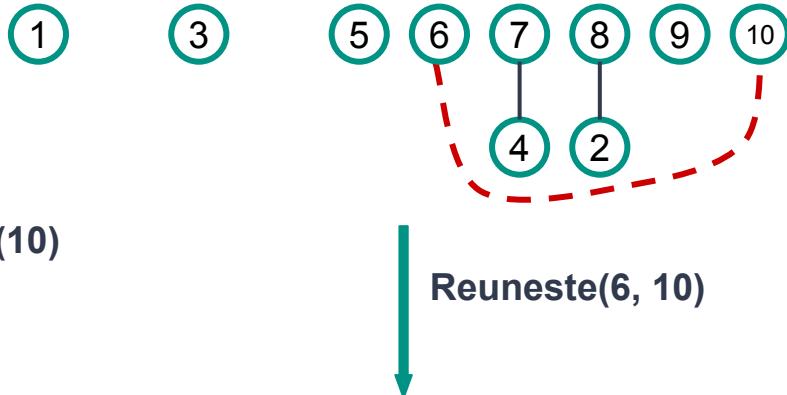
(4, 7)	(4, 6)
(2, 8)	(1, 2)
<b>(6, 10)</b>	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Pădurea de multimi disjuncte la pasul curent

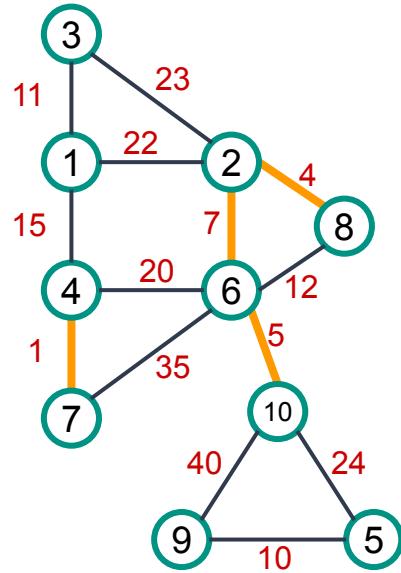
Muchia curentă

**(6, 10):**

**Reprez(6) ≠ Reprez(10)**



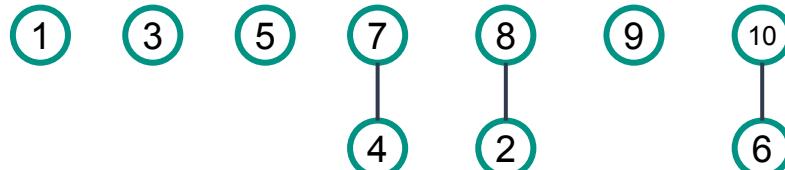
	1	2	3	4	5	6	7	8	9	10
tata	0	8	0	7	0	<b>10</b>	0	0	0	0
h	0	0	0	0	0	0	1	1	0	<b>1</b>



Ordine muchii

- |               |         |
|---------------|---------|
| (4, 7)        | (4, 6)  |
| (2, 8)        | (1, 2)  |
| (6, 10)       | (2, 3)  |
| <b>(2, 6)</b> | (5, 10) |
| (5, 9)        | (6, 7)  |
| (1, 3)        | (9, 10) |
| (6, 8)        |         |
| (1, 4)        |         |

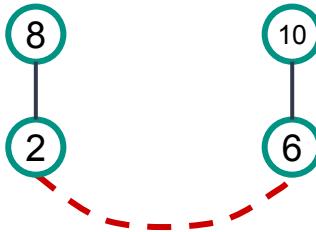
Pădurea de multimi disjuncte la pasul curent

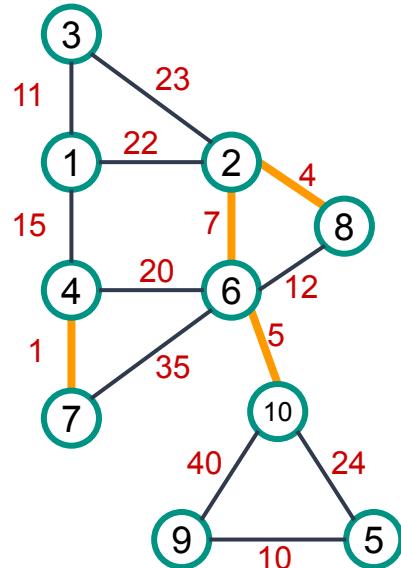


Muchia curentă

**(2, 6):**

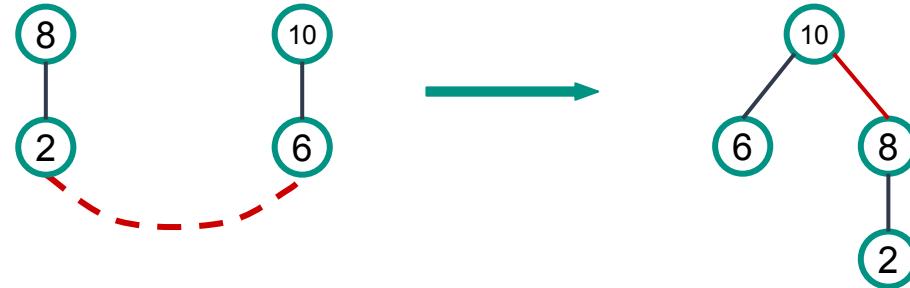
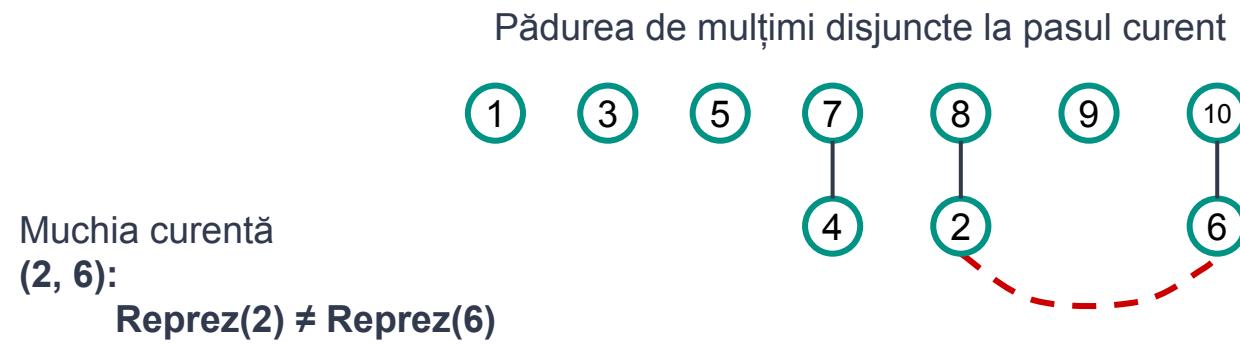
**Reprez(2)  $\neq$  Reprez(6)**

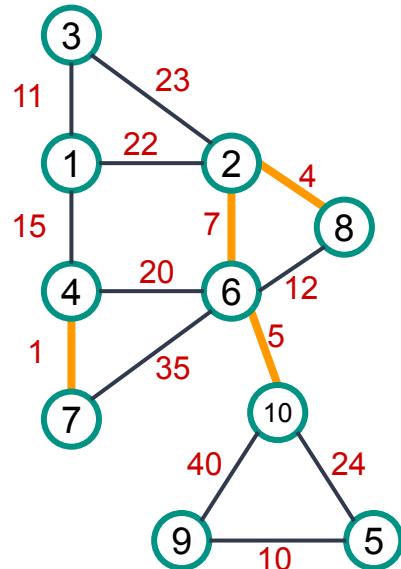




Ordine muchii

- |               |         |
|---------------|---------|
| (4, 7)        | (4, 6)  |
| (2, 8)        | (1, 2)  |
| (6, 10)       | (2, 3)  |
| <b>(2, 6)</b> | (5, 10) |
| (5, 9)        | (6, 7)  |
| (1, 3)        | (9, 10) |
| (6, 8)        |         |
| (1, 4)        |         |





Ordine muchii

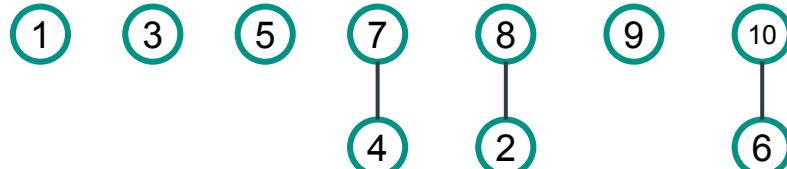
(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
<b>(2, 6)</b>	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Pădurea de multimi disjuncte la pasul curent

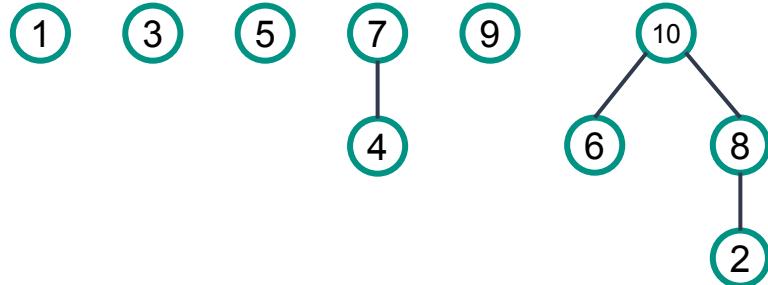
Muchia curentă

**(2, 6):**

**Reprez(2)  $\neq$  Reprez(6)**

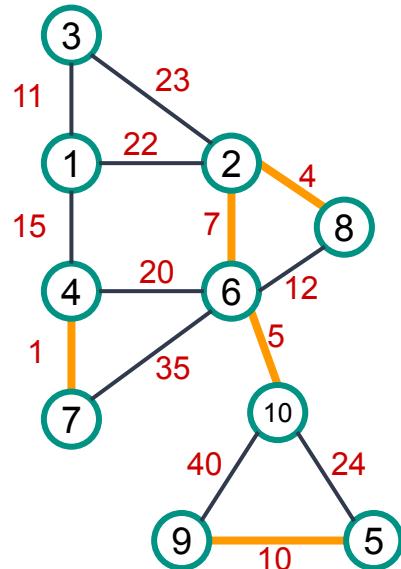


**Reuneste(2, 6)**



1 2 3 4 5 6 7 8 9 10

tata	0	8	0	7	0	10	0	<b>10</b>	0	0
h	0	0	0	0	0	0	1	1	0	<b>1</b>



Ordine muchii

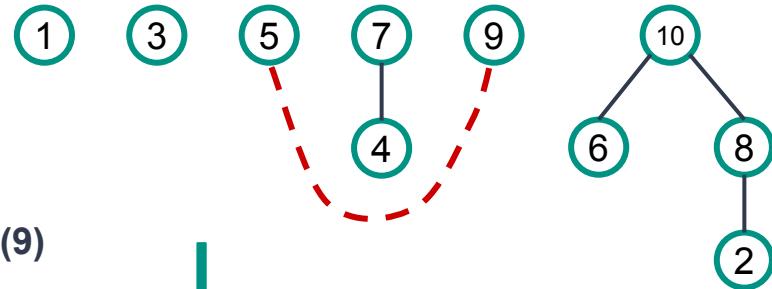
(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
<b>(5, 9)</b>	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Muchia curentă

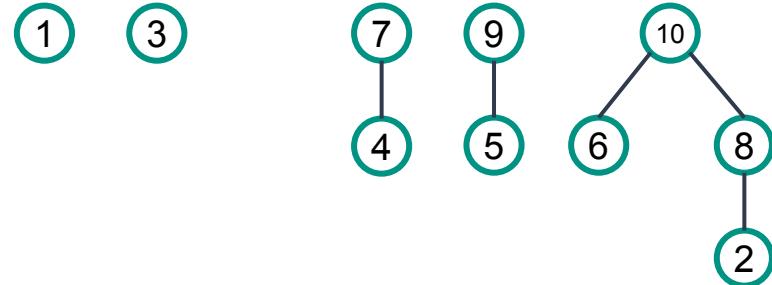
**(5, 9):**

**Reprez(5) ≠ Reprez(9)**

Pădurea de multimi disjuncte la pasul curent

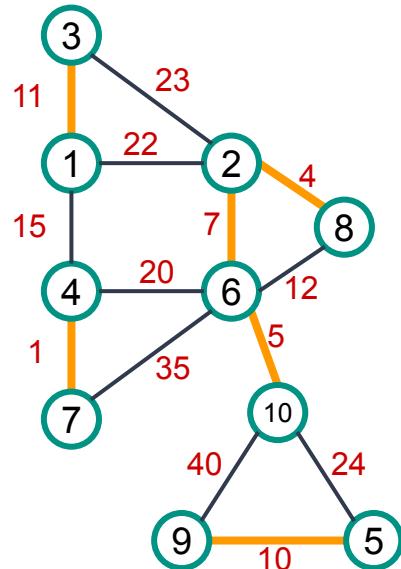


**Reuneste(5, 9)**



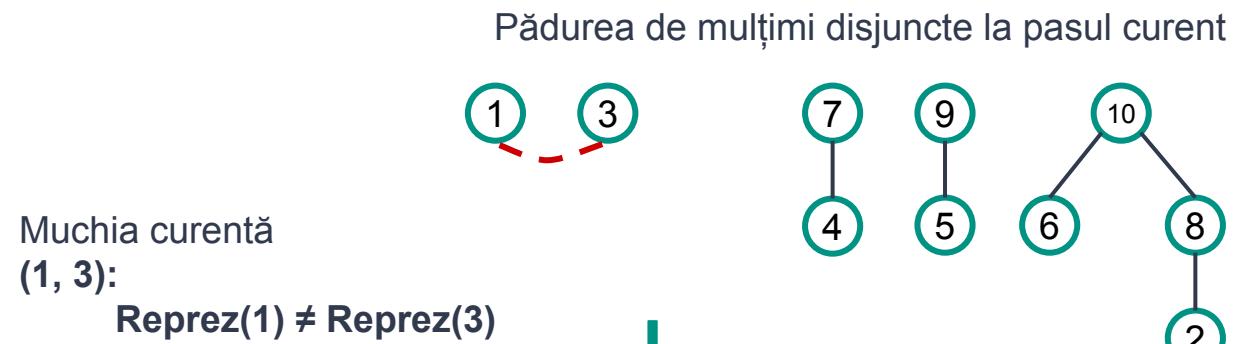
1 2 3 4 5 6 7 8 9 10

tata	0	8	0	7	<b>9</b>	10	0	10	0	0
h	0	0	0	0	0	0	1	1	<b>1</b>	1



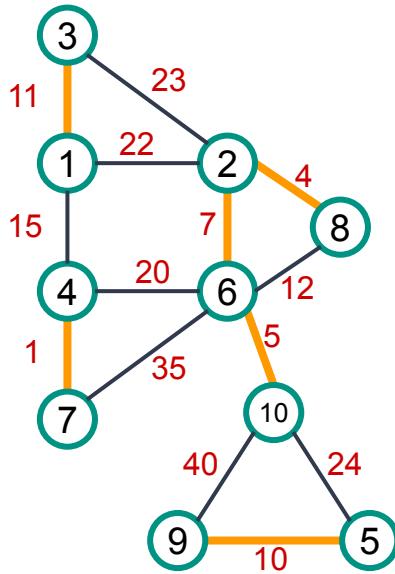
Ordine muchii

(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
<b>(1, 3)</b>	(9, 10)
(6, 8)	
(1, 4)	



1 2 3 4 5 6 7 8 9 10

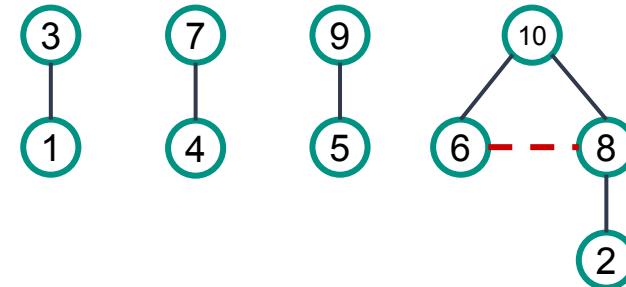
tata	3	8	0	7	9	10	0	10	0	0
h	0	0	1	0	0	0	1	1	1	1



Ordine muchii

(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
<b>(6, 8)</b>	
(1, 4)	

Pădurea de multimi disjuncte la pasul curent



Muchia curentă

**(6, 8):**

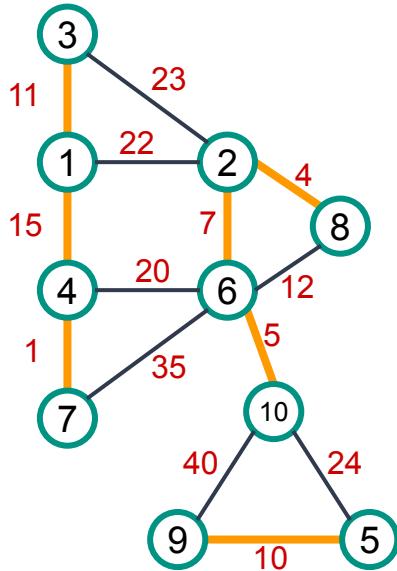
**Reprez(6) = Reprez(8)**

**⇒ nu este selectată**

**Observație:** Până acum, în funcția Reprez nu a fost modificat vectorul tata prin compresie de cale, deoarece vârfurile erau la distanță cel mult 1 față de rădăcină.

1 2 3 4 5 6 7 8 9 10

tata	3	8	0	7	9	10	0	10	0	0
h	0	0	1	0	0	0	1	1	1	1



## Ordine muchii

(4, 7) (4, 6)

(2, 8) (1, 2)

$$(6, 10) \quad (2, 3)$$

(2, 6) (5, 10)

(5, 9) (6, 7)

(1, 3) (9 ,10)

(6, 8)

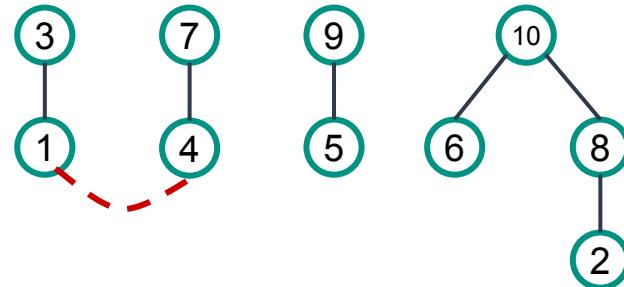
(1, 4)

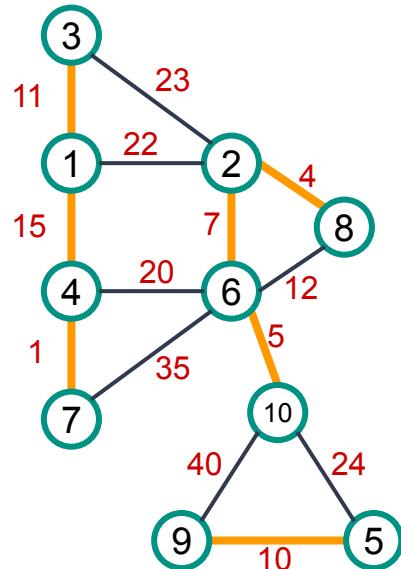
Muchia curentă

(1, 4):

## Reprez(1) ≠ Reprez(4)

#### Pădurea de multimi disjuncte la pasul curent





Ordine muchii

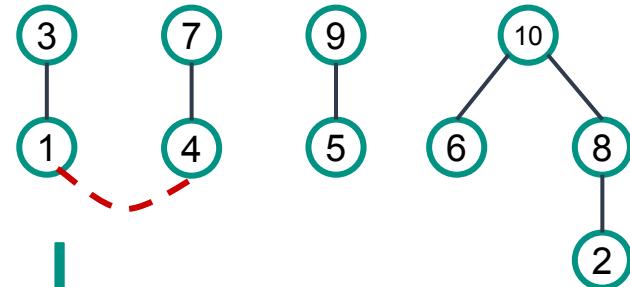
(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
<b>(1, 4)</b>	

Pădurea de multimi disjuncte la pasul curent

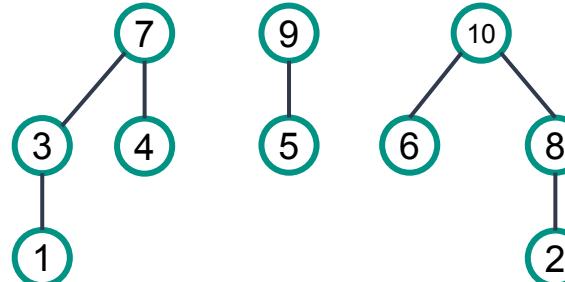
Muchia curentă

**(1, 4):**

**Reprez(1)  $\neq$  Reprez(4)**

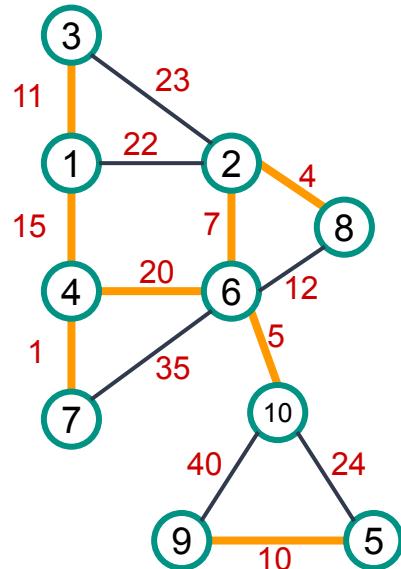


**Reuneste(1, 4)**



1 2 3 4 5 6 7 8 9 10

tata	3	8	7	7	9	10	0	10	0	0
h	0	0	1	0	0	0	2	1	1	1



Ordine muchii

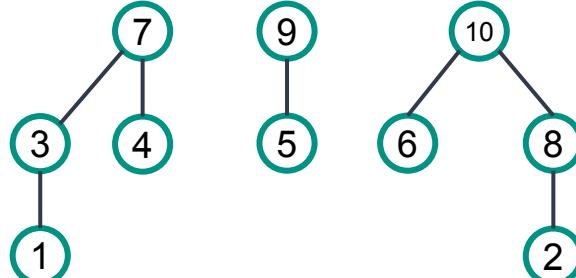
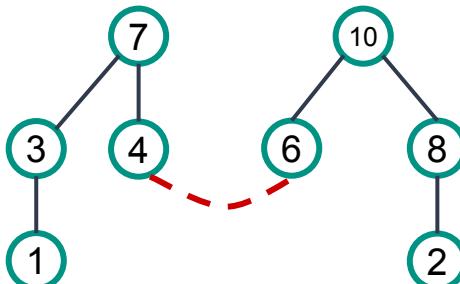
- |         |               |
|---------|---------------|
| (4, 7)  | <b>(4, 6)</b> |
| (2, 8)  | (1, 2)        |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

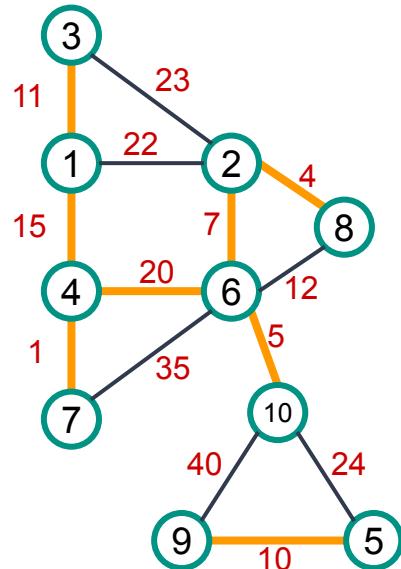
Pădurea de multimi disjuncte la pasul curent

Muchia curentă

(4, 6):

**Reprez(4)  $\neq$  Reprez(6)**





Ordine muchii

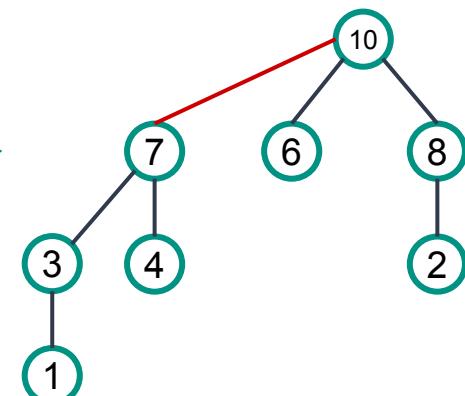
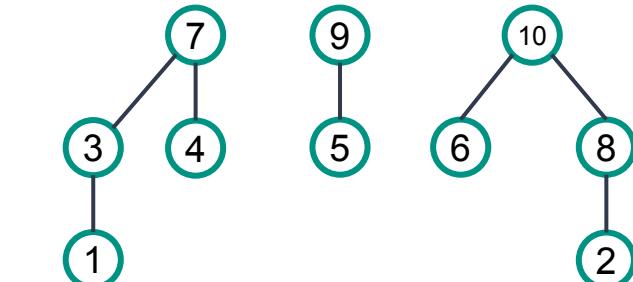
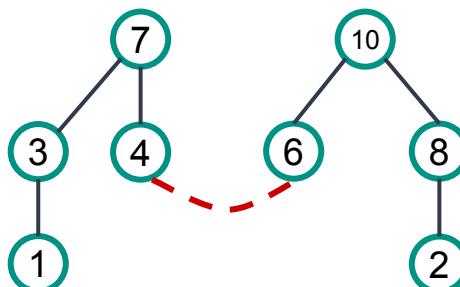
(4, 7)	<b>(4, 6)</b>
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

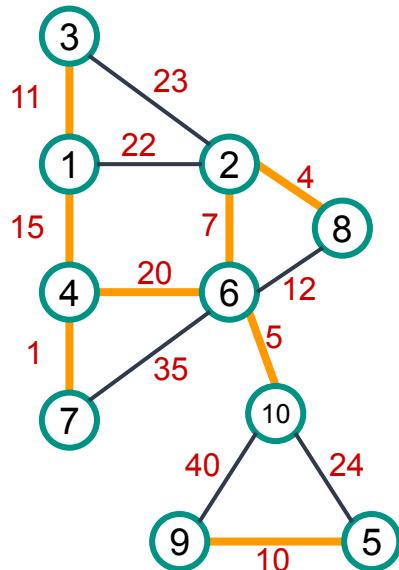
Pădurea de multimi disjuncte la pasul curent

Muchia curentă

**(4, 6):**

**Reprez(4) ≠ Reprez(6)**





Ordine muchii

- |         |               |
|---------|---------------|
| (4, 7)  | <b>(4, 6)</b> |
| (2, 8)  | (1, 2)        |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

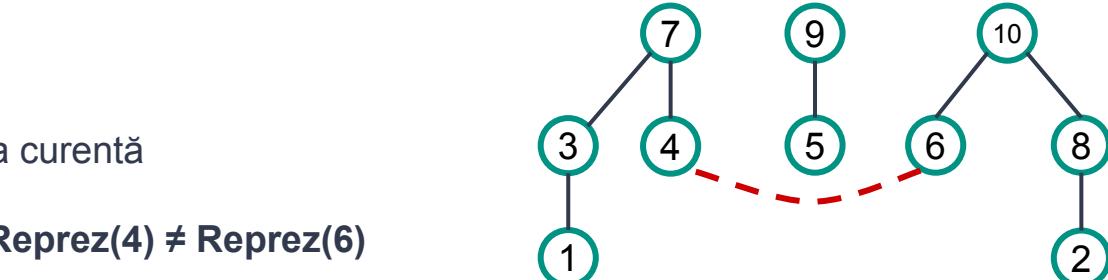
Pădurea de multimi disjuncte la pasul curent

Muchia curentă

(4, 6):

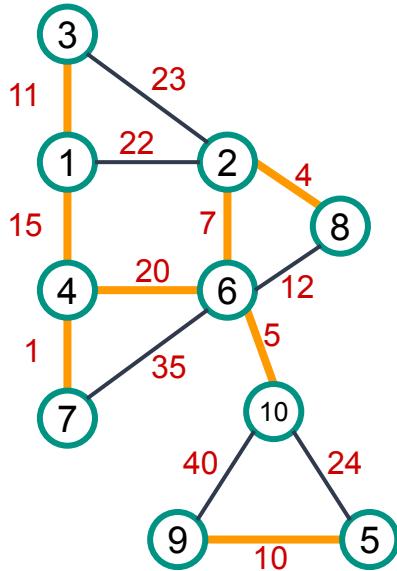
$\text{Reprez}(4) \neq \text{Reprez}(6)$

**Reuneste(4, 6)**



1 2 3 4 5 6 7 8 9 10

tata	3	8	7	7	9	10	<b>10</b>	10	0	0
h	0	0	1	0	0	0	2	1	1	<b>3</b>



Ordine muchii

- |         |               |
|---------|---------------|
| (4, 7)  | (4, 6)        |
| (2, 8)  | <b>(1, 2)</b> |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

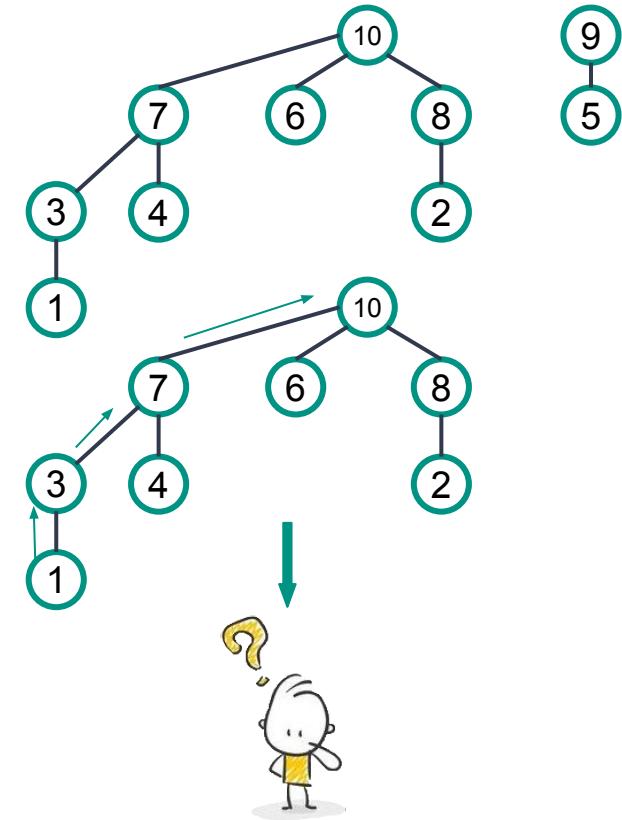
Muchia curentă

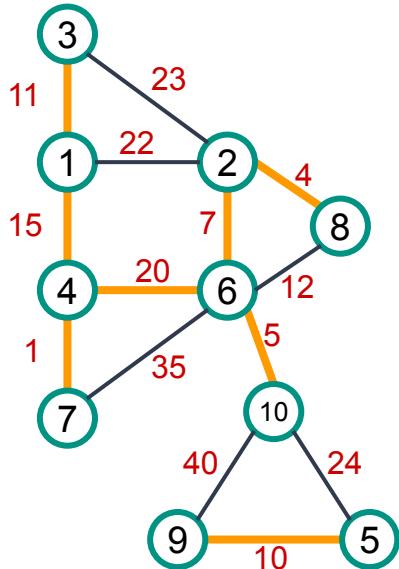
(1, 2):

Reprez(1):  $\Rightarrow 10 +$   
compresie de cale

!! h nu se modifică  
(h[7] rămâne 2)

Pădurea de multimi disjuncte la pasul curent





Ordine muchii

(4, 7)	(4, 6)
(2, 8)	<b>(1, 2)</b>
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

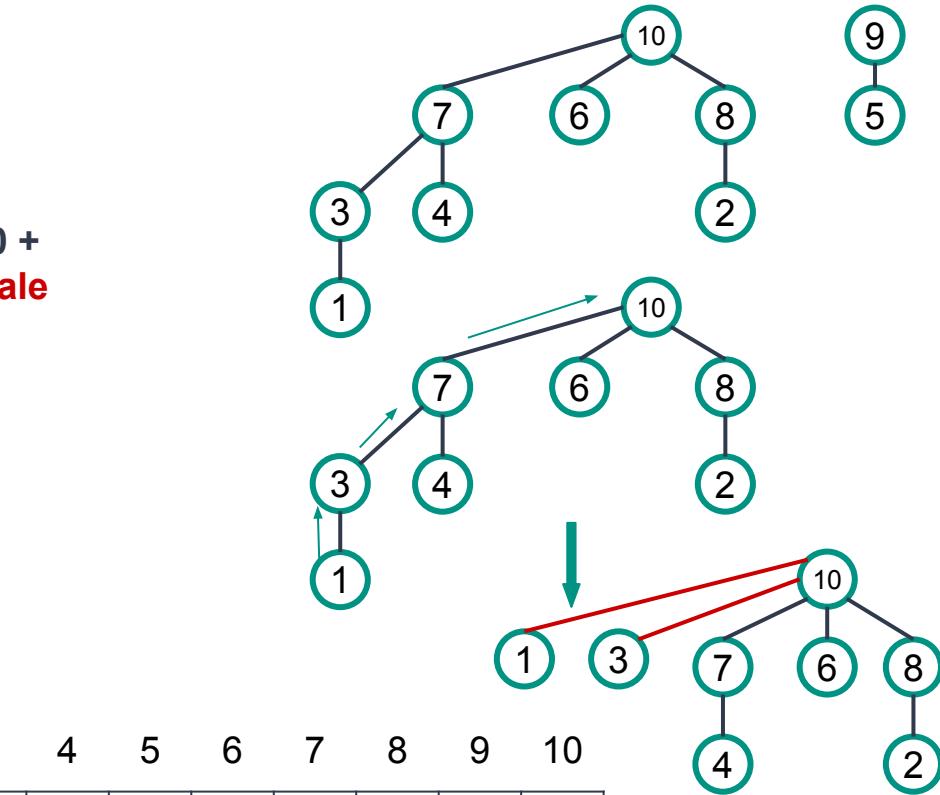
Pădurea de multimi disjuncte la pasul curent

Muchia curentă

(1, 2):

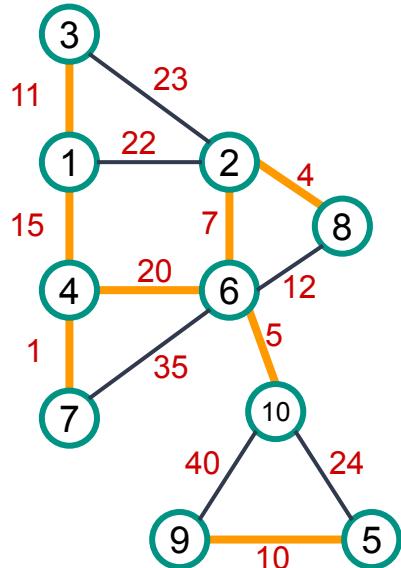
Reprez(1):  $\Rightarrow 10 +$   
compresie de cale

!! h nu se modifică  
(h[7] rămâne 2)



1 2 3 4 5 6 7 8 9 10

tata	10	8	10	7	9	10	10	0	0
h	0	0	1	0	0	0	2	1	1



Ordine muchii

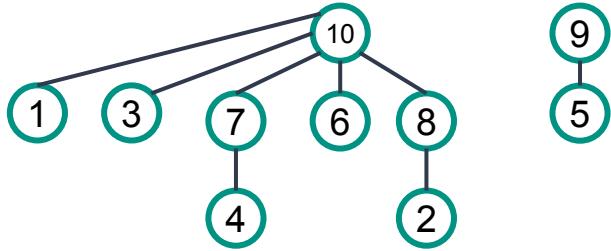
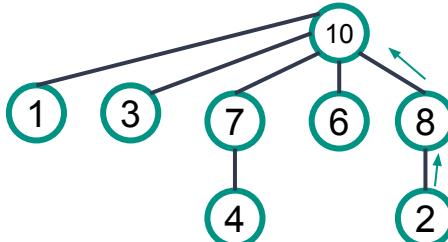
- |         |               |
|---------|---------------|
| (4, 7)  | (4, 6)        |
| (2, 8)  | <b>(1, 2)</b> |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

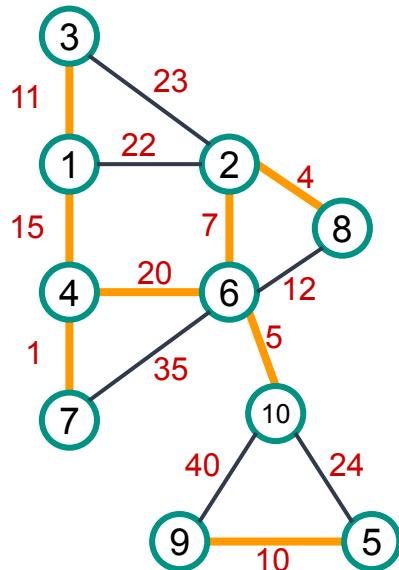
Pădurea de multimi disjuncte la pasul curent

Muchia curentă

(1, 2):

Reprez(2):  $\Rightarrow 10 +$   
**compresie de cale**





Ordine muchii

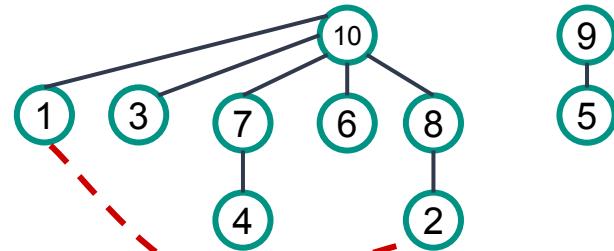
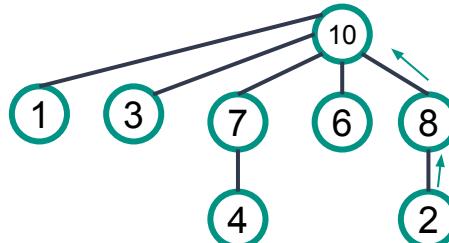
- |         |               |
|---------|---------------|
| (4, 7)  | (4, 6)        |
| (2, 8)  | <b>(1, 2)</b> |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

Pădurea de multimi disjuncte la pasul curent

Muchia curentă

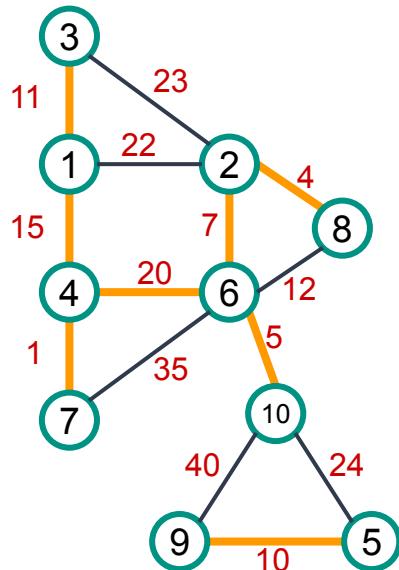
(1, 2):

Reprez(2):  $\Rightarrow 10 +$   
**compresie de cale**



1 2 3 4 5 6 7 8 9 10

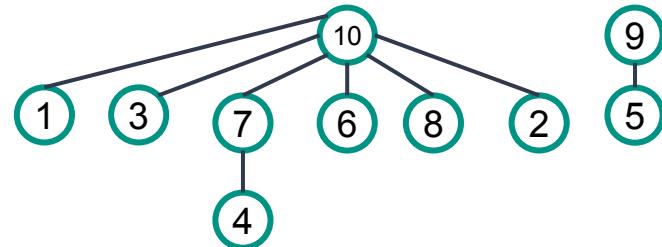
tata	10	<b>10</b>	10	7	9	10	10	0	0
h	0	0	1	0	0	0	2	1	1



Ordine muchii

- |         |               |
|---------|---------------|
| (4, 7)  | (4, 6)        |
| (2, 8)  | <b>(1, 2)</b> |
| (6, 10) | (2, 3)        |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

Pădurea de multimi disjuncte la pasul curent



Muchia curentă

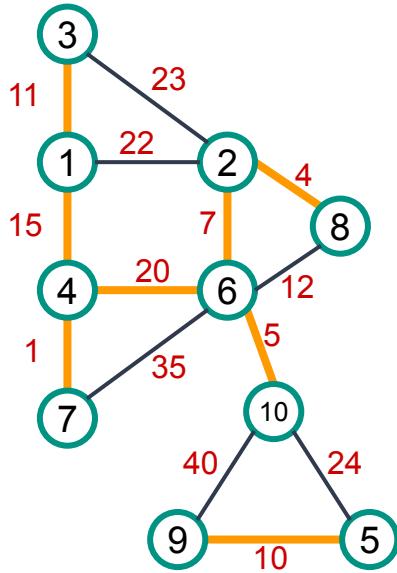
(1, 2):

Reprez(1) = 10

Reprez(2) = 10  $\Rightarrow$  nu este selectată

1 2 3 4 5 6 7 8 9 10

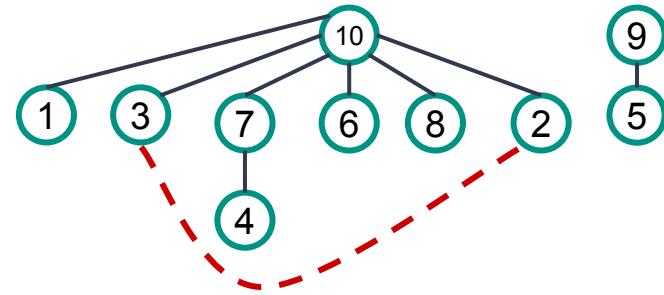
tata	10	10	10	7	9	10	10	10	0	0
h	0	0	1	0	0	0	2	1	1	3



Ordine muchii

- |         |               |
|---------|---------------|
| (4, 7)  | (4, 6)        |
| (2, 8)  | (1, 2)        |
| (6, 10) | <b>(2, 3)</b> |
| (2, 6)  | (5, 10)       |
| (5, 9)  | (6, 7)        |
| (1, 3)  | (9, 10)       |
| (6, 8)  |               |
| (1, 4)  |               |

Pădurea de multimi disjuncte la pasul curent



Muchia curentă

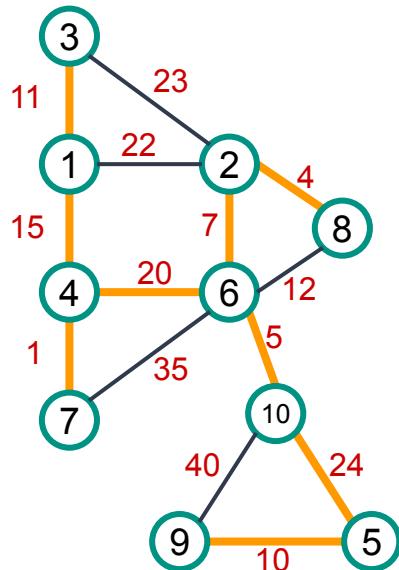
**(2, 3):**

**Reprez(2) = Reprez(3)  
⇒ nu este selectată**

**2 și 3 sunt fii ai rădăcinii, compresia de cale nu modifică vectorul tata**

1 2 3 4 5 6 7 8 9 10

tata	10	10	10	7	9	10	10	0	0
h	0	0	1	0	0	0	2	1	1

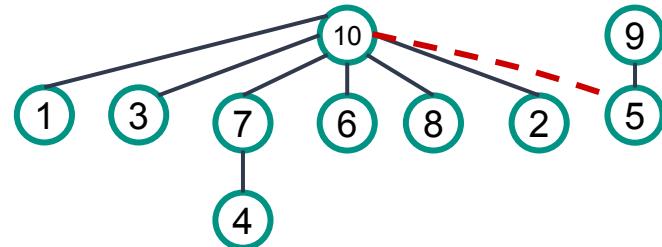


Ordine muchii

(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	<b>(5, 10)</b>
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Muchia curentă  
**(5, 10):**  
**Reprez(5) ≠ Reprez(10)**

Pădurea de multimi disjuncte la pasul curent

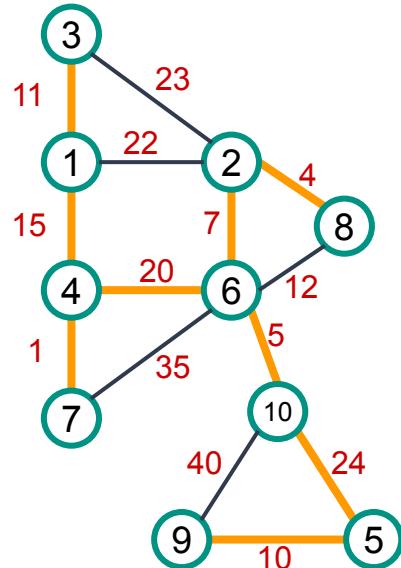


**Reuneste(5, 10)  
reuniune ponderată**



1 2 3 4 5 6 7 8 9 10

tata	10	10	10	7	9	10	10	0	0
h	0	0	1	0	0	0	2	1	1



Ordine muchii

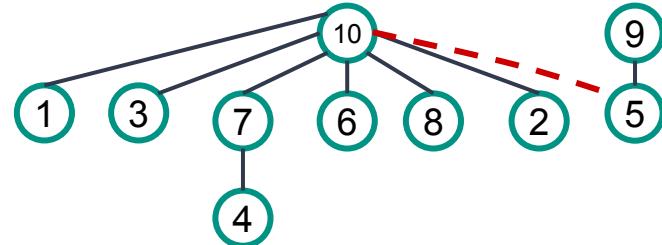
(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	<b>(5, 10)</b>
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Muchia curentă

**(5, 10):**

**Reprez(5) ≠ Reprez(10)**

Pădurea de multimi disjuncte la pasul curent

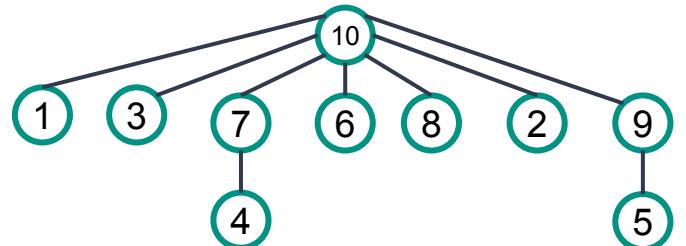


**Reuneste(5, 10)**

**reuniune ponderată**

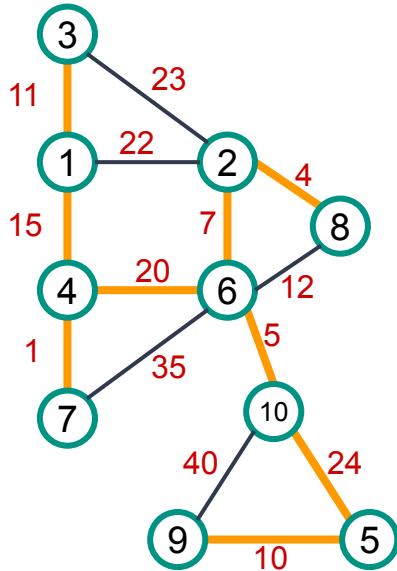
**$h[\text{Reprez}(5)] = h[9] = 1$**

**$< h[\text{Reprez}(10)] = h[10] = 3$**



1 2 3 4 5 6 7 8 9 10

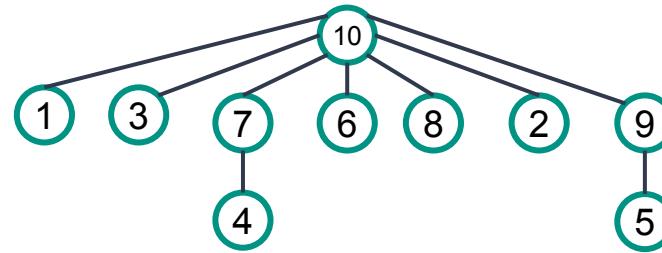
tata	10	10	10	7	9	10	10	<b>10</b>	0
h	0	0	1	0	0	0	2	1	1



Ordine muchii

(4, 7)	(4, 6)
(2, 8)	(1, 2)
(6, 10)	(2, 3)
(2, 6)	(5, 10)
(5, 9)	(6, 7)
(1, 3)	(9, 10)
(6, 8)	
(1, 4)	

Pădurea de multimi disjuncte la pasul curent



STOP - au fost selectate n-1 muchii

Muchii apcm  $\neq$  muchiile din pădurea de multimi disjuncte finală (formată dintr-un singur arbore)





# Drumuri minime în grafuri ponderate



# Aplicații



- Dată o hartă rutieră, să se determine**
  - un drum minim între două orașe date
  - câte un drum minim între oricare două orașe de pe hartă

# Aplicații

- **Determinarea de drumuri minime / distanțe - numeroase aplicații**
  - probleme de rutare
  - robotică
  - procesarea imaginilor
  - strategii jocuri
  - probleme de planificare (drumuri critice)

# Cadru

Fie:

- G - un graf **orientat** ponderat**
- P - un drum**

$$w(P) = \sum_{e \in E(P)} w(e)$$

**costul / ponderea / lungimea drumului P**

# Cadru

Fie:

- G - un graf **orientat** ponderat**
- Presupunem că G nu conține circuite de pondere negativă**

# Cadru

- Fie  $s, v \in V, s \neq v$
- Distanța de la  $s$  la  $v$ 
  - $\delta_G(s, v) = \min \{ w(P) \mid P \text{ este drum de la } s \text{ la } v \}$

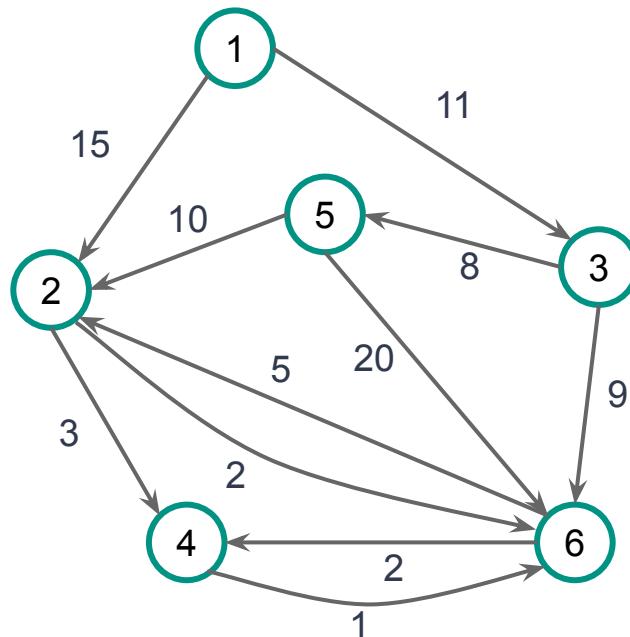
# Cadru

- Fie  $s, v \in V, s \neq v$
- Distanța de la  $s$  la  $v$ 
  - $\delta_G(s, v) = \min \{ w(P) \mid P \text{ este drum de la } s \text{ la } v \}$
  - $\delta_G(s, s) = 0$

Convenție:  $\min \emptyset = \infty$

- Un drum  $P$  de la  $s$  la  $v$  se numește drum minim de la  $s$  la  $v$  dacă  $w(P) = \delta_G(s, v)$

# Exemplu



# Aplicații

## Tipuri de probleme de drumuri minime

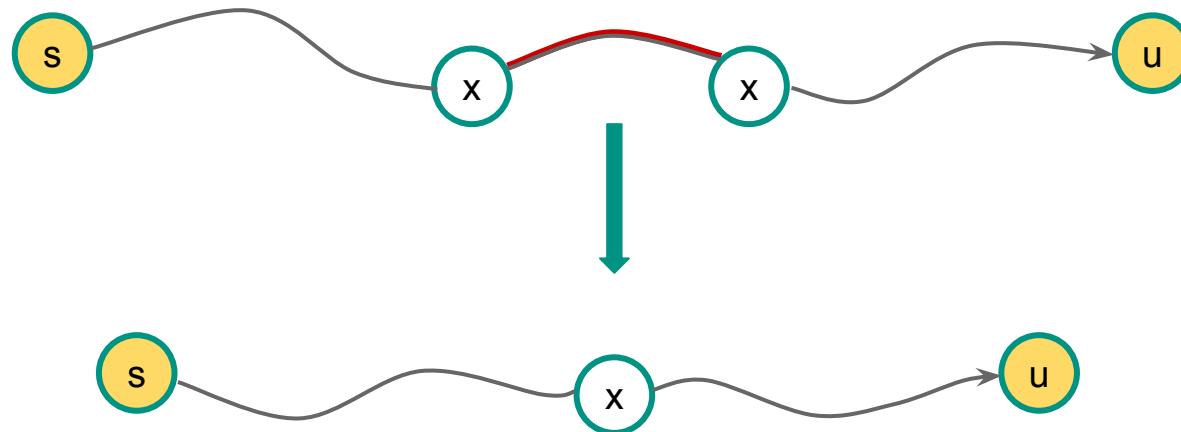
- între două vârfuri date**
- de la un vârf la toate celelalte**
- între oricare două vârfuri**

## Situări

- Sunt acceptate și arce de cost negativ?**
- Graful conține circuite?**
- Graful conține circuite de cost negativ?**

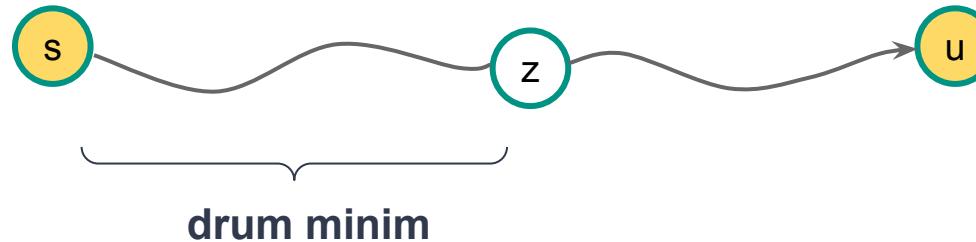
# Observații

**Observația 1.** Dacă  $P$  este un drum minim de la  $s$  la  $v$  și **nu există circuite de cost negativ**, atunci  $P$  este drum elementar.



# Observații

**Observația 2.** Dacă  $P$  este un drum minim de la  $s$  la  $u$  și  $z$  este un vârf al lui  $P$ , atunci subdrumul lui  $P$  de la  $s$  la  $z$  este drum minim de la  $s$  la  $z$ .



# Observații



Drumuri minime de la un  
vârf s dat la celealte vârfuri  
(de sursă unică)

# Problemă

Problema drumurilor minime de sursă unică (de la s la celelalte vârfuri)

Se dau:

- un graf **orientat** ponderat  $G = (V, E, w)$  cu  
 $w : E \rightarrow \mathbb{R}$
- un vârf de start **s**

Să se determine distanța de la s la fiecare vârf x al lui G / la un vârf dat t (și un arbore al distanțelor față de s / un drum minim de la s la t).

# Drumuri minime de sursă unică s



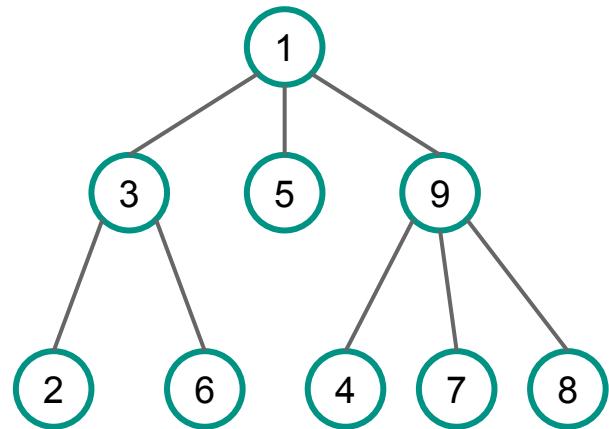
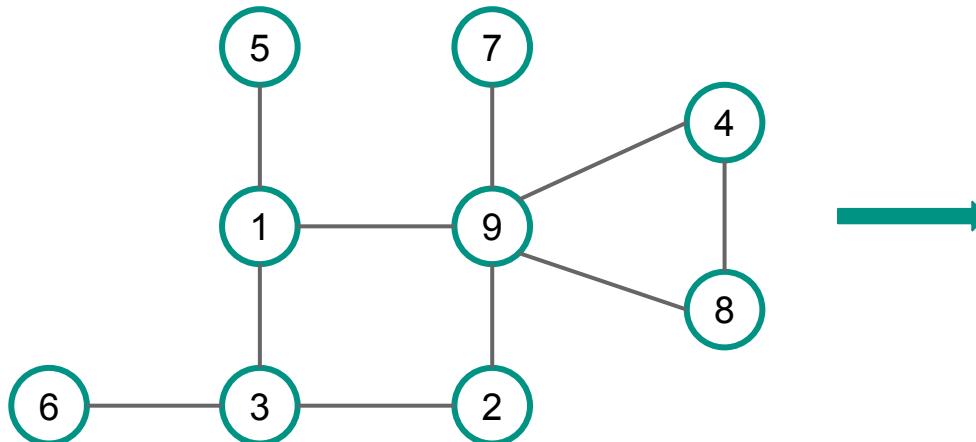
- Dacă  $G$  **nu** este ponderat, cum putem calcula distanțele față de  $s$ ?

# Drumuri minime de sursă unică s

## □ Amintim

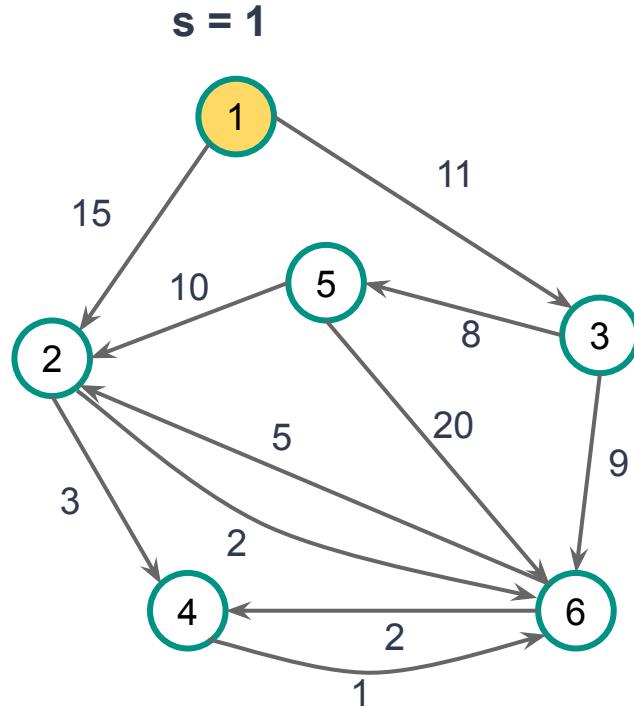
În cazul unui graf neponderat, problema se rezolvă folosind parcurgerea BF din s

⇒ arborele BF (al distanțelor față de s)

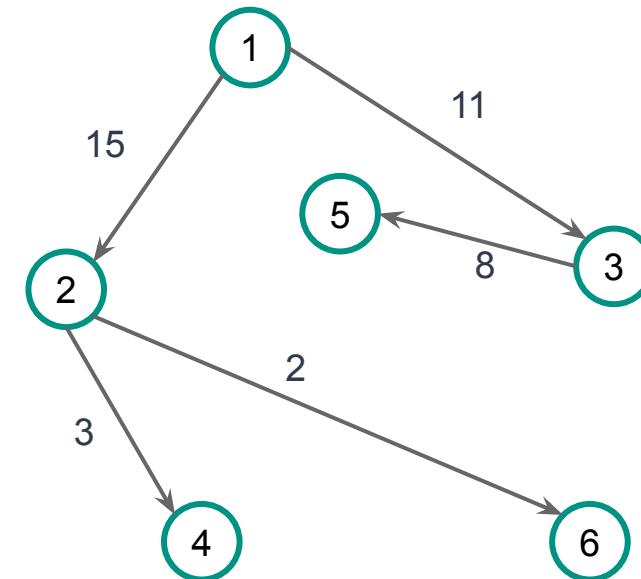
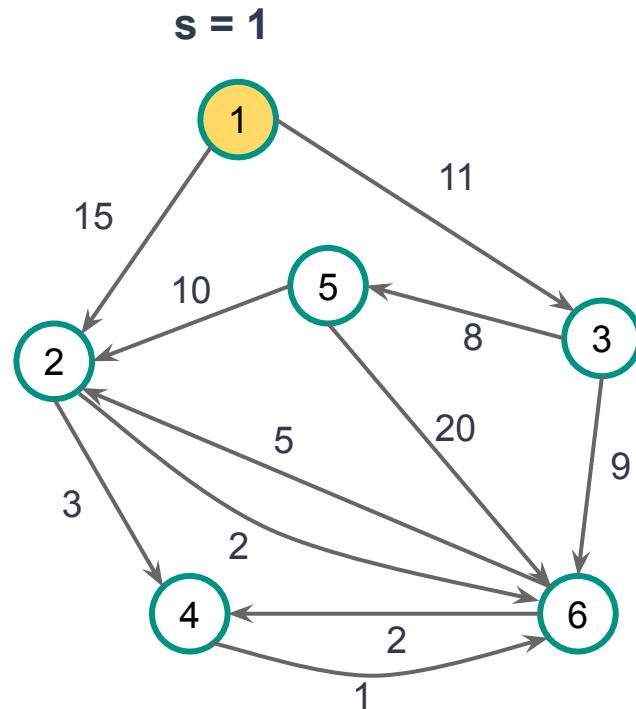


arborele BF

# Exemplu



# Exemplu



arborele distanțelor față de 1

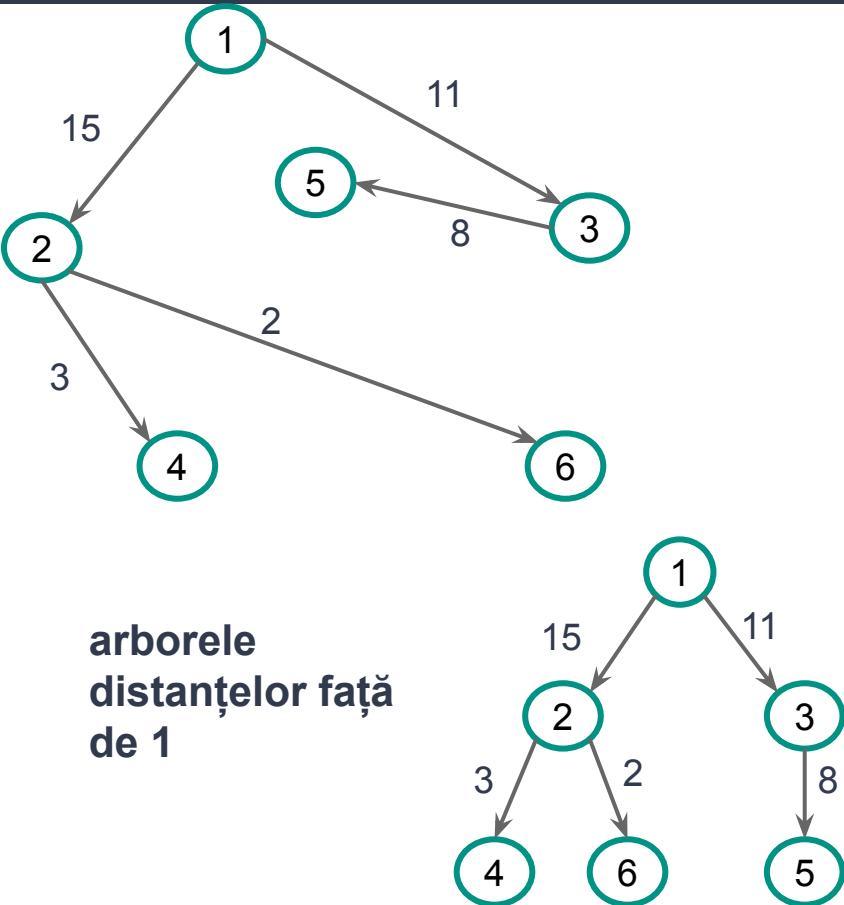
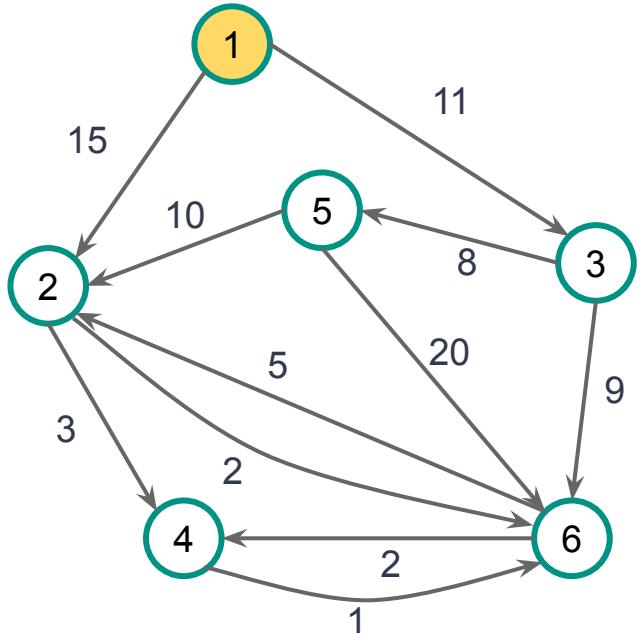
# Arborele distanțelor

**Definiție.** Pentru un vârf dat  $s$ , un arbore al distanțelor față de  $s$  = un subgraf  $T$  al lui  $G$ , care conservă distanțele de la  $s$  la celelalte vârfuri accesibile din  $s$

$$\delta_T(s, v) = \delta_G(s, v), \quad \forall v \in V \text{ accesibil din } s,$$

graful neorientat asociat lui  $T$  fiind arbore cu rădăcina în  $s$  (cu arcele corespunzătoare orientate de la  $s$  la frunze).

# Arborele distanțelor

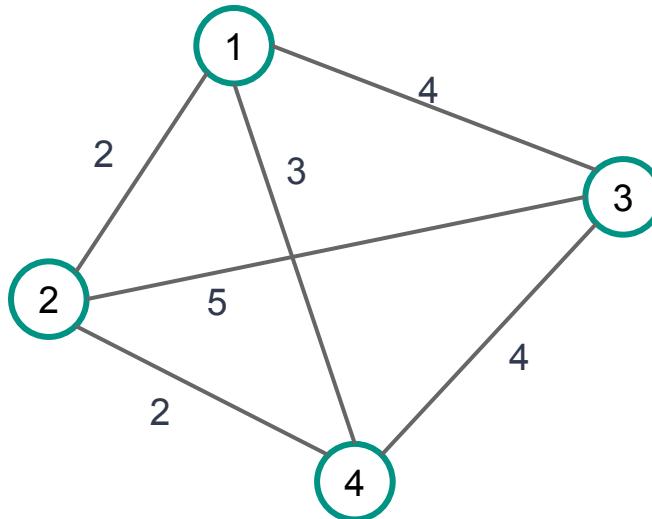


# Arborele distanțelor

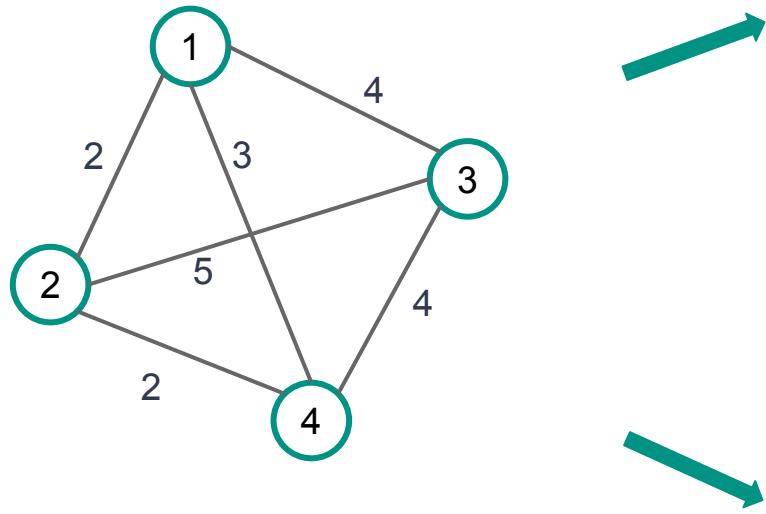
- Presupunem că **toate vârfurile sunt accesibile din s**
- Problema drumurilor minime de sursă unică este echivalentă cu determinarea unui **arbore al distanțelor față de s**

# Arborele distanțelor

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



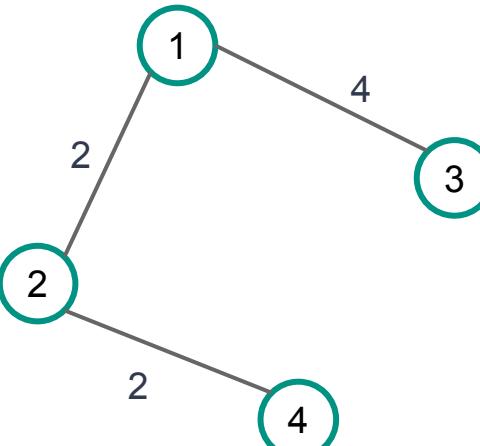
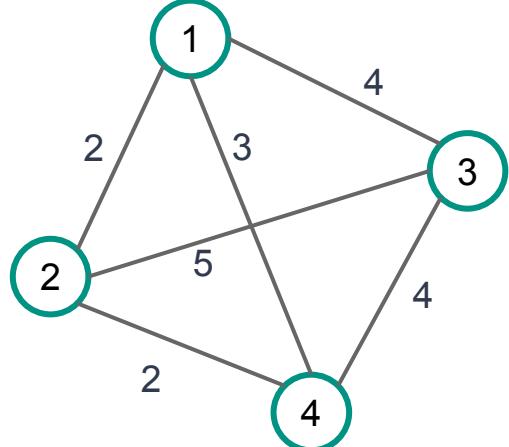
- Un arbore parțial de cost minim nu este neapărat un arbore de distanțe minime



apcm

arbore al  
distanțelor  
față de 1

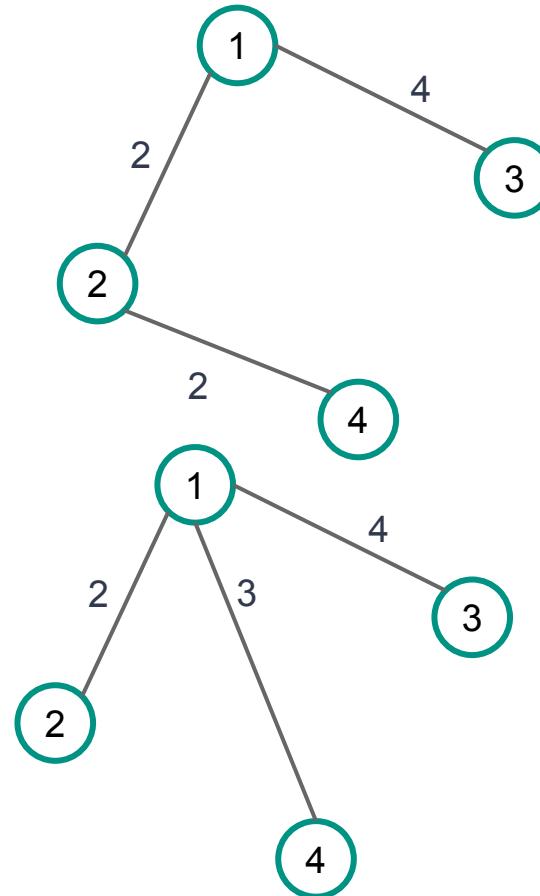
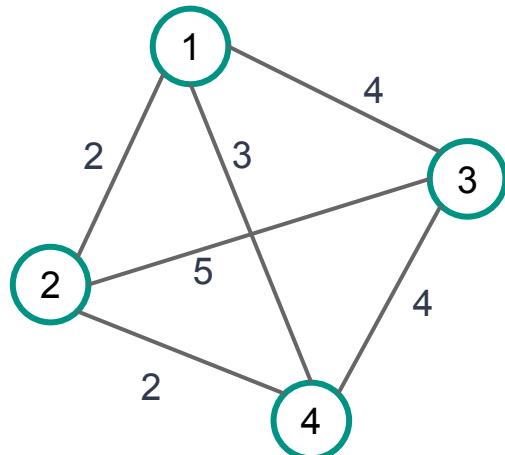
- Un arbore parțial de cost minim nu este neapărat un arbore de distanțe minime



apcm

arbore al  
distanțelor  
față de 1

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



apcm

arbore al  
distanțelor  
față de 1

# Drumuri minime de sursă unică s



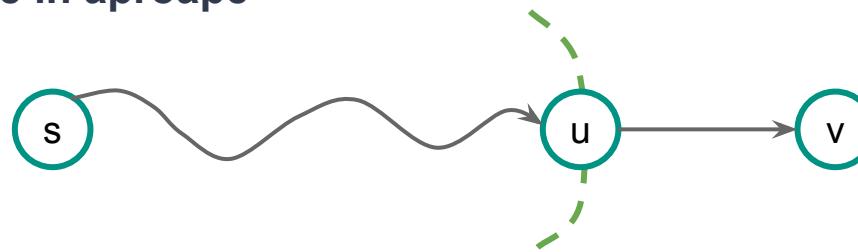
- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

# Drumuri minime de sursă unică s



- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

"din aproape în aproape"



Dacă u este predecesor al lui v pe un drum minim de la s la v  $\Rightarrow$

$$\delta(s, v) = \delta(s, u) + w(uv)$$

Stim  $\delta(s, u) \Rightarrow$  aflăm și  $\delta(s, v)$

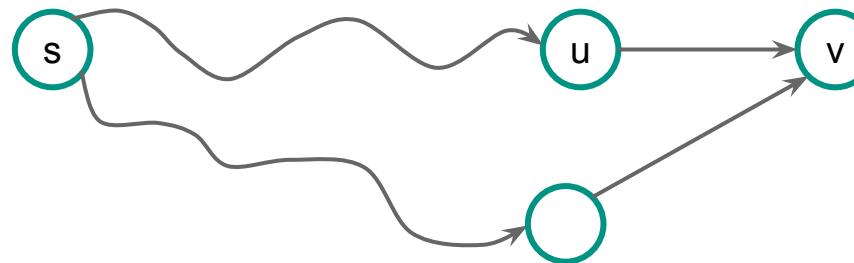
# Drumuri minime de sursă unică s



- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

"din aproape în aproape" ⇒ când considerăm un vârf  $v$ , pentru a calcula  $\delta(s, v)$ , ar fi util să știm deja  $\delta(s, u)$ , pentru  $u$  cu  $uv \in E$  (?! toate)

$$\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \mid uv \in E \}$$



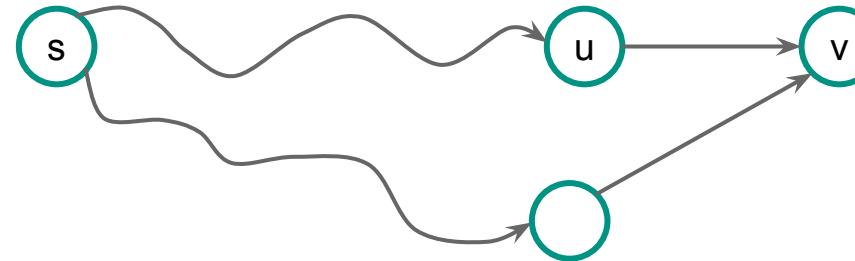
# Drumuri minime de sursă unică s

- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

"din aproape în aproape"  $\Rightarrow$  când considerăm un vârf  $v$ , pentru a calcula  $\delta(s, v)$ , ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$



**Ar fi utilă o ordonare a vârfurilor astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$**



# Drumuri minime de sursă unică s

- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

"din aproape în aproape"  $\Rightarrow$  când considerăm un vârf  $v$ , pentru a calcula  $\delta(s, v)$ , ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$

Ar fi utilă o ordonare a vârfurilor astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$



O astfel de ordonare nu există dacă graful conține circuite

# Drumuri minime de sursă unică s

- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

"din aproape în aproape"  $\Rightarrow$  când considerăm un vârf  $v$ , pentru a calcula  $\delta(s, v)$ , ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$

Ar fi utilă o ordonare a vârfurilor astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$



O astfel de ordonare nu există dacă graful conține circuite

Dacă există circuite - estimăm distanțele pe parcursul algoritmului și considerăm vârful care este estimat a fi cel mai aproape de s

# Drumuri minime de sursă unică s

- Algoritmi pentru grafuri orientate cu circuite, dar cu ponderi pozitive - **Dijkstra**
- Algoritmi pentru grafuri orientate fără circuite (cu ponderi reale) DAGs = Directed Acyclic Graphs
- Algoritmi pentru grafuri orientate cu circuite și ponderi reale, care detectează existența de circuite negative - **Bellman-Ford**

# Algoritmul lui Dijkstra

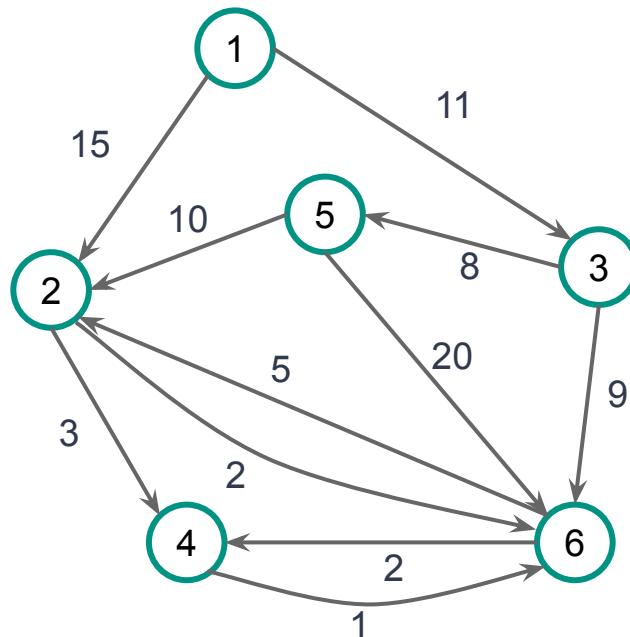


# Algoritmul lui Dijkstra

Ipoteză:

Presupunem că arcele au cost pozitiv (graful poate conține circuite)

# Exemplu



# Algoritmul lui Dijkstra

**Idee:** La un pas, este ales ca vârf curent (vizitat) vârful u care este **estimat a fi cel mai apropiat de s**

- **Estimarea pentru u** = cel mai scurt drum de la s la u determinat până la pasul curent



# Algoritmul lui Dijkstra

**Idee:** La un pas, este ales ca vârf curent (vizitat) vârful u care este **estimat a fi cel mai apropiat de s**

- **Estimarea pentru u** = cel mai scurt drum de la s la u determinat până la pasul curent
- + se descoperă noi drumuri către vecinii lui  $\Rightarrow$  **se actualizează distanțele estimate pentru vecini**



# Algoritmul lui Dijkstra

- Generalizare a ideii de parcurgere BF
- Dacă toate arcele au cost egal  $\Rightarrow$  Dijkstra  $\equiv$  BF

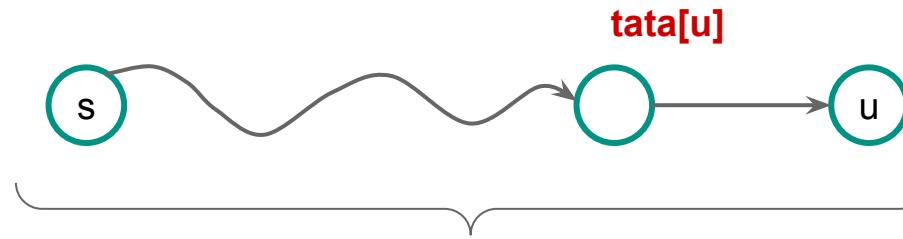
# Pseudocod



# Algoritmul lui Dijkstra

Pentru fiecare vârf, reținem etichetele:

- $d[u]$  - etichetă de distanță
- $tata[u]$



$d[u]$  = costul minim al unui drum de la  $s$  la  $u$  descoperit până la acel moment

$tata[u]$  = predecesorul lui  $u$  pe drumul de cost minim de la  $s$  la  $u$  descoperit până la acel moment

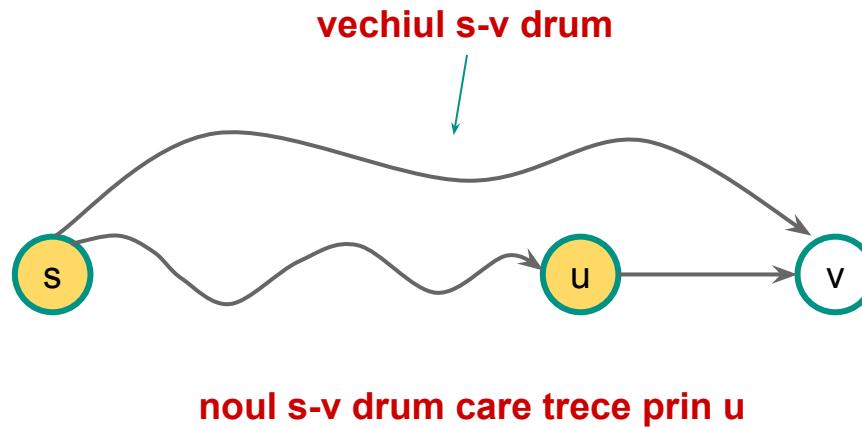
# Algoritmul lui Dijkstra

La un pas:

- este selectat un vârf  $u$  (neselectat) care “pare” cel mai apropiat de  $s \Leftrightarrow$  are eticheta  $d$  minimă
- se actualizează etichetele  $d[v]$  ale vecinilor lui  $u$  - considerând drumuri care trec prin  $u$ 
  - **tehnică de relaxare a arcelor care ies din  $u$**

# Algoritmul lui Dijkstra

Relaxarea unui arc  $(u, v)$  = verificarea dacă  $d[v]$  poate fi îmbunătățit, trecând prin vârful  $u$



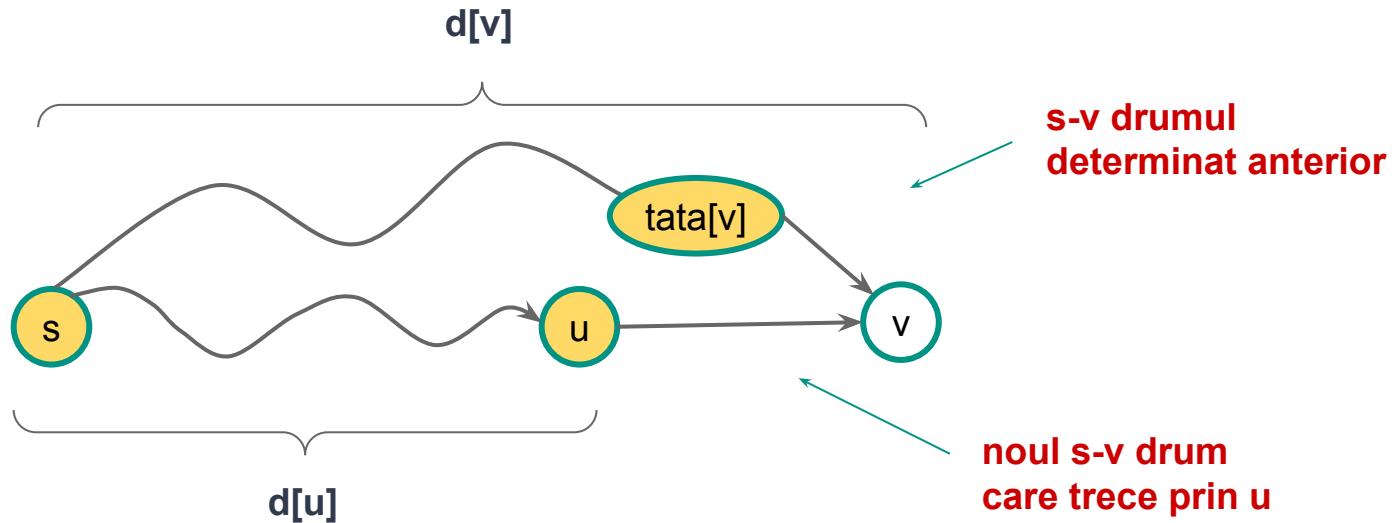
# Algoritmul lui Dijkstra

Relaxarea unui arc  $(u, v)$

dacă  $d[u] + w(u, v) < d[v]$  atunci

$$d[v] = d[u] + w(u, v)$$

$$\text{tata}[v] = u$$



# Algoritmul lui Dijkstra

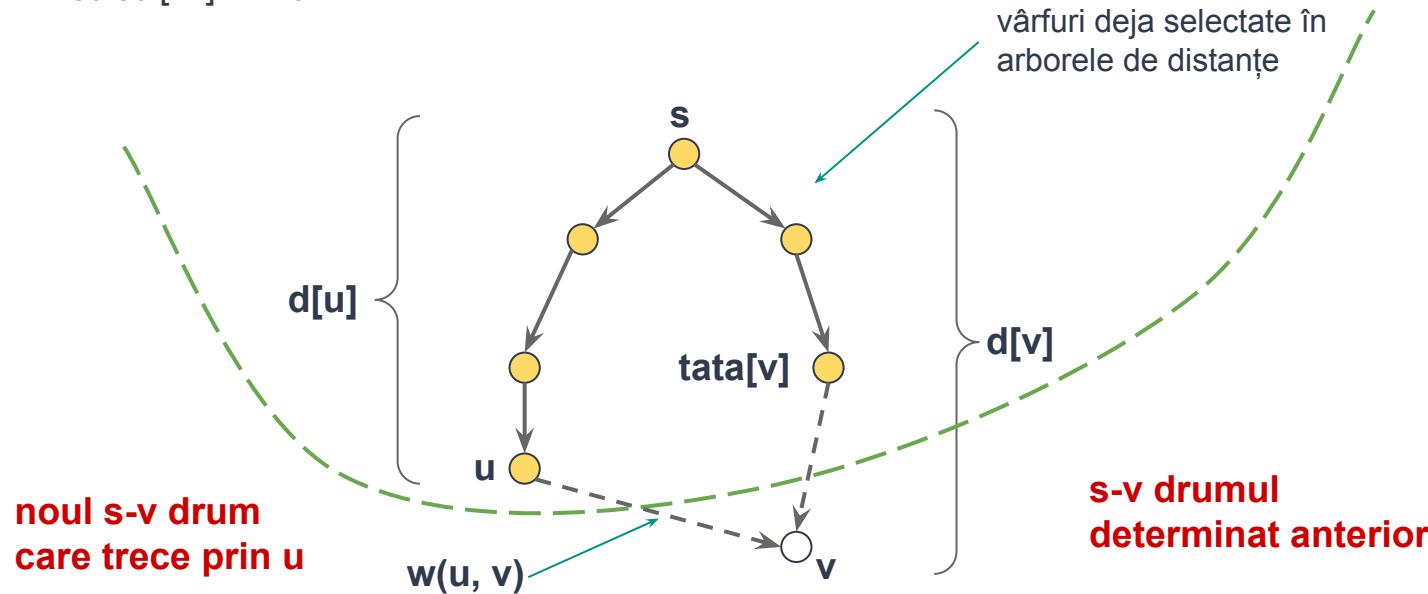
Relaxarea unui arc  $(u, v)$

dacă  $d[u] + w(u, v) < d[v]$  atunci

$$d[v] = d[u] + w(u, v)$$

$$\text{tata}[v] = u$$

Raportat la vârfuri deja selectate - similar Prim



# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

**pentru** fiecare  $u \in V$  **execută**

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută //  $\Leftrightarrow$  pentru  $i=1, n$  execută

# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută

$u =$  extrage vârf cu eticheta  $d$  minimă din  $Q$

# Algoritmul lui Dijkstra

Dijkstra( $G, w, s$ )

  inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută

$u = \text{extrage vârf cu eticheta } d \text{ minimă din } Q$

    pentru fiecare  $uv \in E$  execută // relaxare  $uv$

# Algoritmul lui Dijkstra

Dijkstra(G, w, s)

  inițializează mulțimea vârfurilor neselectate Q cu V

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută

$u =$  extrage vârf cu eticheta d minimă din Q

    pentru fiecare  $uv \in E$  execută

      dacă  ~~$v \in Q$  și~~  $d[u] + w(u, v) < d[v]$  atunci

$d[v] = d[u] + w(u, v)$

$tata[v] = u$

# Algoritmul lui Dijkstra

```
Dijkstra(G, w, s)
```

  inițializează mulțimea vârfurilor neselectate Q cu V

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută

$u = \text{extrage vârf cu eticheta } d \text{ minimă din } Q$

    pentru fiecare  $uv \in E$  execută

      dacă  $d[u] + w(u, v) < d[v]$  atunci

$d[v] = d[u] + w(u, v)$

$tata[v] = u$

scrie  $d$ ,  $tata$

// scrie drum minim de la  $s$  la un vârf  $t$  dat folosind  $tata$

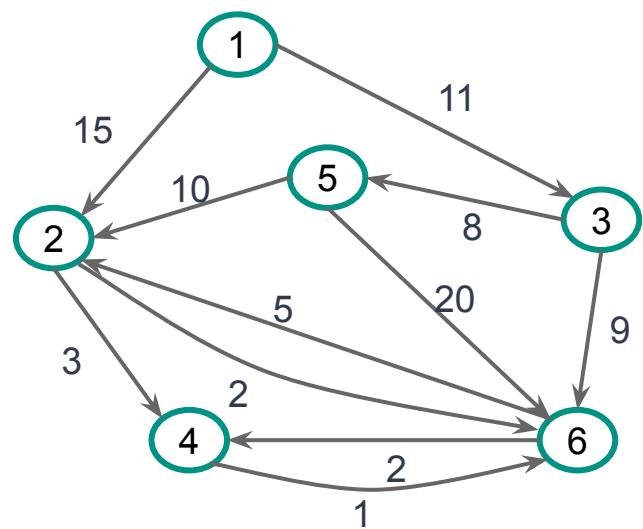
# Algoritmul lui Dijkstra

## Observatie

Vom demonstra că, atunci când  $u$  este extras din  $Q$ , eticheta lui  $d[u]$  este chiar egală cu  $\delta(s, u)$  (este corectă) și **nu se va mai actualiza**  $\Rightarrow v \in Q$

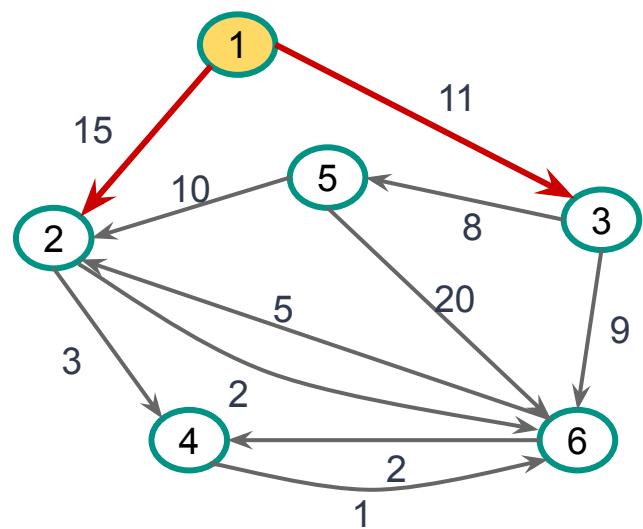
# Exemplu





---

$d/tata = [ 0/0, \infty/0, \infty/0, \infty/0, \infty/0, \infty/0 ]$



1

d/tata

[ 0/0,

$\infty/0$ ,

$\infty/0$ ,

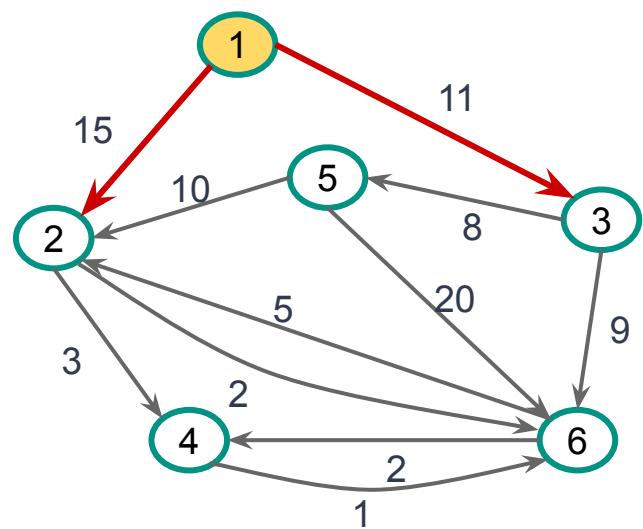
$\infty/0$ ,

$\infty/0$ ,

$\infty/0$  ]

Selectăm 1

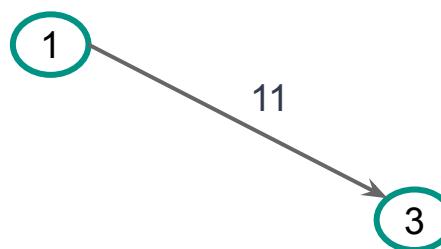
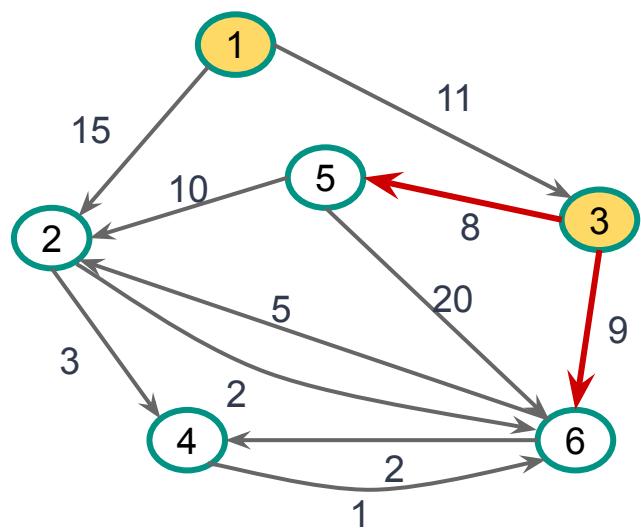
$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



1

d/tata	1	2	3	4	5	6
Selectăm 1	[ 0/0 , - ]	[ 15/1 , 11/1 ]	[ ∞/0 , ∞/0 ]	[ ∞/0 , ∞/0 ]	[ ∞/0 , ∞/0 ]	[ ∞/0 , ∞/0 ]

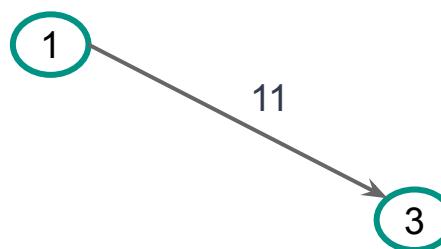
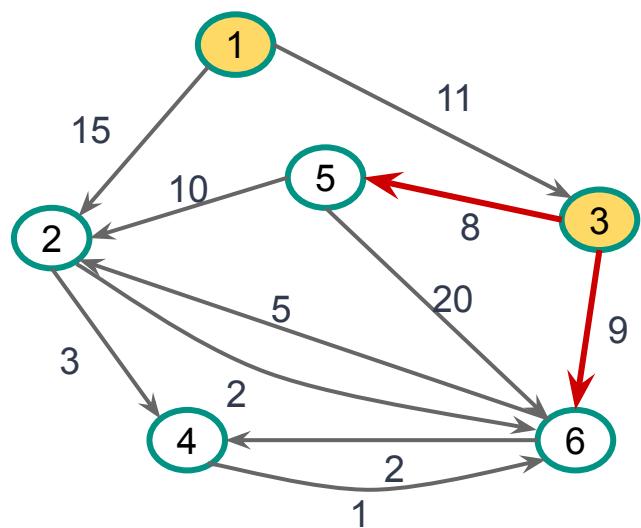
$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



d/tata	1	2	3	4	5	6
Selectăm 1	[ 0/0, - ]	$\infty/0, 15/1$	$\infty/0, 11/1$	$\infty/0, \infty/0$	$\infty/0, \infty/0$	$\infty/0, \infty/0 ]$
Selectăm 3						

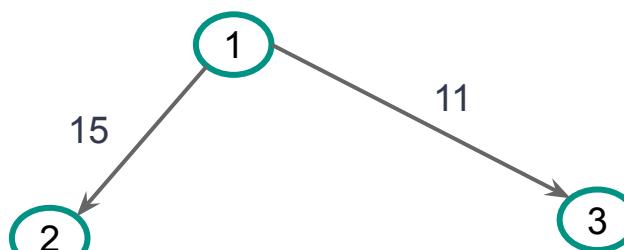
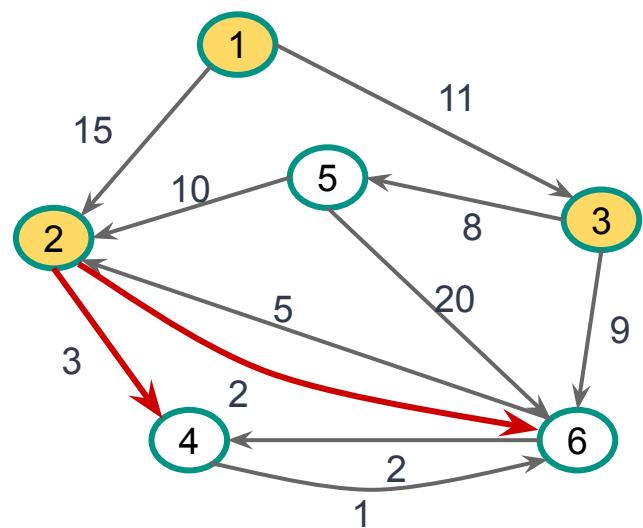
$$d[5] = \min \{ d[5], d[3] + w(3, 5) \}$$

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



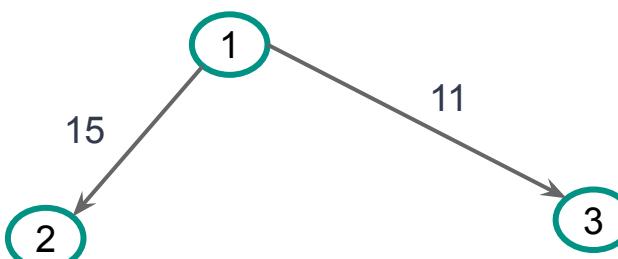
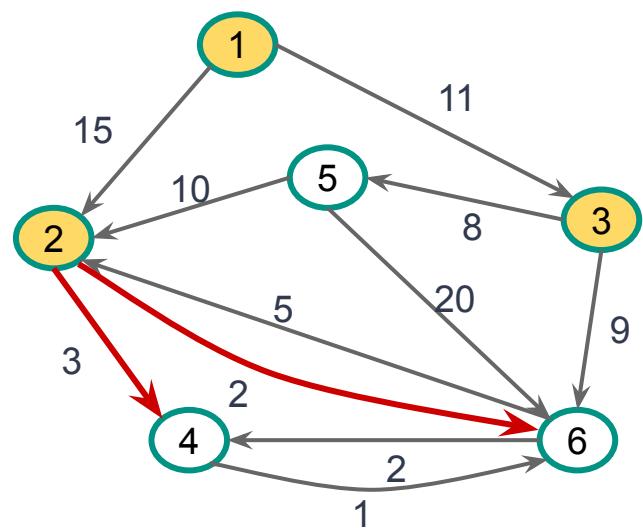
d/tata	1	2	3	4	5	6
Selectăm 1	[ - , $\infty/0$ ]	$15/1$ , $\infty/0$	$11/1$ , $\infty/0$	$\infty/0$ , $\infty/0$	$\infty/0$ , $\infty/0$	$\infty/0$ , $\infty/0$
Selectăm 3	[ - , $15/1$ ]	- , $15/1$	- , $19/3$	- , $20/3$		

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



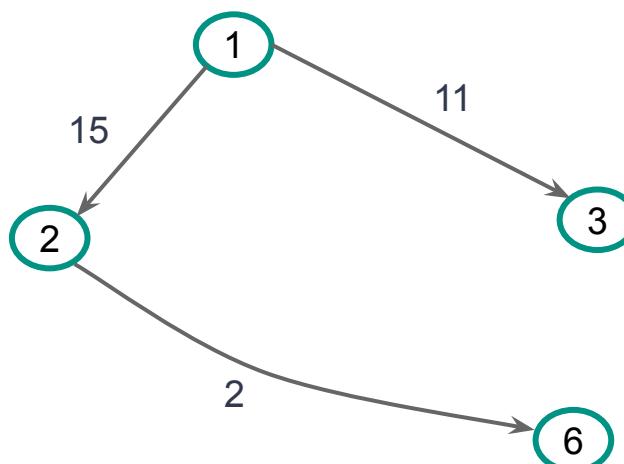
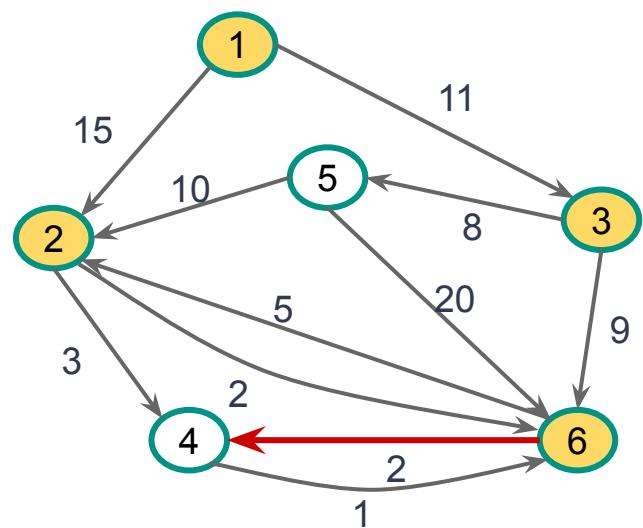
d/tata	1	2	3	4	5	6
Selectám 1	[ - , $\infty/0$ ]	$15/1$	$11/1$	$\infty/0$	$\infty/0$	$\infty/0$
Selectám 3	[ - , $15/1$ ]	-	-	$\infty/0$	$19/3$	$20/3$
Selectám 2						

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



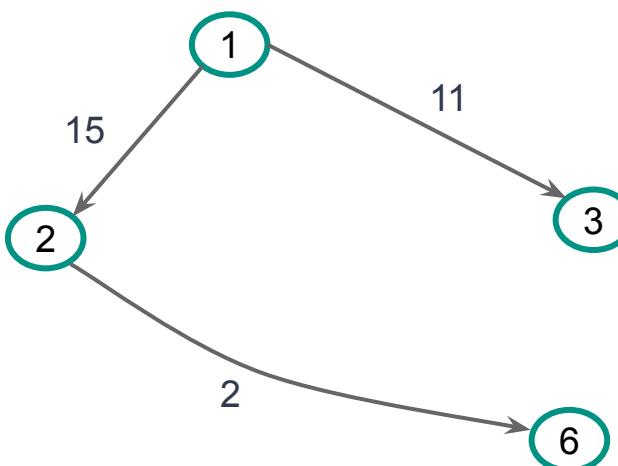
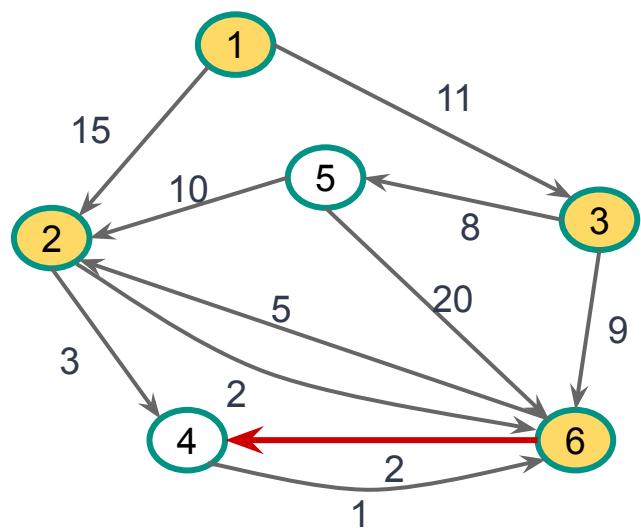
d/tata	1	2	3	4	5	6
Selectám 1	[ - , $\infty/1$ ]	$15/1$	$11/1$	$\infty/0$	$\infty/0$	$\infty/0$
Selectám 3	[ - , $15/1$ ]	-	-	$\infty/0$	$19/3$	$20/3$
Selectám 2	[ - , - ]	-	-	$18/2$	$19/3$	$17/2$

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



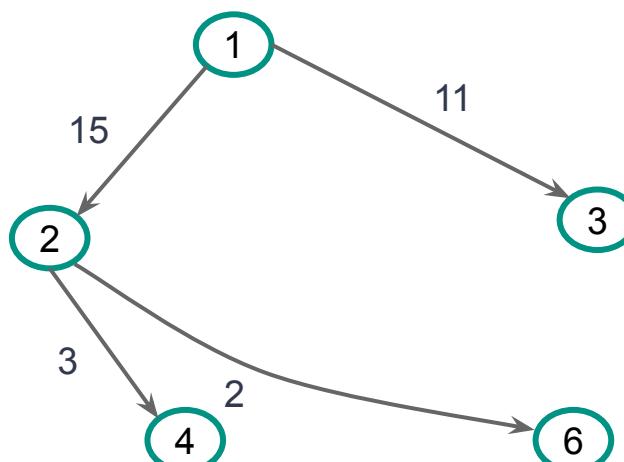
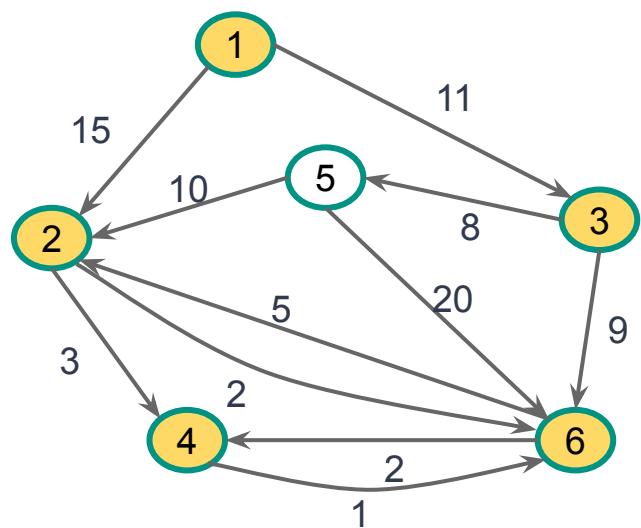
d/tata	1	2	3	4	5	6
Selectám 1	[ - , $\infty/1$ ]	$15/1$	$11/1$	$\infty/0$	$\infty/0$	$\infty/0$
Selectám 3	[ - , $15/1$ ]	-	-	$\infty/0$	$19/3$	$20/3$
Selectám 2	[ - , - ]	-	-	$18/2$	$19/3$	$17/2$
Selectám 6						

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



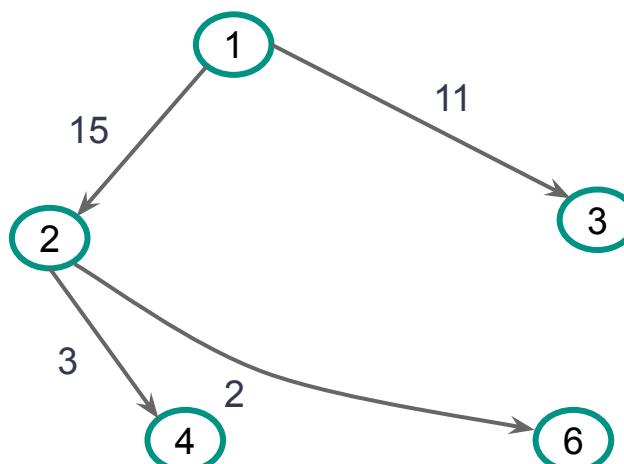
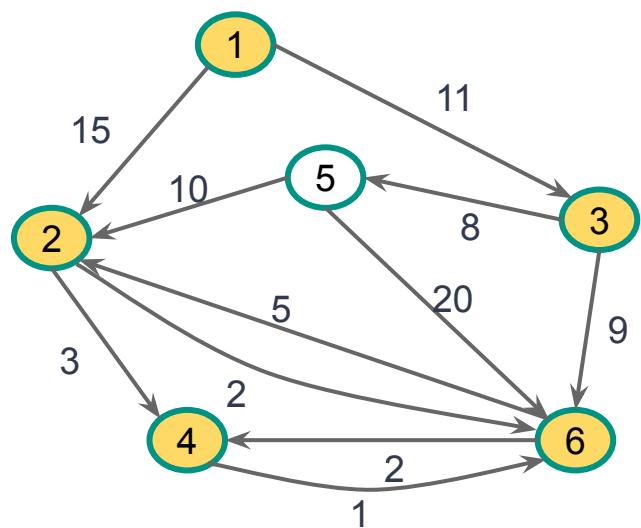
d/tata	1	2	3	4	5	6
Selectám 1	[ - , $\infty/0$ ]	[ $15/1$ , $\infty/0$ ]	[ $11/1$ , $\infty/0$ ]	[ $\infty/0$ , $\infty/0$ ]	[ $\infty/0$ , $\infty/0$ ]	[ $\infty/0$ , $\infty/0$ ]
Selectám 3	[ - , $15/1$ ]	-	-	[ $\infty/0$ , $19/3$ ]	[ $19/3$ , $20/3$ ]	-
Selectám 2	[ - , - ]	-	-	[ $18/2$ , $19/3$ ]	[ $19/3$ , $17/2$ ]	-
Selectám 6	[ - , - ]	-	-	[ $18/2$ , $19/3$ ]	-	-

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



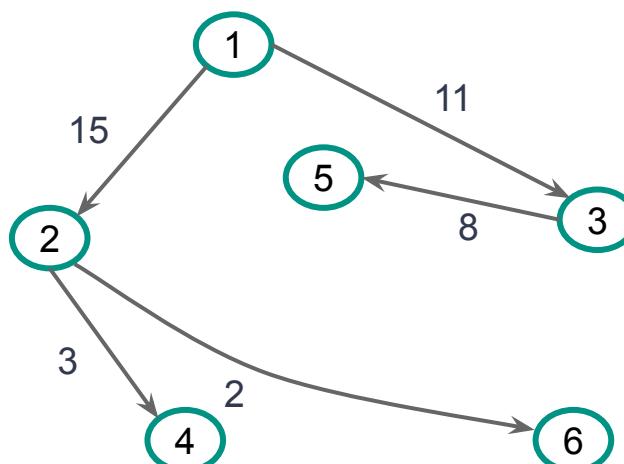
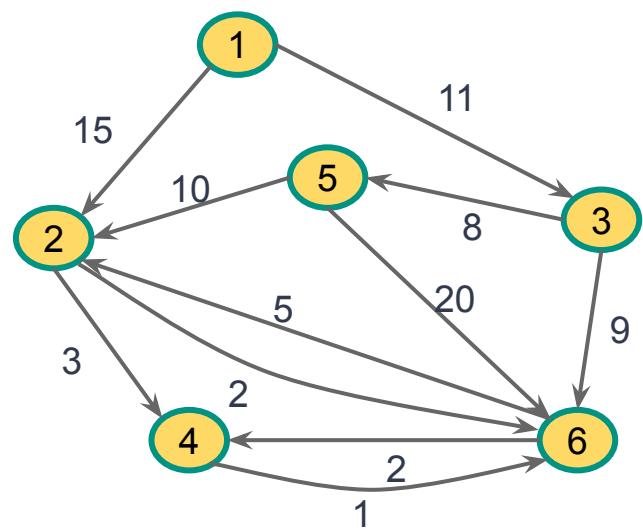
d/tata	1	2	3	4	5	6
Selectám 1	[ 0/0, - , - ]	15/1, - , -	11/1, - , -	∞/0, - , -	∞/0, - , -	∞/0, - , -
Selectám 3	- , - , -	15/1, - , -	- , - , -	∞/0, - , -	19/3, - , -	20/3, - , -
Selectám 2	- , - , -	- , - , -	- , - , -	18/2, - , -	19/3, - , -	17/2, - , -
Selectám 6	- , - , -	- , - , -	- , - , -	18/2, - , -	19/3, - , -	- , - , -
Selectám 4						

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



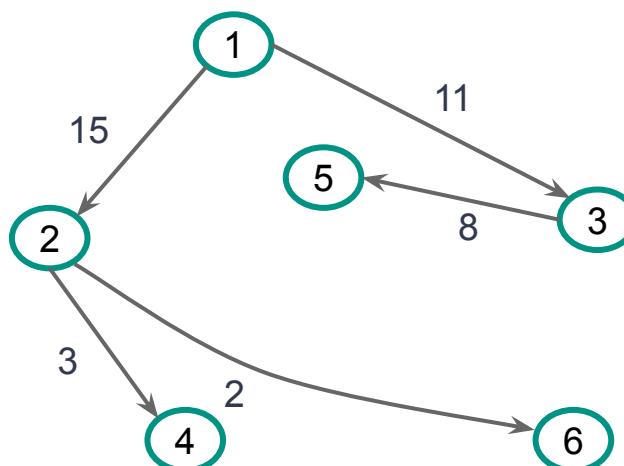
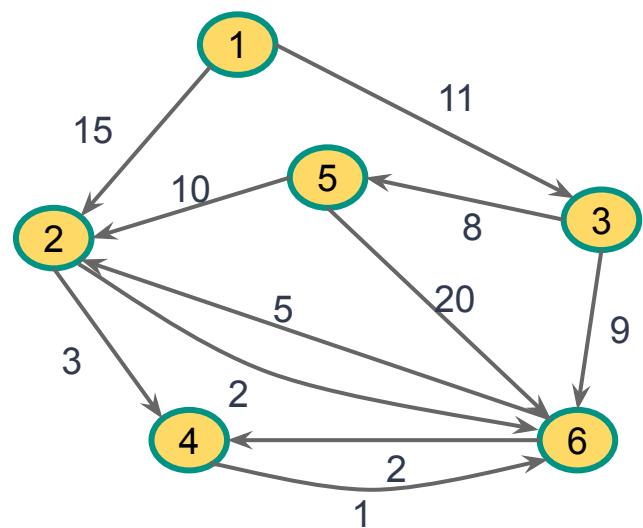
d/tata	1	2	3	4	5	6
Selectám 1	[ 0/0, - , 15/1, - , 11/1, - , - ]					
Selectám 3	[ - , - , 15/1, - , - , 19/3, 20/3 ]					
Selectám 2	[ - , - , - , - , 18/2, 19/3, 17/2 ]					
Selectám 6	[ - , - , - , - , 18/2, 19/3, - ]					
Selectám 4	[ - , - , - , - , - , 19/3, - ]					

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$

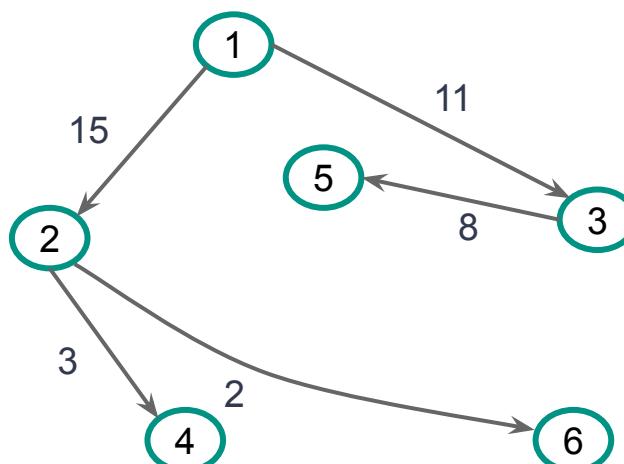
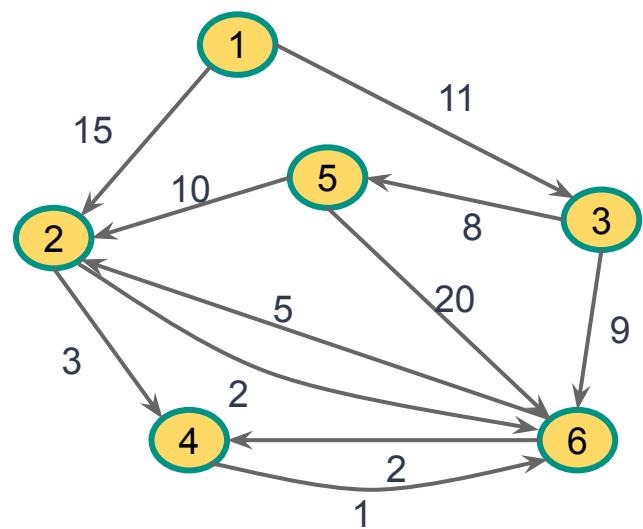


d/tata	1	2	3	4	5	6
Selectám 1	[ 0/0, - , 15/1, - , 11/1, - , ∞/0, ∞/0, ∞/0 ]					
Selectám 3	[ - , - , 15/1, - , - , ∞/0, 19/3, 20/3 ]					
Selectám 2	[ - , - , - , - , 18/2, 19/3, 17/2 ]					
Selectám 6	[ - , - , - , - , 18/2, 19/3, - ]					
Selectám 4	[ - , - , - , - , - , 19/3, - ]					
Selectám 5						

$$d[v] = \min \{ d[v], d[u] + w(u, v) \}$$



d/tata	1	2	3	4	5	6
Selectám 1	[ - , $\infty/0$ ]	$15/1$	$11/1$	$\infty/0$	$\infty/0$	$\infty/0$
Selectám 3	[ - , $15/1$ ]	-	-	$\infty/0$	$19/3$	$20/3$
Selectám 2	[ - , - ]	-	-	$18/2$	$19/3$	$17/2$
Selectám 6	[ - , - ]	-	-	$18/2$	$19/3$	-
Selectám 4	[ - , - ]	-	-	-	$19/3$	-
Selectám 5	[ - , - ]	-	-	-	-	-



d/tata  
**SOLUȚIE**

[ 0/0,      15/1,      11/1,  
   4      18/2,      19/3,      17/2 ]

**Un drum minim de la 1 la 6?**

# Algoritmul lui Dijkstra

## Observații

- **Dacă vârful  $u$  curent are eticheta  $d[u] = \infty$ , algoritmul se poate opri**
- Vectorul **tata** memorează arborele distanțelor față de **s** (vârfurile neaccesibile din s rămân cu tata 0)

# Algoritmul lui Dijkstra - Complexitate

```
Dijkstra(G, w, s)
```

  inițializează mulțimea vârfurilor neselectate Q cu V

  pentru fiecare  $u \in V$  execută

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

  cât timp  $Q \neq \emptyset$  execută

$u = \text{extrage vârf cu eticheta } d \text{ minimă din } Q$

    pentru fiecare  $uv \in E$  execută

      dacă  $d[u] + w(u, v) < d[v]$  atunci

$d[v] = d[u] + w(u, v)$

$tata[v] = u$

scrie  $d$ ,  $tata$

// scrie drum minim de la  $s$  la un vârf  $t$  dat folosind  $tata$

# Algoritmul lui Dijkstra - Complexitate



□ Cum memorăm  $Q = \text{vârfurile încă neselectate}$ ?

# Algoritmul lui Dijkstra - Complexitate

**Q poate fi** (ca și în cazul algoritmului lui Prim)

- **vector**
  - $Q[u] = 1$ , dacă  $u$  este selectat ( $u \notin Q$ )  
0, altfel ( $u \in Q$ )
- **min-ansamblu (heap)**

# Algoritmul lui Dijkstra - Complexitate

**Varianta 1** - reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat } (u \notin Q) \\ 0, & \text{altfel } (u \in Q) \end{cases}$$

- Inițializare Q** →
  - n \* extragere vârf minim** →
  - actualizare etichete vecini** →
-

# Algoritmul lui Dijkstra - Complexitate

**Varianta 1** - reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat } (u \notin Q) \\ 0, & \text{altfel } (u \in Q) \end{cases}$$

- Inițializare Q** →  $O(n)$
  - $n *$  extragere vârf minim** →
  - actualizare etichete vecini** →
-

# Algoritmul lui Dijkstra - Complexitate

**Varianta 1** - reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat } (u \notin Q) \\ 0, & \text{altfel } (u \in Q) \end{cases}$$

- Inițializare Q** →  $O(n)$
  - $n *$  extragere vârf minim** →  $O(n^2)$
  - actualizare etichete vecini** →
-

# Algoritmul lui Dijkstra - Complexitate

**Varianta 1** - reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat } (u \notin Q) \\ 0, & \text{altfel } (u \in Q) \end{cases}$$

- Inițializare Q** →  $O(n)$
  - $n *$  extragere vârf minim** →  $O(n^2)$
  - actualizare etichete vecini** →  $O(m)$
-

# Algoritmul lui Dijkstra - Complexitate

**Varianta 1** - reprezentarea lui Q ca vector

$$Q[u] = \begin{cases} 1, & \text{dacă } u \text{ este selectat } (u \notin Q) \\ 0, & \text{altfel } (u \in Q) \end{cases}$$

- Inițializare Q** →  $O(n)$
- $n *$  extragere vârf minim** →  $O(n^2)$
- actualizare etichete vecini** →  $O(m)$

---

$$O(n^2)$$

# Algoritmul lui Dijkstra - Complexitate

**Varianta 2** - reprezentarea lui Q ca min-heap

- Inițializare Q** →
  - n \* extragere vârf minim** →
  - actualizare etichete vecini** →
-

# Algoritmul lui Dijkstra - Complexitate

```
Dijkstra(G, w, s) // - Q min-heap în raport cu d
    pentru fiecare  $u \in V$  execută
         $d[u] = \infty$ ;  $tata[u] = 0$ 
     $d[s] = 0$ 
    Q = V // creare heap cu cheile din d
    cât timp  $Q \neq \emptyset$  execută
        u = extrage_min(Q)
        pentru fiecare  $uv \in E$  execută
            dacă  $d[u] + w(u, v) < d[v]$  atunci
                 $d[v] = d[u] + w(u, v)$ 
                repara(Q, v)
                tata[v] = u

    scrie d, tata
    // scrie drum minim de la s la un vîrf t dat folosind tata
```

# Algoritmul lui Dijkstra - Complexitate

## Varianta 2 - reprezentarea lui Q ca min-heap

- **Inițializare Q** →  $O(n)$
  - **$n *$  extragere vârf minim** →  $O(n \log n)$
  - **actualizare etichete vecini** →
-

# Algoritmul lui Dijkstra - Complexitate

## Varianta 2 - reprezentarea lui Q ca min-heap

- **Inițializare Q** →  $O(n)$
  - **$n *$  extragere vârf minim** →  $O(n \log n)$
  - **actualizare etichete vecini** →  $O(m \log n)$
- 
- !! + actualizare Q**
- $O(m \log n)$**

# Algoritmul lui Dijkstra

**Observație.** Pentru a determina drumul minim între două vârfuri  $s$  și  $t$  **date**, putem folosi algoritmul lui Dijkstra, cu următoarea modificare:

- dacă vârful  $u$  ales este chiar  $t$ , **algoritmul se oprește**
- drumul de la  $s$  la  $t$  se afișează folosind vectorul  $tata$  (vezi BF)

# Algoritmul lui Dijkstra

- **Dijkstra ≈ Prim** (versiunea  $O(n^2)$  /  $O(m \log n)$ )

# Algoritmul lui Dijkstra



**Algoritmul funcționează și pentru grafuri neorientate?**

# Algoritmul lui Dijkstra



- De ce nu funcționează corect algoritmul dacă avem arce cu cost negativ? Exemplu.
  
- Cum putem rezolva problema dacă avem și arce de cost negativ?

# Algoritmul lui Dijkstra

- De ce nu funcționează corect algoritmul dacă avem arce cu cost negativ? Exemplu.
- Cum putem rezolva problema dacă avem și arce de cost negativ?



Putem aduna o constantă la costul fiecărui arc, astfel încât toate arcele să aibă cost pozitiv. **Drumul minim între 2 vârfuri rămâne la fel?**

# Algoritmul lui Dijkstra

- De ce nu funcționează corect algoritmul dacă avem arce cu cost negativ? Exemplu.
- Cum putem rezolva problema dacă avem și arce de cost negativ?

Putem aduna o constantă la costul fiecărui arc, astfel încât toate arcele să aibă cost pozitiv. Drumul minim între 2 vârfuri rămâne la fel?

- NU



# Algoritmul lui Dijkstra

Cum putem rezolva problema dacă avem și arce de cost negativ?



## Algoritmul BELLMAN-FORD

(suplimentar)

- La un pas, nu relaxăm arcele dintr-un vârf selectat  $u$ , ci **din toate vârfurile** (deci relaxăm toate arcele)

# Algoritmul lui Dijkstra

Cum putem rezolva problema dacă avem și arce de cost negativ?



## Algoritmul BELLMAN-FORD (suplimentar)

- La un pas, nu relaxăm arcele dintr-un vârf selectat  $u$ , ci **din toate vârfurile** (deci relaxăm toate arcele)

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

# Algoritmul lui Dijkstra

Cum putem rezolva problema dacă avem și arce de cost negativ?



## Algoritmul BELLMAN-FORD

- La un pas, nu relaxăm arcele dintr-un vârf selectat u, ci **din toate vârfurile** (deci relaxăm toate arcele)

pentru  $i = 1, n-1$  execută

    pentru fiecare  $uv \in E$  execută

        dacă  $d[u] + w(u, v) < d[v]$  atunci

$d[v] = d[u] + w(u, v)$

            tata[v] = u

- După pasul i  $\rightarrow d[u] =$  costul minim al unui s-u drum cu cel mult i arce

# Algoritmul lui Dijkstra - Corectitudine

# Algoritmul lui Dijkstra - Corectitudine

**Lema 1.** Pentru orice  $u \in V$ , la orice pas al algoritmului lui Dijkstra, avem:

- dacă  $d[u] < \infty$ , există un drum de la  $s$  la  $u$  în  $G$ , de cost  $d[u]$ , iar acesta se poate determina din vectorul tata:
  - $tata[u] = \text{predecesorul lui } u \text{ pe un drum de la } s \text{ la } u \text{ de cost } d[u]$
- $d[u] \geq \delta(s, u)$

# Algoritmul lui Dijkstra - Corectitudine

**Lema 1.** Pentru orice  $u \in V$ , la orice pas al algoritmului lui Dijkstra, avem:

- dacă  $d[u] < \infty$ , există un drum de la  $s$  la  $u$  în  $G$ , de cost  $d[u]$ , iar acesta se poate determina din vectorul tata:
  - $tata[u] = \text{predecesorul lui } u \text{ pe un drum de la } s \text{ la } u \text{ de cost } d[u]$
- $d[u] \geq \delta(s, u)$

**Consecință.** Dacă, la un pas al algoritmului, avem, pentru un vârf  $u$ , relația  $d[u] = \delta(s, u)$ , atunci  $d[u]$  nu se mai modifică până la final.

# Algoritmul lui Dijkstra - Corectitudine

## Teorema

Fie  $G = (V, E, w)$  un graf orientat ponderat cu  
 $w : E \rightarrow \mathbb{R}_+$  și  $s \in V$  fixat.

La finalul algoritmului lui Dijkstra, avem:

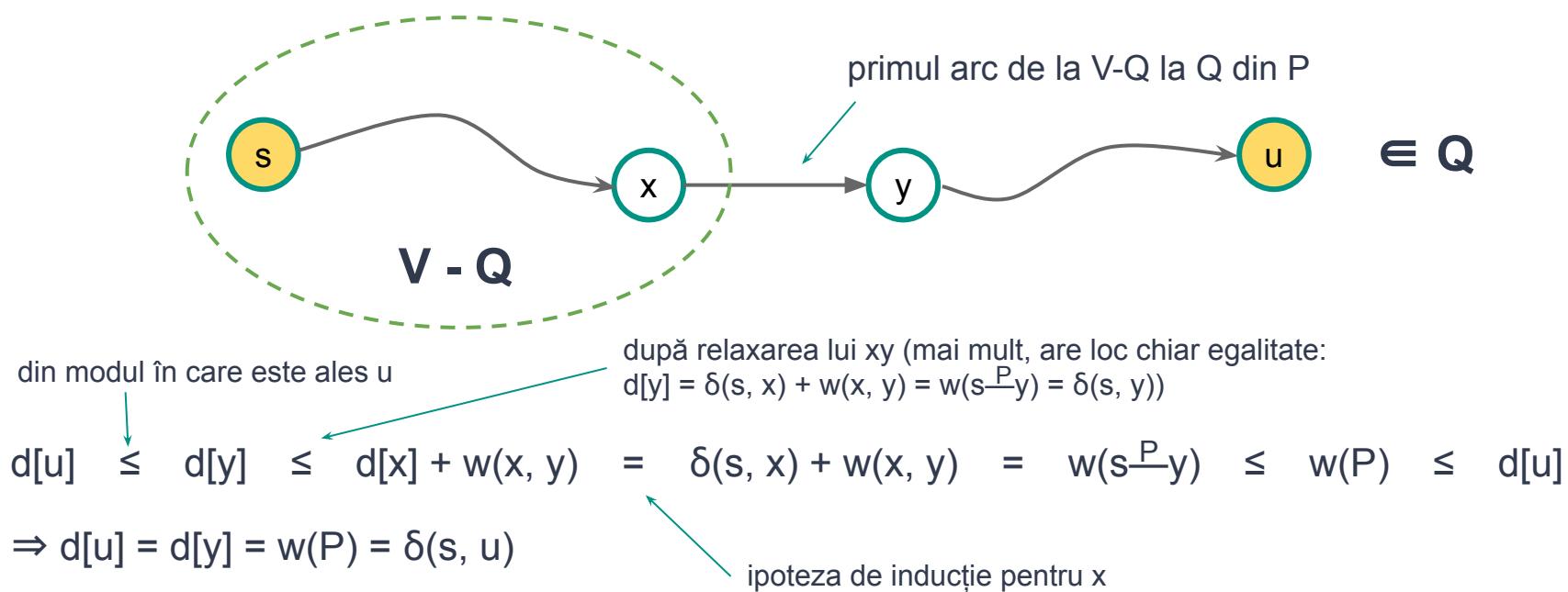
$$d[u] = \delta(s, u), \text{ pentru orice } u \in V$$

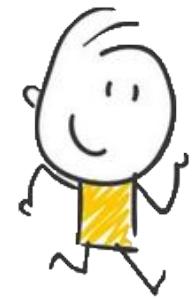
și tata memorează un arbore al distanțelor față de  $s$ .

# Algoritmul lui Dijkstra - Corectitudine

**Demonstrație (idee).** Inducție:  $d[x] = \delta(s, x)$ ,  $\forall x \notin Q$  (= deja selectat)

Când un vârf  $u$  este selectat: fie  $P$  un  $s-u$  drum minim





# Drumuri minime de la un vârf s dat la celealte vârfuri

(de sursă unică)

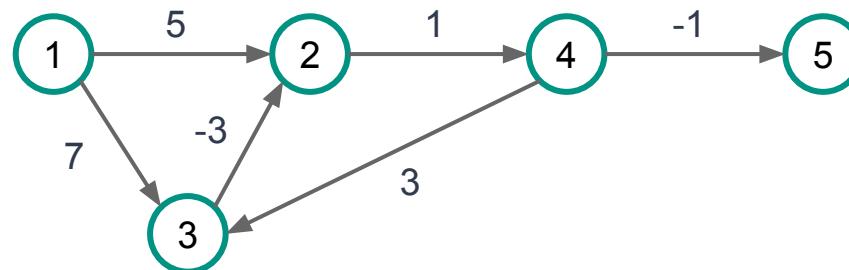
# Algoritmul Bellman-Ford

# Algoritmul Bellman-Ford

Ipoteză:

- Arcele pot avea și cost **negativ**
- Graful **nu** conține circuite de cost negativ
  - (dacă există  $\Rightarrow$  algoritmul le va detecta  $\Rightarrow$  nu are soluție)

# Algoritmul Bellman-Ford



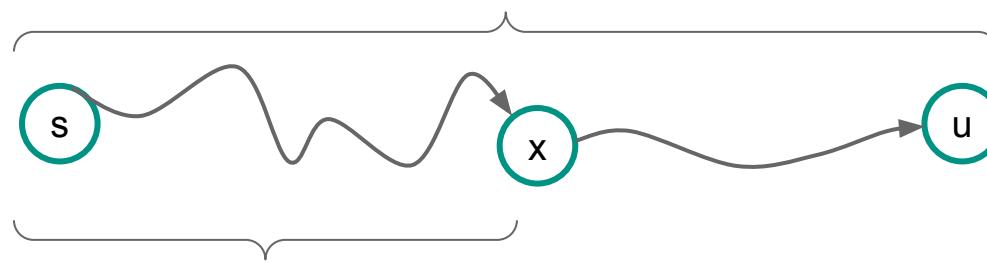
Algoritmul lui Dijkstra - doar pentru ponderi nenegative

# Algoritmul Bellman-Ford

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

De câte ori?

s-y drum minim cu  $\leq k$  arce

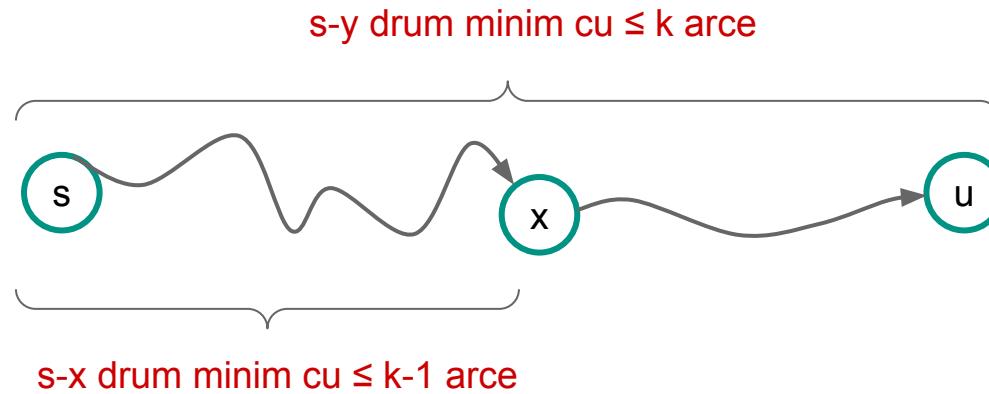


s-x drum minim cu  $\leq k-1$  arce

# Algoritm Bellman-Ford

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

De câte ori?



Dacă  $P$  este  $s-y$  drum cu  $\leq k$  arce  $\Rightarrow [s \xrightarrow{P} x]$  este  $s-x$  drum minim cu  $\leq k-1$  arce

**Un drum minim are cel mult  $n-1$  arce  $\Rightarrow n-1$  etape**

# Algoritmul Bellman-Ford

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

Nu relaxăm arcele dintr-un vârf selectat  $u$ , ci **din toate vârfurile**.

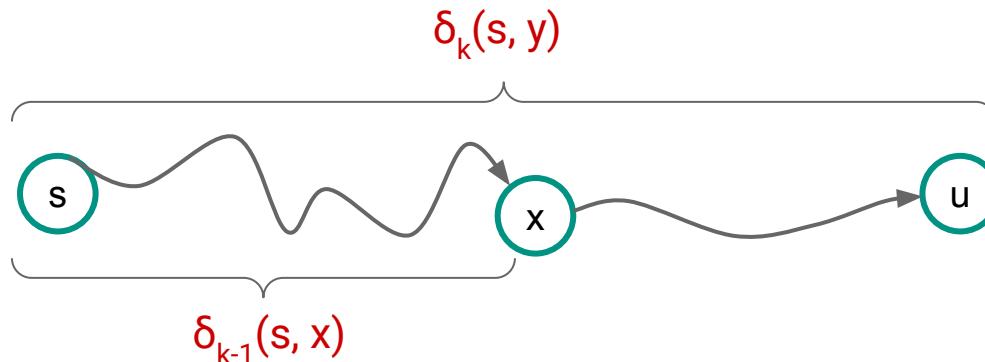
- **n-1 etape** - după  $k$  etape,  $d[u] \leq$  costul minim al unui drum de la  $s$  la  $u$  cu cel mult  $k$  arce (programare dinamică)

⇒ după  $k$  etape, sunt corect calculate etichetele  $d[u]$  pentru arcele vârfuri  $u$  pentru care există un  $s-u$  drum minim cu cel mult  $k$  arce

# Algoritmul Bellman-Ford

Notăm

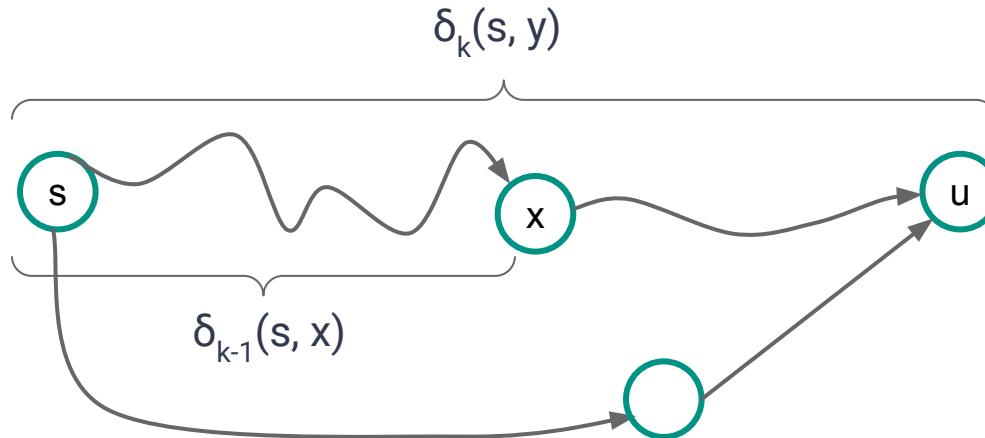
$\delta_k(s, x) = \text{costul minim al unui } s-x \text{ drum cu cel mult } k \text{ arce}$



# Algoritmul Bellman-Ford

Notăm

$\delta_k(s, x) = \text{costul minim al unui } s-x \text{ drum cu cel mult } k \text{ arce}$



Avem:

$$\delta_k(s, y) = \min\{ \delta_{k-1}(s, y), \min\{ \delta_{k-1}(s, x) + w(xy) \mid xy \in E \} \}$$

# Algoritmul Bellman-Ford

**pentru fiecare  $u \in V$  execută**

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

**pentru  $i = 1, n-1$  execută**

**pentru fiecare  $uv \in E$  execută**

**dacă  $d[u] + w(u, v) < d[v]$  atunci**

$d[v] = d[u] + w(u, v)$

$tata[v] = u$

# Algoritmul Bellman-Ford

**pentru fiecare  $u \in V$  execută**

$d[u] = \infty$ ;  $tata[u] = 0$

$d[s] = 0$

**pentru  $i = 1, n-1$  execută**

**pentru fiecare  $uv \in E$  execută**

**dacă  $d[u] + w(u, v) < d[v]$  atunci**

$d[v] = d[u] + w(u, v)$

$tata[v] = u$

**Complexitate:**  $O(nm)$

# Algoritmul Bellman-Ford

## Optimizări

La un pas, este suficient să relaxăm arcele din vârfuri ale căror etichetă s-a modificat anterior

# Algoritmul Bellman-Ford

## Optimizări

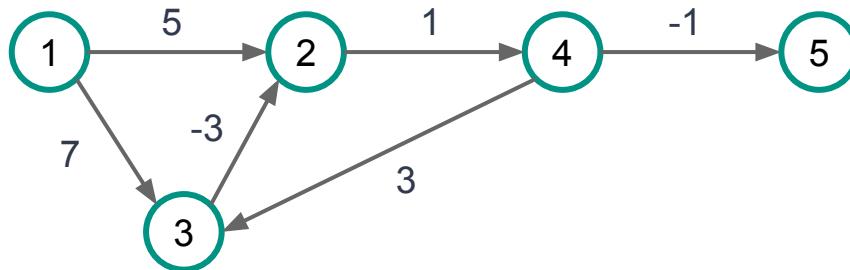
La un pas, este suficient să relaxăm arcele din vârfuri ale căror etichetă s-a modificat anterior

⇒ coadă cu vârfurile ale căror etichetă s-a actualizat

(Detalii - laborator)

# Algoritmul Bellman-Ford

**Etapa 1**



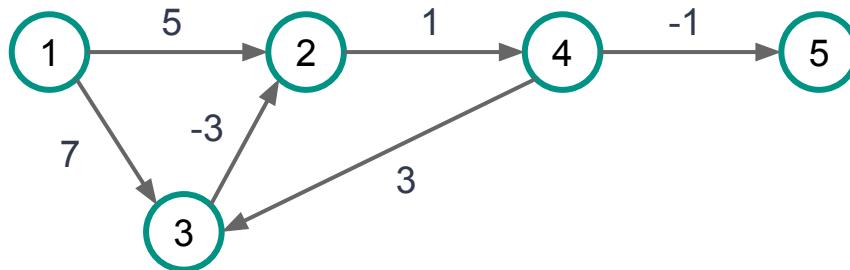
Relaxăm

1 2  
1 3

	1	2	3	4	5
<b>d</b>	0	5	7	$\infty$	$\infty$

# Algoritmul Bellman-Ford

**Etapa 1**



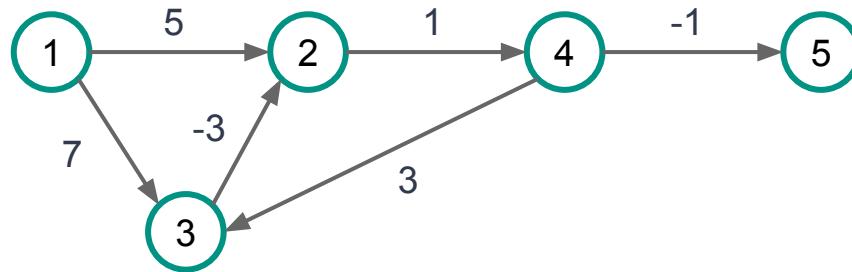
Relaxăm

1 2  
1 3  
2 4

	1	2	3	4	5
<b>d</b>	0	5	7	6	$\infty$

# Algoritmul Bellman-Ford

**Etapa 1**



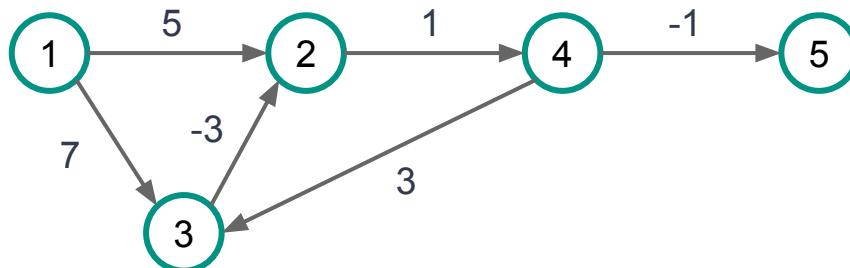
**Relaxăm**

1 2  
1 3  
2 4  
4 3  
4 5

	1	2	3	4	5
<b>d</b>	0	5	7	6	<b>5</b>

# Algoritmul Bellman-Ford

**Etapa 1**



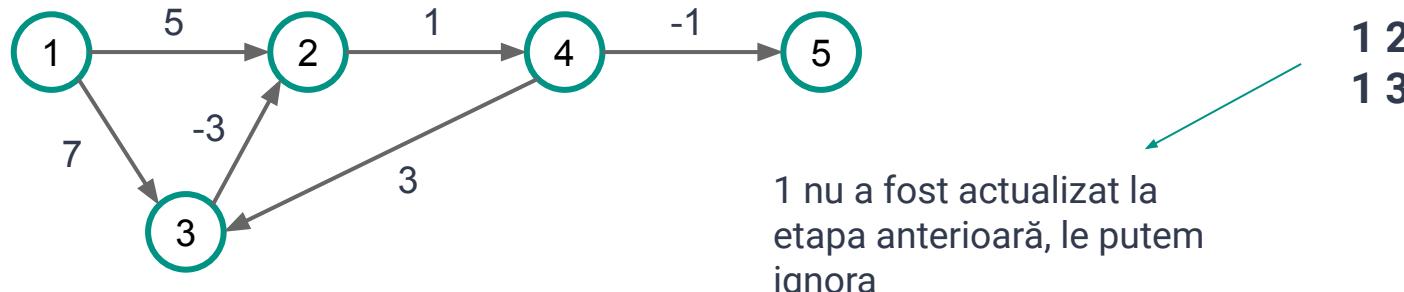
Relaxăm

1 2  
1 3  
2 4  
4 3  
4 5  
3 2

	1	2	3	4	5
d	0	4	7	6	5

# Algoritmul Bellman-Ford

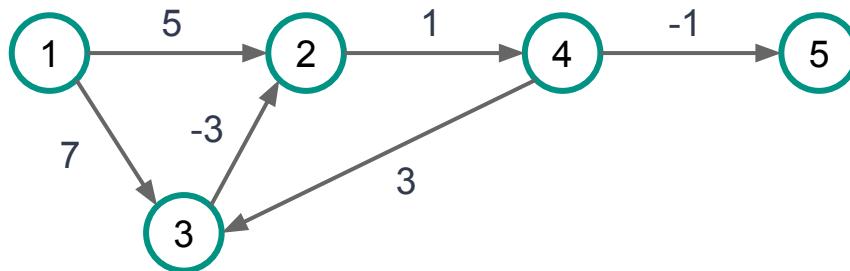
## Etapa 2



	1	2	3	4	5
d	0	4	7	6	5

# Algoritmul Bellman-Ford

**Etapa 2**



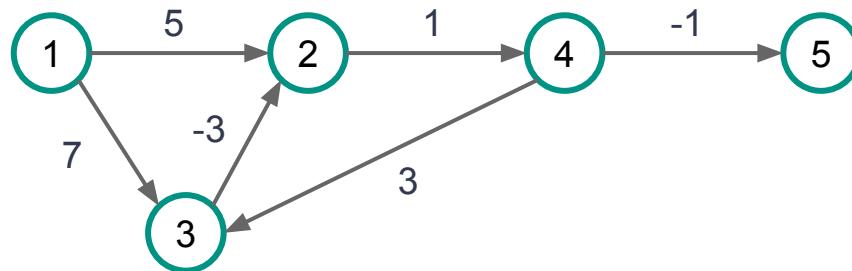
Relaxăm

1 2  
1 3  
2 4

	1	2	3	4	5
<b>d</b>	0	4	7	<b>5</b>	5

# Algoritmul Bellman-Ford

**Etapa 2**



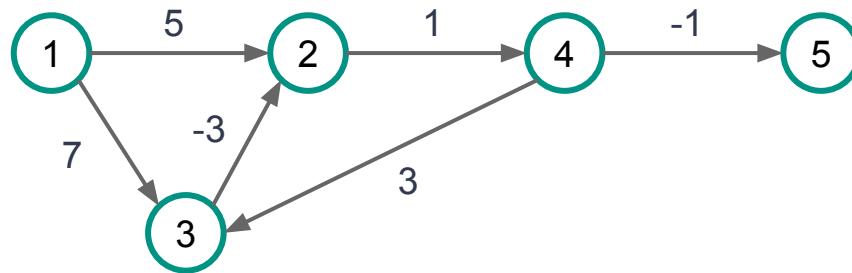
**Relaxăm**

1 2  
1 3  
2 4  
4 3

	1	2	3	4	5
<b>d</b>	0	4	7	5	5

# Algoritmul Bellman-Ford

**Etapa 2**



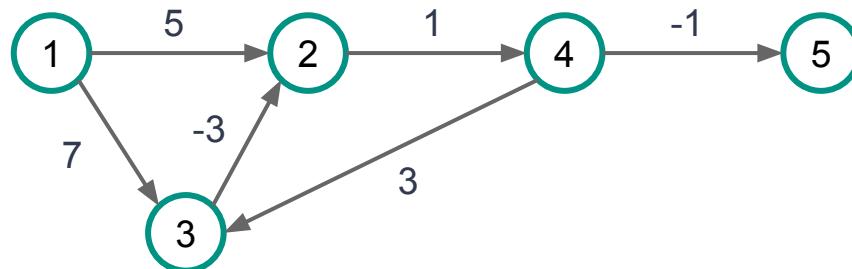
Relaxăm

1 2  
1 3  
2 4  
4 3  
4 5

	1	2	3	4	5
d	0	4	7	5	4

# Algoritmul Bellman-Ford

**Etapa 2**



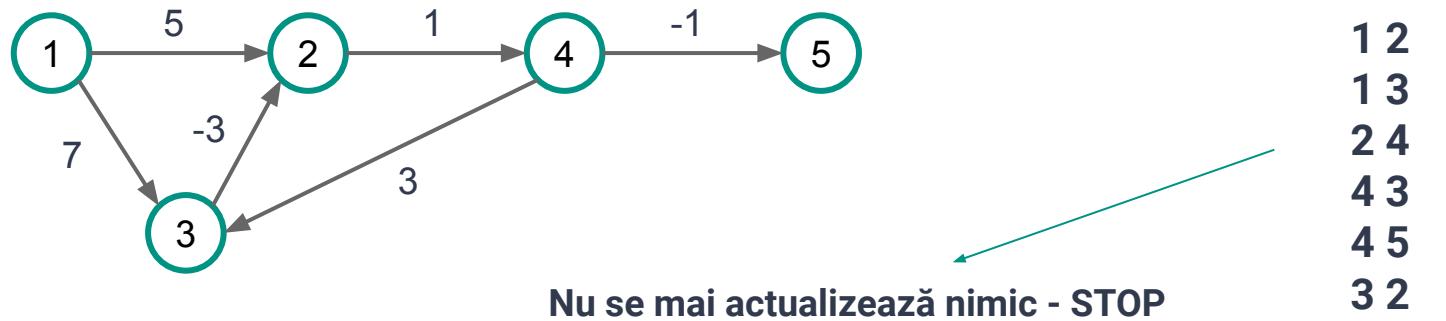
**Relaxăm**

1 2  
1 3  
2 4  
4 3  
4 5  
3 2

	1	2	3	4	5
d	0	4	7	5	4

# Algoritmul Bellman-Ford

**Etapa 3**



	1	2	3	4	5
d	0	4	7	5	4

# Corectitudinea algoritmului Bellman-Ford

# Corectitudine

**Lema 1 (comună).**

Pentru orice  $u \in V$ , la orice pas al algoritmului, avem:

- dacă  $d[u] < \infty$ , există un drum de la s la u în G de cost  $d[u]$  și acesta se poate determina din vectorul tata
  - $tata[u] = \text{predecesorul lui } u \text{ pe un drum de la } s \text{ la } u \text{ de cost } d[u]$
- $d[u] \geq \delta(s, u)$

# Corectitudine

## Lema 1 (comună) - Demonstrație

Inducție după numărul de etape (o etapă = relaxarea tuturor muchiilor)

După k iterări

$$d[x] \leq \delta_k(s, x)$$

= costul minim al unui s-x drum cu cel mult k muchii

# Corectitudine

$$d[x] \leq \delta_k(s, x)$$

= costul minim al unui s-x drum cu cel mult k muchii

$$k = 0: d[s] = 0 = w([s])$$

$k-1 \Rightarrow k$ : Presupunem că, înainte de iteratăia k:

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

Eticheta unui vârf y la iteratăia k se actualizează astfel:

# Corectitudine

$$d[x] \leq \delta_k(s, x)$$

= costul minim al unui s-x drum cu cel mult k muchii

$$k = 0: d[s] = 0 = w([s])$$

$k-1 \Rightarrow k$ : Presupunem că, înainte de iteratăia k:

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

ipoteza de inducție

Eticheta unui vârf y la iteratăia k se actualizează astfel:

$$d[y] \leq \min\{ d[y], \min\{ d[x]+w(x,y) \mid xy \in E \} \}$$

# Corectitudine

$$d[x] \leq \delta_k(s, x)$$

= costul minim al unui s-x drum cu cel mult k muchii

$$k = 0: d[s] = 0 = w([s])$$

$k-1 \Rightarrow k$ : Presupunem că, înainte de iteratăia k:

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

ipoteza de inducție

Eticheta unui vârf y la iteratăia k se actualizează astfel:

$$d[y] \leq \min\{ d[y], \min\{ d[x]+w(x,y) \mid xy \in E \} \}$$



$$\leq \min\{ \delta_{k-1}(s, y), \min\{ \delta_{k-1}(s, x)+w(x,y) \mid xy \in E \} \}$$

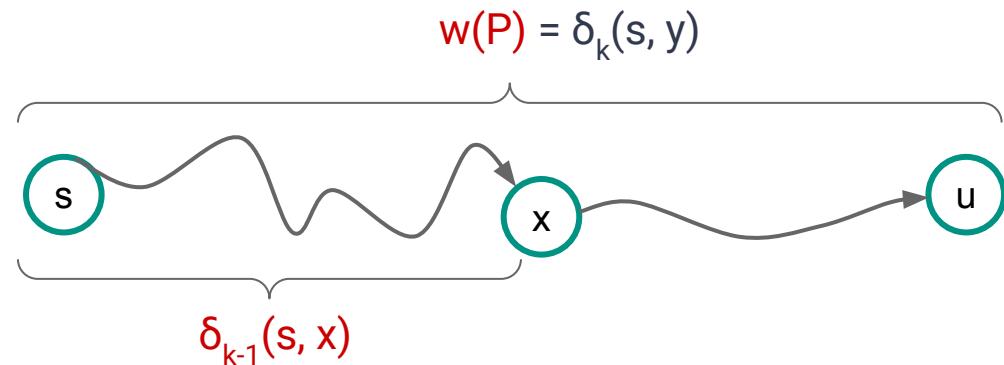
$$= \delta_k(s, y)$$

# Corectitudine

$k-1 \Rightarrow k$ : Presupunem că, înainte de iterăția  $k$ :

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

**Varianta 2:** Fie un  $s-y$  drum cu cost minim printre cele cu cel mult  $k$  arce (  $w(P) = \delta_k(s, y)$  )



# Corectitudine

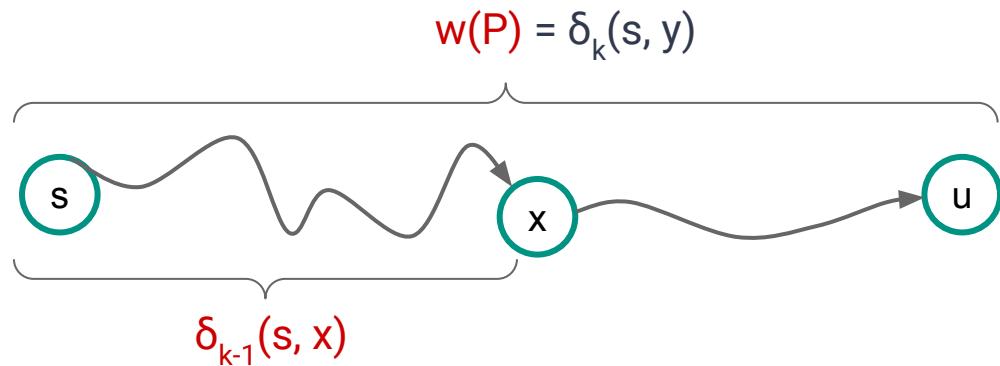
$k-1 \Rightarrow k$ : Presupunem că, înainte de iterăția  $k$ :

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

**Varianta 2:** Fie un  $s-y$  drum cu cost minim printre cele cu cel mult  $k$  arce (  $w(P) = \delta_k(s, y)$  )

$\Rightarrow [s \xrightarrow{P} x]$  este un  $s-x$  drum cu cost minim  
printre cele cu cel mult  $k-1$  arce, deci  
are cost  $\delta_{k-1}(s, x)$

$\Rightarrow d[x] \leq \delta_{k-1}(s, x)$



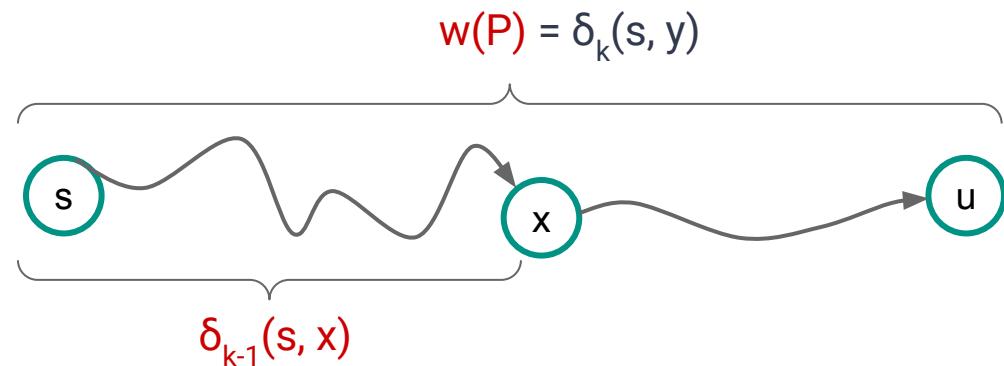
# Corectitudine

După relaxarea arcului  $xy$ :

$$d[y] \leq d[x] + w(xy)$$

$$\leq \delta_{k-1}(s, x) + w(xy)$$

$$= w([s \xrightarrow{P} x]) + w(xy) = w(P) = \delta_k(s, y)$$



# Detectarea de circuite negative

# Detectarea de circuite negative

Există circuit negativ în G

↔ dacă algoritmul ar mai face o iteratie, s-ar mai actualiza etichete de distanță

↔ după  $n-1$  iteratii, există un arc  $uv$  cu

$$d[v] > d[u] + w(uv)$$

# Detectarea de circuite negative

## Demonstrație

Arătăm că:

nu există cicluri negative  $\Leftrightarrow$  nu se mai fac actualizări la pasul n

# Detectarea de circuite negative

## Demonstrație

Arătăm că:

nu există cicluri negative  $\Leftrightarrow$  nu se mai fac actualizări la pasul n

- Dacă nu există cicluri negative  $\Rightarrow$  nu se mai fac actualizări la pasul n (din corectitudine)

# Detectarea de circuite negative

## Demonstrație

Arătăm că:

nu există cicluri negative  $\Leftrightarrow$  nu se mai fac actualizări la pasul n

- Dacă nu există cicluri negative  $\Rightarrow$  nu se mai fac actualizări la pasul n (din corectitudine)
- Dacă nu se mai fac actualizări la pasul n, pentru orice ciclu  $C = [v_0, \dots, v_p, v_0] \Rightarrow d[v_i] \leq w(v_i v_{i+1})$

Însumând pe ciclu:

# Detectarea de circuite negative

## Demonstrație

Arătăm că:

nu există cicluri negative  $\Leftrightarrow$  nu se mai fac actualizări la pasul n

- Dacă nu există cicluri negative  $\Rightarrow$  nu se mai fac actualizări la pasul n (din corectitudine)
- Dacă nu se mai fac actualizări la pasul n, pentru orice ciclu  $C = [v_0, \dots, v_p, v_0] \Rightarrow d[v_i] \leq w(v_i v_{i+1})$

Însumând pe ciclu:

$$d[v_0] + \dots + d[v_p] \leq d[v_0] + \dots + d[v_p] + w(v_0 v_1) + \dots + w(v_p v_0)$$

$$\Rightarrow 0 \leq w(v_0 v_1) + \dots + w(v_p v_0) = w(C)$$

# Detectarea de circuite negative

Afișarea ciclului negativ detectat - folosind tata:

# Detectarea de circuite negative

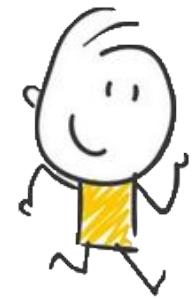
Afișarea ciclului negativ detectat - folosind tata:

- Fie  $v$  un vârf al cărei etichetă s-a actualizat la pasul  $k$

# Detectarea de circuite negative

Afișarea ciclului negativ detectat - folosind tata:

- Fie  $v$  un vârf al cărei etichetă s-a actualizat la pasul  $k$
- Facem  $n$  pași înapoi din  $v$ , folosind vectorul tata (către  $s$ ); fie  $x$  vârful în care am ajuns
- Afișăm ciclul care conține pe  $x$ , folosind tata (din  $x$  până ajungem iar în  $x$ )



# Drumuri minime de sursă unică în grafuri aciclice



# Drumuri minime de sursă unică în grafuri aciclice

## Ipoteze:

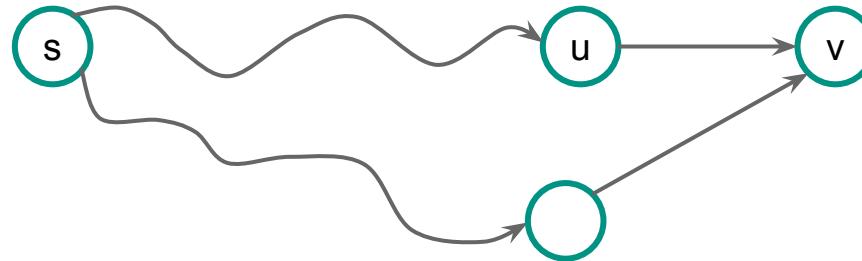
- Graful **nu** conține circuite
- Arcele pot avea **și cost negativ**

# Drumuri minime de sursă unică în grafuri aciclice

**Amintim:**

- Când considerăm un vârf  $v$ , pentru a calcula  $d(s, v)$  ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$
- atunci, putem calcula distanțele după relația

$$\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \mid uv \in E \}$$



# Drumuri minime de sursă unică în grafuri aciclice

## Amintim:

- Când considerăm un vârf  $v$ , pentru a calcula  $d(s, v)$  ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$

$\Rightarrow$

**ar fi utilă o ordonare a vârfurilor, astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$**

# Drumuri minime de sursă unică în grafuri aciclice

## Amintim:

- Când considerăm un vârf  $v$ , pentru a calcula  $d(s, v)$  ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$

⇒

**ar fi utilă o ordonare a vârfurilor, astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$**



**O astfel de ordonare nu există dacă graful conține circuite**

# Drumuri minime de sursă unică în grafuri aciclice

## Amintim:

- Când considerăm un vârf  $v$ , pentru a calcula  $d(s, v)$  ar fi util să știm deja  $\delta(s, u)$ , pentru orice  $u$  cu  $uv \in E$

⇒

ar fi utilă o ordonare a vârfurilor, astfel încât, dacă  $uv \in E$ , atunci  $u$  se află înaintea lui  $v$



O astfel de ordonare există dacă graful nu conține circuite  
= sortarea topologică

# Pseudocod



# Drumuri minime de sursă unică în grafuri aciclice

- Considerăm vârfurile în ordinea dată de sortarea topologică
- Pentru fiecare vârf  $u$ , relaxăm arcele  $uv$  către vecinii săi (pentru a găsi drumuri noi către aceştia)

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

// inițializăm distanțe - ca la Dijkstra

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

// determinăm o sortare topologică a vârfurilor
// este suficient să păstrăm vârfurile din sortare începând cu s
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

// determinăm o sortare topologică a vârfurilor
// este suficient să păstrăm vârfurile din sortare începând cu s
SortTop = sortare_topologică( $G$ , s)

pentru fiecare  $u \in SortTop$  execută
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

// determinăm o sortare topologică a vârfurilor
// este suficient să păstrăm vârfurile din sortare începând cu s
SortTop = sortare_topologică( $G$ , s)

pentru fiecare  $u \in \text{SortTop}$  execută
    pentru fiecare  $uv \in E$  execută
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

// determinăm o sortare topologică a vârfurilor
// este suficient să păstrăm vârfurile din sortare începând cu s
SortTop = sortare_topologică(G, s)

pentru fiecare  $u \in \text{SortTop}$  execută
    pentru fiecare  $uv \in E$  execută
        dacă  $d[u] + w(u, v) < d[v]$  atunci // relaxăm uv
             $d[v] = d[u] + w(u, v)$ 
             $tata[v] = u$ 
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

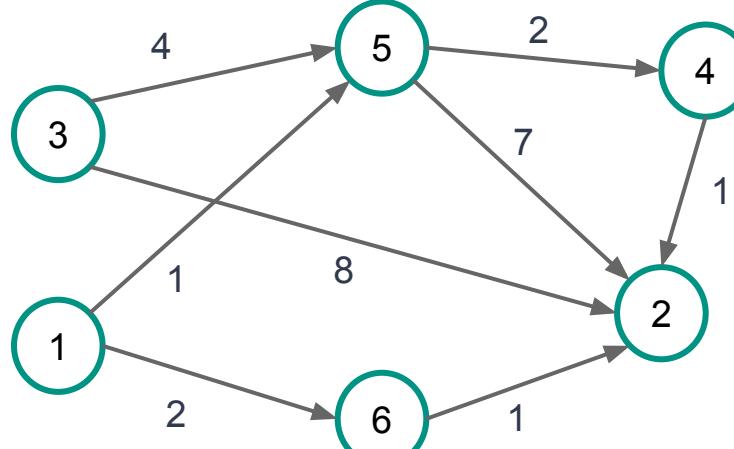
```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

// determinăm o sortare topologică a vârfurilor
// este suficient să păstrăm vârfurile din sortare începând cu s
SortTop = sortare_topologică(G, s)

pentru fiecare  $u \in \text{SortTop}$  execută
    pentru fiecare  $uv \in E$  execută
        dacă  $d[u] + w(u, v) < d[v]$  atunci // relaxăm uv
             $d[v] = d[u] + w(u, v)$ 
             $tata[v] = u$ 

scrie d, tata
```

# Exemplu

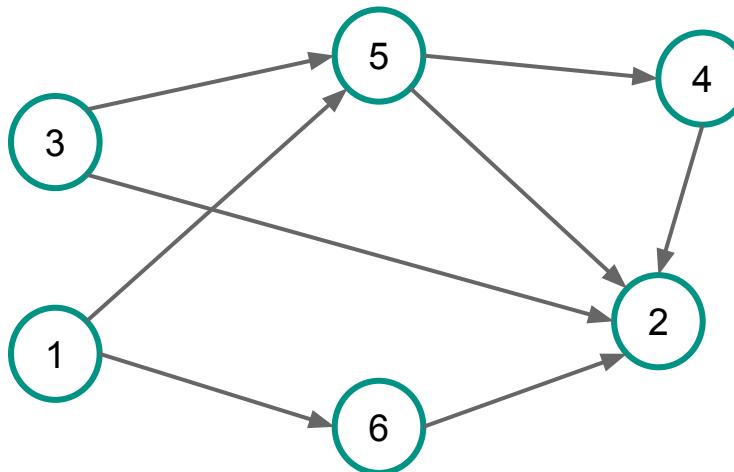


# Exemplu

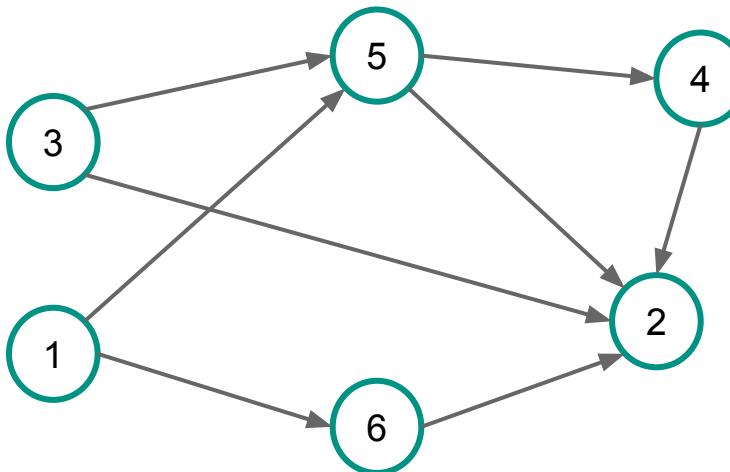
- **Etapa 1** - determinăm o ordonare topologică a vârfurilor
- Amintim algoritm

```
SortTop ← ∅  
coada C ← ∅  
adaugă în C toate vârfurile v cu  $d^-[v] = 0$   
cât timp  $C \neq \emptyset$  execută  
     $i \leftarrow \text{extrage}(C)$   
    adaugă  $i$  în SortTop  
pentru  $ij \in E$  execută  
         $d^-[j]--$   
        dacă  $d^-[j] = 0$  atunci  
            adaugă( $j$ , C)  
  
returnează SortTop
```

# Sortare topologică - Exemplu

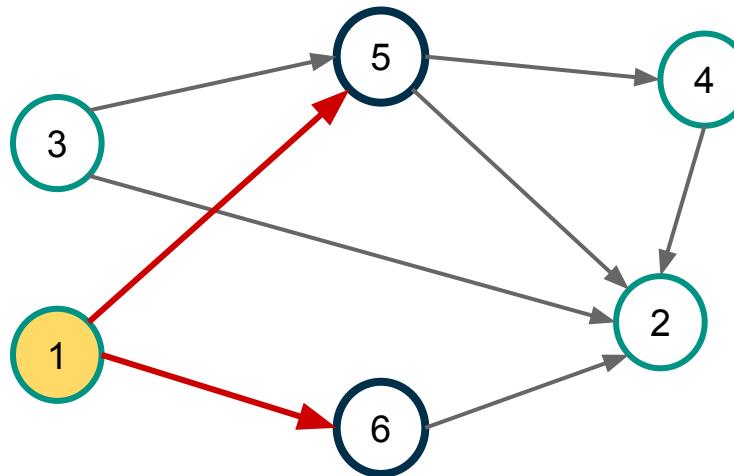


# Sortare topologică - Exemplu



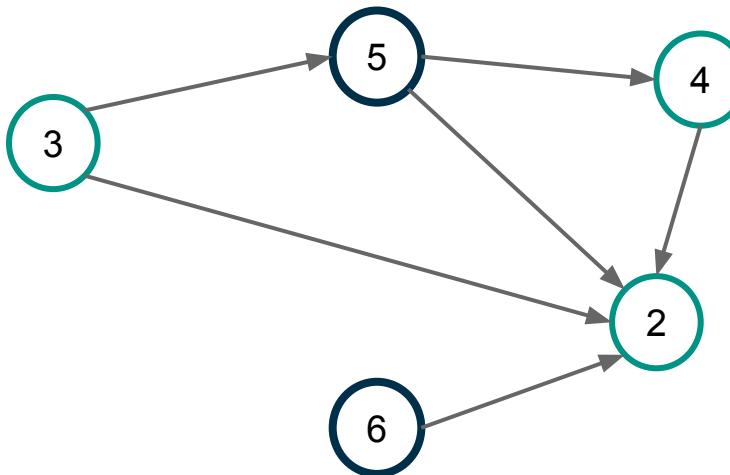
C: 1 3

# Sortare topologică - Exemplu



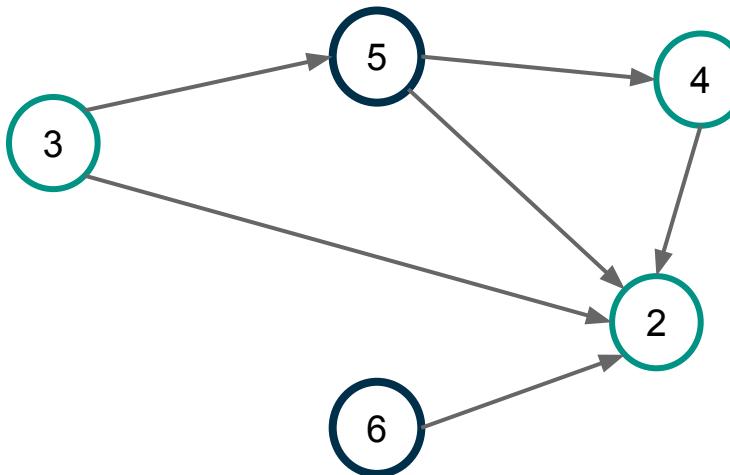
C: **1** 3

# Sortare topologică - Exemplu



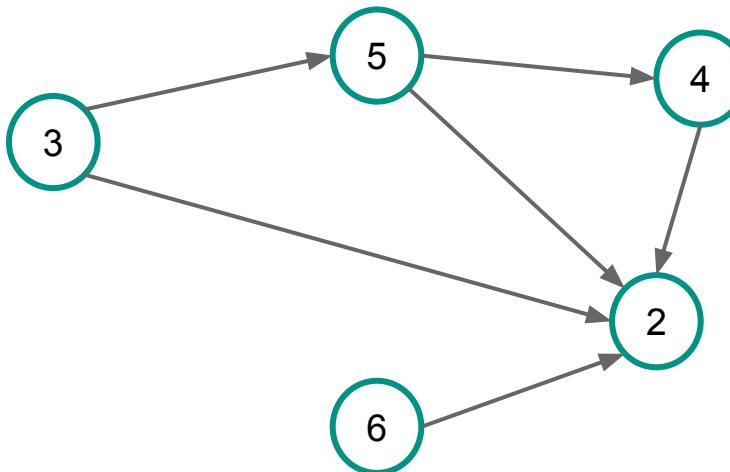
C: **1** 3

# Sortare topologică - Exemplu



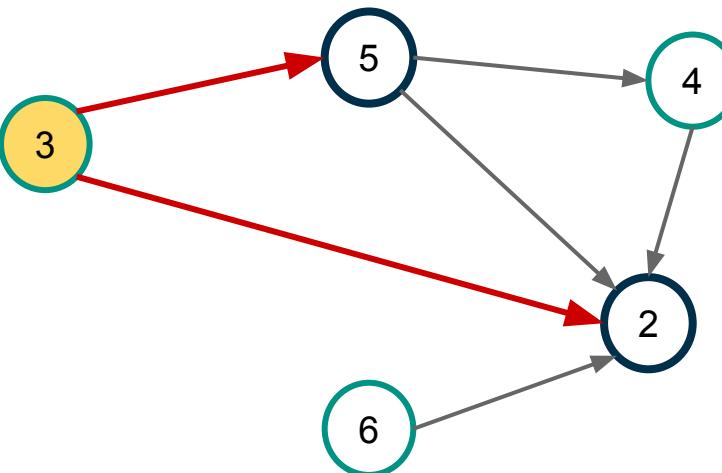
C: **1** 3 6

# Sortare topologică - Exemplu



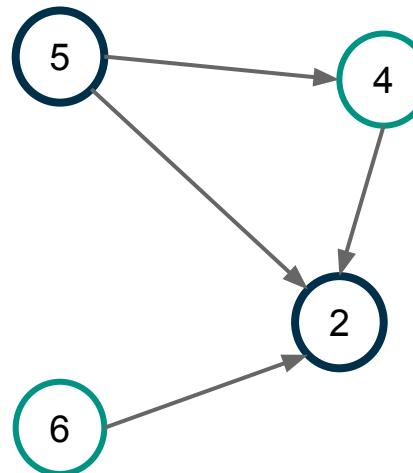
C: **1** 3 6

# Sortare topologică - Exemplu



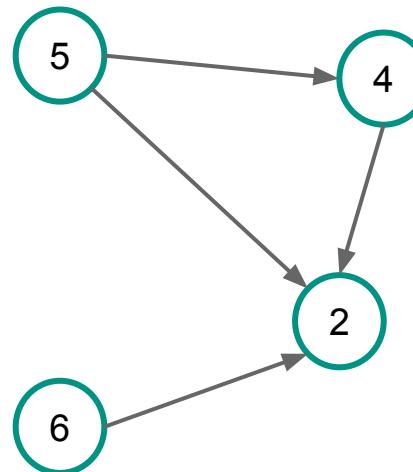
C: **1 3 6**

# Sortare topologică - Exemplu



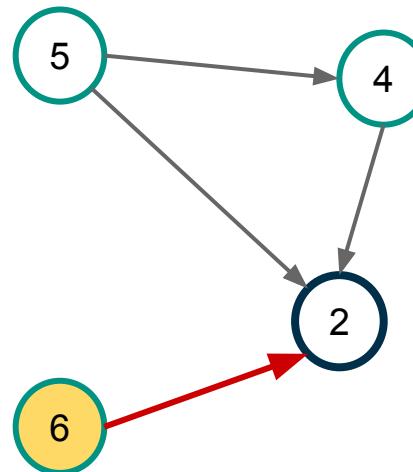
C: **1 3 6**

# Sortare topologică - Exemplu



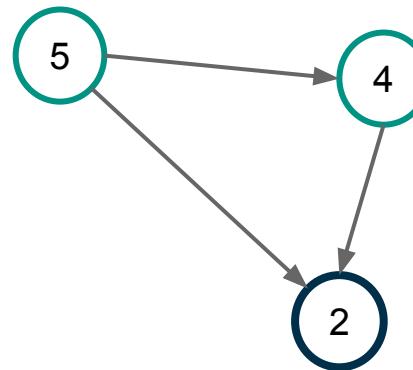
C: **1 3 6 5**

# Sortare topologică - Exemplu



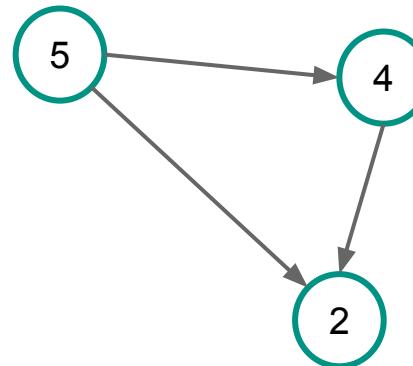
C: **1 3 6 5**

# Sortare topologică - Exemplu



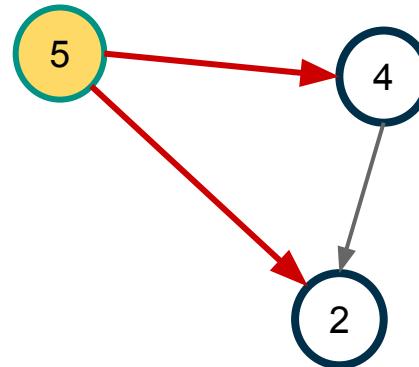
C: **1 3 6 5**

# Sortare topologică - Exemplu



c: **1 3 6 5**

# Sortare topologică - Exemplu



c: **1 3 6 5**

# Sortare topologică - Exemplu



c: **1 3 6 5**

# Sortare topologică - Exemplu



C: 1 3 6 5 4

# Sortare topologică - Exemplu



c: 1 3 6 5 4

# Sortare topologică - Exemplu

2

c: 1 3 6 5 4

# Sortare topologică - Exemplu

2

C: 1 3 6 5 4 2

# Sortare topologică - Exemplu

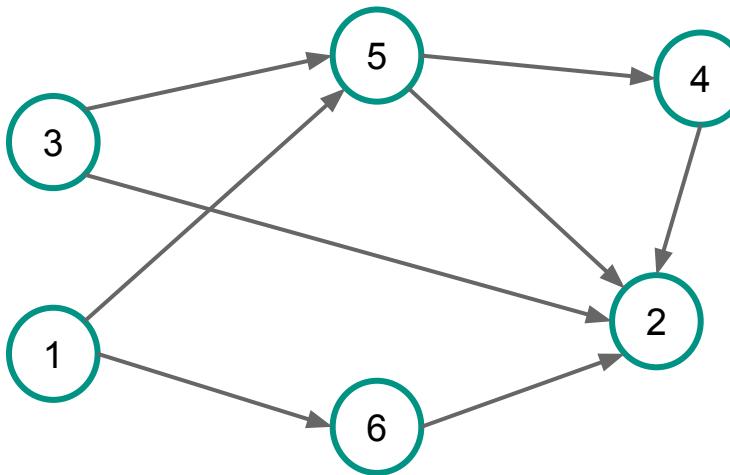


c: 1 3 6 5 4 2

# Sortare topologică - Exemplu

C: 1 3 6 5 4 2

# Sortare topologică - Exemplu



SORTARE TOPOLOGICĂ: 1 3 6 5 4 2

# Sortare topologică - Algoritm

coada C  $\leftarrow \emptyset$

adăugă în C toate vârfurile v cu  $d^-[v]=0$

cât timp  $C \neq \emptyset$  execută

i  $\leftarrow$  extrage(C)

adăugă i în sortare

pentru  $ij \in E$  execută

$d^-[j]--$

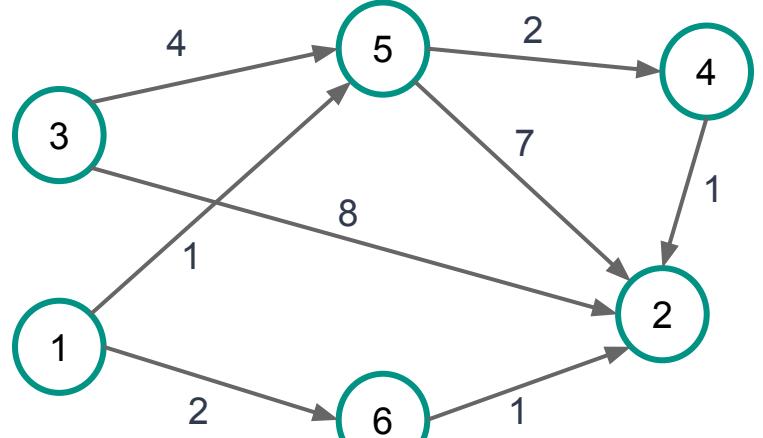
dacă  $d^-[j] = 0$  atunci

adăugă(j, C)

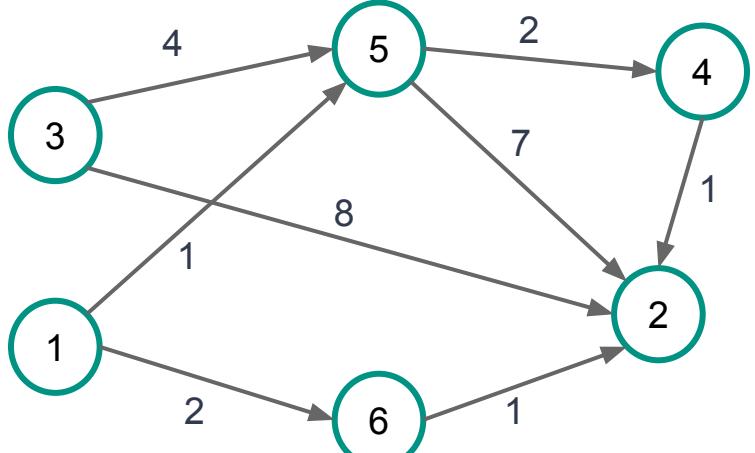
return C

# Exemplu

- **Etapa 2** - parcurgem vârfurile în ordinea dată de sortarea topologică și relaxăm, pentru fiecare vârf, arcele care ies din acesta



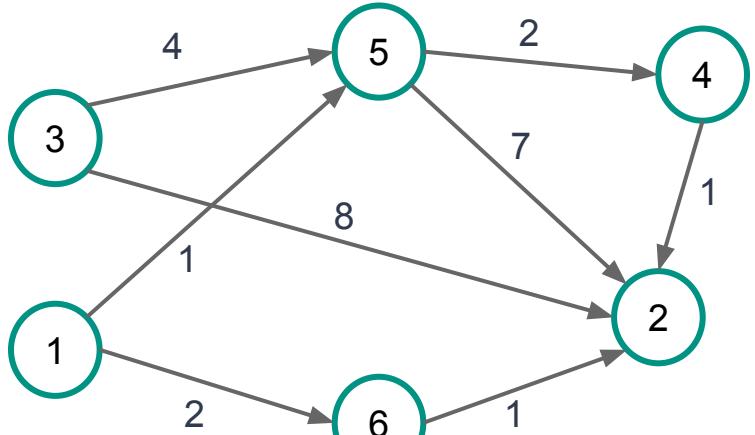
**Sortare topologică**  
1, 3, 6, 5, 4, 2



Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vîrf de start

Ordine de calcul distanțe  
1, 3, 6, 5, 4, 2

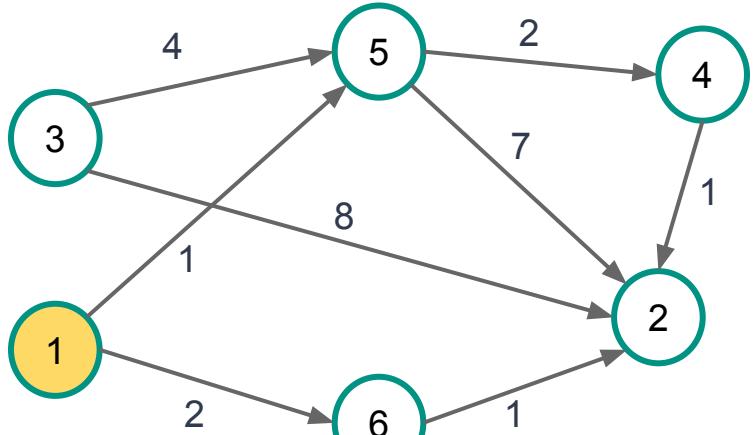


Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vârf de start

Ordine de calcul distanțe  
1, 3, 6, 5, 4, 2

d/tata	1 [ $\infty/0$ ,	2 $\infty/0$ ,	3 0/0,	4 $\infty/0$ ,	5 $\infty/0$ ,	6 $\infty/0$ ]
--------	---------------------	-------------------	-----------	-------------------	-------------------	-------------------



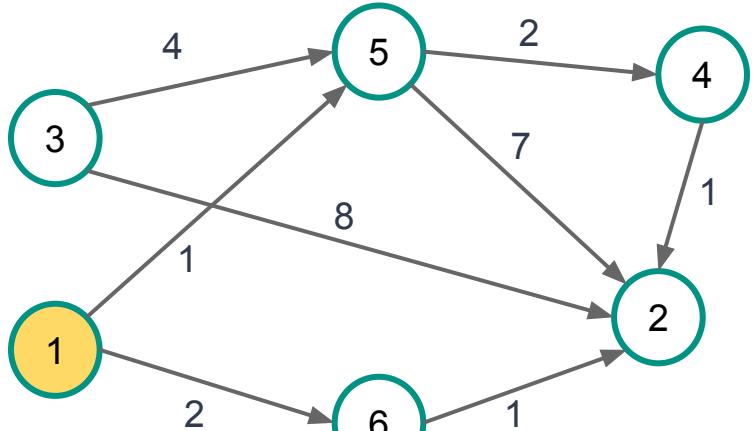
**Sortare topologică**  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ ,	$\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



Sortare topologică  
1, 3, 6, 5, 4, 2

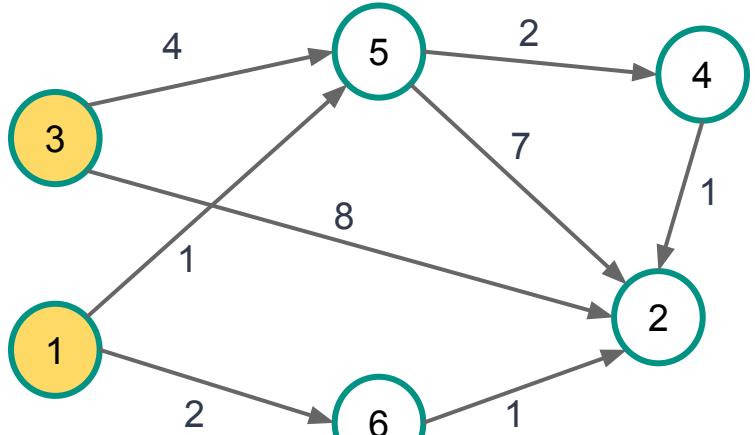
**s = 3 - vârf de start**

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

d/tata	1	2	3	4	5	6
$u = 1$	[ $\infty/0$ , [ $\infty/0$ ,	$\infty/0$ , $\infty/0$ ,	$0/0$ , $0/0$ ,	$\infty/0$ , $\infty/0$ ,	$\infty/0$ , $\infty/0$ ,	$\infty/0$ ]

**1 nu este accesibil din s - putem să nu îl considerăm  
(să ignorăm vâfurile din ordonarea topologică aflate înaintea lui s)**

$$d[v] = \min \{ d[v], d[u]+w(u,v) \}$$



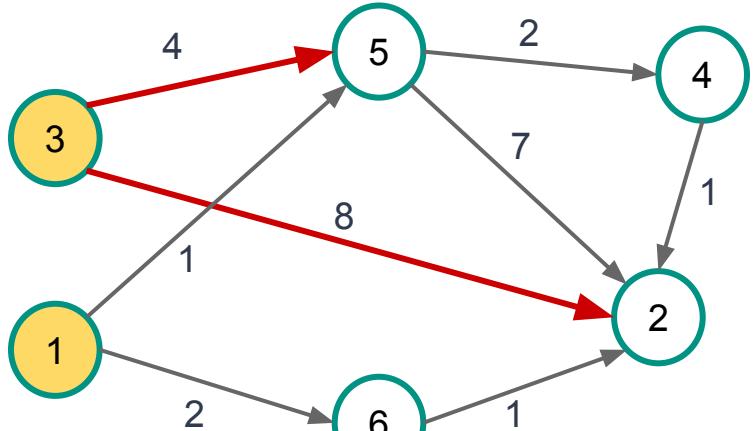
Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vîrf de start

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

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

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



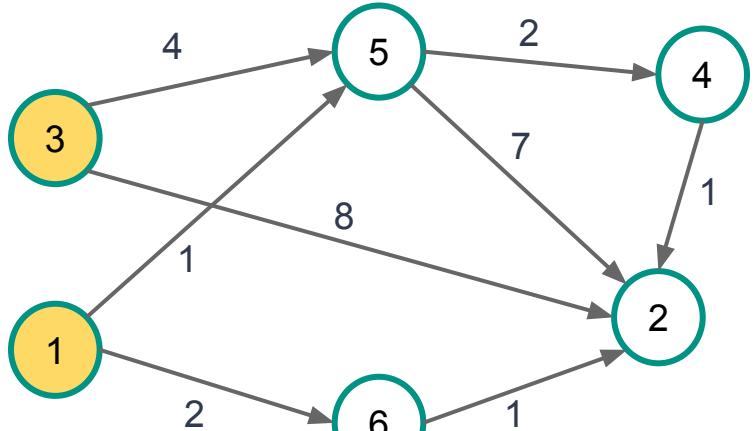
Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vîrf de start

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

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

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



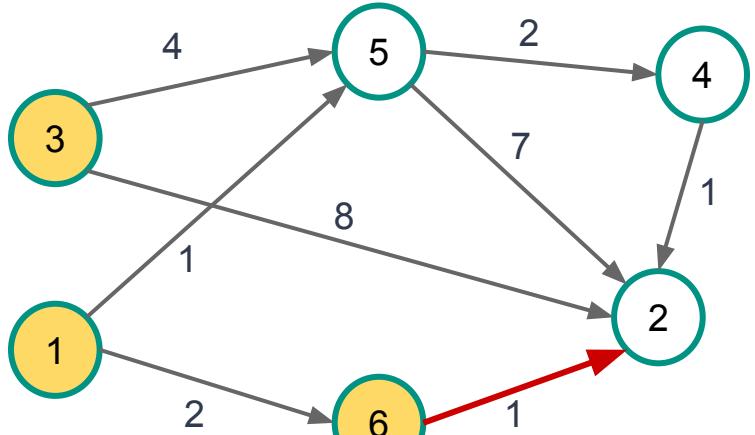
Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vârf de start

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

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

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



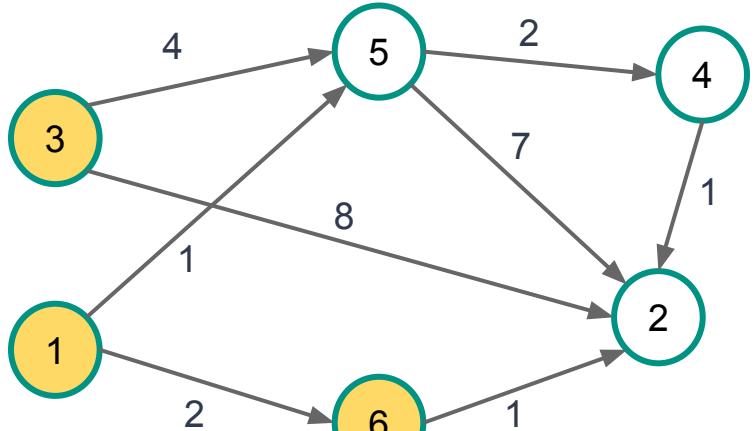
Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vârf de start

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

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

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



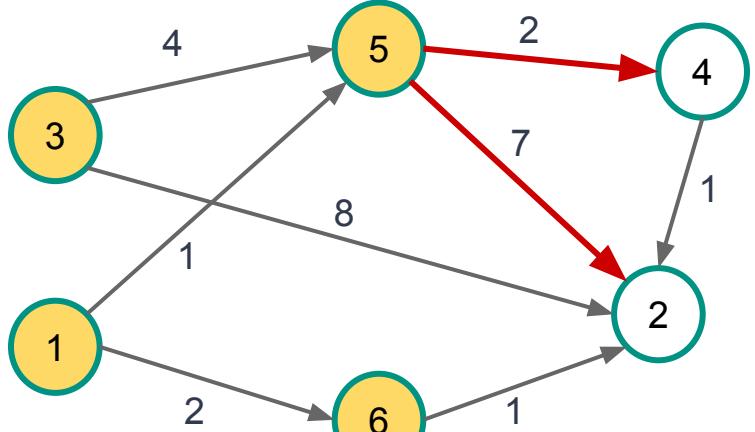
Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]	[ $0/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]
u = 3	[ $\infty/0$ , $8/3$ ,      ]	[ $0/0$ , $\infty/0$ ,      ]	[ $0/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $4/3$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]
u = 6	[ $\infty/0$ , <b><math>8/3</math></b> ,      ]	[ $0/0$ , $\infty/0$ ,      ]	[ $0/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $4/3$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]	[ $\infty/0$ , $\infty/0$ ,      ]

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



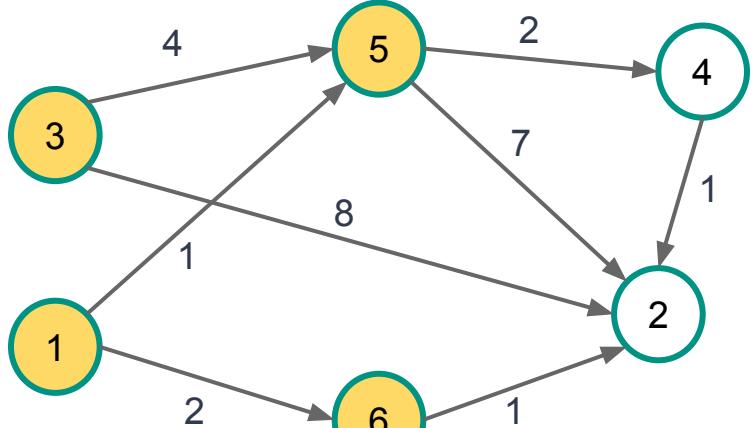
Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3** - vârf de start

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

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

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



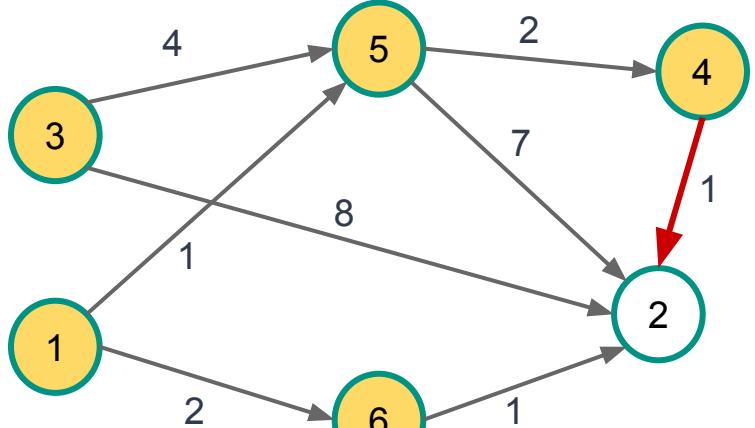
Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,    ]					
u = 3	[ $\infty/0$ , $8/3$ ,    ]					
u = 6	[ $\infty/0$ , $8/3$ ,    ]					
u = 5	[ $\infty/0$ , <b><math>8/3</math></b> ,    ]					

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



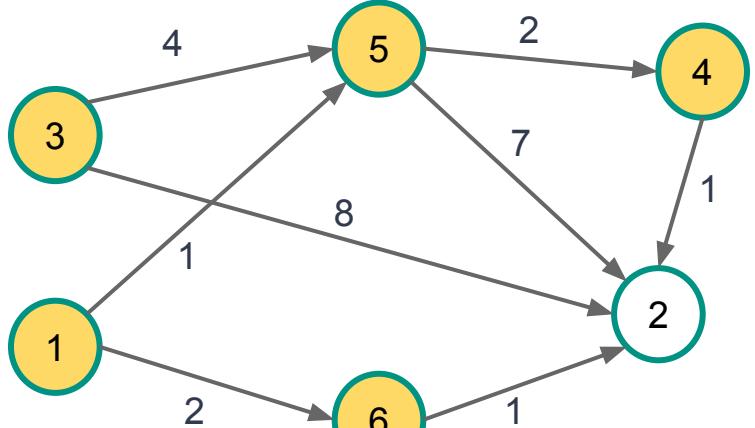
Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,	$\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
u = 3	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 6	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 5	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 4						

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



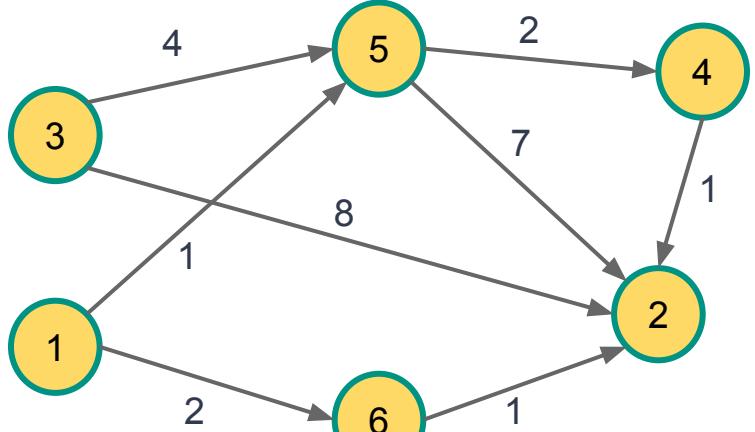
Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,	$\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
u = 3	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 6	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 5	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 4	[ $\infty/0$ , $\infty/0$ ,	<b>7/4</b> ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$



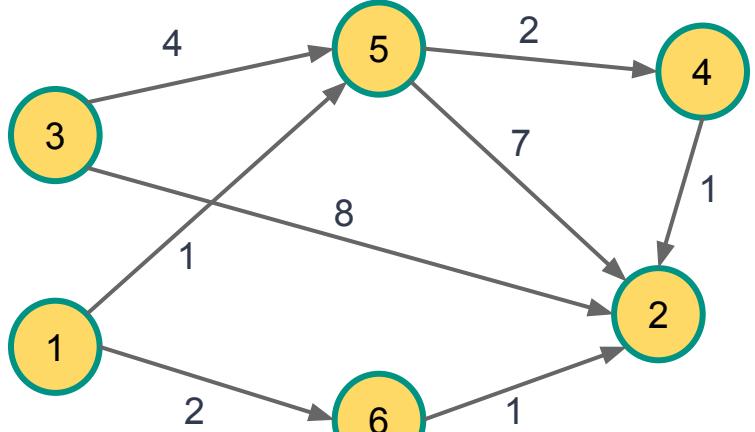
Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,	$\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
u = 3	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 6	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 5	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 4	[ $\infty/0$ , $\infty/0$ ,	$7/4$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 2						

$$d[v] = \min \{ d[v], d[u] + w(u,v) \}$$

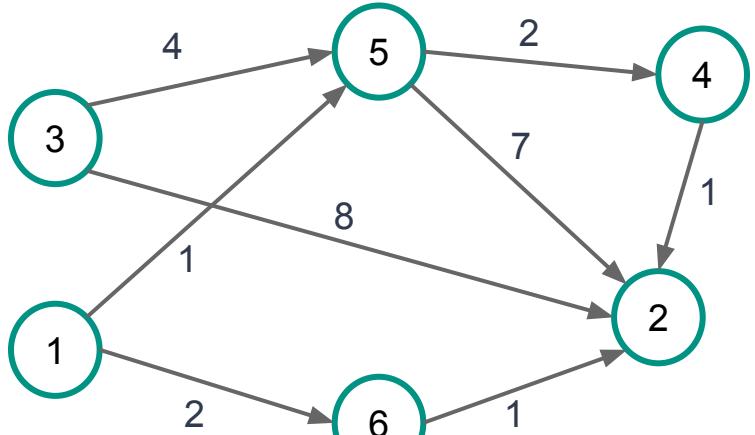


Sortare topologică  
1, 3, 6, 5, 4, 2

**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	[ $\infty/0$ , $\infty/0$ ,	$\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
u = 3	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 6	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$\infty/0$ ,	$4/3$ ,	$\infty/0$ ]
u = 5	[ $\infty/0$ , $\infty/0$ ,	$8/3$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 4	[ $\infty/0$ , $\infty/0$ ,	$7/4$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]
u = 2	[ $\infty/0$ , $\infty/0$ ,	$7/4$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]



Sortare topologică  
1, 3, 6, 5, 4, 2

**s = 3 - vârf de start**

Ordine de calcul distanțe  
**1, 3, 6, 5, 4, 2**

**Soluție**

[  $\infty/0$ ,       $7/4$ ,

$0/0$ ,

$6/5$ ,

$4/3$ ,

$\infty/0$  ]

**Un drum minim de la 3 la 2?**

# Drumuri minime de sursă unică în grafuri aciclice

## Observație

- Este suficient să considerăm, în ordonarea topologică, doar vârfurile accesibile din s
- În exemplu** - fără 1 și 6

# Complexitate



# Drumuri minime de sursă unică în grafuri aciclice

```
s - vârful de start
void df(int i) {
    viz[i] = 1;
    for ij ∈ E
        if (viz[j] == 0)
            df(j);
    // i este finalizat
    push(S, i)
}
for (i=1; i<=n; i++)
    if (viz[i] == 0)
        df(i);
while (not S.empty()) {
    u = S.pop();
    adaugă u în sortare
}
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
// inițializăm distanțe - ca la Dijkstra
pentru fiecare  $u \in V$  execută
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
```

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

```
pentru fiecare  $u \in SortTop$  execută
    pentru fiecare  $uv \in E$  execută
        dacă  $d[u] + w(u,v) < d[v]$  atunci // relaxăm uv
             $d[v] = d[u] + w(u,v)$ 
             $tata[v] = u$ 
```

scrie  $d$ ,  $tata$

# Drumuri minime de sursă unică în grafuri aciclice

## Complexitate

- **Initializare** →  $O(?)$
  - **Sortare topologică** →  $O(?)$
  - **$m * \text{relaxare } uv$**  →  $O(?)$
-

# Drumuri minime de sursă unică în grafuri aciclice

## Complexitate

- **Initializare** →  $O(n)$
- **Sortare topologică** →  $O(m + n)$
- **$m * \text{relaxare } uv$**  →  $O(m)$

---

**$O(m + n)$**

# Corectitudine



# Drumuri minime de sursă unică în grafuri aciclice

Algoritmul funcționează corect și dacă există arce cu cost negativ



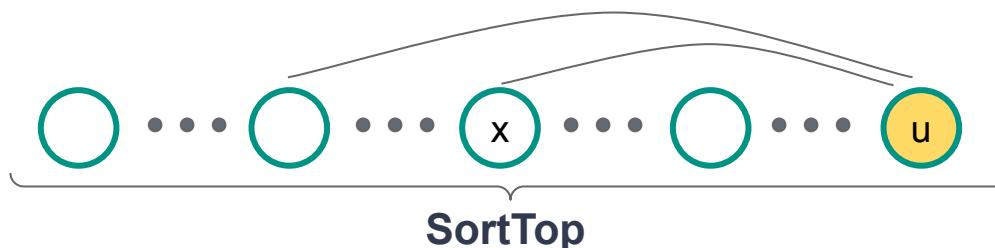
$$\delta(s, u) = \min \{ \delta(s, x) + w(x, u) \mid xu \in E \}$$

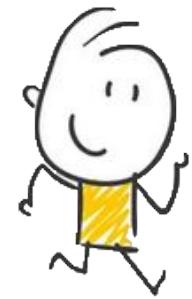
Când algoritmul ajunge la vârful  $u$ , avem:

$$d[u] = \min \{ d[x] + w(x, u) \mid xu \in E \}$$



relaxare arce  $xu$





# Concluzii

## Drumuri minime de sursă unică - algoritmi

# Algoritmi - $G=(V, E)$ graf orientat

$G$  - neponderat

**Parcuregere în lătime (BF)**

**BF(s)**

```
coada C ← ∅  
adauga(s, C)
```

$G$  - ponderat, ponderi  $> 0$

**Algoritmul lui Dijkstra**

**Dijkstra(s)**

```
(min-heap) Q ← V  
// se putea începe doar cu  $Q \leftarrow \{s\}$  +  
vector viz;  $v \in Q \Leftrightarrow v$  nevizitat
```

$G$  - ponderat fără circuite

**DAGS(s)**

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

# Algoritmi - $G=(V, E)$ graf orientat

$G$  - neponderat

**Parcuregere în lătime (BF)**

**BF(s)**

```
coada C  $\leftarrow \emptyset$ 
adauga(s, C)
```

```
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ;  $tata[u] = \text{viz}[u] = 0$ 
 $\text{viz}[s] = 1$ ;  $d[s] = 0$ 
```

$G$  - ponderat, ponderi  $> 0$

**Algoritmul lui Dijkstra**

**Dijkstra(s)**

```
(min-heap)  $Q \leftarrow V$ 
// se putea începe doar cu  $Q \leftarrow \{s\}$  +
vector viz;  $v \in Q \Leftrightarrow v$  nevizitat
```

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

$G$  - ponderat fără circuite

**DAGS(s)**

```
SortTop  $\leftarrow \text{sortare_topologica}(G)$ 
```

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

# Algoritmi - $G=(V, E)$ graf orientat

$G$  - neponderat

**Parcursere în lătime (BF)**

**BF(s)**

```
coada C  $\leftarrow \emptyset$ 
adauga(s, C)
```

```
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ;  $tata[u] = \text{viz}[u] = 0$ 
 $\text{viz}[s] = 1$ ;  $d[s] = 0$ 
```

```
cât timp  $C \neq \emptyset$ 
     $u \leftarrow \text{extrage}(C)$ 
    pentru fiecare  $uv \in E$ 
```

$G$  - ponderat, ponderi  $> 0$

**Algoritmul lui Dijkstra**

**Dijkstra(s)**

```
(min-heap)  $Q \leftarrow V$ 
// se putea începe doar cu  $Q \leftarrow \{s\}$  +
vector viz;  $v \in Q \Leftrightarrow v$  nevizitat
```

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

```
cât timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu
        eticheta  $d$  minimă
    pentru fiecare  $uv \in E$ 
```

$G$  - ponderat fără circuite

**DAGS(s)**

```
SortTop  $\leftarrow \text{sortare_topologica}(G)$ 
```

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

```
pentru fiecare  $u \in \text{SortTop}$ 
    pentru fiecare  $uv \in E$ 
```

# Algoritmi - $G=(V, E)$ graf orientat

$G$  - neponderat

**Parcursere în lătime (BF)**

**BF(s)**

```
coada C ← ∅  
adauga(s, C)
```

```
pentru fiecare u∈V  
    d[u]=∞; tata[u]=viz[u]=0  
  
viz[s]=1; d[s]=0
```

```
cât timp C ≠ ∅  
    u ← extrage(C)  
  
    pentru fiecare uv∈E  
        dacă viz[v] = 0  
            d[v] = d[u] + 1  
            tata[v]=u  
            adauga(v, C)  
            viz[v]=1
```

scrie  $d$ ,  $tata$

$G$  - ponderat, ponderi  $> 0$

**Algoritmul lui Dijkstra**

**Dijkstra(s)**

```
(min-heap) Q ← V  
// se putea începe doar cu Q ← {s} +  
vector viz; v∈Q ⇔ v nevizitat
```

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

```
cât timp Q ≠ ∅  
    u = extrage(Q) vârf cu  
        eticheta d minimă  
    pentru fiecare uv∈E  
        dacă v∉Q și  
            d[u]+w(u,v)<d[v]  
                d[v] = d[u] + w(u,v)  
                tata[v] = u  
                repara(v, Q)
```

scrie  $d$ ,  $tata$

$G$  - 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 uv∈E  
    dacă d[u]+w(u,v) < d[v]  
        d[v] = d[u] + w(u,v)  
        tata[v] = u
```

scrie  $d$ ,  $tata$

# Algoritmi - $G=(V, E)$ graf orientat

$G$  - neponderat

**Parcursere în lătime (BF)**

**BF(s)**

```
coada C ← ∅  
adauga(s, C)
```

```
pentru fiecare u∈V  
    d[u]=∞; tata[u]=viz[u]=0  
  
viz[s]=1; d[s]=0
```

```
cât timp C ≠ ∅  
    u ← extrage(C)  
  
    pentru fiecare uv∈E  
        dacă 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)**

```
(min-heap) Q ← V  
// se putea începe doar cu Q ← {s} +  
vector viz; v∈Q ⇔ v nevizitat
```

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

```
cât timp Q ≠ ∅  
    u = extrage(Q) vârf cu  
        eticheta d minimă  
    pentru fiecare uv∈E  
        dacă v∉Q și  
        d[u]+w(u,v)<d[v]  
            d[v] = d[u] + w(u,v)  
            tata[v] = u  
            repară(v, Q)
```

scrie  $d$ ,  $tata$

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

$G$  - 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

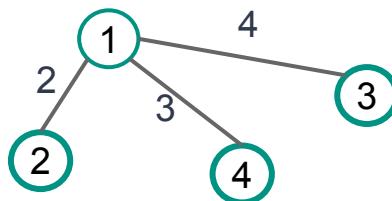
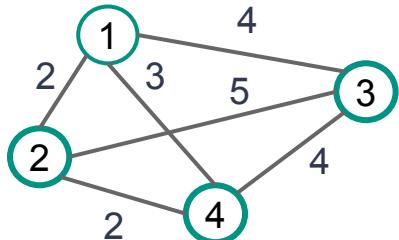
```
pentru fiecare uv∈E  
    dacă d[u]+w(u,v) < d[v]  
        d[v] = d[u] + w(u,v)  
        tata[v] = u
```

scrie  $d$ ,  $tata$

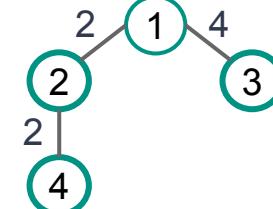
$O(n+m)$

# Drumuri minime vs APCM

Drumuri minime din  $s \Rightarrow$  arbore de drumuri minime (distanțe) din  $s \neq$  APCM - minimizează costul total



arbore al  
drumurilor  
minime față  
de 1

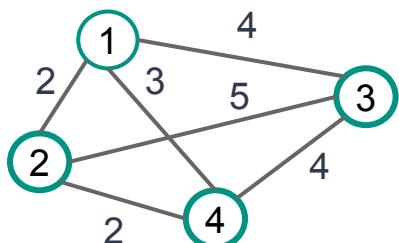


arbore parțial  
de cost minim

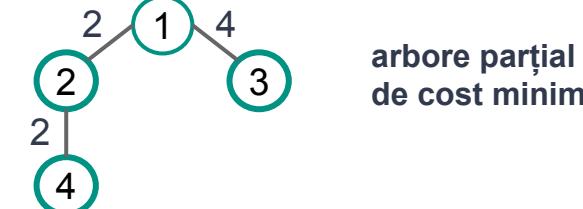
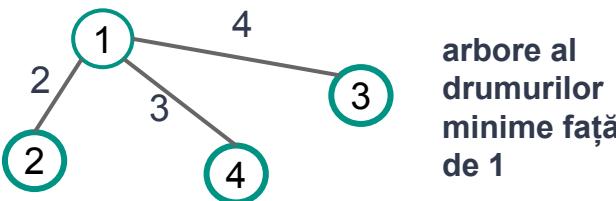
# Drumuri minime vs APCM

Drumuri minime din  $s \Rightarrow$  arbore de drumuri minime (distanțe) din  $s \neq$  APCM - minimizează costul total

G - (ne)orientat ponderat, ponderi  $> 0$   
Drumuri minime din  $s$   
**Algoritmul lui Dijkstra**

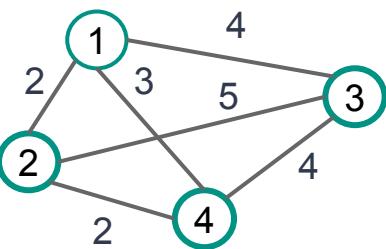


G - neorientat ponderat, ponderi reale  
Arbore parțial de cost minim  
**Algoritmul lui Prim**



# Drumuri minime vs APCM

Drumuri minime din  $s \Rightarrow$  arbore de drumuri minime (distanțe) din  $s \neq$  APCM - minimizează costul total



G - (ne)orientat ponderat, ponderi  $> 0$

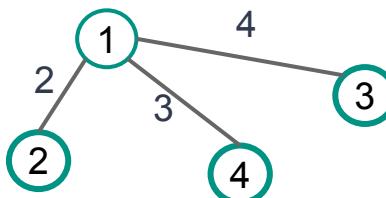
Drumuri minime din  $s$

**Algoritmul lui Dijkstra**

Dijkstra( $s$ )

```
(min-heap) Q ← V
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
cât timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu
        eticheta  $d$  minimă
    pentru fiecare  $uv \in E$ 
        dacă  $v \notin Q$  și  $d[u] + w(u, v) < d[v]$ 
             $d[v] = d[u] + w(u, v)$ 
             $tata[v] = u$ 
            repară( $v$ ,  $Q$ )
scrie  $d$ ,  $tata$ 
```

scrie  $d$ ,  $tata$



arbore al  
drumurilor  
minime față  
de 1

G - neorientat ponderat, ponderi reale

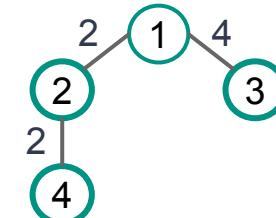
Arbore parțial de cost minim

**Algoritmul lui Prim**

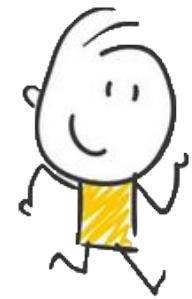
Prim( $s$ )

```
(min-heap) Q ← V
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
cât timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu
        eticheta  $d$  minimă
    pentru fiecare  $uv \in E$ 
        dacă  $v \in Q$  și  $w(u, v) < d[v]$ 
             $d[v] = w(u, v)$ 
             $tata[v] = u$ 
            repară( $v$ ,  $Q$ )
scrie  $d$ ,  $tata$ , pentru  $u \neq s$ 
```

scrie  $d$ ,  $tata$ , pentru  $u \neq s$



arbore parțial  
de cost minim



# Drumuri minime între toate perechile de varfuri

Algoritmul Floyd-Warshall

# Problemă

Se dă:

- un graf **orientat** ponderat  $G = (V, E, w)$

Pentru oricare două vârfuri  $x$  și  $y$  ale lui  $G$ , să se determine distanța de la  $x$  la  $y$  și un drum minim de la  $x$  la  $y$ .

Ponderile pot fi și **negative** dar **NU** există **circuite cu cost negativ** în  $G$ .

# Problemă



## Soluția 1

Se aplică algoritmul lui Dijkstra pentru fiecare vârf x.

# Problemă



## Soluția 1

Se aplică algoritmul lui Dijkstra pentru fiecare vârf x.

**!!! funcționează dacă ponderile sunt pozitive**

**Complexitate =  $n * \text{complexitate Dijkstra}$**

# Problemă



## Soluția 2

Se aplică algoritmul lui Bellman Ford pentru fiecare vârf x.

# Problemă



## Soluția 2

Se aplică algoritmul lui Bellman Ford pentru fiecare vârf x.

**Complexitate =  $n * \text{Complexitate Bellman Ford}$**

$$\rightarrow n * n * m = n^2 * m$$



# Problemă



## Soluția 3

Algoritmul Floyd-Warshall

# Algoritmul Floyd-Warshall

# Floyd-Warshall

Fie  $W = (w_{ij})_{i,j=1,\dots,n}$  **matricea costurilor** grafului G:

$$w_{ij} = \begin{cases} 0, & \text{dacă } i = j \\ w(i, j), & \text{dacă } ij \in E \\ \infty, & \text{dacă } ij \notin E \end{cases}$$

Vrem să calculăm **matricea distanțelor**  $D = (d_{ij})_{i,j=1,\dots,n}$ :

$$d_{ij} = \delta(i, j)$$

# Floyd-Warshall

Fie  $W = (w_{ij})_{i,j=1,\dots,n}$  **matricea costurilor** grafului G:

$$w_{ij} = \begin{cases} 0, & \text{dacă } i = j \\ w(i, j), & \text{dacă } ij \in E \\ \infty, & \text{dacă } ij \notin E \end{cases}$$

Vrem să calculăm **matricea distanțelor**  $D = (d_{ij})_{i,j=1,\dots,n}$ :

$$d_{ij} = \delta(i, j)$$

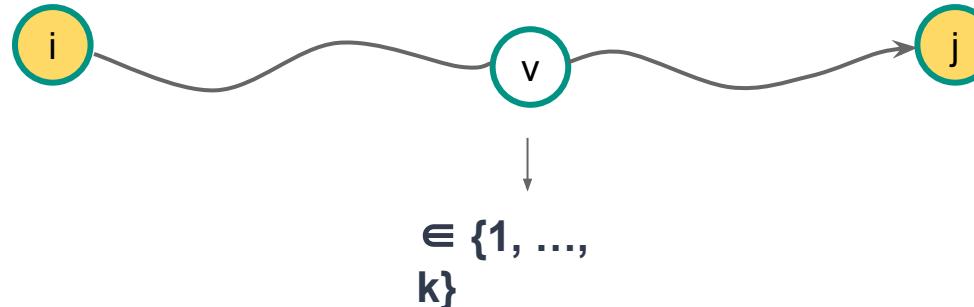
- **Observație:**  $w_{ij}$  = costul minim al unui i-j drum, fără vârfuri intermediare (cu cel mult un arc)

# Floyd-Warshall

## □ Ideea algoritmului Floyd-Warshall:

Pentru  $k = 1, 2, \dots, n$ , calculăm, pentru oricare două vârfuri  $i, j$ :

**costul minim al unui drum de la  $i$  la  $j$ , care are ca vârfuri intermedii doar vârfuri din mulțimea  $\{1, 2, \dots, k\}$**



# Floyd-Warshall

## □ Ideea algoritmului Floyd-Warshall:

Astfel, pentru  $k = 1, 2, \dots, n$ , calculăm matricea

$$D^{(k)} = (d_{ij}^k)_{i,j=1,\dots,n}$$

$d_{ij}^k$  = costul minim al unui drum de la  $i$  la  $j$ , care are vârfurile intermediare în  $\{1, 2, \dots, k\}$

## □ Inițializare:



# Floyd-Warshall

## □ Ideea algoritmului Floyd-Warshall:

Astfel, pentru  $k = 1, 2, \dots, n$ , calculăm matricea

$$D^{(k)} = (d_{ij}^k)_{i,j=1,\dots,n}$$

$d_{ij}^k$  = costul minim al unui drum de la  $i$  la  $j$ , care are vârfurile intermediare în  $\{1, 2, \dots, k\}$

## □ Inițializare: $D^{(0)} = W$



**Care este matricea distanțelor?**

# Floyd-Warshall

## □ Ideea algoritmului Floyd-Warshall:

Astfel, pentru  $k = 1, 2, \dots, n$ , calculăm matricea

$$D^{(k)} = (d_{ij}^k)_{i,j=1,\dots,n}$$

$d_{ij}^k$  = costul minim al unui drum de la  $i$  la  $j$ , care are vârfurile intermediare în  $\{1, 2, \dots, k\}$

## □ Inițializare: $D^{(0)} = W$

## □ Avem $D^{(n)} = D$

# Floyd-Warshall

- Ideea algoritmului Floyd-Warshall:

Pentru a reține și un drum minim



# Floyd-Warshall

## □ Ideea algoritmului Floyd-Warshall:

Pentru a reține și un drum minim

- matrice de predecesori  $P^{(k)} = (p_{ij}^k)_{i,j=1,\dots,n}$
- $p_{ij}^k$  = predecesorul lui  $j$  pe drumul minim curent găsit de la  $i$  la  $j$ , care are vârfurile intermediare în  $\{1, 2, \dots, k\}$

# Floyd-Warshall



**Cum calculăm elementele matricei  $D^{(k)}$ ?**

# Floyd-Warshall

**Cum calculăm elementele matricei  $D^{(k)}$ ?**



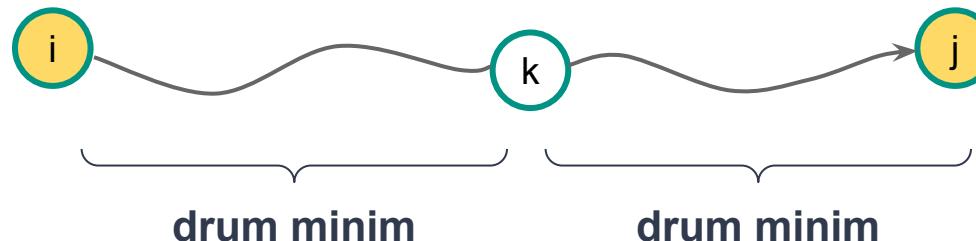
**Relație de recurență**

# Floyd-Warshall

## □ Ideea de calcul a matricei $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$ , cu vârfurile intermediare în multimea  $\{1, 2, \dots, k\}$ .

- Dacă vârful  $k$  este vârf intermediar al lui  $P$

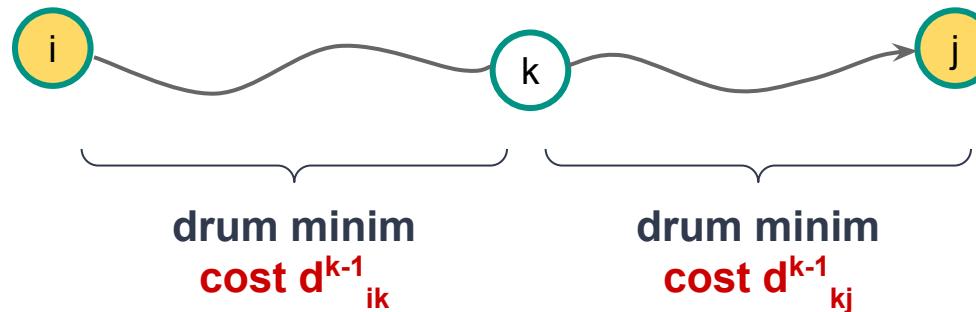


# Floyd-Warshall

## □ Ideea de calcul a matricei $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$ , cu vârfurile intermediare în multimea  $\{1, 2, \dots, k\}$ .

- Dacă vârful  $k$  este vârf intermediar al lui  $P$

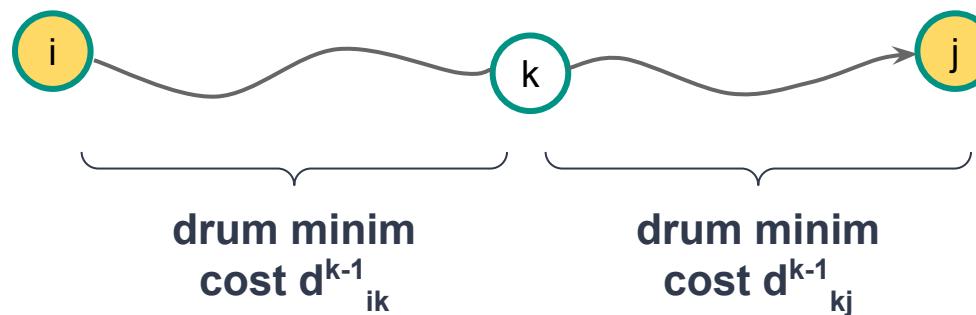


# Floyd-Warshall

## □ Ideea de calcul a matricei $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$ , cu vârfurile intermediare în multimea  $\{1, 2, \dots, k\}$ .

- Dacă vârful  $k$  este vârf intermediar al lui  $P$



$$\Rightarrow d^k_{ij} = \min \{ d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj} \}$$

# Floyd-Warshall

- Se obține, astfel, relația:

$$d^k_{ij} = \min \{ d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj} \}$$

- Observații

- Avem:

$$d^k_{ik} = d^{k-1}_{ik}$$

$$d^k_{kj} = d^{k-1}_{kj}$$

De aceea, în implementarea algoritmului, putem folosi o singură matrice

# Floyd-Warshall

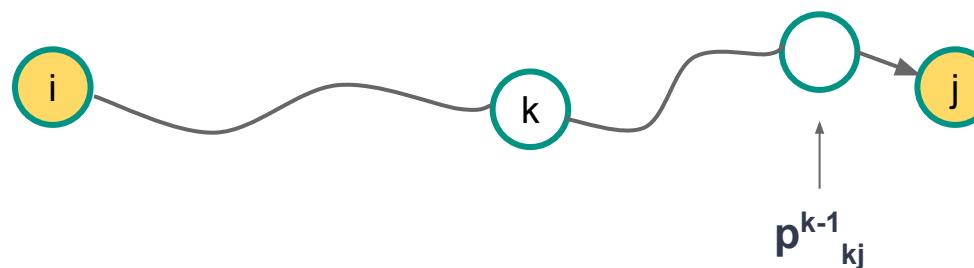
- Se obține, astfel, relația:

$$d^k_{ij} = \min \{ d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj} \}$$

- Observații

Când se actualizează  $d^k_{ij} = d^{k-1}_{ik} + d^{k-1}_{kj}$ , trebuie actualizat și  $p^k_{ij}$

$$p^k_{ij} = p^{k-1}_{kj}$$



# Implementare



# Floyd-Warshall

Conform observațiilor anterioare, putem folosi o unică matrice D

## □ Inițializare

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

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

# Floyd-Warshall

```
for (i=1; i <= n; i++)          // initial, d = matricea costurilor
    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;
    }
```

# Floyd-Warshall

```
for (i=1; i <= n; i++)          // initial, d = matricea costurilor
    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++)      // varfuri intermediiare
```

# Floyd-Warshall

```
for (i=1; i <= n; i++)           // initial, d = matricea costurilor
    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++)
```

# Floyd-Warshall

```
for (i=1; i <= n; i++)          // initial, d = matricea costurilor
    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];
            }
```

# Floyd-Warshall

Ieșire: matricea  $d$  = matricea distanțelor minime

Afișarea unui drum de la  $i$  la  $j$ , dacă  $d[i][j] < \infty$ , se face folosind matricea  $p$

# Floyd-Warshall

Ieșire: matricea  $d = \text{matricea distanțelor minime}$

Afișarea unui drum de la  $i$  la  $j$ , dacă  $d[i][j] < \infty$ , se face folosind matricea  $p$

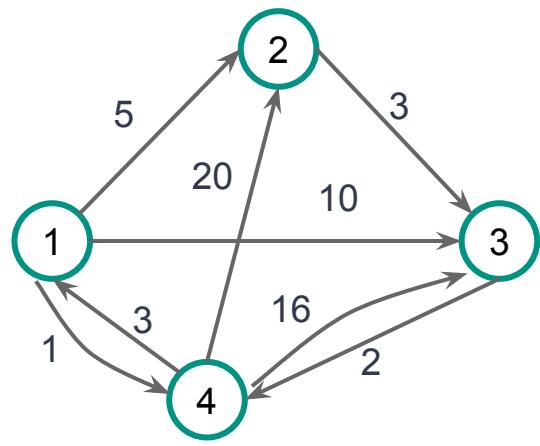
```
void drum(int i, int j) {
    if (i != j)
        drum(i, p[i][j]);
    cout << j << " ";
}
```

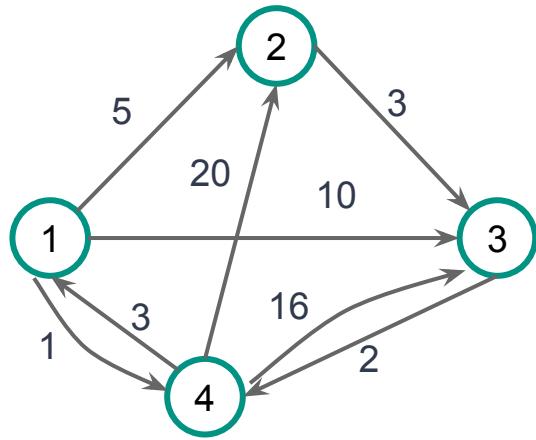
# Floyd-Warshall

**Complexitate - ?**

# Floyd-Warshall

**Complexitate -  $O(n^3)$**



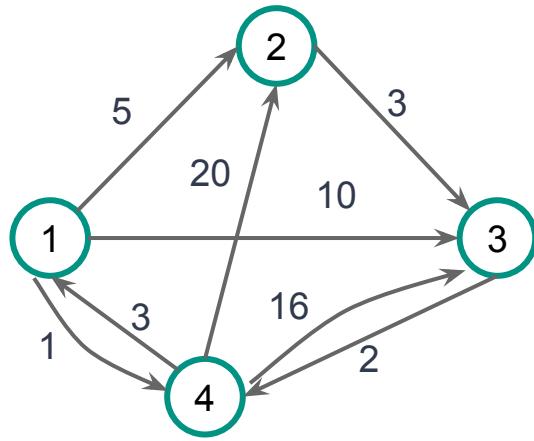


$$W = d =$$

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	20	16	0

$$p =$$

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0



$$W = d =$$

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	20	16	0

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

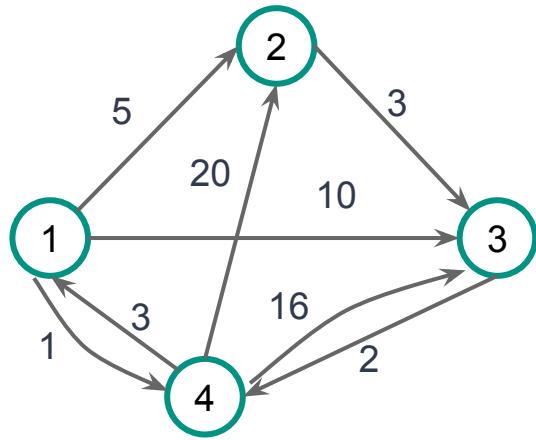
**k=1**

d =

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

p =

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0



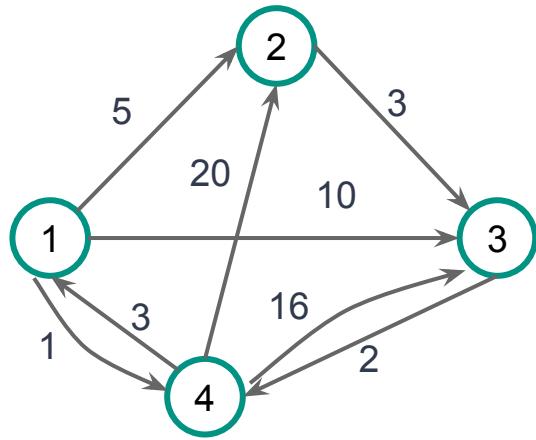
$$W = d =$$

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

$$p =$$

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

$k=1$	$d =$	$p =$
0 5 10 1	$\infty$ 0 3 $\infty$	0 1 1 1
$\infty$ 0 3 $\infty$	0 0 2 0	0 0 2 0
$\infty$ $\infty$ 0 2	0 0 0 3	0 0 0 3
3 8 13 0	3 8 11 0	4 4 4 0
$k=2$		
0 5 10 1	0 5 8 1	0 1 2 1
$\infty$ 0 3 $\infty$	$\infty$ 0 3 $\infty$	0 0 2 0
$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2	0 0 0 3
3 8 11 0	3 8 11 0	4 1 2 0



$$W = d =$$

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

$$p =$$

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

**k=1**

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

**d =**

**k=2**

0	5	8	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	11	0

**k=3**

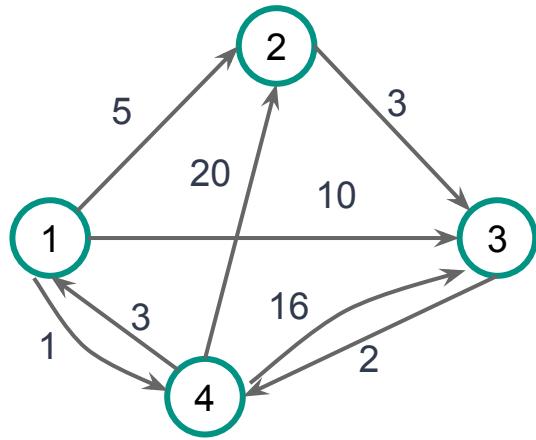
0	5	8	1
$\infty$	0	3	5
$\infty$	$\infty$	0	2
3	8	11	0

**p =**

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0

0	1	2	1
0	0	2	3
0	0	0	3
4	1	2	0



$$W = d =$$

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

**k=1**

0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	13	0

**d =**

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

**p =**

**k=2**

0	5	8	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	8	11	0

**k=3**

0	5	8	1
$\infty$	0	3	5
$\infty$	$\infty$	0	2
3	8	11	0

**k=4**

0	5	8	1
8	0	3	5
5	10	0	2
3	8	11	0

0	1	2	1
4	0	2	3
4	1	0	3
4	1	2	0

# Floyd-Warshall

Algoritmul funcționează corect chiar dacă arcele au și costuri negative (dar graful nu are circuite negative).



- Cum putem detecta, pe parcursul algoritmului, existența unui circuit negativ? ( $\Rightarrow$  datele de intrare nu sunt corecte)

# Floyd-Warshall

Algoritmul funcționează corect chiar dacă arcele au și costuri negative (dar graful nu are circuite negative).



- Cum putem detecta, pe parcursul algoritmului, existența unui circuit negativ? ( $\Rightarrow$  datele de intrare nu sunt corecte)
- Reușim să optimizăm diagonala principală să obținem un cost negativ!

# Aplicație Închiderea tranzitivă a unui graf orientat

Algoritmul Roy-Warshall

# Roy-Warshall

**Aplicație: Închiderea tranzitivă a unui graf orientat  $G=(V, E)$  (**!!! neponderat**):**

$G^* = (V, E^*)$ , unde

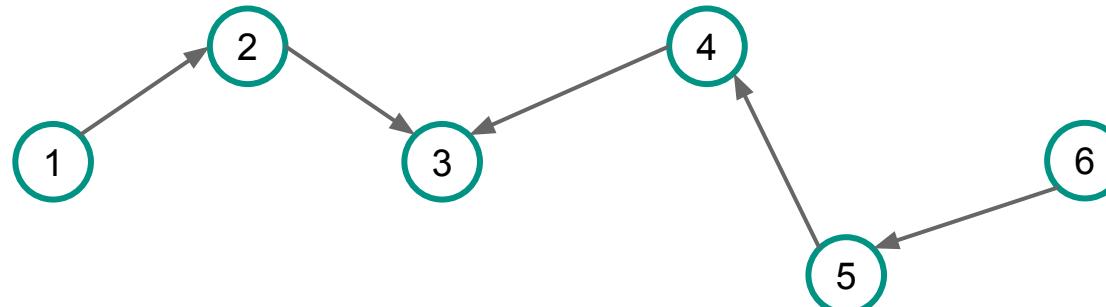
$E^* = \{ (i, j) \mid \text{există drum (de lungime minim 1) de la } i \text{ la } j \text{ în } G \}$

**Utilitate:**

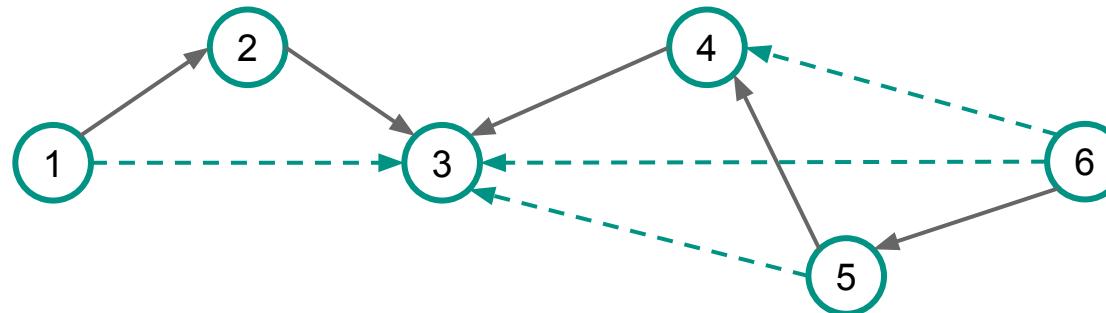
- grupări de obiecte aflate în relație (directă sau indirectă):** optimizări în baze de date, analize în rețele, logică

# Roy-Warshall

## Exemplul 1

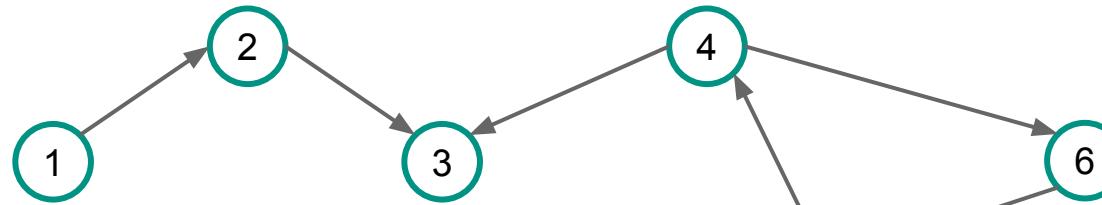


Închiderea tranzitivă

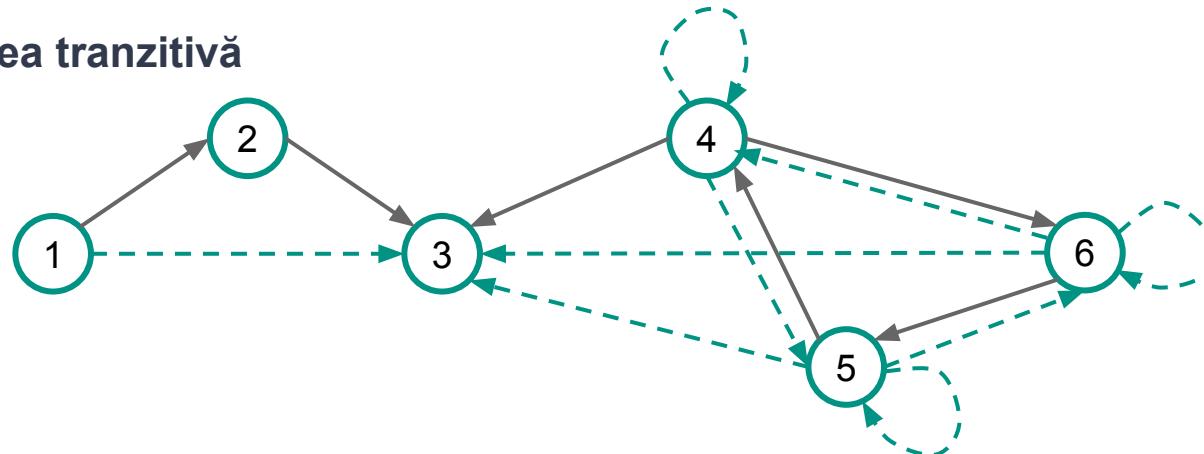


# Roy-Warshall

## Exemplul 2



Închiderea tranzitivă



# Roy-Warshall

Închiderea tranzitivă  $\Leftrightarrow$  calculăm matricea existenței drumurilor (matricea de adiacență a înciderii tranzitive)

$D = (d_{ij})_{i, j = 1, \dots, n}$ :

$$d_{ij} = \begin{cases} 1, & \text{dacă există drum nevid de la } i \text{ la } j \\ 0, & \text{altfel} \end{cases}$$

# Roy-Warshall

initial d = matricea de adiacență

```
for (k=1; k<=n; k++)
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            d[i][j] = d[i][j]  V  (d[i][k]  ^  d[k][j]);
```

**SAU**

**SI**

# Roy-Warshall

## Observație

Dacă A este matricea de adiacență a unui graf și

$$A^k = (a_{ij}^k)_{i,j=1,\dots,n} : \text{puterea } k \text{ a matricei } (k < n)$$

atunci  $a_{ij}^k = \text{numărul de drumuri distințe de lungime } k \text{ de la } i \text{ la } j$  (! nu neapărat elementare)

**Demonstrație - Inducție. Temă**

# Roy-Warshall

## Observație

Dacă A este matricea de adiacență a unui graf și

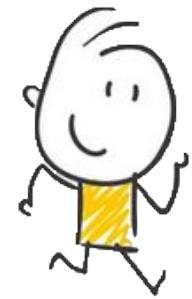
$$A^k = (a_{ij}^k)_{i,j=1,\dots,n} : \text{puterea } k \text{ a matricei } (k < n)$$

atunci  $a_{ij}^k = \text{numărul de drumuri distințe de lungime } k \text{ de la } i \text{ la } j$  (! nu neapărat elementare)

## Consecință

$$D = A \vee A^2 \vee \dots \vee A^{n-1}$$

unde o valoare diferită de 0 se interpretează ca **true**



# Fluxuri maxime în rețele de transport



# Fluxuri maxime în rețele de transport



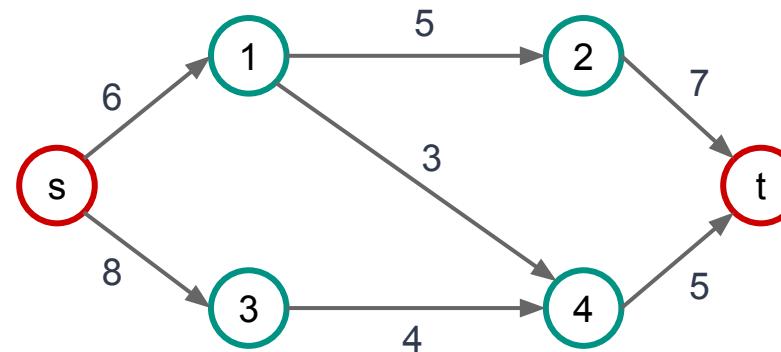
# Problemă



Avem o rețea în care

- arcele au limitări de capacitate
- nodurile = joncțiune

Care este cantitatea maximă care poate fi transmisă prin rețea, de la surse la destinații? (în unitatea de timp)



# Aplicații

## □ Rețea de comunicare

- Transferul de informații - limitat de lățimea de bandă

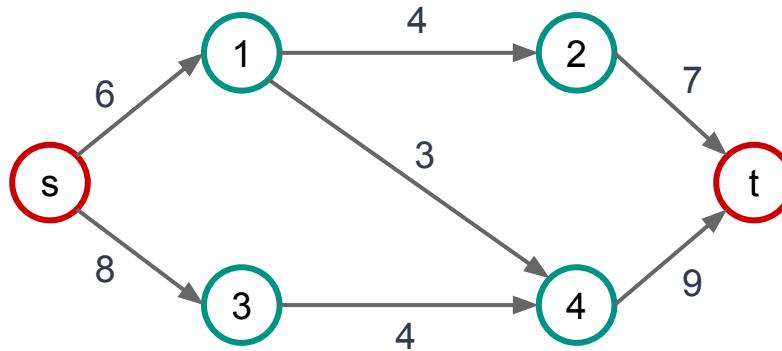
## □ Rețele de transport / evacuare în caz de urgență

- Limitare - număr de mașini / persoane în unitatea de timp

## □ Rețele de conducte

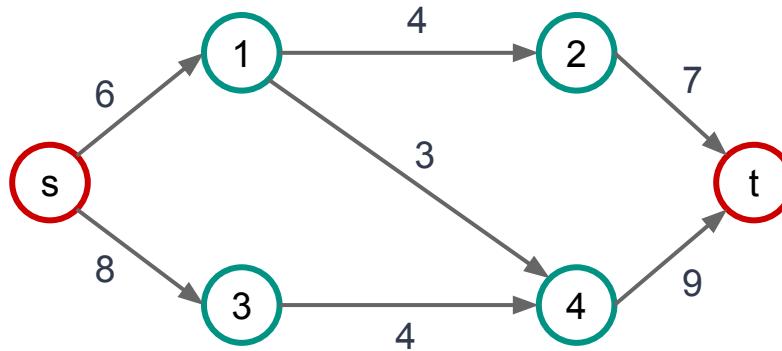
□ ...

# Fluxuri în rețele de transport



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

# Fluxuri în rețele de transport

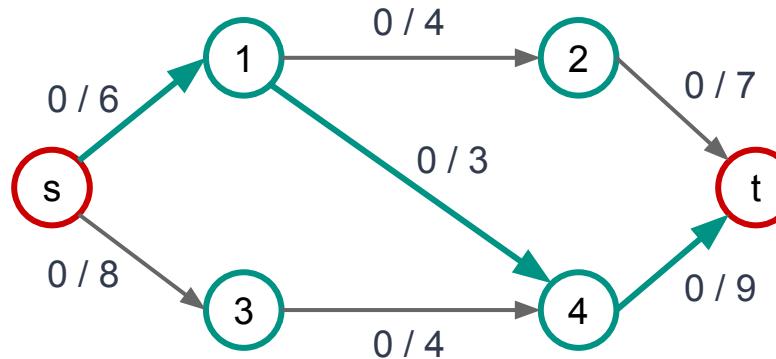


Încercăm să trimitem marfă (flux) de la vârful sursă s la destinația t

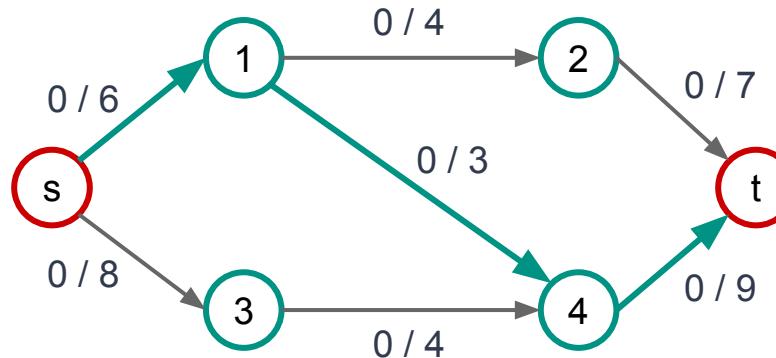


Determinăm drumuri de la s la t pe care mai putem trimite marfă

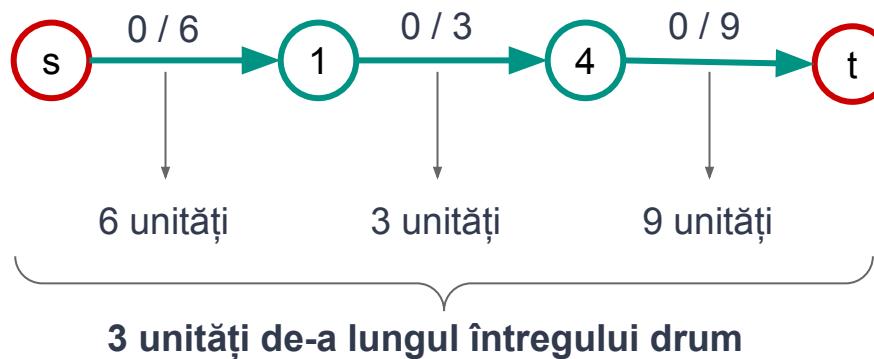
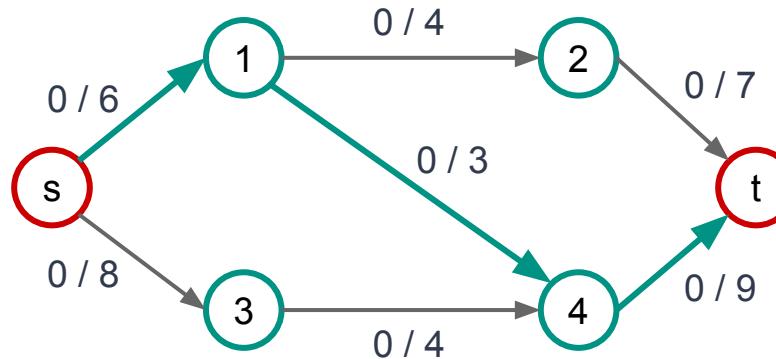
# Fluxuri în rețele de transport



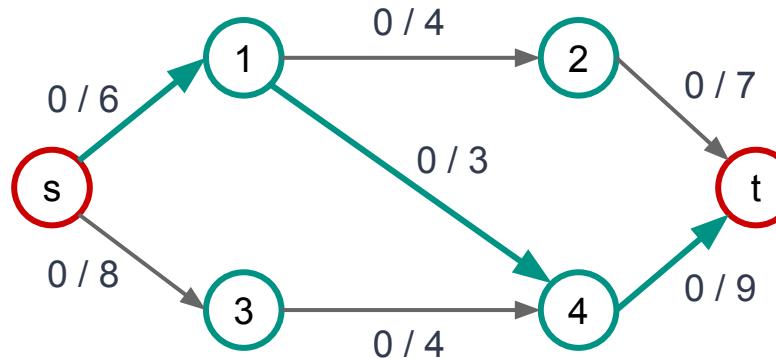
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

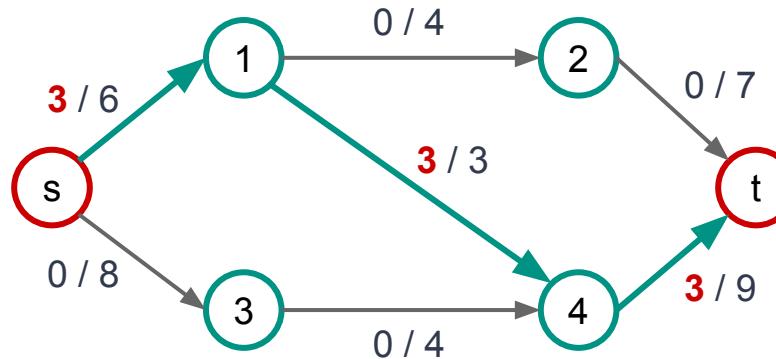


# Fluxuri în rețele de transport

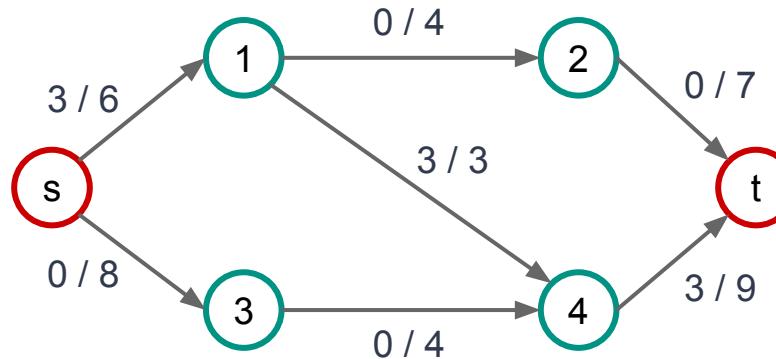


Putem trimite 3 unități de-a lungul întregului drum

# Fluxuri în rețele de transport

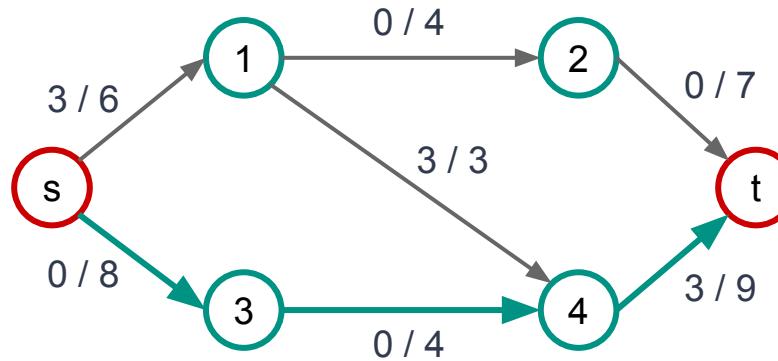


# Fluxuri în rețele de transport

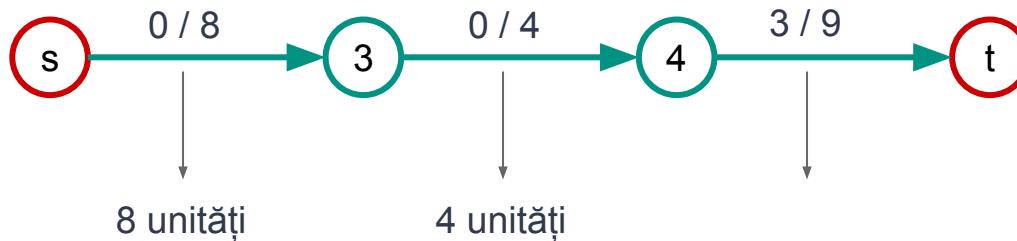
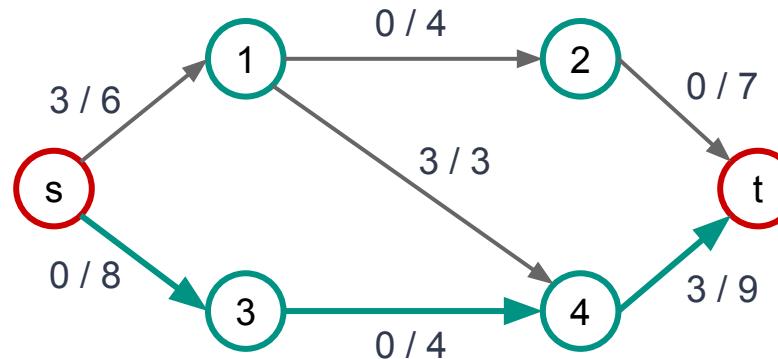


Căutăm alt drum de la s la t pe care mai putem trimite flux

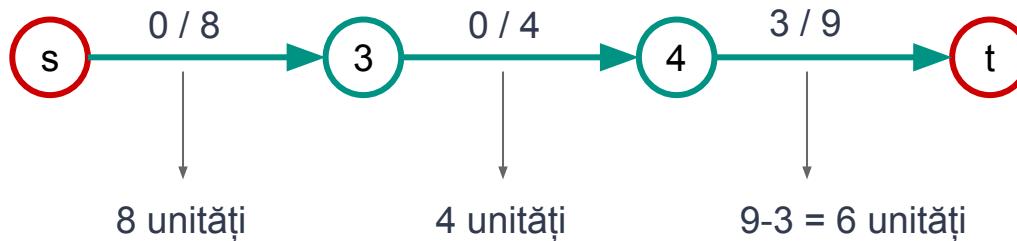
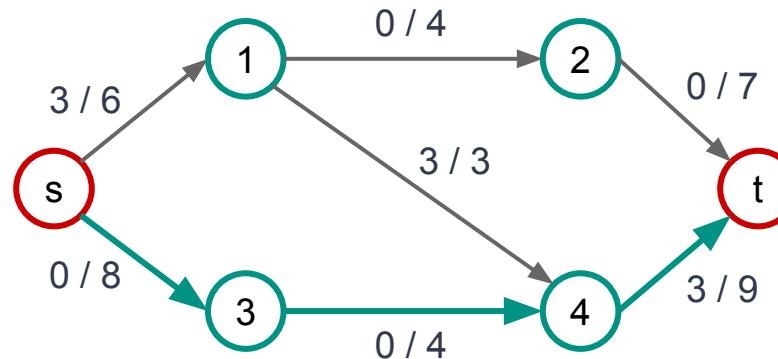
# Fluxuri în rețele de transport



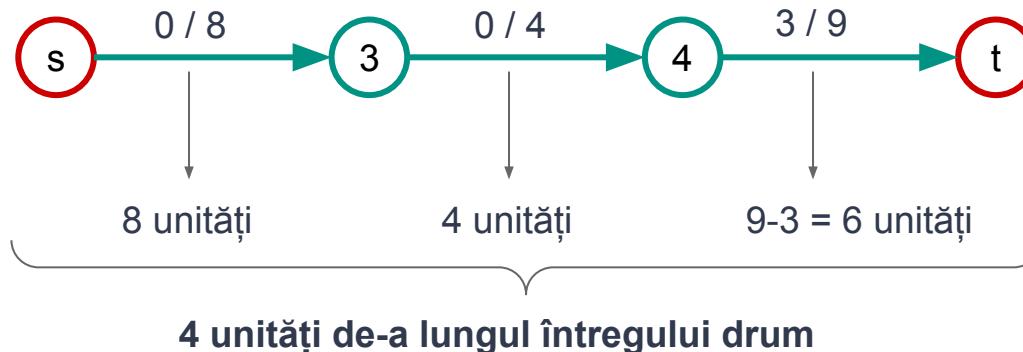
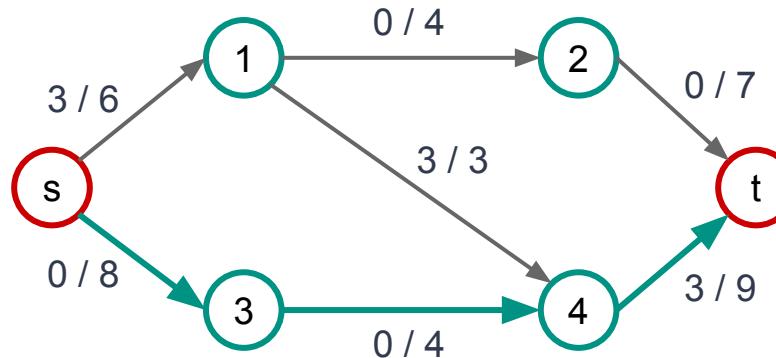
# Fluxuri în rețele de transport



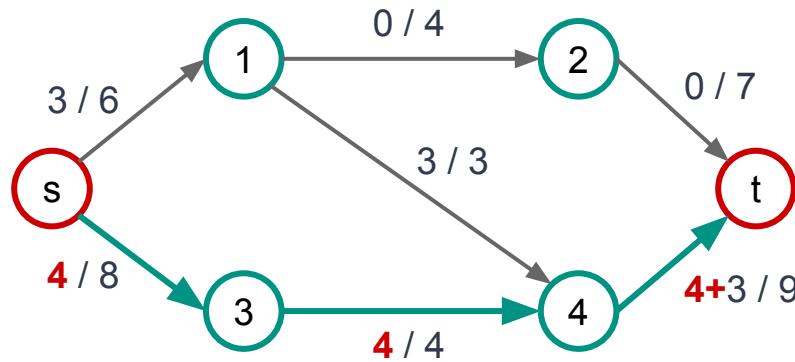
# Fluxuri în rețele de transport



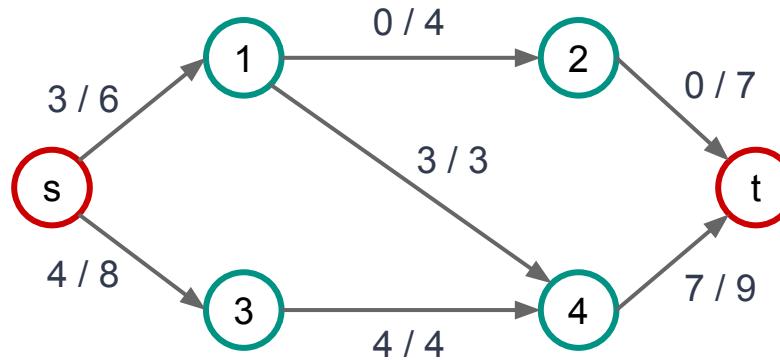
# Fluxuri în rețele de transport



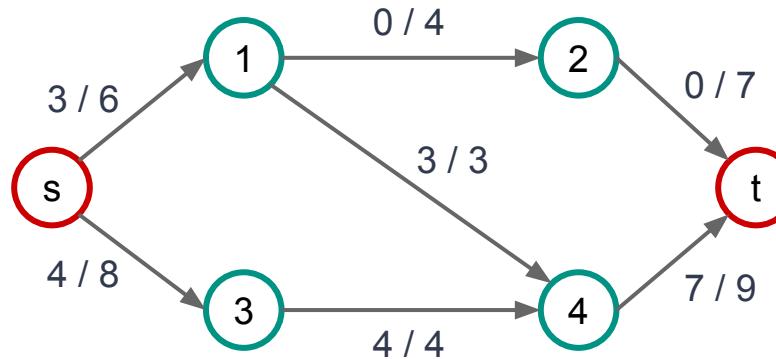
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

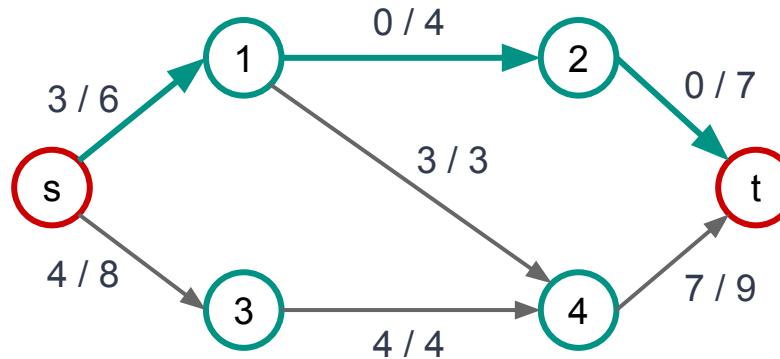


# Fluxuri în rețele de transport

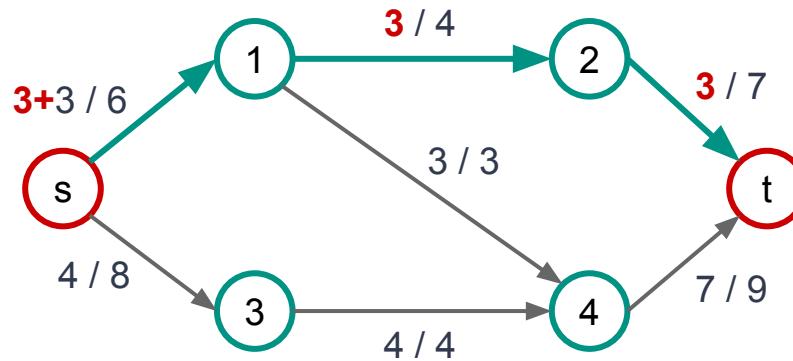


Căutăm alt drum de la s la t pe care mai putem trimite flux

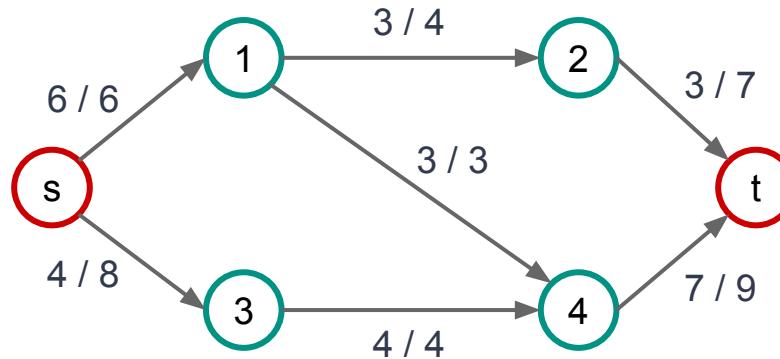
# Fluxuri în rețele de transport



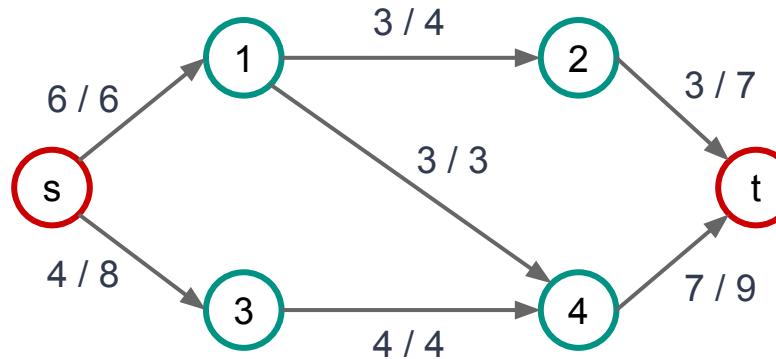
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

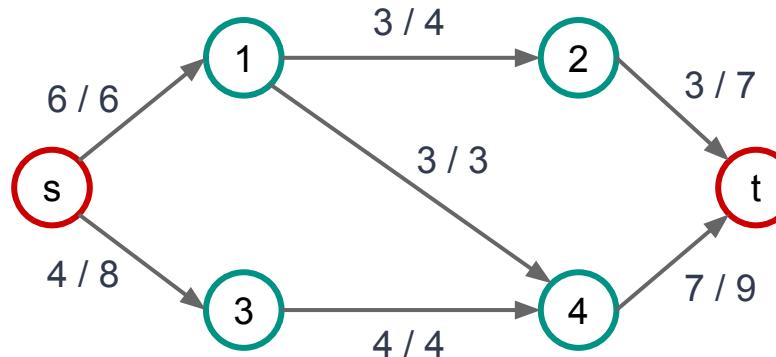


# Fluxuri în rețele de transport



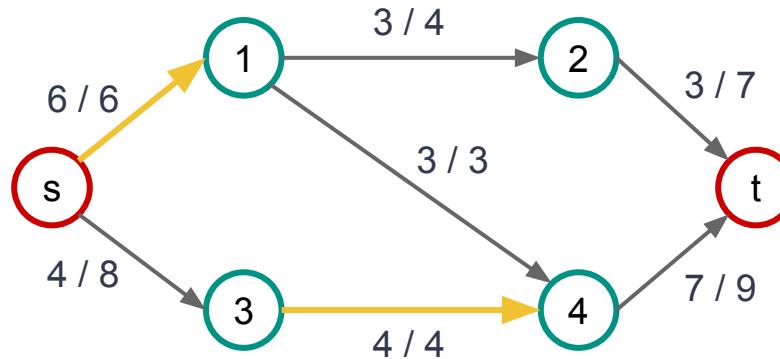
Nu mai există drumuri de la s la t pe care mai putem trimite flux

# Fluxuri în rețele de transport

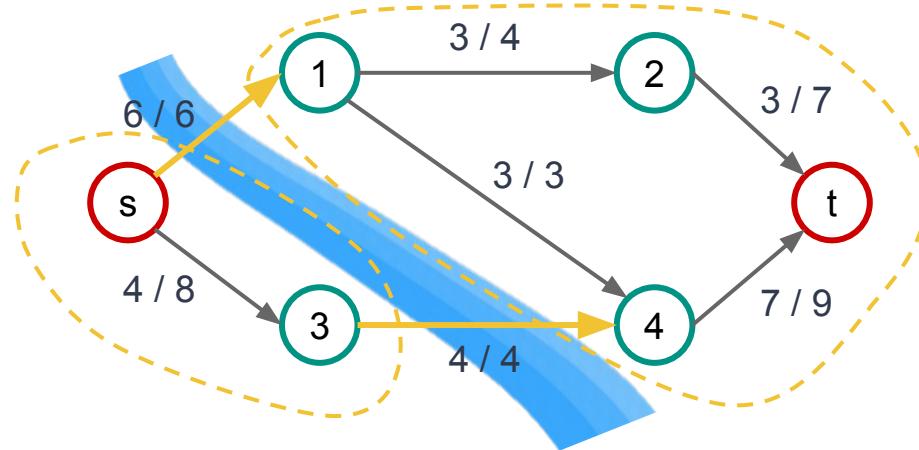


Este maxim fluxul?

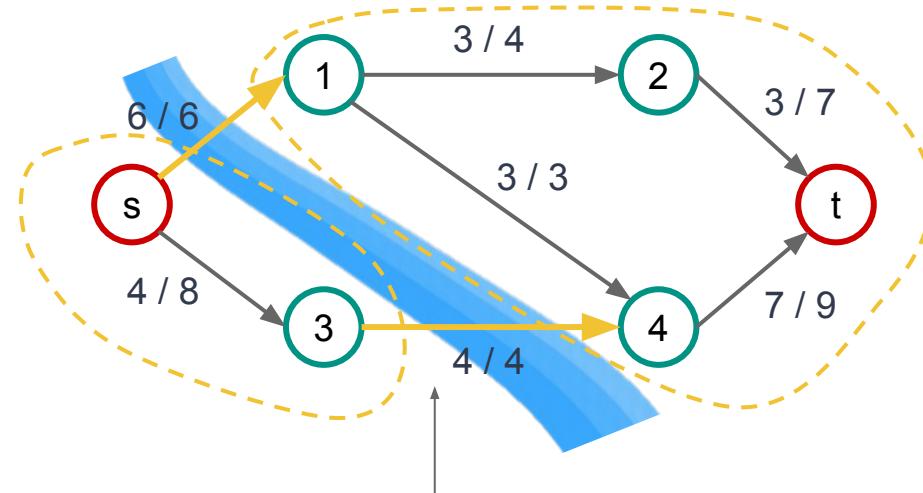
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

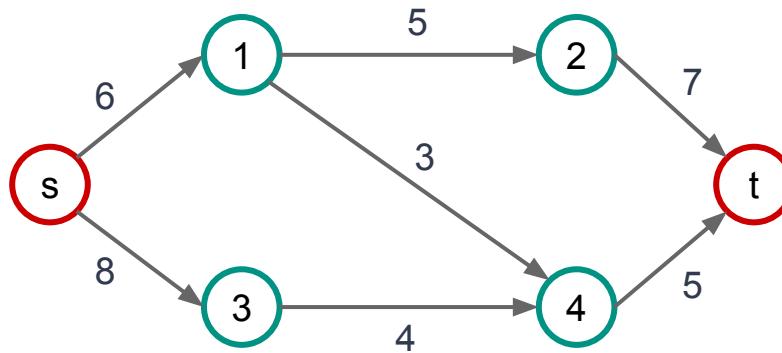


- Singurele arce ("poduri") care trec din regiunea lui  $s$  în cea a lui  $t$  nu mai pot fi folosite pentru a trimite flux (au flux = capacitate)  
⇒ fluxul este maxim
- **s-t tăietură**

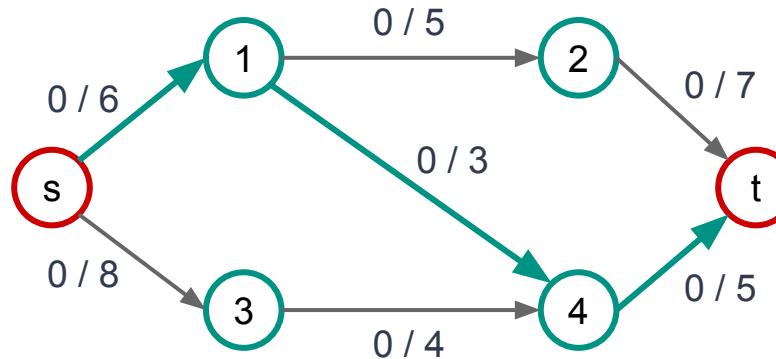
# Alt exemplu



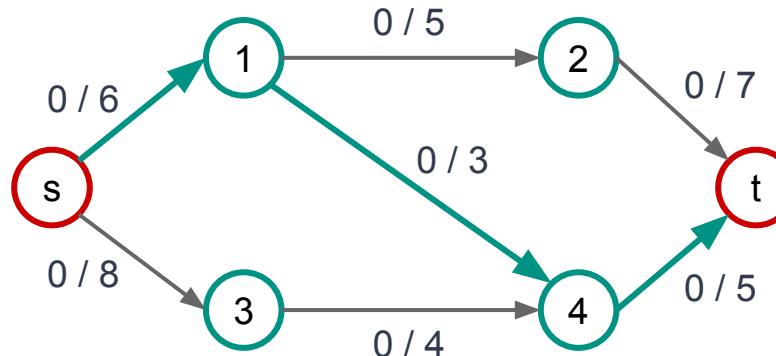
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

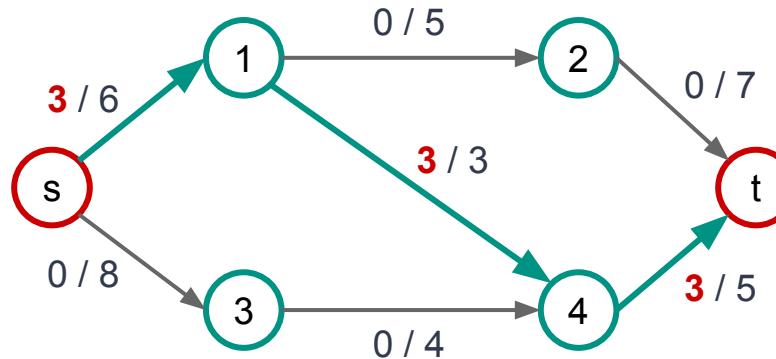


# Fluxuri în rețele de transport

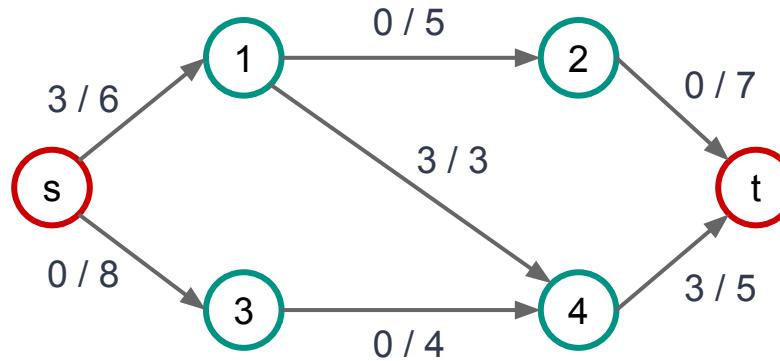


Putem trimite 3 unități de-a lungul întregului drum

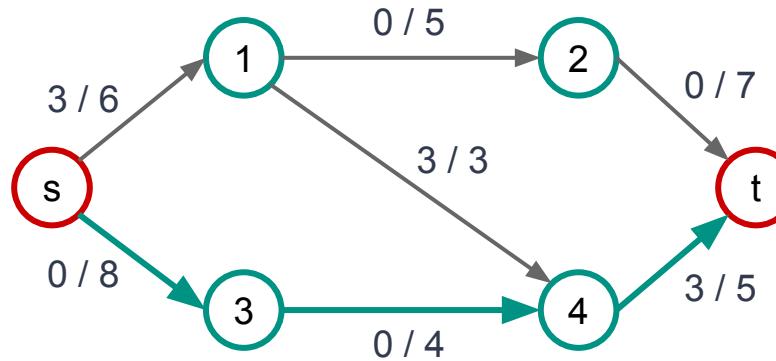
# Fluxuri în rețele de transport



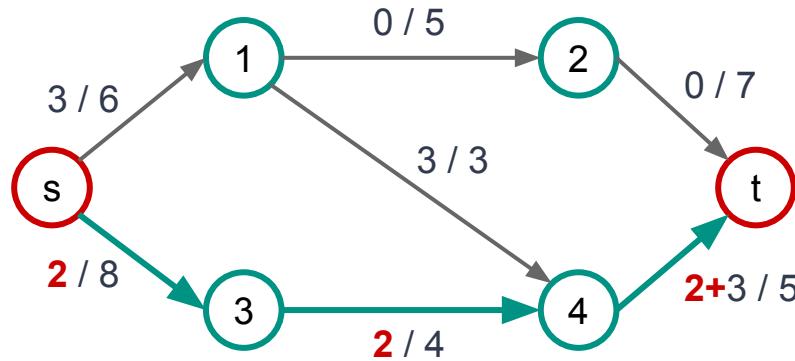
# Fluxuri în rețele de transport



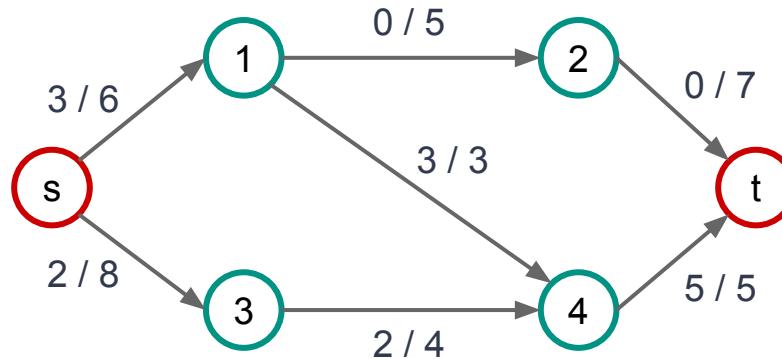
# Fluxuri în rețele de transport



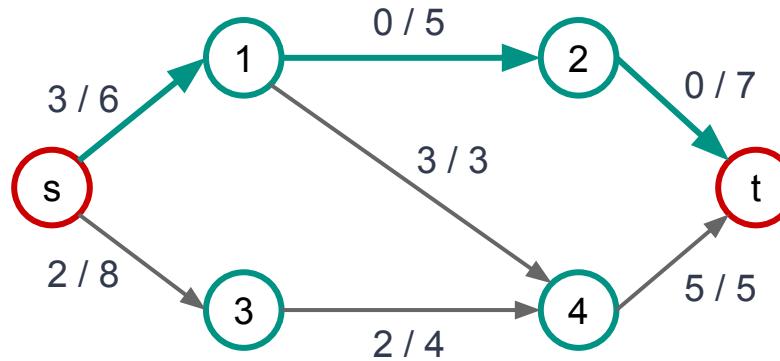
# Fluxuri în rețele de transport



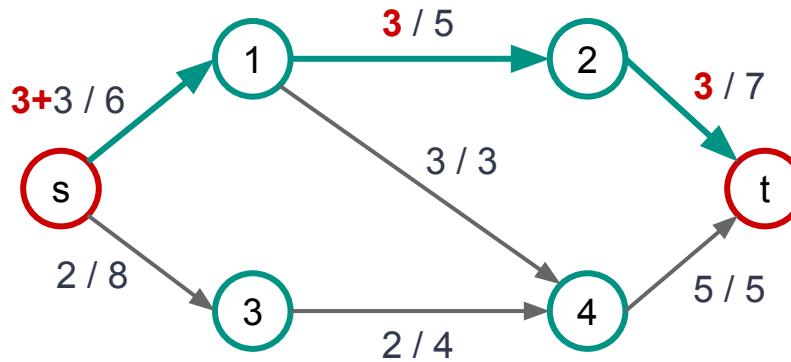
# Fluxuri în rețele de transport



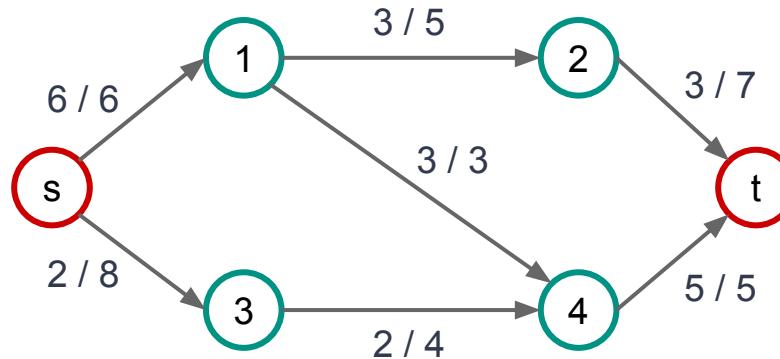
# Fluxuri în rețele de transport



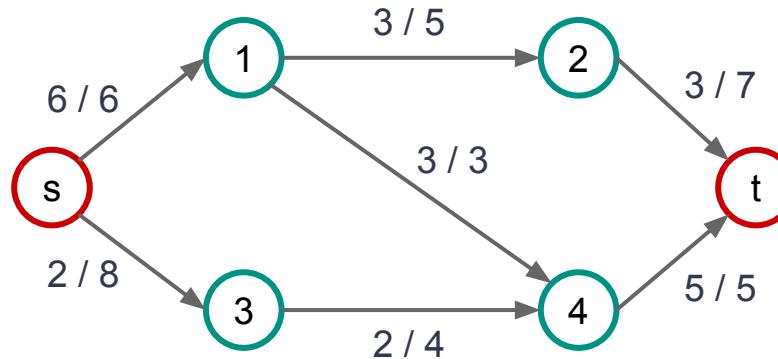
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

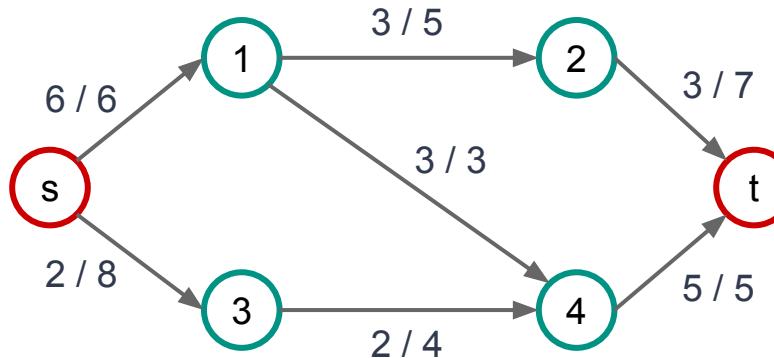


Nu mai există drumuri de la s la t pe care mai putem trimite flux



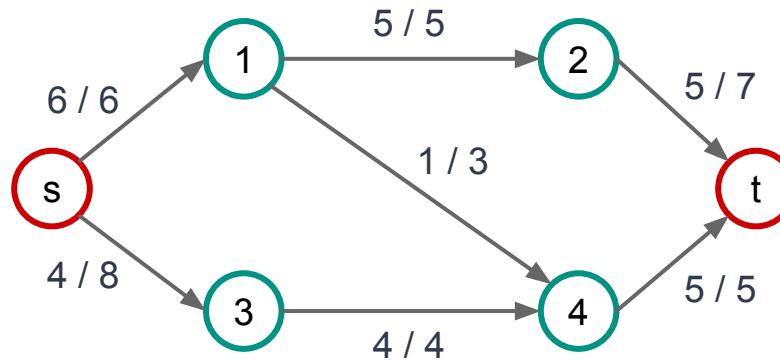
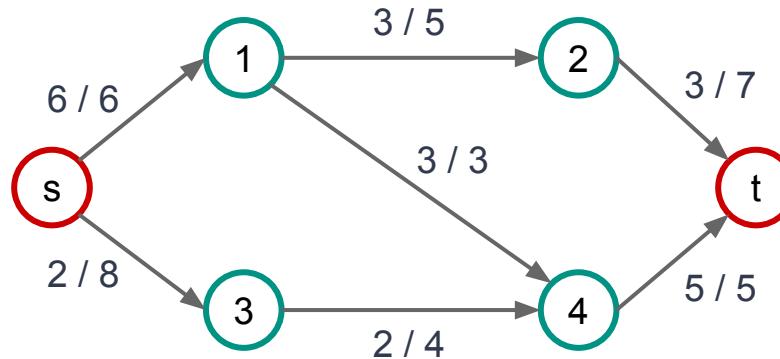
Este maxim fluxul?

# Fluxuri în rețele de transport

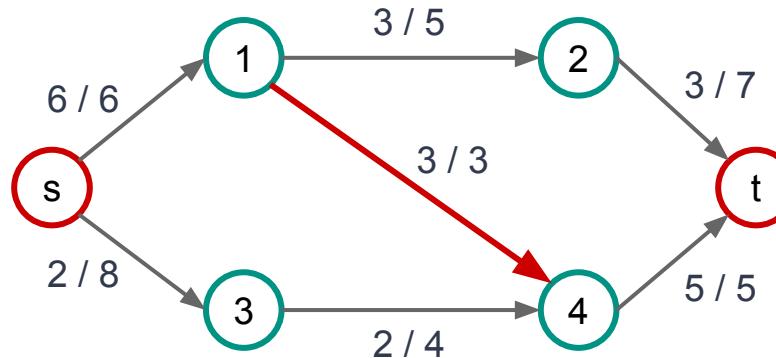


Nu este cantitatea maximă pe care o putem trimite, am trimis greșit pe arcul  $(1,4)$ , adică pe drumul  $[s, 1, 4, t]$

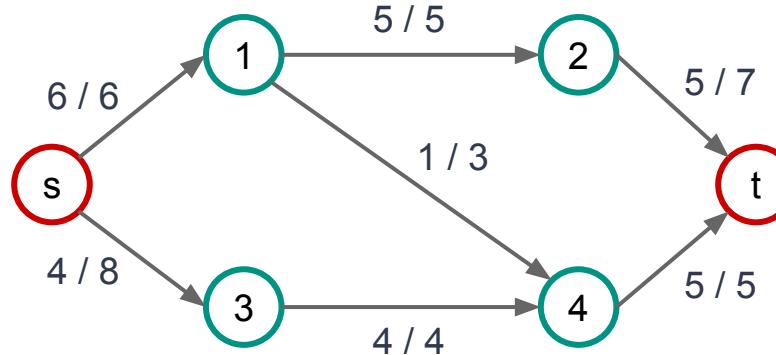
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport

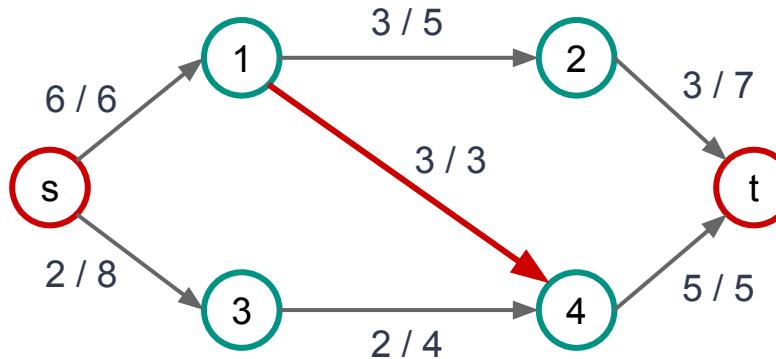


fluxul obținut



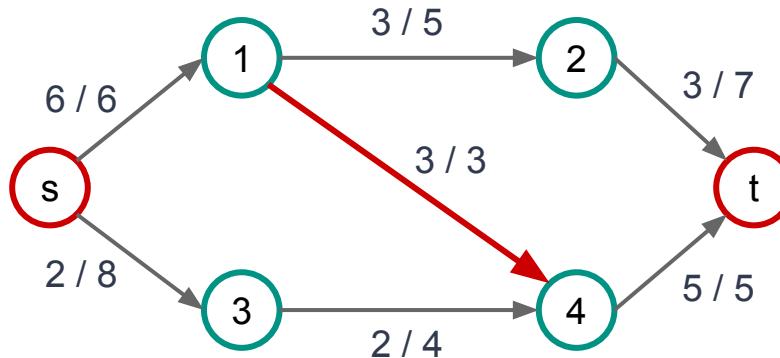
un flux "mai mare"

# Fluxuri în rețele de transport



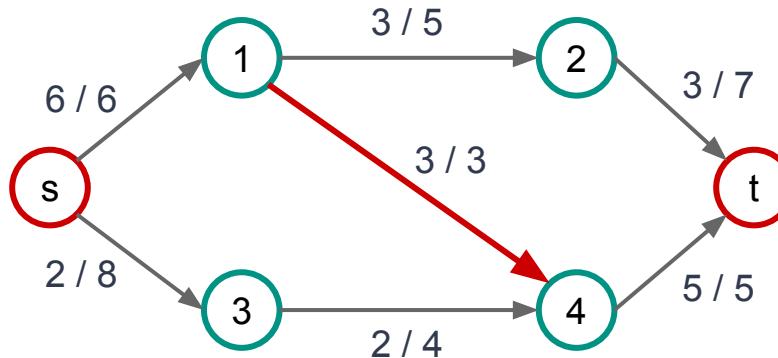
Trebuie să putem **corecta** (să trimitem flux înapoi pe un arc, pentru a fi direcționat prin alte arce către destinație)

# Fluxuri în rețele de transport



Trimitem unități de flux înapoi pe arcul (1, 4)

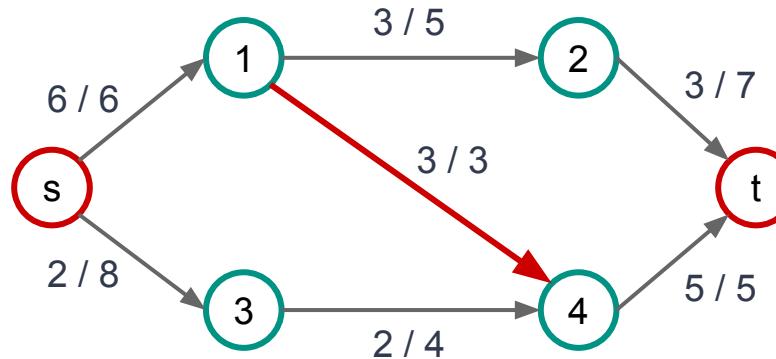
# Fluxuri în rețele de transport



Trimitem unități de flux înapoi pe arcul (1, 4)

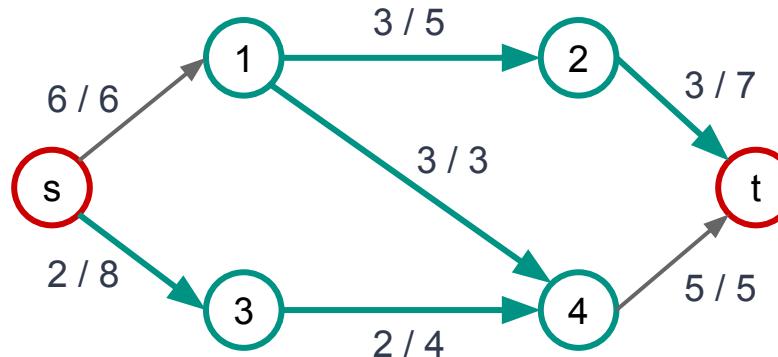
**Corecția trebuie făcută pe un lanț de la s la t, nu doar pe un arc, altfel fluxul (marfa) va rămâne într-un vârf intermediu**

# Fluxuri în rețele de transport



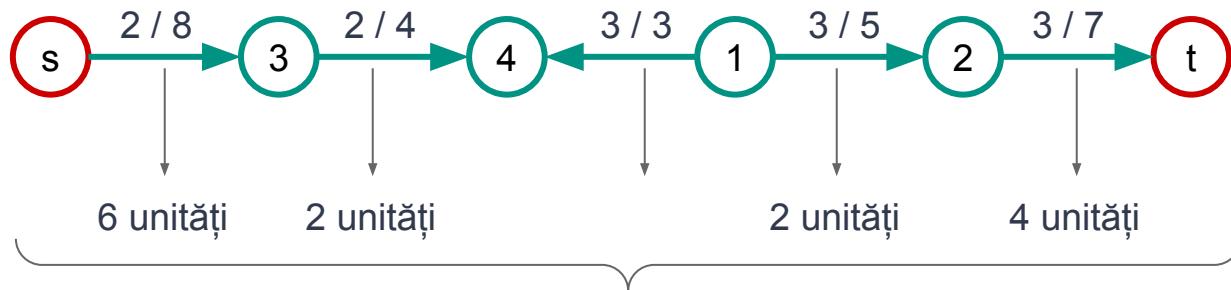
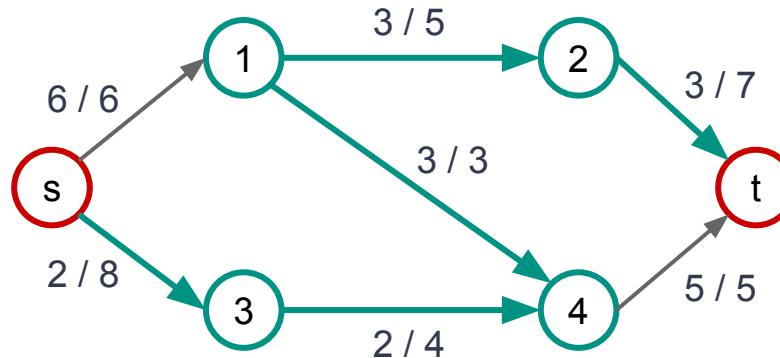
Determinăm un **LANT** (nu drum) de la s la t pe care putem modifica fluxul

# Fluxuri în rețele de transport

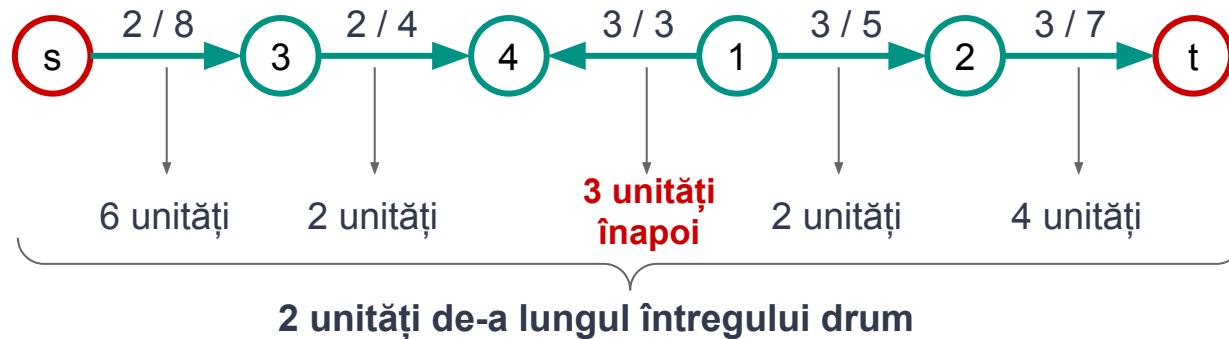
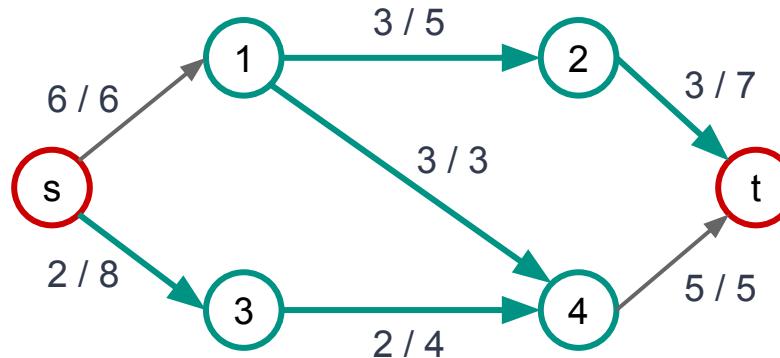


Cu cât putem modifica fluxul pe acest lanț?

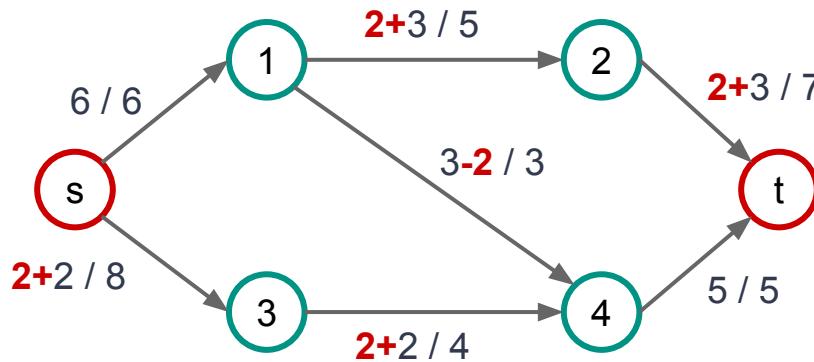
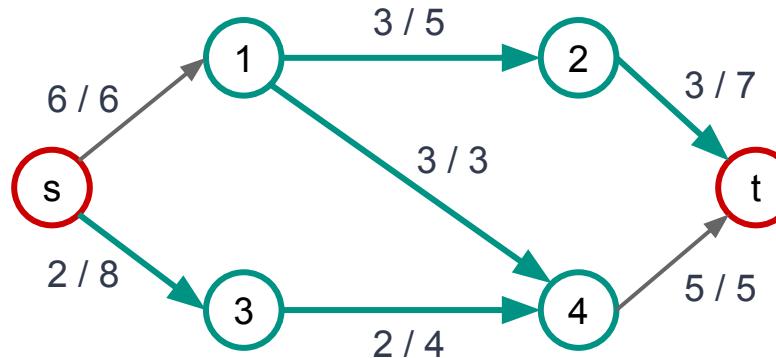
# Fluxuri în rețele de transport



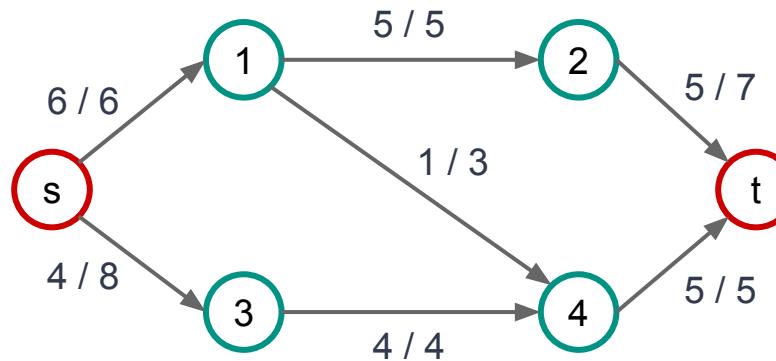
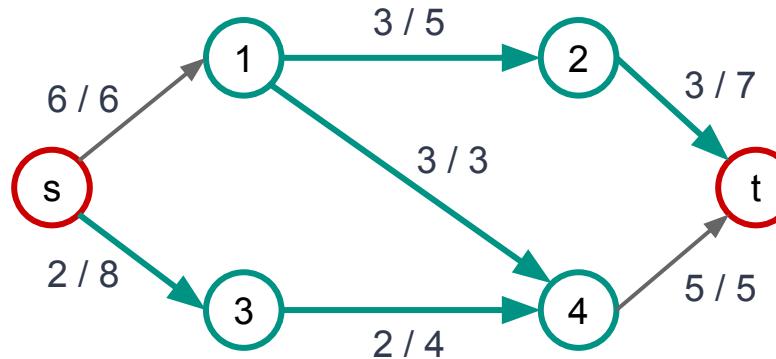
# Fluxuri în rețele de transport



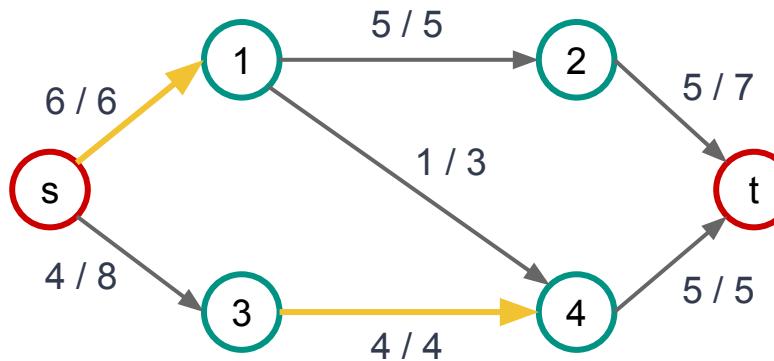
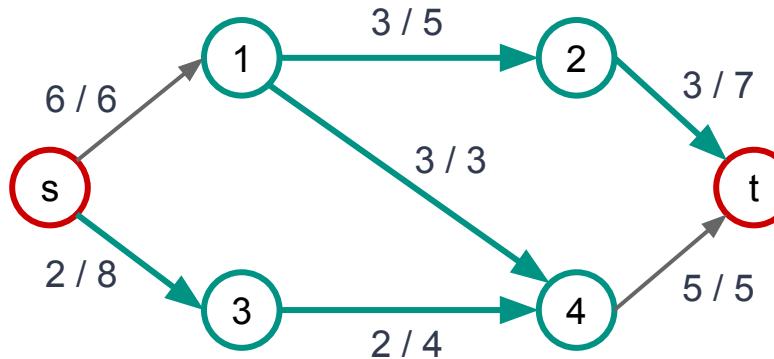
# Fluxuri în rețele de transport



# Fluxuri în rețele de transport



# Fluxuri în rețele de transport



# Definiții

# Fluxuri în rețele de transport

**Rețea de transport**  $N = (G, S, T, I, c)$ , unde

- $G = (V, E)$  - graf orientat cu
  - $V = S \cup I \cup T$

# Fluxuri în rețele de transport

**Rețea de transport**  $N = (G, S, T, I, c)$ , unde

- $G = (V, E)$  - graf orientat cu
  - $V = S \cup I \cup T$
  - $S, I, T$  disjuncte, nevide
  - $S$  - mulțimea surselor (intrărilor)
  - $T$  - mulțimea destinațiilor (ieșirilor)
  - $I$  - mulțimea vârfurilor intermediare

# Fluxuri în rețele de transport

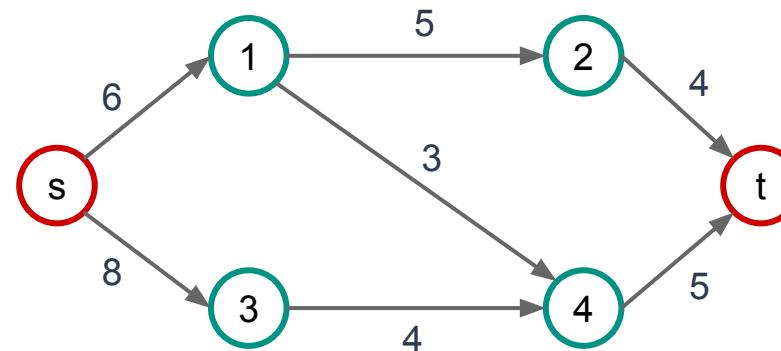
**Rețea de transport**  $N = (G, S, T, I, c)$ , unde

- $G = (V, E)$  - graf orientat cu
  - $V = S \cup I \cup T$
  - $S, I, T$  disjuncte, nevide
  - $S$  - mulțimea surselor (intrărilor)
  - $T$  - mulțimea destinațiilor (ieșirilor)
  - $I$  - mulțimea vârfurilor intermediare
- $c : E \rightarrow \mathbb{N}$  - funcția de **capacitate** (cantitatea maximă care poate fi transportată prin fiecare arc)

# Fluxuri în rețele de transport

## Ipoteze pentru rețeaua N

- $S = \{s\}$  - o singură sursă
- $T = \{t\}$  - o singură destinație
- $d^-(s) = 0$  - în sursă nu intră arce
- $d^+(t) = 0$  - din destinație nu ies arce



# Fluxuri în rețele de transport

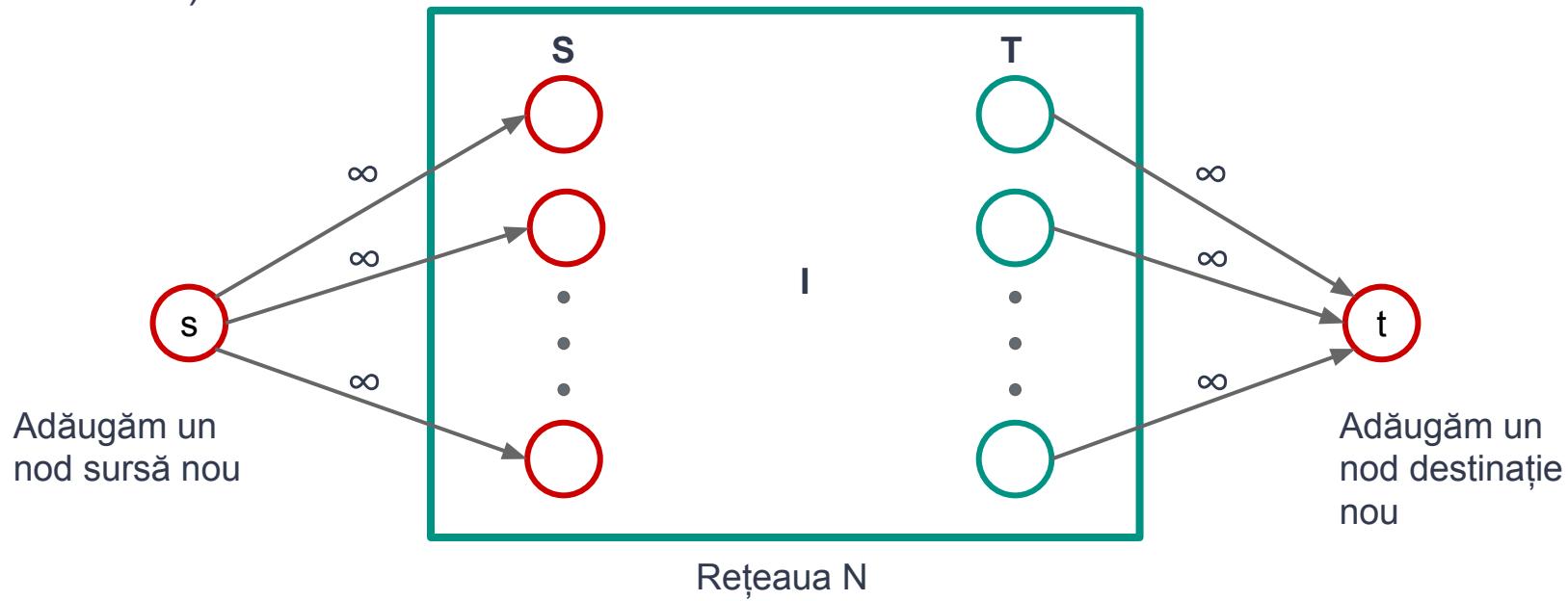
## Ipoteze pentru rețeaua N

- $S = \{s\}$  - o singură sursă
- $T = \{t\}$  - o singură destinație
- $d^-(s) = 0$  - în sursă nu intră arce
- $d^+(t) = 0$  - din destinație nu ies arce

**Ipotezele nu sunt restrictive** - vom arăta că studiul fluxului într-o rețea cu mai multe surse și destinații se poate reduce la studiul fluxului într-o rețea de acest tip.

# Fluxuri în rețele de transport

**Ipotezele nu sunt restrictive** - vom arăta că studiul fluxului într-o rețea cu mai multe surse și destinații se poate reduce la studiul fluxului într-o rețea de acest tip (din punct de vedere al valorii fluxului)



# Fluxuri în rețele de transport

## Ipoteze pentru rețeaua N

- $S = \{s\}$  - o singură sursă
- $T = \{t\}$  - o singură destinație
- $d^-(s) = 0$  - în sursă nu intră arce
- $d^+(t) = 0$  - din destinație nu ies arce
- orice vârf este accesibil din s**

# Fluxuri în rețele de transport

Un **flux** într-o rețea de transport  $N = (G, S, T, I, c)$  este o funcție  $f : E \rightarrow \mathbb{N}$  cu proprietățile:

# Fluxuri în rețele de transport

Un **flux** într-o rețea de transport  $N = (G, S, T, I, c)$  este o funcție  $f : E \rightarrow \mathbb{N}$  cu proprietățile:

- $0 \leq f(e) \leq c(e), \quad \forall e \in E(G)$       *condiția de **mărginire***

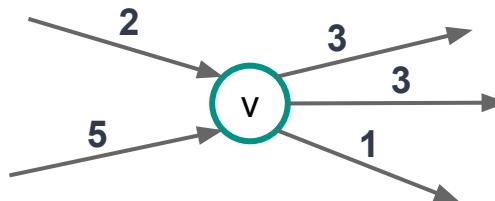
# Fluxuri în rețele de transport

Un **flux** într-o rețea de transport  $N = (G, S, T, I, c)$  este o funcție  $f : E \rightarrow \mathbb{N}$  cu proprietățile:

- $0 \leq f(e) \leq c(e), \quad \forall e \in E(G)$  *condiția de mărginire*
- pentru orice vârf **intermediar**  $v \in I$

$$\sum_{uv \in E} f(uv) = \sum_{vu \in E} f(vu) \quad \text{i} \text{condiția de conservare a fluxului}$$

(fluxul total care intră în  $v$  = fluxul total care iese din  $v$ )



# Fluxuri în rețele de transport

Notății:

- $\bar{X}$
- $f^-(v), f^+(v)$
- $f(X, Y), X, Y \subseteq V$
- $f^+(X), X \subseteq V$

În general, pentru orice funcție  $g : E \rightarrow \mathbb{N}$  **vom folosi notății similare**

# Fluxuri în rețele de transport

Notății:

$$f^+(v) = \sum_{vu \in E} f(vu) \quad = \text{fluxul care ieșe din } v$$

$$f^-(v) = \sum_{uv \in E} f(uv) \quad = \text{fluxul care intră în } v$$

# Fluxuri în rețele de transport

Notății:

$$f^+(v) = \sum_{vu \in E} f(vu) \quad = \text{fluxul care ieșe din } v$$

$$f^-(v) = \sum_{uv \in E} f(uv) \quad = \text{fluxul care intră în } v$$

Condiția de **conservare a fluxului** devine:

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

# Fluxuri în rețele de transport

Pentru  $X, Y \subseteq V$  disjuncte:

$$f(X, Y) = \sum_{\substack{uv \in E \\ u \in X, v \in Y}} f(uv) \quad = \text{fluxul de la } X \text{ la } Y$$

(pe arcele care ies din  $X$  către  $Y$ )

# Fluxuri în rețele de transport

Pentru  $X \subseteq V$  disjuncte:

$$f^+(X) = \sum_{\substack{uv \in E \\ u \in X, v \notin X}} f(uv) \quad = \text{fluxul care ieșe din } X$$

(din vârfurile din  $X$ )

$$f^-(X) = \sum_{\substack{vu \in E \\ u \in X, v \notin X}} f(vu) \quad = \text{fluxul care intră în } X$$

(în nodurile din  $X$ )

# Fluxuri în rețele de transport

Pentru  $X, Y \subseteq V$  disjuncte:

$$f(X, Y) = \sum_{\substack{uv \in E \\ u \in X, v \in Y}} f(uv) \quad = \text{fluxul de la } X \text{ la } Y$$

(pe arcele care ies din  $X$  către  $Y$ )

Avem

$$f^+(X) = f(X, V - X) = f(X, \overline{X})$$

$$f^-(X) = f(\overline{X}, X)$$

# Fluxuri în rețele de transport

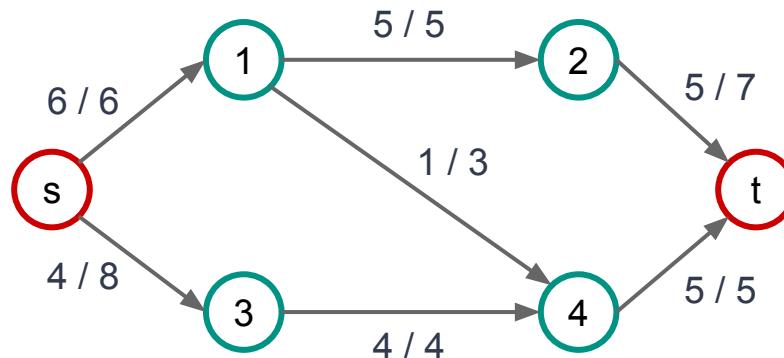
**Valoarea fluxului  $f$  se definește ca fiind**

$$val(f) = f^+(s) = \sum_{su \in E} f(su)$$

# Fluxuri în rețele de transport

Valoarea fluxului  $f$  se definește ca fiind

$$val(f) = f^+(s) = \sum_{su \in E} f(su)$$

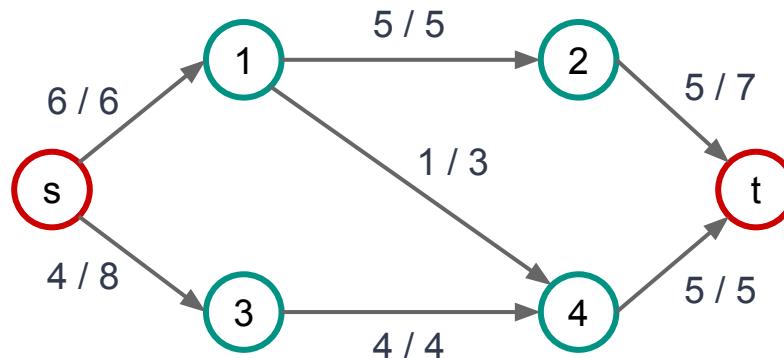


$$val(f) = ?$$

# Fluxuri în rețele de transport

Valoarea fluxului  $f$  se definește ca fiind

$$val(f) = f^+(s) = \sum_{su \in E} f(su)$$



$$val(f) = f(s, 1) + f(s, 3) = 6 + 4 = 10$$

# Fluxuri în rețele de transport

Valoarea fluxului  $f$  se definește ca fiind

$$val(f) = f^+(s)$$

Vom demonstra ulterior că are loc relația

$$val(f) = f^+(s) = f^-(t)$$

# Problema fluxului maxim

Fie  $N$  o rețea.

Un flux  $f^*$  se numește **flux maxim în  $N$**  dacă

$$\text{val}(f^*) = \max \{ \text{val}(f) \mid f \text{ este flux în } N \}$$

# Problema fluxului maxim

Fie  $N$  o rețea.

Un flux  $f^*$  se numește **flux maxim în  $N$**  dacă

$$\text{val}(f^*) = \max \{ \text{val}(f) \mid f \text{ este flux în } N \}$$

**Observație.** Orice rețea admite cel puțin un flux, spre exemplu fluxul vid:

$$f(e) = 0, \quad \forall e \in E$$

# Problema fluxului maxim

Fie  $N$  o rețea.

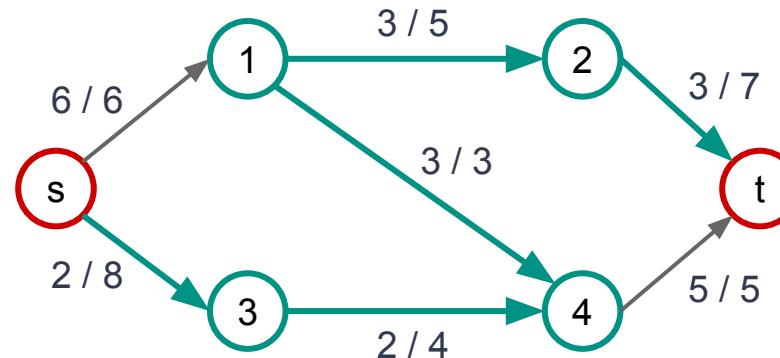
Să se determine  $f^*$  un **flux maxim** în  $N$ .

# Algoritmul Ford-Fulkerson

de determinare a unui flux maxim  
+ a unei tăieturi minime

# Algoritmul Ford-Fulkerson

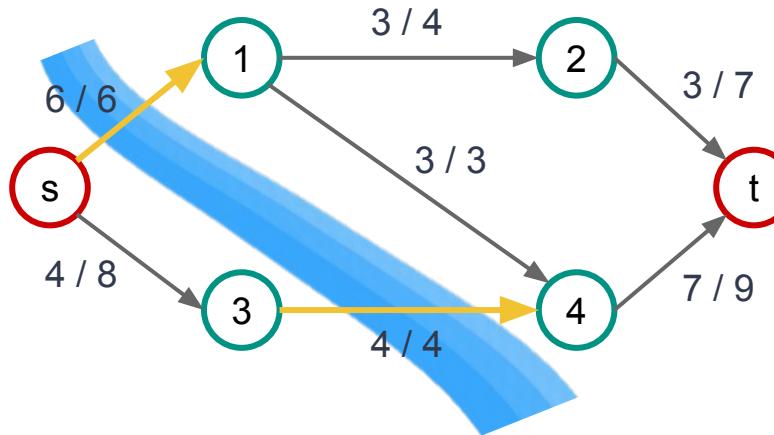
Amintim din exemplele anterioare:



arc în sens invers, putem trimite  
înapoi 3 unități de flux

arc în sens direct, mai putem  
trimite  $5-3=2$  unități de flux

# Algoritmul Ford-Fulkerson



Fluxul este maxim - în mulțimea de arce evidențiată, toate arcele au flux=capacitate și nu putem construi drumuri de la s la t care nu conțin arce din această mulțime (**s-t tăietură**)

# Algoritmul Ford-Fulkerson

Definim noțiunile necesare descrierii și studiului algoritmului:

- **s-t lanț f-nesaturat**
  - arc direct
  - arc invers
  - capacitate reziduală arc, lanț
- Operația de **revizuire a fluxului** de-a lungul unui s-t lanț *f-nesaturat*
- **Tăietură în rețea**
  - capacitatea unei tăieturi

# Algoritmul Ford-Fulkerson

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

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

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

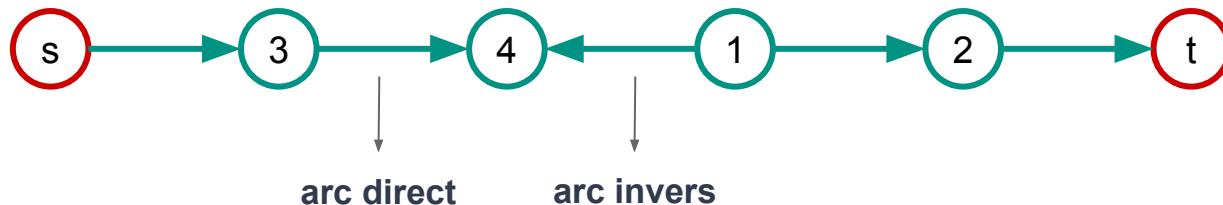
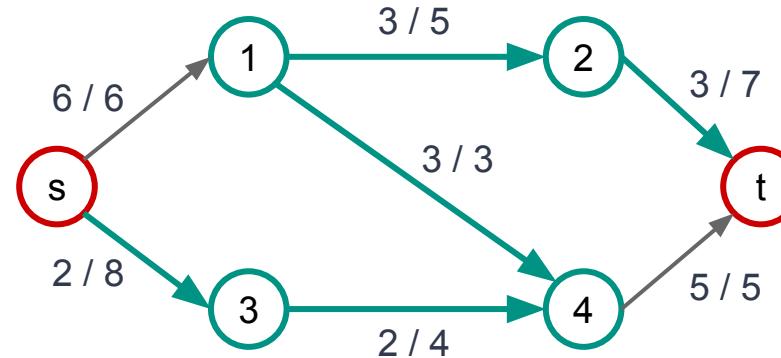
unde arcul  $e_i$  este fie  $v_{i-1}v_i$ , fie  $v_iv_{i-1}$ .

(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
- **Dacă nu există confuzii, vom omite arcele în scrierea lanțului P**

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

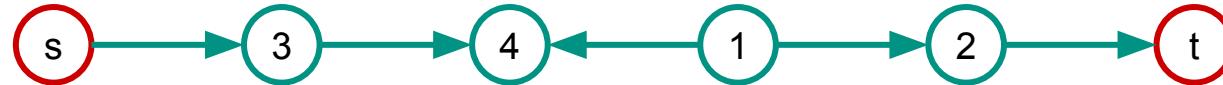
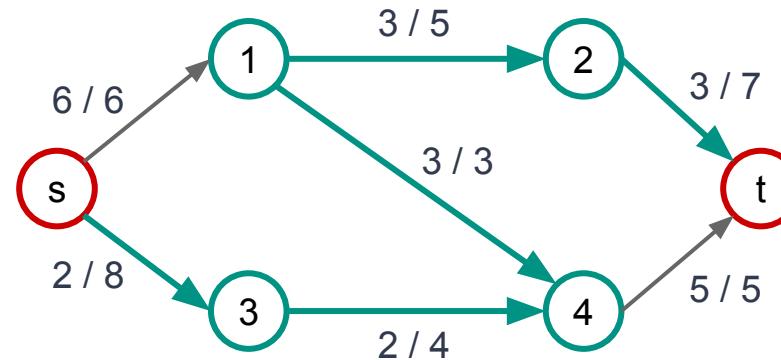
# Algoritmul Ford-Fulkerson



# Algoritmul Ford-Fulkerson

Fie  $N$  rețea,  $f$  flux în  $N$ ,  $P$  un  $s$ - $t$  lanț.

Asociem fiecărui arc  $e$  din  $P$  o pondere, numită **capacitate reziduală** în  $P$ .

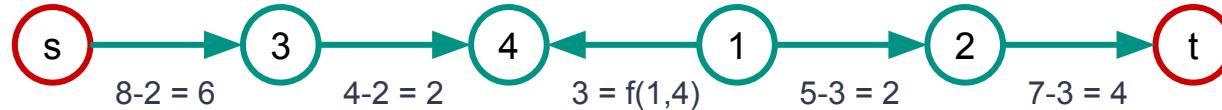
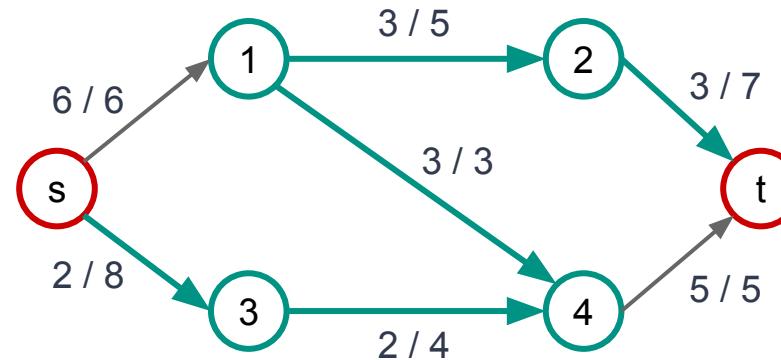


capacități reziduale?

# Algoritmul Ford-Fulkerson

Fie  $N$  rețea,  $f$  flux în  $N$ ,  $P$  un  $s$ - $t$  lanț.

Asociem fiecărui arc  $e$  din  $P$  o pondere, numită **capacitate reziduală** în  $P$ .



capacități reziduale?

# Algoritmul Ford-Fulkerson

Fie  $N$  rețea,  $f$  flux în  $N$ ,  $P$  un s-t lanț.

Asociem fiecărui arc  $e$  din  $P$  o pondere, numită **capacitate reziduală** în  $P$ .

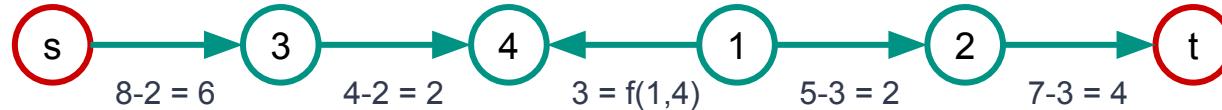
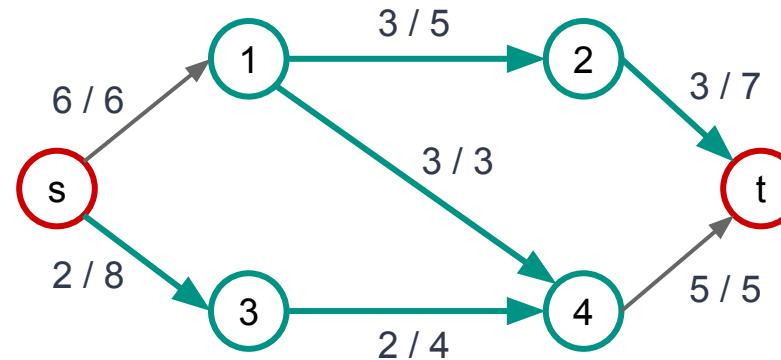
$$i_P(e) = \begin{cases} c(e) - f(e), & \text{daca } e \text{ este arc direct in } P \\ f(e), & \text{daca } e \text{ este arc invers in } P \end{cases}$$

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

# Algoritmul Ford-Fulkerson

Fie  $N$  rețea,  $f$  flux în  $N$ ,  $P$  un  $s$ - $t$  lanț.

Asociem fiecărui arc  $e$  din  $P$  o pondere, numită **capacitate reziduală** în  $P$ .



capacități reziduale?

# Algoritmul Ford-Fulkerson

Capacitatea reziduală a lanțului P



$$i(P) = ?$$

= cu cât putem revizui maxim fluxul de-a lungul lui P

# Algoritmul Ford-Fulkerson

Capacitatea reziduală a lanțului P



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

# Algoritmul Ford-Fulkerson

**Capacitatea reziduală** a lanțului P este

$$i(P) = \min\{i_P(e) | e \in E(P)\}$$

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

P se numește

- f-saturat**,      dacă  $i(P) = 0$
- f-nesaturat**,    dacă  $i(P) \neq 0$

# Algoritmul Ford-Fulkerson

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  $f_P : E \rightarrow \mathbb{N}$ ,

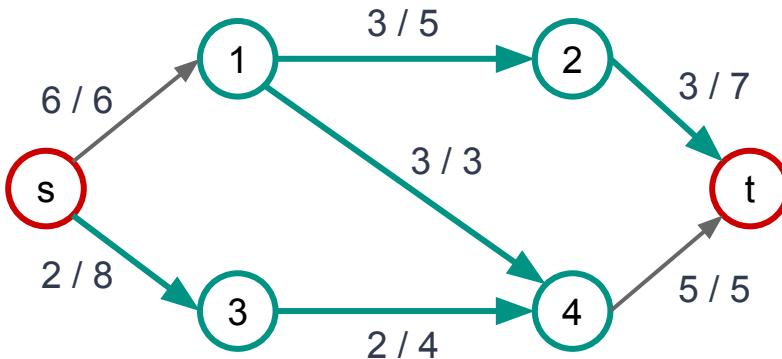
# Algoritmul Ford-Fulkerson

Fie  $N$  - rețea,  $f$  flux în  $N$ ,  $P$  un s-t lanț **f-nesaturat**.

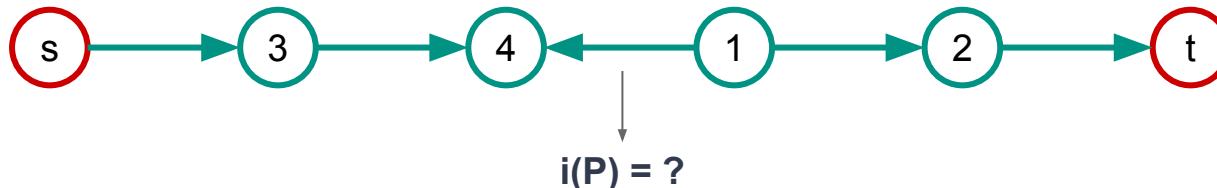
Fluxul revizuit de-a lungul lanțului  $P$  se definește ca fiind  $f_P : E \rightarrow \mathbb{N}$ ,

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

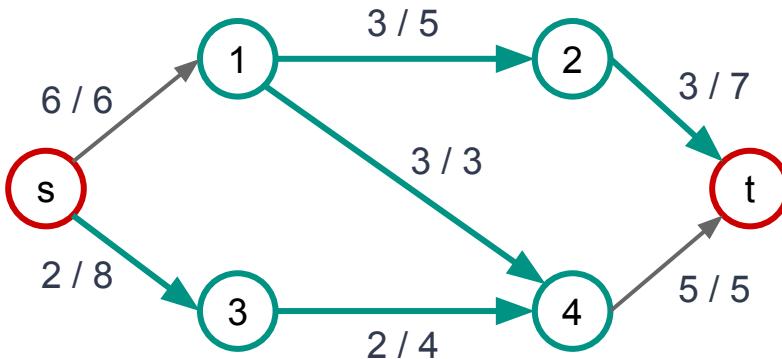
# Algoritmul Ford-Fulkerson



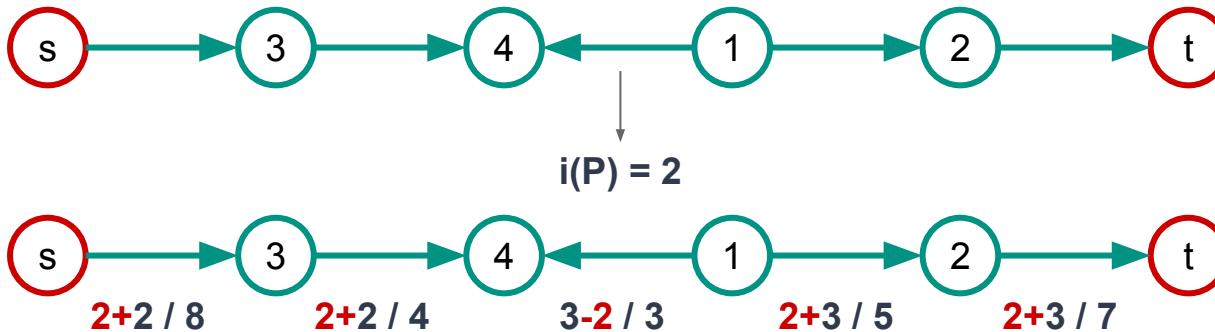
Considerăm s-t  
lanțul P evidențiat



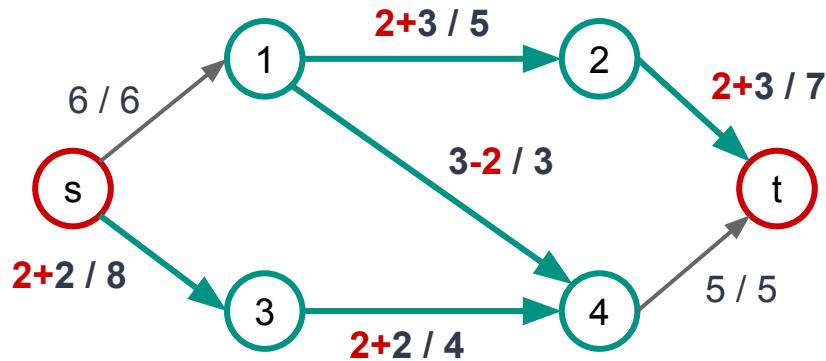
# Algoritmul Ford-Fulkerson



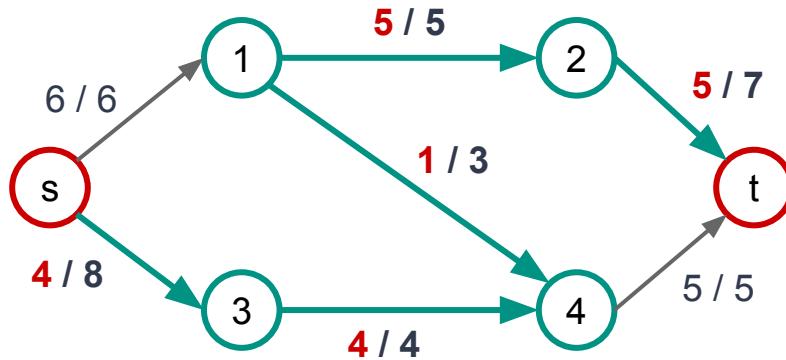
Considerăm s-t  
lanțul P evidențiat  
și revizuim fluxul



# Algoritmul Ford-Fulkerson



# Algoritmul Ford-Fulkerson



Fluxul după revizuirea de-a lungul lanțului P

# Algoritmul Ford-Fulkerson

## Proprietăți ale fluxului revizuit

Fie  $N = (G, \{s\}, \{t\}, l, c)$  o rețea și  $f$  flux în  $N$ .

Fie  $P$  un  $s$ - $t$  lanț  $f$ -nesaturat în  $G$  și  $f_P$  fluxul revizuit de-a lungul lanțului  $P$ .

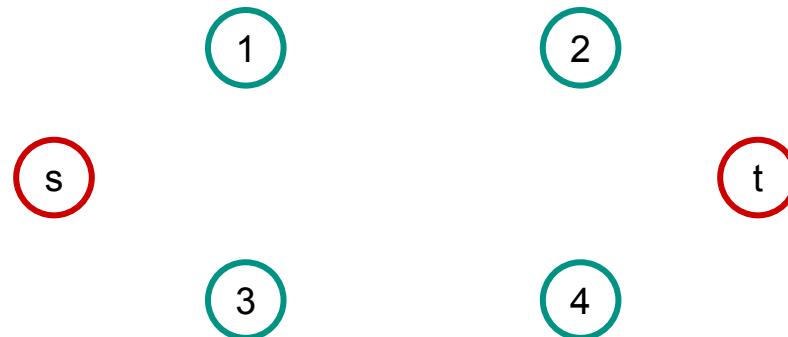
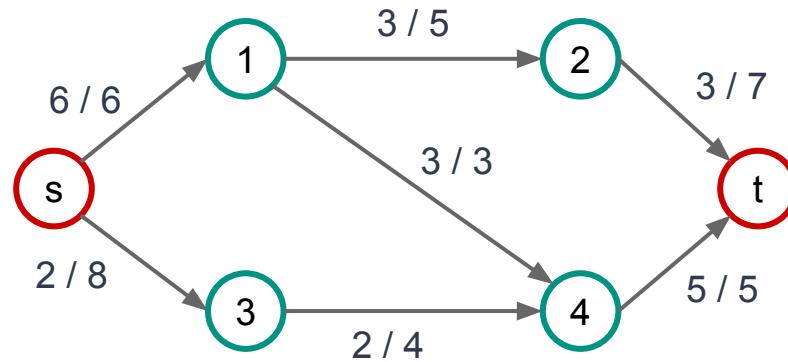
Atunci:

$f_P$  este flux în  $G$

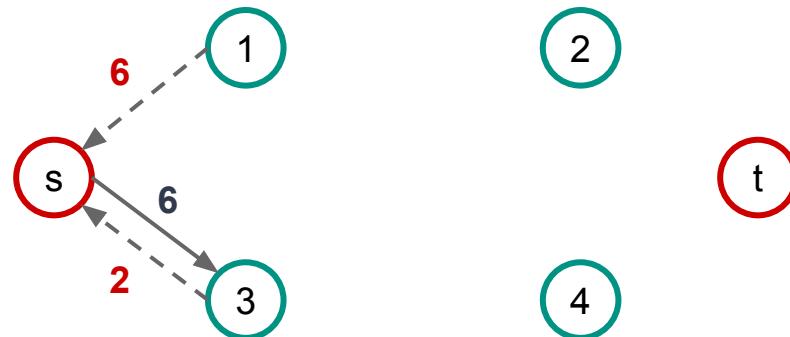
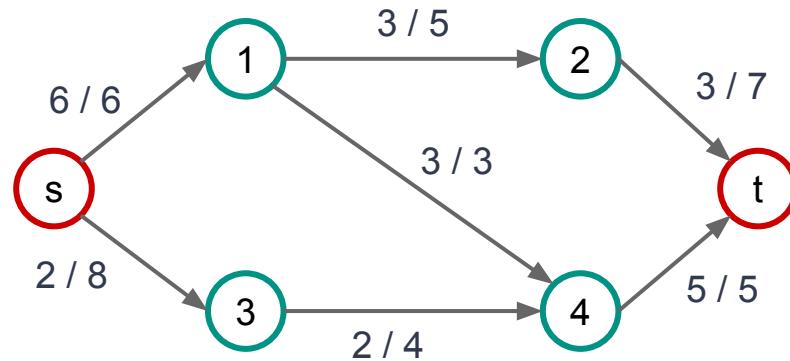
și

$\text{val}(f_P) = \text{val}(f) + i(P) \geq \text{val}(f) + 1$

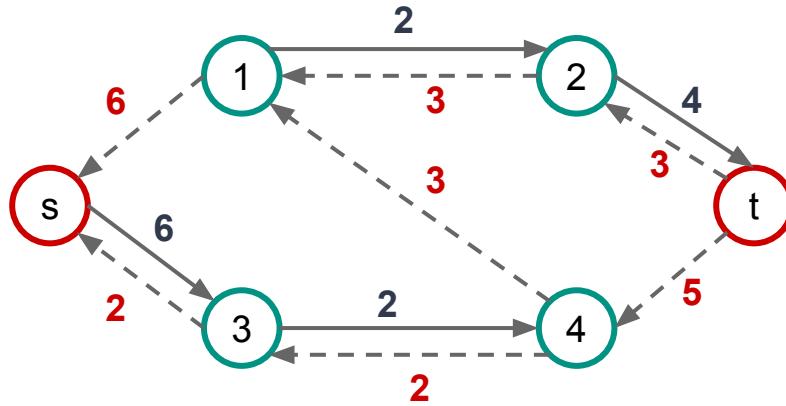
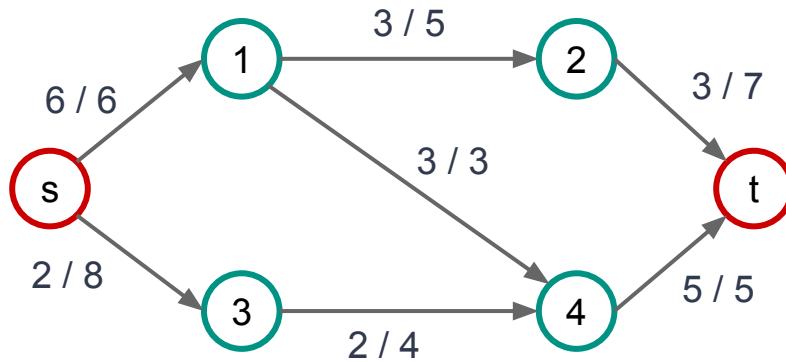
# Graf rezidual



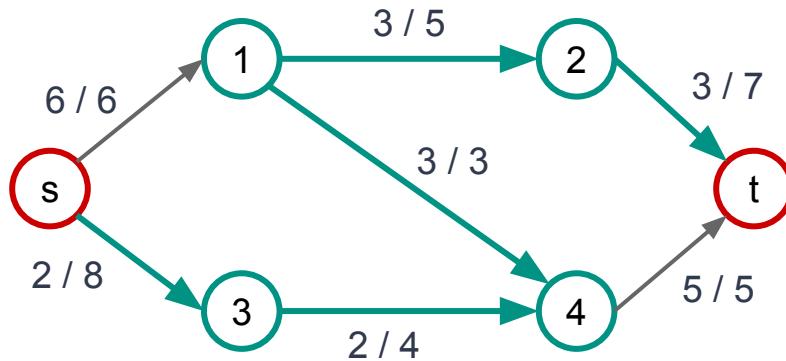
# Graf rezidual



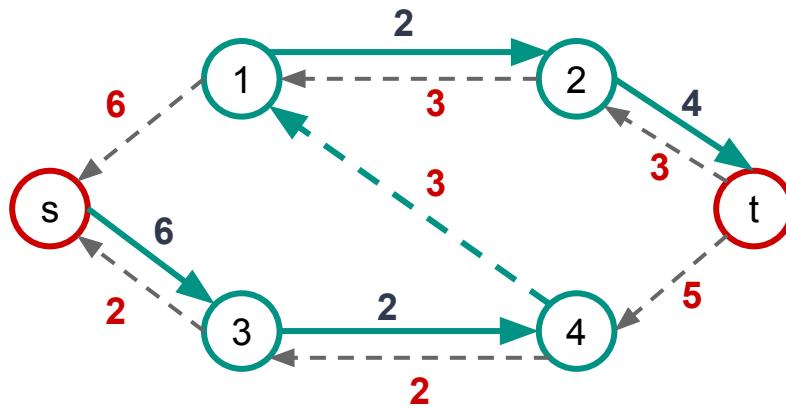
# Graf rezidual



# Graf rezidual



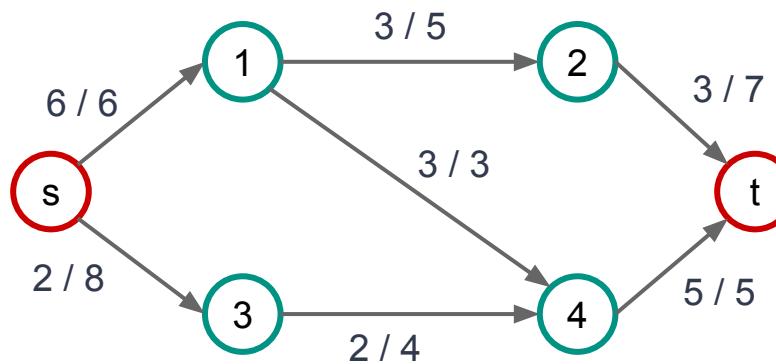
s-t lanț f-nesaturat



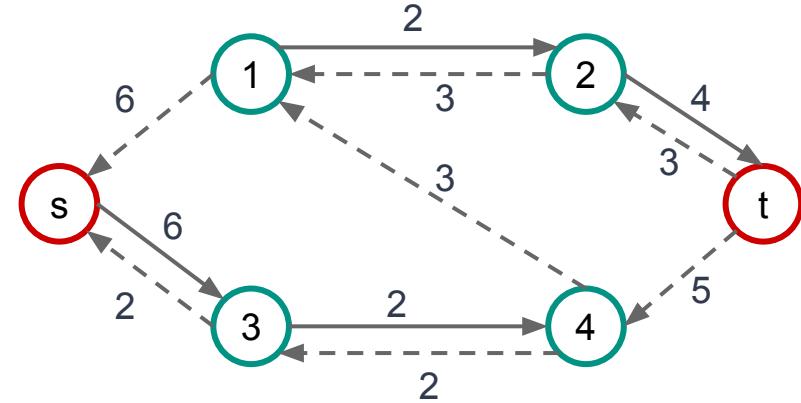
s-t drum în graful rezidual

# Graf rezidual

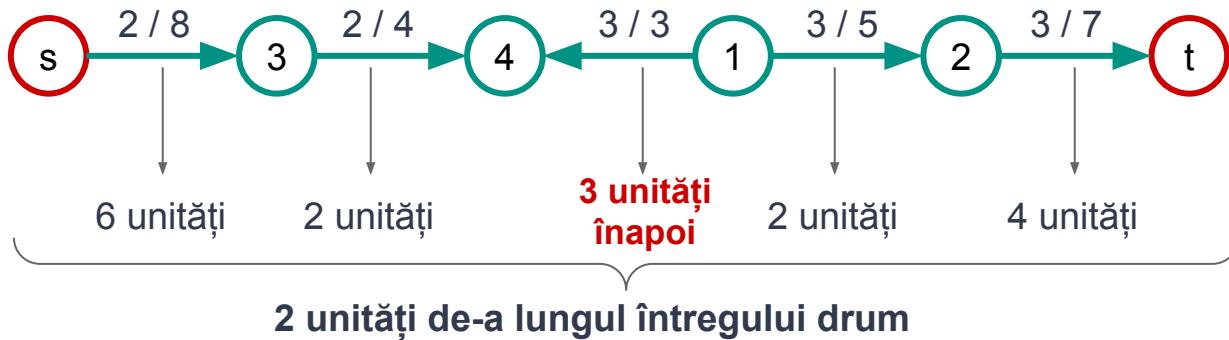
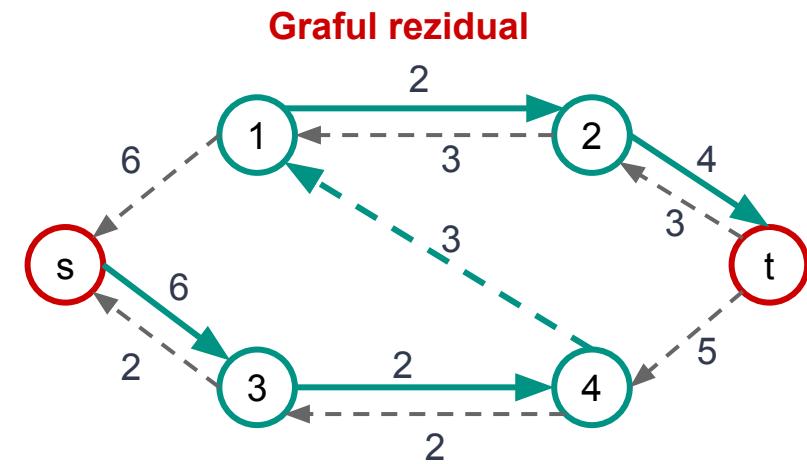
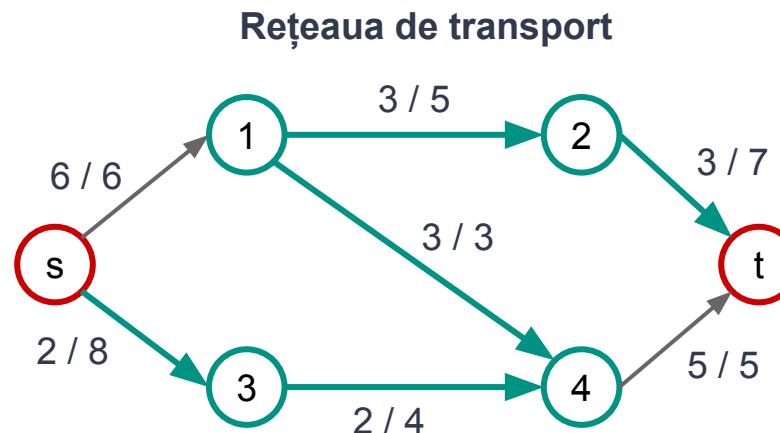
Rețeaua de transport



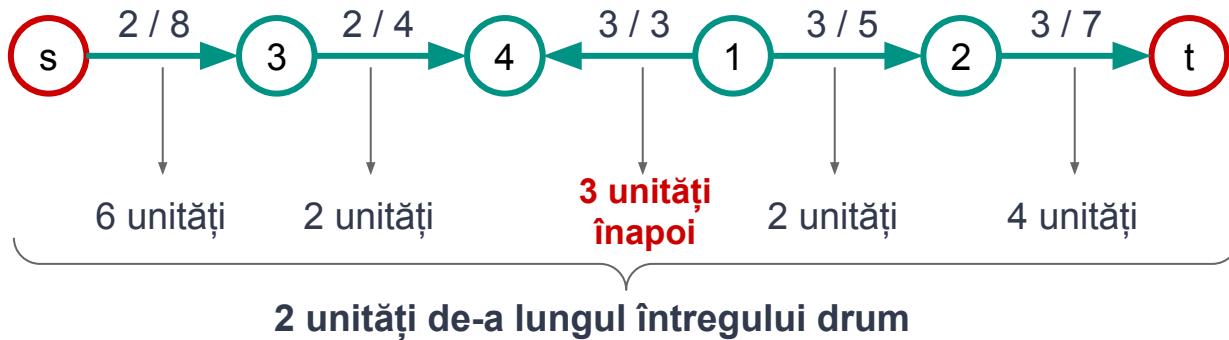
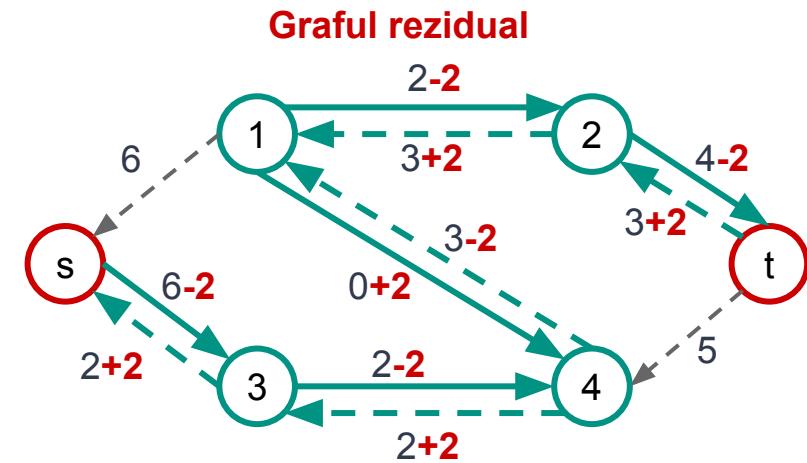
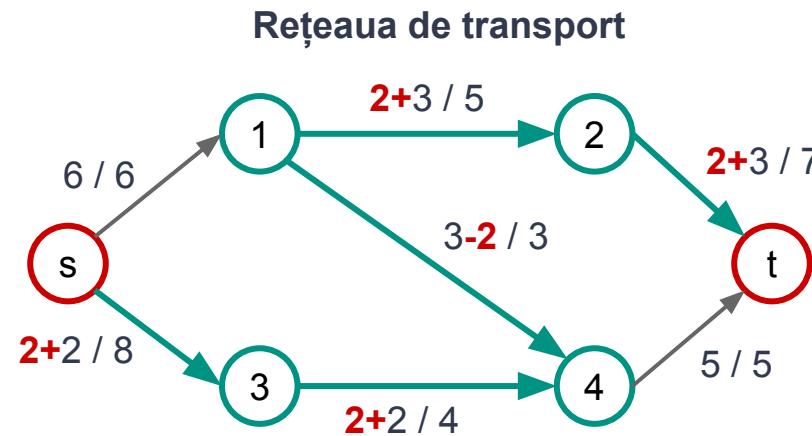
Graful rezidual



# Graf rezidual

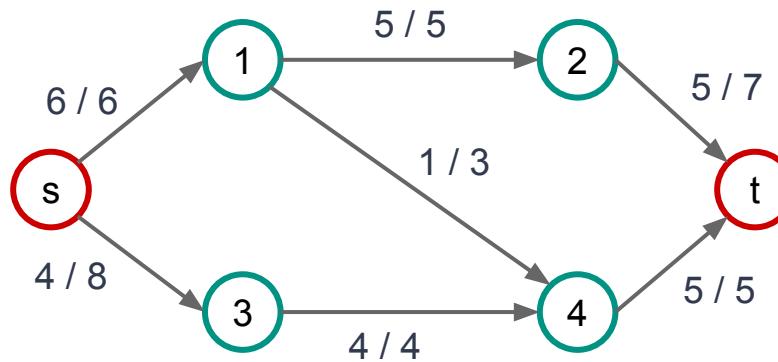


# Graf rezidual

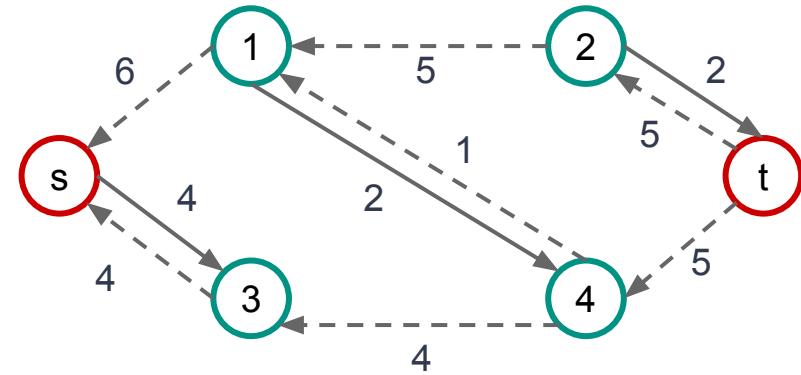


# Graf rezidual

Rețeaua de transport



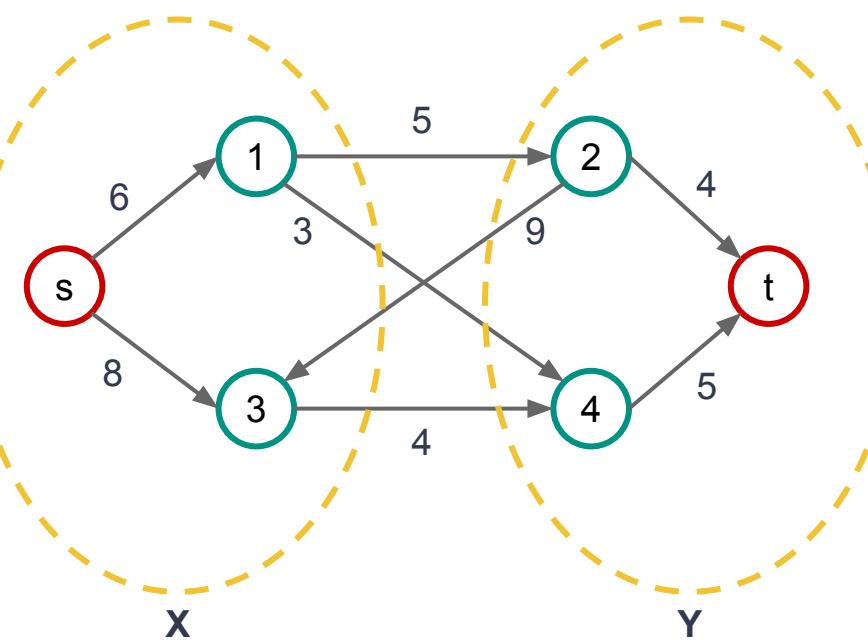
Graful rezidual



# Tăietură în rețea

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

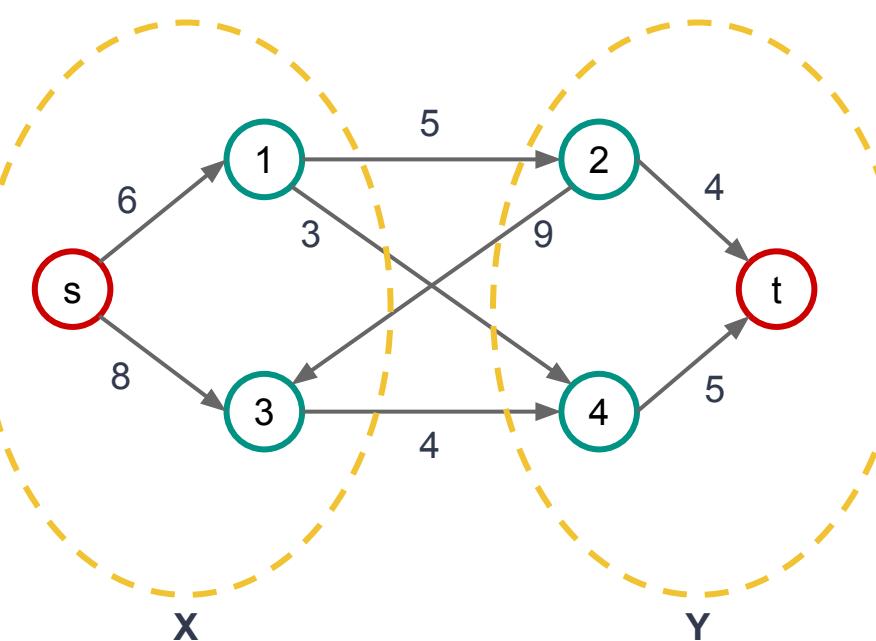
O tăietură  $K = (X, Y)$  în rețea



# Tăietură în rețea

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

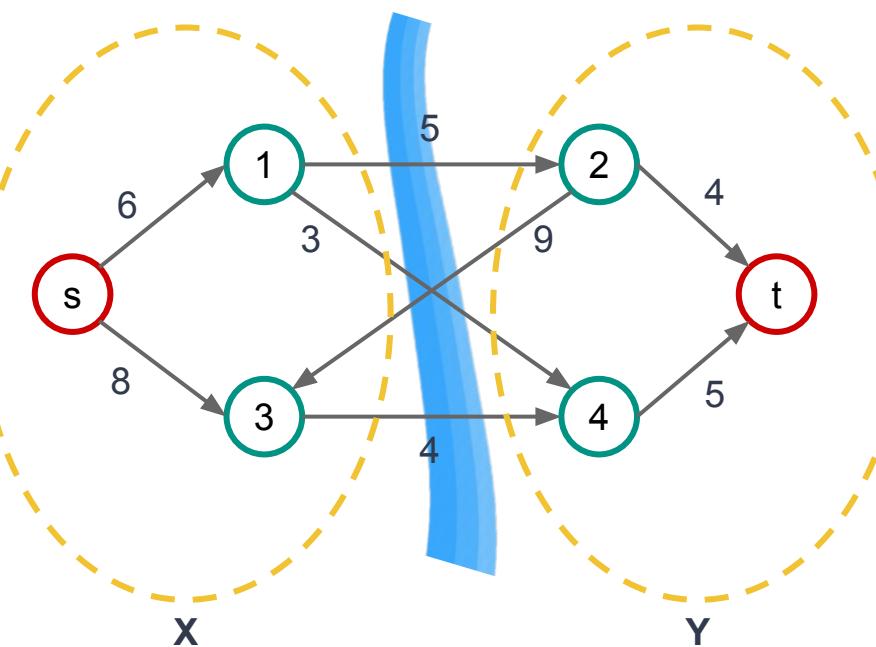
O tăietură  $K = (X, Y)$  în rețea este o (bi)partiție  $(X, Y)$  a multimii vârfurilor  $V$ , astfel încât  $s \in X$  și  $t \in Y$ .



# Tăietură în rețea

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

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



# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

Capacitatea tăieturii  $K = (X, Y)$

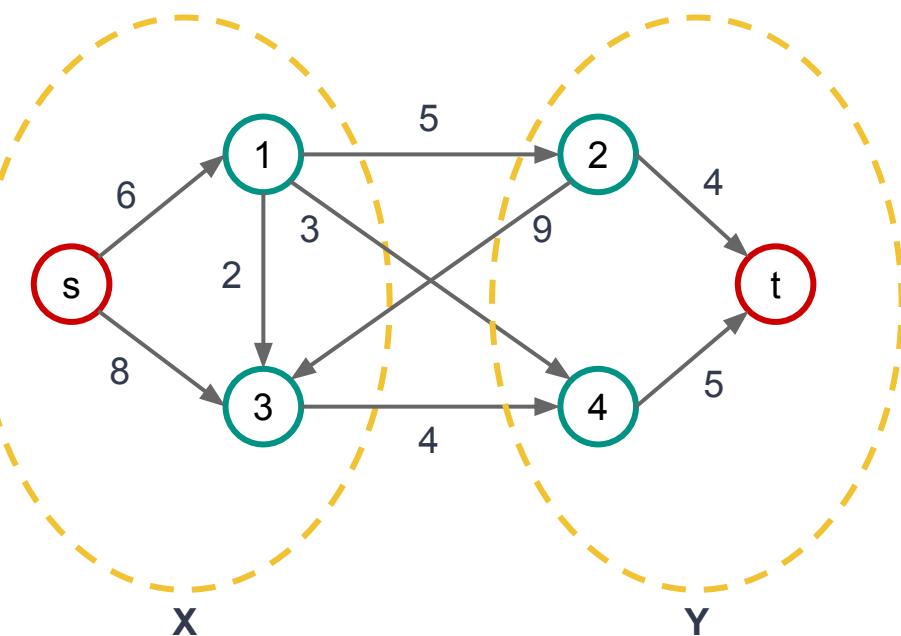
$$c(K) = c(X, Y) = \sum_{\substack{x \in X, y \in Y \\ xy \in E}} c(xy)$$

= suma capacitațiilor arcelor care ies din  $X$  către  $Y$

# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

Capacitatea tăierii  $K = (X, Y)$

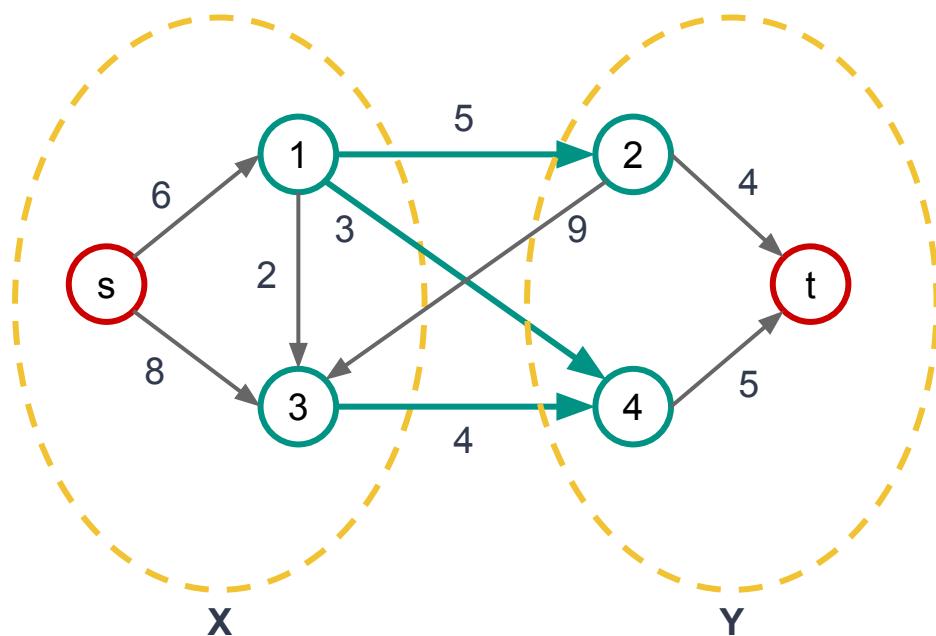


$$c(K) = c(X, Y) = ?$$

# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

Capacitatea tăierii  $K = (X, Y)$



$$c(K) = c(X, Y) = 5 + 3 + 4 = 12$$

# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

Capacitatea tăieturii  $K = (X, Y)$

$$c(K) = \sum_{\substack{x \in X, y \in Y \\ xy \in E}} c(xy)$$

= suma capacitaților arcelor care ies din  $X$  către  $Y$

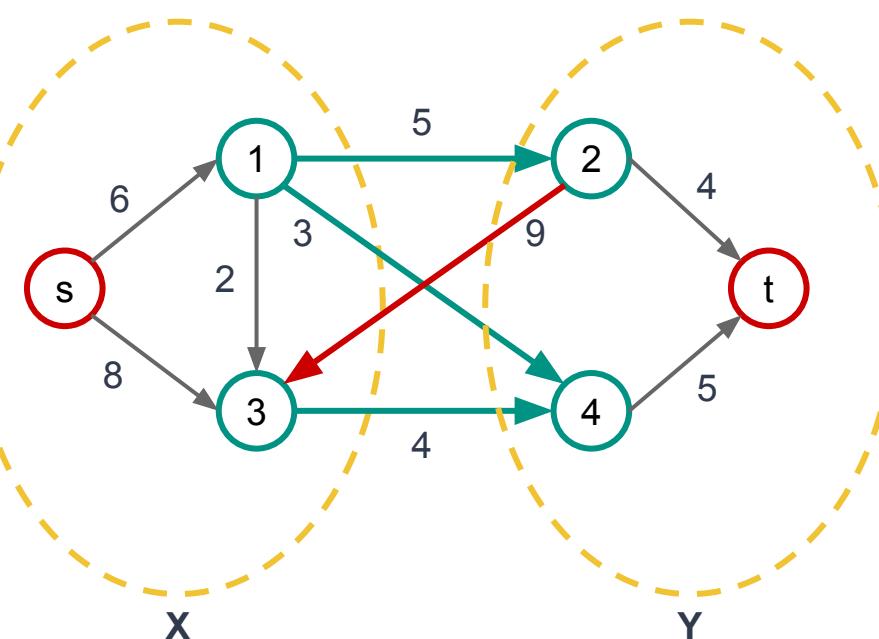
Notăm:

- $E^+(K) =$  mulțimea arcelor de la  $X$  la  $Y$   
 $= \{ xy \in E \mid x \in X, y \in Y \} =$  **arce directe** ale lui  $K$
- $E^-(K) =$  mulțimea arcelor de la  $Y$  la  $X$   
 $= \{ yx \in E \mid x \in X, y \in Y \} =$  **arce inverse** ale lui  $K$

# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

- $xy \in E$  cu  $x \in X, y \in Y$  = **arc direct** al lui  $K$  →
- $yx \in E$  cu  $x \in X, y \in Y$  = **arc invers** al lui  $K$  ←



# Tăietură în rețea

Fie  $K = (X, Y)$  o tăietură.

Notăm:

- $E^+(K) =$  mulțimea arcelor de la  $X$  la  $Y$   
 $= \{ xy \in E \mid x \in X, y \in Y \} =$  **arce directe** ale lui  $K$
- $E^-(K) =$  mulțimea arcelor de la  $Y$  la  $X$   
 $= \{ yx \in E \mid x \in X, y \in Y \} =$  **arce inverse** ale lui  $K$

Atunci avem

$$c(K) = c(E^+(K))$$

# Tăietură minimă

Fie  $N$  o rețea.

O tăietură  $\tilde{K}$  se numește **tăietură minimă în  $N$**  dacă

$$c(\tilde{K}) = \min \{ c(K) \mid K \text{ este tăietură în } N \}$$

# Tăietură minimă

Vom demonstra

$$val(f) \leq c(K)$$

Dacă avem egalitate  $\Rightarrow$  f flux maxim, K tăietură minimă

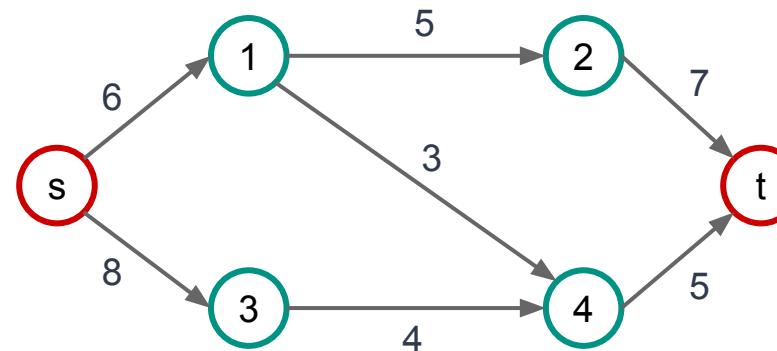
# Tăietură minimă - Aplicații

Determinarea unui flux maxim  $\Rightarrow$  determinarea unei tăieturi minime

## Aplicații

- Arce = poduri
- Capacitate = costul dărâmării podului

**Ce poduri trebuie dărâmate a.î. teritoriul sursă să nu mai fie conectat cu destinația, iar costul distrugerilor să fie minim?**



# Tăietură minimă - Aplicații

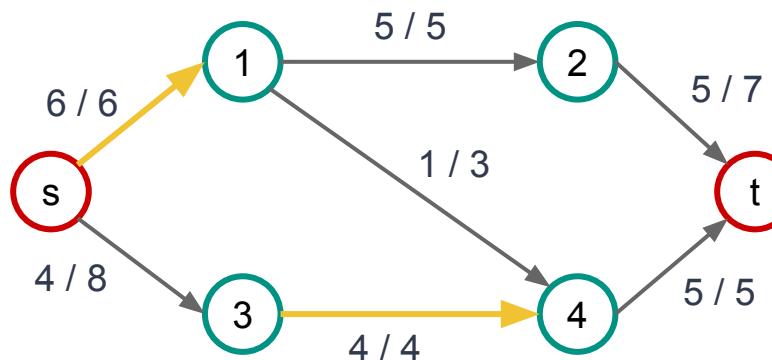
Determinarea unui flux maxim  $\Rightarrow$  determinarea unei tăieturi minime

## Aplicații

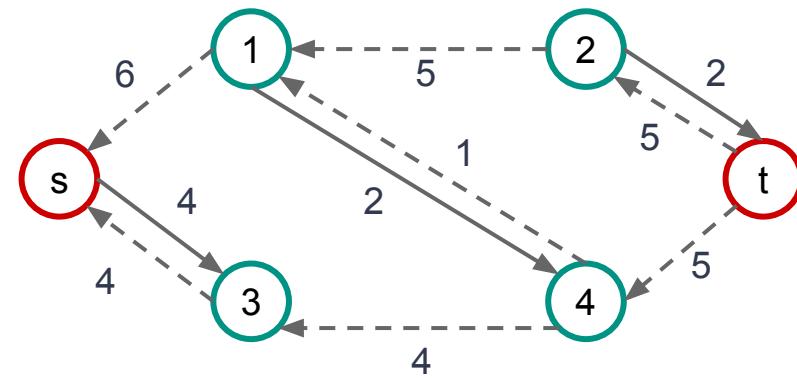
- Fiabilitatea rețelelor**
- Probleme de proiectare, planificare**
- Segmentarea imaginilor**

# Tăietură minimă

Rețeaua de transport



Graful rezidual



s-t tăietură saturată rezidual

↔ nu mai există s-t drum în graful rezidual

↔ **s-t flux maxim**

# Algoritmul Ford-Fulkerson

Pseudocod

# Algoritmul Ford-Fulkerson

## Algoritm generic de determinare a unui flux maxim

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  $G$



# Algoritmul Ford-Fulkerson

## Algoritm generic de determinare a unui flux maxim

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  $G$

- determinăm un astfel de lanț  $P$
-

# Algoritmul Ford-Fulkerson

## Algoritm generic de determinare a unui flux maxim

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  $G$

- determinăm un astfel de lanț  $P$
- revizuieste fluxul  $f$  de-a lungul lanțului  $P$

Returnează  $f$

# Algoritmul Ford-Fulkerson

## Algoritm generic de determinare a unui flux maxim

Pentru a determina și o s-t tăietură minimă, la finalul algoritmului considerăm:

- $X$  = multimea vârfurilor accesibile din  $s$  prin lanțuri f-nesaturate
- $K = (X, V-X)$

# Algoritmul Ford-Fulkerson

Complexitate

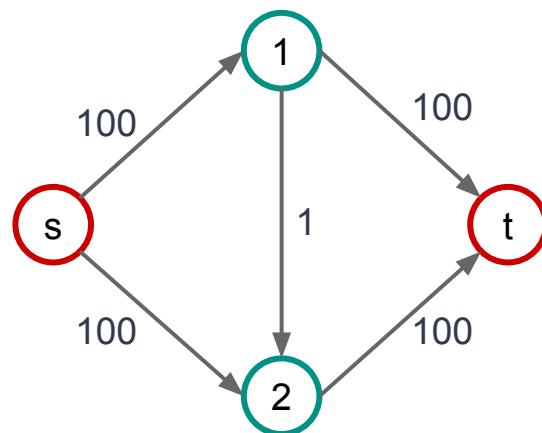
# Algoritmul Ford-Fulkerson - Complexitate



- Algoritmul se termină?**
- De ce este necesară ipoteza că fluxul are valori întregi?**
- Care este numărul maxim de etape?**
  - Cum determinăm un lanț f-nesaturat?
  - Criteriul după care construim lanțul f-nesaturat influențează numărul de etape (iterații "cât timp")?

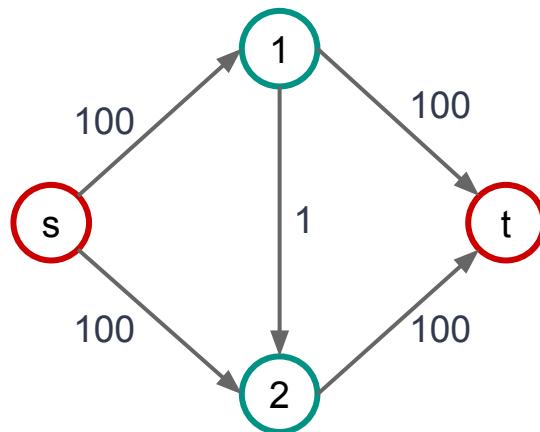
# Algoritmul Ford-Fulkerson - Complexitate

Criteriul după care construim lanțul f-nesaturat influențează numărul de etape (iterații "cât timp")?



# Algoritmul Ford-Fulkerson - Complexitate

Criteriul după care construim lanțul f-nesaturat influențează numărul de etape (iterații "cât timp")?



Pasul 1:  $[s, 1, 2, t] - i(P) = 1$

Pasul 2:  $[s, 2, 1, t] - i(P) = 1$

Pasul 3:  $[s, 1, 2, t] - i(P) = 1$

Pasul 4:  $[s, 2, 1, t] - i(P) = 1$

...

# Algoritmul Ford-Fulkerson - Complexitate

**Complexitate**

# Algoritmul Ford-Fulkerson - Complexitate

## Complexitate

□  $O(mL)$ , unde

$$L = \text{capacitatea minimă a unei căi} \leq \sum_{su \in E} c(su)$$

□  $O(nmC)$ , unde

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

# Algoritmul Ford-Fulkerson



**Cum determinăm un lanț f-nesaturat?**

# Algoritmul Ford-Fulkerson



- Spre exemplu, prin parcurgerea grafului, pornind din vârful  $s$  și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanțurile construite prin parcurgere, memorate cu vectorul  $tata$ )  
**= s-t drum în graful rezidual**

# Algoritmul Ford-Fulkerson



- Spre exemplu, prin parcurgerea grafului, pornind din vârful  $s$  și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanțurile construite prin parcurgere, memorate cu vectorul tata)

Parcugerea BF ⇒

determinăm s-t lanțuri f-nesaturate de **lungime minimă**

⇒ **Algoritmul EDMONDS-KARP**

= Ford-Fulkerson, în care lanțul  $P$  ales la un pas are lungime minimă

# Algoritmul Ford-Fulkerson



- **Spre exemplu, prin parcurgerea grafului, pornind din vârful s și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanturile construite prin parcurgere, memorate cu vectorul tata)**
- Alte criterii de construcție lanț  $\Rightarrow$  alți algoritmi**

# Algoritmul Ford-Fulkerson - Complexitate

**Complexitate  $O(mL)$ , unde**

$$L = \sum_{su \in E} c(su)$$

# Algoritmul Ford-Fulkerson - Complexitate

Complexitate  $O(mC)$ , unde

$$C = c(\{s\}, V-\{s\}) = c^+(s)$$

# Algoritmul Ford-Fulkerson

Corectitudine

# Algoritmul Ford-Fulkerson



- Fluxul determinat de algoritm are valoarea maximă, sau putem determina un flux de valoare mai mare, prin alte metode?

Trebuie să arătăm că

∅ s-t lanț f-nesaturat  $\Rightarrow$  f flux maxim

# Algoritmul Ford-Fulkerson

Vom demonstra că

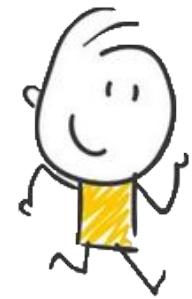
- $\text{val}(f) \leq c(K)$  pentru orice  $f$  flux,  $K$  tăietură



# Algoritmul Ford-Fulkerson

Vom demonstra că

- $\text{val}(f) \leq c(K)$  pentru orice  $f$  flux,  $K$  tăietură
- $\nexists$  s-t lanț f-nesaturat  $\Rightarrow \exists K$  cu  $\text{val}(f) = c(K) \Rightarrow f$  flux maxim



# Implementarea algoritmului Ford-Fulkerson

# Algoritmul Ford-Fulkerson



**Cum determinăm un lanț f-nesaturat?**

# Algoritmul Ford-Fulkerson



- Spre exemplu, prin parcurgerea grafului, pornind din vârful  $s$  și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanțurile construite prin parcurgere, memorate cu vectorul  $tata$ )  
**= s-t drum în graful rezidual**

# Algoritmul Ford-Fulkerson



- Spre exemplu, prin parcurgerea grafului, pornind din vârful s și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanțurile construite prin parcurgere, memorate cu vectorul tata)

Parcugerea BF ⇒

determinăm s-t lanțuri f-nesaturate de **lungime minimă**

⇒ **Algoritmul EDMONDS-KARP**

= Ford-Fulkerson, în care lanțul P ales la un pas are lungime minimă

# Algoritmul Ford-Fulkerson



- **Spre exemplu, prin parcurgerea grafului, pornind din vârful s și considerând doar arce cu capacitatea reziduală pozitivă (în raport cu lanturile construite prin parcurgere, memorate cu vectorul tata)**
- Alte criterii de construcție lanț  $\Rightarrow$  alți algoritmi**

# Algoritmul Edmonds-Karp



# Implementare

## Schema:

initializează\_flux\_nul()

cât timp (construiește\_s-t\_lanț\_nesat\_BF() == true) execută

revizuește\_flux\_lanț()

afisează\_flux()

# Implementare

## Schema:

initializează\_flux\_nul()

cât timp (construiește\_s-t\_lanț\_nesat\_BF() == true) execută

revizuește\_flux\_lanț()

afisează\_flux()

**Amintim:** a determina un s-t lanț nesaturat folosind BF în G  $\Leftrightarrow$  a determina un s-t drum folosind BF în graful rezidual  $G_f$

# Varianta 1 de implementare

revizuirea fluxului folosind s-t lanțuri din G  
(fără a folosi graful rezidual)

# Implementare - Varianta 1

**construieste\_s-t\_lant\_nesat\_BF()** - construiește un s-t lanț nesaturat, prin parcurgerea BF din s

- sunt considerate în parcursere **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)

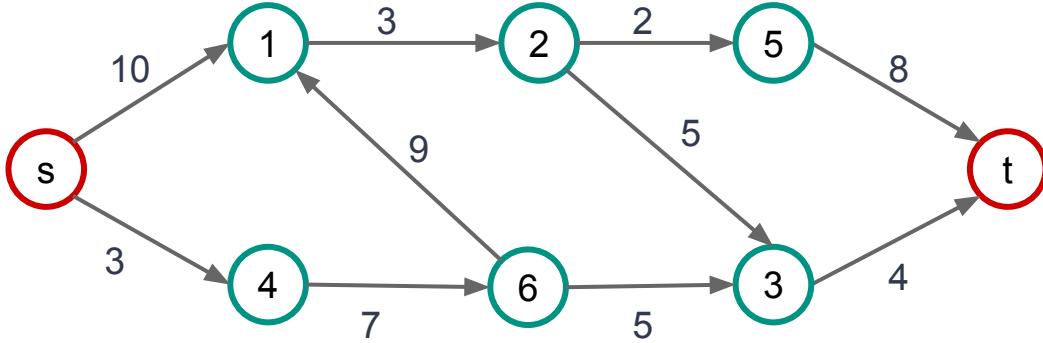
# Implementare - Varianta 1

## revizuește\_flux\_lanț()

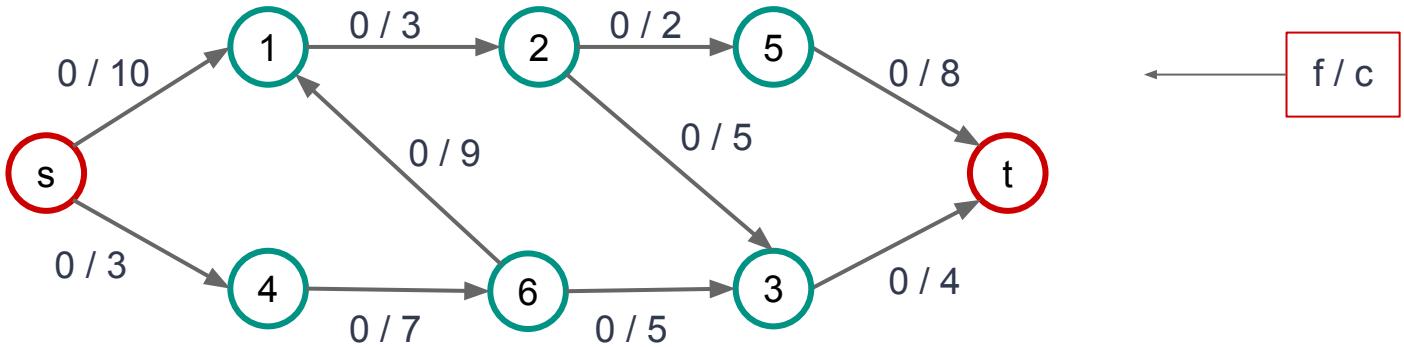
- fie  $P$  s-t lanțul găsit în **construiește\_s-t\_lant\_nesat\_BF()**
- calculăm  $i(P)$
- pentru fiecare arc  $e$  al lanțului  $P$ 
  - **creștem** cu  $i(P)$  fluxul pe  $e$  dacă este **arc direct**
  - **scădem** cu  $i(P)$  fluxul pe  $e$  dacă este arc invers

# Exemplu

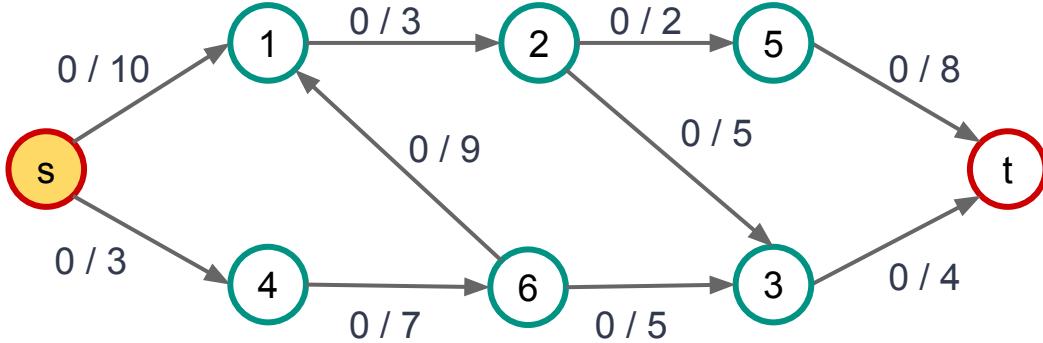
# Algoritmul Edmonds-Karp



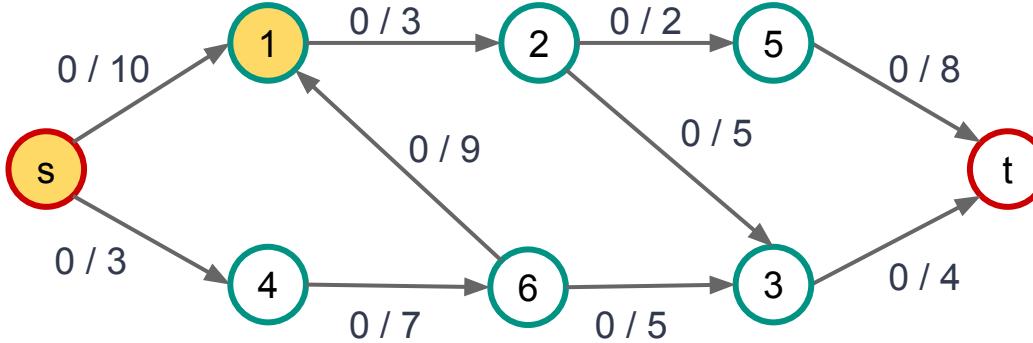
**initializează\_flux\_nul()**



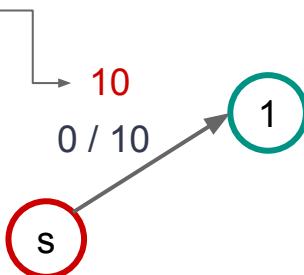
**construiește\_s-t\_lanț\_nesat\_BF()**

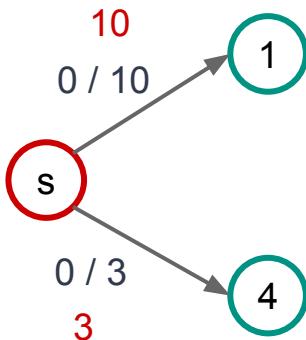
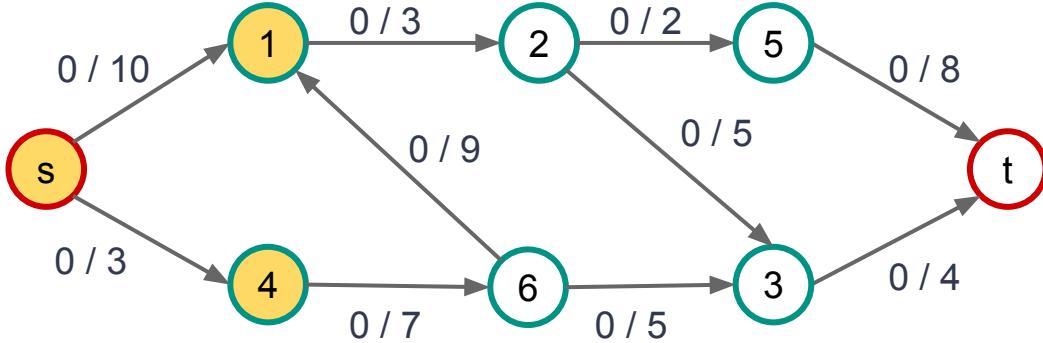


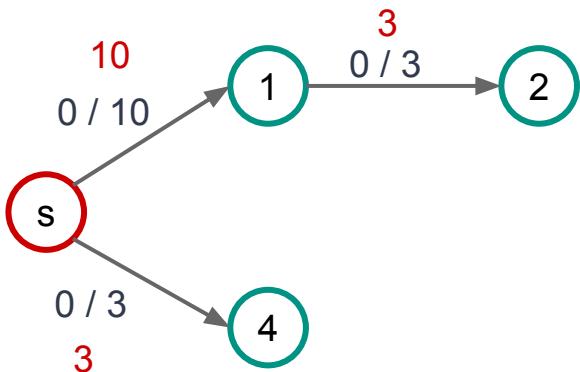
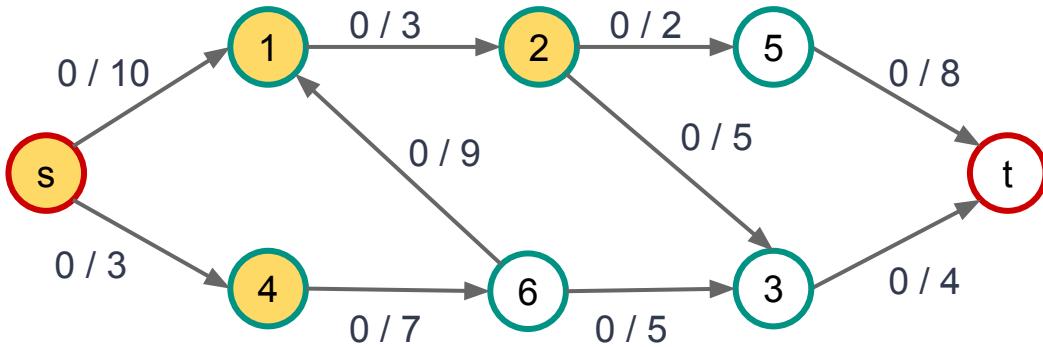
s

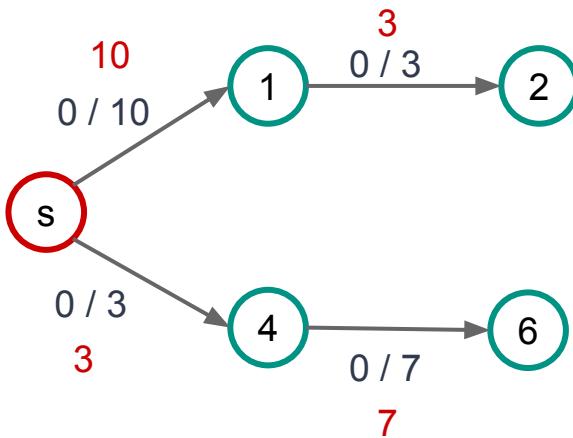
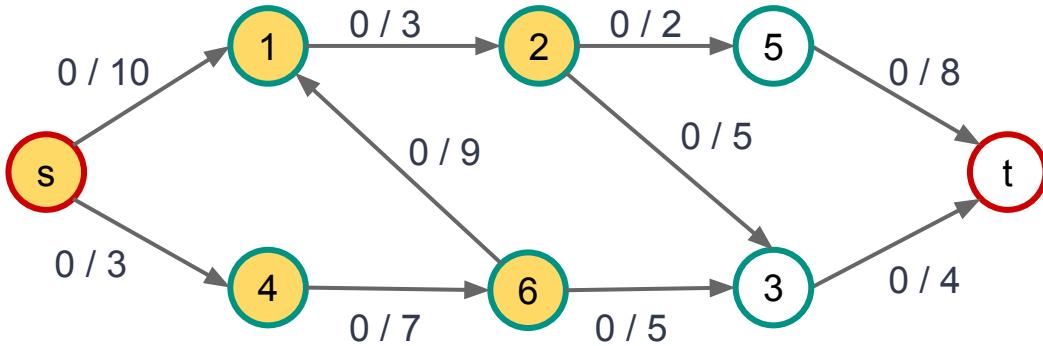


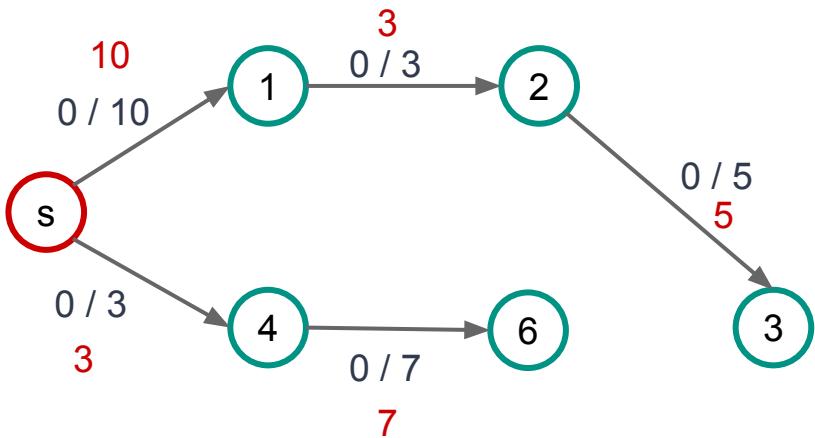
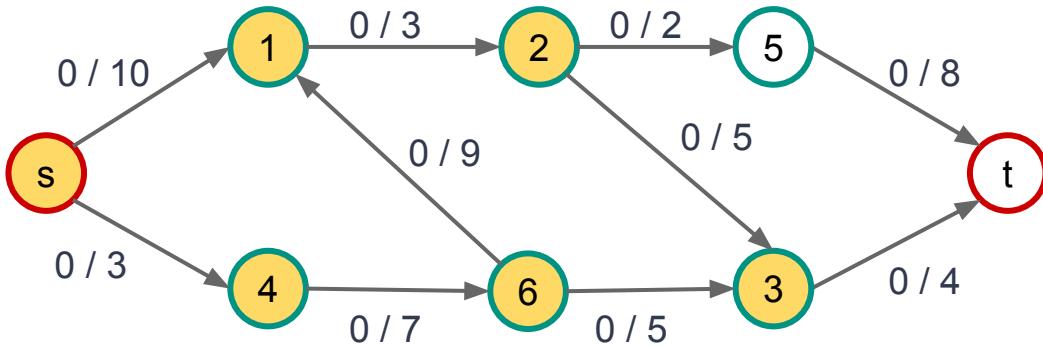
Capacitatea reziduală

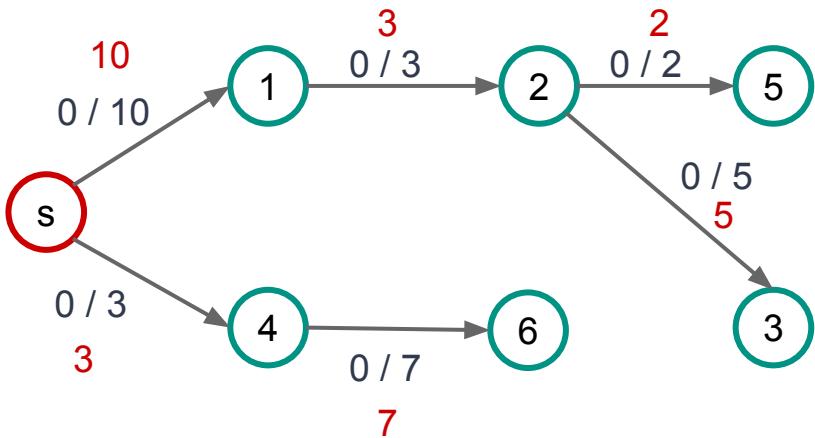
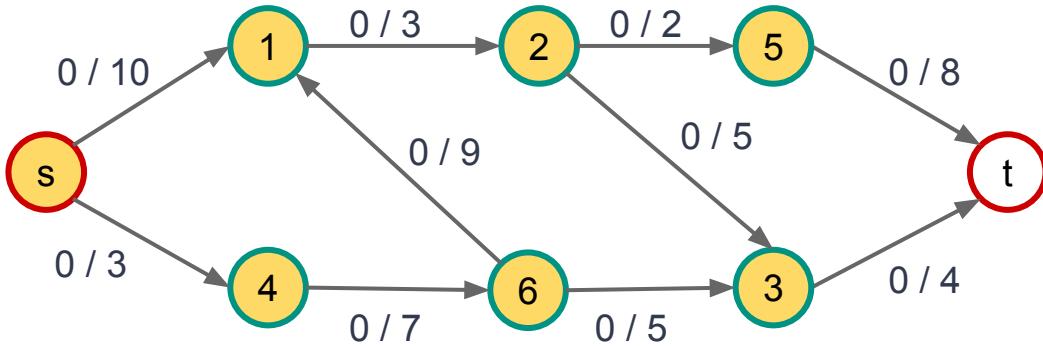


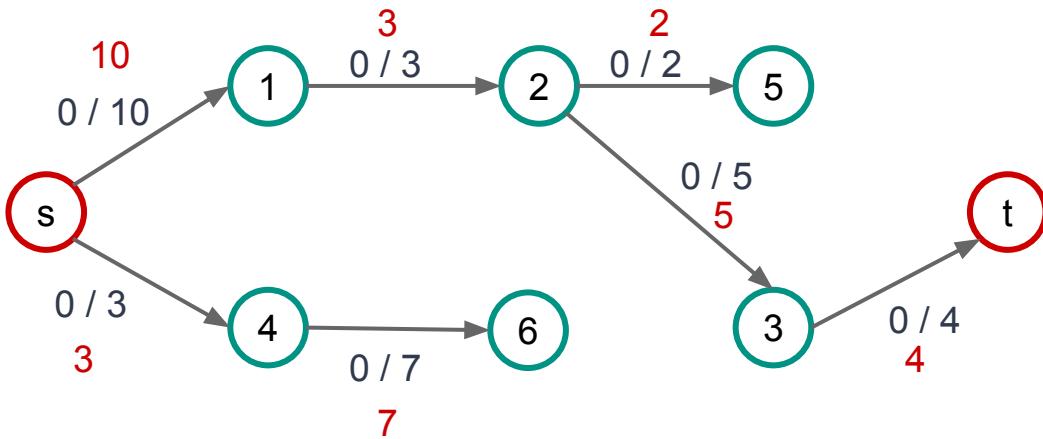
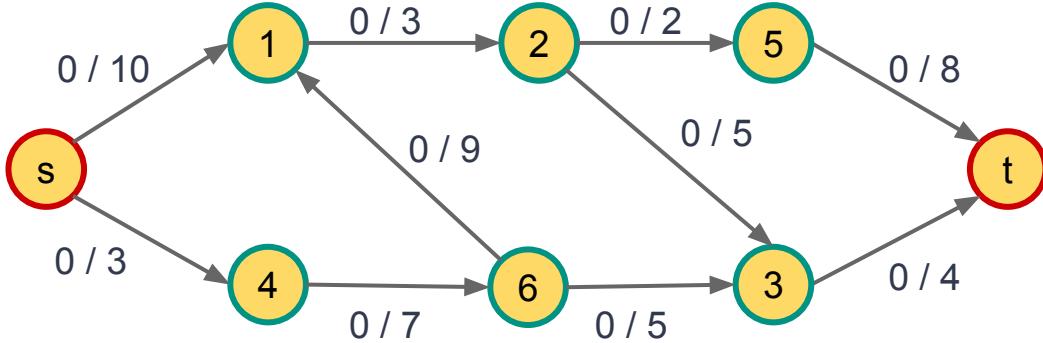


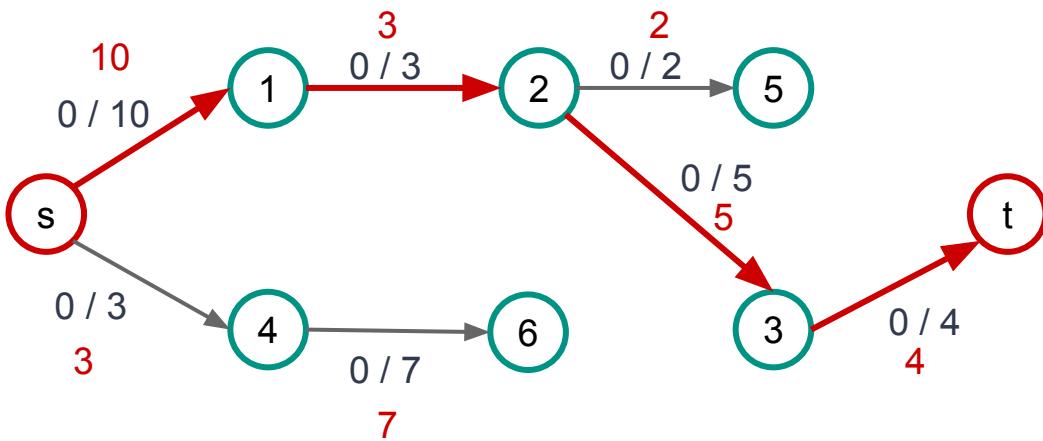
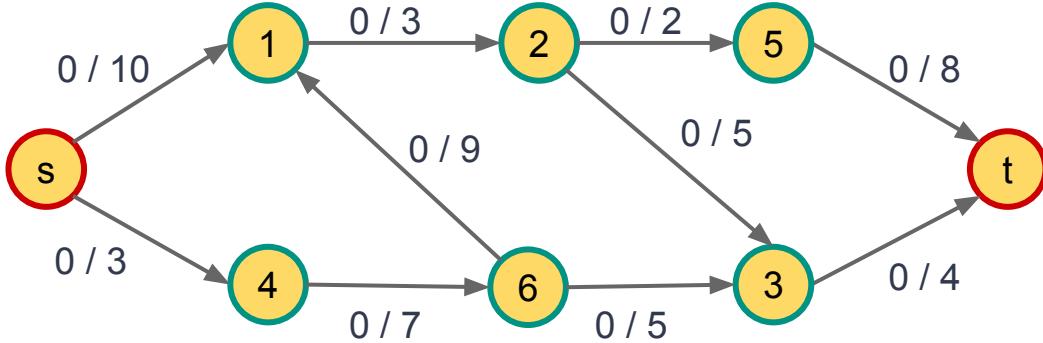




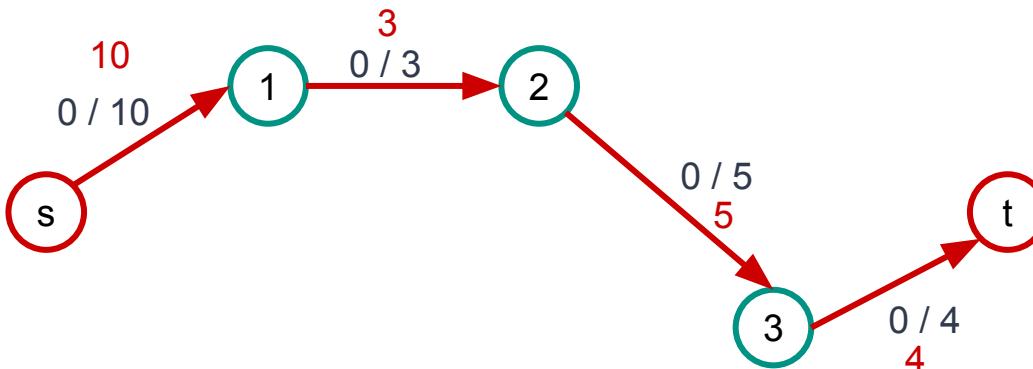
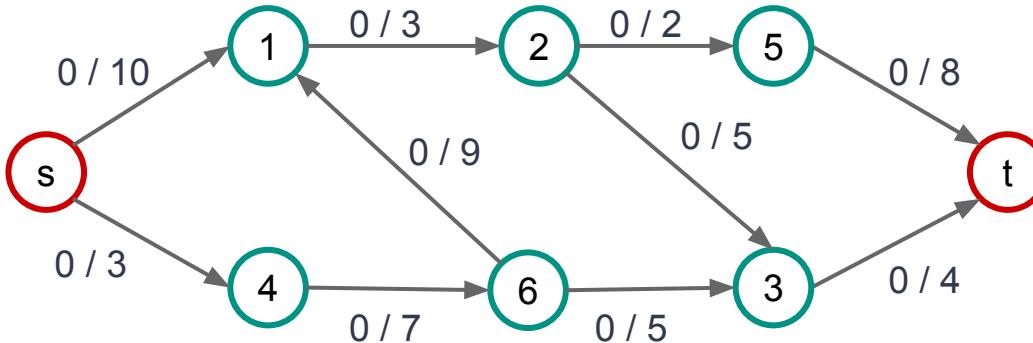




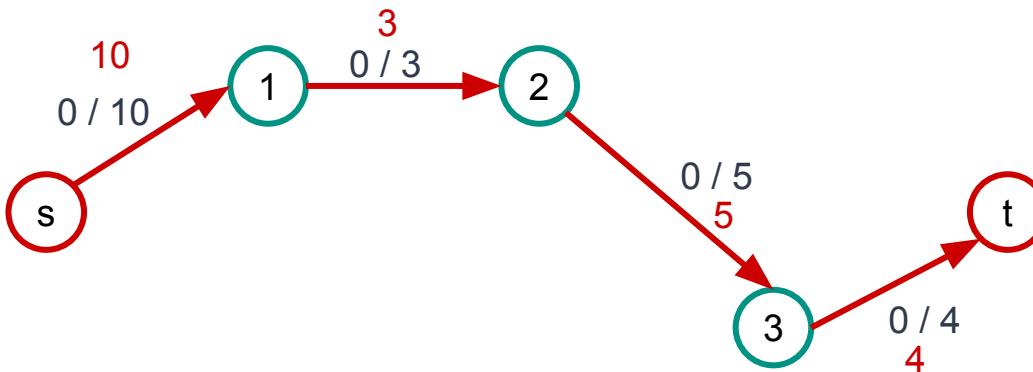
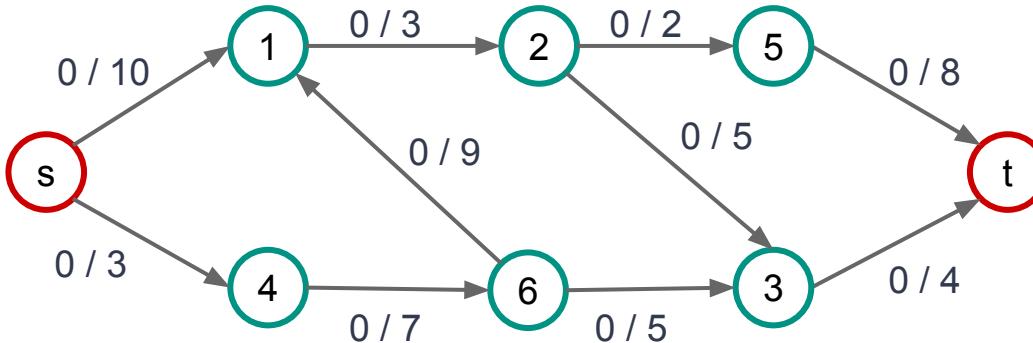




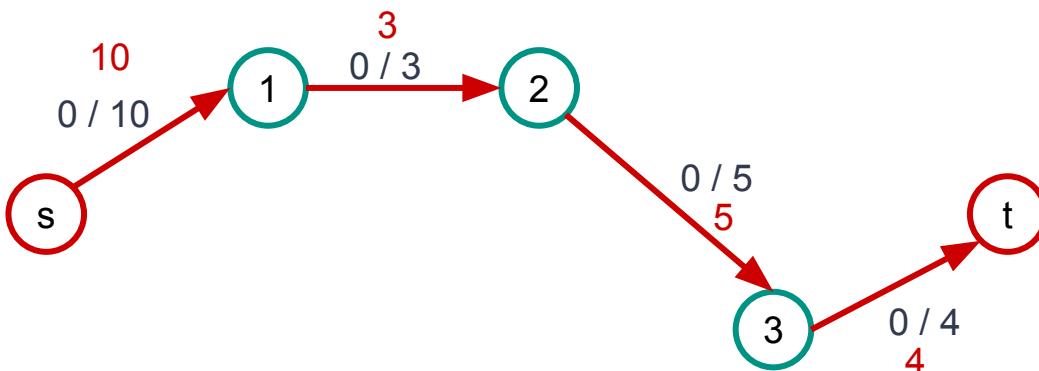
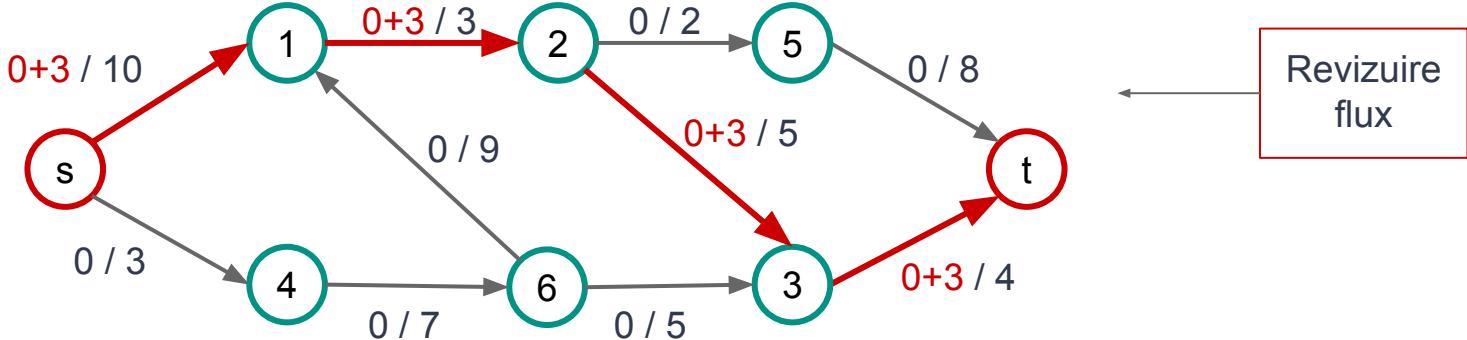
**revizuiеște\_flux\_lant()**



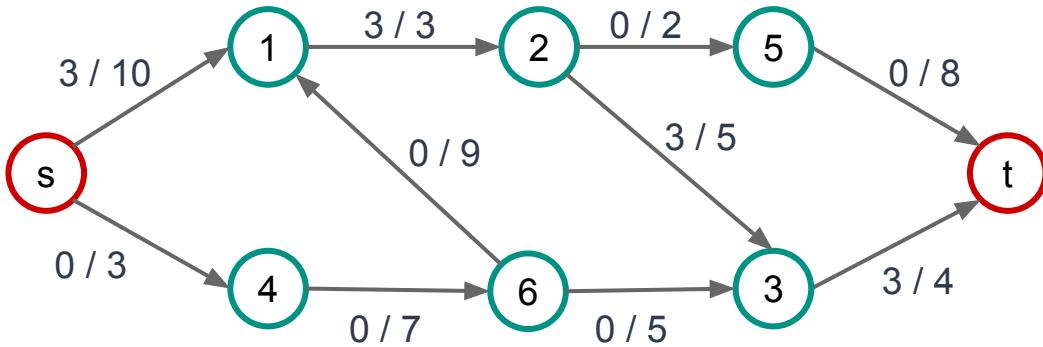
$$i(P) = ?$$



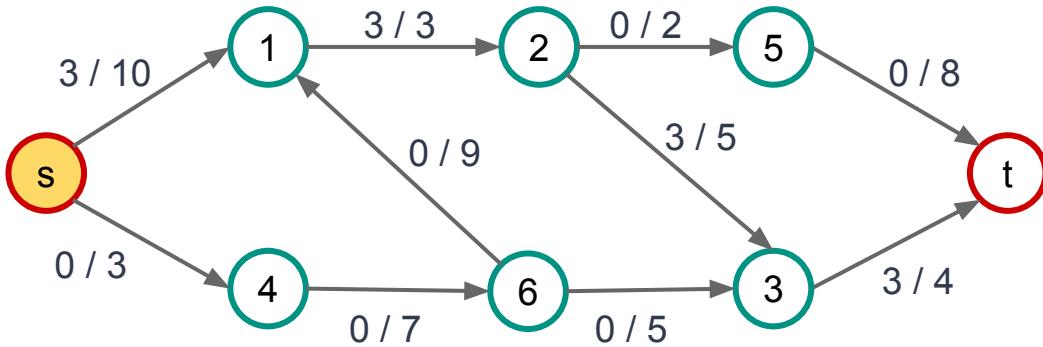
$$i(P) = \min \{ 10, 3, 5, 4 \} = 3$$



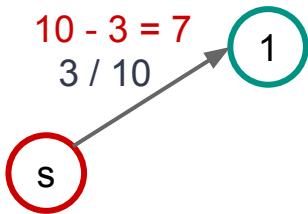
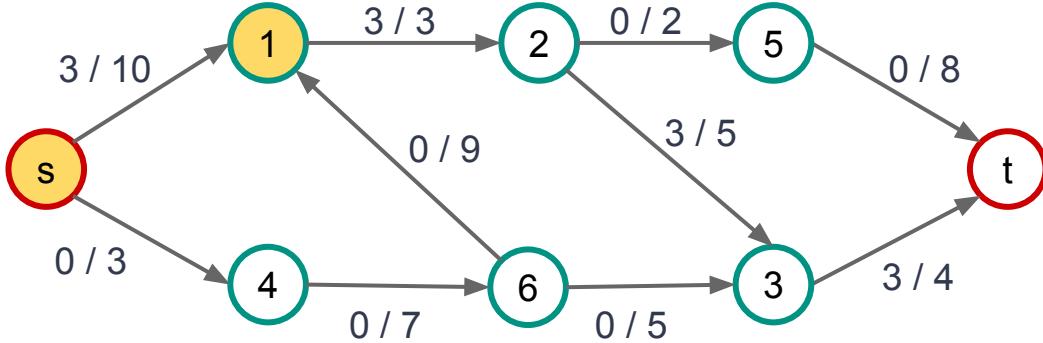
$$i(P) = \min \{ 10, 3, 5, 4 \} = 3$$

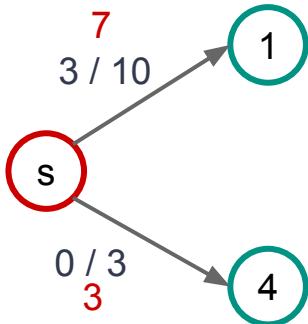
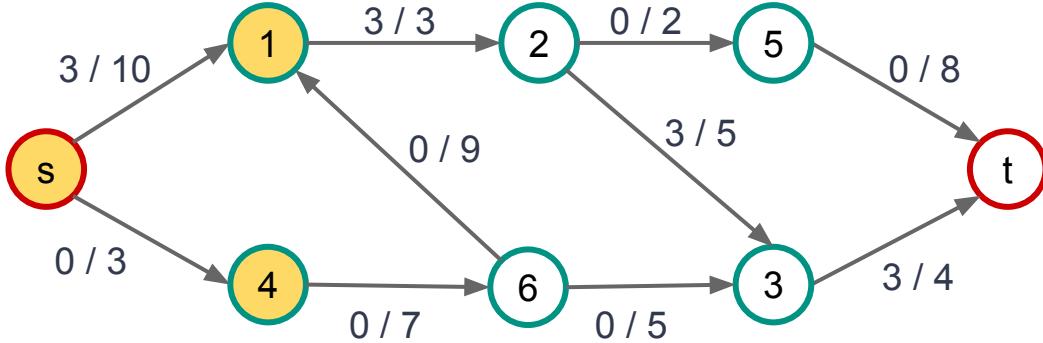


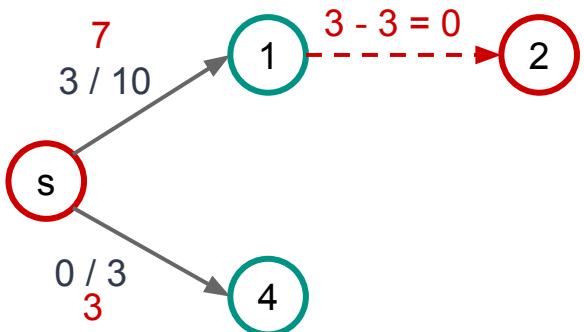
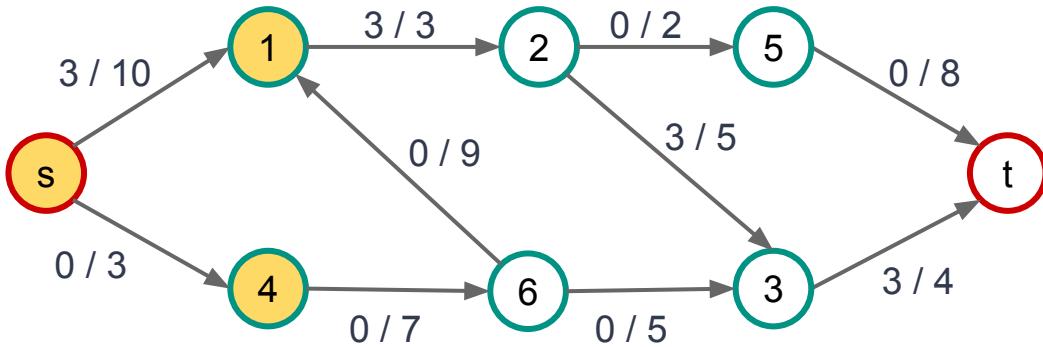
**construiește\_s-t\_lanț\_nesat\_BF()**

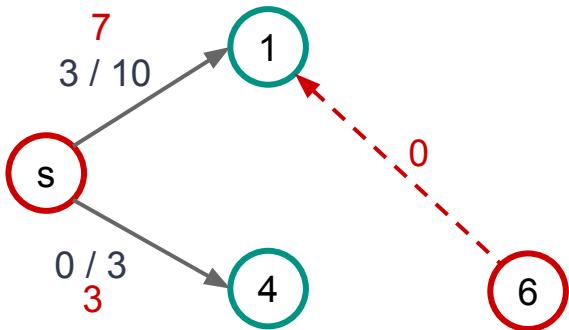
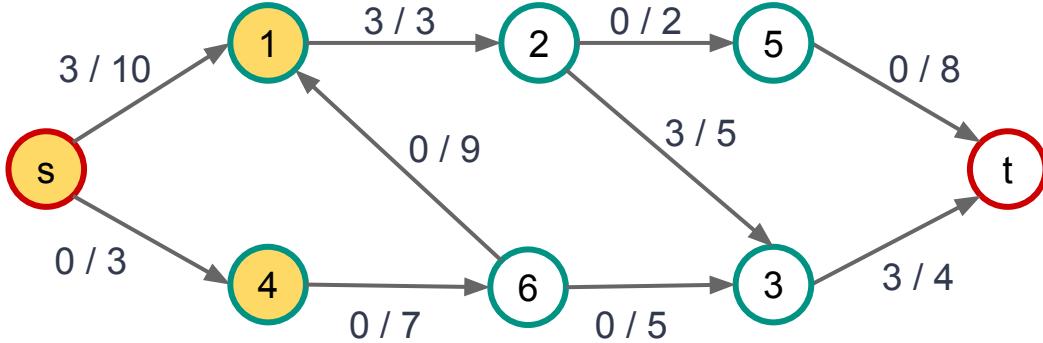


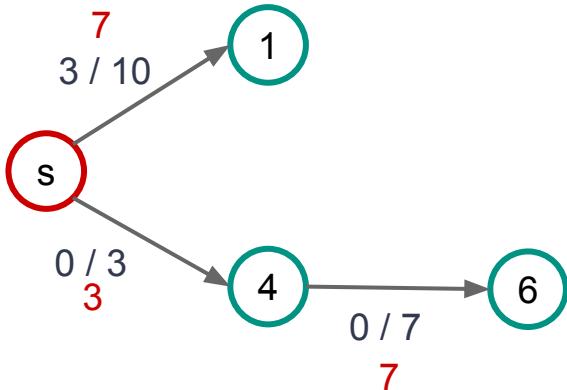
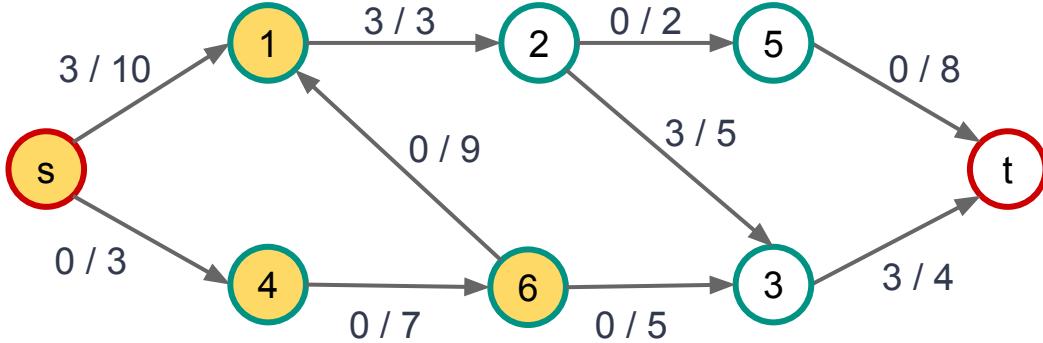
s

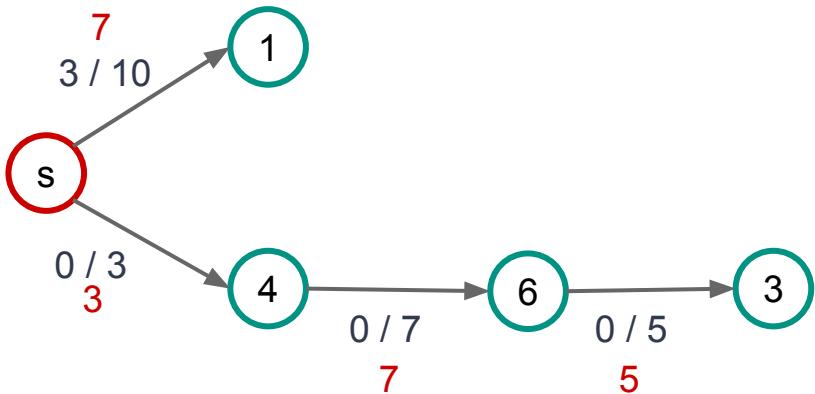
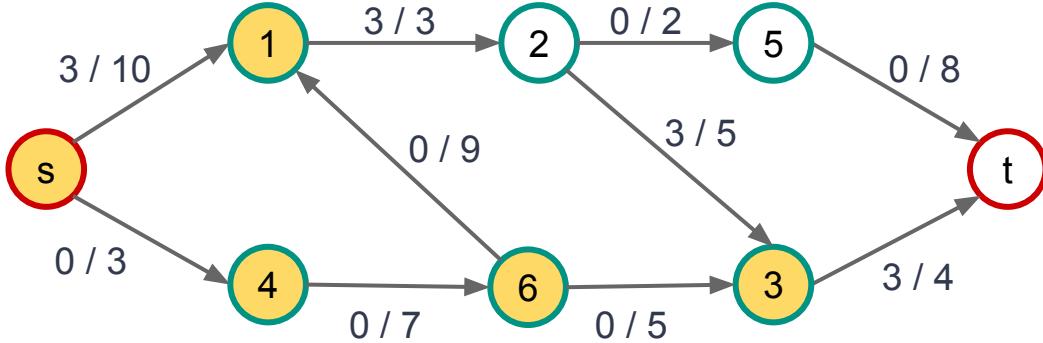


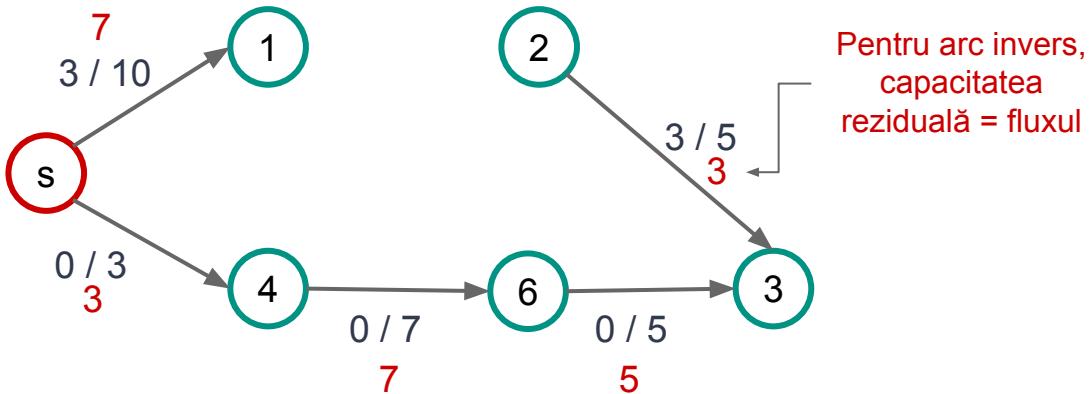
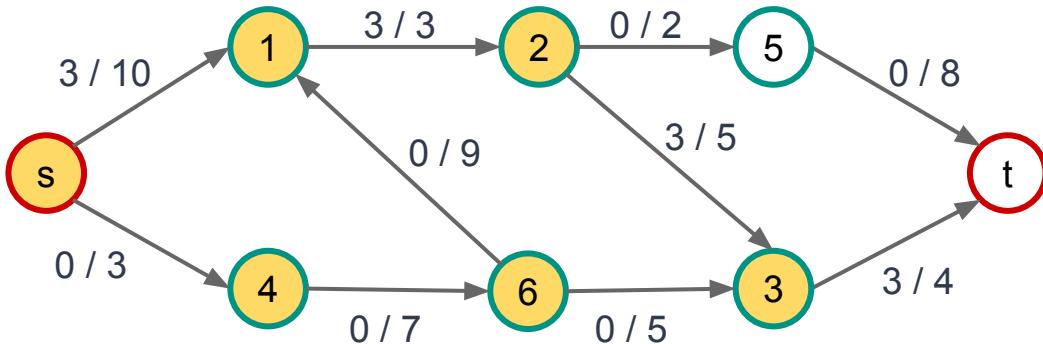


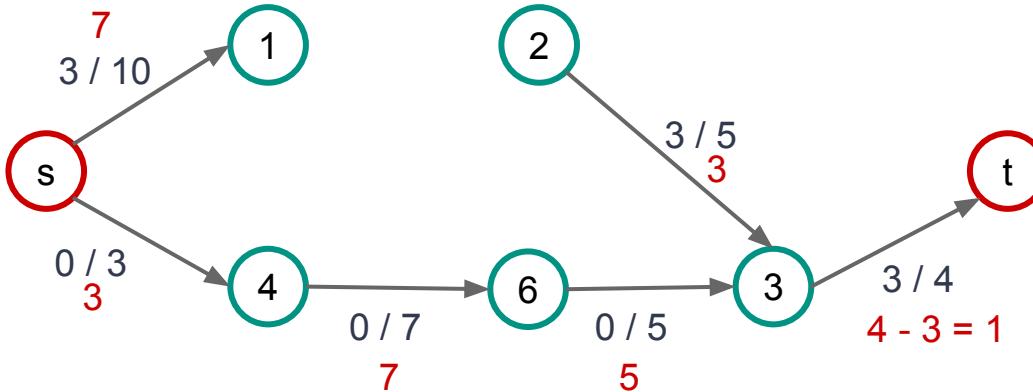
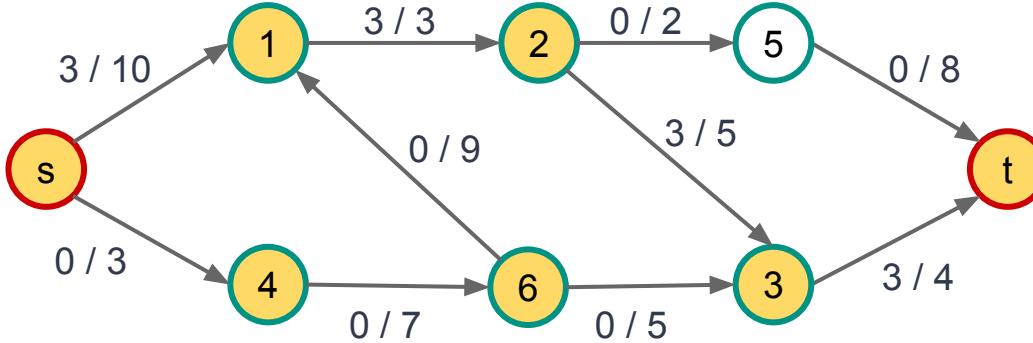


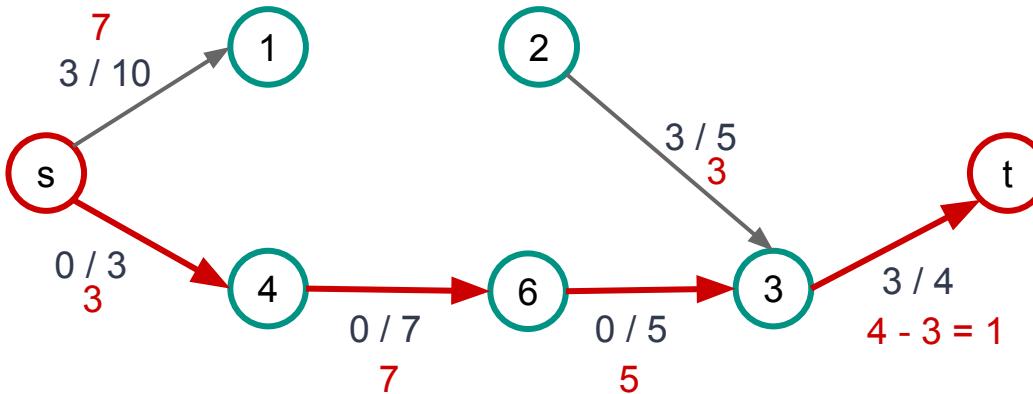
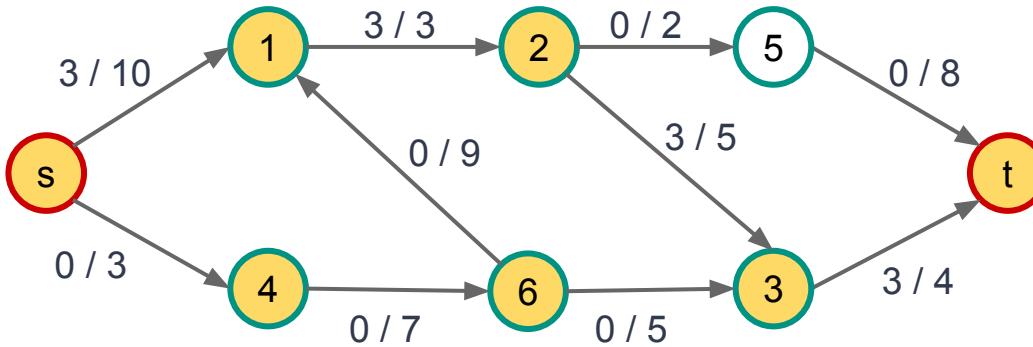




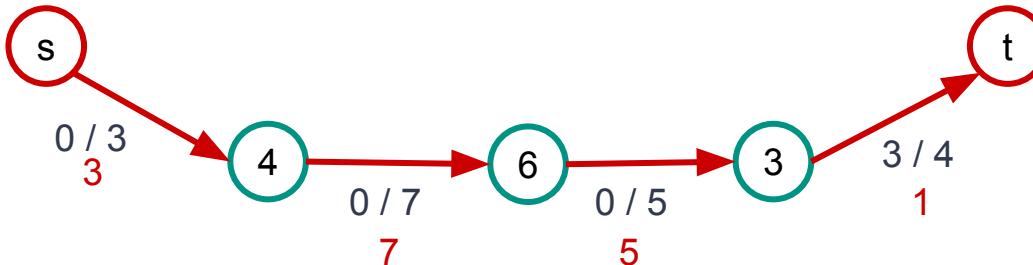
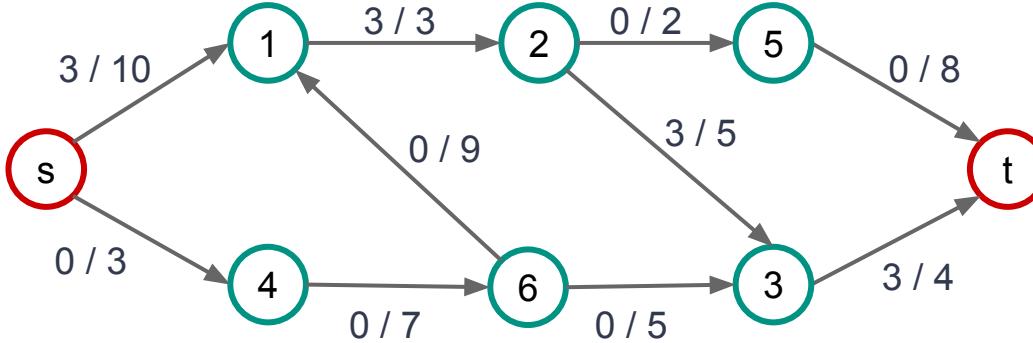




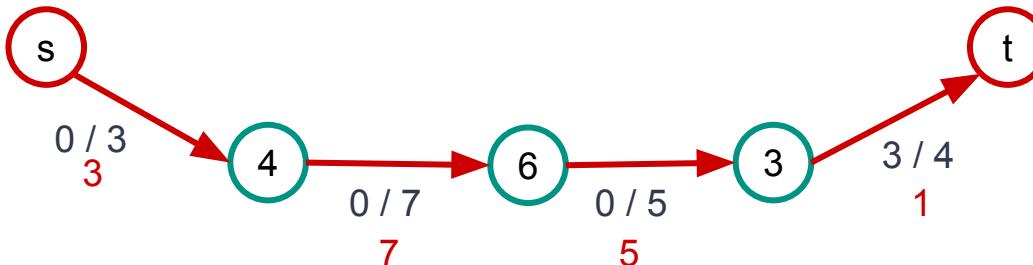
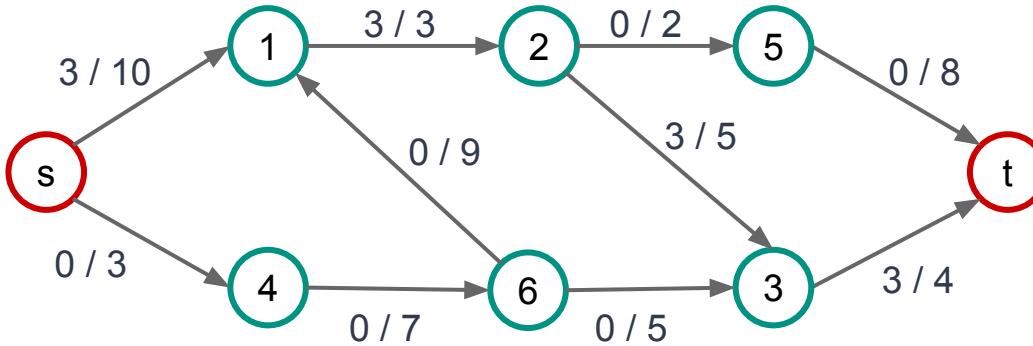




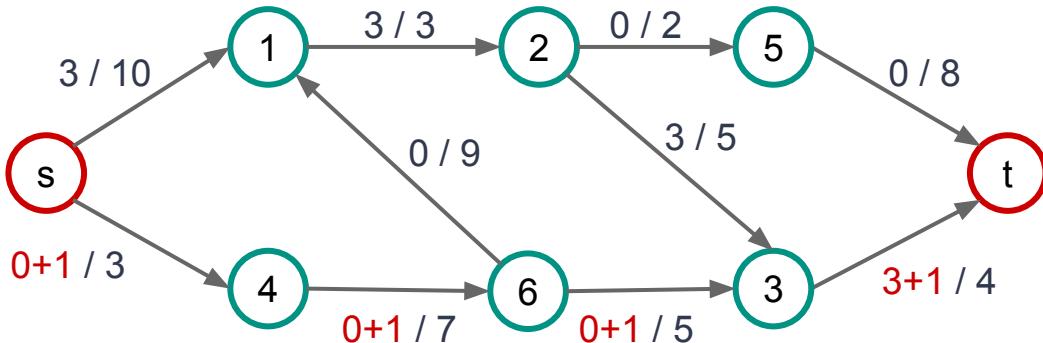
**revizuiеște\_flux\_lant()**



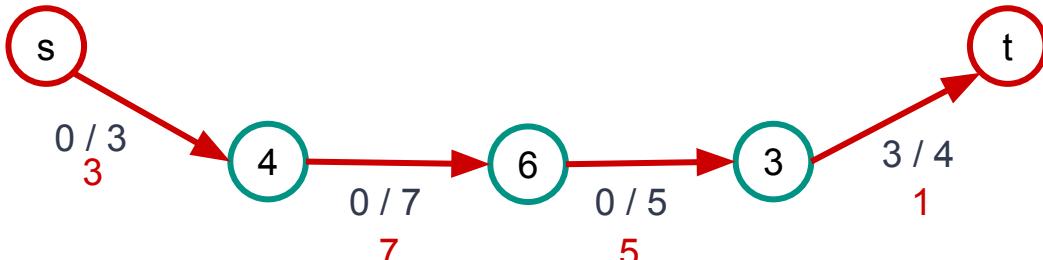
$$i(P) = ?$$



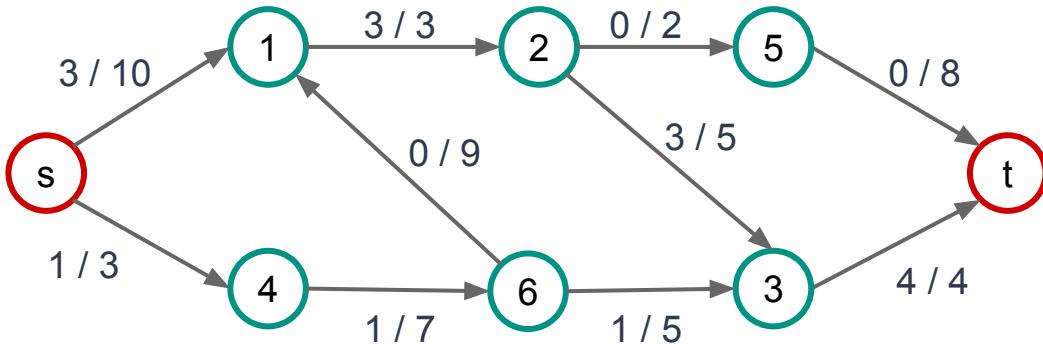
$$i(P) = \min \{ 3, 7, 5, 1 \} = 1$$



Revizuire  
flux

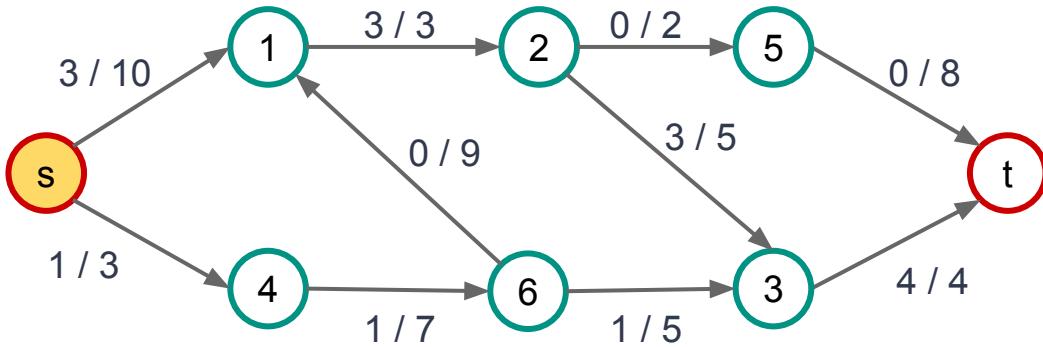


$$i(P) = \min \{ 3, 7, 5, 1 \} = 1$$

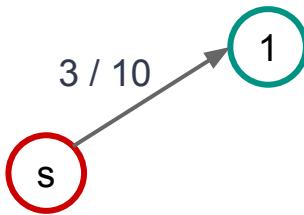
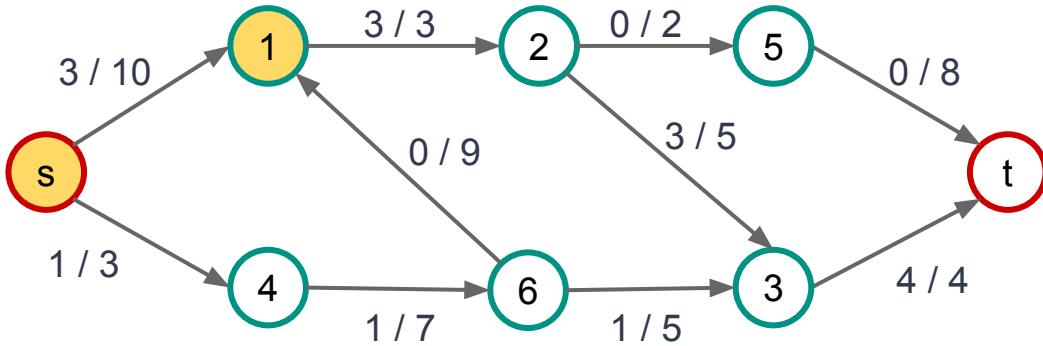


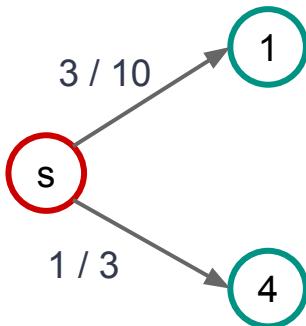
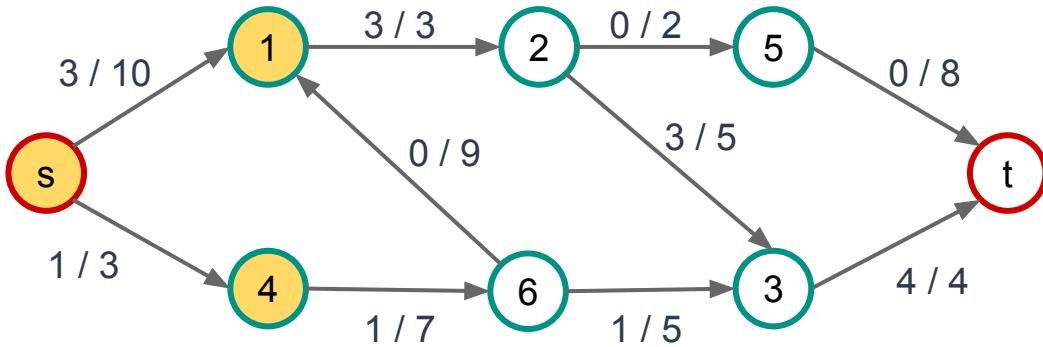
Revizuire  
flux

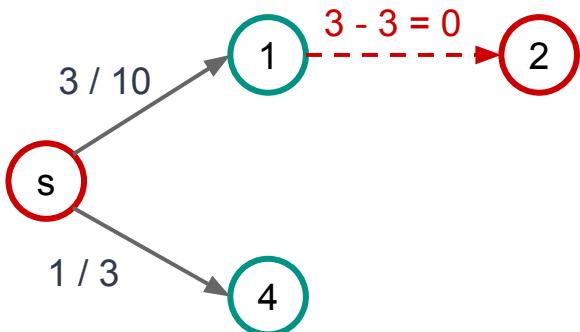
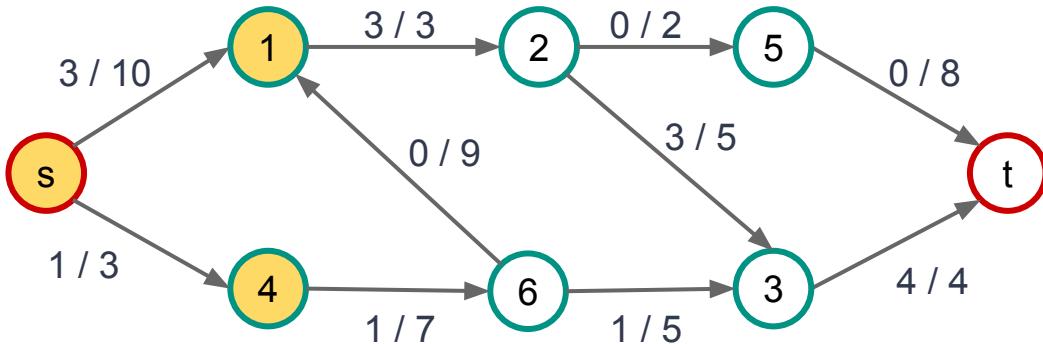
**construiește\_s-t\_lanț\_nesat\_BF()**

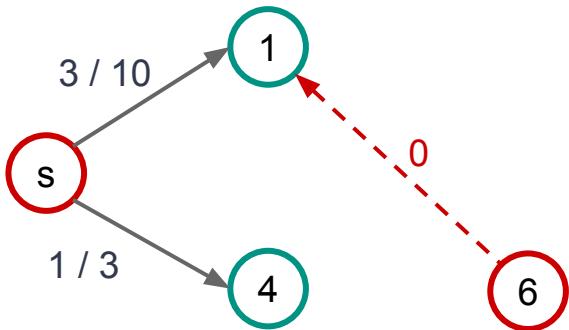
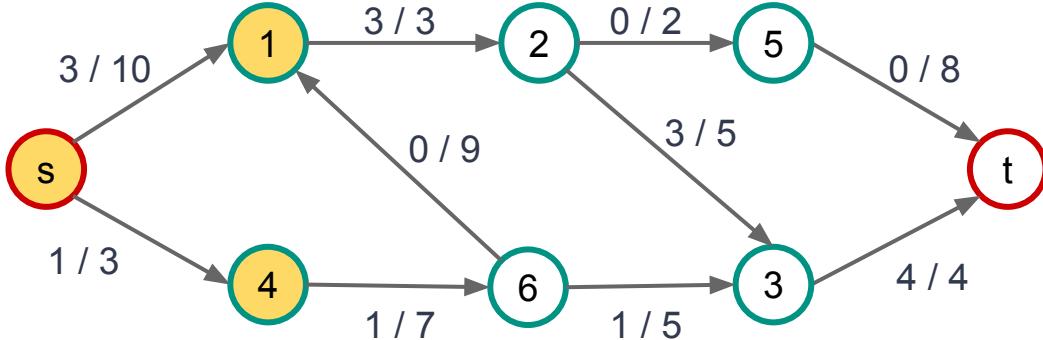


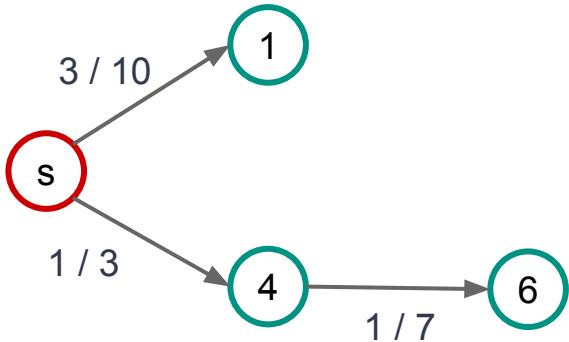
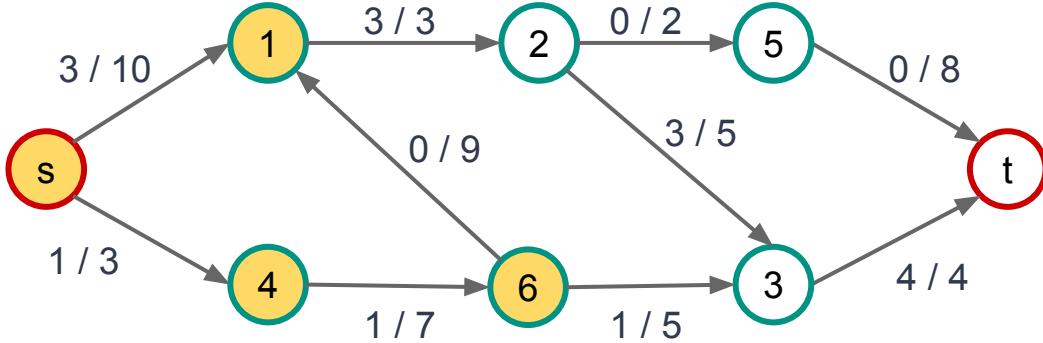
s

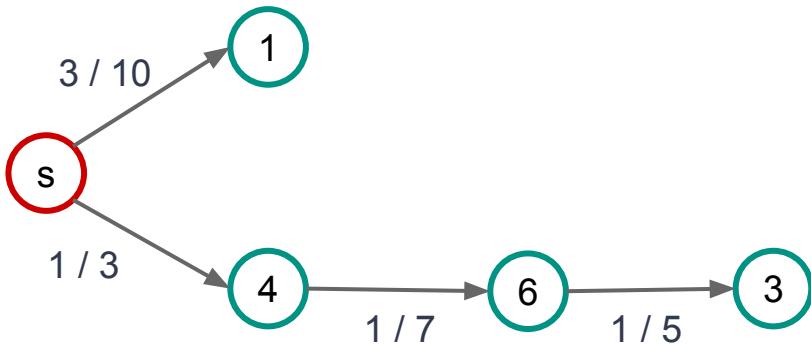
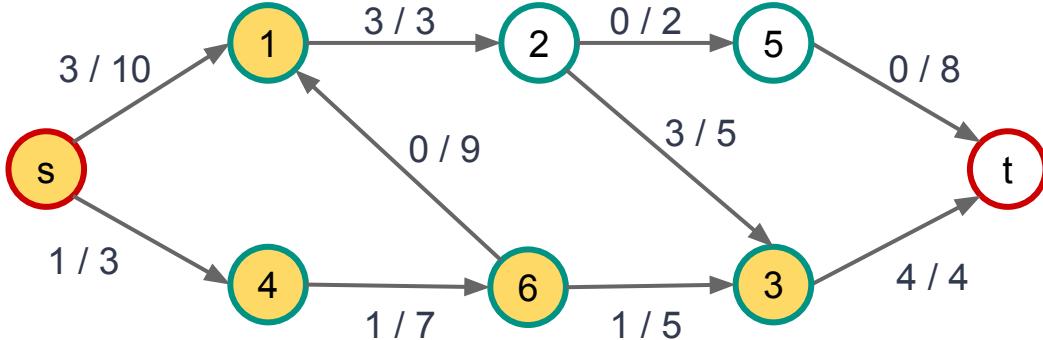


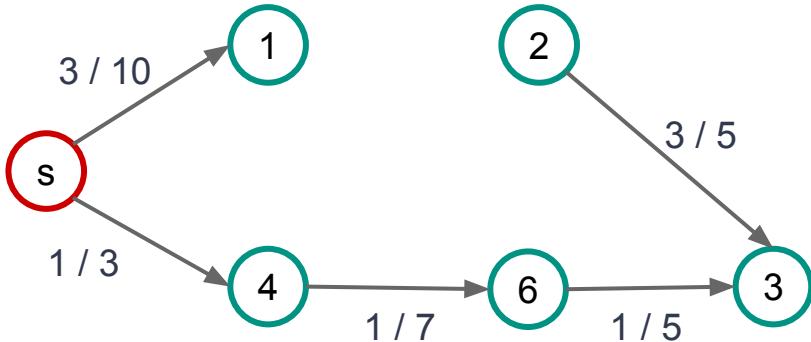
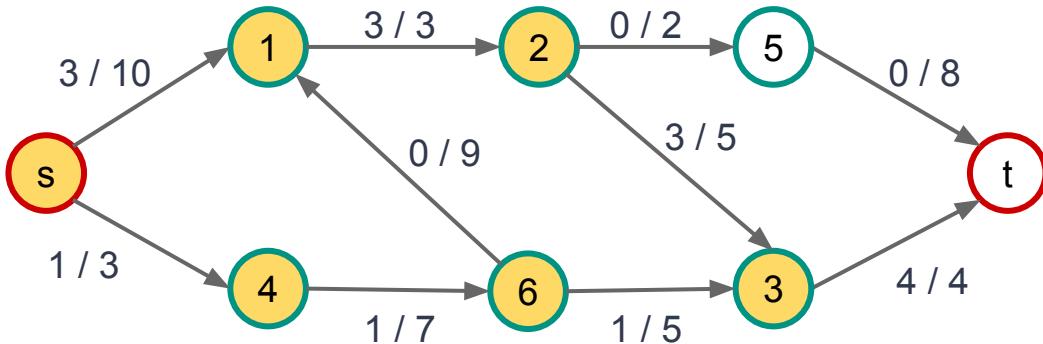


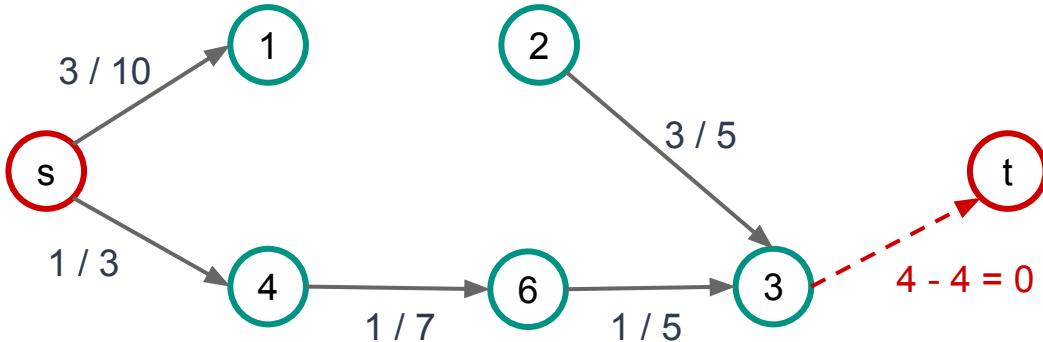
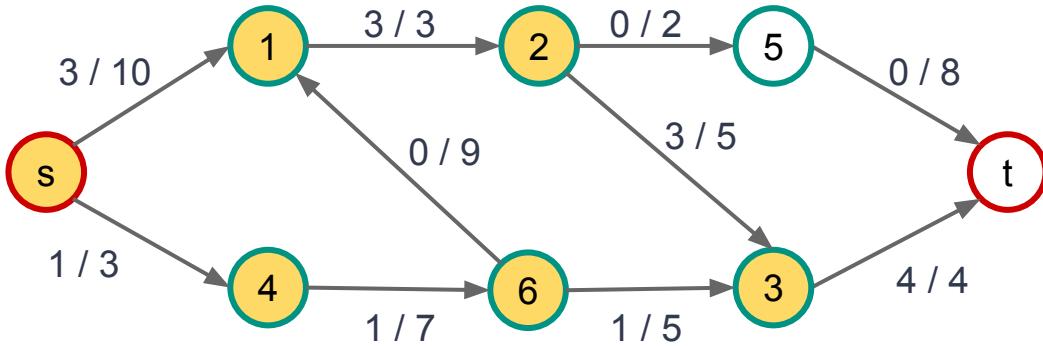


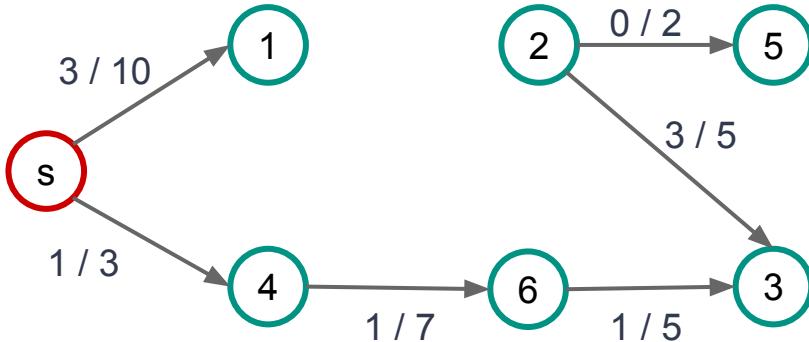
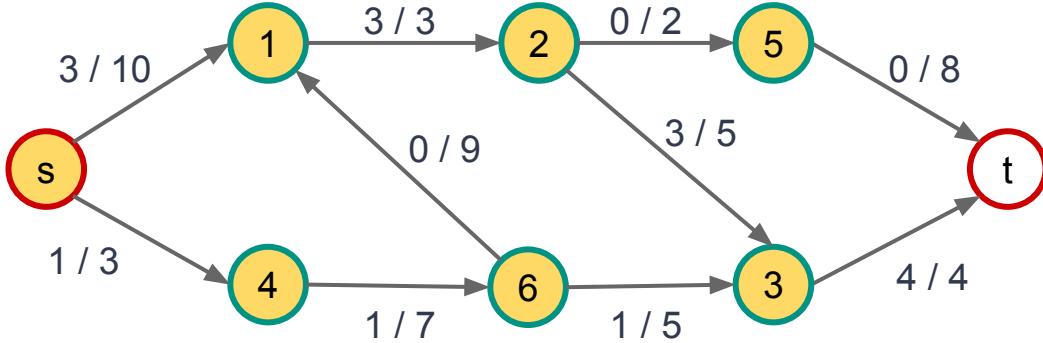


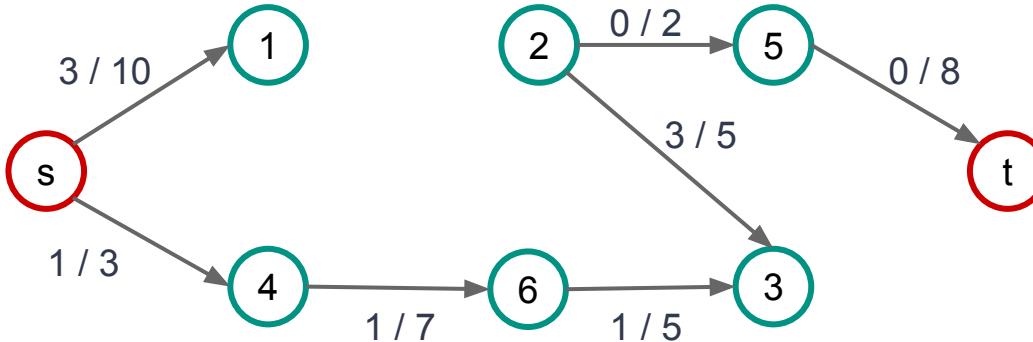
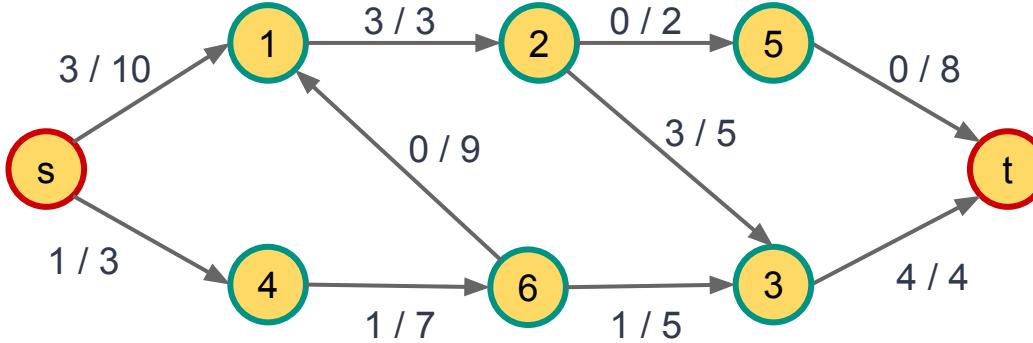


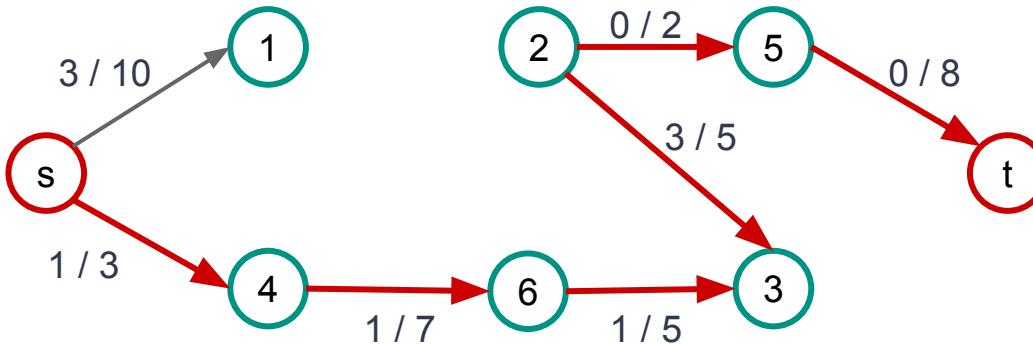
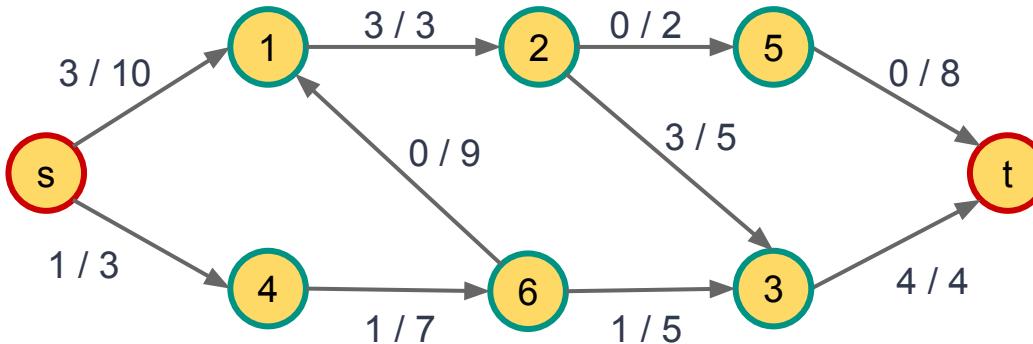




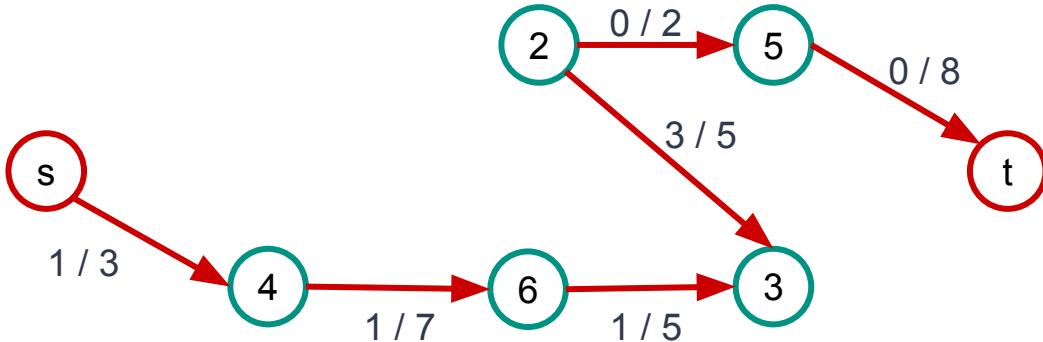
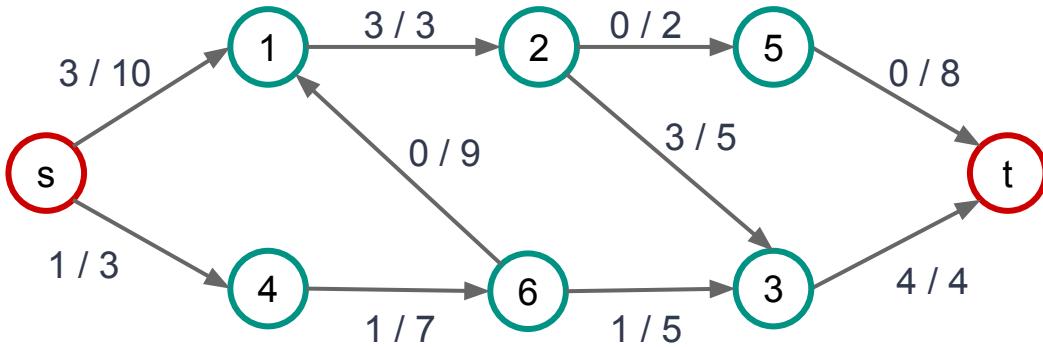


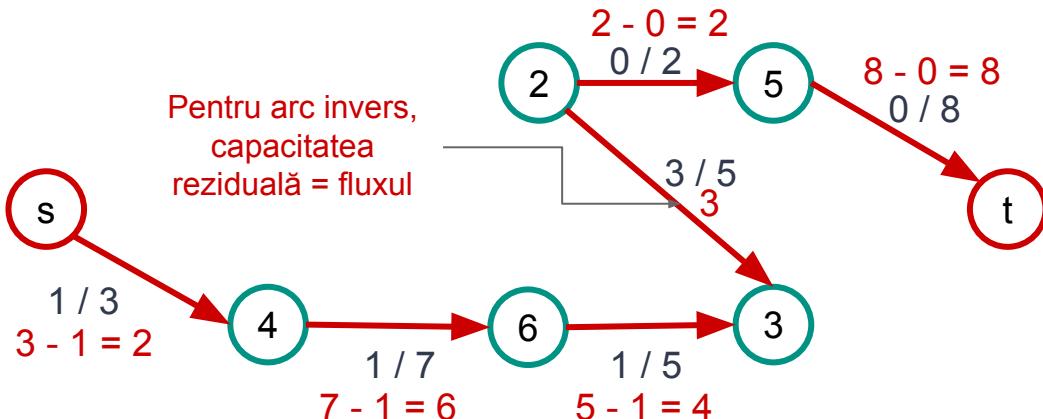
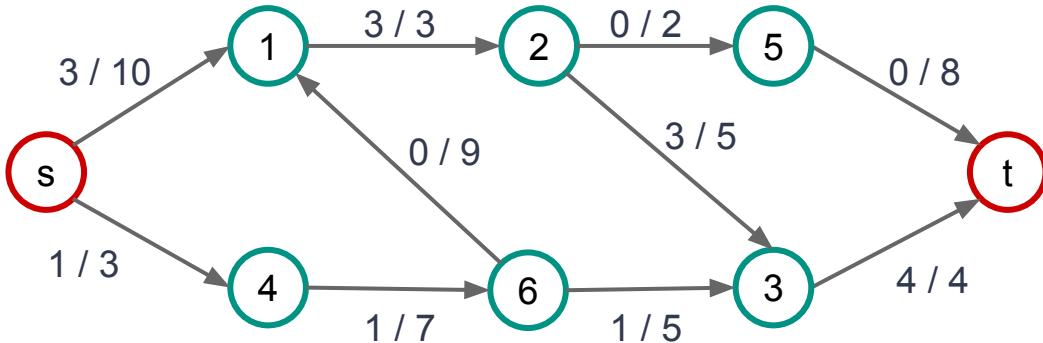


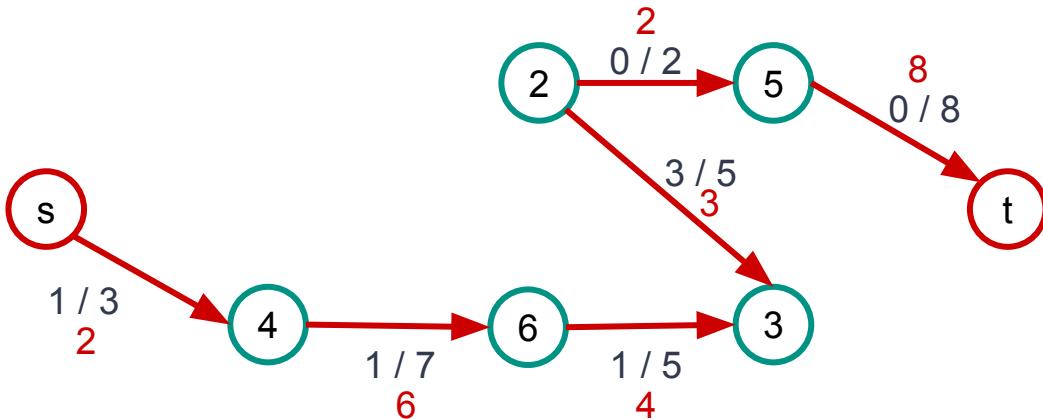
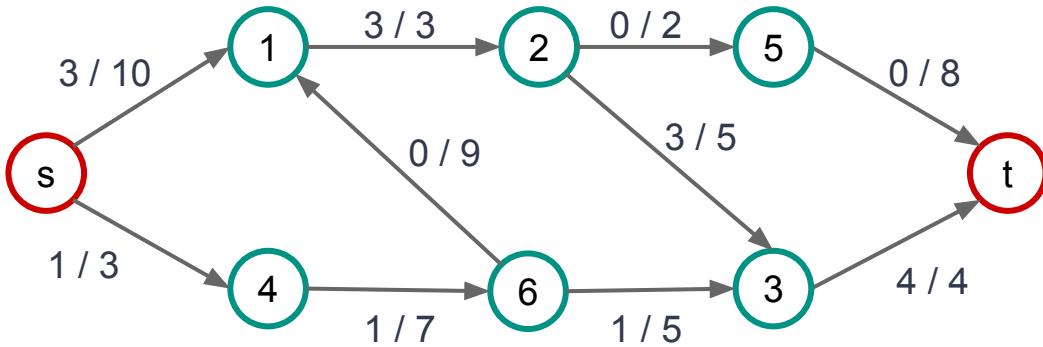




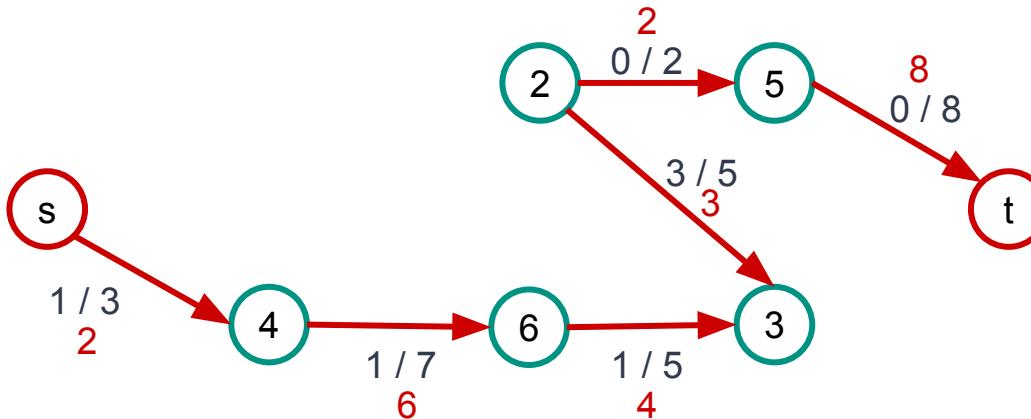
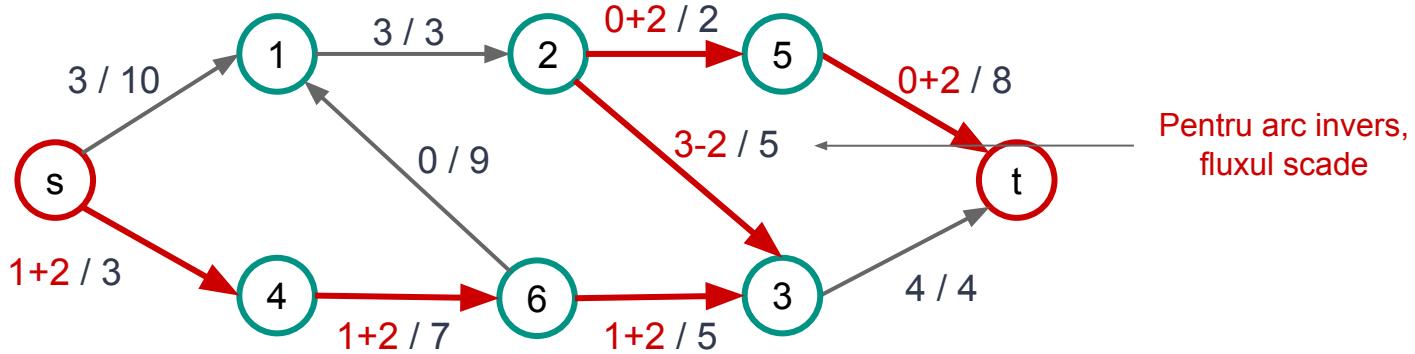
**revizuiеște\_flux\_lant()**



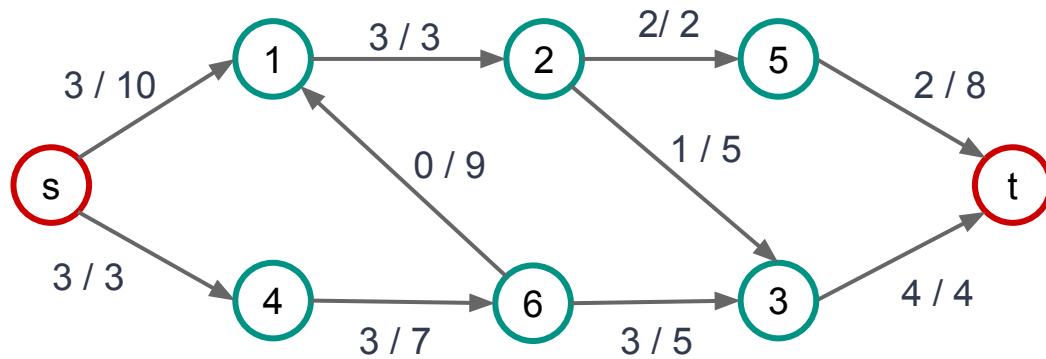




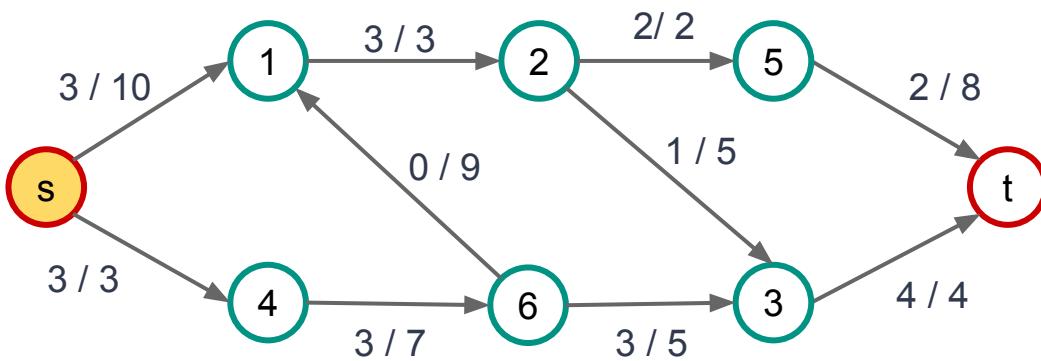
$$i(P) = \min \{ 2, 6, 4, 3, 2, 8 \} = 2$$



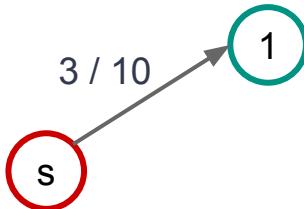
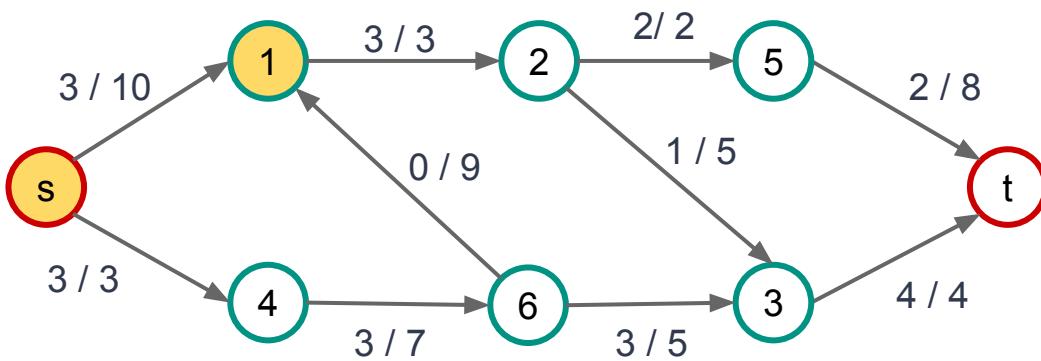
$$i(P) = \min \{ 2, 6, 4, 3, 2, 8 \} = 2$$

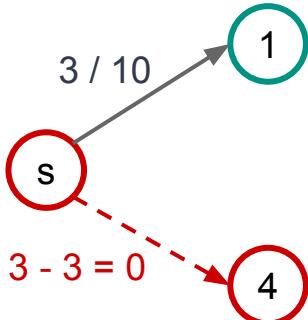
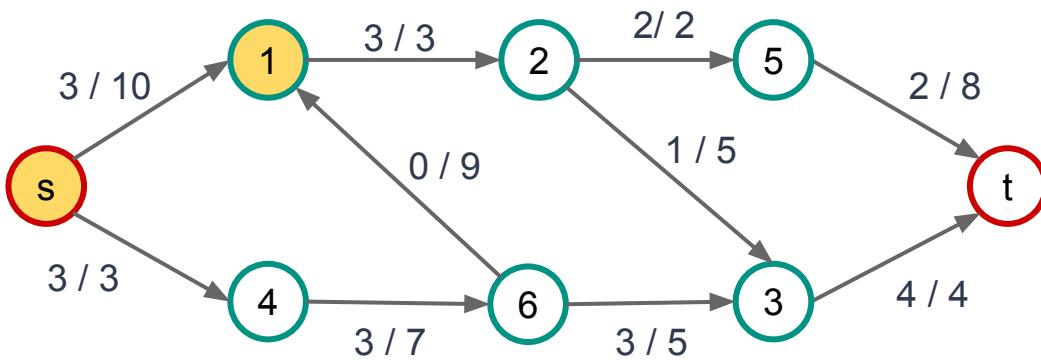


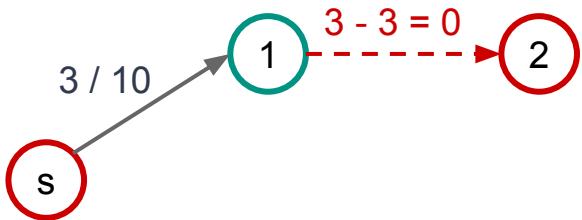
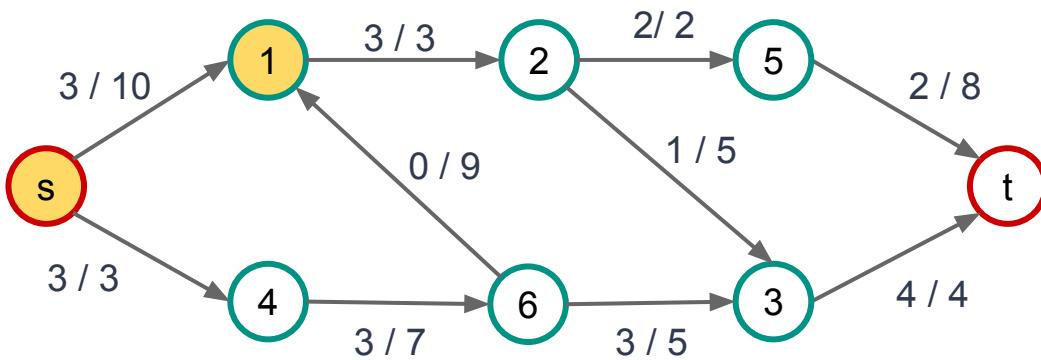
**construiește\_s-t\_lanț\_nesat\_BF()**

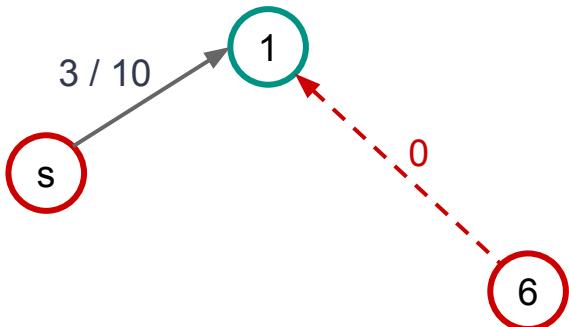
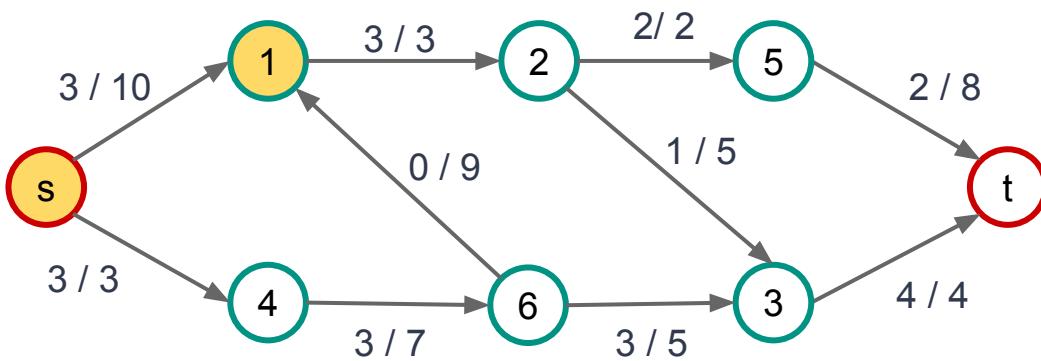


s









$t$  nu este accesibil din  $s \Rightarrow \text{STOP}$

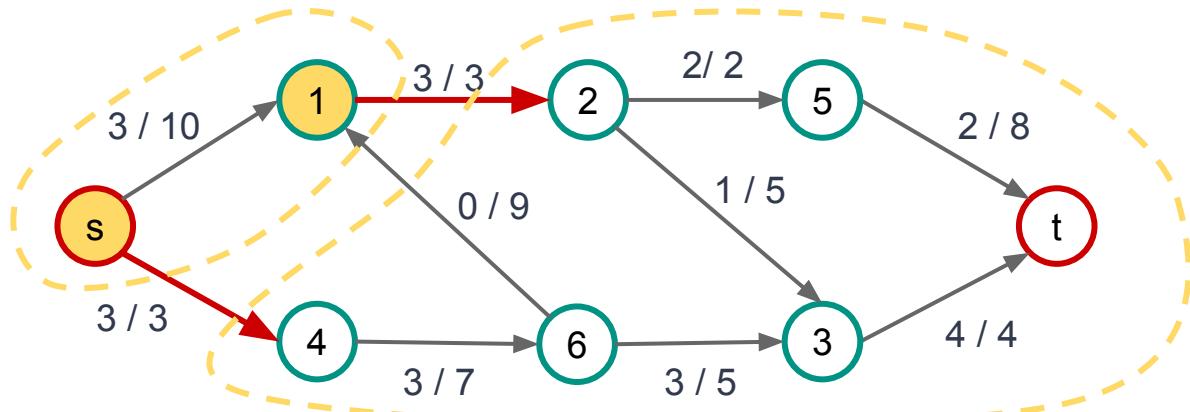
$f$  este flux maxim

$t$  nu este accesibil din  $s \Rightarrow$  STOP

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

Vom demonstra!





Tājetură minimă

# Sugestii de implementare

Algoritmul Edmonds-Karp

# Implementare

Memorăm lanțurile (arborele BF), folosind vectorul **tata**

**Convenție** - pentru arcele inverse (i, j) ținem minte tatăl cu semnul minus

$$\text{tata}[j] = -i$$

**construieste\_s-t\_lant\_nesat\_BF()**

**construiește\_s-t\_lanț\_nesat\_BF()**

pentru  $v \in V$  execută

  tata[v]  $\leftarrow \emptyset$ ; viz[v]  $\leftarrow 0$

**construiește\_s-t\_lanț\_nesat\_BF()**

pentru  $v \in V$  execută

$tata[v] \leftarrow 0$ ;    $viz[v] \leftarrow 0$

coada  $C \leftarrow \emptyset$

**adaugă(s, C)**

$viz[s] \leftarrow 1$

**construiește\_s-t\_lanț\_nesat\_BF()**

pentru  $v \in V$  execută

$tata[v] \leftarrow 0$ ;    $viz[v] \leftarrow 0$

coada  $C \leftarrow \emptyset$

adăugă( $s, C$ )

$viz[s] \leftarrow 1$

cât timp  $C \neq \emptyset$  execută

$i \leftarrow \text{extrage}(C)$

```
construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s, C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
```

```
construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s, C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
                adaugă( $j, C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow i$ 
```

```
construiește_s-t_lanț_nesat_BF()
    pentru v ∈ V execută
        tata[v] ← 0; viz[v] ← 0
    coada C ← ∅
    adaugă(s, C)
    viz[s] ← 1
    cât timp C ≠ ∅ execută
        i ← extrage(C)
        pentru ij ∈ E execută          // arc direct
            dacă viz[j] = 0 și c(ij) > f(ij) atunci
                adaugă(j, C)
                viz[j] ← 1
                tata[j] ← i
            dacă j = t atunci
                STOP și returnează true (1)
```

```

construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s$ ,  $C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
                adaugă( $j$ ,  $C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow i$ 
                dacă  $j = t$  atunci
                    STOP și returnează true (1)
            pentru  $ji \in E$  execută // arc invers

```

```

construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s$ ,  $C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
                adaugă( $j$ ,  $C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow i$ 
                dacă  $j = t$  atunci
                    STOP și returnează true (1)
            pentru  $ji \in E$  execută // arc invers
                dacă  $viz[j] = 0$  și  $f(ji) > 0$  atunci

```

```

construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s$ ,  $C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
                adaugă( $j$ ,  $C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow i$ 
            dacă  $j = t$  atunci
                STOP și returnează true (1)
        pentru  $ji \in E$  execută // arc invers
            dacă  $viz[j] = 0$  și  $f(ji) > 0$  atunci
                adaugă( $j$ ,  $C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow -i$ 

```

```

construiește_s-t_lanț_nesat_BF()
    pentru  $v \in V$  execută
         $tata[v] \leftarrow 0$ ;  $viz[v] \leftarrow 0$ 
    coada  $C \leftarrow \emptyset$ 
    adaugă( $s$ ,  $C$ )
     $viz[s] \leftarrow 1$ 
    cât timp  $C \neq \emptyset$  execută
         $i \leftarrow \text{extrage}(C)$ 
        pentru  $ij \in E$  execută // arc direct
            dacă  $viz[j] = 0$  și  $c(ij) > f(ij)$  atunci
                adaugă( $j$ ,  $C$ )
                 $viz[j] \leftarrow 1$ 
                 $tata[j] \leftarrow i$ 
                dacă  $j = t$  atunci
                    STOP și returnează true (1)
            pentru  $ji \in E$  execută // arc invers
                dacă  $viz[j] = 0$  și  $f(ji) > 0$  atunci
                    adaugă( $j$ ,  $C$ )
                     $viz[j] \leftarrow 1$ 
                     $tata[j] \leftarrow -i$ 
                    dacă  $j = t$  atunci
                        STOP și returnează true (1)
    returnează false (0)

```

# Algoritmul Edmonds-Karp

## Complexitate

- Algoritm generic Ford-Fulkerson -  $O(mL) / O(nmC)$
- Implementare Edmonds-Karp -  $O(nm^2)$

# Implementare

## Schema:

initializează\_flux\_nul()

cât timp (construiește\_s-t\_lanț\_nesat\_BF() == true) execută

revizuește\_flux\_lanț()

afisează\_flux()

**Amintim:** a determina un s-t lanț nesaturat folosind BF în G  $\Leftrightarrow$  a determina un s-t drum folosind BF în graful rezidual  $G_f$

# Varianta 2 de implementare

revizuirea fluxului folosind s-t drumuri din  
 $G_f$  (în graful rezidual)

# Implementare folosind graf rezidual

Schema devine:

inițializează\_flux\_nul()

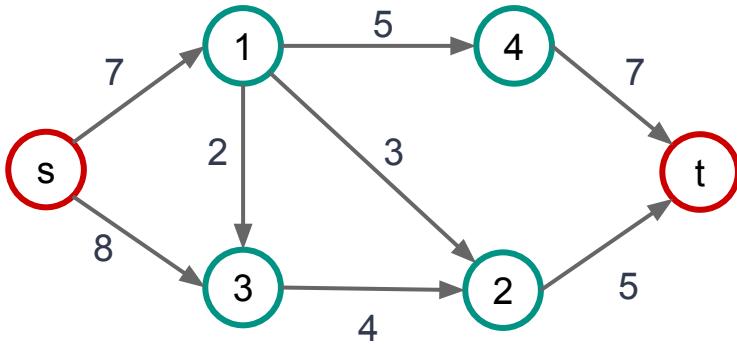
cât timp (construiește\_s-t\_drum\_în\_G\_fBF() == true) execută

revizuește\_flux\_lanț()

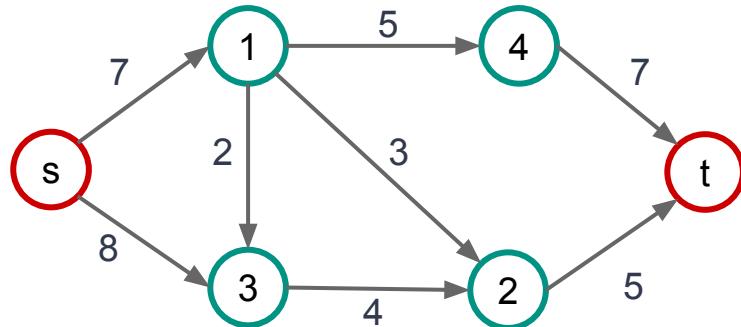
actualizează  $G_f$

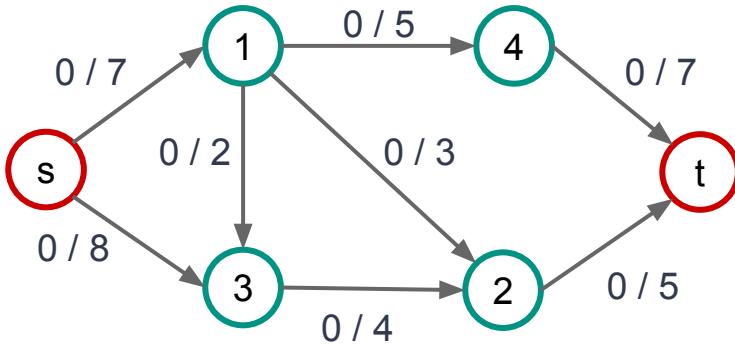
afișează\_flux()

Detaliem această schemă.

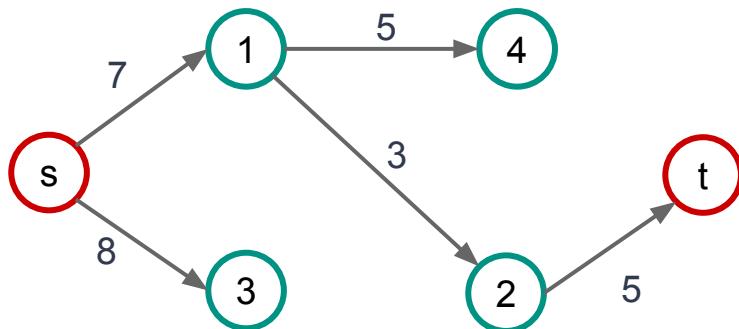
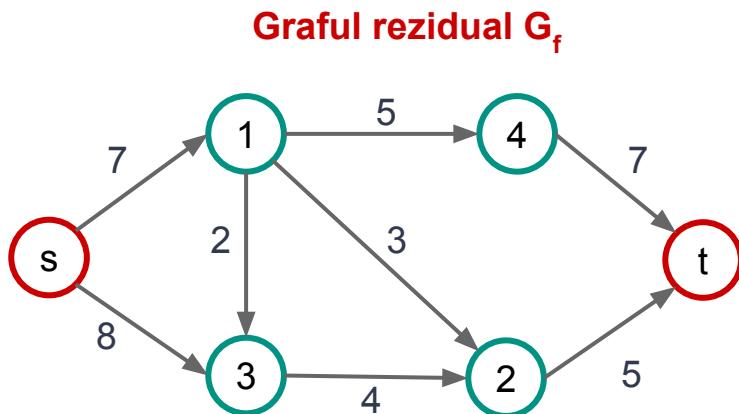


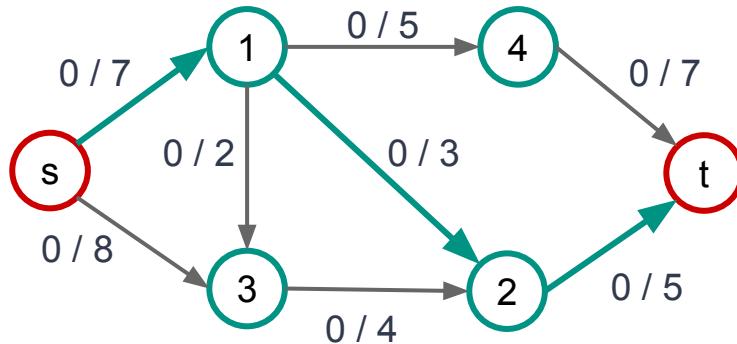
Graful rezidual  $G_f$





$BF(s)$  - în graful rezidual



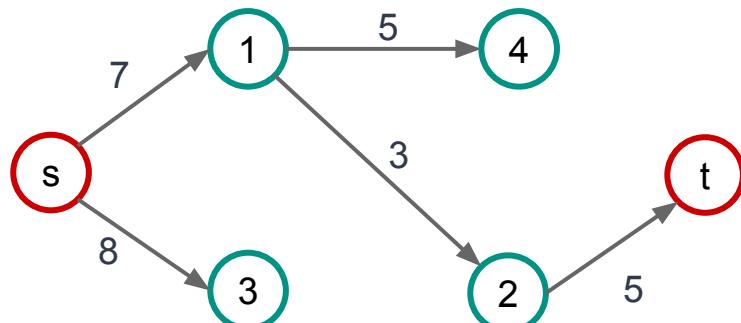
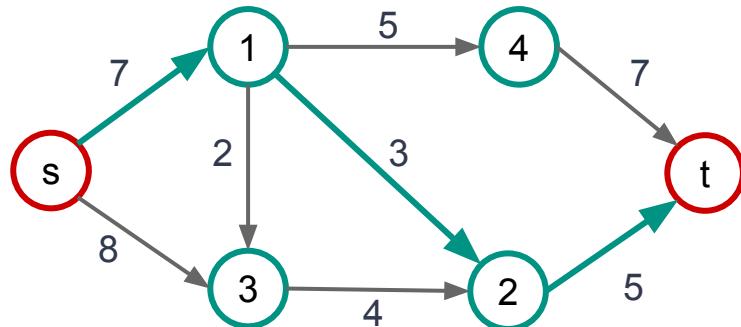


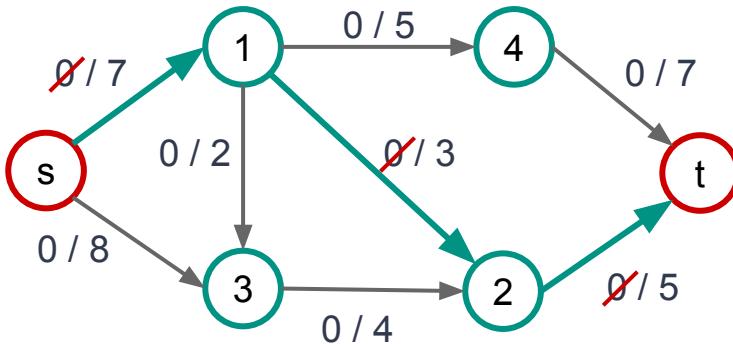
$BF(s)$  - în graful rezidual

Drumul de creștere  $[s, 1, 2, t]$  - capacitate reziduală 3

Revizuim fluxul

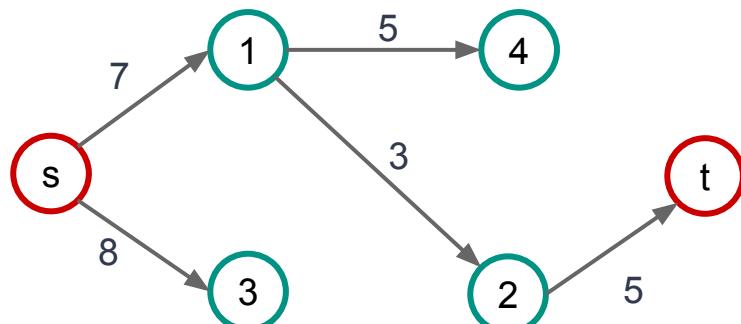
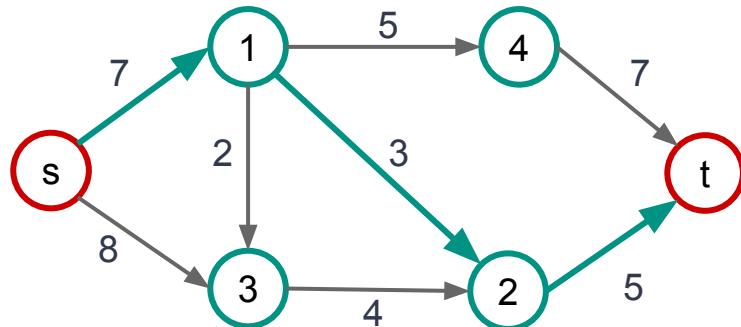
Graful rezidual  $G_f$





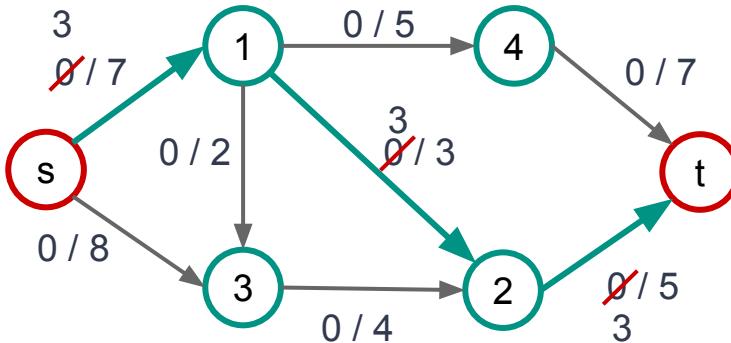
BF(s) - în graful rezidual

Graful rezidual  $G_f$



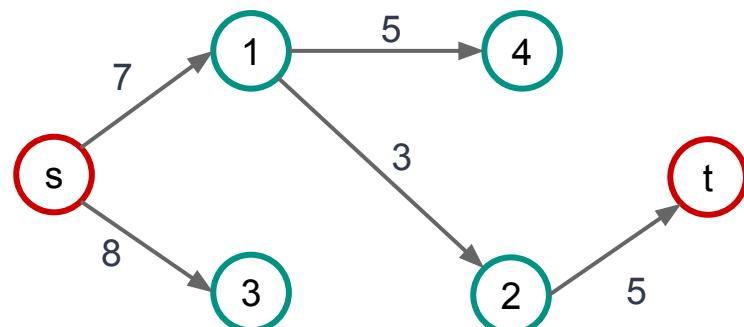
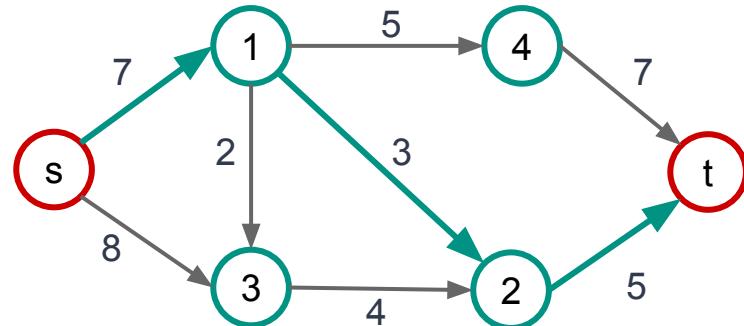
Drumul de creștere [s, 1, 2, t] - capacitate reziduală 3

Revizuim fluxul



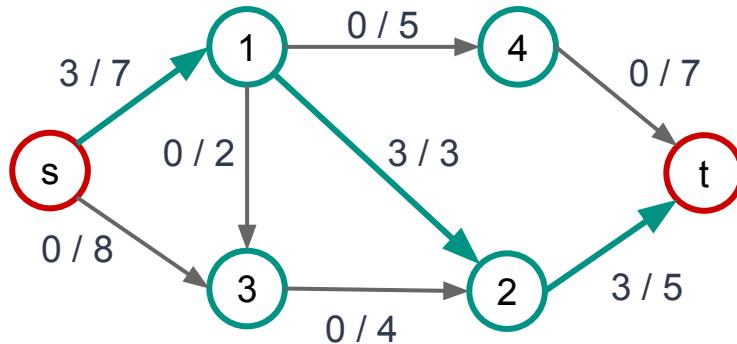
$BF(s)$  - în graful rezidual

Graful rezidual  $G_f$

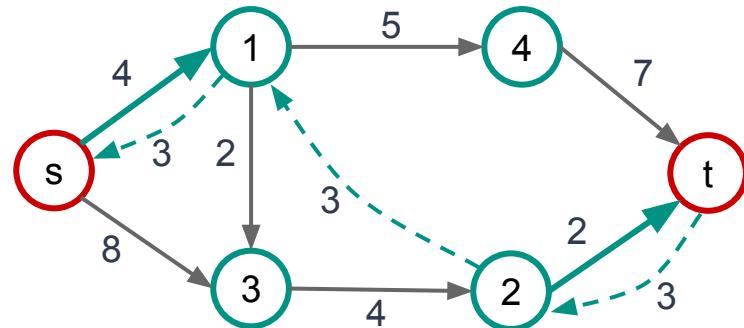


Drumul de creștere  $[s, 1, 2, t]$  - capacitate reziduală 3

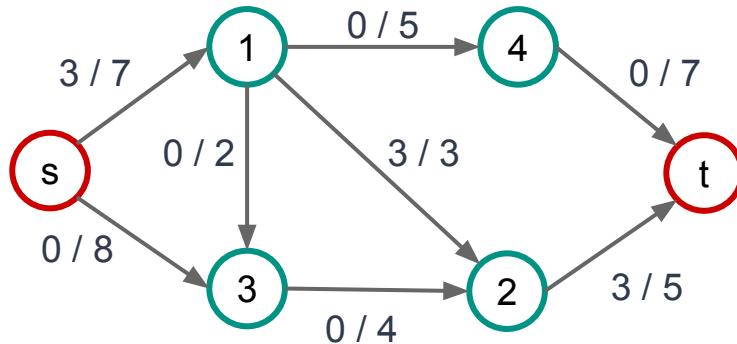
Revizuim fluxul



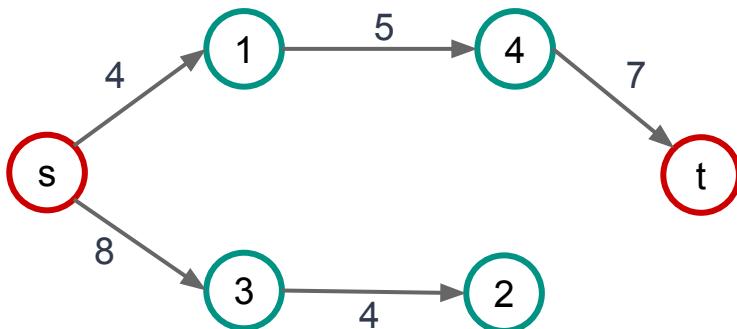
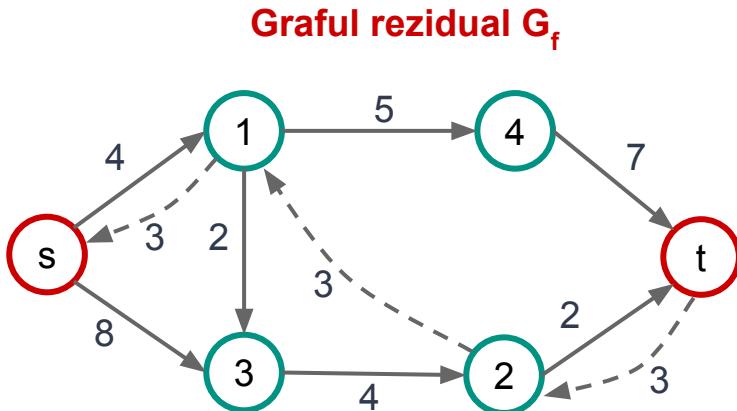
Graful rezidual  $G_f$

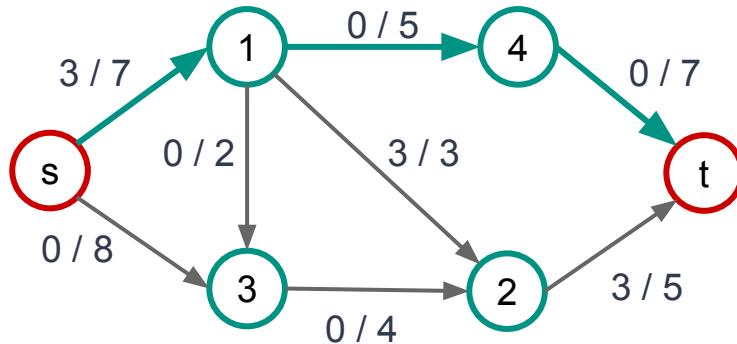


Actualizăm rețeaua reziduală

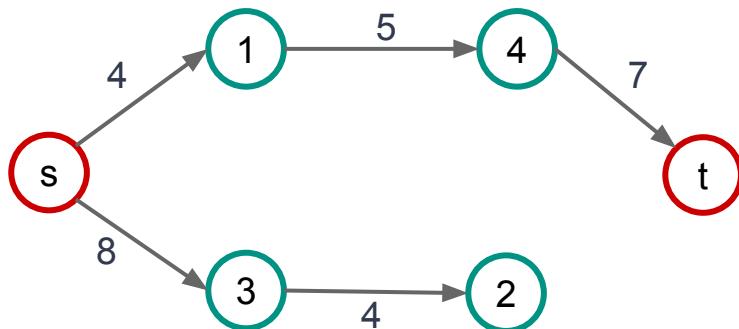
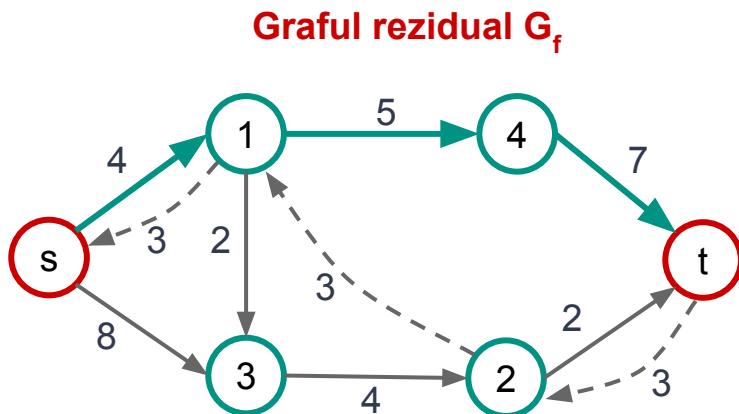


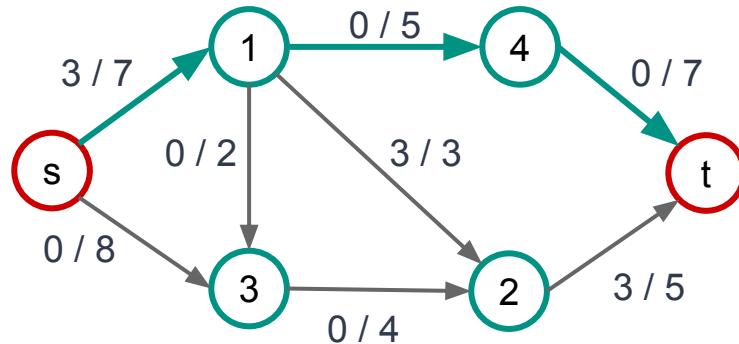
$BF(s)$  - în graful rezidual





$BF(s)$  - în graful rezidual



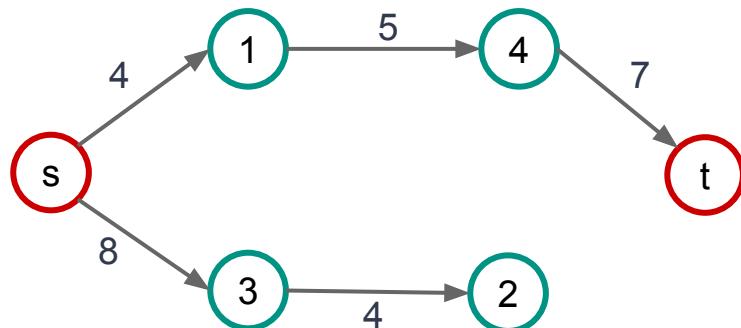
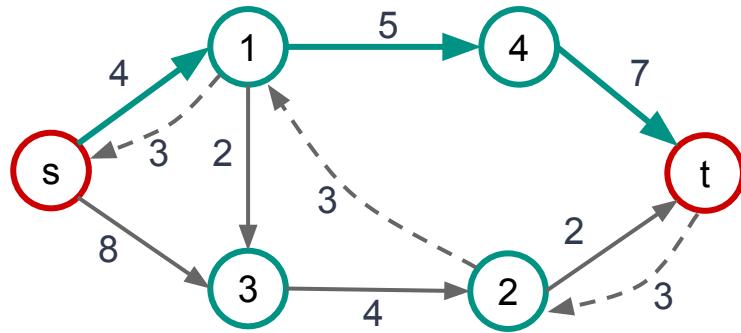


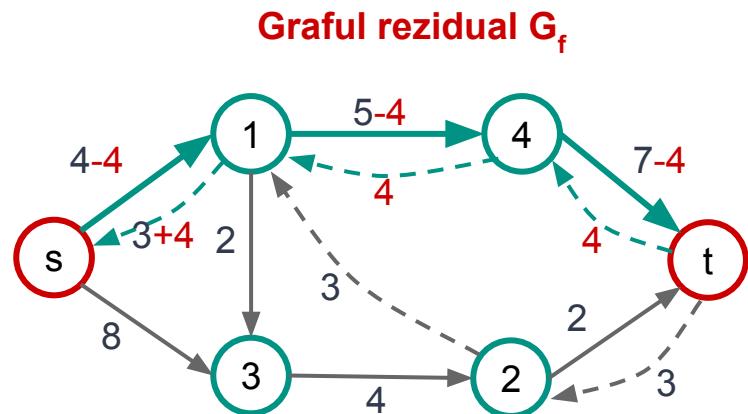
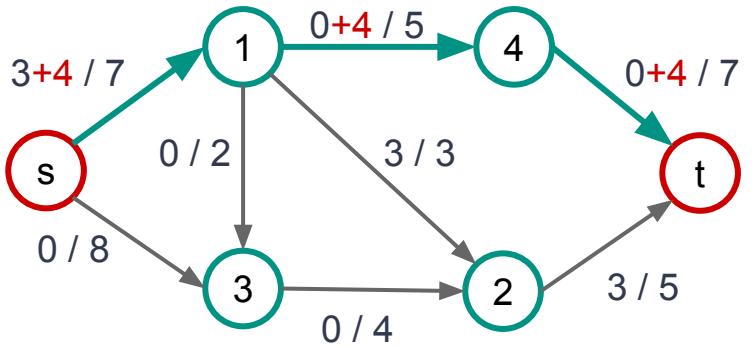
$BF(s)$  - în graful rezidual

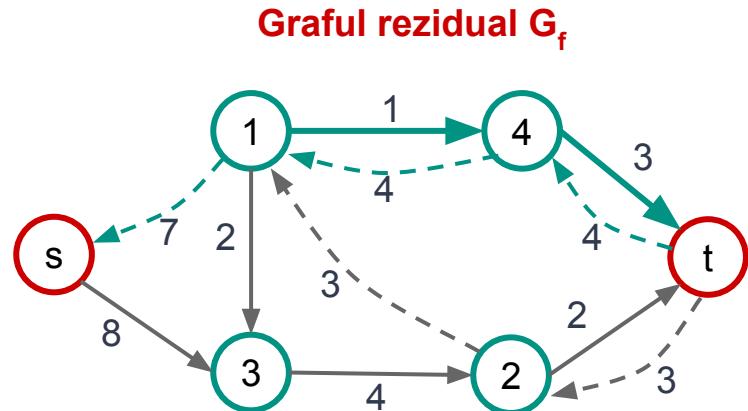
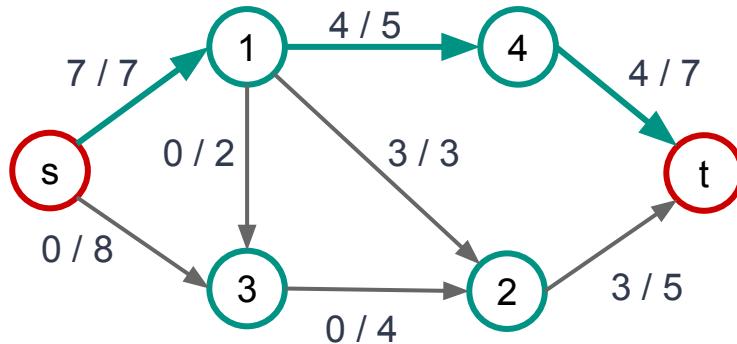
Drumul de creștere  $[s, 1, 4, t]$  - capacitate reziduală 4

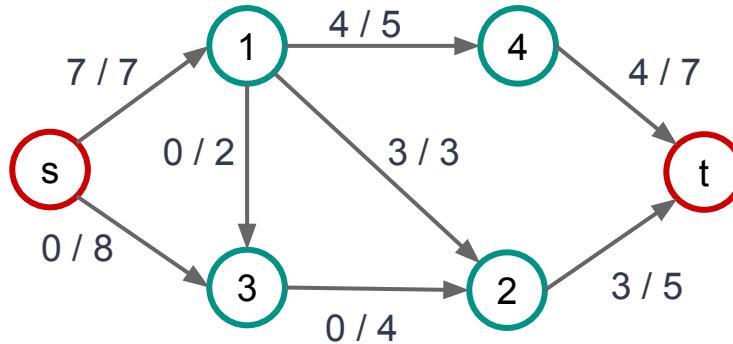
Revizuim fluxul

Graful rezidual  $G_f$

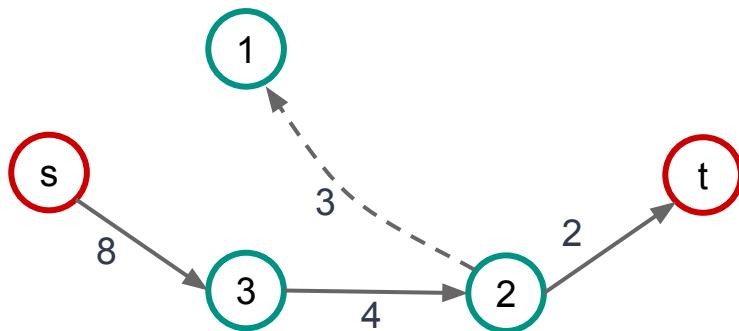
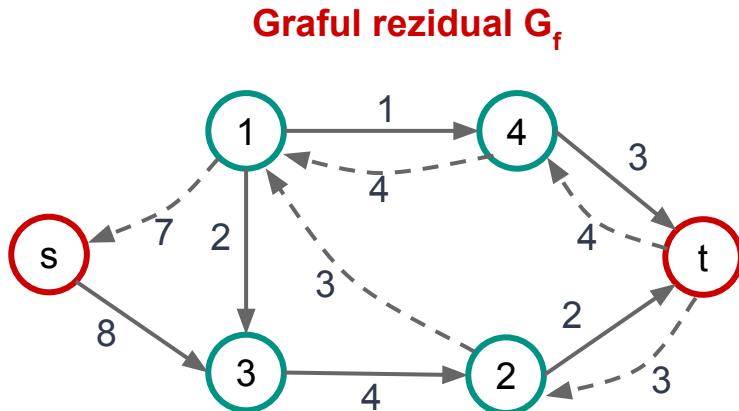


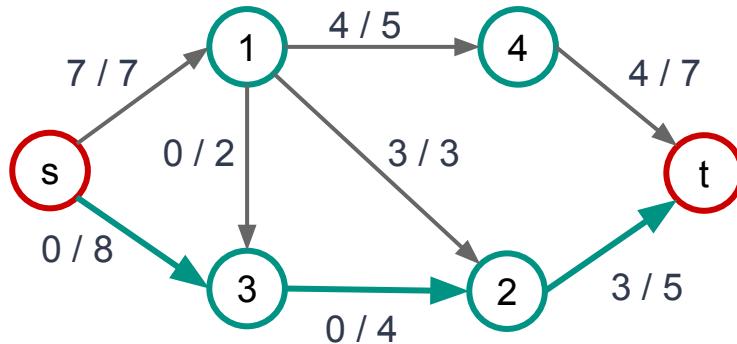




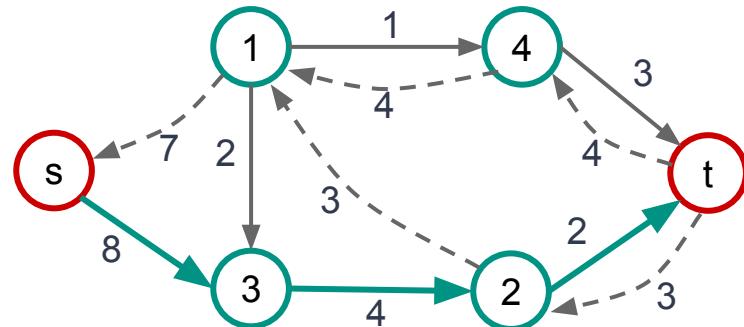


BF( $s$ ) - în graful rezidual

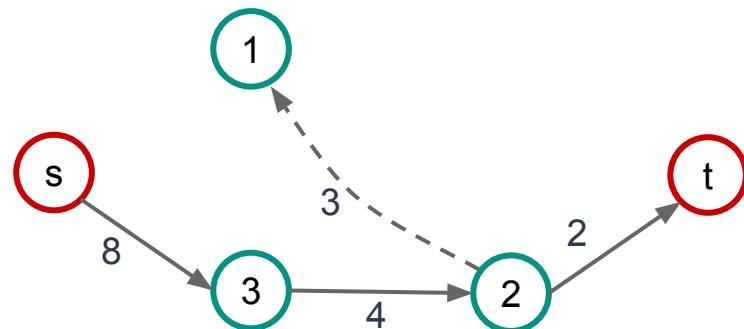




Graful rezidual  $G_f$

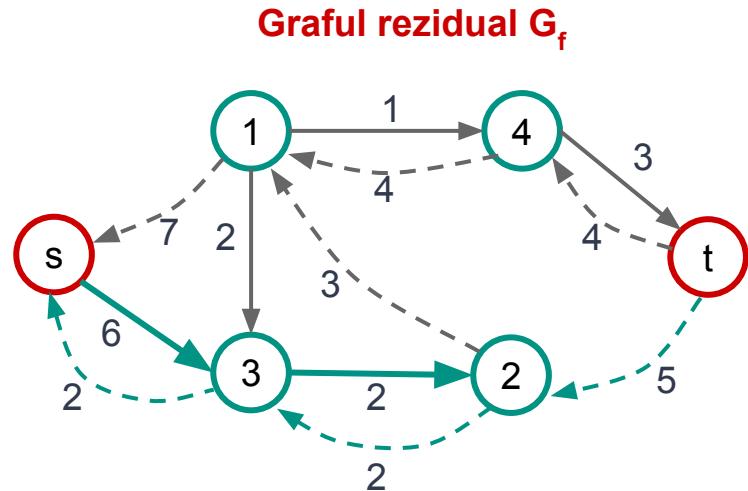
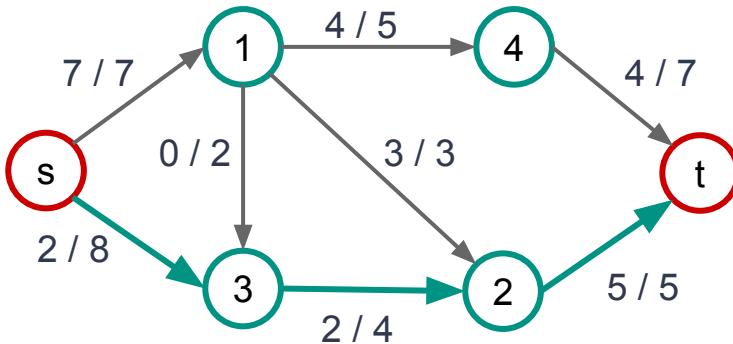


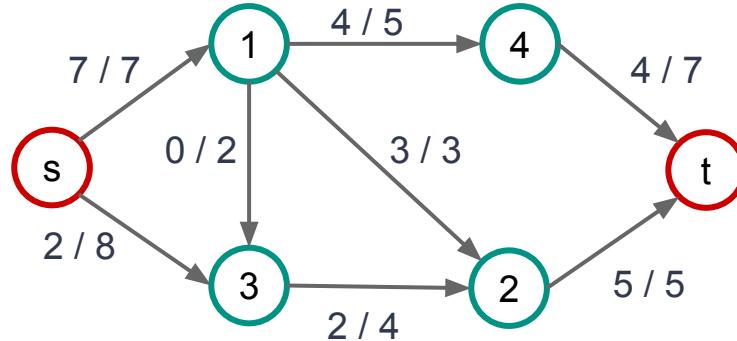
$BF(s)$  - în graful rezidual



Drumul de creștere  $[s, 3, 2, t]$  - capacitate reziduală 2

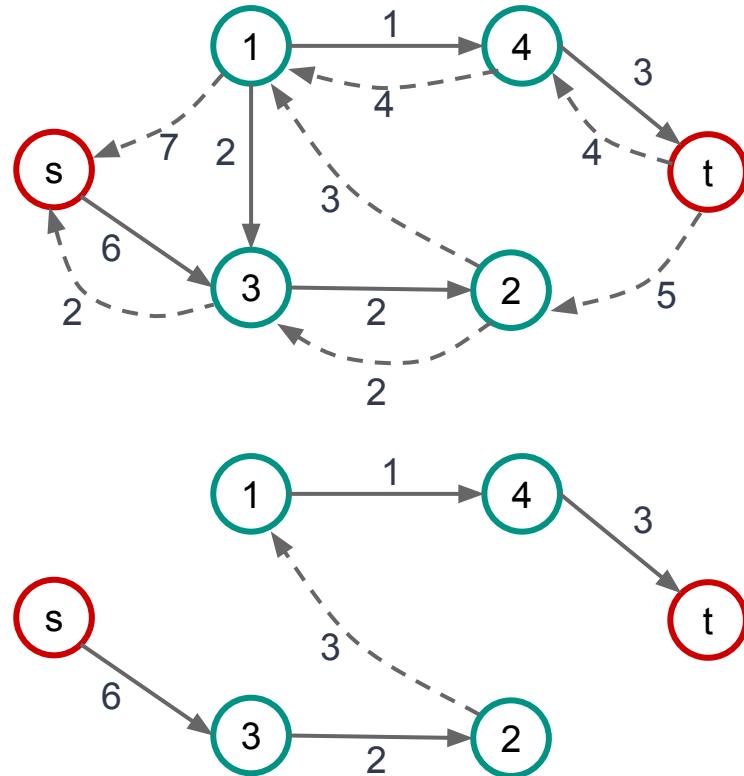
Revizuim fluxul

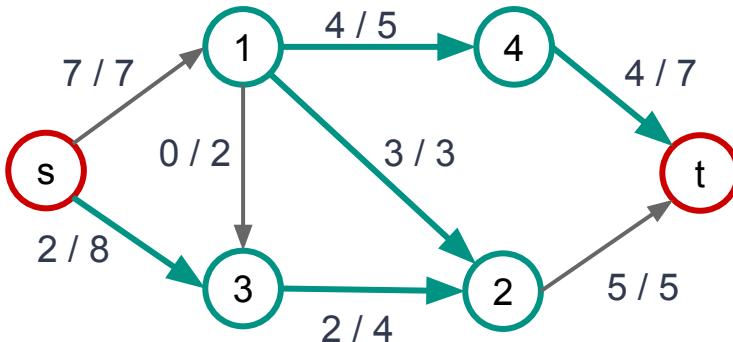




BF(s) - în graful rezidual

## Graful rezidual $G_f$

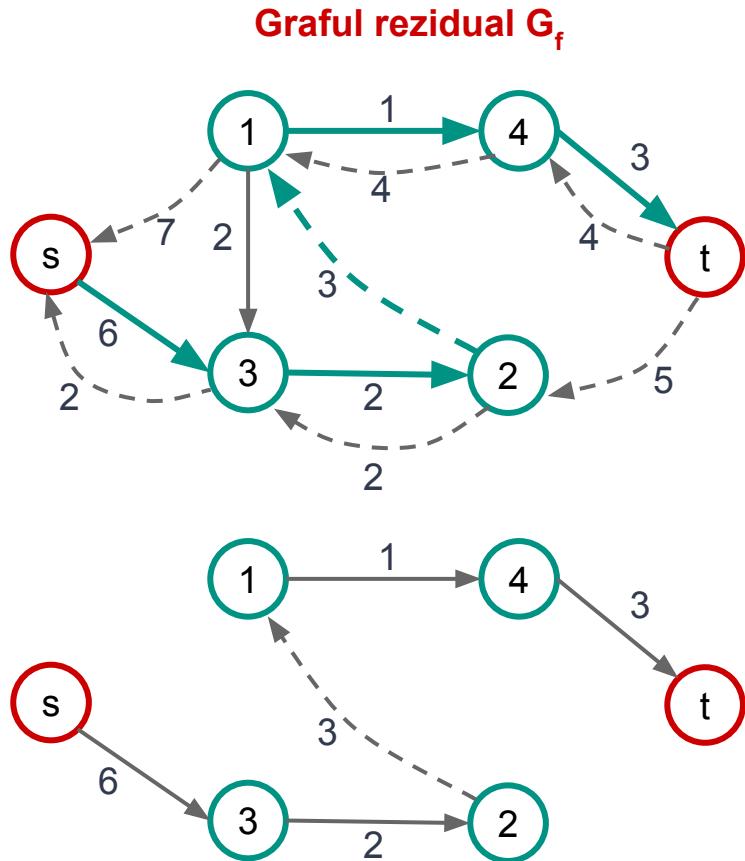


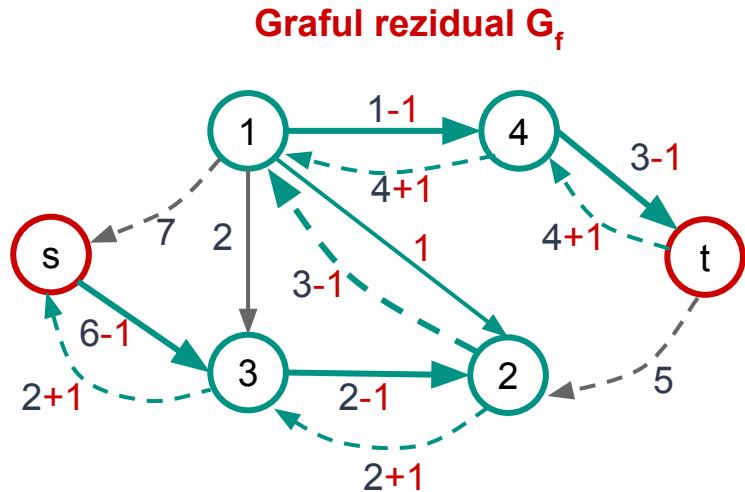
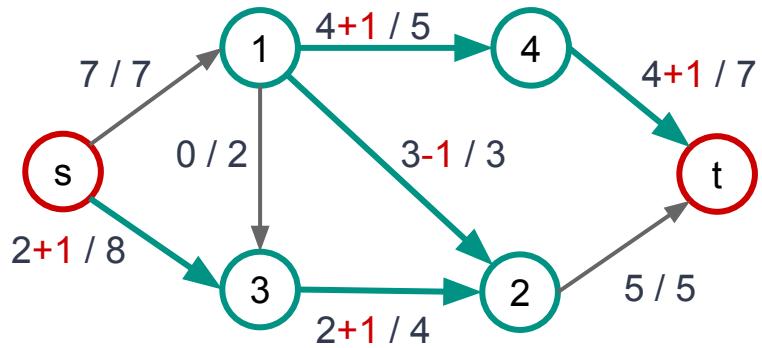


$BF(s)$  - în graful rezidual

Drumul de creștere  $[s, 3, 2, 1, 4, t]$  - capacitate reziduală 1

Revizuim fluxul





actualizare  $G_f$ :

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

$$c_f(e) \leftarrow c_f(e) - cfp$$

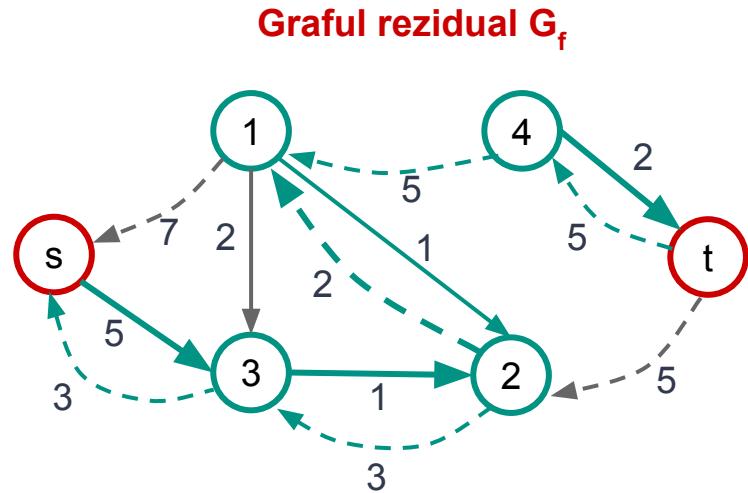
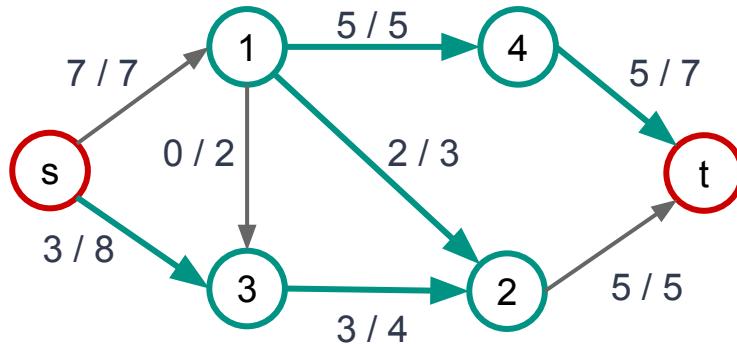
dacă  $c_f(e) = 0$  atunci

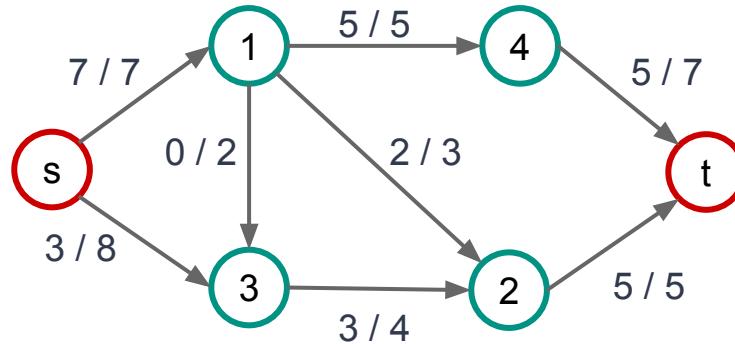
elimină  $e$  din  $G_f$  // se ignoră în parcurgere

$$c_f(e^{-1}) \leftarrow c_f(e^{-1}) + cfp$$

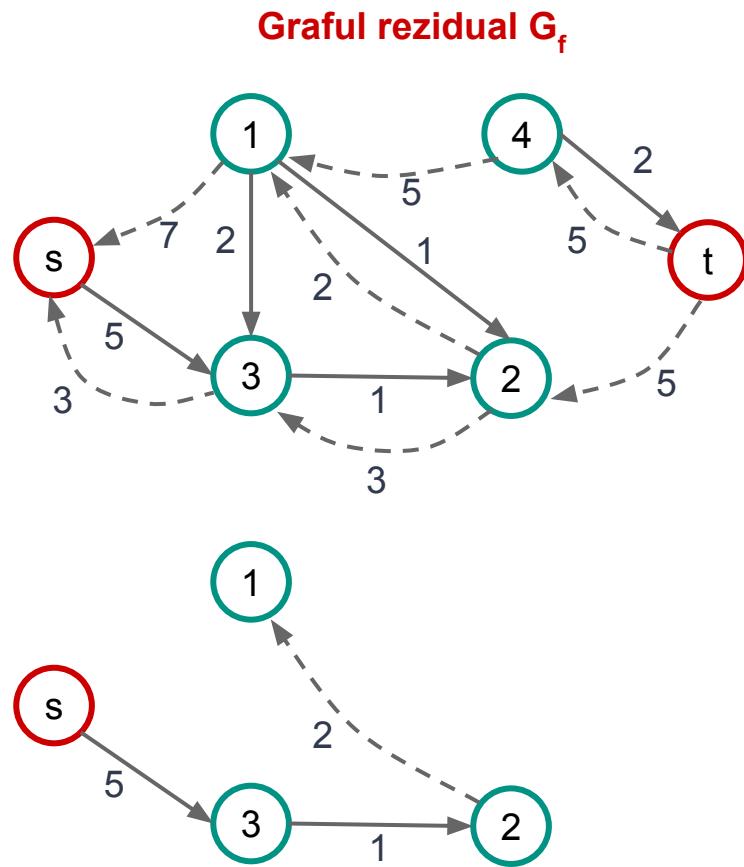
dacă  $c_f(e^{-1}) > 0$  și  $e^{-1} \notin G_f$  atunci

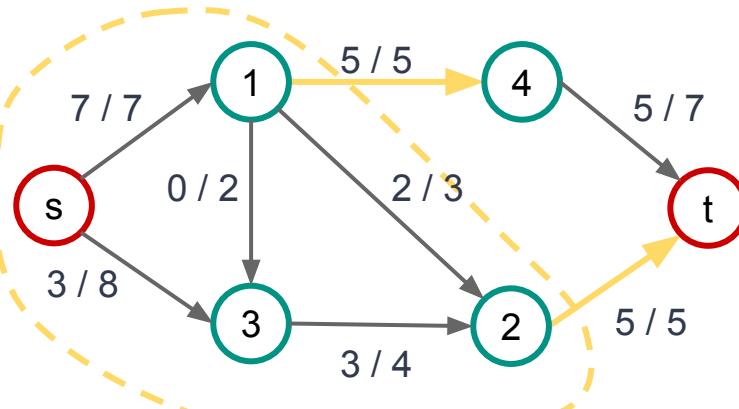
adaugă  $e^{-1}$  la  $G_f$





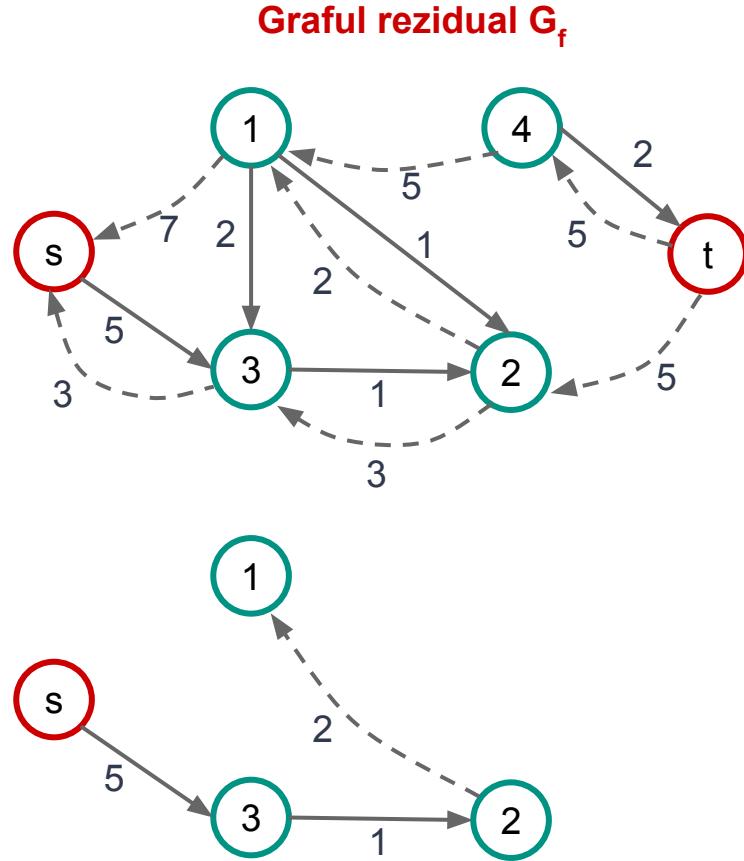
$BF(s)$  - în graful rezidual

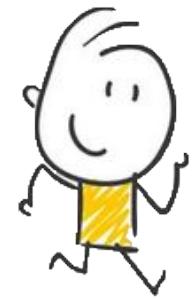




BF(s) - în graful rezidual

Nu mai există drum de creștere  $\Rightarrow$  s-t flux maxim (valoare 10) + s-t tăietură minimă (de capacitate tot 10, determinată de vârfurile accesibile din s în  $G_f$ :  $S = \{ s, 1, 3, 2 \}$ )





# Aplicație

Flux maxim  $\rightarrow$  Cuplaj maxim  
în grafuri bipartite

# Cuplaje



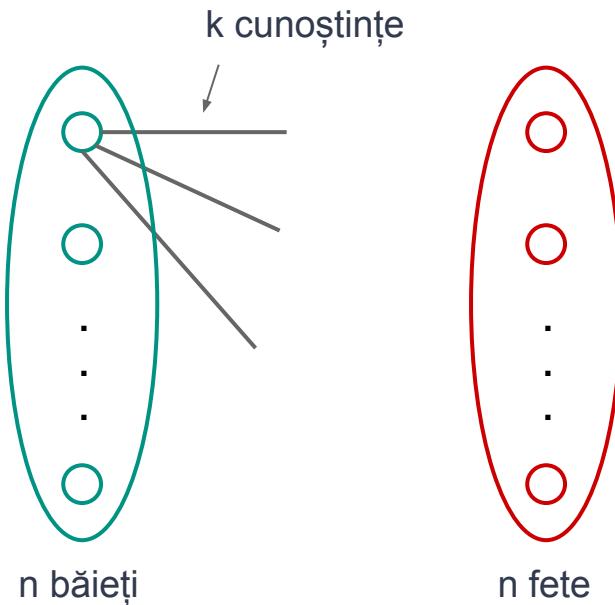
# Repere istorice. Aplicații

## □ Problema seratei (perechilor) - sec XIX

- $n$  fete,  $n$  băieți
- un băiat cunoaște exact  $k$  fete
- o fată cunoaște exact  $k$  băieți

# Repere istorice. Aplicații

## □ Problema seratei (perechilor)



# Repere istorice. Aplicații

## □ **Problema seratei (perechilor) - sec XIX**

- Se poate organiza o repriză de dans astfel încât fiecare participant să danseze cu o cunoștință a sa?

# Repere istorice. Aplicații

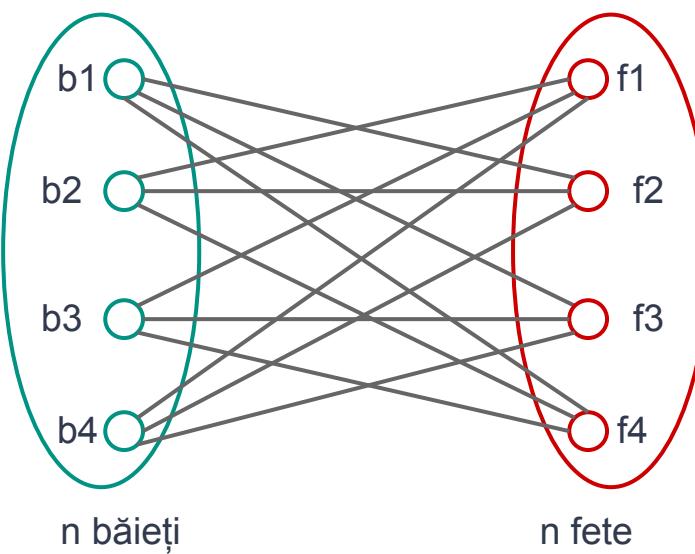
- **Problema seratei (perechilor) - sec XIX**
  - Se poate organiza o repriză de dans astfel încât fiecare participant să danseze cu o cunoștință a sa?
  - Se pot organiza **k** reprize de dans în care fiecare participant să danseze câte un dans cu fiecare cunoștință a sa?

# Repere istorice. Aplicații

## □ Problema seratei (perechilor) - sec XIX

$$n = 4$$

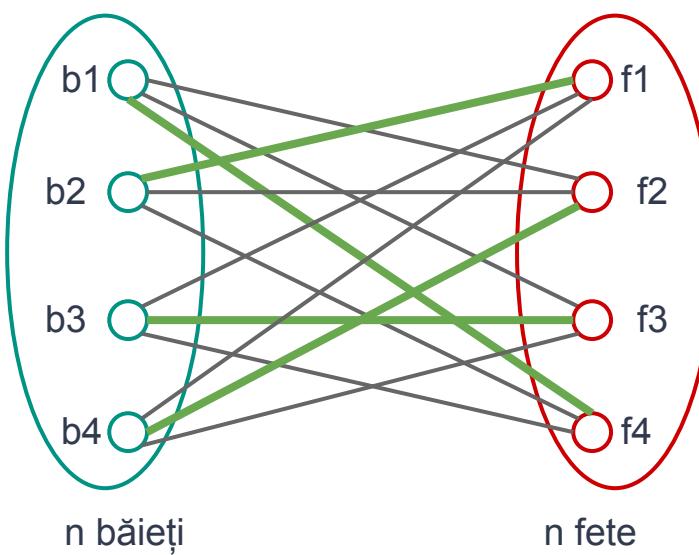
$$k = 3$$



# Repere istorice. Aplicații

## □ O repreză de dans

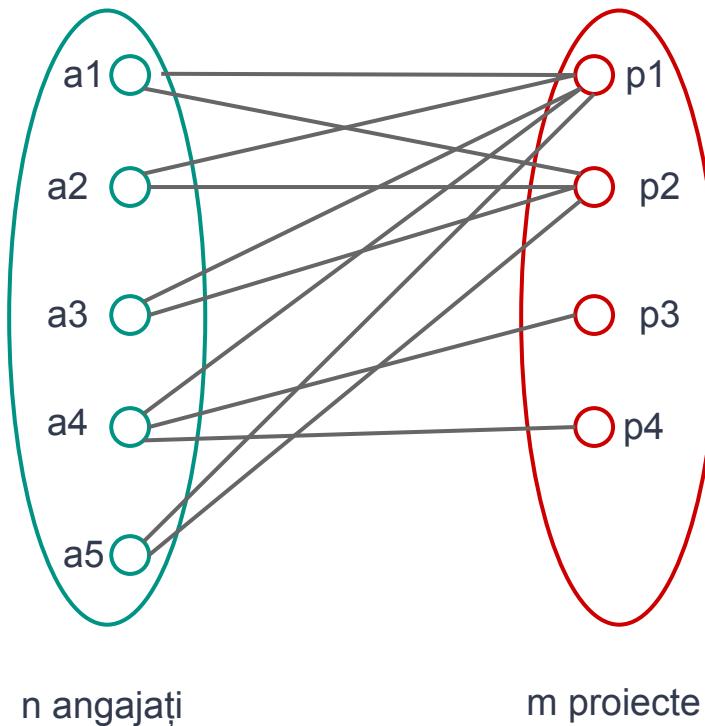
b<sub>1</sub>, f<sub>4</sub>  
b<sub>2</sub>, f<sub>1</sub>  
b<sub>3</sub>, f<sub>3</sub>  
b<sub>4</sub>, f<sub>2</sub>



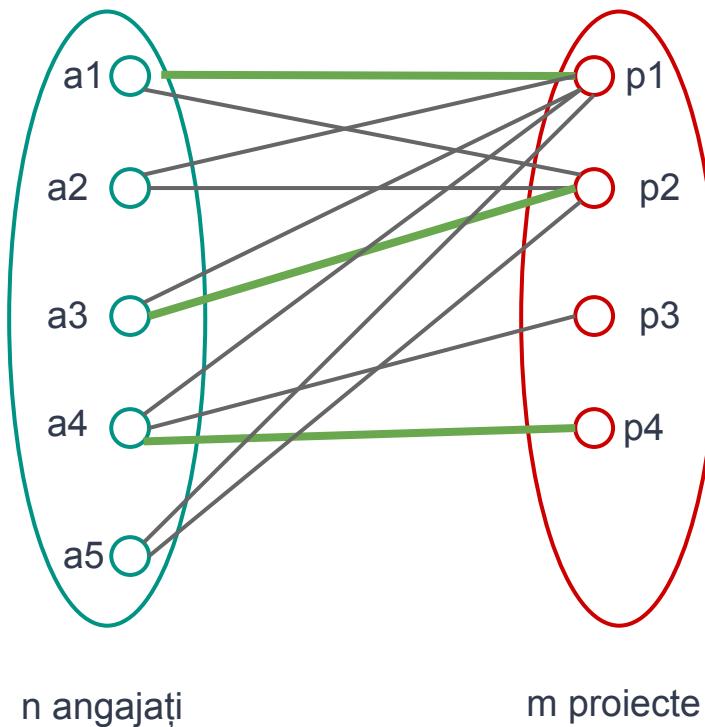
# Repere istorice. Aplicații

- **Problema seratei**
- **Organizare de competiții**
- **Probleme de repartiție**
  - lucrători - locuri de muncă
  - profesori - examene / conferințe
  - **Problema orarului**

# Alte aplicații



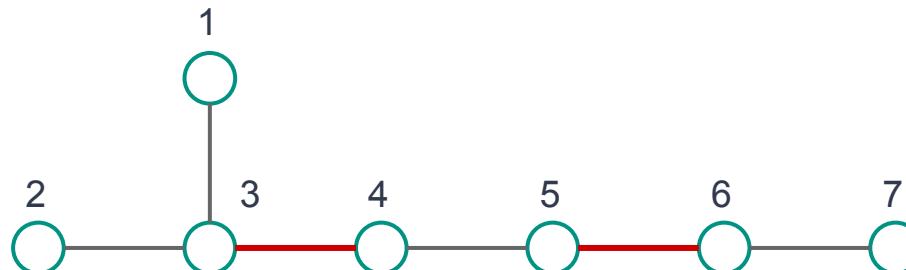
# Alte aplicații



# Cuplaje

Fie  $G = (V, E)$  un graf și  $M \subseteq E$ .

- $M$  se numește cuplaj** dacă orice două muchii din  $M$  sunt neadiacente.

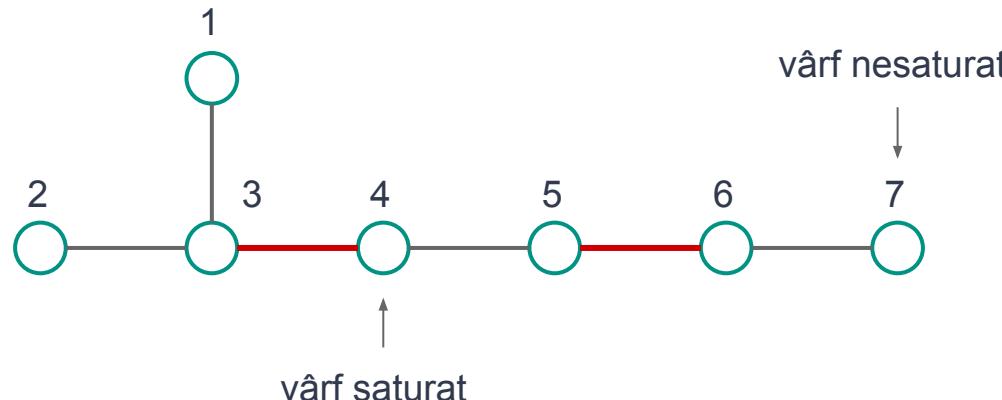


cuplaj  $M = \{ \{3, 4\}, \{5, 6\} \}$

# Cuplaje

Fie  $G = (V, E)$  un graf și  $M \subseteq E$ .

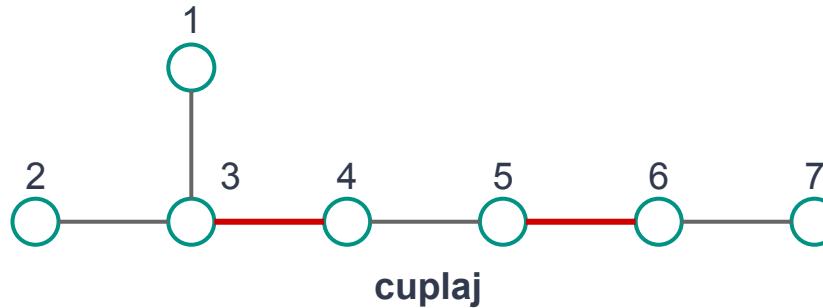
- $M$  se numește cuplaj** dacă orice două muchii din  $M$  sunt neadiacente.
- $V(M) =$  mulțimea vârfurilor  **$M$ -saturate**
- $V(G) - V(M) =$  mulțimea vârfurilor  **$M$ -nesaturate**



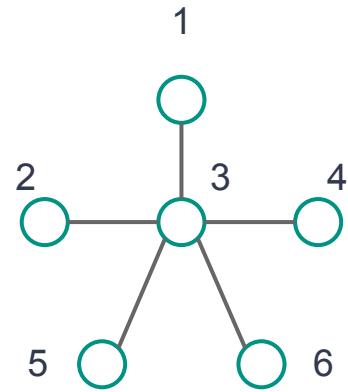
# Cuplaje

- Un cuplaj  $M^*$  se numește **cuplaj de cardinal maxim (cuplaj maxim)**:

$$|M^*| \geq |M|, \quad \forall M \subseteq E \text{ cuplaj}$$



# Cuplaje



cuplaj de cardinal maxim?

# Grafuri bipartite

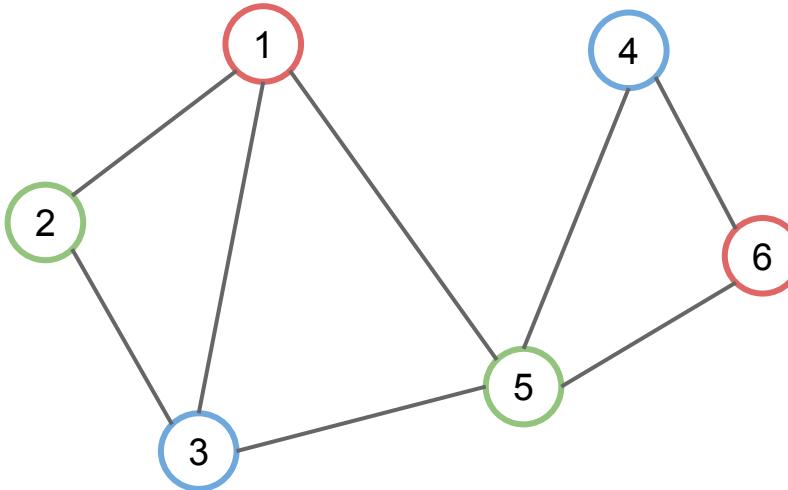
# Colorări ale grafurilor

Fie  $G = (V, E)$  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.

# Colorări ale grafurilor



**3-colorabil, dar nu și 2-colorabil (!)**

# Graf bipartit

$G = (V, E)$  graf neorientat s. n. **bipartit**  $\Leftrightarrow$  există o partiție a lui  $V$  în două submulțimi  $V_1$  și  $V_2$  (**partiție**) cu:

$$V = V_1 \cup V_2$$

$$V_1 \cap V_2 = \emptyset$$

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

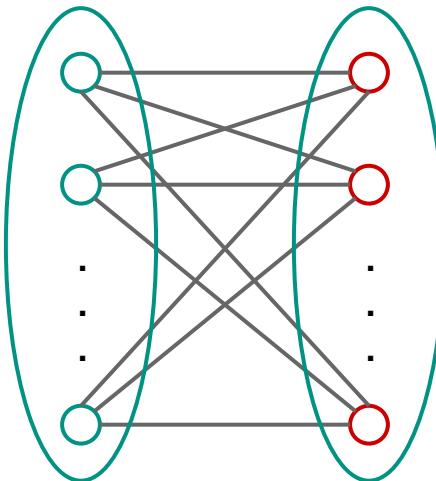
Notăm  $G = (V_1 \cup V_2, E)$

# Graf bipartit

$G = (V, E)$  graf neorientat s. n. **bipartit complet**  $\Leftrightarrow$

este bipartit și  $E = \{ xy \mid x \in V_1, y \in V_2 \}$

Notăm  $K_{p, q}$  dacă  $p = |V_1|$  și  $q = |V_2|$



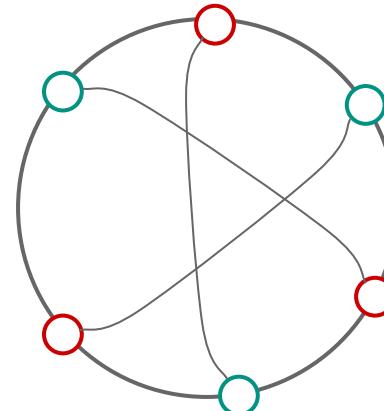
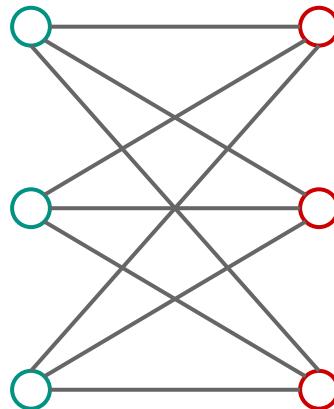
# Graf bipartit

$G = (V, E)$  graf neorientat s. n. **bipartit complet**  $\Leftrightarrow$

este bipartit și  $E = \{ xy \mid x \in V_1, y \in V_2 \}$

Notăm  $K_{p, q}$  dacă  $p = |V_1|$  și  $q = |V_2|$

□  $K_{3, 3}$



# Graf bipartit

## Observație

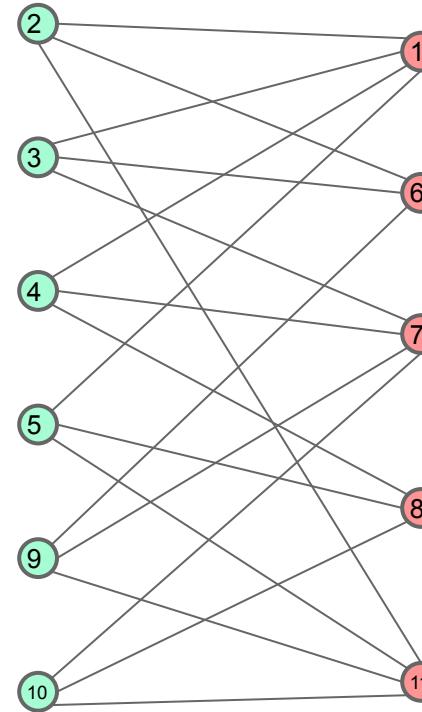
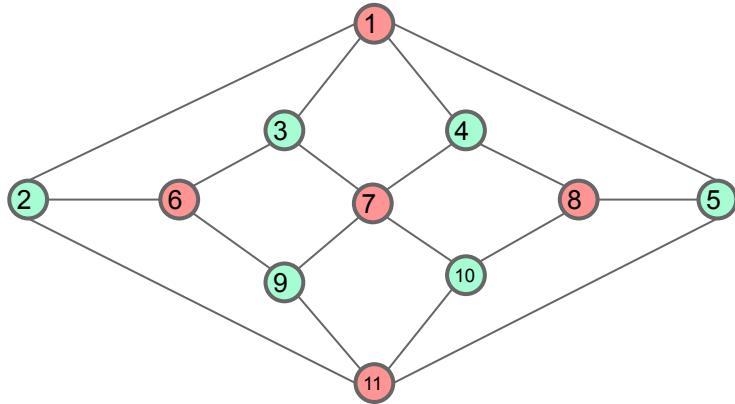
$G = (V, E)$  bipartit  $\Leftrightarrow$

există o 2-colorare proprie a vârfurilor (**bicolorare**):

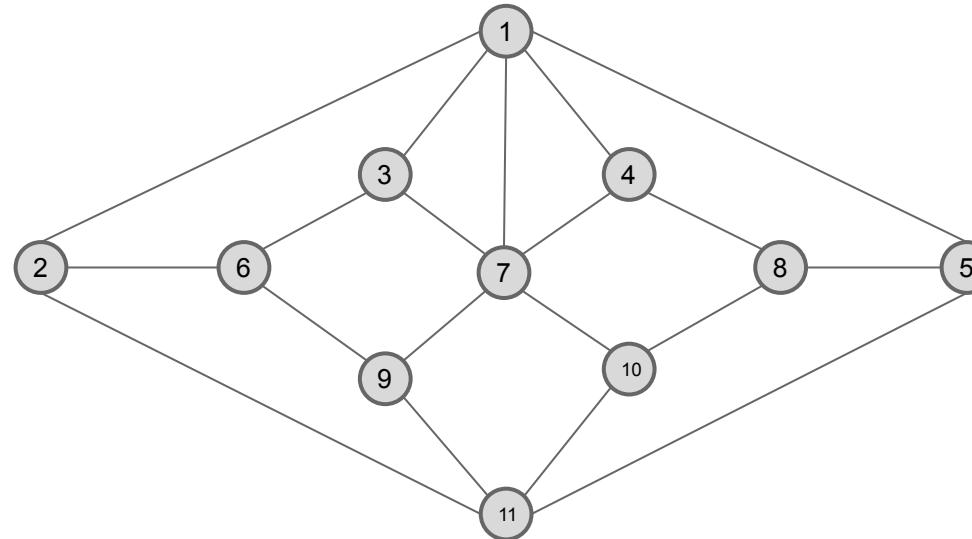
$$c : V \rightarrow \{ 1, 2 \}$$

( i. e. astfel încât, pentru orice muchie  $e = xy \in E$  avem  $(x) \neq c(y)$  )

# Graf bipartit

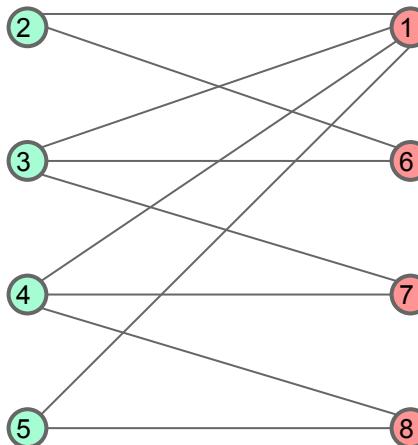


# Graf bipartit



nu este bipartit

# Modelare



Profesori    *predau la*    Cursuri

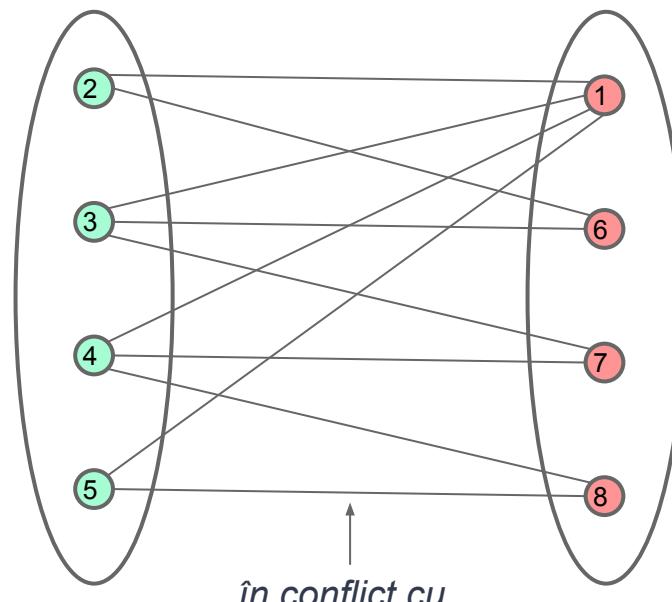
Candidați    *depun CV la*    Joburi

# Aplicații

**Graf de conflicte** (exemplu - substanțe care interacționează, activități incompatibile, relații în rețele de socializare)

**submulțimi fără conflicte** (evenimente compatibile)

**submulțimi fără conflicte** (evenimente compatibile)



*în conflict cu*

Cuplaje, rețele ...

# Aplicații p-colorări

**Exemplu** - 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)

# Aplicații p-colorări

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

Conf. 1: interval (1, 4)

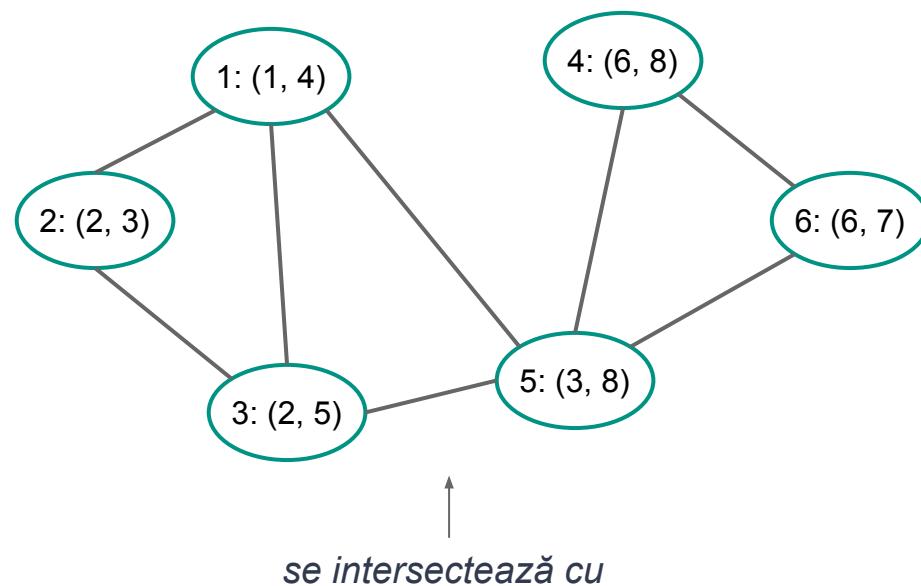
Conf. 2: interval (2, 3)

Conf. 3: interval (2, 5)

Conf. 4: interval (6, 8)

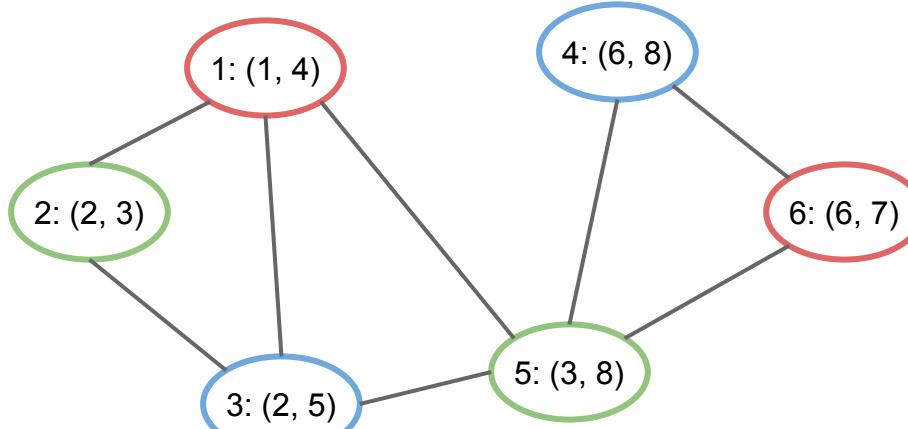
Conf. 5: interval (3, 8)

Conf. 6: interval (6, 7)



# Aplicații p-colorări

Graful intersecției intervalelor este 3-colorabil



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

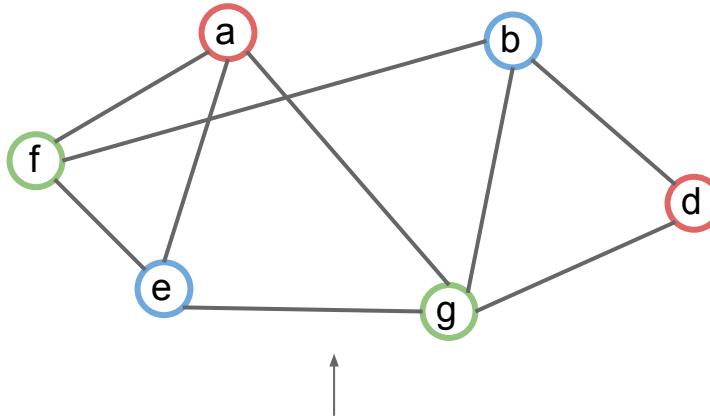
**Sala 1:** (1, 4), (6, 7)

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

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

# Aplicații p-colorări

## Alocare de registri (Register allocation problem)



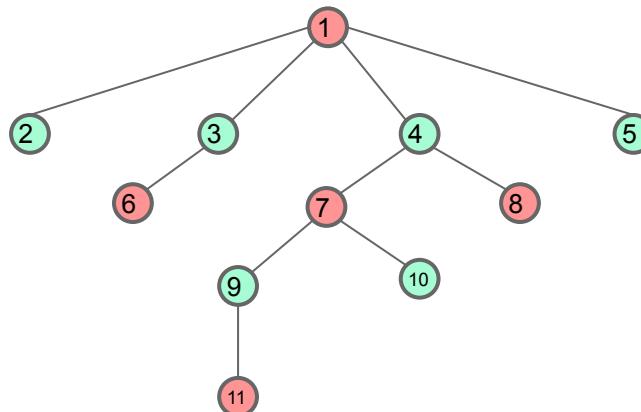
*pot fi simultan active  
(nu pot fi memorate în același regisztr)*

- Numărul de culori = numărul de registri
- Vârfuri de aceeași culoare = pot fi memorate în același regisztr

# Graf bipartit

## Propoziție

Un arbore este graf bipartit.



- ← - Fixăm o rădăcină  
- Colorăm alternativ nivelurile

# 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

# Graf bipartit

## Teorema König - Caracterizarea grafurilor bipartite

**Demonstrație "⇒":**

Evident, deoarece un ciclu impar nu poate fi colorat propriu cu două culori.

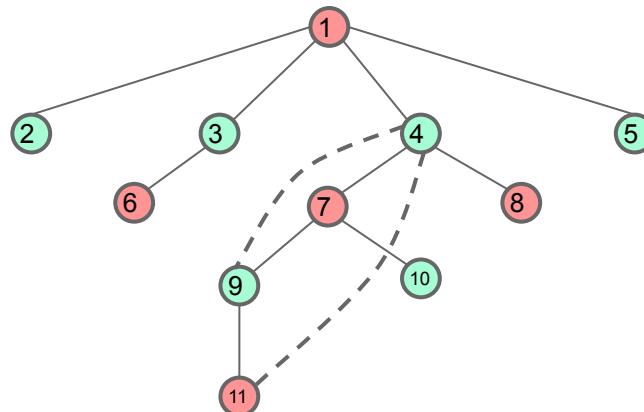
# Graf bipartit

**Teorema König - Caracterizarea grafurilor bipartite**

**Demonstrație "↔":** Presupunem  $G$  conex.

Colorăm propriu cu 2 culor un arbore parțial  $T$  al său.

Orice altă muchie  $uv$  din graf are extremitățile colorate diferit, deoarece



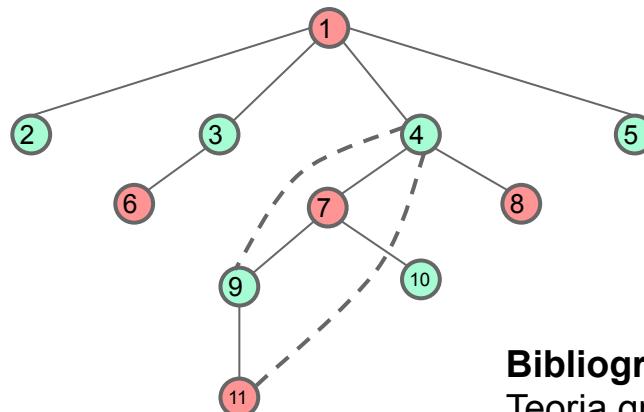
# Graf bipartit

## Teorema König - Caracterizarea grafurilor bipartite

**Demonstrație** " $\Leftarrow$ ": Presupunem  $G$  conex.

Colorăm propriu cu 2 culor un arbore parțial  $T$  al său.

Orice altă muchie  $uv$  din graf are extremitățile colorate diferit, deoarece formează un ciclu elementar cu lanțul de la  $u$  la  $v$  din arbore, iar acest ciclu are lungime pară, deci  $u$  și  $v$  se află pe niveluri de paritate diferită în  $T$ .



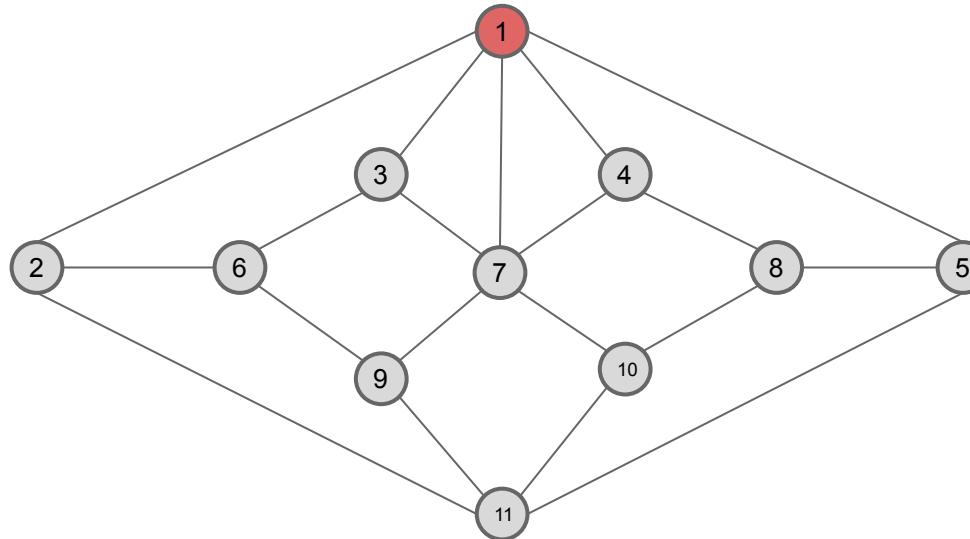
**Bibliografie:** DR Popescu Combinatorică și Teoria grafurilor (Teorema 4.18)

# Graf bipartit

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

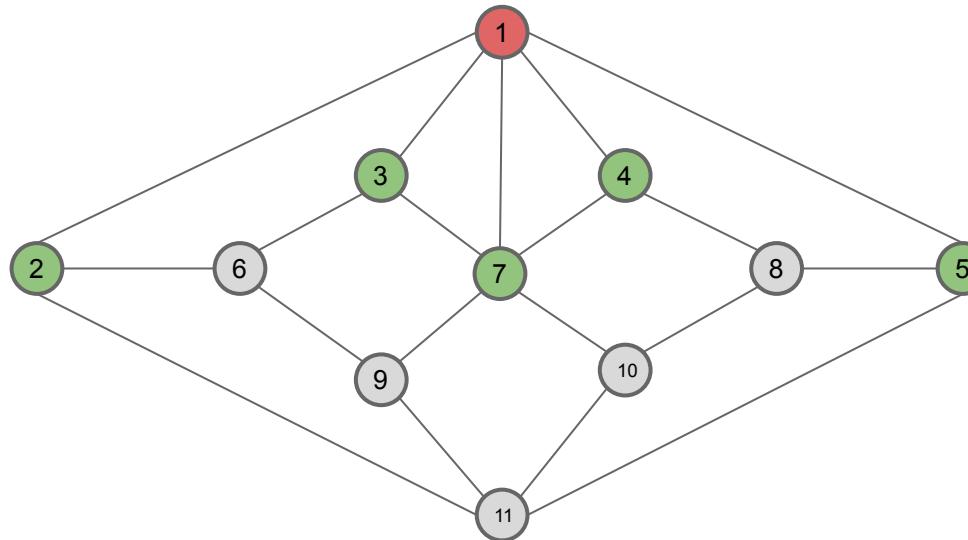
- Colorăm (propriu) cu 2 culori un arbore parțial al său, printr-o **parcursere** (colorăm orice vecin  $j$  nevizitat al vârfului curent  $i$  cu o culoare diferită de cea a lui  $i$ )
- Testăm dacă celelalte muchii - de la  $i$  la **vecini  $j$  deja vizitați** (colorați) au extremitățile  $i$  și  $j$  colorate diferit

# Exemplu de test - graf bipartit BF

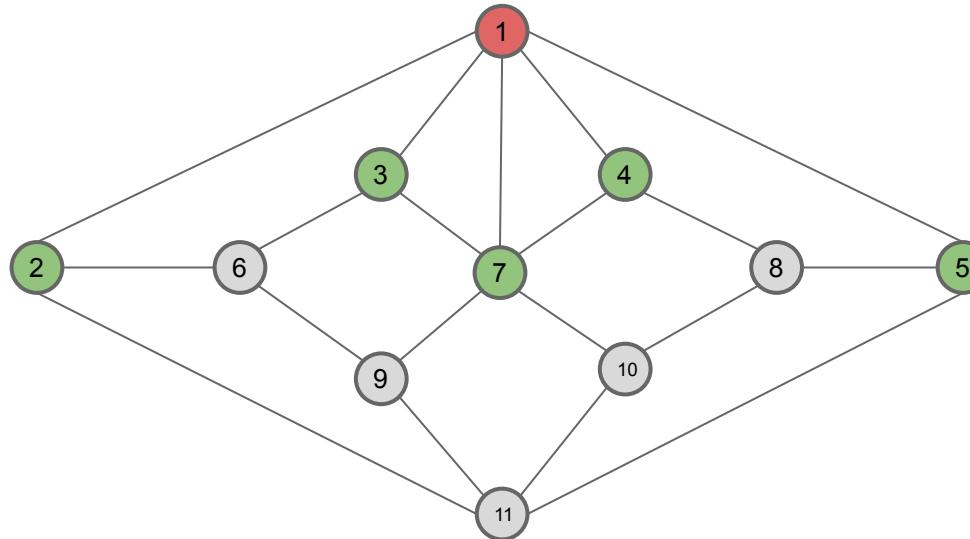


$i = 1$

# Exemplu de test - graf bipartit BF

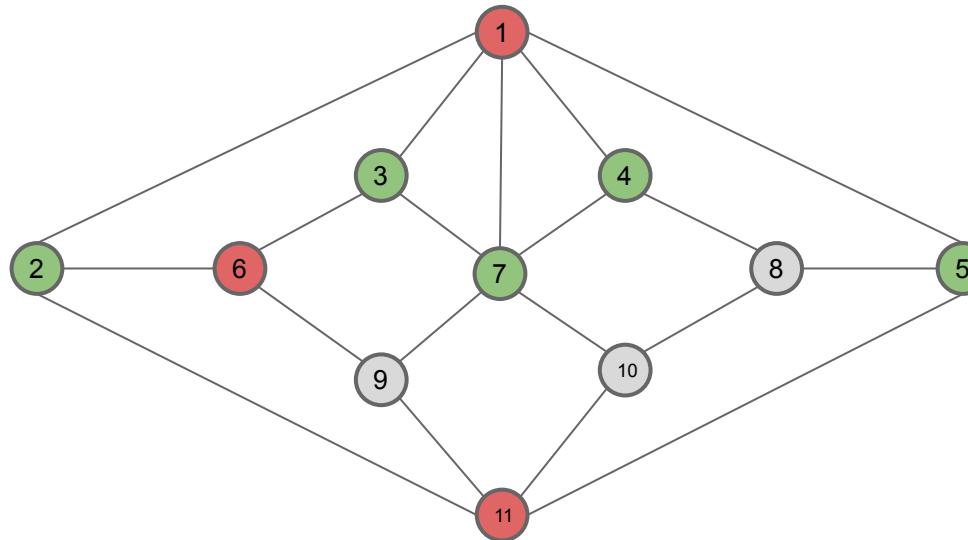


# Exemplu de test - graf bipartit BF

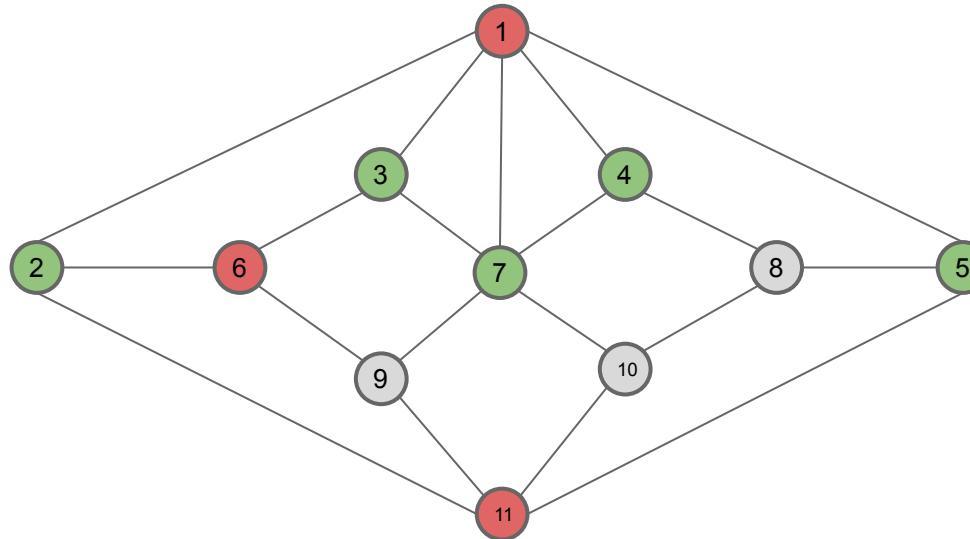


$i = 2$

# Exemplu de test - graf bipartit BF

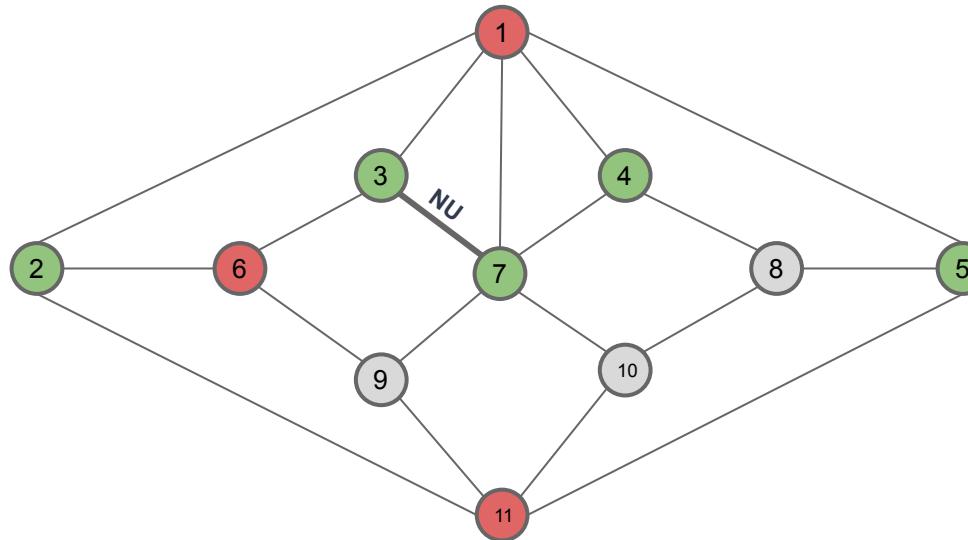


# Exemplu de test - graf bipartit BF



$i = 3$

# Exemplu de test - graf bipartit BF



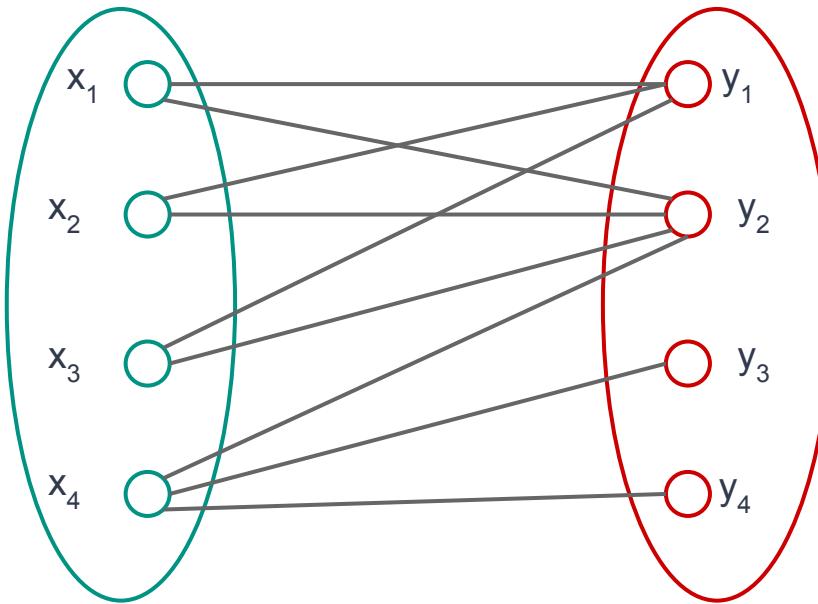
# Algoritm

Flux maxim → Cuplaj maxim  
în grafuri bipartite

# Algoritm de determinare a unui cuplaj maxim

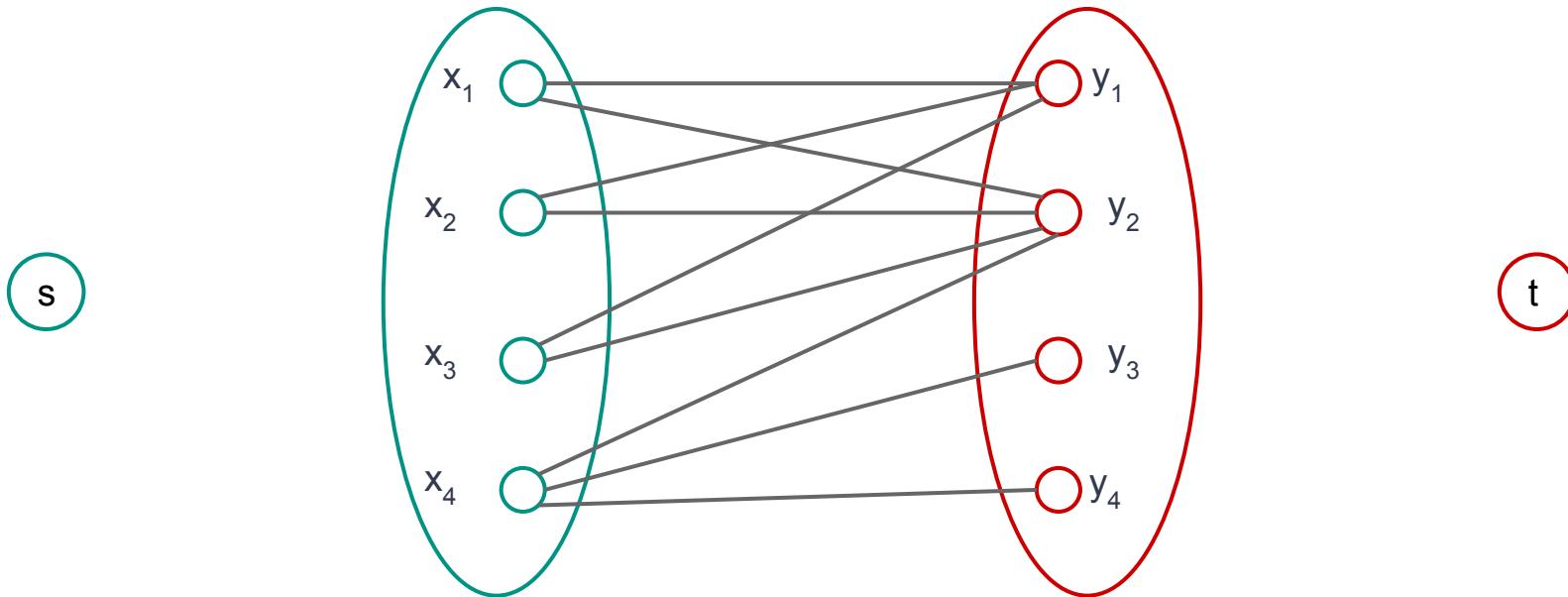
- Reducem problema determinării unui cuplaj maxim într-un graf bipartit  $G$  la determinarea unui flux maxim într-o rețea de transport asociată lui  $G$
- Construim rețeaua de transport  $N_G$  asociată lui  $G$  astfel:

# Algoritm de determinare a unui cuplaj maxim



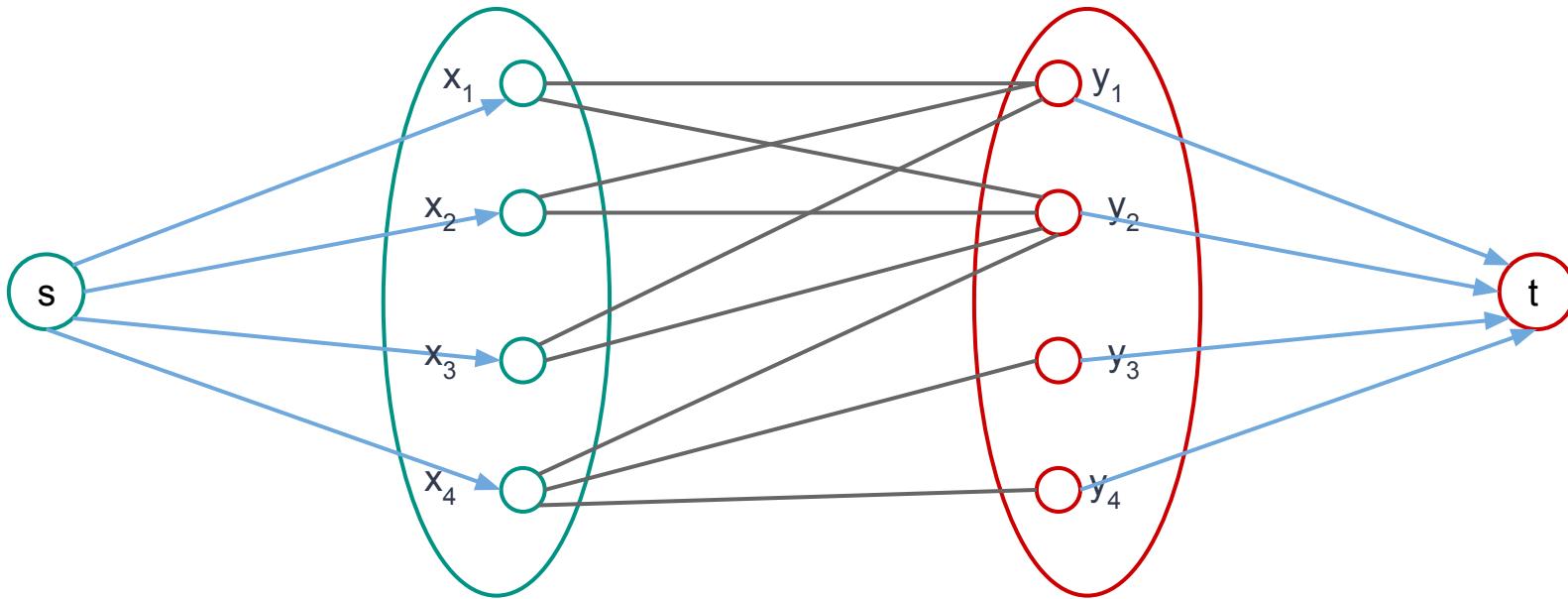
# Algoritm de determinare a unui cuplaj maxim

Adăugăm două noduri noi  $s$  și  $t$



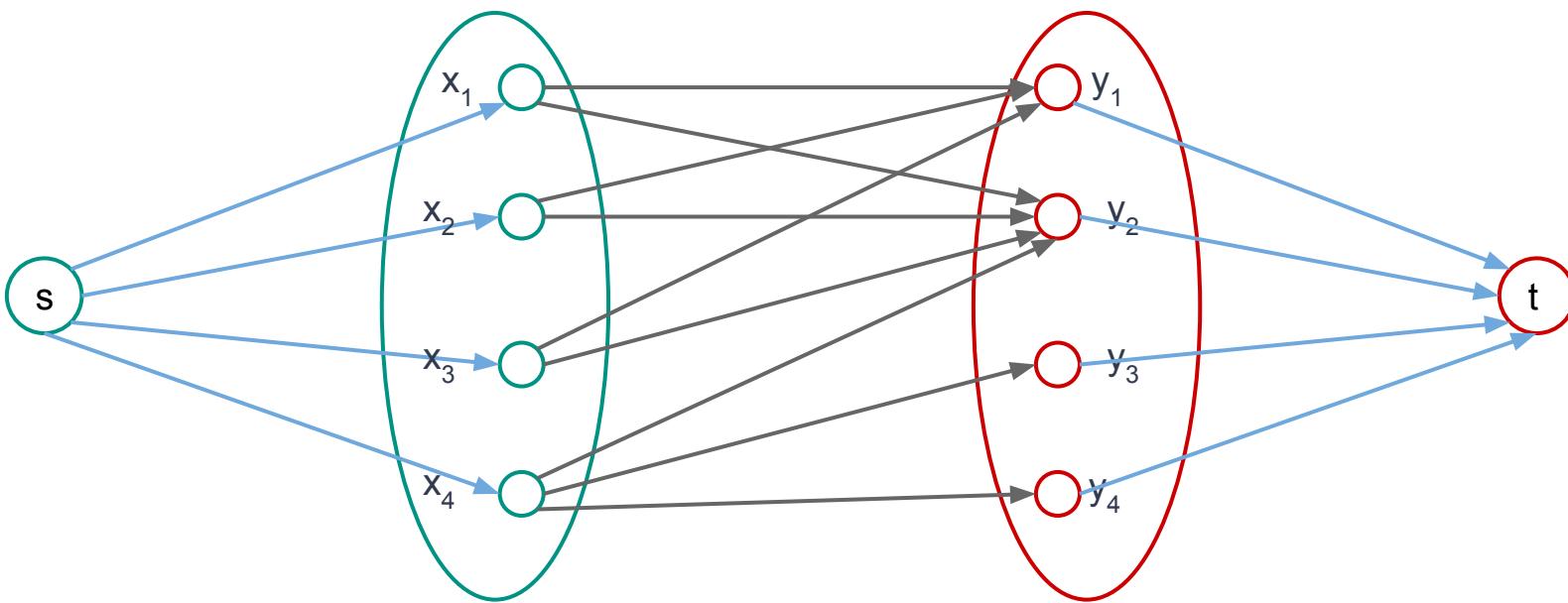
# Algoritm de determinare a unui cuplaj maxim

Adăugăm arce  $(s, x_i)$ , pentru  $x_i \in X$  și  $(y_j, t)$  pentru  $y_j \in Y$



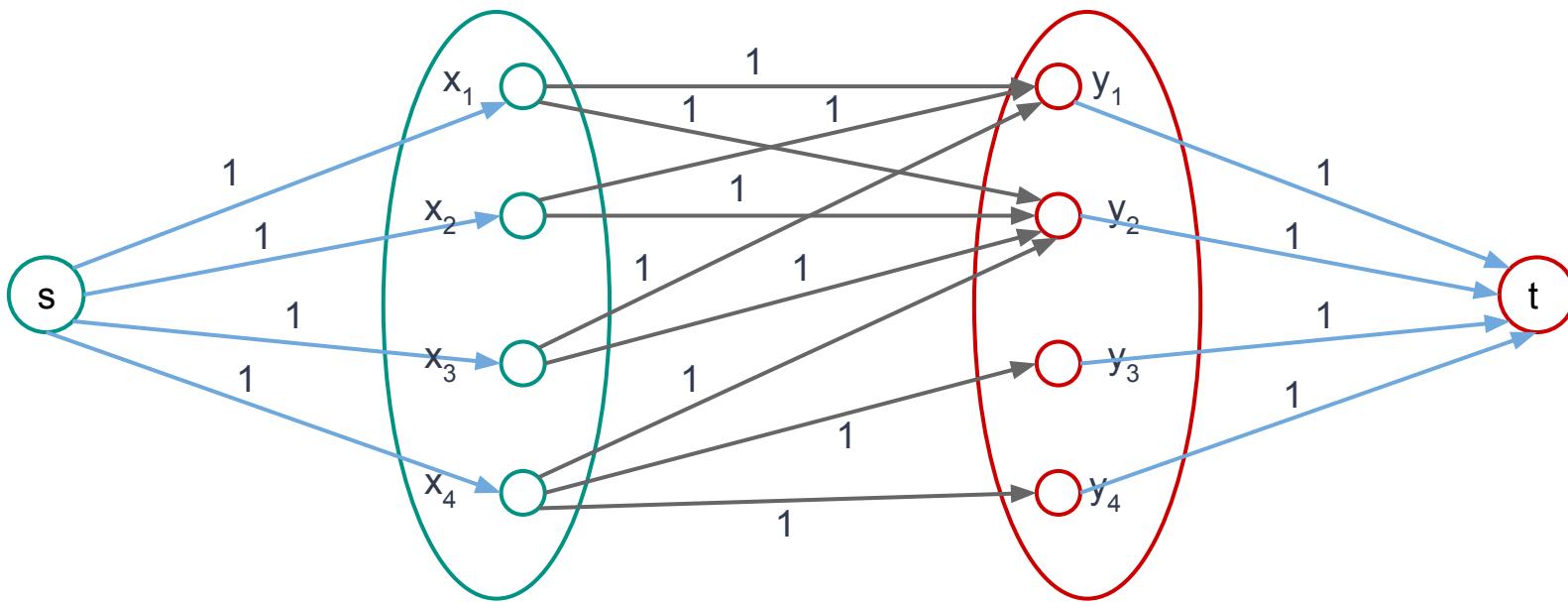
# Algoritm de determinare a unui cuplaj maxim

Transformăm muchiile  $x_iy_j$  în arce (de la X la Y)



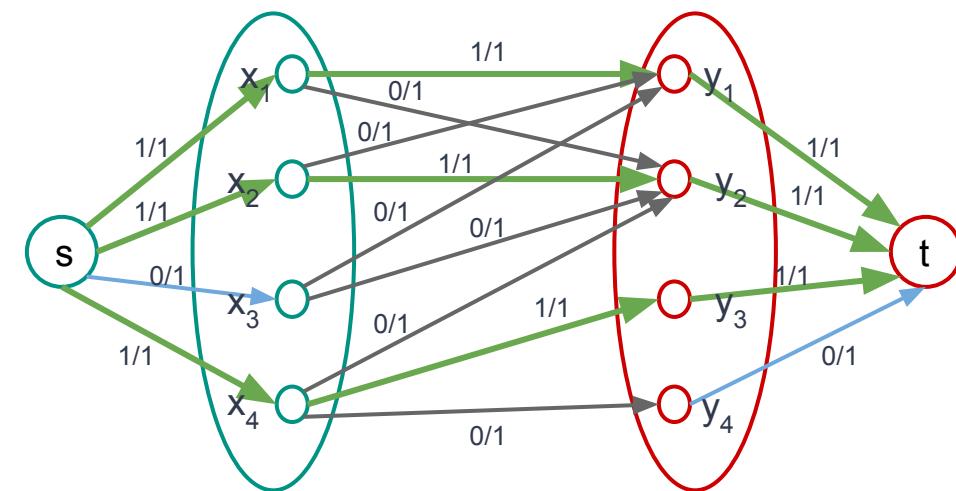
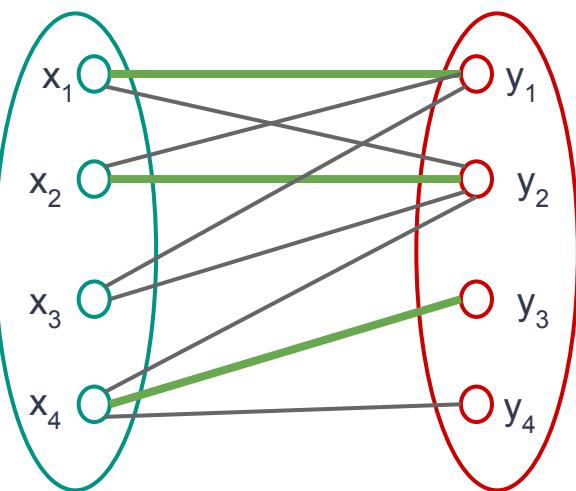
# Algoritm de determinare a unui cuplaj maxim

Asociem fiecărui arc capacitatea 1



# Algoritm de determinare a unui cuplaj maxim

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



# Algoritm de determinare a unui cuplaj maxim

## Proprietatea 1.

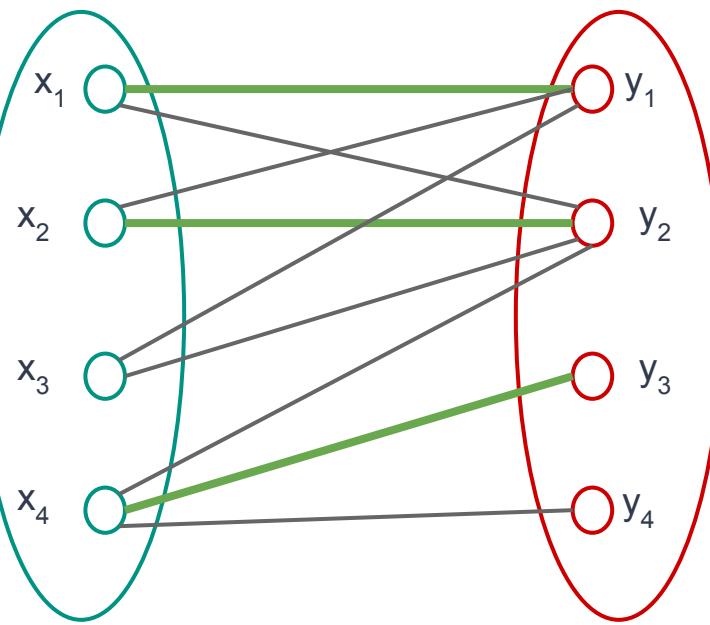
Fie  $G = (X \cup Y, E)$  un graf bipartit și  $M$  un cuplaj în  $G$ .

Atunci există un flux  $f$  în rețeaua de transport asociată  $N_G$  cu  $\text{val}(f) = |M|$ .

**Justificare:** Dat un cuplaj  $M$  în  $G$ , se poate construi un flux  $f$  în  $N_G$  cu  $\text{val}(f) = |M|$  astfel:

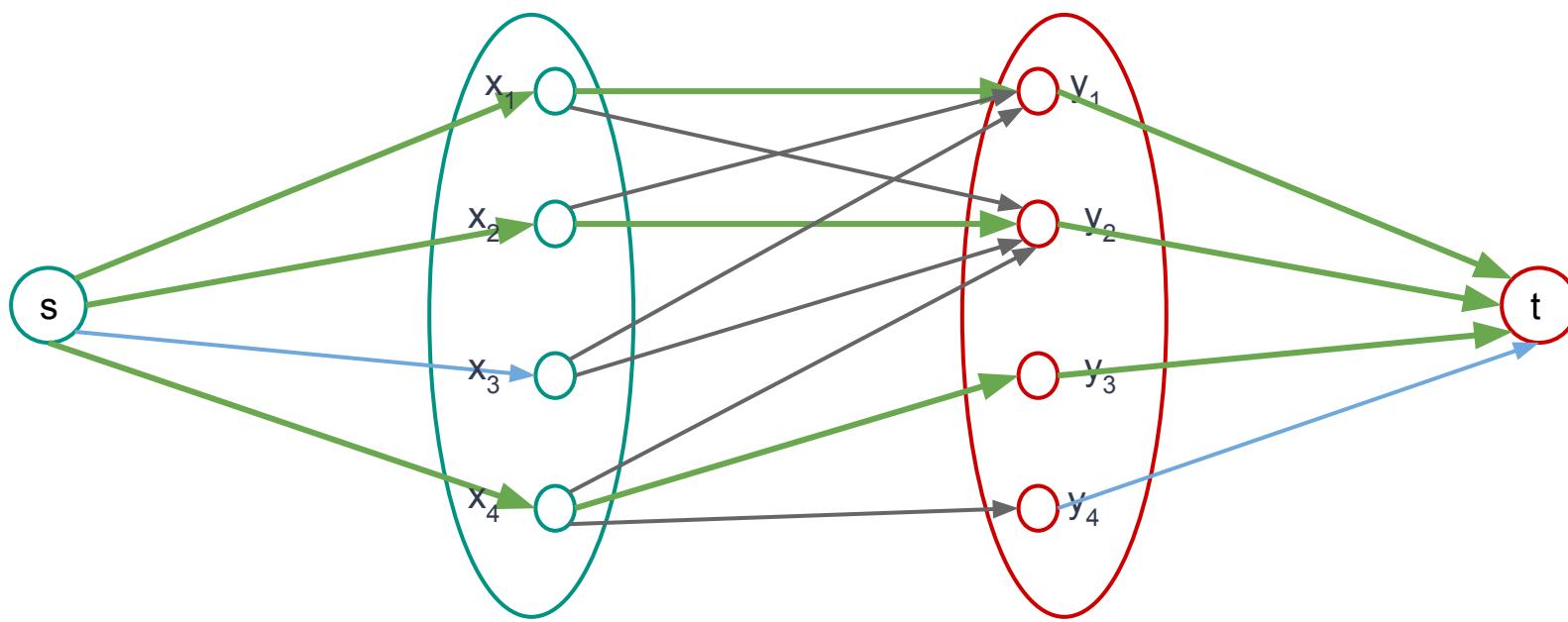
# Algoritm de determinare a unui cuplaj maxim

Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



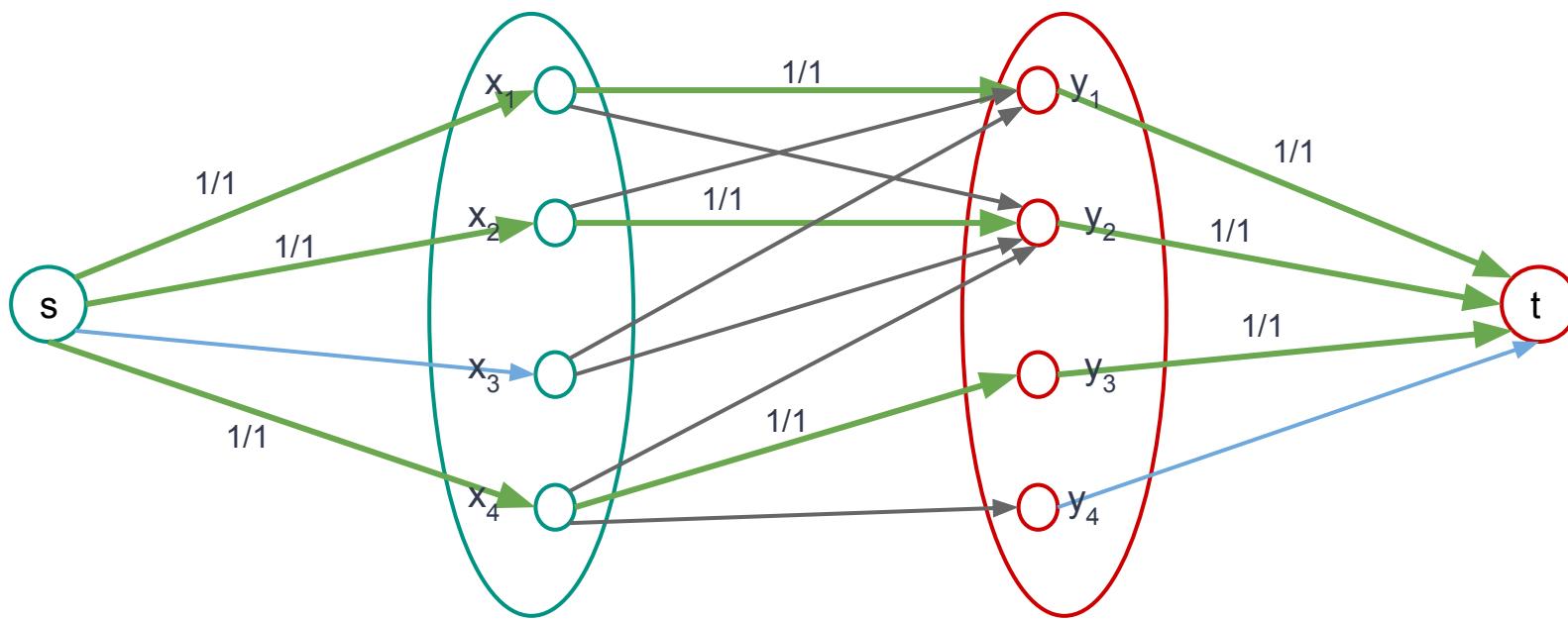
# Algoritm de determinare a unui cuplaj maxim

Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



# Algoritm de determinare a unui cuplaj maxim

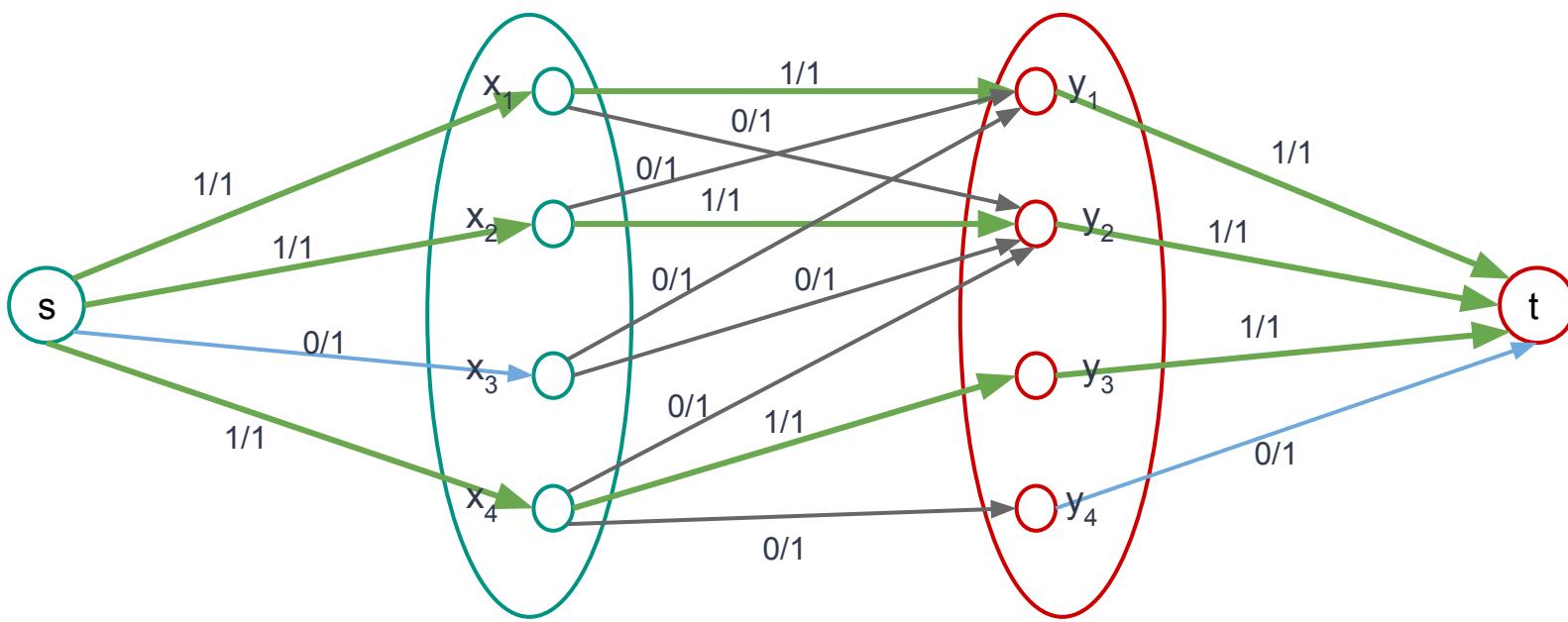
Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



Celealte arce au flux 0 și capacitate 1

# Algoritm de determinare a unui cuplaj maxim

Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



# Algoritm de determinare a unui cuplaj maxim

## Proprietatea 2.

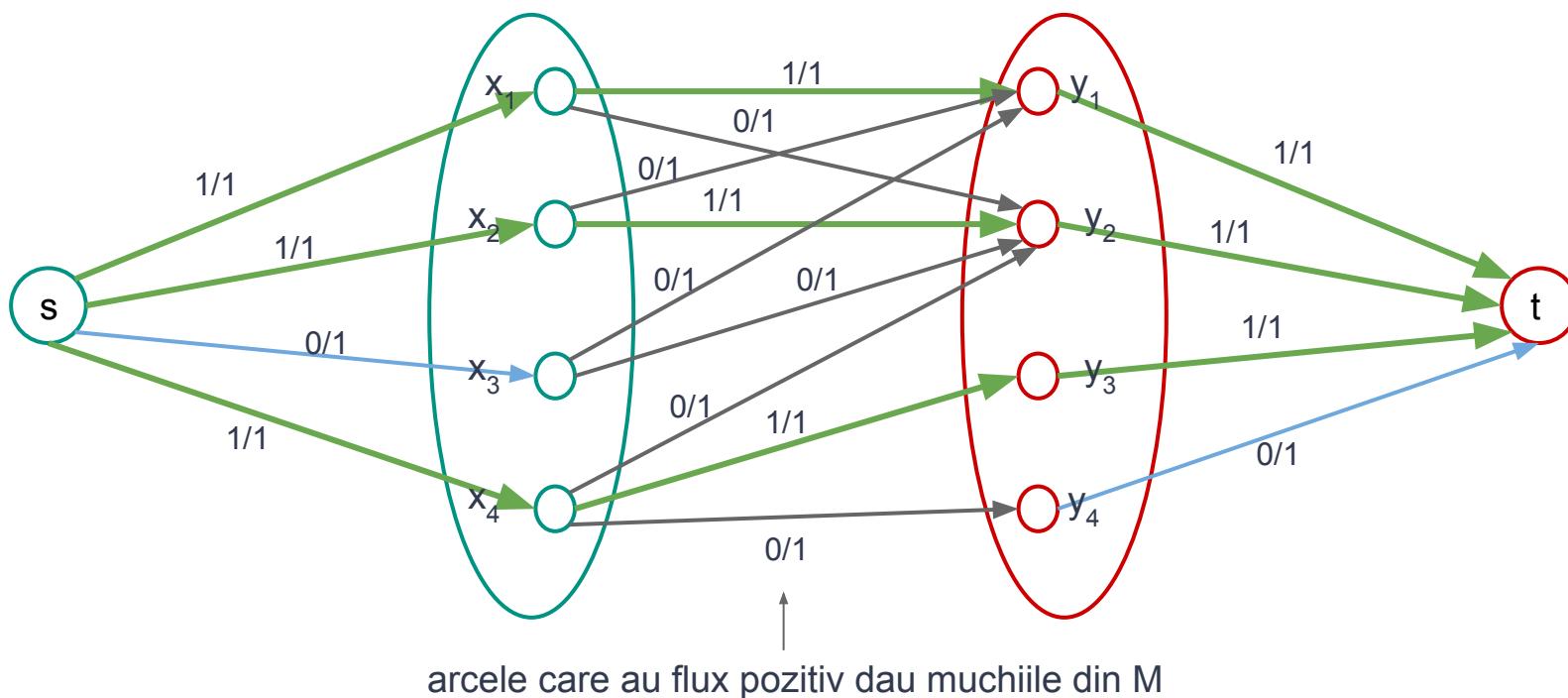
Fie  $G = (X \cup Y, E)$  un graf bipartit și  $f$  un flux în rețeaua de transport  $N_G$  asociată.

Atunci există  $M$  un cuplaj în  $G$  cu  $\text{val}(f) = |M|$ .

**Justificare:** Dat un flux  $f$  în  $N$ , se poate construi un cuplaj  $M$  în  $G$  cu  $\text{val}(f) = |M|$  astfel:

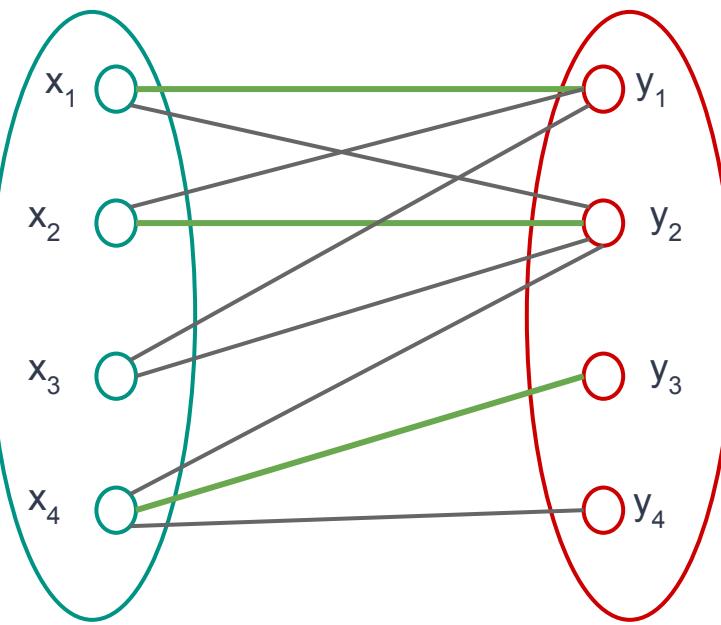
# Algoritm de determinare a unui cuplaj maxim

Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



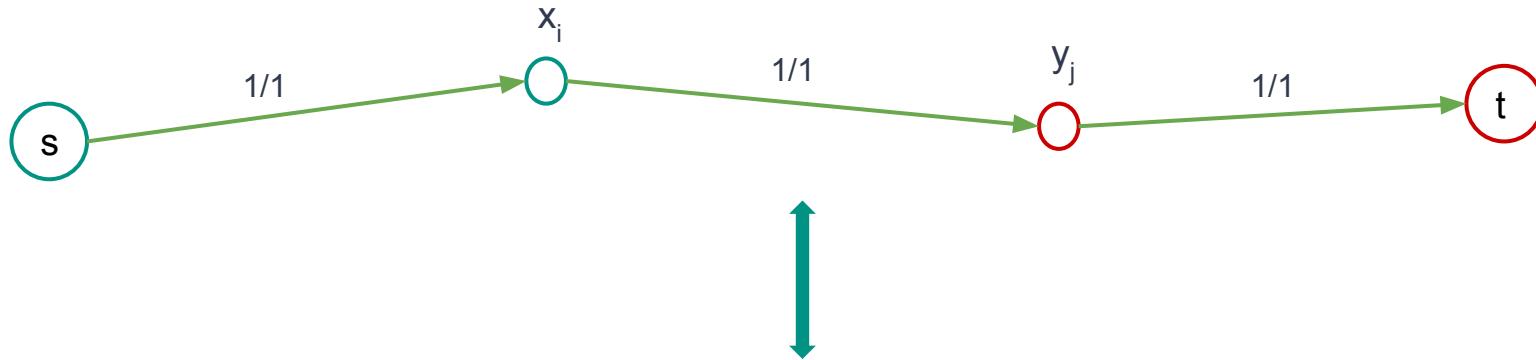
# Algoritm de determinare a unui cuplaj maxim

Dat cuplaj în graf  $\Rightarrow$  definim un flux în rețea



# Algoritm de determinare a unui cuplaj maxim

Concluzie: Flux în rețea  $\Leftrightarrow$  cuplaj în graf



Muchie în cuplaj



# Algoritm de determinare a unui cuplaj maxim

## Consecință

$f^*$  flux maxim în  $N \Rightarrow$  cuplajul corespunzător  $M^*$  este cuplaj maxim în  $G$

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

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  $xy$  cu flux nenul din  $N$ , care nu este incident în  $s$  sau în  $t$ , muchia  $xy$  corespunzătoare din  $G$  se adaugă la  $M$ )
4. return  $M$

# Algoritm de determinare a unui cuplaj maxim

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  $xy$  cu flux nenul din  $N$ , care nu este incident în  $s$  sau în  $t$ , muchia  $xy$  corespunzătoare din  $G$  se adaugă la  $M$ )
4. return  $M$

**Complexitate?**

# Algoritm de determinare a unui cuplaj maxim

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

4. return  $M$

Complexitate?  $C = 1$  (sau  $L \leq c^+(s) \leq n$ )  $\Rightarrow O(mn)$

# Aplicație

## Construcția unui graf orientat din secvențele de grade

# Aplicație

Se dau secvențele

- $s_0^+ = \{d_1^+, \dots, d_n^+\}$
- $s_0^- = \{d_1^-, \dots, d_n^-\}$

cu  $d_1^+ + \dots + d_n^+ = d_1^- + \dots + d_n^-$

Să se construiască, **dacă se poate**, un grad orientat  $G$ , cu  $s^+(G) = s_0^+$  și  $s^-(G) = s_0^-$

**Exemplu:**

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

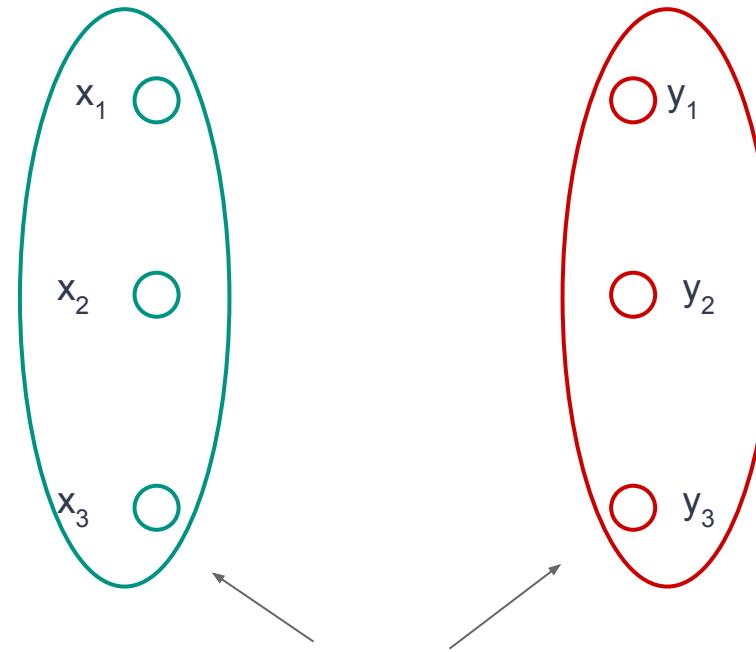
# Aplicație

**Exemplu:**

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

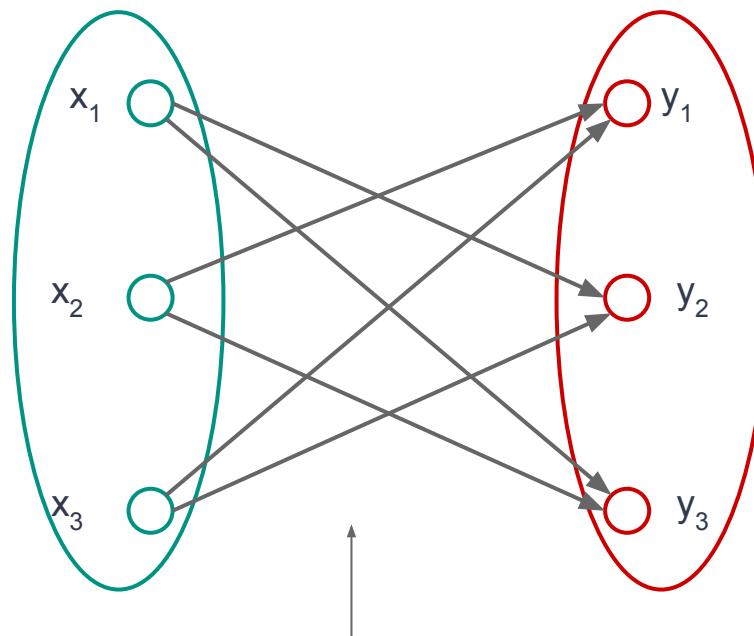
**Construim o rețea asociată celor două secvențe** a.î. din fluxul maxim în rețea să putem deduce dacă  $G$  se poate construi + arcele grafului  $G$  (în caz afirmativ).

# Aplicație



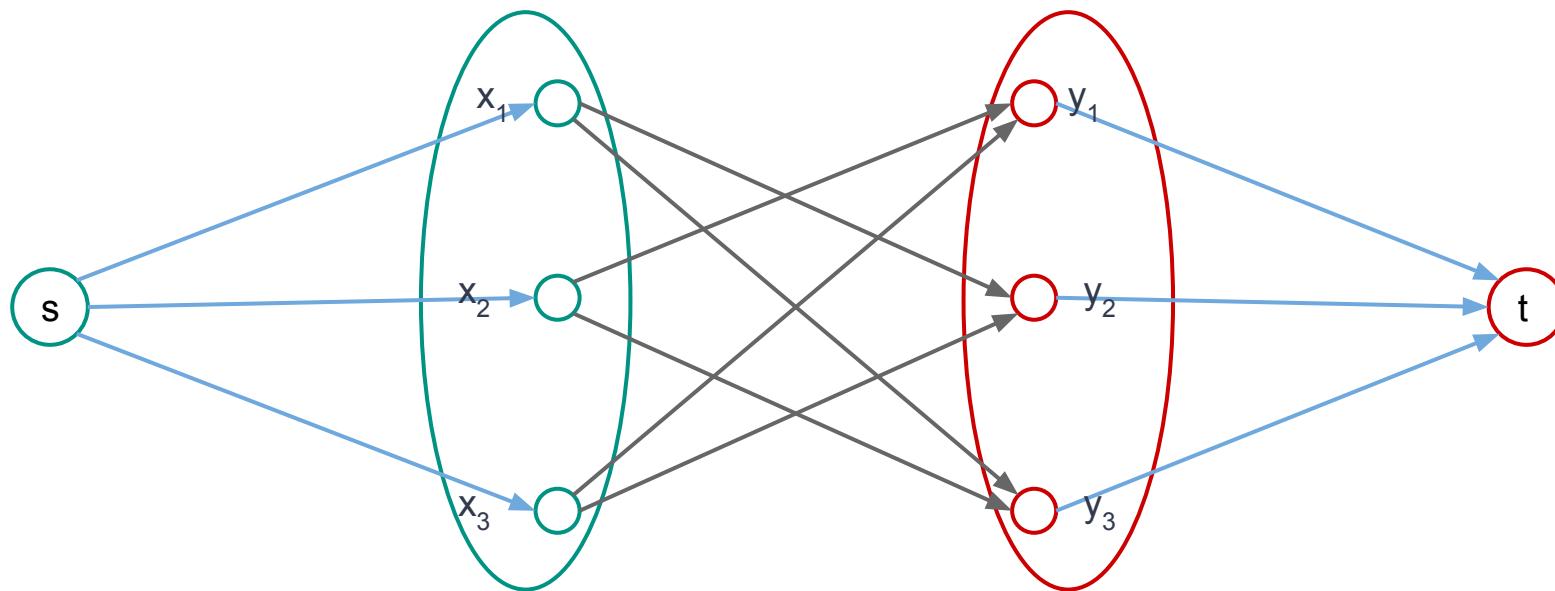
Vârfurile 1, 2, ..., n se pun în ambele clase ale bipartiției (câte o copie)

# Aplicație



arce  $x_i y_j$  cu  $i \neq j$   
(fluxul pe arcul  $x_i y_j$  va fi nenul  $\Leftrightarrow ij \in E(G)$ )

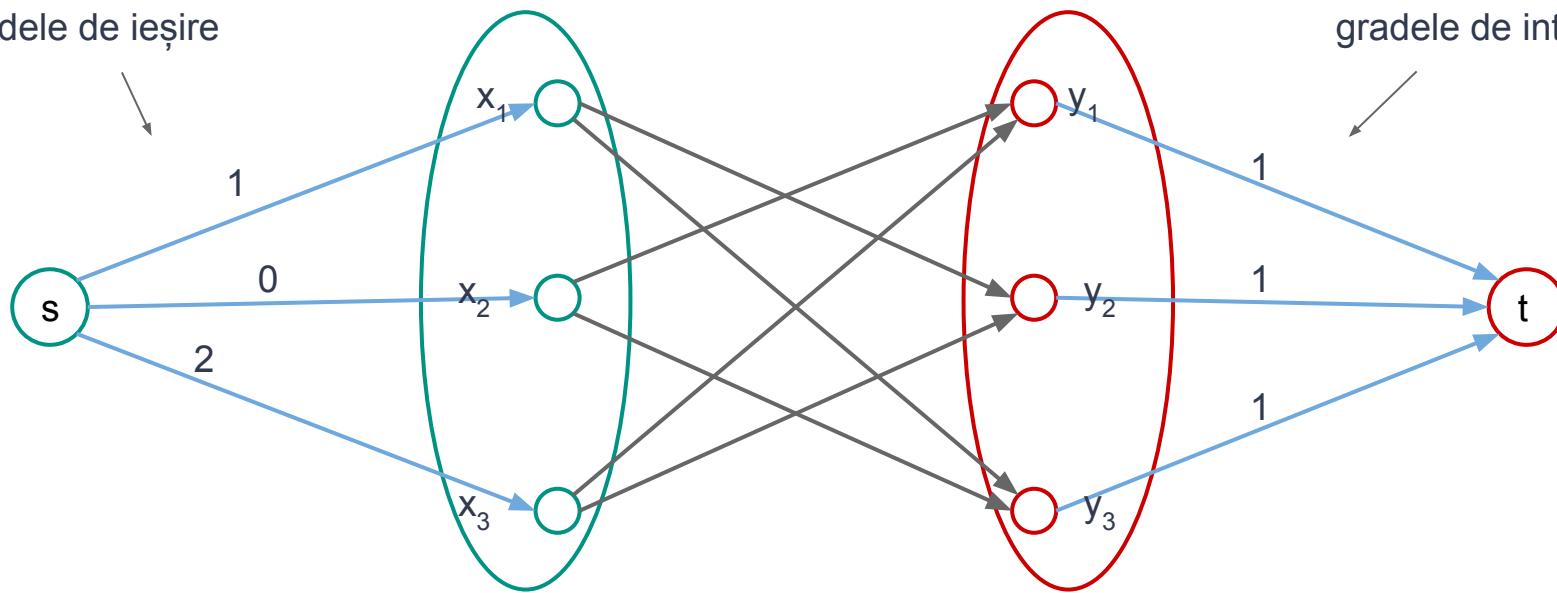
# Aplicație



# Aplicație

gradele de ieșire

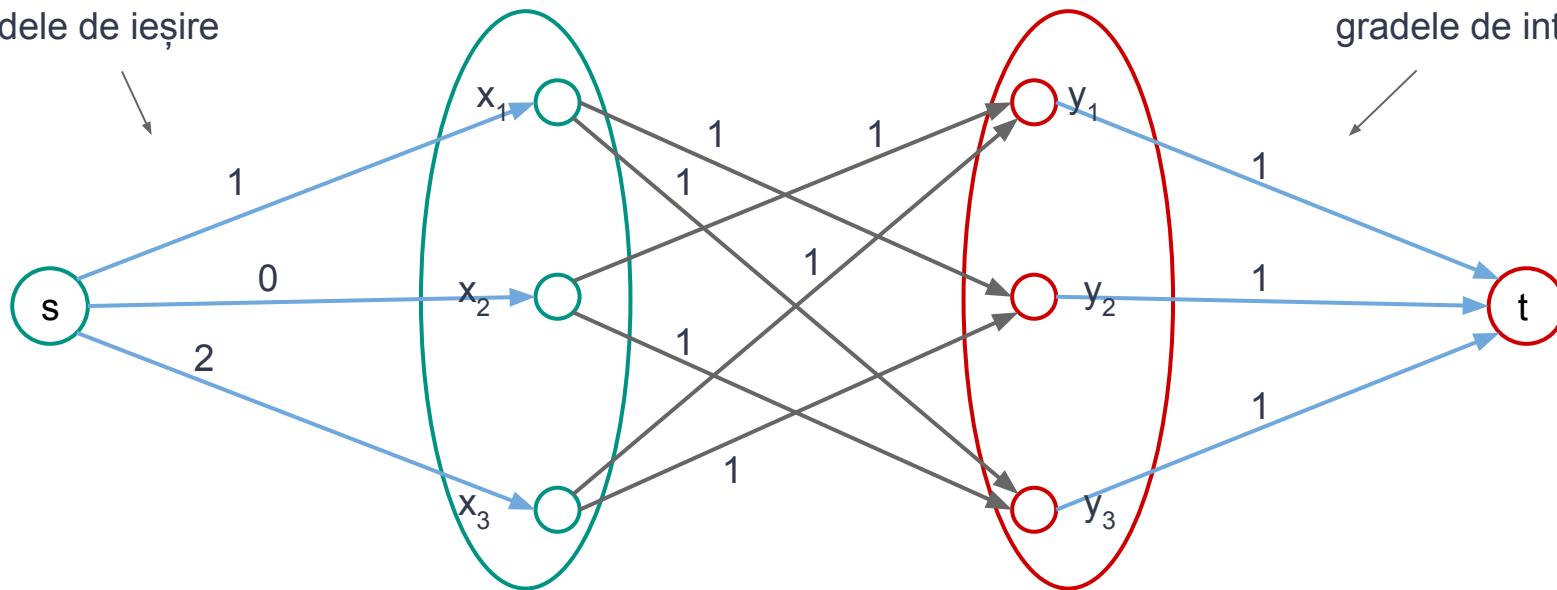
gradele de intrare



# Aplicație

gradele de ieșire

gradele de intrare



# Aplicație

## Proprietate

Există graf cu secvențele date  $\Leftrightarrow$  în rețeaua asociată, **fluxul de valoare maximă are**

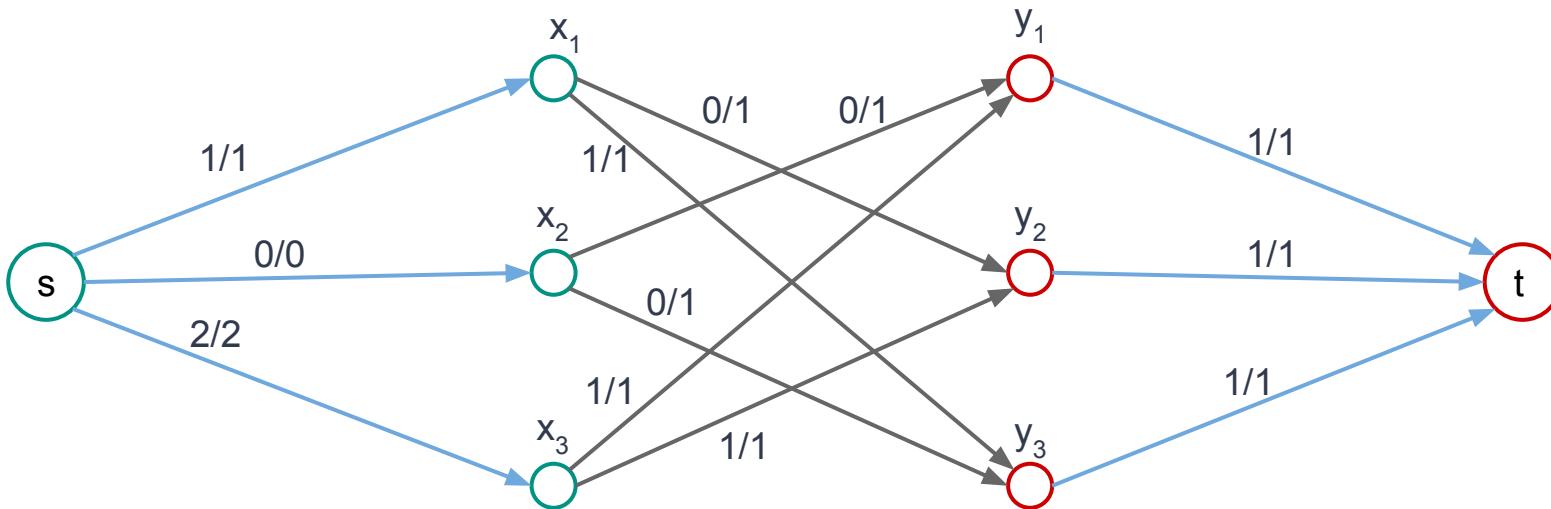
$$\text{val}(f) = d_1^+ + \dots + d_n^+ = d_1^- + \dots + d_n^-$$

**(satereză toate arcele care ies din s și toate arcele care intră în t)**

**Tăieturile ( $\{s\}$ ,  $V-\{s\}$ ) , ( $V-\{t\}$ , t) sunt minime**

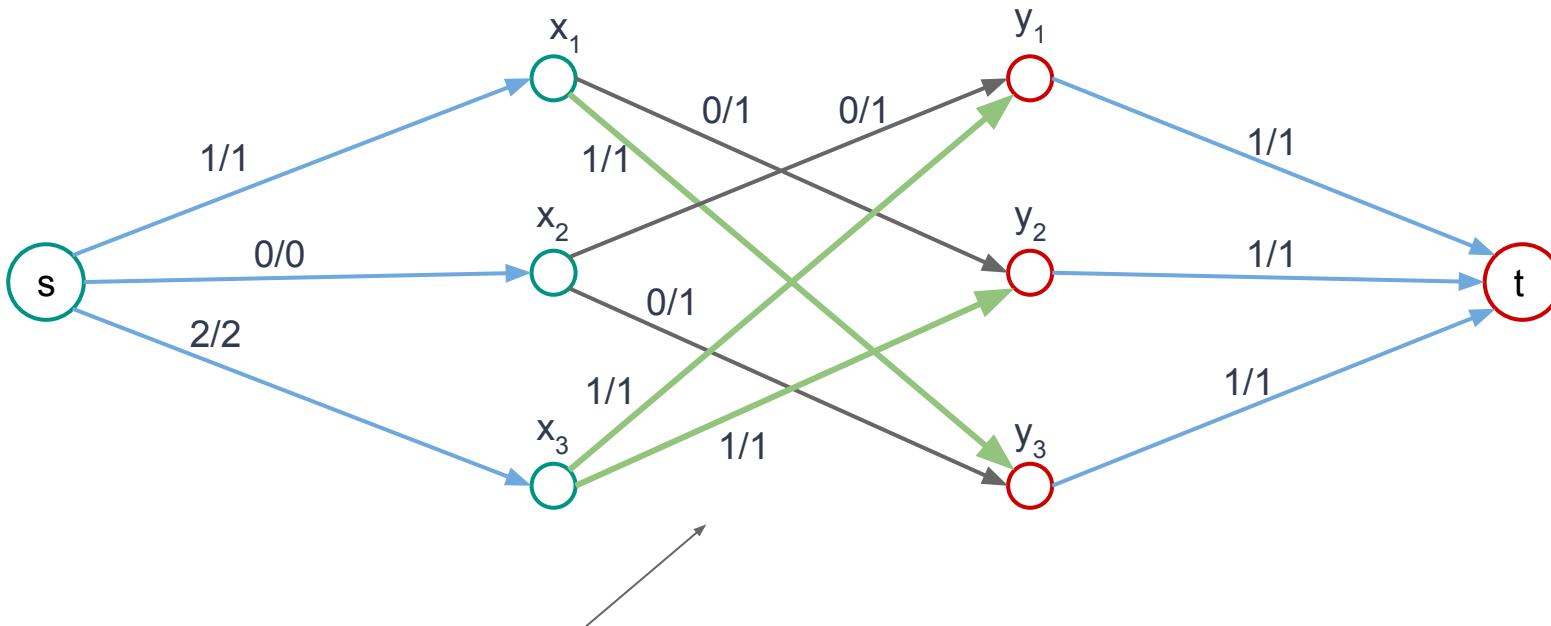
# Aplicație

Flux în rețea care saturează arcele din  $s$  și  $t \Rightarrow G$



# Aplicație

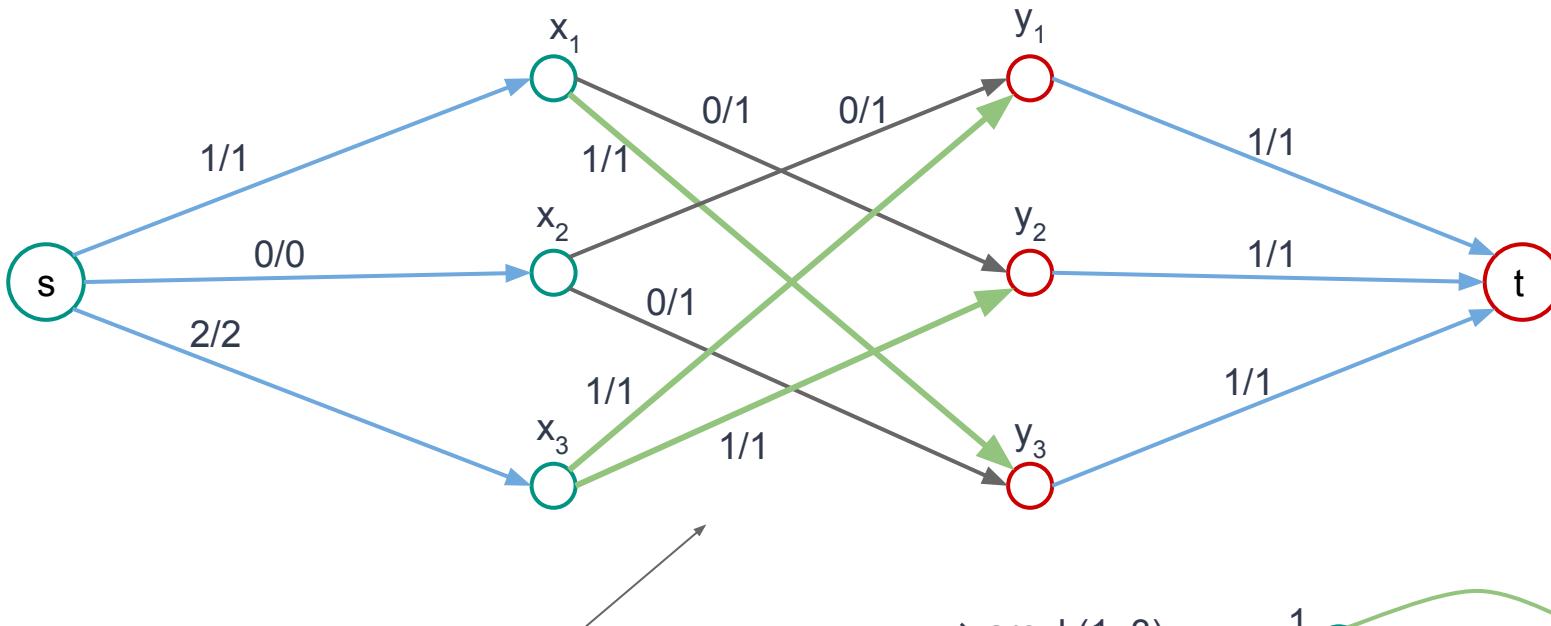
Flux în rețea care saturează arcele din  $s$  și  $t \Rightarrow G$



arcele care au flux pozitiv dău arcele lui  $G$

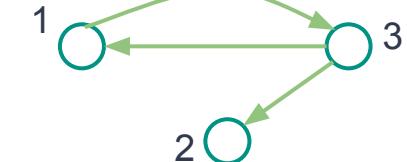
# Aplicație

Flux în rețea care saturează arcele din  $s$  și  $t \Rightarrow G$



arcele care au flux pozitiv dău arcele lui  $G$

- $x_1y_3 \Rightarrow$  arcul  $(1, 3)$
- $x_3y_1 \Rightarrow$  arcul  $(3, 1)$
- $x_3y_2 \Rightarrow$  arcul  $(3, 2)$



# Aplicație

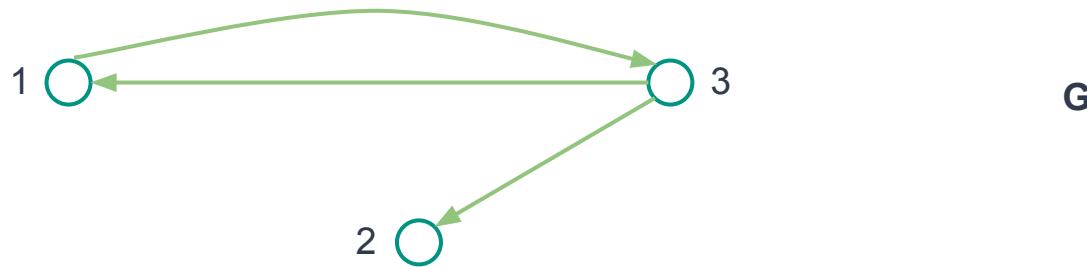
## Reciproc

**Flux în rețea care saturează arcele din  $s$  și  $t \Leftarrow G$**

# Aplicație

## Reciproc

Flux în rețea care saturează arcele din  $s$  și  $t \Leftarrow G$

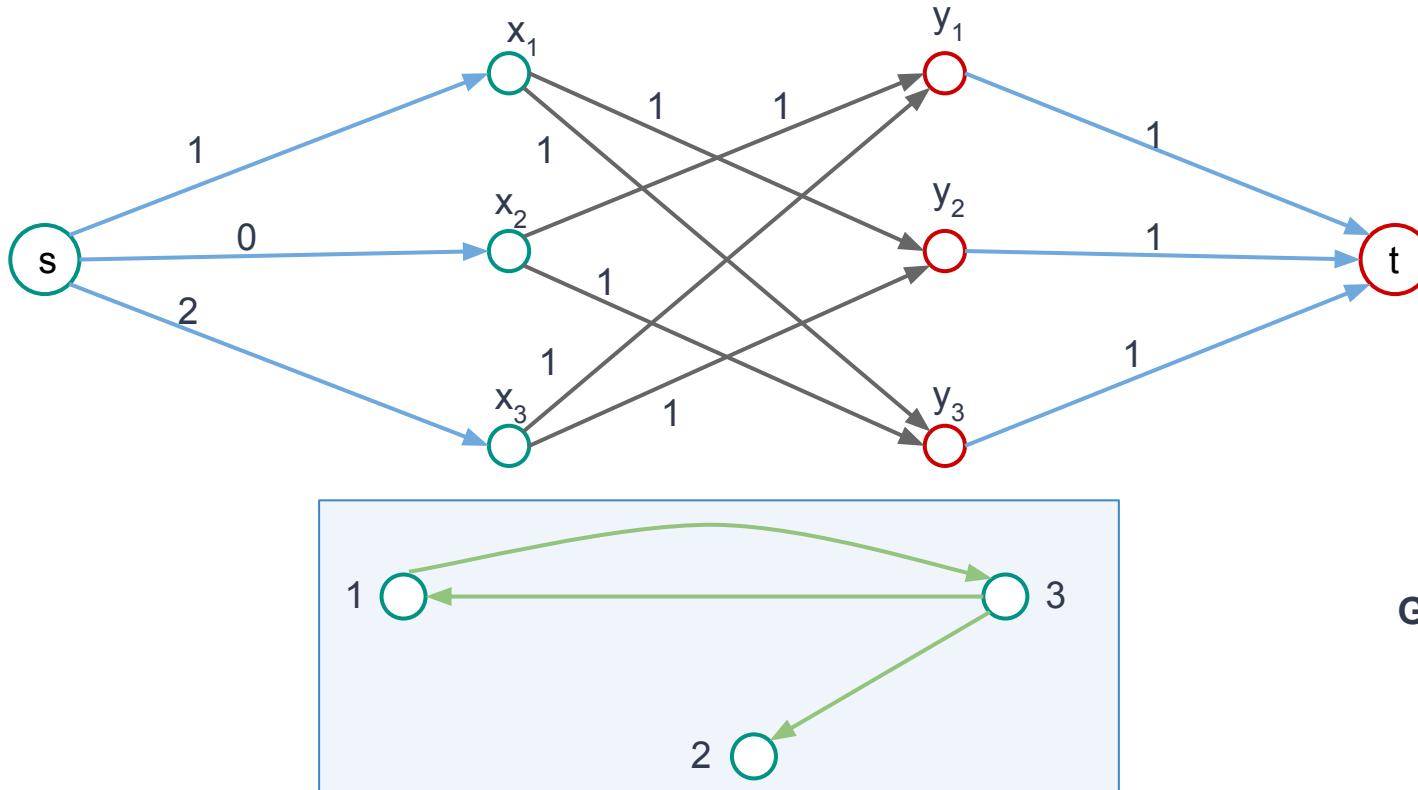


$$s_0^+ = \{1, 0, 2\}$$

$$s_0^- = \{1, 1, 1\}$$

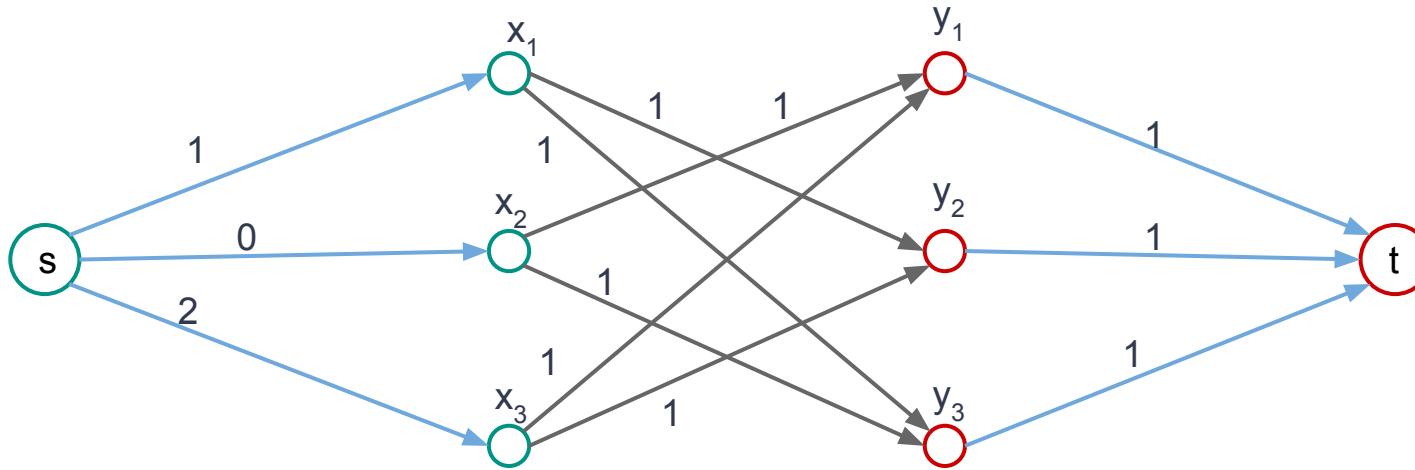
# Aplicație

Flux în rețea care saturează arcele din  $s \neq t \in G$



# Aplicație

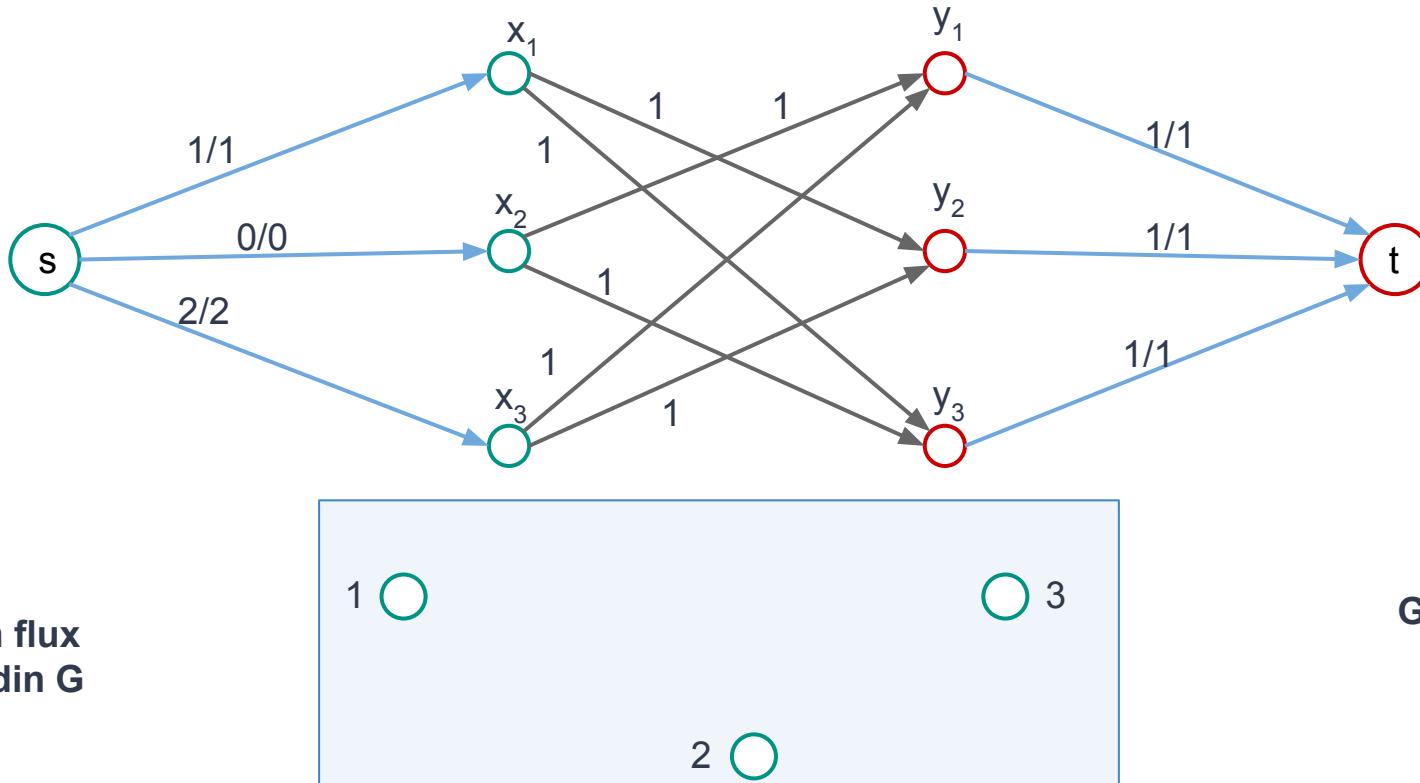
Flux în rețea care saturează arcele din  $s$  și  $t \in G$



saturăm arcele  
din  $s$  și  $t$

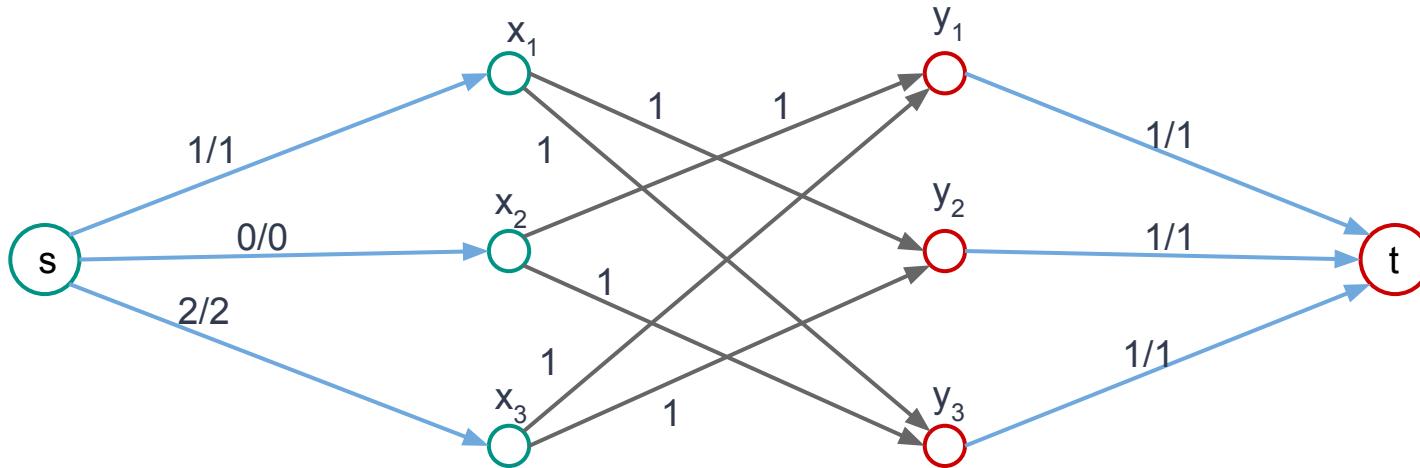
# Aplicație

Flux în rețea care saturează arcele din  $s \text{ și } t \in G$

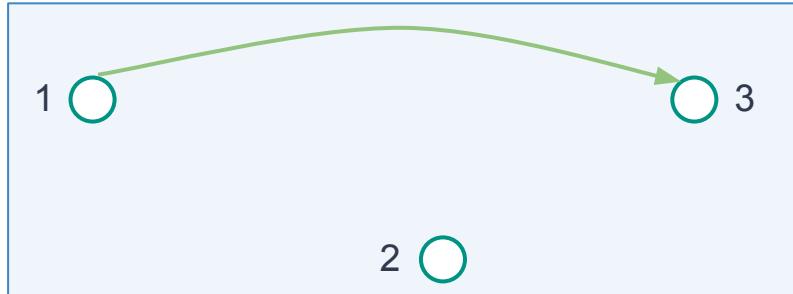


# Aplicație

Flux în rețea care saturează arcele din  $s \neq t \in G$



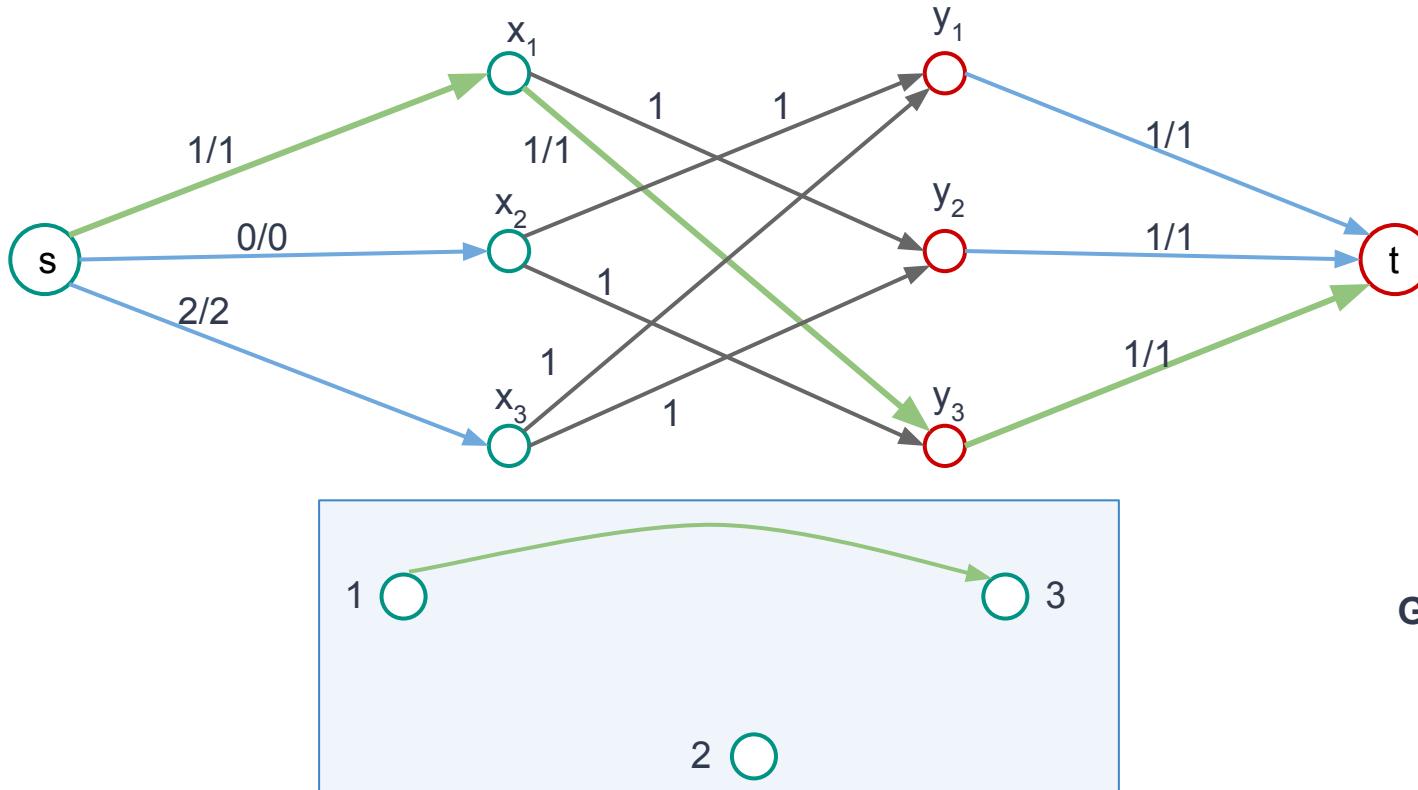
Arc  $(1, 3) \Rightarrow$  flux  
pe arcul  $x_1y_3$



$G$

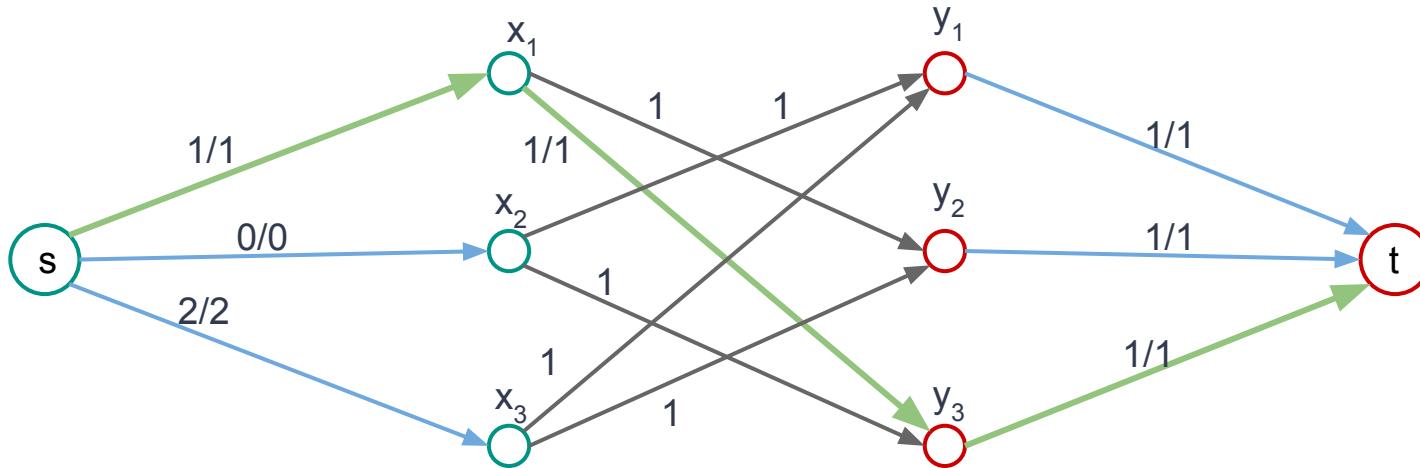
# Aplicație

Flux în rețea care saturează arcele din  $s \neq t \in G$

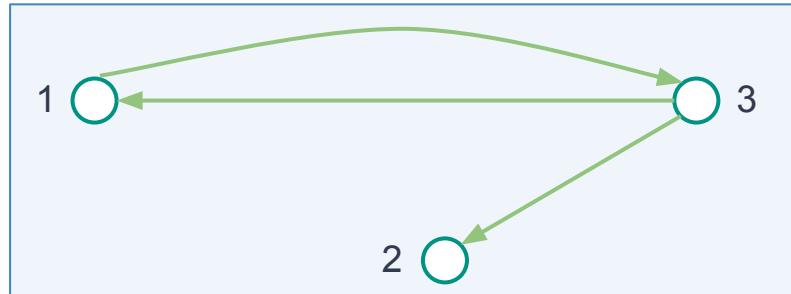


# Aplicație

Flux în rețea care saturează arcele din  $s \neq t \in G$



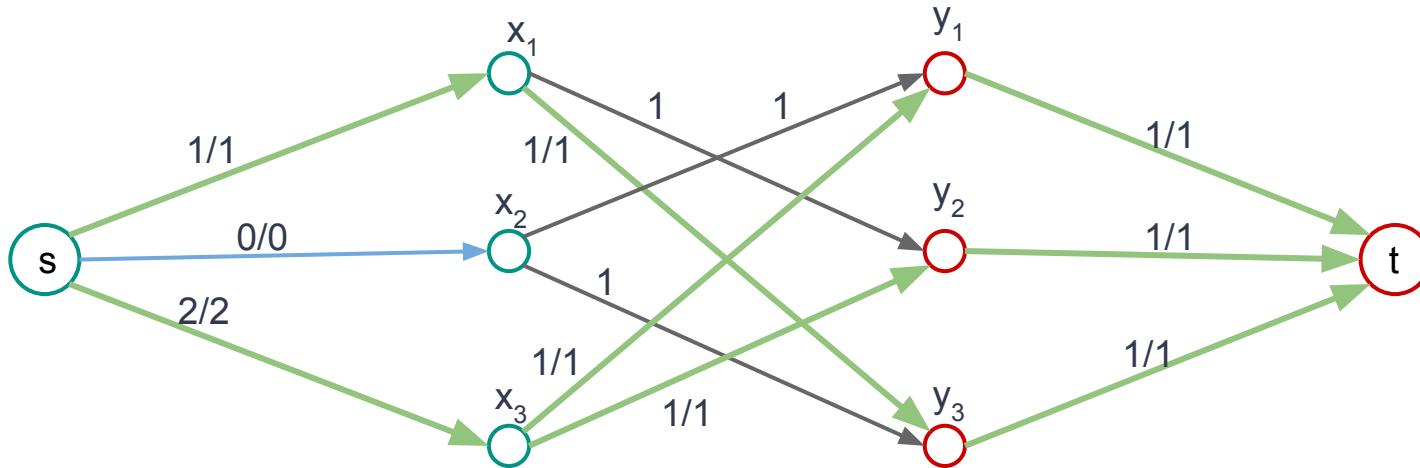
Arce  $(3, 1), (3, 2)$   
⇒ ?



$G$

# Aplicație

Flux în rețea care saturează arcele din  $s \neq t \in G$



Restul arcelor au fluxul 0

# Algoritm

**Algoritm de determinare a unui graf orientat  $G$  cu  $s^+(G) = s_0^+$  și  $s^-(G) = s_0^-$**

- 1.** Construim  $N$  rețeaua de transport asociată
- 2.** Determinăm  $f^*$  flux maxim în  $N$

# Algoritm

**Algoritm de determinare a unui graf orientat  $G$  cu  $s^+(G) = s_0^+$  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

# Algoritm

**Algoritm de determinare a unui graf orientat  $G$  cu  $s^+(G) = s_0^+$  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.**  $V(G) = \{1, \dots, n\}$

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

**Complexitate:**  $L \leq c^+(s) = d_1^+ + \dots + d_n^+ = m \Rightarrow O(m^2)$

# Aplicații

## Alte probleme de asociere

# Probleme de asociere (temă)

Se dau 2 multimi de obiecte, spre exemplu produse (aflate în fabrici) și clienți (joburi / mașini, pagini web / servere, echipe turneu etc).

- Pentru fiecare produs  $x$  se cunoaște
  - $c(x)$  = numărul de unități disponibile din produsul  $x$
- Pentru fiecare client  $y$  se cunoaște
  - $c(y)$  = numărul maxim de unități de produse pe care le poate primi (în total, din toate produsele)
- Pentru fiecare pereche produs-client ( $x, y$ ) se cunoaște
  - $c(x, y)$  = numărul maxim de unități din produsul  $x$  pe care le poate primi clientul  $y$

**Să se determine o modalitate de a distribui cât mai multe produse (unități de produse) clienților cu respectarea constrângerilor.**

# Probleme de asociere (temă)

**Observație:** Problema determinării unui cuplaj de cardinal maxim într-un graf bipartit  $G=(X \cup Y, E)$  este un caz particular al acestei probleme, pentru:

- $c(x) = c(y) = 1, \quad \forall x \in X, y \in Y$
- $c(x, y) = 1, \quad \text{dacă } xy \in E$
- $c(x, y) = 0, \quad \text{dacă } xy \notin E$

# Aplicație

Drumuri arc-disjuncte între  
două vârfuri.

Conecțivitatea unui graf

SUPLIMENTAR

# Drumuri arc-disjuncte

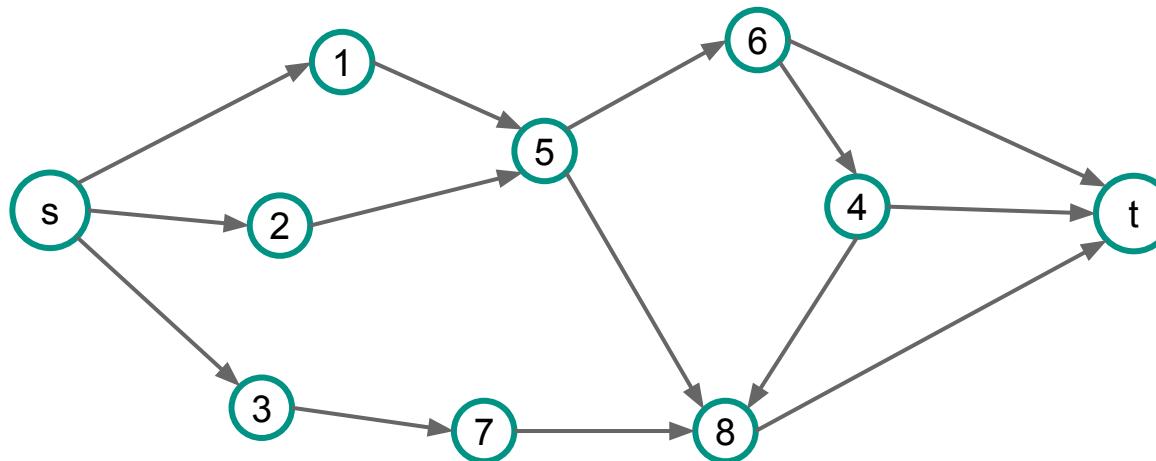
Se dau:

- $G = (V, E)$  - orientat, conex (graful neorientat suport)
- $s, t$  - două vârfuri

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

Două drumuri  $P_1, P_2$  se numesc **arc-disjuncte** dacă  $E(P_1) \cap E(P_2) = \emptyset$

# Drumuri arc-disjuncte



# Drumuri arc-disjuncte

Se dau:

- $G = (V, E)$  - orientat, conex (graful neorientat suport)
- $s, t$  - două vârfuri

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

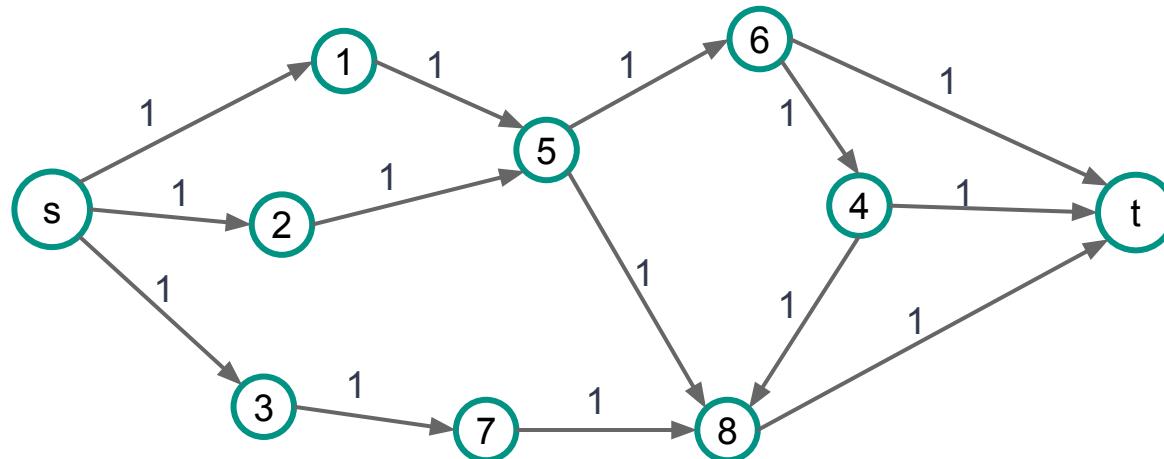
**Aplicații:**

- Fiabilitatea rețelelor, conectivitate
- Probleme de strategie
- Măsuri de centralitate (a unui nod) în rețele sociale

# Drumuri arc-disjuncte

Intuitiv:

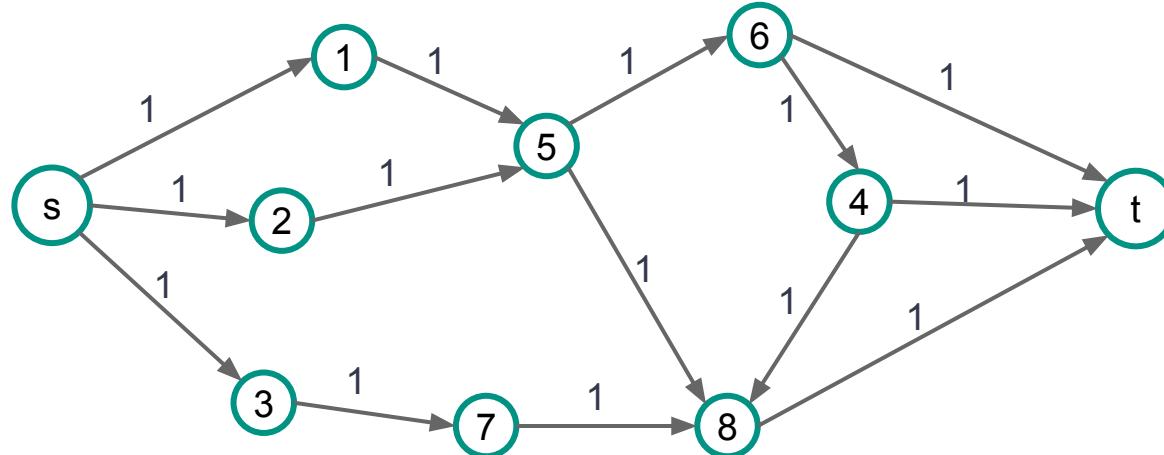
- asociem fiecărui arc capacitatea 1
- fluxul maxim:  $f(e) \in \{0, 1\}$



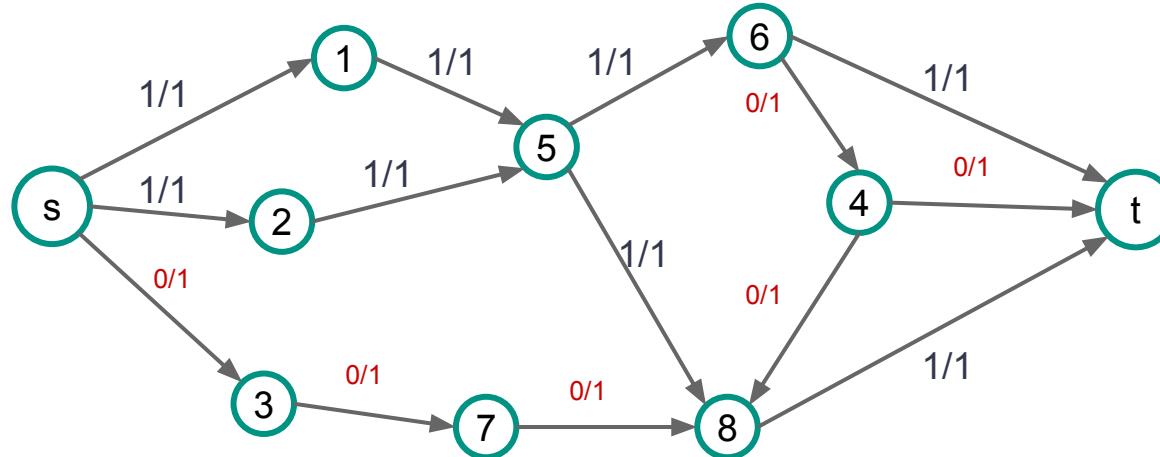
# Drumuri arc-disjuncte

Intuitiv:

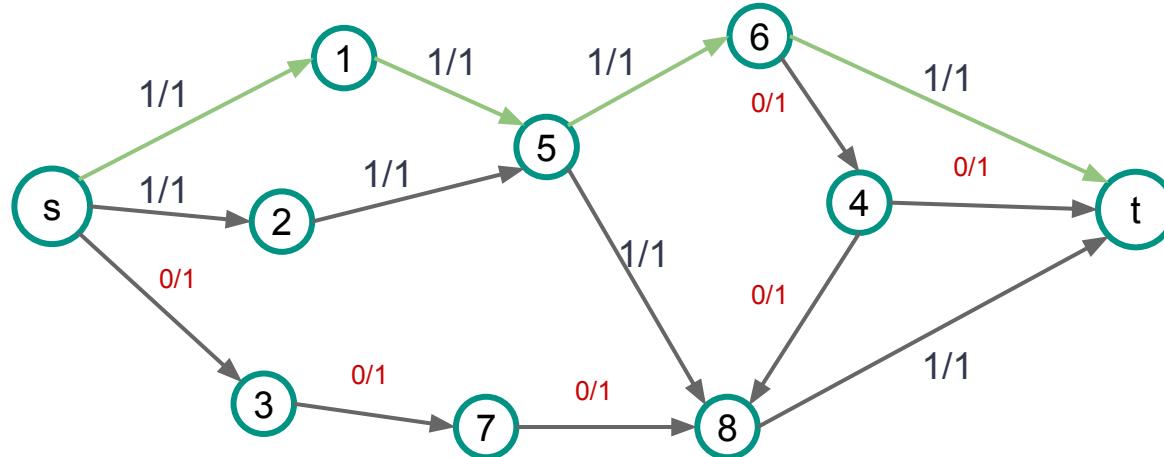
- asociem fiecărui arc capacitatea 1
- fluxul maxim:  $f(e) \in \{0, 1\}$
- un drum de la s la t = **traseul parcurs de o unitate de flux de la s la t**
- numărul de s-t drumuri arc-disjuncte** = valoarea fluxului maxim



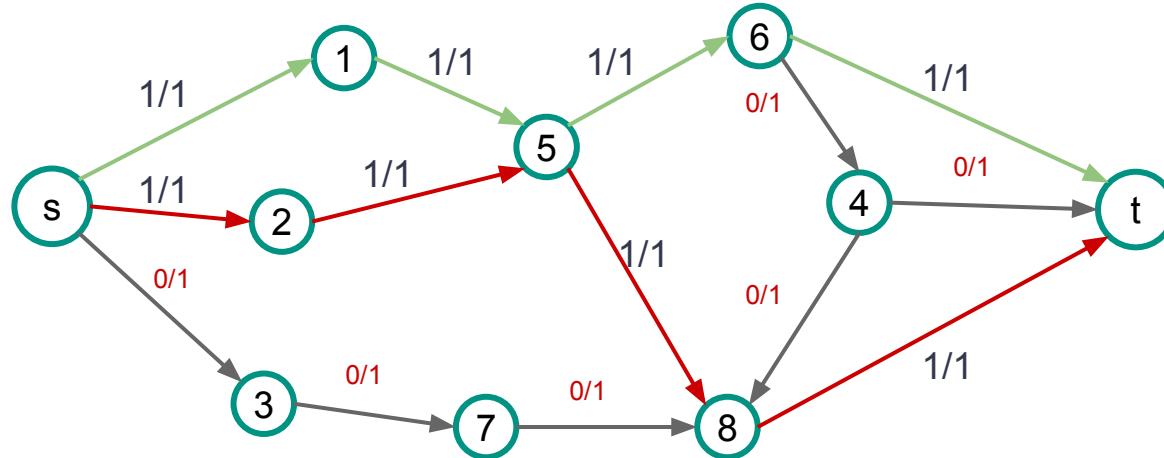
# Drumuri arc-disjuncte



# Drumuri arc-disjuncte



# Drumuri arc-disjuncte



# Drumuri arc-disjuncte

## Teorema lui Menger

Fie  $G$  graf orientat,  $s, t$  două vârfuri distincte în  $G$ .

Numărul minim de arce care trebuie eliminate pentru ca  $s$  și  $t$  să nu mai fie conectate printr-un drum (să fie **separate**) =

numărul maxim de drumuri arc-disjuncte de la  $s$  la  $t$

# Drumuri arc-disjuncte

## Teorema lui Menger

Fie  $G$  graf orientat,  $s, t$  două vârfuri distincte în  $G$ .

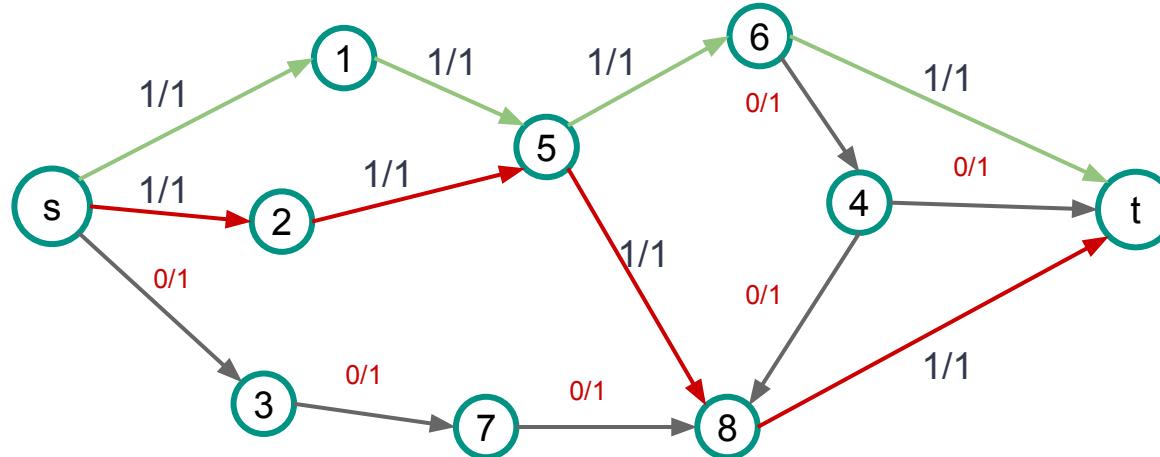
Numărul minim de arce care trebuie eliminate pentru ca  $s$  și  $t$  să nu mai fie conectate printr-un drum (să fie **separate**) =

numărul maxim de drumuri arc-disjuncte de la  $s$  la  $t$



O astfel de mulțime de arce se poate determina cu algoritmul Ford-Fulkerson?

# Drumuri arc-disjuncte



# Drumuri arc-disjuncte

## Teorema lui Menger

Fie  $G$  graf orientat,  $s, t$  două vârfuri distincte în  $G$ .

Numărul minim de arce care trebuie eliminate pentru ca  $s$  și  $t$  să nu mai fie conectate printr-un drum (să fie **separate**) =

numărul maxim de drumuri arc-disjuncte de la  $s$  la  $t$

O astfel de mulțime de arce se poate determina cu algoritmul  
**Ford-Fulkerson?**

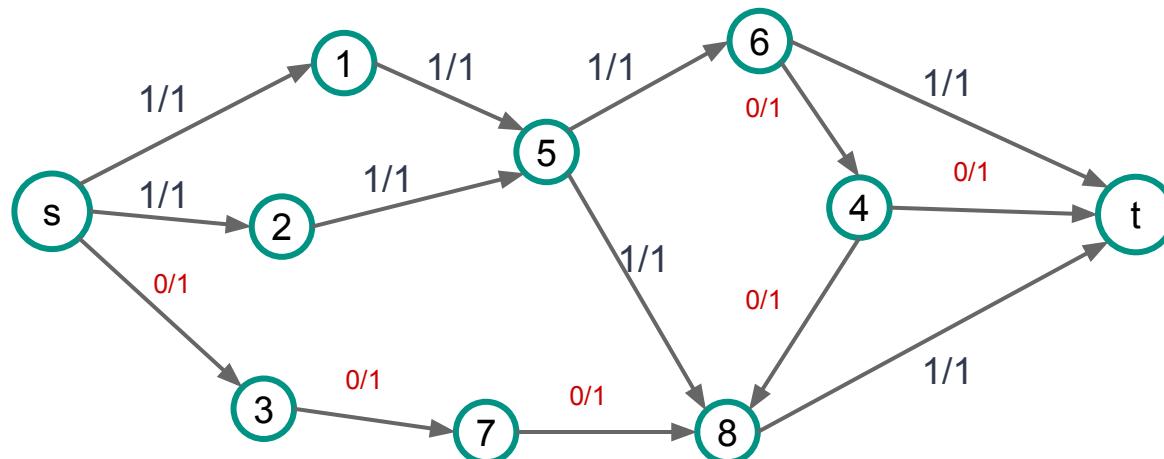


**Sunt arcele directe ale tăieturii minime**

# Drumuri arc-disjuncte



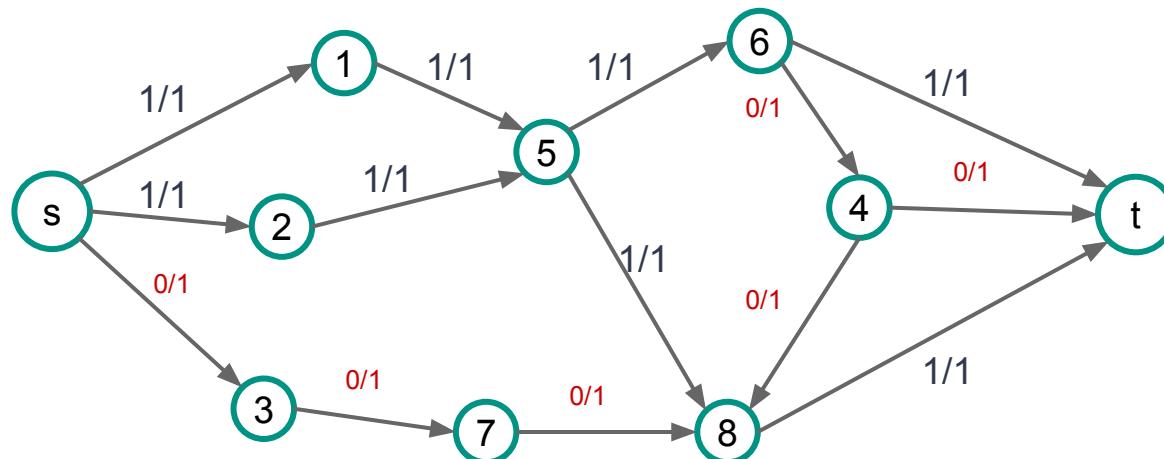
Cum determinăm tăietura minimă?



# Drumuri arc-disjuncte



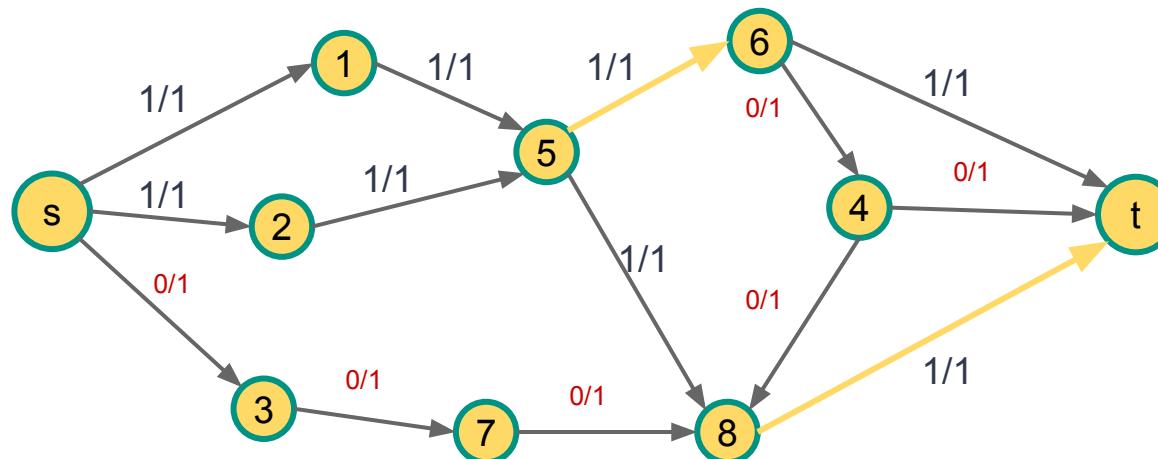
Mulțimea vârfurilor accesibile din  $s$  prin lanțuri  $f$ -nesaturate



# Drumuri arc-disjuncte



Mulțimea vârfurilor accesibile din  $s$  prin lanțuri  $f$ -nesaturate



# Drumuri arc-disjuncte

## Variante

- Aceeași problemă pentru

$G = (V, E)$  - **neorientat** conex,  $|E| > 2$

- Aceeași problemă pentru **vârfuri** (s-t drumuri care nu au vârfuri interne în comun)

# Drumuri arc-disjuncte

- **Muchie - conectivitatea lui  $G$**   $k'(G)$  = cardinalul **minim** al unei multimi 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**
  - Amintim (laborator + seminar)
    - există muchie critică  $\Rightarrow G$  este 1-conex
    - nu există muchie critică  $\Rightarrow G$  este 2-conex
- **Cu ajutorul algoritmului de flux maxim, putem determina (muchie)-conectivitatea unui graf**

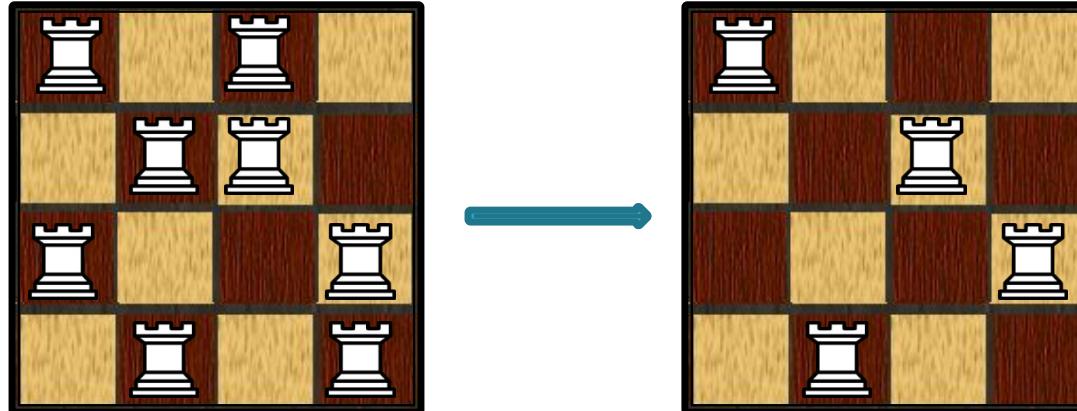


# Aplicații cuplaje

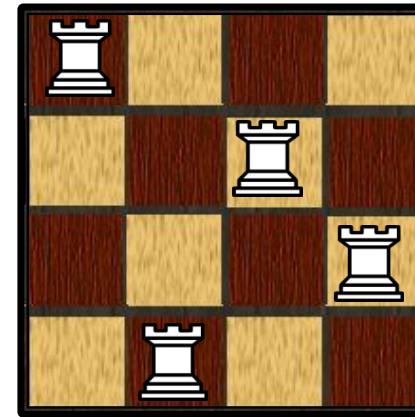
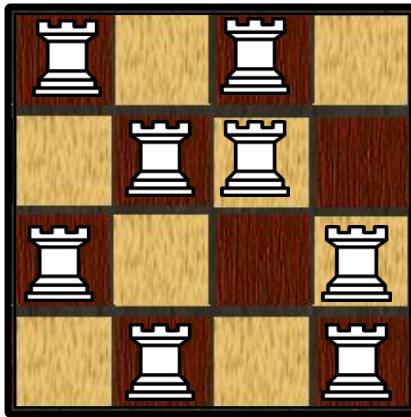
# Aplicație: Matrice de permutări

# Probleme

Pe o tablă de tip șah de dimensiuni  $n \times n$  sunt așezate ture, astfel încât pe fiecare linie și fiecare coloană se află **același număr de ture**. Să se arate că se pot păstra pe tablă  $n$  dintre aceste ture, care nu se atacă două câte două.



# Probleme



$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$



$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

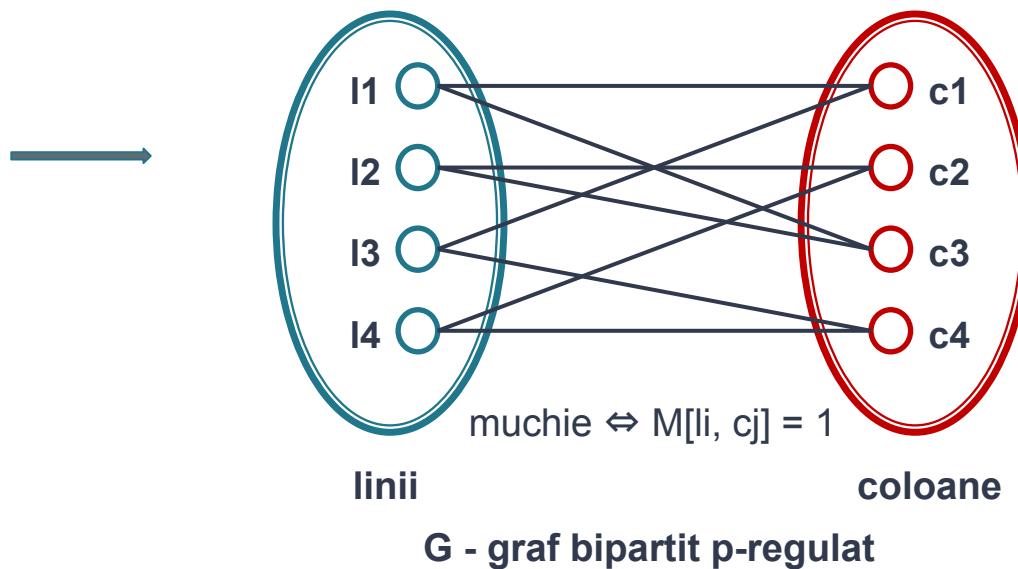
# Reformulare cu matrice

Fie  $p > 1$  și  $M$  o matrice  $n \times n$  cu elemente  $\{0, 1\}$ , astfel încât pe fiecare linie și pe fiecare coloană sunt exact  $p$  elemente 1.

Atunci  $M$  conține o matrice de permutări (având un unic 1 pe fiecare linie și coloană).

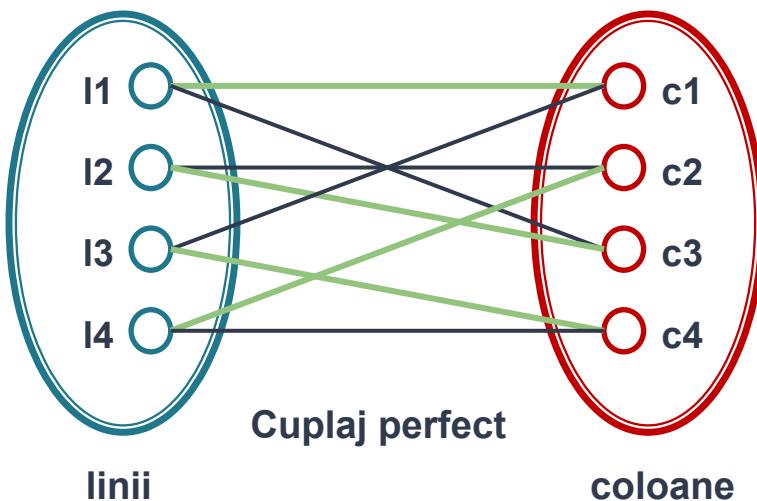
# Reformulare cu matrice

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$



- există matrice de permutări în  $M \Leftrightarrow$  există cuplaj perfect în  $G$
  - rezultă din consecințele teoremei lui HALL (teorema căsătoriei)

# Reformulare cu matrice

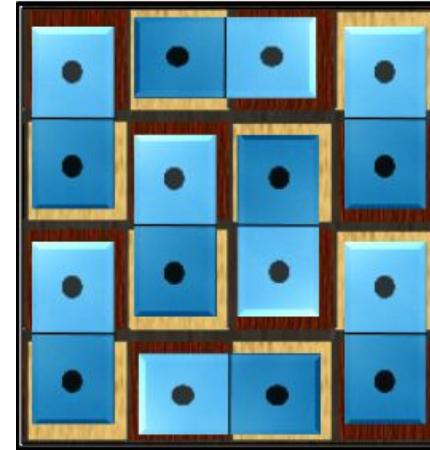
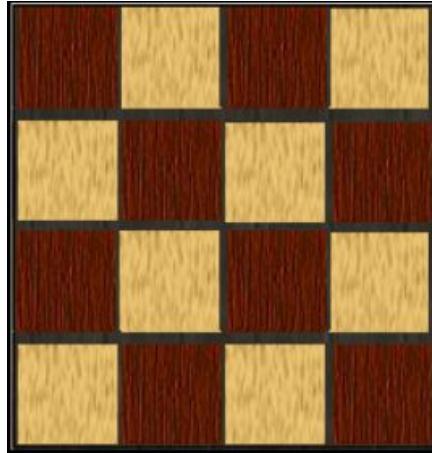


$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Aplicație:  
Acoperire tablă cu piese de  
domino

# Probleme

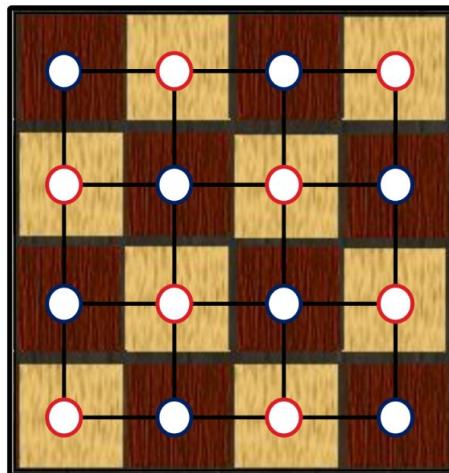
Acoperirea unei table cu piese de domino



# Probleme

Tabla

Acoperire

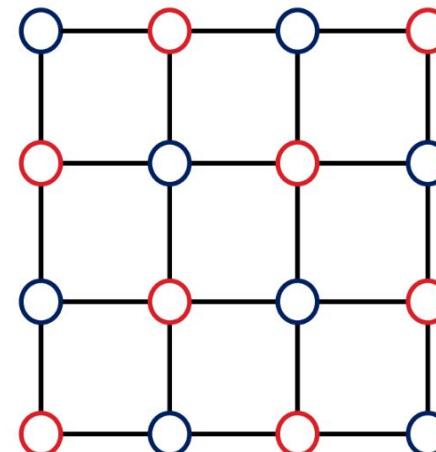


⇒

Graful grid

⇒

Cuplaj perfect

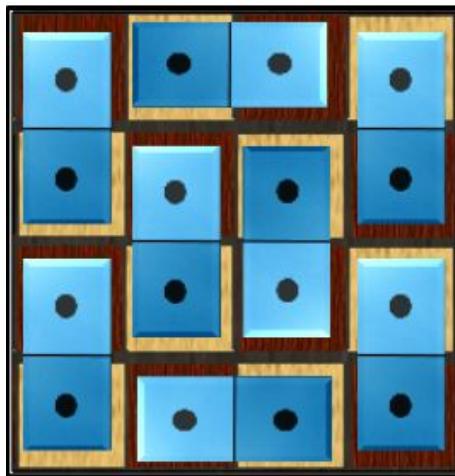


Graful grid

# Probleme

Tabla

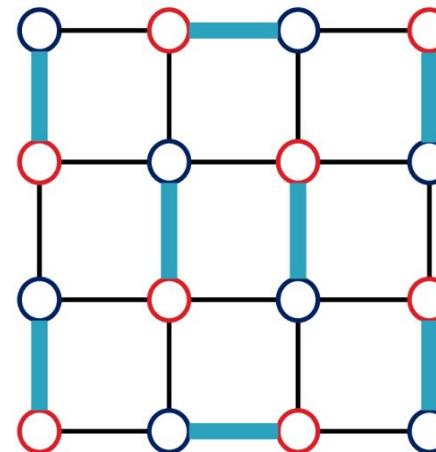
Acoperire



Graful grid



Cuplaj perfect



Graful grid

# Probleme

## Acoperirea unei table cu piese de domino

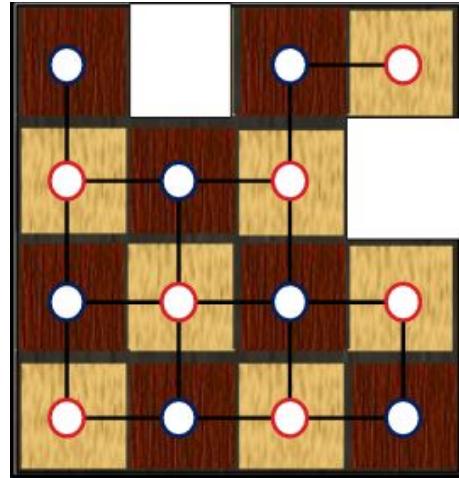
- Tabla poate fi acoperită  $\Leftrightarrow m \times n$  par



Dacă tabla de șah poate fi acoperită, dar eliminăm două pătrățele din ea, în ce condiții rămâne acoperibilă?

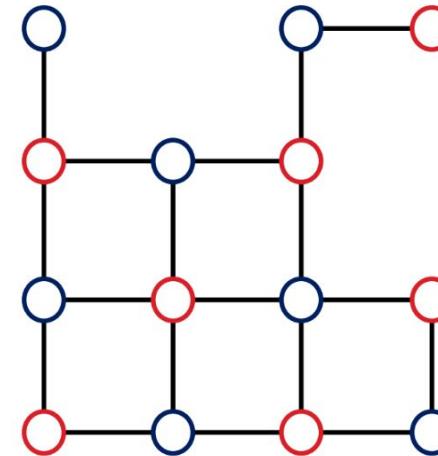
# Probleme

Tabla  
Acoperire



⇒  
⇒

Graful grid  
Cuplaj perfect



# Probleme

## Acoperirea unei table cu piese de domino

- Tabla poate fi acoperită  $\Leftrightarrow m \times n$  par

Dacă tabla de șah poate fi acoperită, dar eliminăm două pătrățele din ea, în ce condiții rămâne acoperibilă?



**Dacă și numai dacă pătrățelele au culori diferite**

Aplicație:  
Sistem de reprezentanți  
distincți pentru submulțimi

# Sistem de reprezentanți distincți

Fie A - mulțime finită cu n elemente

$$X_1, X_2, \dots, X_m \subseteq A$$

S. n. **sistem de reprezentanți distincți** pentru colecția de submulțimi  $(X_1, X_2, \dots, X_m)$  un vector  $(r_1, r_2, \dots, r_m)$  cu proprietățile:

- $r_i \in X_i, \quad \forall i = 1, \dots, m$
- $r_i \neq r_j, \quad \forall i, j = 1, \dots, m, \quad i \neq j$

$$A = \{1, 2, 3, 4\}$$

$$X_1 = \{2, 3\} \quad \Rightarrow r_1 = 2$$

$$X_2 = \{1, 3, 4\} \quad \Rightarrow r_2 = 3$$

$$X_3 = \{2, 4\} \quad \Rightarrow r_3 = 4$$

# Sistem de reprezentanți distincți

Nu orice colecție de submulțimi admite un sistem de reprezentanți distincți.

**Exemplu:**

$$A = \{1, 2, 3, 4\}$$

$$X_1 = \{2, 3\}$$

$$X_2 = \{3\}$$

$$X_3 = \{2\}$$

# Sistem de reprezentanți distincți



**Condiții necesare și suficiente pentru existența unui sistem de reprezentanți distincți ai unei colecții de submulțimi din A?**

# Sistem de reprezentanți distincți

**Condiții necesare și suficiente pentru existența unui sistem de reprezentanți distincti ai unei colectii de submultimi din A?**



## Modelăm problema cu ajutorul unui graf bipartit

- vârf  $x_i$  - asociat submulțimii  $X_i$ ,  $i = 1, \dots, m$   
⇒ mulțimea  $\mathbf{X}$  de vârfuri
  - vârf  $a_j$  - asociat fiecărui element din A,  $j = 1, \dots, n$   
⇒ mulțimea  $\mathbf{Y}$  de vârfuri
  - muchie de la  $x_i$  la  $a_j$     $\Leftrightarrow$      $a_j \in X_i$

# Sistem de reprezentanți distinți

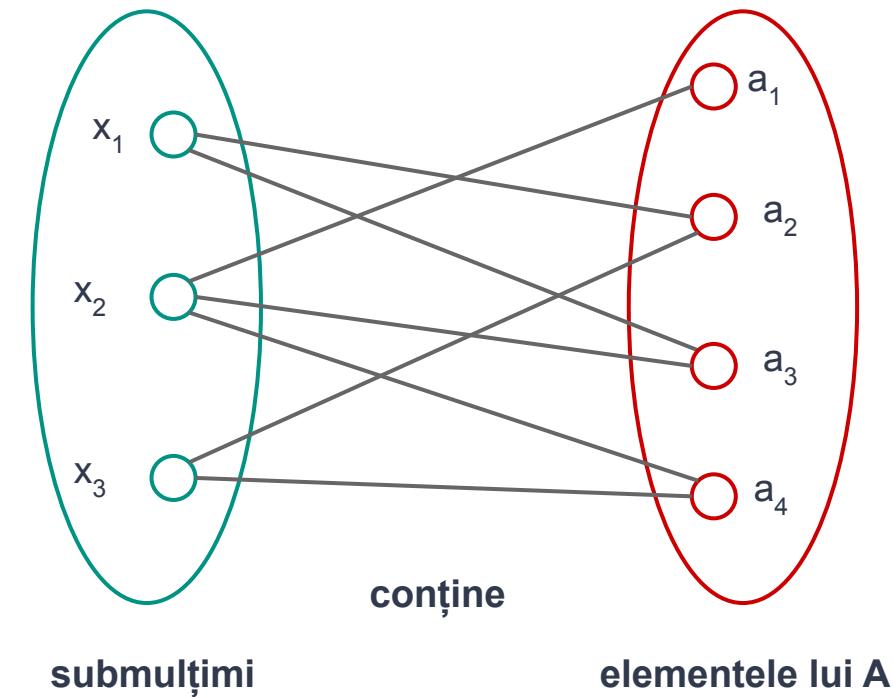
**Exemplu:**

$$A = \{1, 2, 3, 4\}$$

$$X_1 = \{2, 3\}$$

$$X_2 = \{1, 3, 4\}$$

$$X_3 = \{2, 4\}$$



# Sistem de reprezentanți distincți

## **Observatie:**

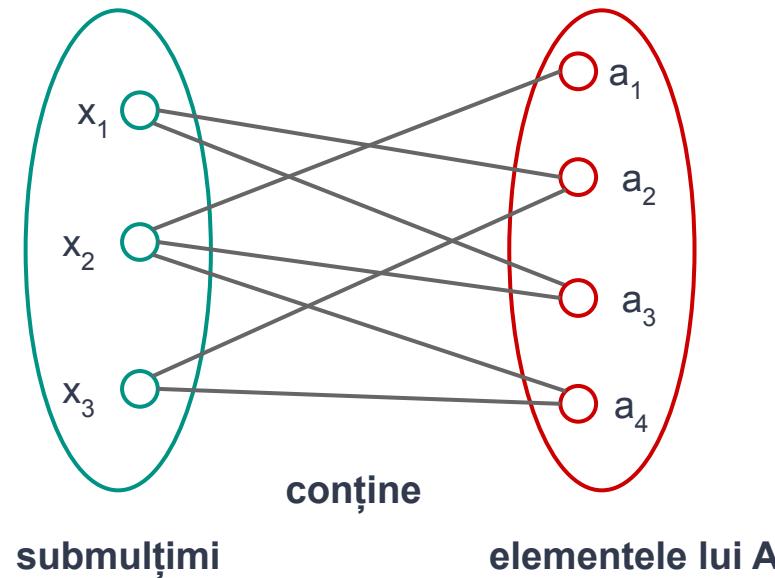
Există un sistem de reprezentanți pentru colecția de submulțimi ( $X_1, X_2, \dots, X_m$ ) ale lui A  $\Leftrightarrow$   
**există un cuplaj al lui X în Y în graful asociat**

$$X_1 = \{2, 3\}$$

$$X_2 = \{1, 3, 4\}$$

$$X_3 = \{2, 4\}$$

$$r = (2, 3, 4)$$



# Sistem de reprezentanți distincți

## Observație:

Există un sistem de reprezentanți pentru colecția de submulțimi ( $X_1, X_2, \dots, X_m$ ) ale lui A  $\Leftrightarrow$  există un cuplaj al lui X în Y în graful asociat

## Teorema lui HALL:

Dacă pentru orice submulțime  $S = \{x_{i1}, x_{i2}, \dots, x_{ik}\} \subseteq X$  avem

$$|N(S)| \geq |S| = k \text{ (imaginării lui } S)$$

$$N(S) = X_{i1} \cup X_{i2} \cup \dots \cup X_{ik}$$

Astfel, are loc următorul rezultat

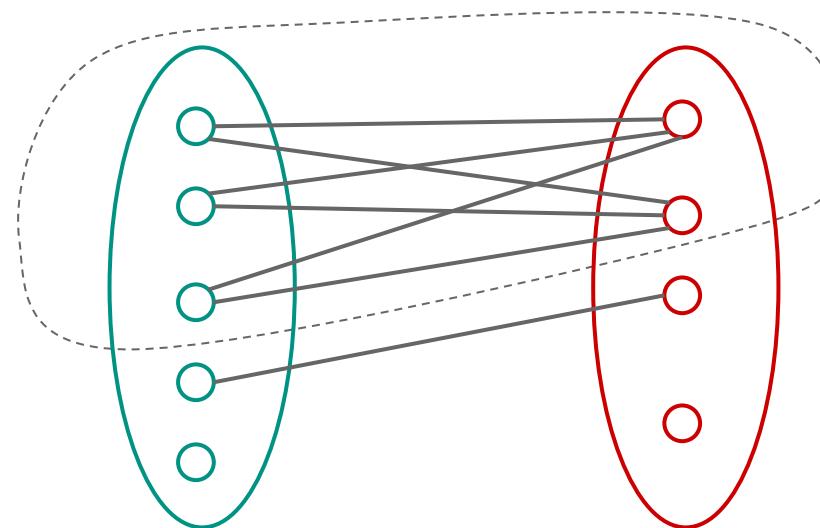
# Sistem de reprezentanți distincți

**Teoremă - existența unui sistem de reprezentanți distincți**

Fie A o mulțime finită și  $(X_1, X_2, \dots, X_m)$  o colecție de submulțimi din A.

Colecția nu are un sistem de reprezentanți distincți  $\Leftrightarrow$

$\exists k$  submulțimi în colecție a căror reuniune are mai puțin de k elemente





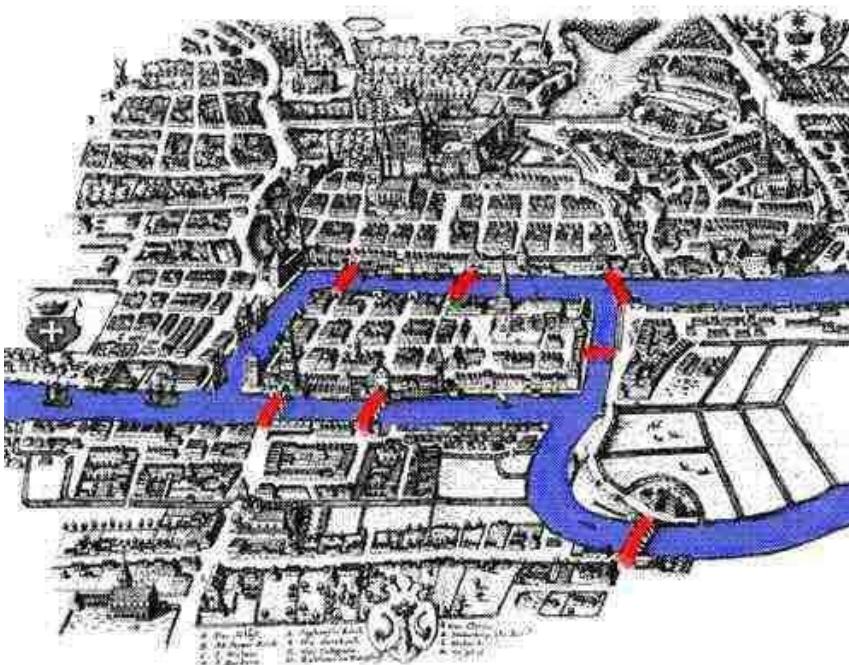
# Grafuri euleriene



# Istoric. Aplicații

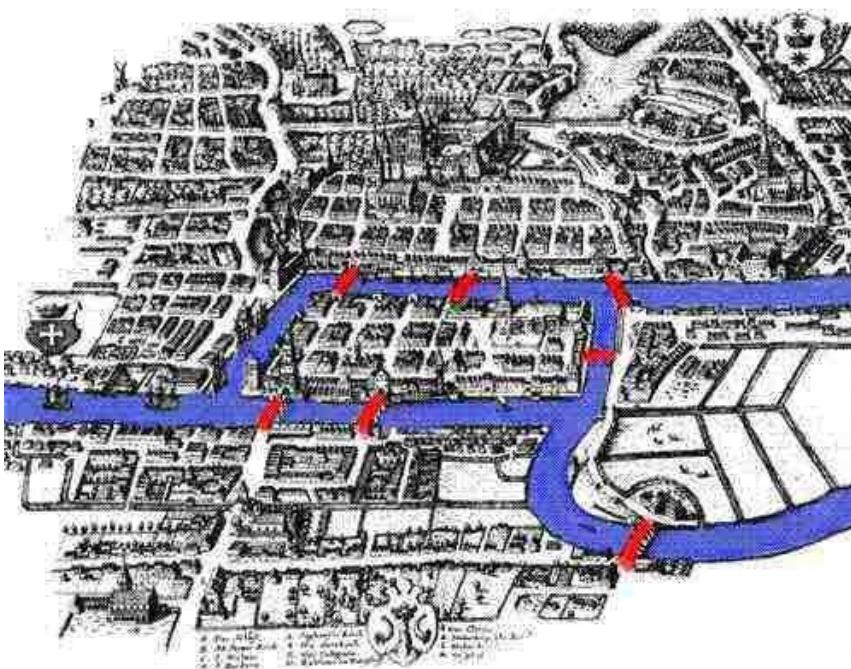
din cursul 1

# Problema celor 7 poduri din Königsberg

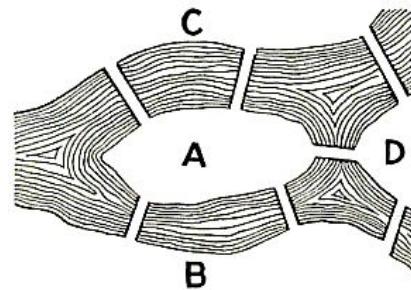


Este posibil ca un om să facă o plimbare în care să treacă pe toate cele 7 poduri, o singură dată?

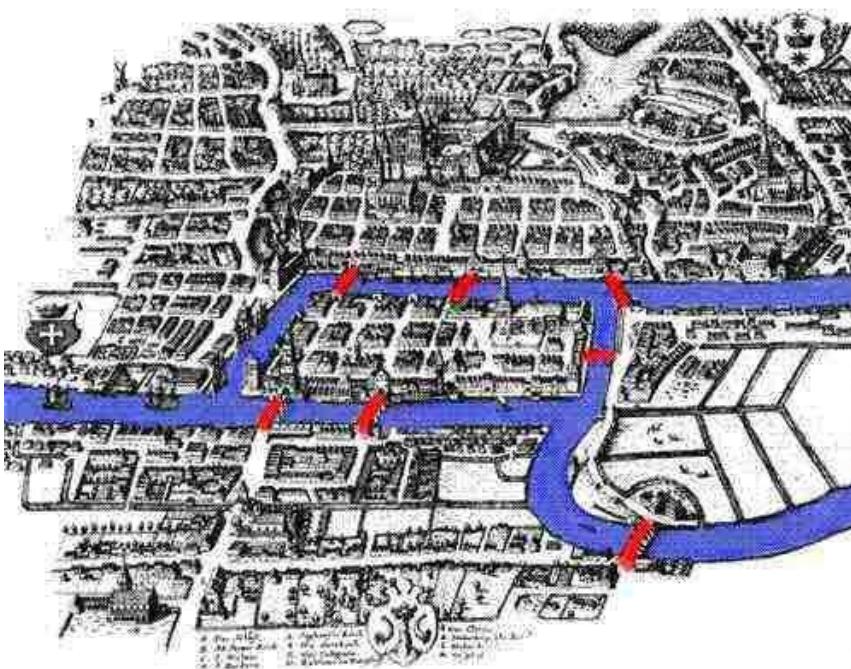
# Problema celor 7 poduri din Königsberg



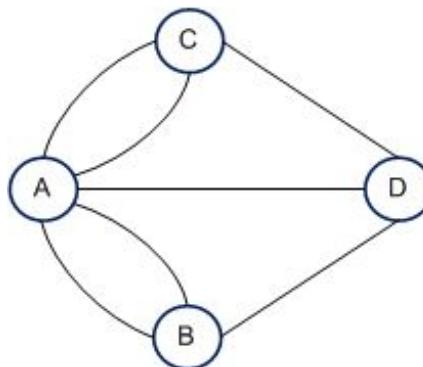
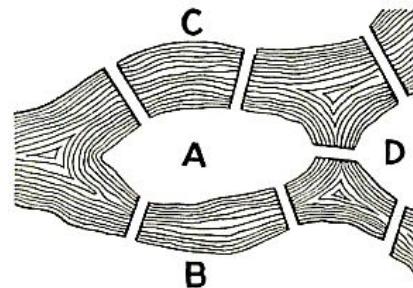
Modelare:



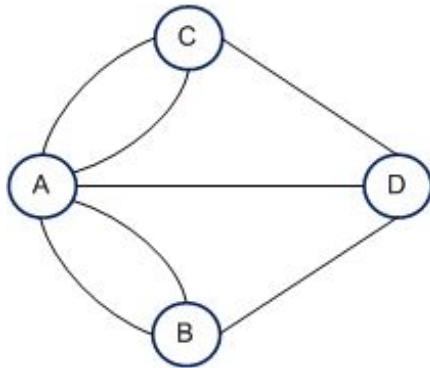
# Problema celor 7 poduri din Königsberg



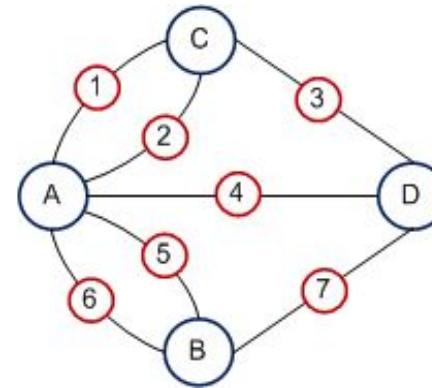
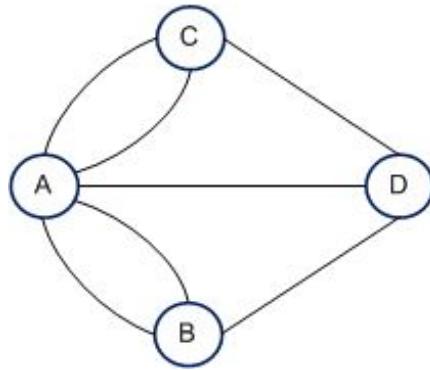
Modelare:



# Problema celor 7 poduri din Königsberg

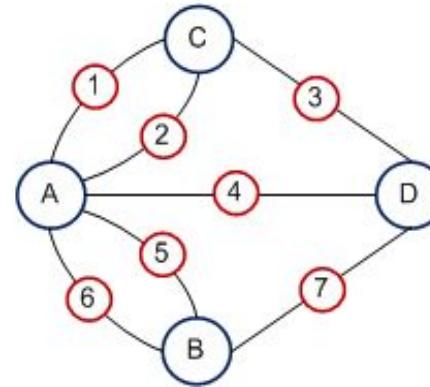
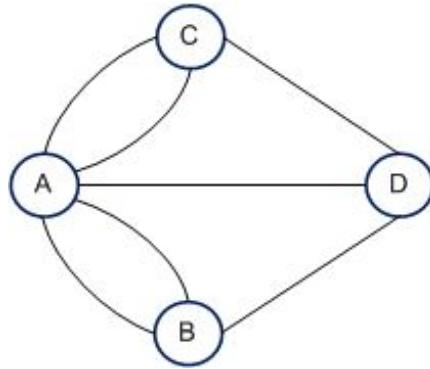


# Problema celor 7 poduri din Königsberg



graf simplu

# Problema celor 7 poduri din Königsberg



□ 1736 - Leonhard Euler

*Solutio problematis ad geometriam situs pertinentis*

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

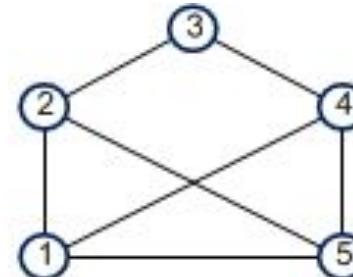
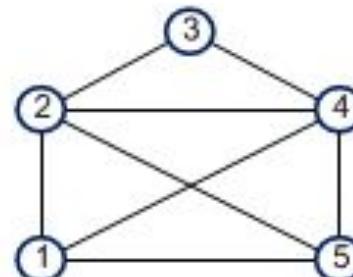
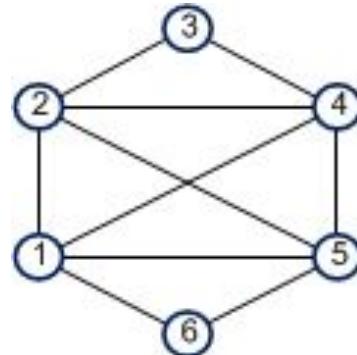
**Graf eulerian**

# Problema celor 7 poduri din Königsberg

## Interpretare

Se poate desena diagrama printr-o curbă continuă închisă, fără a ridica pixul de pe hârtie și fără a desena o linie de două ori (în plus: să terminăm desenul în punctul în care l-am început)?

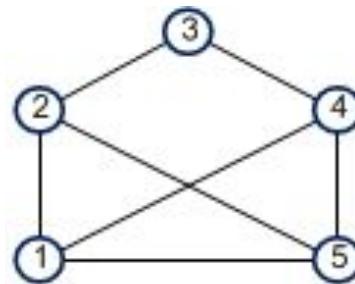
- tăierea unui material



# Problema celor 7 poduri din Königsberg

## Interpretare

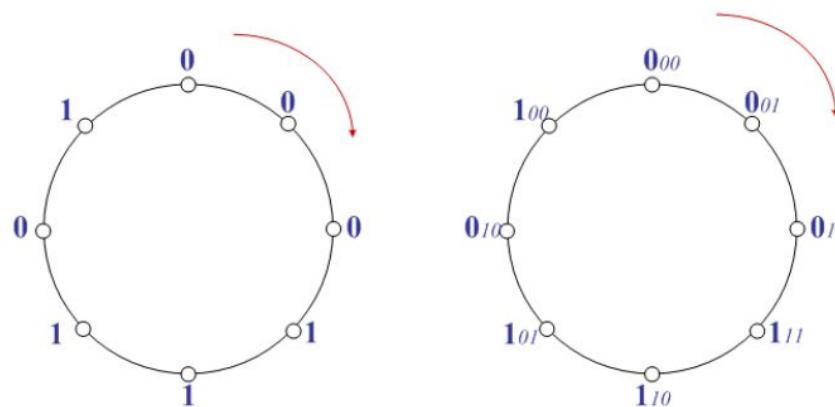
De câte ori (minim) trebuie să ridicăm pixul de pe hârtie pentru a desena diagrama?



# Grafuri euleriene

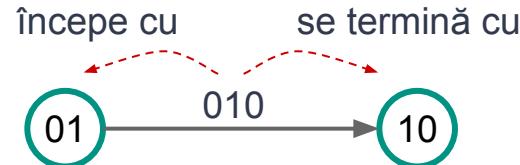
## Problema lui POSTHUMUS

- $f(n) =$  numărul minim de cifre de 0 și 1 care se pot dispune circular a.î. între cele  $f(n)$  secvențe de lungime  $n$  de cifre successive apar toți cei  $2^n$  vectori de lungime  $n$  peste  $\{0, 1\}$  (citite în același sens).
- Evident,  $f(n) \geq 2^n$ . **Are loc chiar egalitate?**

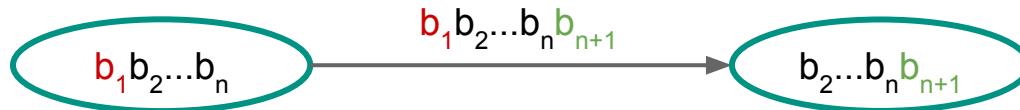


# Grafuri de Bruijn

- Etichete pe arce:

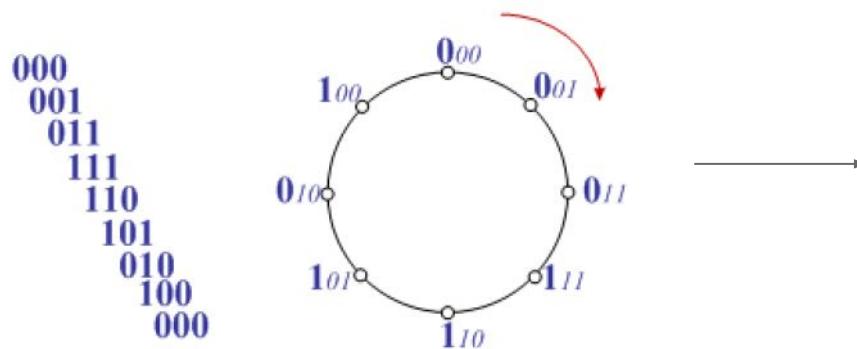
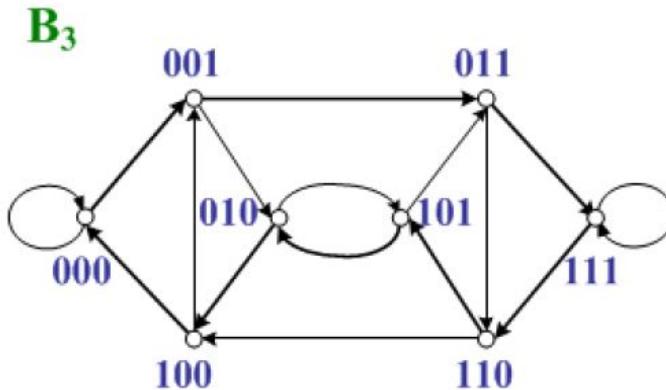
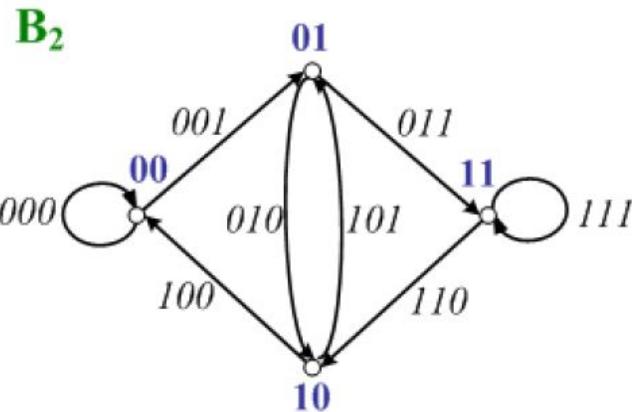


- În general:



$$b_i \in \{0, 1\}$$

# Grafuri de Bruijn



Soluția la problema lui  
**POSTHUMUS** pentru  $n = 3 \Leftrightarrow$   
etichetele arcelor unui circuit  
eulerian în graful  $B_2$

# Grafuri euleriene

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  
 $E(P) = E(G)$

# Grafuri euleriene

## Observație:

Fie  $P = [v_1, \dots, v_k]$

- Dacă  $v_1 \neq v_k$ , atunci vârfurile interne din  $P$  au gradul în  $P$  par, iar extremitățile au gradul în  $P$  impar
- Dacă  $v_1 = v_k$ , atunci toate vârfurile din  $P$  au gradul în  $P$  par

# Grafuri euleriene

Lemă:

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

Atunci, pentru orice  $x \in V$ , există un ciclu  $C$  în  $G$  cu  $x \in V(C)$

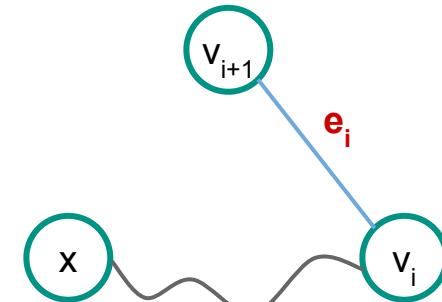
(ciclu care conține  $x$ , nu neapărat eulerian, nici neapărat elementar).

# Grafuri euleriene

Demonstrație - Algoritm de determinare a unui ciclu care conține  $x$ :

- $i = 1, v_1 = x$
- $E(C) = \emptyset$
- Repetă

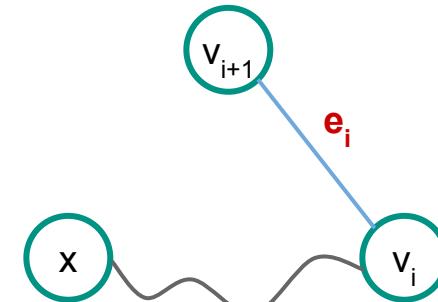
până când  $v_i = x$



# Grafuri euleriene

Demonstrație - Algoritm de determinare a unui ciclu care conține x:

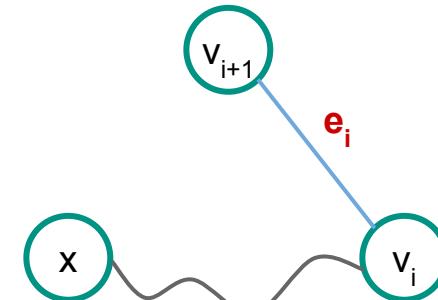
- $i = 1, v_1 = x$
- $E(C) = \emptyset$
- Repetă
  - selectează  $e_i = v_i v_{i+1} \in E(G) - E(C)$
  - $E(C) = E(C) \cup \{e_i\}$
  - $i = i + 1$
- până când  $v_i = x$



# Grafuri euleriene

Demonstrație - Algoritm de determinare a unui ciclu care conține  $x$ :

- $i = 1, v_1 = x$
- $E(C) = \emptyset$
- Repetă
  - selectează  $e_i = v_i v_{i+1} \in E(G) - E(C)$
  - $E(C) = E(C) \cup \{e_i\}$
  - $i = i + 1$până când  $v_i = x$



Algoritmul este corect deoarece:

# Grafuri euleriene

Demonstrație - Algoritm de determinare a unui ciclu care conține x:

- $i = 1, v_1 = x$
- $E(C) = \emptyset$
- Repetă

- selectează  $e_i = v_i v_{i+1} \in E(G) - E(C)$  ←————
- $E(C) = E(C) \cup \{e_i\}$
- $i = i + 1$

până când  $v_i = x$

Dacă  $v_i \neq x$ , atunci  $d_C(v_i)$  este impar.  
Din ipoteză,  $d_G(v_i)$  este par, deci  
 $d_{G-E(C)}(v_i)$  este impar deci  
 $d_{G-E(C)}(v_i) > 0$

⇒ muchia  $e_i$  există

# Grafuri euleriene

Demonstrație - Algoritm de determinare a unui ciclu care conține x:

- $i = 1, v_1 = x$
- $E(C) = \emptyset$
- Repetă

- selectează  $e_i = v_i v_{i+1} \in E(G) - E(C)$  ←
- $E(C) = E(C) \cup \{e_i\}$
- $i = i + 1$

până când  $v_i = x$



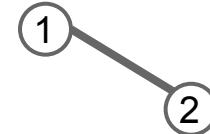
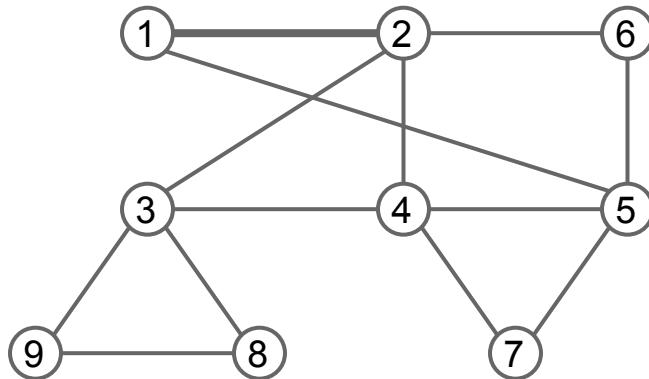
Dacă  $v_i \neq x$ , atunci  $d_C(v_i)$  este impar.  
Din ipoteză,  $d_G(v_i)$  este par, deci  
 $d_{G-E(C)}(v_i) > 0$

⇒ muchia  $e_i$  există

$|E(G)| < \infty$ , deci algoritmul se termină  
( $v_i$  ajunge egal cu x)

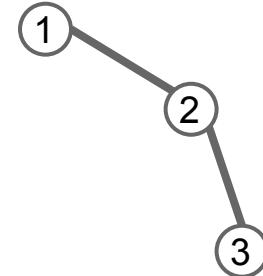
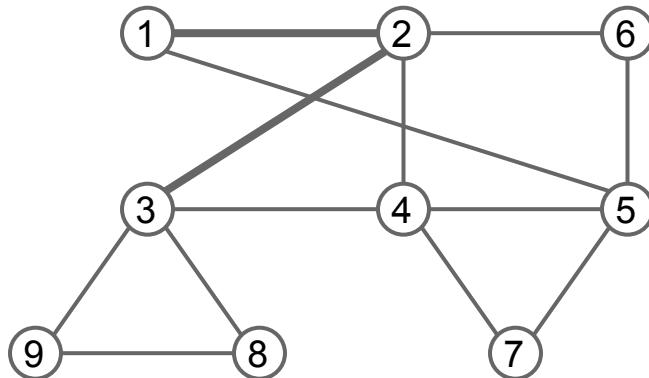
# Grafuri euleriene

$x = 1$



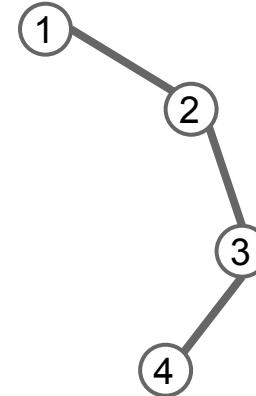
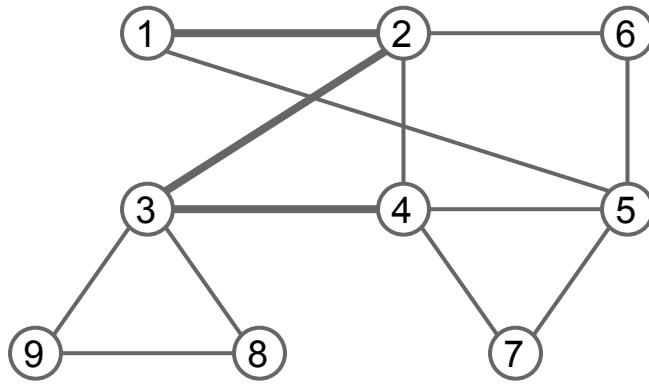
# Grafuri euleriene

$x = 1$



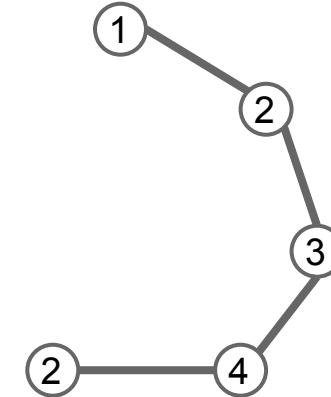
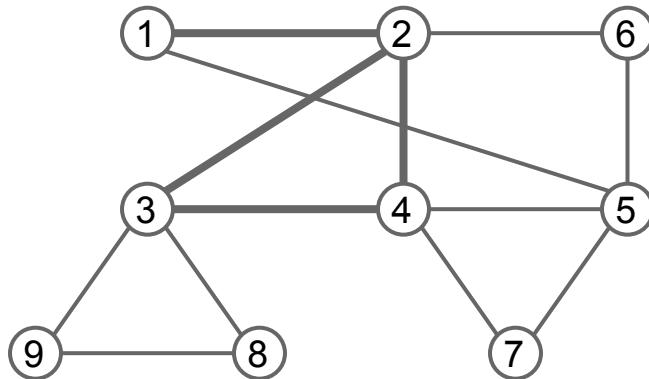
# Grafuri euleriene

$x = 1$



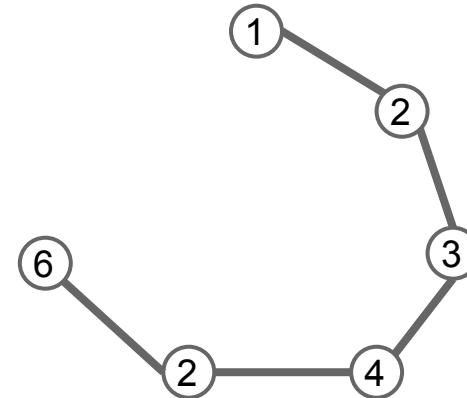
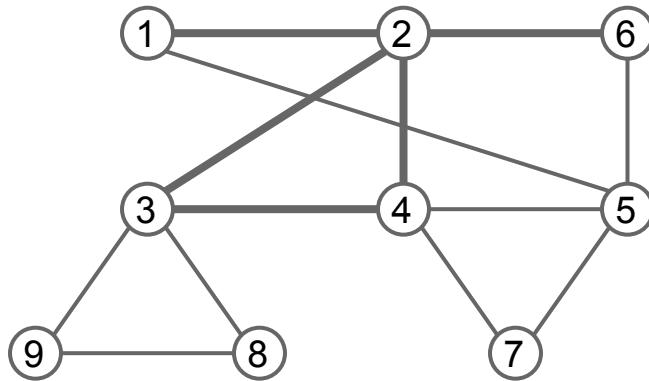
# Grafuri euleriene

$x = 1$



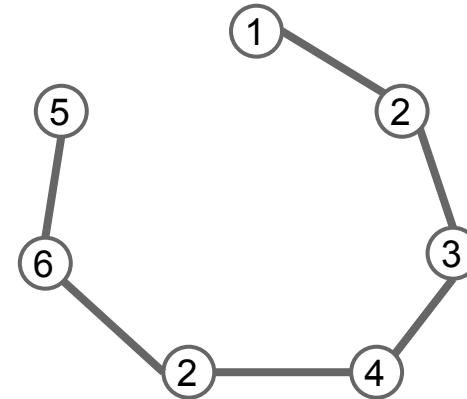
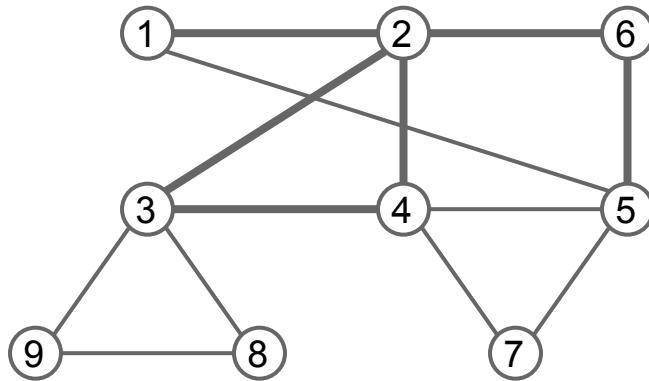
# Grafuri euleriene

$x = 1$



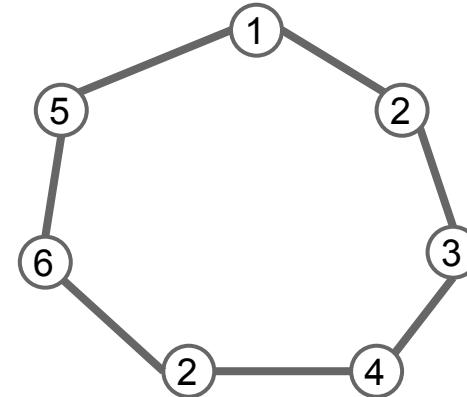
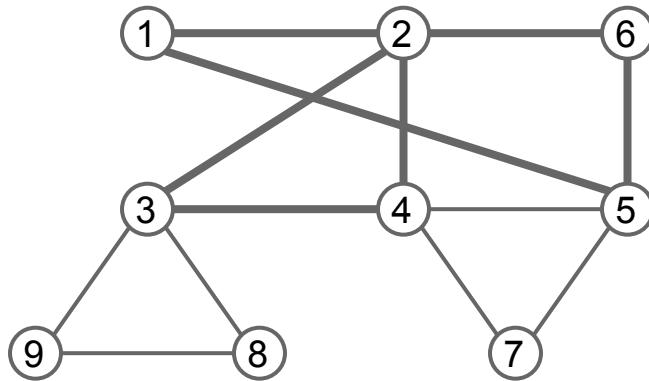
# Grafuri euleriene

$x = 1$



# Grafuri euleriene

$x = 1$



# Grafuri euleriene

## Teorema lui Euler

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

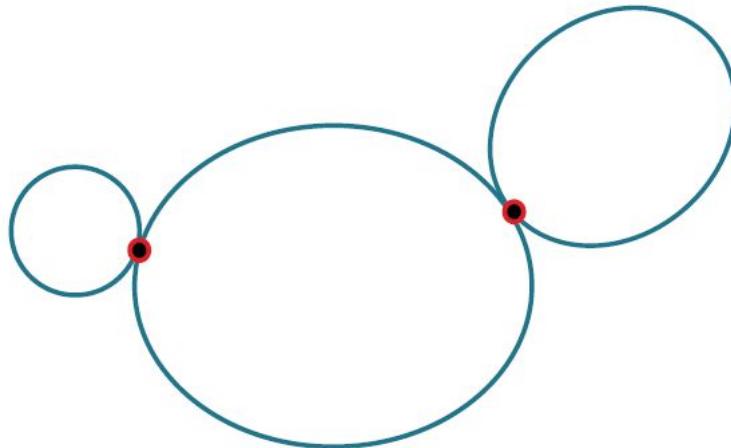
Atunci

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

# Algoritmul lui Hierholzer

Determinarea unui ciclu eulerian într-un graf conex (sau un graf conex + vârfuri izolate) cu toate vârfurile de grad par.

- bazat pe ideea demonstrației teoremei lui Euler - **fuziune de cicluri (succesiv)**



# Algoritmul lui Hierholzer

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

# 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 Lemă)

# 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 Lemă)
- **cât timp  $|E(C)| < |E(G)|$  execută**
  - selectează  $v \in V(C)$  cu  $d_{G-E(C)}(v) > 0$  (în care sunt indicate muchii care nu aparțin lui  $C$ )

# 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 Lemă)
- **cât timp  $|E(C)| < |E(G)|$  execută**
  - selectează  $v \in V(C)$  cu  $d_{G-E(C)}(v) > 0$  (în care sunt indicate muchii care nu aparțin lui  $C$ )
  - construiește  $C'$  un ciclu în  $G - E(C)$  care începe cu  $v$

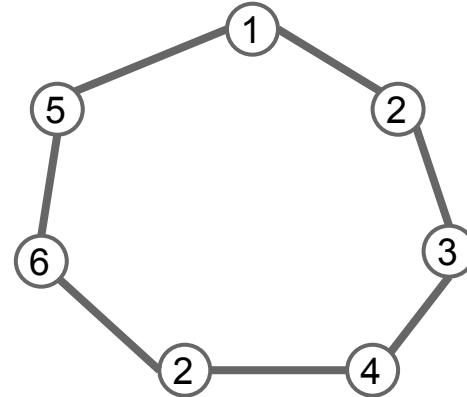
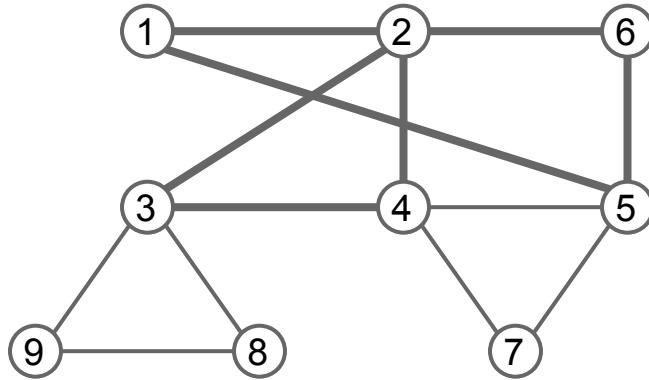
# Algoritmul lui Hierholzer

- **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 Lemă)
- **cât timp  $|E(C)| < |E(G)|$  execută**
  - selectează  $v \in V(C)$  cu  $d_{G-E(C)}(v) > 0$  (în care sunt indicate 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  $v$

# Algoritmul lui Hierholzer

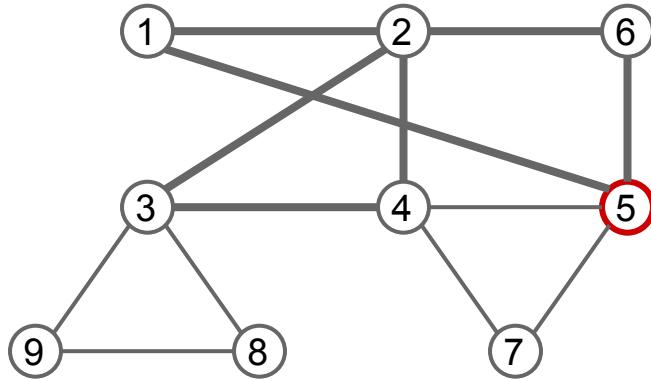
- **Pasul 0** - verificare condiții (conex + vf. izolate, grade pare)
- **Pasul 1**
  - alege  $v \in V$  arbitrar
  - construiește  $C$  un ciclu în  $G$  care începe cu  $v$  (cu algoritmul din Lemă)
- cât timp  $|E(C)| < |E(G)|$  execută
  - selectează  $v \in V(C)$  cu  $d_{G-E(C)}(v) > 0$  (în care sunt indicate 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  $C$

# Algoritmul lui Hierholzer

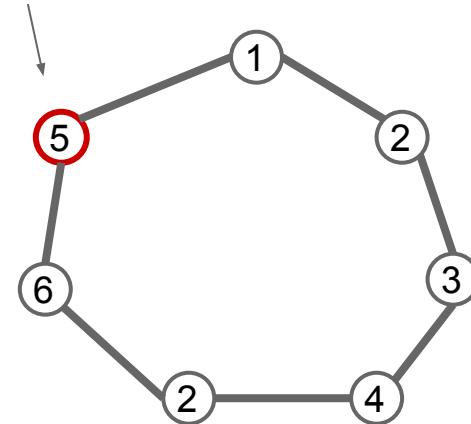


$$C = [1, 2, 3, 4, 2, 6, 5, 1]$$

# Algoritmul lui Hierholzer

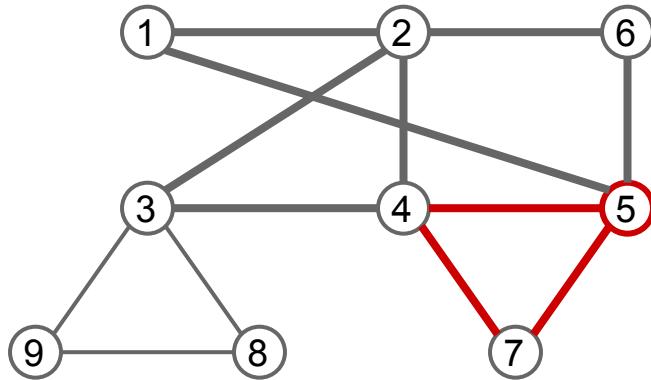


Alegem un vîrf din C în care  
mai sunt incidente muchii, de  
exemplu  $v = 5$

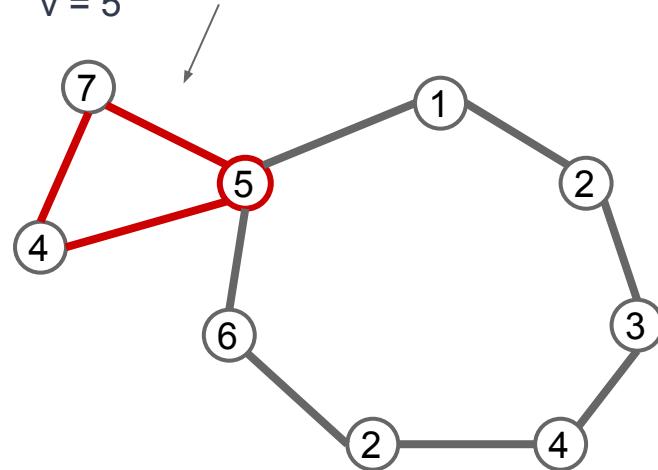


$$C = [1, 2, 3, 4, 2, 6, \mathbf{5}, 1]$$

# Algoritmul lui Hierholzer



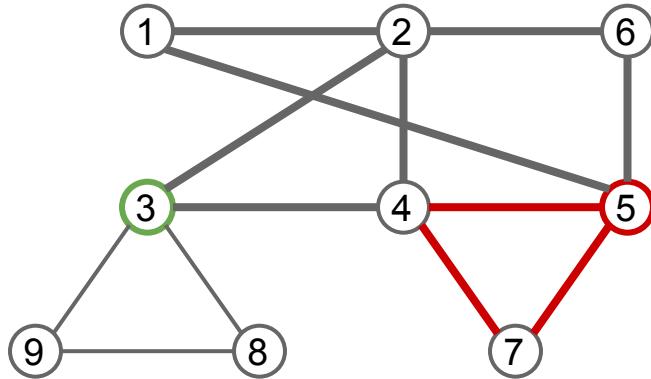
Construim un ciclu  $C'$  cu  
muchiile rămase, care conține  
 $v = 5$



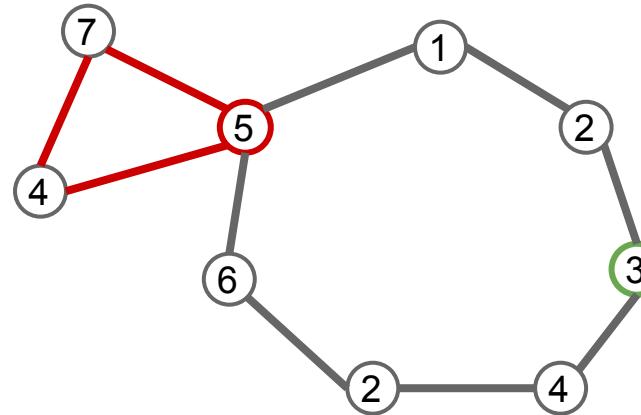
⇒ Un nou ciclu obținut prin fuziunea  
celor două cicluri

$$C = [1, 2, 3, 4, 2, 6, \mathbf{5}, \mathbf{4}, \mathbf{7}, \mathbf{5}, 1]$$

# Algoritmul lui Hierholzer

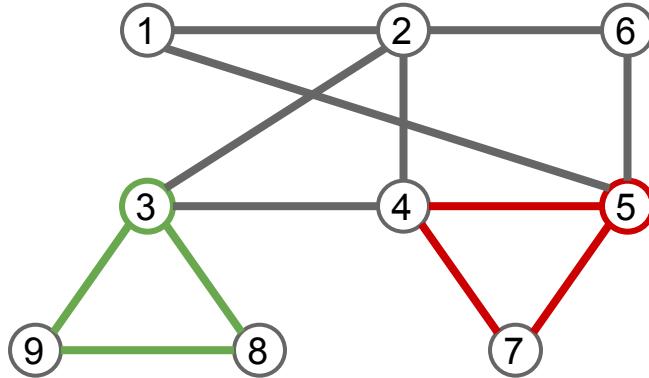


Alegem un vîrf din C în care mai sunt incidente muchii, de exemplu  $v = 3$

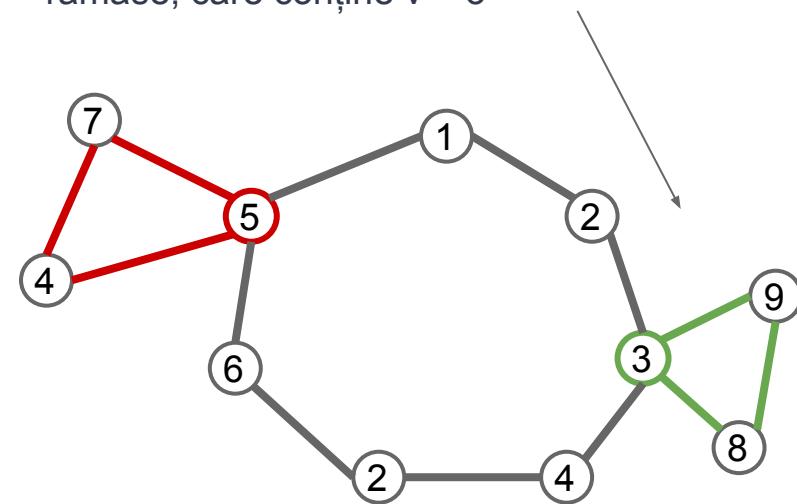


$$C = [1, 2, 3, 4, 2, 6, 5, 4, 7, 5, 1]$$

# Algoritmul lui Hierholzer



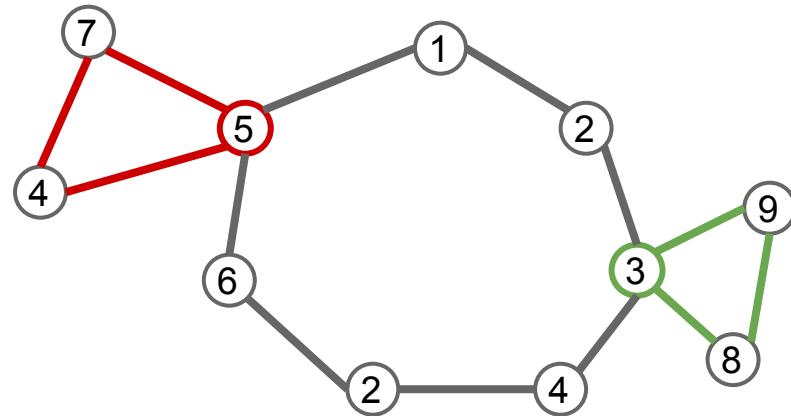
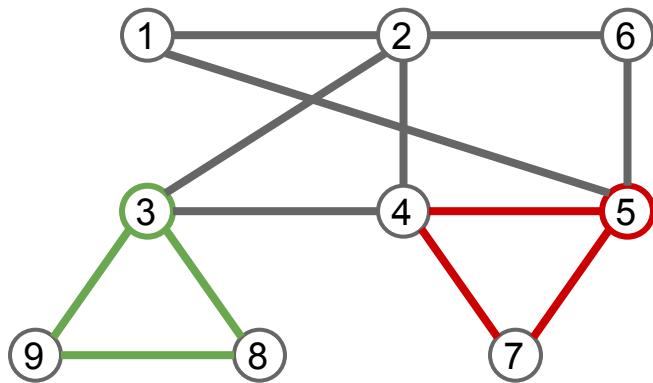
Construim un ciclu  $C'$  cu muchiile rămase, care conține  $v = 3$



⇒ Un nou ciclu obținut prin fuziunea celor două cicluri

$$C = [1, 2, \mathbf{3}, \mathbf{8}, \mathbf{9}, \mathbf{3}, 4, 2, 6, 5, 4, 7, 5, 1]$$

# Algoritmul lui Hierholzer



$$C = [1, 2, 3, 8, 9, 3, 4, 2, 6, 5, 4, 7, 5, 1]$$

Ciclul conține toate muchiile  
⇒ este eulerian

# Algoritmul lui Hierholzer

**Complexitate -  $O(m)$**

# Algoritmul lui Hierholzer

## Posibile implementări

- Varianta 1** - stiva (dfs)
- Muchiile folosite - marcate (nu neapărat șterse)

# Algoritmul lui Hierholzer

## Posibile implementări

### □ Varianta 2 - posibilă implementare recursivă

```
euler(nod n)
    cât timp  $d(v) > 0$ 
        alege vw o muchie incidentă în v
        șterge muchia vw din G
        euler(w)
    C = C + v // adăugăm v la ciclul C
```

### Inițial

```
C =  $\emptyset$ 
euler(1) // pornim construcția din vârful 1
```

# Algoritmul lui Hierholzer

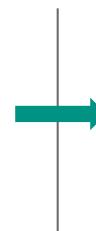
## Possible implementări

### □ Varianta 2 - posibilă implementare recursivă

```
euler(nod n)
    cât timp  $d(v) > 0$ 
        alege vw o muchie incidentă în v
        șterge muchia vw din G
        euler(w)
    C = C + v // adăugăm v la ciclul C
```

**Observație** - putem alege muchiile incidente în v, de exemplu, în ordinea dată de listele de adiacență

```
cât timp  $d(v) > 0$ 
    alege vw o muchie incidentă în v
    șterge muchia vw din G
    euler(w)
```



```
pentru  $vw \in E$ 
    șterge muchia vw din G
    euler(w)
```

# Algoritmul lui Hierholzer

## Posibile implementări

### □ Varianta 2 - posibilă implementare recursivă

```
euler(nod n)
    cât timp  $d(v) > 0$ 
        alege vw o muchie incidentă în v
        șterge muchia vw din G
        euler(w)
    C = C + v // adăugăm v la ciclul C
```

### Inițial

```
C =  $\emptyset$ 
euler(1) // pornim construcția din vârful 1
```

# Lanțuri euleriene

## 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

# Descompuneri euleriene în lanțuri (suplimentar)

**k-descompunere euleriană** în lanțuri a unui graf  $G =$

o mulțime de  $k$  lanțuri simple, muchie-disjuncte

$$\Delta = \{P_1, P_2, \dots, P_k\}$$

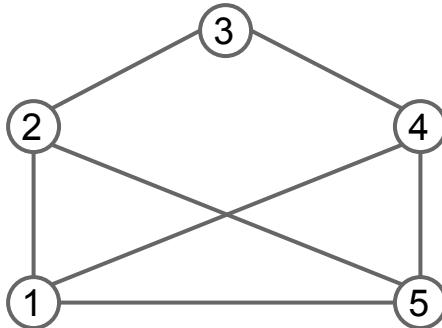
ale căror muchii induc o  $k$ -partiție a lui  $E(G)$

$$E(G) = E(P_1) \cup E(P_2) \cup \dots \cup E(P_k)$$

# Descompuneri euleriene în lanțuri (suplimentar)

## Interpretare

De câte ori (minim) trebuie să ridicăm creionul de pe hârtie, pentru a desena diagrama?



# Descompuneri euleriene în lanțuri (suplimentar)

## Teoremă - Descompunerea 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.

# Grafuri orientate euleriene

## Teorema lui Euler

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

Atunci

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

# Lanțuri euleriene

## Teorema lui Euler

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

Atunci

$G$  are un drum eulerian  $\Leftrightarrow$

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

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

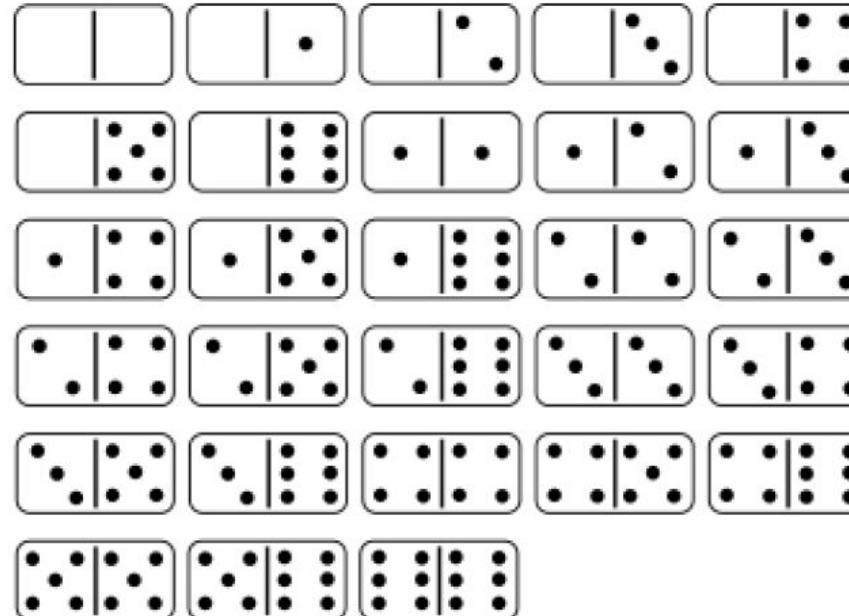
$$\exists y \in V \text{ cu } d_G^-(y) = d_G^+(y) + 1,$$

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

# Grafuri euleriene

## Problemă - joc domino

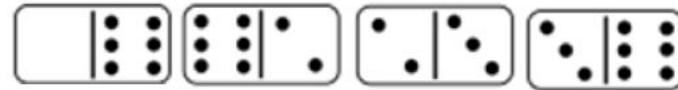
Piesă de domino - două fețe, numere 0, ..., n, de obicei  $n = 6$



# Grafuri euleriene

## Problemă - joc domino

Şir de piese de domino - respectă regula de construcție: primul număr de pe piesa adăugată la şir = al doilea număr de pe ultima piesă din şir



# Grafuri euleriene

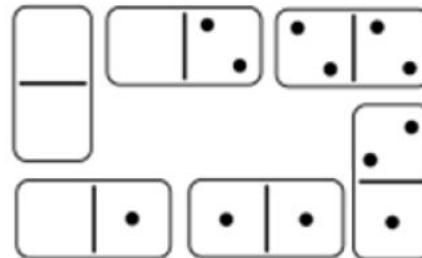
## Problemă - joc domino

Se poate forma un sir de piese de domino care sa contină toate piesele + să se termine cu același număr cu care a început (un sir circular)?

# Grafuri euleriene

## Problemă - joc domino

Exemplu - dacă folosim doar piese cu numere 0, ..., 2, putem forma un ciclu

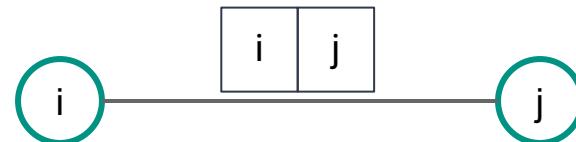


# Grafuri euleriene

## Problemă - joc domino

Graf asociat

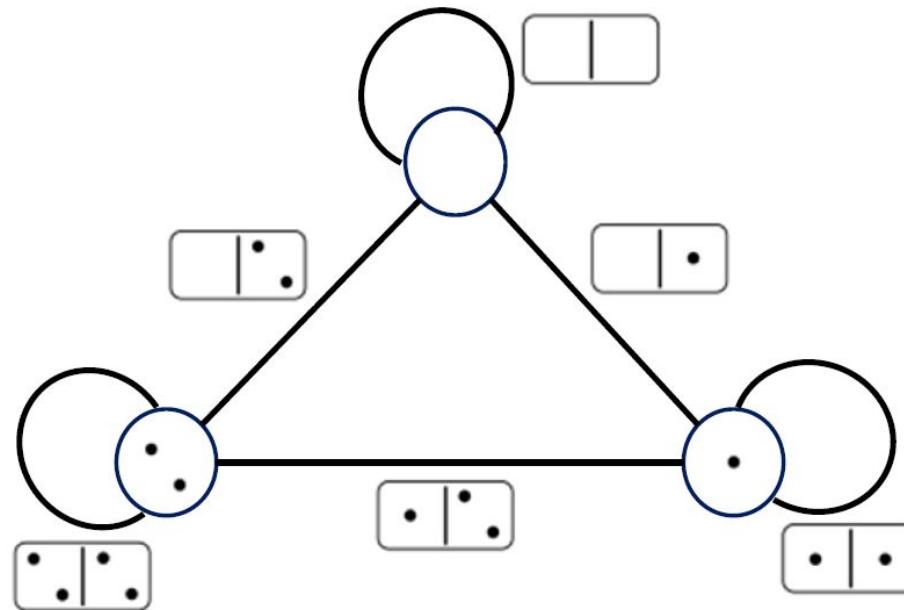
- vârfuri - numerele de pe piese
- muchii - perechi de numere (piesele)
- se pot lipi doar piese asociate muchiilor adiacente



# Grafuri euleriene

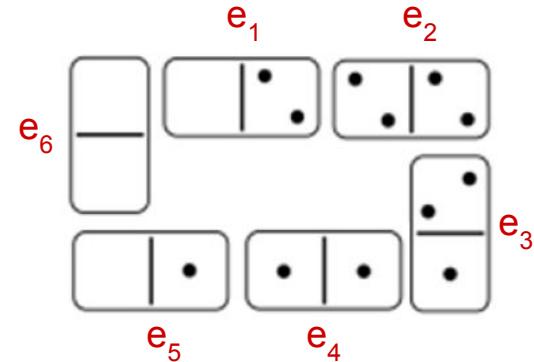
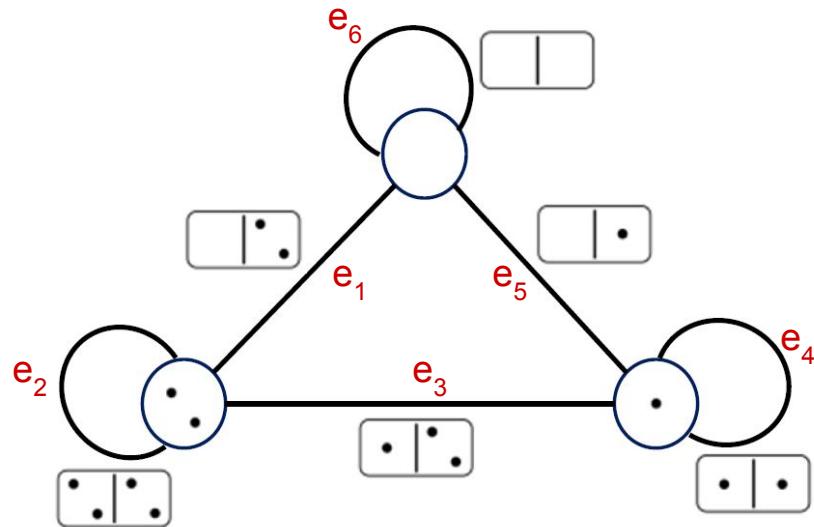
Problemă - joc domino

$$n = 2$$



# Grafuri euleriene

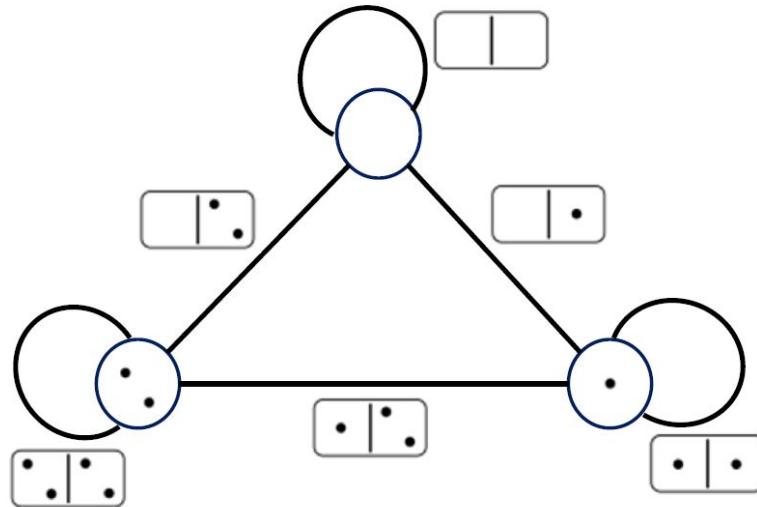
Există ciclu de piese  $\Leftrightarrow$  există ciclu eulerian în (multi)graf



# Grafuri euleriene



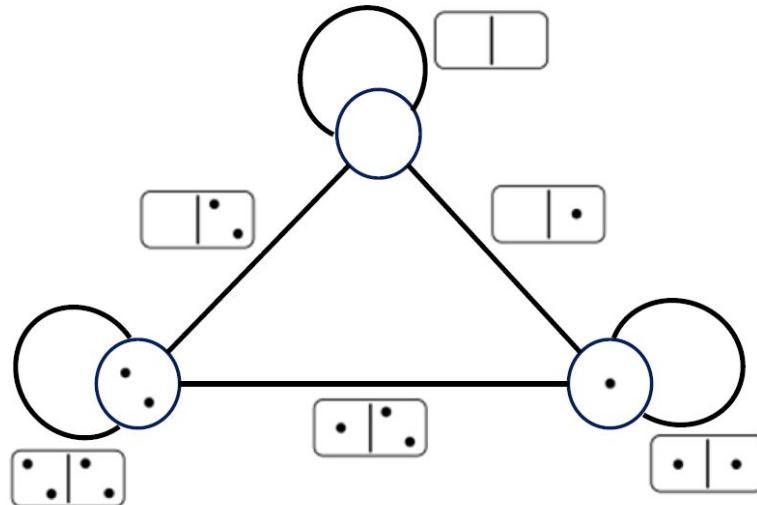
Pentru ce valori ale lui  $n$  există un ciclu eulerian în (multi)graful asociat?



# Grafuri euleriene



Pentru ce valori ale lui  $n$  există un ciclu eulerian în (multi)graful asociat?

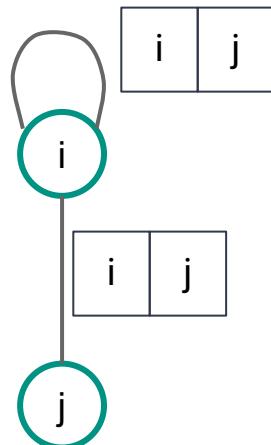


$$d(i) = ?, \text{ pentru } i = 0, \dots, n$$

# Grafuri euleriene



Pentru ce valori ale lui  $n$  există un ciclu eulerian în (multi)graful asociat?



$$d(i) = n + 2$$

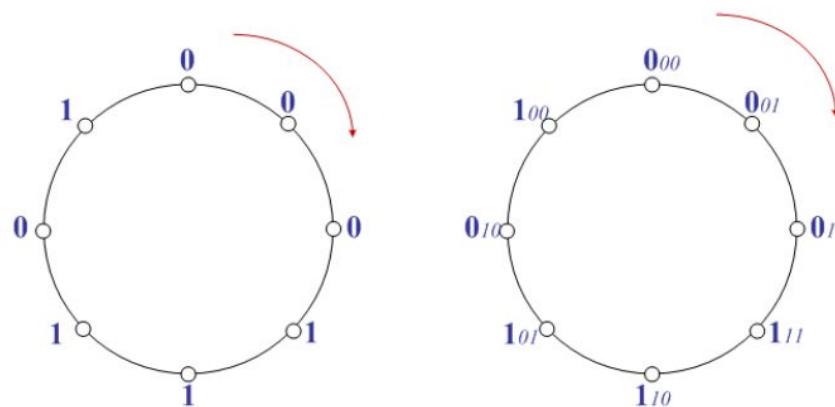
muchiile incidente în  $i$  sunt:  
bucla neetichetată  $(i, i)$  și  
muchiile etichetate  $\{i, j\}$  cu  $i \neq j$ ,  $j \in \{0, \dots, n\}$

$\Rightarrow$  trebuie ca  $n$  să fie par

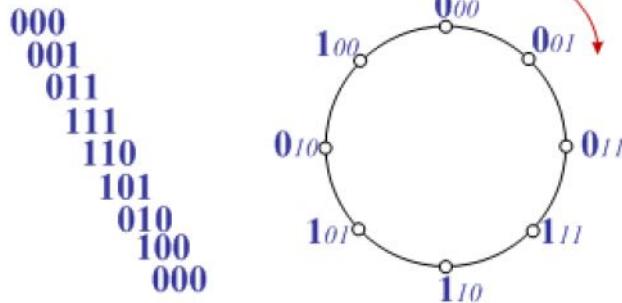
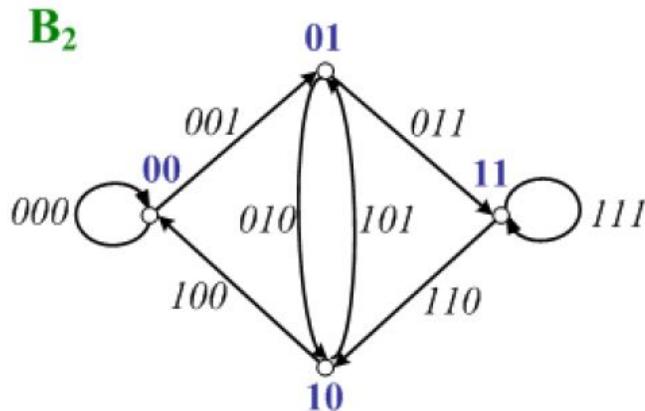
# Grafuri euleriene

## Problema lui POSTHUMUS (Suplimentar)

- $f(n) =$  numărul minim de cifre de 0 și 1 care se pot dispune circular a.î. între cele  $f(n)$  secvențe de lungime  $n$  de cifre successive apar toți cei  $2^n$  vectori de lungime  $n$  peste  $\{0, 1\}$  (citite în același sens).
- Evident,  $f(n) \geq 2^n$ . **Are loc chiar egalitate?**



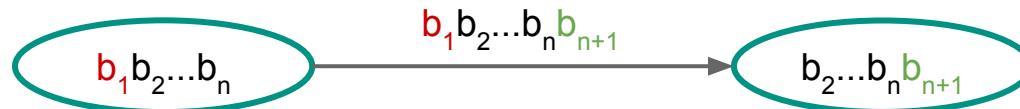
# Grafuri de Bruijn



Prima cifră din etichetele arcelor unui circuit eulerian în  $B_{n-1}$  - soluție pentru problema lui Posthumus

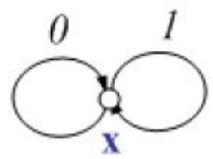
# Grafuri de Bruijn

- Multigraf
- $V(B_n) = \{0, 1\}^n$  (mai general  $\{0, 1, \dots, p\}^n$ )  
(sau cuvinte de lungime n peste un alfabet finit)
- $E(B_n)$  etichetate cu  $\{0, 1\}^{n+1}$  ( $\{0, 1, \dots, p\}^{n+1}$ )  
 $b_1 b_2 \dots b_n b_{n+1}$  etichetează arcul de la  
 $b_1 b_2 \dots b_n$  la  $b_2 \dots b_n b_{n+1}$

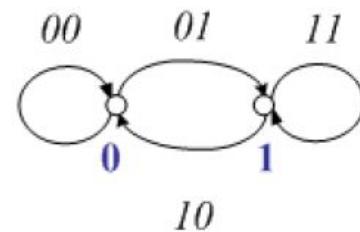


# Grafuri de Bruijn

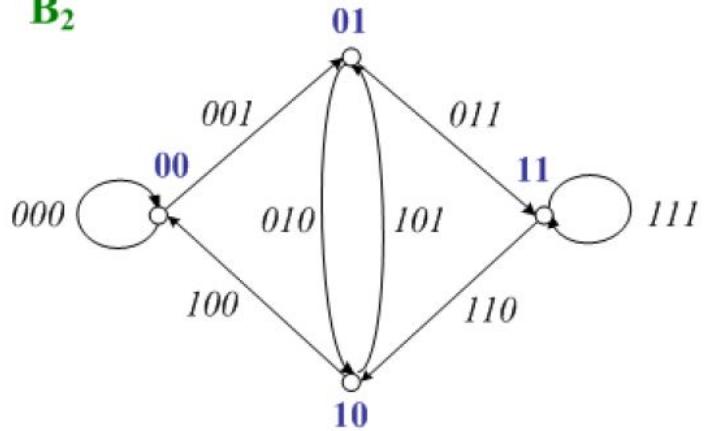
**B<sub>0</sub>**



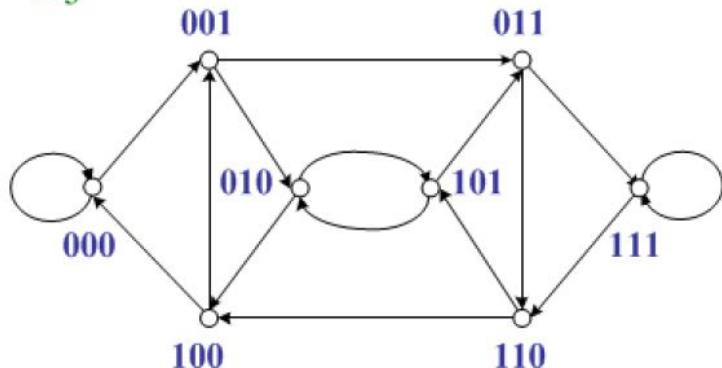
**B<sub>1</sub>**



**B<sub>2</sub>**



**B<sub>3</sub>**

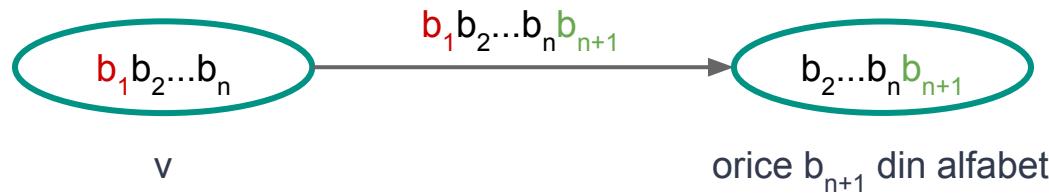


# Grafuri de Bruijn

$B_n$  este eulerian?

$$d^+(v) = ?$$

$$d^-(v) = ?$$

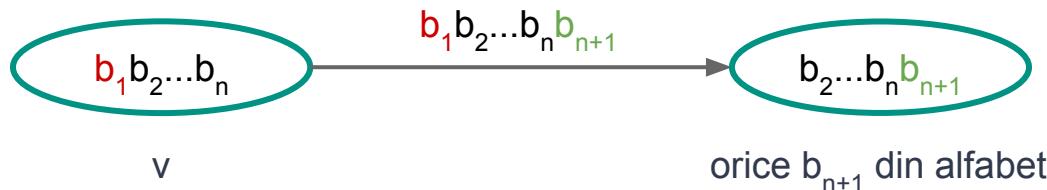


# Grafuri de Bruijn

$B_n$  este eulerian

$$d^+(v) = |\{0, 1\}| = 2 \quad (\text{mai general} = |\{0, 1, \dots, p\}|)$$

$$d^-(v) = d^+(v)$$



# Grafuri de Bruijn

- Prima cifră din etichetele arcelor unui circuit eulerian în  $B_{n-1}$  - soluție pentru problema lui Posthumus

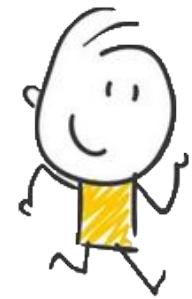
$$\Rightarrow f(n) = 2^n$$

- Observație

Circuit eulerian în  $B_{n-1} \leftrightarrow$  circuit hamiltonian în  $B_n$

# Grafuri de Bruijn

Aplicație - genetică (*Genome Assembly*)



# Grafuri planare

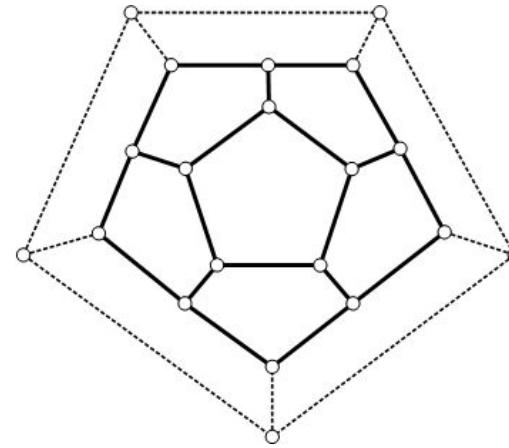
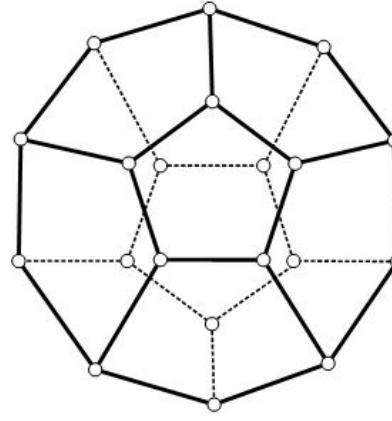
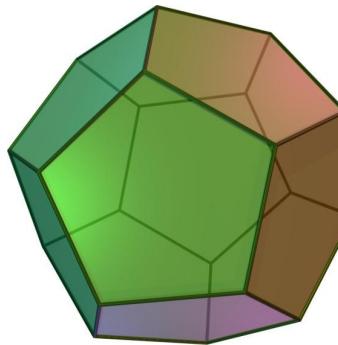


# Graf planar

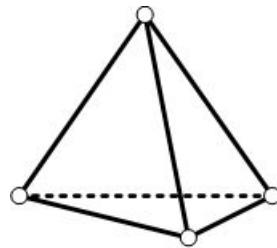


**Amintiri din primul curs**

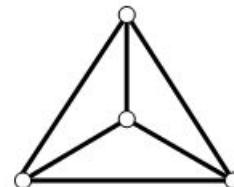
# Dodecaedrul



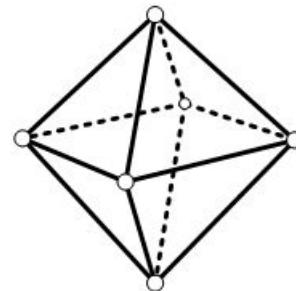
# Corpuri platonice - grafuri planare



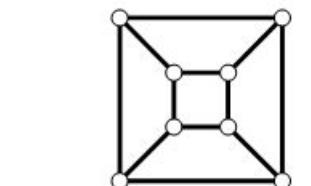
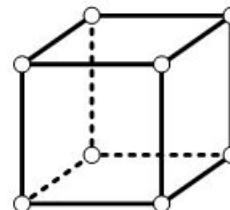
Tetraedru



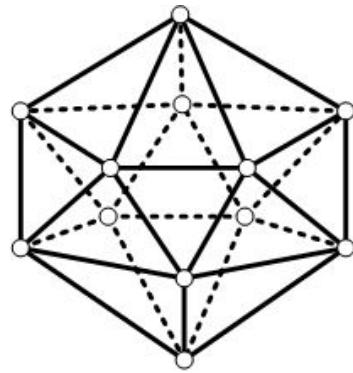
Octaedru



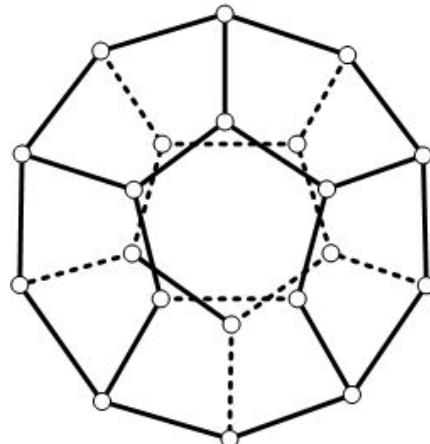
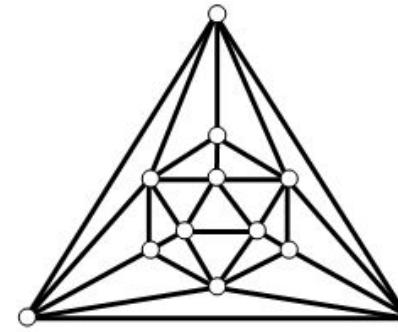
Cub



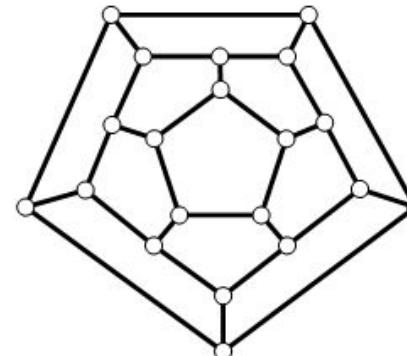
# Corpuri platonice - grafuri planare



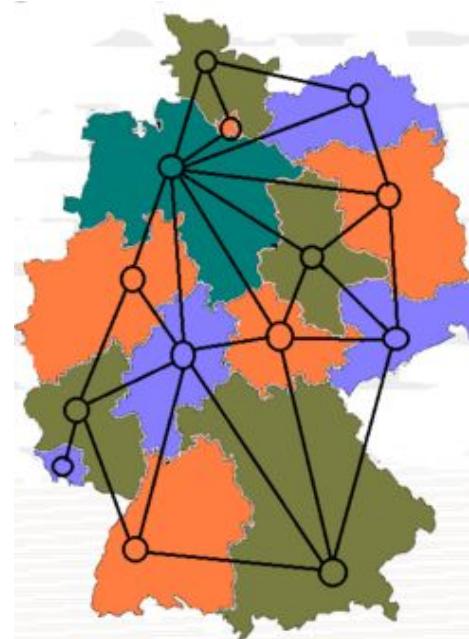
Icosaedru



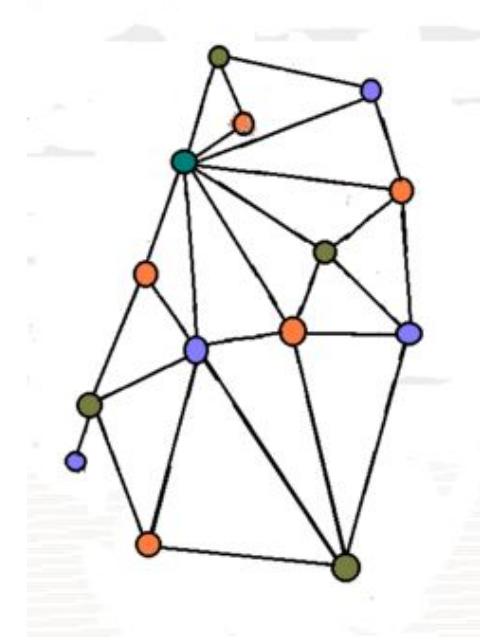
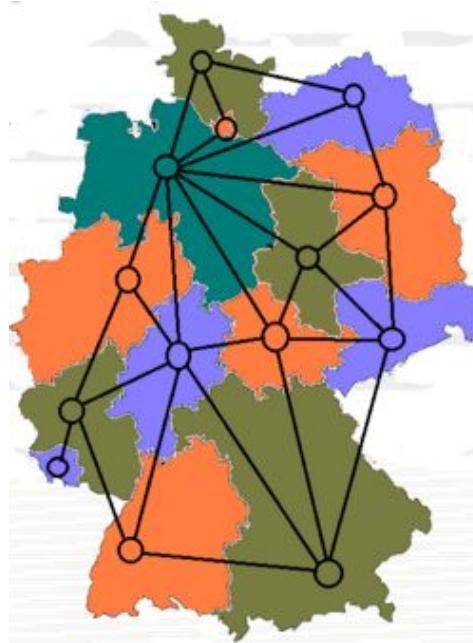
Dodecaedru



# Colorarea hărților



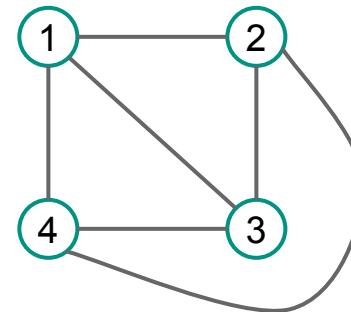
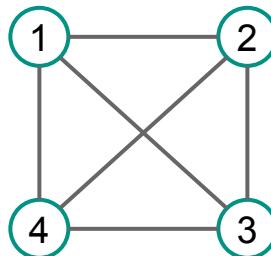
# Colorarea hărților



# Graf planar

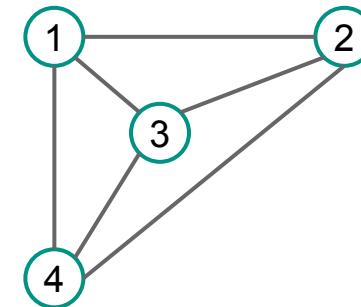
$G = (V, E)$  graf neorientat s.n. planar  $\Leftrightarrow$  admite o reprezentare în plan, a.î. **muchiilor** le corespund segmente de curbe continue care **nu se intersectează** în interior unele pe altele

O astfel de reprezentare s.n. hartă a lui G



$G \sim K_4$

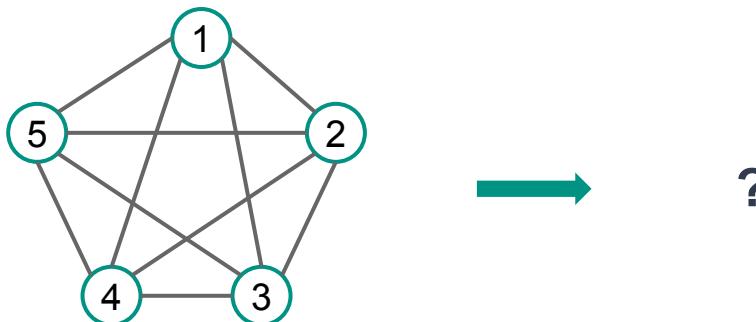
hărți



# Graf planar

$G = (V, E)$  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



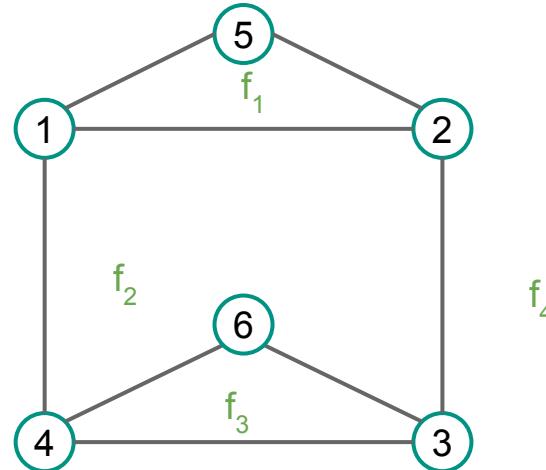
$$G \sim K_5$$

# Graf planar

Fie  $G = (V, E)$  graf planar,  $M$  o hartă a sa.

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

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

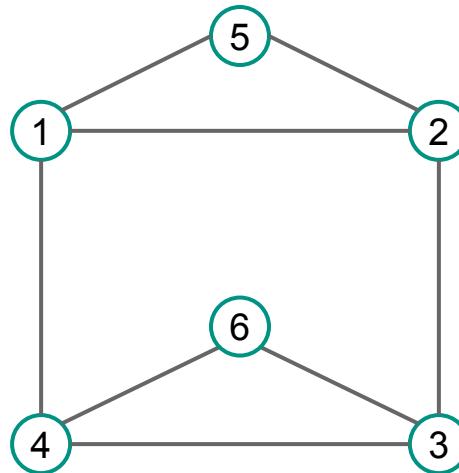


# Graf planar

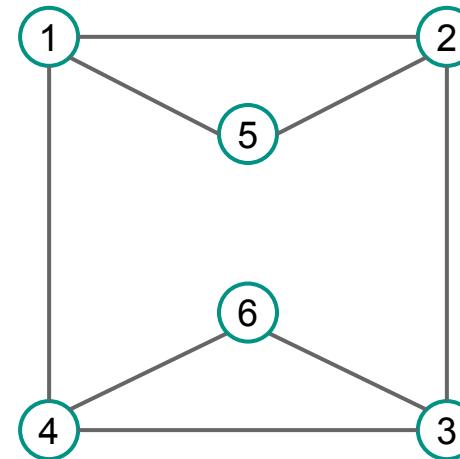
$M = (V, E, F)$  hartă

Pentru o față  $f \in F$  definim

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

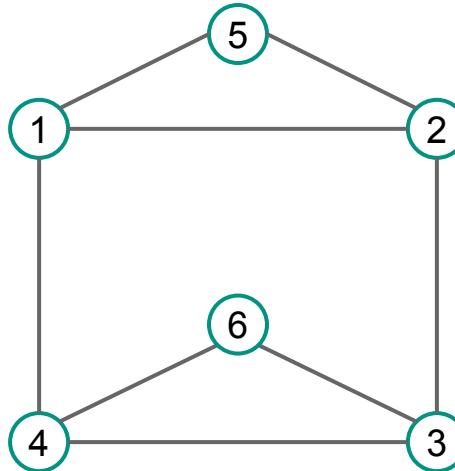


~

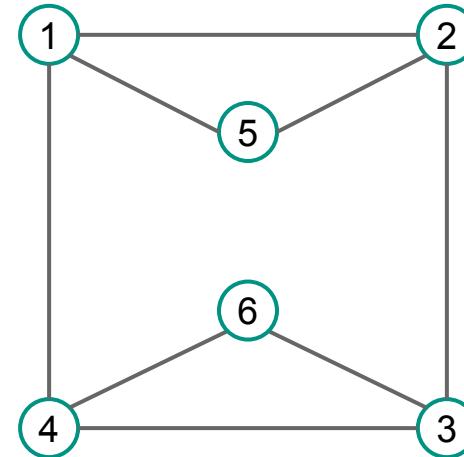


# Graf planar

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



~



**Poate să difere și numărul de fețe (între 2 hărți ale aceluiași graf)?**

# Graf planar

$M = (V, E, F)$  hartă

Avem:

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

# Graf planar

## 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

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

# Graf planar

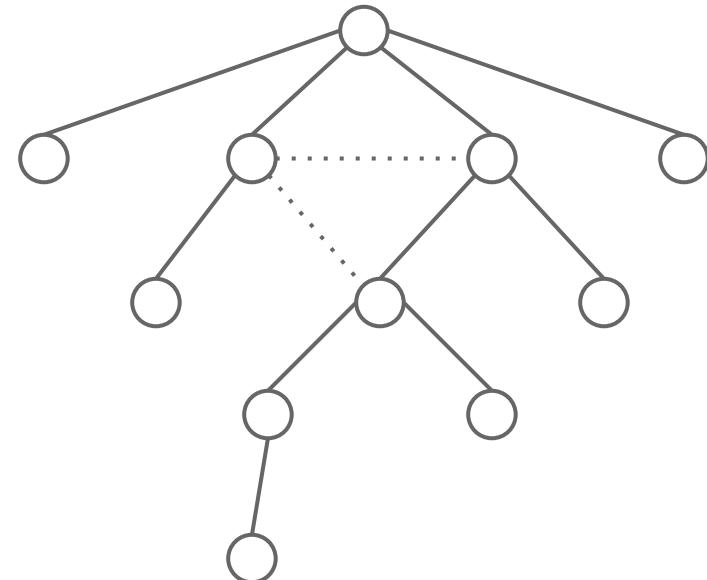
## Teorema poliedrală a lui EULER

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

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

## Inducție

- Arboare parțial + muchii



# Graf planar

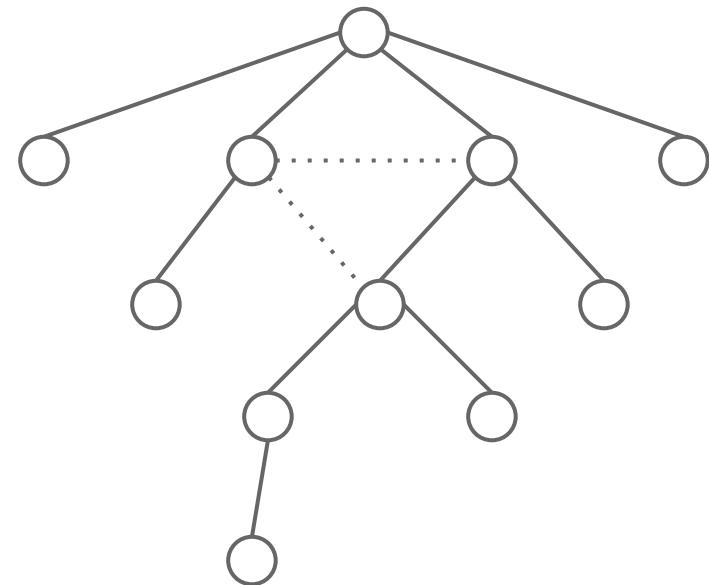
## Teorema poliedrală a lui EULER

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

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

### Inducție

- Arbore parțial + muchii**
- Într-un arbore**
  - $|E| = |V| - 1$
  - $|F| = 1$
  - $|V| - |E| + F = |V| - (|V| - 1) + (1) = 1 + 1 = 2 \rightarrow \text{OK}$



# Graf planar

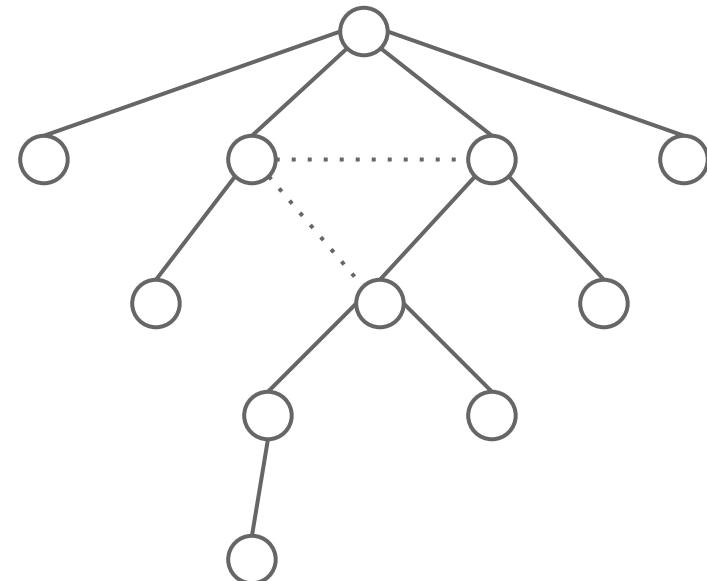
## Teorema poliedrală a lui EULER

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

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

### Inducție

- Pentru arbore e OK
- Când adăugăm o muchie ...
  - $V$  rămâne constant
  - $E$  crește cu 1
  - $F$  crește cu 1
  - Relația rămâne validă



# Graf planar

## 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.

# Graf planar

## Proprietăți

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

Atunci:

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

# Graf planar

## Proprietăți

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

Atunci:

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

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

$$d_M(f) \geq 3$$

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

# Graf planar

## Proprietăți

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

Atunci:

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

Demonstrație

$$\sum_{f \in F} d_M(f) = 2|E|$$
$$|V| - |E| + |2E|/3 \leq 2$$
$$d_M(f) \geq 3$$
$$3|V| - |E| \leq 6 \Rightarrow a)$$
$$|V| - |E| + |F| = 2$$

# Graf planar

## Proprietăți

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

Atunci:

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

## Consecință

$K_5$  nu este graf planar (exercițiu)

# Graf planar

## Proprietăți (temă)

Fie  $G = (V, E)$  un graf planar convex 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

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

## **Algoritm de colorare a unui graf planar cu 6 culori**

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

## **Algoritm de colorare a unui graf planar cu 6 culori**

colorare( $G$ )

dacă  $|V(G)| \leq 6$ , atunci colorează vîrfurile cu culori distincte din  $\{1, \dots, 6\}$

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

### **Algoritm de colorare a unui graf planar cu 6 culori**

colorare( $G$ )

**dacă**  $|V(G)| \leq 6$ , **atunci** colorează vîrfurile cu culori distincte din  $\{1, \dots, 6\}$

**altfel**

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

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

### **Algoritm de colorare a unui graf planar cu 6 culori**

colorare( $G$ )

**dacă**  $|V(G)| \leq 6$ , **atunci** colorează vîrfurile cu culori distincte din  $\{1, \dots, 6\}$

**altfel**

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

colorare( $G-x$ )

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

### **Algoritm de colorare a unui graf planar cu 6 culori**

colorare( $G$ )

**dacă**  $|V(G)| \leq 6$ , **atunci** colorează vârfurile 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

# Graf planar

## Teorema celor 6 culori

Orice graf planar conex este 6-colorabil.

### Algoritm de colorare a unui graf planar cu 6 culori

colorare( $G$ )

**dacă**  $|V(G)| \leq 6$ , **atunci** colorează vârfurile 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

**Sugestie de implementare** - determinarea iterativă a ordinii în care sunt colorate vârfurile (similar parcursere BF, sortare topologică)

# Graf planar

## Teorema celor 5 culori

Orice graf planar conex este 5-colorabil.

**Suplimentar - Temă (+ algoritm de 5-colorare)**

# Graf planar

## Teorema celor 4 culori

Orice graf planar conex este 4-colorabil.

Peste nivelul cursului.



# Distanțe de editare



# Distanțe de editare

Se dau două șiruri de caractere  $s_1$  și  $s_2$ .

**Distanță de editare** - numărul minim de operații (inserări, modificări, ștergeri de caractere etc) necesar pentru a transforma șirul  $s_1$  în șirul  $s_2$ .

**Distanță de editare Levenshtein** - sunt permise operații de inserare, modificare și stergere.

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



# Distanțe de editare

- La fiecare nepotrivire a unui caracter cu cel din destinație, avem 3 operații posibile.
- Dacă analizăm, pe rând, fiecare variantă  $\Rightarrow$  backtracking  $\Rightarrow$  **ineficient**

**Soluție - Programare dinamică**

# Distanțe de editare

## Principiu de optimalitate:

Considerăm o transformare cu număr minim de operații:

$x_1 x_2 \dots x_n$

$y_1 y_2 \dots y_m$

Evidențiem ultima operatie.

# Distanțe de editare

$x_1 x_2 \dots x_n \Rightarrow y_1 y_2 \dots y_m$

**Evidențiem ultima operație**

- $x_n = y_m$  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$   
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) + \text{păstrăm } x_n$

# Distanțe de editare

$$x_1 x_2 \dots x_n \Rightarrow y_1 y_2 \dots y_m$$

## Evidențiem ultima operație

- $x_n = y_m$**  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$   
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) + \text{păstrăm } x_n$
- $x_n$  a fost șters** - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_m$  (după care se șterge  $x_n$ )  
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_m) + \text{ștergem } x_n$

# Distanțe de editare

$x_1 x_2 \dots x_n \Rightarrow y_1 y_2 \dots y_m$

## Evidențiem ultima operație

- $x_n = y_m$**  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$   
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) +$  **păstrăm  $x_n$**
- $x_n$  a fost șters** - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_m$  (după care se șterge  $x_n$ )  
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_m) +$  **ștergem  $x_n$**
- $x_n$  a fost modificat în  $y_m$**  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$  (după care se modifică  $x_n$  în  $y_m$ )  
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) +$  **modificăm  $x_n \leftrightarrow y_m$**

# Distanțe de editare

$x_1 x_2 \dots x_n \Rightarrow y_1 y_2 \dots y_m$

## Evidențiem ultima operație

- $x_n = y_m$**  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$   
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) + \text{păstrăm } x_n$
- $x_n$  a fost șters** - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_m$  (după care se șterge  $x_n$ )  
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_m) + \text{ștergem } x_n$
- $x_n$  a fost modificat în  $y_m$**  - problema se reduce la a transforma  $x_1 \dots x_{n-1}$  în  $y_1 \dots y_{m-1}$  (după care se modifică  $x_n$  în  $y_m$ )  
 $(x_1 \dots x_{n-1} \Rightarrow y_1 \dots y_{m-1}) + \text{modificăm } x_n \leftrightarrow y_m$
- a fost inserat  $y_m$**  - problema se reduce la a transforma  $x_1 \dots x_n$  în  $y_1 \dots y_{m-1}$  (după care se inserează  $y_m$ )  
 $(x_1 \dots x_n \Rightarrow y_1 \dots y_{m-1}) + \text{inserăm } y_m$

# Distanțe de editare

Avem, deci, 4 cazuri. Problema se reduce la a transforma un prefix  $x_1 \dots x_i$  al primului cuvânt într-un prefix  $y_1 \dots y_j$  al celui de-al doilea cuvânt (subprobleme PD).

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

# 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:

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

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

↑                      ↑                      ↑  
ștergem  $x_i$          $x_i \leftrightarrow y_j$         inserăm  $y_j$

# Distanțe de editare

**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 din cuvinte este vid):
  - $x_1 \dots x_{i-1} \Rightarrow$  secvență vidă      prin  $i$  ștergeri 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:**  $i = 0, \dots, n; j = 0, \dots, m$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

i = 0: c[0][j] = j  
j = 0: c[i][0] = i

# Distanțe de editare

**Exemplu:**  $s_1 = \text{care} \Rightarrow s_2 = \text{antet}$

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

$i = 1, j = 1:$

$s1[i] = c, s2[j] = a - \text{sunt diferite} \Rightarrow$

# Distanțe de editare

**Exemplu:**  $s_1 = \text{care} \Rightarrow s_2 = \text{antet}$

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

$i = 1, j = 1:$

$s1[i] = c, s2[j] = a - \text{sunt diferite} \Rightarrow$

$c[i][j] = 1 + \min(c[i-1][j], c[i][j-1], c[i-1][j-1]) = 1 + 0 = 1$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

$\leftarrow c \text{ nu aparține cuvântului antet}$

$$c[i][j] = 1 + \min(c[i-1][j], c[i][j-1], c[i-1][j-1])$$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c: ← c nu aparține cuvântului antet

i = 2, j = 1:

s1[i] = a, s2[j] = a - sunt egale  $\Rightarrow$   
c[i][j] = c[i-1][j-1] = 1

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c:



$$c[i][j] = 1 + \min(c[i-1][j], c[i][j-1], c[i-1][j-1])$$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

i = 3, j = 1:

s1[i] = r, s2[j] = a - sunt diferite  $\Rightarrow$

c[i][j] = 1 + min(c[i-1][j], c[i][j-1], c[i-1][j-1]) = 1 + 1 = 2

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c: ← r nu aparține cuvântului **antet**

$$c[i][j] = 1 + \min(c[i-1][j], c[i][j-1], c[i-1][j-1])$$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

i = 4, j = 4:

s1[i] = e, s2[j] = e - sunt egale  $\Rightarrow$   
c[i][j] = c[i-1][j-1] = 3

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

Soluția:  $c[4][5] = 4$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

Soluția:  $c[4][5] = 4$   
Cum determinăm operațiile?

# Distanțe de editare

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	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	3	4

Soluția:  $c[4][5] = 4$   
Cum determinăm operațiile?

Mergând succesiv înapoi de la  $(4, 5)$  în celula pentru care s-a obținut egalitatea în relația de recurență:

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

↑                      ↑                      ↑  
ștergem  $x_i$          $x_i \leftrightarrow y_j$         inserăm  $y_j$

# Distanțe de editare

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	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	

Soluția:  $c[4][5] = 4$

$s1[4] \neq s2[5] \Rightarrow$   
 $c[4][5] = 1 + \min(c[4][4], c[3][5], c[3][4])$   
 $= 1 + c[4][4] \Rightarrow$   
(4, 5) s-a obținut din (4, 4) prin inserarea caracterului  $s2[5] = t$

# Distanțe de editare

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	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	

inserăm t

$$s1[4] = s2[4] \Rightarrow$$

$c[4][4] = c[3][3]$  și nu s-a făcut nicio operație (păstrăm e)

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c:

păstrăm e

inserăm t

$s1[3] \neq s2[3] \Rightarrow$   
 $c[3][3] = 1 + \min(c[2][3], c[3][2], c[2][2])$   
 $= 1 + c[3][2] = 1 + c[2][2] \Rightarrow$

!! soluția nu este unică  $\Rightarrow$  alegem una:  $c[3][3] = 1 + c[3][2] \Rightarrow (3,3)$  s-a obținut din  $(3,2)$  prin inserarea caracterului  $s2[3] = t$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

inserăm t, păstrăm e

inserăm t

$s1[3] \neq s2[2] \Rightarrow$

$$\begin{aligned} c[3][2] &= 1 + \min(c[2][2], c[3][1], c[2][1]) \\ &= 1 + c[2][1] \Rightarrow \end{aligned}$$

(3,2) s-a obținut din (2,1) prin modificarea  $s1[3] = r \leftrightarrow s2[2] = n$

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

modificăm r  $\leftrightarrow$  n

inserăm t, păstrăm e

inserăm t

$s1[2] = s2[1] \Rightarrow$

$c[2][1] = c[1][0]$  și nu s-a făcut nicio operație (păstrăm a)

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c:

păstrăm a

modificăm r  $\leftrightarrow$  n

inserăm t, păstrăm e

inserăm t

i = 1, j = 0:

Deoarece j = 0, c[1][0] = 1 + c[0][0] corespunzător unei ștergeri  $\rightarrow$  a caracterului s1[i] = c

# Distanțe de editare

Exemplu: care  $\Rightarrow$  antet

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

c:

ştergem c

păstrăm a

modificăm r  $\leftrightarrow$  n

inserăm t, păstrăm e

inserăm t

# Distanțe de editare

**Complexitate?**

# Distanțe de editare

Complexitate?       $O(nm)$

