

# Algoritmi Fundamentali

Ruxandra Marinescu & Marius Dumitran

[marius.dumitran@unibuc.ro](mailto:marius.dumitran@unibuc.ro)

# Programa



# Programa (Grafuri + Stringuri)

- Parcurgeri**
- Secvențe de grade**
- Conecțivitate**
- Arbore, arbori parțiali de cost minim**
- Drumuri minime**
- Fluxuri în rețele de transport**
- Cuplaje**
- Grafuri planare**
- Kmp ???**
- Dinamici pe șiruri de caractere**

# Obiectiv general

- Dezvoltarea gândirii algoritmice prin familiarizarea cu algoritmi fundamentali de grafuri și siruri de caractere și aplicații ale acestora și cu noi tipuri de abordare ale problemelor dificile de algoritmică

# Obiective specifice

- Prezentarea principalelor noțiuni și rezultate + utilitatea acestora**
- Modelarea problemelor cu ajutorul grafurilor și elaborarea de algoritmi de grafuri pentru rezolvarea acestora**
- Justificarea corectitudinii algoritmilor propuși + estimarea eficienței acestora**
- Implementarea eficientă a algoritmilor**

# Motivații

- Este un domeniu fundamental
- Apar în numeroase aplicații în diverse domenii
- Sunt folosiți în procesarea imaginilor, bioinformatică, rețele, baze de date, proiectare, strategii
- Există instrumente pentru a dezvolta algoritmi eficienți
- Apar des la interviuri
- Vor apărea în cursuri viitoare

# Structura cursului

## □ Curs

- 2 ore pe săptămână
- finalizat cu examen scris

## □ Laborator

- 2 ore la două săptămâni
- limbaje de programare: C / C++ / Python

## □ Seminar

- 2 ore la două săptămâni
- discuții probleme curs / laborator, calcul complexități, exerciții

# Evaluare



# Evaluare

- Laborator 50%
  - Nota **minim 5**
- Seminar 10% + 1p bonus
  - Prezență, activitate, teme
- Examen 40%
  - Nota **minim 5**
  - Scris
  - Cam ca anul trecut :)
- Donam contam
  - O donare de sange ne da o prezenta in plus la seminar

# Evaluare

## □ Laborator 50%

- Nota **minim 5**
- teme obligatorii – punctaj teme\_oblig  $\leq 4$
- teme suplimentare (mai dificile) – punctaj tema\_suplim  $\leq 6$
- test de laborator – punctaj test  $\leq 6$
- Punctajul de la temele suplimentare poate înlocui punctajul de la testul de laborator sau poate contribui la creșterea acestuia . Astfel, nota finală la laborator va fi
  - $\text{tema\_oblig} = \min(6, \max(\text{tema\_suplim}/3, \text{test}))$
- Penalizare 50% pentru depasirea deadlineului ... pana cand poti trimite ?
- Prezentat fizic la labul cu deadline sau urmatorul....

# Bibliografie



# Bibliografie – curs

- Douglas B. West, **Introduction to Graph Theory**, Prentice Hall 1996, 2001
- J.A. Bondy, U. S. R Murty, **Graph theory with applications**, The Macmillan Press 1976 / Springer 2008
- Dragoș-Radu Popescu, **Combinatorică și teoria grafurilor**, Editura Societatea de Științe Matematice din România, București, 2005

# Bibliografie – curs + seminar

- Dragoș-Radu Popescu, R. Marinescu-Ghemeci, **Combinatorică și teoria grafurilor prin exerciții și probleme**, Editura Matrixrom, 2014
- Ioan Tomescu, **Probleme de combinatorică și teoria grafurilor/ Problems in Combinatorics and Graph Theory**

# Bibliografie – algoritmi + laborator

- Jon Kleinberg, Éva Tardos, **Algorithm Design**, Addison-Wesley, 2005  
<http://www.cs.princeton.edu/~wayne/kleinberg-tardos/>
- T.H. Cormen, C.E. Leiserson, R.R. Rivest, **Introducere în algoritmi**, MIT Press, trad. Computer Libris Agora
- H. Georgescu, **Tehnici de programare**, Editura Universității din București, 2005

# Bibliografie

- coursera.org
- infoarena.ro
- csacademy.com

# Resurse

- O să fie pe [teams](#).
- Consultații
  - [marius.dumitran@unibuc.ro](mailto:marius.dumitran@unibuc.ro)
  - [marius.dumitran@gmail.com](mailto:marius.dumitran@gmail.com)
  - Sau vorbiți cu profesorii de laborator/seminar

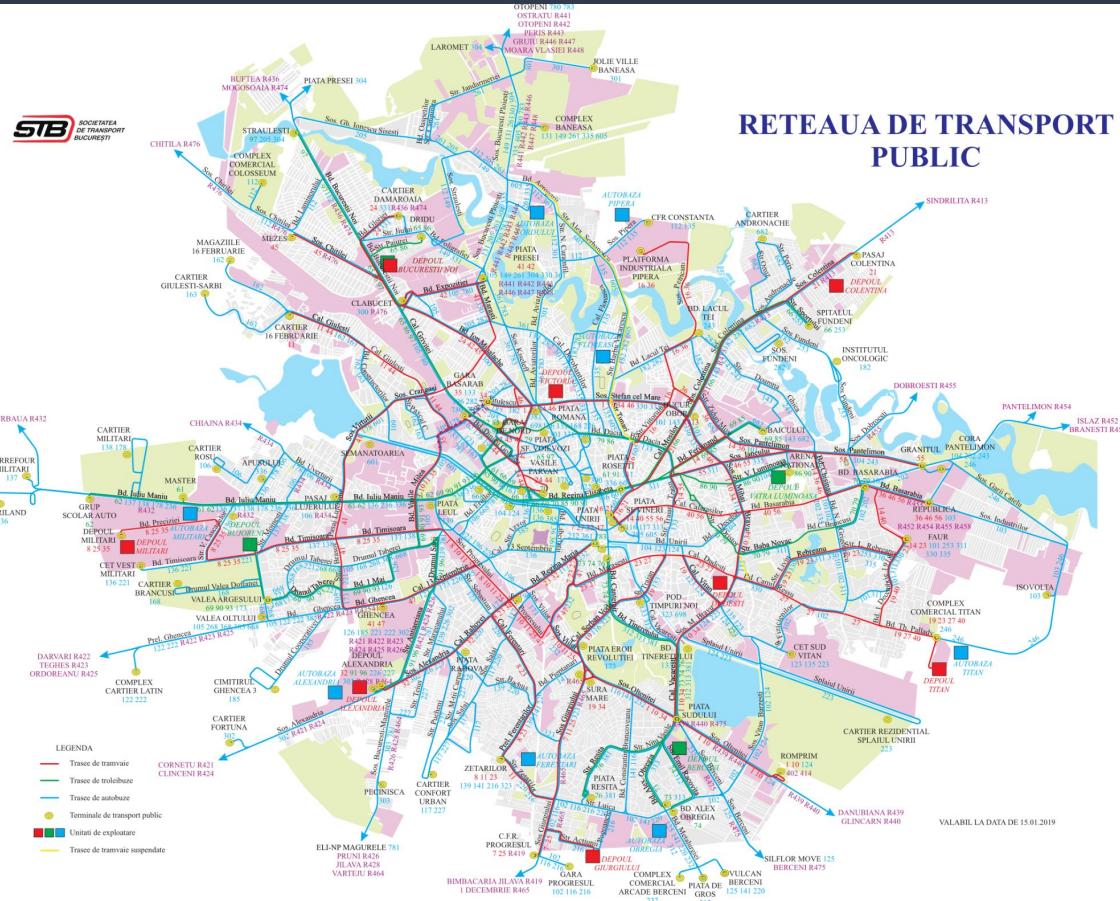


0:30

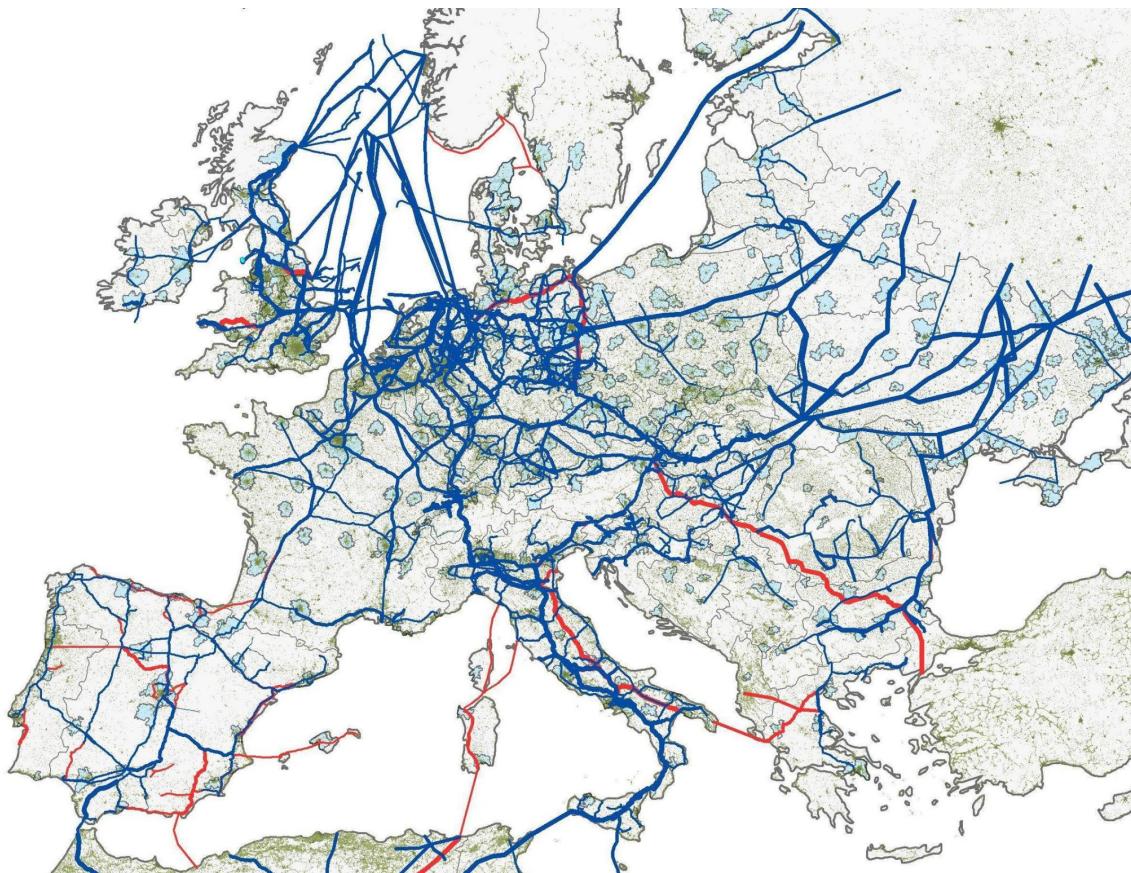
# Aplicații ale grafurilor



# Rețele de transport în comun, trasee turistice, GPS



# Rețele de transport în comun, trasee turistice, GPS



**Rețeaua de distribuție a  
gazelor naturale în  
Europa**

[https://britishbusinessenergy.co.uk/  
blog/europe-natural-gas-network/](https://britishbusinessenergy.co.uk/blog/europe-natural-gas-network/)

# Analiza rețelelor

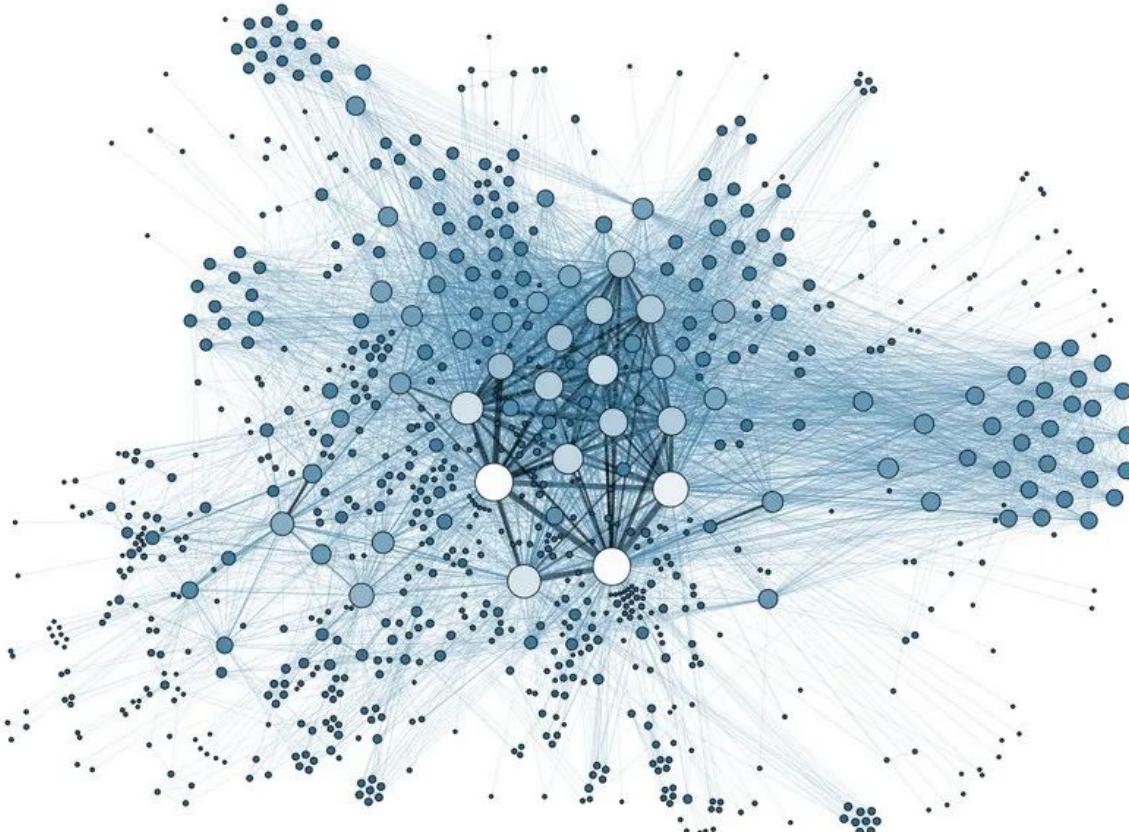
## □ Interacțiuni

- Rețele sociale
- Rețele biologice
- Retele de citări, de știri, de spionaj etc



<https://github.com/XinyueTan/Social-Network-Analysis->

# Software pentru vizualizarea și analiza rețelelor

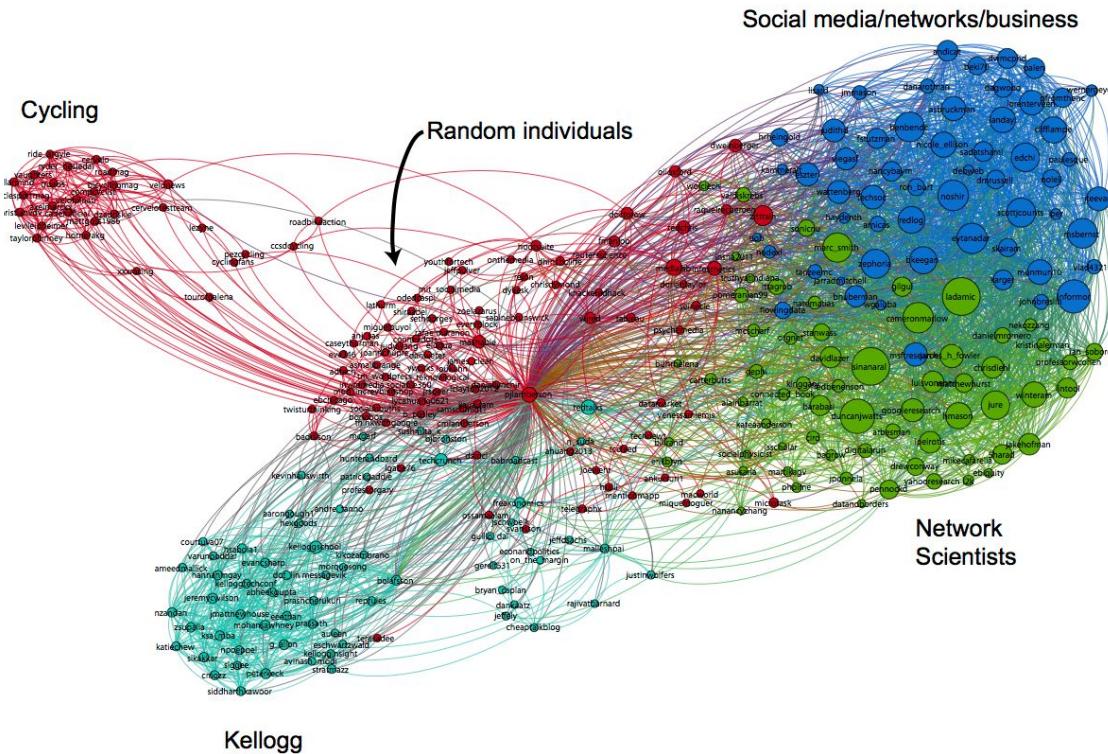


Software:

- <https://archive.codeplex.com/?p=nodelx>
- <https://gephi.org/>

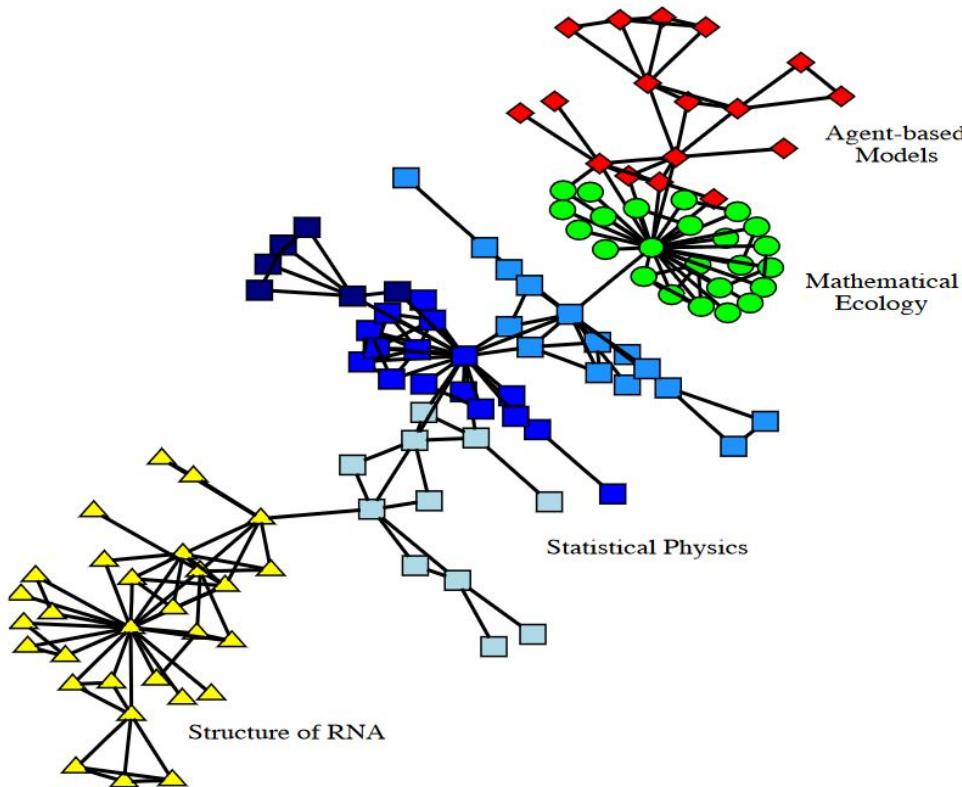
<https://www.interaction-design.org/literature/article/how-to-display-complex-network-data-with-information-visualization>

# Rețele sociale



<http://social-dynamics.org/twitter-network-data/>

# Rețele sociale



**Rețea de colaborări între cercetătorii de la Institutul Santa Fe**

**Clusterele – corespund departamentelor de cercetare**

Santo Fortunato, **Community detection in graphs**, Physics Reports 486 (2010) 75–174

<https://arxiv.org/pdf/0906.0612.pdf>

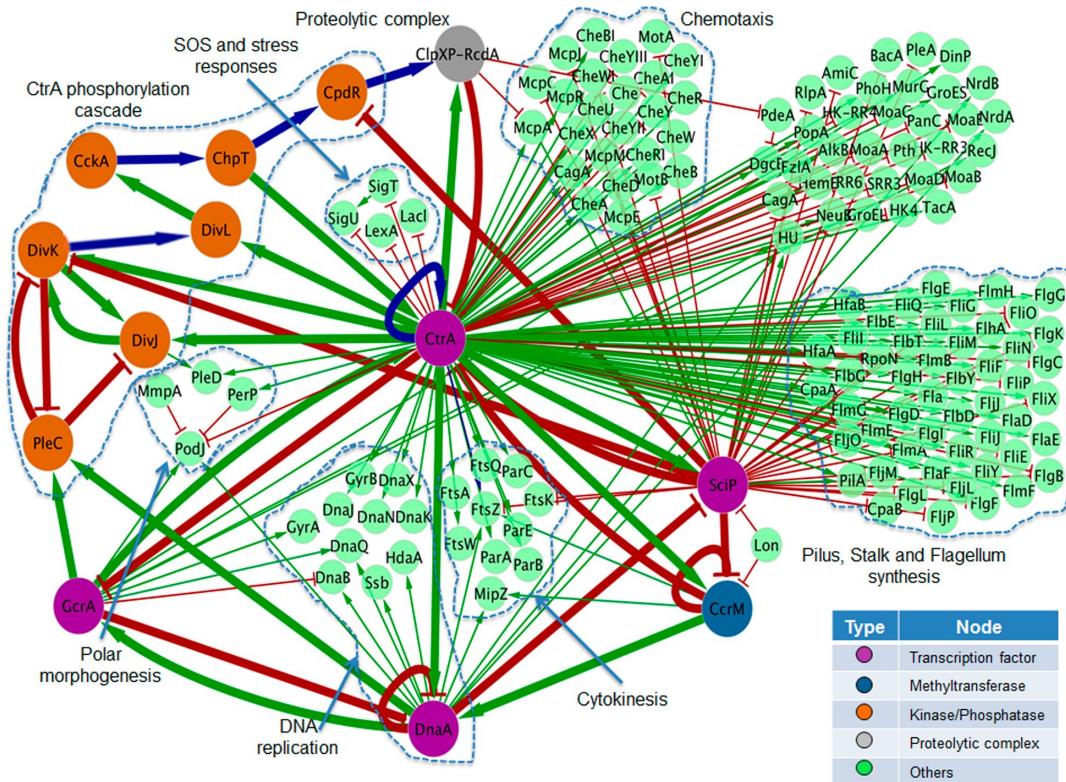
# Rețele

- **Rețele de știri - detectarea de știri false**
  - <https://neo4j.com/blog/machine-learning-graphs-fake-news-epidemic-part-2/>
  - <https://cambridge-intelligence.com/detecting-fake-news/>
- **Rețele de teroriști**
  - Palantir :)

# Bioinformatică

- Grafuri de interacțiuni între gene/proteine
  - [https://domaingraph.bioinf.mpi-inf.mpg.de/docu/dg\\_network.php](https://domaingraph.bioinf.mpi-inf.mpg.de/docu/dg_network.php)
- Clustering
- Grafuri de intersecție, grafuri De Bruijn
- Arbori filogenetici

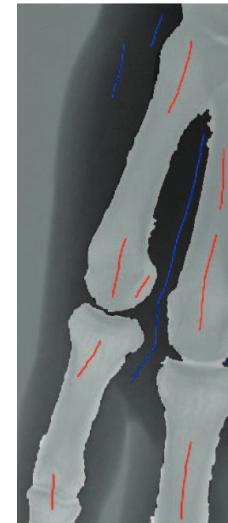
# Bioinformatică



[https://openi.nlm.nih.gov/detailedresult?img=PMC4219702\\_pone.0111116.q002&req=4](https://openi.nlm.nih.gov/detailedresult?img=PMC4219702_pone.0111116.q002&req=4)

# Image segmentation

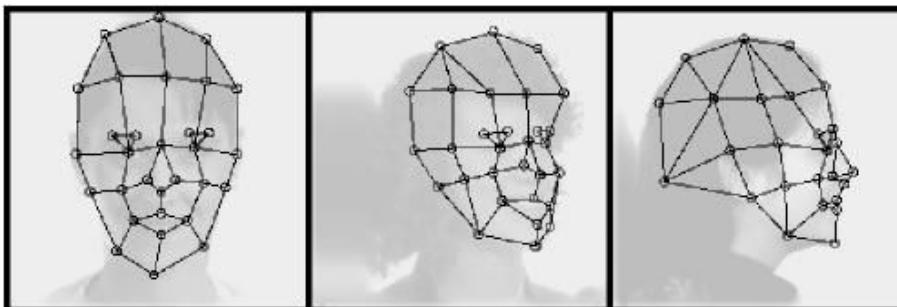
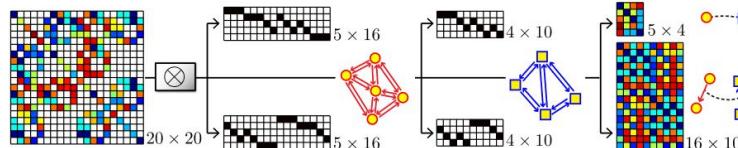
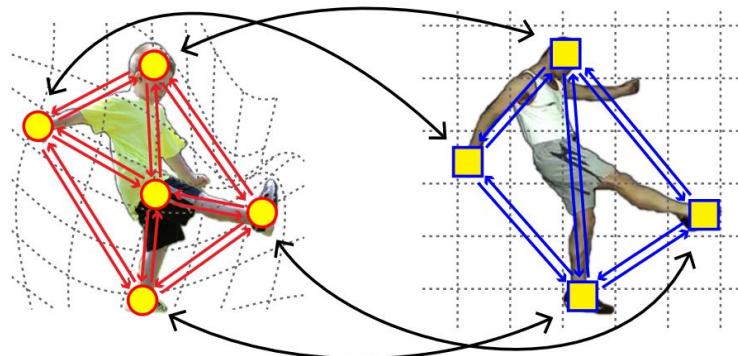
- tăietură minimă - fluxuri în rețele de transport
- medicină



**Spatially Varying Color Distributions for Interactive Multi-Label Segmentation** (C. Nieuwenhuis, D. Cremers), In IEEE Transactions on Pattern Analysis and Machine Intelligence, volume 35, 2013

[https://vision.in.tum.de/\\_media/spezial/bib/nieuwenhuis-cremers-pami12\\_2.pdf](https://vision.in.tum.de/_media/spezial/bib/nieuwenhuis-cremers-pami12_2.pdf)

# Computer vision



F. Zhou and F. De la Torre, **Deformable Graph Matching**, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2013

[http://www.f-zhou.com/gm/2013\\_CVPR\\_DGM.pdf](http://www.f-zhou.com/gm/2013_CVPR_DGM.pdf)

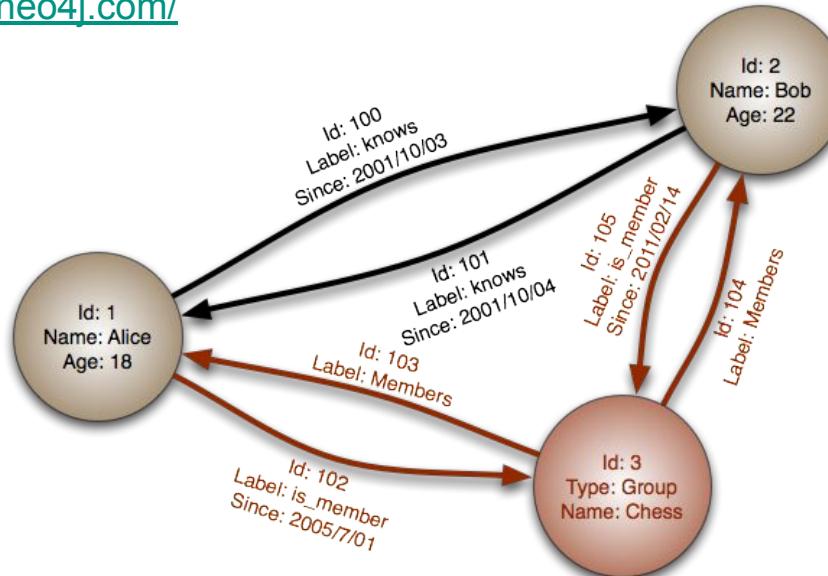
Face recognition by elastic bunch graph matching

<https://www.ini.rub.de/PEOPLE/wiskott/Projects/EGMFaceRecognition.html>

# Baze de date

## □ Graph database

- Neo4J: <https://neo4j.com/>



[https://en.wikipedia.org/wiki/Graph\\_database](https://en.wikipedia.org/wiki/Graph_database)

# Probleme de planificare, orar

**Exemplu:** Care este numărul minim de săli necesare 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)**

# Probleme de planificare, orar

**Exemplu:** Care este numărul minim de săli necesare pentru programarea într-o zi a n conferințe cu intervale de desfășurare date?

Conf. 1: interval **(1,4)**

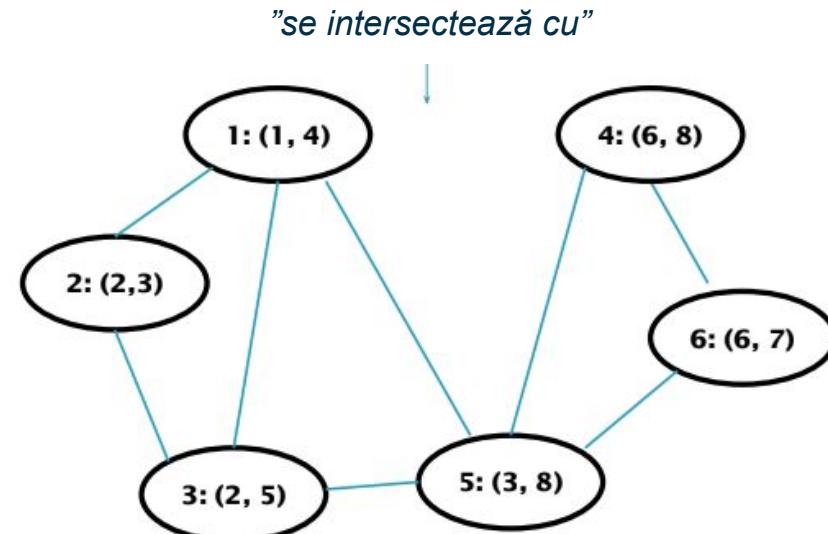
Conf. 2: interval **(2,3)**

Conf. 3: interval **(2,5)**

Conf. 4: interval **(6,8)**

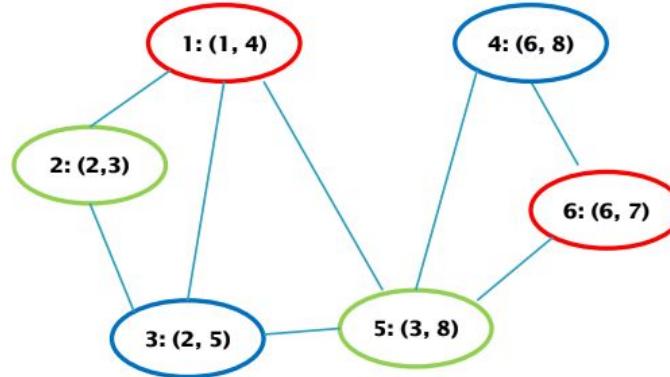
Conf. 5: interval **(3,8)**

Conf. 6: interval **(6,7)**



# Probleme de planificare, orar

Graful intersecției intervalor este **3-colorabil**



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

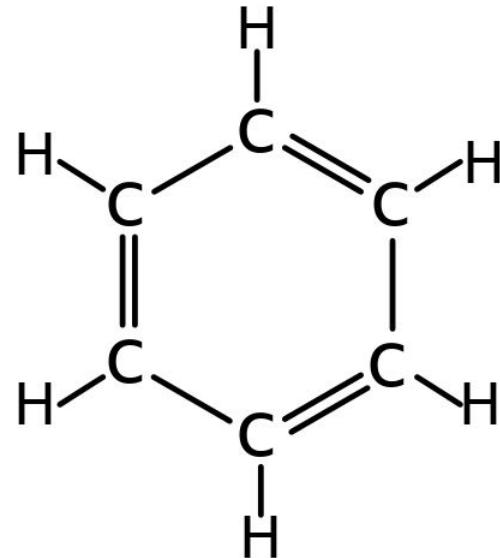
**Sala 1:** (1, 4), (6, 7)

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

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

# Chimie

- Graf ← "notație grafică" din chimie
  - J. Silvester, 1878



# Matematică

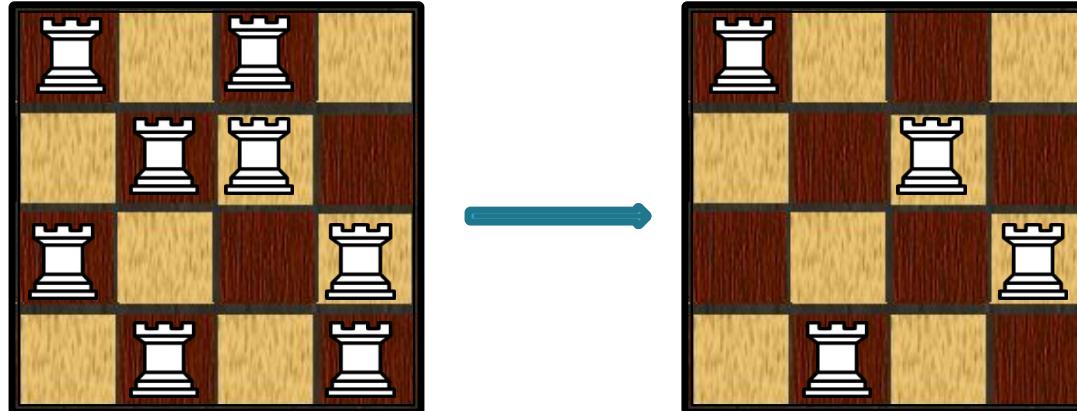
- Demonstrarea unor rezultate matematice
  - Matrice → graf
  - Diagonală / Matrice de permutări - cuplaj

# Probleme

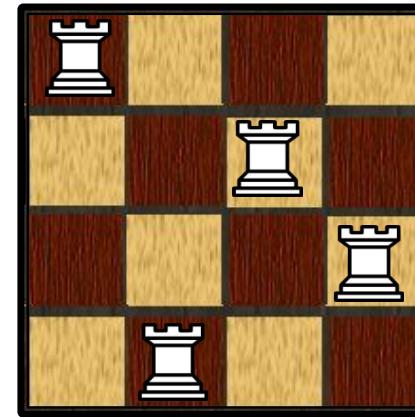
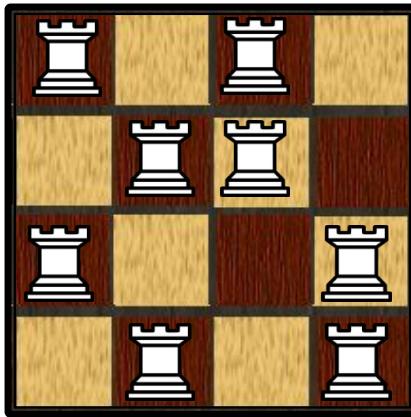


# Probleme

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**. Găsiți numărul maxim de ture care nu se atacă două câte două - **Cuplaje**



# Probleme



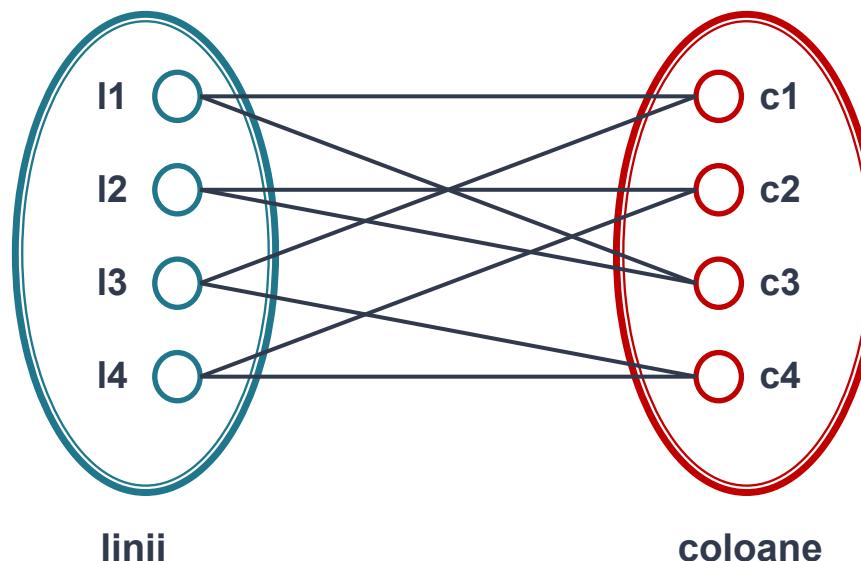
$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$



$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

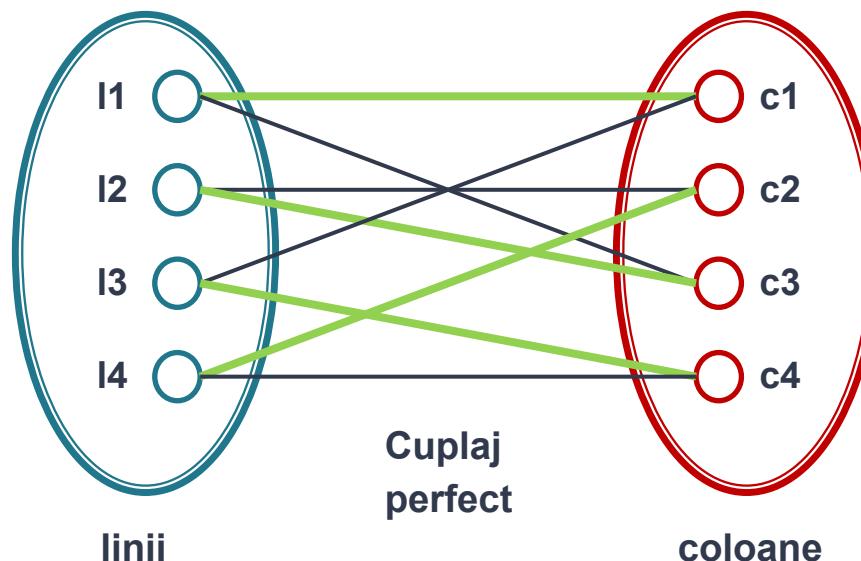
# Probleme

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \xrightarrow{\hspace{1cm}} \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$



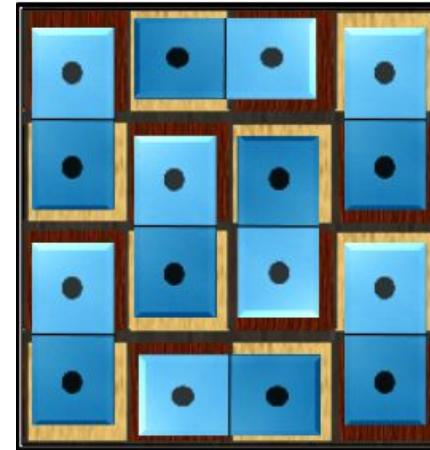
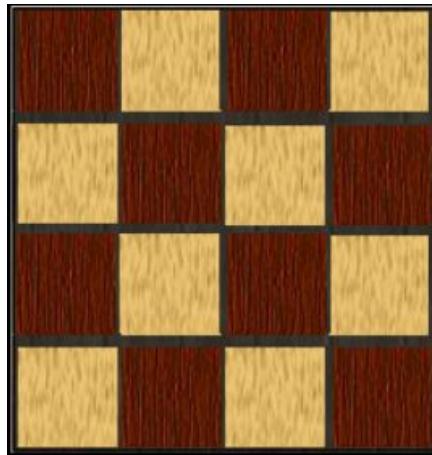
# Probleme

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \xrightarrow{\hspace{1cm}} \quad P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$



# Probleme

Acoperirea unei table cu piese de domino

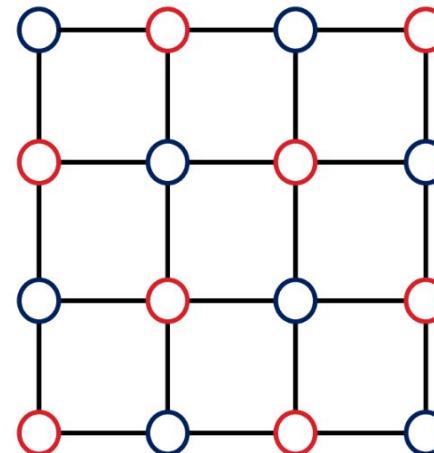
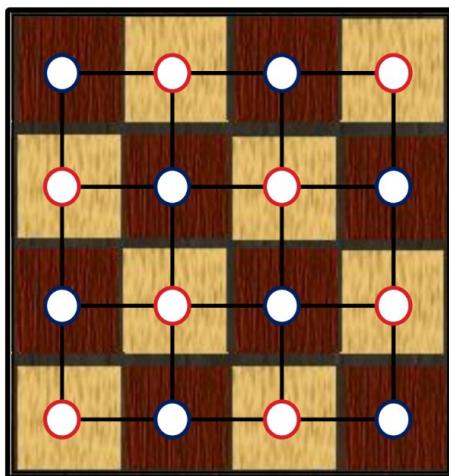


# Probleme

Tabla

⇒

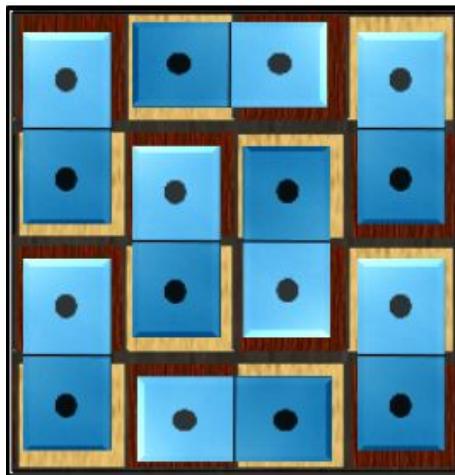
Graful grid



# Probleme

Tabla

Acoperire

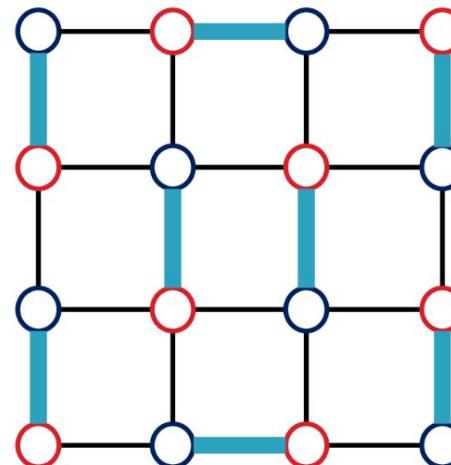


⇒

Graful grid

⇒

Cuplaj perfect



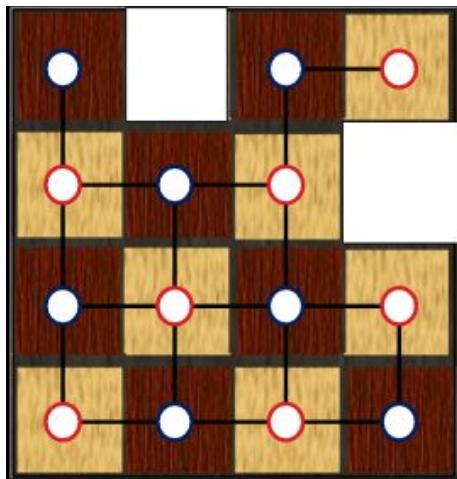
# Probleme

## 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?

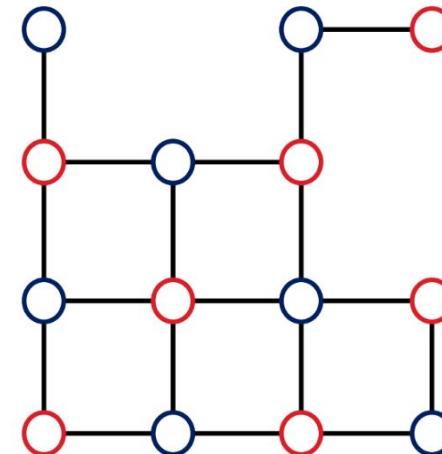
# Probleme

Tabla



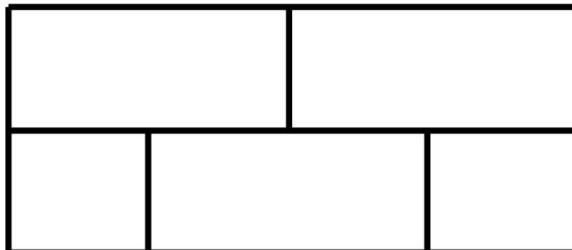
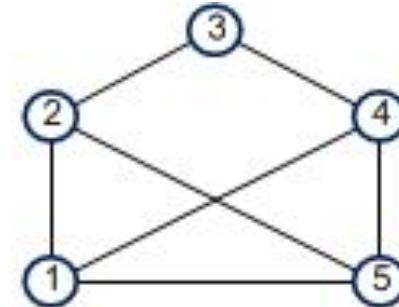
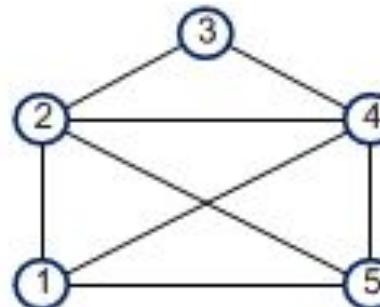
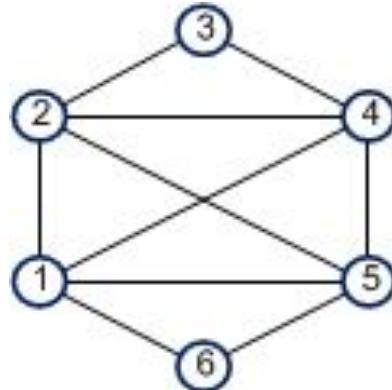
⇒

Graful grid



# Probleme

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?



Există o linie continuă, neînchisă, care să intersecteze, în interior, fiecare segment, o singură dată?

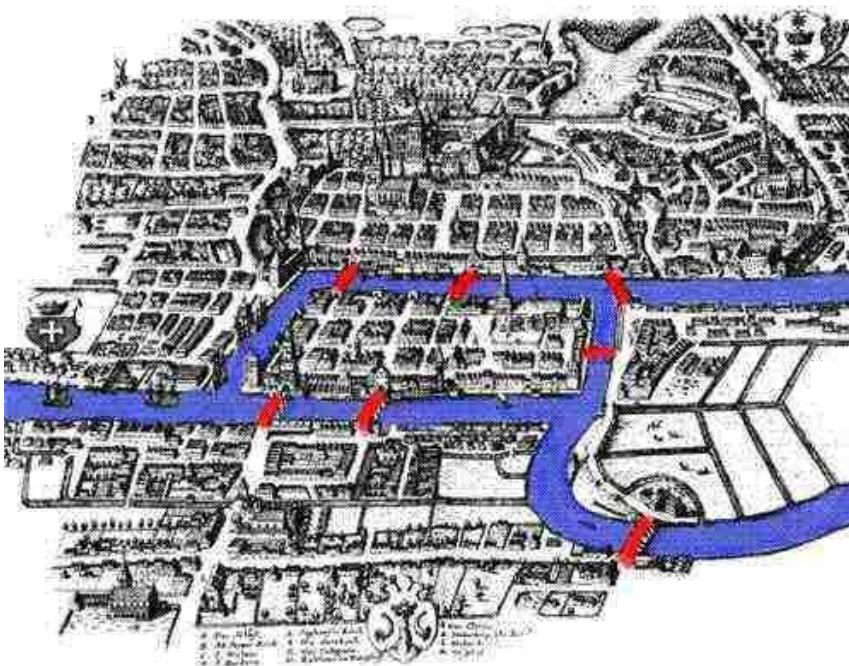
# Alte aplicații

- Rețele de calculatoare
- Limbaje formale
- Probleme de planificări, repartiții etc
- Teoria jocurilor

# Istoric

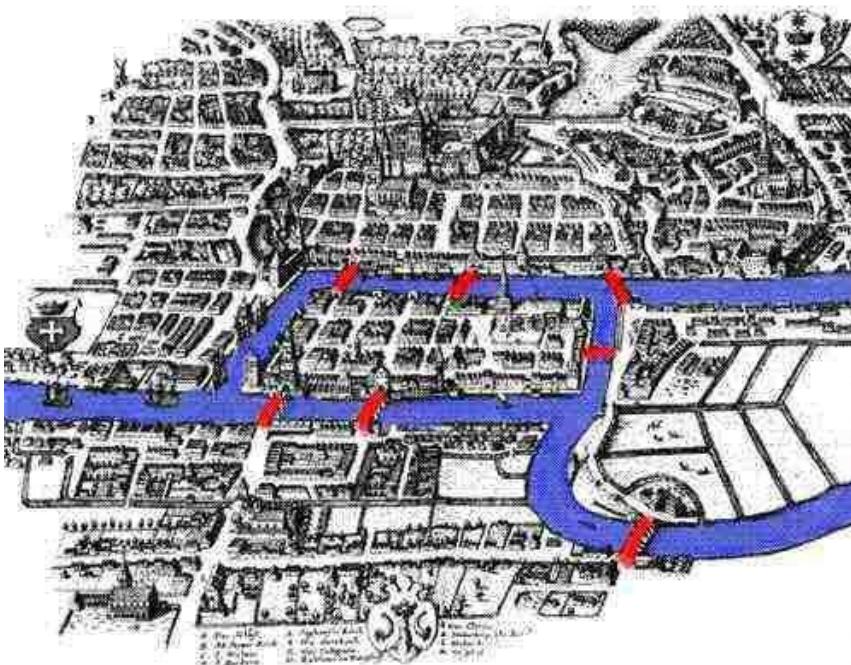


# Problema celor 7 poduri din Königsberg



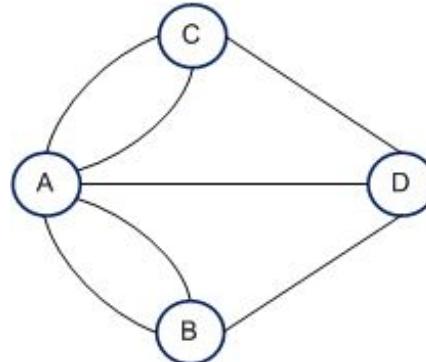
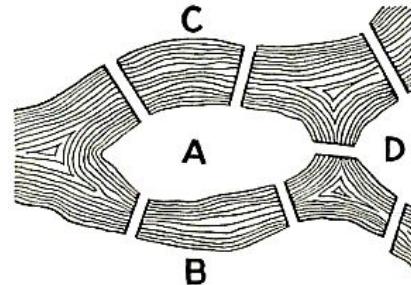
Este posibil ca un om să facă o plimbare în care să treacă pe toate cele 7 poduri, o singură dată?

# Problema celor 7 poduri din Königsberg

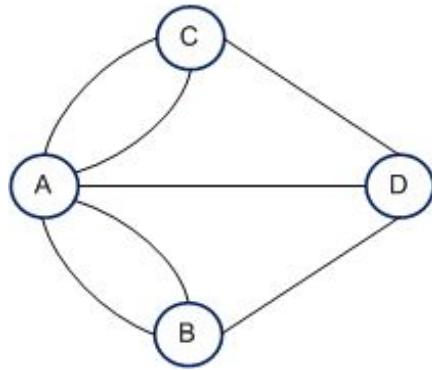


<https://www.maa.org/book/export/html/116597>

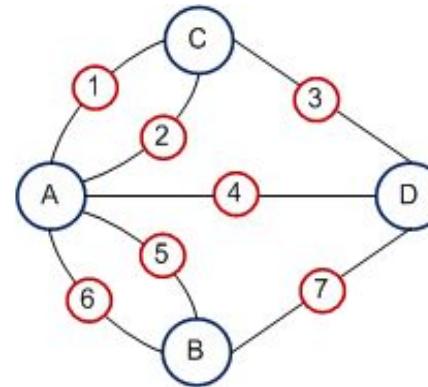
**Modelare:**



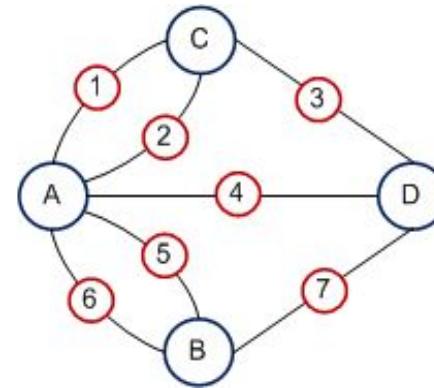
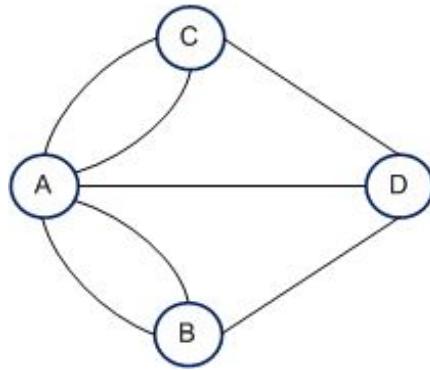
# Problema celor 7 poduri din Königsberg



Multigraf



# 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

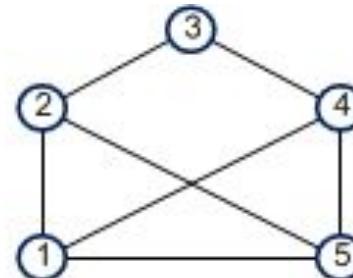
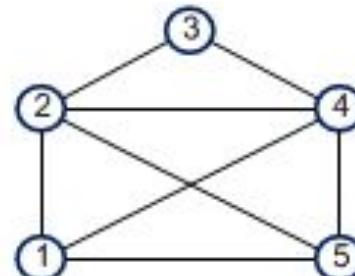
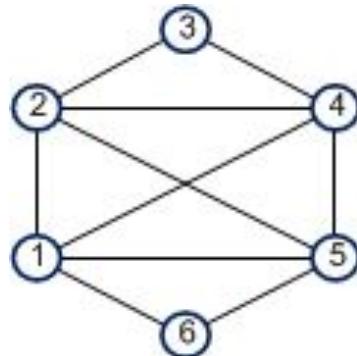
**Graf eulerian**

# 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)?

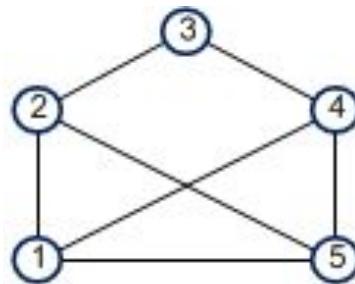
- tăierea unui material



# Problema celor 7 poduri din Königsberg

## Interpretare

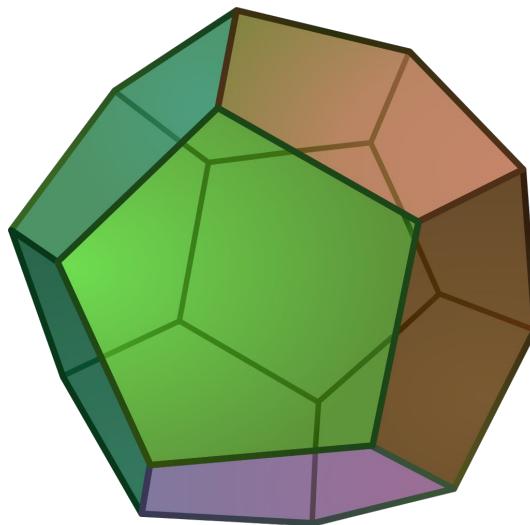
De câte ori (minim) trebuie să ridicăm pixul de pe hârtie pentru a desena diagrama?



# Jocul icosian

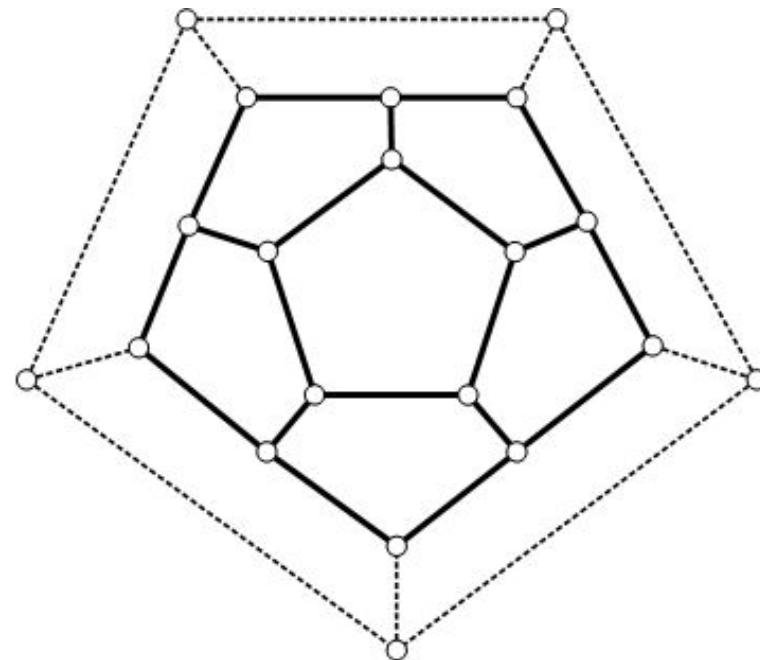
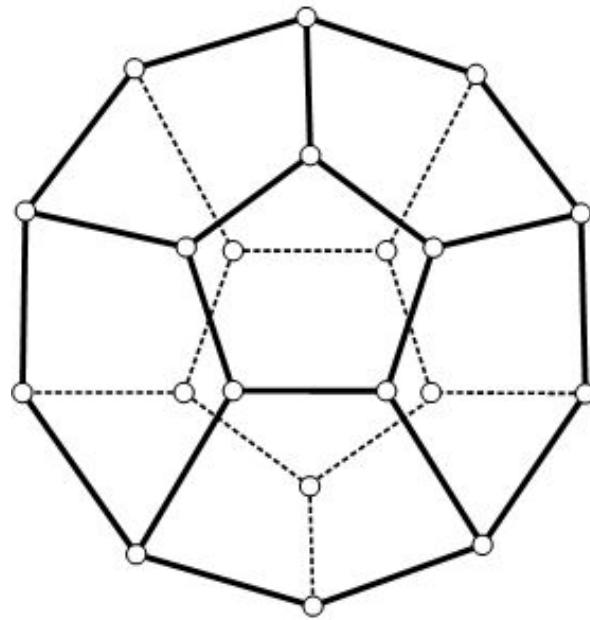
- 1856 - **Hamilton** - "voiaj în jurul lumii"

Există un traseu închis pe muchiile dodecaedrului care să treacă prin fiecare vârf o singură dată?

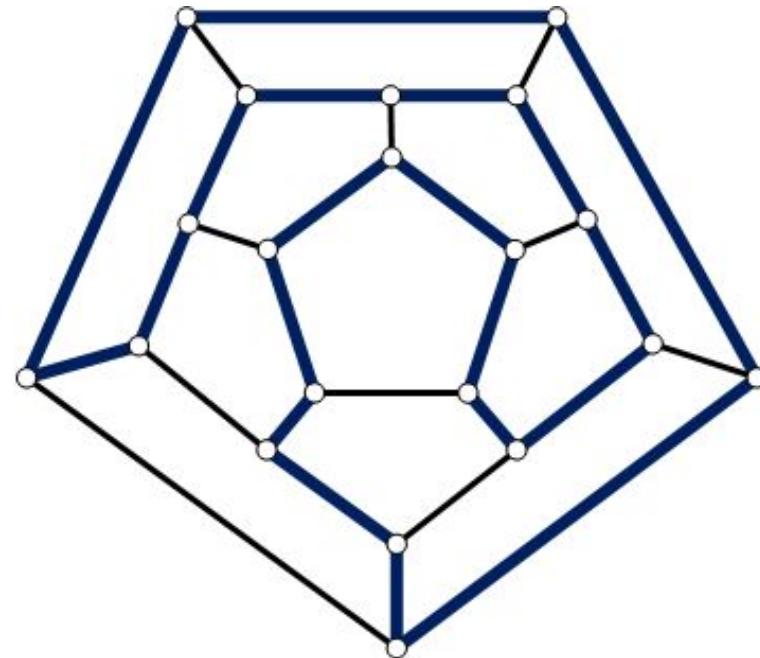
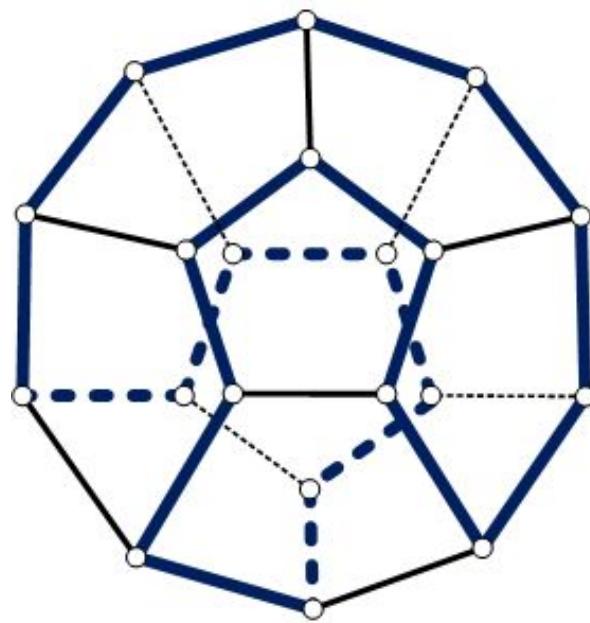


<https://ro.wikipedia.org/wiki/Dodecaedru>

# Jocul icosian



# Jocul icosian



# Jocul icosian

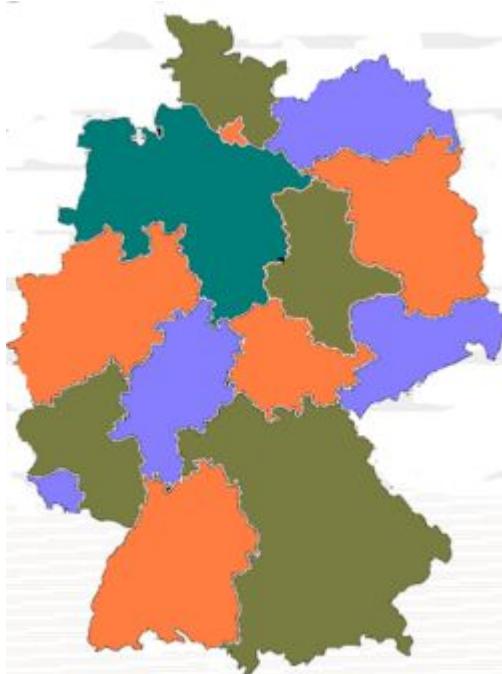
**Ciclu hamiltonian** - trece o singură dată prin toate vârfurile

**Graf hamiltonian**

**Problema comis-voiajorului**

# Problema celor 4 culori

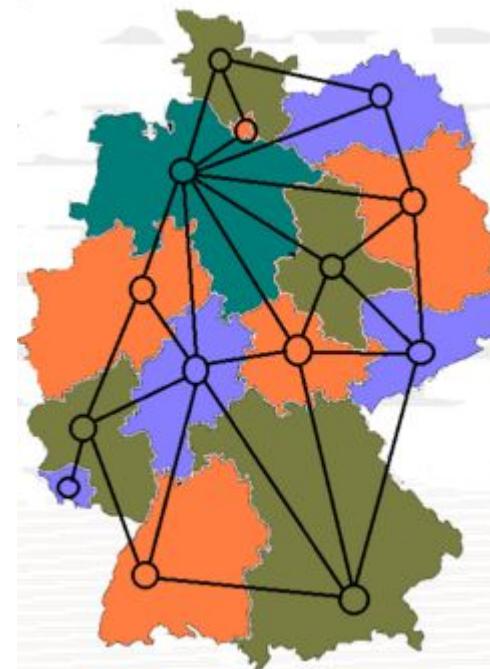
- 1852 - DeMorgan - "Problema celor 4 culori"



Se poate colora o hartă cu patru culori, astfel încât orice două țări, care au frontieră comună și care **nu se reduc la un punct**, să aibă culori diferite?

# Problema celor 4 culori

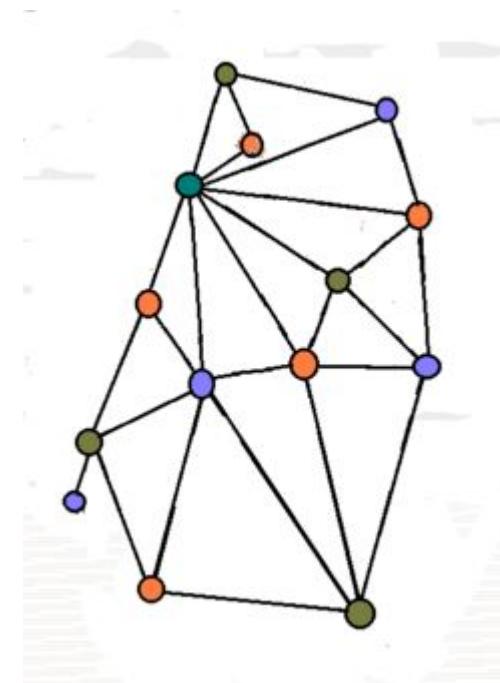
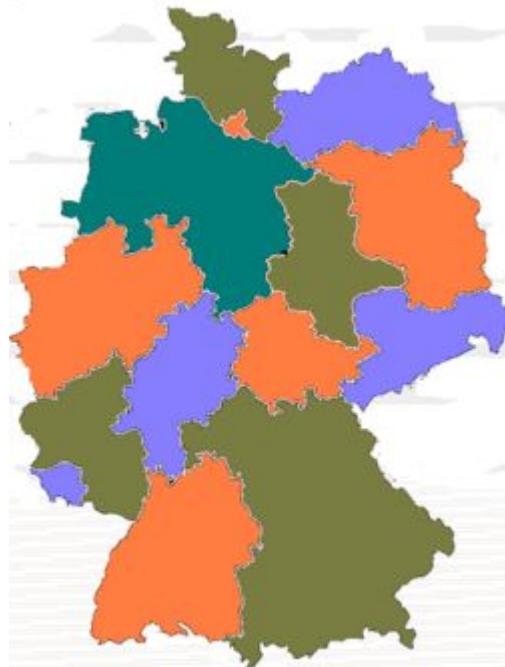
- 1852 - DeMorgan - "Problema celor 4 culori"



[https://en.citizendium.org/wiki/Four\\_color\\_theorem](https://en.citizendium.org/wiki/Four_color_theorem)

# Problema celor 4 culori

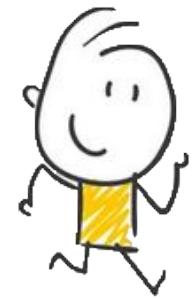
- 1852 - DeMorgan - "Problema celor 4 culori"



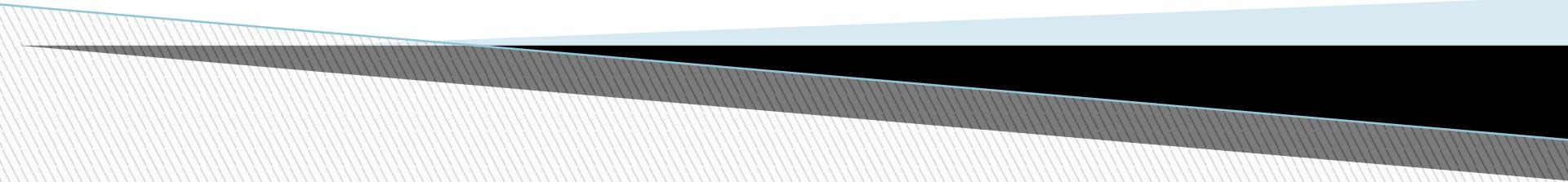
[https://en.citizendium.org/wiki/Four\\_color\\_theorem](https://en.citizendium.org/wiki/Four_color_theorem)

# Problema celor 4 culori

- **Problema celor 4 culori - Appel și Haken au răspuns afirmativ, în 1976, cu ajutorul calculatorului**



# **Noțiuni introductive**



# Multiset

- S o mulțime (finită) nevidă
- **Multiset**
  - **Intuitiv: O “mulțime” unde elementele se pot repeta**

# Multiset

□ S o mulțime (finită) nevidă

□ **Multiset**

- $R = (S, r)$ ,  $r : S \rightarrow \mathbb{N}$  **funcție de multiplicitate**

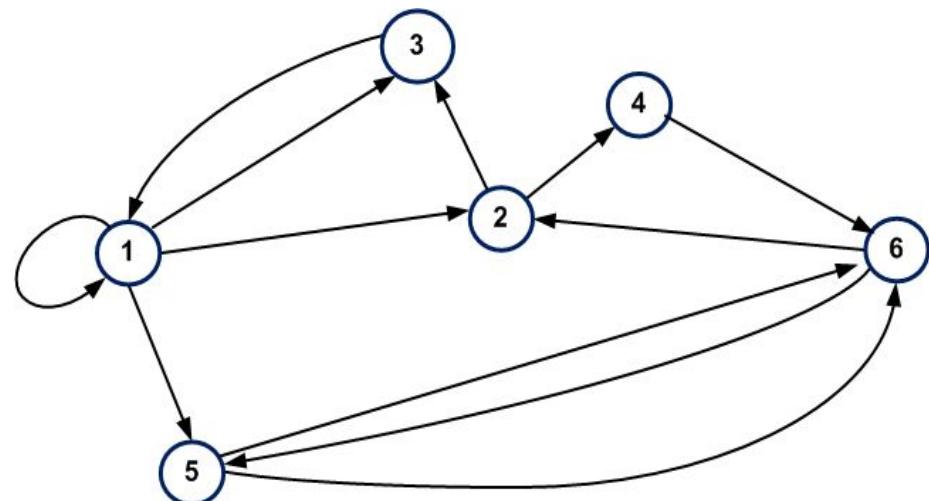
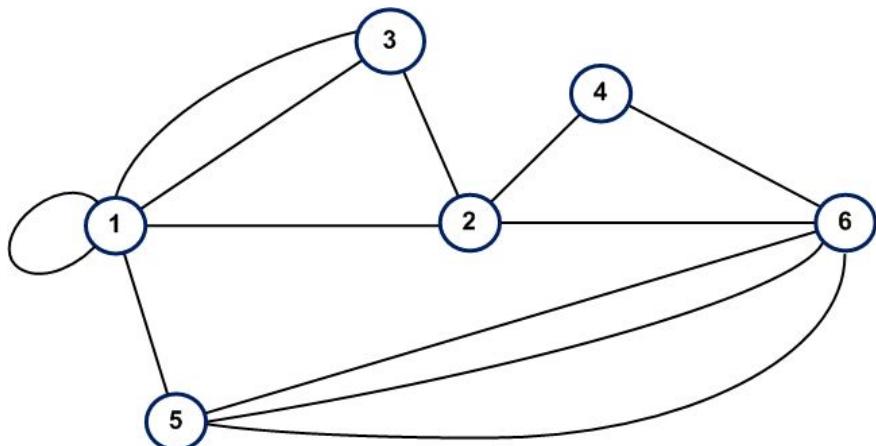
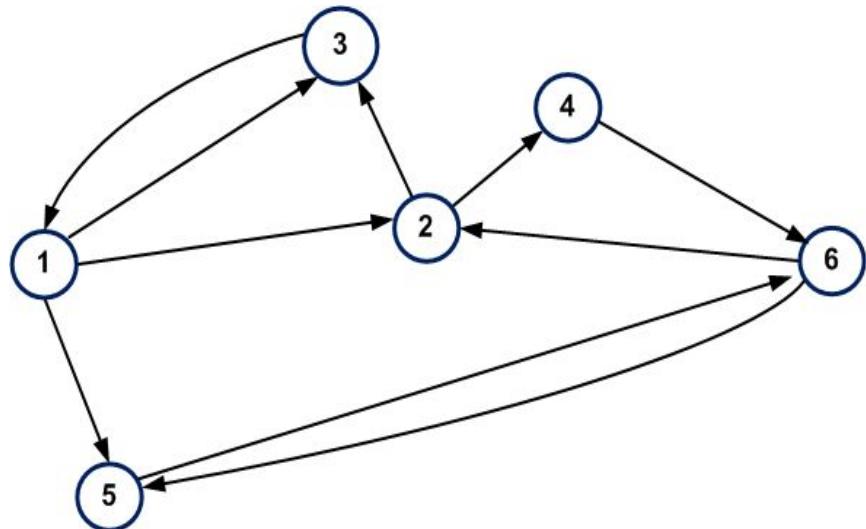
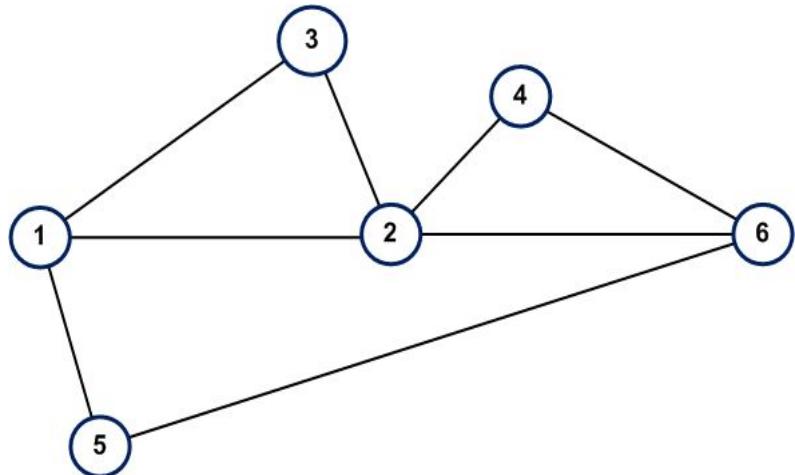
□ **Notație**

- $R = \{x^{r(x)} \mid x \in S\}$

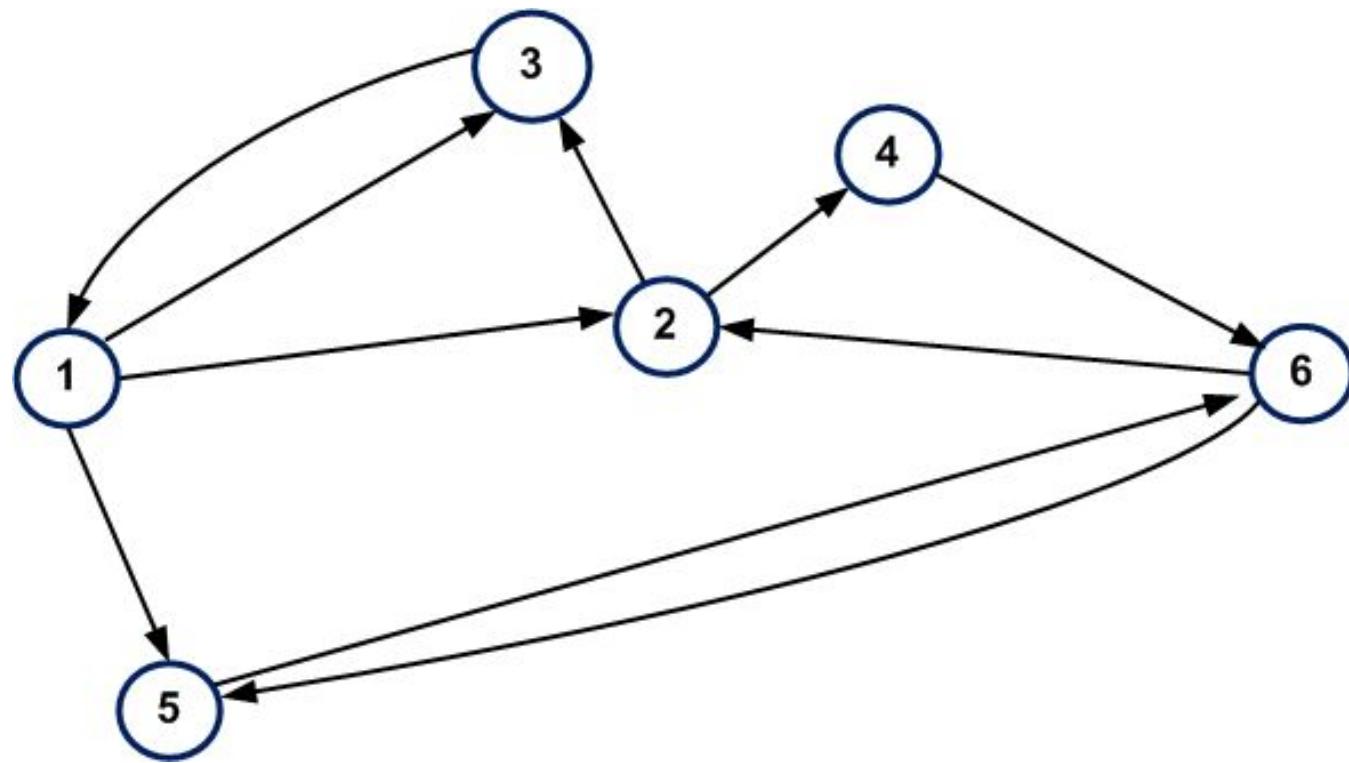
# Multiset

## □ Exemplu

- $S = \{1, 2, 3, 4, 5\}$
  - $R = \{2^2, 3, 5^3\}$
- $|R| = 2+1+3 = 6$  – suma multiplicităților
- $1 \notin R$



# Graf orientat



# Graf orientat

- **Graf orientat:**  $G = (V, E)$ 
  - **V – finită**
  - **E - perechi (ordonate) de 2 elemente distincte din V**
  - $v \in V$  - **vârf**
  - $e = (u, v) = uv$  - **arc**
    - $u = e^-$  - **vârf inițial / origine / extremitate inițială**
    - $v = e^+$  - **vârf final / terminus / extremitate finală**

# Graf orientat

□  $G = (V, E)$

◦  $d_G^-(u)$  - **grad interior**

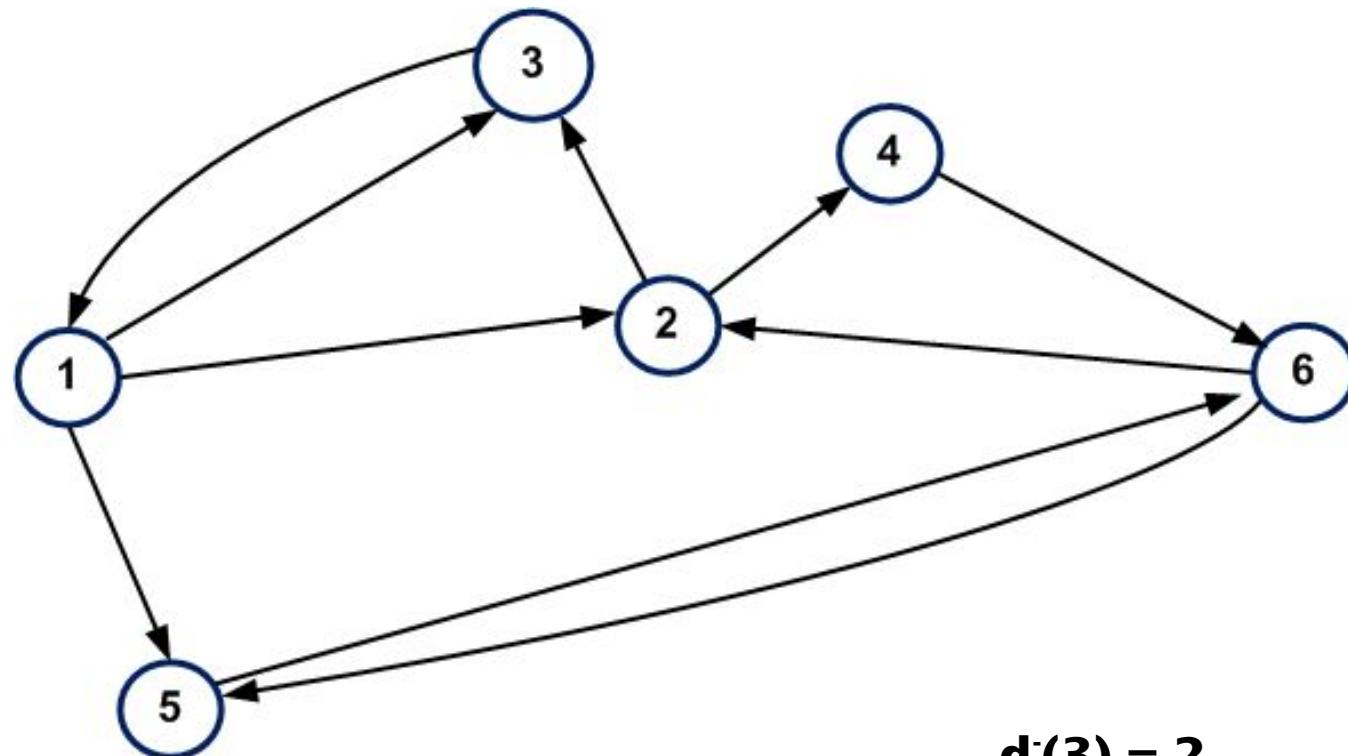
$$d_G^-(u) = |\{e \in E \mid u \text{ extremitate finala pentru } e\}|$$

◦  $d_G^+(u)$  - **grad exterior**

$$d_G^+(u) = |\{e \in E \mid u \text{ extremitate initiala pentru } e\}|$$

◦  $d_G(u)$  - **grad**

$$d_G(u) = d_G^+(u) + d_G^-(u)$$



$$d^-(3) = 2$$

$$d^+(3) = 1$$

# Graf orientat

- Are loc relația

$$\sum_{u \in V} d_G^-(u) = \sum_{u \in V} d_G^+(u) = |E|$$

# Multisetul gradelor

□ **G orientat,  $V = \{v_1, v_2, \dots, v_n\}$**

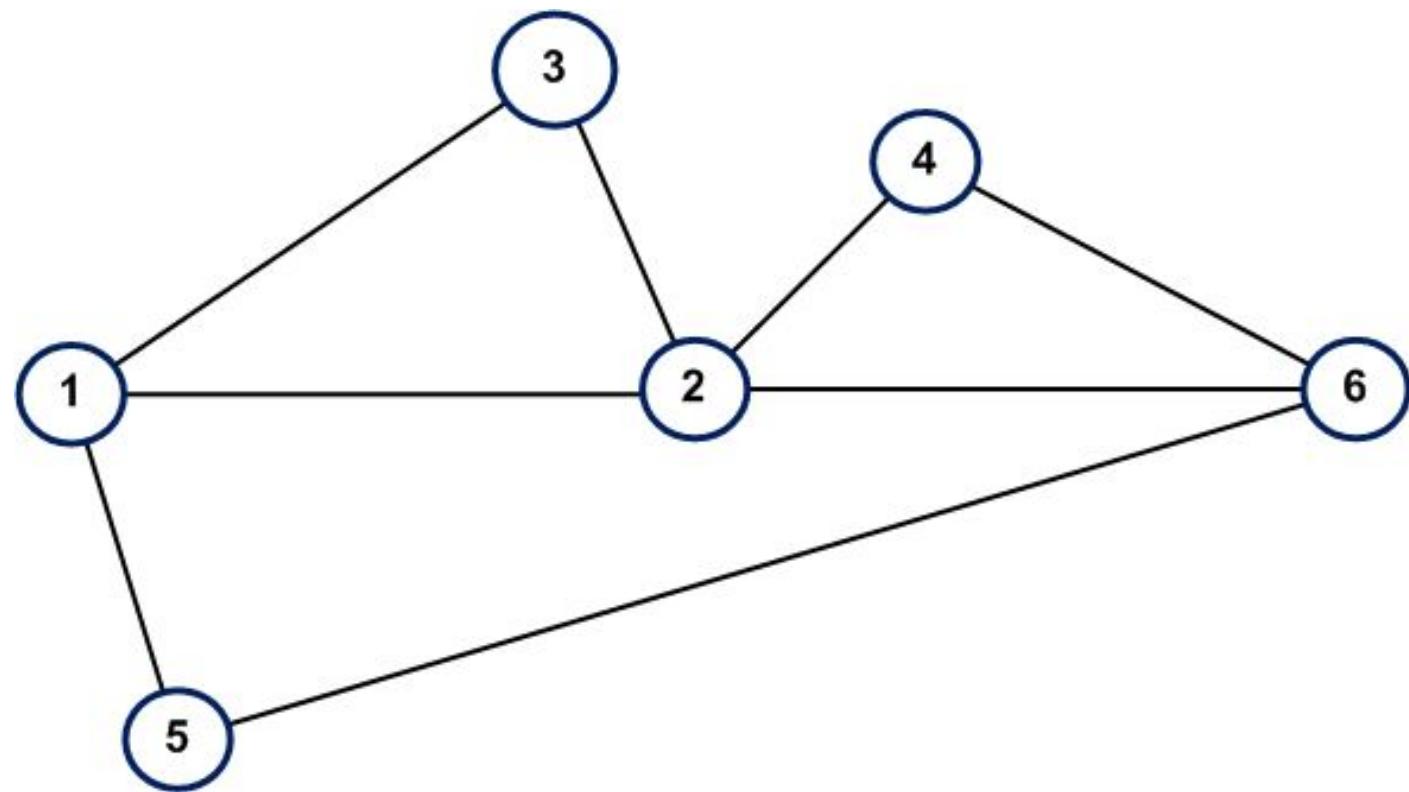
- **Multisetul gradelor interioare**

$$s^-(G) = \{d_G^-(v_1), \dots, d_G^-(v_n)\}$$

- **Multisetul gradelor exterioare**

$$s^+(G) = \{d_G^+(v_1), \dots, d_G^+(v_n)\}$$

# Graf neorientat



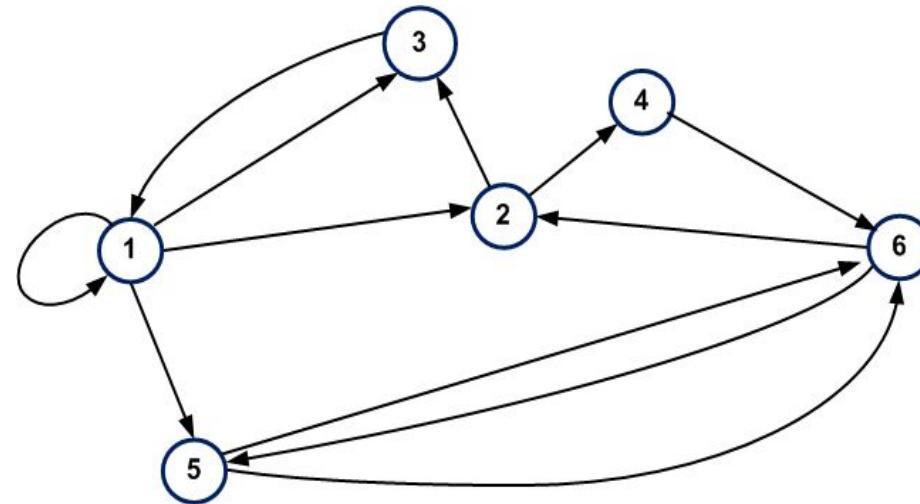
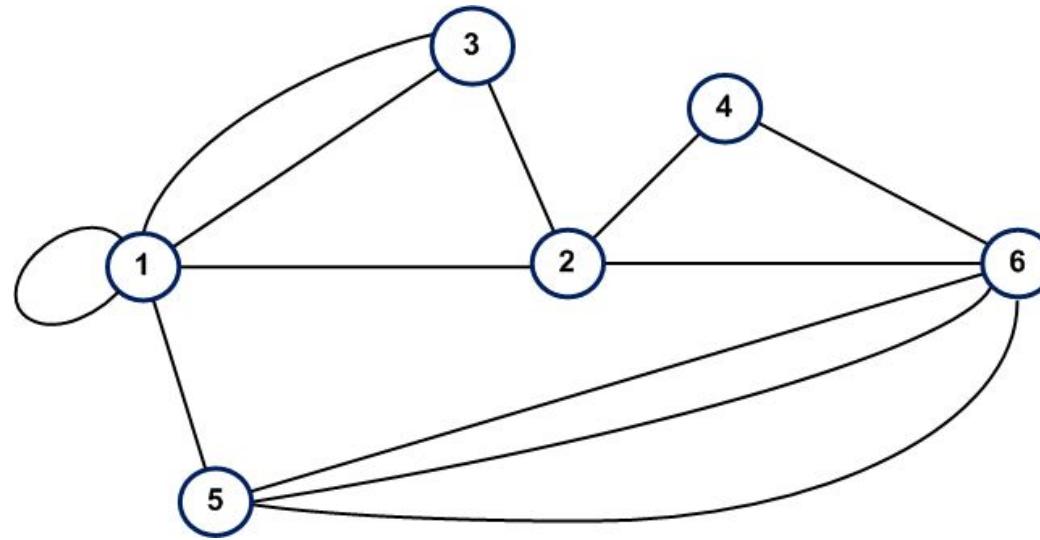
# Graf neorientat

- **Graf neorientat:**  $G = (V, E)$ 
  - **V - finită**
  - **E - submulțimi de 2 elemente (distingute) din V**
  - $v \in V$  - **vârf / nod**
  - $e = \{u,v\} = uv$  - **muchie**
    - $u, v$  - **capete / extremități**

# Notății

- $V(G), E(G)$
- $e = uv$

# Multigraf neorientat/orientat



# Multigraf

□  $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{r})$

$r(e)$  – **multiplicitatea muchiei e**

# Multigraf neorientat

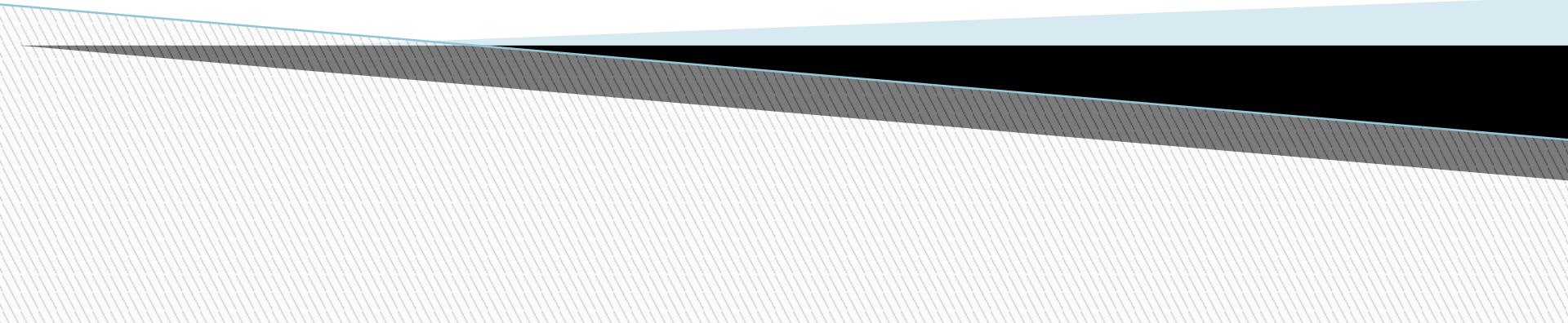
□  $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{r})$

$r(e)$  – multiplicitatea muchiei  $e$

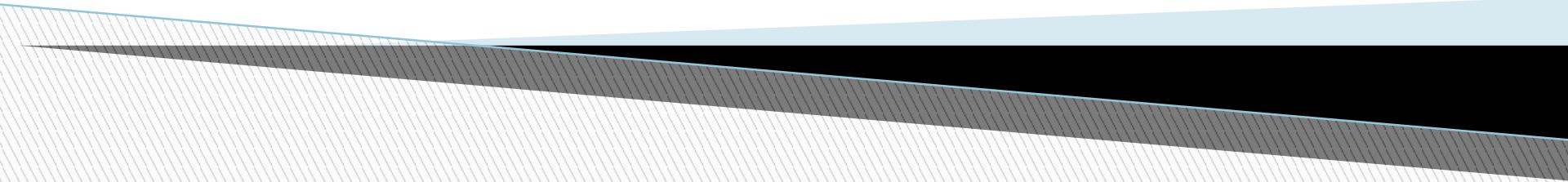
- $e = \{u, u\}$  = **bucă**
- $e$  cu  $r(e) > 1$  = **muchie multiplă**

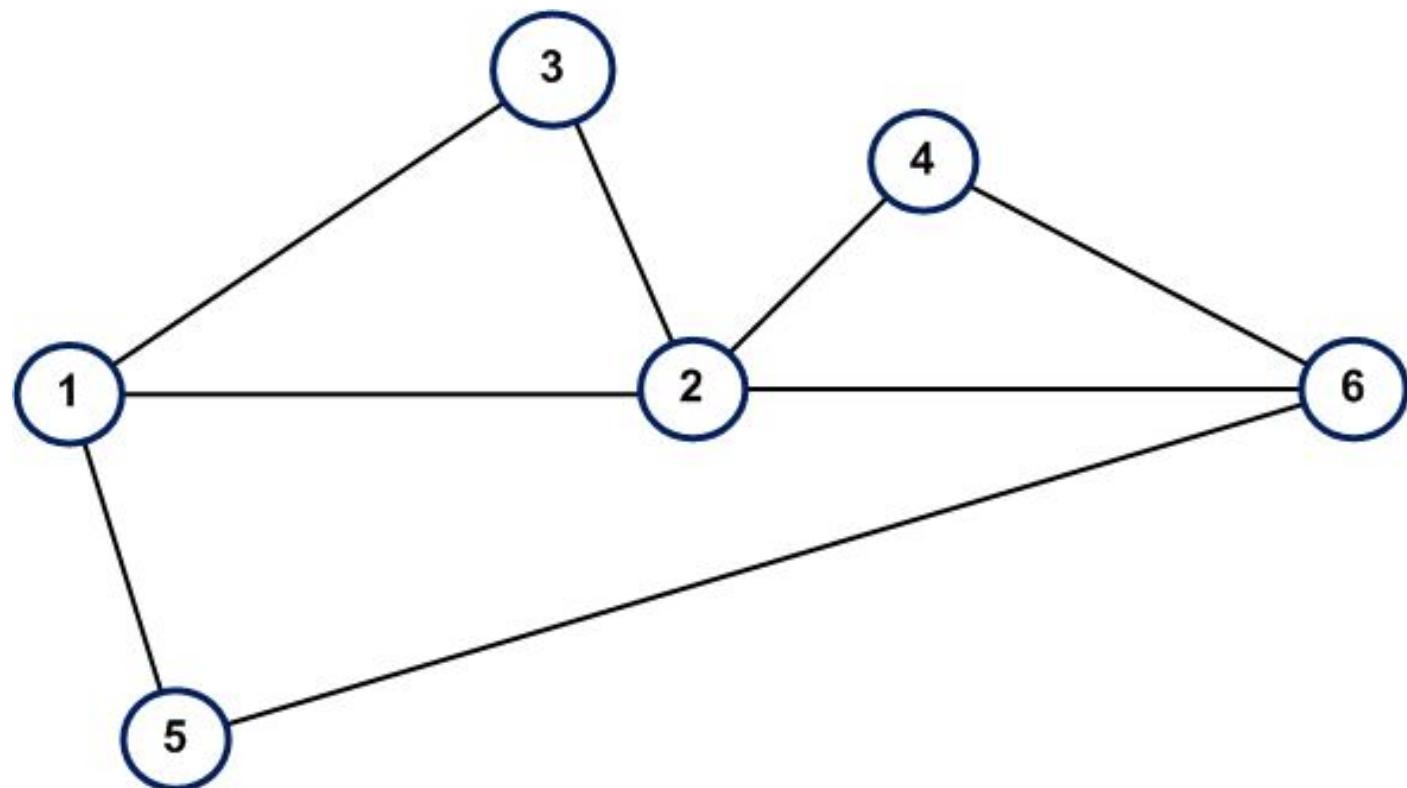
$$d_G(u) = |\{e \in E \mid e \text{ nu este buclă a lui } u\}| + 2 \cdot |\{e \in E \mid e \text{ este buclă a lui } u\}|$$

# **Alte noțiuni fundamentale**



# **Adiacență. Încidență**





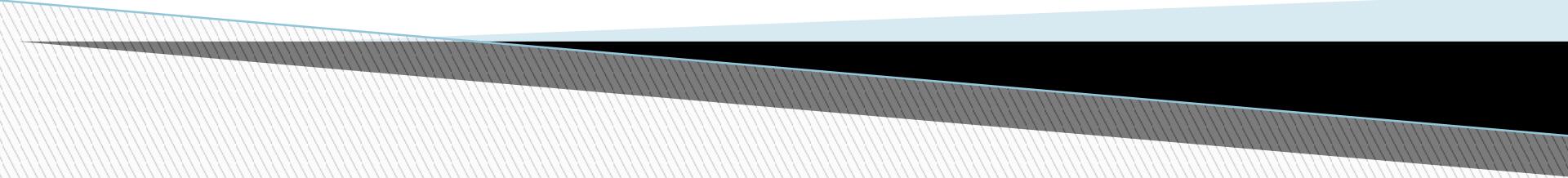
# Adiacență. Incidentă

- Fie  $G = (V, E)$  un graf **neorientat**
  - $u$  și  $v \in V$  sunt **adiacente** dacă  $uv \in E$
  - Un **vecin** al lui  $u \in V$  este un vârf adiacent cu el
  - **Notatie**  $N_G(u) =$  mulțimea vecinilor lui  $u$

# Adiacență. Incidentă

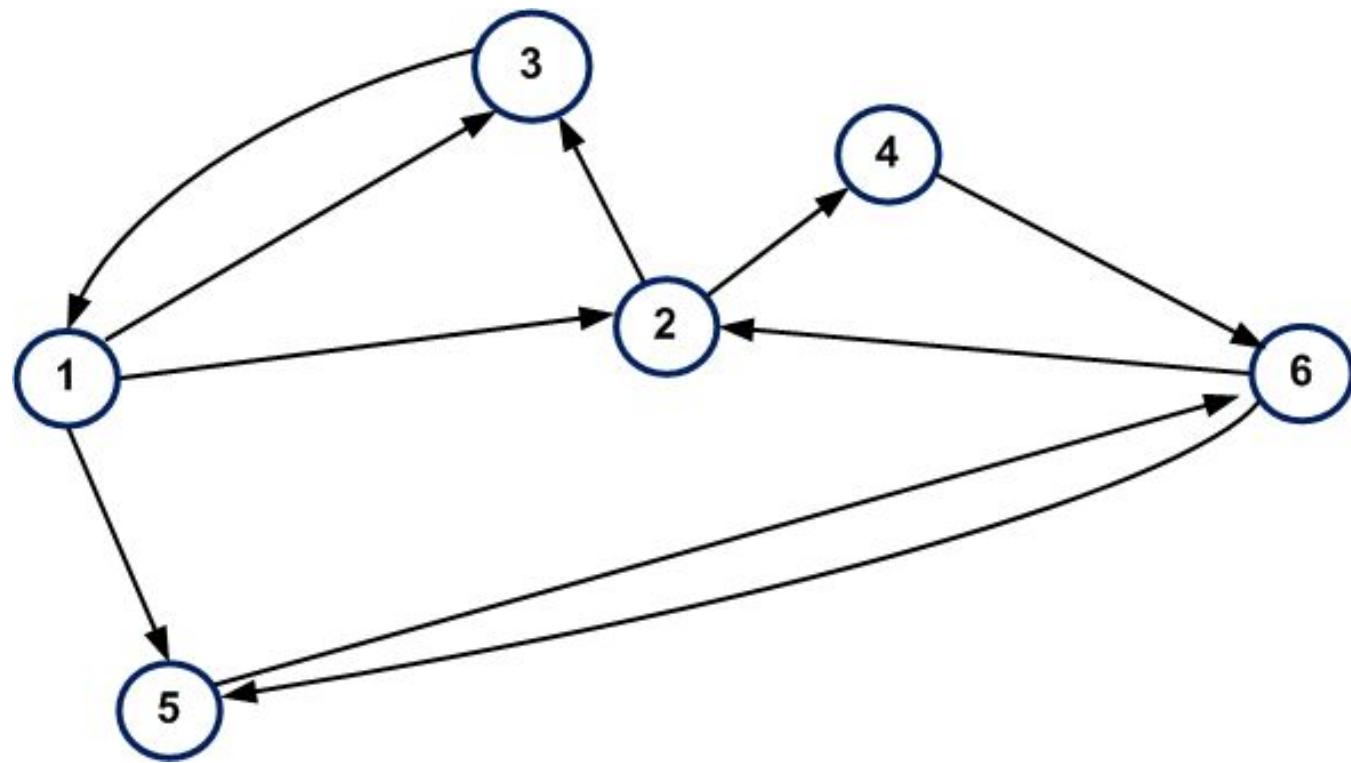
- Fie  $G = (V, E)$  un graf **neorientat**
  - O muchie  $e \in E$  este **incidentă** cu un vârf  $u$  dacă  $u$  este extremitate a lui  $e$
  - $e$  și  $f \in E$  sunt **adiacente** dacă există un vârf în care sunt incidente (au o extremitate în comun)

# **Drumuri. Circuite**



# Drumuri. Circuite

- **Drum (walk)**
- **Drum simplu (trail)**
- **Drum elementar (path)**
- **Circuit + elementar**
- **Lungimea unui drum**
- **Distanță între două vârfuri**



# Drumuri. Circuite

Fie  $G$  un graf **orientat**

- Un **drum** este o secvență  $P$  de vârfuri

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$

unde  $v_1, \dots, v_k \in V(G)$

cu proprietatea că între oricare două vârfuri consecutive există arc:

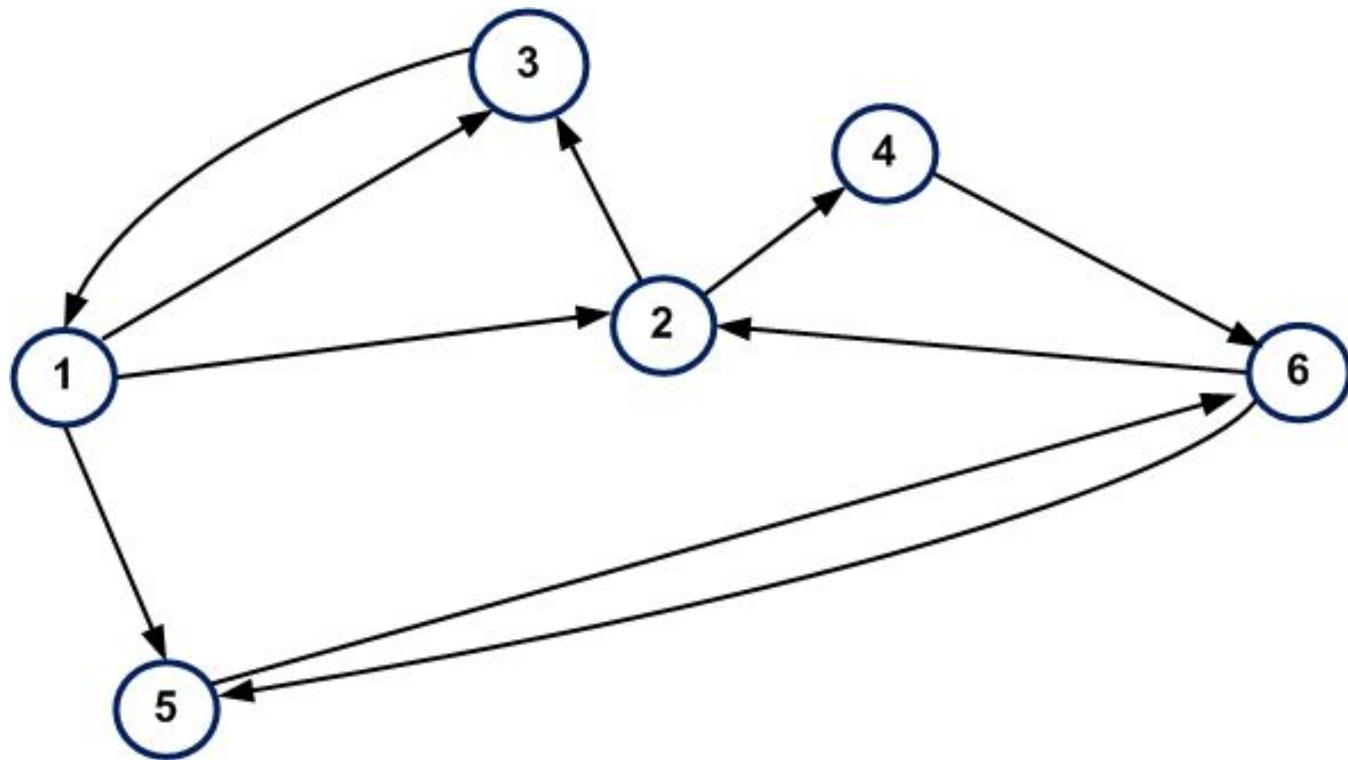
$$(v_i, v_{i+1}) \in E(G), \forall i \in \{1, \dots, k - 1\}$$

# Drumuri. Circuite

Fie  $G$  un graf **orientat și un drum**

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$

- $P$  este **drum simplu** dacă nu conține un arc de mai multe ori ( $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$ ,  $\forall i \neq j$ )
- $P$  este **drum elementar** dacă nu conține un vârf de mai multe ori ( $v_i \neq v_j$ ,  $\forall i \neq j$ )



**[1, 2, 4, 6, 2, 4] - drum care nu este simplu**

**[1, 2, 4, 6, 2, 3] - drum simplu care nu este elementar**

**[1, 2, 4, 6] - drum elementar**

# Drumuri. Circuite

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$

- **Lungimea** lui  $P = l(P) = k-1$  (cardinalul multisetului arcelor lui  $P$ )
- $v_1$  și  $v_k$  se numesc **capetele/ extremitățile** lui  $P$
- $P$  se numește și  **$v_1-v_k$  lanț**

# Drumuri. Circuite

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$

## □ Notăm

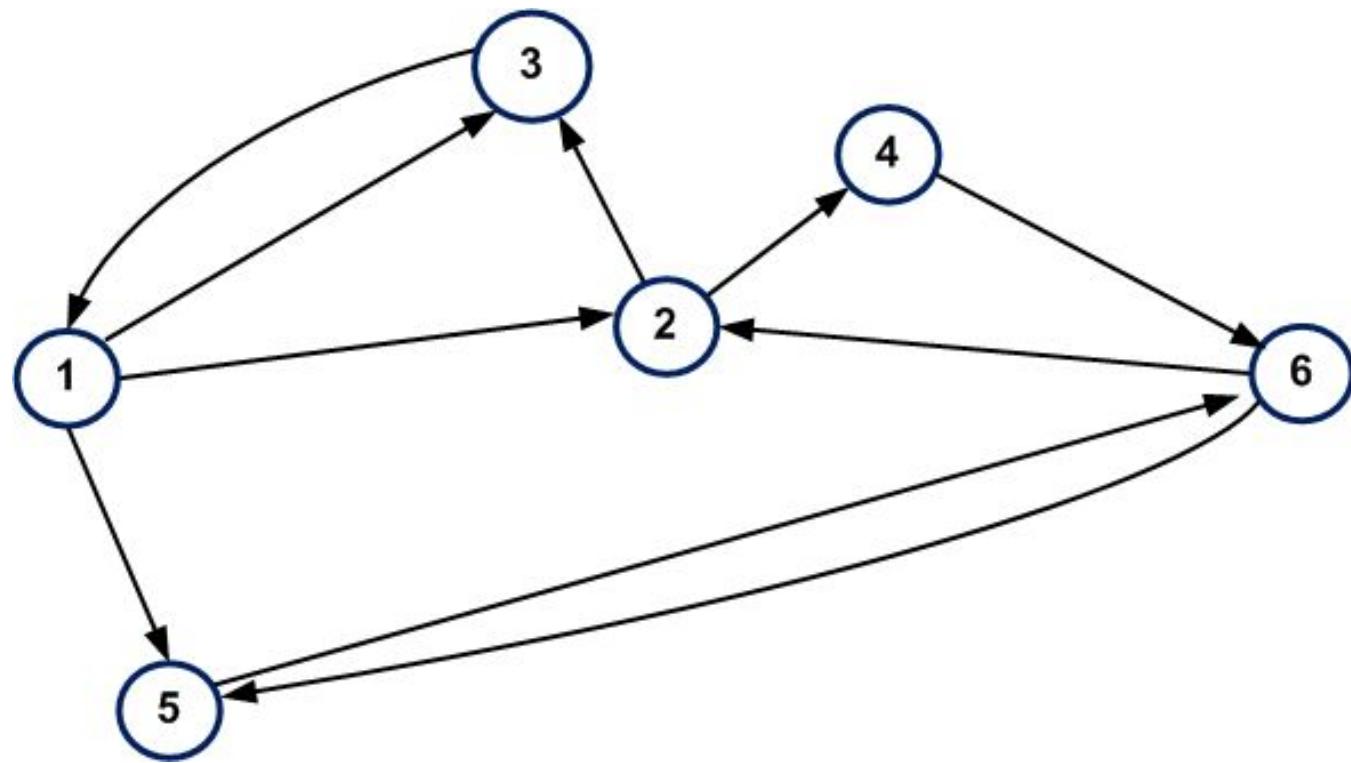
- $V(P) = \{v_1, v_2, \dots, v_k\}$
- $e_i = (v_i, v_{i+1})$
- $E(P) = \{e_1, e_2, \dots, e_{k-1}\}$

# Drumuri. Circuite

- Pentru două vârfuri  $u$  și  $v$  definim **distanța de la  $u$  la  $v$**  astfel:

$$d_G(u, v) = \begin{cases} 0, & \text{daca } u = v \\ \infty, & \text{daca nu există } u - v \text{ drum în } G \\ \min\{l(P) \mid P \text{ este } u - v \text{ drum în } G\}, & \text{altfel} \end{cases}$$

(cea mai mică lungime a unui  $u$ - $v$  drum)



# Drumuri. Circuite

- Pentru două vârfuri  $u$  și  $v$  definim **distanța de la  $u$  la  $v$**  astfel:

$$d_G(u, v) = \begin{cases} 0, & \text{daca } u = v \\ \infty, & \text{daca nu există } u - v \text{ drum în } G \\ \min\{l(P) \mid P \text{ este } u - v \text{ drum în } G\}, & \text{altfel} \end{cases}$$

(cea mai mică lungime a unui  $u - v$  drum)

- ▶ Un  $u - v$  drum de lungime  $d_G(u, v)$  se numește **drum minim de la  $u$  la  $v$**
- ▶ Vom nota și  $d(u, v)$  dacă  $G$  se deduce din context

# Drumuri. Circuite

- Un **circuit** este un drum simplu cu capetele identice

$$C = [v_1, v_2, \dots, v_{k-1}, v_k, v_1]$$

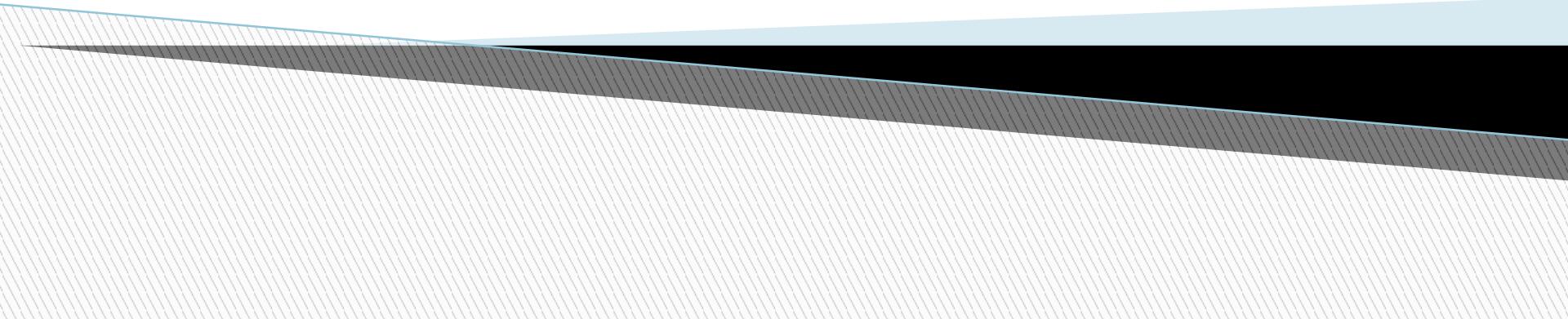
- **Circuit elementar**

- **Notării**  $V(C)$ ,  $E(C)$

# Drumuri. Circuite

- Un **circuit** este un drum simplu cu capetele identice
- $C = [v_1, v_2, \dots, v_{k-1}, v_k, v_1]$
- C este **circuit simplu** - dacă drumul asociat este simplu
- **Circuit elementar**
- **Notării**  $V(C)$ ,  $E(C)$

# **Lanțuri. Cicluri**



# Lanțuri. Cicluri

Pentru  $G$  graf **neorientat** – **noțiuni similare**

- Un **lanț** este o secvență  $P$  de vârfuri cu proprietatea că oricare două vârfuri consecutive sunt adiacente

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$

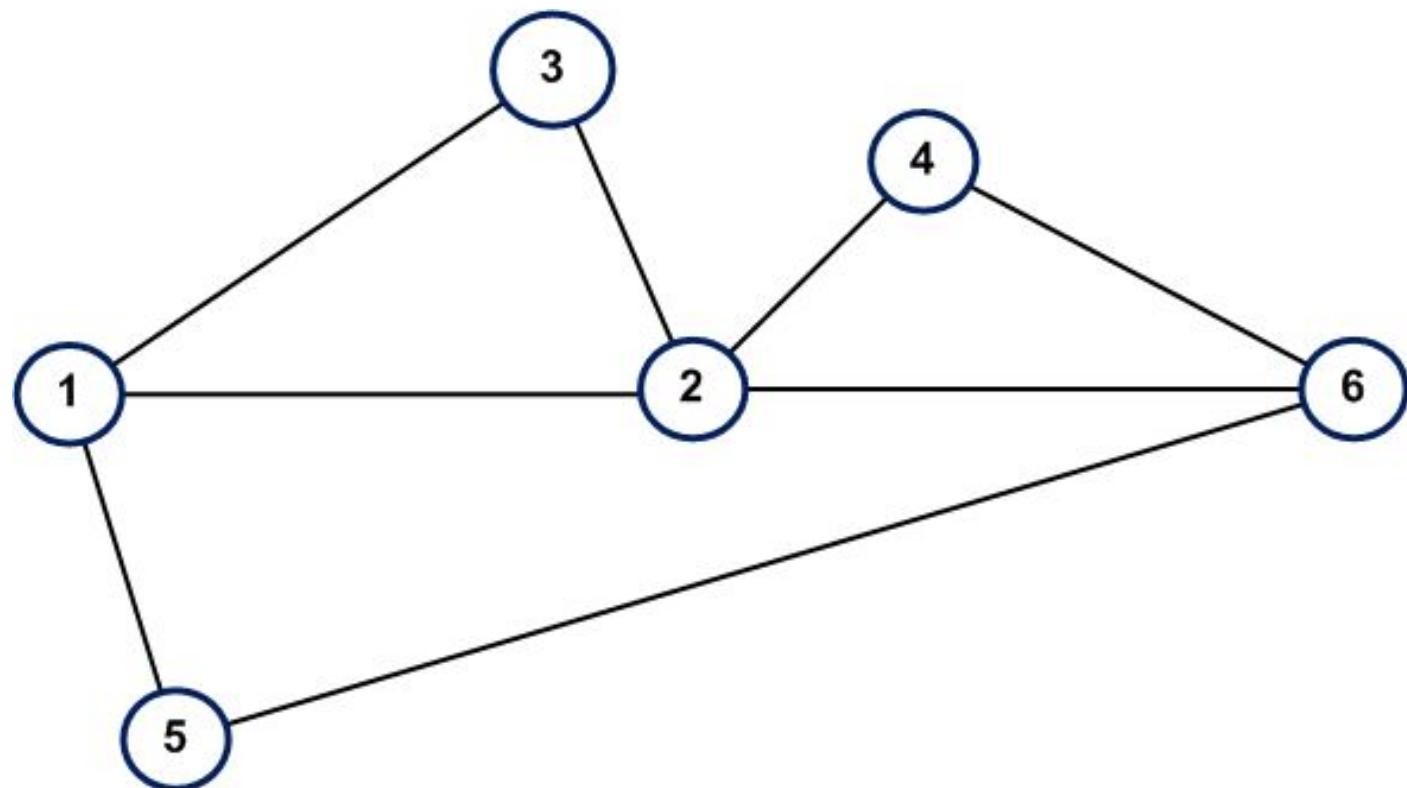
- **lanț simplu / lanț elementar / lungime**
- **ciclu / ciclu elementar**
- **distanță / lanț minim**

# Lanțuri. Cicluri

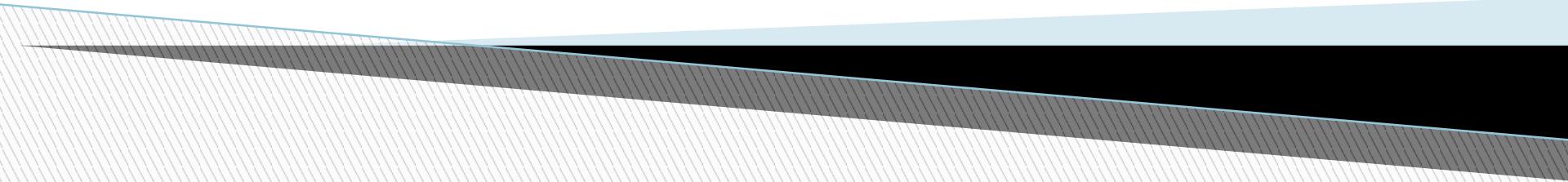
## Observație

- În cazul unui graf simplu putem descrie un lanț/ciclu doar ca o **succesiune de vârfuri** (fără a mai preciza și muchiile):

$$P = [v_1, v_2, \dots, v_{k-1}, v_k]$$



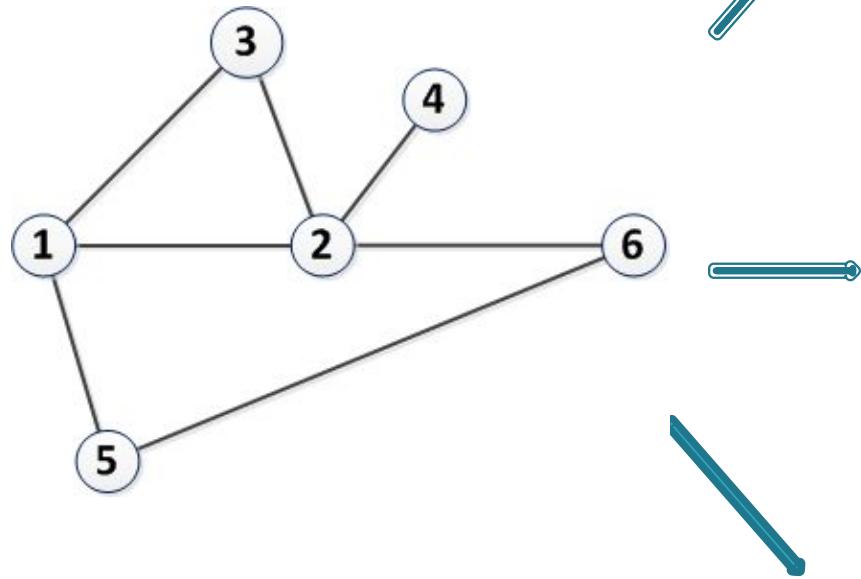
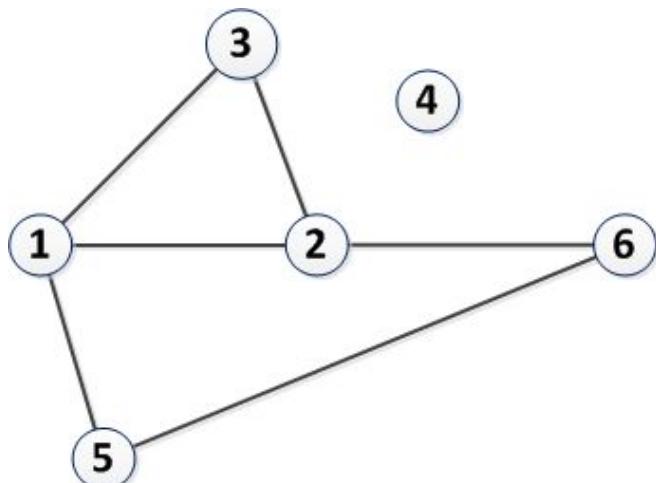
# **Graf parțial. Subgraf. Conexitate**

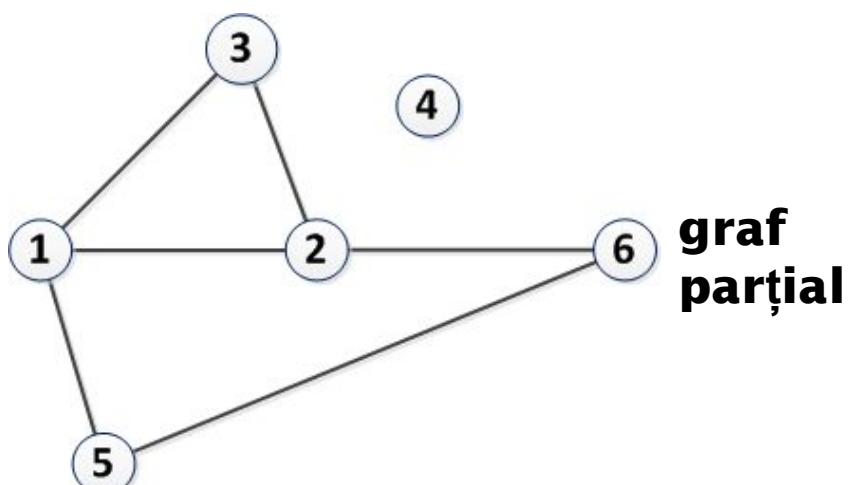


# Graf parțial. Subgraf

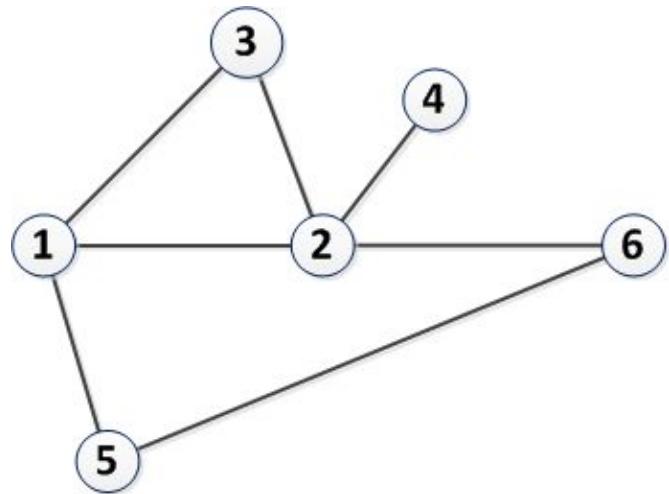
- **graf parțial**
- **subgraf**
- **subgraf induc**

**graf  
parțial**



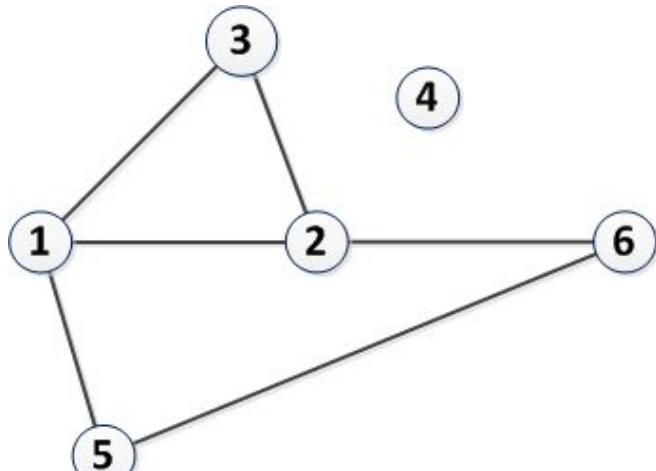


**graf  
parțial**

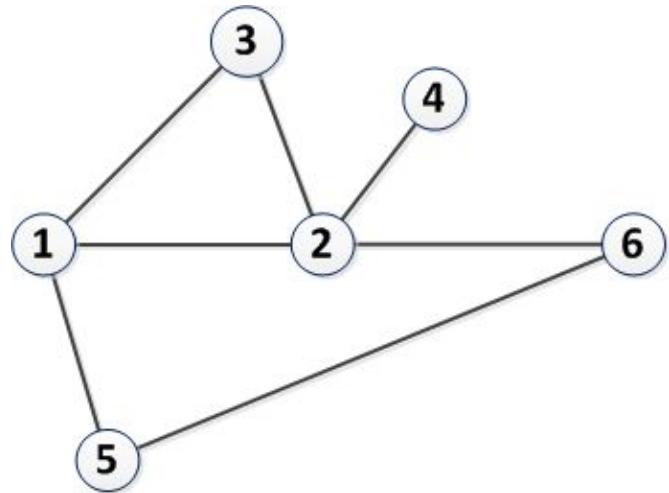


**subgr  
af**

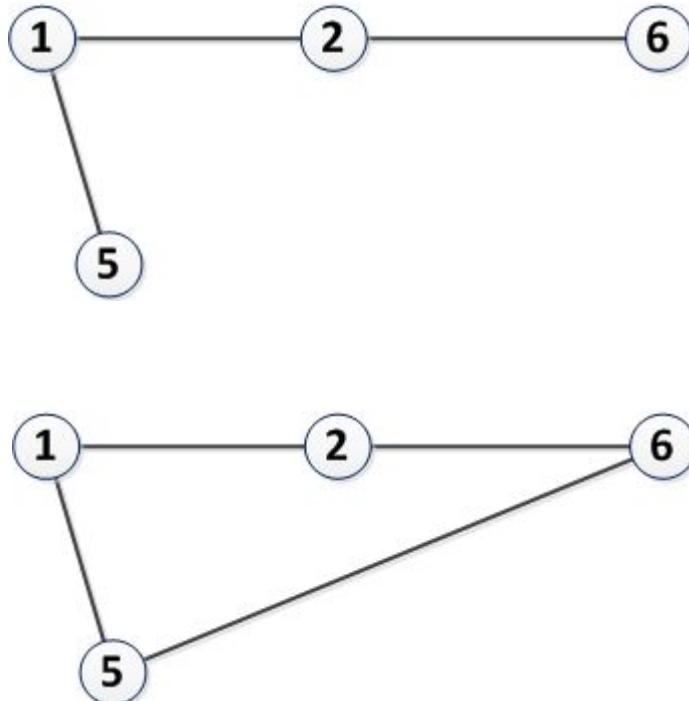
**graf  
parțial**



**subgr  
af**



**subgra  
f  
indus  
de  
{1,2,5,6  
}**



# Graf parțial. Subgraf

Fie  $G = (V, E)$  și  $G_1 = (V_1, E_1)$  două grafuri

- $G_1$  este **graf parțial** al lui  $G$  (vom nota  $G_1 \leq G$ ) dacă  
 $V_1 = V, E_1 \subseteq E$

# Graf parțial. Subgraf

□

Fie  $G = (V, E)$  și  $G_1 = (V_1, E_1)$  două grafuri

- ▶  $G_1$  este **graf parțial** al lui  $G$  (vom nota  $G_1 \leq G$ ) dacă

$$V_1 = V, \quad E_1 \subseteq E$$

- ▶  $G_1$  este **subgraf** al lui  $G$  (vom nota  $G_1 \prec G$ ) dacă

$$V_1 \subseteq V, \quad E_1 \subseteq E$$

# Graf parțial. Subgraf

□

Fie  $G = (V, E)$  și  $G_1 = (V_1, E_1)$  două grafuri

- ▶  $G_1$  este **graf parțial** al lui  $G$  (vom nota  $G_1 \leq G$ ) dacă

$$V_1 = V, \quad E_1 \subseteq E$$

- ▶  $G_1$  este **subgraf** al lui  $G$  (vom nota  $G_1 \prec G$ ) dacă

$$V_1 \subseteq V, \quad E_1 \subseteq E$$

- ▶  $G_1$  este **subgraf induș de  $V_1$  în  $G$**  (vom nota  $G_1 = G[V_1]$ ) dacă

$$V_1 \subseteq V,$$

$$E_1 = \{e \mid e \in E(G), e \text{ are ambele extremități în } V_1\}$$

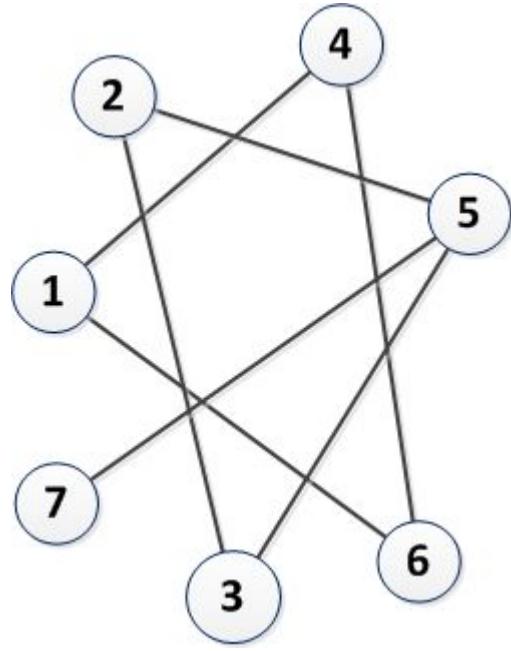
(toate arcele/muchiile cu extremități în  $V_1$ )

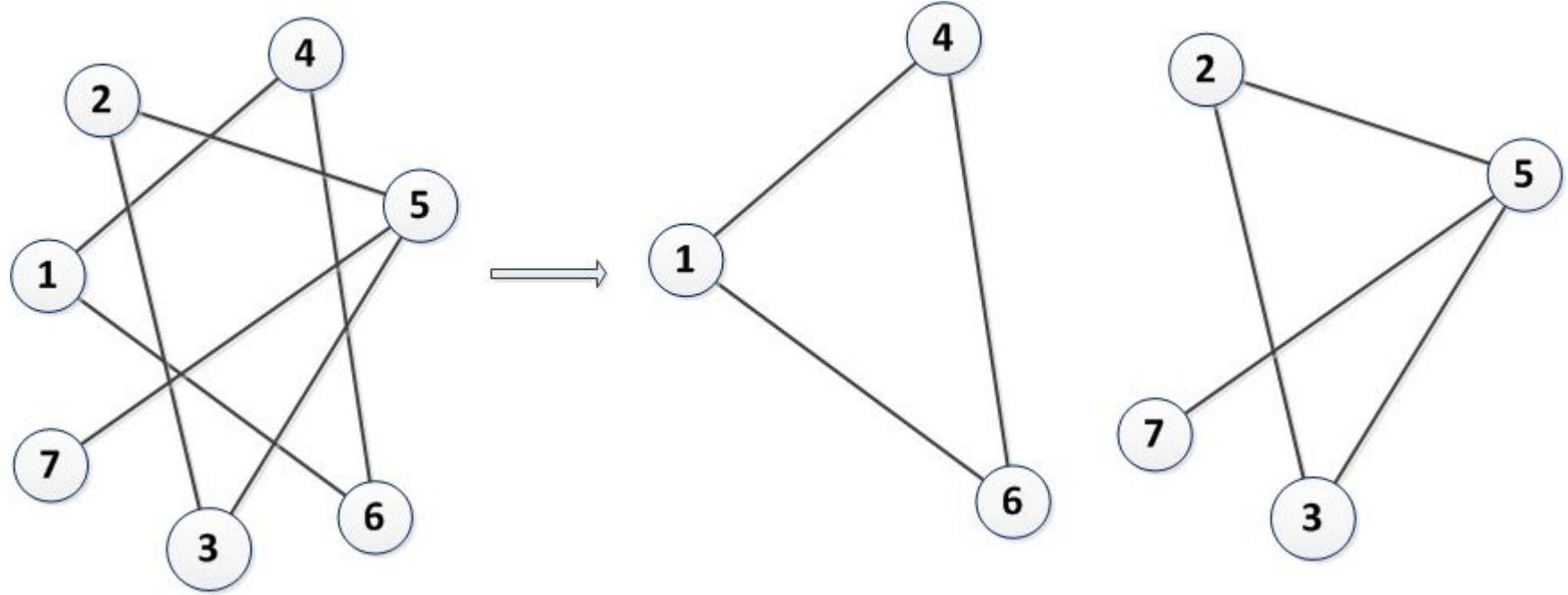
# Conexitate

Fie  $G = (V, E)$  un graf neorientat

- **graf conex**

- **componentă conexă**





**două componente  
conexe**

# Conexitate

Fie  $G = (V, E)$  un graf neorientat

- $G$  este **graf conex** dacă între orice două vârfuri distincte există un lanț

# Conexitate

Fie  $G = (V, E)$  un graf neorientat

- $G$  este **graf conex** dacă între orice două vârfuri distincte există un lanț
- O **componentă conexă** a lui  $G$  este un subgraf **indus** conex maximal (care nu este inclus în alt subgraf conex)

# Conexitate

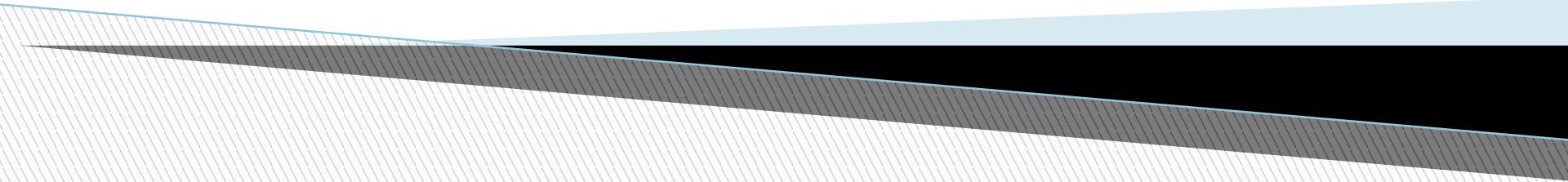
Fie  $G = (V, E)$  un graf neorientat

- $G$  este **graf conex** dacă între orice două vârfuri distincte există un lanț
- O **componentă conexă** a lui  $G$  este un subgraf inducător conex maximal (care nu este inclus în alt subgraf conex)
- Pentru cazul orientat – **tare-conexitate**

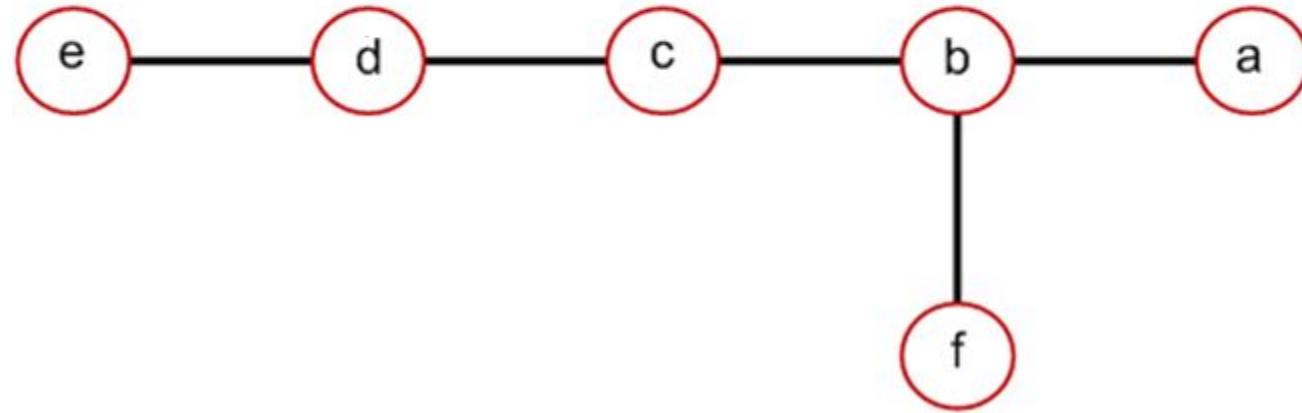
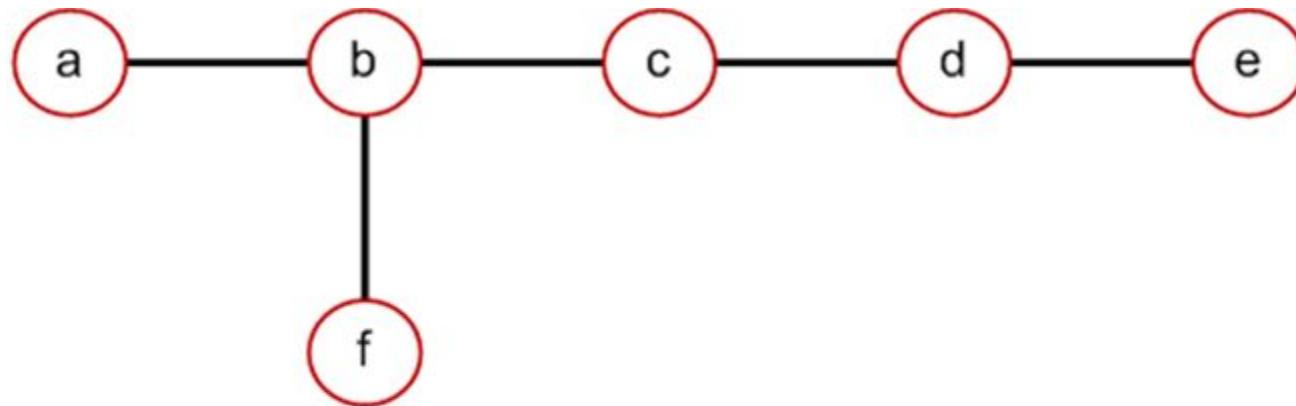
# Notări

- $\mathbf{G} - v$ ,  $v \in V(G)$
- $\mathbf{G} - e$ ,  $e \in E(G)$
- $\mathbf{G} - V'$ ,  $V' \subseteq V(G)$
- $\mathbf{G} - E'$ ,  $E' \subseteq E(G)$
- $\mathbf{G} + e$

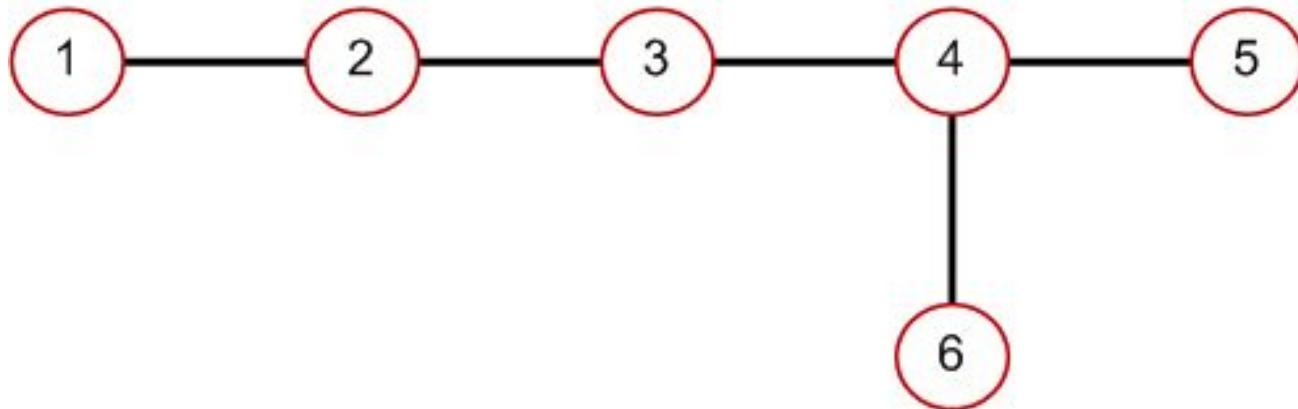
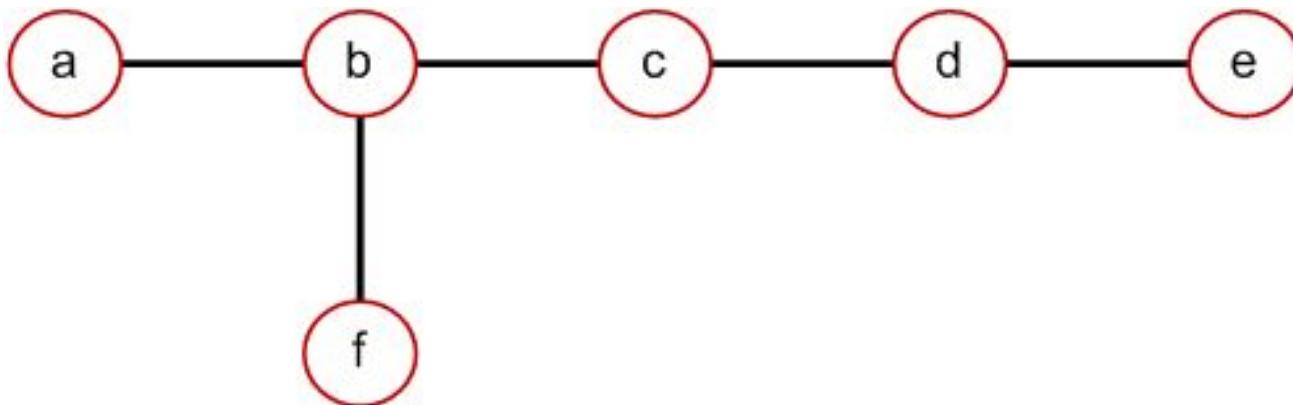
# **Egalitate. Izomorfism**



# Egalitate



# Egalitate?



# Izomorfism

Fie  $G_1, G_2$  două grafuri

- $G_1 = (V_1, E_1)$
- $G_2 = (V_2, E_2)$

Grafurile  $G_1$  și  $G_2$  sunt **izomorfe** ( $G_1 \sim G_2$ )  $\Leftrightarrow$   
există  $f : V_1 \rightarrow V_2$  bijectivă cu

$$uv \in E_1 \Leftrightarrow f(u)f(v) \in E_2$$

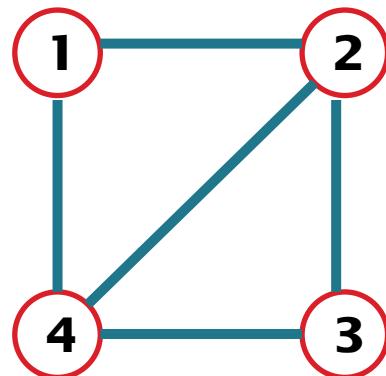
pentru orice  $u, v \in V_1$

**(f conservă adiacența și neadiacența)**

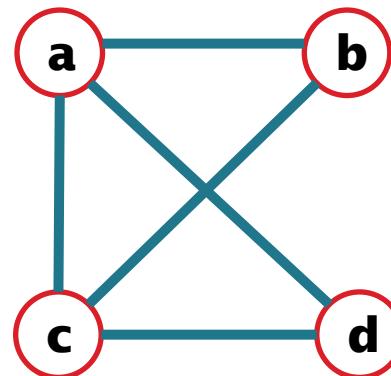
# Izomorfism

**Interpretare:** se pot reprezenta în plan prin același desen

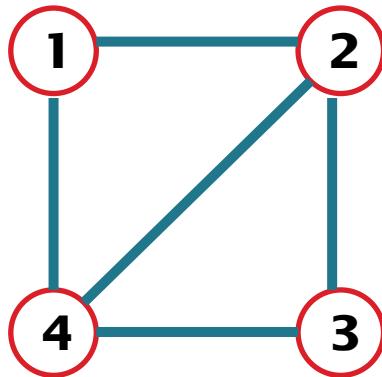
# Izomorfism



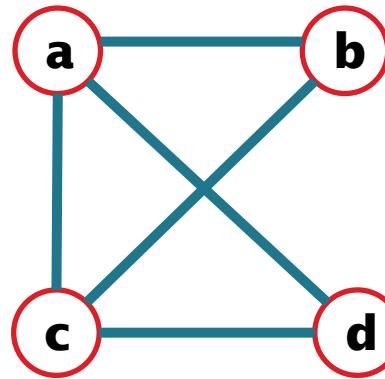
~



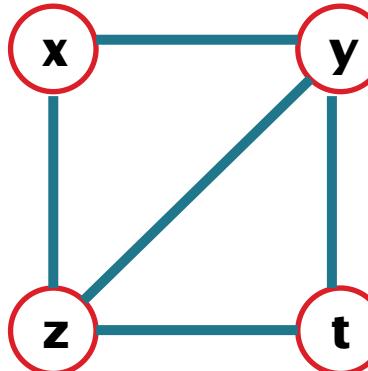
# Izomorfism



~



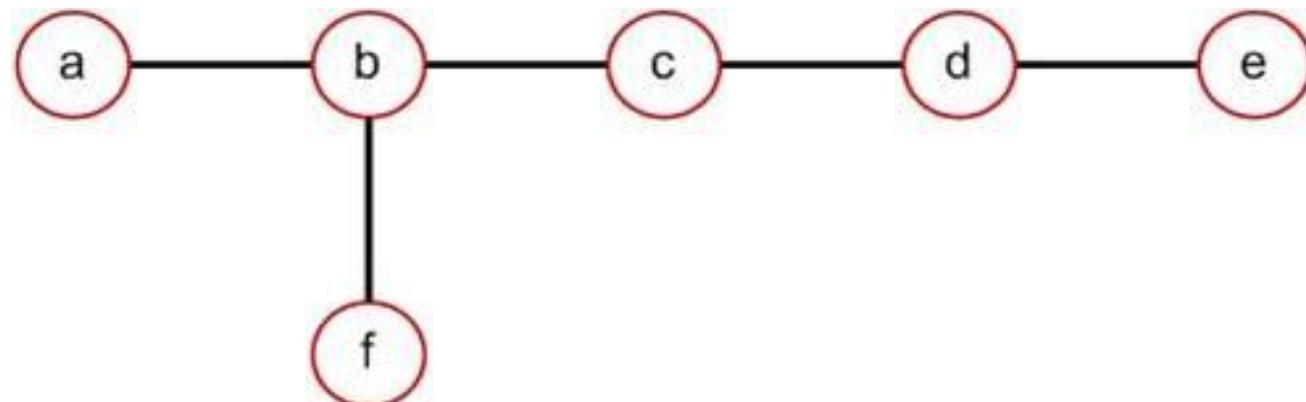
~



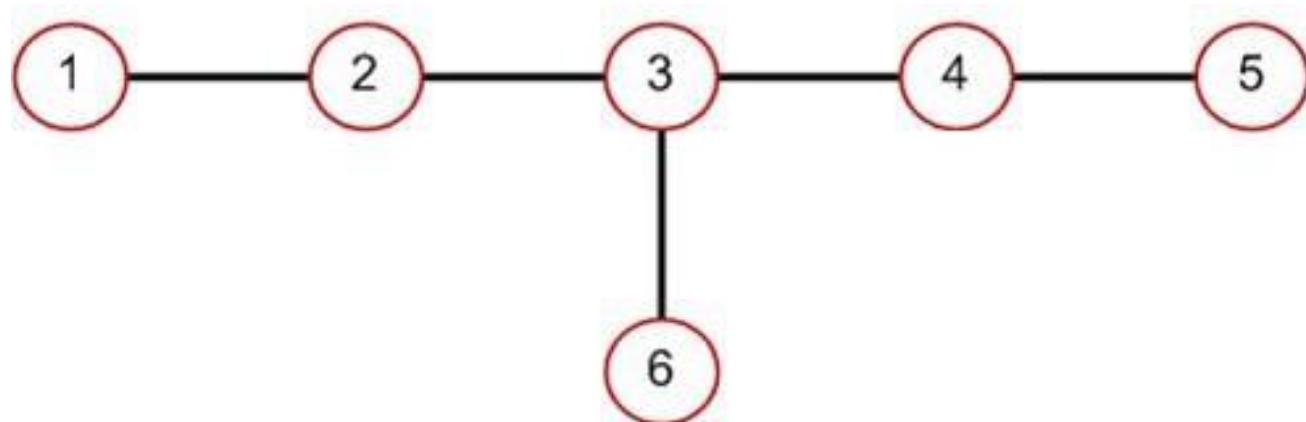
# Izomorfism

- $G_1 \sim G_2 \Rightarrow s(G_1) = s(G_2)$
- $s(G_1) = s(G_2) \not\Rightarrow G_1 \sim G_2 ?$

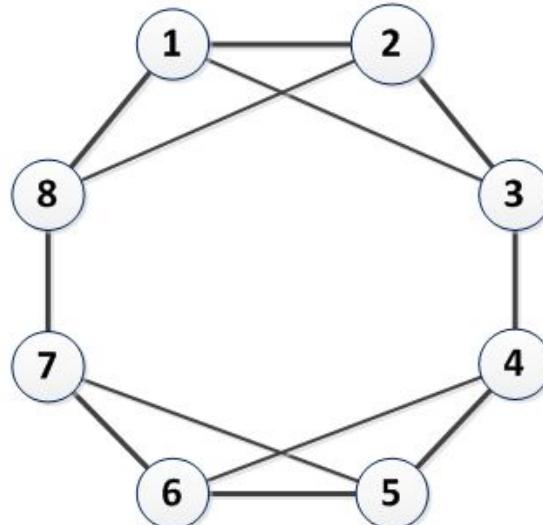
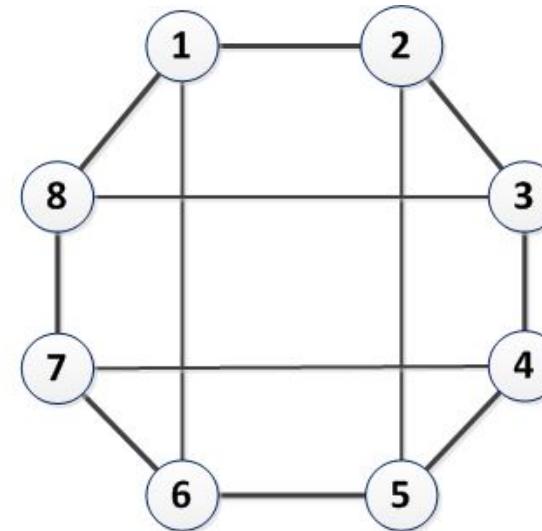
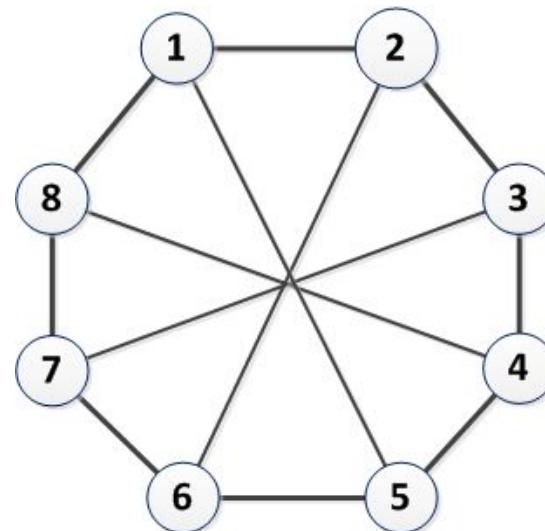
# Izomorfism



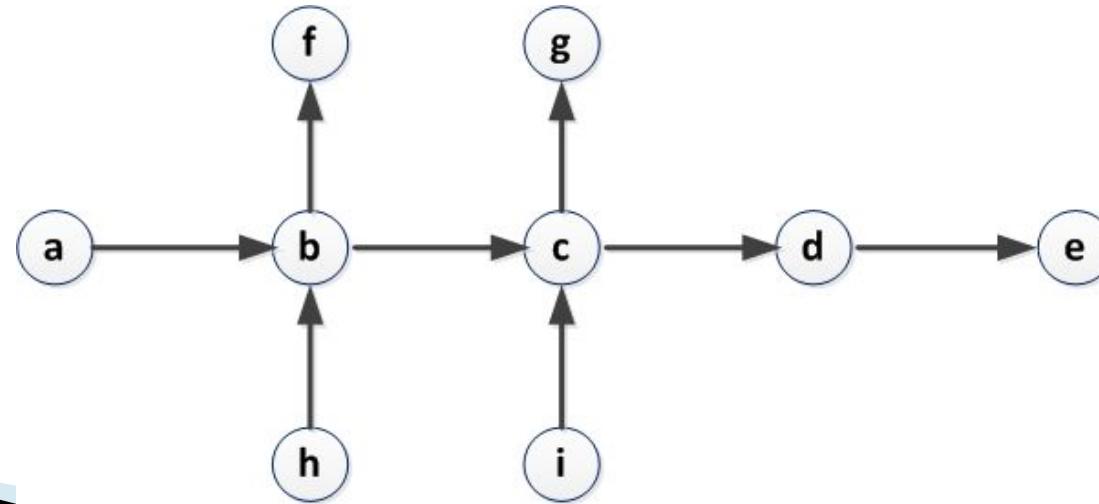
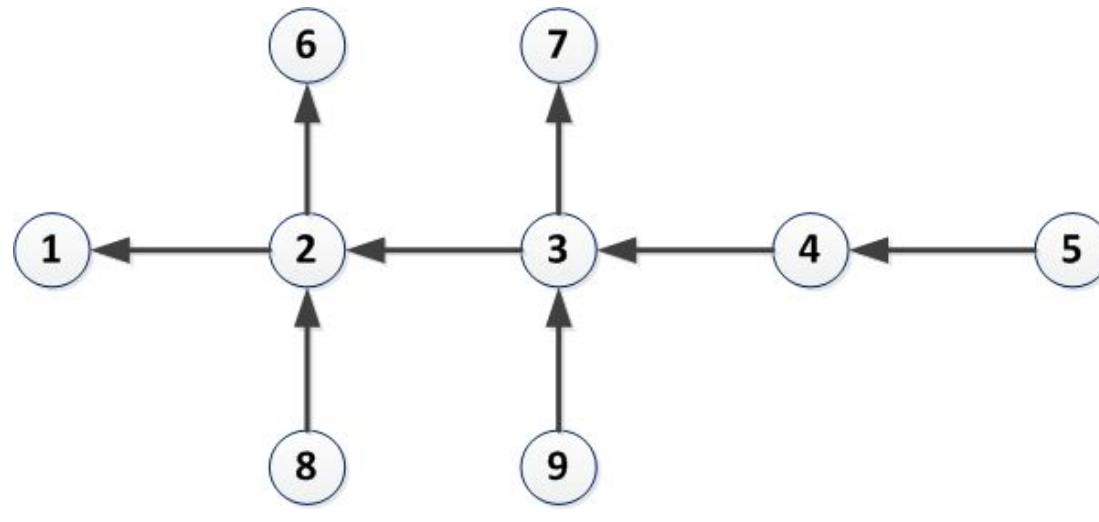
Izomorf  
e?



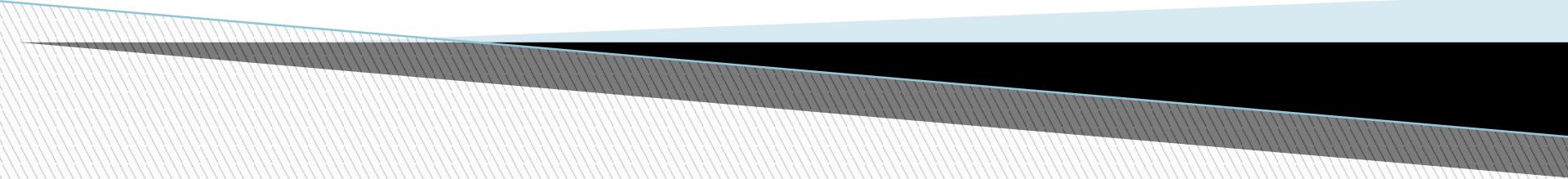
# Care dintre aceste grafuri sunt izomorfe?



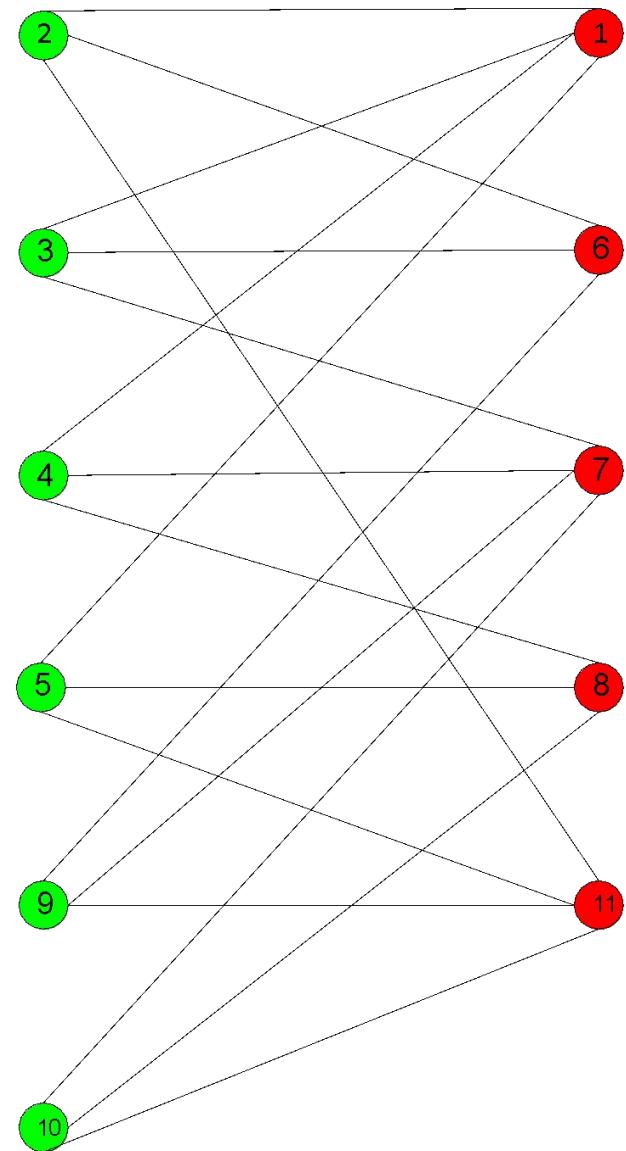
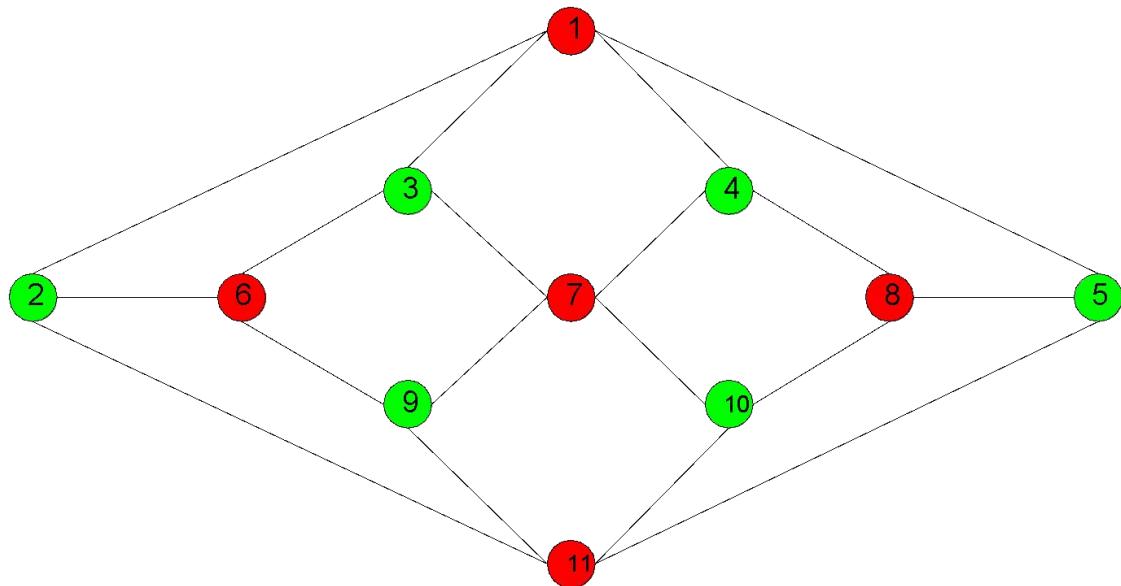
# Sunt aceste grafuri izomorfe?



# **Grafuri standard**



# Graf bipartit



# Graf bipartit

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

$$V = V_1 \cup V_2$$

$$V_1 \cap V_2 = \emptyset$$

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

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

# Graf bipartit

## Observații

e  $\square$   $G = (V, E)$  **bipartit**  $\Leftrightarrow$

există o colorare a vârfurilor cu două culori:

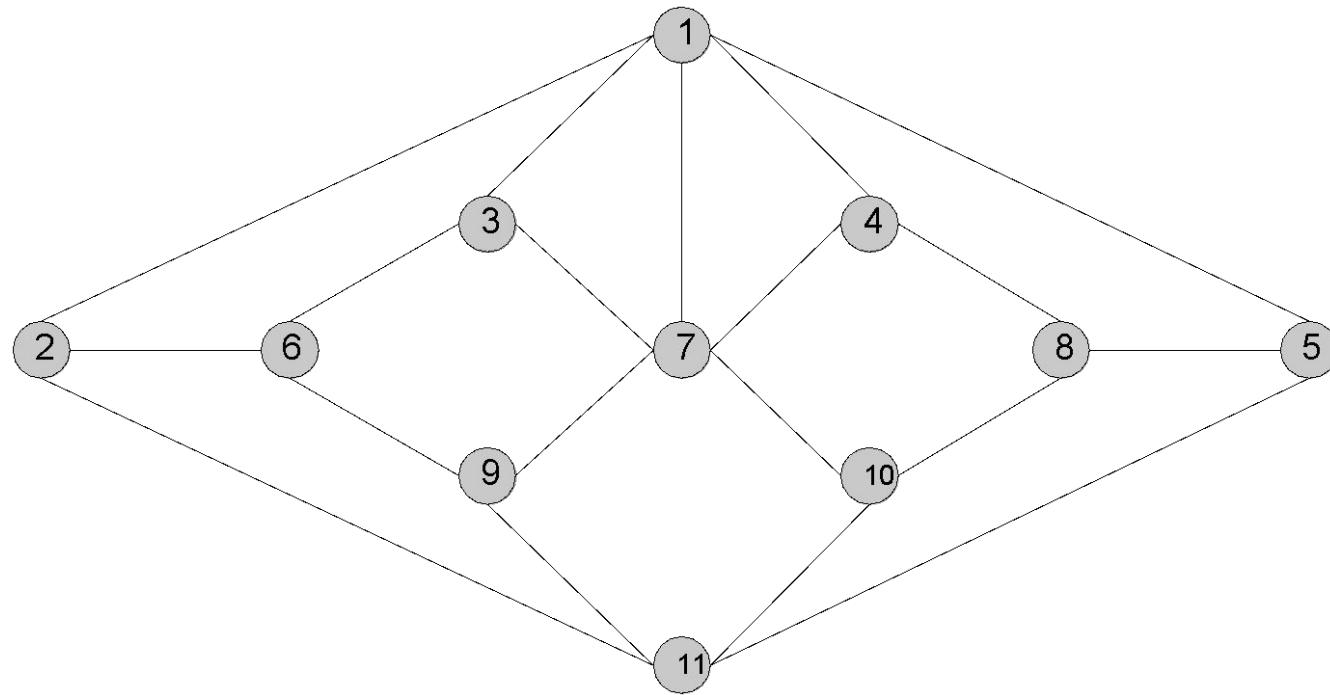
$$c : V \rightarrow \{1, 2\}$$

astfel încât pentru orice muchie  $e=xy \in E$  avem

$$c(x) \neq c(y)$$

**(bicolorare)**

# Graf bipartit



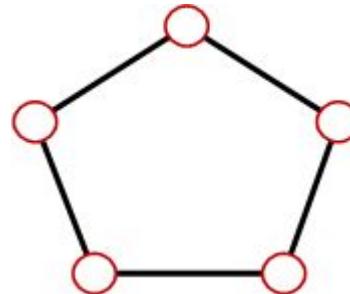
**nu este  
bipartit**

# Grafuri standard

□  $P_n$  – lanț elementar

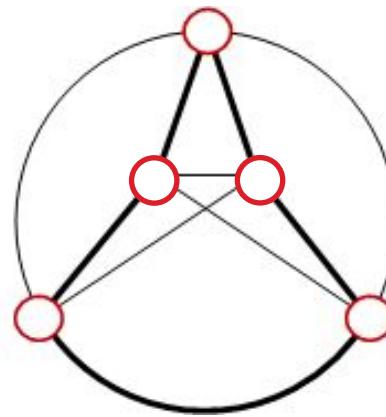
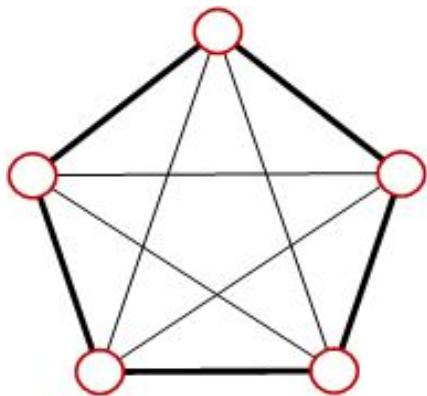


□  $C_n$  – ciclu elementar



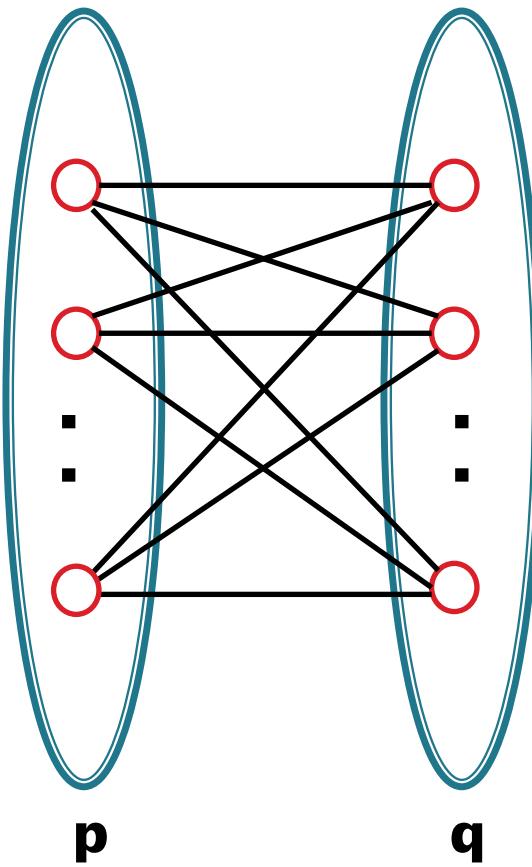
# Grafuri standard

## □ **$K_n$ – graf complet**



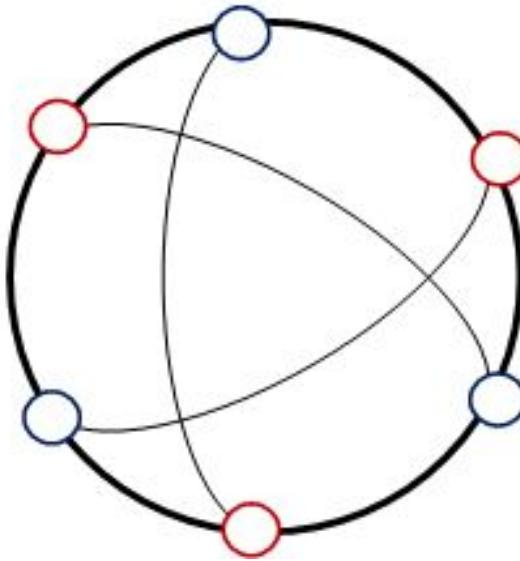
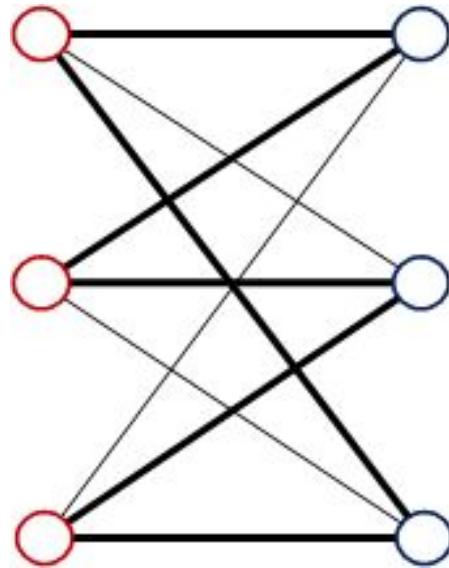
# Grafuri standard

□  **$K_{p,q}$  – graf bipartit complet**



# Grafuri standard

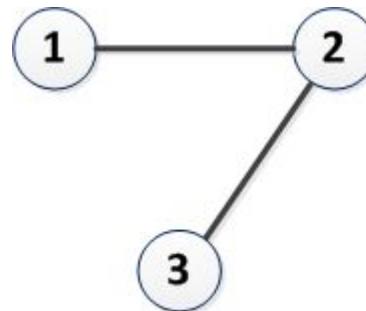
□  $K_{3,3}$



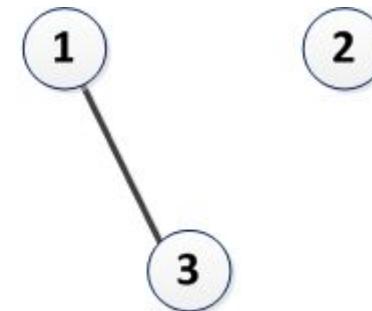
# Graful complementar al unui graf neorientat

□  $G = (V, E)$  graf neorientat

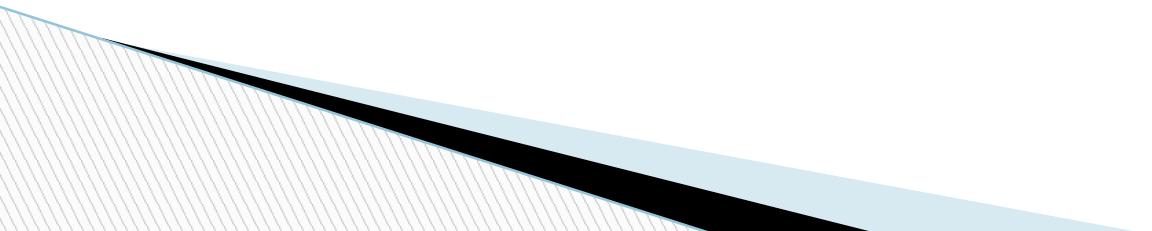
$\overline{G} = (V, E)$  **graful complementar** al lui  
 $G$



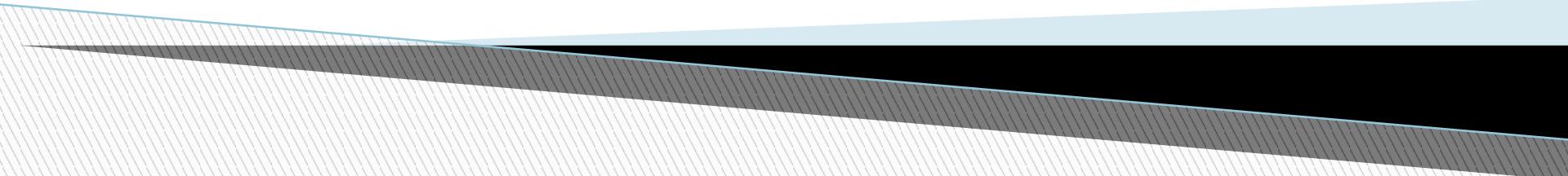
$G$



$\overline{G}$



# **Construcția de grafuri cu secvență gradelor dată**



# Secvențe de grade



**Dată o formulă chimică, există un compus chimic cu această formulă? Dar unul aciclic? Ce structuri poate avea un astfel de compus?**

- $C_m H_n$  - poate exista moleculă **aciclică** cu această formulă?

# Secvențe de grade



**Din studii empirice, chestionare, analize ⇒ 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 ⇒ 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 construască 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 secvență 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 și suficiente

... totusi puteam crea un multigraf

# Construcția de grafuri cu secvență gradelor dată



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

- **începem construcția de la vârful cu gradul cel mai mare**
- **îi alegem ca vecini vârfurile cu gradele cele mai mari**

# Construcția de grafuri cu secvență gradelor dată

## Exemplu algoritm

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

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

- începem construcția de la vârful cu gradul cel mai mare
- îi alegem ca vecini vârfurile cu gradele cele mai mari
- 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 algoritm de construcție a unui graf  $G$  cu  $s(G) = s_0$**

- începem construcția de la vârful cu gradul cel mai mare
- îi alegem ca vecini vârfurile cu gradele cele mai mari
- 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 secvență gradelor dată

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

- începem construcția de la vârful cu gradul cel mai mare
- îi alegem ca vecini vârfurile cu gradele cele mai mari
- 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

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

**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

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

**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

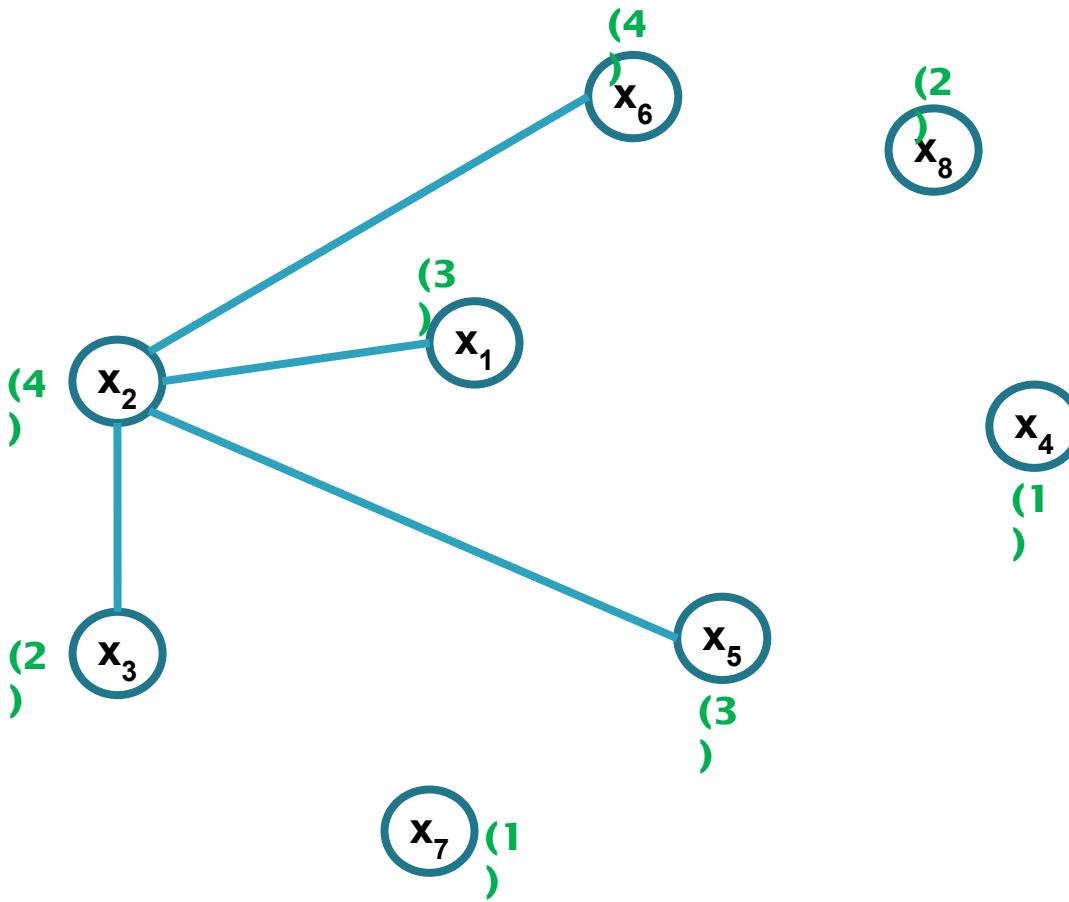
Pasul 1.

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

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

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



- **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  $x_6 \ x_1 \ x_5 \ x_3 \ x_8 \ x_4 \ x_7$

# Exemplu algoritm Havel-Hakimi

## Pasul 1.

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

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



□ **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  $x_6 \ x_1 \ x_5 \ x_3 \ x_8 \ x_4 \ x_7$

**Secvența rămasă ordonată descrescător:**

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

etichete vârfuri  $x_6 \ x_1 \ x_5 \ x_8 \ x_3 \ x_4 \ x_7$

# Exemplu algoritm Havel-Hakimi

## Pasul 2.

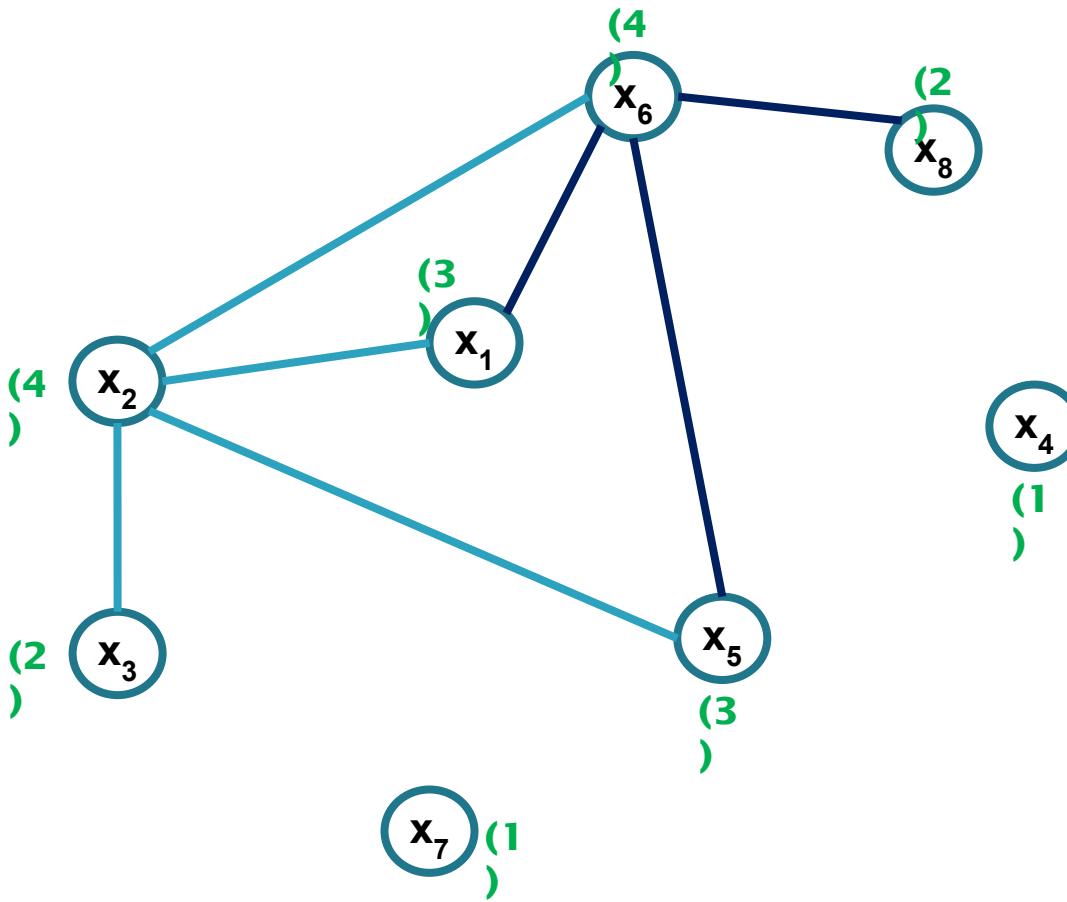
$s'_0 = \{ 3, 2, 2, 2, 1, 1, 1 \}$   
etichete vârfuri  $x_6 \ x_1 \ x_5 \ x_8 \ x_3 \ x_4 \ x_7$   


- Muchii construite:  $x_6x_1, x_6x_5, x_6x_8$
- Secvența rămasă:

$s''_0 = \{ 1, 1, 1, 1, 1, 1 \}$   
etichete vârfuri  $x_1 \ x_5 \ x_8 \ x_3 \ x_4 \ x_7$

(este ordonată descrescător)

# Exemplu algoritm Havel-Hakimi



# Exemplu algoritm Havel-Hakimi

## Pasul 3.

$s''_0 = \{ 1, 1, 1, 1, 1, 1 \}$   
etichete vârfuri  $x_1, x_5, x_8, x_3, x_4, x_7$

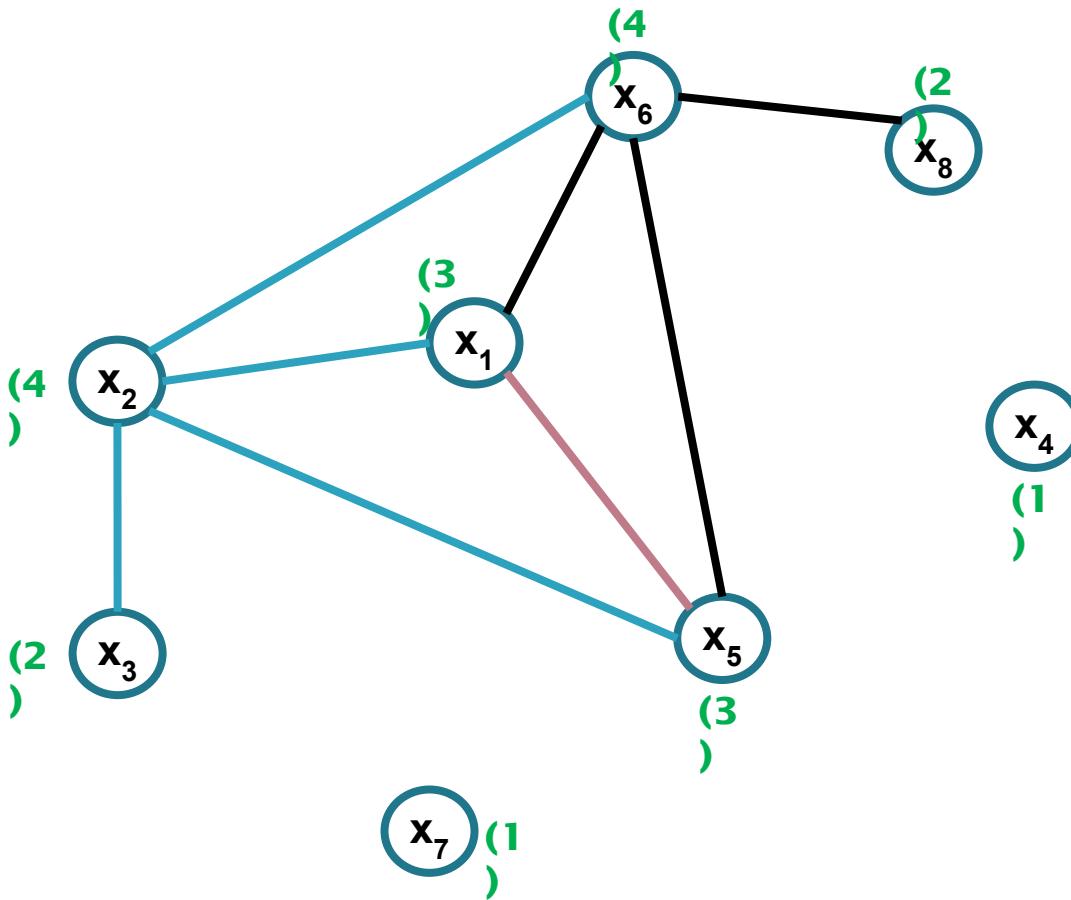
- Muchii construite:  $x_1x_5$
- Secvența rămasă:

$s'''_0 = \{ 0, 1, 1, 1, 1 \}$   
etichete vârfuri  $x_5, x_8, x_3, x_4, x_7$

Secvența rămasă ordonată descrescător:

$s'''_0 = \{ 1, 1, 1, 1, 0 \}$   
etichete vârfuri  $x_7, x_3, x_4, x_8, x_5$

# Exemplu algoritm Havel-Hakimi



# Exemplu algoritm Havel-Hakimi

## Pasul 4.

$s'''_0 = \{ 1, 1, 1, 1, 0 \}$   
etichete vârfuri  $x_7 \ x_3 \ x_4 \ x_8 \ x_5$

□ **Muchii construite:**  $x_7x_3$

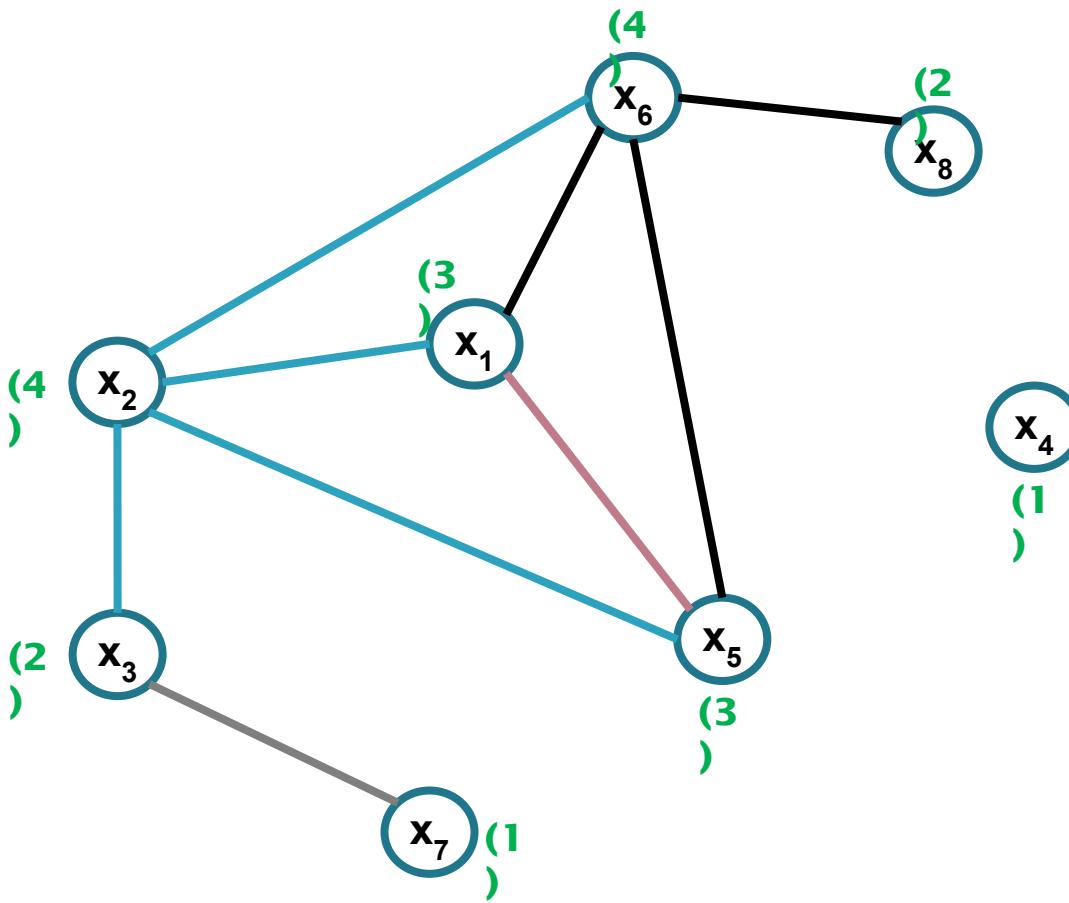
□ **Secvența rămasă:**

$s^{iv}_0 = \{ 0, 1, 1, 0 \}$   
etichete vârfuri  $x_3 \ x_4 \ x_8 \ x_5$

**Secvența rămasă ordonată descrescător:**

$s'''_0 = \{ 1, 1, 0, 0 \}$   
etichete vârfuri  $x_4 \ x_8 \ x_3 \ x_5$

# Exemplu algoritm Havel-Hakimi



# Exemplu algoritm Havel-Hakimi

Pasul 5.

$s^{iv}_0 = \{$   
**etichete vârfuri**

1, 1, 0, 0}  
 $x_4 \quad x_8 \quad x_3 \quad x_5$

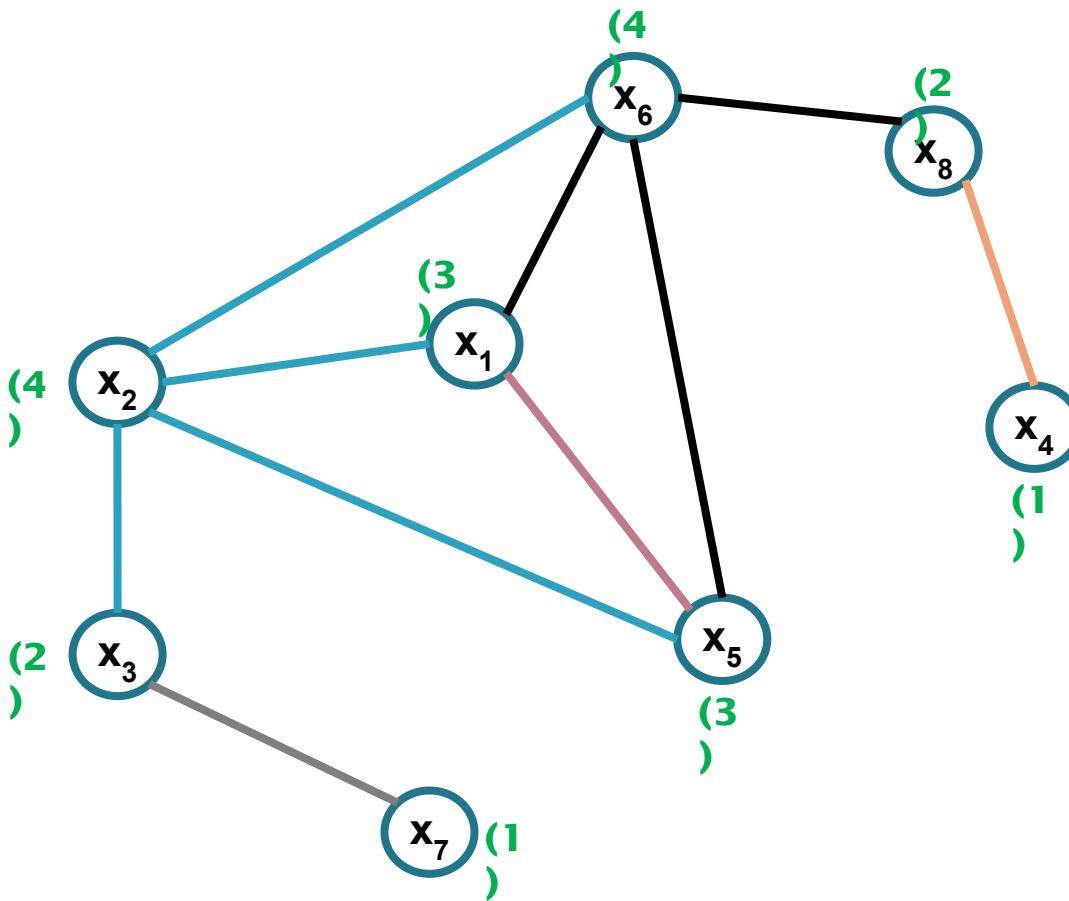
- **Muchii construite:**  $x_4x_8$
- **Secvența rămasă:**

$s^{iv}_0 = \{$   
**etichete vârfuri**

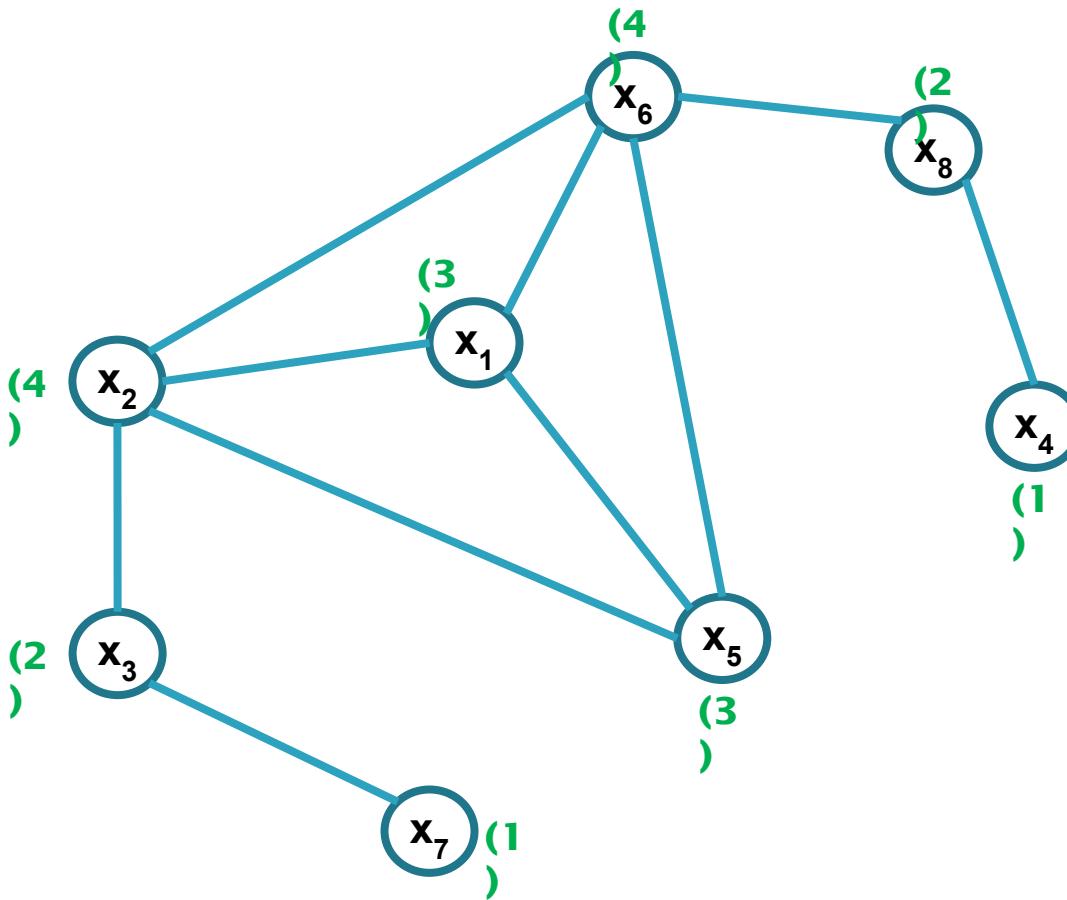
0, 0, 0}  
 $x_8 \quad x_3 \quad x_5$

**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 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  $d_{i_1}, \dots, d_{i_{d_k}}$  **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  $d_{i_1}, \dots, d_{i_{d_k}}$  **cele mai mari  $d_k$  numere** din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{d_k}\}$  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  $d_{i_1}, \dots, d_{i_{d_k}}$  **cele mai mari  $d_k$  numere** din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{d_k}\}$  execută:
    - adaugă la G muchia  $x_k x_j$
    - 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  $d_{i_1}, \dots, d_{i_{d_k}}$  **cele mai mari  $d_k$  numere** din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{d_k}\}$  execută:
    - adaugă la G muchia  $x_k x_j$
    - 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

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  $d_{i_1}, \dots, d_{i_{d_k}}$  **cele mai mari  $d_k$  numere** din  $s_0$
  - pentru  $j \in \{i_1, \dots, i_{d_k}\}$  execută:
    - adaugă la G muchia  $x_k x_j$
    - 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 - 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).**

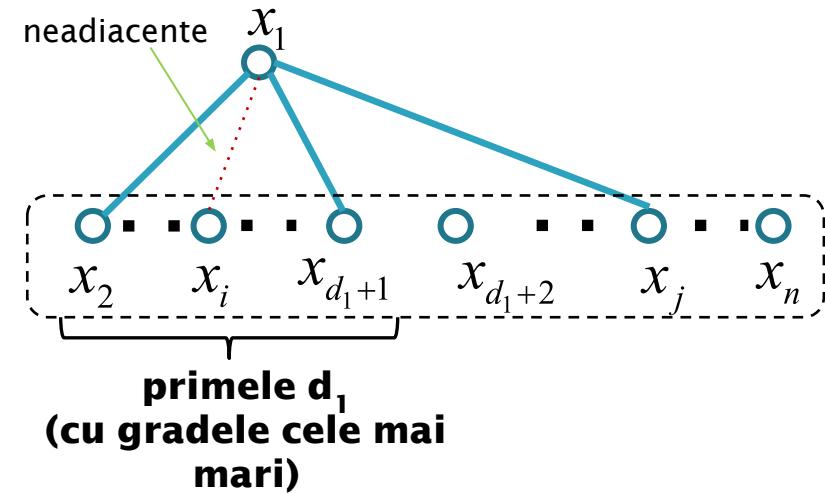
**Observație:** Secvența  $s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$  se obține din  $s_0$  **eliminând primul element (adică  $d_1$ ) și scăzând 1 din primele  $d_1$  elemente rămase** – acestea au indicii 2, 3, ...,  $d_1 + 1$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi – Demonstrație

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

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



# Algoritm Havel-Hakimi - Corectitudine

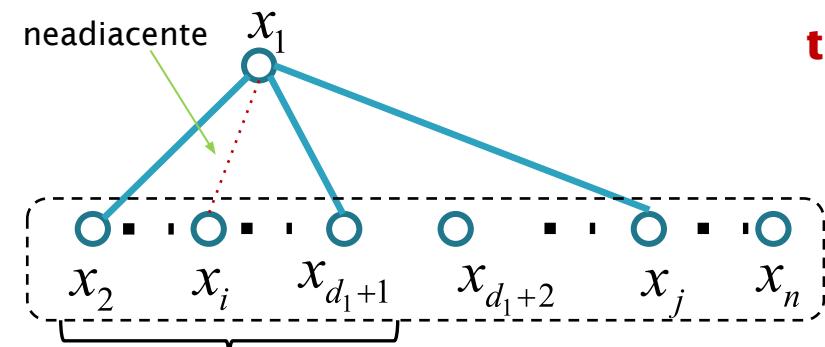
## Teorema Havel-Hakimi – Demonstrație

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

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

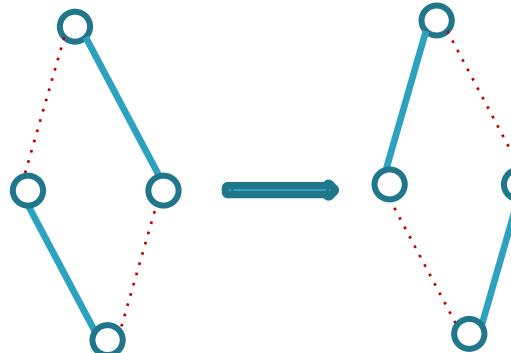
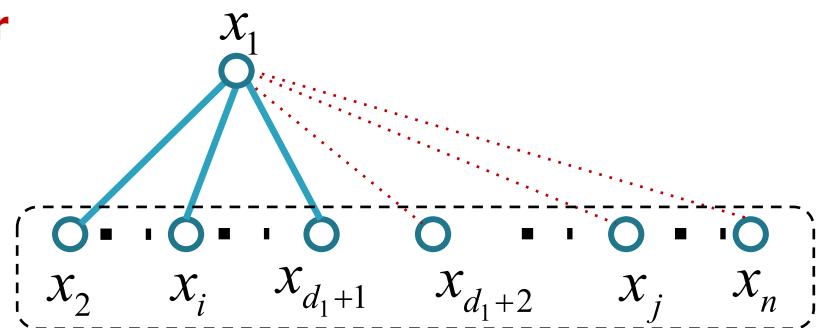
$$G^*, s(G^*) = s_0$$

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



primele  $d_1$   
(cu gradele cele mai mari)

transformar  
e t  
pe pătrat



# Algoritm Havel-Hakimi - Corectitudine

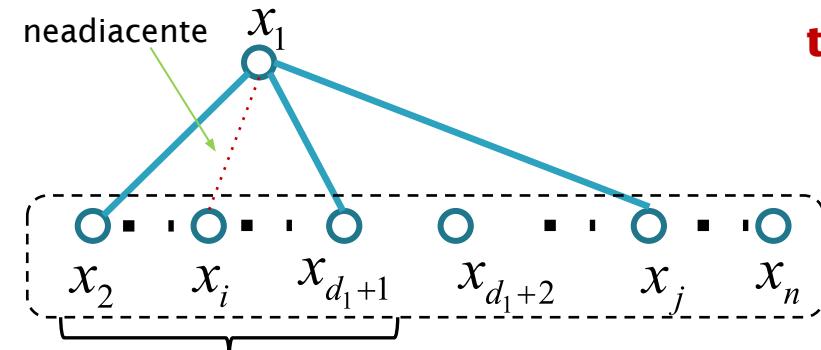
## Teorema Havel-Hakimi – Demonstrație

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

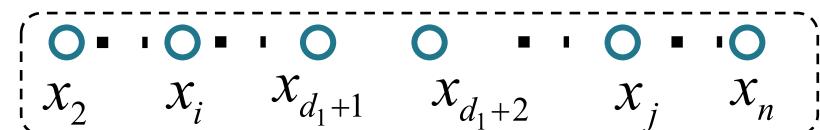
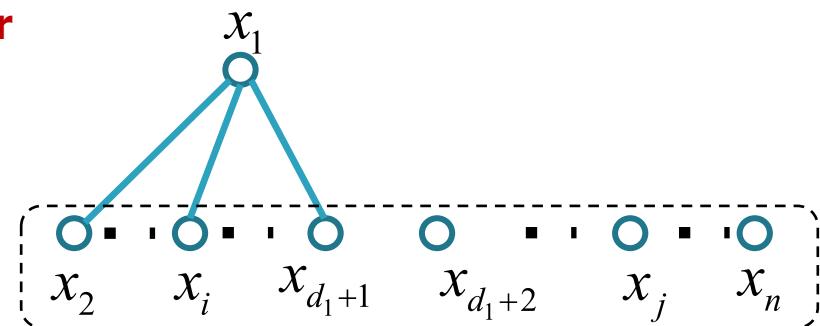
$$G, s(G) = s_0$$

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

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



transformar  
e t  
pe pătrat



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

# **Algoritm Havel-Hakimi - Corectitudine**

## **Teorema Havel-Hakimi – Demonstrație**

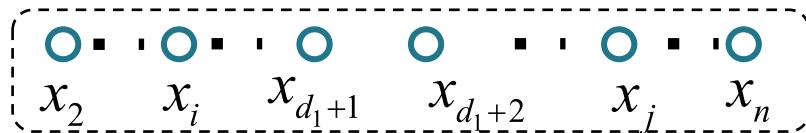
$$s_0 = \{d_1 \geq \dots \geq d_n\} \iff s'_0 = \{d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n\}$$

# Algoritm Havel-Hakimi - Corectitudine

## Teorema Havel-Hakimi – Demonstrație

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

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

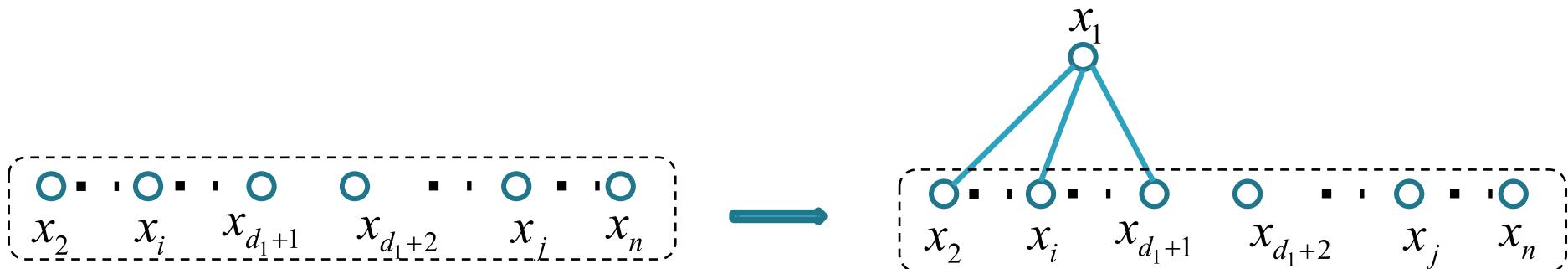
## Teorema Havel-Hakimi – Demonstrație

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

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

$$G: V(G) = V(G') \cup \{x_1\}$$

$$E(G) = E(G') \cup \{x_1 x_2, \dots, x_1 x_{d_1+1}\}$$



adăugăm un vârf

pe care îl unim cu

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 cu mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat. Fie  $s_0^{(i)}$  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 cu mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat. Fie  $s_0^{(i)}$  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 cu mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat. Fie  $s_0^{(i)}$  secvența obținută din  $s_0$  astfel:

- eliminăm elementul  $d_i$
- scădem o unitate din primele  $d_i$  componente în ordine descrescătoare ale 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 cu mai mici sau egale cu  $n-1$  și fie  $i \in \{1, \dots, n\}$  fixat. Fie  $s_0^{(i)}$  secvența obținută din  $s_0$  astfel:

- eliminăm elementul  $d_i$
- scădem o unitate din primele  $d_i$  componente în ordine descrescătoare ale 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 Algoritmului 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 secvență gradelor date

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

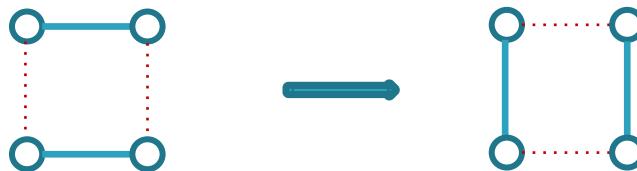


# Construcția de grafuri cu secvență gradelor date

- Cu ajutorul transformării  $t$  pe pătrat putem obține pornind de la un graf  $G$  toate grafurile cu secvență gradelor  $s(G)$  (și mulțimea vârfurilor  $V(G)$ )
- Mai exact, ar 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  $G_1$ .**

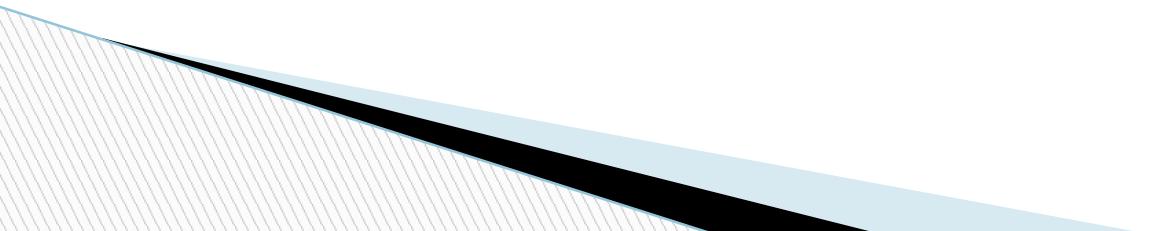


# Construcția de grafuri cu secvență gradelor dată

## Teorema Erdös - 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
- $d_1 + \dots + d_k \leq k(k-1) + \sum_{i=k+1}^n \min\{d_i, k\}, \forall 1 \leq k \leq n$



# Parcurgeri în Grafuri

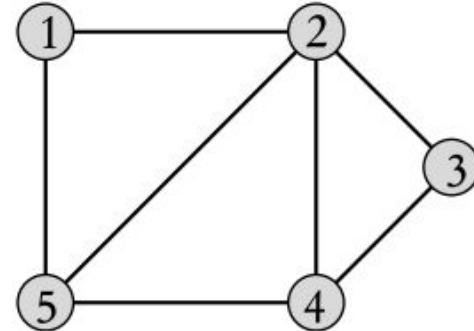
# Reprezentari

Ce reprezentări cunoasteti ?

# Reprezentari

Ce reprezentări cunoașteți ?

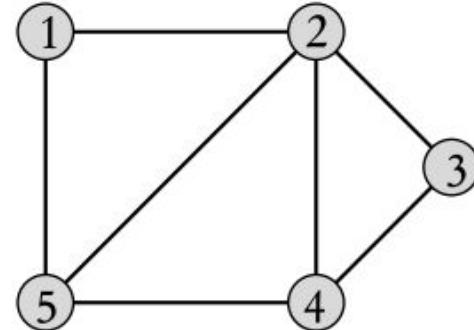
- Reprezentare cu matrice de adiacență
- Liste de adiacență
- Lista de muchii



# Reprezentari

Ce reprezentări cunoașteți ?

- Reprezentare cu matrice de adiacență
  - În graf neorientat matricea nu mai este simetrică

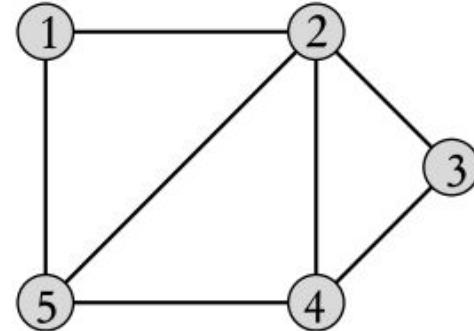
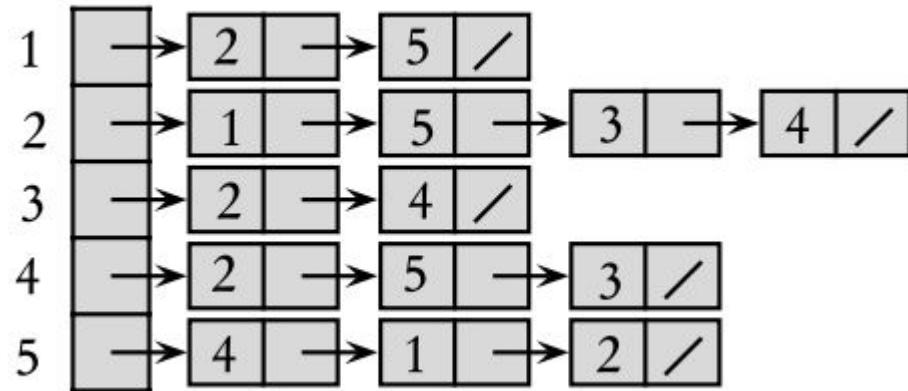


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

# Reprezentări

Ce reprezentări cunoașteți ?

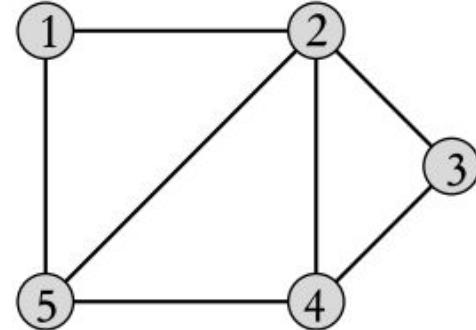
- Liste de adiacență



# Reprezentări

Ce reprezentări cunoașteți ?

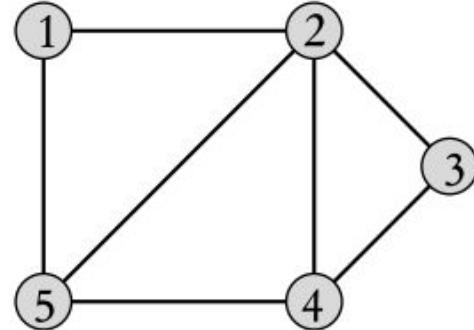
- Lista de muchii
  - $[(1,2), (1,5), (2,5), (2,4), (2,3), (3,4), (4,5)]$



# Reprezentări

Ce reprezentări cunoașteți ?

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



# Reprezentari

- De ce avem mai multe reprezentări ?

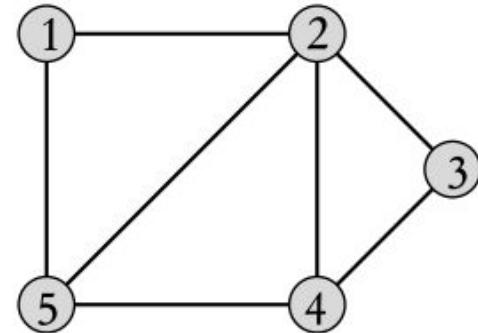
# Reprezentari

Graful Transpus

- 

	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

0	0	1	1	0
0	0	0	0	0
1	0	0	0	1
1	0	0	0	0
0	0	1	0	0

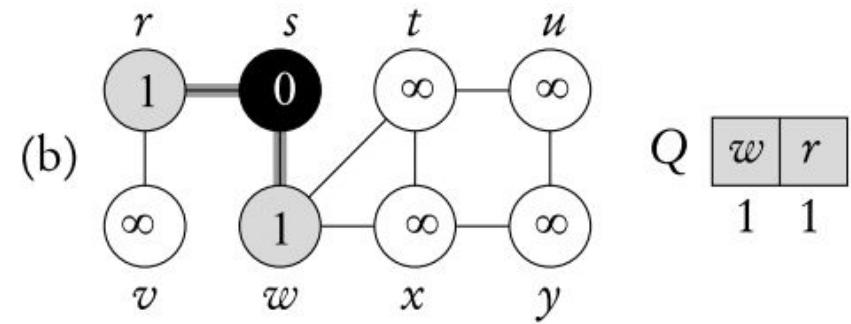
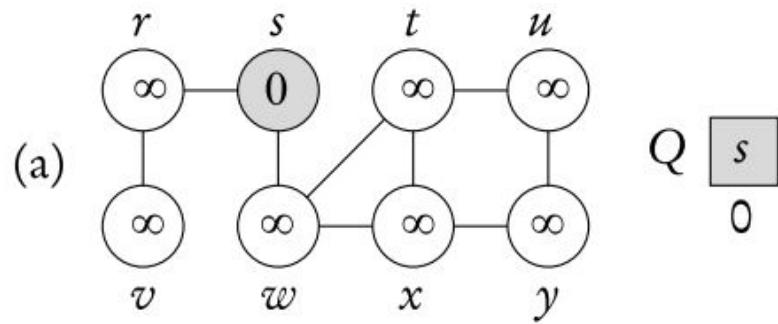


# Parcuregerea în lățime

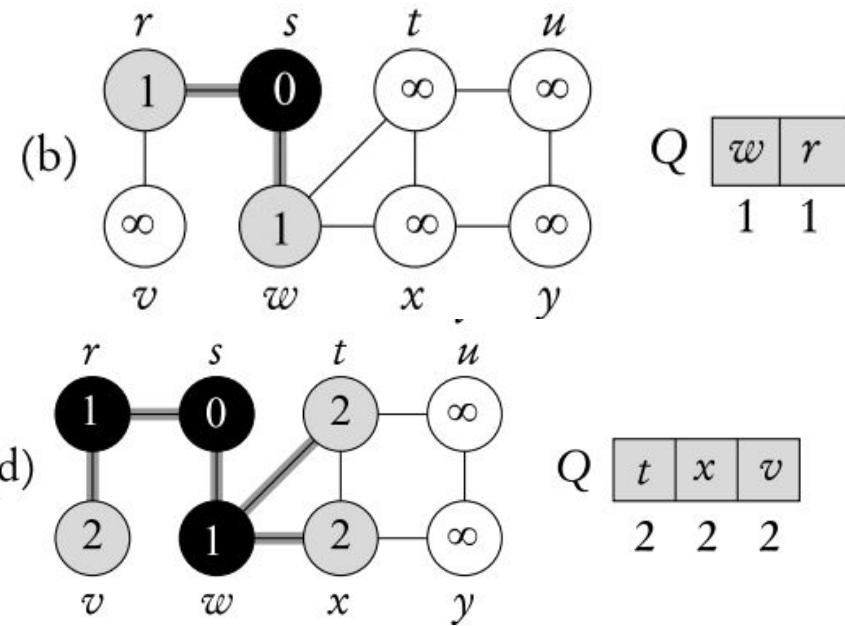
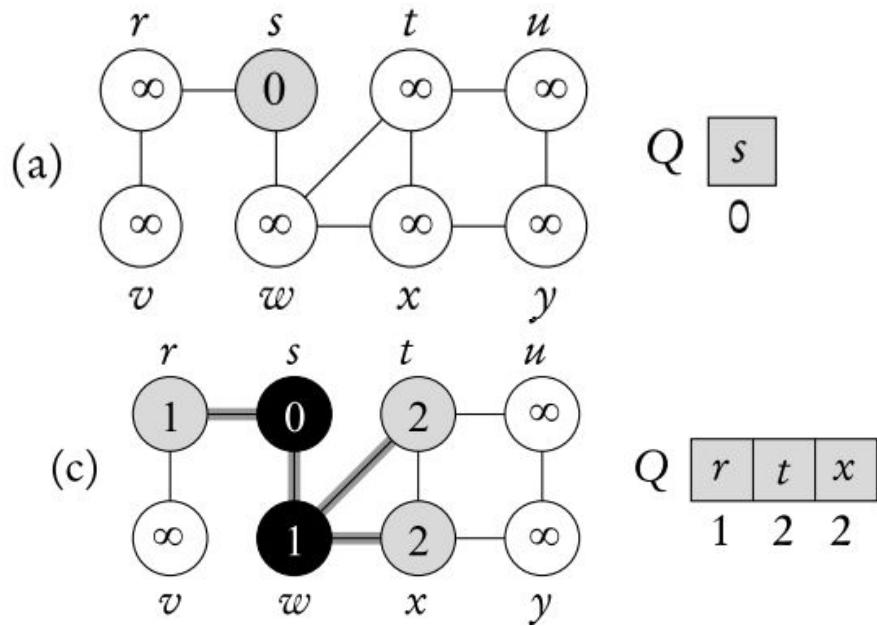
Dat fiind un graf  $G = (V, E)$  și un nod sursă  $s$ , căutarea în lățime explorează sistematic muchiile lui  $G$  pentru a “descoperi” fiecare nod care este accesibil din  $s$ .

De asemenea algoritmul găsește distanța minima de la sursa la toate nodurile din graf.

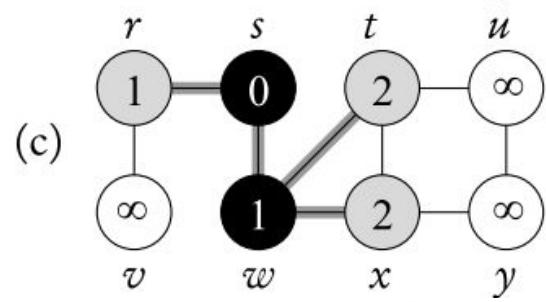
# Parcurea în lățime : Exemplu



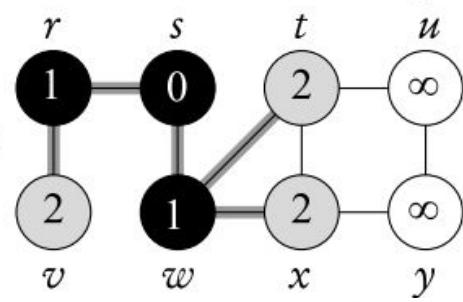
# Parcurea în lățime : Exemplu



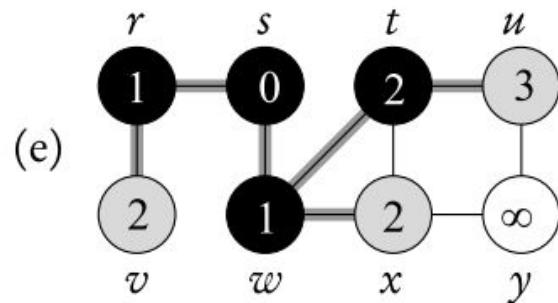
# Parcurea în lățime : Exemplu



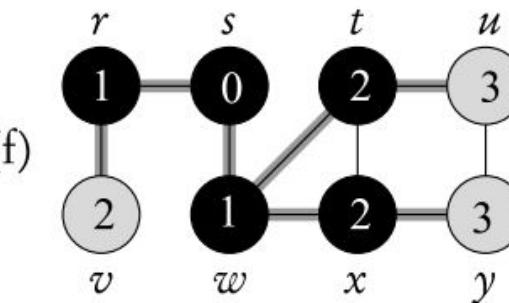
Q	<table border="1"><tr><td><math>r</math></td><td><math>t</math></td><td><math>x</math></td></tr></table>	$r$	$t$	$x$
$r$	$t$	$x$		
	1    2    2			



Q	<table border="1"><tr><td><math>t</math></td><td><math>x</math></td><td><math>v</math></td></tr></table>	$t$	$x$	$v$
$t$	$x$	$v$		
	2    2    2			

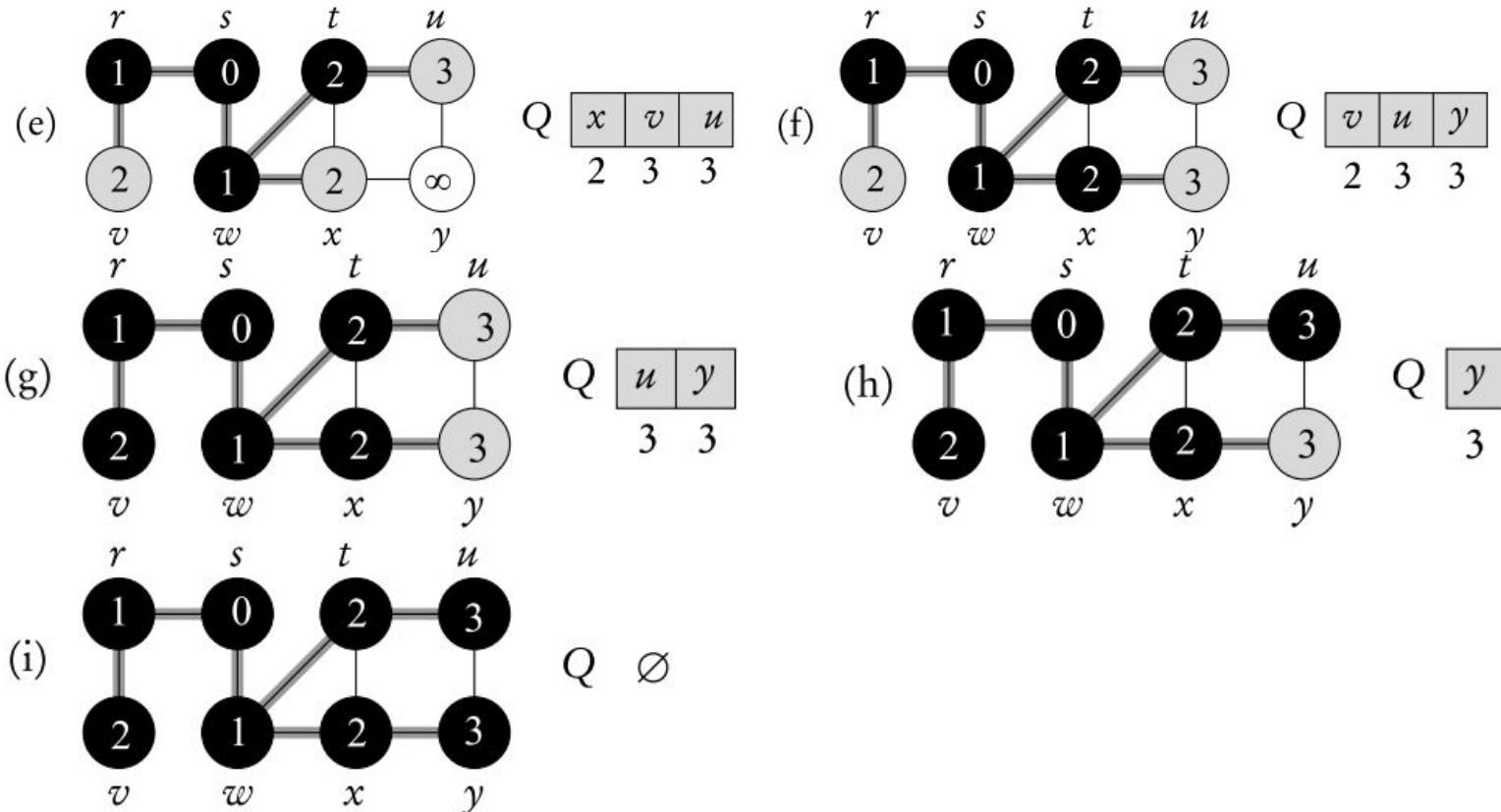


Q	<table border="1"><tr><td><math>x</math></td><td><math>v</math></td><td><math>u</math></td></tr></table>	$x$	$v$	$u$
$x$	$v$	$u$		
	2    3    3			



Q	<table border="1"><tr><td><math>v</math></td><td><math>u</math></td><td><math>y</math></td></tr></table>	$v$	$u$	$y$
$v$	$u$	$y$		
	2    3    3			

# Parcurea în lățime : Exemplu



# Parcurea în lățime: algoritm

Sa scriem impreuna pseudocodul:

# Parcurea în lățime

Algoritm Cormen:

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

??

# Parcurgerea în lățime: complexitate

$O(V+E)$  sau  $O(n+m)$

# Parcurgerea în lățime: complexitate

$O(V+E)$  sau  $O(n+m)$

# Parcurea în lățime: aplicații

Ieșirea din labirint în 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

# Parcursarea în lățime: aplicații

Ieșirea din labirint în 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

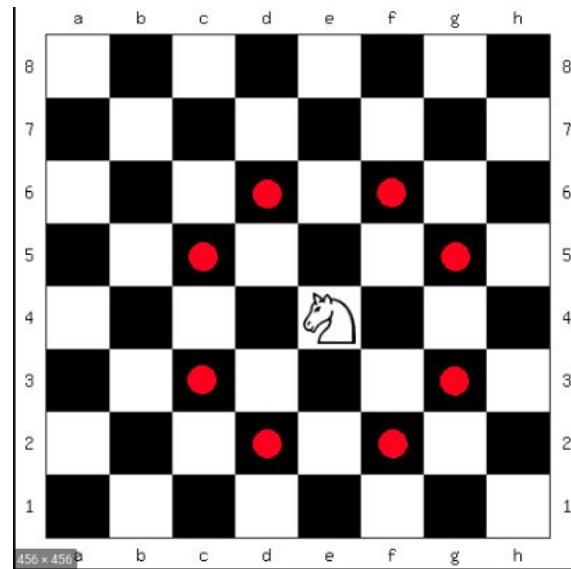
2 3 4 5 6 7 -1 13

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

0 0 0 0 -1 9 10 11

# Parcurea în lățime: aplicații

Drum de lungime minima a calului pe tabla de sah.



# Parcuregerea în adâncime : algoritm

1. Se începe explorarea dintr-un nod nevizitat
2. Se cauta un vecin nevizitat care devine noul nod curent . Cat timp noul curent are un vecin nevizitat repetam pasul 2 -> intrăm în adâncime.
3. Cand nodul curent nu are nici un vecin nevizitat, ne întoarcem la strămoșii lui (tatăl, bunicul... ) pana găsim un nod care are vecini vizitati și reluăm pasul 2.
4. Dacă există noduri nevizitate reluăm pasul 1...

Cand se intampla cazul 4?

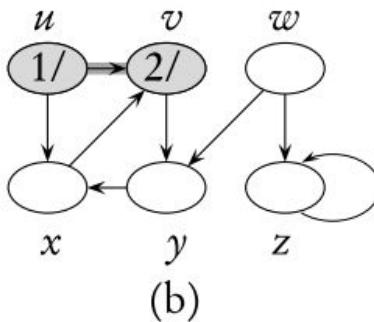
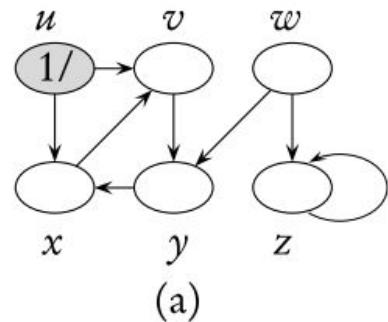
# Parcursarea în adâncime: cu timpi de intrare și ieșire

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

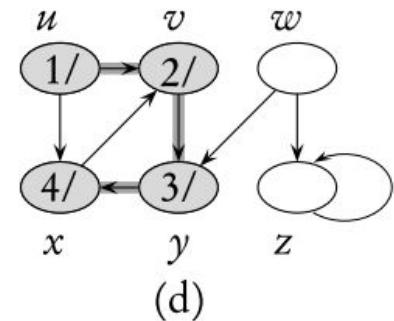
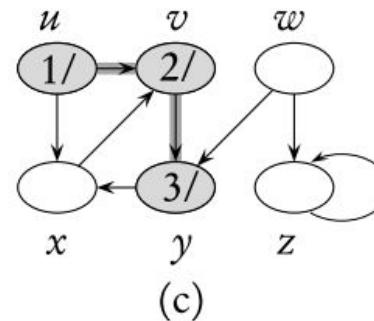
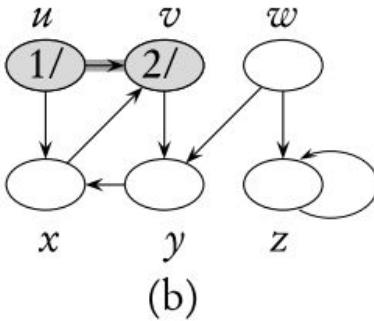
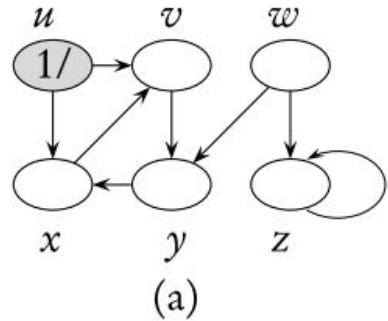
1. De fiecare dată cand ajungem într-un nod sau cand terminam de vizitat toți vecinii incrementam un contor și ținem minte aceste informații ...

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

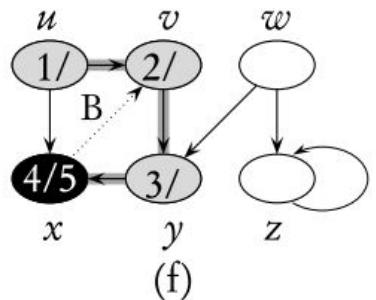
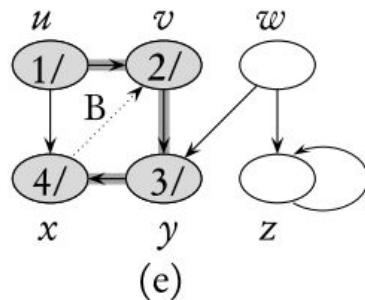
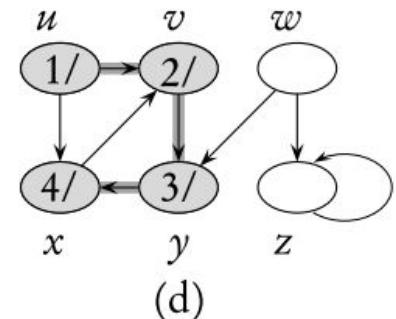
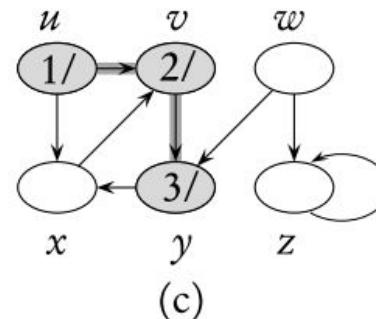
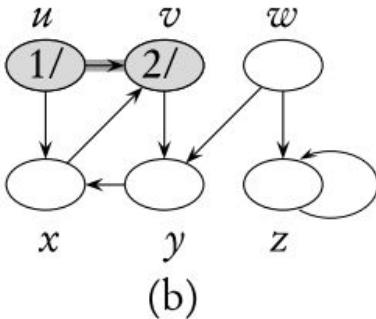
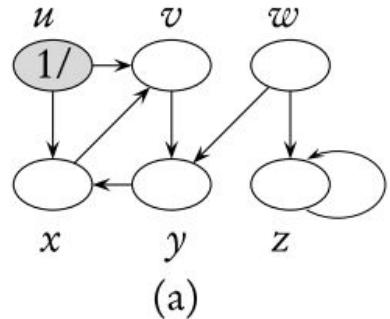
# Parcurea în adâncime: exemplu



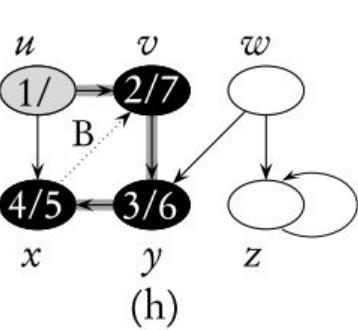
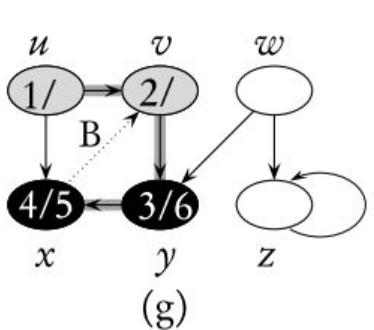
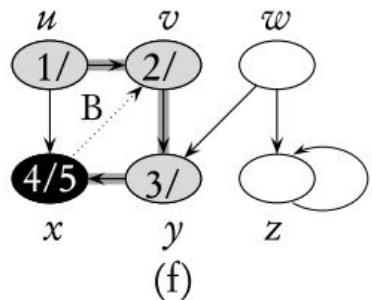
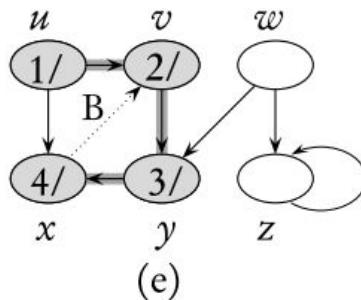
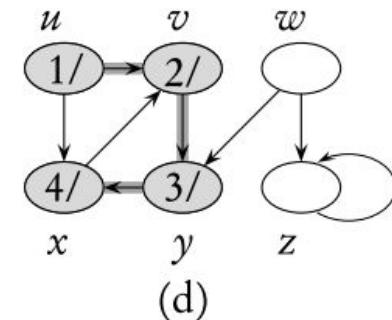
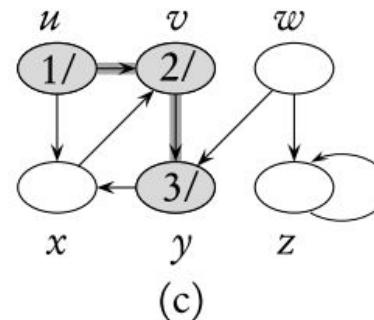
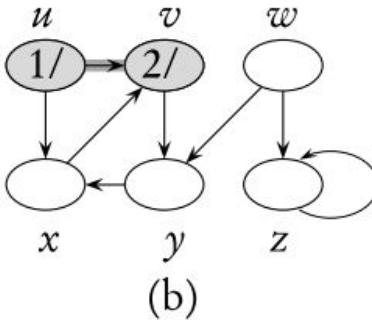
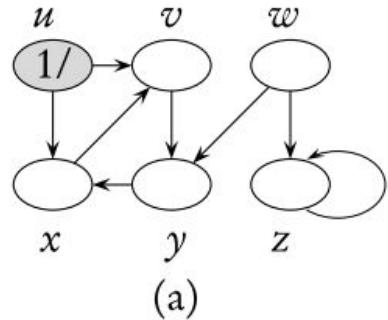
# Parcurea în adâncime: exemplu



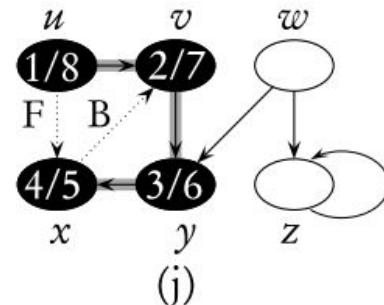
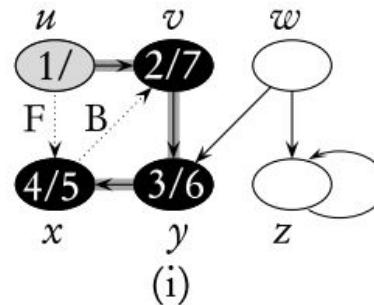
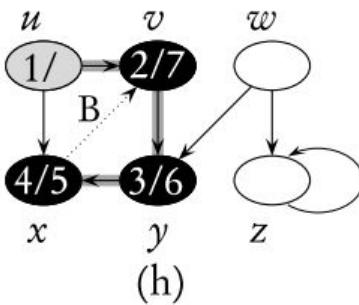
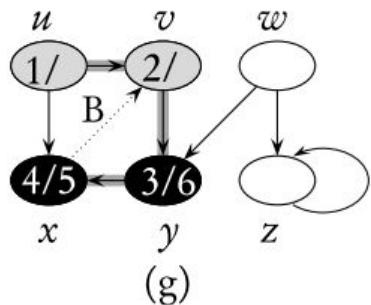
# Parcurea în adâncime: exemplu



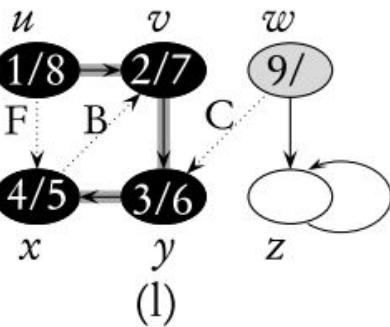
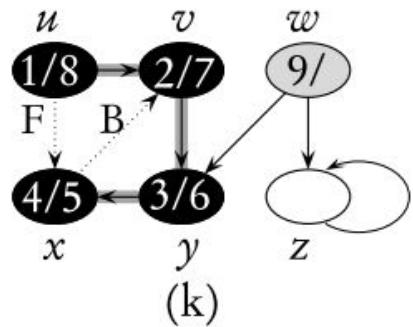
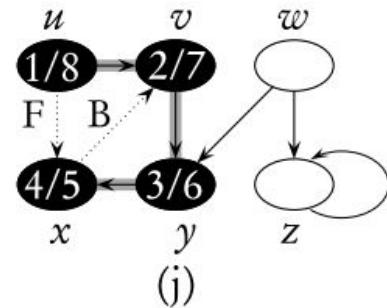
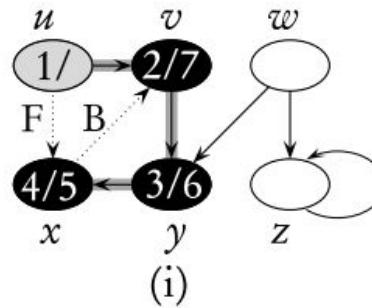
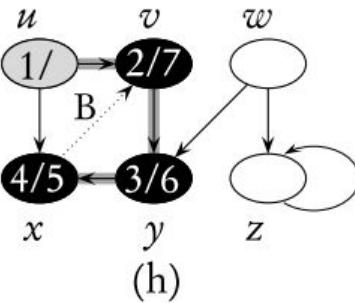
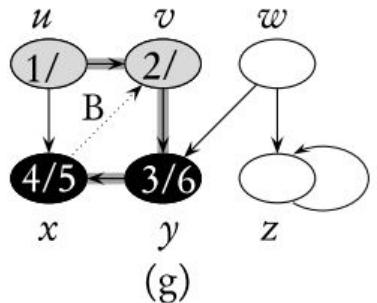
# Parcurea în adâncime: exemplu



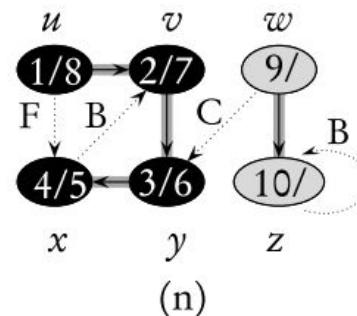
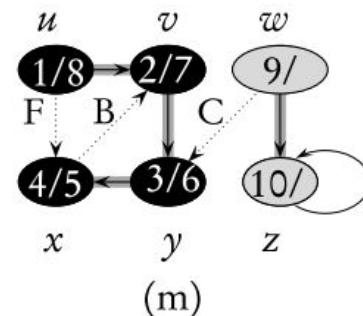
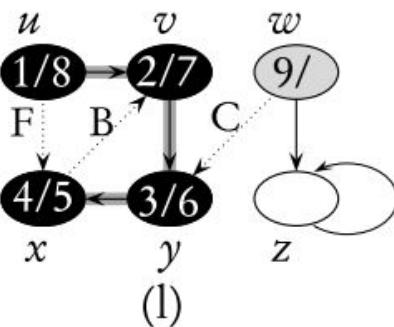
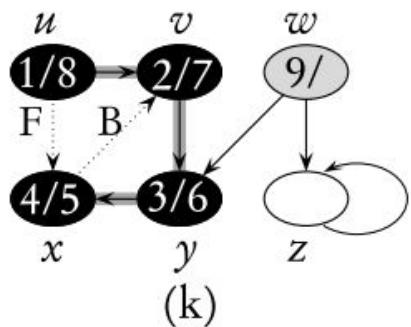
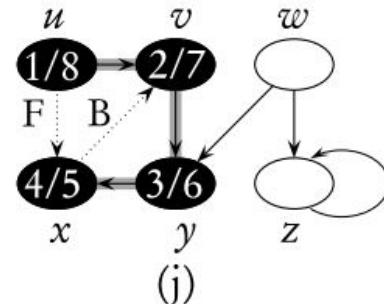
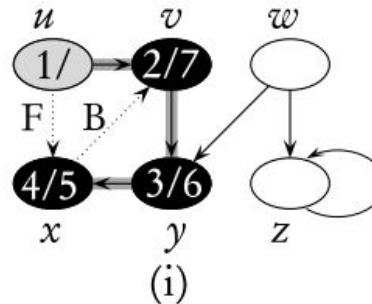
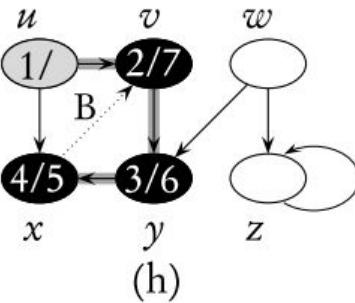
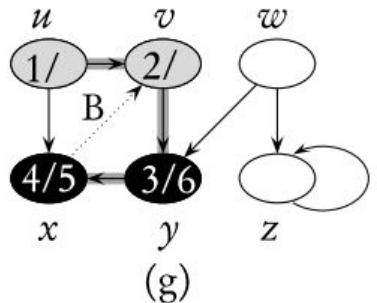
# Parcurea în adâncime: exemplu



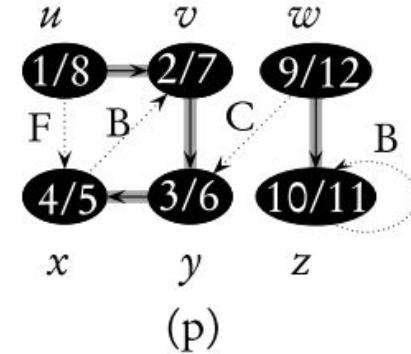
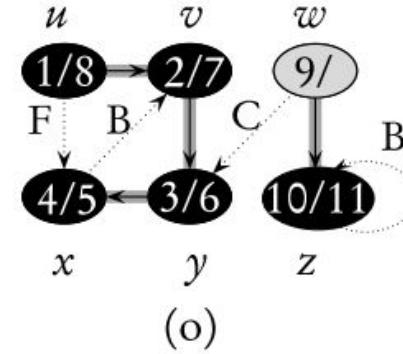
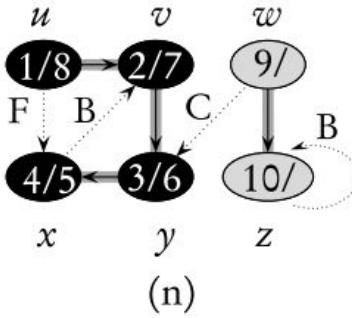
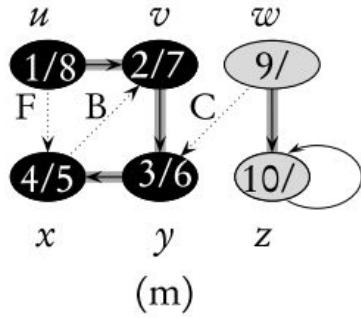
# Parcurea în adâncime: exemplu



# Parcurea în adâncime: exemplu



# Parcurea în adâncime: exemplu



# Parcurea în adâncime: algoritm

Să scriem împreună pseudocodul:

# Parcurea în adâncime: algoritm Cormen

CA( $G$ )

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

# Parcursarea în adâncime: Teorema parantezelor

**Teorema parantezelor:** În orice căutare în adâncime a unui graf (orientat sau neorientat)  $G = (V, E)$ , pentru orice 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 descendant al lui  $v$  în arborele de adâncime, sau
- intervalul  $[d[v], f[v]]$  este conținut, în întregime, în intervalul  $[d[u], f[u]]$ , iar  $v$  este un descendant al lui  $u$  în arborele de adâncime.

# Parcursarea în adâncime: 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. 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  $[d[u], f[u]]$ . În celălalt subcaz  $f[u] < d[v]$  și inegalitatea (23.1) implică faptul că intervalele  $[d[u], f[u]]$  și  $[d[v], f[v]]$  sunt disjuncte.

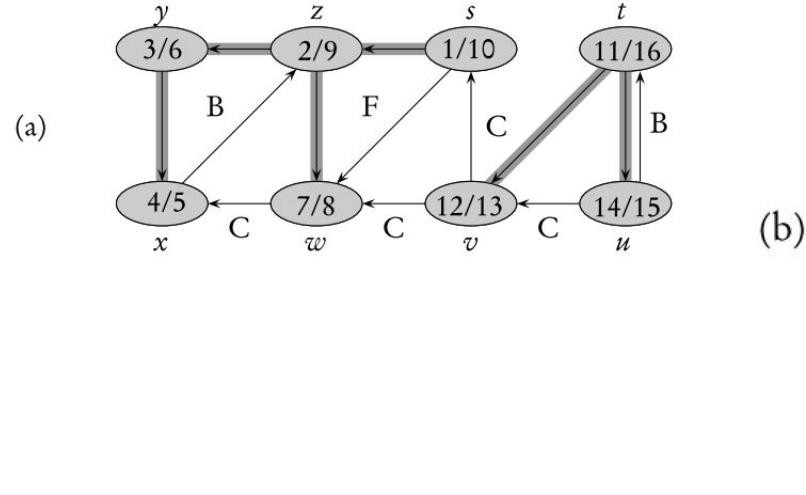
Cazul în care  $d[v] < d[u]$  este similar, inversând rolurile lui u și v în argumentația de mai sus.

# Parcurea în adâncime: Teorema parantezelor

Corolarul 23.7 (Interclasarea intervalelor descendenților)

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

# Parcurea în adâncime: proprietăți



(b)

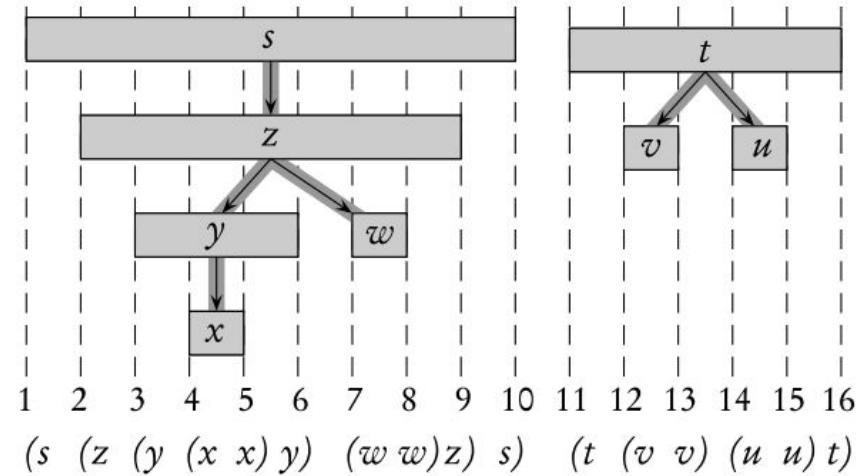


Diagrama b exemplifica foarte bine teorema anterioara.

# Parcurea în adâncime: proprietăți

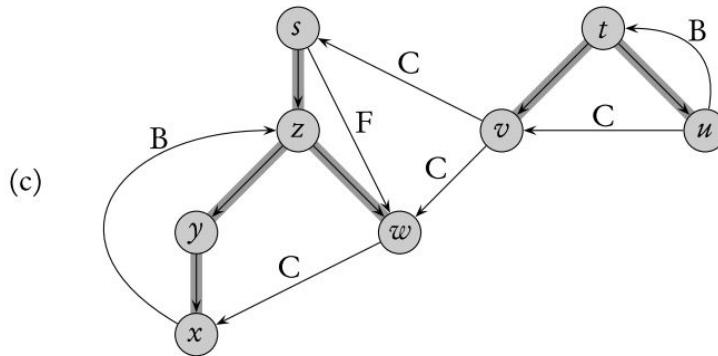
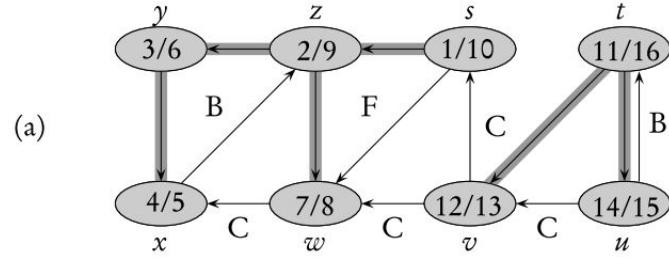


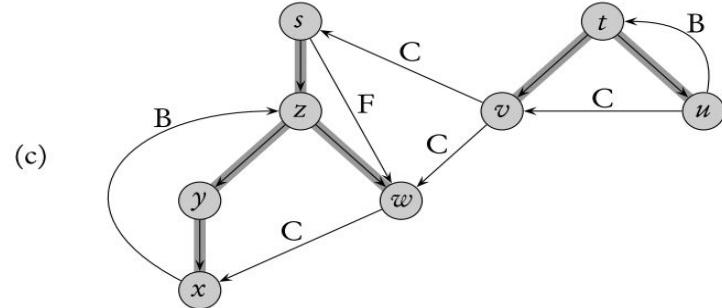
Diagrama ce exemplifica muchiile de întoarcere despre care vom vorbi imediat.

# Parcursarea în adâncime

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.

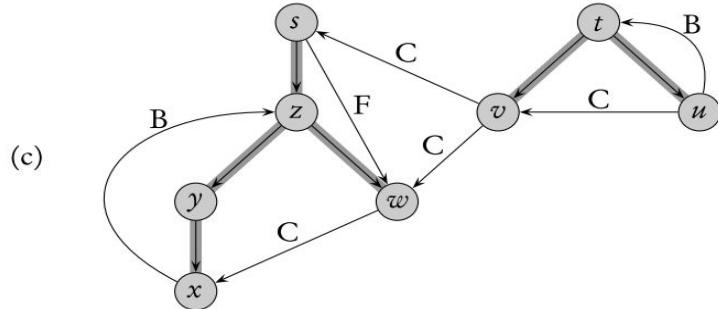
# Parcurea în adâncime



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. Bucurile (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, diferenți.

# Parcurea în adâncime

Într-un graf orientat nu vom avea toate cele 4 categorii...  
Ce categorii vom avea?

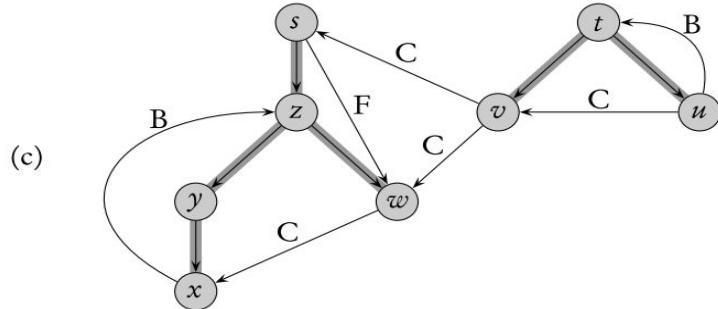


# Parcurea în adâncime

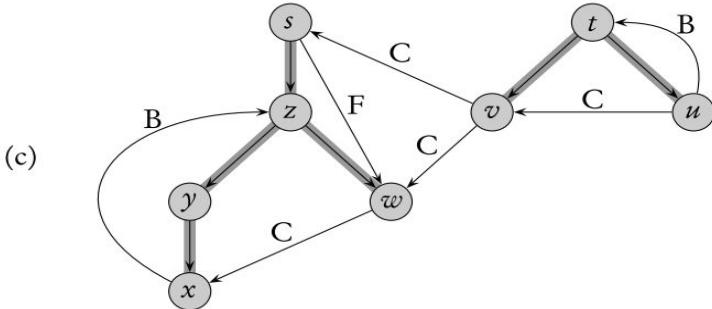
Într-un graf orientat nu vom avea toate cele 4 categorii...

Ce categorii vom avea?

- Doar primele 2 categorii.



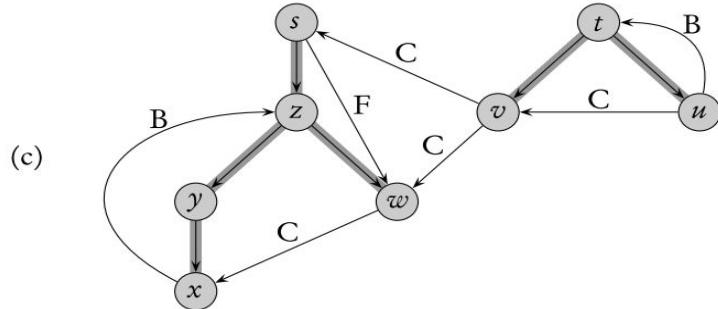
# Parcurea în adâncime



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

-

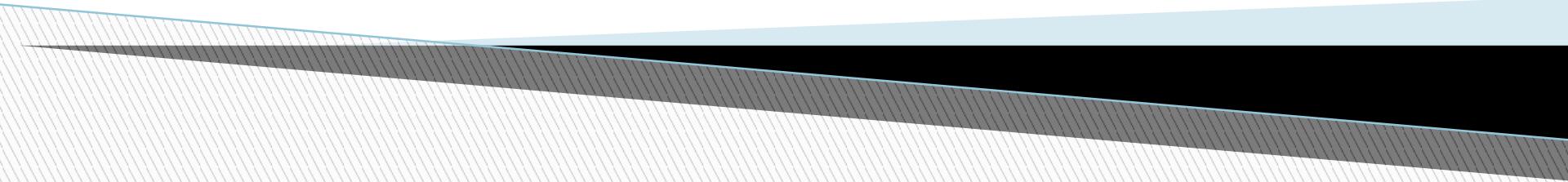
# Parcurea în adâncime



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

- Muchie inapoi...

# **Sortare topologică**



# Sortare topologică

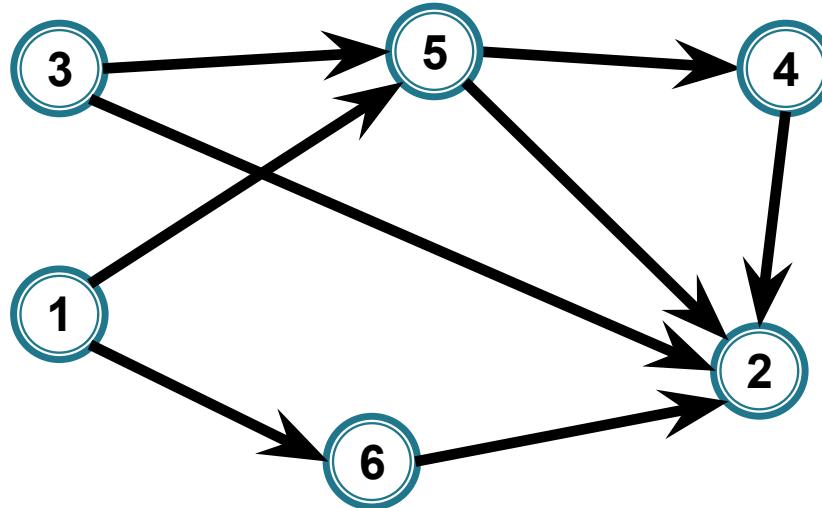
- **Fie  $G = (V, E)$  graf orientat**
- **Sortare topologică a lui  $G$  =**
  - ordonare a vârfurilor astfel încât dacă  $uv \in E$  atunci  $u$  se află înaintea lui  $v$  în ordonare
  - **Nu este neapărat unică**

# Sortare topologică

□ Fie  $G = (V, E)$  graf orientat

□ Sortare topologică a lui  $G =$

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

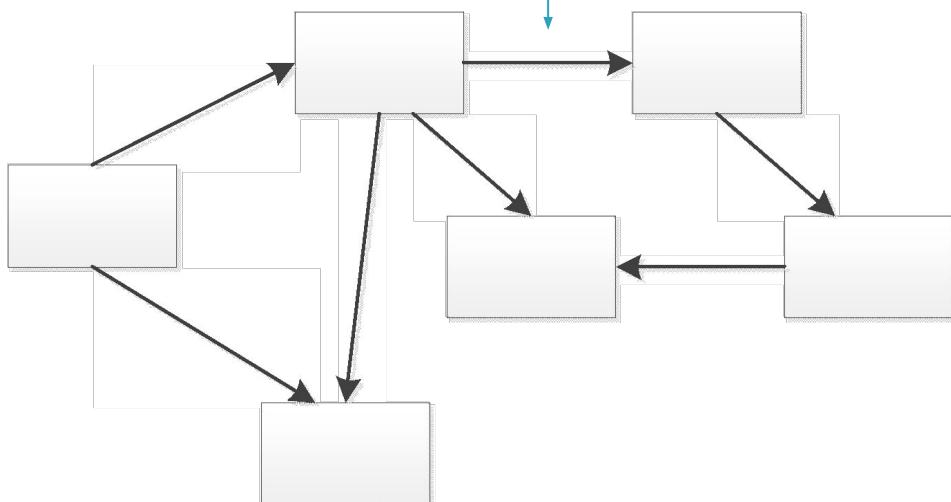


# Sortare topologică

## Aplicatii

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

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



# **În ce ordine trebuie executate activitățile?**

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

## **formulele din celulele B2..D2**

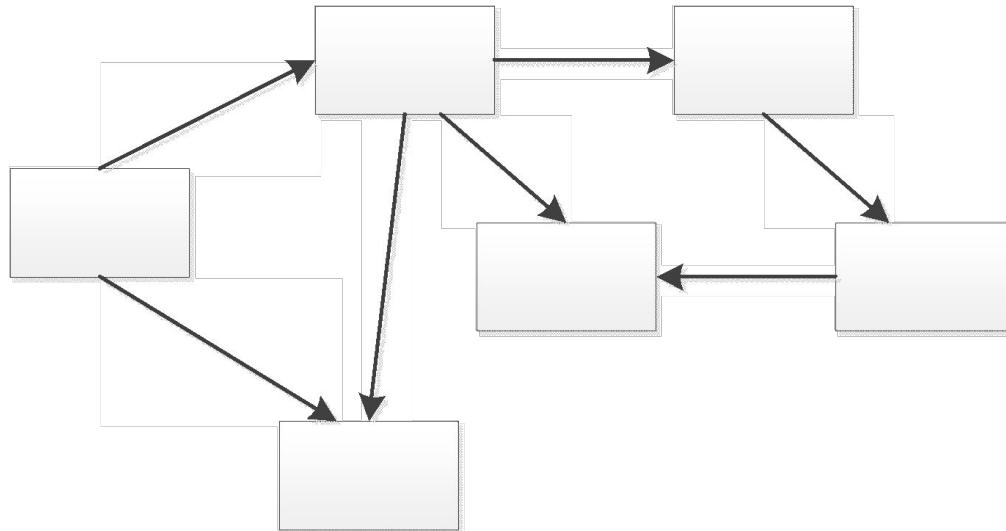
## **În ce ordine se evaluatează formulele?**

### **Probleme - dacă există dependențe circulare**

# Sortare topologică

## Aplicații

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



În ce ordine trebuie executate activitățile?

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

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

# Sortare topologică

- **Fie  $G = (V, E)$  graf orientat**
- **Sortare topologică a lui  $G$  =**
  - ordonare a vârfurilor astfel încât dacă  $uv \in E$  atunci  $u$  se află înaintea lui  $v$  în ordonare

# Sortare topologică

- **Fie  $G = (V, E)$  graf orientat**
- **Sortare topologică a lui  $G$  =**  
ordonare a vârfurilor astfel încât dacă  $uv \in E$  atunci  $u$  se află înaintea lui  $v$  în ordonare
- **Propoziție.** Dacă  $G$  este aciclic atunci  $G$  are o sortare topologică

# Sortare topologică

- **Fie  $G = (V, E)$  graf orientat**
- **Sortare topologică a lui  $G$  =**
  - ordonare a vârfurilor astfel încât dacă  $uv \in E$  atunci  $u$  se află înaintea lui  $v$  în ordonare
- **Propoziție.** Dacă  $G$  este aciclic atunci  $G$  are o sortare topologică
  - **Demonstrație  $\Rightarrow$  Algoritm?**



# Sortare topologică

- Fie  $G = (V, E)$  graf orientat
- Sortare topologică a lui  $G$  =  
ordonare a vârfurilor astfel încât dacă  $uv \in E$  atunci  $u$  se află înaintea lui  $v$  în ordonare
- **Propoziție.** Dacă  $G$  este aciclic atunci  $G$  are o sortare topologică

• **Demonstrație  $\Rightarrow$  Algoritm?**



Care vârf poate fi primul în sortarea topologică?



# Sortare topologică - Algoritm

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

# Sortare topologică - Algoritm

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

cât timp  $|V(G)| > 0$  execută

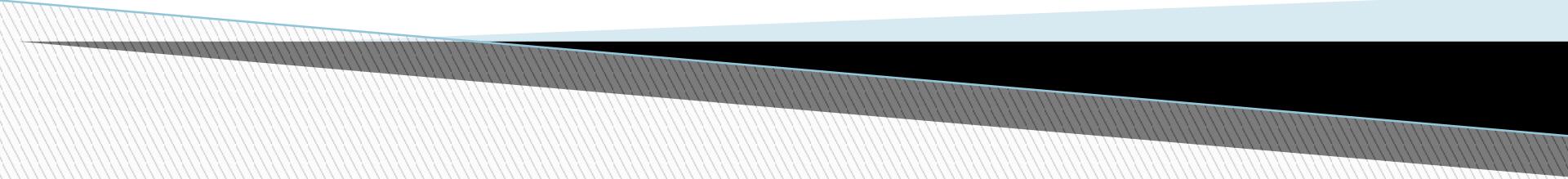
alege  $v$  cu  $d^-(v) = 0$

adauga  $v$  in ordonare

$G \leftarrow G - v$

- **Corectitudinea** – rezultă din **Lemă + inducție**

# Pseudocod



# Sortare topologică - Algoritm

## Algoritm

cât timp  $|V(G)| > 0$  execută

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

adauga  $v$  in ordonare

$G \leftarrow G - v$



Implementare? Complexitate?

# Sortare topologică - Algoritm

## Algoritm

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
-

# Sortare topologică - Algoritm

## Algoritm

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)
  -

# Sortare topologică - Algoritm

## Algoritm

cât timp  $|V(G)| > 0$  execută

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

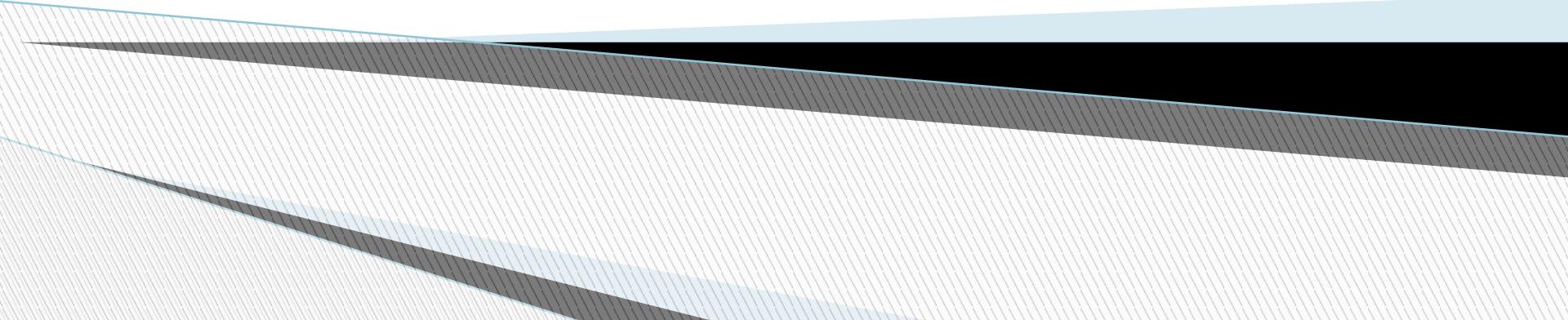
adauga  $v$  in ordonare

$G \leftarrow G - v$

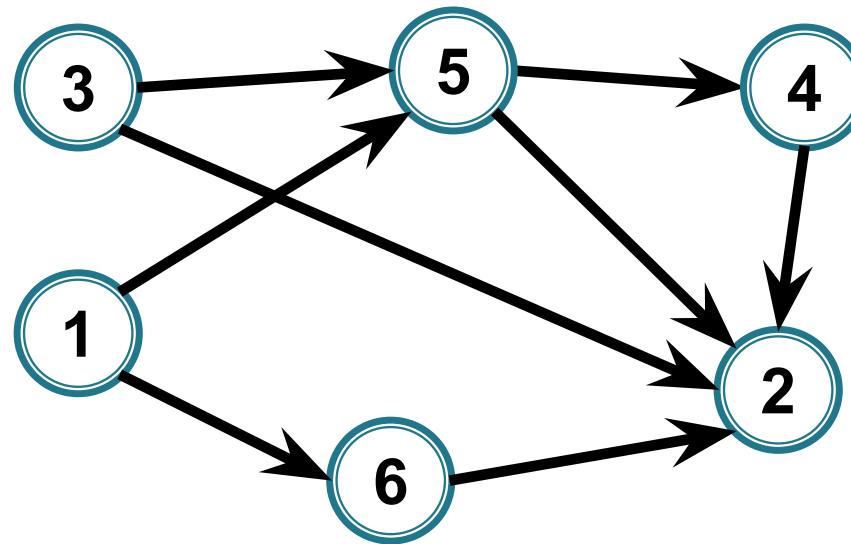
## Implementare - similar BF

- 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

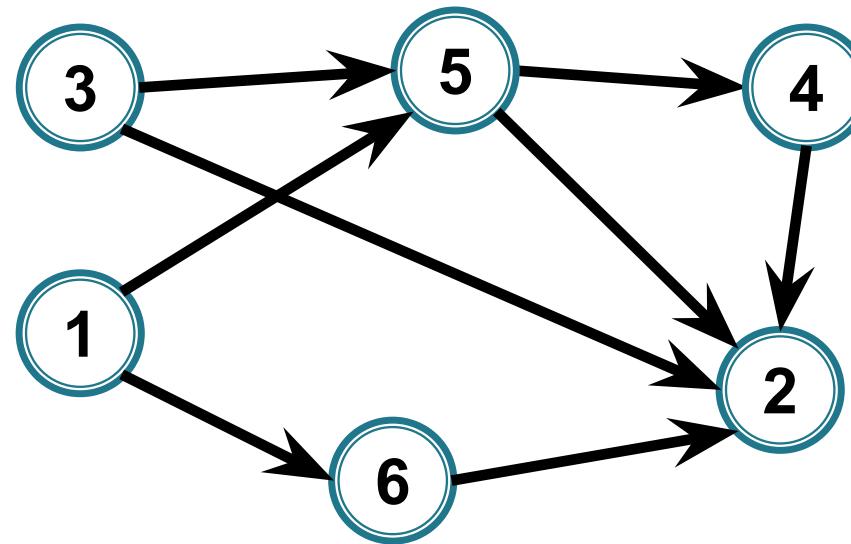
# **Exemplu**



# Sortare topologică

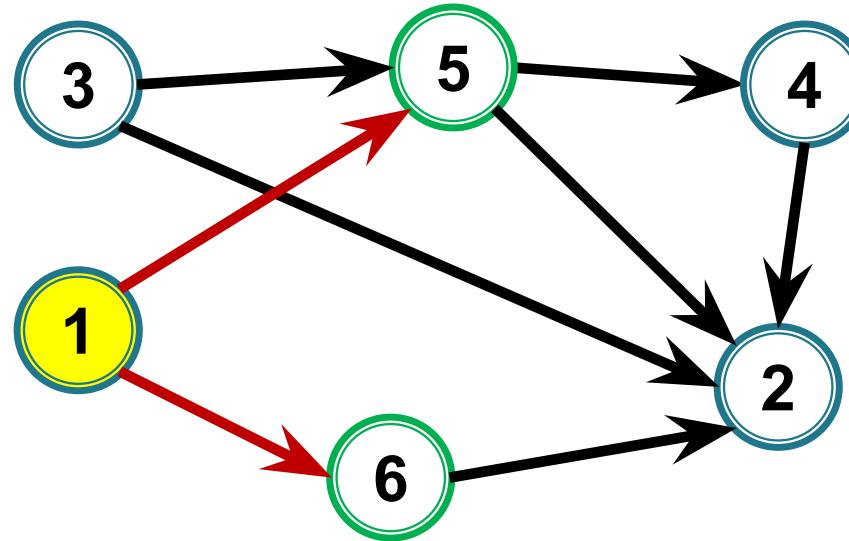


# Sortare topologică



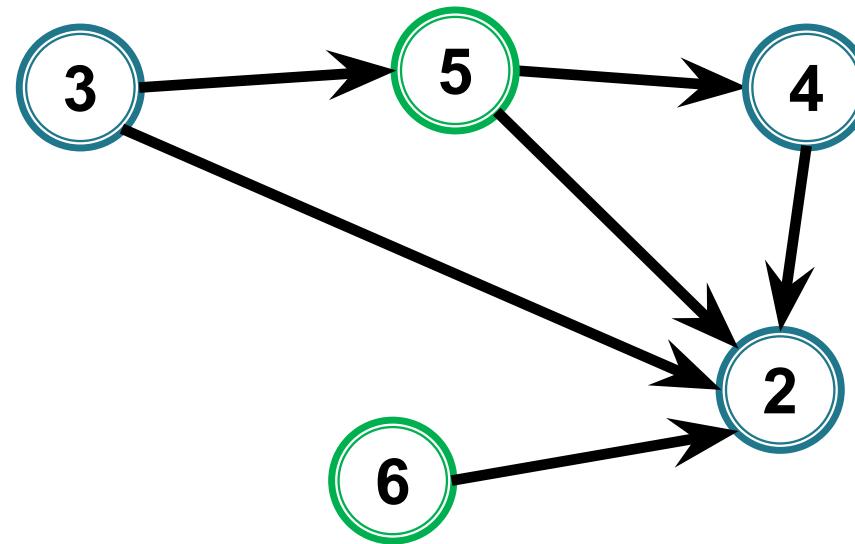
C: 1 3

# Sortare topologică



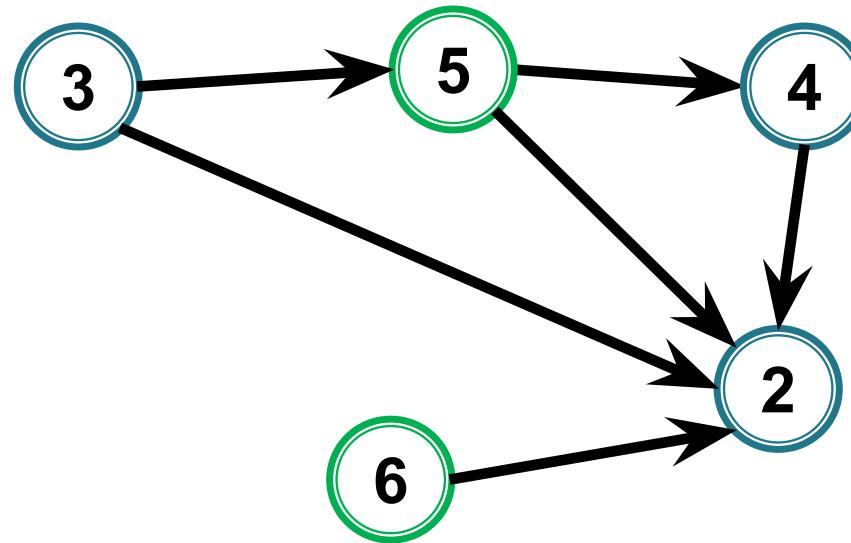
C: 1 3

# Sortare topologică



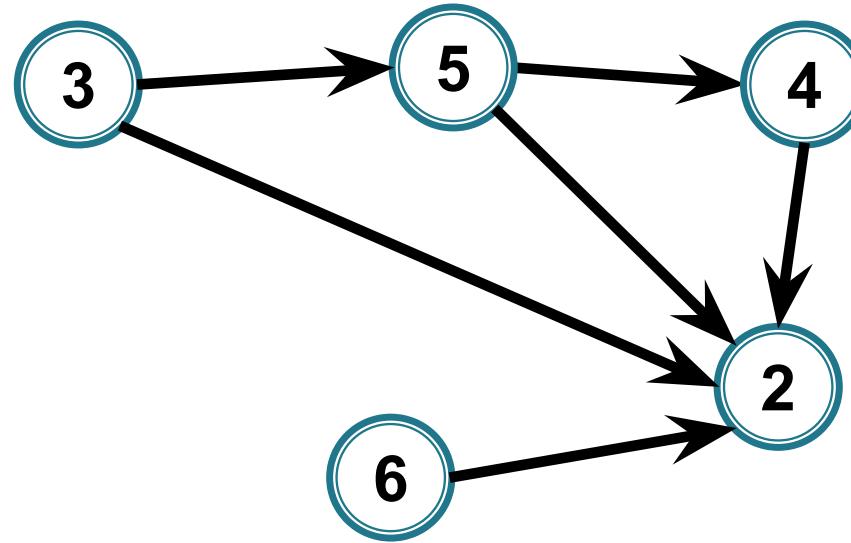
C: 1 3

# Sortare topologică



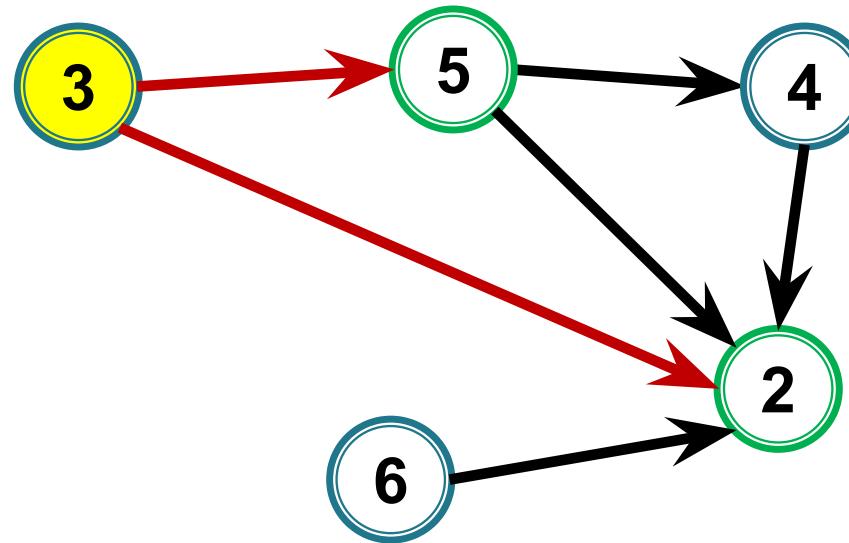
C: **1** 3 6

# Sortare topologică



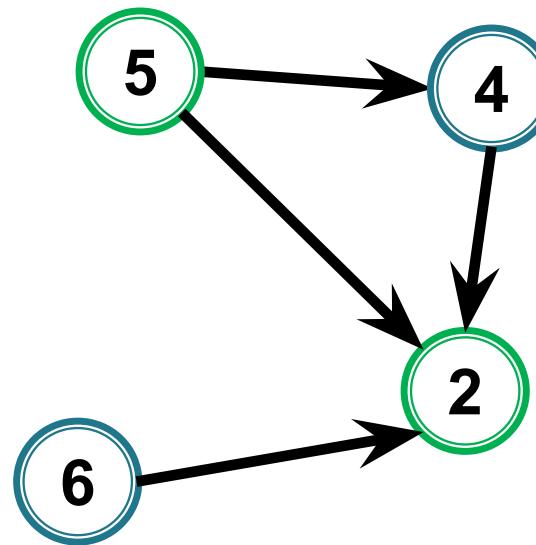
C: **1** 3 6

# Sortare topologică



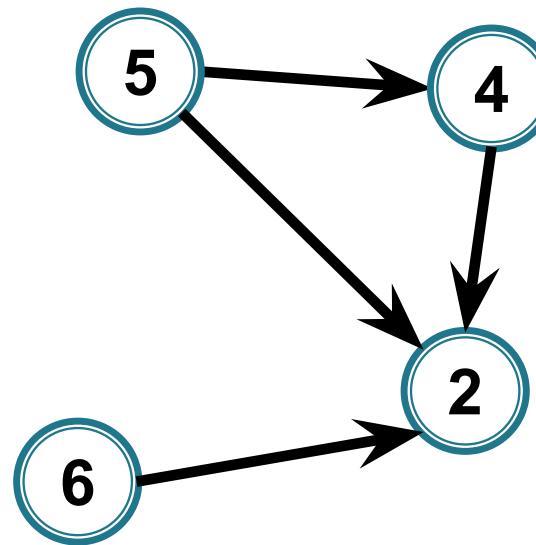
C: 1 3 6

# Sortare topologică



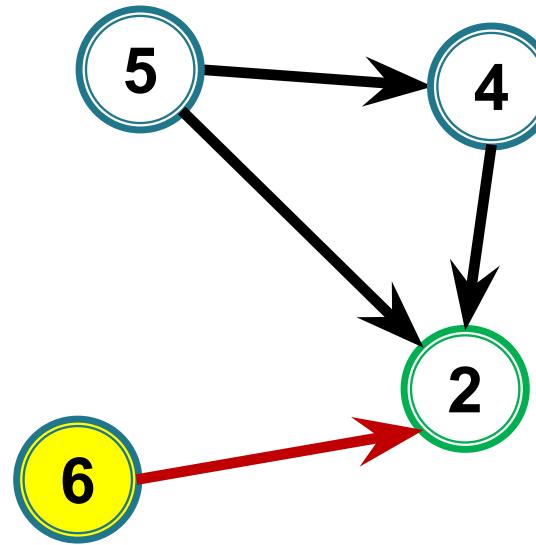
C: 1 3 6

# Sortare topologică



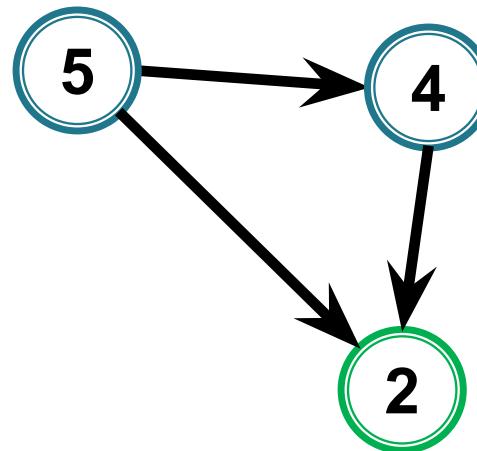
C: **1 3 6**  
**5**

# Sortare topologică



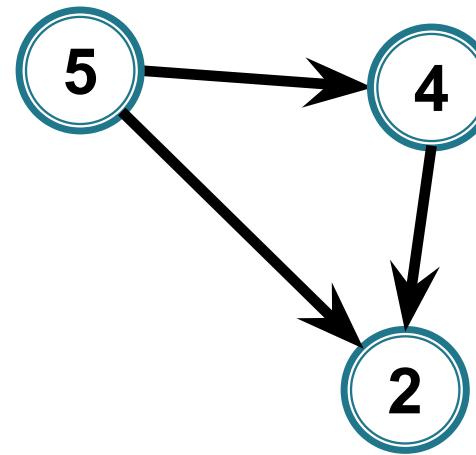
C: 1 3 6 5

# Sortare topologică



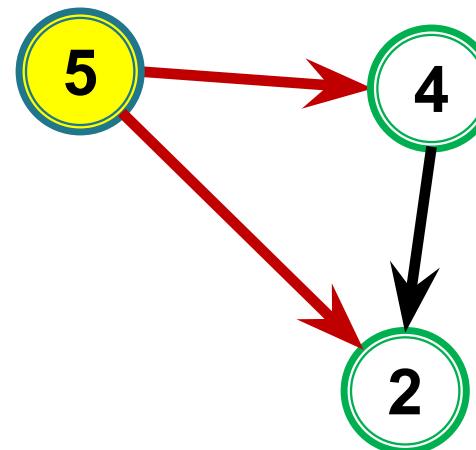
C: 1 3 6 5

# Sortare topologică



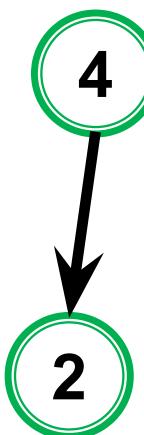
C: 1 3 6 5

# Sortare topologică



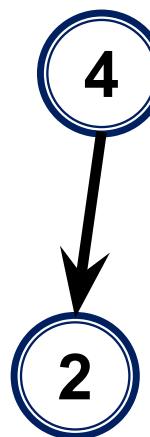
C: 1 3 6 5

# Sortare topologică



C: 1 3 6 5

# Sortare topologică



C: 1 3 6 5 4

# Sortare topologică



C: 1 3 6 5 4

# Sortare topologică

2

C: 1 3 6 5 4

# Sortare topologică

2

C: 1 3 6 5 4 2

# Sortare topologică

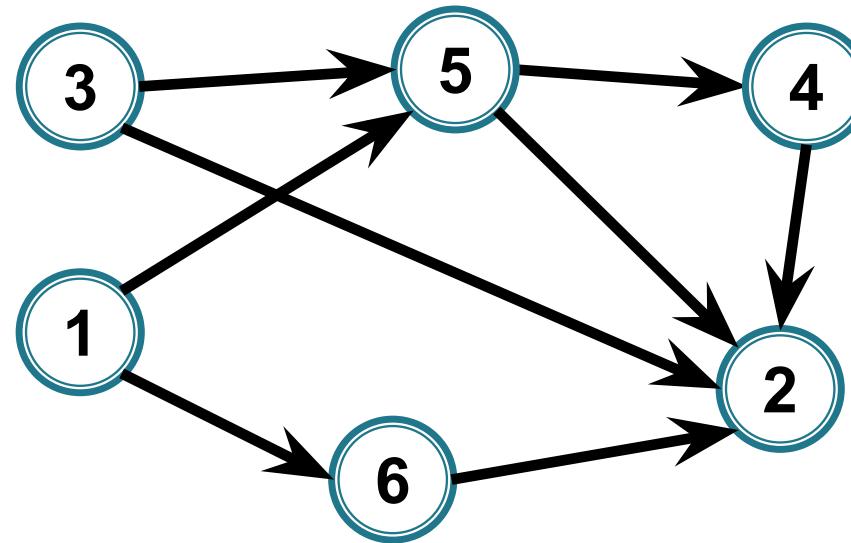
2

C: 1 3 6 5 4 2

# Sortare topologică

C: 1 3 6 5 4 2

# Sortare topologică



**Sortare topologică:** **1 3 6 5 4**  
**2**

# Sortare topologică - Algoritm

*coada*  $C \leftarrow \emptyset;$

*adauga* in  $C$  toate vârfurile  $v$  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

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

    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

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

    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

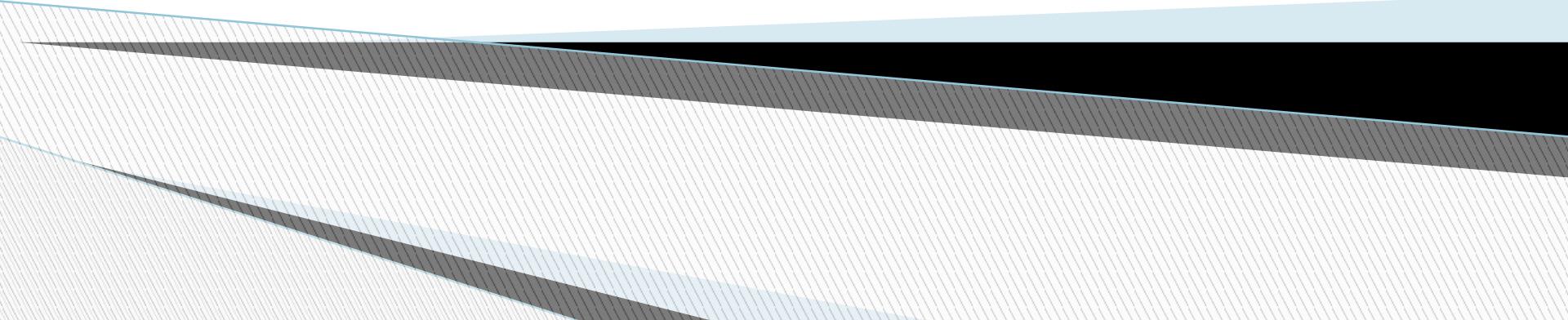
            adauga( $j$ ,  $C$ )

# Sortare topologică



- **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  $f[u] = \text{momentul la care a fost } \underline{\text{finalizat}} \text{ vârful } u \text{ în parcurgerea DF}$  avem:

$$uv \in E \Rightarrow f[u] > f[v]$$



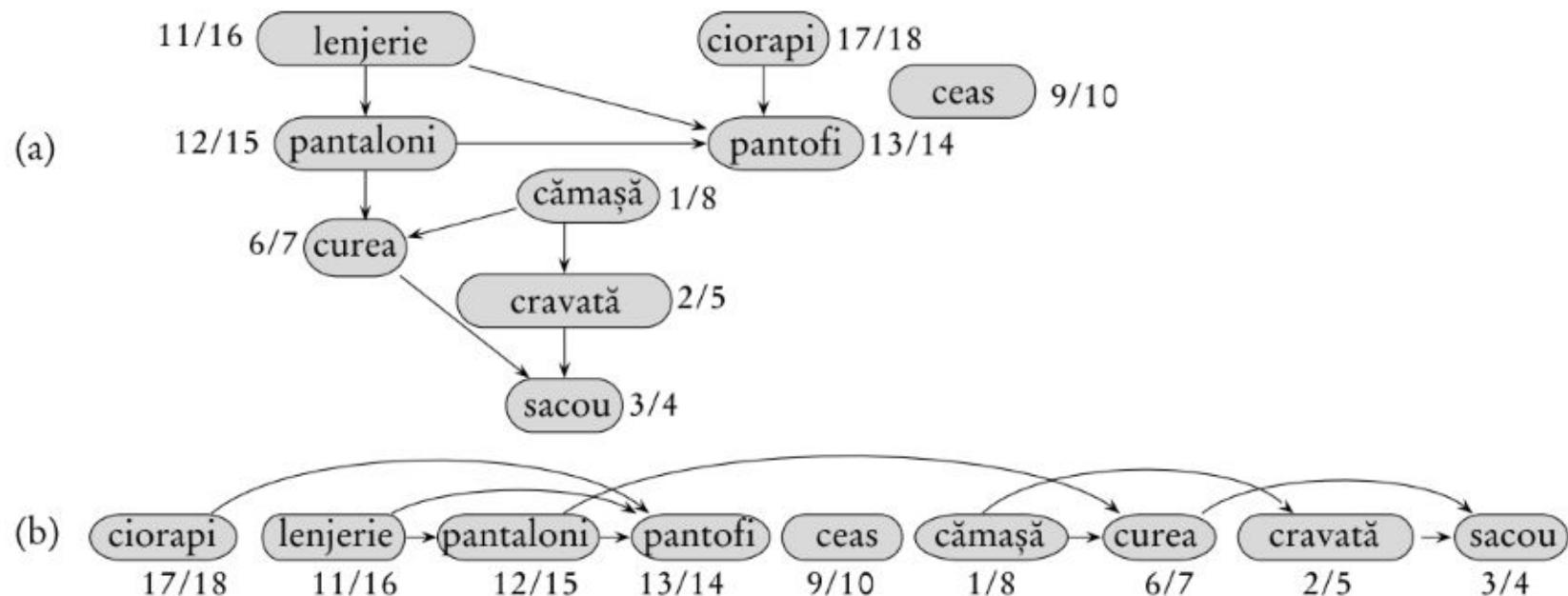
# Sortare topologică – Alt algoritm

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



**Figura 23.7** (a) Profesorul Bumstead își sortează topologic îmbrăcămintea când se îmbracă. 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 **final**[ $u$ ] = momentul la care a fost finalizat vârful  $u$  în parcurgerea DF avem:  $uv \in E \Rightarrow f[u] > f[v]$ 
  - Atunci sortare topologică = sortare descrescătoare în raport cu final

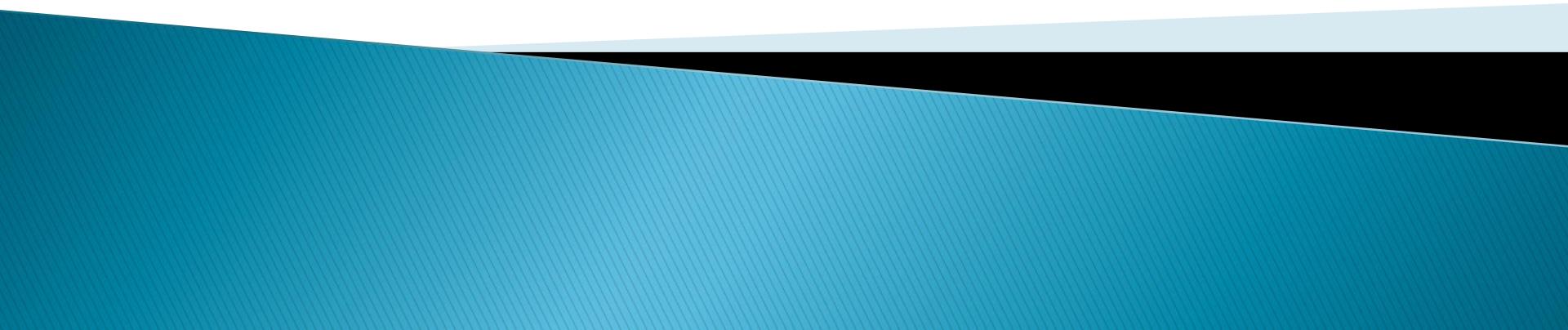
## Sortare-Topologică( $G$ )

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

# 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 vector tata), prin care se descoperă vârfuri noi
  - **muchii de întoarcere** – închid ciclu, nu pot fi critice

# Muchii critice



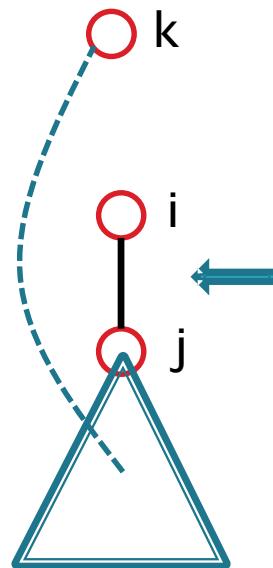
**Cum testăm dacă o muchie de avansare  $(i,j)$  este critică?**

# 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

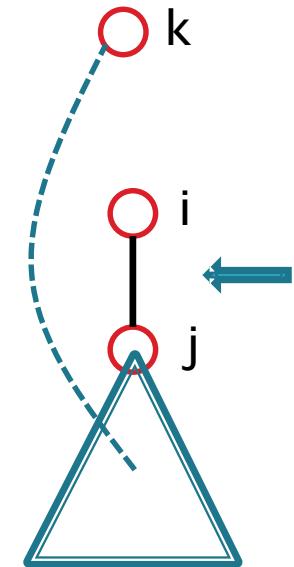
$\Leftrightarrow$

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 descendant  
al lui  $j$  și
- 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  $i$ )

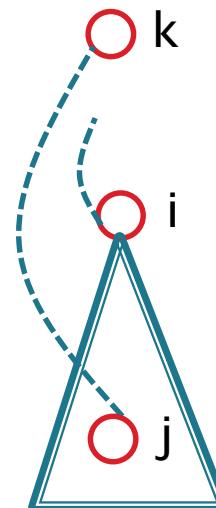


# Muchii critice

Memorăm pentru fiecare vârf  $i$ :

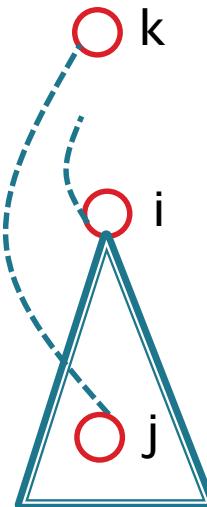
$niv\_min[i]$  = nivelul minim al unui vârf care este extremitate a unei muchii de întoarcere din  $i$  sau dintr-un descendant al lui  $i$

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



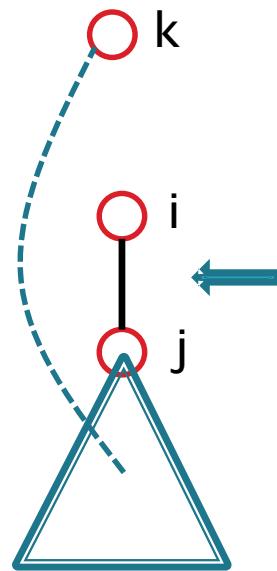
# Muchii critice

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



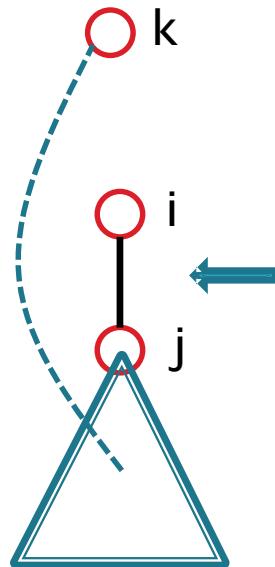
# Muchii critice

O muchie de avansare  $ij$  este critică  $\Leftrightarrow$



# Muchii critice

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



# Muchii critice



Cum calculăm eficient `niv_min[i]` ?

`niv_min[i] = min { nivel[i], A, B}`

`A = min {nivel[k] | ik muchie de întoarcere}`

`B = min {nivel[k] | jk descendent al lui i,  
jk muchie de întoarcere}`

# Muchii critice

Cum calculăm eficient `niv_min[i]` ?

`niv_min[i] = min { nivel[i], A, B}`

`A = min {nivel[k] | ik muchie de întoarcere}`

`B = min {nivel[k] | jk descendent al lui i,  
jk muchie de întoarcere}`



**B se poate calcula recursiv**

# Muchii critice

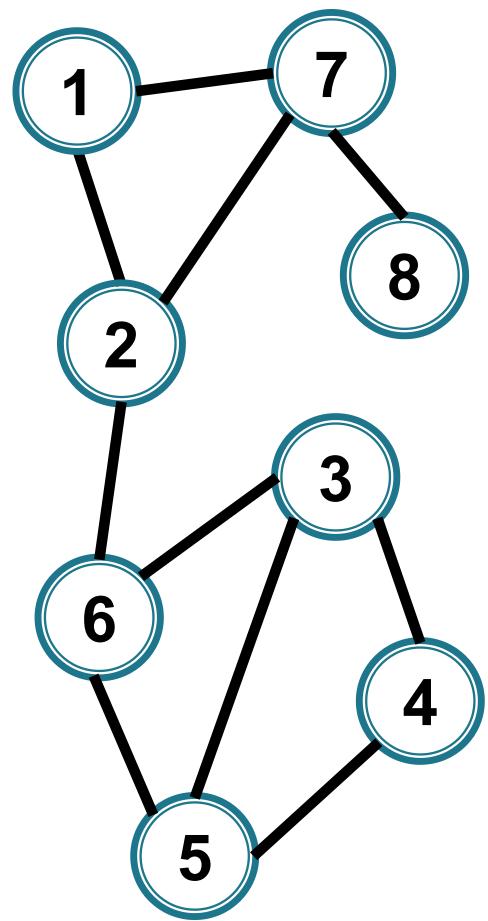
Cum calculăm eficient `niv_min[i]` ?

`niv_min[i] = min { nivel[i], A, B}`

`A = min {nivel[k] | ik muchie de întoarcere}`

`B = min {nivel[k] | j descendent al lui i,  
jk muchie de întoarcere}`

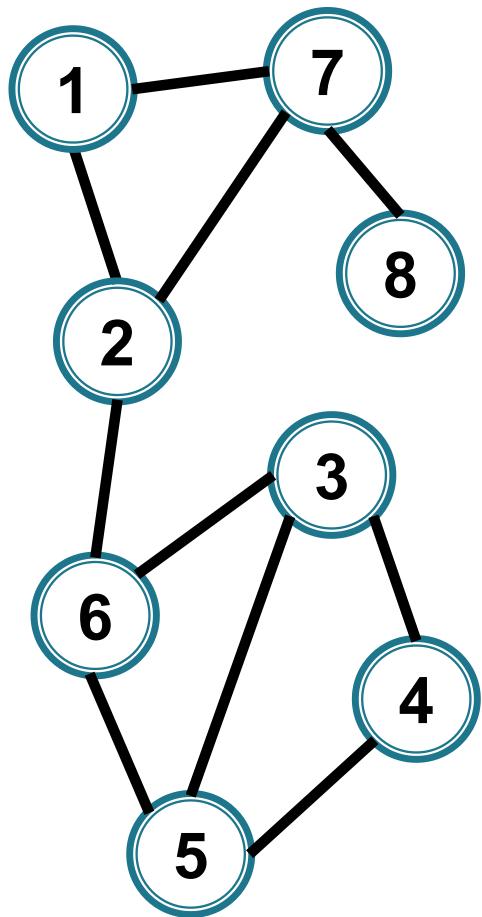
**`B = min {niv_min[j] | j fiu al lui i}`**

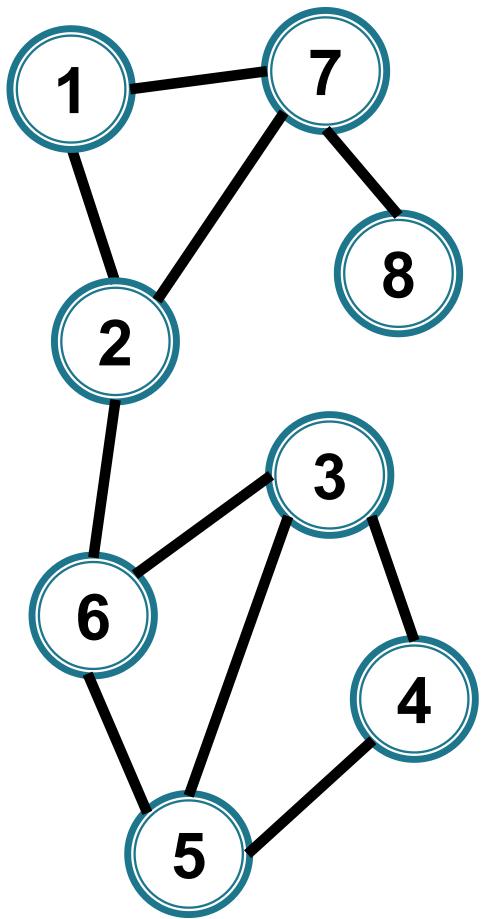


nivel/niv\_min

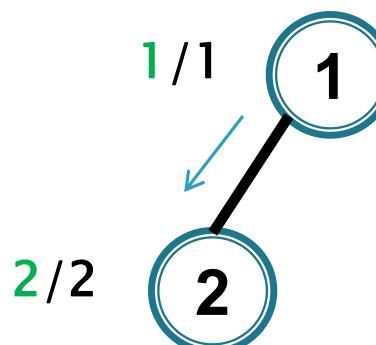
1 / 1

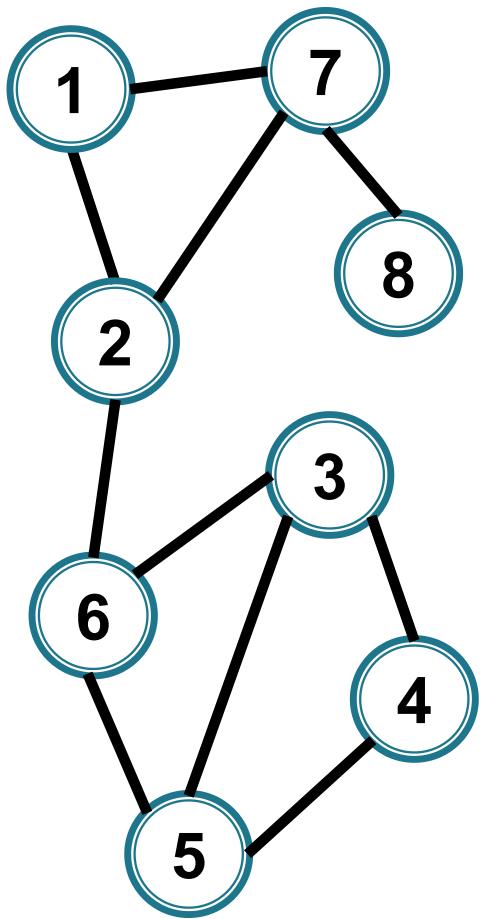
1



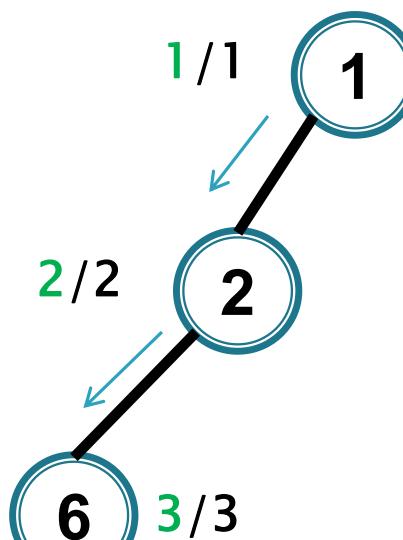


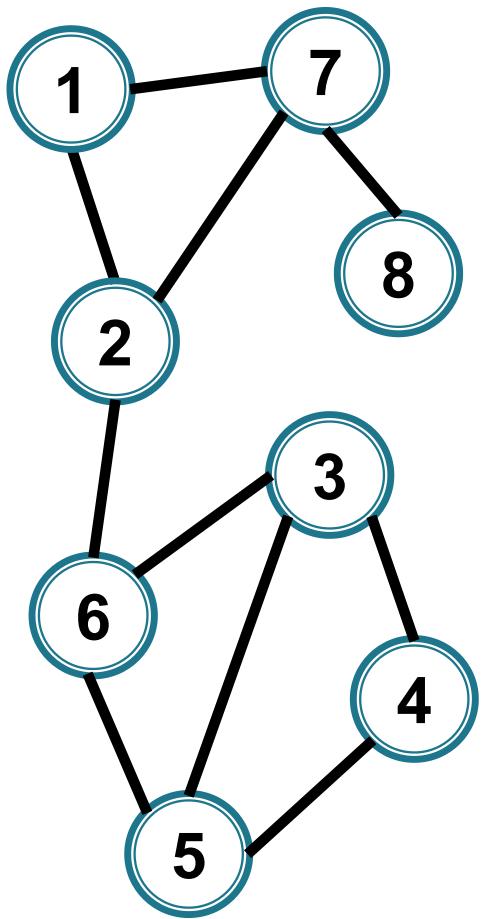
nivel/niv\_min



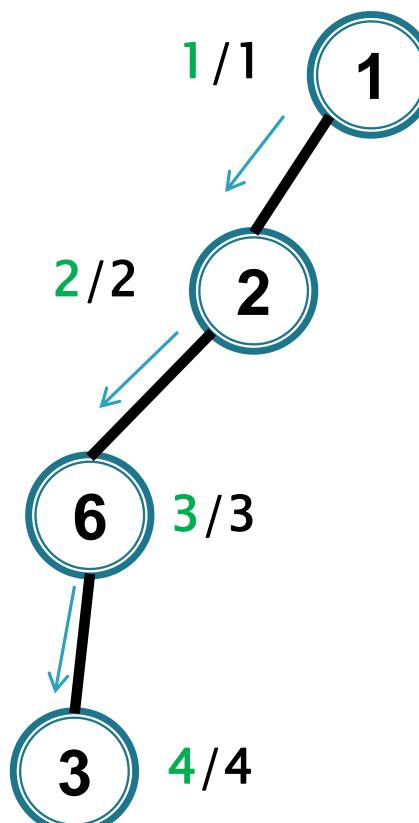


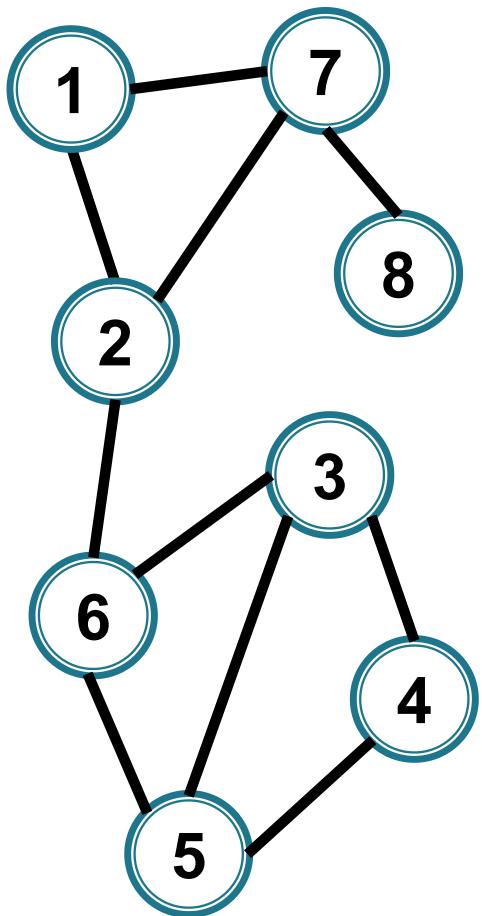
nivel/niv\_min



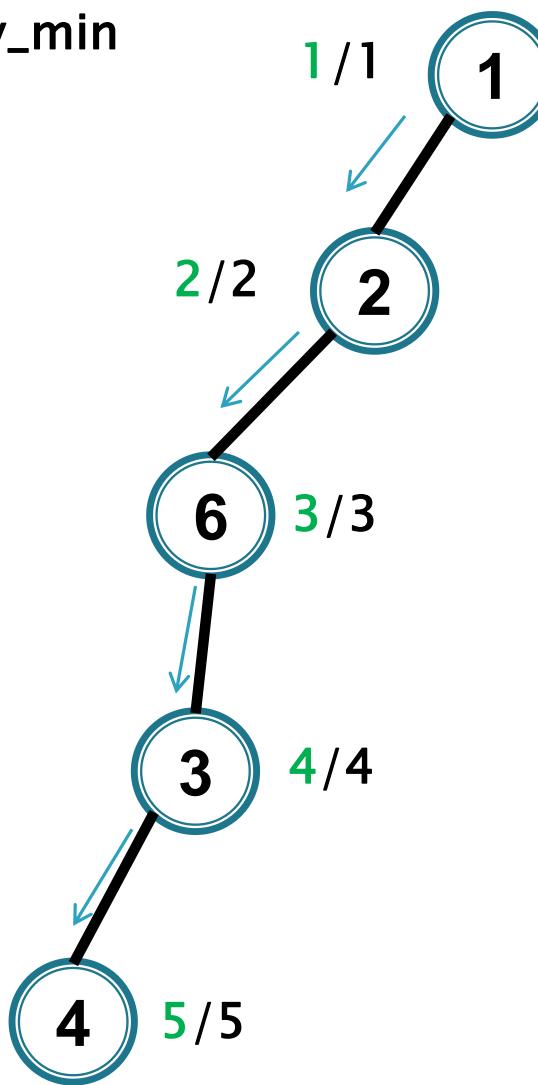


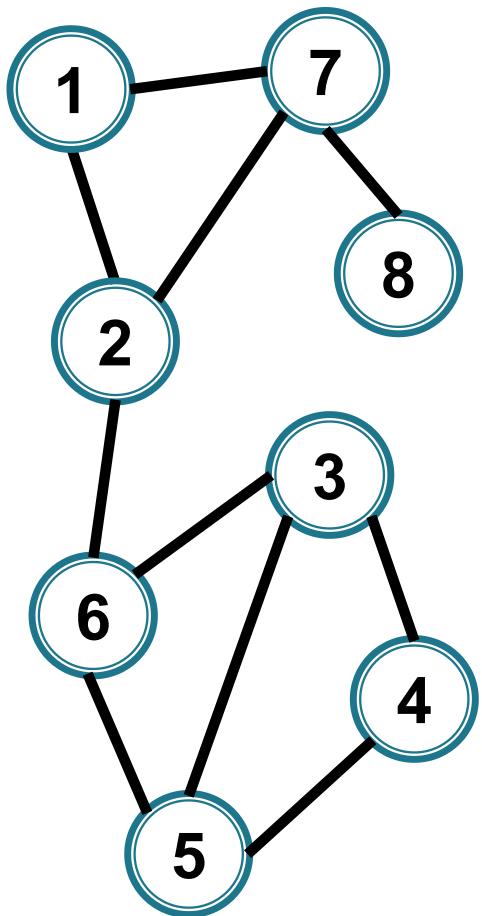
nivel/niv\_min



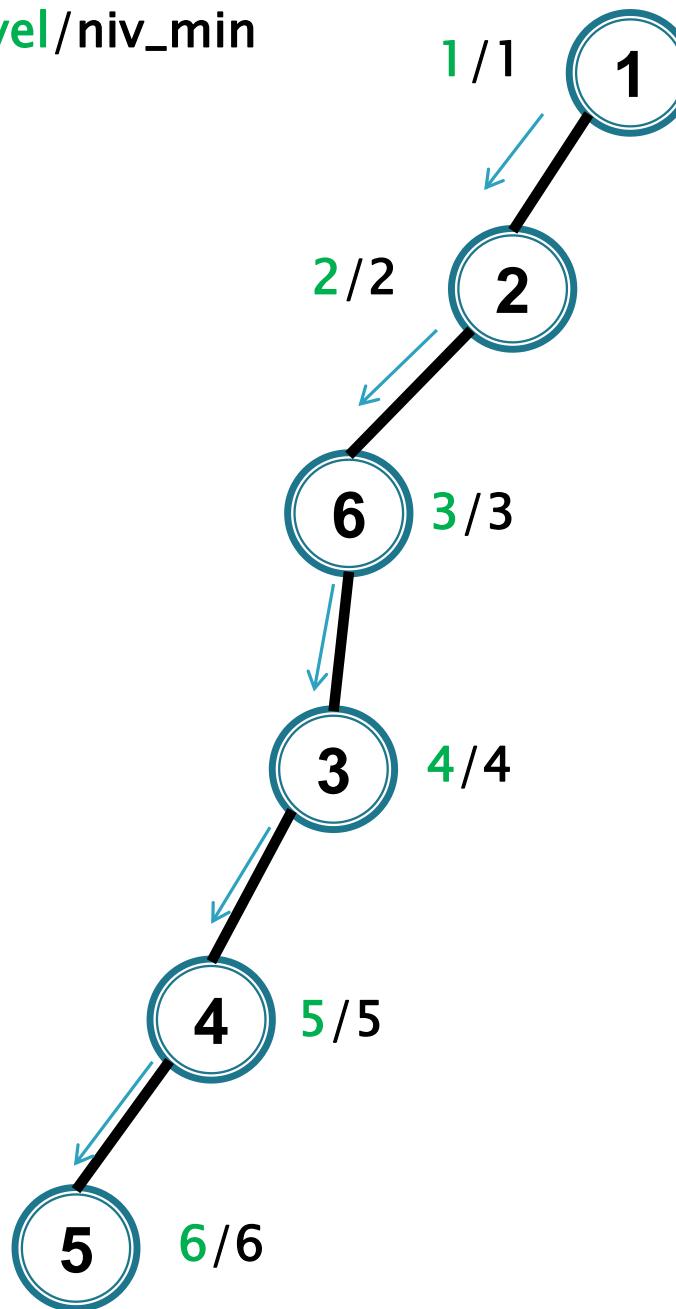


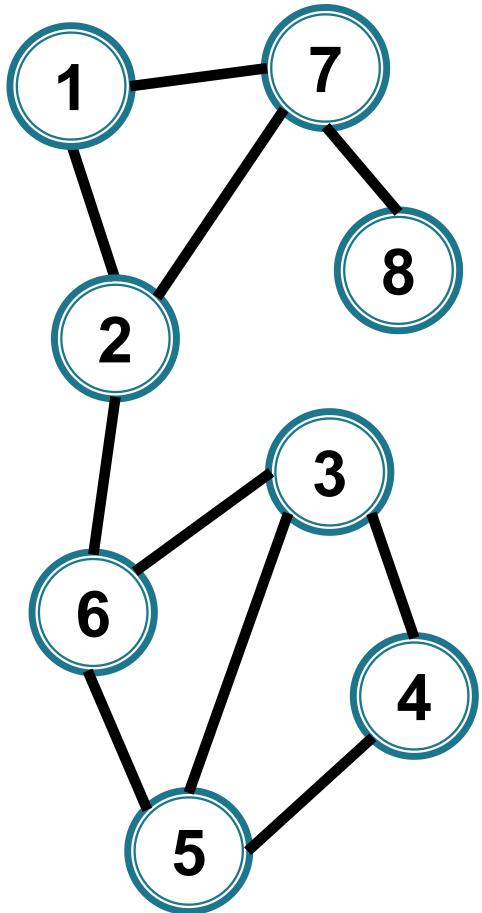
nivel/niv\_min



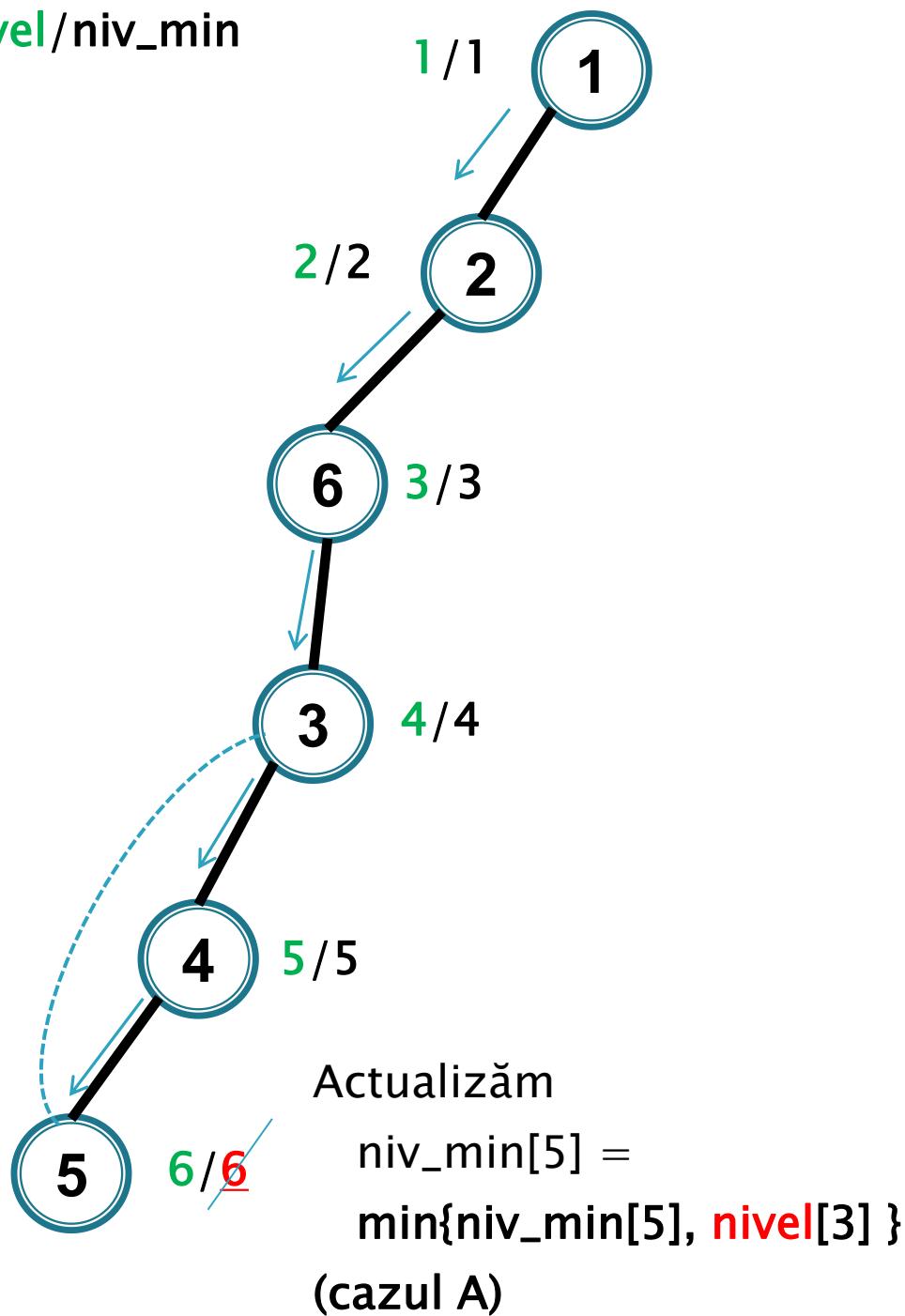


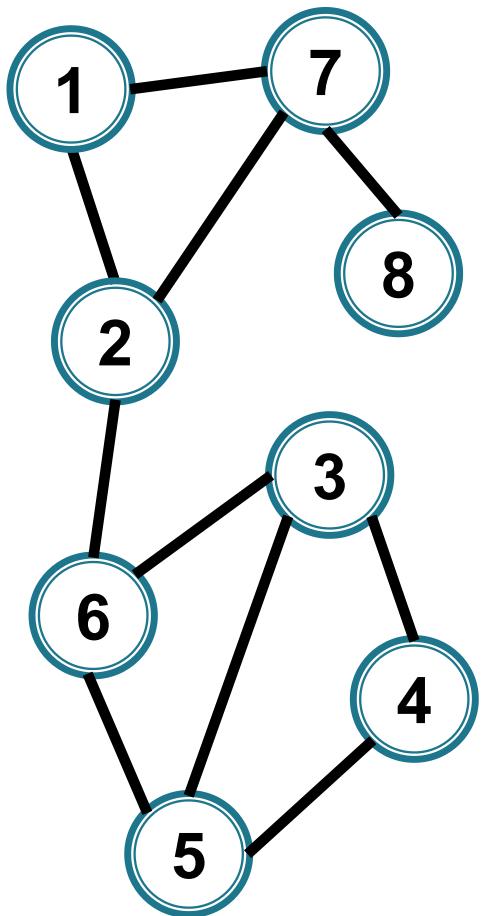
nivel/niv\_min



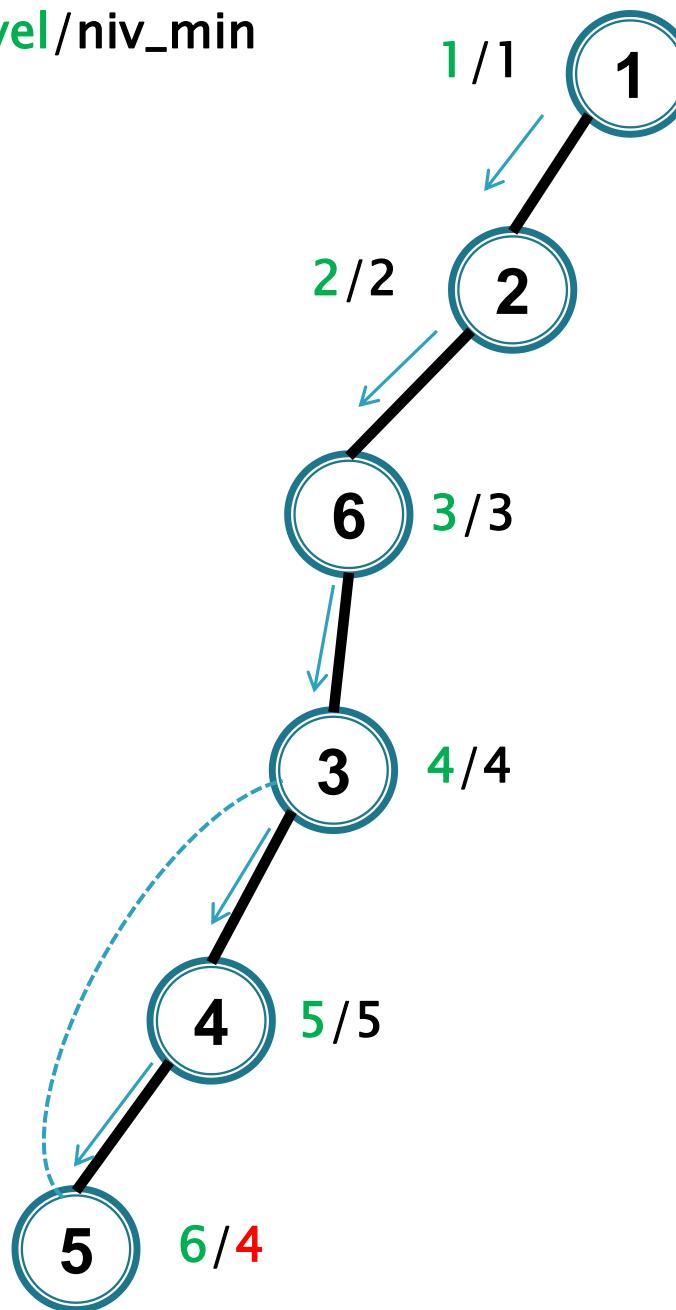


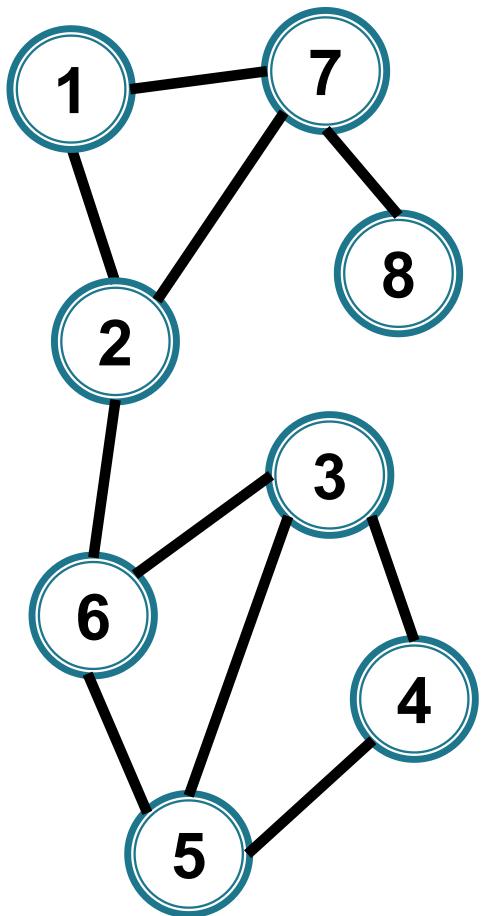
nivel/niv\_min



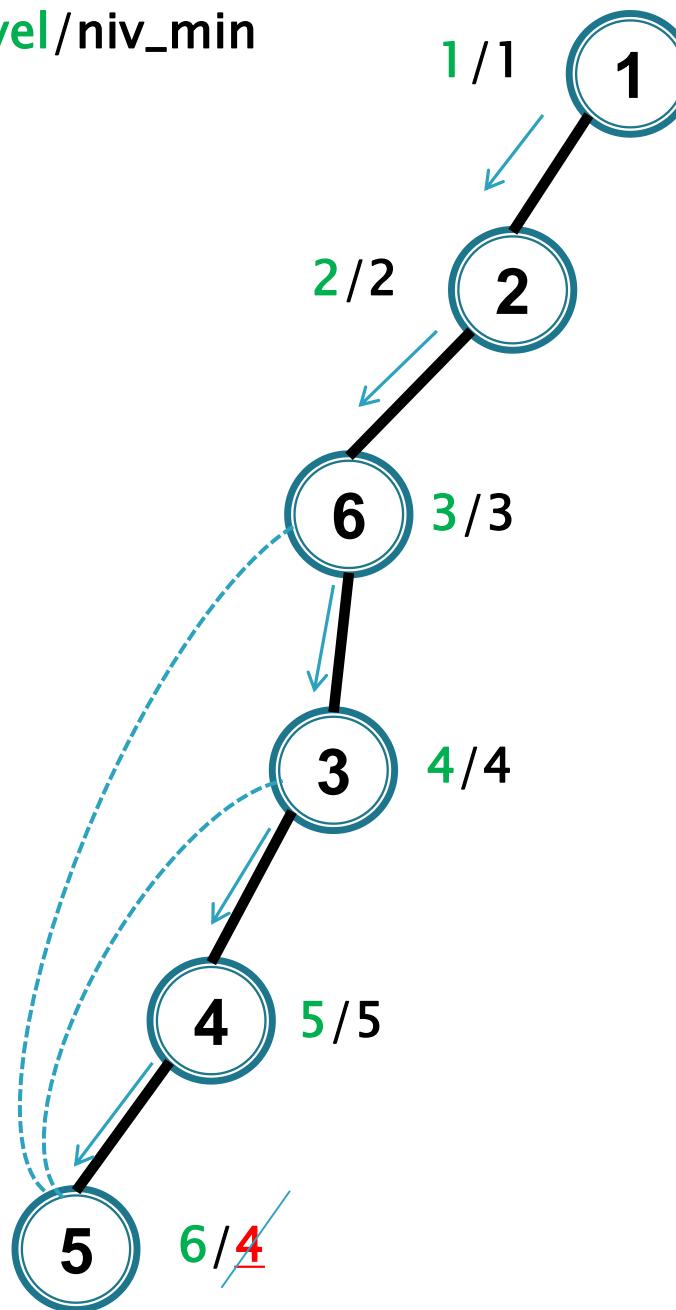


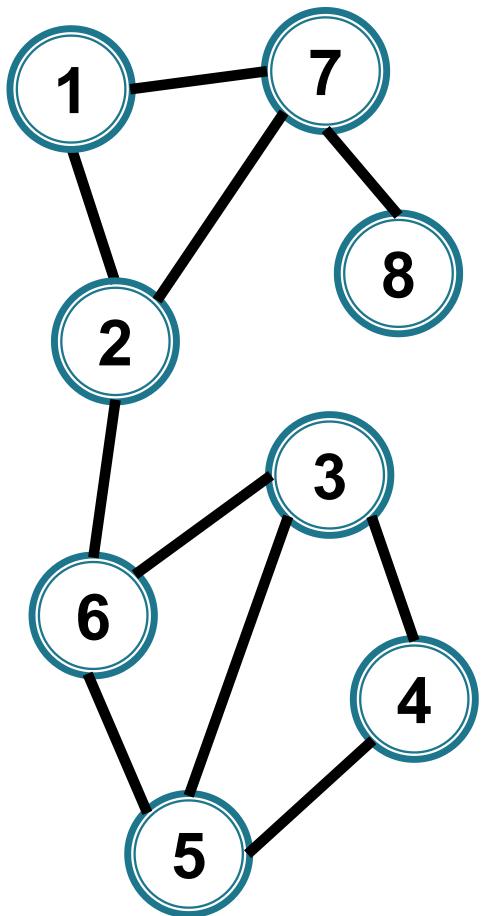
nivel/niv\_min



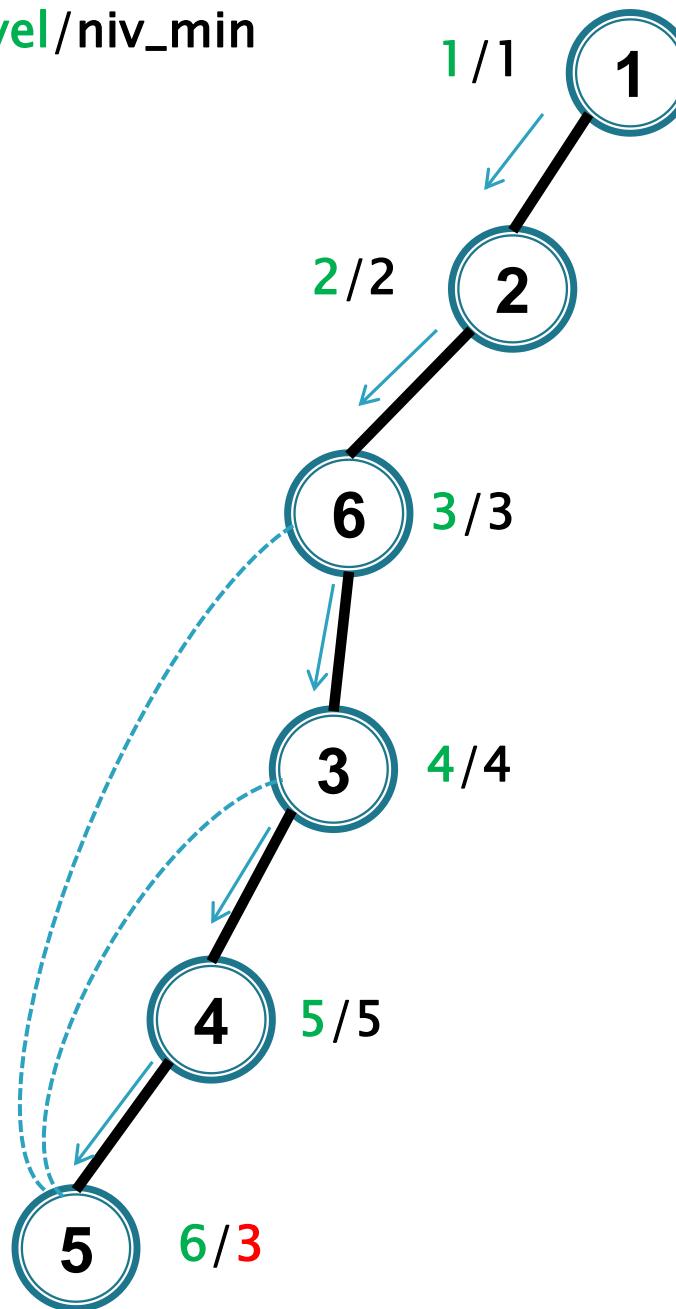


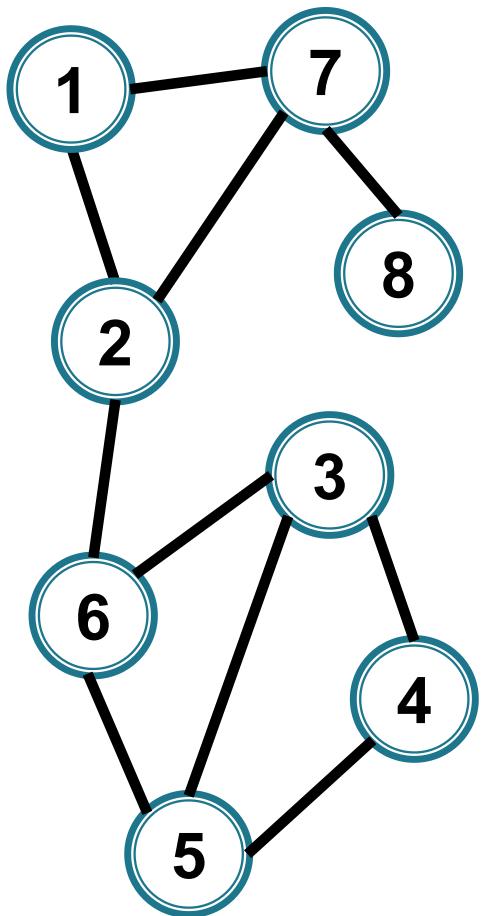
nivel/niv\_min



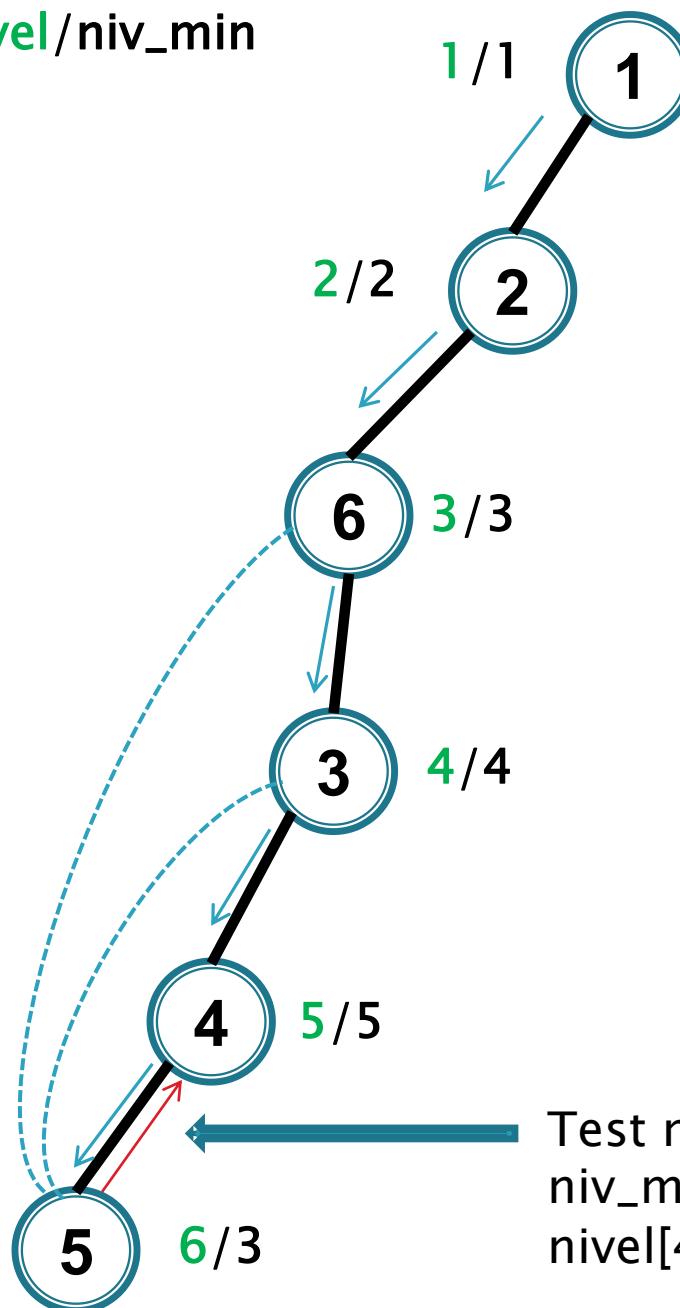


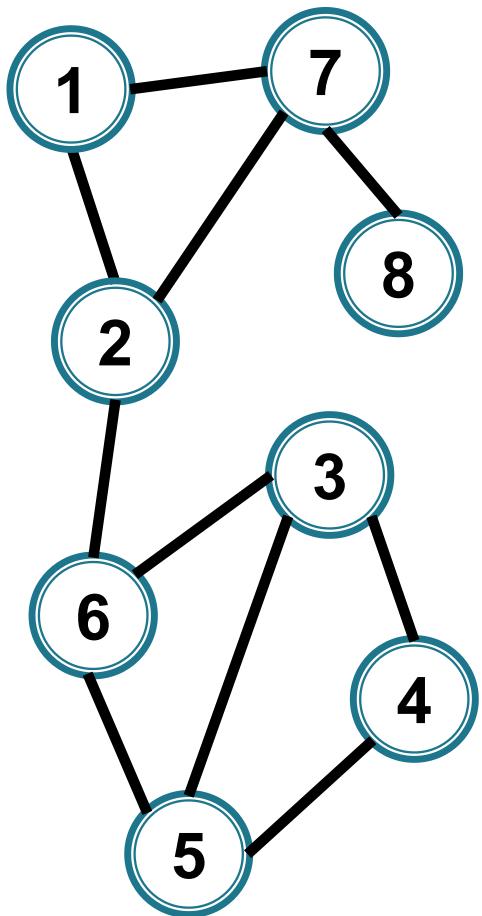
nivel/niv\_min



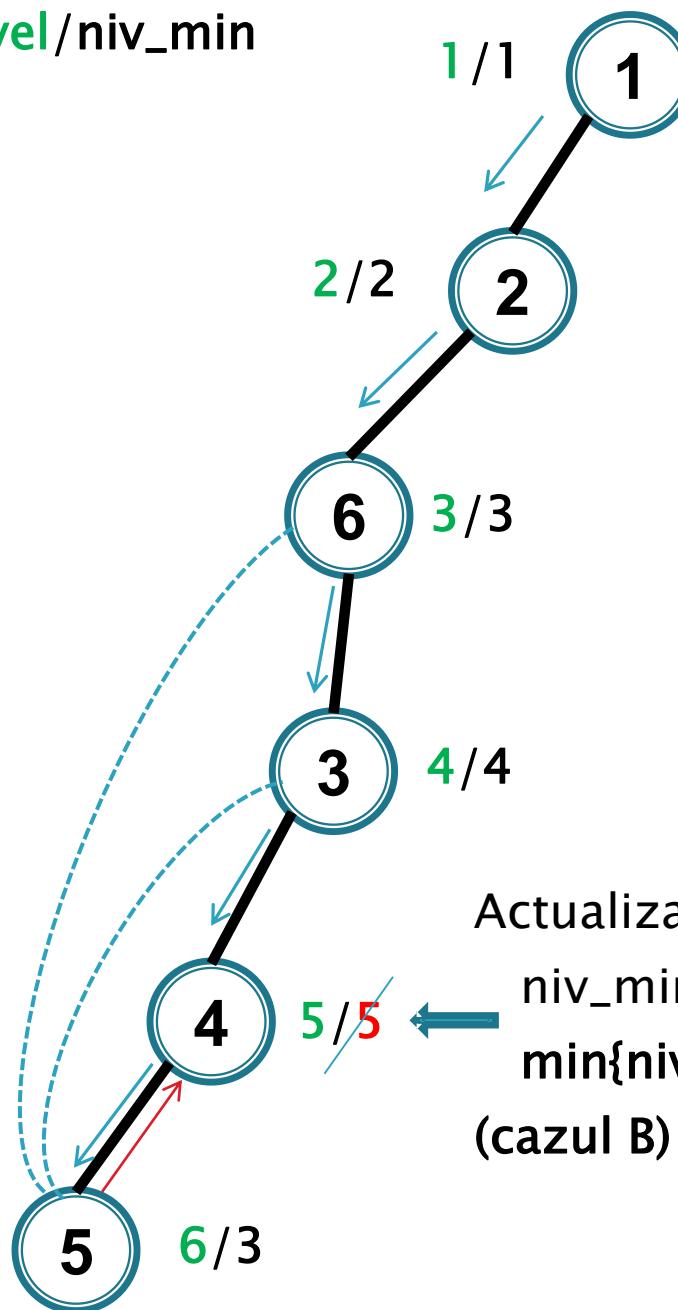


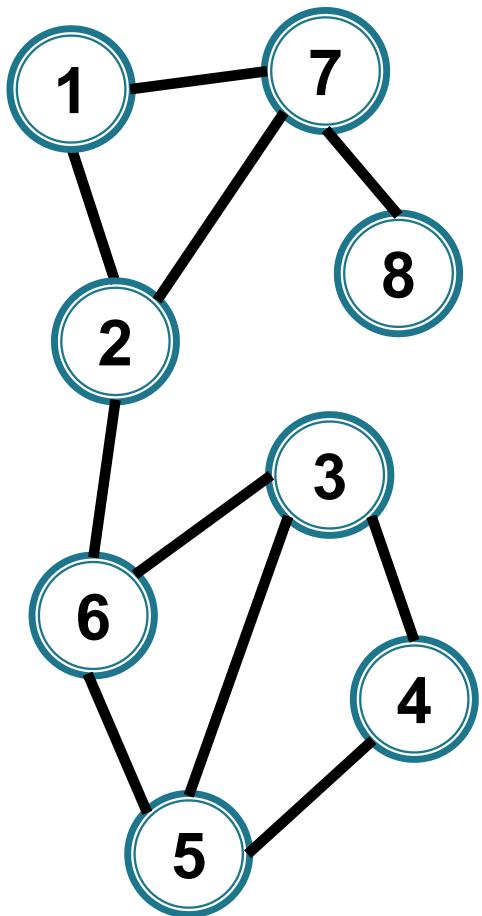
nivel/niv\_min



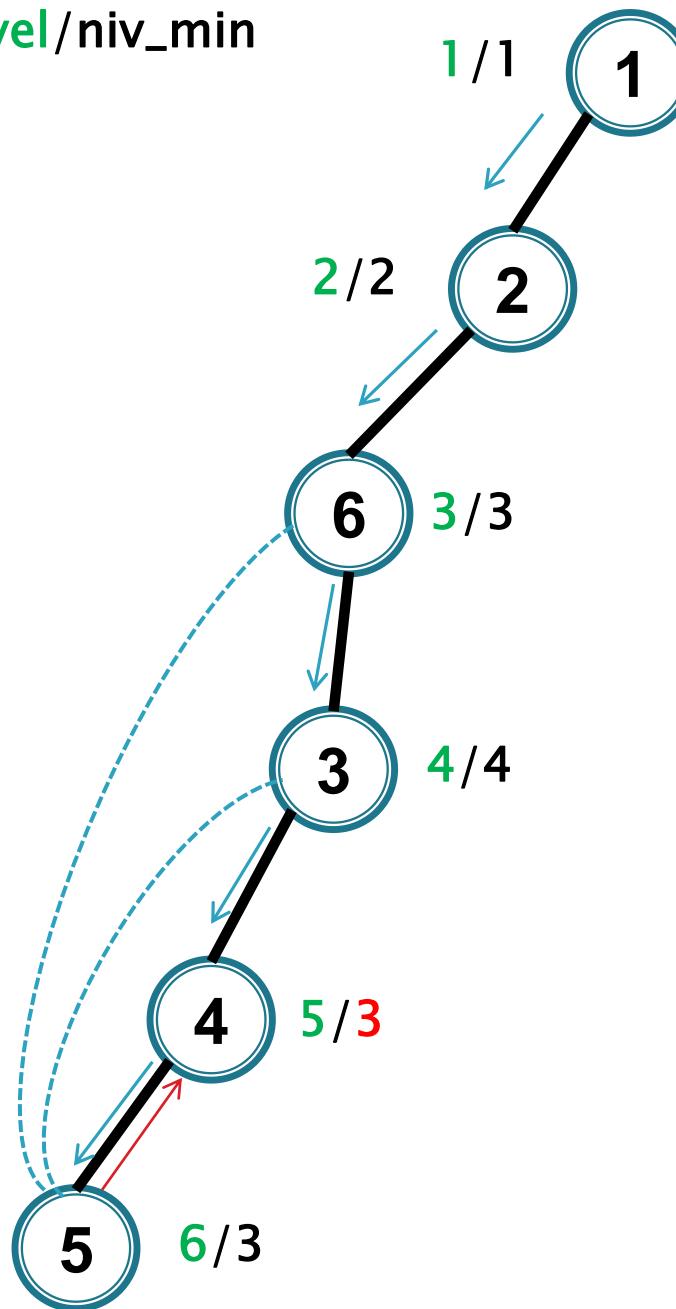


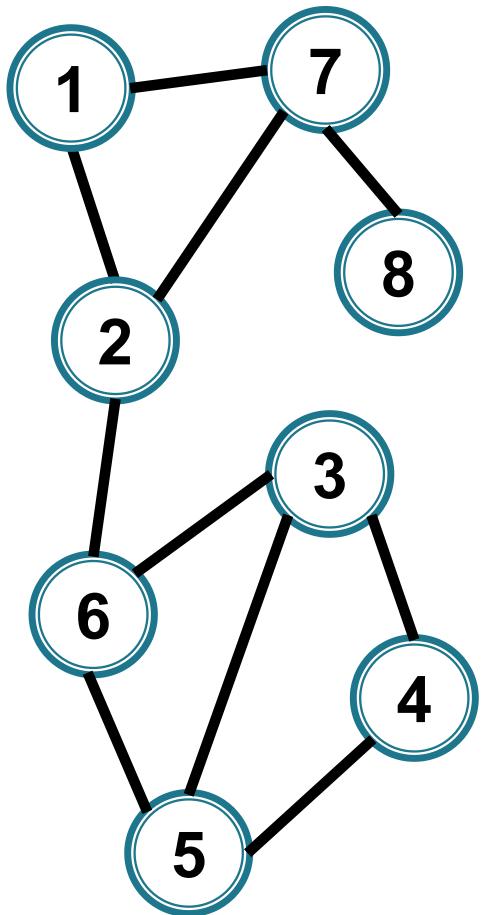
nivel/niv\_min



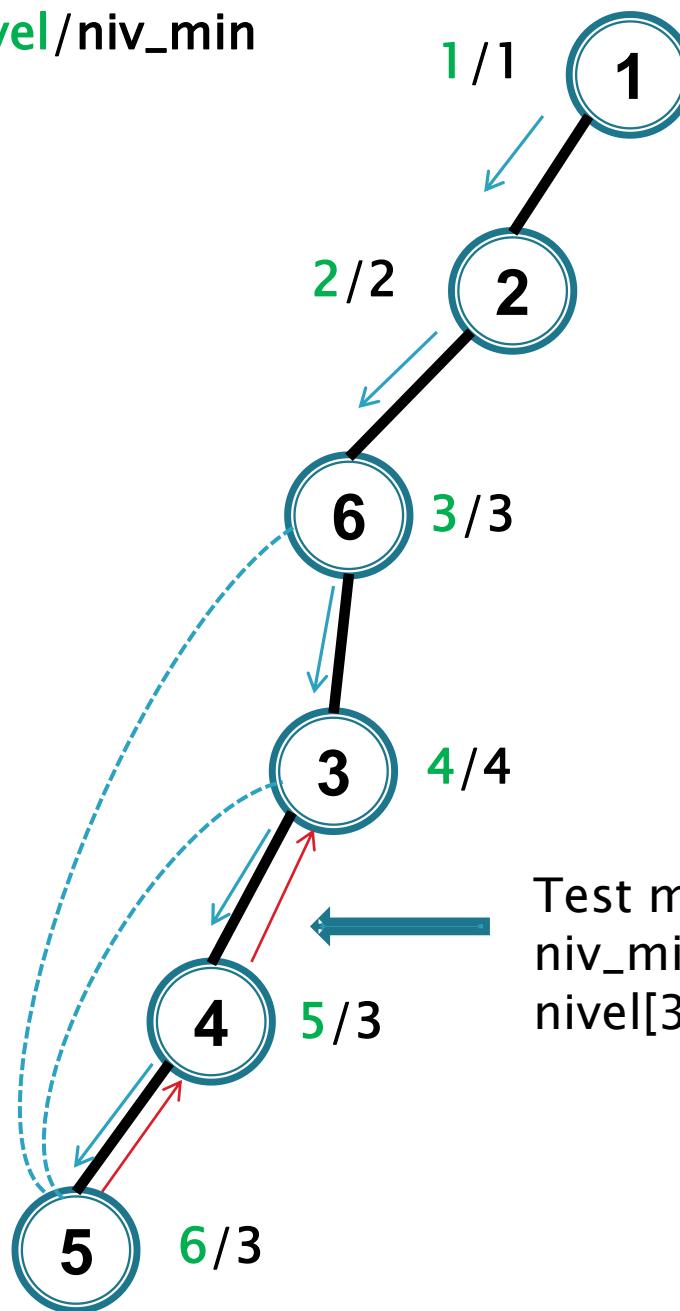


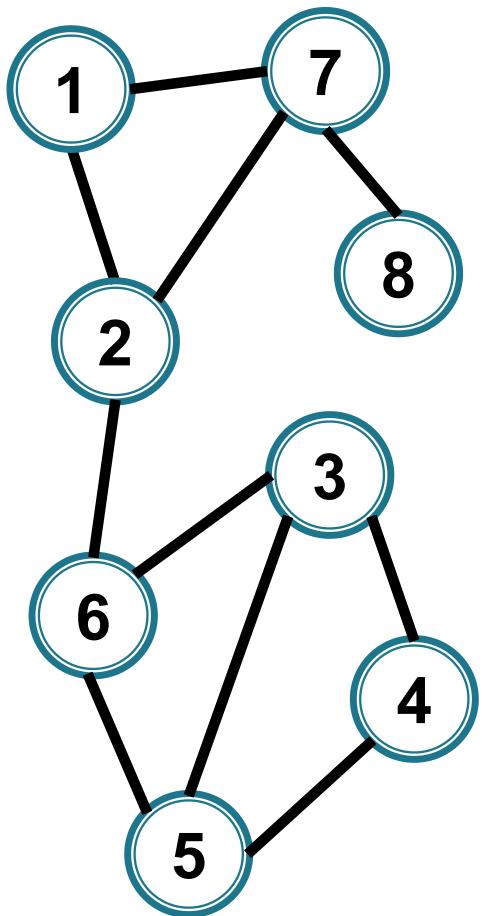
nivel/niv\_min



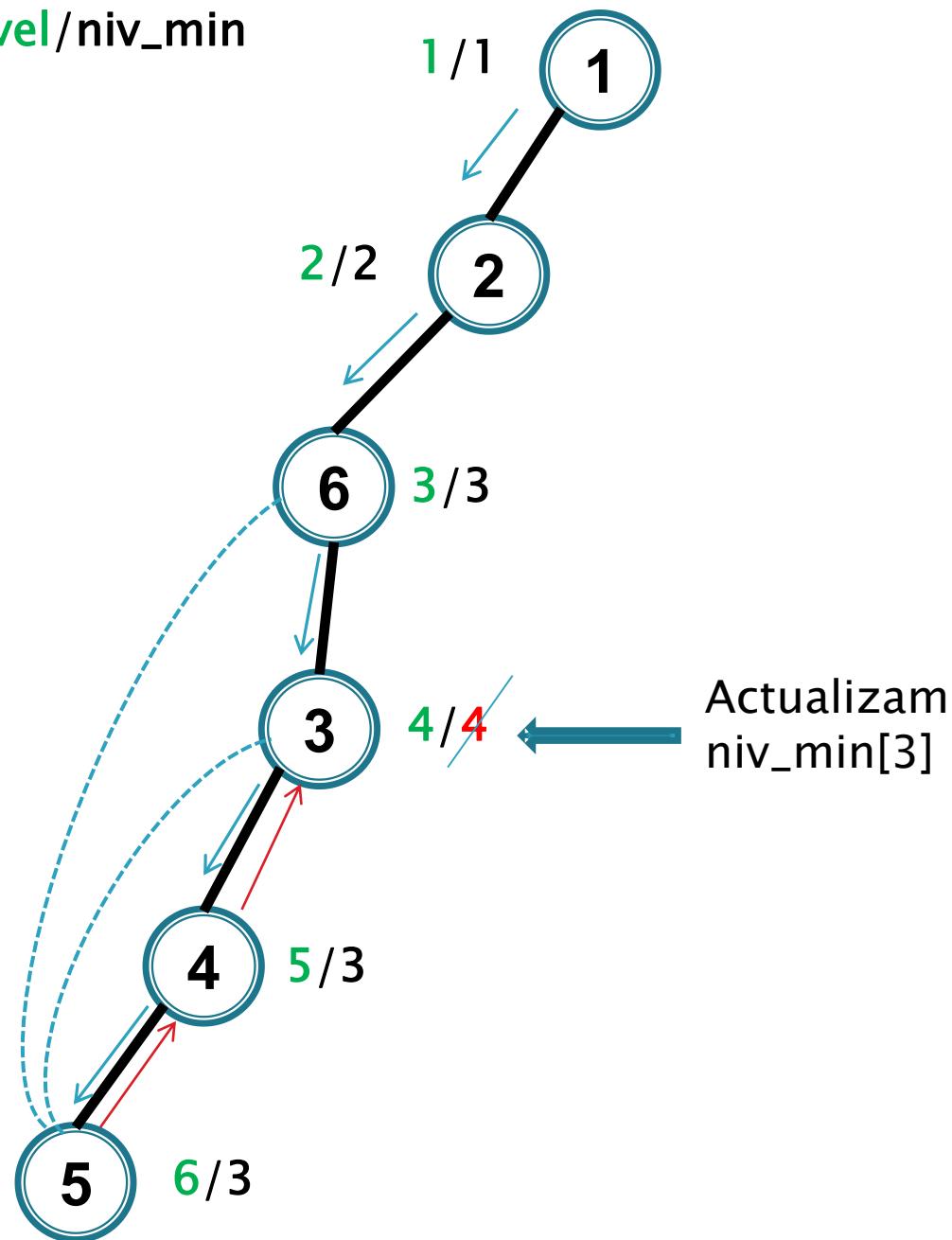


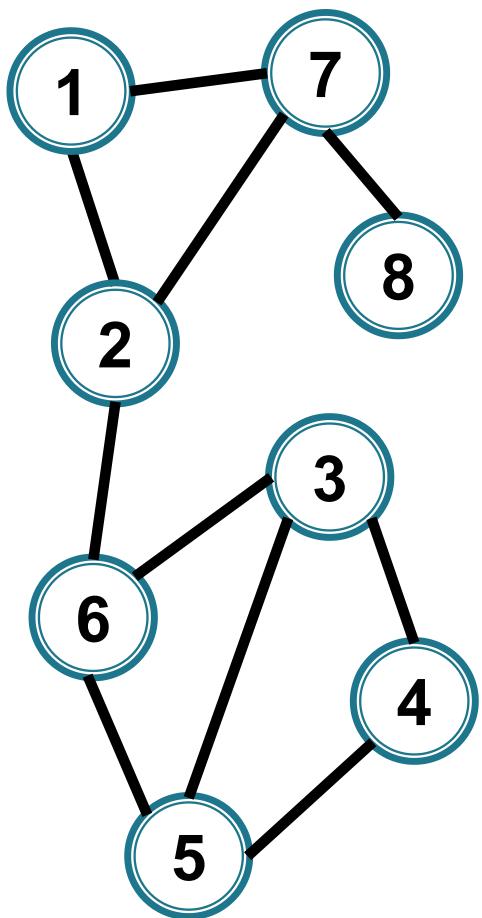
nivel/niv\_min



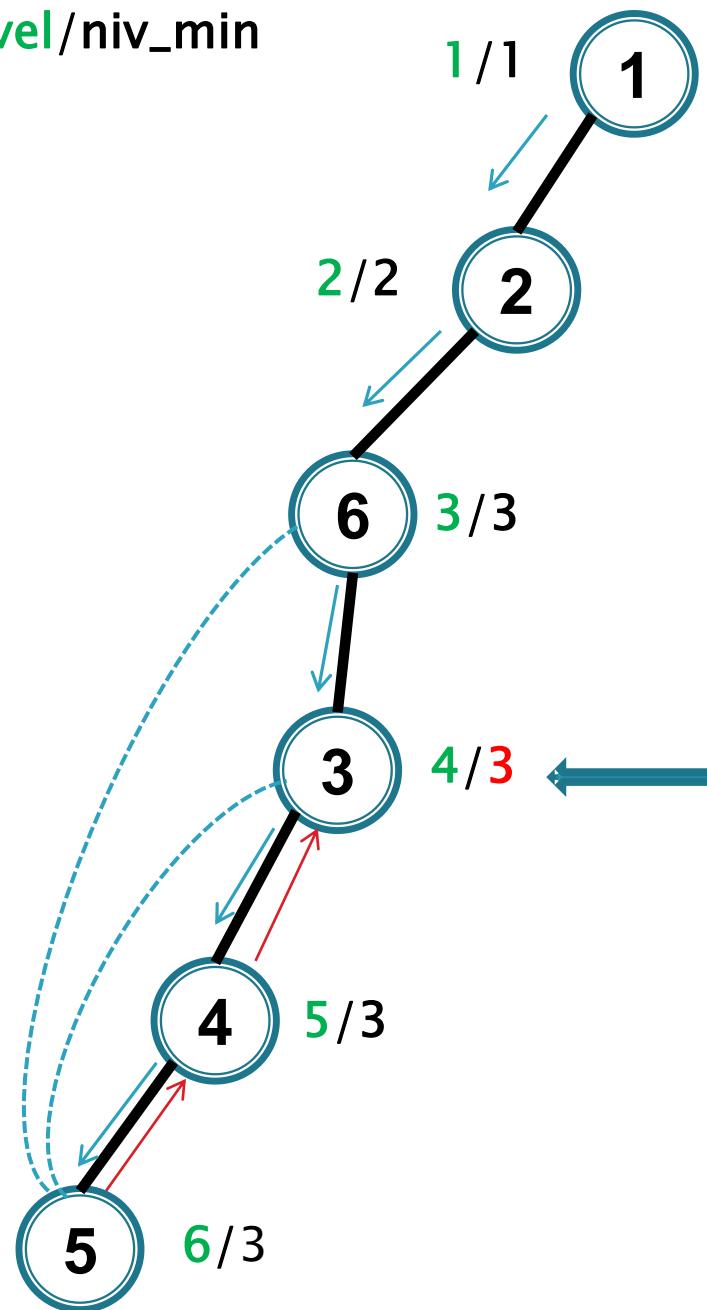


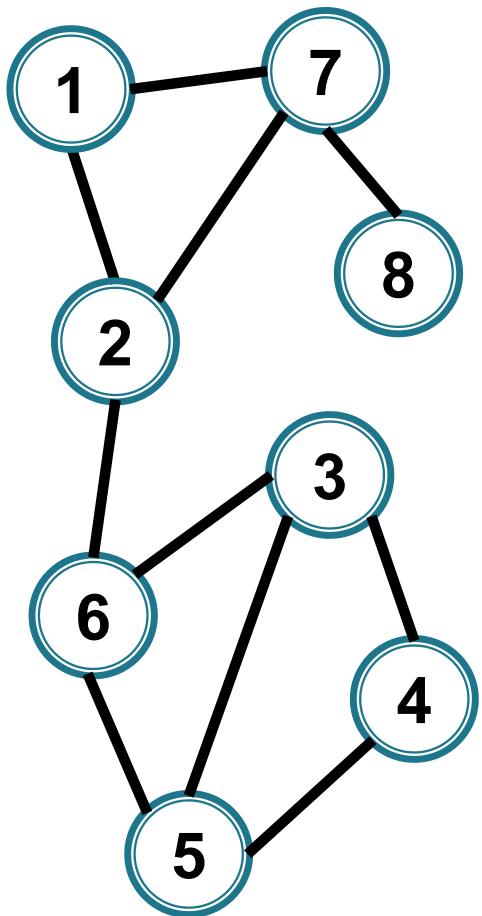
nivel/niv\_min



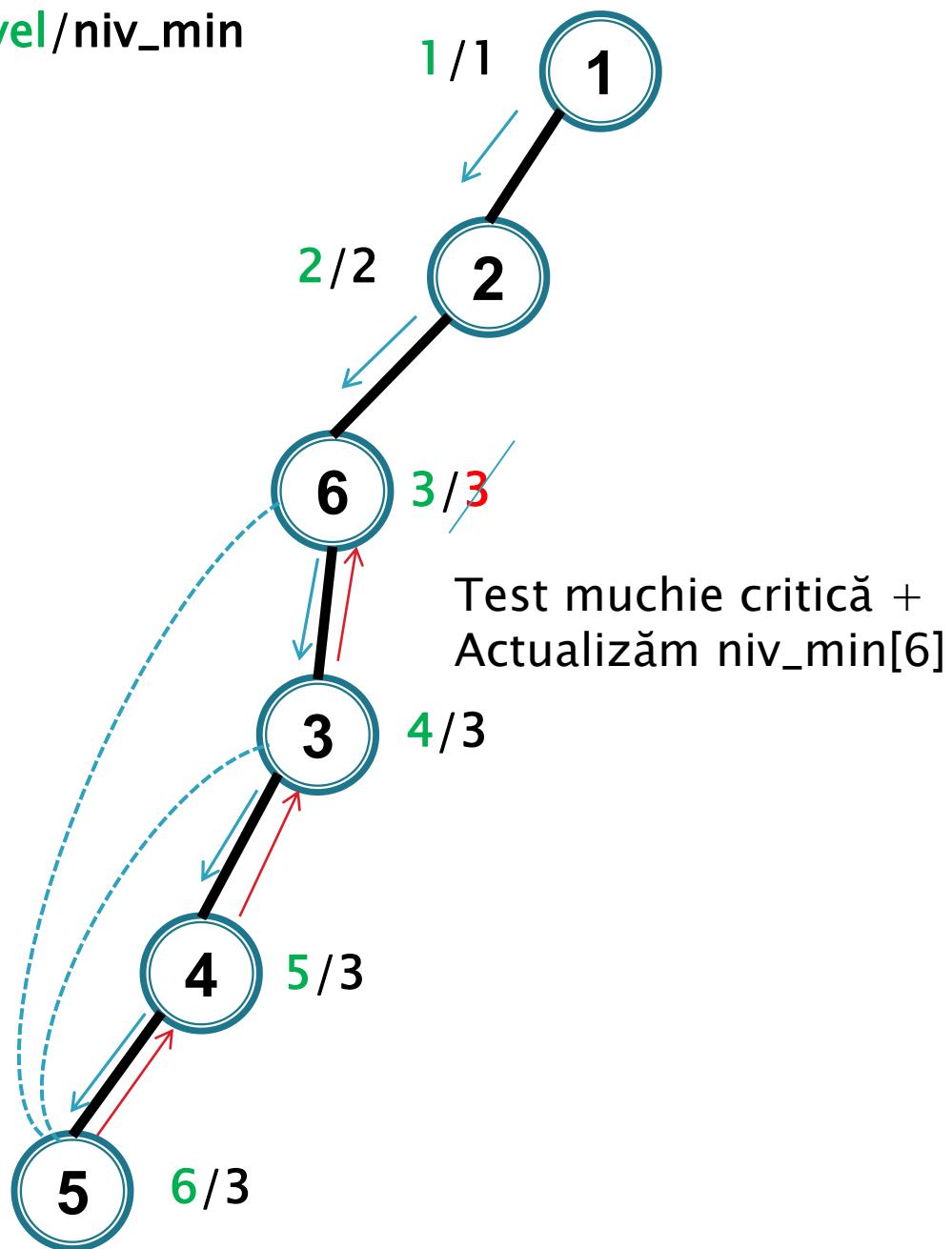


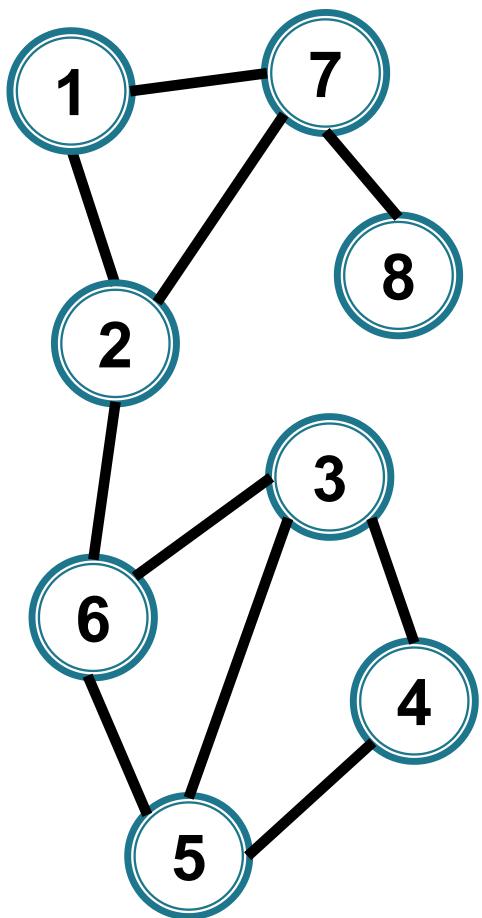
nivel/niv\_min



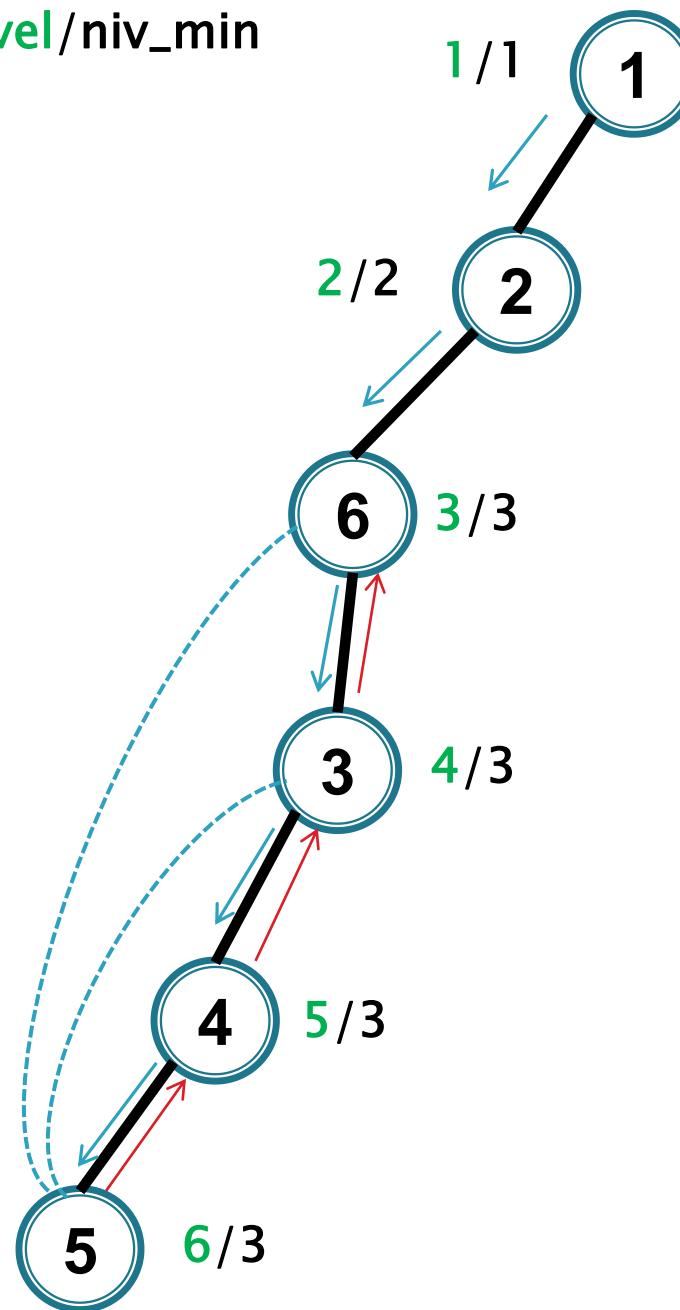


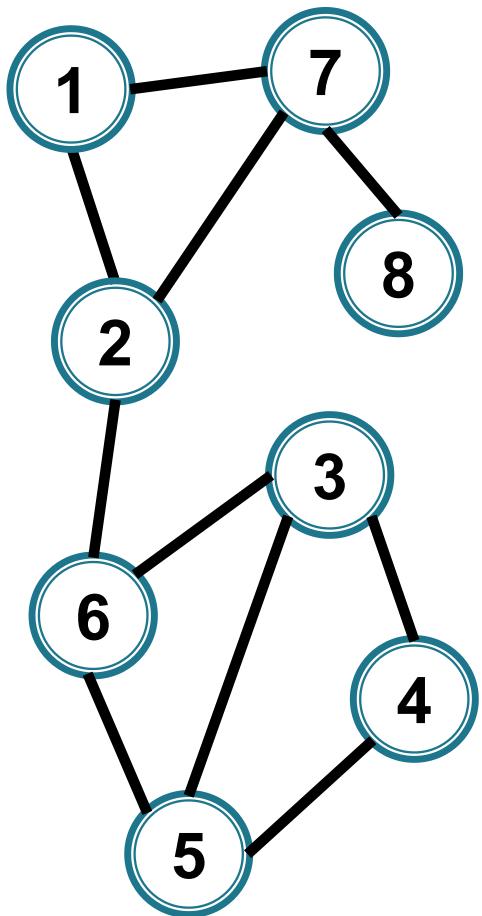
nivel/niv\_min



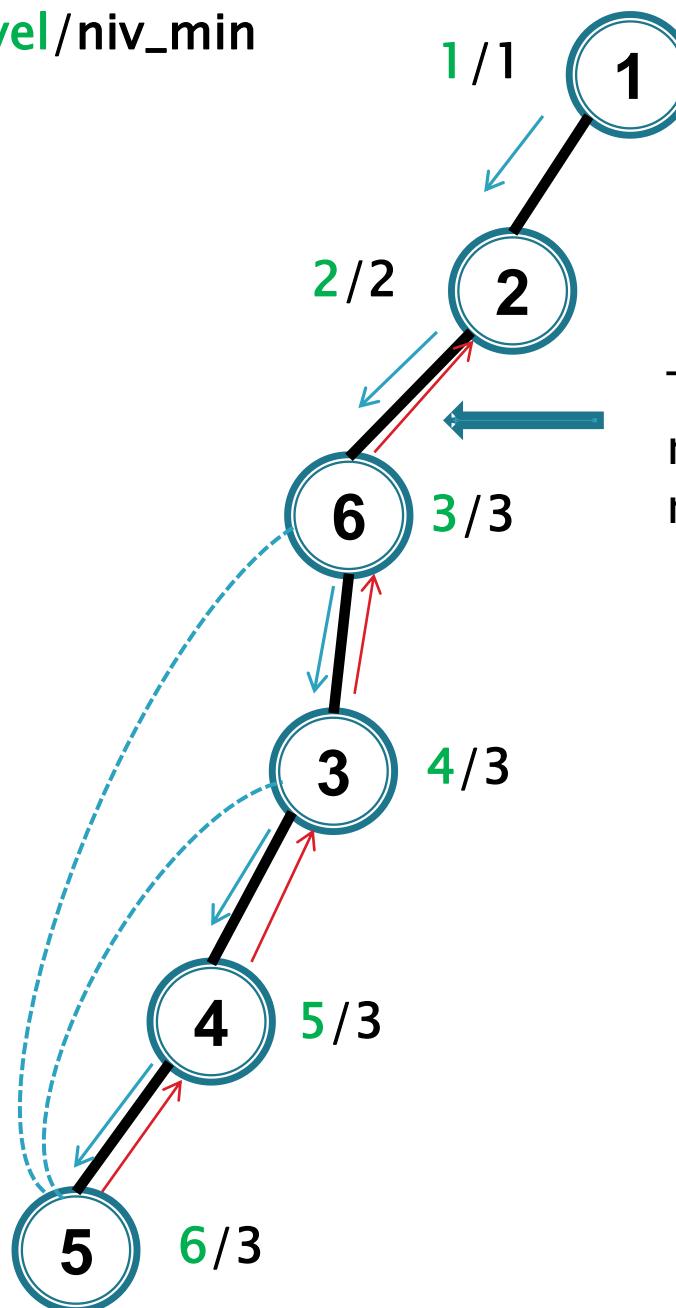


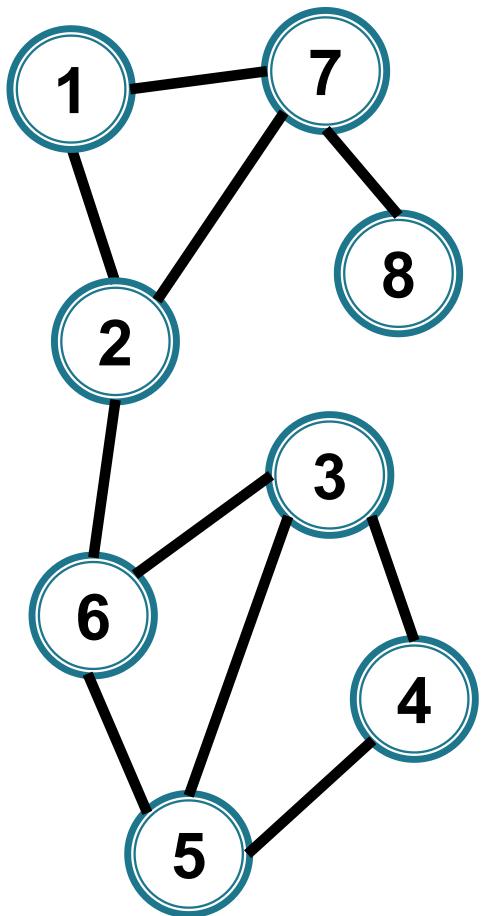
nivel/niv\_min



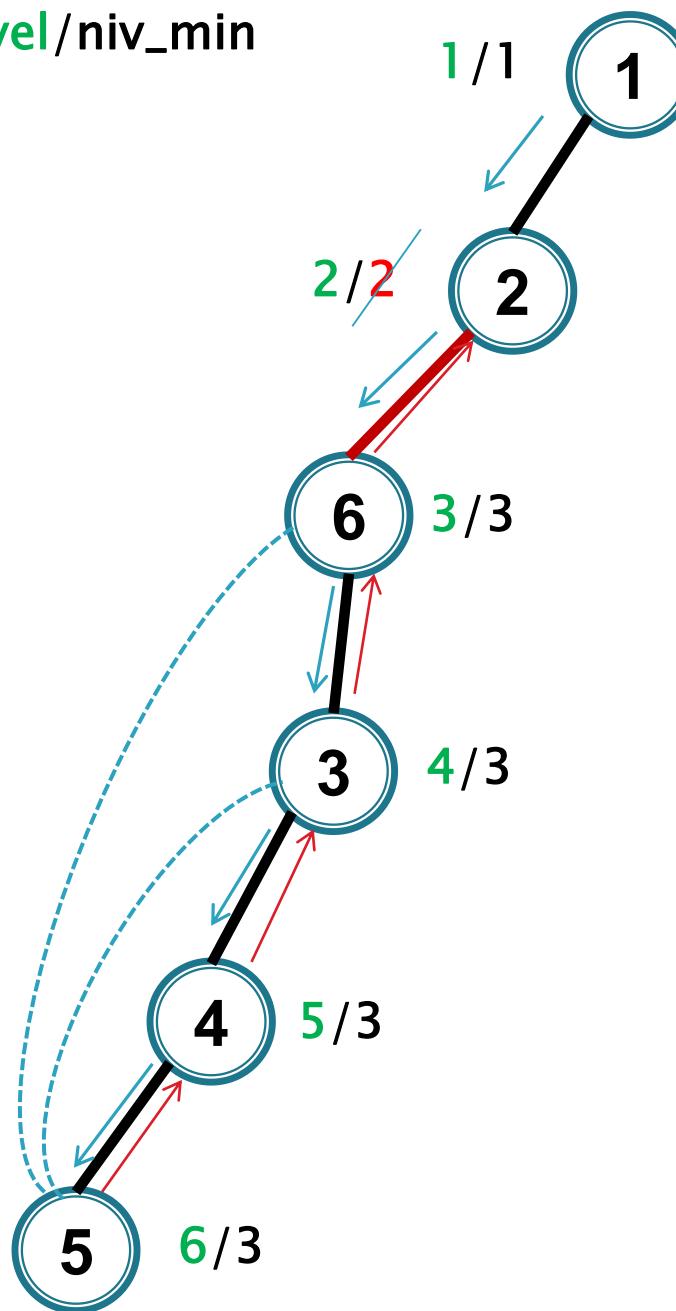


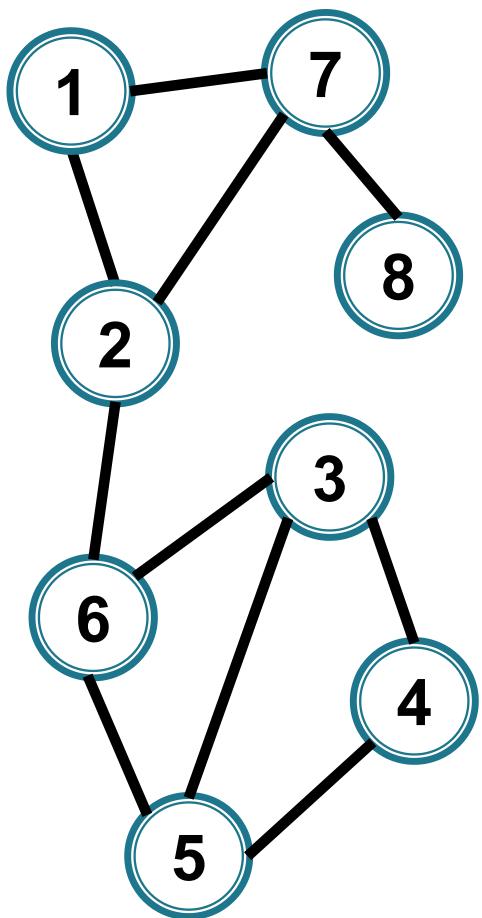
nivel/niv\_min



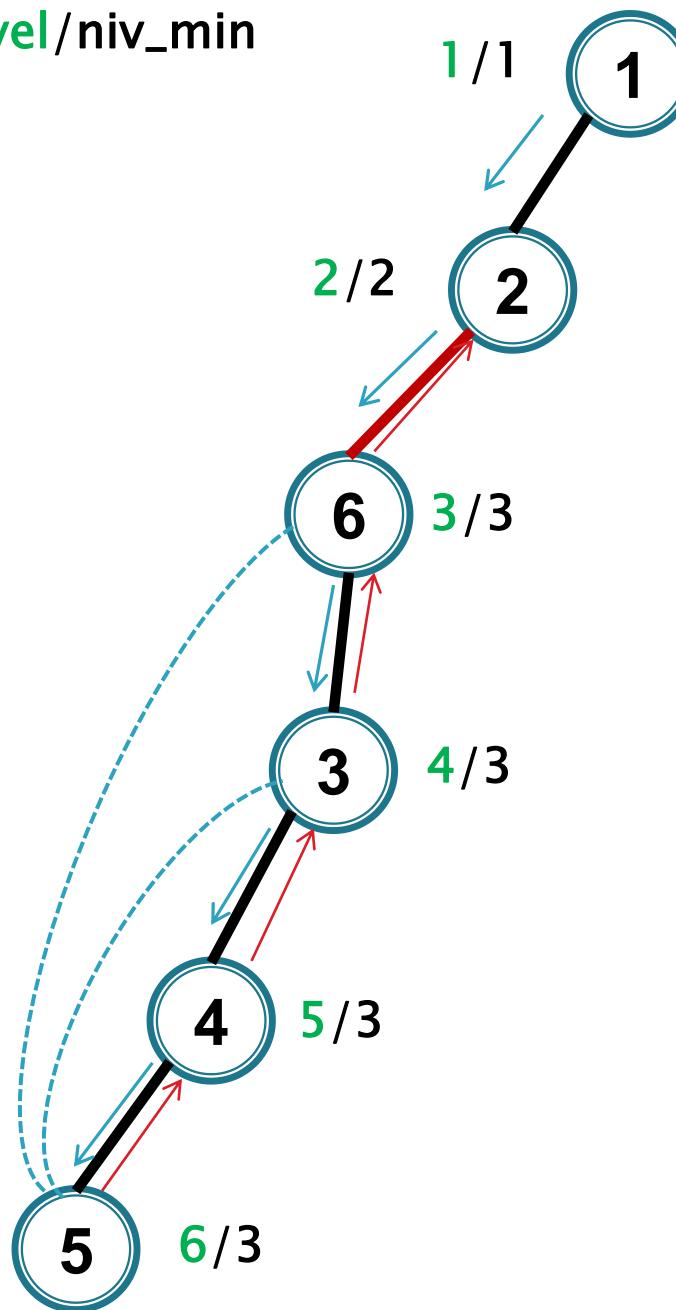


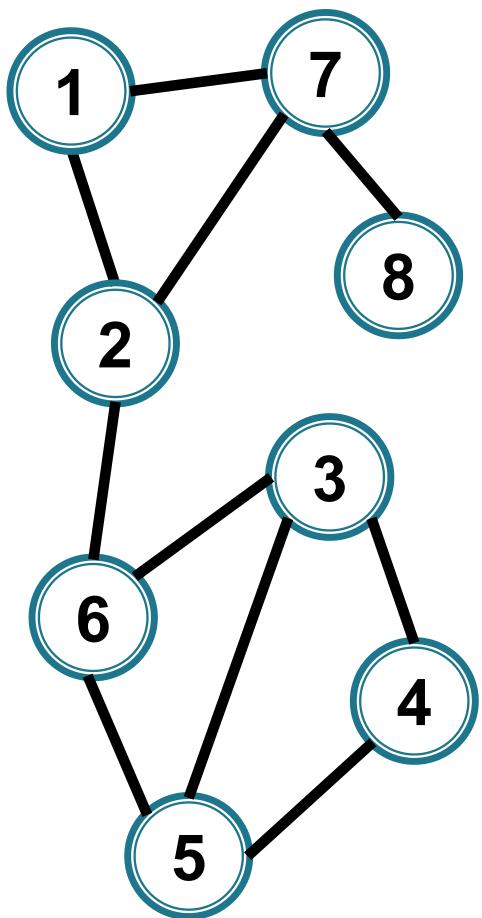
nivel/niv\_min



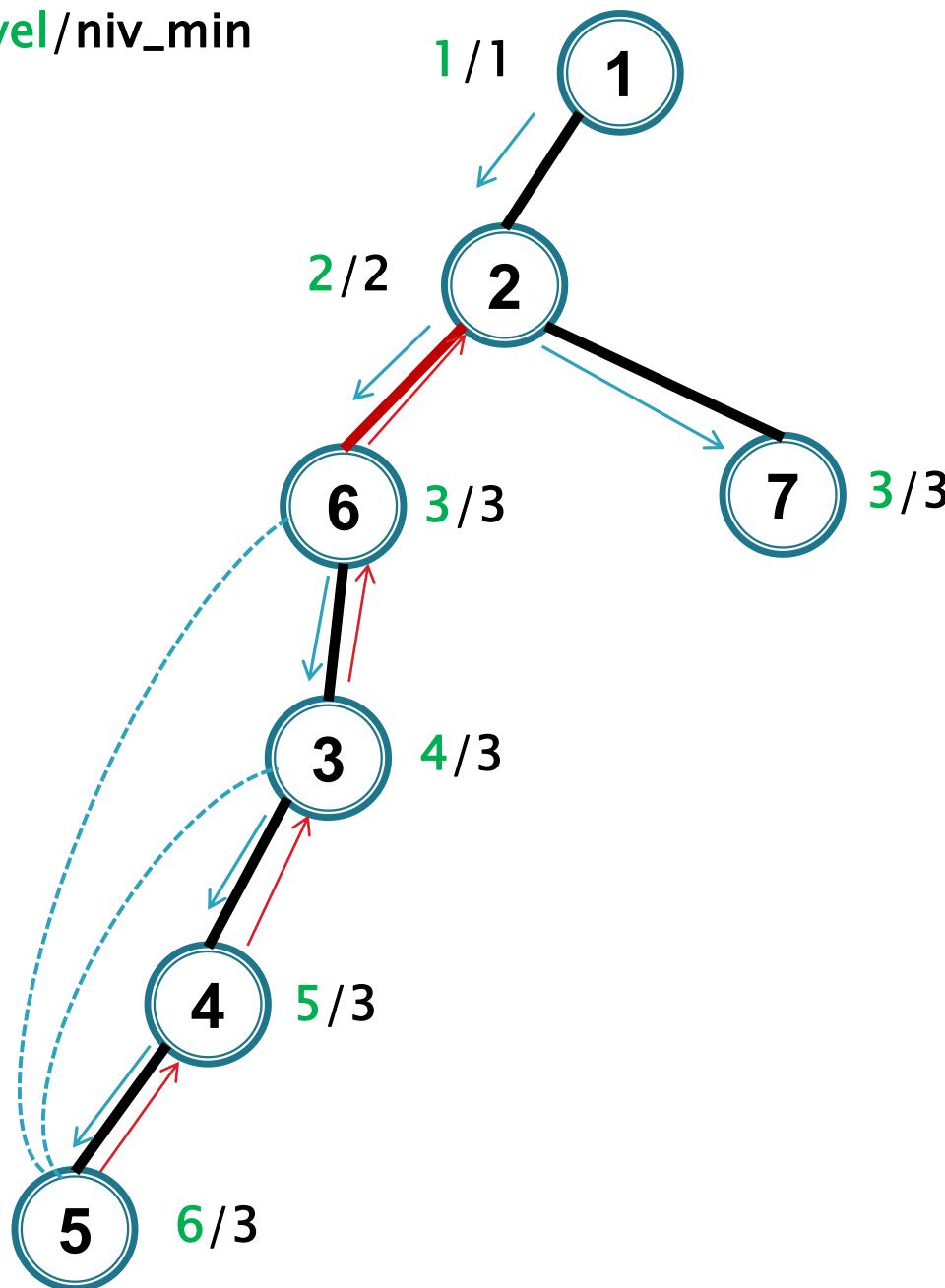


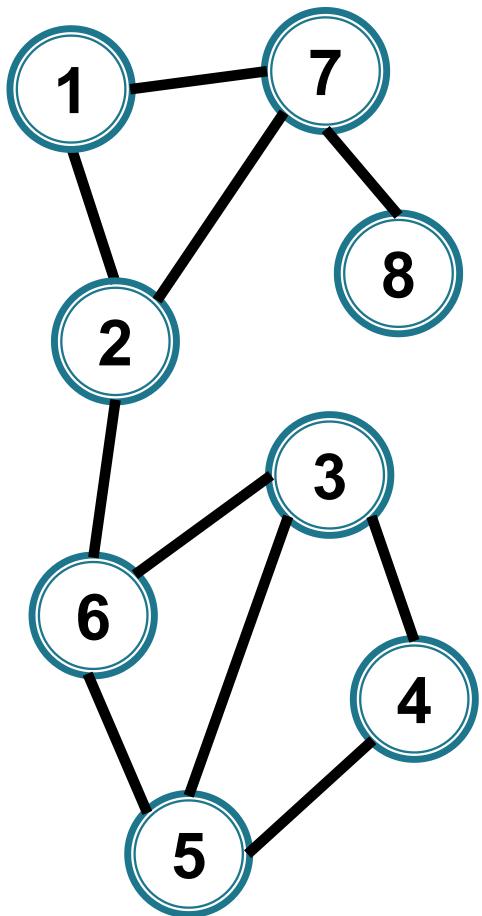
nivel/niv\_min



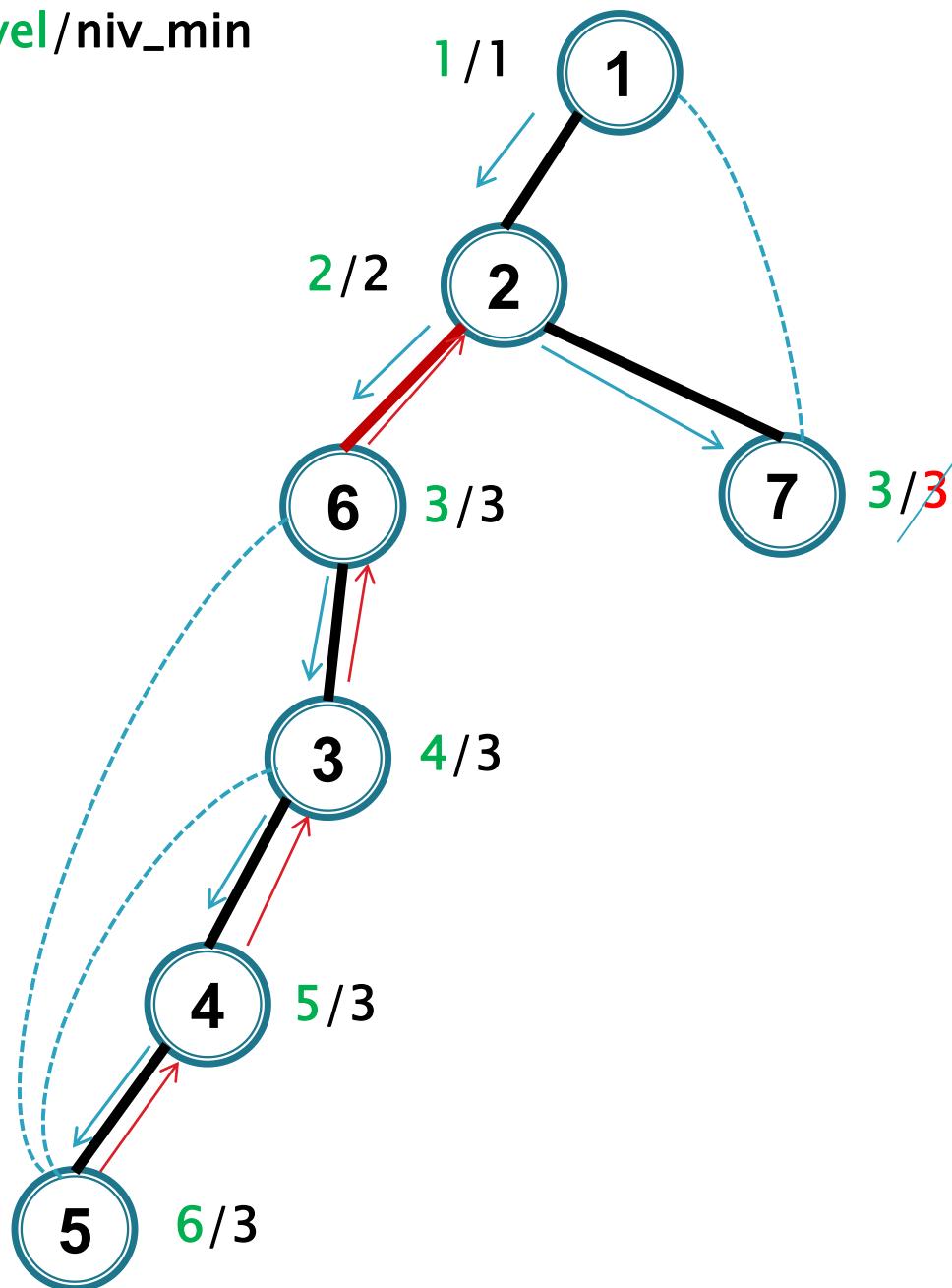


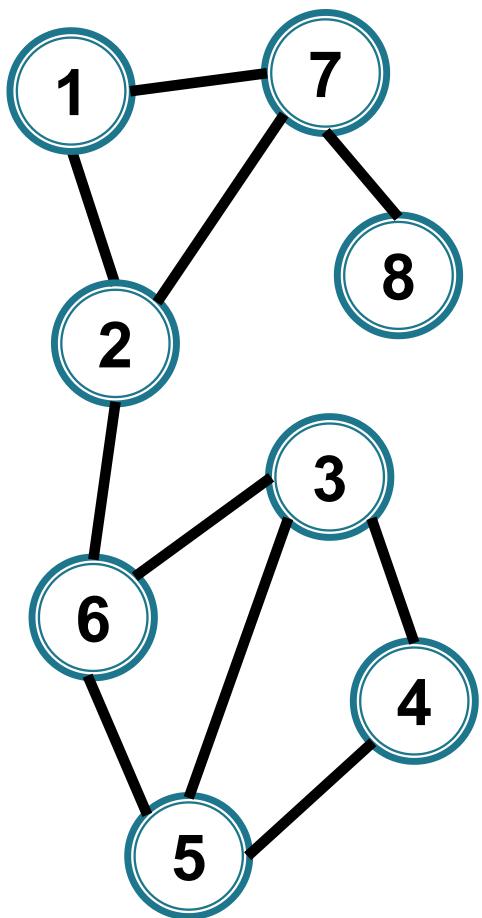
nivel/niv\_min



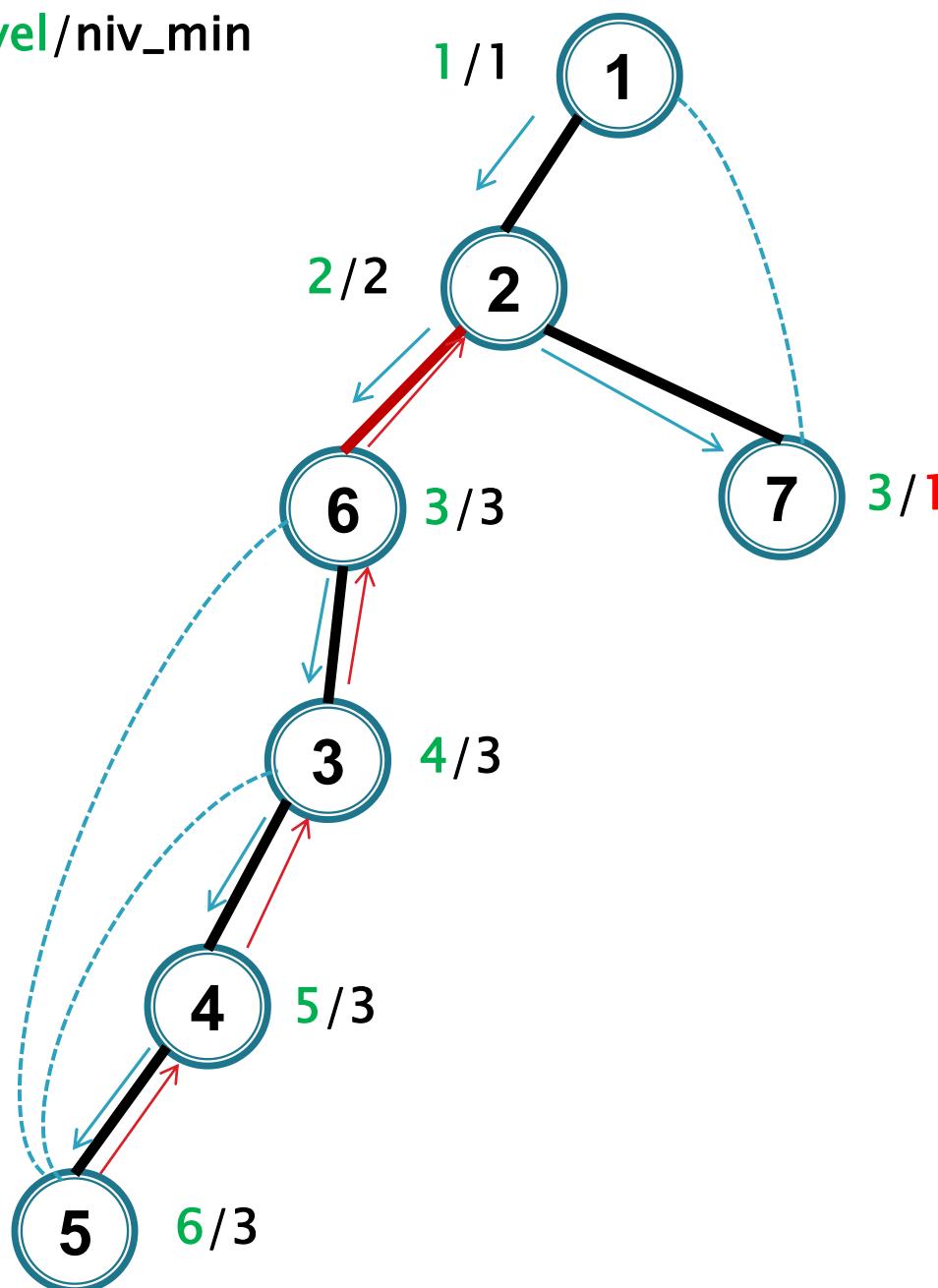


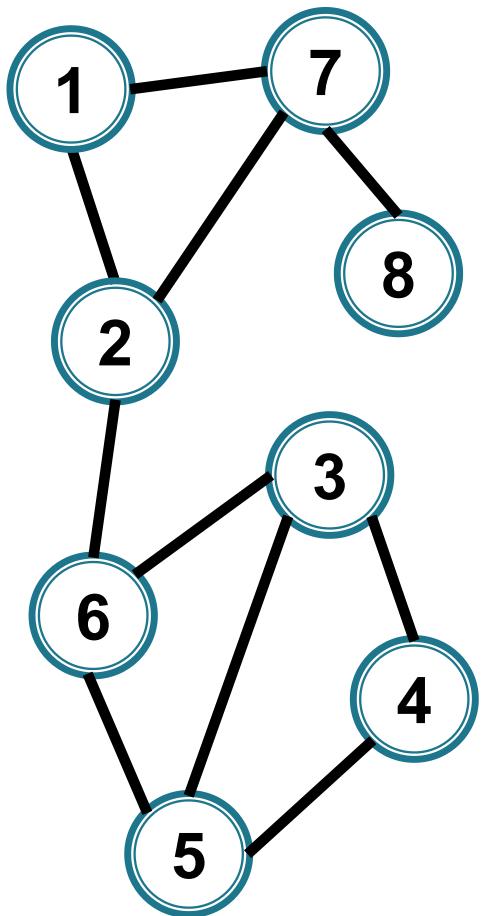
nivel/niv\_min



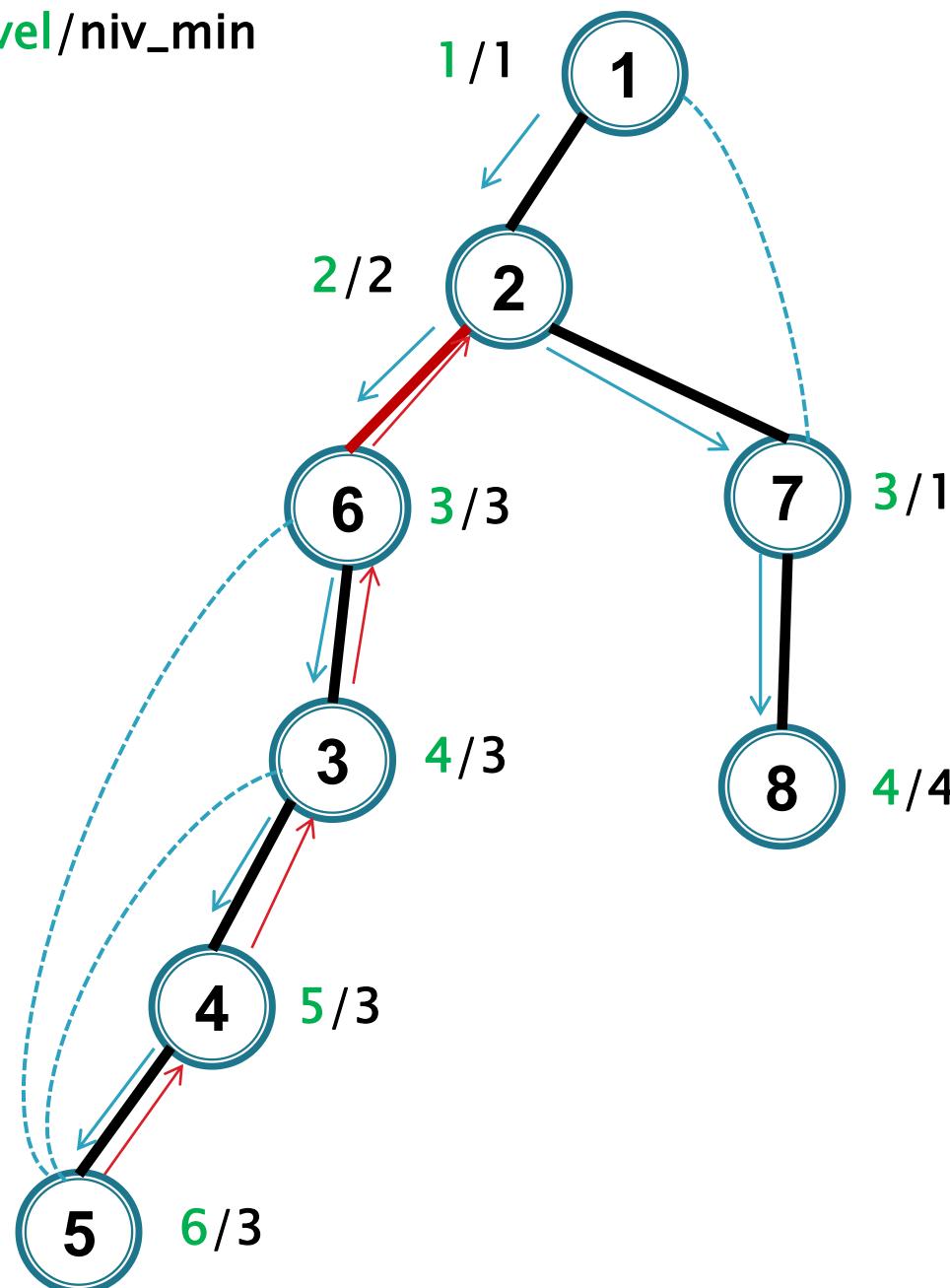


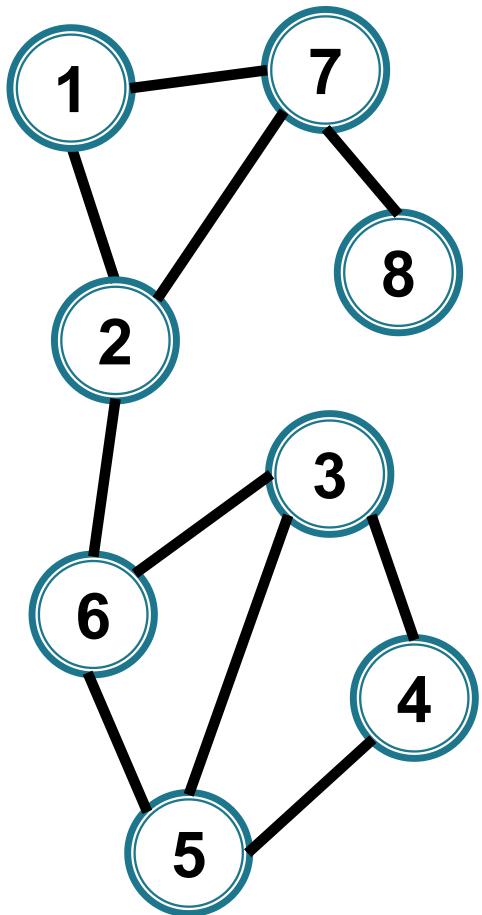
nivel/niv\_min



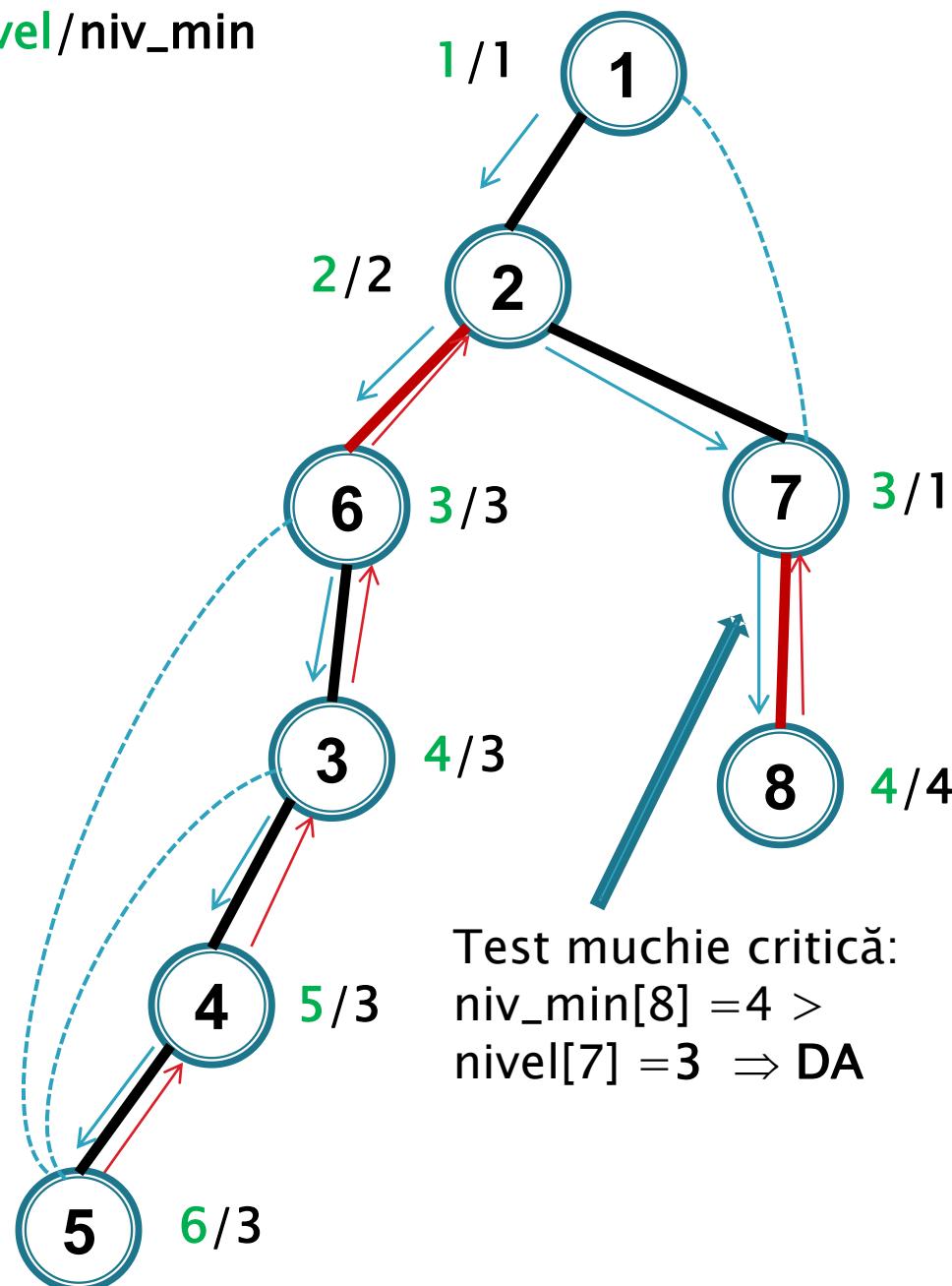


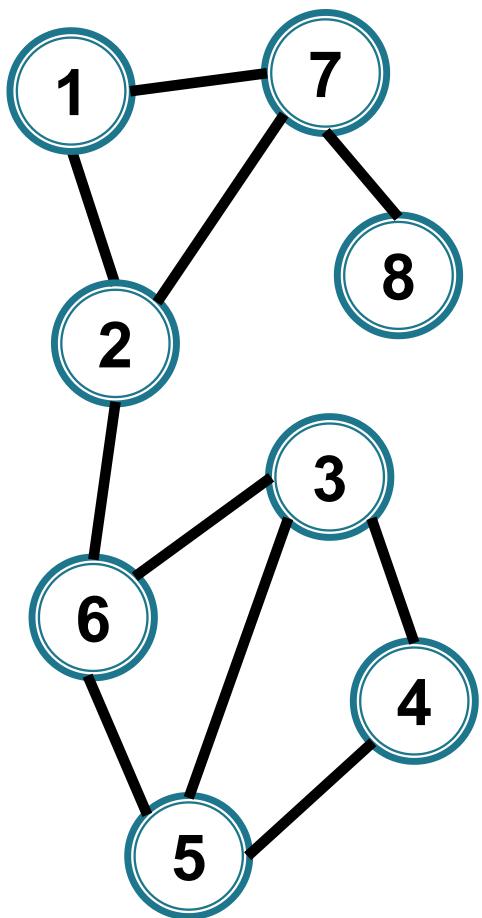
nivel/niv\_min



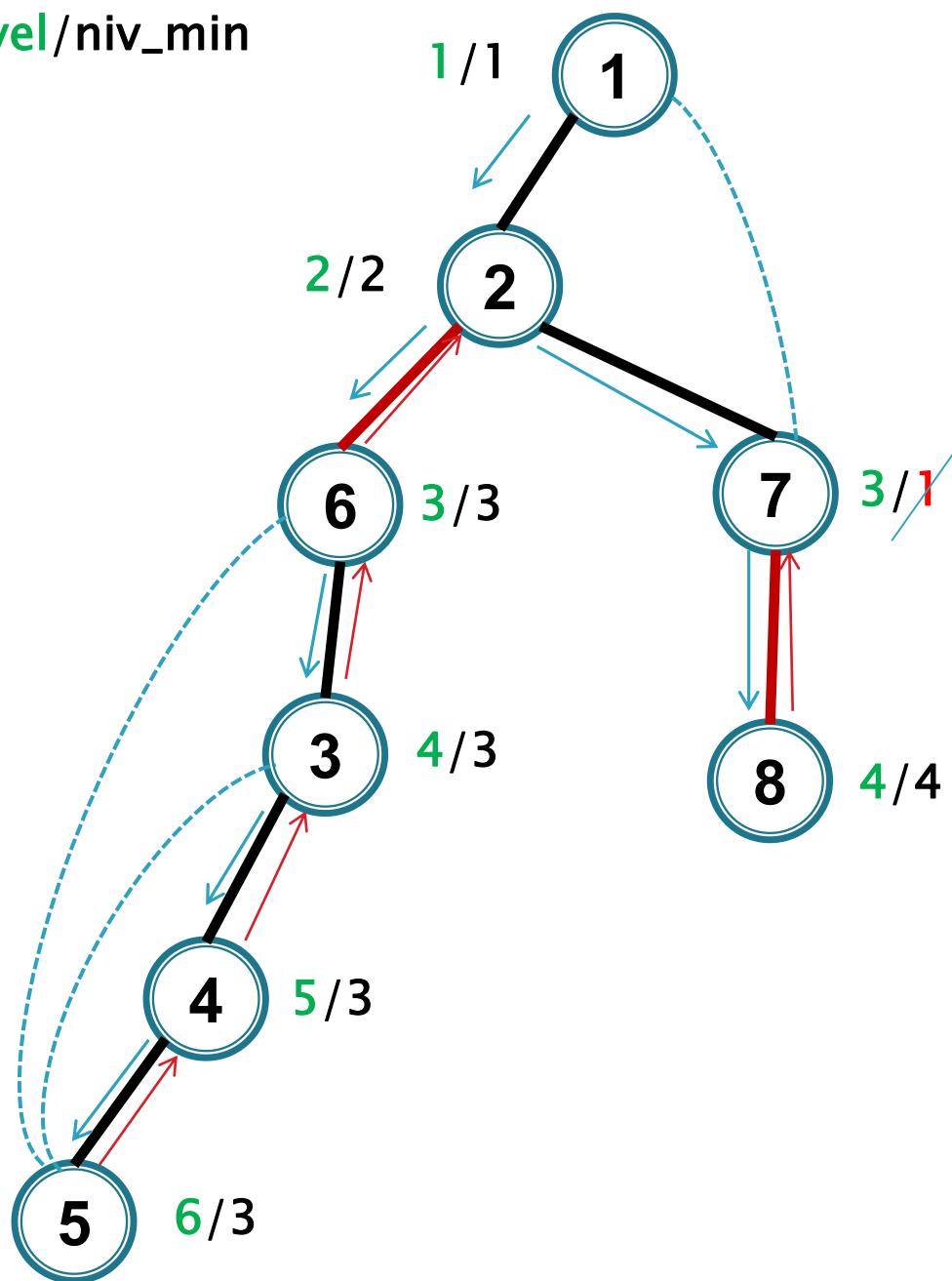


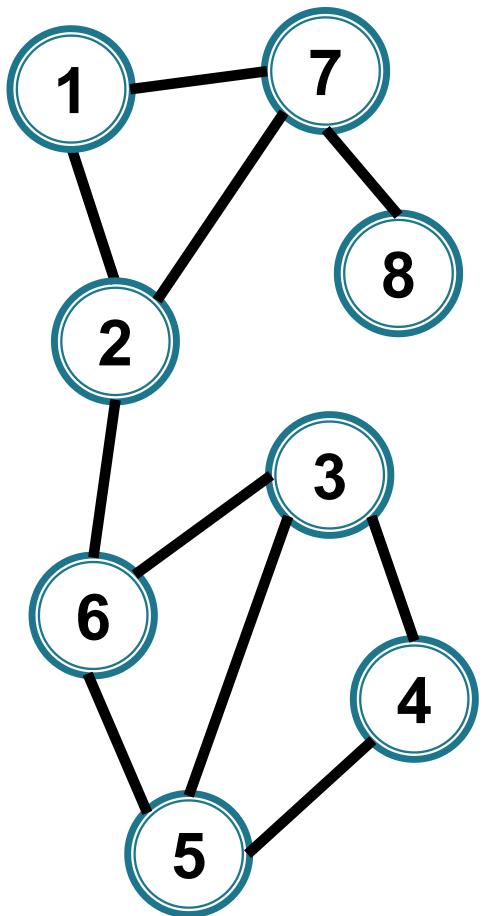
nivel/niv\_min



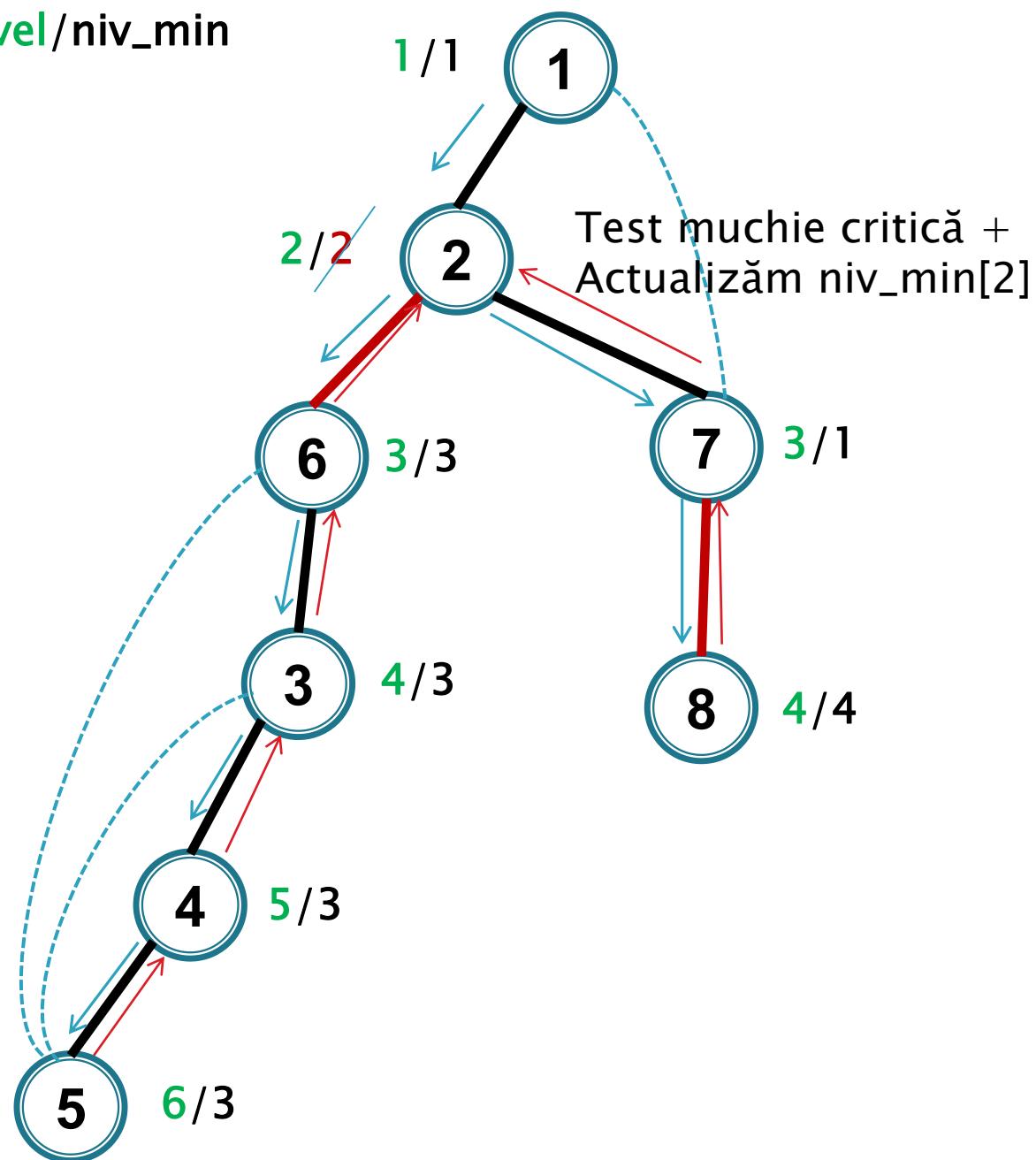


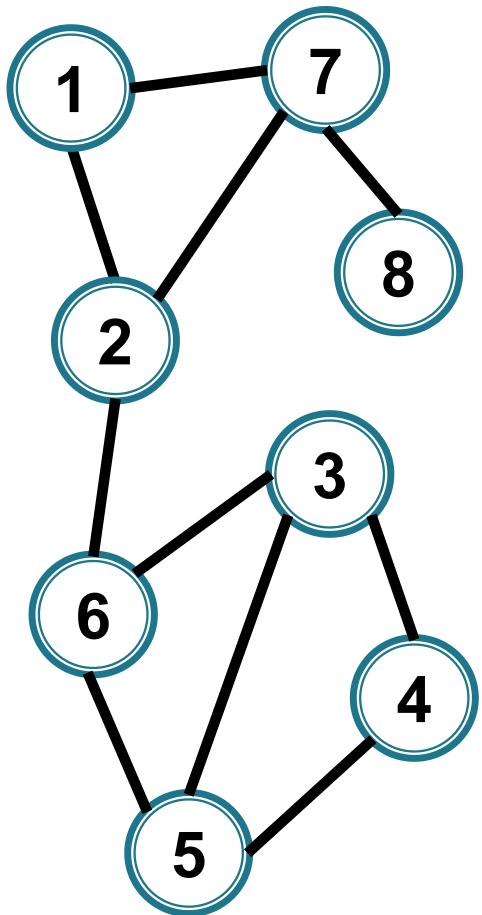
nivel/niv\_min



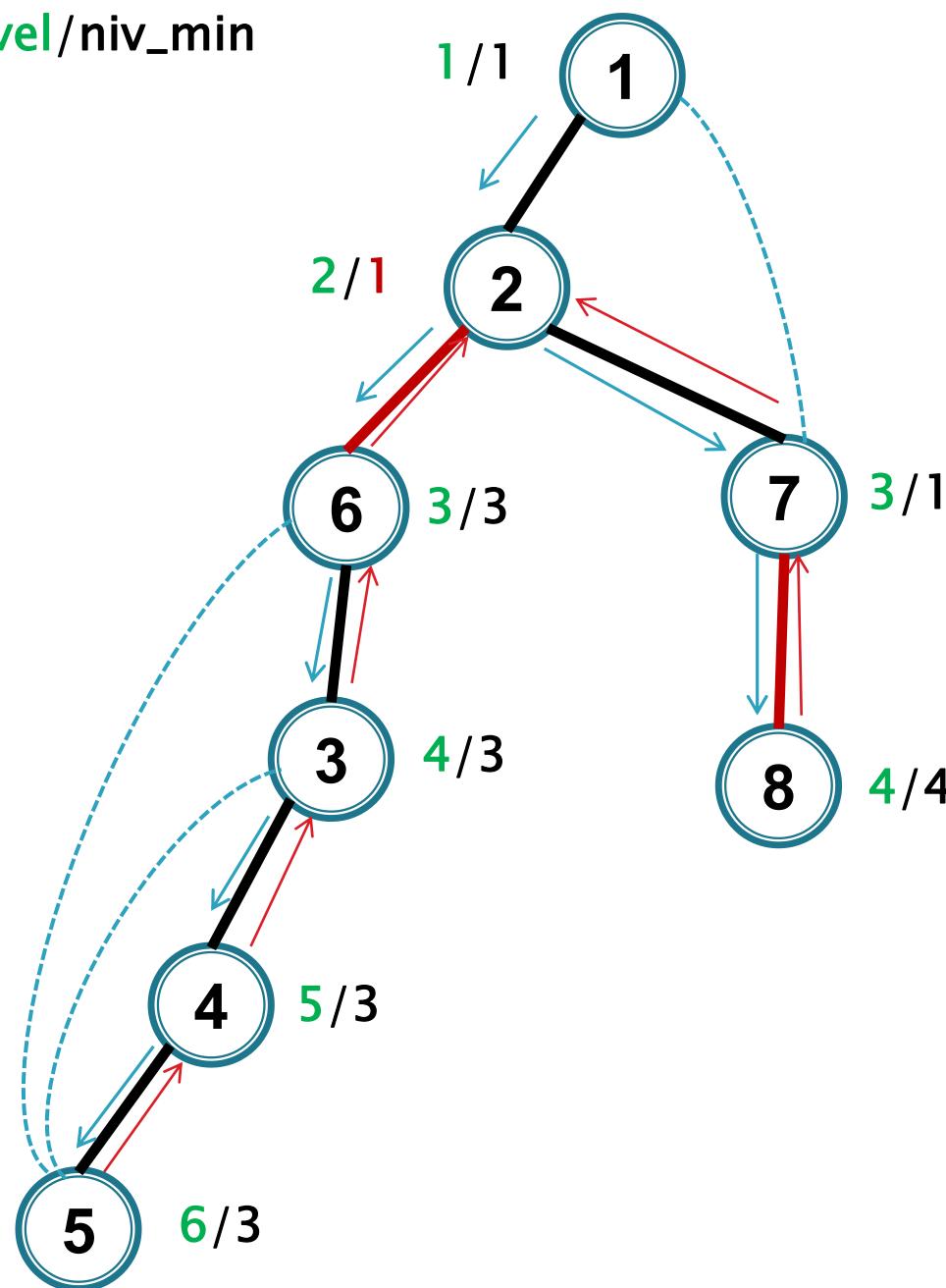


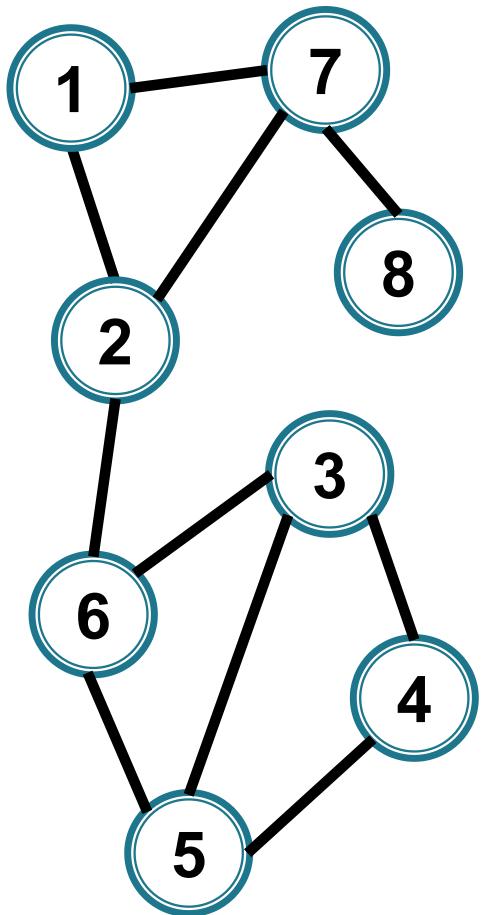
nivel/niv\_min





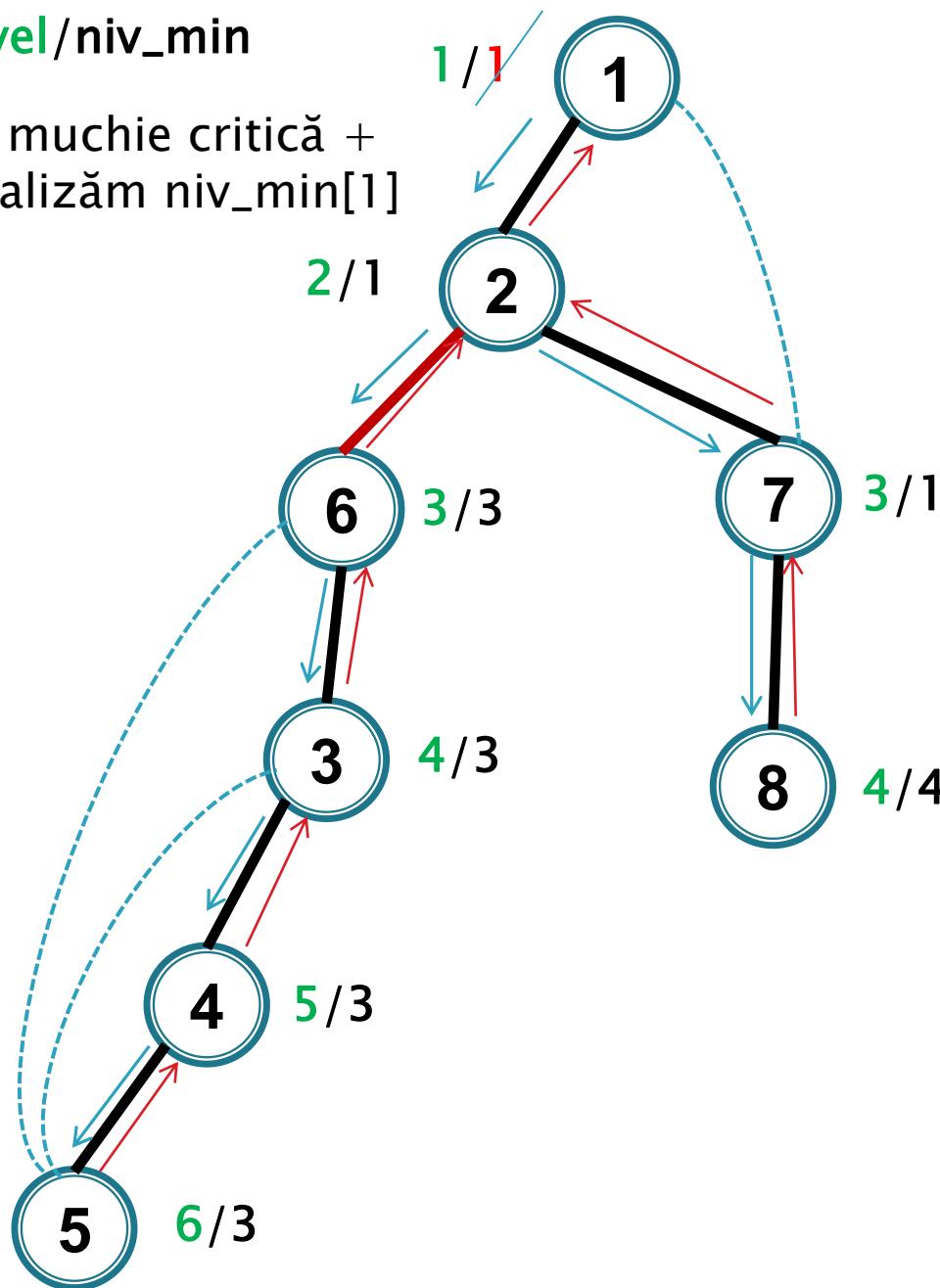
nivel/niv\_min

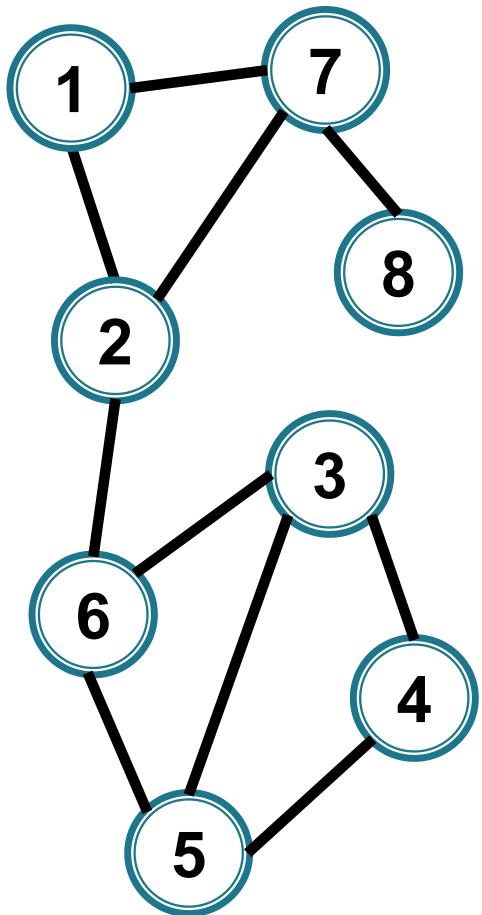




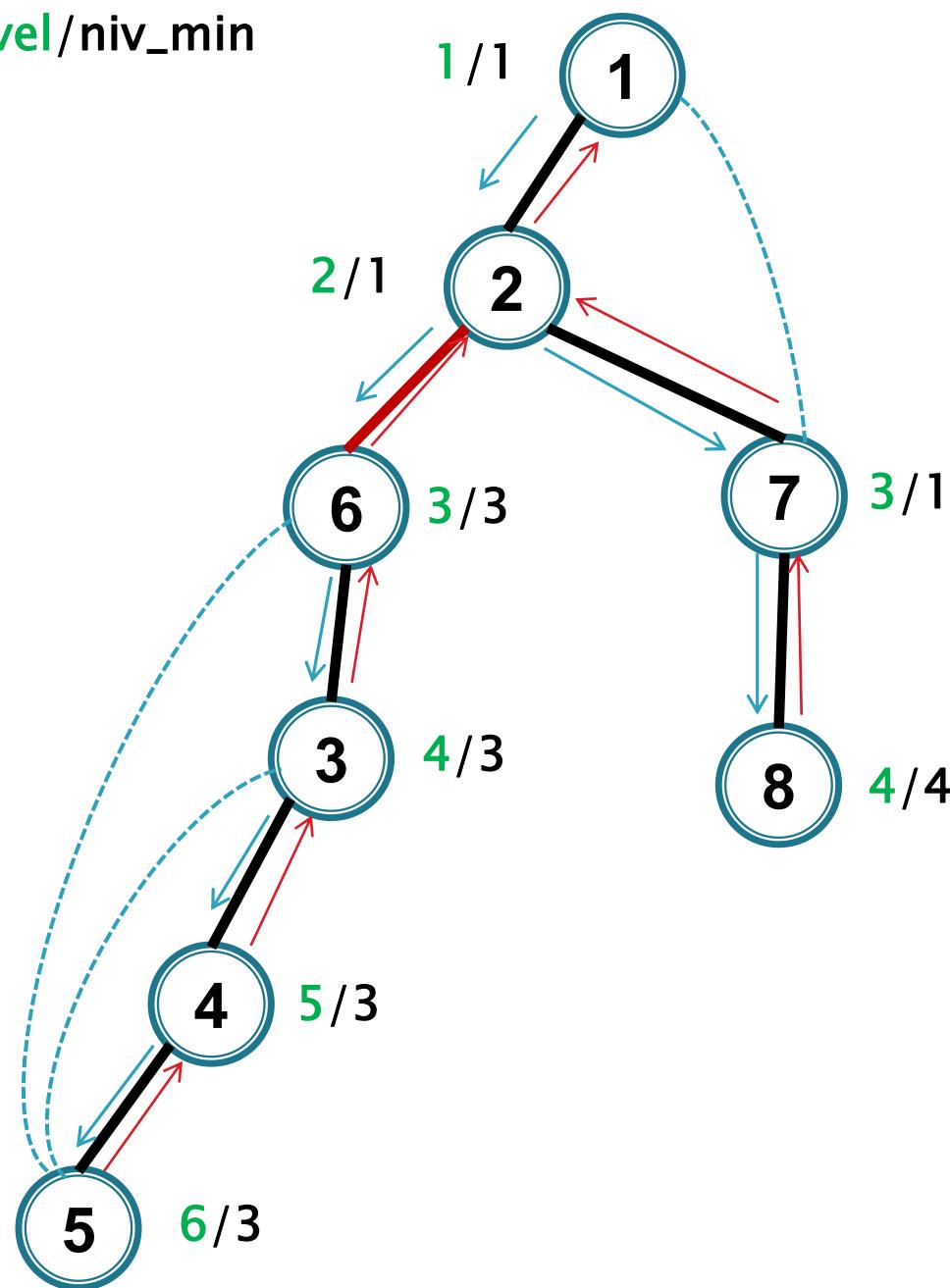
nivel/niv\_min

Test muchie critică +  
Actualizăm niv\_min[1]





nivel/niv\_min



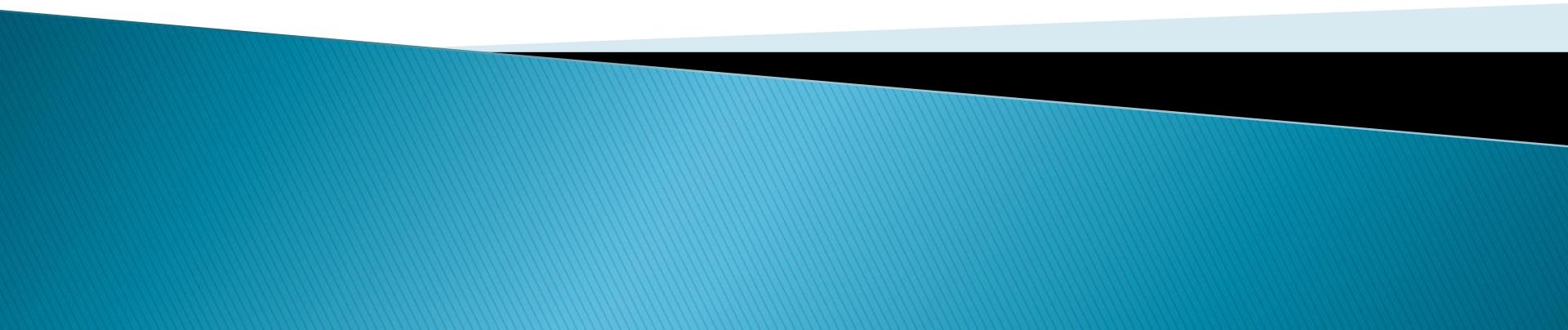
# Indicații implementare

```
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 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



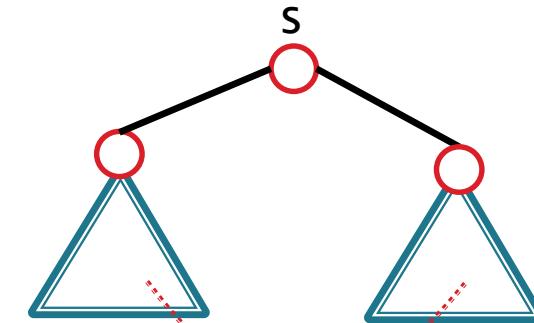
# Puncte critice

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

- ▶ Arborele DF

- rădăcina  $s$  este punct critic  $\Leftrightarrow$

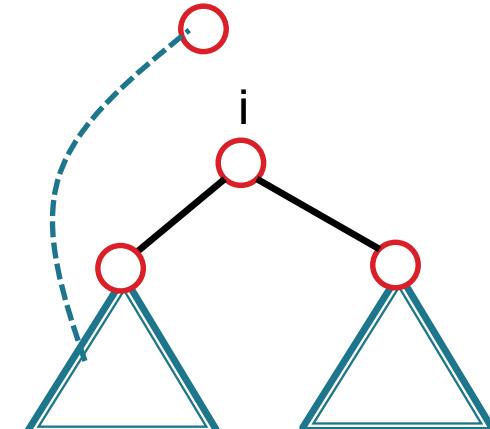
?



nu există muchii între subarbore (de traversare)

- un alt vârf  $i$  din arbore este critic  $\Leftrightarrow$

?



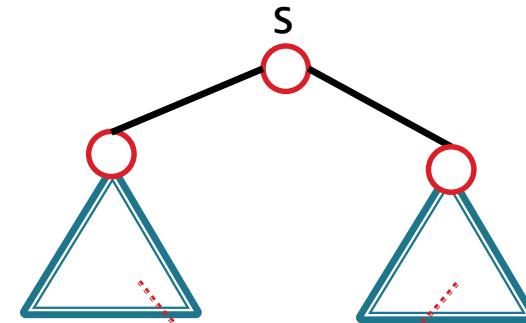
# Puncte critice

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

## ▶ Arborele DF

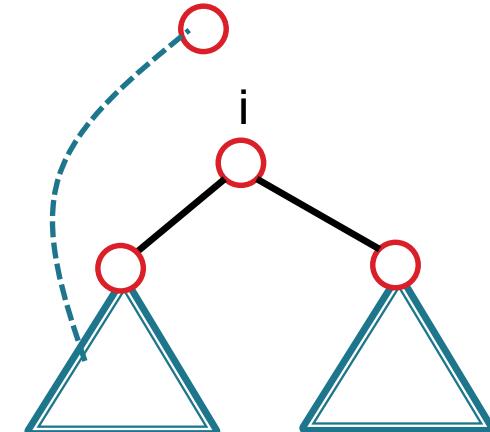
- rădăcina  $s$  este punct critic  $\Leftrightarrow$

**are cel puțin 2 fii în arborele DF**



nu există muchii între subarbore (de traversare)

- un alt vârf  $i$  din arbore este critic  $\Leftrightarrow$   
**are cel puțin un fiu  $j$  cu  $niv\_min[j] \geq nivel[i]$**



# Componente tare conexă

# Componente tare conexe

Într-un graf orientat avem 2 definiții de conexitate.

Un graf orientat este **slab conex** dacă există un drum de la oricare nod la oricare altul **considerând muchile grafului neorientate**.

Un graf orientat este **tare conex** dacă există un drum de la oricare nod la oricare altul.

# Componente tare conexe

Într-un graf orientat avem 2 definiții de conexitate.

Un graf orientat este **slab conex** dacă există un drum de la oricare nod la oricare altul **considerand muchile grafului neorientate**.

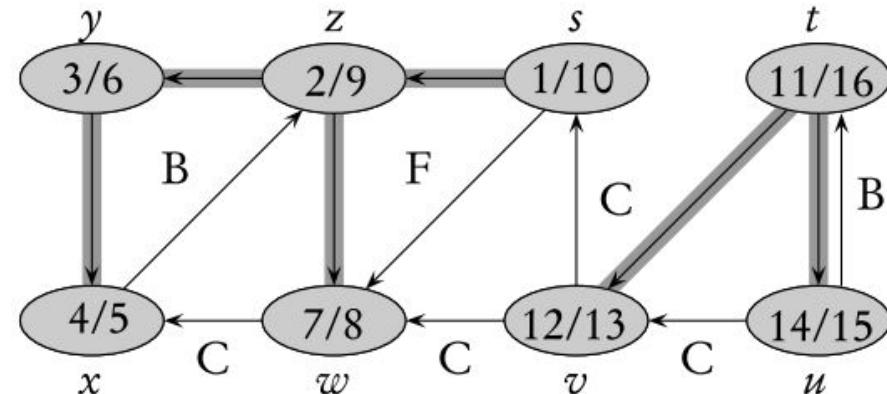
Un graf orientat este **tare conex** dacă există un drum de la oricare nod la oricare altul.

Graful este slab conex

Graful nu este tare conex

- drumul  $s \rightarrow v$  nu există

(a)



# Componente tare conexe: algoritm

- Următorul algoritm de timp liniar (adică  $\Theta(V + E)$ ) determină componentele tare conexe ale unui graf orientat  $G = (V, E)$  folosind două căutări în adâncime, una în  $G$  și una în  $GT$ .
- Componete-Tare-Conexe( $G$ )
- 1: apeleză  $CA(G)$  pentru a calcula timpii de terminare  $f[u]$  pentru fiecare vârf  $u$
- 2: calculează  $GT$
- 3: apeleză  $CA(GT)$ , dar în bucla principală a lui  $CA$ , consideră vâfurile în ordinea descrescătoare a timpilor  $f[u]$  (calculați în linia 1)
- 4: afișeză vâfurile fiecărui arbore în pădurea de adâncime din pasul 3 că o componentă tare conexă separată

# Componente tare conexe: algoritm Kosaraju

- Următorul algoritm de timp liniar (adică  $\Theta(V + E)$ ) determină componentele tare conexe ale unui graf orientat  $G = (V, E)$  folosind două căutări în adâncime, una în  $G$  și una în  $GT$ .
- Componete-Tare-Conexe( $G$ )
- 1: apeleză  $CA(G)$  pentru a calcula timpii de terminare  $f[u]$  pentru fiecare vârf  $u$
- 2: calculează  $GT$
- 3: apeleză  $CA(GT)$ , dar în bucla principală a lui  $CA$ , consideră vâfurile în ordinea descrescătoare a timpilor  $f[u]$  (calculați în linia 1)
- 4: afișeză vâfurile fiecărui arbore în pădurea de adâncime din pasul 3 că o componentă tare conexă separată

# Componente tare conexe: algoritm Kosaraju

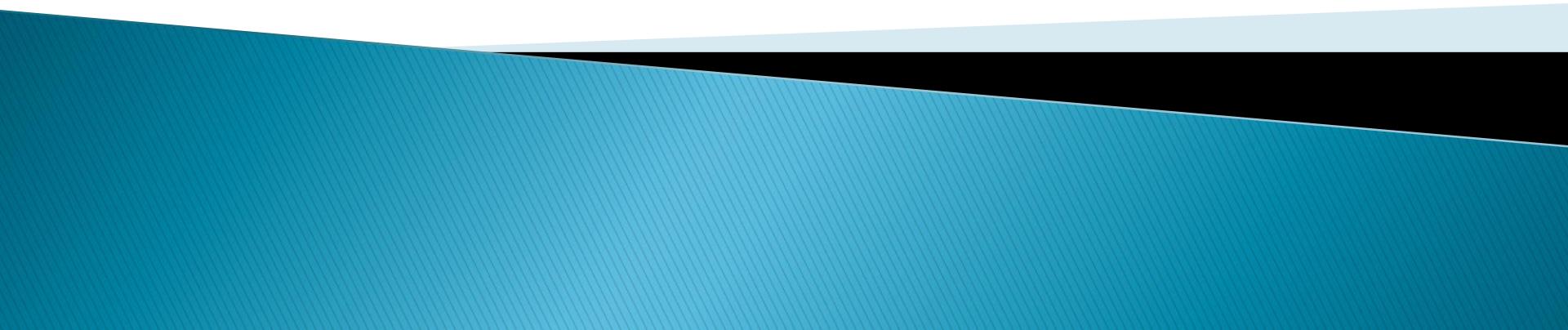
- 90-100 p pe infoarena :)



# Componente tare conexe: schita demonstratie

- Lema: Dacă două vârfuri se află în aceeași componentă tare conexă, atunci nici un drum între ele nu părăsește, vreodată, componentă tare conexă.
- Demonstrație: Fie  $u$  și  $v$  două noduri din componenta tare conexă.  
Presupunem ca există  $w$  în afara componentei și există drum  $u \rightarrow v$  prin  $w$ .  
Atunci avem drum de la  $u$  la  $w$  dar avem și drumul  $w \rightarrow v \rightarrow u$  deci și drum de la  $w$  la  $u$  deci  $w$  este în componenta tare conexă.

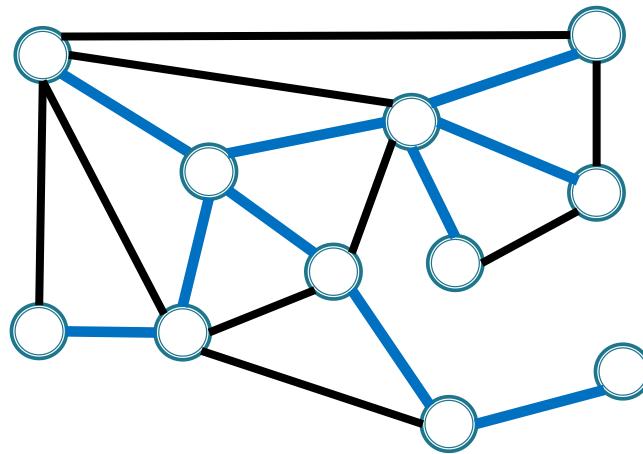
# Arbore parțiali



# 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)

# 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)
$T \leftarrow (V, \emptyset)$ cat timp $T$ nu este conex executa <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> returneaza $T$	

# 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)
$T \leftarrow (V, \emptyset)$ cat timp $T$ nu este conex executa <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> returneaza $T$	
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 (bottom - up)	Prin eliminare de muchii (cut - down)
$T \leftarrow (V, \emptyset)$ cat timp $T$ nu este conex executa <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> returneaza $T$	$T \leftarrow (V, E)$ cat timp $T$ conține cicluri executa <ul style="list-style-type: none"><li>• alege <math>e \in E(T)</math> o muchie dintr-un ciclu</li><li>• <math>E(T) \leftarrow E(T) - \{e\}</math></li></ul> returneaza $T$
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 (bottom - up)	Prin eliminare de muchii (cut - down)
$T \leftarrow (V, \emptyset)$ cat timp $T$ nu este conex executa <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> returneaza $T$	$T \leftarrow (V, E)$ cat timp $T$ conține cicluri executa <ul style="list-style-type: none"><li>• alege <math>e \in E(T)</math> o muchie dintr-un ciclu</li><li>• <math>E(T) \leftarrow E(T) - \{e\}</math></li></ul> returneaza $T$
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

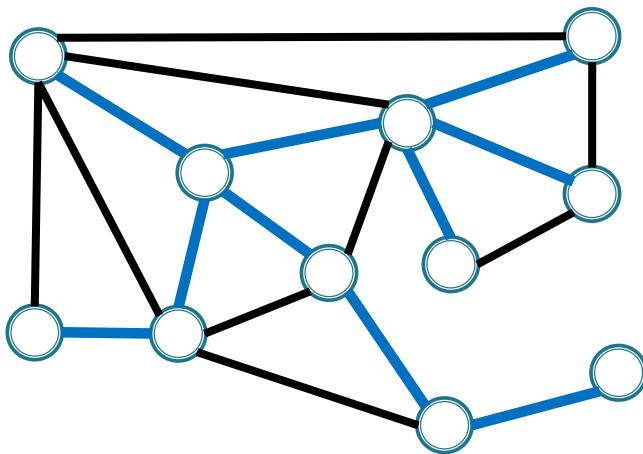
Algoritmi de determinare a unui arbore parțial al unui graf conex

Complexitate algoritm?



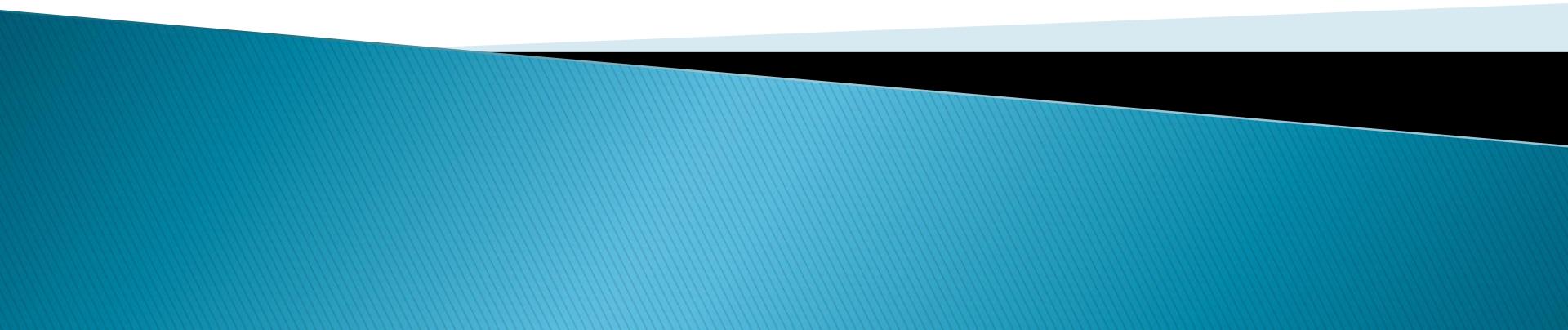
arborele asociat unei parcurgeri este arbore parțial  $\Rightarrow$   
determinăm un arbore parțial printr-o parcuregere

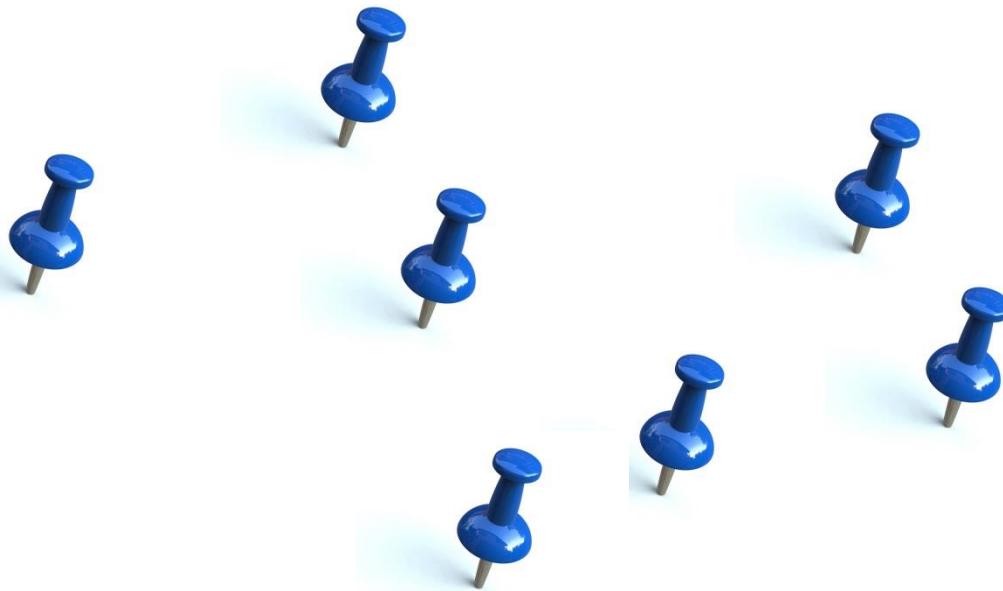
# Arborei parțiali



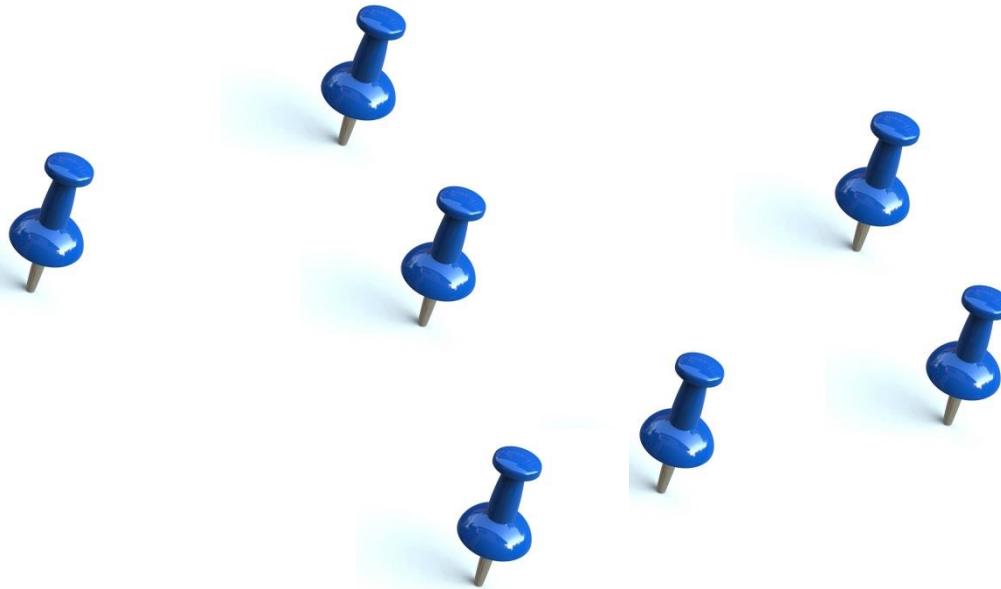
- “Scheletul” grafului
- Transmiterea de mesaje în rețea astfel încât mesajul să ajungă o singură dată în fiecare vârf
- Conectare fără redundanță + cu cost minim

# **Arbore parțiali de cost minim**

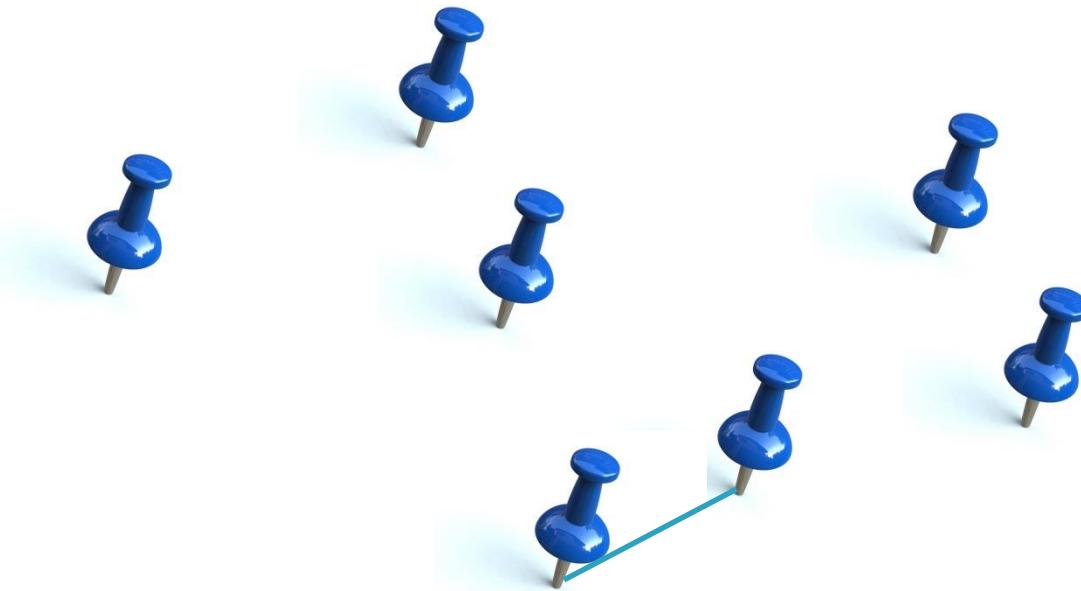


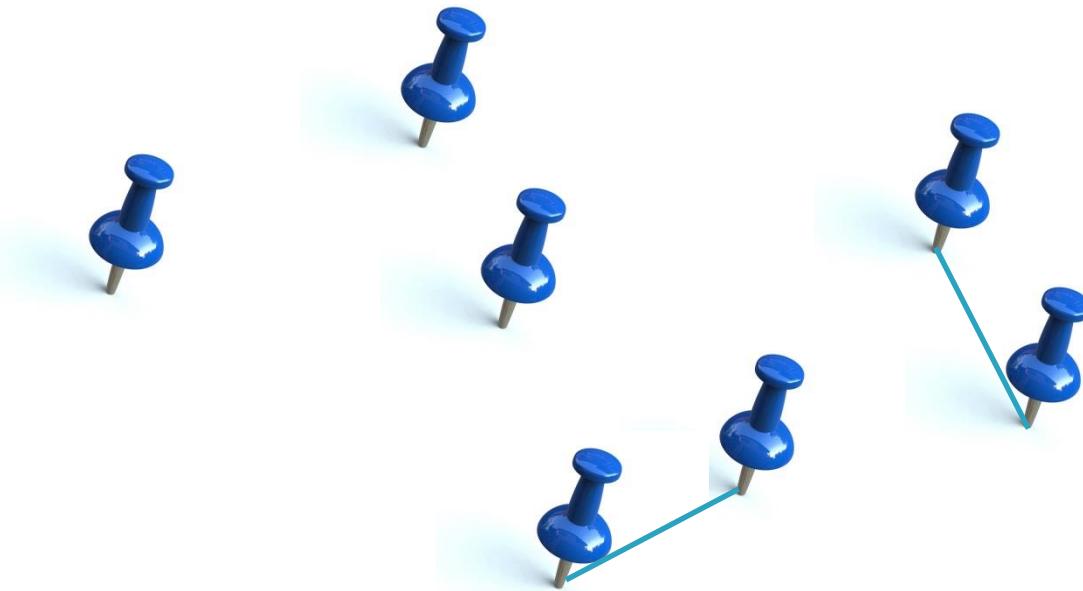


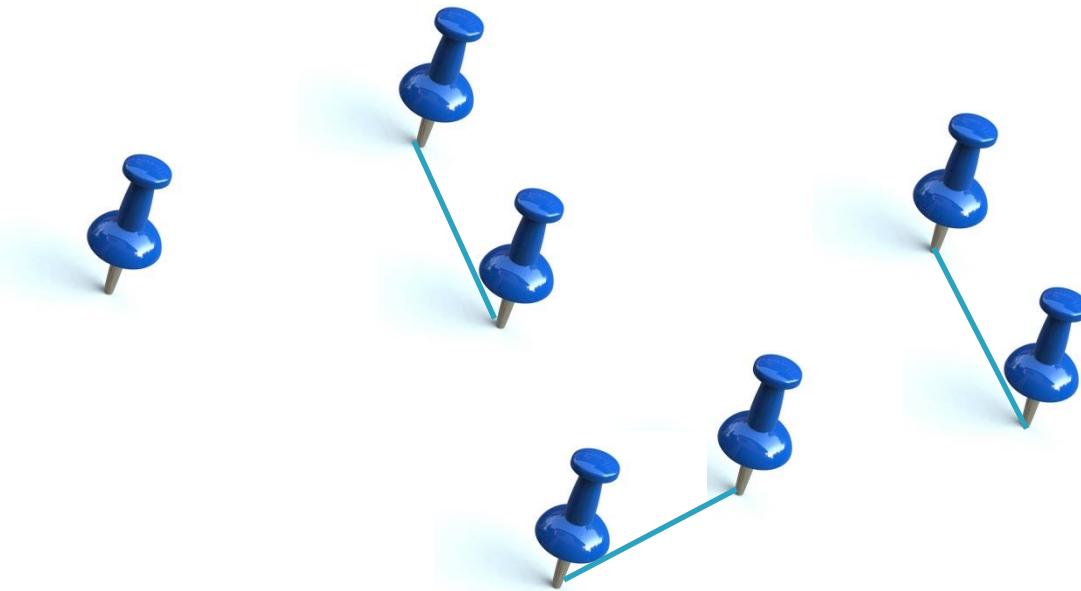
Conectați pinii astfel încât să folosiți cât mai puțin cablu

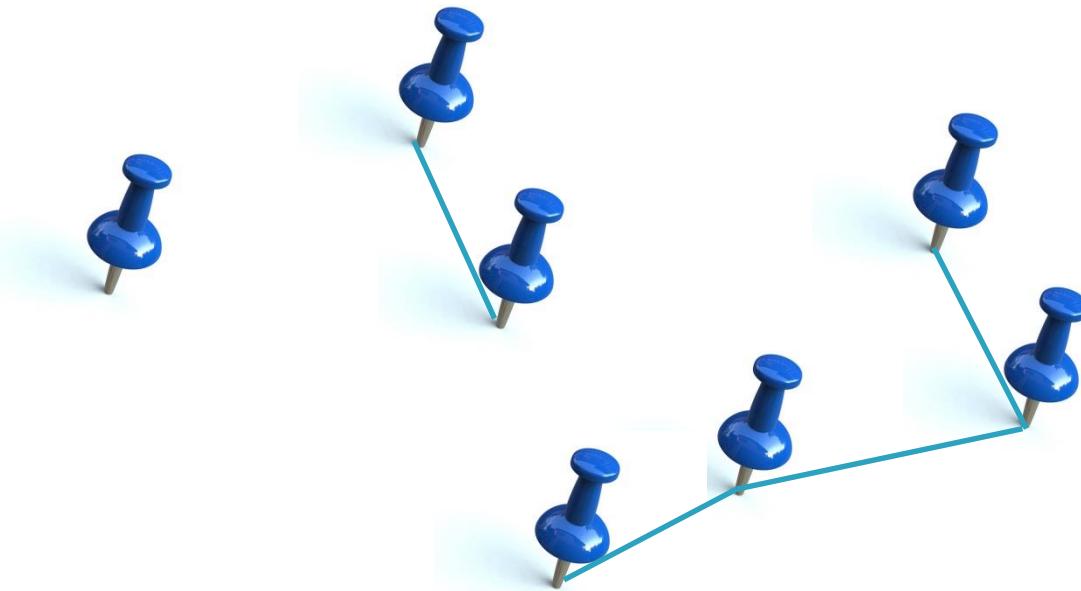


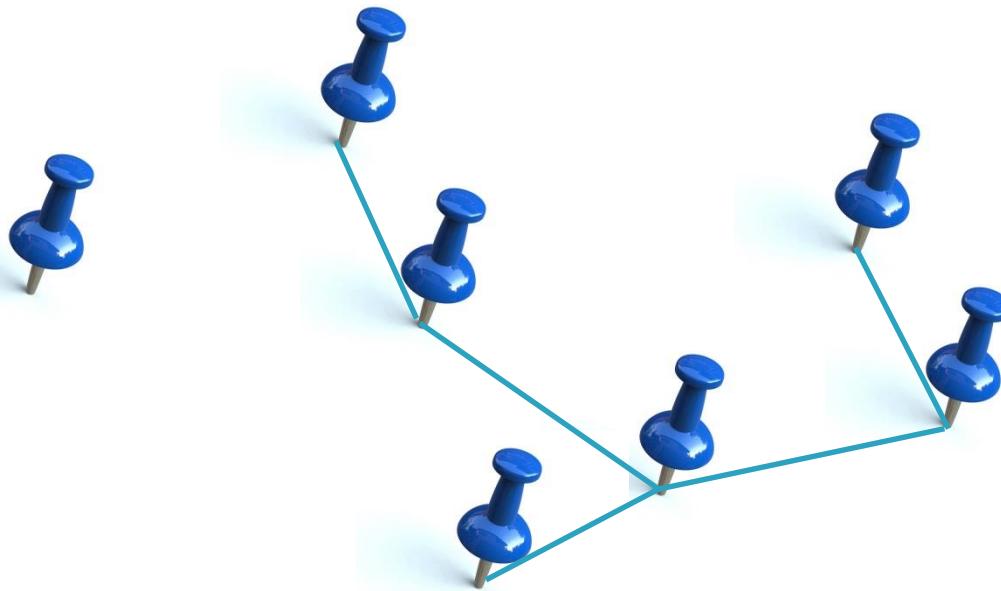
- Legăm pini apropiati
- Nu închidem cicluri

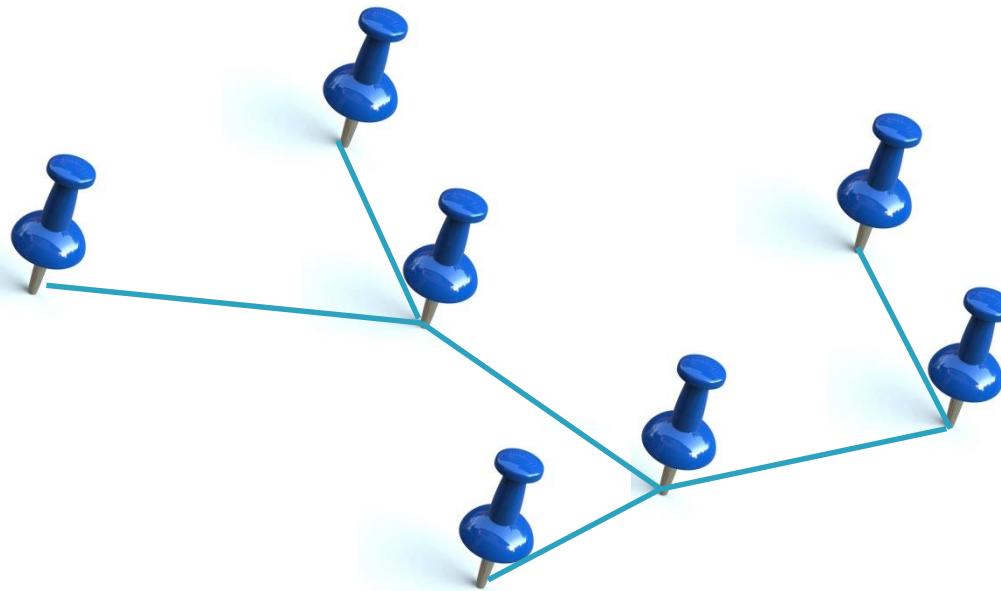














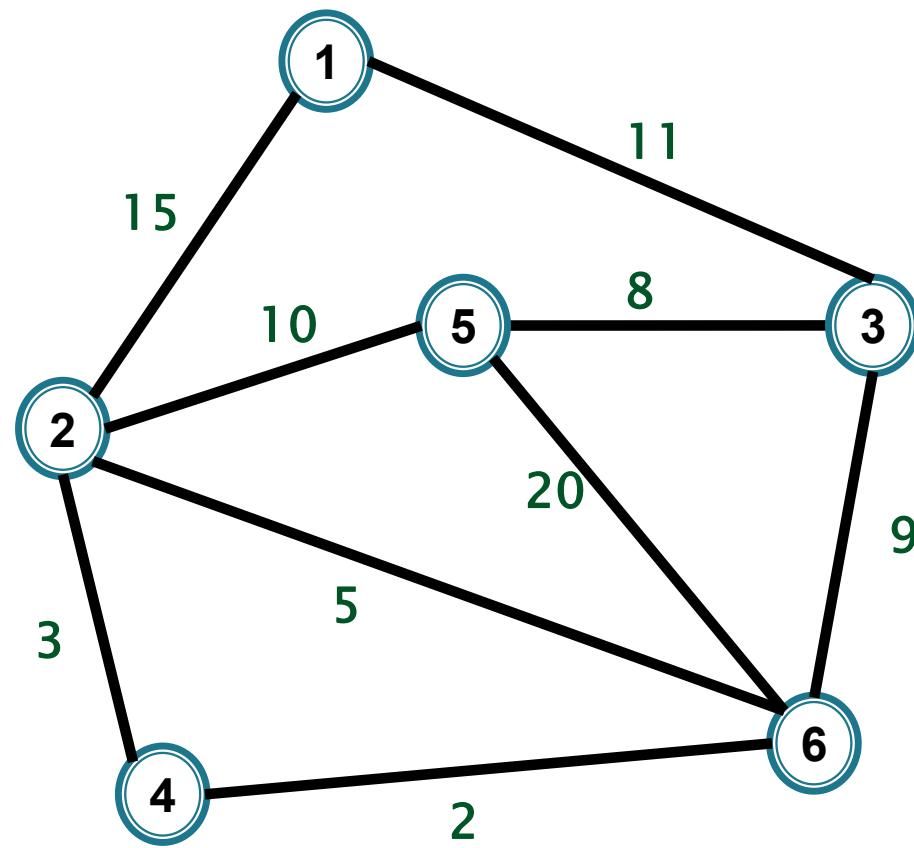
**conectare cu cost minim  $\Rightarrow$  evităm ciclurile**

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)**
- ▶ notat  $G = (V, E, w)$

# Grafuri ponderate

►  $G = (V, E, w)$  **graf ponderat**

► Pentru  $A \subseteq E$

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

# Grafuri ponderate

►  $G = (V, E, w)$  **graf ponderat**

► 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

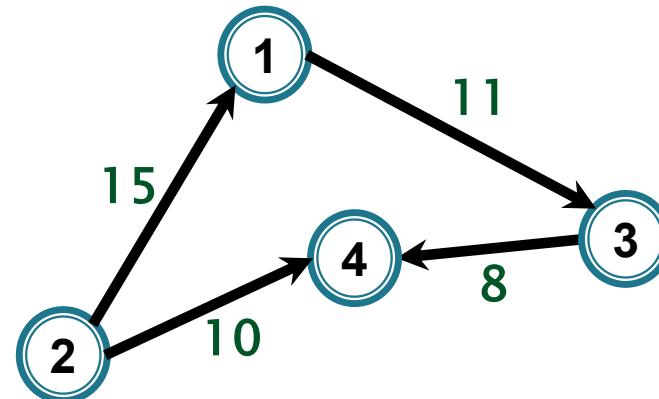
## Reprezentarea grafurilor ponderate

# Grafuri ponderate

## Reprezentarea grafurilor ponderate

- Matrice de costuri (ponderi)  $W = (w_{ij})_{i,j=1..n}$

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

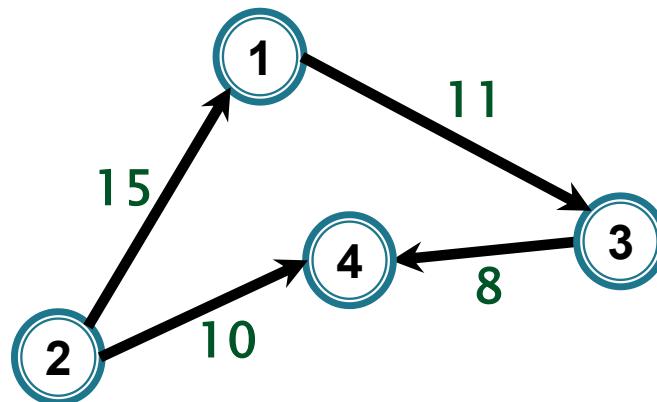


0	$\infty$	11	$\infty$
15	0	$\infty$	10
$\infty$	$\infty$	0	8
$\infty$	$\infty$	$\infty$	0

# Grafuri ponderate

## Reprezentarea grafurilor ponderate

- ▶ Matrice de costuri (ponderi)
- ▶ Liste de adiacență

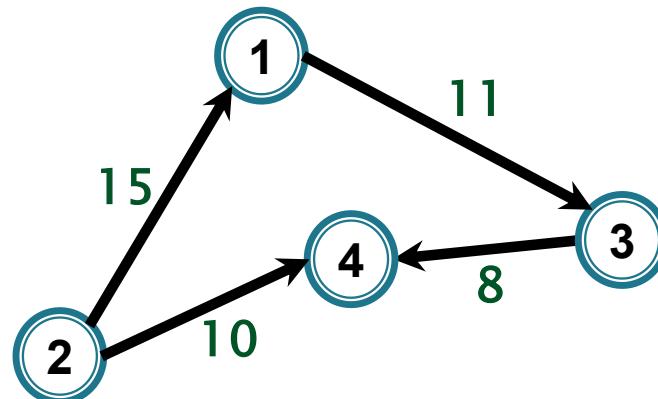


1: 3 / 11  
2: 1 / 15, 4 / 10  
3: 4 / 8  
4:

# Grafuri ponderate

## Reprezentarea grafurilor ponderate

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



1 3 11

2 1 15

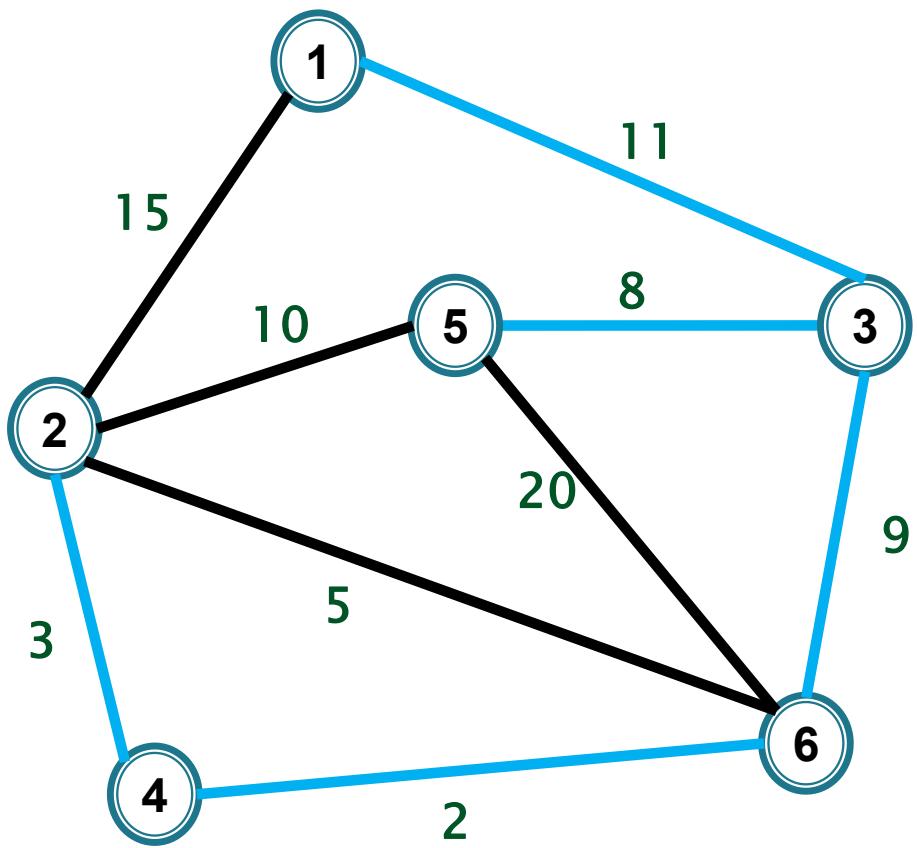
2 4 10

3 4 8

# A.p.c.m

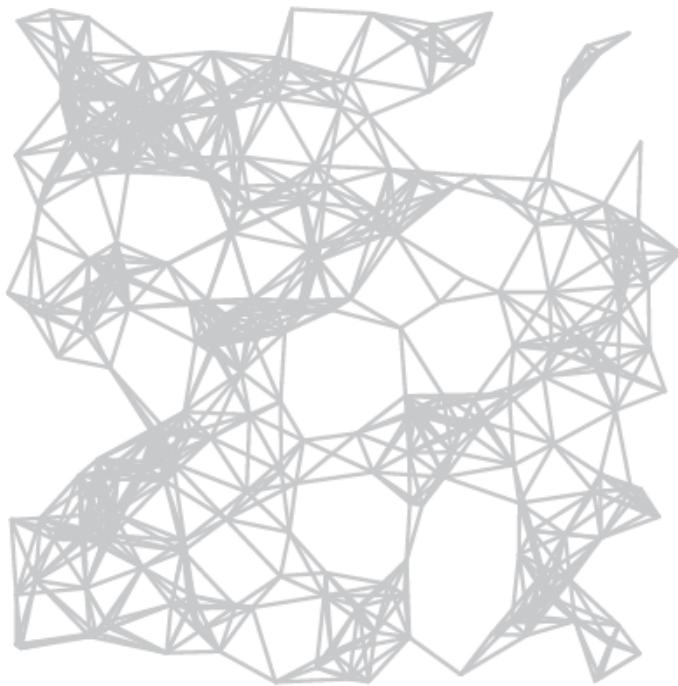
- ▶  $G = (V, E, w)$  **conex ponderat**
- ▶ **Arbore parțial de cost minim** al lui  $G$  = un arbore parțial  $T_{\min}$  al lui  $G$  cu

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

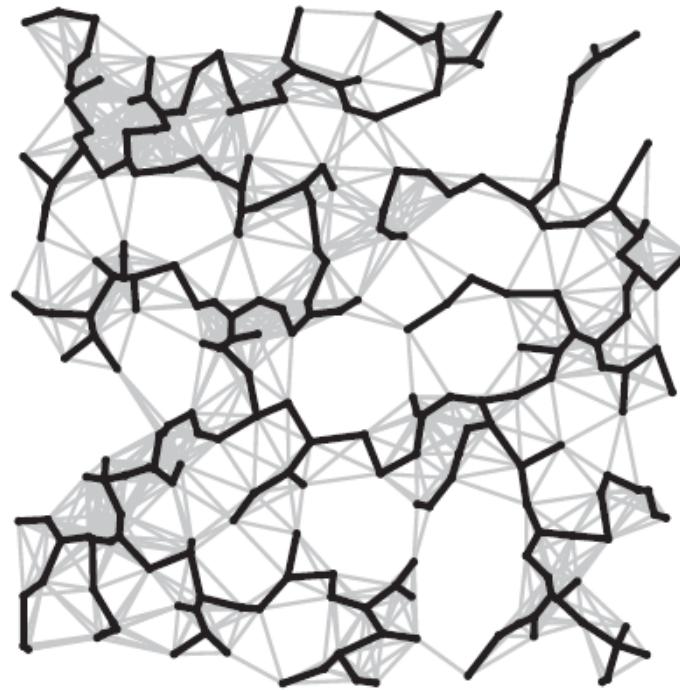


# A.p.c.m.

graf 250 noduri



apcm



Imagine din

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

# Aplicații a.p.c.m.

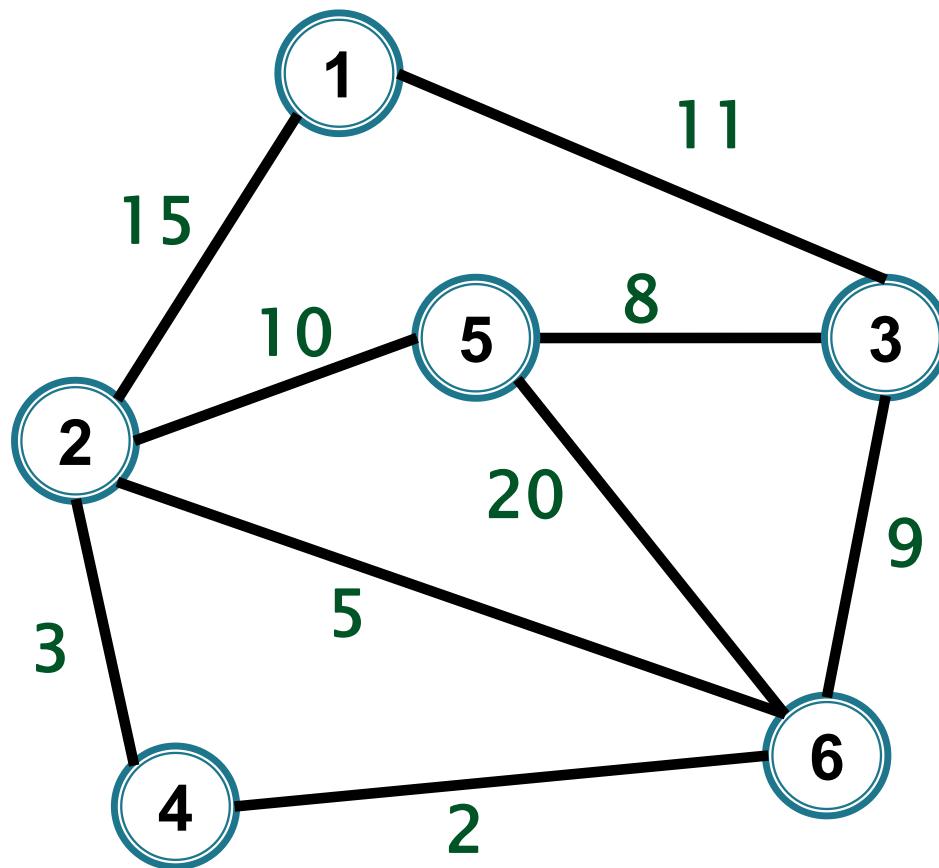
- ▶ **Construcția/renovarea unui sistem de căi ferate a.î.:**
  - oricare două stații să fie conectate (prin căi renovate)
  - sistem economic (costul total minim)
- ▶ **Proiectarea de rețele, circuite electronice**
  - conectarea pinilor cu cost minim/ fără cicluri
- ▶ **Clustering**
- ▶ **Subrutină 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



**Idee:** Prin **adăugare** **succesivă** de muchii, astfel încât multimea 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

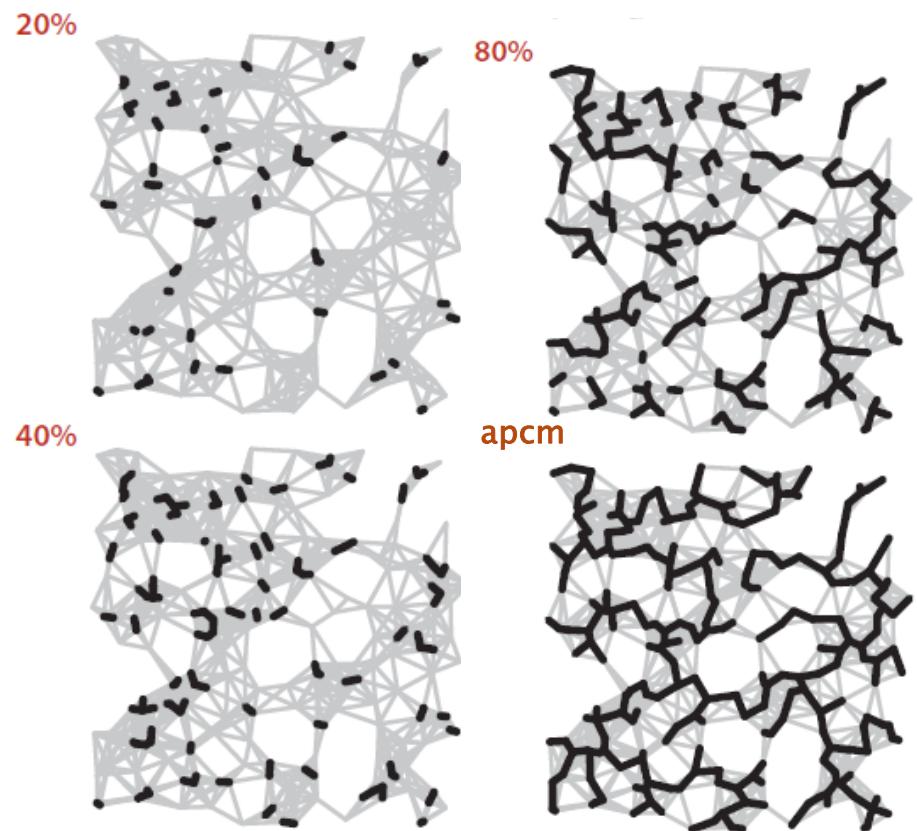


După ce criteriu selectăm muchiile?

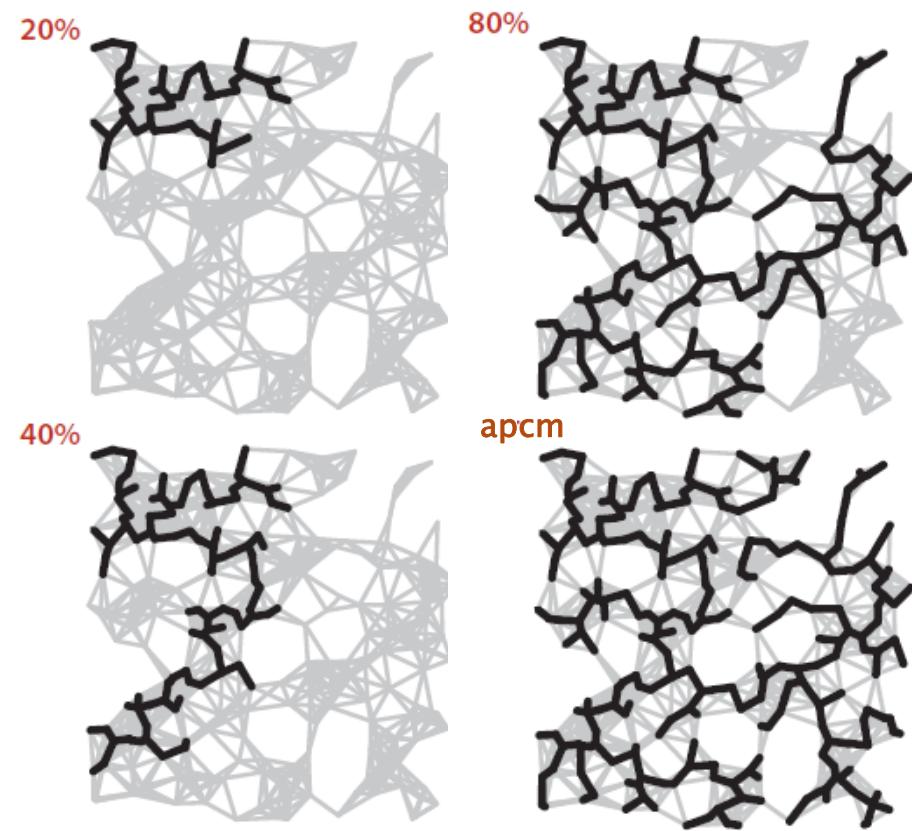
⇒ diversi algoritmi

# Arbore parțiali de cost minim

Kruskal



Prim



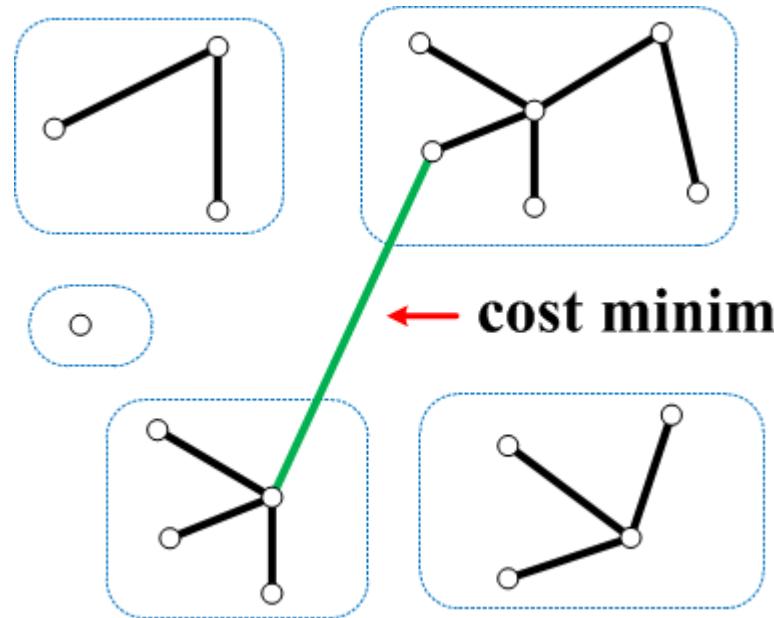
Imagine din

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)



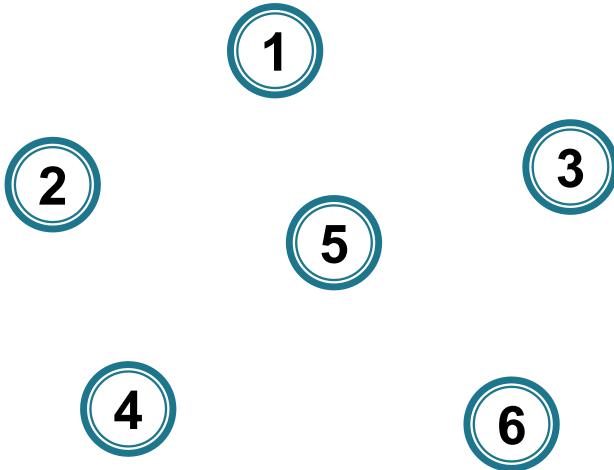
## ► O primă formă a algoritmului

### 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>)
  - $E(T) = E(T) \cup \{uv\}$

# Kruskal

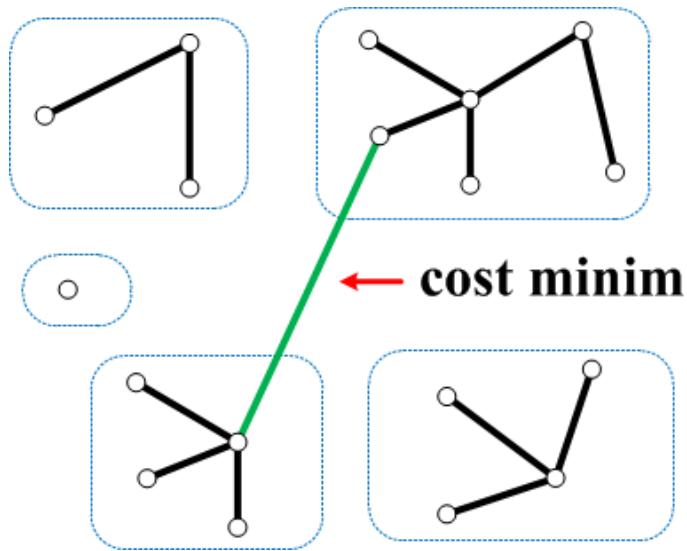
- Initial: cele n vârfuri sunt izolate, fiecare formând o componentă conexă



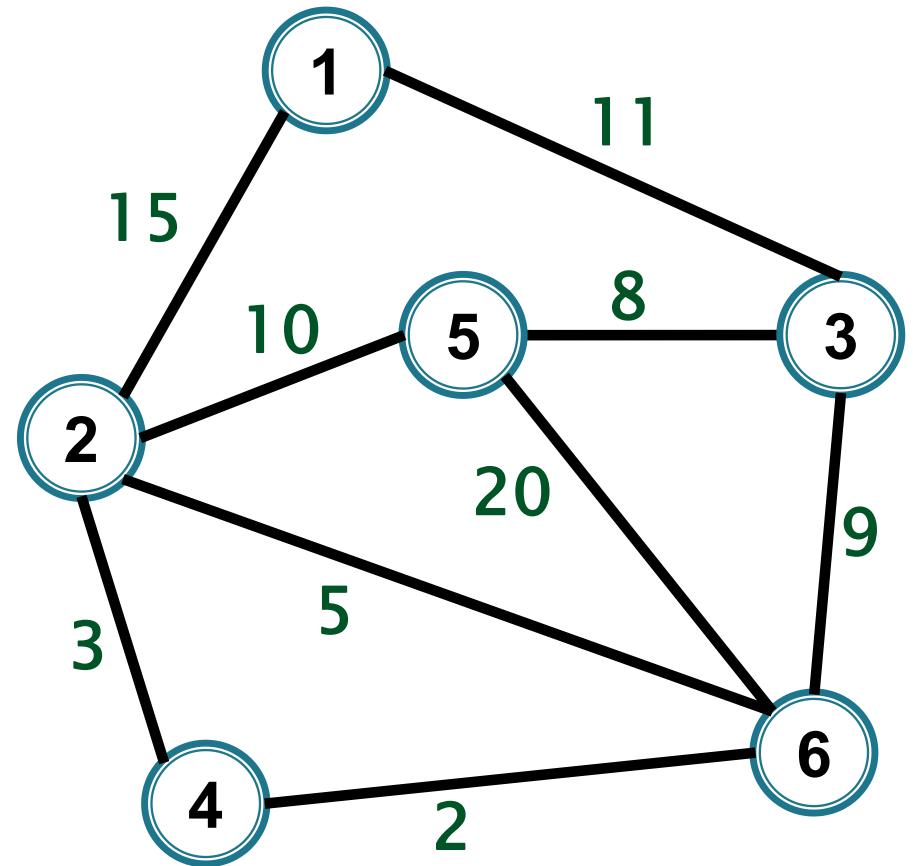
# Kruskal

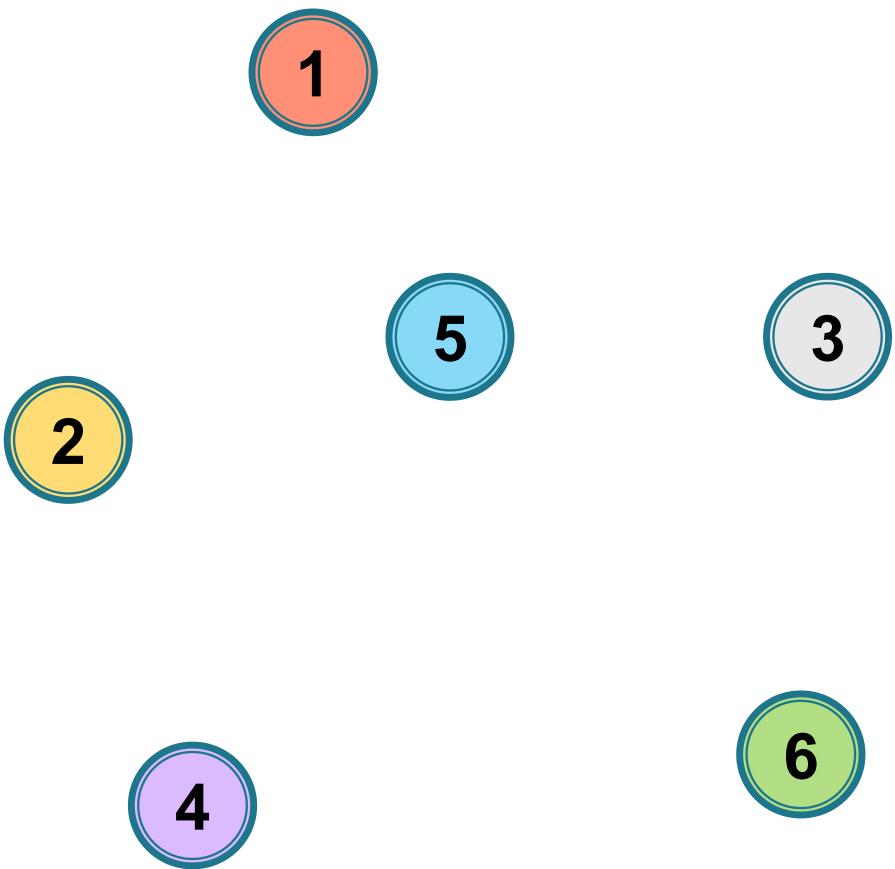
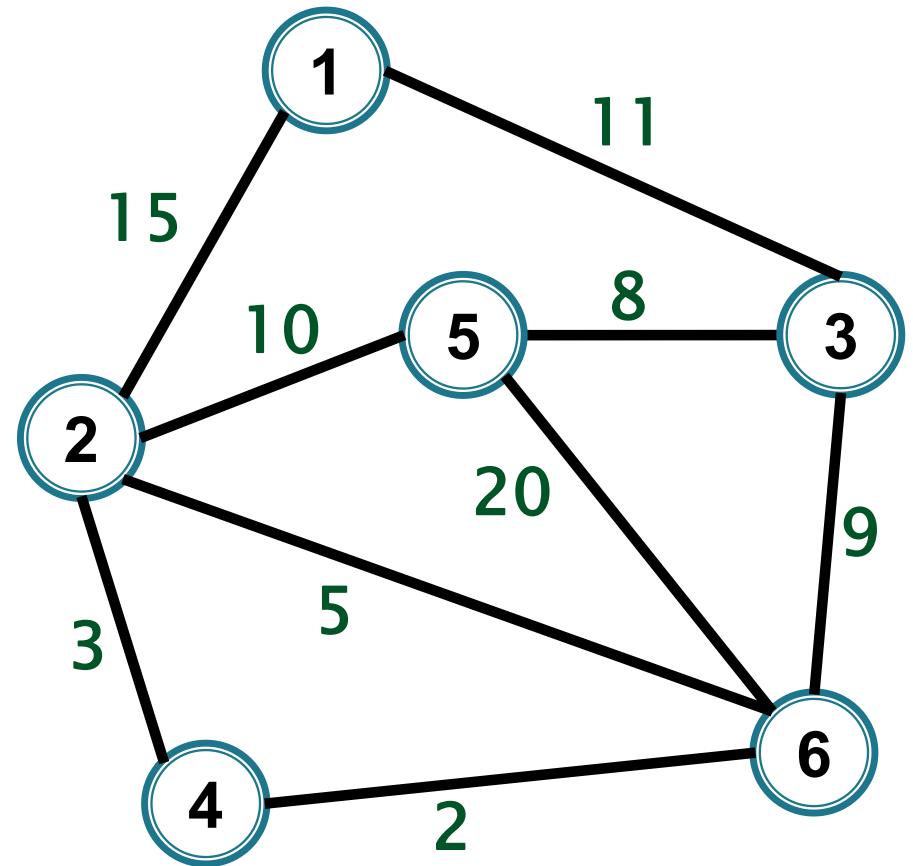
- La un pas:

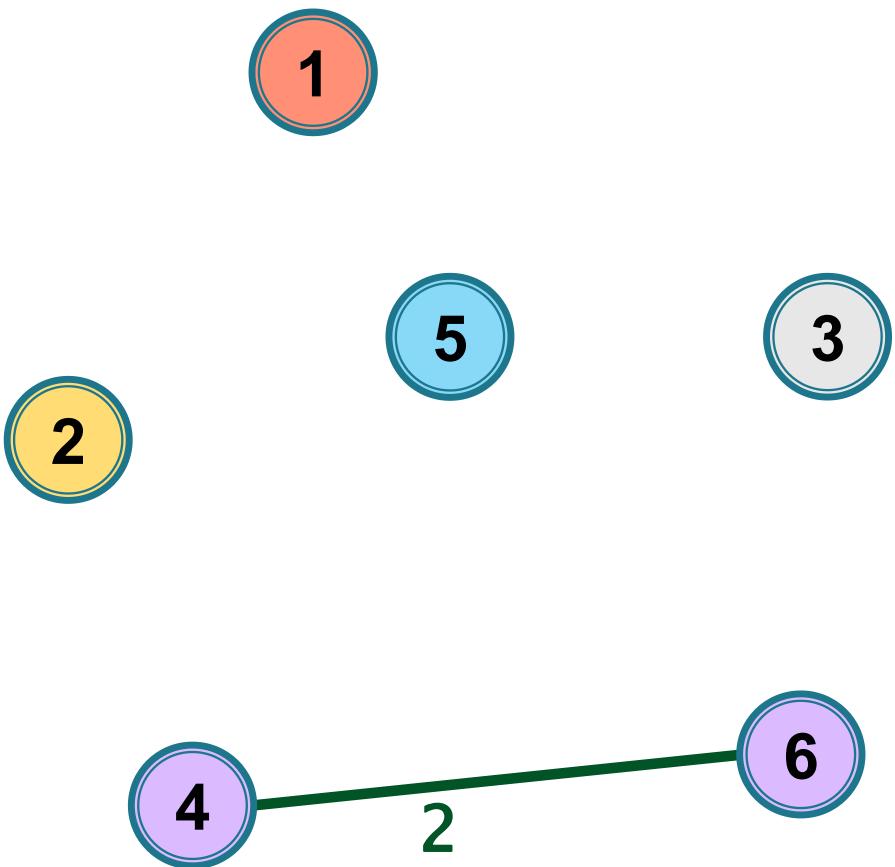
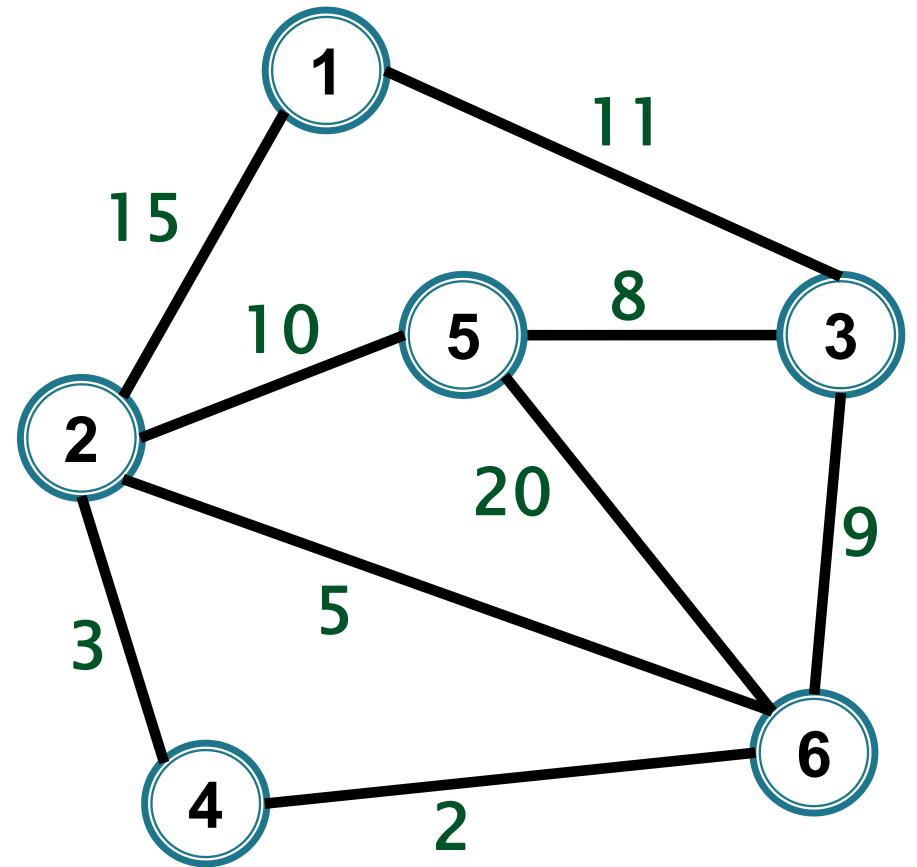
Muchiile selectate formează o  
pădure

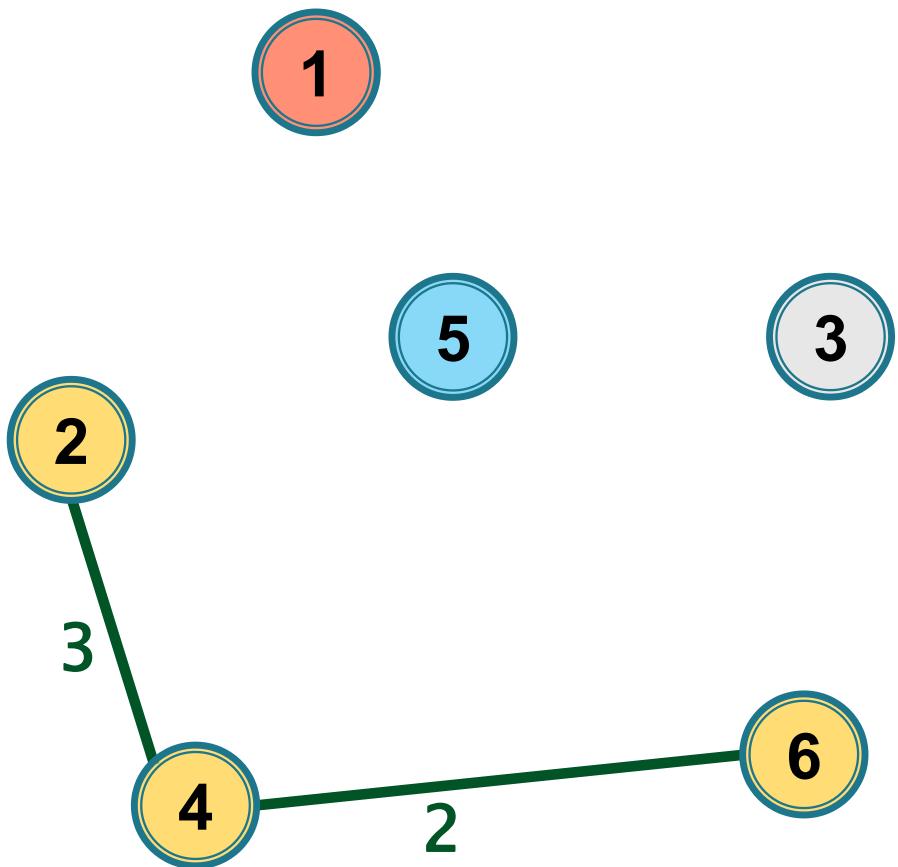
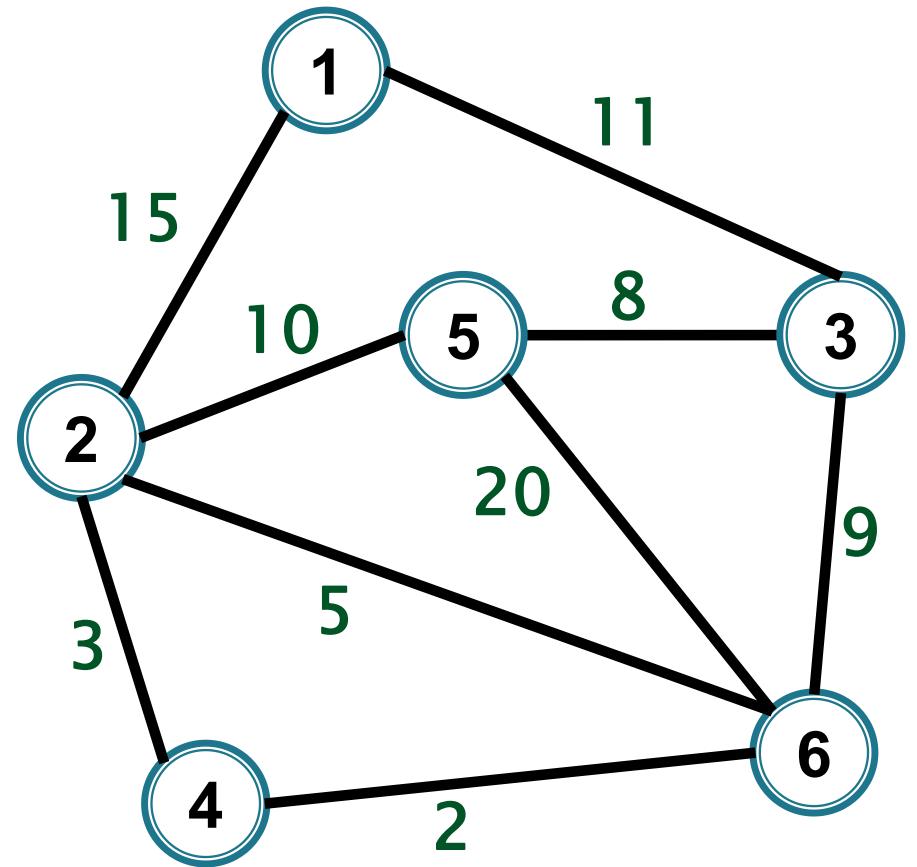


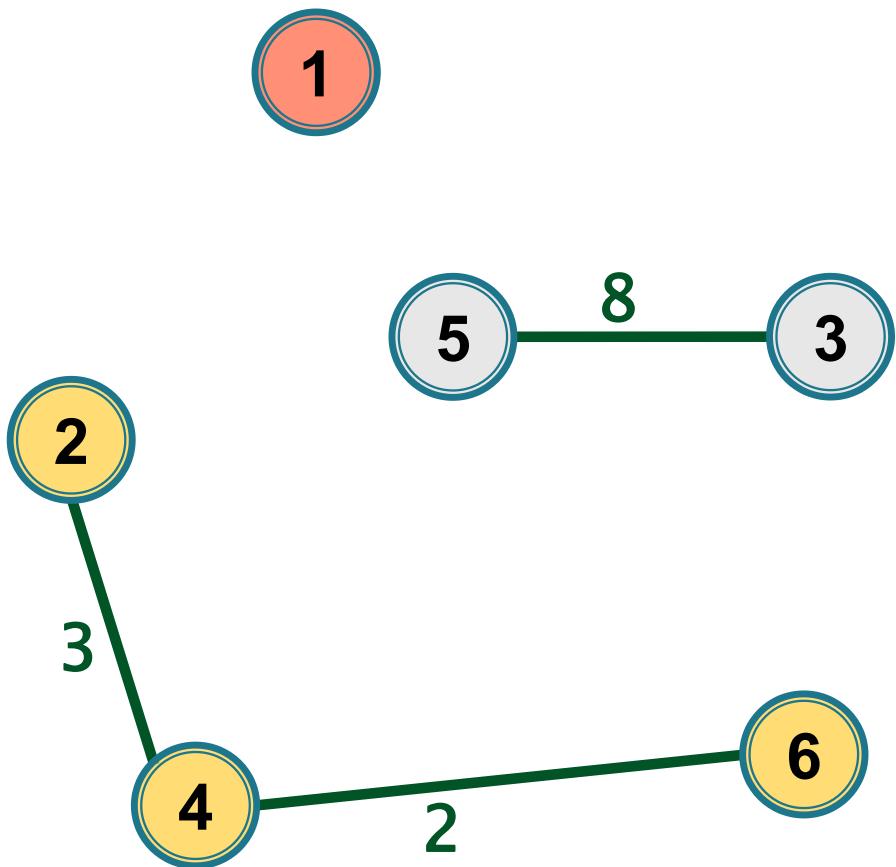
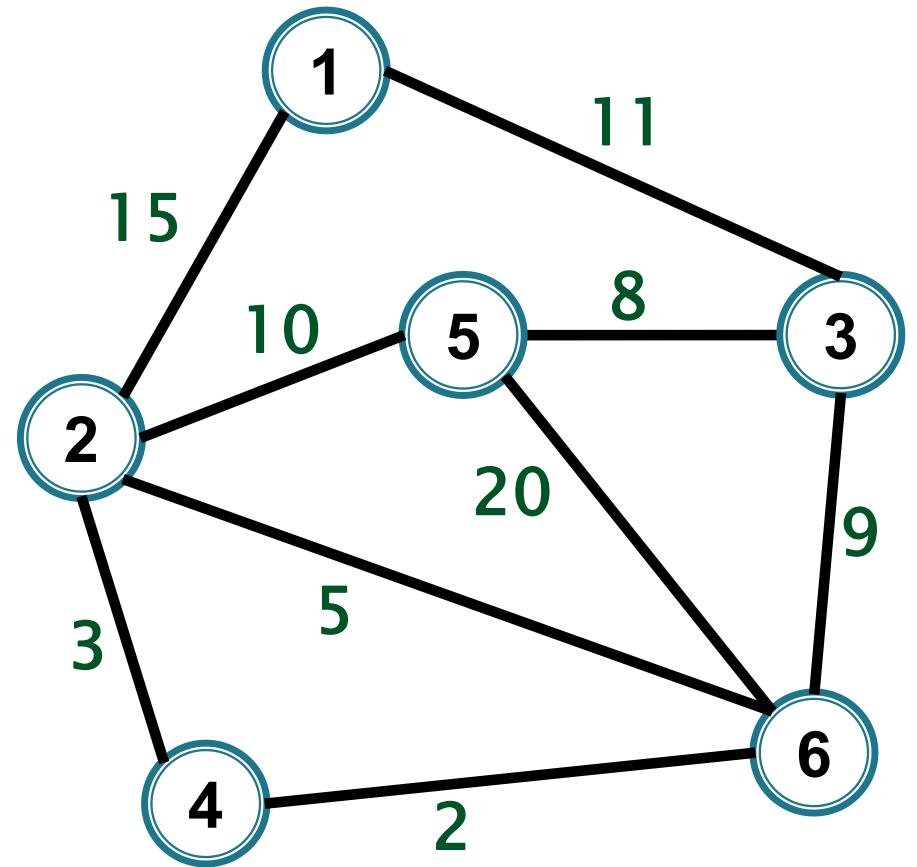
Este selectată o muchie de cost minim  
care unește doi arbori din pădurea  
currentă (două componente conexe)

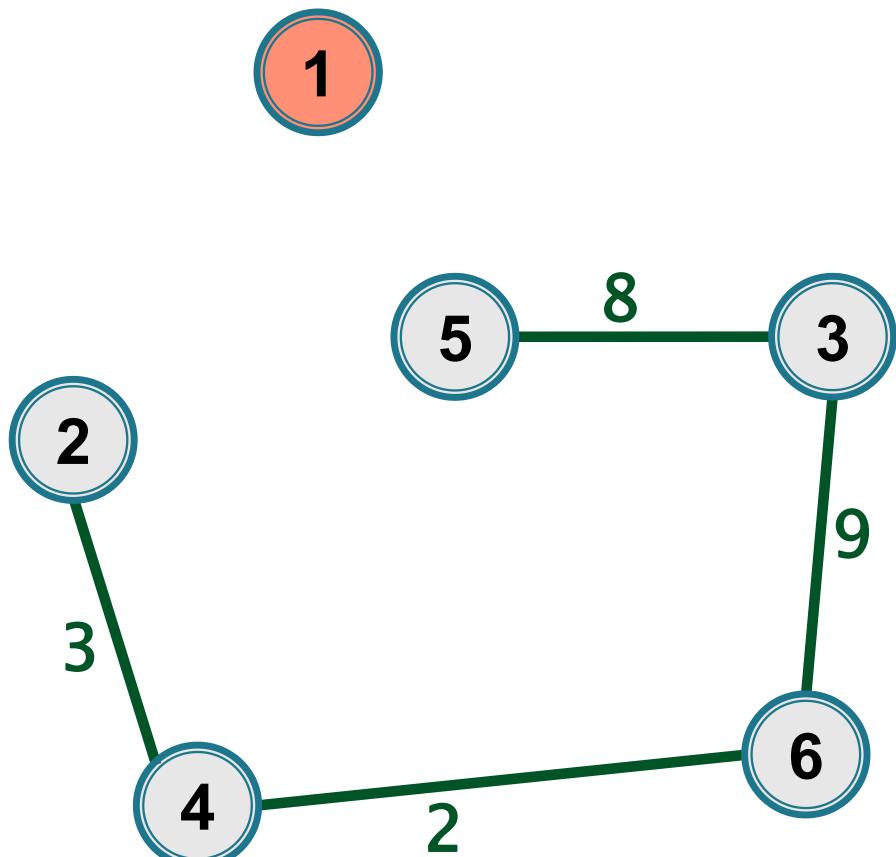
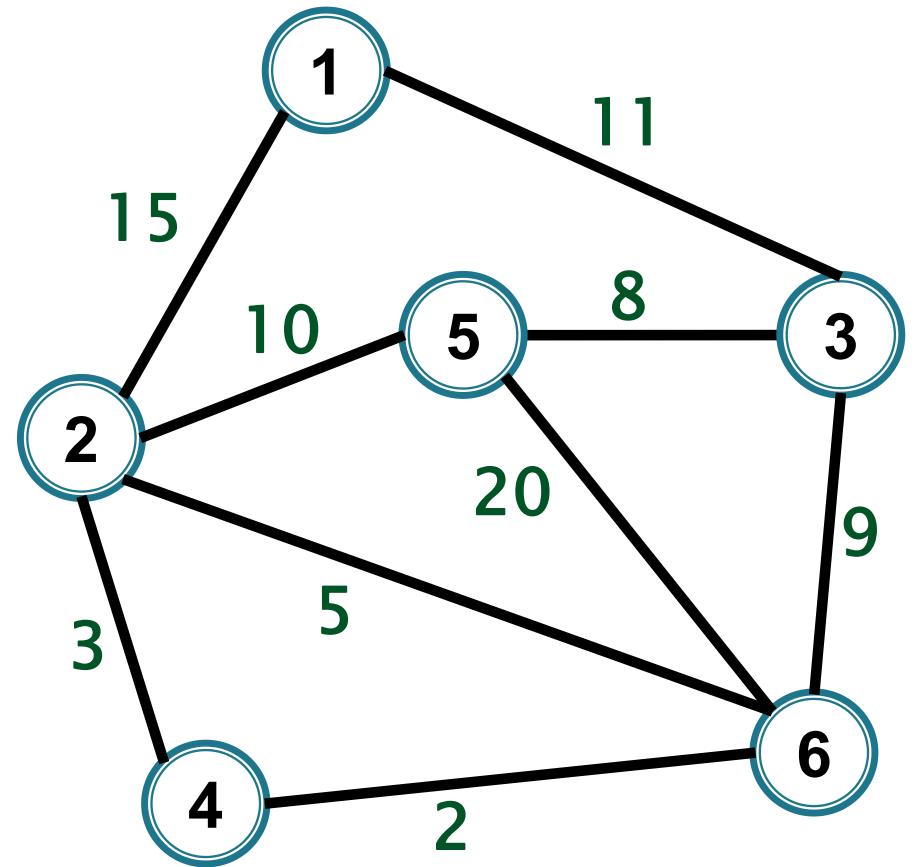


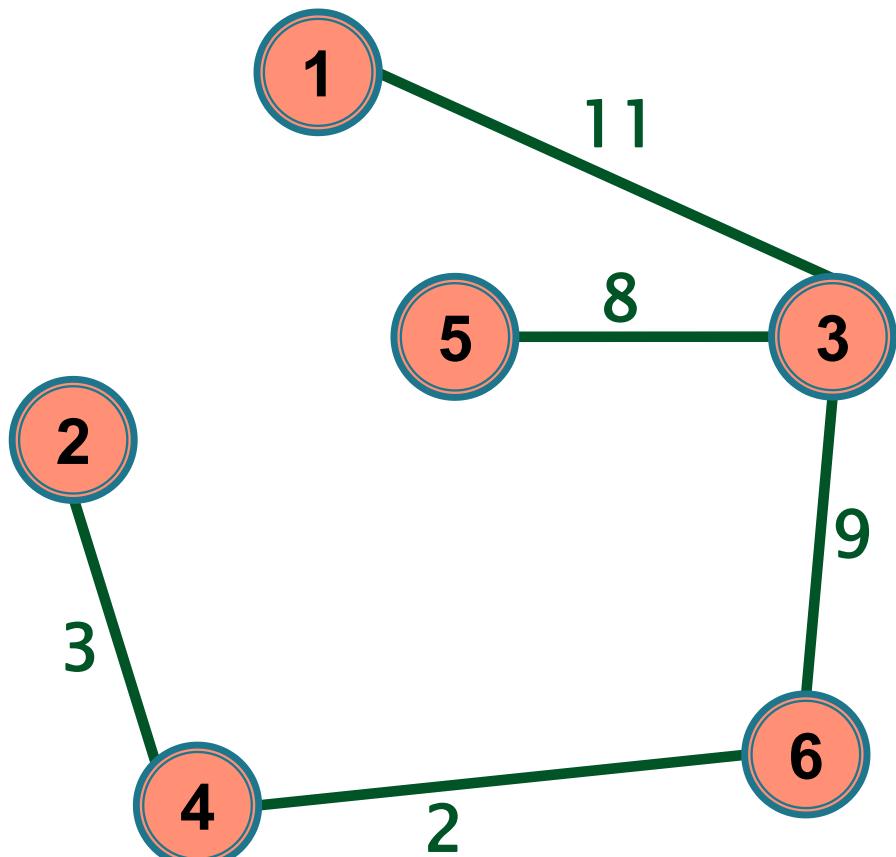
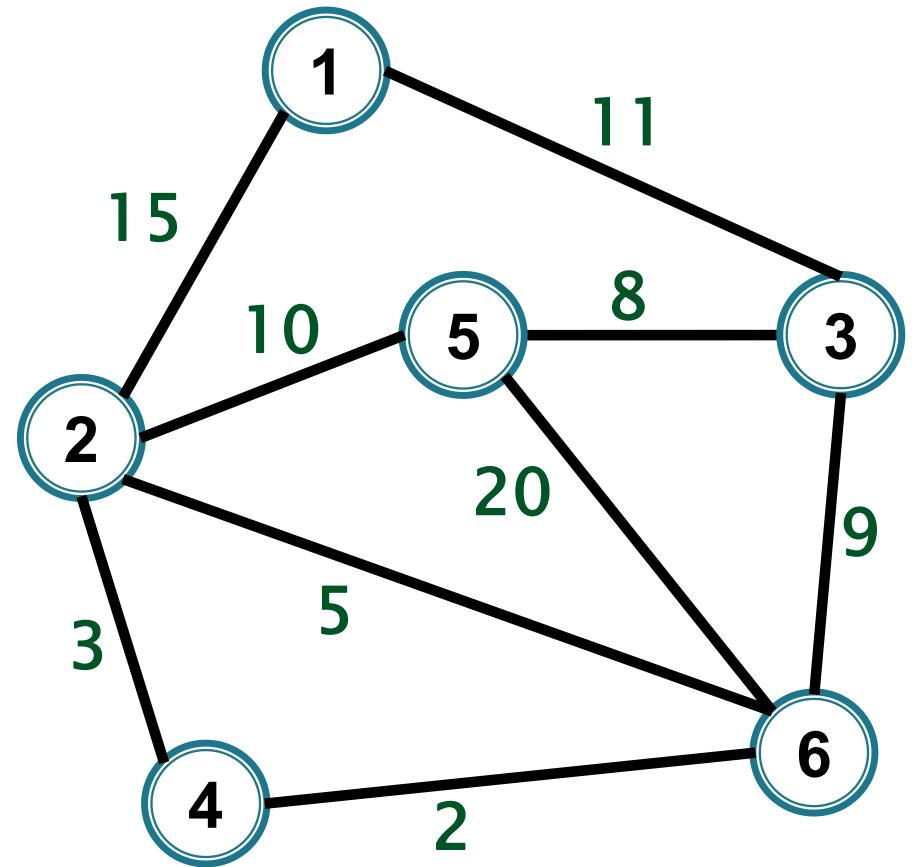












# Kruskal – Implementare



**1. Cum reprezentăm graful în memorie?**

**2. Cum selectăm ușor o muchie:**

- de cost minim
- care unește două componente (nu formează cicluri cu muchiile deja selectate)

# Kruskal



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

# Kruskal



## Reprezentarea grafului ponderat

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

# Kruskal



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

# Kruskal



**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



**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,**

**$\Rightarrow O(mn)$  – ineficient**



# Kruskal



Componentele sunt multimi disjuncte din V  
(partiie a lui V)

⇒ **structuri pentru multimi disjuncte**

- asociem fiecărei componente un reprezentant  
(o culoare)

# Kruskal

## ► Operații necesare:

- **Initializare(u)** –
- **Reprez(u)** –
- **Reuneste(u,v)** –

# Kruskal

## ► Operații necesare:

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

# Kruskal

- ▶ O muchie uv unește două componente dacă și numai dacă

# Kruskal

- ▶ O muchie  $uv$  unește două componente dacă și numai dacă

**Reprez(u)  $\neq$  Reprez(v)**

# Kruskal

**sorteaza (E)**

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

**Initializare (v) ;**

# Kruskal

```
sorteaza (E)
for (v=1 ; v<=n ; v++)
    Initializare (v) ;
nrmsel=0
for (uv  ∈  E)
    if (Reprez (u) !=Reprez (v) )
    {
}
}
```

# Kruskal

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

# Kruskal

## Complexitate



De câte ori se execută fiecare operație?

# Kruskal

## Complexitate

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

Depinde de modalitatea de memorare a componentelor conexe

# Kruskal



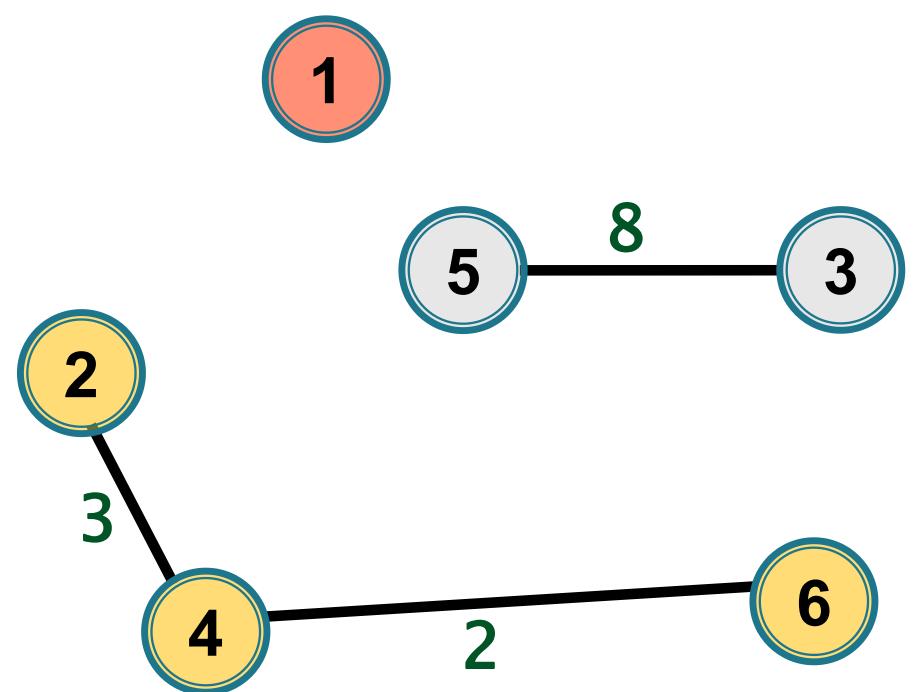
**Cum memorăm componentele + reprezentantul / culoarea componentei în care se află un vârf?**

# Kruskal

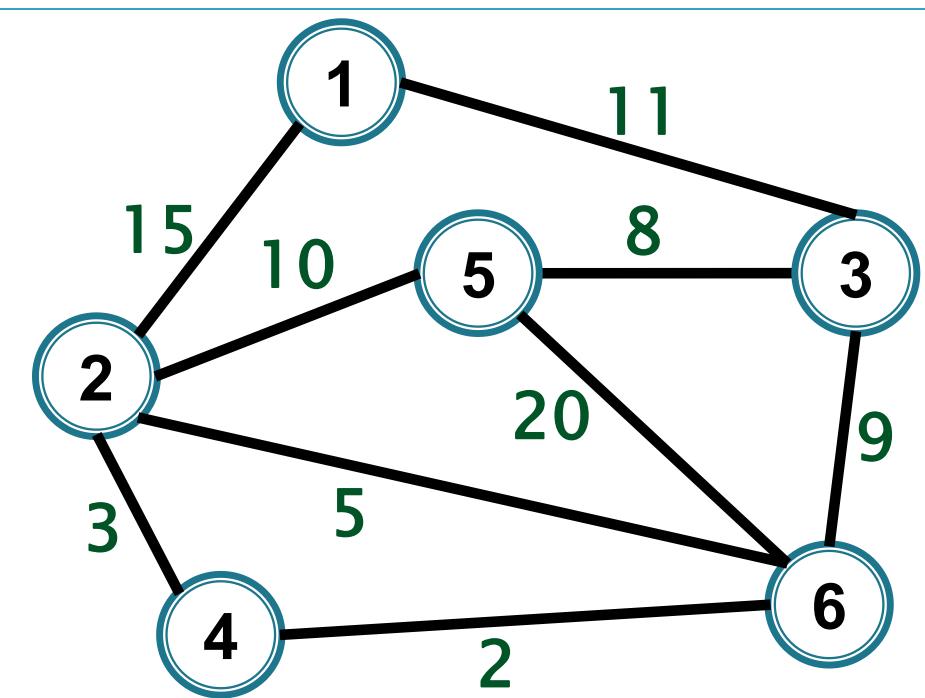


**Varianta 1** – memorăm într-un vector pentru fiecare vârf reprezentantul/culoarea componentei din care face parte

$r[u]$  = 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];  
}
```

- ▶ Reuneste – O(n)

```
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;  
}
```

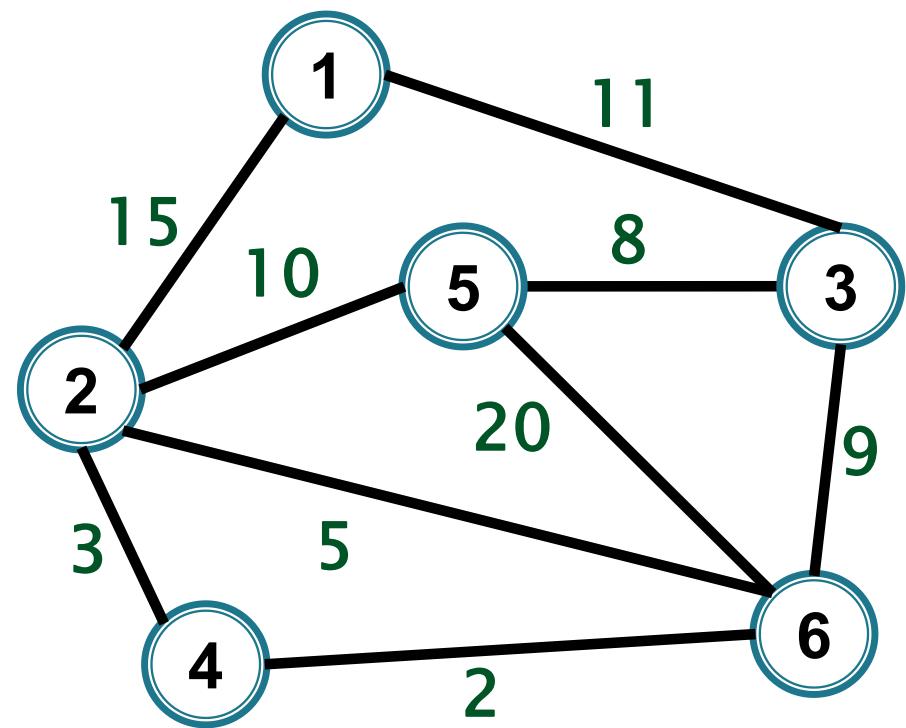
# Kruskal

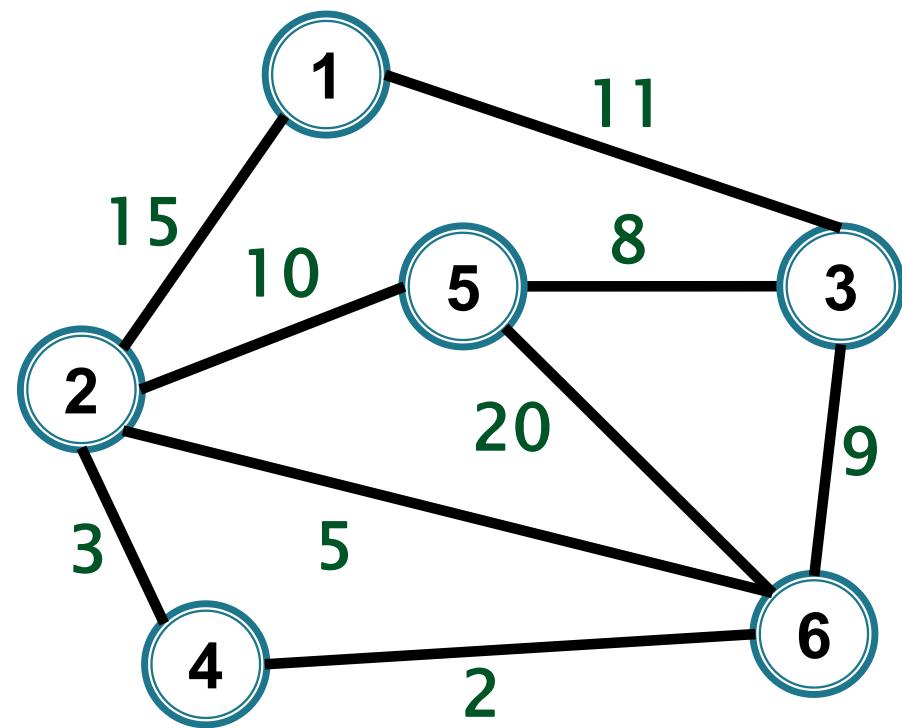
## Complexitate

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

- **Sortare**  $\rightarrow O(m \log m) = O(m \log n)$
  - **n \* Initializare**  $\rightarrow O(n)$
  - **2m \* Reprez**  $\rightarrow O(m)$
  - **(n-1) \* Reuneste**  $\rightarrow O(n^2)$
- 

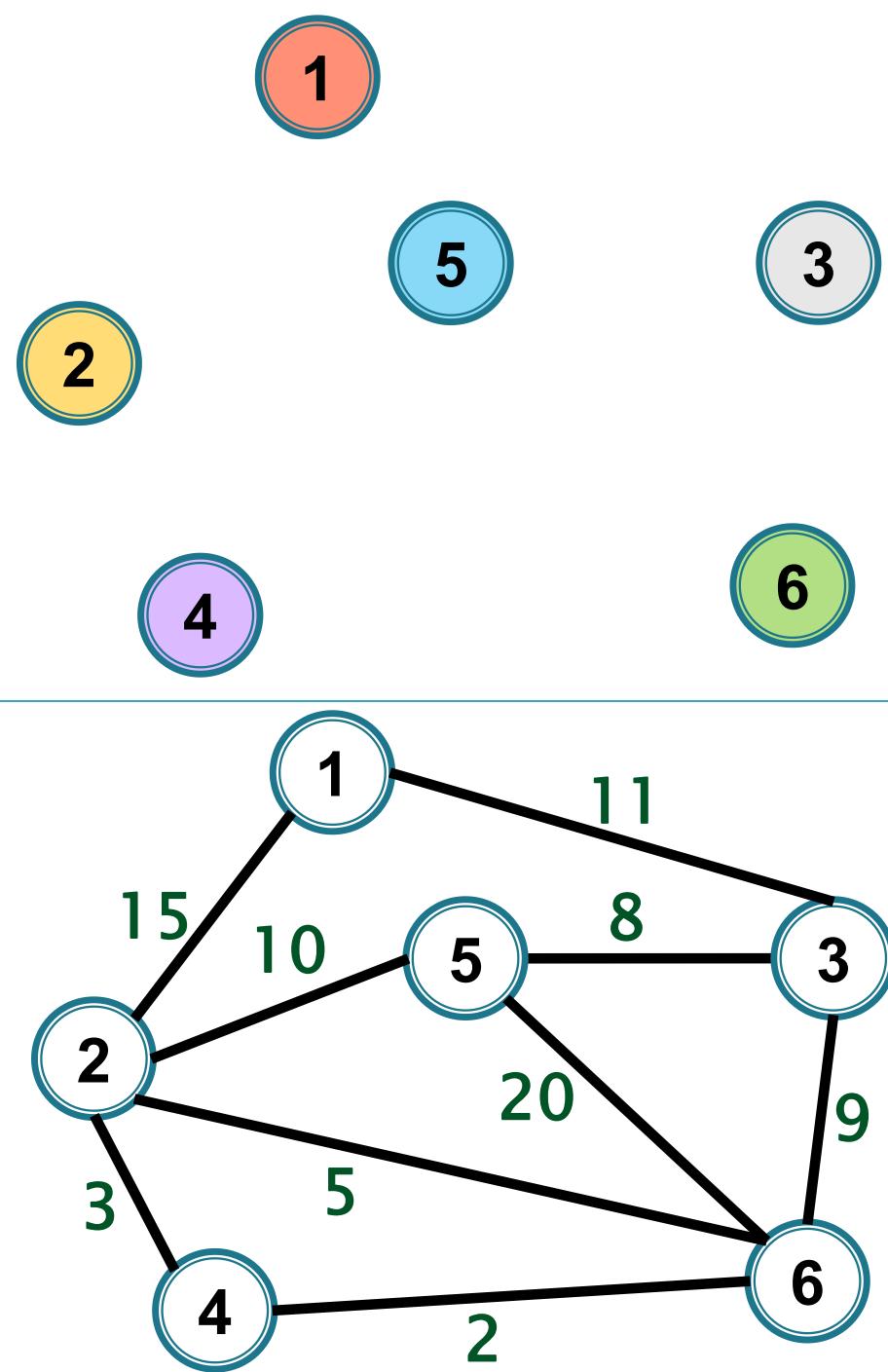
$$O(m \log n + n^2)$$



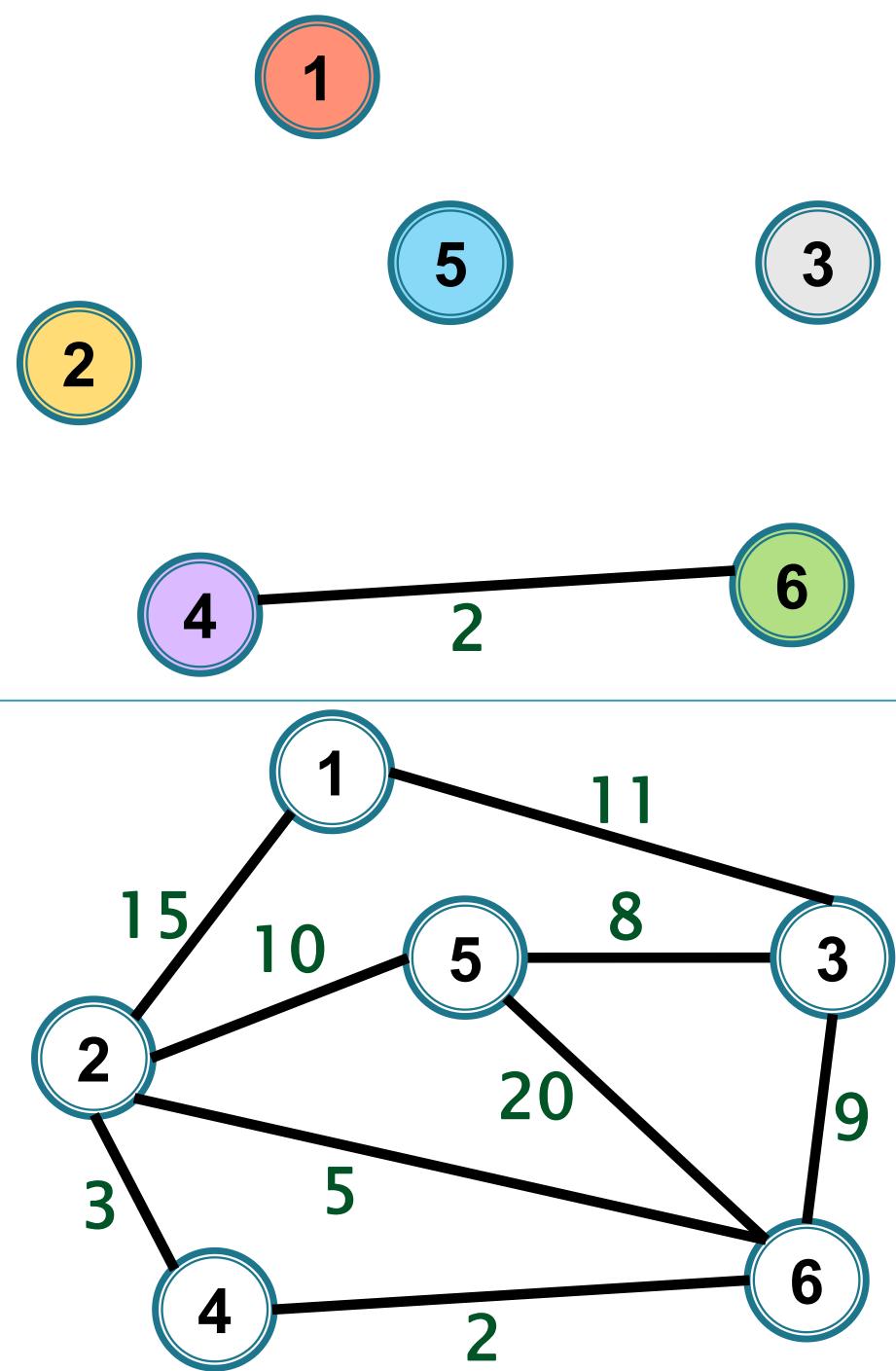


(4,6)  
(2,4)  
(2,6)  
(3,5)  
(3,6)  
(2,5)  
(1,3)  
(1,2)  
(5,6)

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



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



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

(4,6)

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

(2,4)

(2,6)

(3,5)

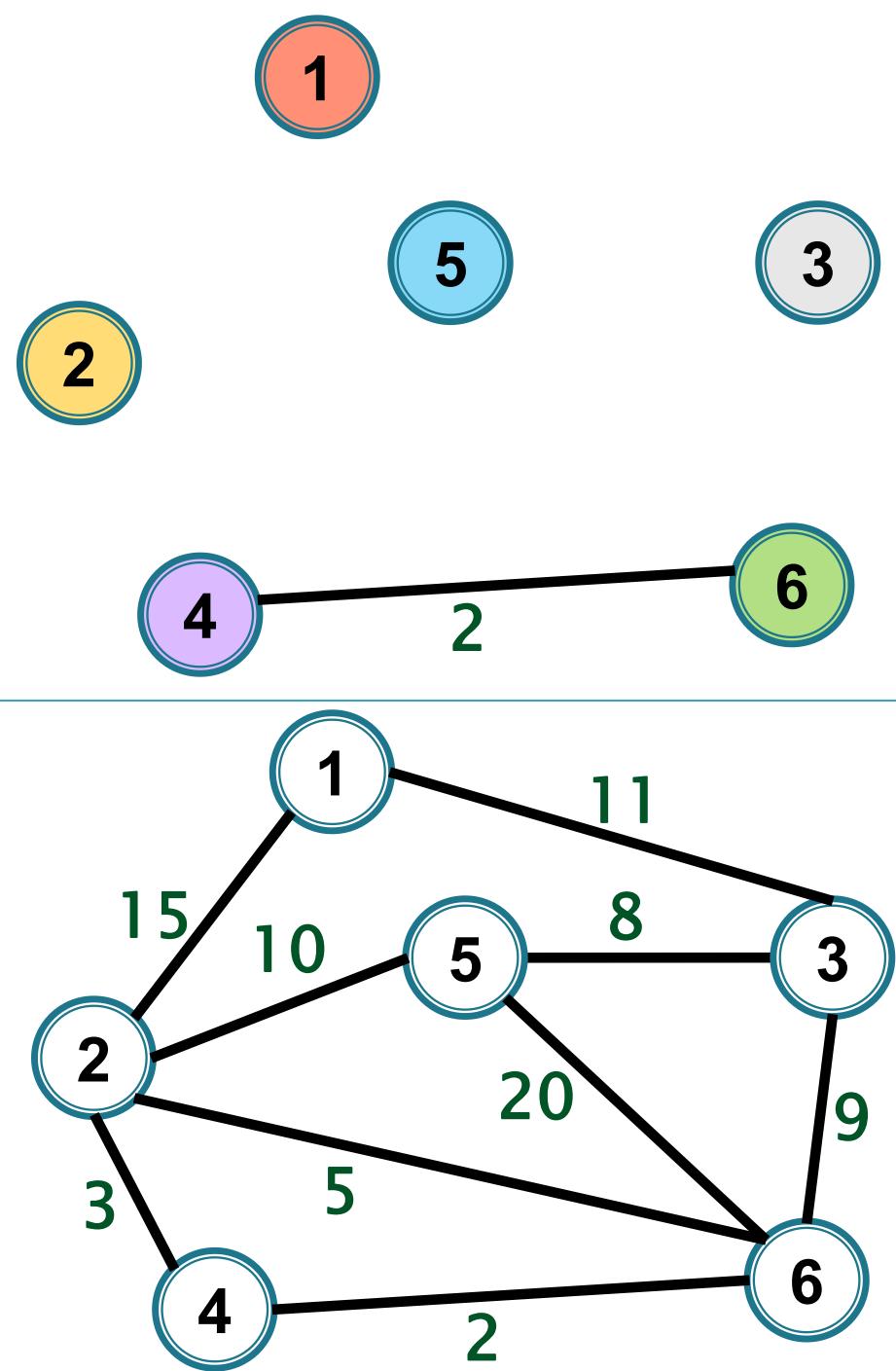
(3,6)

(2,5)

(1,3)

(1,2)

(5,6)



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

(4,6)

(2,4)

(2,6)

(3,5)

(3,6)

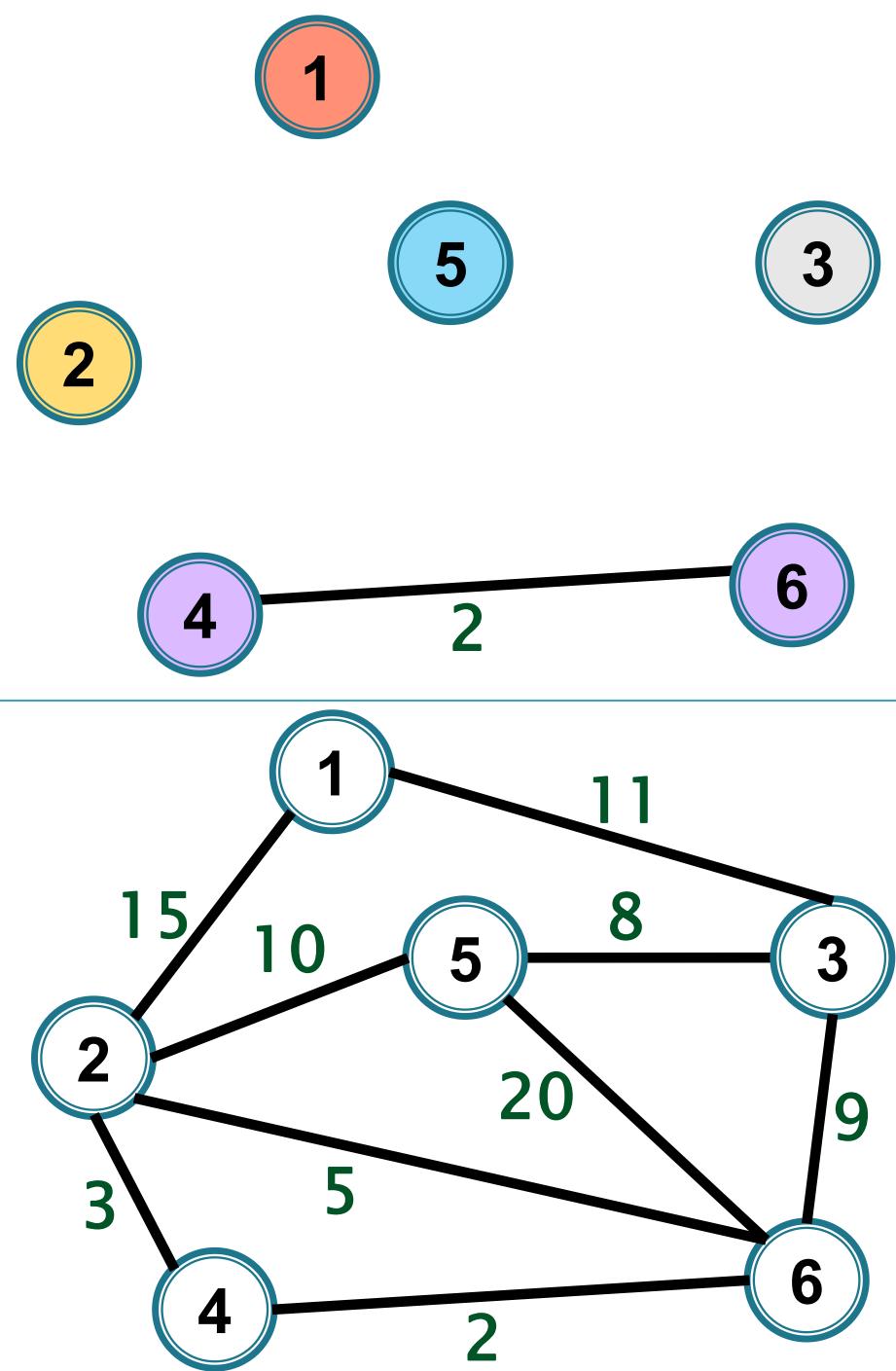
(2,5)

(1,3)

(1,2)

(5,6)

Reuneste(4, 6)



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

(4,6)

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

(2,4)

(2,6)

(3,5)

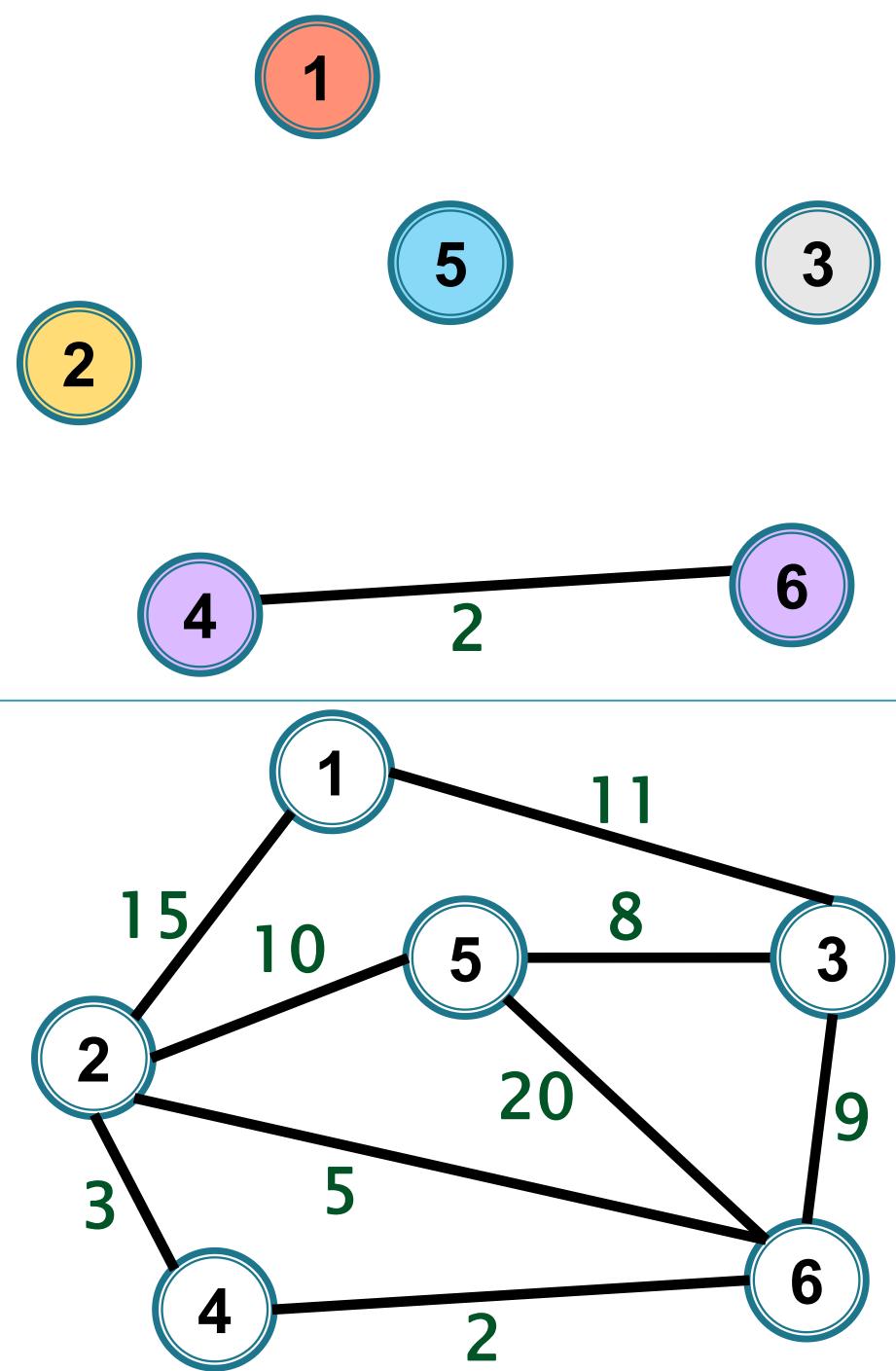
(3,6)

(2,5)

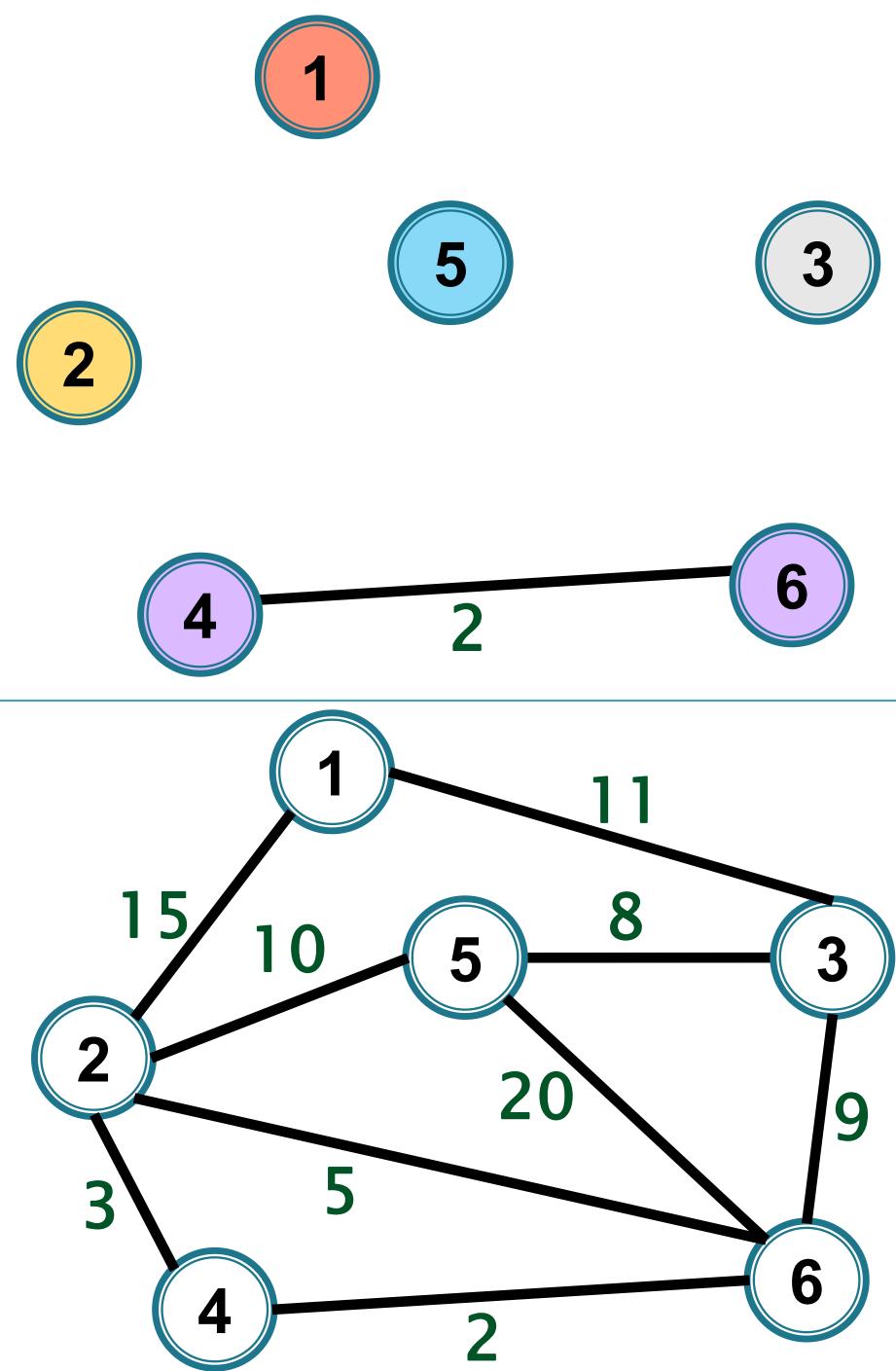
(1,3)

(1,2)

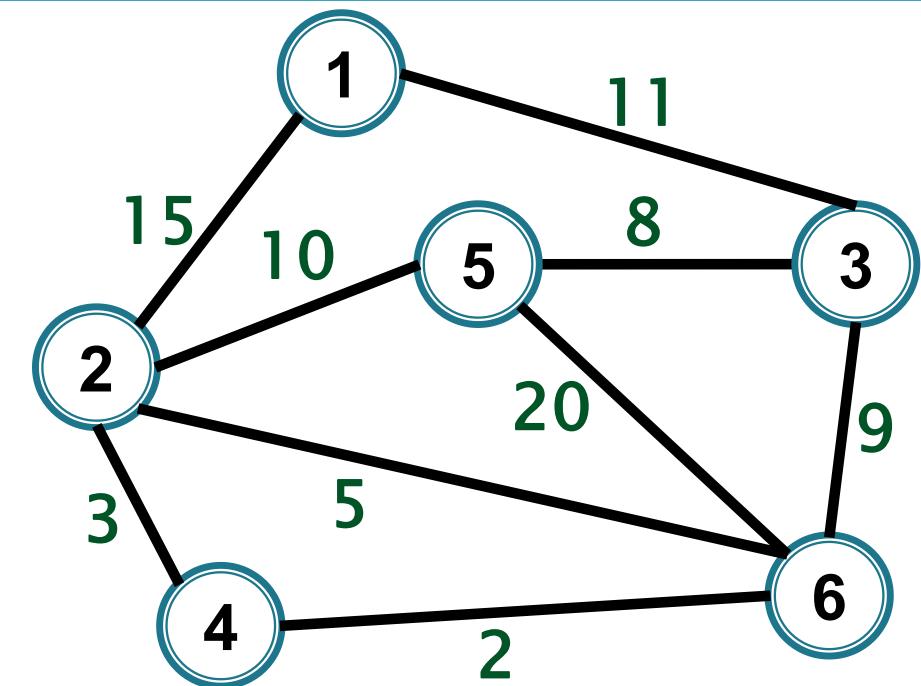
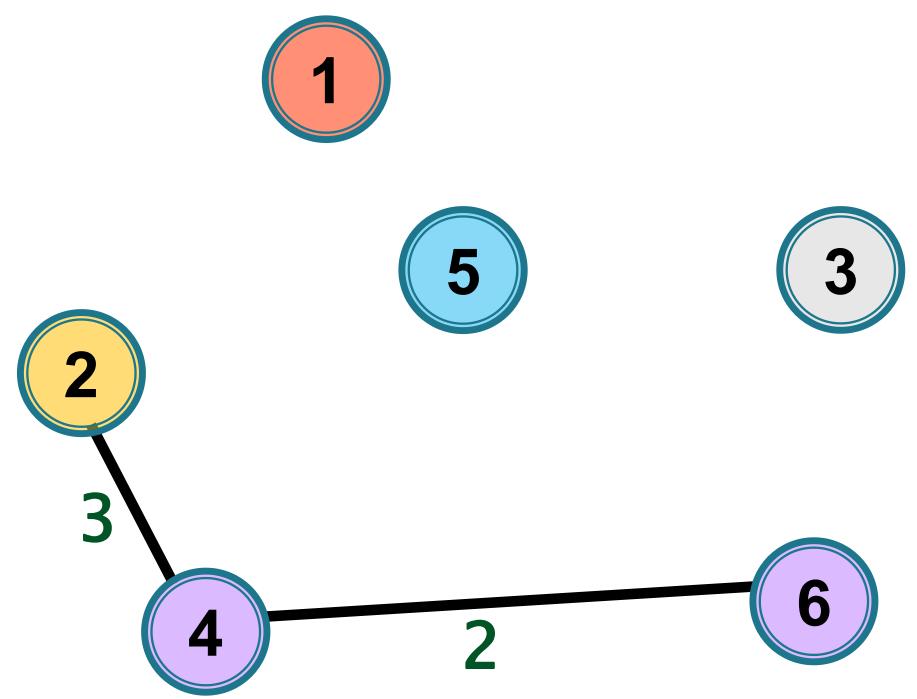
(5,6)



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



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



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

(4,6)

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

(2,4)

(2,6)

(3,5)

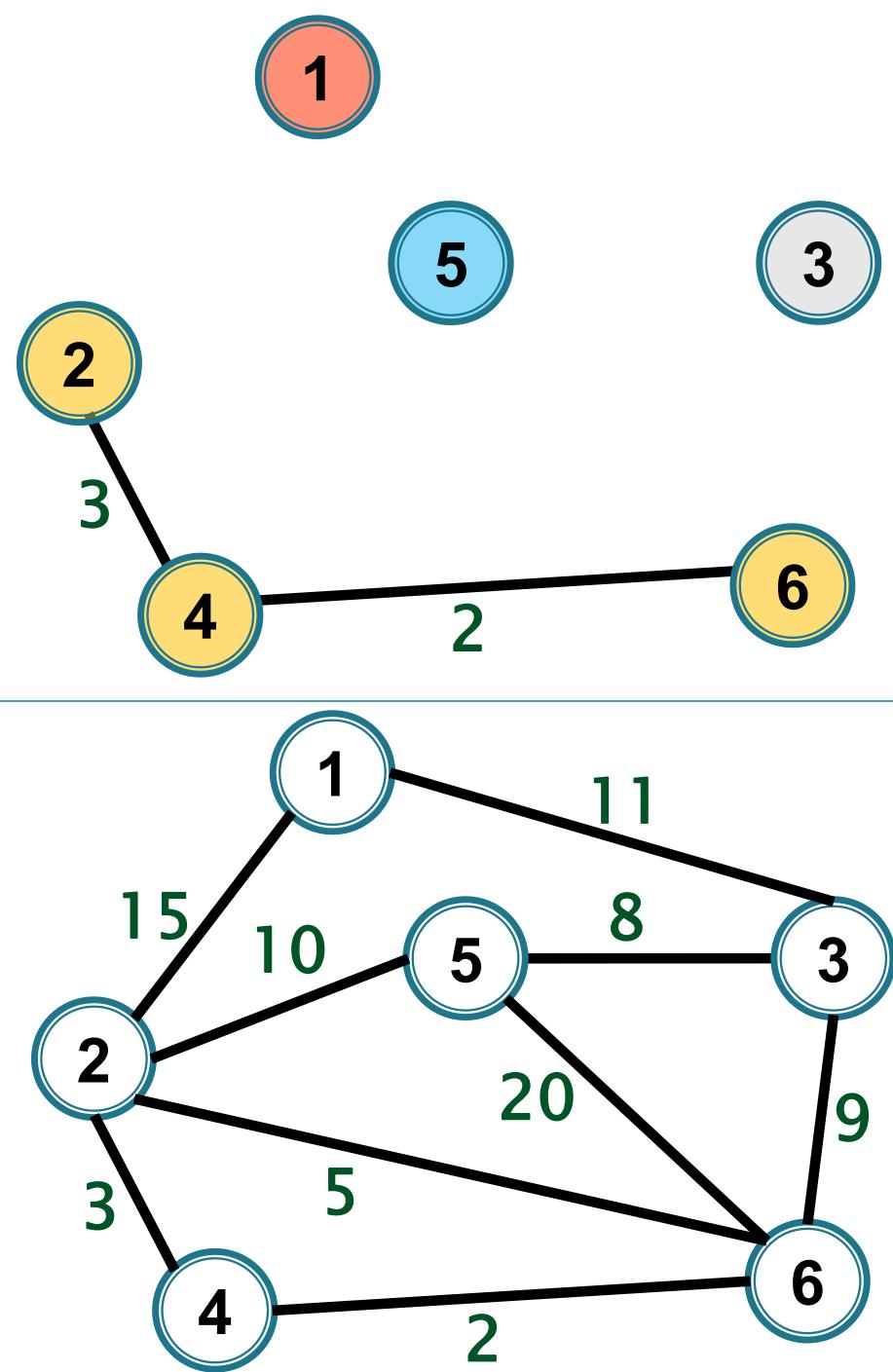
(3,6)

(2,5)

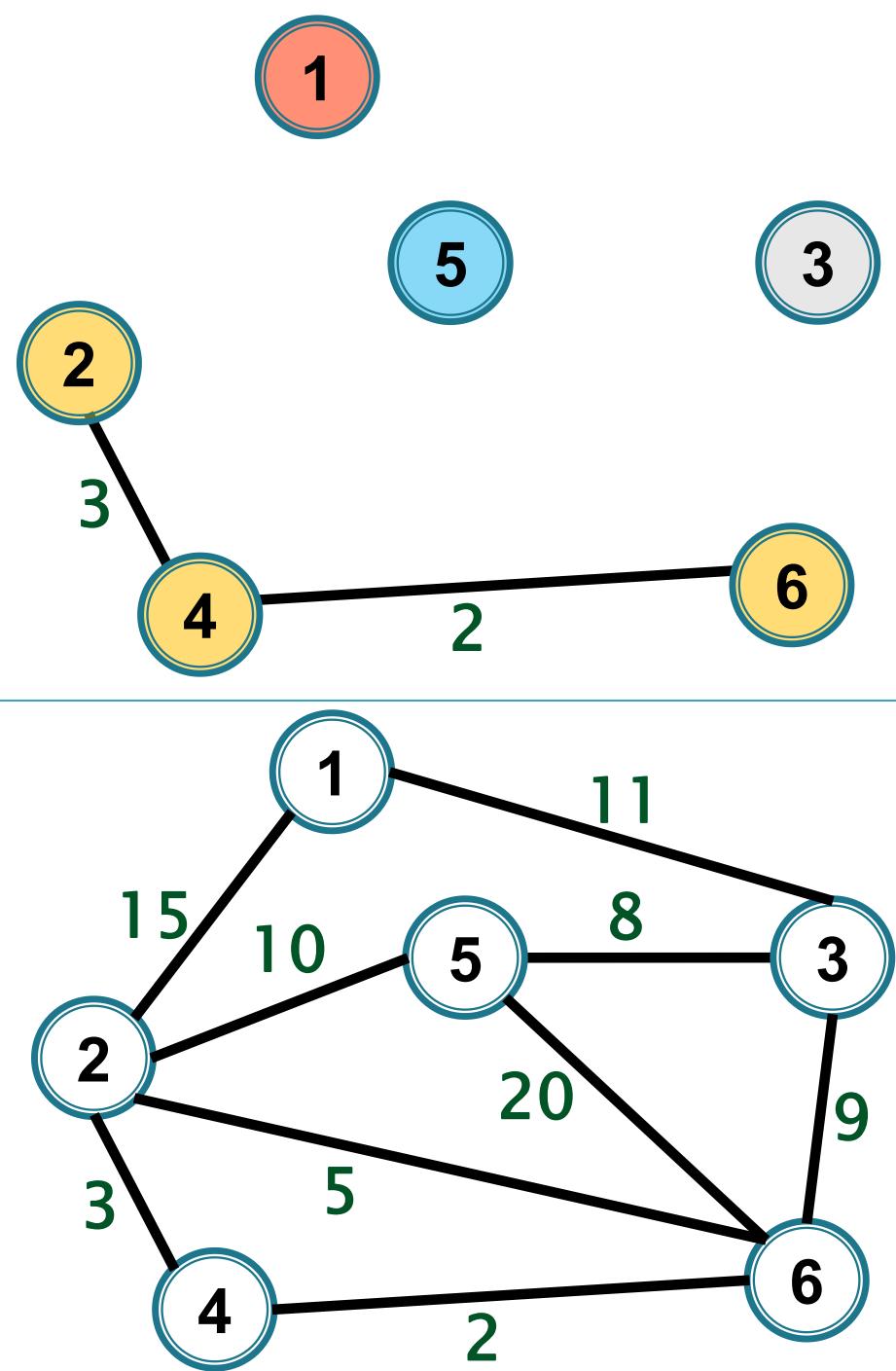
(1,3)

(1,2)

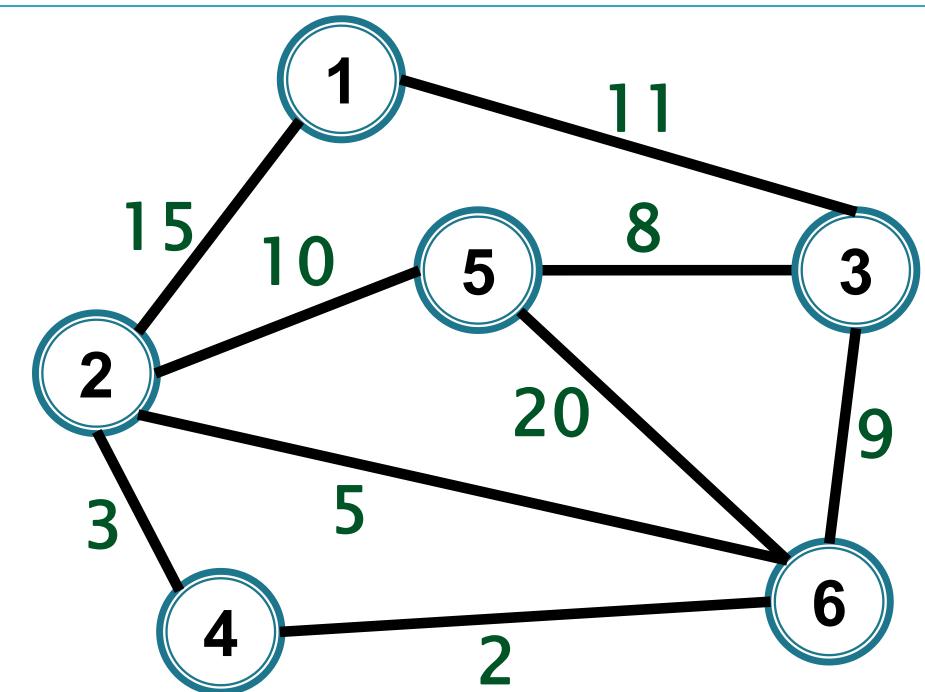
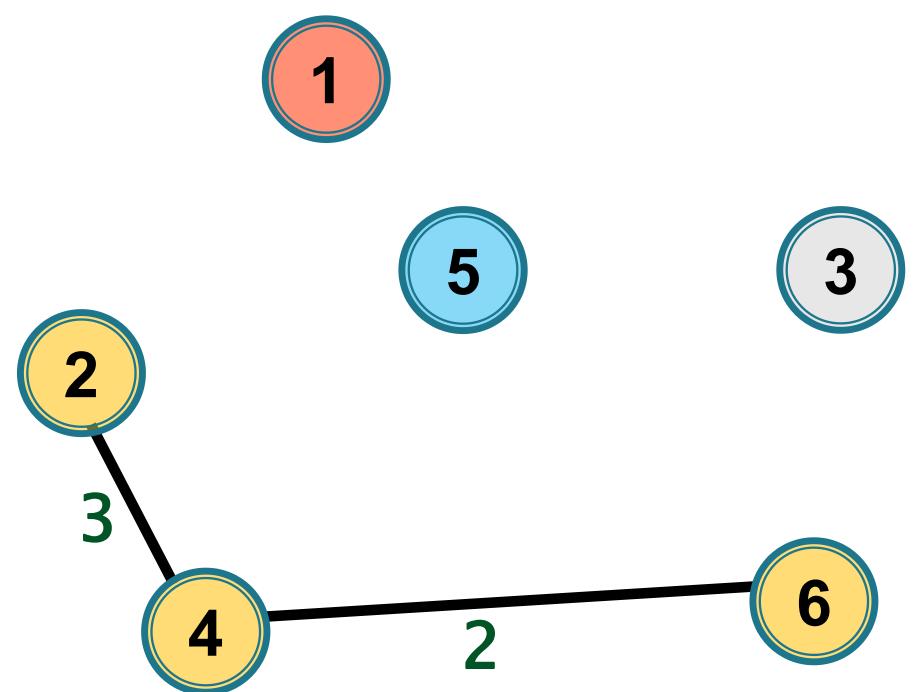
(5,6)



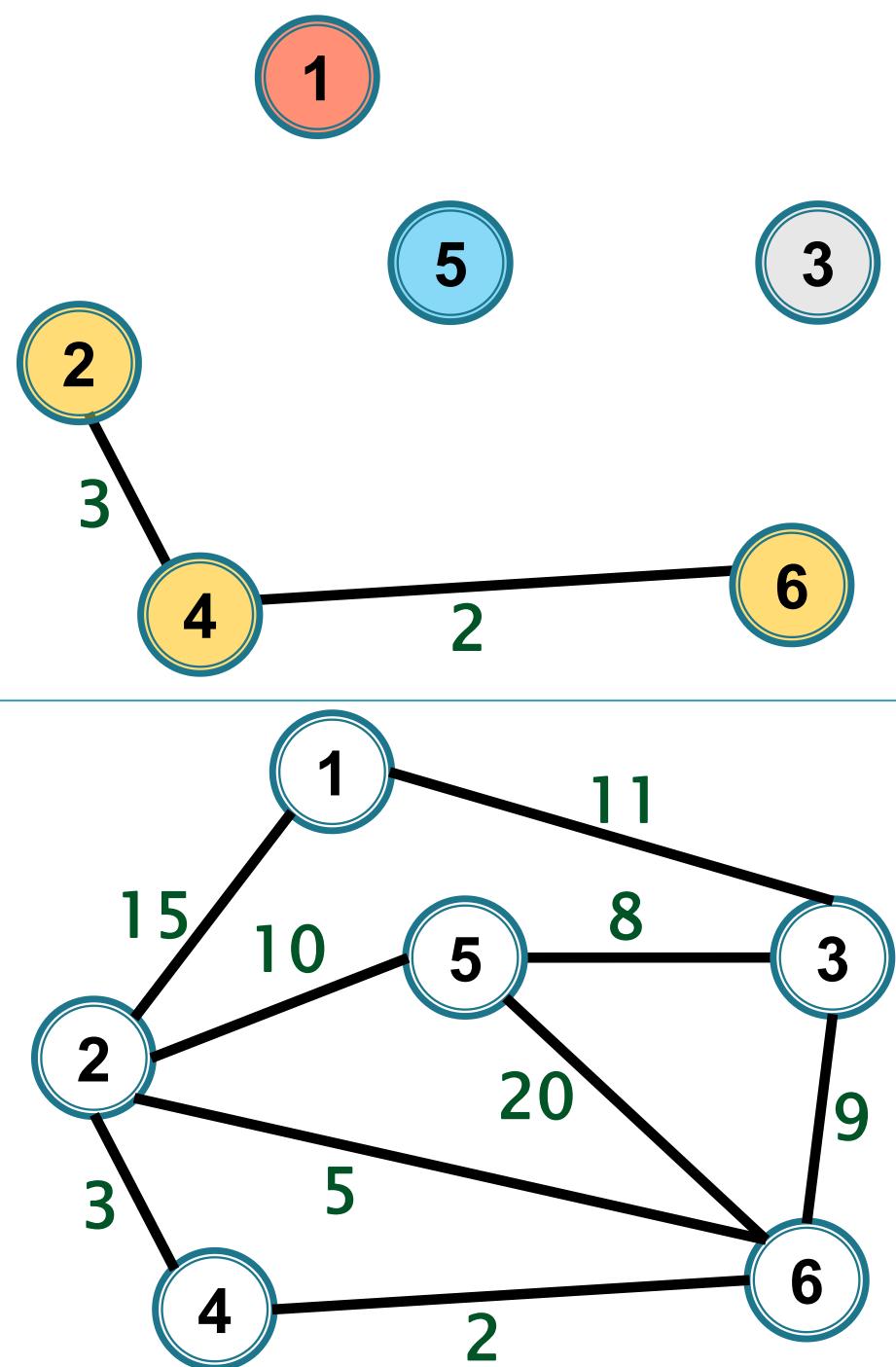
$r = [1,2,3,4,5,6]$   
 $(4,6)$        $r = [1,\underline{2},3,\underline{4},5,4]$   
 $(2,4)$        $r = [1,2,3,\textcolor{teal}{2},5,2]$   
 $(2,6)$   
 $(3,5)$   
 $(3,6)$   
 $(2,5)$   
 $(1,3)$   
 $(1,2)$   
 $(5,6)$



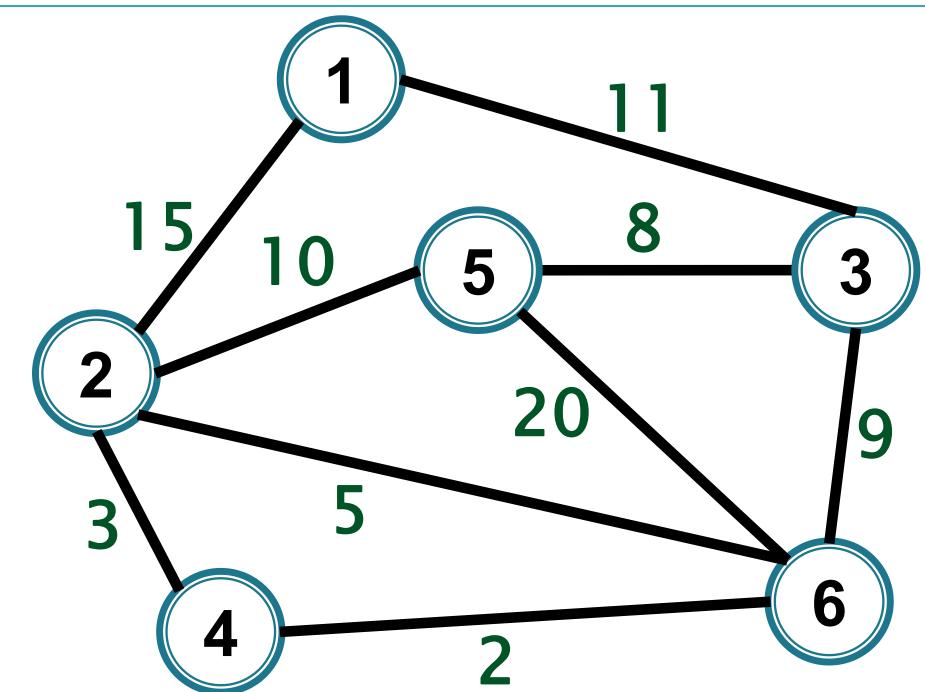
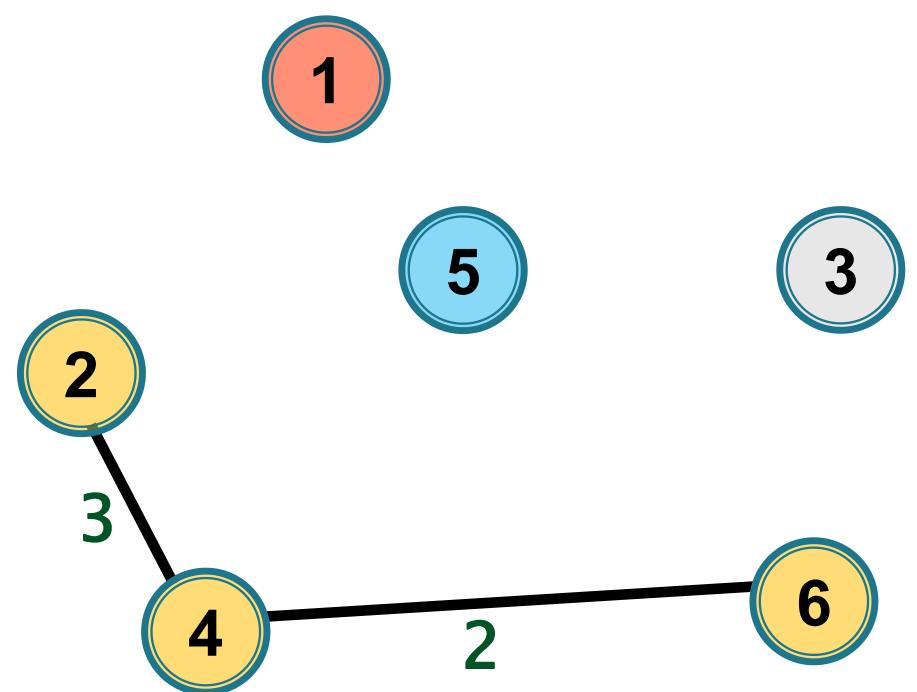
$r = [1,2,3,4,5,6]$   
(4,6)       $r = [1,2,3,4,5,4]$   
(2,4)       $r = [1,2,3,2,5,2]$   
(2,6)  
(3,5)  
(3,6)  
(2,5)  
(1,3)  
(1,2)  
(5,6)



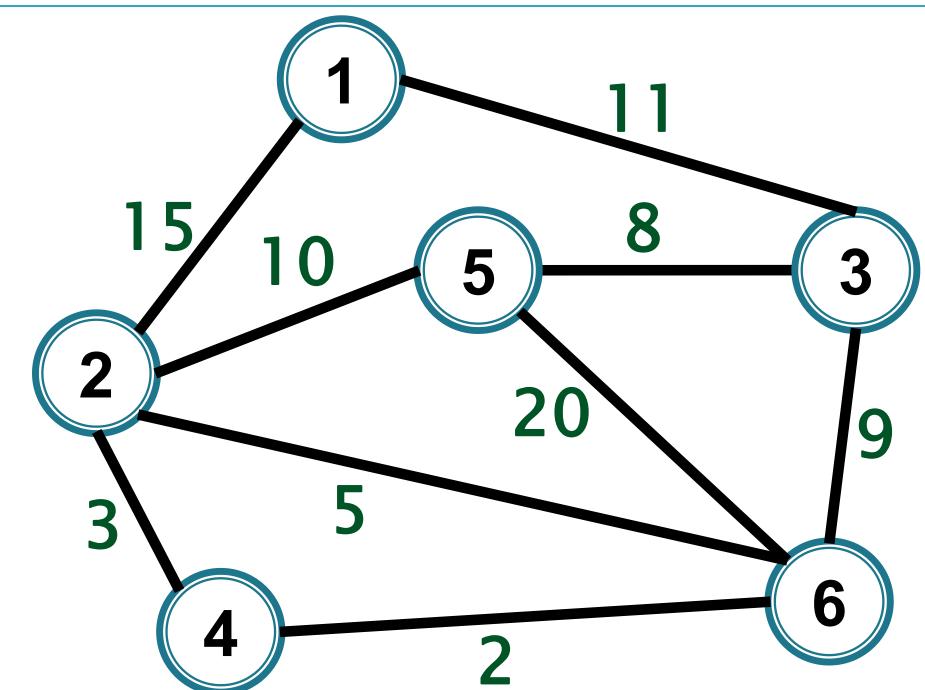
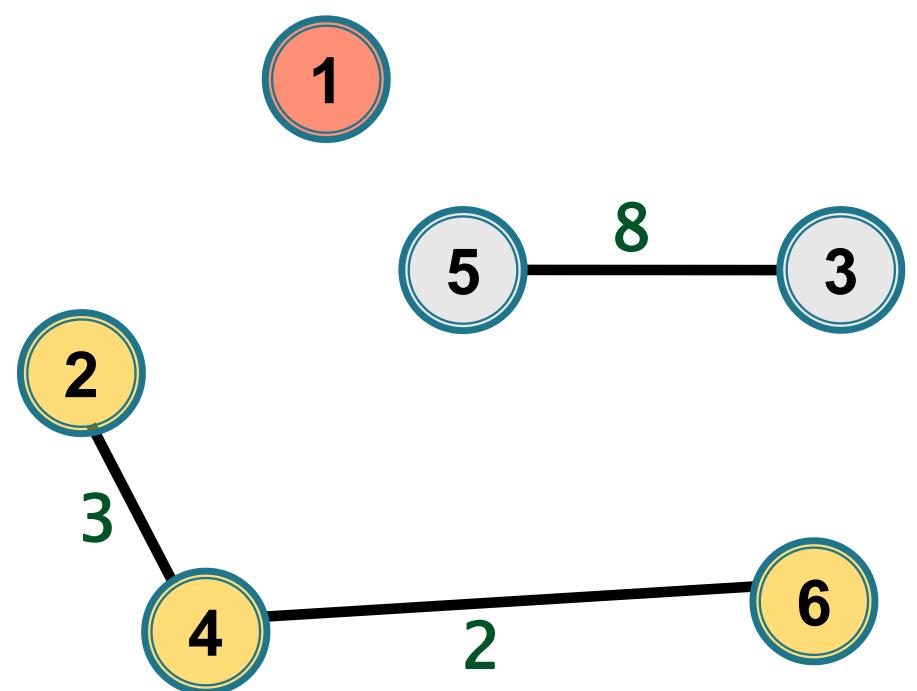
- |                     |   |
|---------------------|---|
| $r = [1,2,3,4,5,6]$ |   |
| $(4,6)$             | $r = [1,2,3,4,5,4]$                         |
| $(2,4)$             | $r = [1,\underline{2},3,2,5,\underline{2}]$ |
| $(2,6)$             | $r(2) = r(6) \rightarrow \text{NU}$         |
| $(3,5)$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



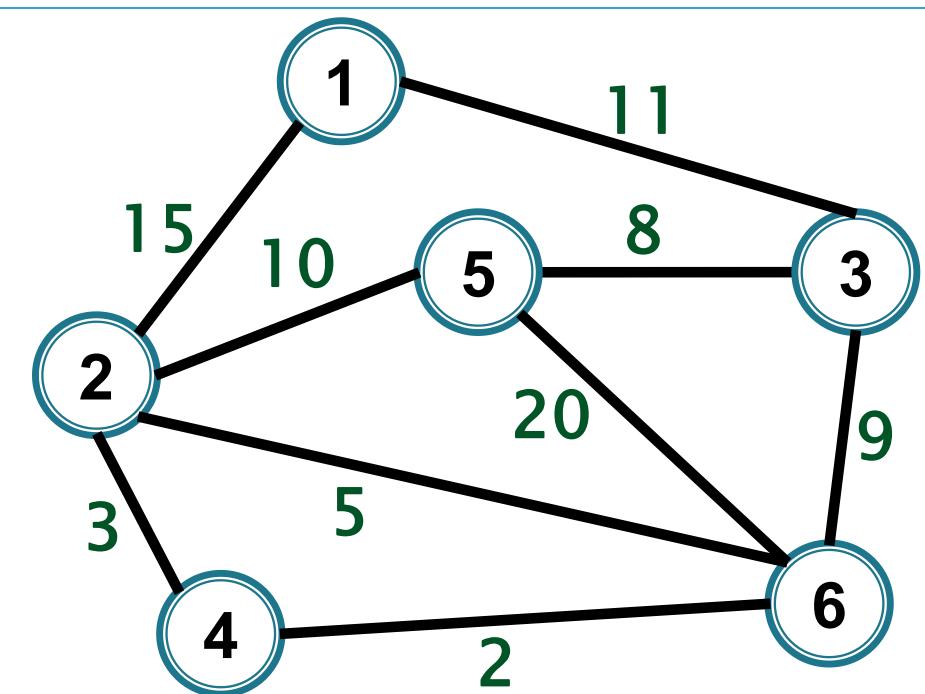
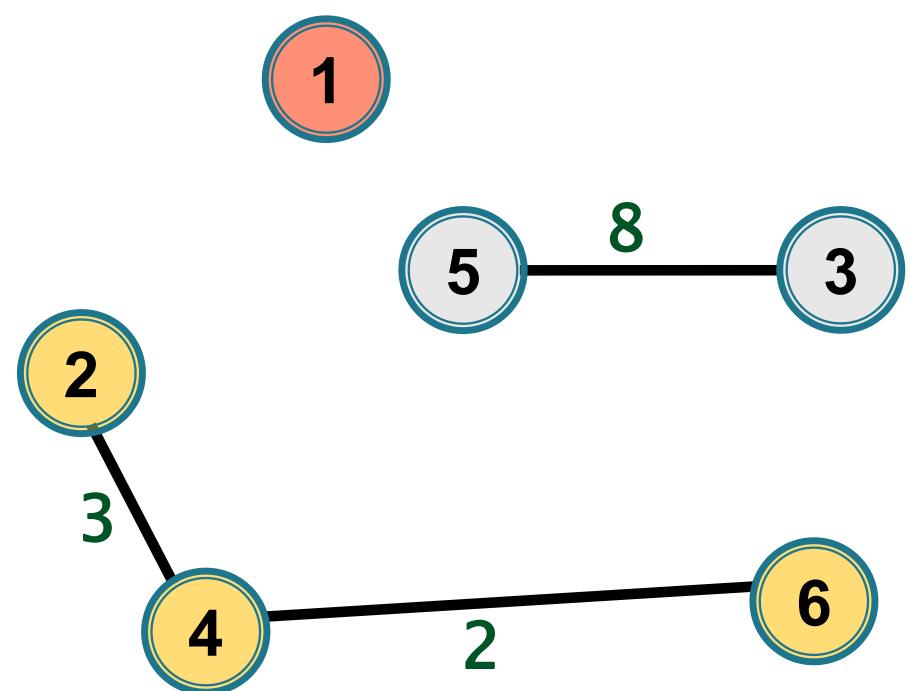
- $r = [1,2,3,4,5,6]$   
 $(4,6) \quad r = [1,2,3,4,5,4]$   
 $(2,4) \quad r = [1,2,3,2,5,2]$   
 $(2,6) \quad r(2) = r(6) \rightarrow \text{NU}$   
 $(3,5)$   
 $(3,6)$   
 $(2,5)$   
 $(1,3)$   
 $(1,2)$   
 $(5,6)$



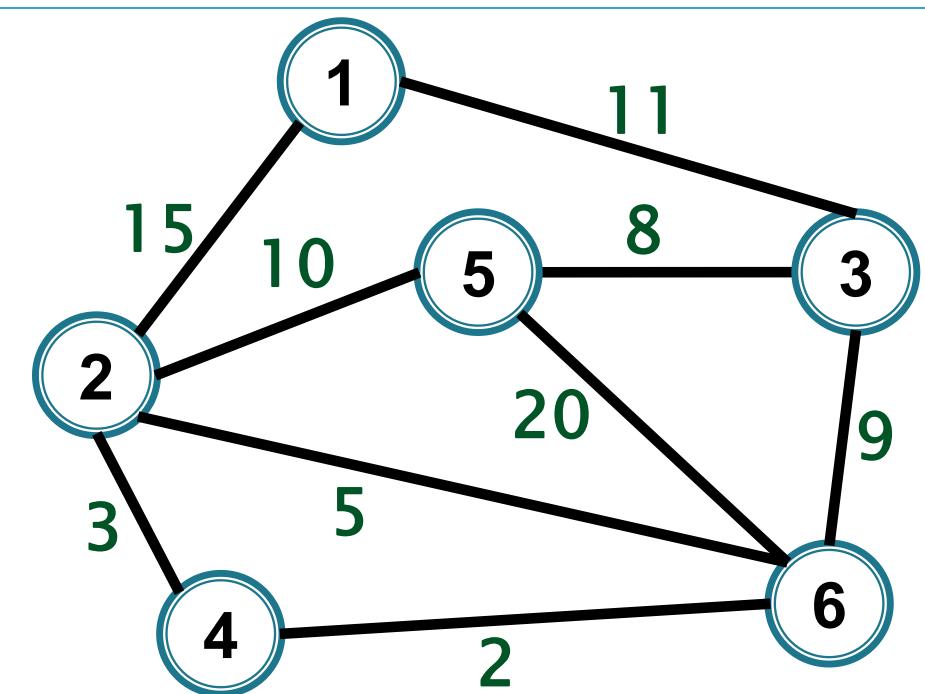
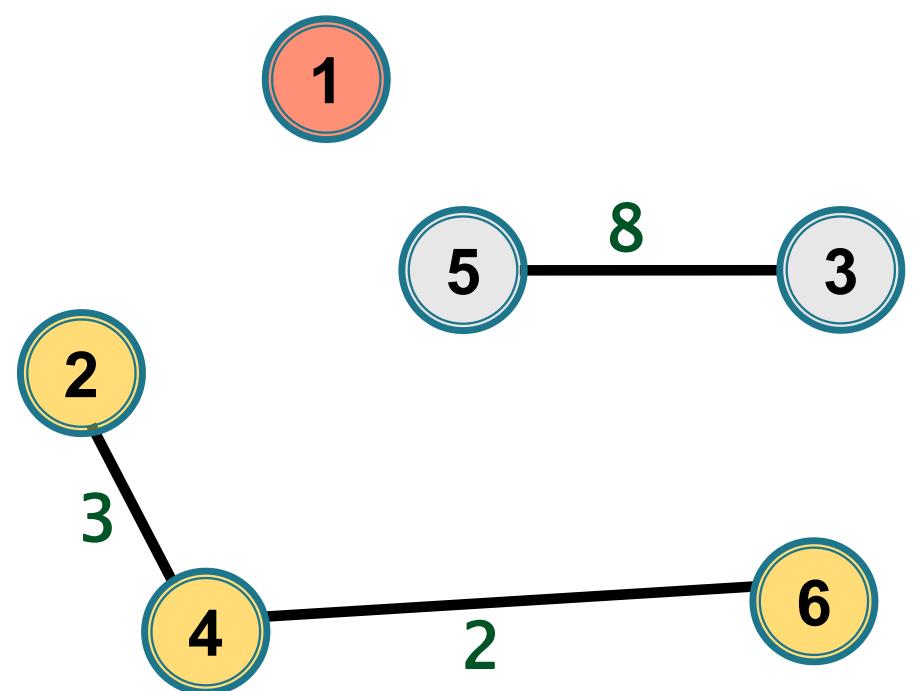
- |                     |   |
|---------------------|---|
| $r = [1,2,3,4,5,6]$ |   |
| $(4,6)$             | $r = [1,2,3,4,5,4]$                         |
| $(2,4)$             | $r = [1,2,\underline{3},2,\underline{5},2]$ |
| $(2,6)$             | $r(2) = r(6) \rightarrow \text{NU}$         |
| $(3,5)$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



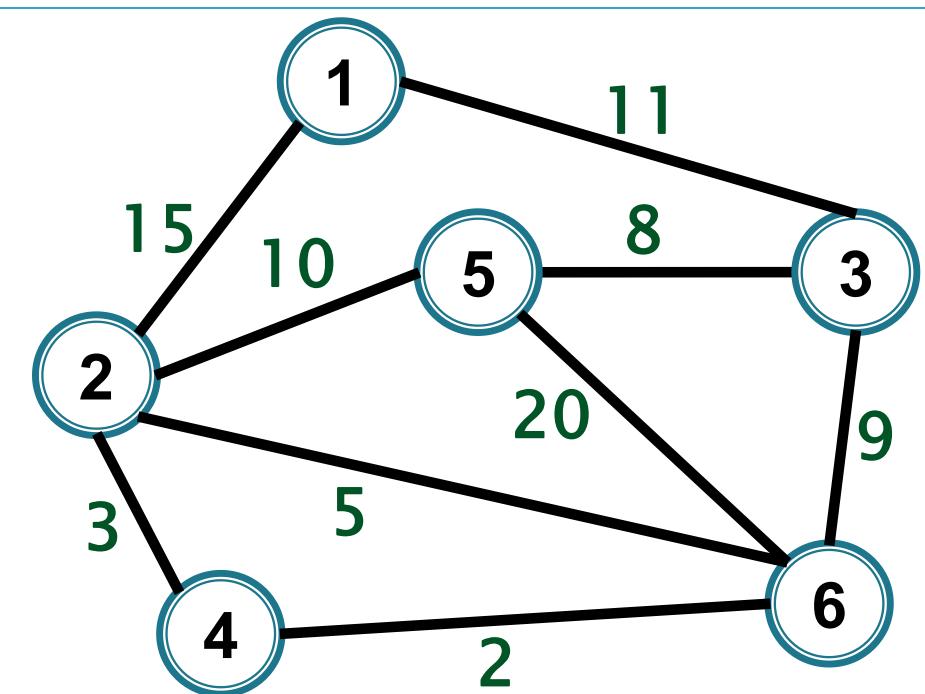
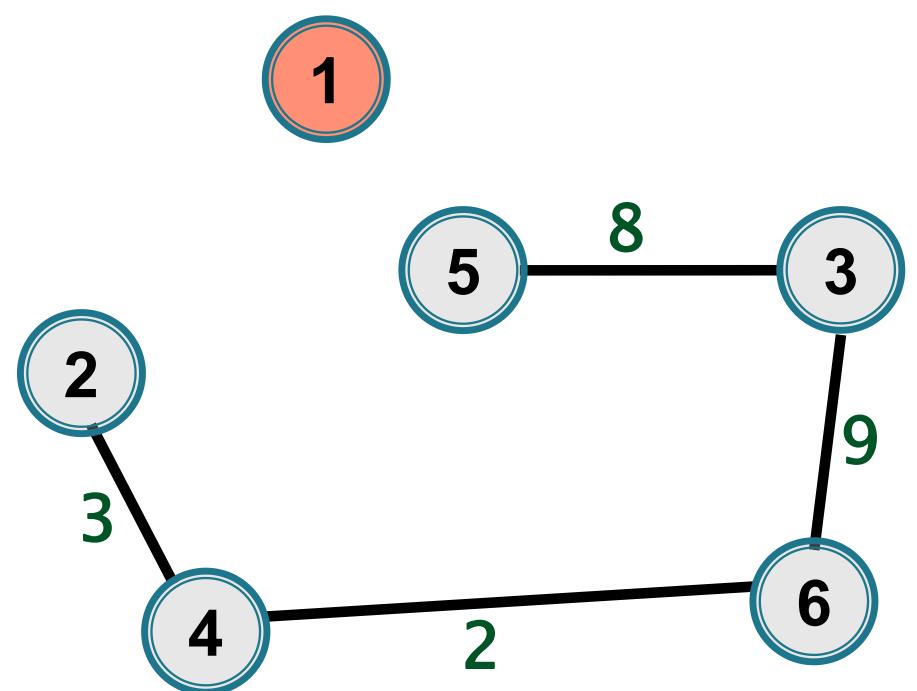
- |                     |   |
|---------------------|---|
| $r = [1,2,3,4,5,6]$ |   |
| $(4,6)$             | $r = [1,2,3,4,5,4]$                         |
| $(2,4)$             | $r = [1,2,\underline{3},2,\underline{5},2]$ |
| $(2,6)$             | $r(2) = r(6) \rightarrow \text{NU}$         |
| $(3,5)$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



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



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



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

$(4,6) \quad r = [1,2,3,4,5,4]$

$(2,4) \quad r = [1,2,3,2,5,2]$

$(2,6) \quad r(2) = r(6) \rightarrow \text{NU}$

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

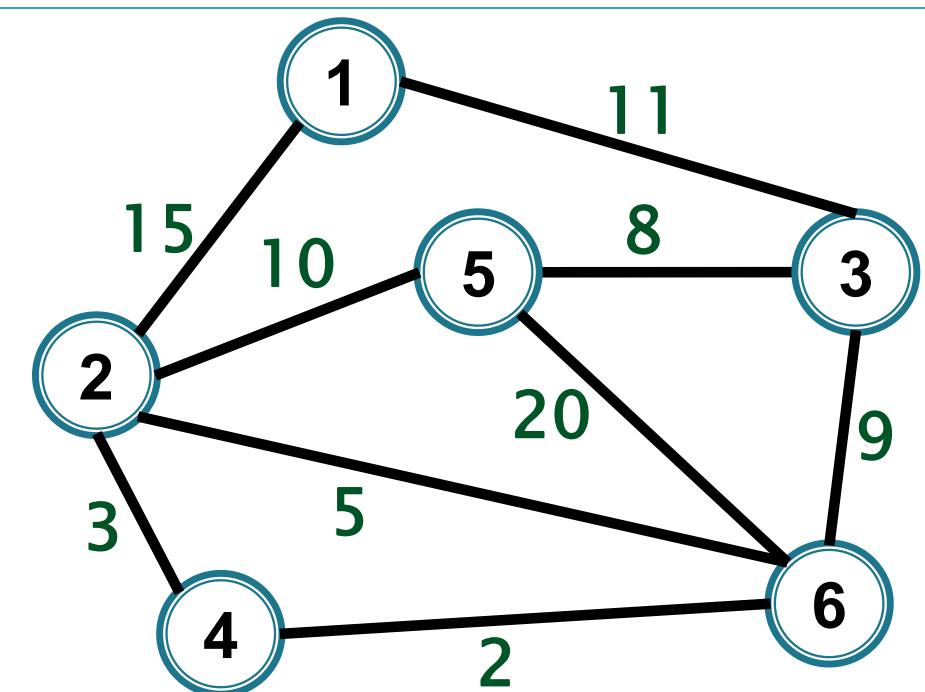
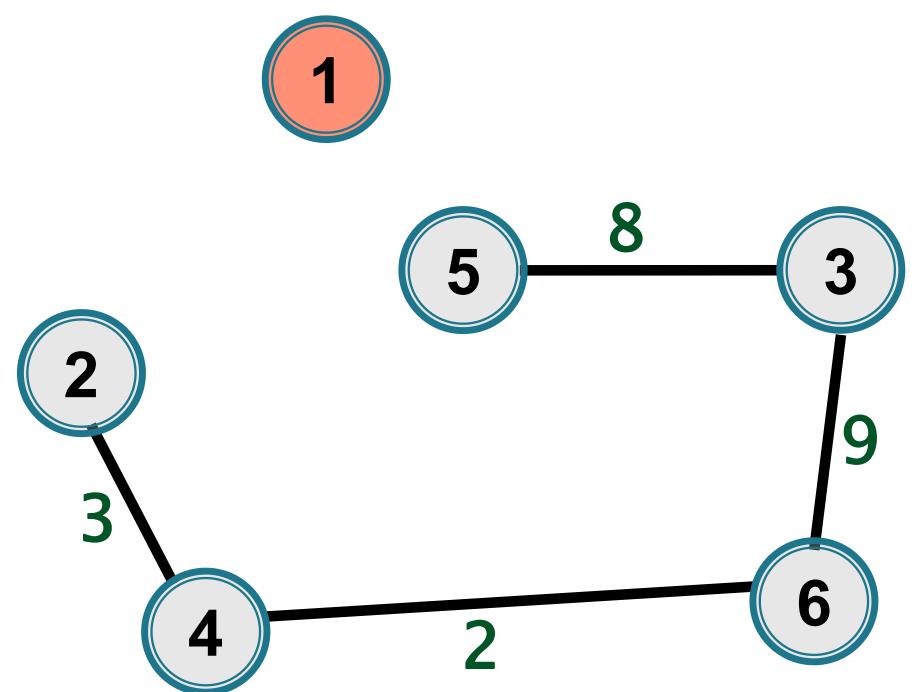
$(3,6) \quad r = [1,3,3,3,3,3]$

$(2,5)$

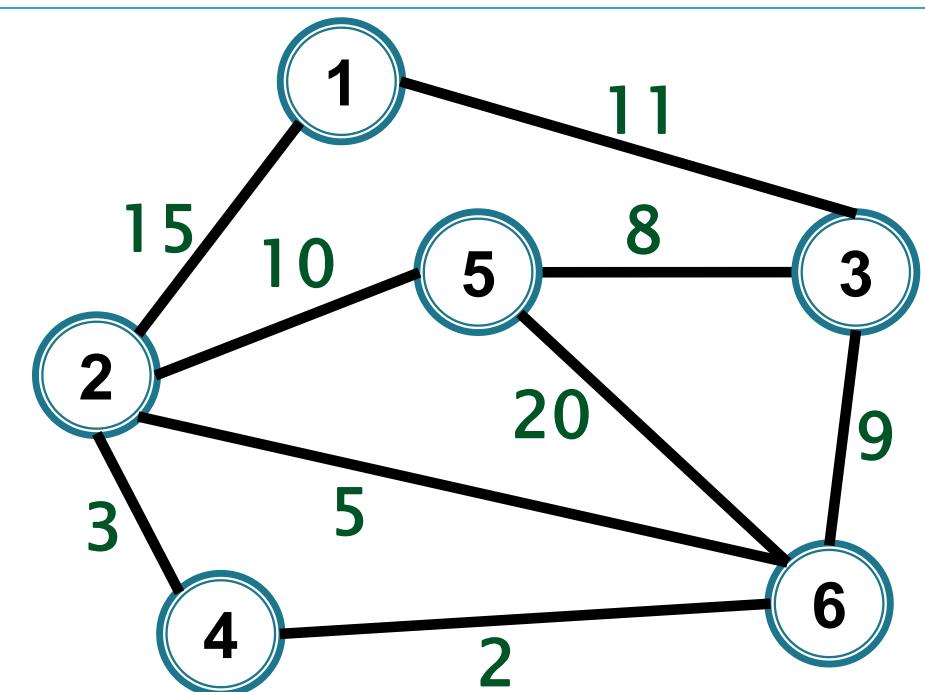
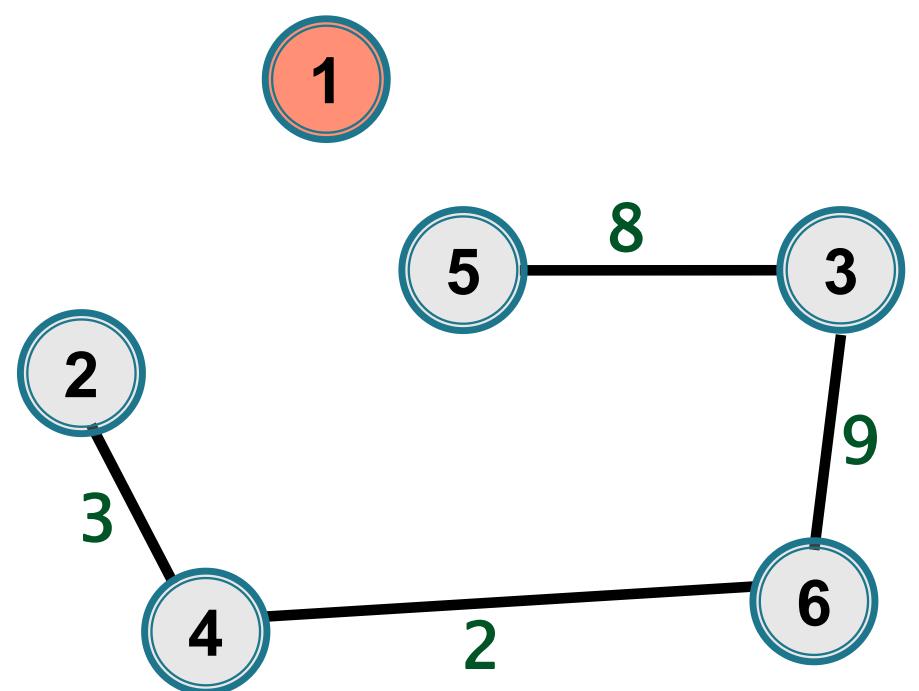
$(1,3)$

$(1,2)$

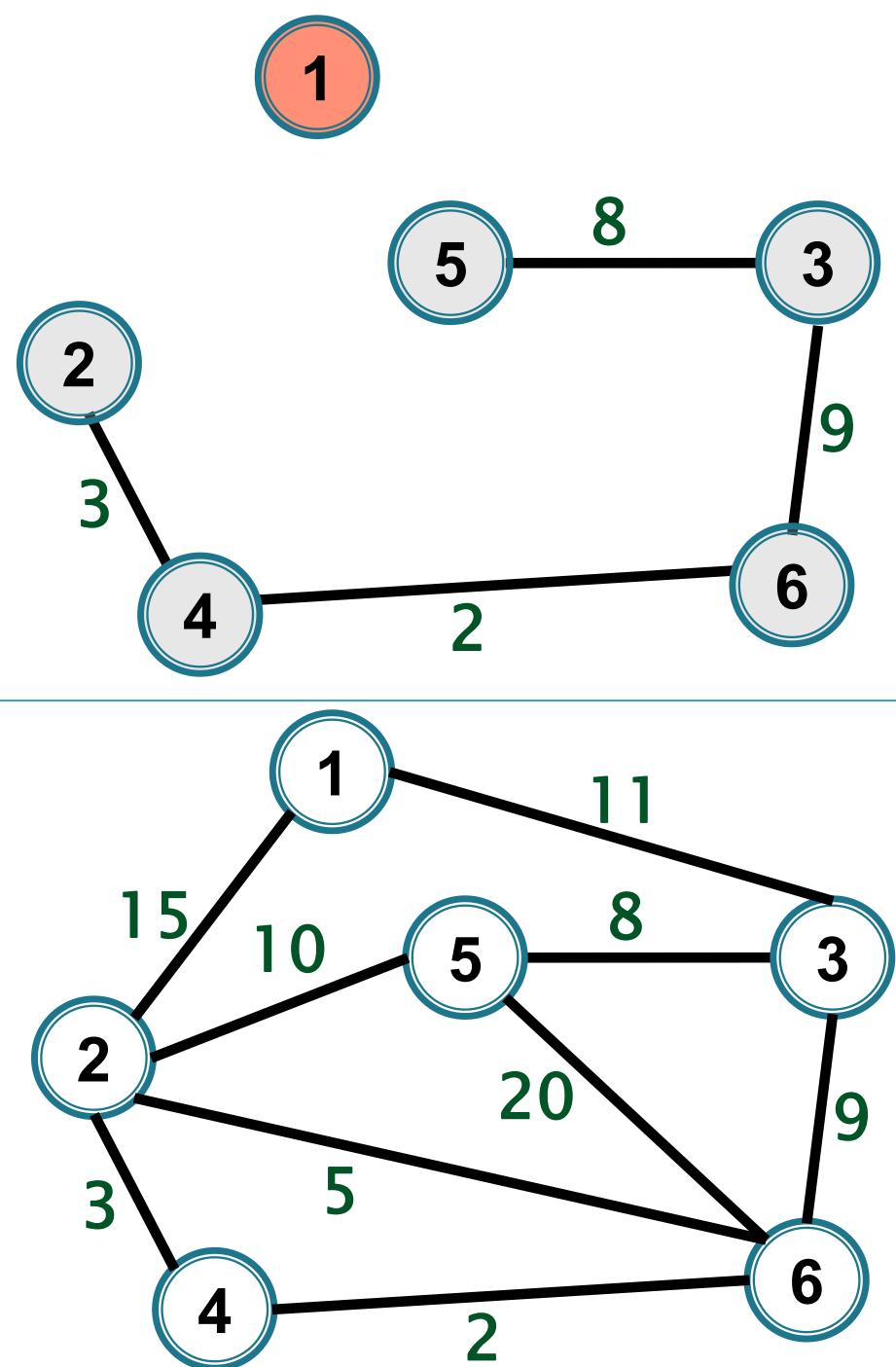
$(5,6)$



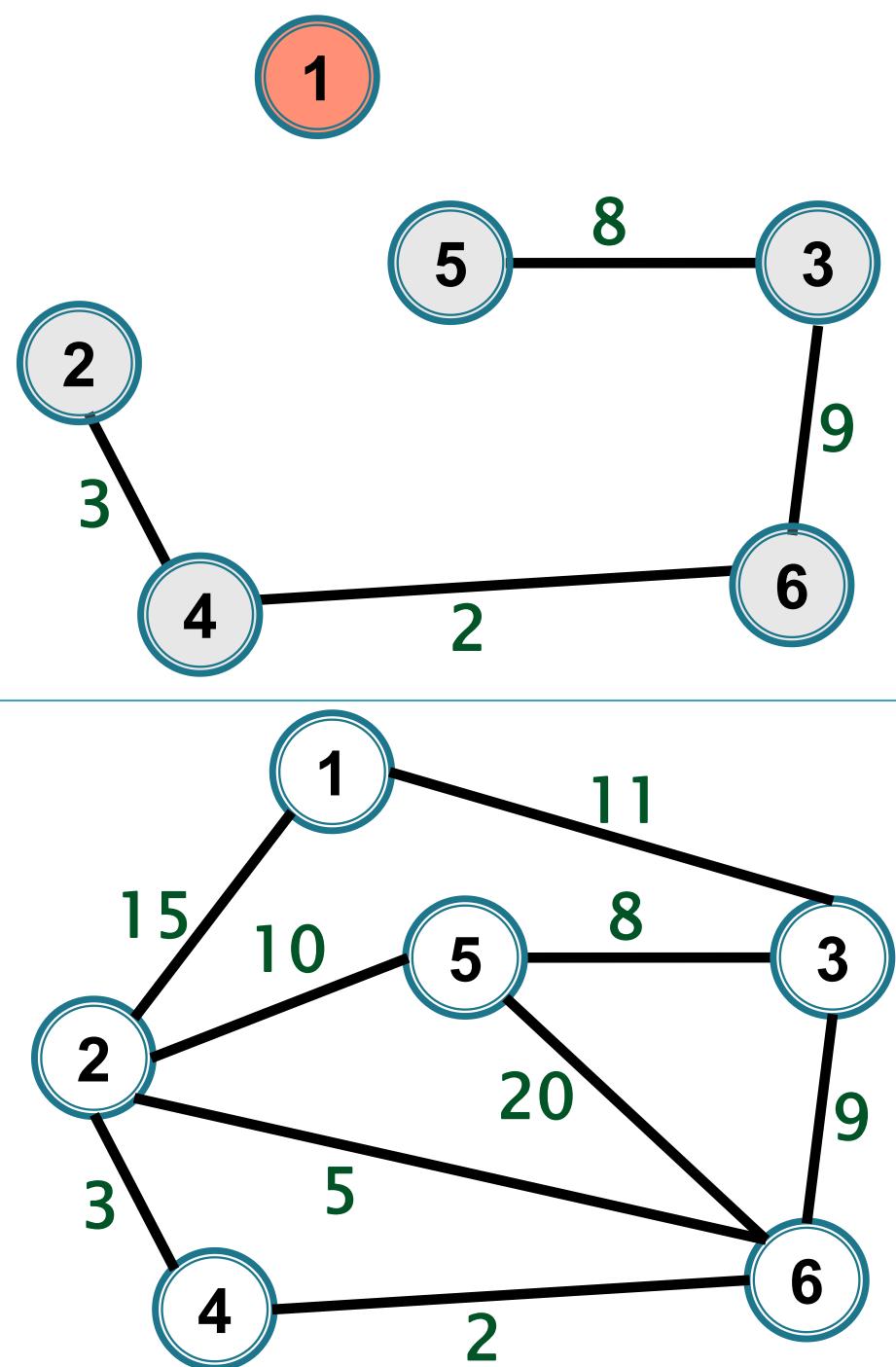
- |       |                                     |
|-------|-------------------------------------|
|       | $r = [1,2,3,4,5,6]$                 |
| (4,6) | $r = [1,2,3,4,5,4]$                 |
| (2,4) | $r = [1,2,3,2,5,2]$                 |
| (2,6) | $r(2) = r(6) \rightarrow \text{NU}$ |
| (3,5) | $r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



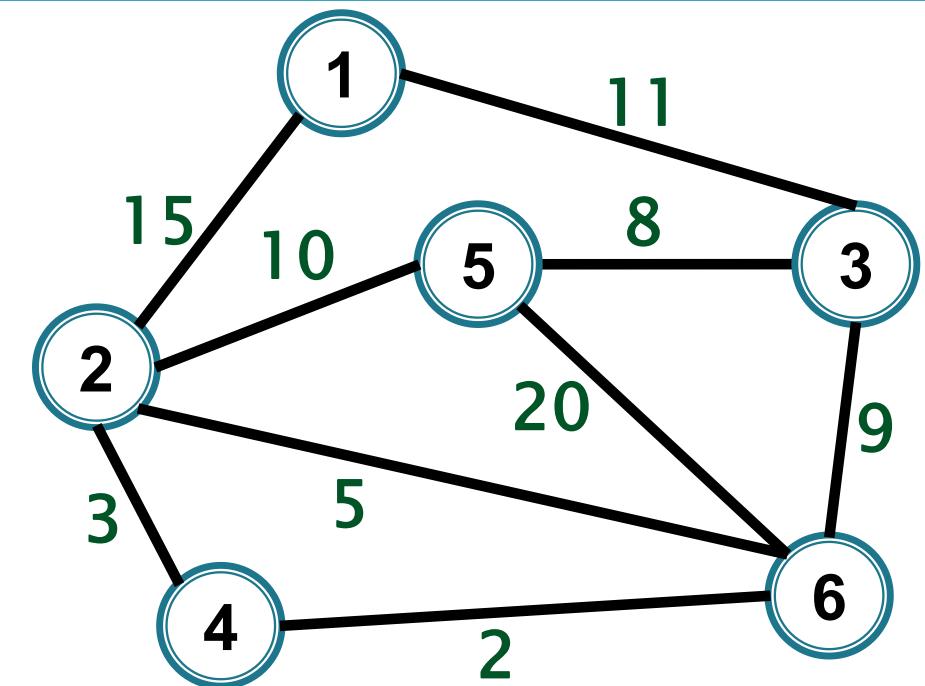
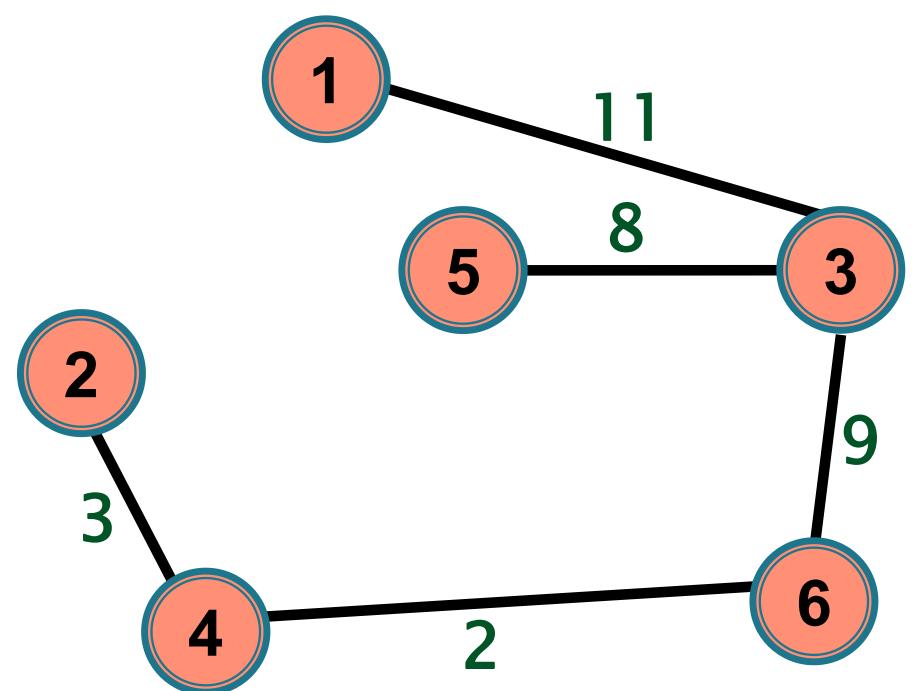
- |       |   |
|-------|---|
|       | $r = [1,2,3,4,5,6]$                         |
| (4,6) | $r = [1,2,3,4,5,4]$                         |
| (2,4) | $r = [1,2,3,2,5,2]$                         |
| (2,6) | $r(2) = r(6) \rightarrow \text{NU}$         |
| (3,5) | $r = [1,2,3,2,3,2]$                 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



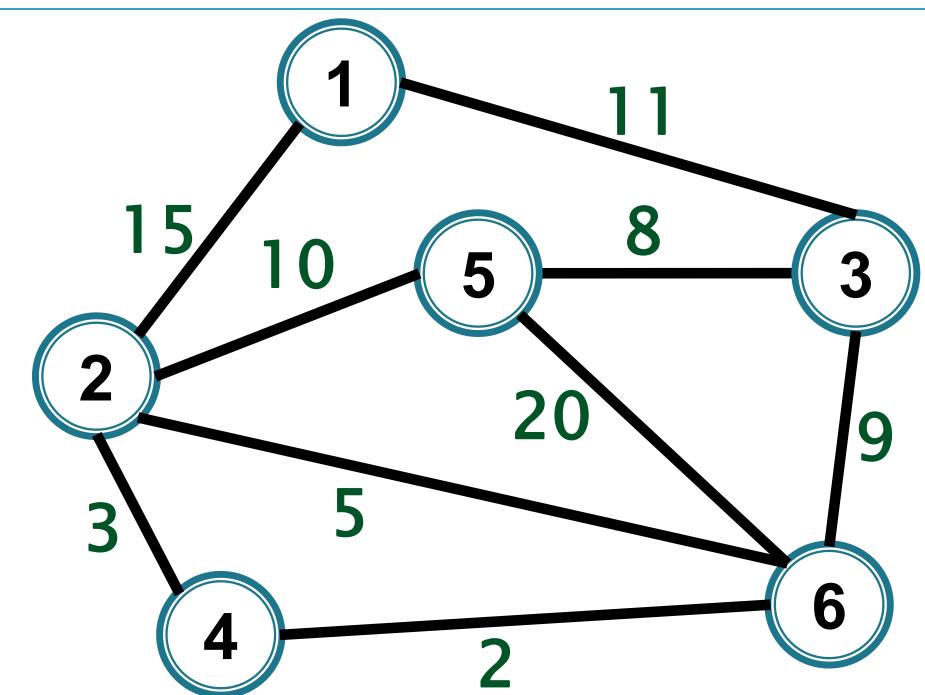
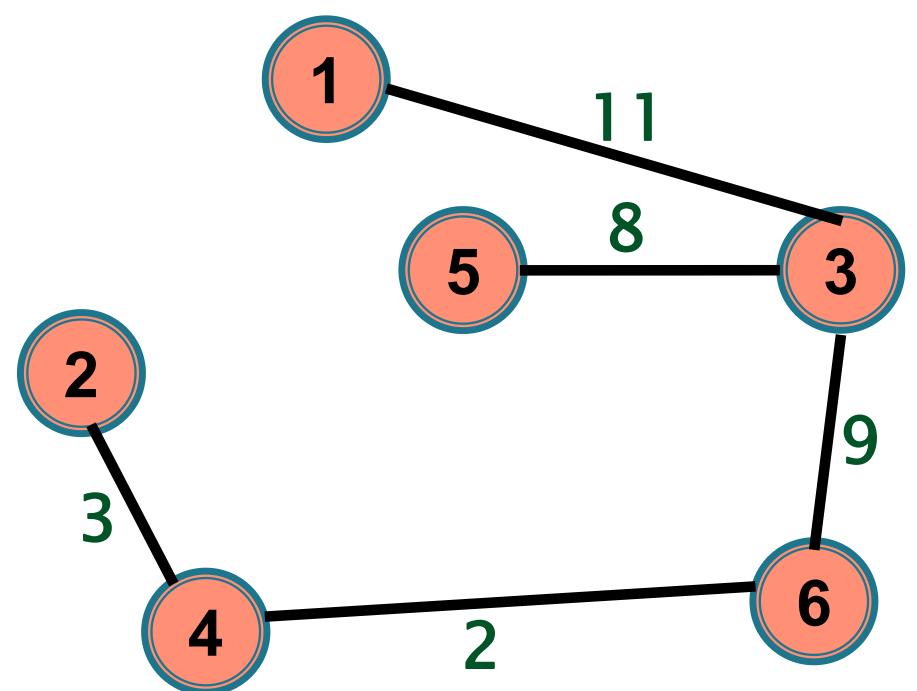
- |                          |                                     |
|--------------------------|-------------------------------------|
| $r = [1, 2, 3, 4, 5, 6]$ |                                     |
| (4,6)                    | $r = [1, 2, 3, 4, 5, 4]$            |
| (2,4)                    | $r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



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



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



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

# Kruskal



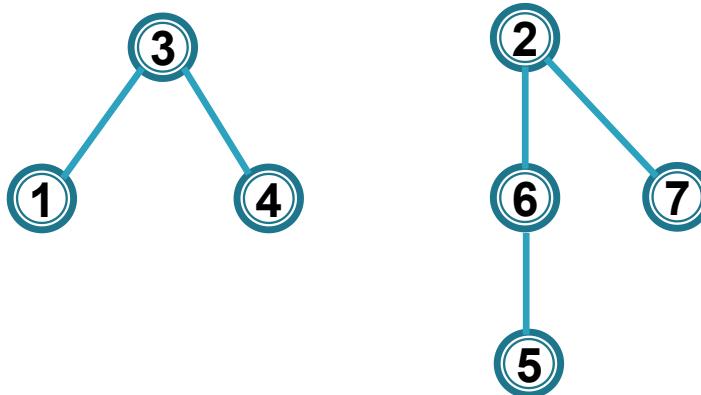
**Varianta 2 – Structuri pentru mulțimi disjuncte  
Union/Find**

# Kruskal



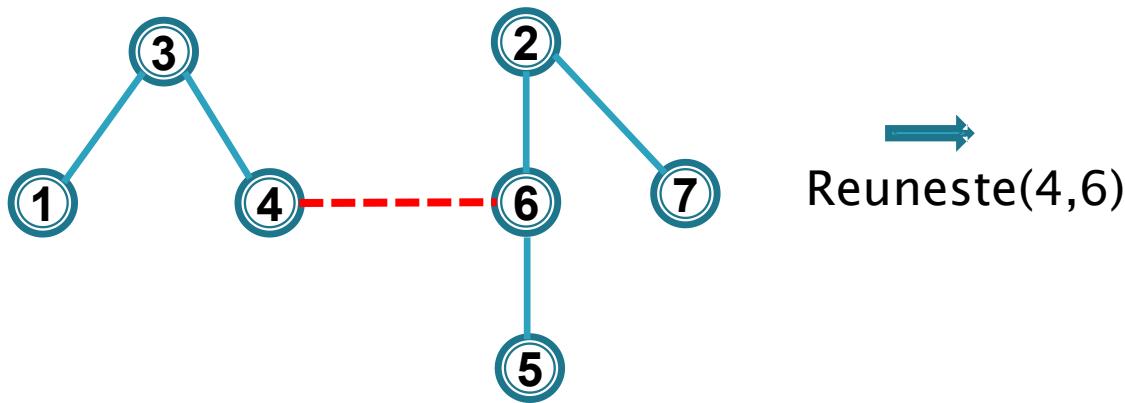
## Varianta 2 – Structuri pentru mulțimi disjuncte Union/Find – arbori

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



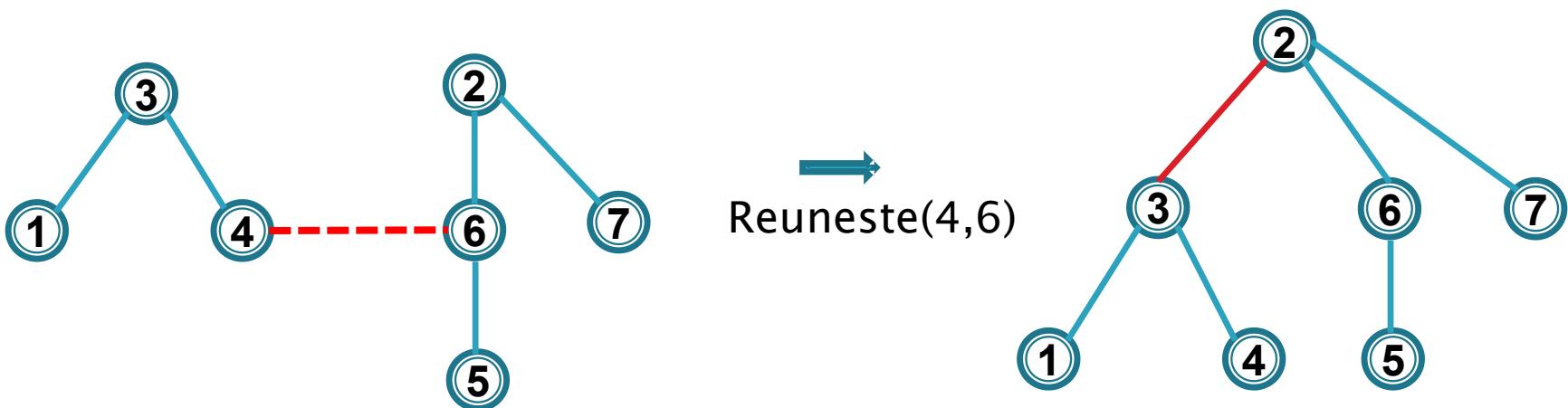
# Kruskal

- **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)



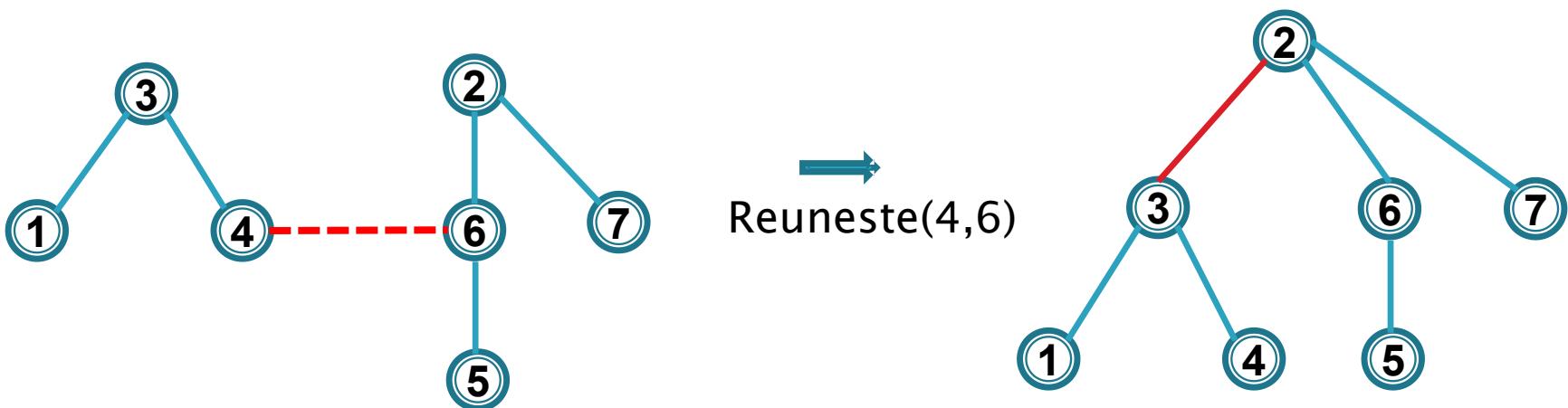
- arborele cu înălțimea mai mică devine subarbore al rădăcinii celuilalt arbore

# 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 mulțimi disjuncte:

- **Initializare**
- **Reprez(u)**  $\Rightarrow$  determinarea rădăcinii arborelui care conține u
- **Reuneste(u,v)**  $\Rightarrow$  reuniune ponderată

ASD + laborator AG

# 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 Initialize(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])  
        h[ru]=h[rv];  
    else  
        h[rv]=h[ru];  
    tata[rv]=u;
```

# Kruskal

```
void Initialize(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 Initialize(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**                   $\rightarrow O(m \log m) = O(m \log n)$
  - **$n * \text{Initializare}$**      $\rightarrow O(n)$
  - **$2m * \text{Reprez}$**          $\rightarrow$
  - **$(n-1) * \text{Reuneste}$**   $\rightarrow$
-

# Kruskal

## Complexitate

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

- **Sortare**                   $\rightarrow O(m \log m) = O(m \log n)$
  - **n \* Initializare**       $\rightarrow O(n)$
  - **2m \* Reprez**             $\rightarrow O(m \log n)$
  - **(n-1) \* Reuneste**     $\rightarrow O(n \log n)$
-

# Kruskal

## Complexitate

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

- **Sortare**                   $\rightarrow O(m \log m) = O(m \log n)$
- **n \* Initializare**       $\rightarrow O(n)$
- **2m \* Reprez**             $\rightarrow O(m \log n)$
- **(n-1) \* Reuneste**     $\rightarrow O(n \log n)$

---

$O(m \log n)$

# 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 diferite*

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 diferite*

**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

# Aplicații – 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
  - $k = \text{numărul de clase}$

# Aplicații – Clustering

## Definiții

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

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

# Aplicații – Clustering

## Definiții

- ▶ Un **k-clustering** a 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  $\mathcal{C}$

= distanța minimă dintre două obiecte aflate în clase diferite

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

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

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

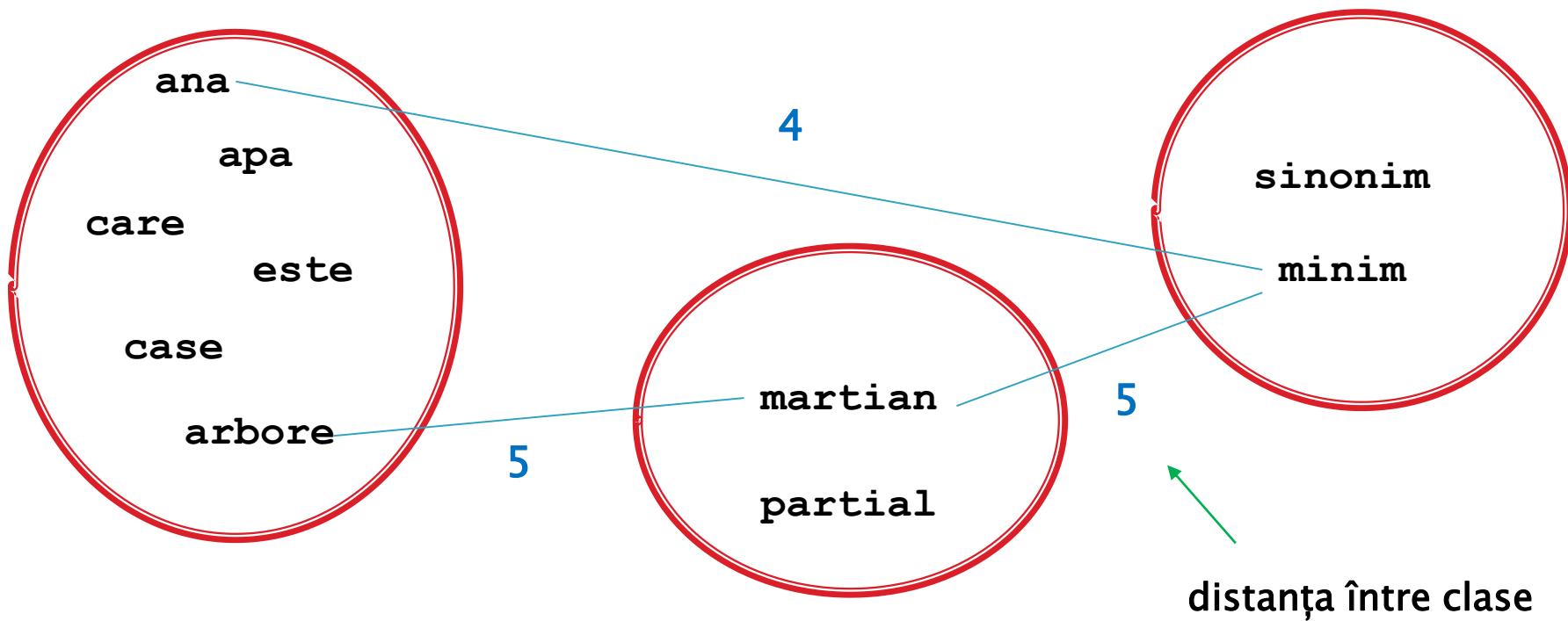
# Aplicații - Clustering

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

este	martian	care
ana	apa	sinonim
minim	partial	
arbore		case

# Aplicații – Clustering

- ▶ obiecte = cuvinte,
- ▶  $d$  = distanță de editare
- ▶  $k = 3$



3-clustering cu gradul de separare = 4

# Aplicații – Clustering

## Problemă Clustering:

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

# Aplicații – Clustering



# Idee

este

## **martian**

care

ana

apa

**sinonim**

## minim

partial

## case

## arbore

# Aplicații – 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

# Aplicații – 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

## Aplicații – Clustering

# Cuvinte – distanță de editare

martian

care

este

apa

### **sinonim**

ana

## minim

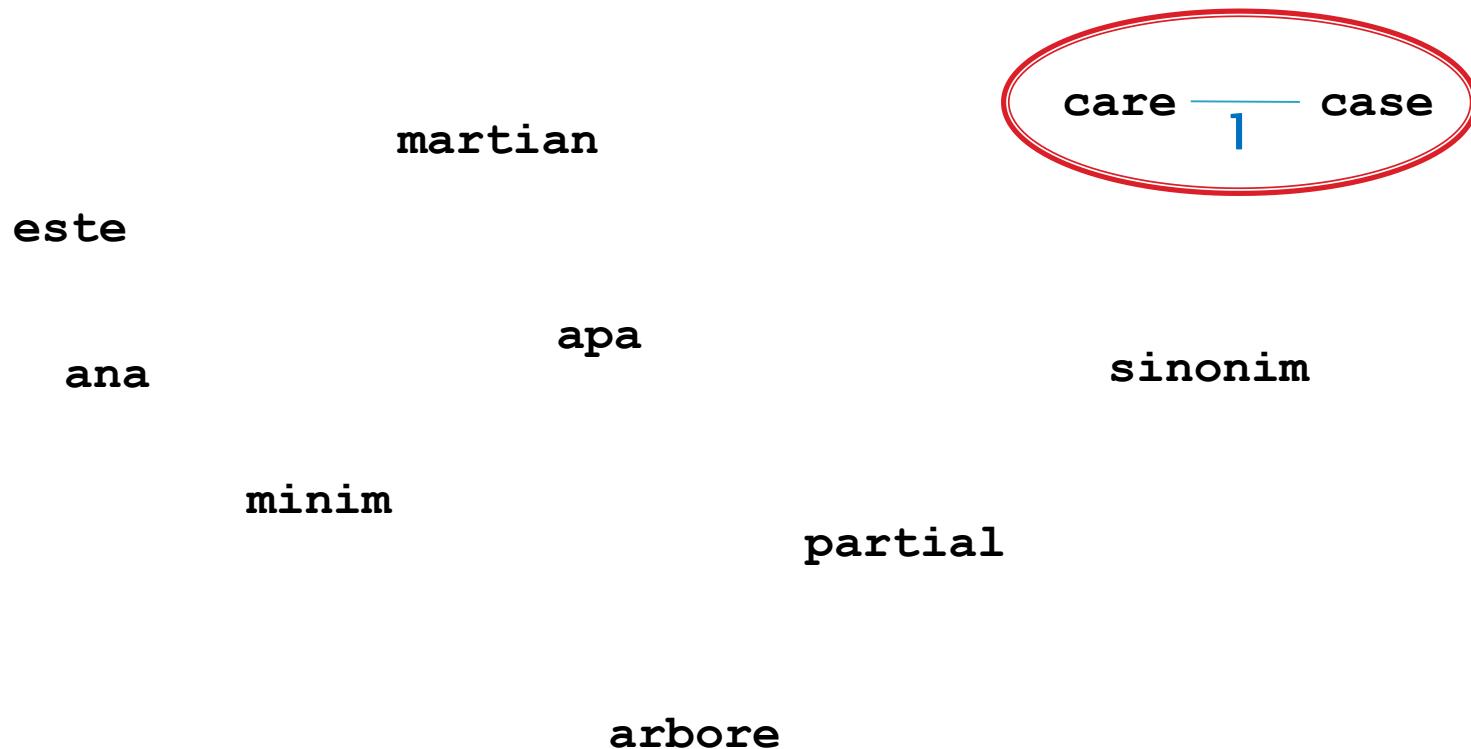
**partial**

arbore

## case

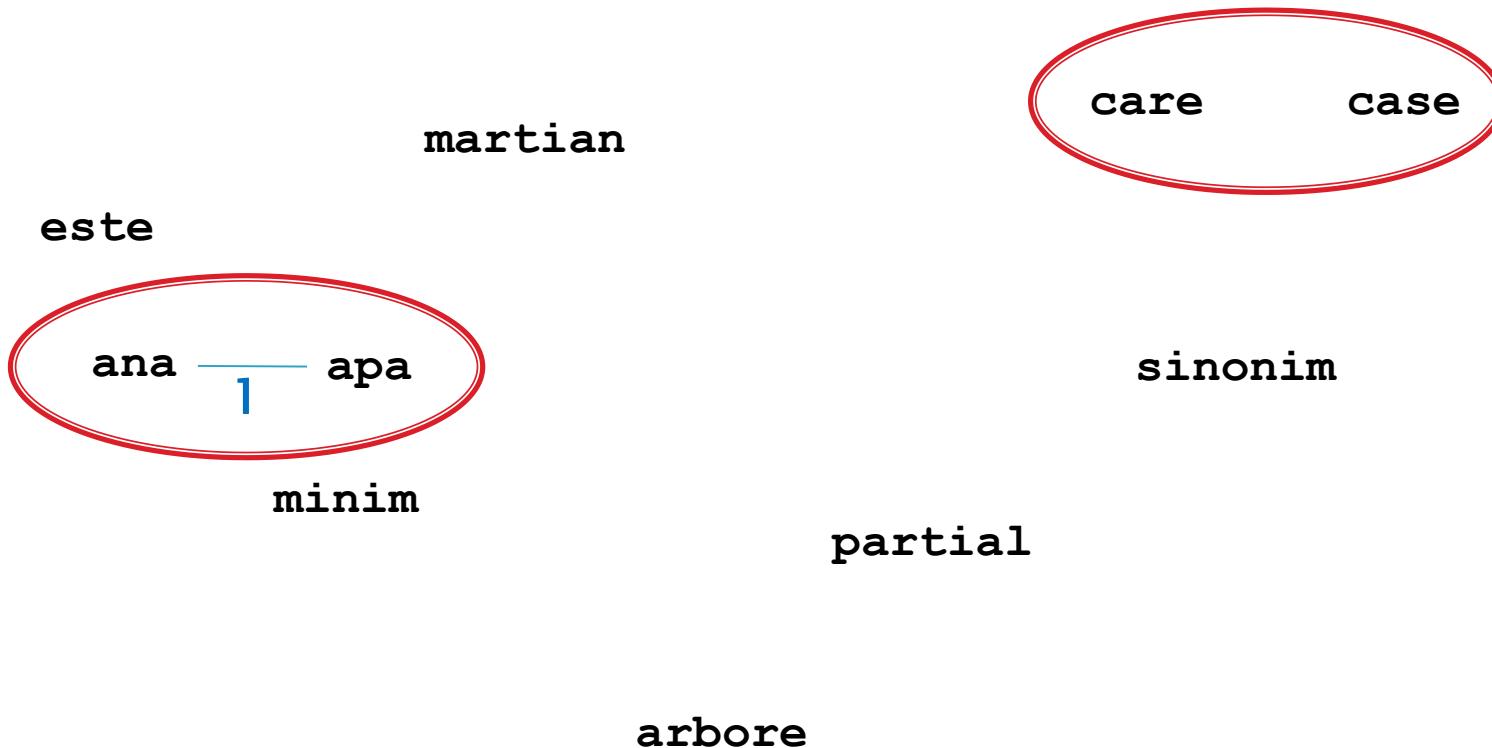
## K = 3 clusterer

# Aplicații – Clustering



K = 3 clustere

# Aplicații – Clustering



K = 3 clustere

# Aplicații – Clustering

este

ana apa

minim

arbore

care case

sinonim

partial — 2 — martian

K = 3 clustere

# Aplicații – Clustering

ana      apa

minim

este      care      case

3

sinonim

partial      martian

arbore

K = 3 clustere

# Aplicații – Clustering

ana      apa

minim

este      care      case

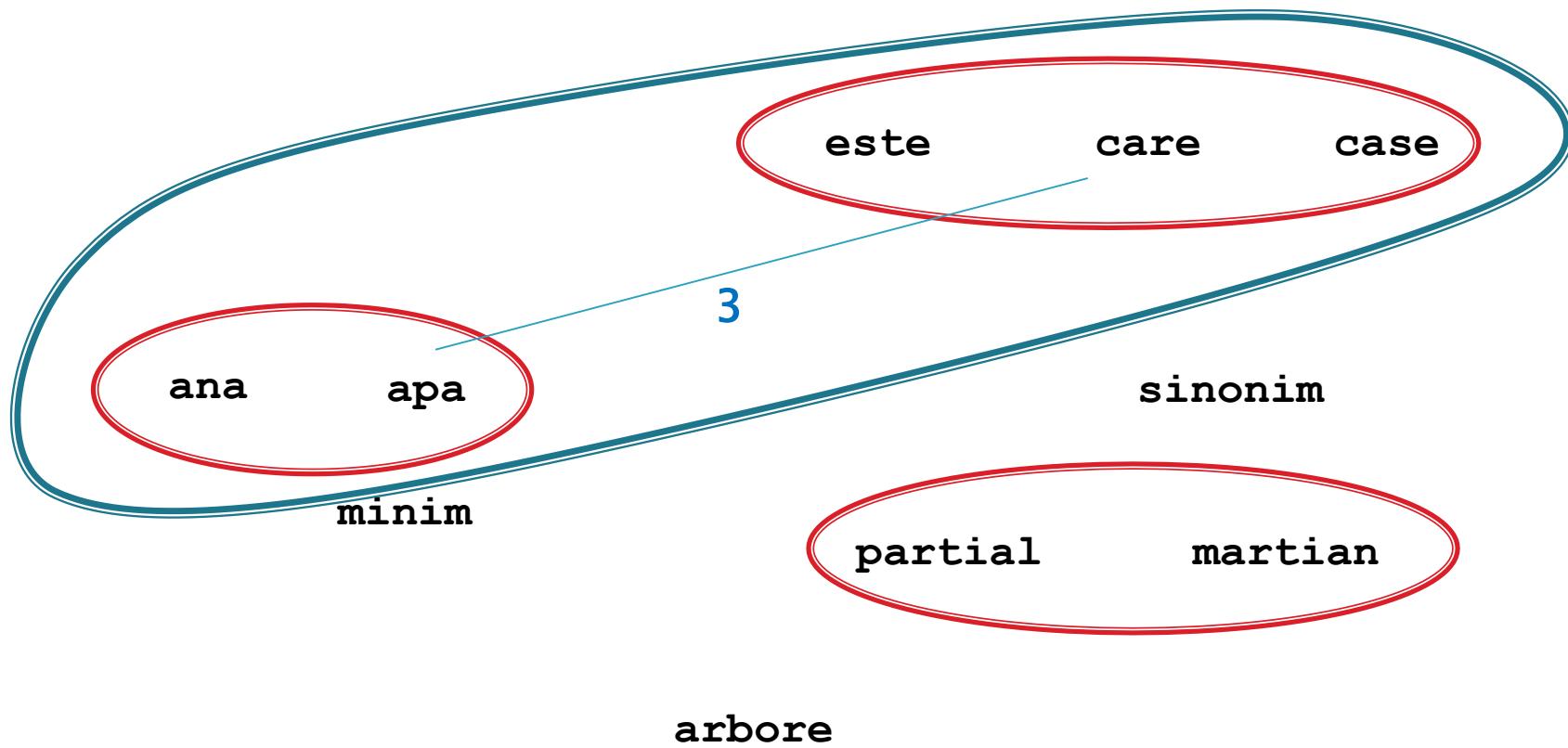
sinonim

partial      martian

arbore

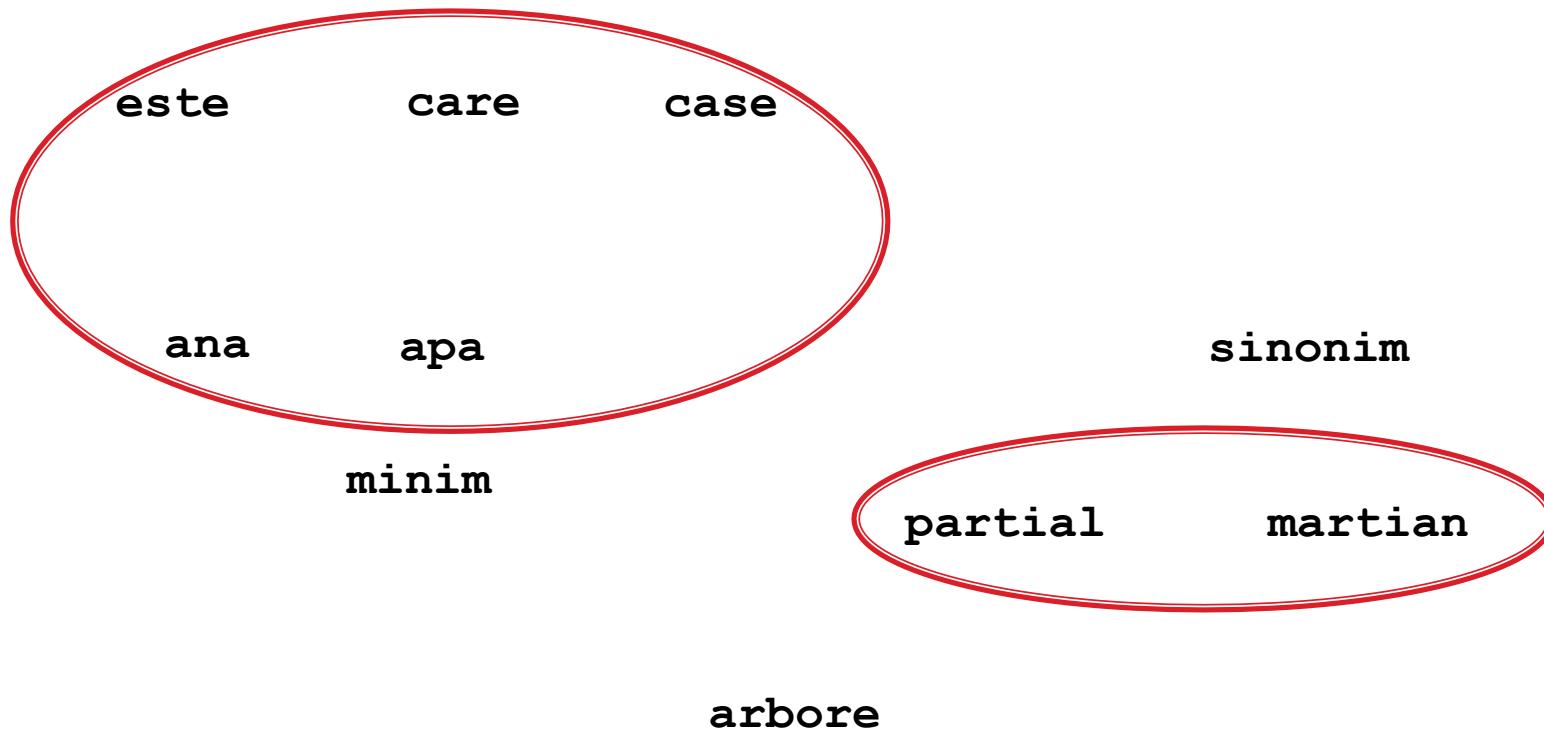
K = 3 clustere

# Aplicații – Clustering



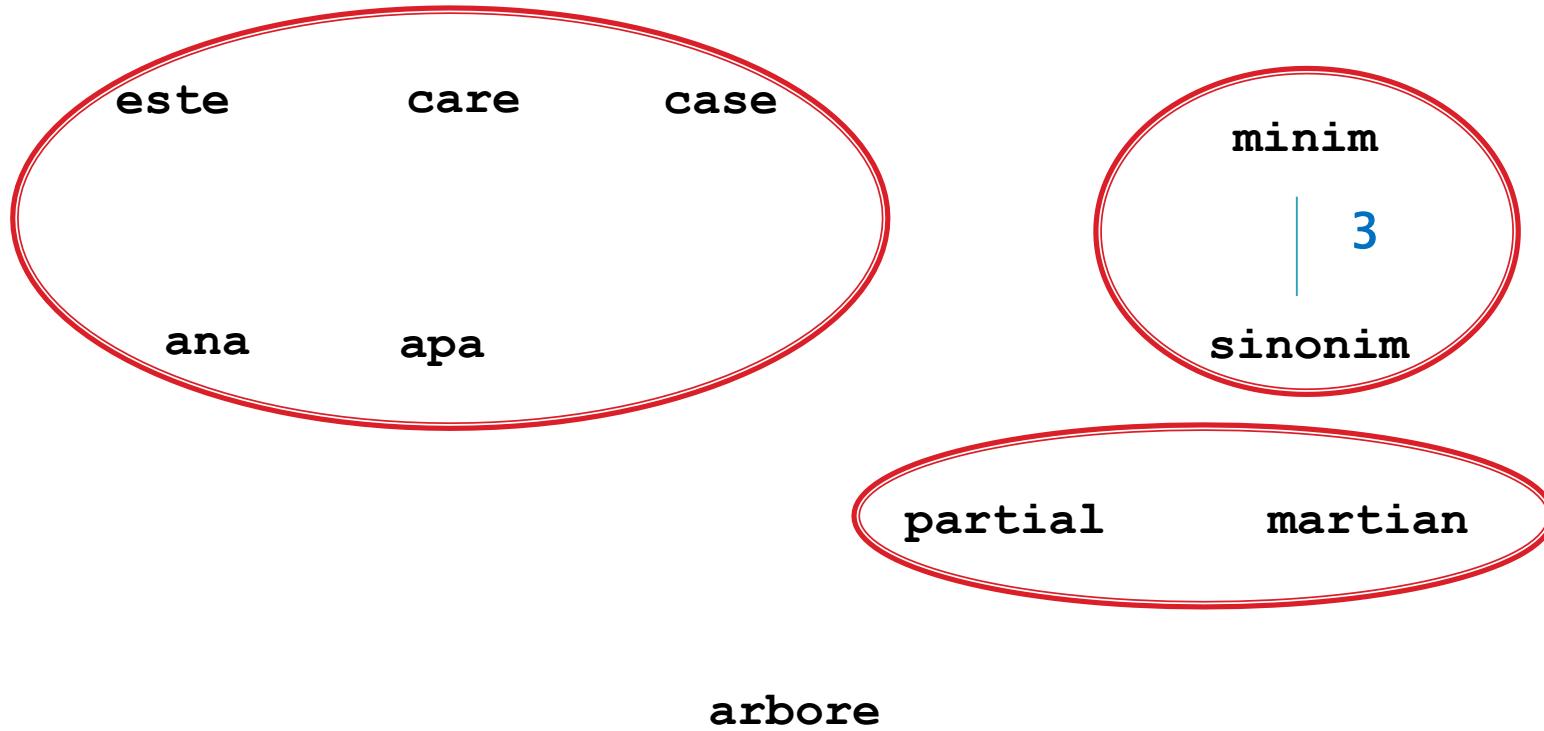
K = 3 clustere

# Aplicații – Clustering



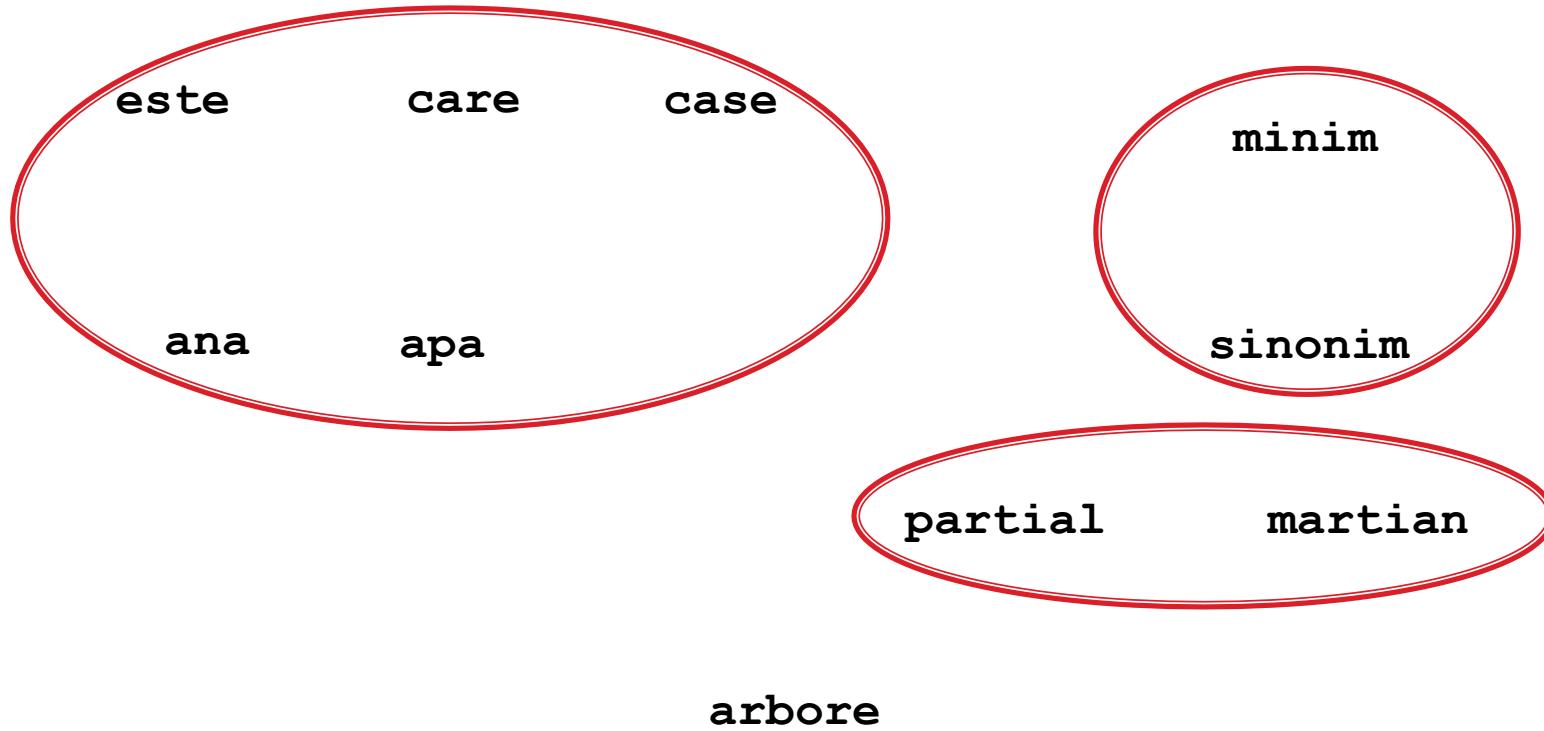
K = 3 clustere

# Aplicații – Clustering



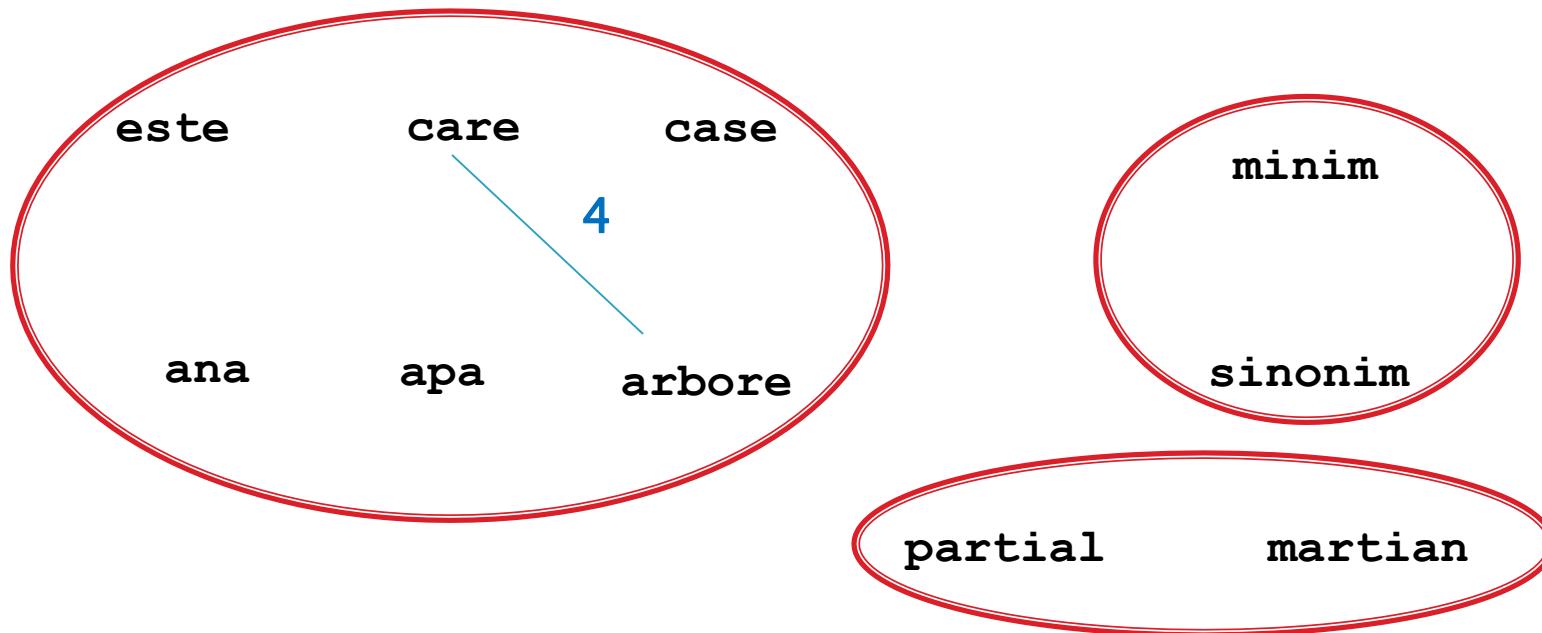
**K = 3 clustere**

# Aplicații – Clustering



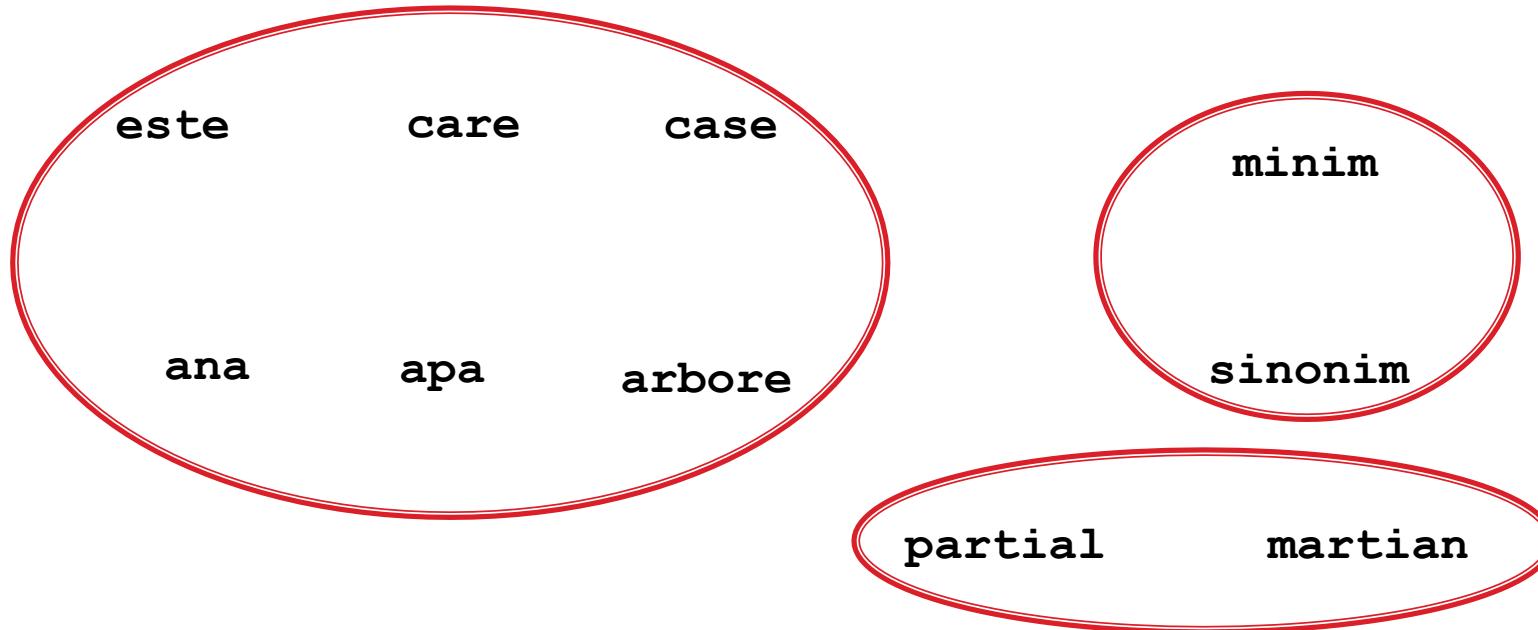
**K = 3 clustere**

# Aplicații – Clustering



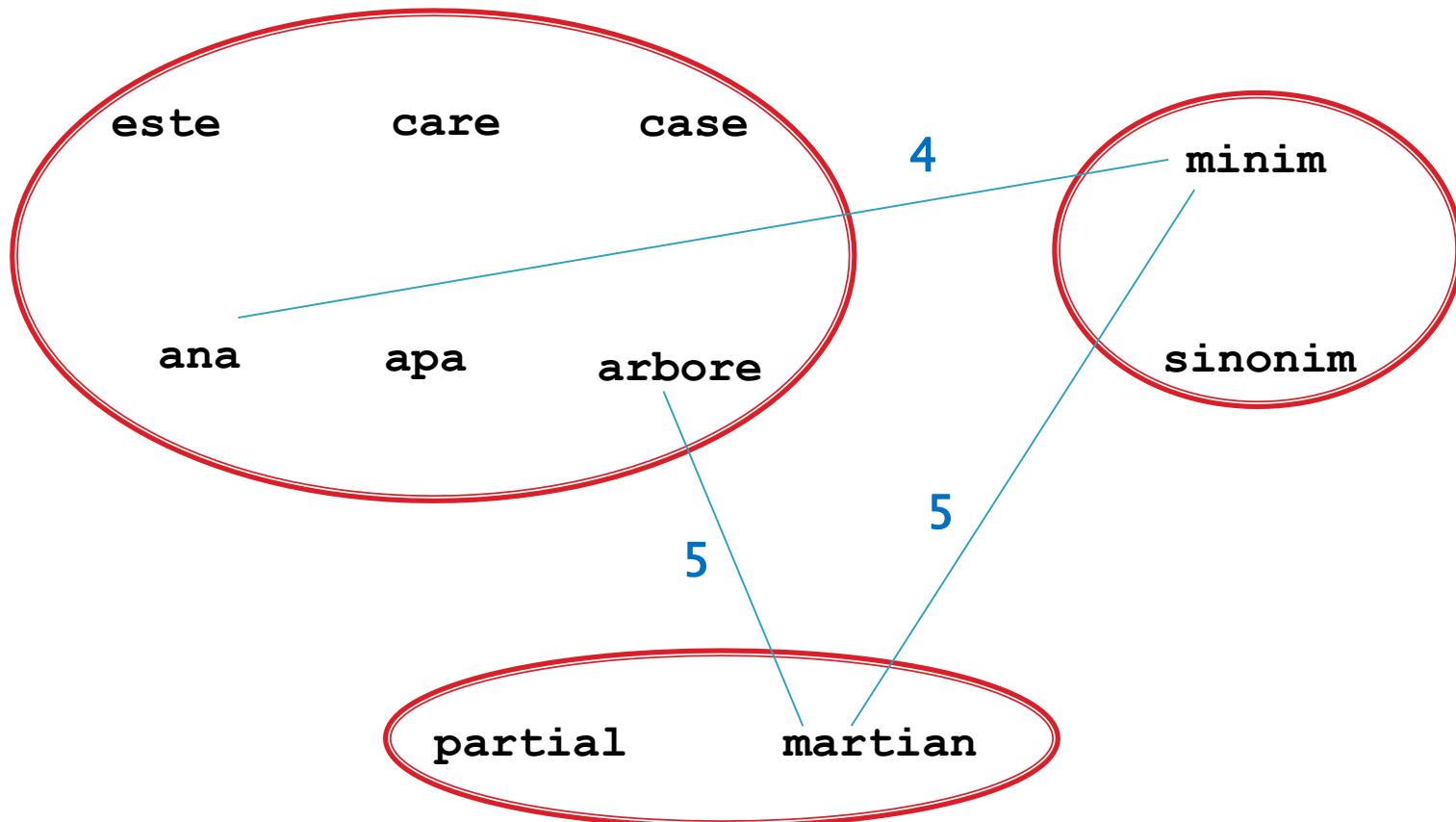
**K = 3 clustere**

# Aplicații – Clustering



Soluția cu  $k= 3$  clustere

# Aplicații – Clustering



Grad de separare =4

# Aplicații – 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  $o_r$ , clasa lui  $o_t$ )
- afișează cele k clase obținute

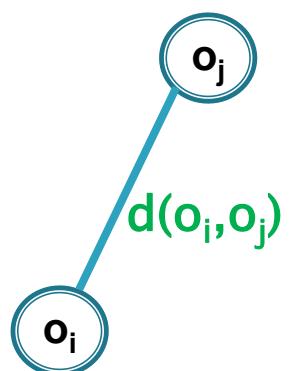
# Aplicații – Clustering

## 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  $o_r$ , clasa lui  $o_t$ )
- afișează cele  $k$  clase obținute



**Modelare cu graf ponderat (complet)**  
 $\Rightarrow$   $n - k$  pași din algoritmul lui Kruskal



# Aplicații – 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:

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

# Aplicații – 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 vârf formează o componentă conexă (clasă):  $T' = (V, \emptyset)$

# Aplicații – 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 vârf 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:  $E(T') = E(T') \cup \{uv\}$

# Aplicații – 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 vârf 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  $T'$

# Aplicații – Clustering

- ▶ Observație. Algoritmul este echivalent cu următorul, mai general
  - determină un apcm  $T$  al grafului complet  $G$
  - 
  - 
  -

# Aplicații – Clustering

- ▶ **Observație.** Algoritmul este echivalent cu următorul, mai general
  - determină un apcm  $T$  al grafului complet  $G$
  - consideră mulțimea  $\{e_{n-k+1}, \dots, e_{n-1}\}$  formată cu  $k-1$  muchii cu cele mai mari ponderi în  $T$
  - fie pădurea  $T' = T - \{e_{n-k+1}, \dots, e_{n-1}\}$
  -

# Aplicații – Clustering

- ▶ **Observație.** Algoritmul este echivalent cu următorul, mai general
  - determină un apcm  $T$  al grafului complet  $G$
  - consideră mulțimea  $\{e_{n-k+1}, \dots, e_{n-1}\}$  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

# Aplicații – Clustering

## Corectitudine – v. curs

- k-clusteringul obținut de algoritm are grad de separare maxim

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

[http://www.cs.princeton.edu/~wayne/kleinberg-  
tardos/pdf/04GreedyAlgorithmsII-2x2.pdf](http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsII-2x2.pdf)



# Arbore parțiali de cost minim



# **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 la arbore la unul neadăugat

## ► O primă formă a algoritmului

### Kruskal

- Inițial  $T = (V; \emptyset)$
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** 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

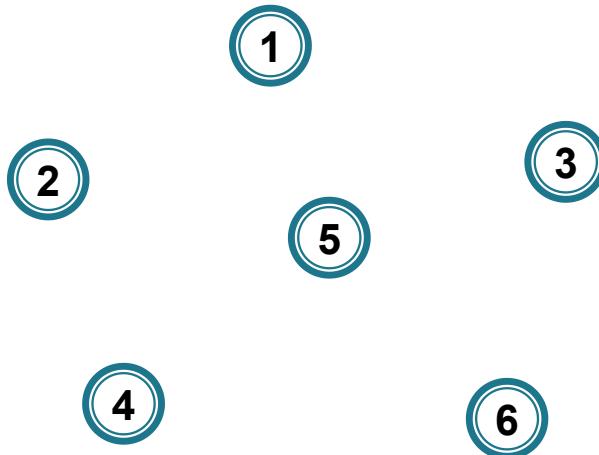
- Inițial  $T = (V; \emptyset)$
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** a.î.  $u, v$  sunt în **componente conexe diferite** ( $T+uv$  aciclic)
  - $E(T) = E(T) \cup uv$

### Prim

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

## Kruskal

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



- Se încearcă unirea acestor componente prin muchii de cost minim

## Prim

- Inițial: se pornește de la un vârf de start



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

## Kruskal

- La un pas:

Muchiile selectate formează o pădure

## Prim

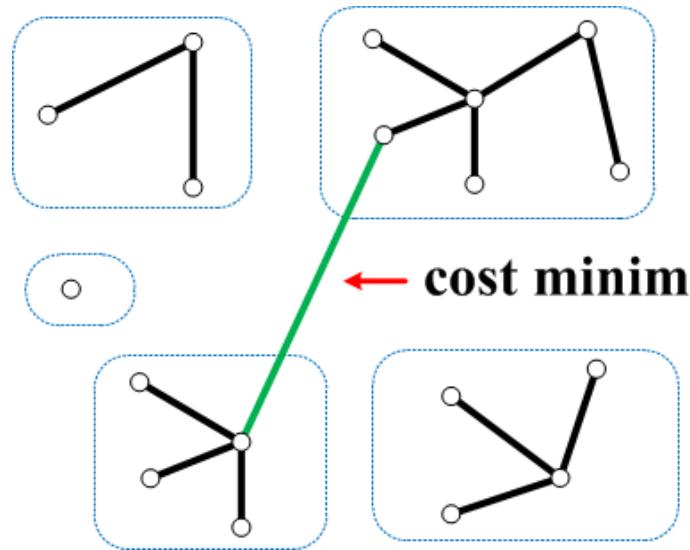
- La un pas:

Muchiile selectate formează un arbore

# Kruskal

- La un pas:

Muchiile selectate formează o pădure

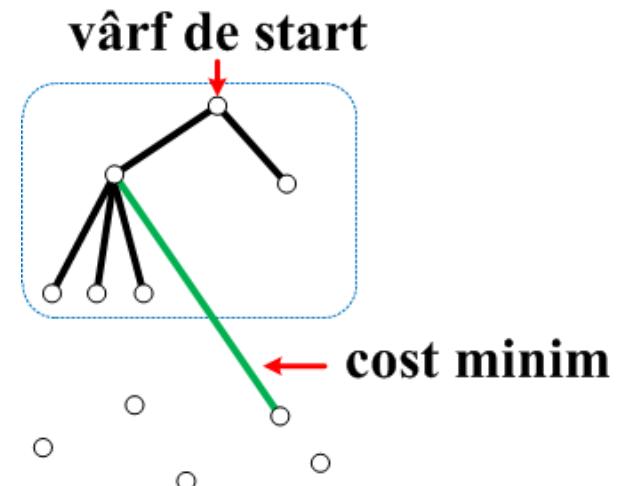


Este selectată o muchie de cost minim care unește doi arbori din pădurea curentă (două componente conexe)

# Prim

- La un pas:

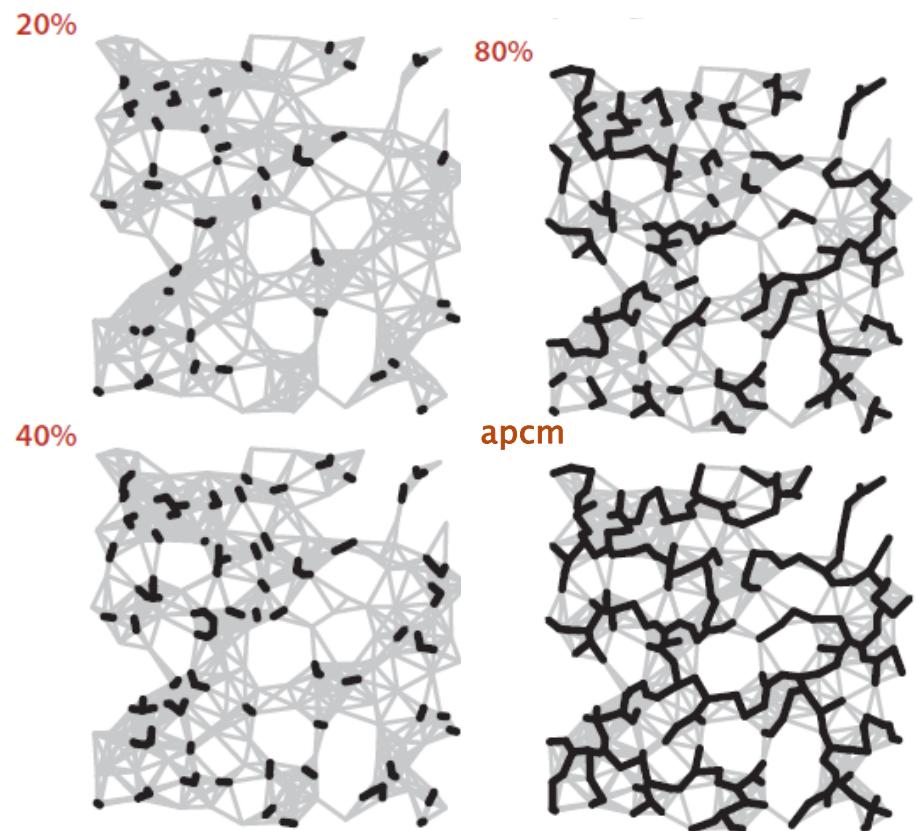
Muchiile selectate formează un arbore



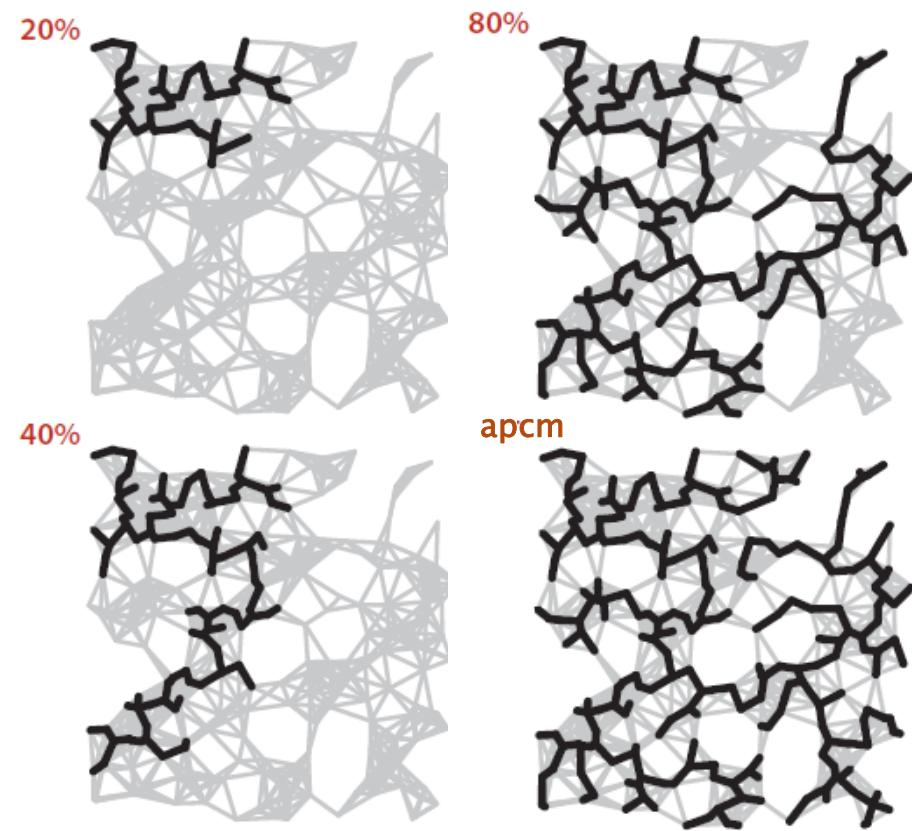
Este selectată o muchie de cost minim care unește un vârf din arbore cu unul care nu este în arbore(neselectat)

# Arbore parțiali de cost minim

Kruskal

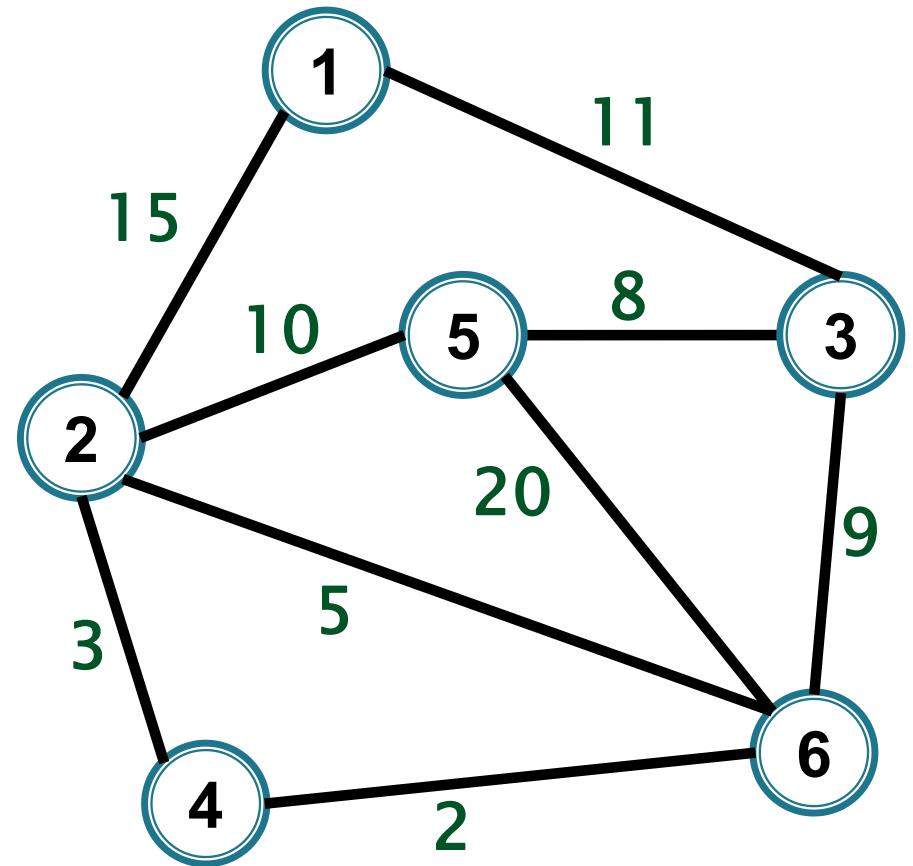


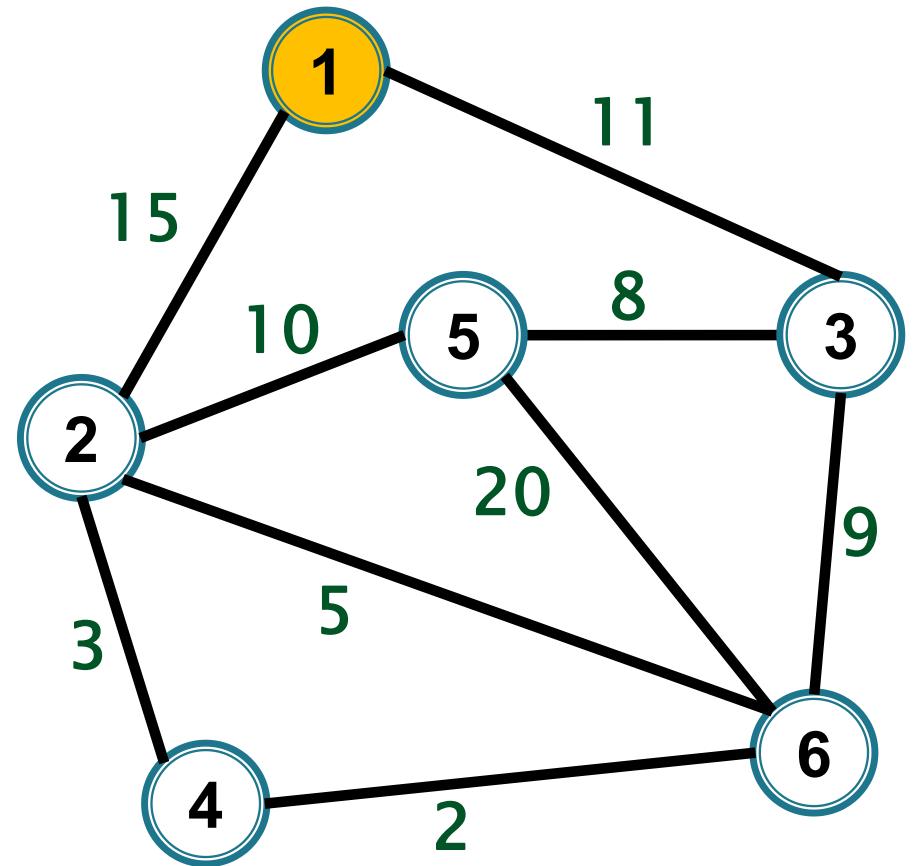
Prim



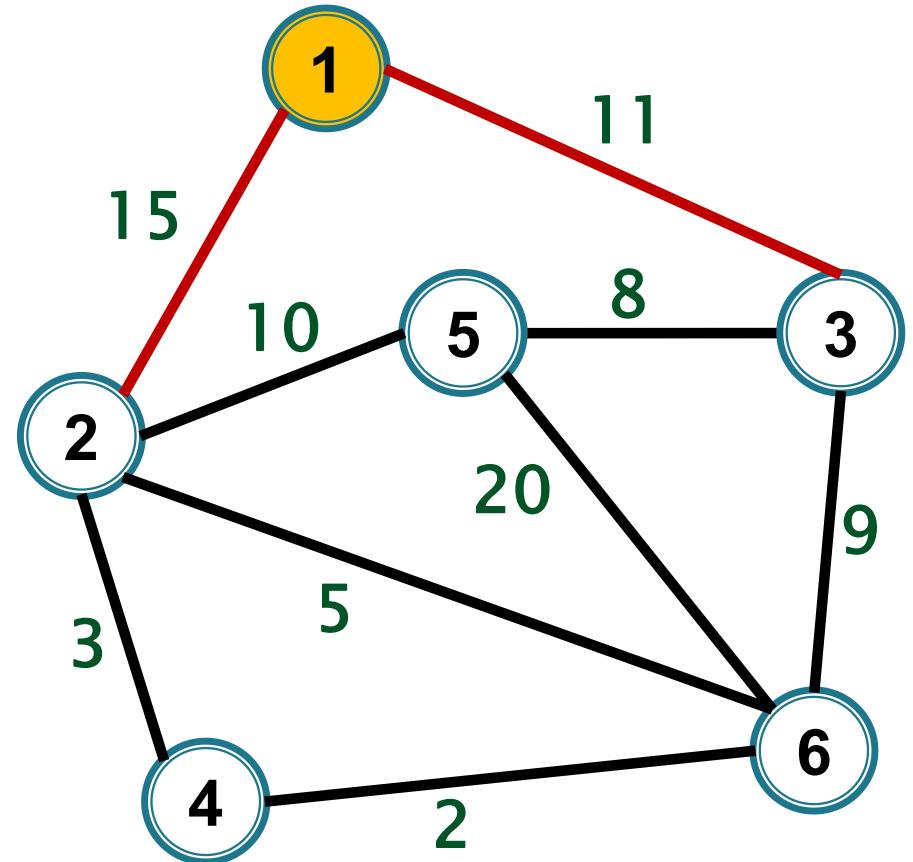
Imagine din

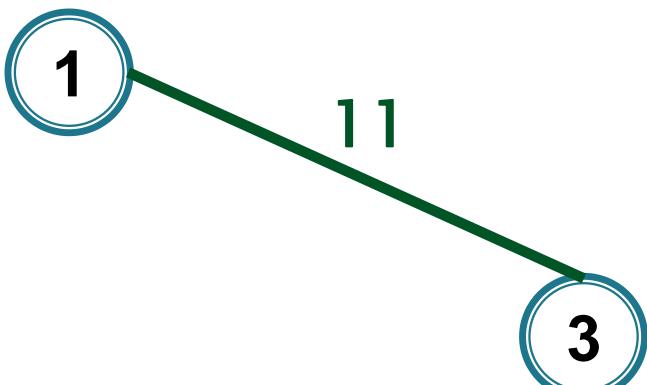
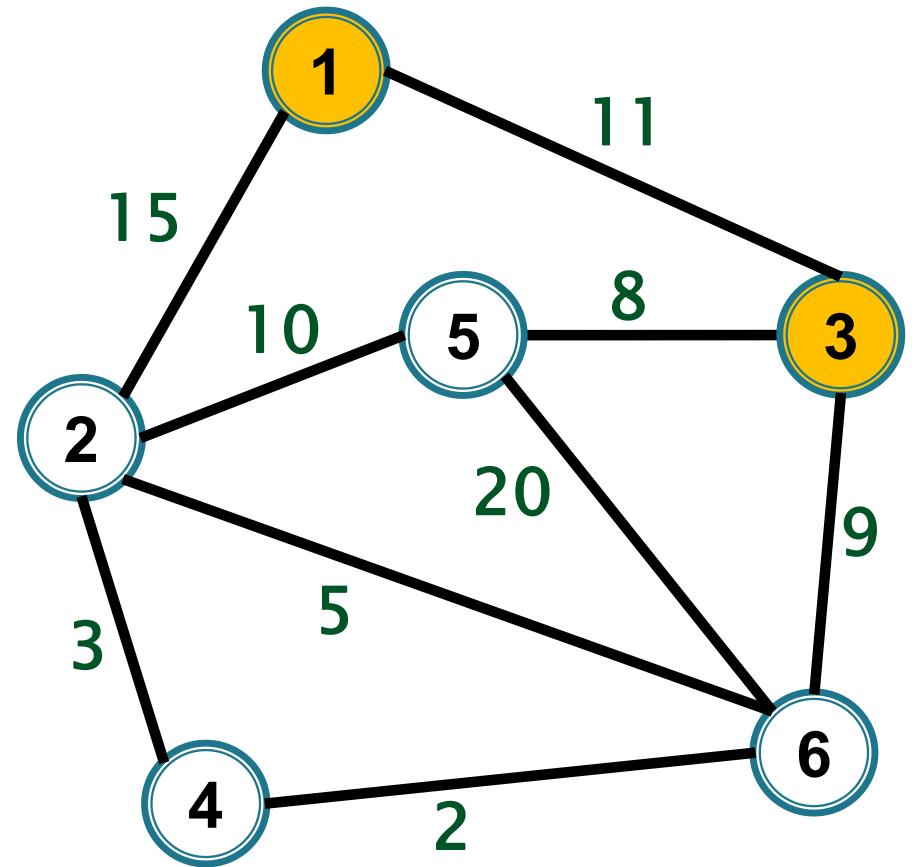
R. Sedgewick, K. Wayne – Algorithms, 4th edition, Pearson Education, 2011

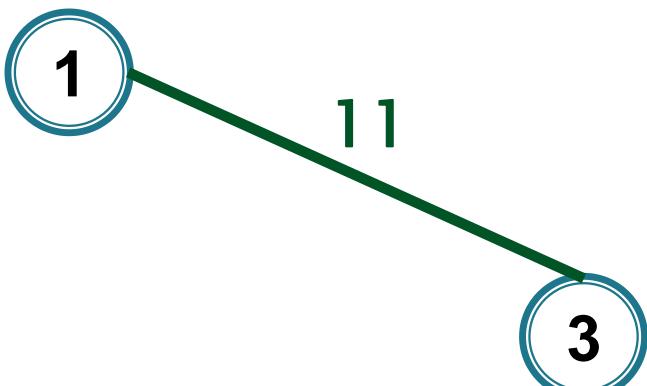
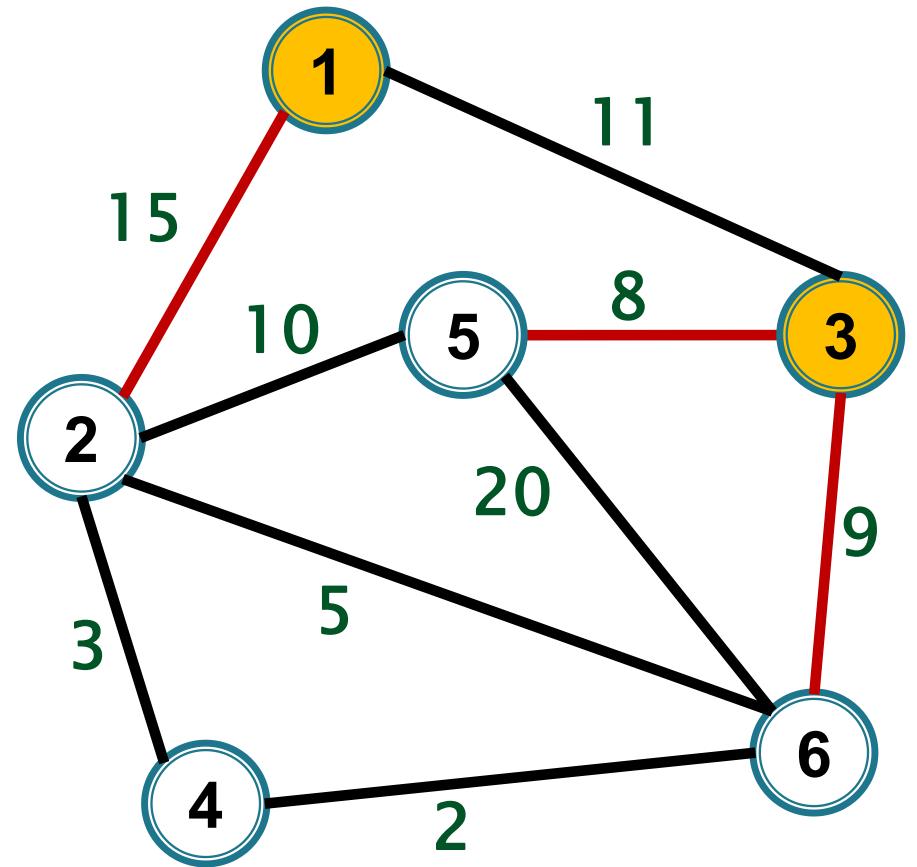


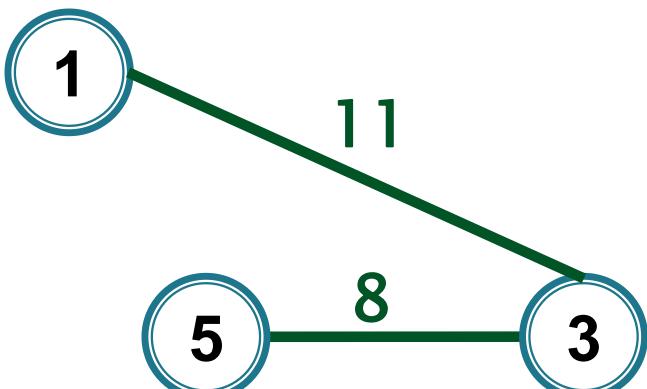
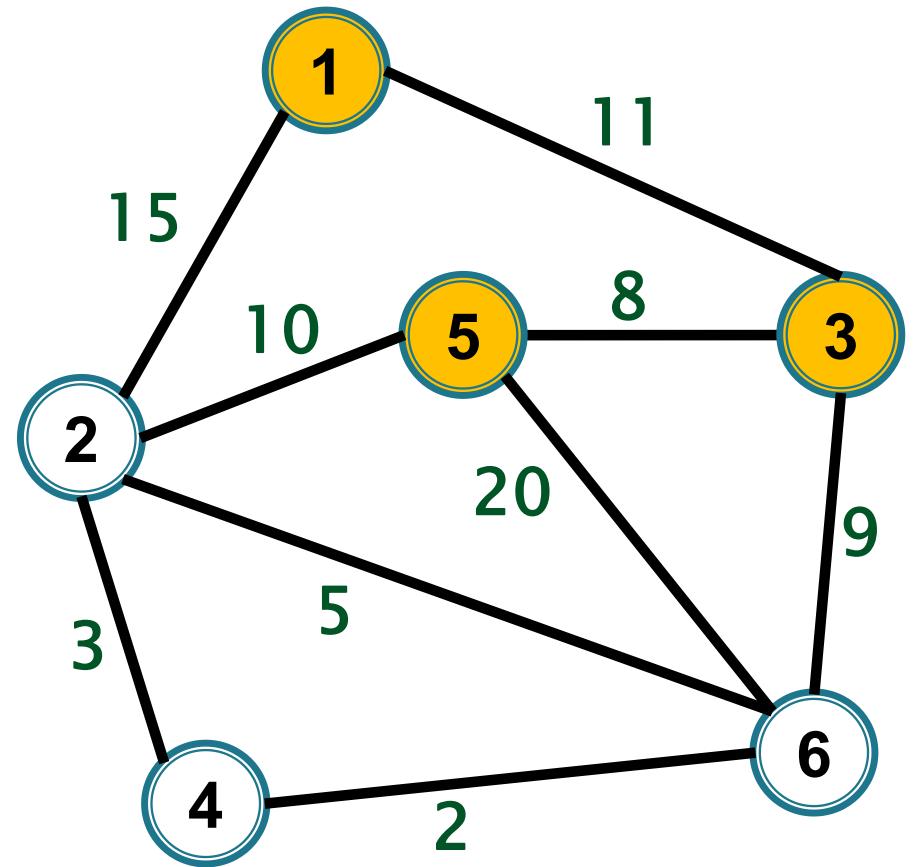


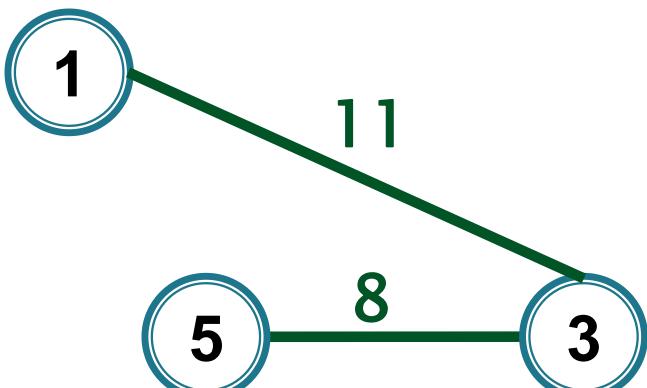
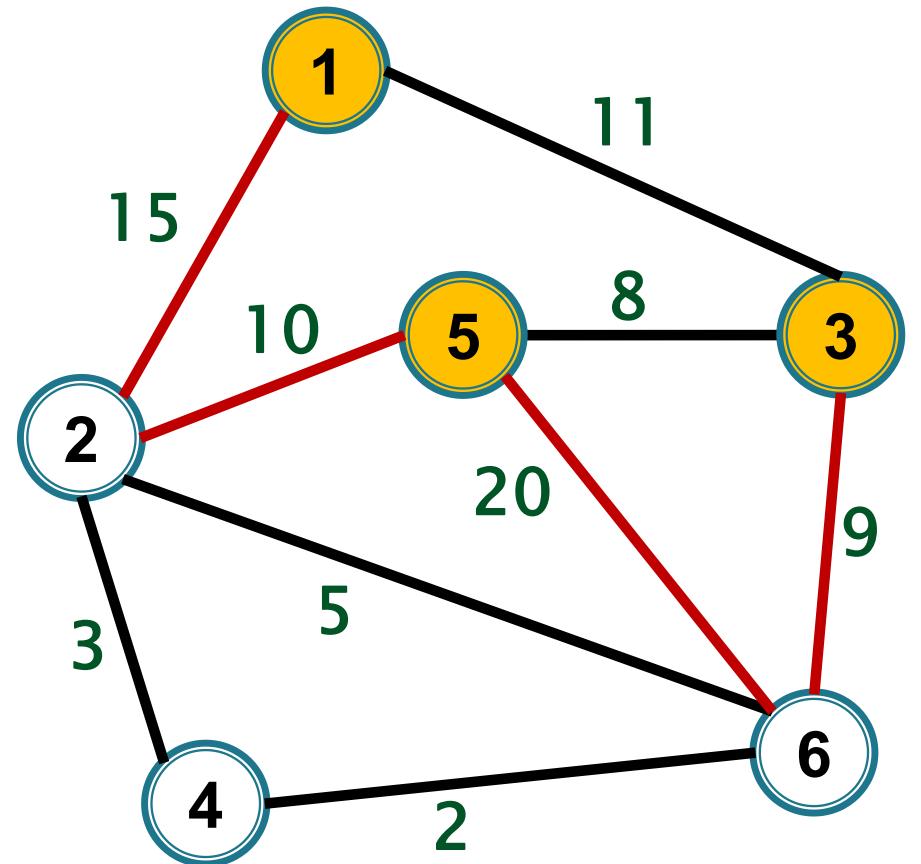
$$s = \textcircled{1}$$

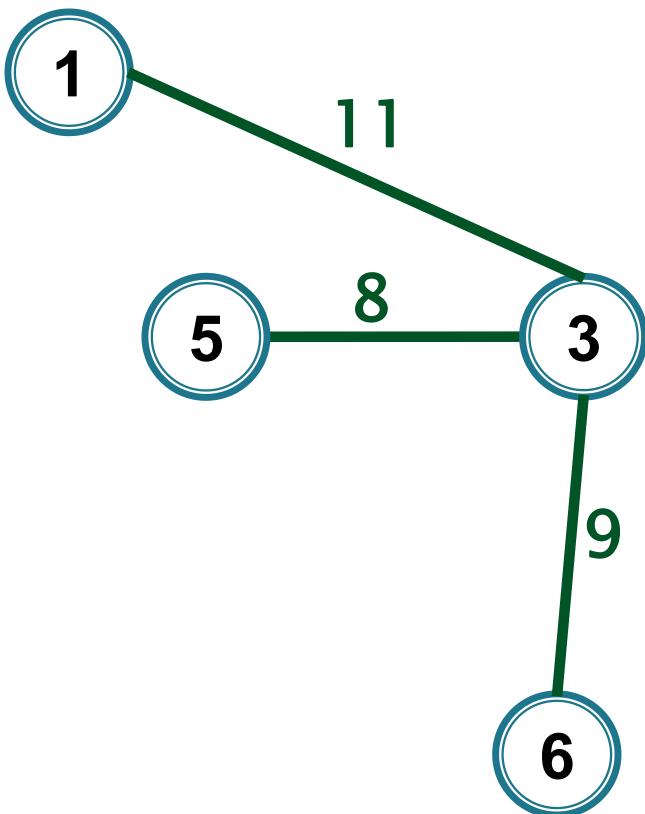
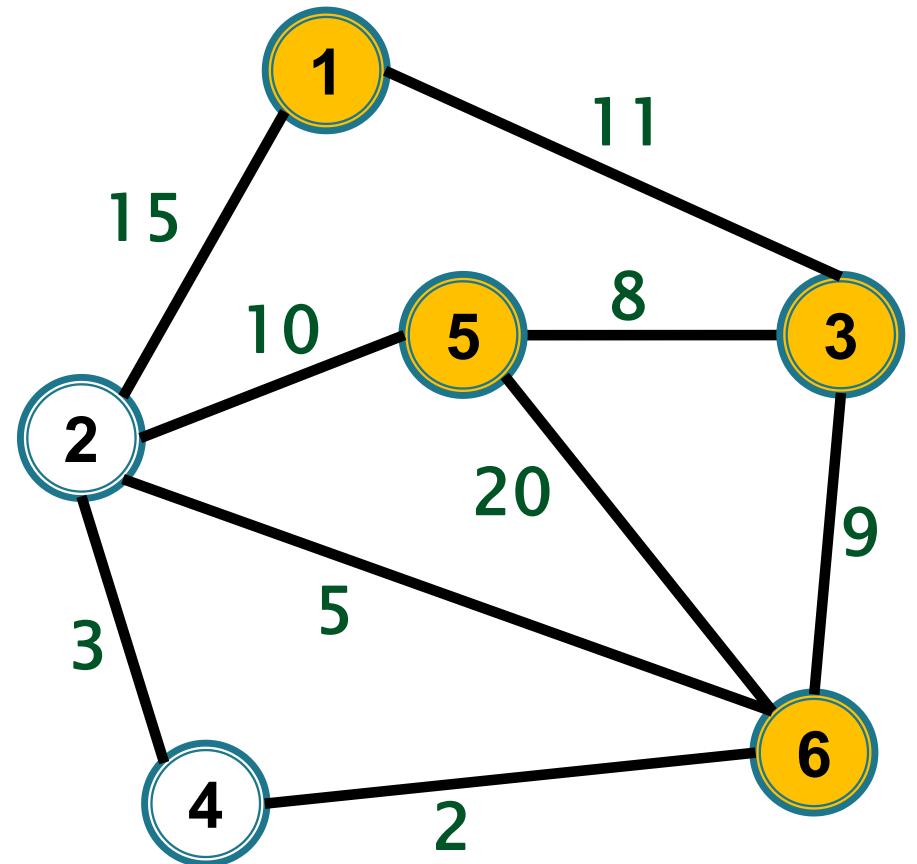


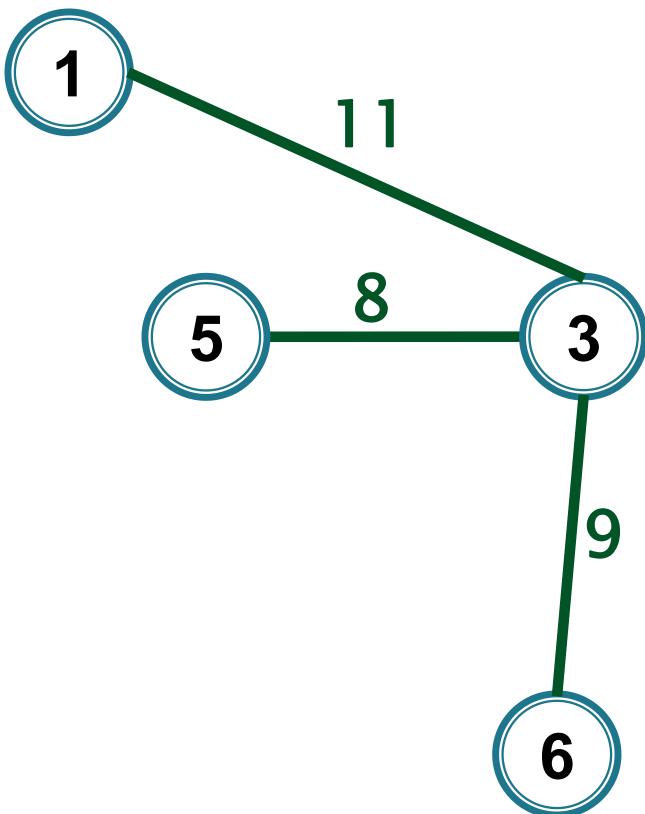
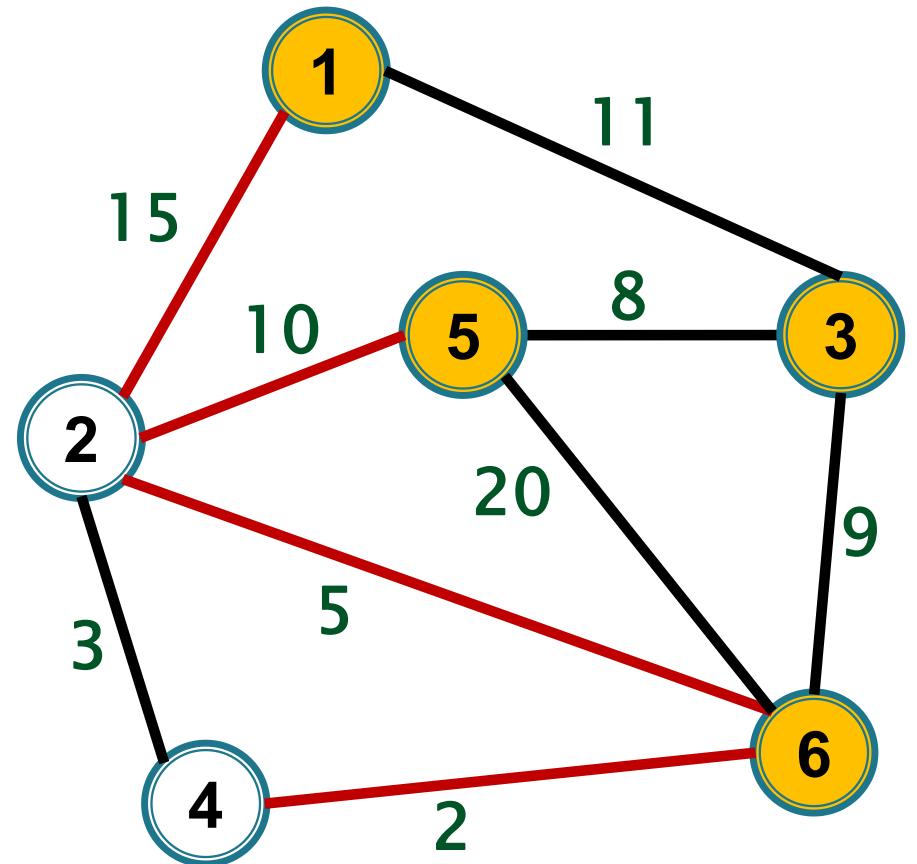


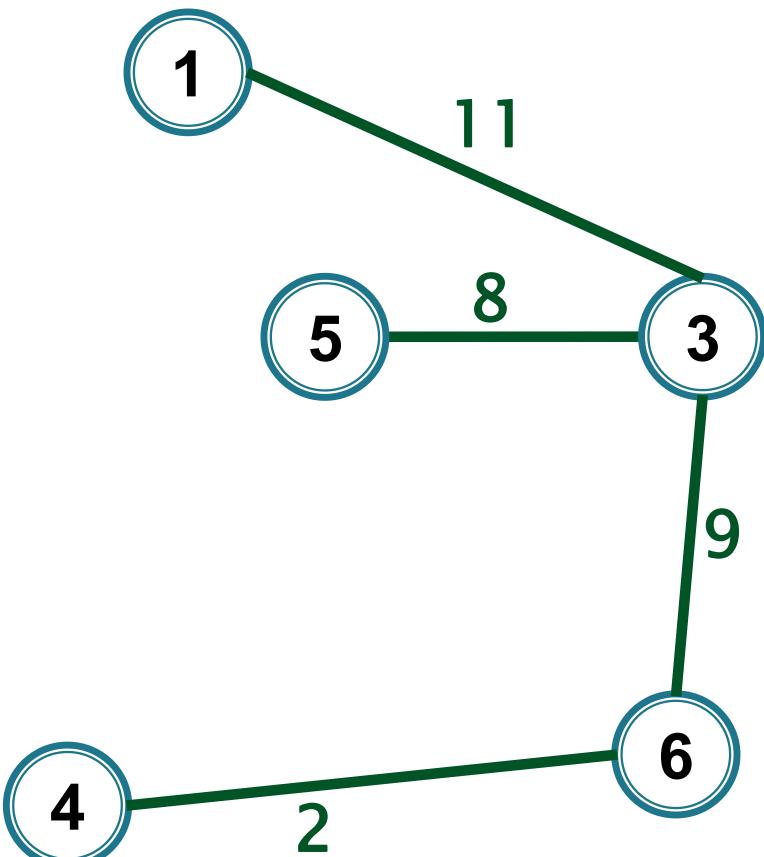
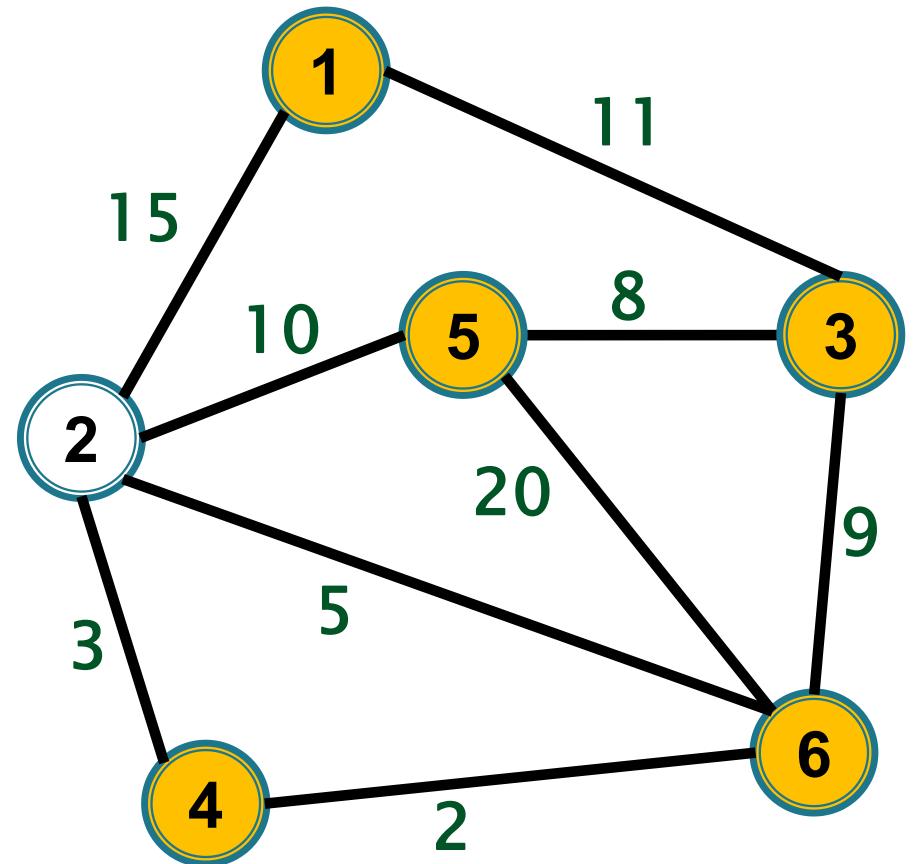


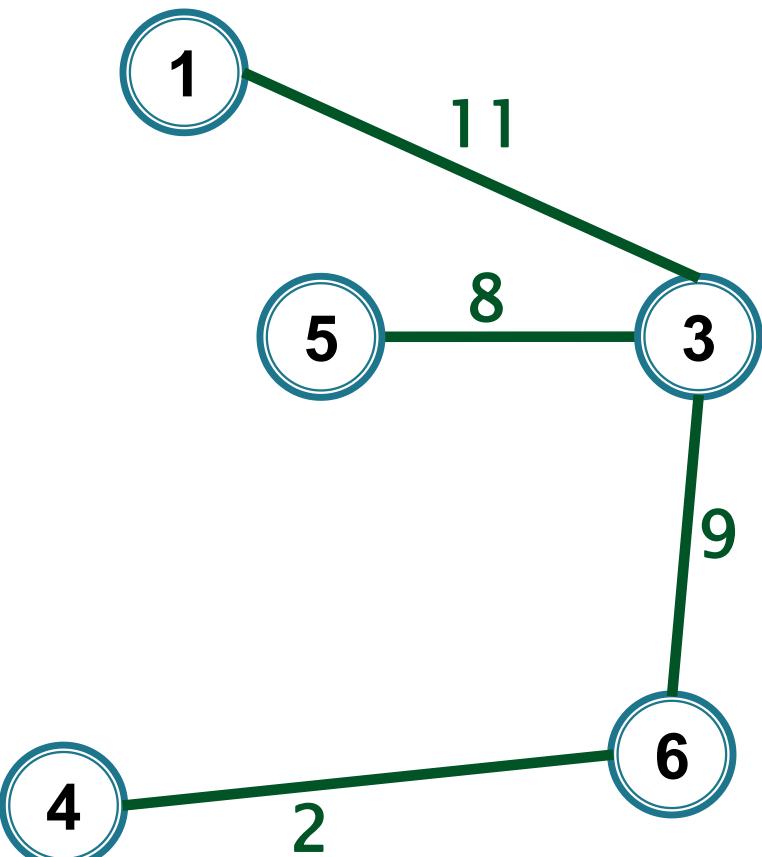
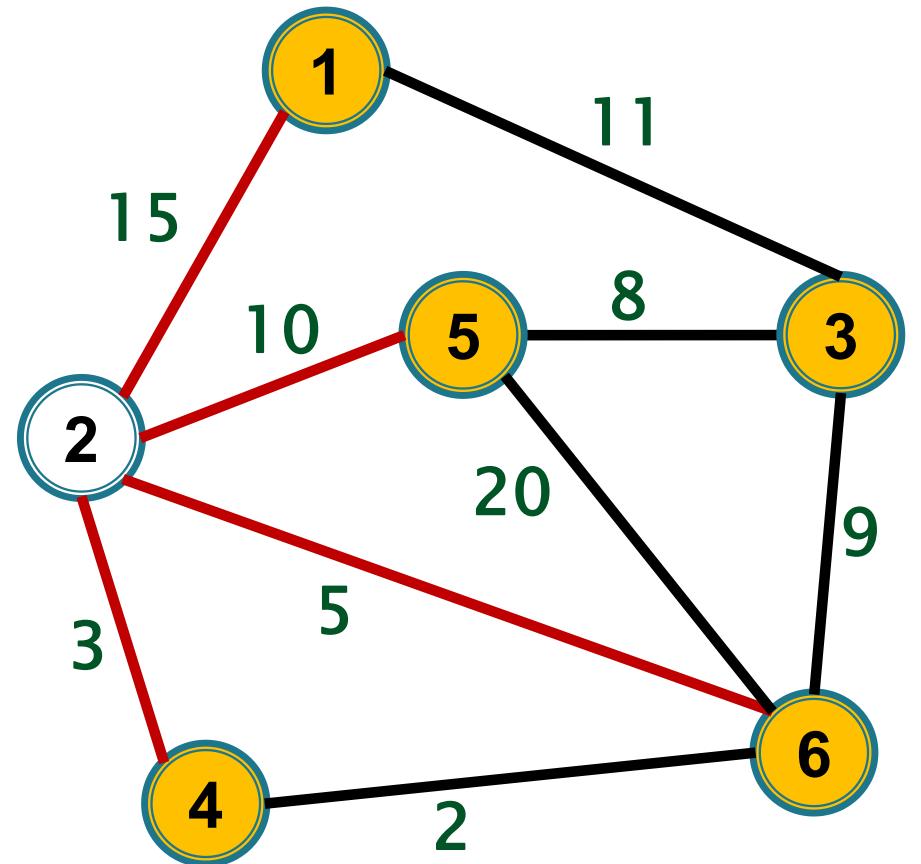


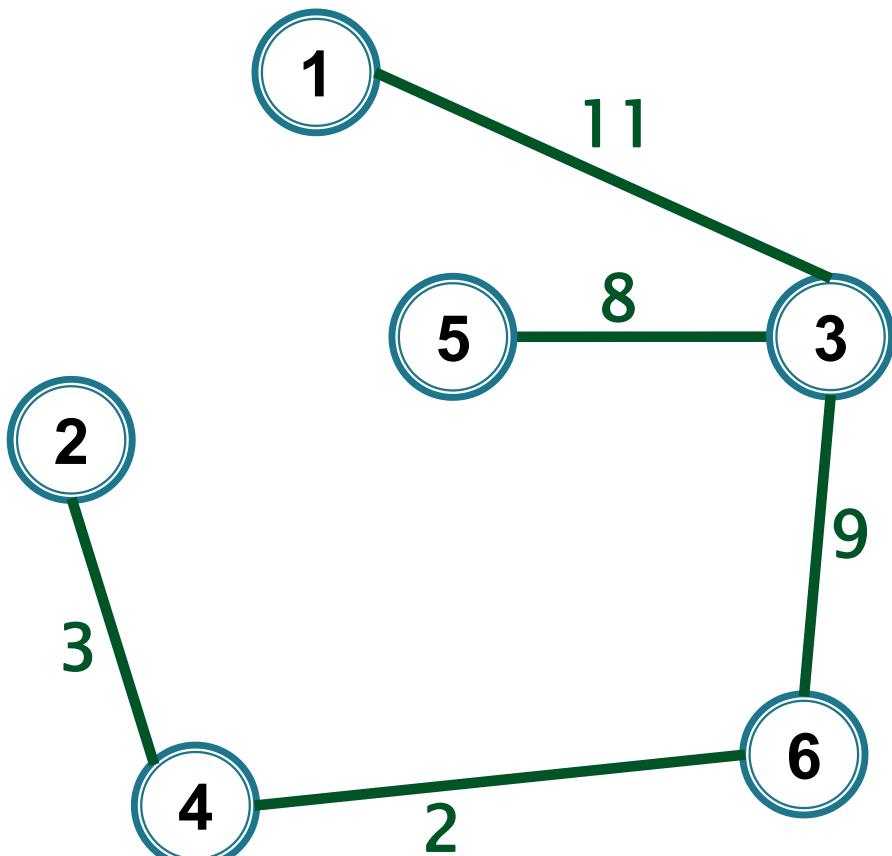
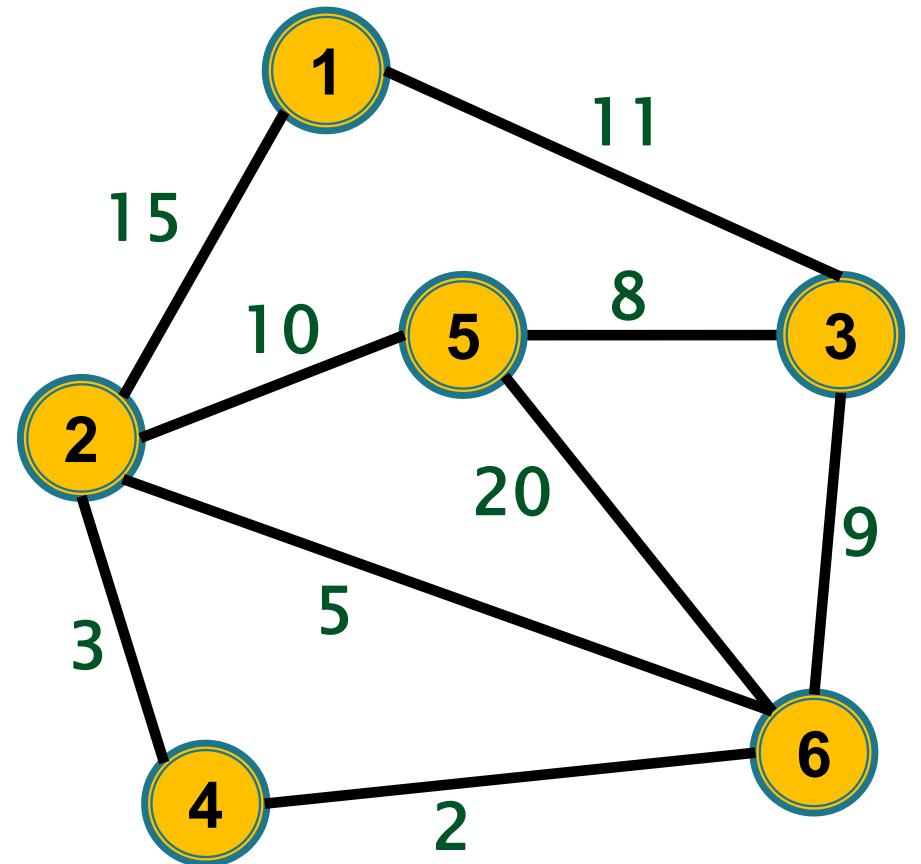


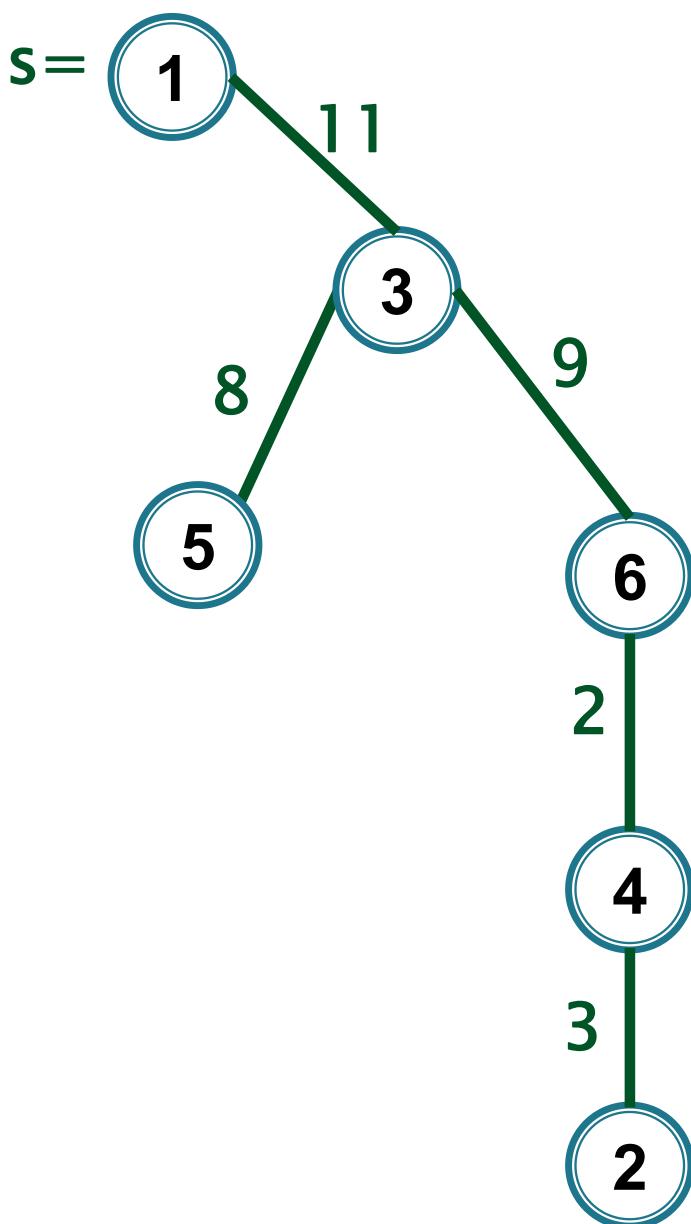
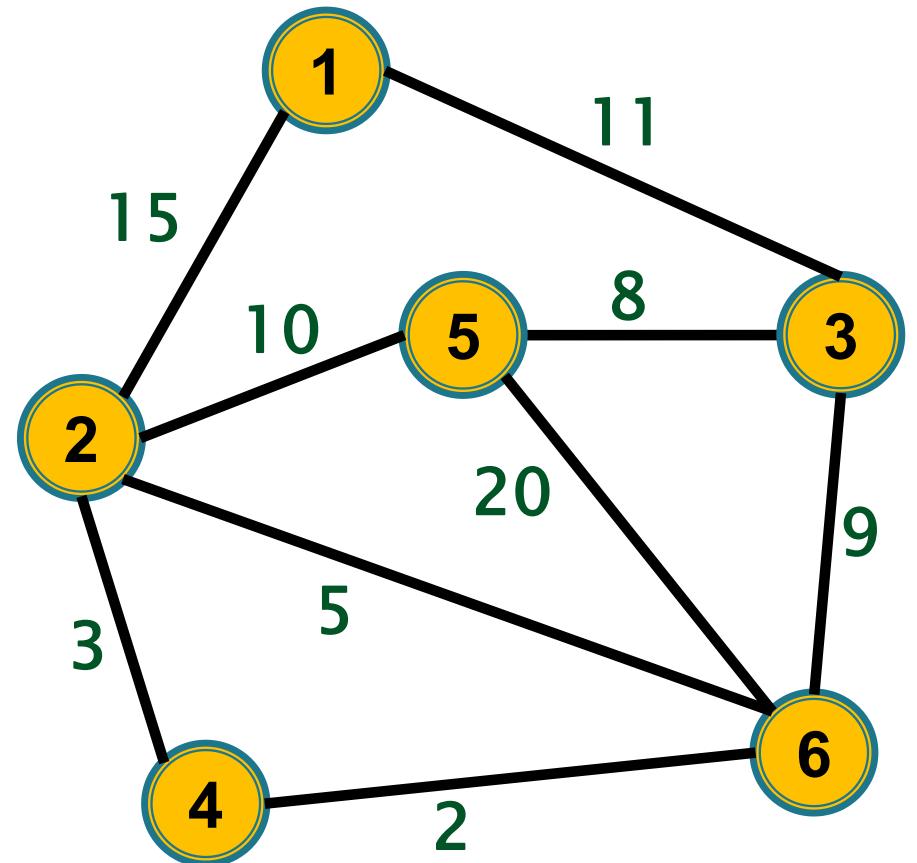












# Implementare+Complexitate



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

# Implementare+Complexitate



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

# Implementare+Complexitate



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

$O(nm)$



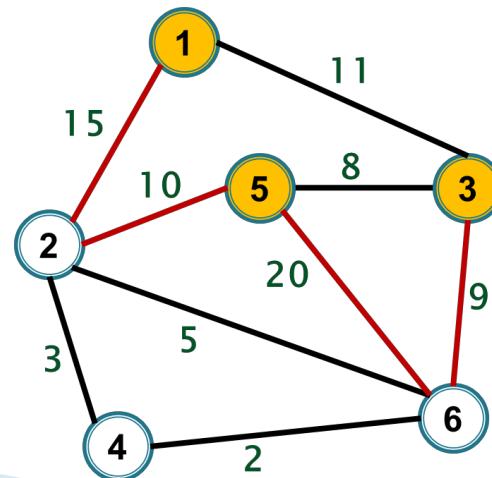
# Implementare Prim



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

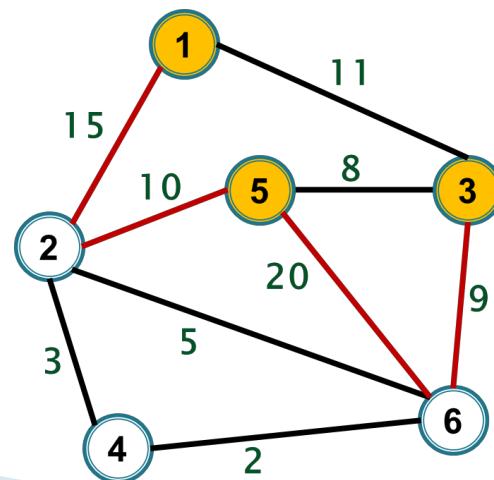


# Implementare Prim

Cum evităm să comparam de fiecare dată toate muchiile cu o extremitate în arbore și cealaltă nu.



Pentru un 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)

# Implementare+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. si laborator+seminar + alg. Dijkstra)

# **Detalii implementare**

## **Algoritmul lui Prim**

# Implementare Prim

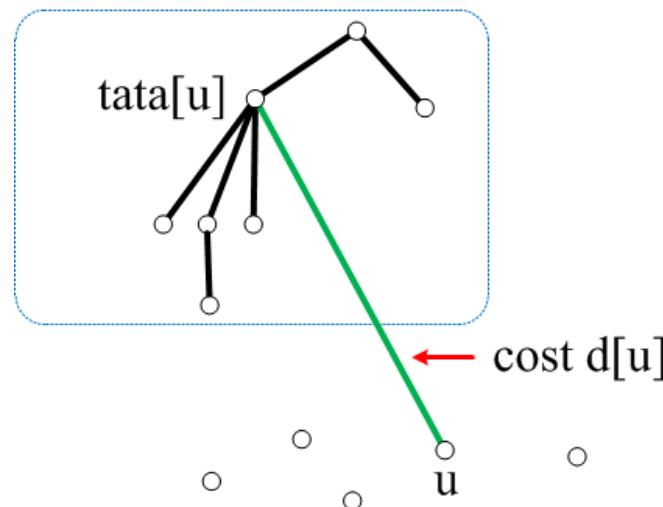
Asociem fiecărui vârf u următoarele informații (etichete) – pentru a reține **muchia de cost minim care îl unește de un vârf selectat deja în arbore**:



# Implementare Prim

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

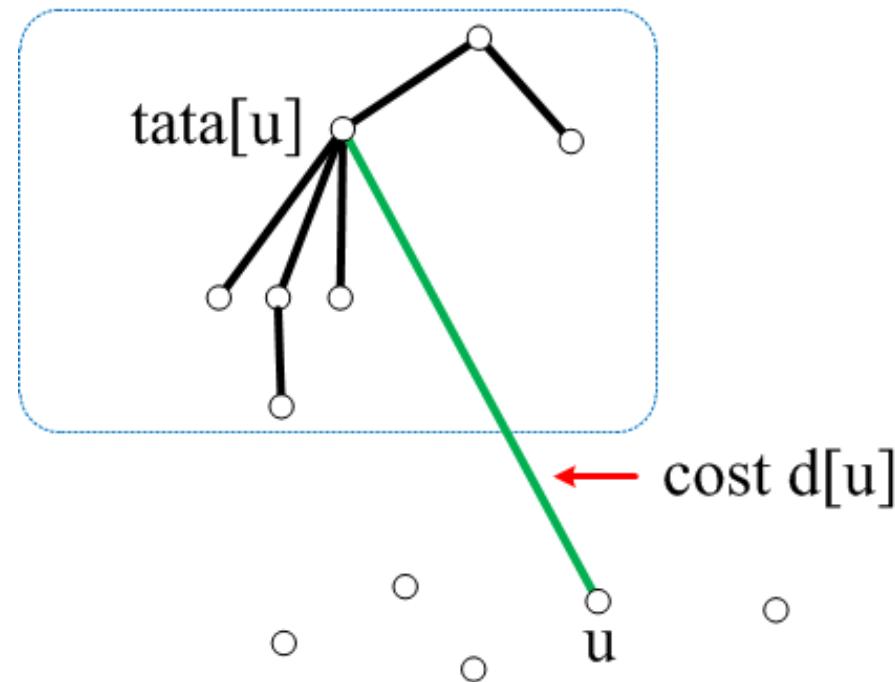
- ▶  $d[u]$  = costul minim al unei muchii de la  $u$  la un vârf selectat deja în arbore
- ▶  $tata[u]$  = acest vârf din arbore pentru care se realizează minimul



# Implementare Prim

## ► Avem

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



# Implementare Prim

Atunci 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  $(\text{tata}[u], u)$ 
    - aceasta este muchia de cost minim care unește un vârf neselectat de un vârf din arbore

# Implementare Prim

Atunci algoritmul se modifică astfel:

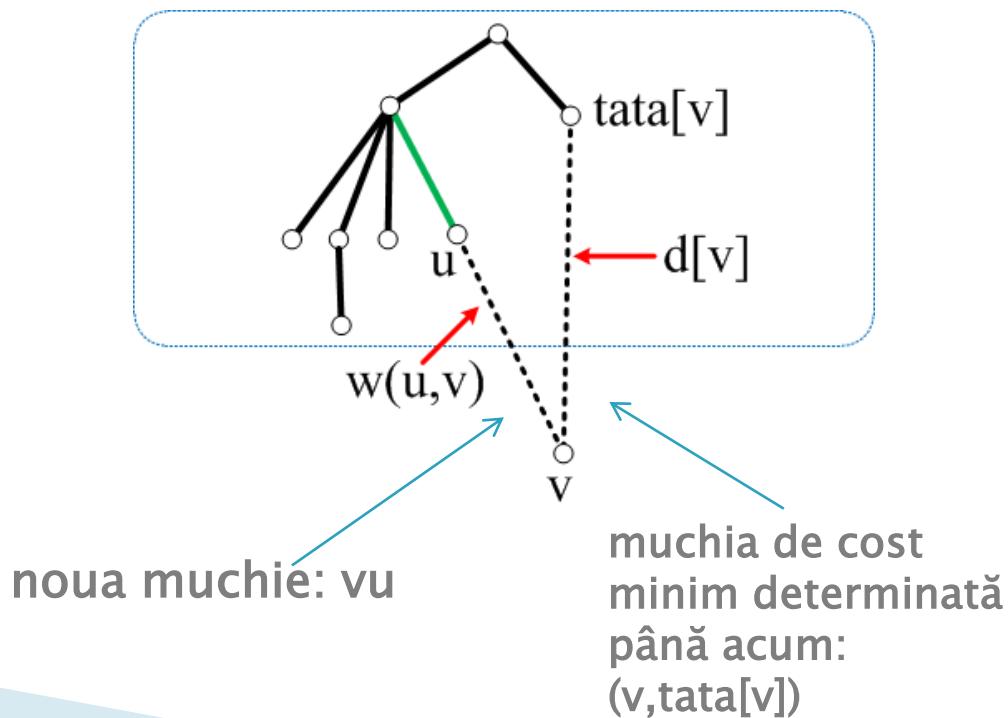
► La un pas

- se alege un vârf  $u$  cu eticheta  $d$  minimă care nu este încă în arbore și se adaugă la arbore muchia  $(\text{tata}[u], u)$
- se actualizează etichetele vârfurilor  $v \notin V(T)$  vecine cu  $u$  astfel:

dacă  $w(u, v) < d[v]$  atunci

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

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



# Implementare Prim

- ▶ Muchiile arborelui vor fi în final  
 $(u, \text{tata}[u]), u \neq s$

# Prim

Notăm  $Q = V(G) - V(T)$  = multimea vîrfurilor neselectate încă în arbore

## ► Prim

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

## ► Prim

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

## ► Prim

- s - vârful de start
- initializează  $Q$  cu  $V$
- pentru fiecare  $u \in V$  executa
  - $d[u] = \infty$ ;  $tata[u] = 0$
  - $d[s] = 0$
- cat timp  $Q \neq \emptyset$  executa
  - extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă

## ▶ Prim

- $s$  - vârful de start
- initializează  $Q$  cu  $V$
- pentru fiecare  $u \in V$  execută  
 $d[u] = \infty$ ;  $tata[u] = 0$   
 $d[s] = 0$
- cat timp  $Q \neq \emptyset$  execută  
  extrage un vârf  $u \in Q$  cu eticheta  $d[u]$  minimă  
  pentru fiecare  $uv \in E$  execută

## ▶ Prim

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

## ▶ Prim

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

# Prim

## Complexitate

- ▶ Inițializări →
- ▶  $n * \text{extragere vârf minim}$  →
- ▶ actualizare etichete vecini →

# Prim



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

# Prim



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

- vector?
- heap?

# Prim

## Varianta 1 – Folosim vector de vizitat

$Q[u] = 1$ , dacă  $u \notin Q$

0, altfel

# Prim

## Complexitate

### Varianță 1 - cu vector de vizitat

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

# Prim

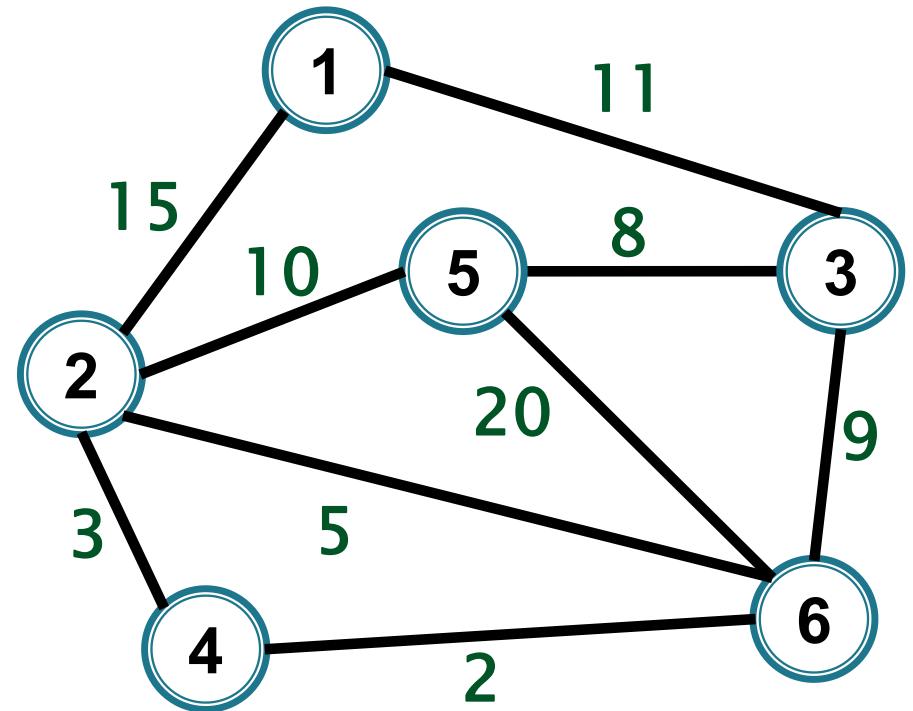
# Complexitate

## **Varianta 1 – cu vector de vizitat**

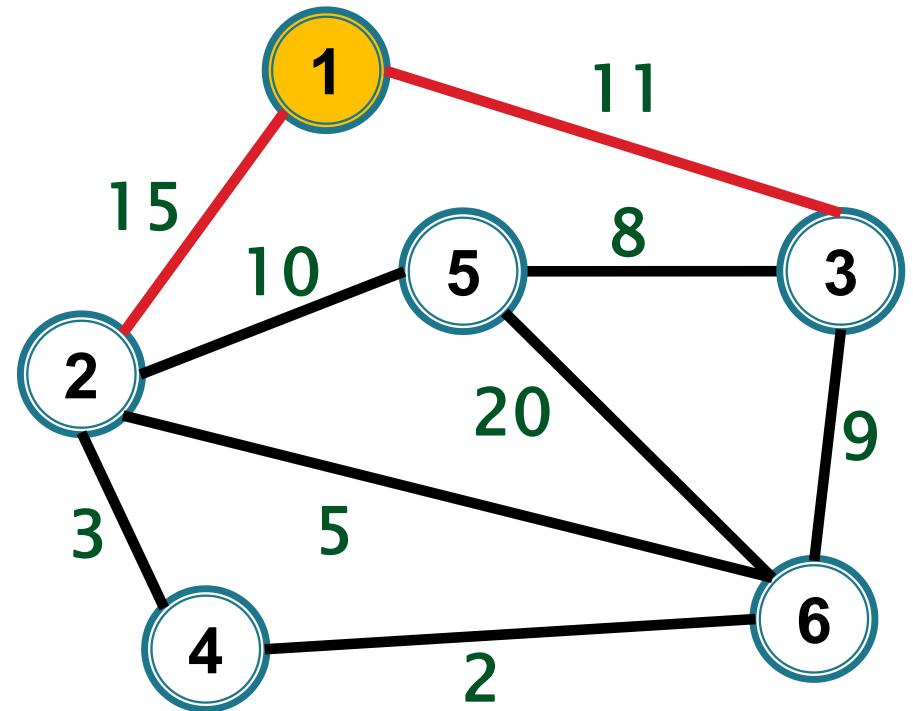
- ▶ Inițializări  $\rightarrow O(n)$
  - ▶  $n * \text{extragere vârf minim}$   $\rightarrow O(n^2)$
  - ▶ actualizare etichete vecini  $\rightarrow O(m)$

---

$O(n^2)$



$d/tata = [0/0,$   $1/\infty,$   $2/\infty,$   $3/\infty,$   $4/\infty,$   $5/\infty,$   $6/\infty ]$



d/tata

1  
[ 0/0,

2  
 $\infty/0$ ,

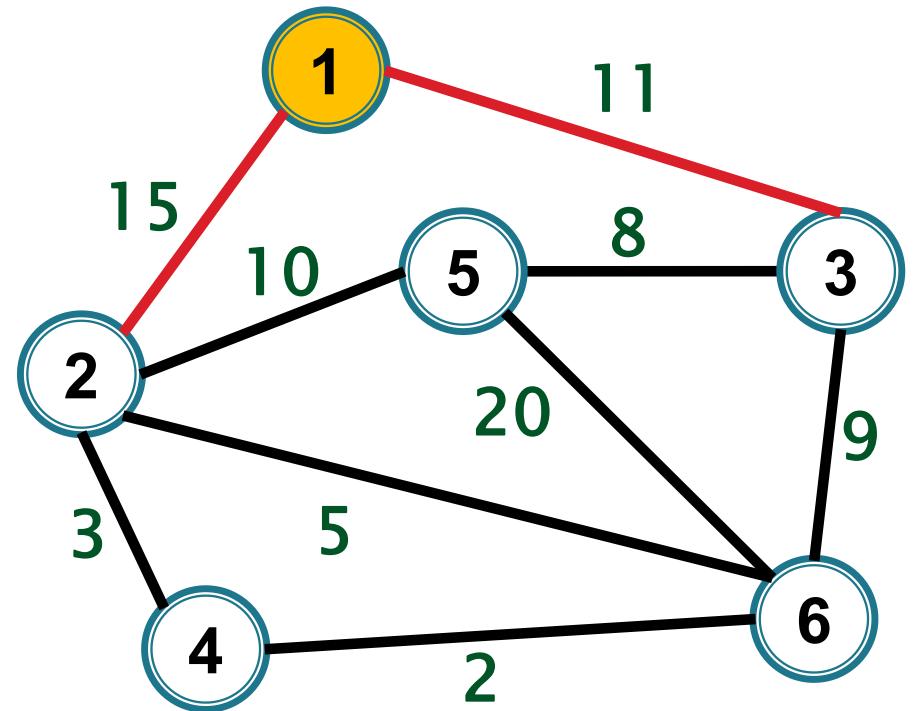
3  
 $\infty/0$ ,

4  
 $\infty/0$ ,

5  
 $\infty/0$ ,

6  
 $\infty/0$  ]

Sel. 1:



d/tata

$[ \frac{1}{0/0},$

$\frac{2}{\infty/0},$

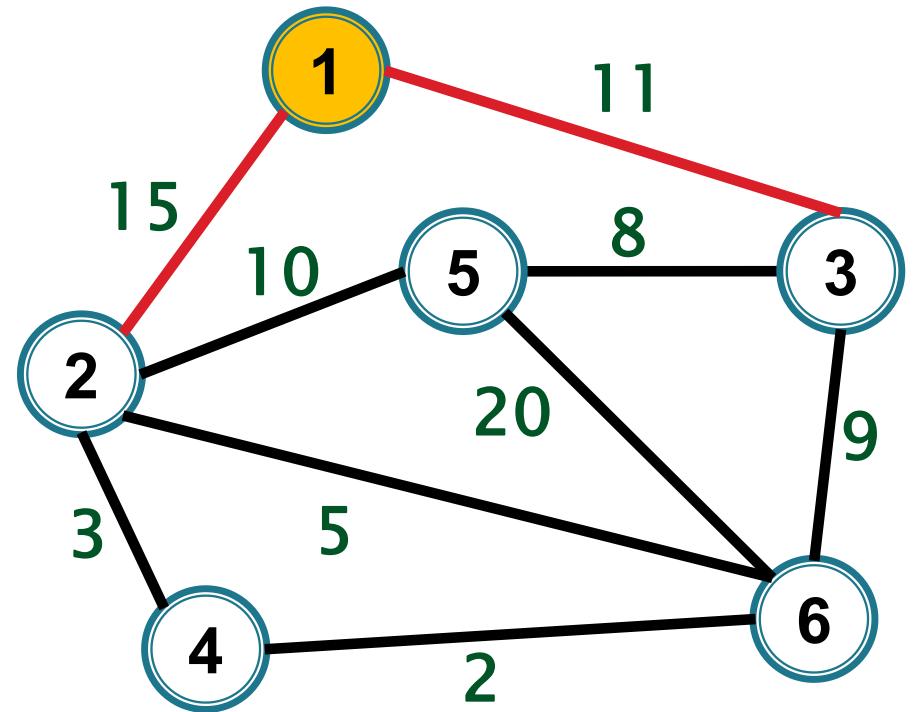
$\frac{3}{\infty/0},$

$\frac{4}{\infty/0},$

$\frac{5}{\infty/0},$

$\frac{6}{\infty/0} ]$

Sel. 1: viz [1] = 1 (vom reprezenta prin - )



d/tata

1  
[ 0/0,

2  
 $\infty/0$ ,

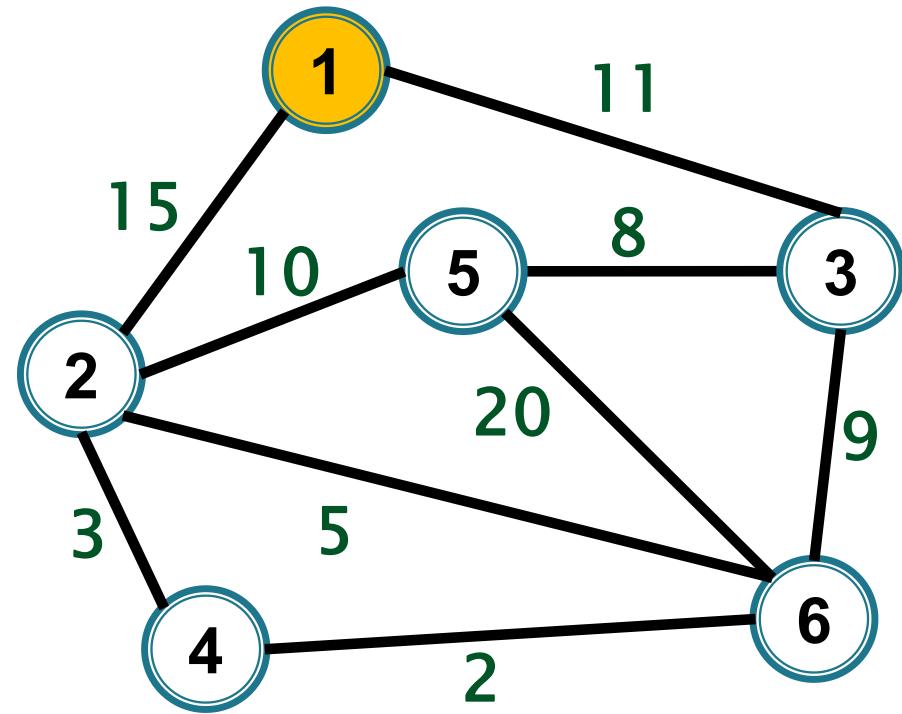
3  
 $\infty/0$ ,

4  
 $\infty/0$ ,

5  
 $\infty/0$ ,

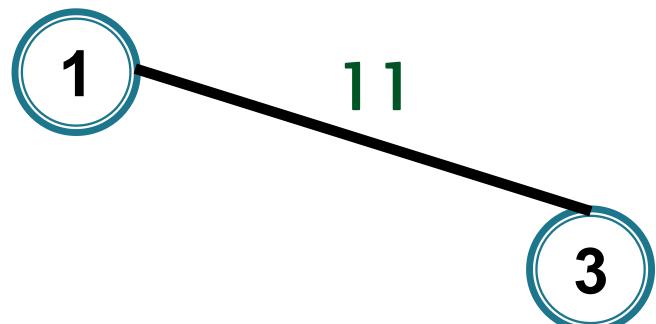
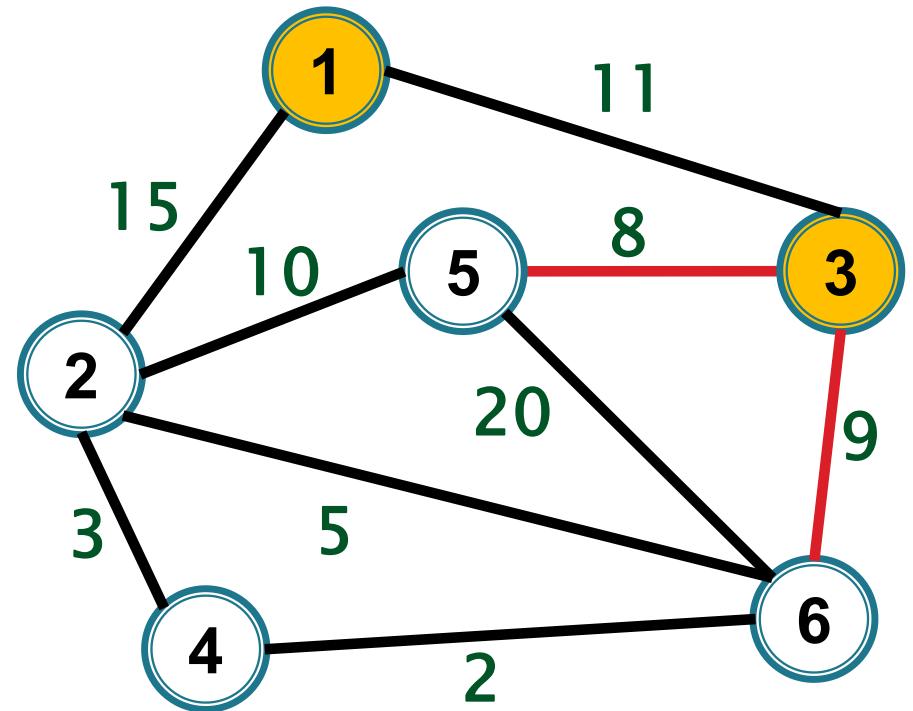
6  
 $\infty/0$  ]

**Sel. 1: actualizăm etichetele vecinilor**



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

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

[  $\frac{1}{0/0}$ ,

$\infty/0$ ,

$\infty/0$ ,

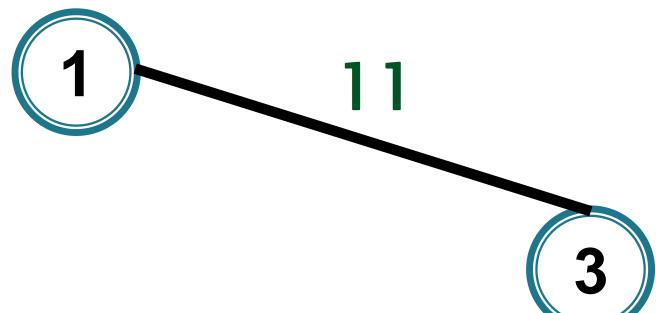
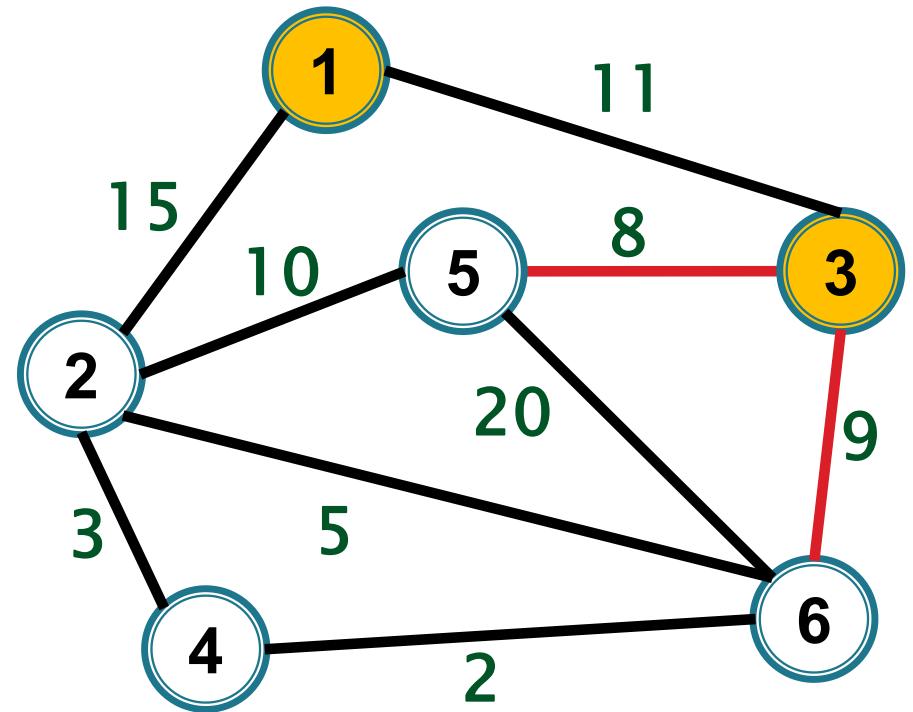
$\infty/0$ ,

$\infty/0$ ,

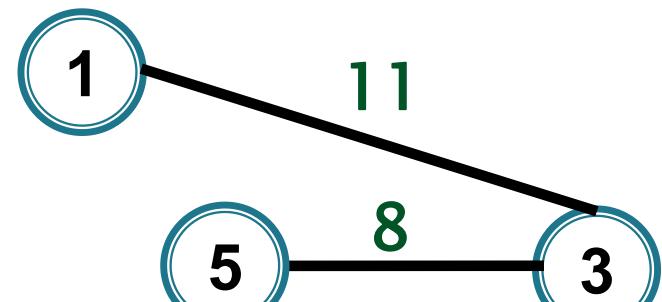
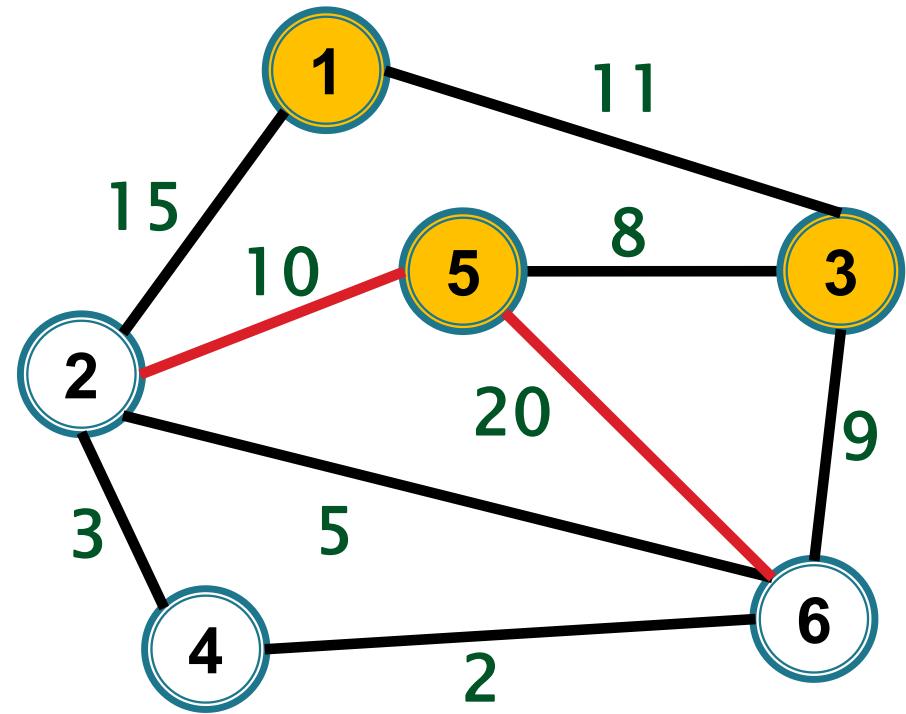
$\infty/0$  ]

Sel. 1: [ - ,  $15/1$ ,  $\frac{11}{1}$ ,

Sel. 3:



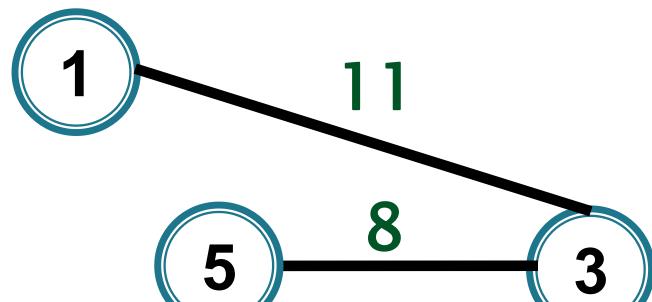
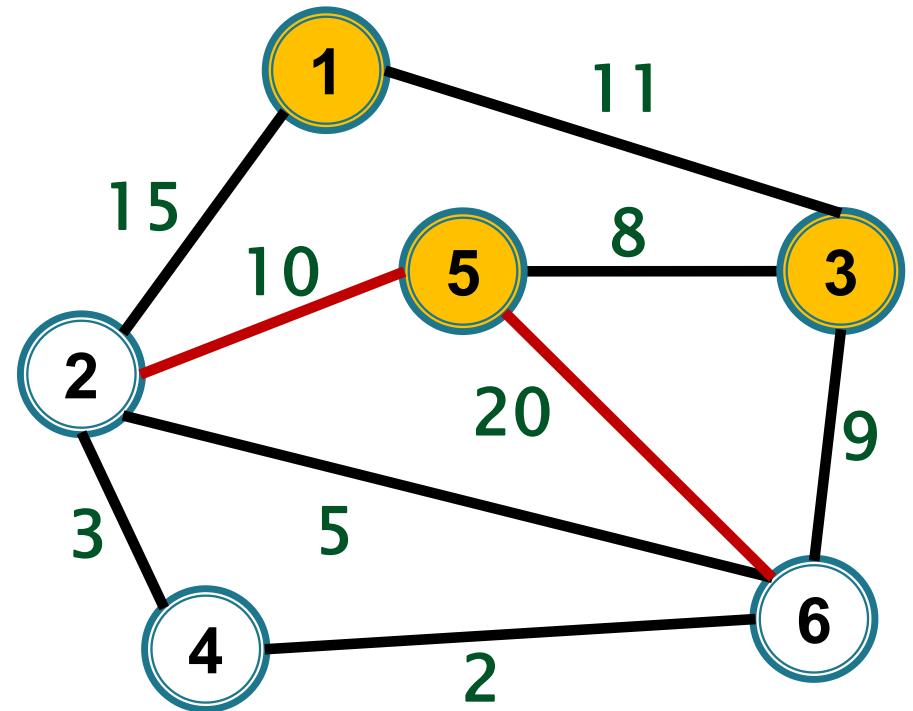
d/tata	[ $\frac{1}{0/0}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 1:	[ -, $\frac{15}{1}$ , $\frac{11}{1}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 3:	[ -, $\frac{15}{1}$ , -, $\infty/0$ , $\frac{8}{3}$ , $\frac{9}{3}$ ]



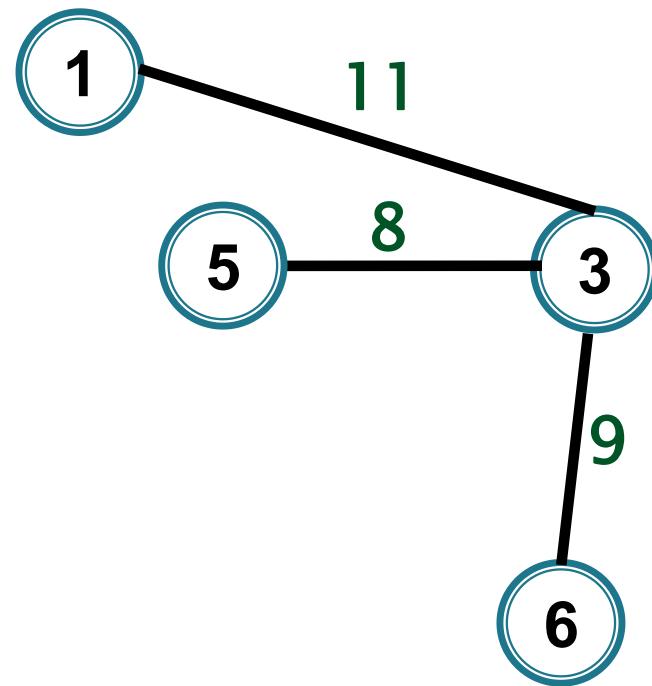
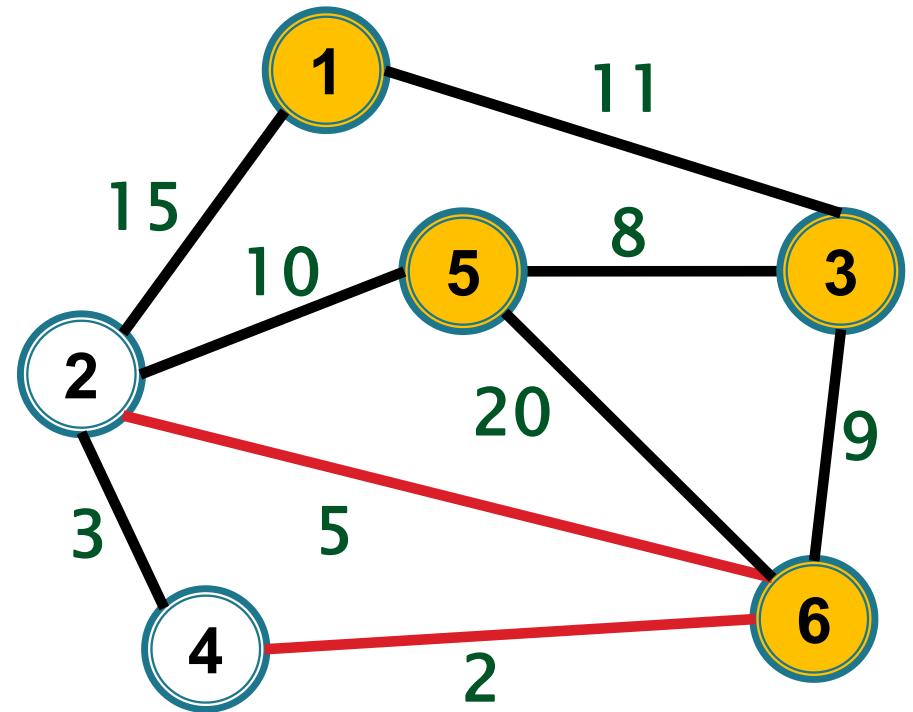
d/tata	[ $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 1:	[ $-$ , $15/1$ , $11/1$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 3:	[ $-$ , $15/1$ , $-$ , $\infty/0$ , $8/3$ , $9/3$ ]
Sel. 5:	

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

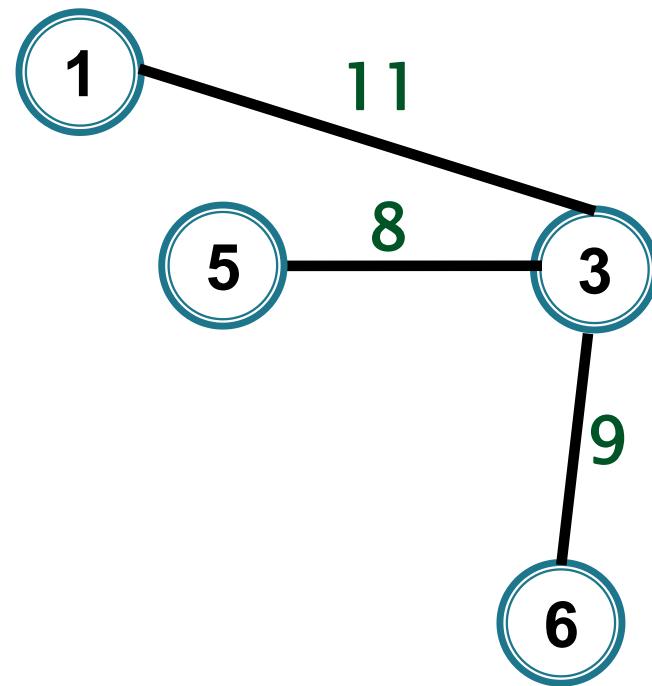
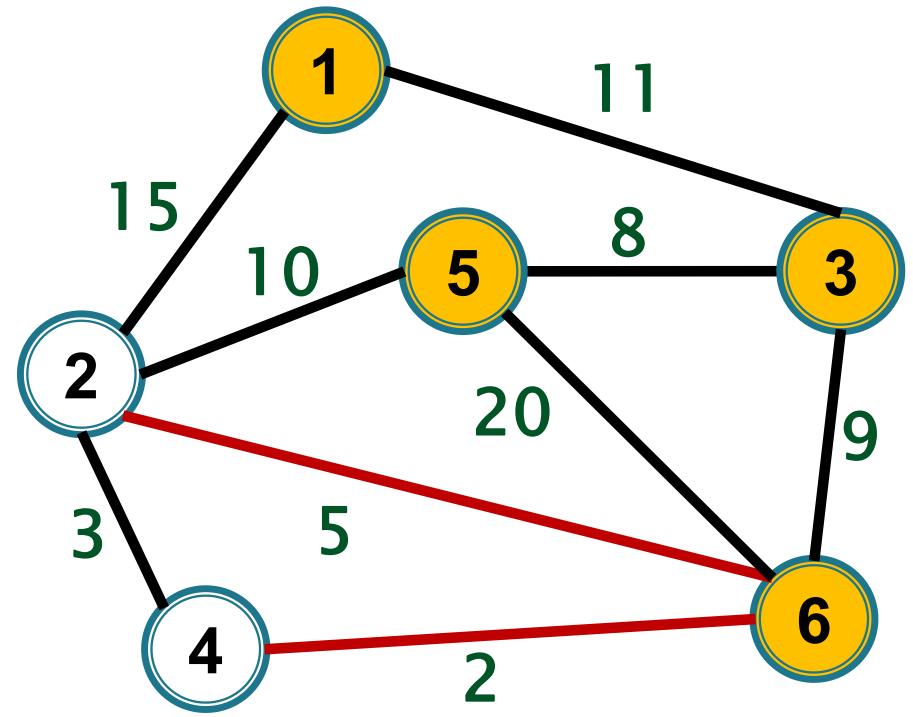
$d[6] = \min\{d[6], w(6,5)\}$



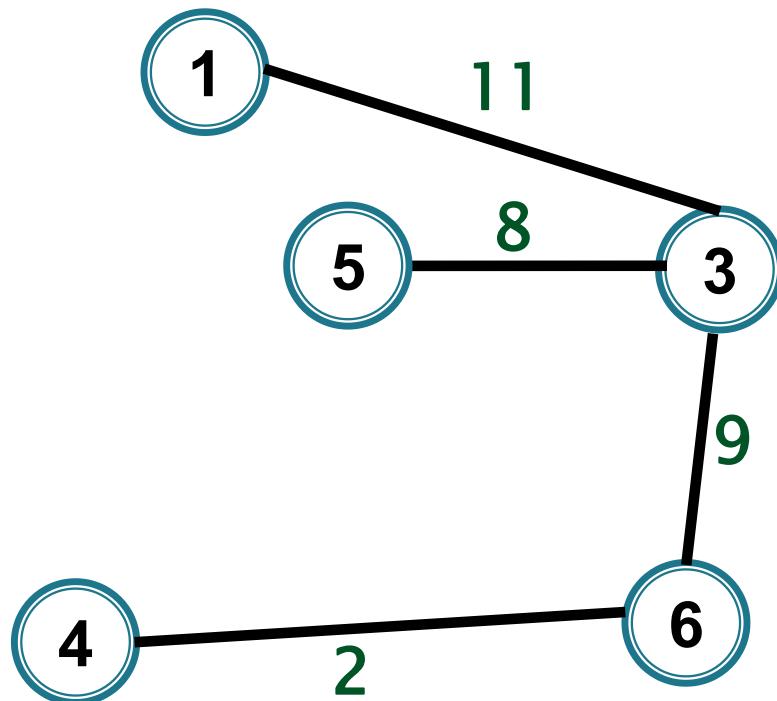
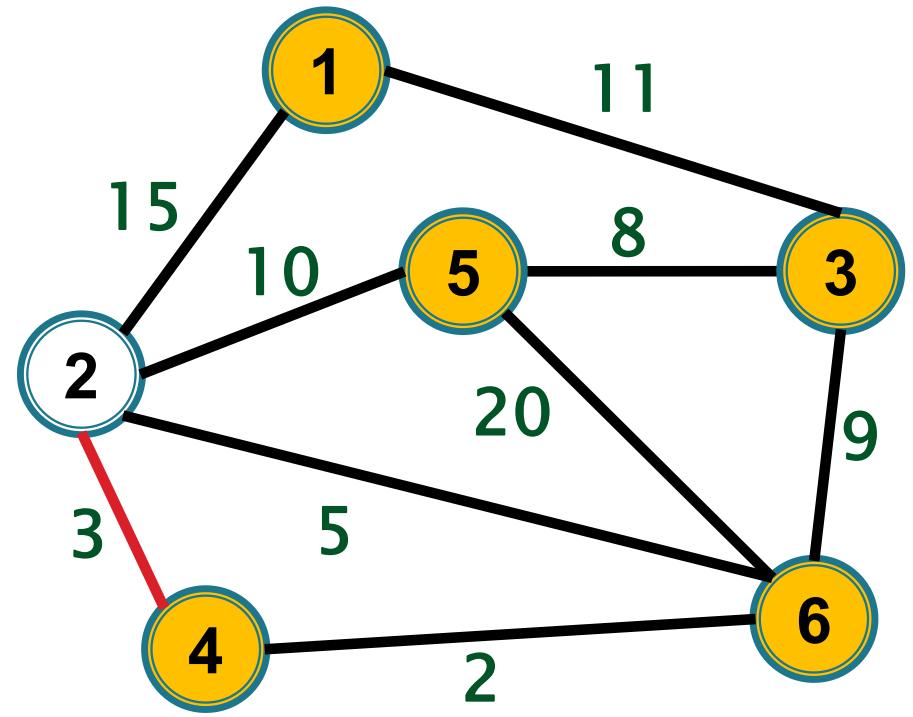
d/tata	[ <b>1</b> <b>0/0</b> ,	<b>2</b> $\infty/0$ ,	<b>3</b> $\infty/0$ ,	<b>4</b> $\infty/0$ ,	<b>5</b> $\infty/0$ ,	<b>6</b> $\infty/0$ ]
Sel. 1:	[ - , <b>15/1</b> ,	<b>11/1</b> ,	- ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
Sel. 3:	[ - , <b>15/1</b> ,	- ,	$\infty/0$ ,	<b>8/3</b> ,	<b>9/3</b> ]	
Sel.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



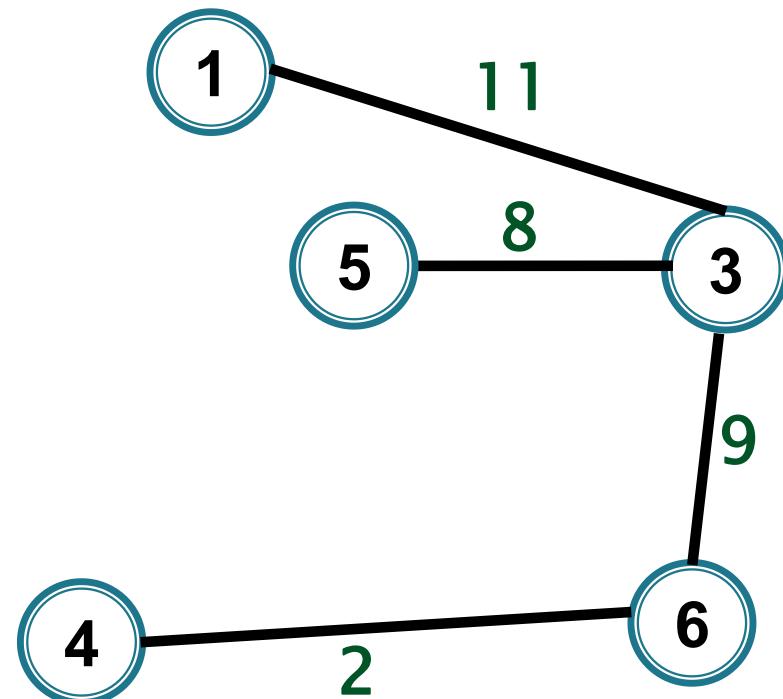
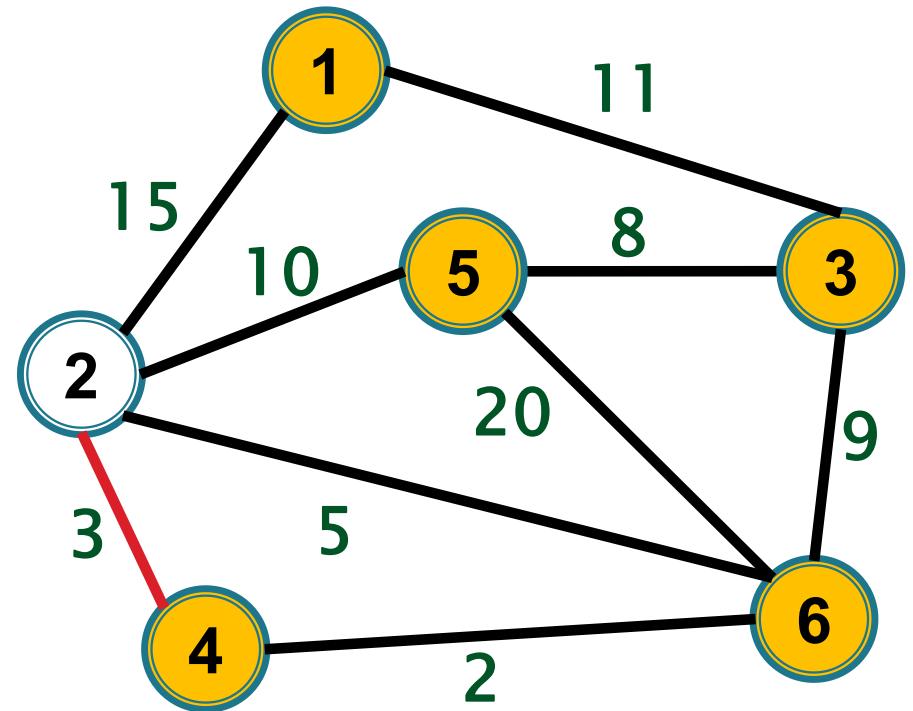
d/tata	[ $\frac{1}{0/0}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 1:	[ -, $\frac{15}{1}$ , $\frac{11}{1}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 3:	[ -, $\frac{15}{1}$ , -, $\infty/0$ , $\frac{8}{3}$ , $\frac{9}{3}$ ]
Sel. 5:	[ -, $\frac{10}{5}$ , -, $\infty/0$ , -, $\frac{9}{3}$ ]
Sel. 6:	



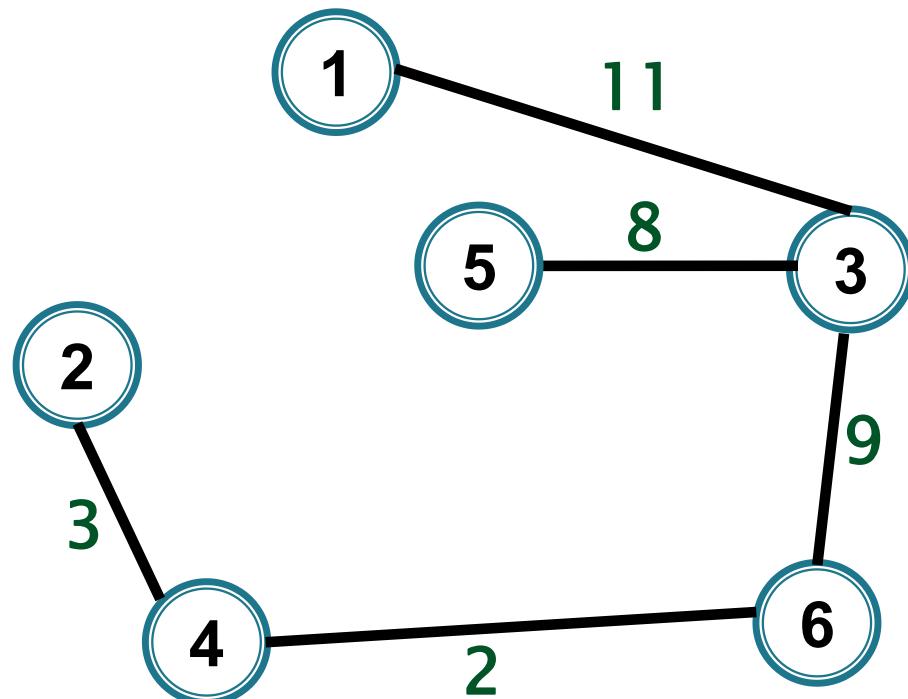
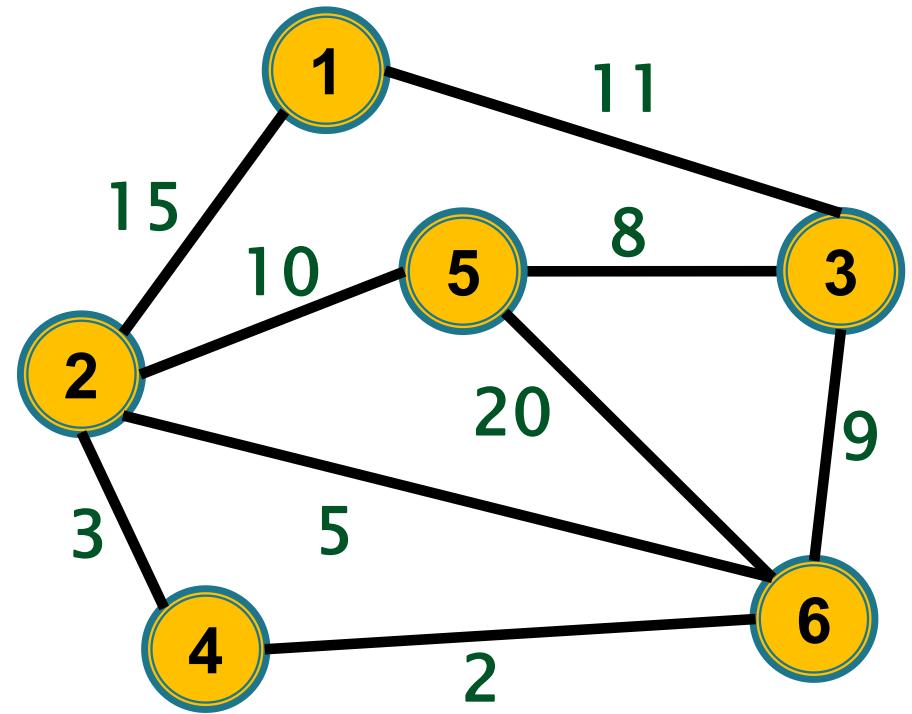
d/tata	[ $\frac{1}{0/0}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 1:	[ -, $15/1$ , $11/1$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 3:	[ -, $15/1$ , -, $\infty/0$ , $8/3$ , $9/3$ ]
Sel. 5:	[ -, $10/5$ , -, $\infty/0$ , -, $9/3$ ]
Sel. 6:	[ -, $5/6$ , -, $2/6$ , -, - ]



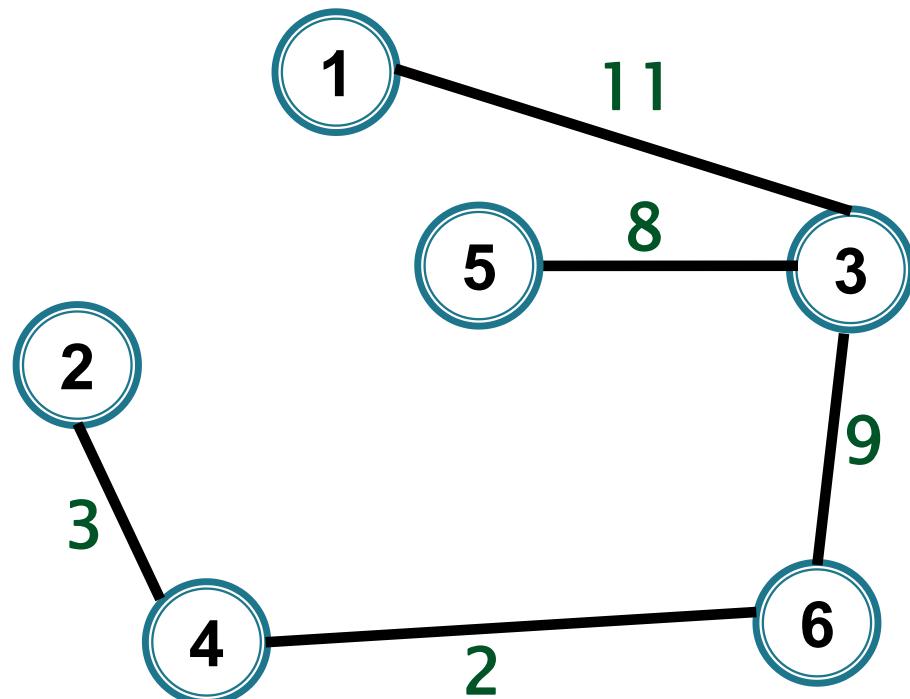
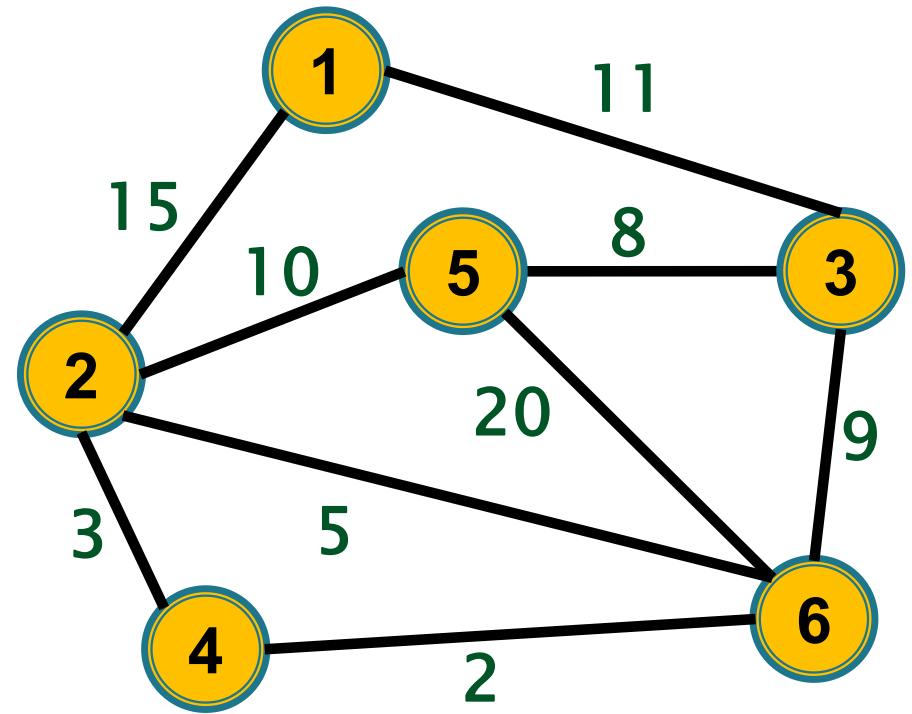
d/tata	[ $\frac{1}{0/0}$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 1:	[ -, $15/1$ , $11/1$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
Sel. 3:	[ -, $15/1$ , -, $\infty/0$ , $8/3$ , $9/3$ ]
Sel. 5:	[ -, $10/5$ , -, $\infty/0$ , -, $9/3$ ]
Sel. 6:	[ -, $5/6$ , -, $2/6$ , -, - ]
Sel. 4:	



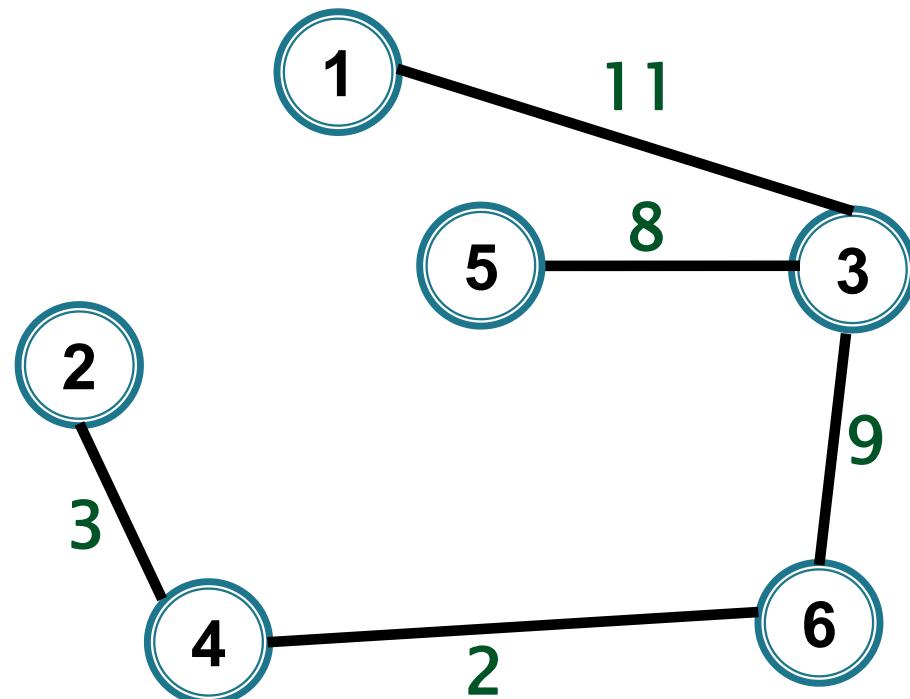
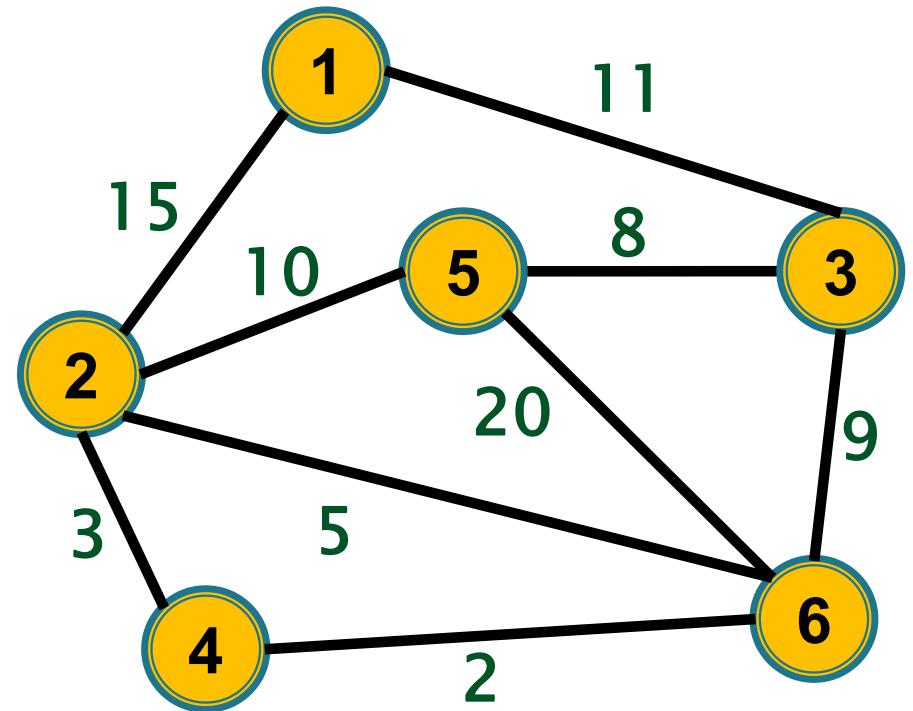
d/tata	$\frac{1}{0/0}$	$\frac{2}{\infty/0}$	$\frac{3}{\infty/0}$	$\frac{4}{\infty/0}$	$\frac{5}{\infty/0}$	$\frac{6}{\infty/0}$
Sel. 1:	[ - , $15/1$ , $11/1$ ]					
Sel. 3:	[ - , $15/1$ , - , $\infty/0$ , $8/3$ ]					
Sel. 5:	[ - , $10/5$ , - , $\infty/0$ , - , $9/3$ ]					
Sel. 6:	[ - , $5/6$ , - , $2/6$ , - , - ]					
Sel. 4:	[ - , $3/4$ , - , - , - , - ]					



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



d/tata	$[ \textcolor{red}{0/0}, \infty/0, \infty/0, \infty/0, \infty/0, \infty/0 ]$
Sel. 1:	$[ -, 15/1, \textcolor{red}{11/1}, \infty/0, \infty/0, \infty/0 ]$
Sel. 3:	$[ -, 15/1, -, \infty/0, \textcolor{red}{8/3}, 9/3 ]$
Sel. 5:	$[ -, 10/5, -, \infty/0, -, \textcolor{red}{9/3} ]$
Sel. 6:	$[ -, 5/6, -, \textcolor{red}{2/6}, -, - ]$
Sel. 4:	$[ -, \textcolor{red}{3/4}, -, -, -, - ]$
Sel. 2:	$[ -, -, -, -, -, - ]$



d/tata  
FINAL

1	2	3	4	5	6
$0/0$	$3/4$	$11/1$	$2/6$	$8/3$	$9/3$

Vectorul tata  $\Rightarrow$  muchiile arborelui  $(u, \text{tata}[u])$ ,  $u \neq s$

# Prim

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

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

$\text{Prim}(G, w, s)$

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

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

pentru fiecare  $v$  adiacent cu  $u$  executa

daca  $v \in Q$  si  $w(u, v) < d[v]$  atunci

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

$tata[v] = u$

???

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

**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  $d$  minimă din  $Q$

pentru fiecare  $v$  adiacent cu  $u$  executa

daca  $v \in Q$  si  $w(u, v) < d[v]$  atunci

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

$tata[v] = u$

//actualizeaza  $Q$  - pentru  $Q$  heap

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

# Prim

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

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

# Prim

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

$\Rightarrow O(n^2)$  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 maxim 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**

# 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 ?

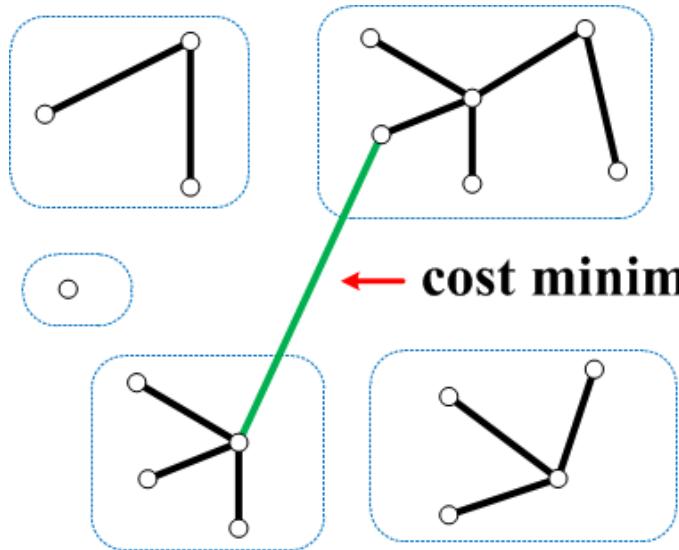
# Corectitudine Kruskal + Prim

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

## ► Amintim

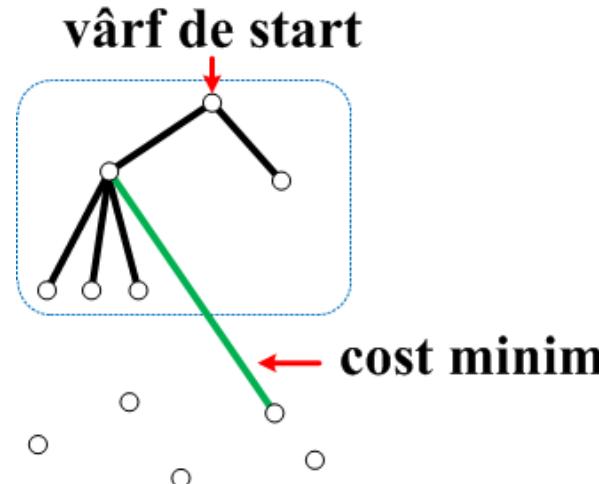
### Kruskal

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



### Prim

- $s$  - vârful de start
- Inițial  $T = (\{s\}; \emptyset)$
- pentru  $i = 1, n-1$ 
  - alege o muchie  $uv$  cu **cost minim** 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

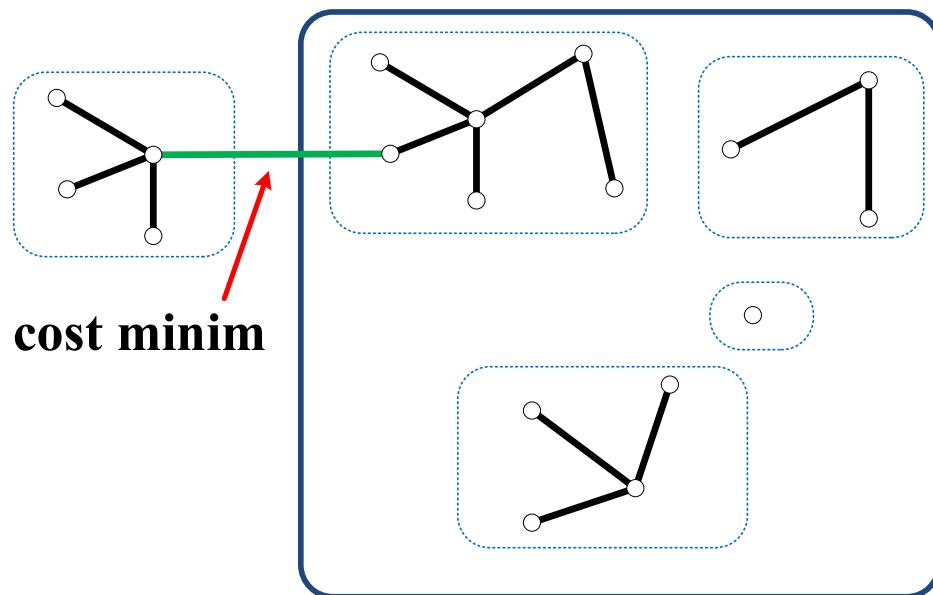
- ▶ vom demonstra un criteriu de alegere a muchiei  $e$  la un pas astfel încât:

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

și

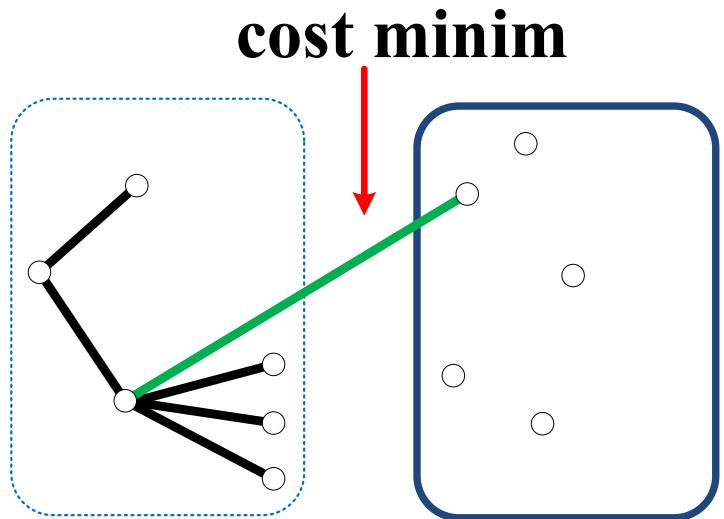
- ▶ vom demonstra că algoritmii lui Kruskal și Prim aplică acest criteriu.

## Kruskal



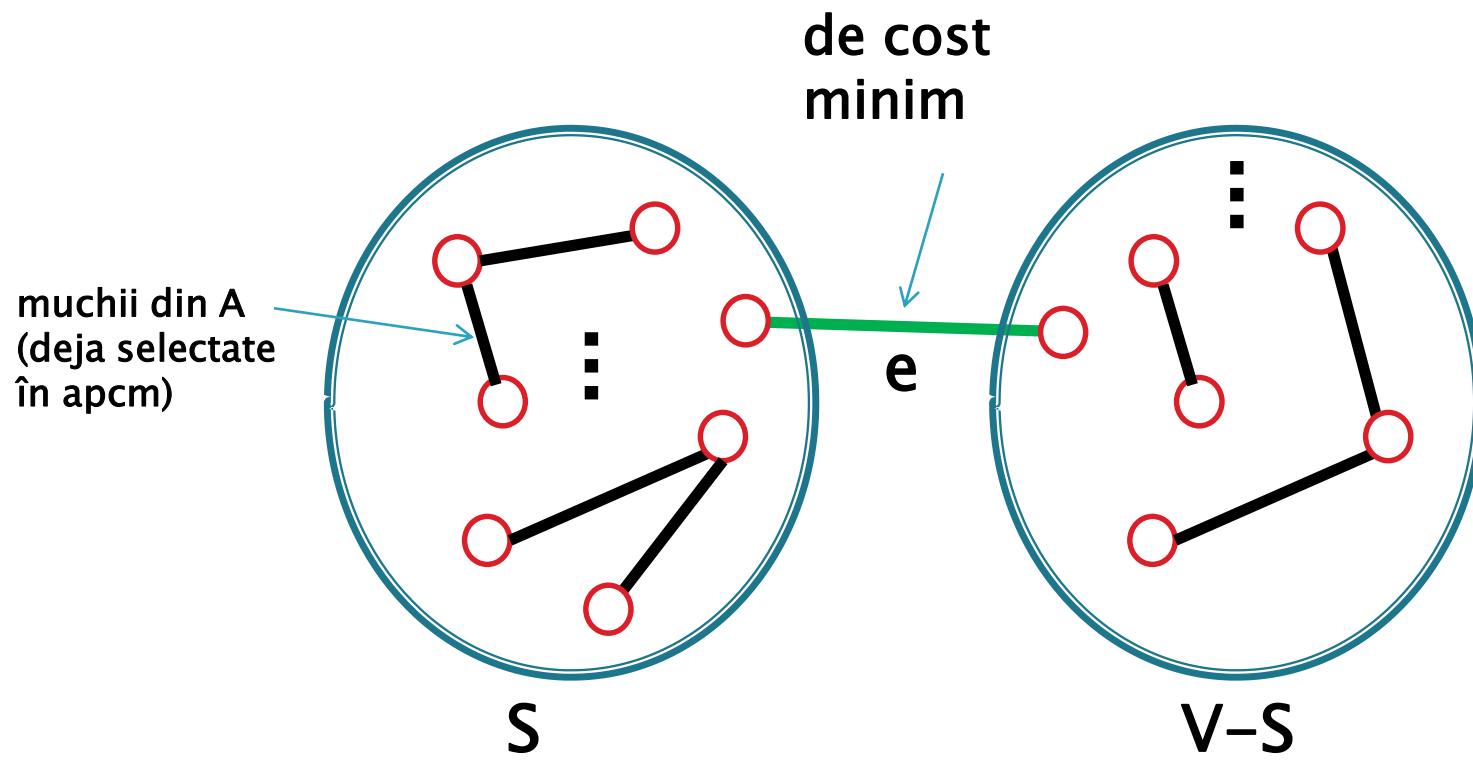
**cost minim**

## Prim



**cost minim**

# Corectitudine Kruskal + Prim

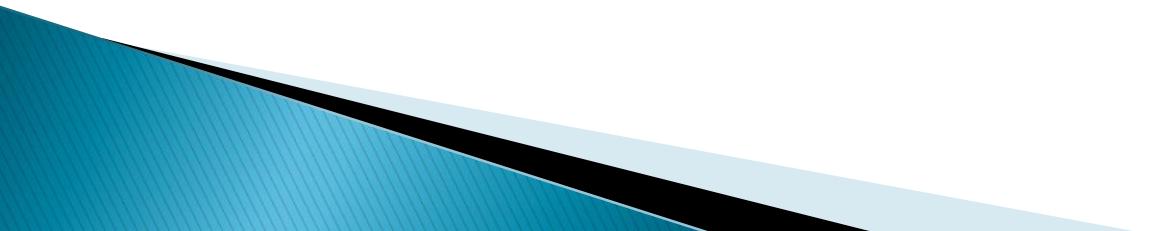


$$A \subseteq \text{apcm} \Rightarrow A \cup \{e\} \subseteq \text{apcm}$$

# Corectitudine Kruskal + Prim

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

- ▶ **Propoziție.** Algoritmul Kruskal determină un apcm
- ▶ **Propoziție.** Algoritmul Prim determină un apcm



# **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:

- ▶  $G$  – graf **orientat** ponderat
- ▶  $P$  – drum

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

**costul/ponderea/lungimea** drumului  $P$

# Cadru

Fie:

- ▶  $G$  – graf **orientat** ponderat
- ▶ **Presupunem că  $G$  nu conține circuite de pondere negativă**

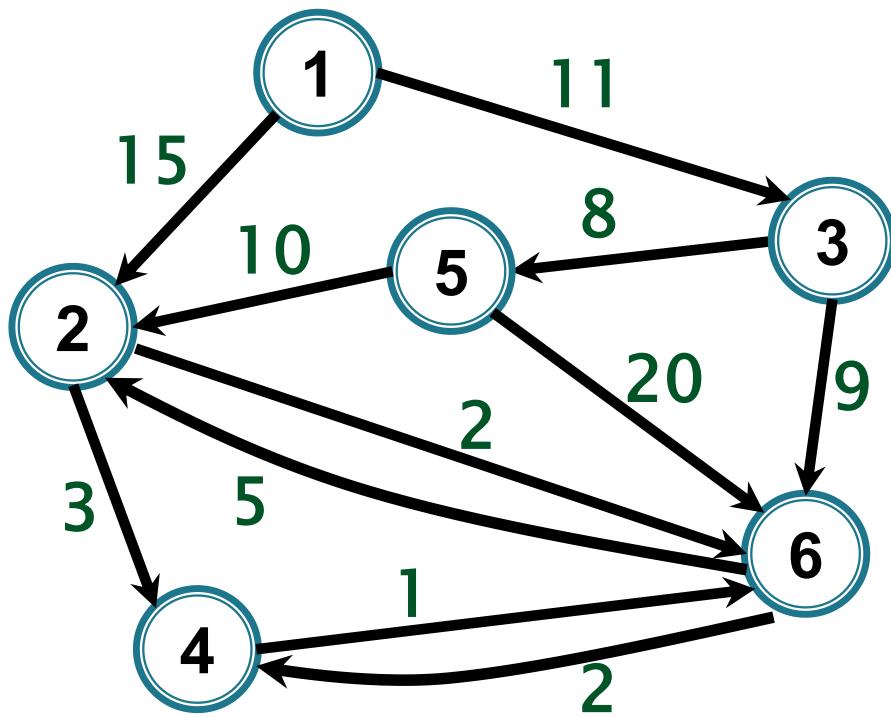
# Cadru

- ▶ 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  $w(P) = \delta_G(s, v)$



## **Tipuri de probleme de drumuri minime**

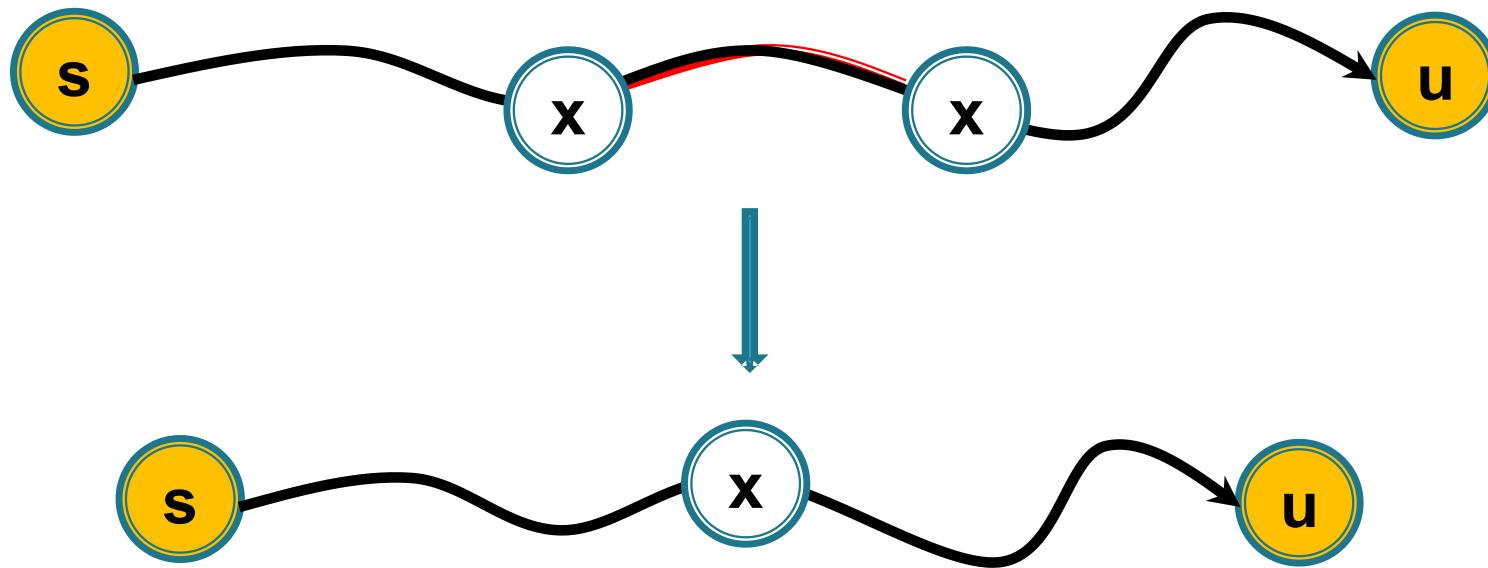
- Între două vârfuri date
- de la un vârf la toate celelalte
- Între oricare două vârfuri

### **Situări:**

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

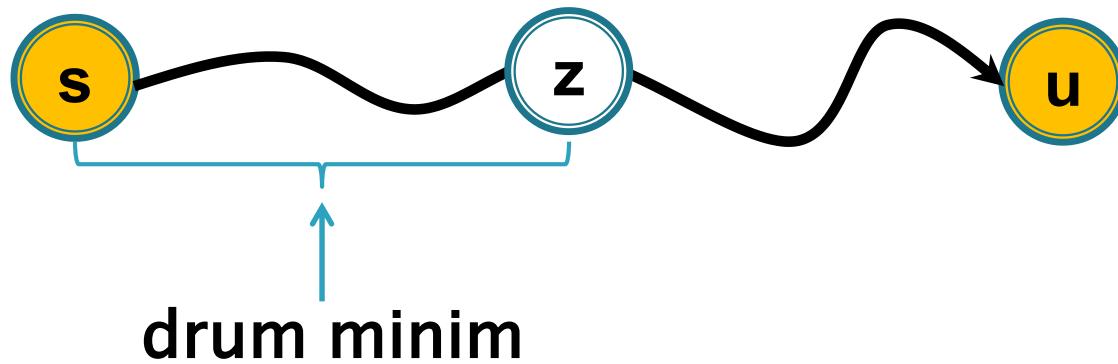
# Observații

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

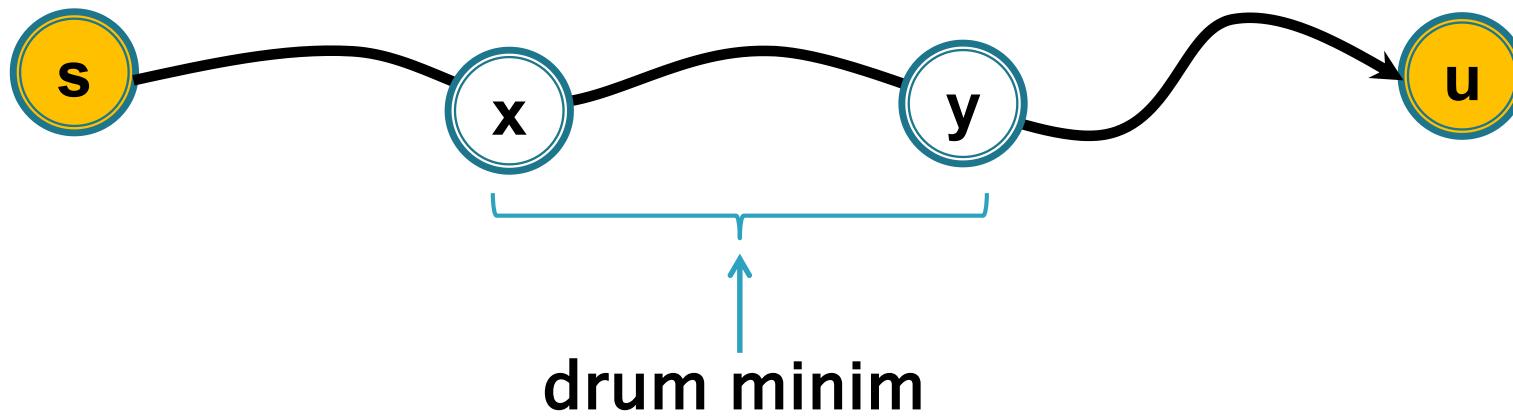


# Observații

- ▶ **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)**

# Problema drumurilor minime de sursă unică (de la s la celealte 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



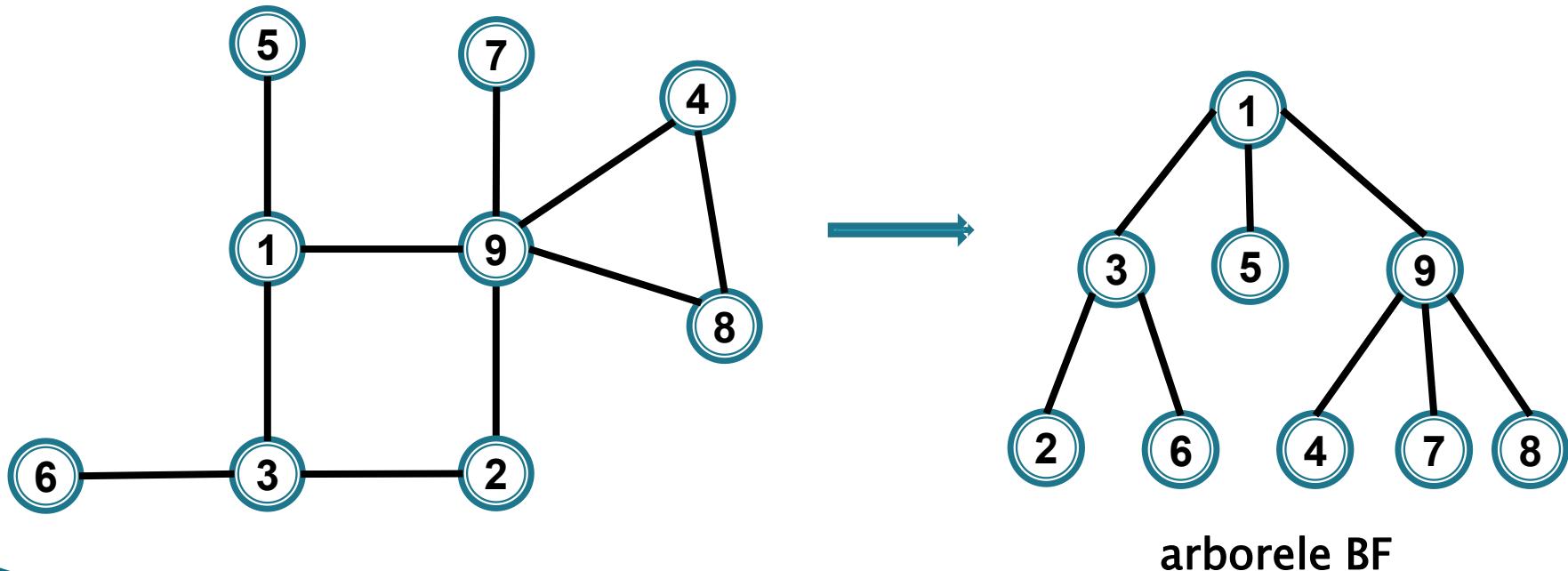
► Dacă  $G$  **nu** este ponderat, cum putem calcula distanțele față de  $s$ ?

# Drumuri minime de sursă unică s

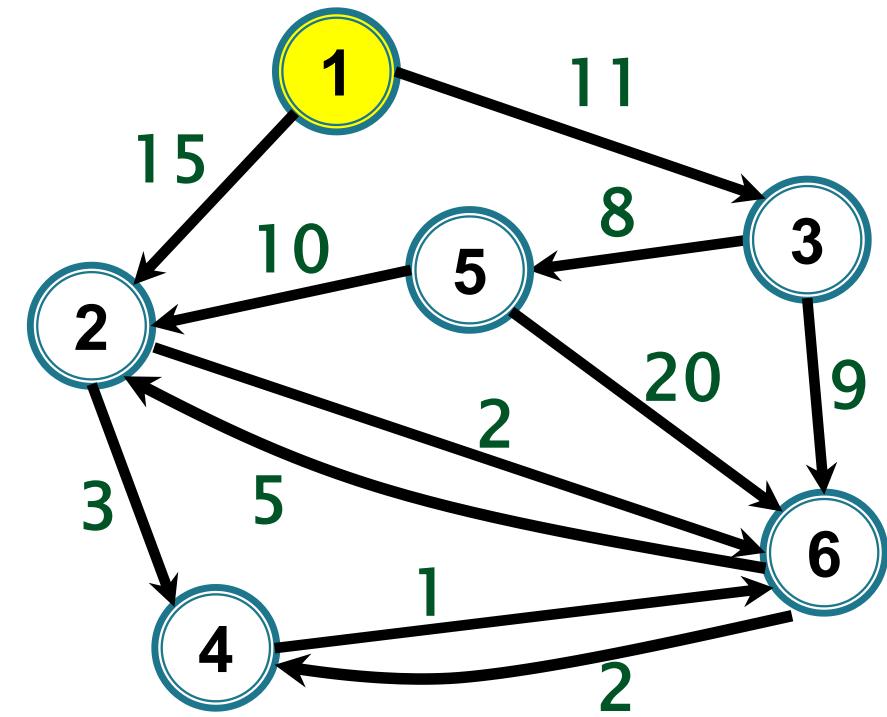
## ► Amintim

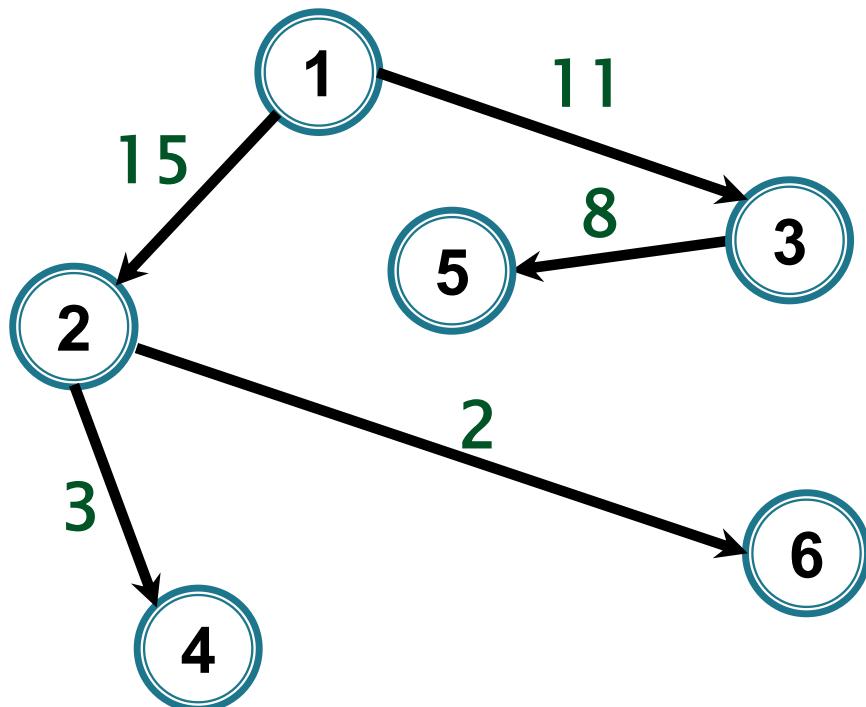
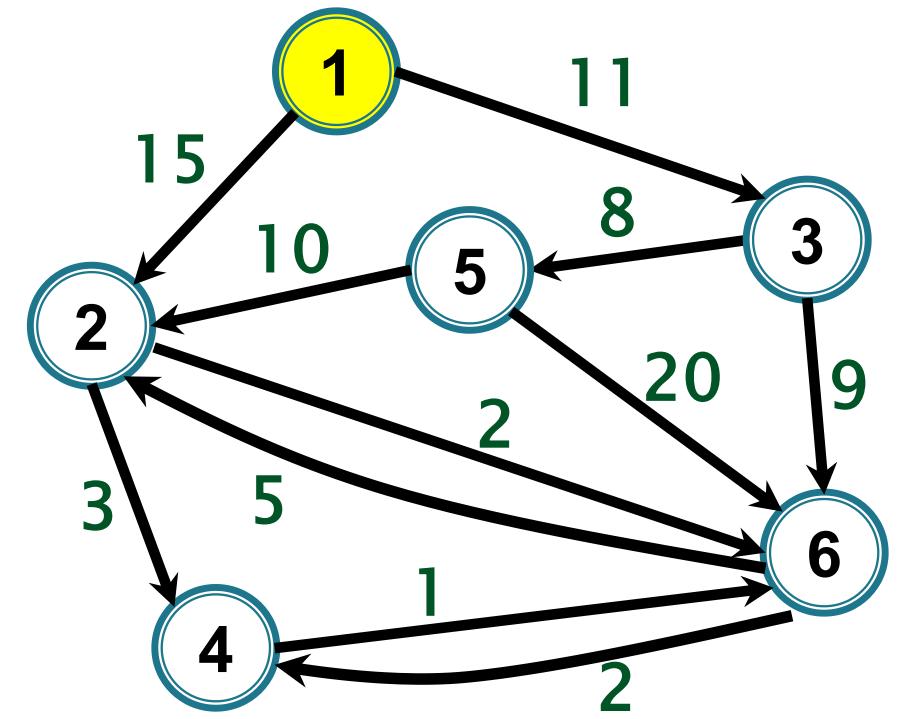
În cazul unui graf neponderat, problema se rezolvă folosind parcurgerea BF din s

⇒ arborele BF (al distanțelor față de s)



$s = 1$



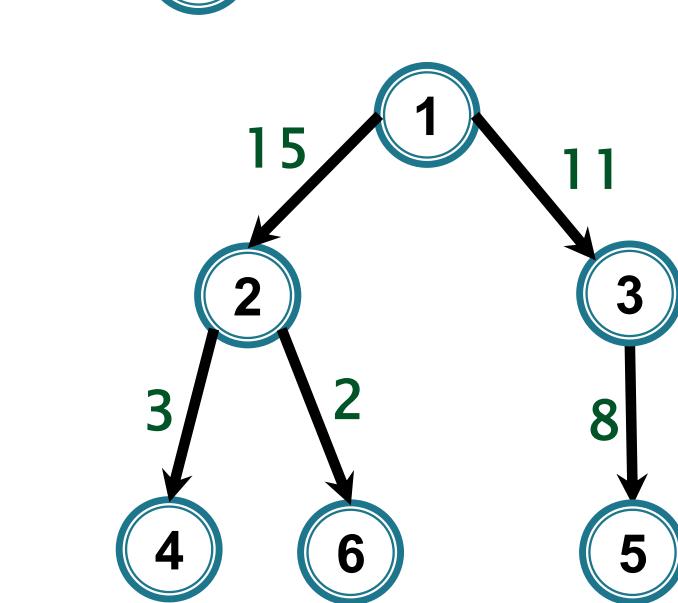
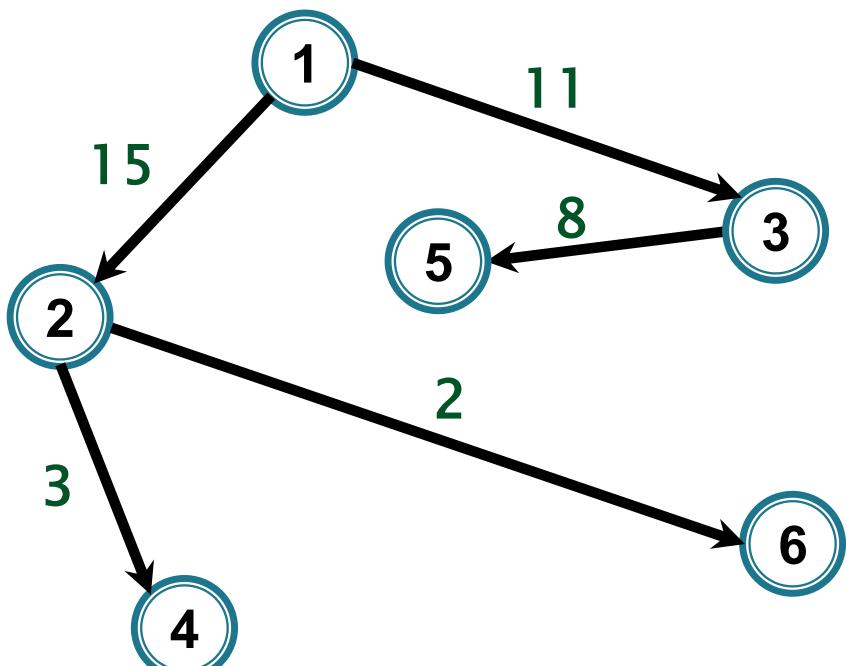
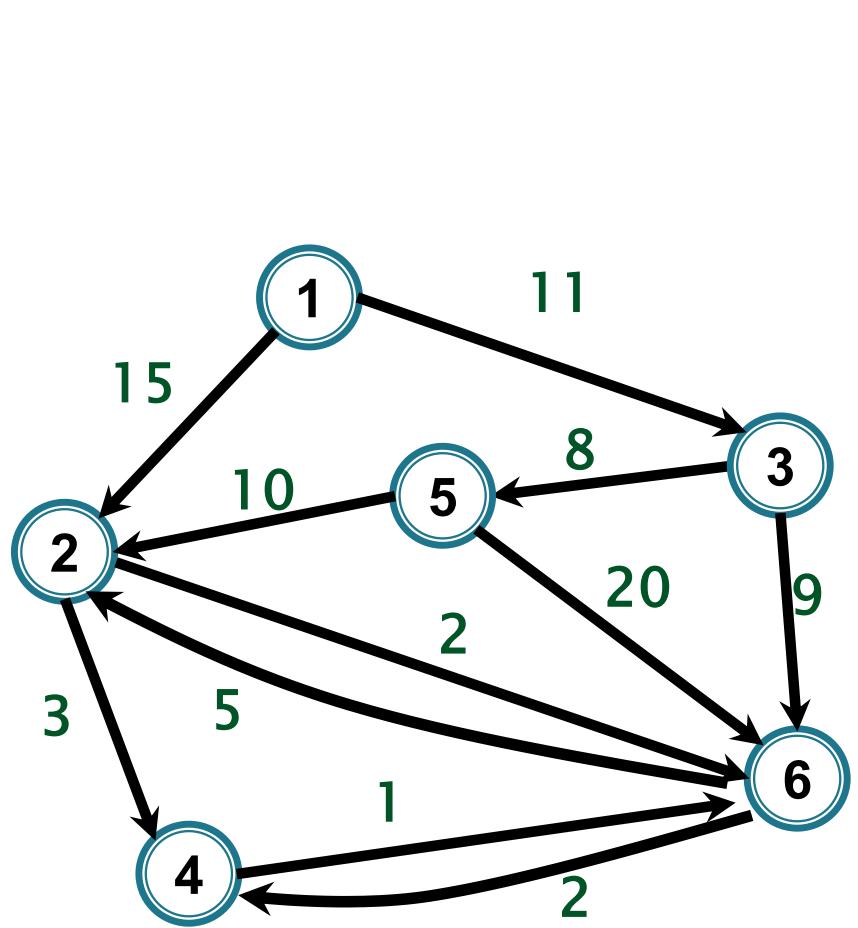


arbore al distanțelor față de 1

**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 celealte vârfuri accesibile din s

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

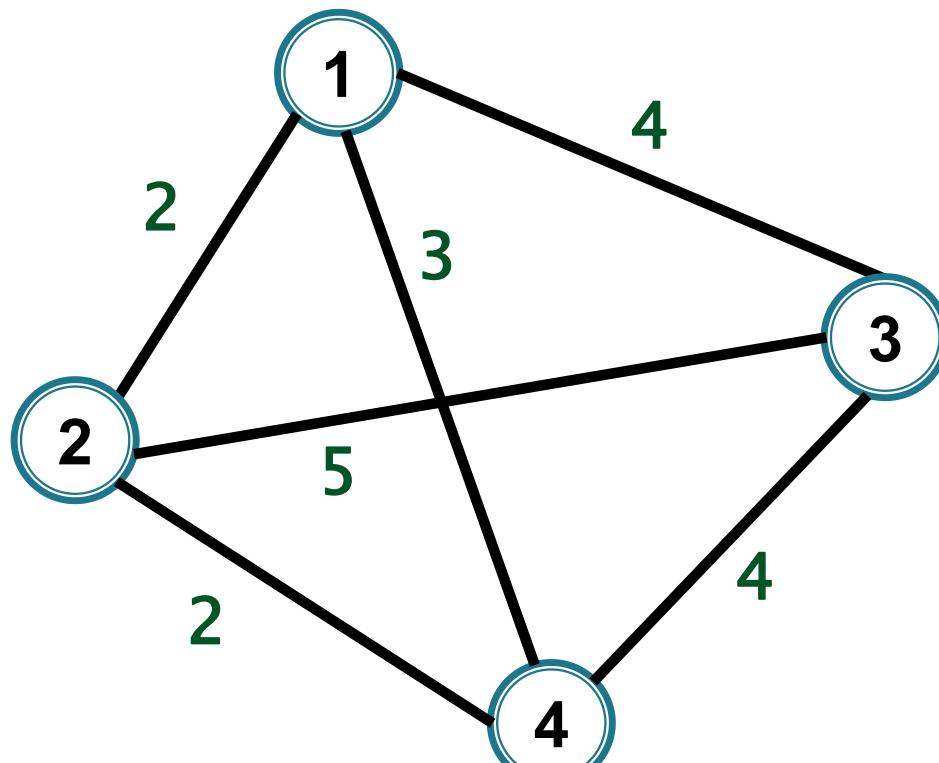
graful neorientat asociat lui T fiind arbore cu rădăcina în s  
(cu arcele corespunzătoare orientate de la s la frunze)



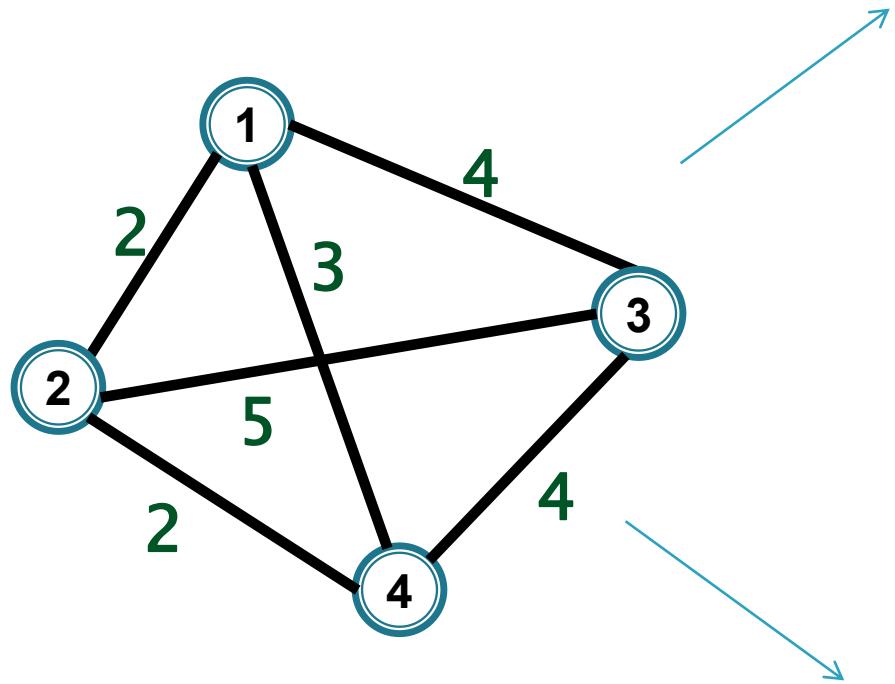
arbore al distanțelor față de 1

- ▶ 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**

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



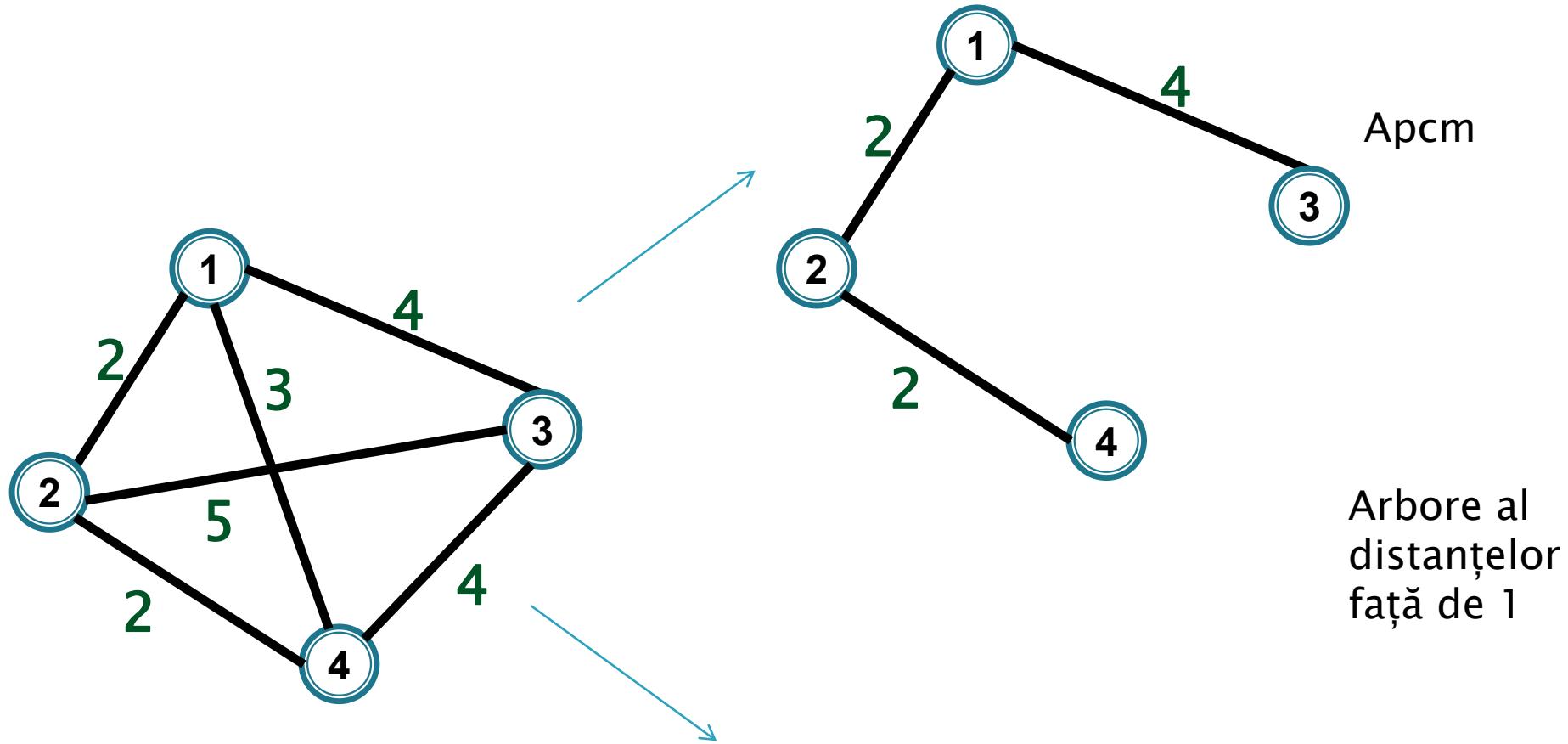
Un arbore parțial de cost minim nu este neapărat un arbore de distanțe minime



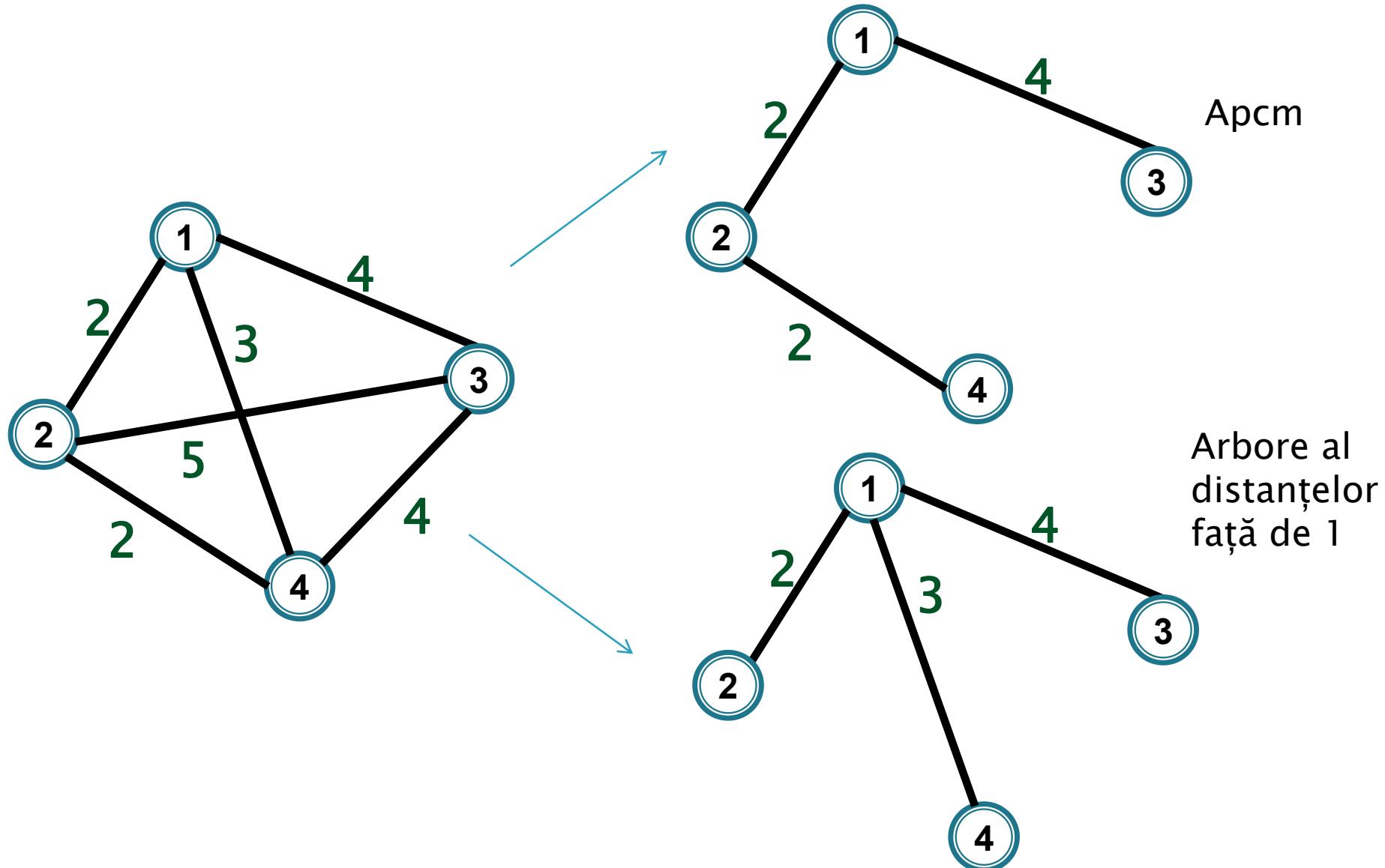
Apcm

Arbore al  
distanțelor  
față de 1

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



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



# Drumuri minime de sursă unică s



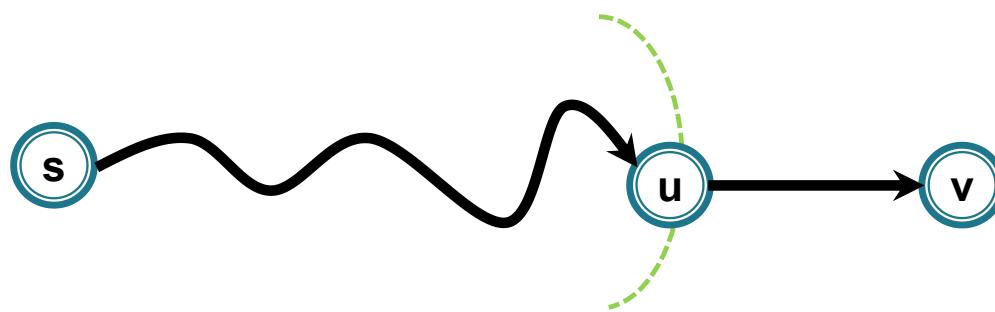
- ▶ În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

# 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  $\Rightarrow$

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

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

# Drumuri minime de sursă unică s

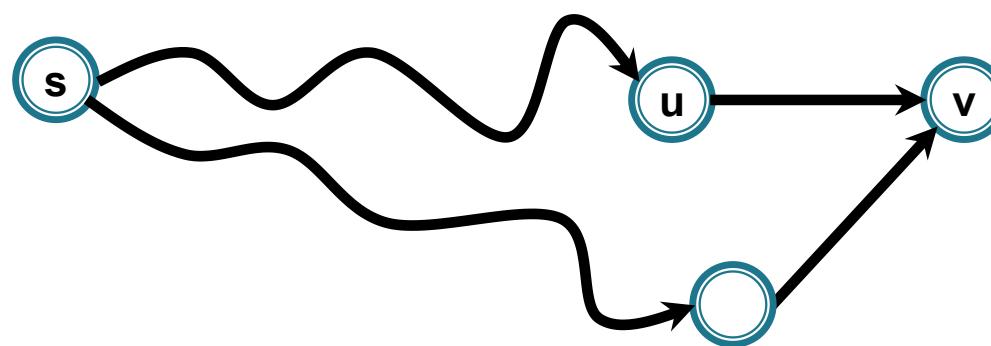
- În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?



“din aproape în aproape”  $\Rightarrow$

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

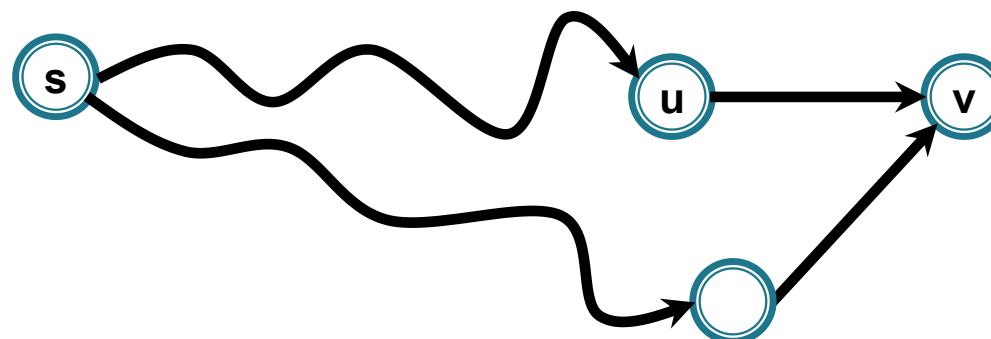
- ▶ În ce ordine considerăm vârfurile pentru a calcula distanțele față de s?

“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

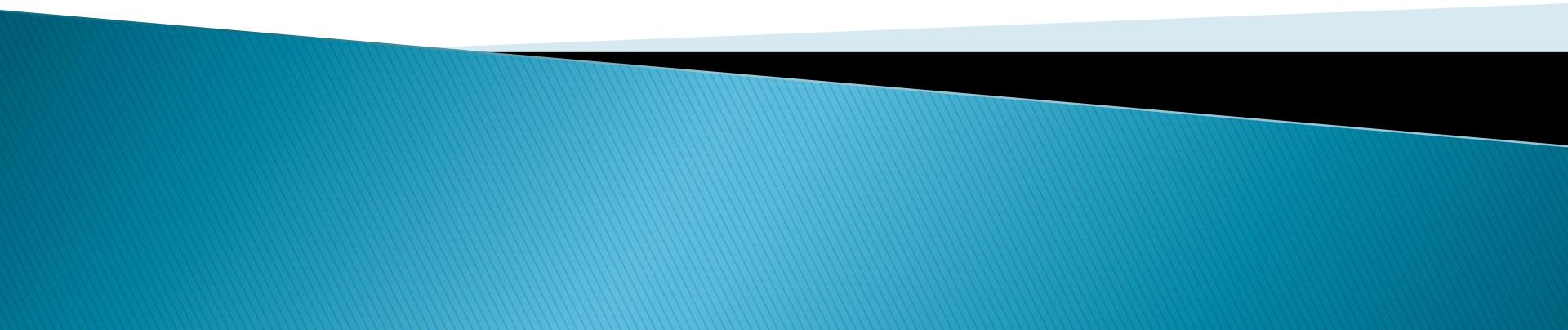


Dacă există circuite – estimăm distanțele pe parcursul algoritmului și considerăm vârful care este **estimat** a fi cel mai aproape de s

# Drumuri minime de sursă unică și căi minime

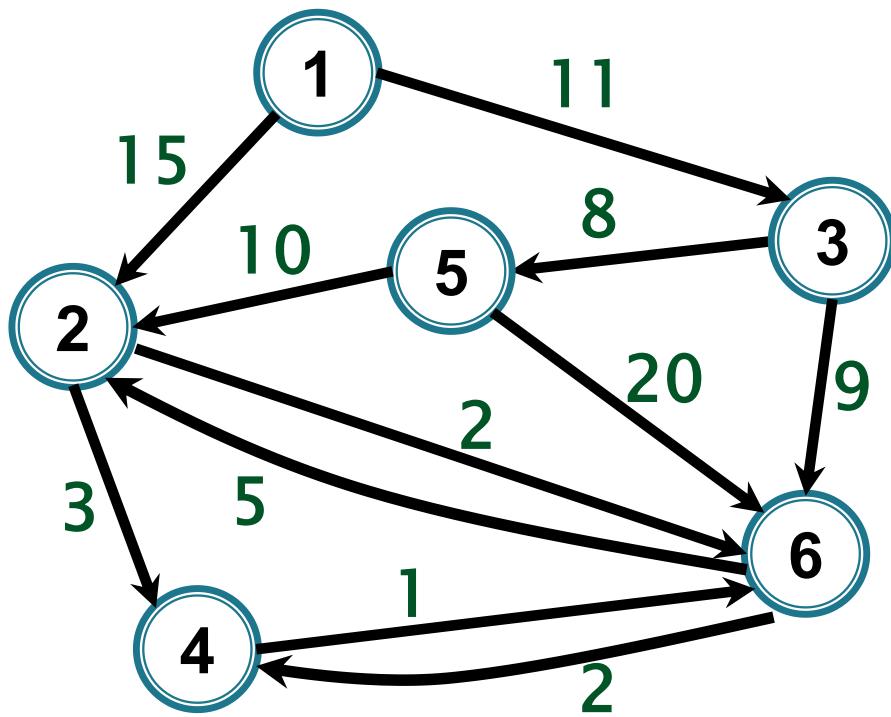
- ▶ Algoritmi pentru grafuri orientate cu circuite, dar cu ponderi pozitive – Dijkstra
- ▶ Algoritmi pentru grafuri orientate fără circuite (cu ponderi reale) DAGs = Directed Acyclic Graphs
- ▶ Algoritmi pentru grafuri orientate cu circuite și ponderi reale, care detectează existența de circuite negative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

# Algoritmul lui Dijkstra



## ► Ipoteză:

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



# Algoritmul lui Dijkstra

Idee: La un pas este ales ca vârf curent (vizitat) vârful u care 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 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

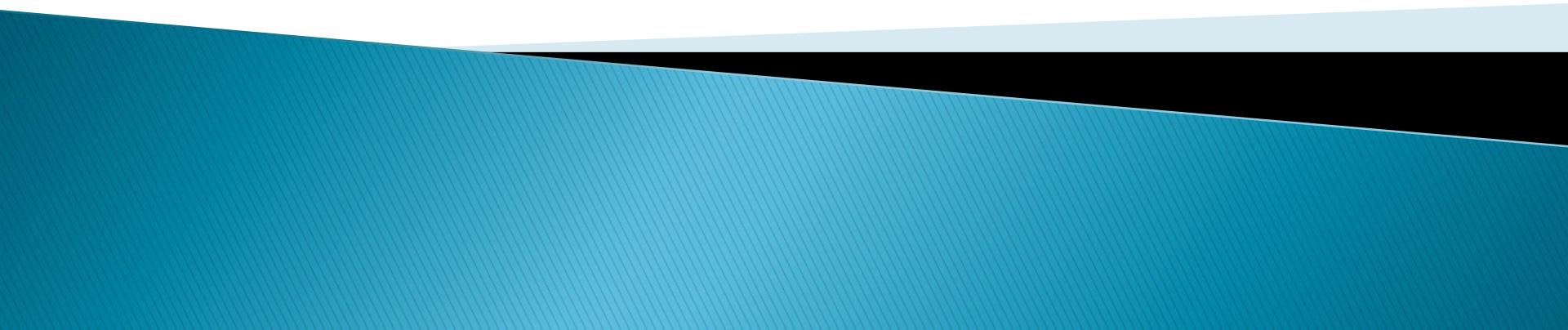
+ se descoperă noi drumuri către vecinii lui  $\Rightarrow$   
se actualizează distanțele estimate pentru vecini



# Algoritmul lui Dijkstra

- generalizare a ideii de parcurgere BF
- dacă toate arcele au cost egal  $Dijkstra = BF$

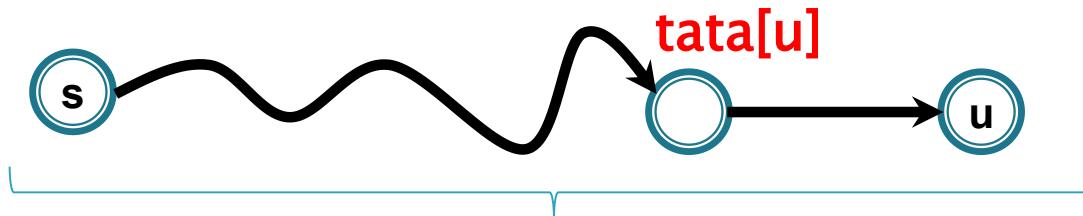
# Pseudocod



# Algoritmul lui Dijkstra

## ► Reținem pentru fiecare vârf etichetele

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



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

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

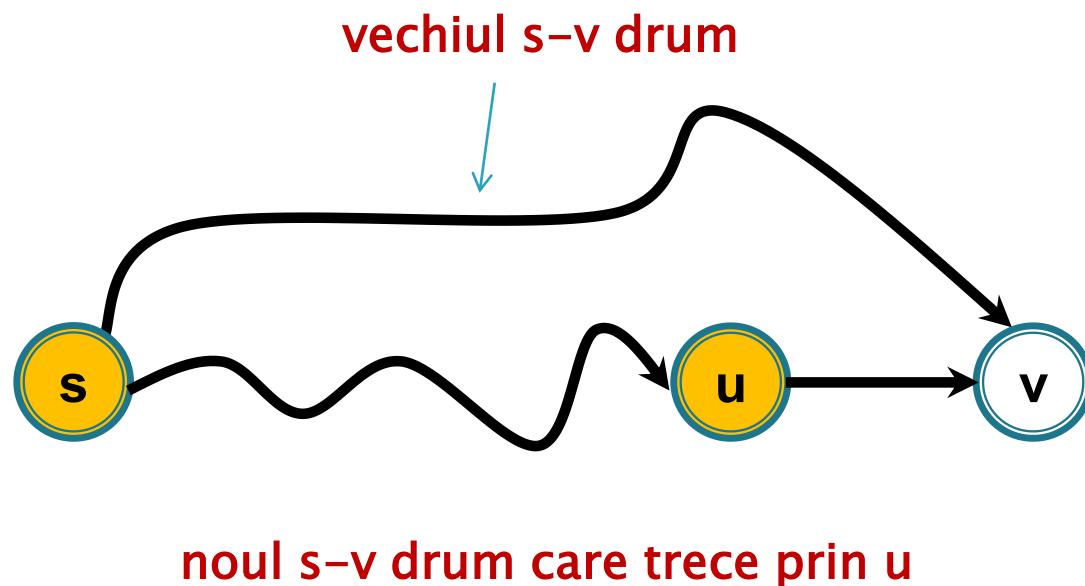
# Algoritmul lui Dijkstra

## ► 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  $(u, v) =$  a verifica dacă  $d[v]$  poate fi îmbunătățit trecând prin vârful  $u$



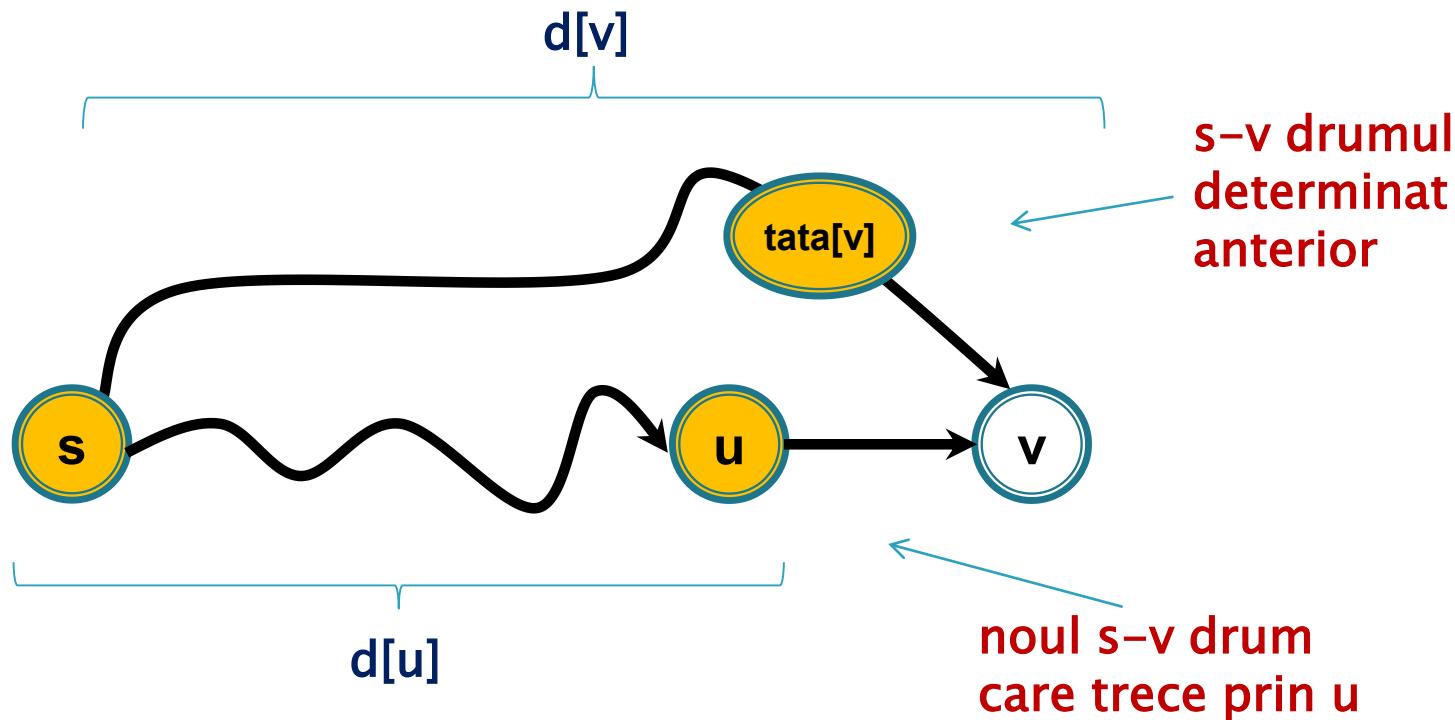
# Algoritmul lui Dijkstra

- Relaxarea unui arc  $(u, v)$  :

dacă  $d[u] + w(u, v) < d[v]$  atunci

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

$tata[v] = u$



# Algoritmul lui Dijkstra

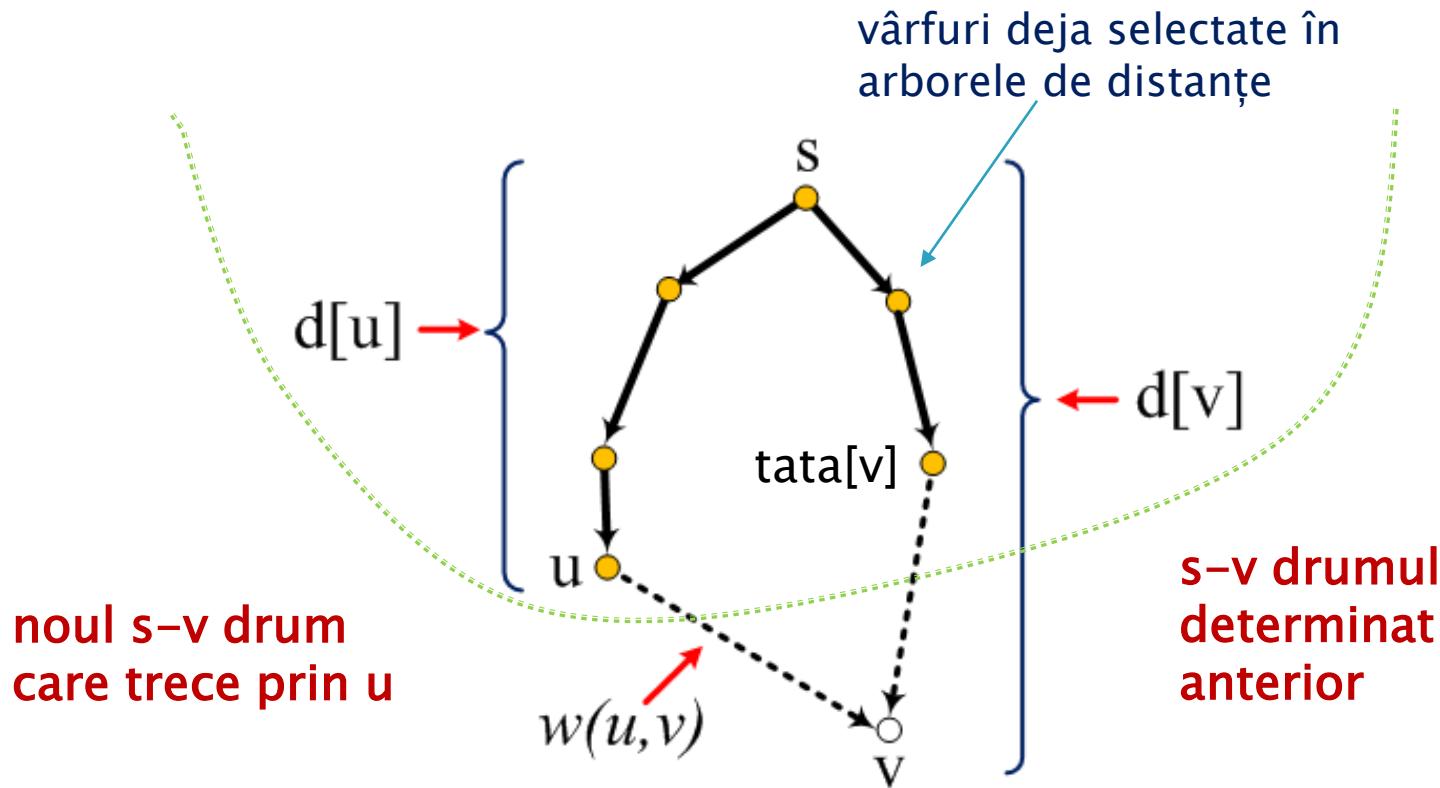
- Relaxarea unui arc  $(u, v)$  :

dacă  $d[u] + w(u, v) < d[v]$  atunci

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

$tata[v] = u$

Raportat la vîrfuri deja selectate – similar Prim



Dijkstra( $G, w, s$ )

initializează multimea  $vârfurilor neselectate Q$  cu  $V$

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

cat timp  $Q \neq \emptyset$  executa  $\Leftrightarrow$  pentru  $i=1, n$  executa

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

căt timp  $Q \neq \emptyset$  executa

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

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

cat timp  $Q \neq \emptyset$  executa

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

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

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

căt timp  $Q \neq \emptyset$  executa

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

pentru fiecare  $uv \in E$  executa

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

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

$tata[v] = u$

Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

cat timp  $Q \neq \emptyset$  executa

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

pentru fiecare  $uv \in E$  executa

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

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

tata[v] = u

scrie d, tata

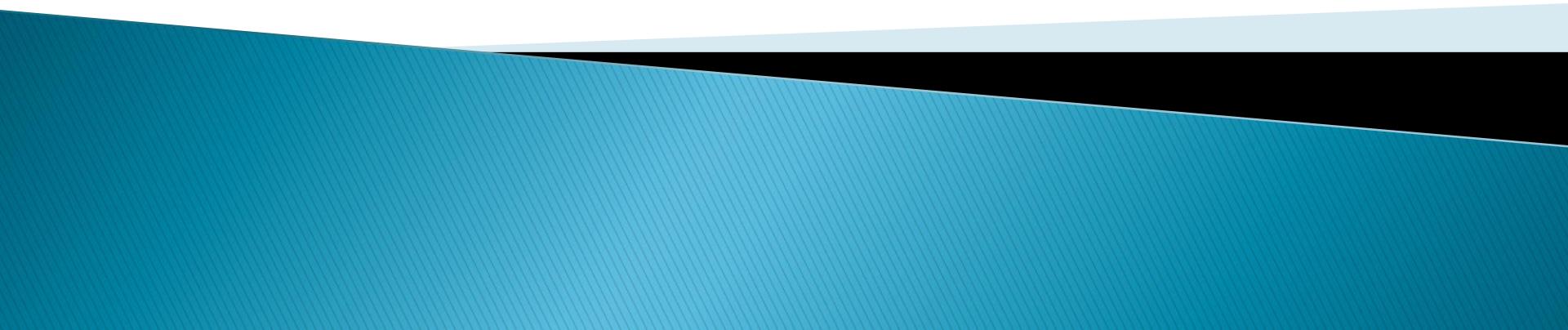
//scrie drum minim de la s la t un varf t dat folosind tata

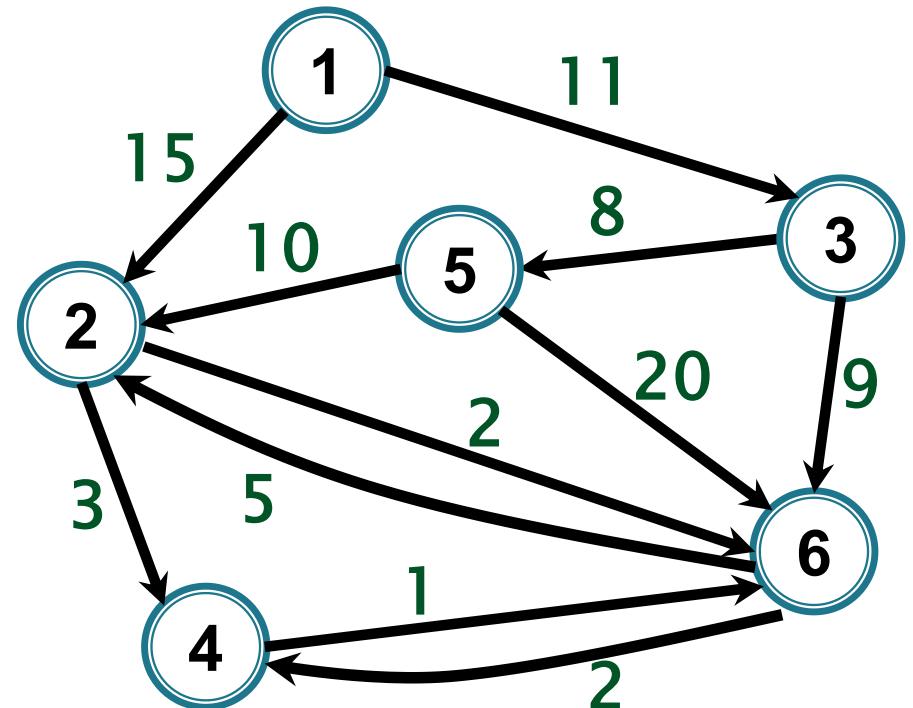
# Algoritmul lui Dijkstra

## ▶ Observație

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

# Exemplu





d/tata

[ 0/0,

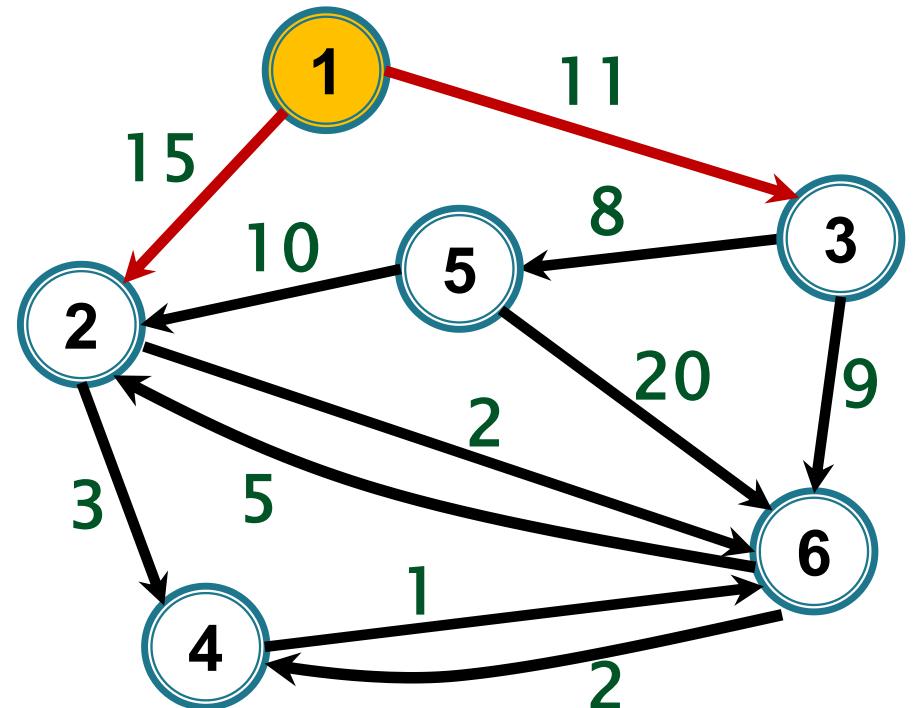
1  
∞/0,

2  
∞/0,

3  
∞/0,

4  
∞/0,

5  
∞/0,  
6  
∞/0 ]



d/tata

[ 1  
0/0,

2  
 $\infty/0$ ,

3  
 $\infty/0$ ,

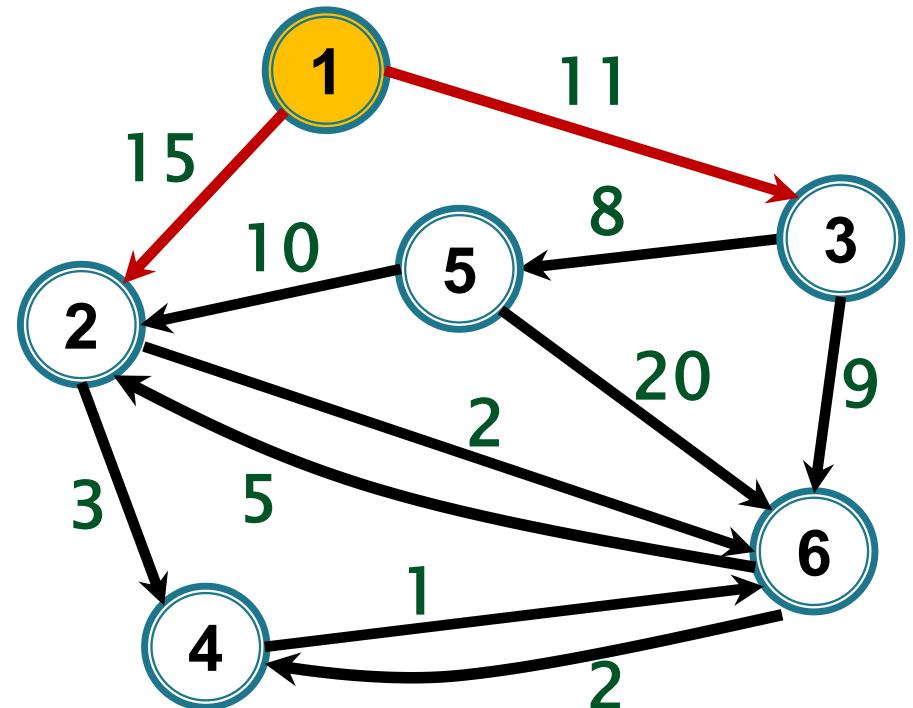
4  
 $\infty/0$ ,

5  
 $\infty/0$ ,

6  
 $\infty/0$  ]

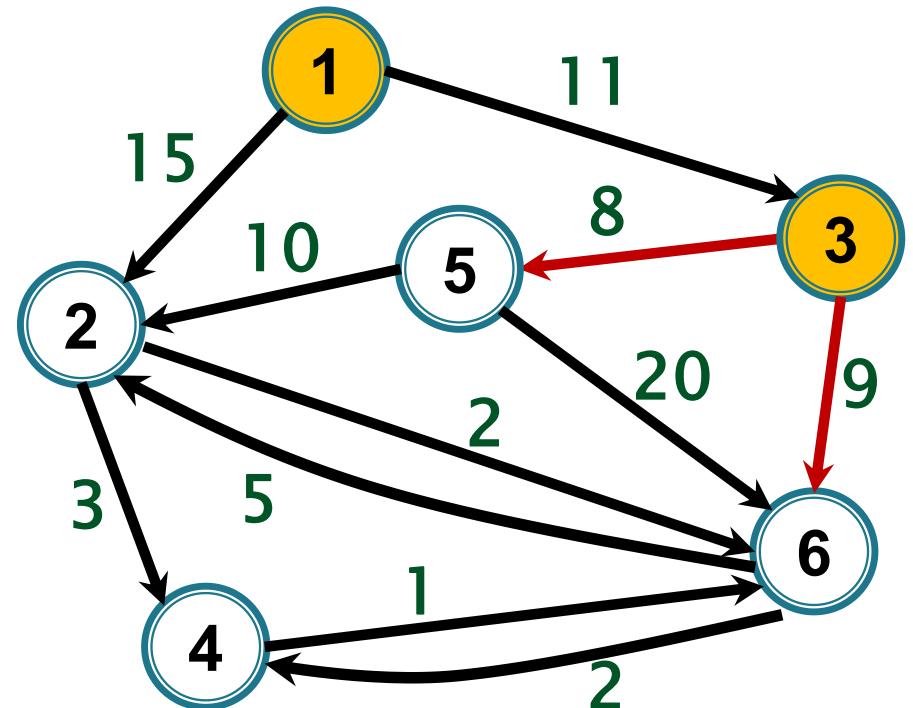
Sel. 1:

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



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

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



$d/tata$

[  $0/0$ ,    ]

$\infty/0$  ]

Sel. 1:

[ - ,

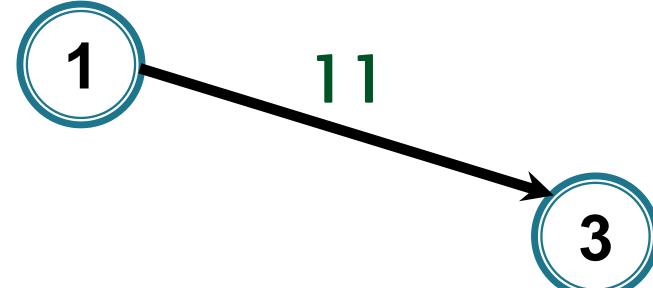
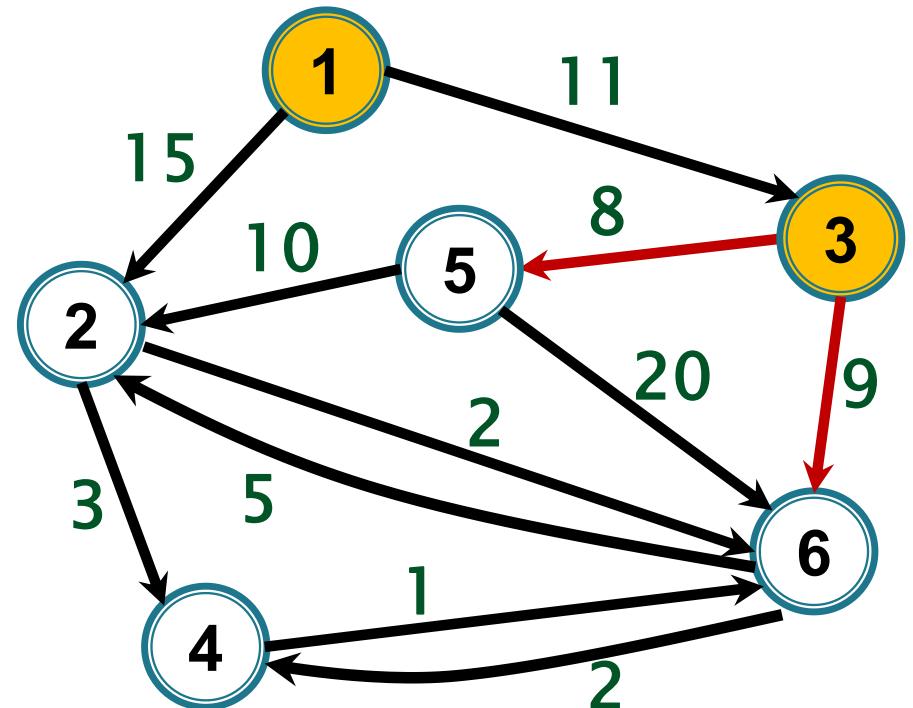
$15/1$ ,

$11/1$ ,

Sel. 3

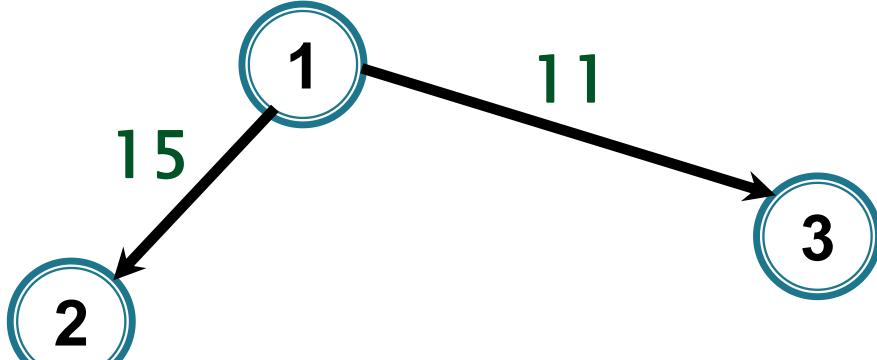
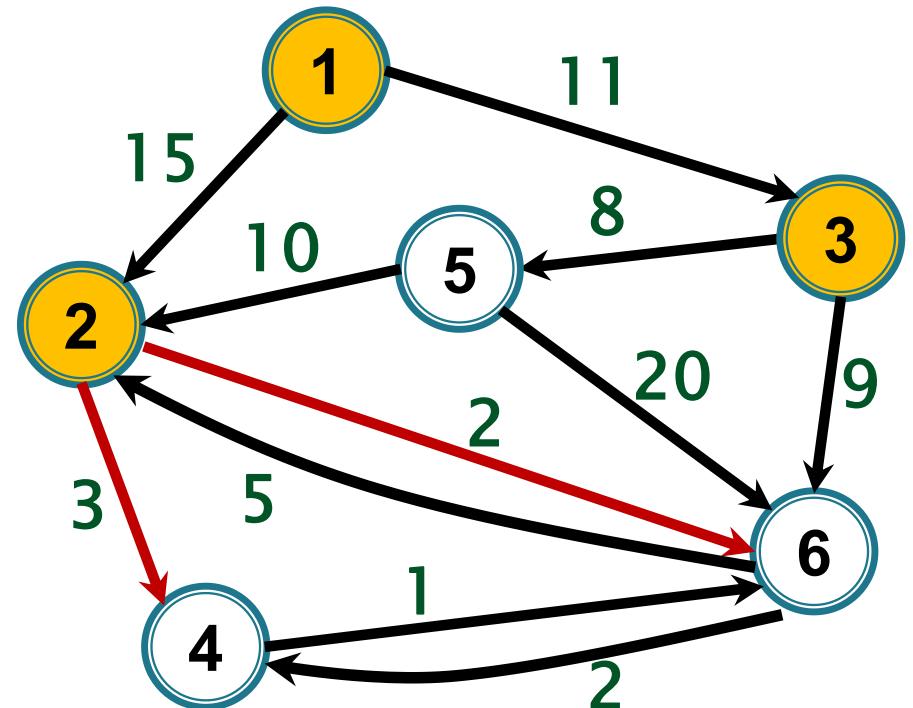
$$d[5] = \min\{d[5], d[3] + w(3, 5)\}$$

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



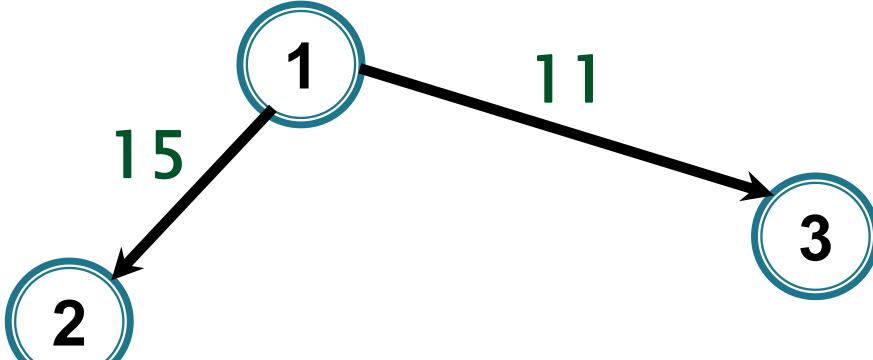
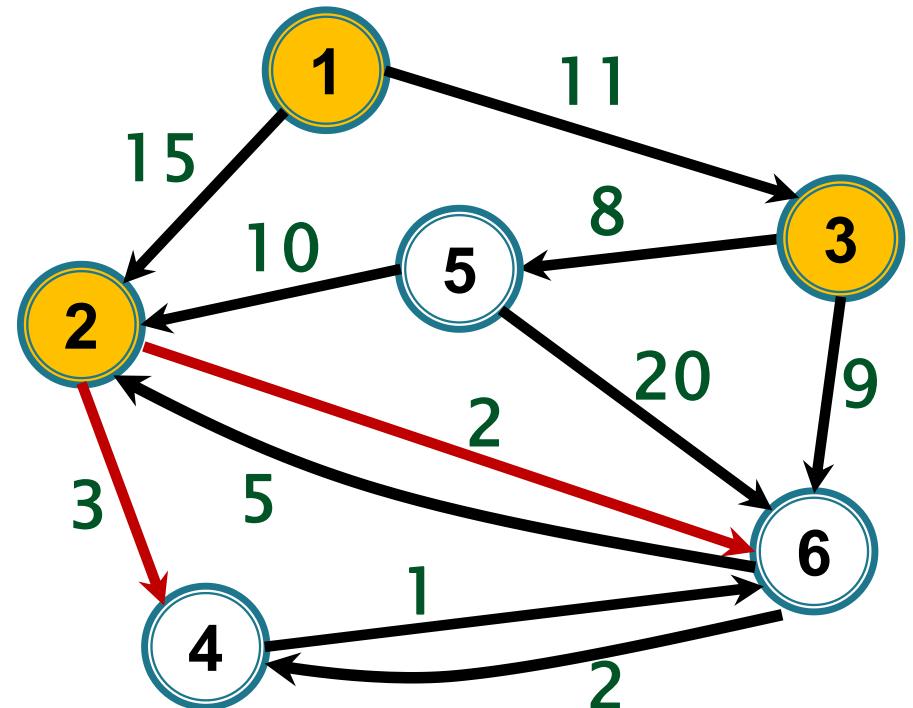
d/tata	1	2	3	4	5	6
Sel. 1:	[ <b>0/0</b> ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]
Sel. 3:	[ - ,	$15/1$ ,	<b>11/1</b> ,	- ,	$\infty/0$ ,	$19/3$ ,
						$20/3$ ]

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



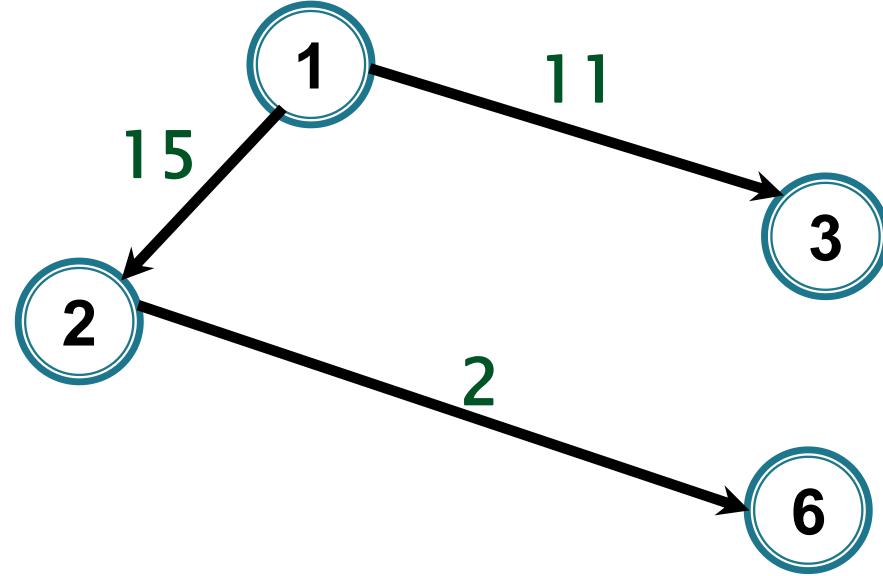
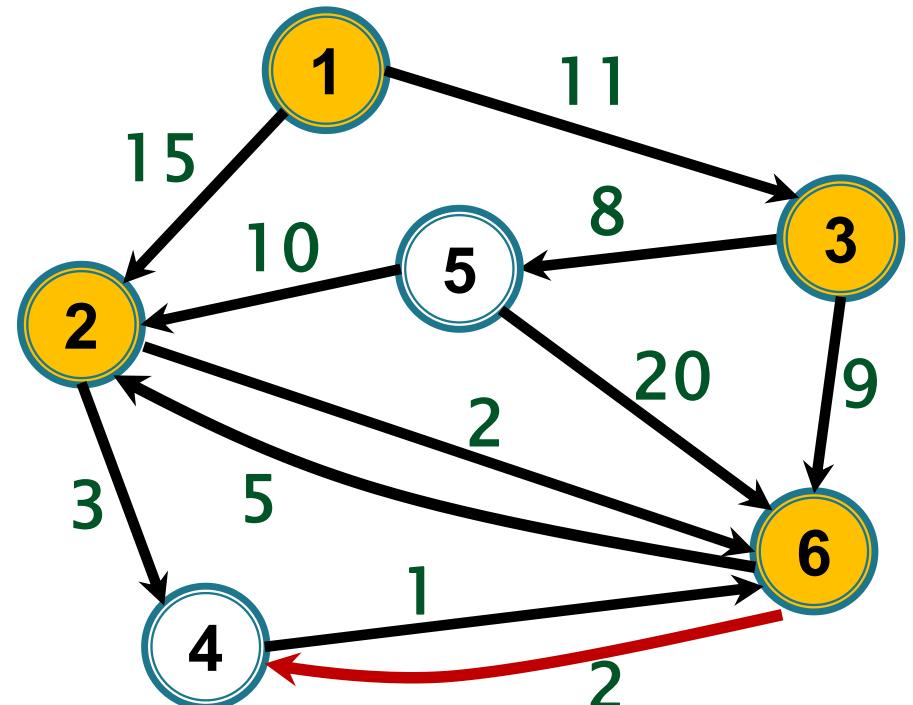
$d/tata$	1	2	3	4	5	6
Sel. 1:	[ $0/0$ ,	$\infty/0$ ]				
Sel. 3:	[ - ,	$15/1$ ,	$11/1$ ,	- ,	$\infty/0$ ,	$19/3$ ,
Sel. 2:						$20/3$ ]

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



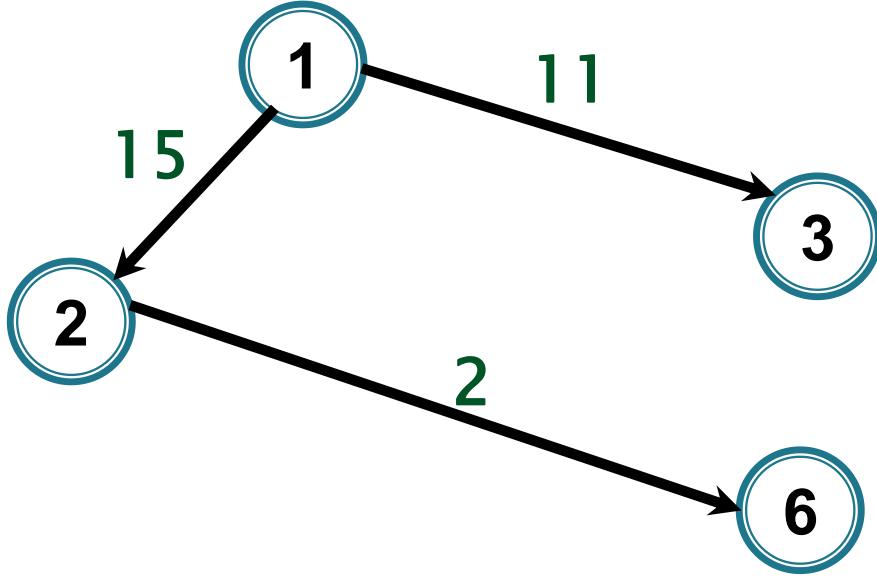
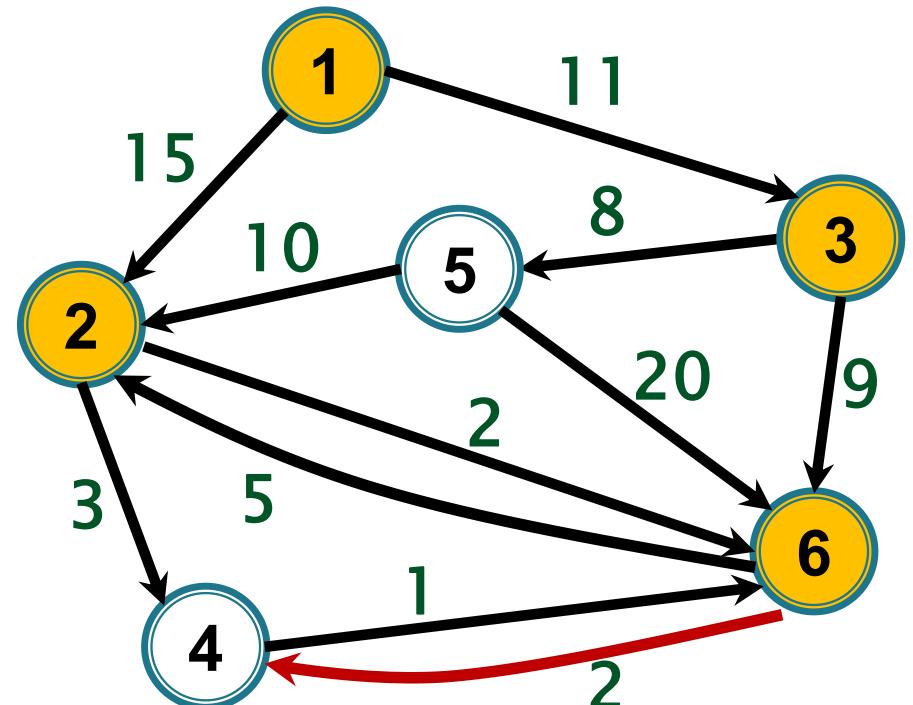
$d/tata$	1	2	3	4	5	6
Sel. 1:	[ <b>0/0</b> , - ]	$\infty/0$ , <b>15/1</b>	$\infty/0$ , <b>11/1</b>	$\infty/0$ , -	$\infty/0$ , <b>19/3</b>	$\infty/0$ , <b>20/3</b>
Sel. 3:	[ - , <b>15/1</b> ]	-	-	$\infty/0$ , <b>18/2</b>	$\infty/0$ , <b>19/3</b>	$\infty/0$ , <b>17/2</b>
Sel. 2:	[ - , - ]	-	-	-	-	-

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



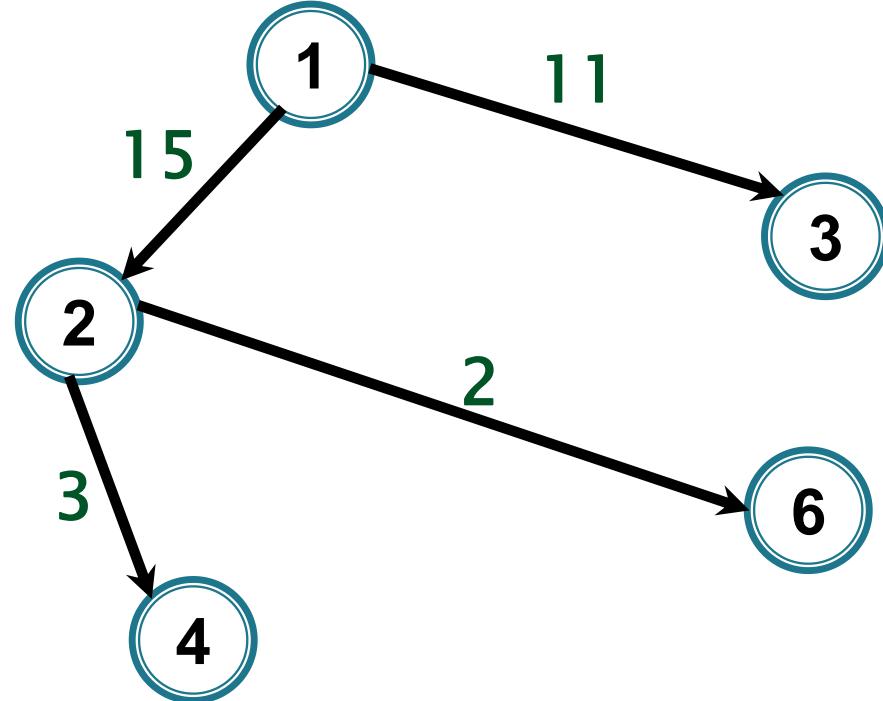
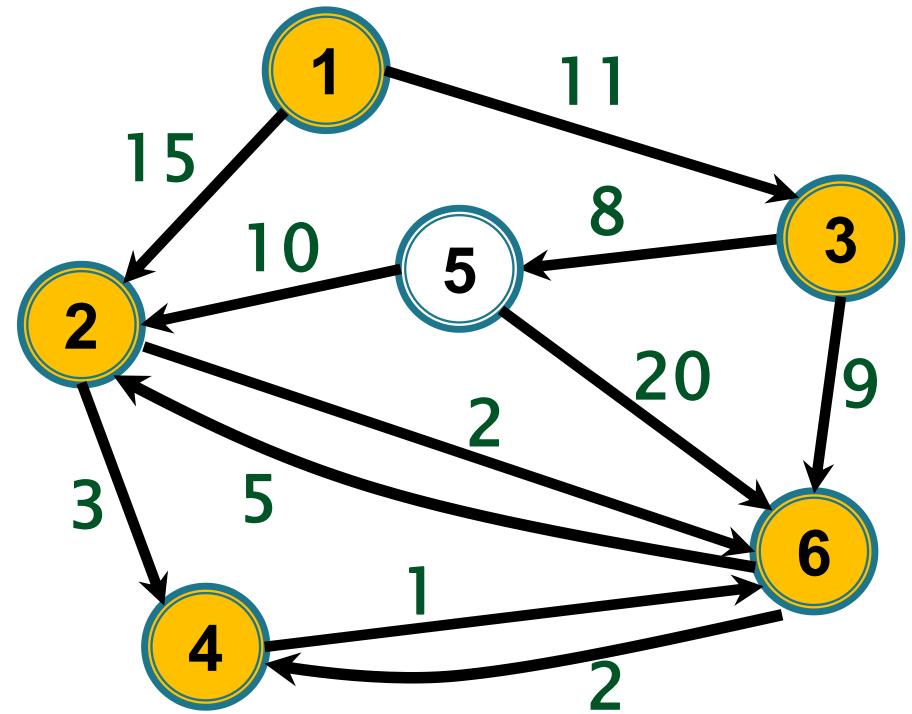
d/tata	1	2	3	4	5	6
	[ 0/0,	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 1:	[ - ,	15/1,	11/1,	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 3:	[ - ,	15/1,	- ,	$\infty/0,$	19/3,	20/3 ]
Sel. 2:	[ - ,	- ,	- ,	18/2,	19/3,	17/2 ]
Sel. 6:						

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



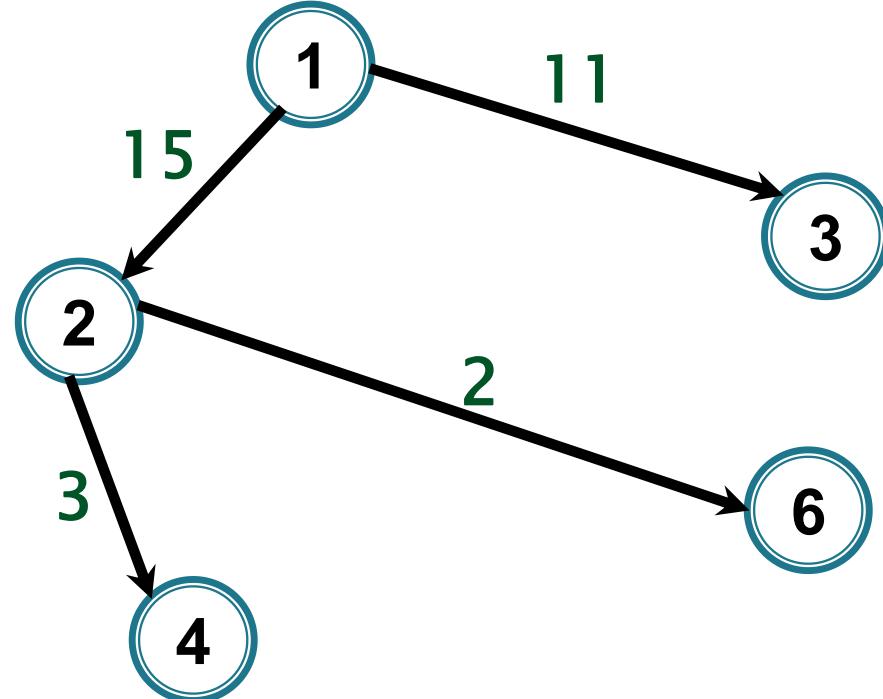
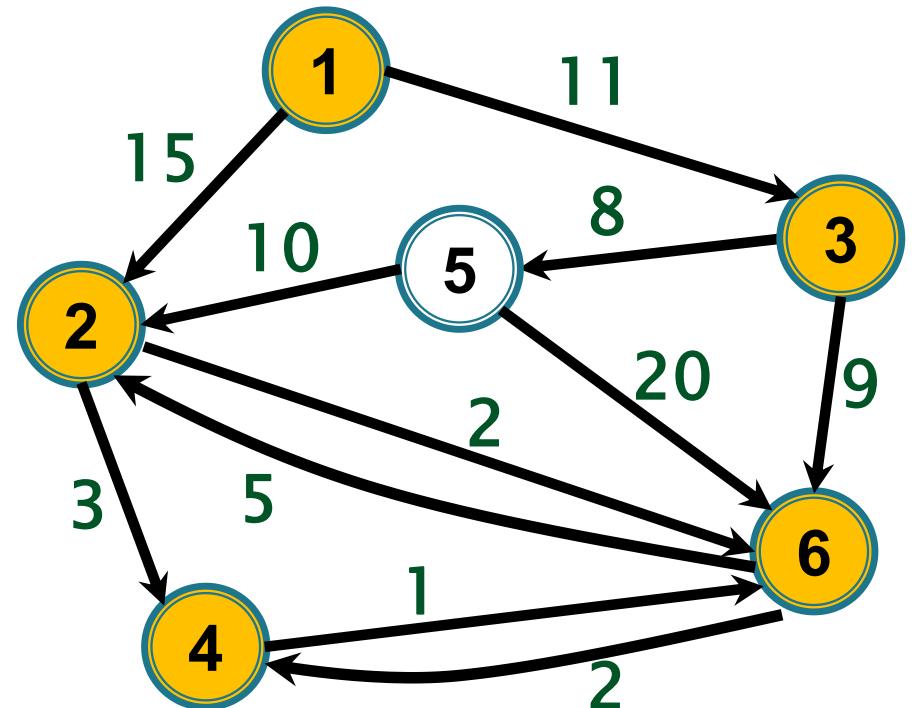
d/tata	1	2	3	4	5	6
	[ 0/0,	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 1:	[ - ,	15/1,	11/1,	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 3:	[ - ,	15/1,	- ,	$\infty/0,$	19/3,	20/3 ]
Sel. 2:	[ - ,	- ,	- ,	18/2,	19/3,	17/2 ]
Sel. 6:	[ - ,	- ,	- ,	18/2,	19/3,	- ]

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



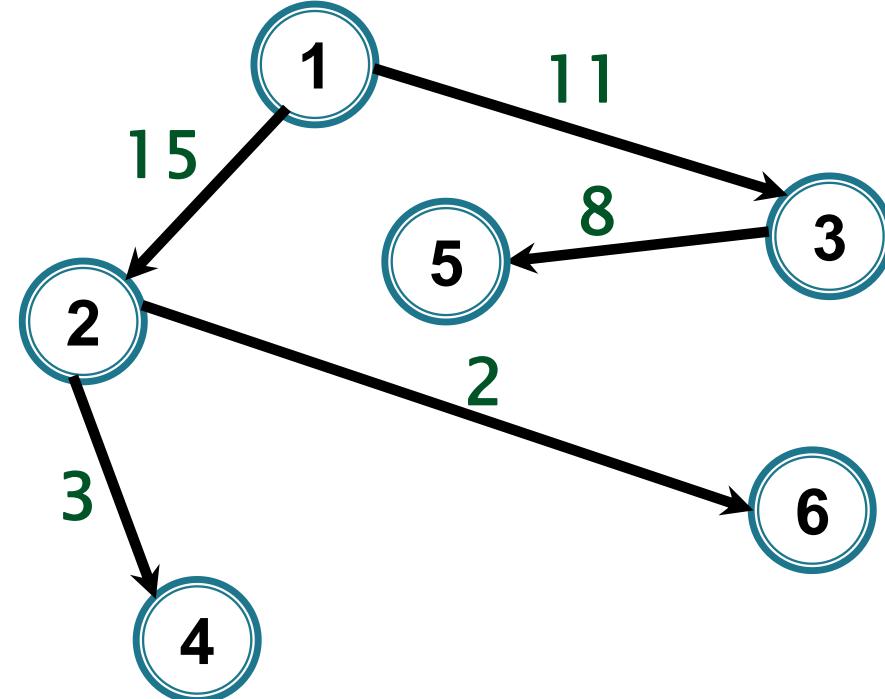
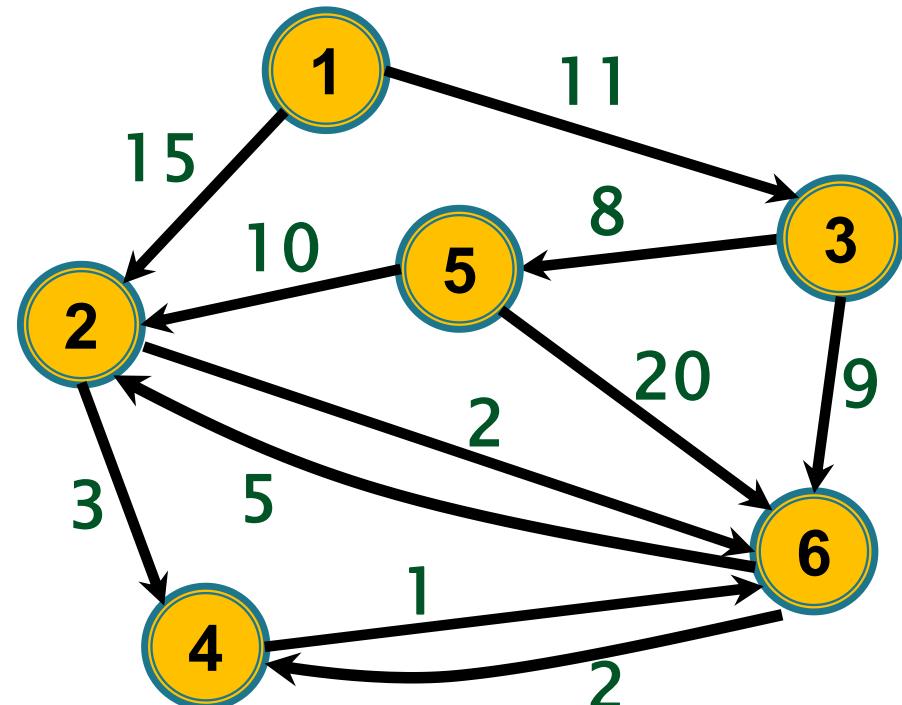
d/tata	1	2	3	4	5	6
	[ 0/0,	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 1:	[ - ,	15/1,	11/1,	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 3:	[ - ,	15/1,	- ,	$\infty/0,$	19/3,	20/3 ]
Sel. 2:	[ - ,	- ,	- ,	18/2,	19/3,	17/2 ]
Sel. 6:	[ - ,	- ,	- ,	18/2,	19/3,	- ]
Sel. 4:						

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



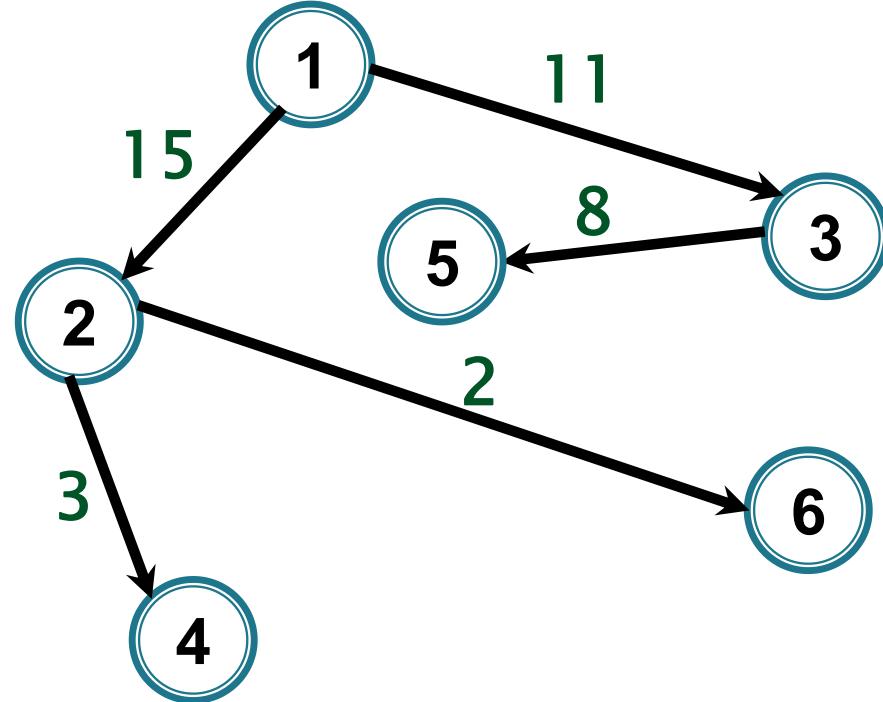
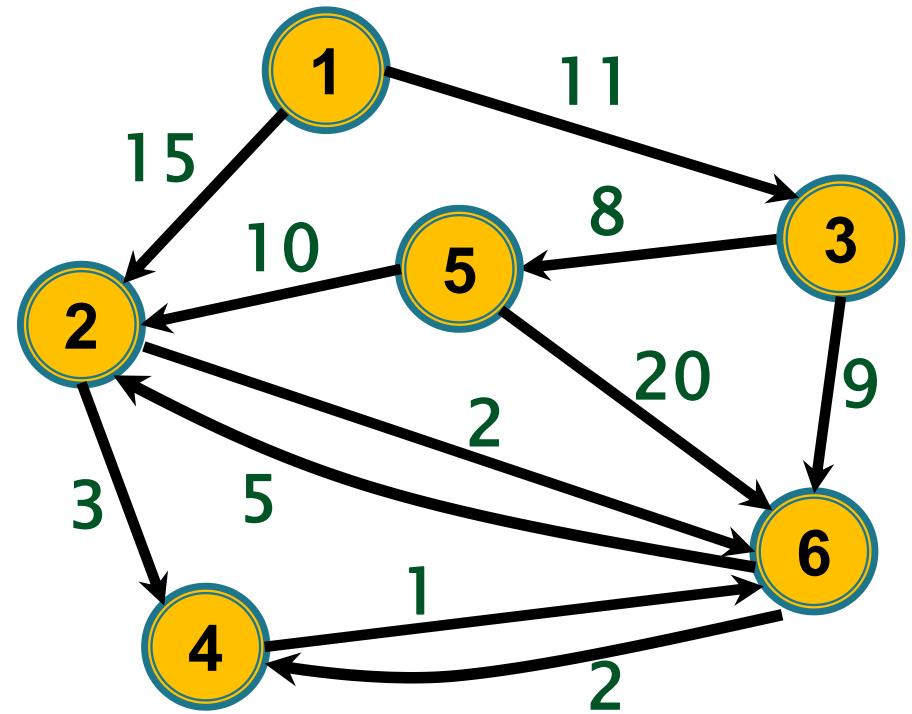
d/tata	1	2	3	4	5	6
	[ 0/0,	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 1:	[ - ,	15/1,	11/1,	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 3:	[ - ,	15/1,	- ,	$\infty/0,$	19/3,	20/3 ]
Sel. 2:	[ - ,	- ,	- ,	18/2,	19/3,	17/2 ]
Sel. 6:	[ - ,	- ,	- ,	18/2,	19/3,	- ]
Sel. 4:	[ - ,	- ,	- ,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

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

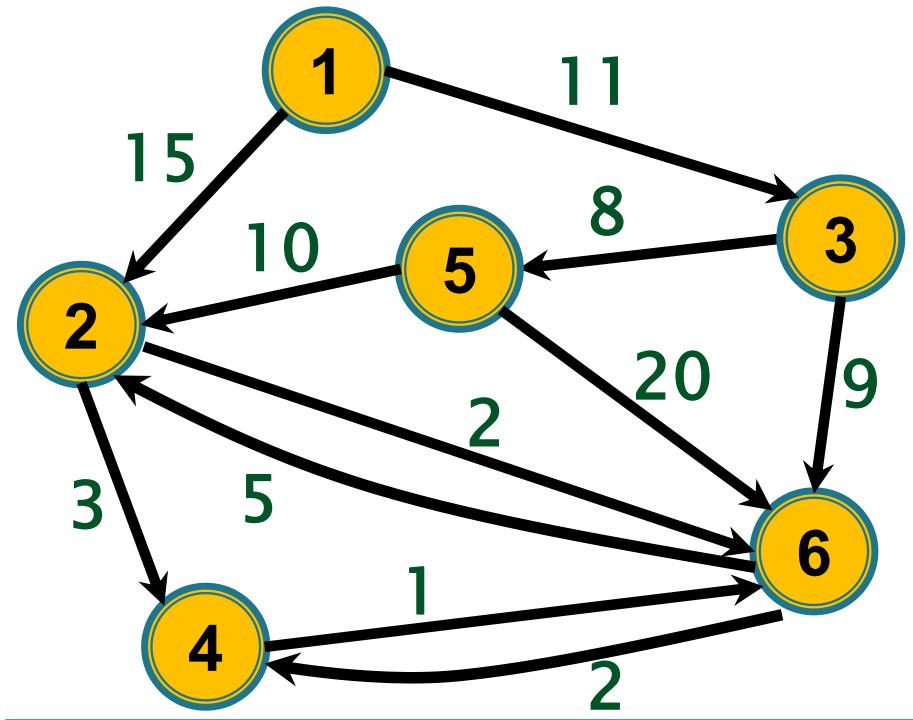


d/tata	1	2	3	4	5	6
	[ 0/0,	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 1:	[ - ,	15/1,	11/1,	$\infty/0,$	$\infty/0,$	$\infty/0 ]$
Sel. 3:	[ - ,	15/1,	- ,	$\infty/0,$	19/3,	20/3 ]
Sel. 2:	[ - ,	- ,	- ,	18/2,	19/3,	17/2 ]
Sel. 6:	[ - ,	- ,	- ,	18/2,	19/3,	- ]
Sel. 4:	[ - ,	- ,	- ,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

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



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

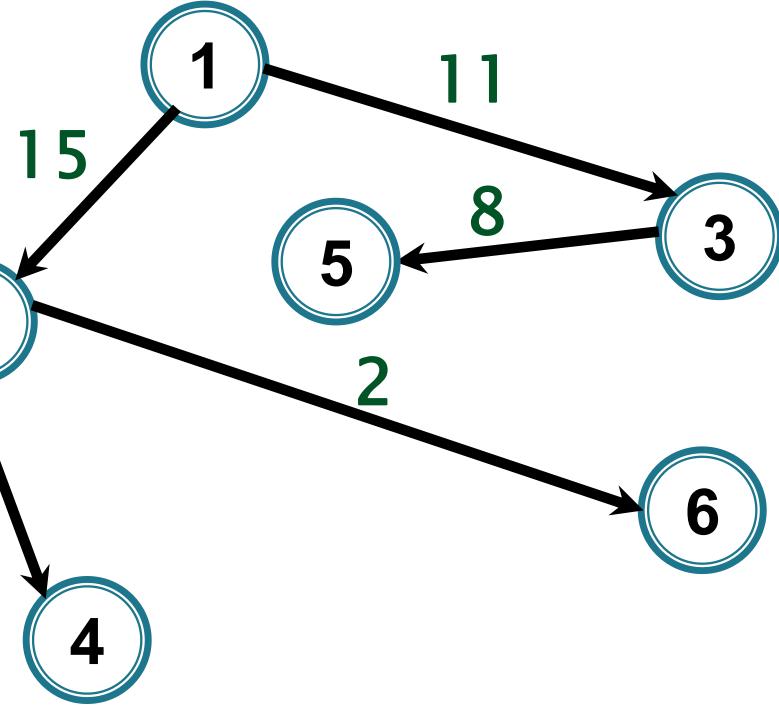


d/tata

1

2

3



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

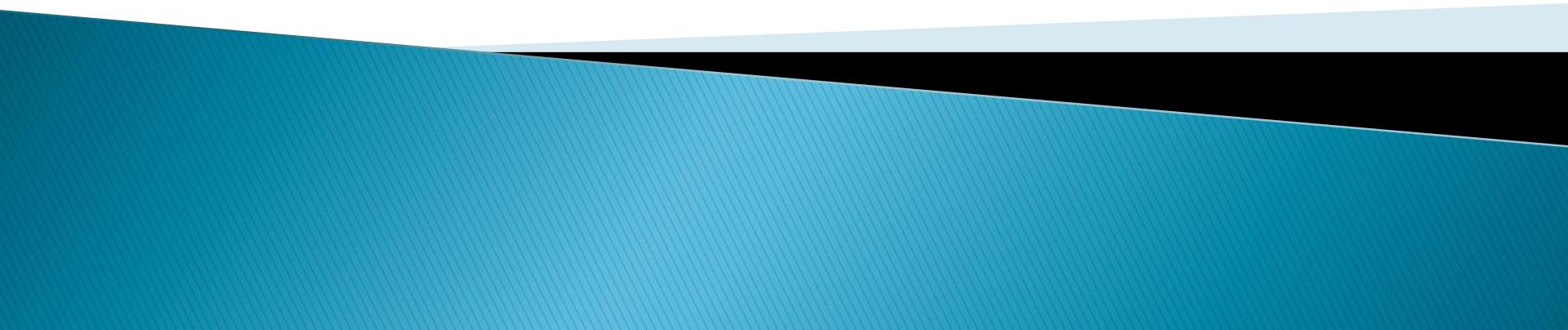
Un drum minim de la 1 la 6?

# Algoritmul lui Dijkstra

## ▶ Observații.

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

# Complexitate



Dijkstra(G, w, s)

initializează multimea vârfurilor neselectate Q cu V  
pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

cat timp  $Q \neq \emptyset$  executa

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

pentru fiecare  $uv \in E$  executa

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

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

tata[v] = u

scrie d, tata

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

# Algoritmul lui Dijkstra



**Cum memorăm  $Q =$  vârfurile încă neselectate?**

# Algoritmul lui Dijkstra

**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** - reprezentarea lui Q ca vector

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

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

# Algoritmul lui Dijkstra

# **Complexitate – reprezentarea lui Q ca vector**

$Q[u] = 1$ , dacă  $u$  este selectat în  $V(T)$   
 $0$ , altfel ( $u \in Q$ )

- ▶ Inițializare Q → O(n)
  - ▶  $n * \text{extragere vârf minim}$  →
  - ▶ actualizare etichete vecini →

# Algoritmul lui Dijkstra

**Complexitate** – reprezentarea lui Q ca vector

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

- ▶ Inițializare Q  $\rightarrow O(n)$
- ▶  $n * \text{extragere vârf minim}$   $\rightarrow O(n^2)$
- ▶ actualizare etichete vecini  $\rightarrow$  

---

# Algoritmul lui Dijkstra

**Complexitate** - reprezentarea lui Q ca vector

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

- ▶ Inițializare Q  $\rightarrow O(n)$
  - ▶  $n * \text{extragere vârf minim}$   $\rightarrow O(n^2)$
  - ▶ actualizare etichete vecini  $\rightarrow O(m)$
-

# Algoritmul lui Dijkstra

**Complexitate** - reprezentarea lui Q ca vector

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

- ▶ Inițializare Q -> O(n)
  - ▶  $n * \text{extragere vârf minim}$  -> O( $n^2$ )
  - ▶ actualizare etichete vecini -> O(m)
- 
- $O(n^2)$

# Algoritmul lui Dijkstra

**Complexitate** – Q min-heap

- ▶ Inițializare Q →
  - ▶  $n * \text{extragere vârf minim}$  →
  - ▶ actualizare etichete vecini →
-

Dijkstra(G, w, s) - Q min-heap in raport cu d

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

$Q = V$  //creare heap cu cheile din d

cat timp  $Q \neq \emptyset$  executa

$u = \text{extrage\_min}(Q)$

pentru fiecare  $uv \in E$  executa

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

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

**repara( $Q, v$ )**

$tata[v] = u$

scrie d, tata

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

# Algoritmul lui Dijkstra

**Complexitate** – Q min-heap

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

# Algoritmul lui Dijkstra

**Complexitate** – Q min-heap

- ▶ Inițializare Q →  $O(n)$
  - ▶  $n * \text{extragere vârf minim}$  →  $O(n \log n)$
  - ▶ 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  $\approx$  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 executa
    pentru fiecare uv∈E executa
        daca 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 (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 executa
    pentru fiecare uv∈E executa
        daca d[u]+w(u,v)< d[v] atunci
            d[v] = d[u]+w(u,v)
            tata[v] = u
```

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

# **Corectitudinea Algoritmului lui Dijkstra**

# Corectitudine

- ▶ **Lema 1.** Pentru orice  $u \in V$ , la orice pas al algoritmului lui Dijkstra avem:
  - a) 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:  
 $tata[u] = \text{predecesorul lui } u \text{ pe un drum de la } s \text{ la } u \text{ de cost } d[u]$
  - b)  $d[u] \geq \delta(s, u)$

# Corectitudine

- ▶ **Lema 1.** Pentru orice  $u \in V$ , la orice pas al algoritmului lui Dijkstra avem:
  - a) 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:  
 $tata[u] = \text{predecesorul lui } u \text{ pe un drum de la } s \text{ la } u \text{ de cost } d[u]$
  - b)  $d[u] \geq \delta(s, u)$
- ▶ **Consecință.** Dacă la un pas al algoritmului avem pentru un vîrf  $u$  relația  $d[u] = \delta(s, u)$ , atunci  $d[u]$  nu se mai modifică până la final.

# Corectitudine

## ► Teoremă

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

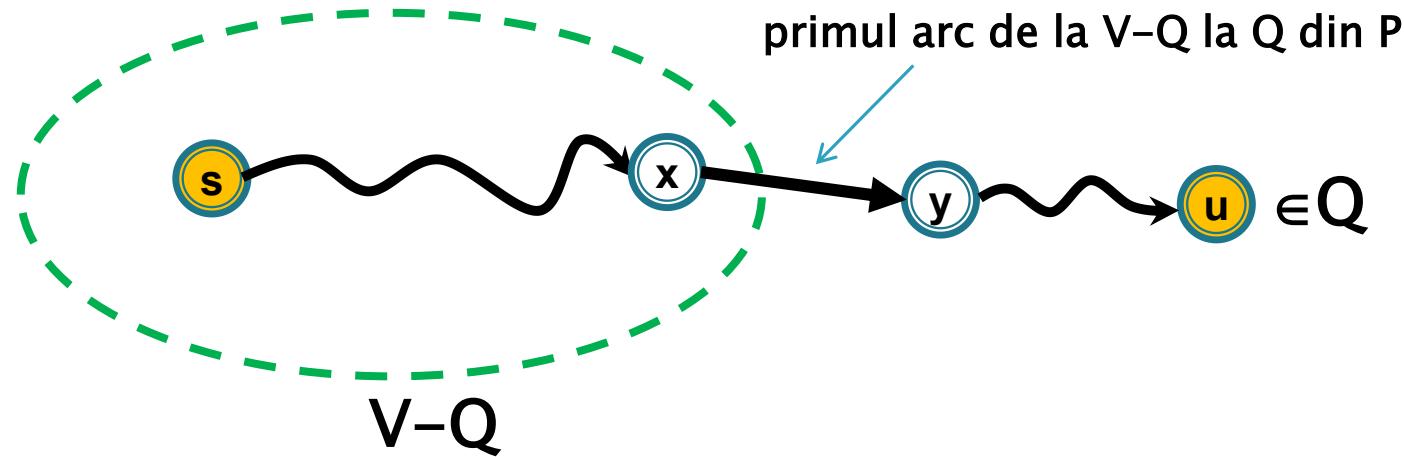
La finalul algoritmului lui Dijkstra avem:

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

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

# Corectitudine

- ▶ Demonstrație (idee). Inducție:  $d[x] = \delta(s, x) \quad \forall x \notin Q$  (=deja selectat)
- ▶ Când un vârf  $u$  este selectat: fie  $P$  un  $s-u$  drum minim



din modul în care este ales  $u$

$$d[u] \leq d[y] \leq d[x] + w(x, y) = \delta(s, x) + w(x, y) = w(s - y) = \delta(s, y)$$

$$\leq w(P) \leq d[u]$$

dupa relaxarea lui  $xy$  (mai mult, are loc chiar egalitate:  $d[y] = \delta(s, x) + w(x, y) = w(s - y) = \delta(s, y)$ )

$$w(s - y) = \delta(s, y) \quad \text{ipoteza de inducție pentru } x$$

$$\Rightarrow d[u] = d[y] = w(P) = \delta(s, u)$$



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

# Algoritmul lui BELLMAN – FORD

## ► Ipoteză:

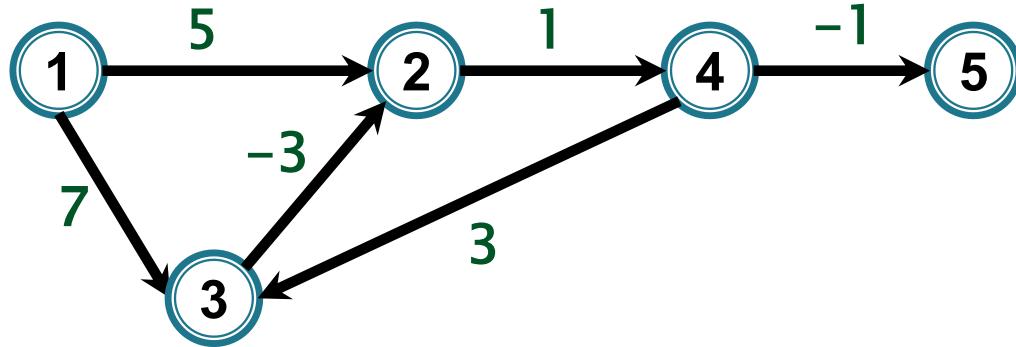
Arcele pot avea și cost negativ

Graful nu conține circuite de cost negativ

(dacă există – algoritmul le va detecta => nu soluție)

# **Algoritmul Bellman Ford**

# Algoritmul lui BELLMAN – FORD

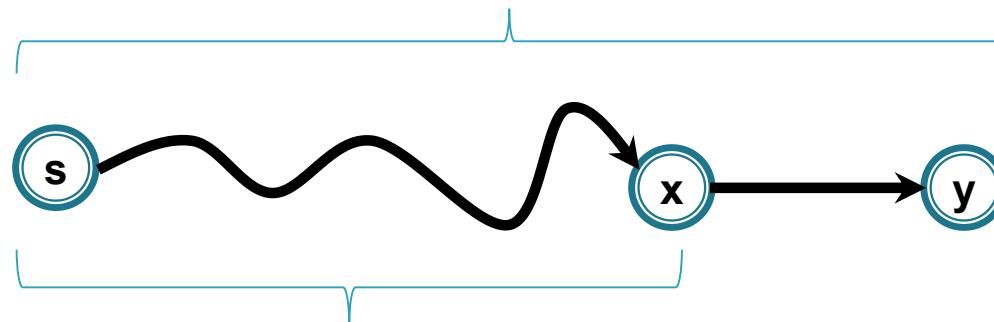


Algoritmul lui Dijkstra – doar pentru ponderi nenegative

# Algoritmul lui BELLMAN – FORD

- ▶ Idee: La un pas relaxăm toate arcele
- ▶ De câte ori?

s–y drum minim cu  $\leq k$  arce

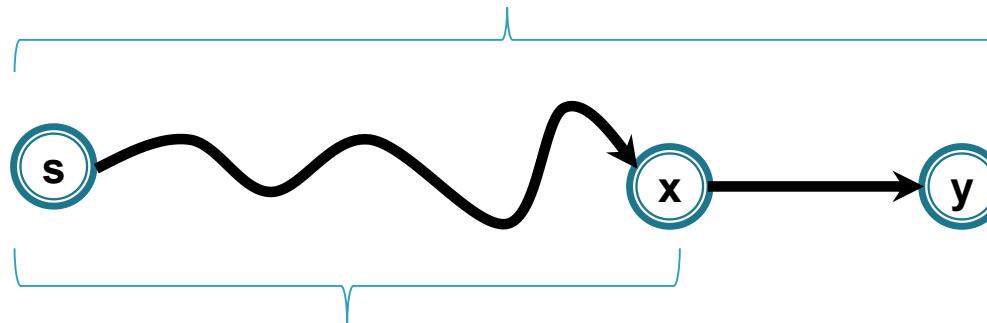


s–x drum minim cu  $\leq k-1$  arce

# Algoritmul lui 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

Dacă  $P$  este  $s-y$  drum minim cu  $\leq k$  arce  $\Rightarrow$

$[s \underline{P} x]$  este  $s-x$  drum minim cu  $\leq k-1$  arce

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

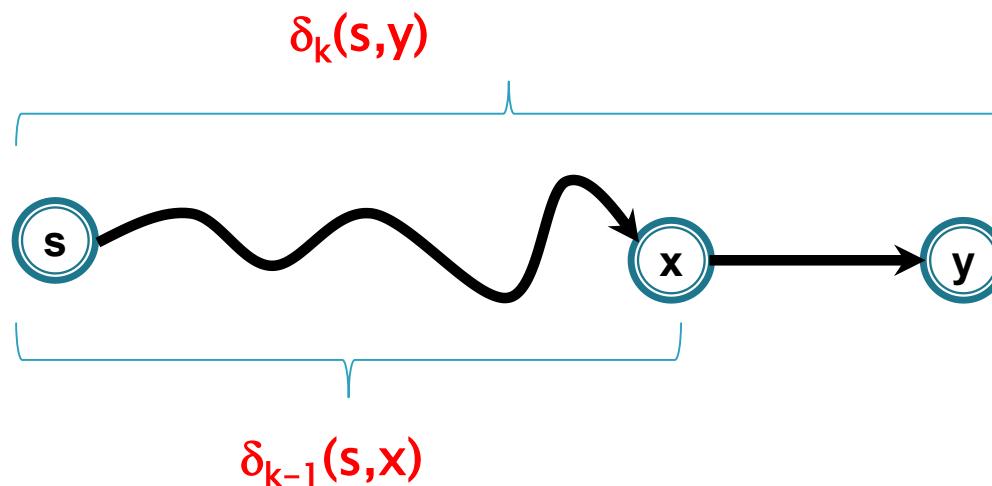
# Algoritmul lui BELLMAN – FORD

- ▶ 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 acele vârfuri  $u$  pentru care **există un  $s-u$  drum minim cu cel mult  $k$  arce**

# Algoritmul lui BELLMAN – FORD

▶ Notăm

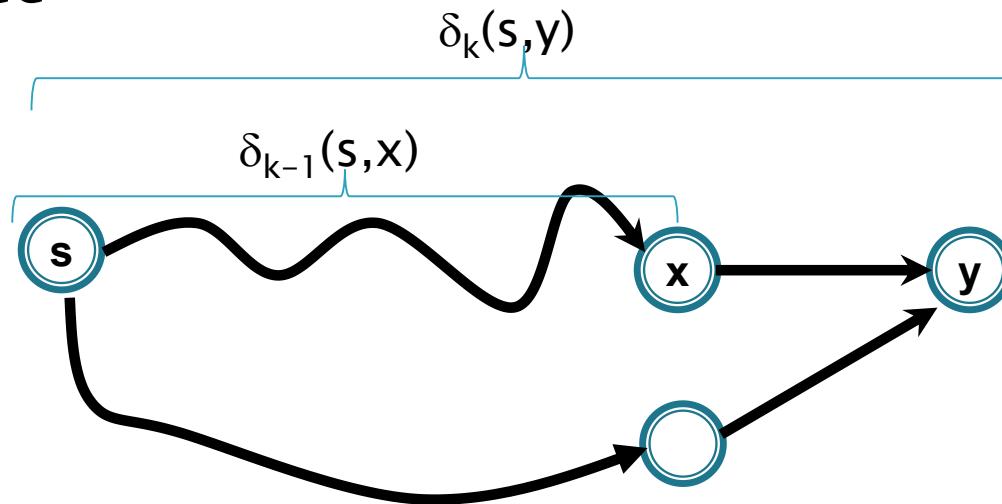
$\delta_k(s,x) = \text{costul minim al unui } s-x \text{ drum cu cel mult } k \text{ arce}$



# Algoritmul lui 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 lui BELLMAN – FORD

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

pentru  $i = 1, n-1$  executa

pentru fiecare  $uv \in E$  executa

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

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

$tata[v] = u$

# Algoritmul lui BELLMAN – FORD

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

pentru  $i = 1, n-1$  executa

pentru fiecare  $uv \in E$  executa

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

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

$tata[v] = u$

**Complexitate:**  $O(nm)$

# Algoritmul lui BELLMAN – FORD

- ▶ Optimizări

La un pas este suficient să relaxăm arcele din vârfuri ale căror etichetă s-a modificat anterior=>

# Algoritmul lui BELLMAN – FORD

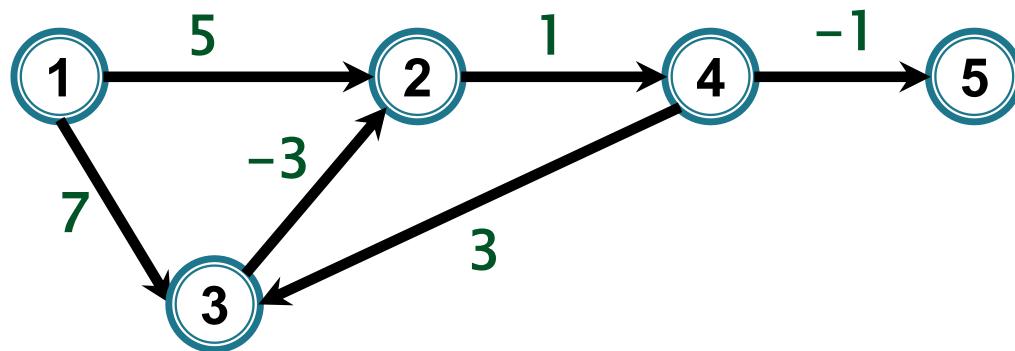
## ▶ Optimizări

**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

# Etapa 1

Relaxăm

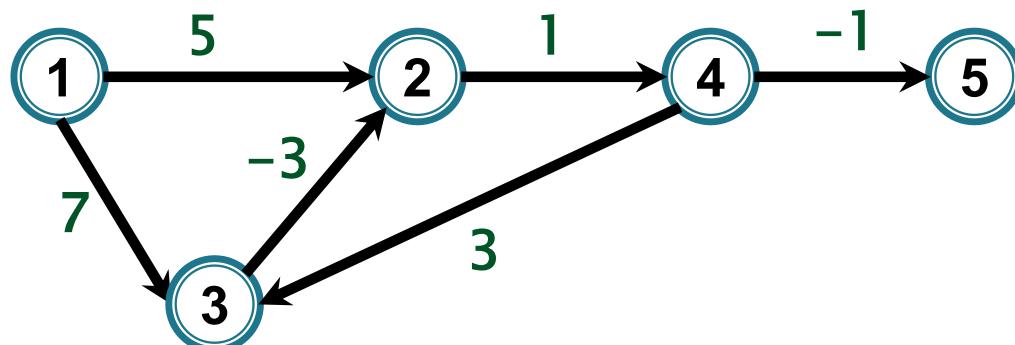


1 2  
1 3

	1	2	3	4	5
d	0	5	7	$\infty$	$\infty$

# Etapa 1

Relaxăm

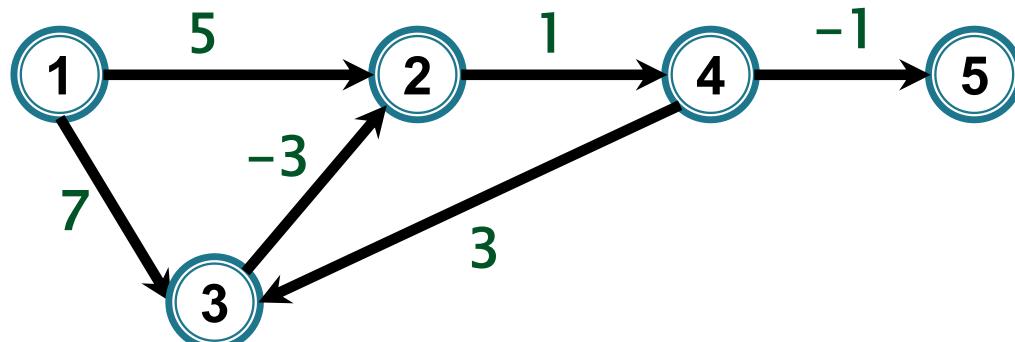


1 2  
1 3  
2 4

	1	2	3	4	5
d	0	5	7	6	$\infty$

# Etapa 1

Relaxăm

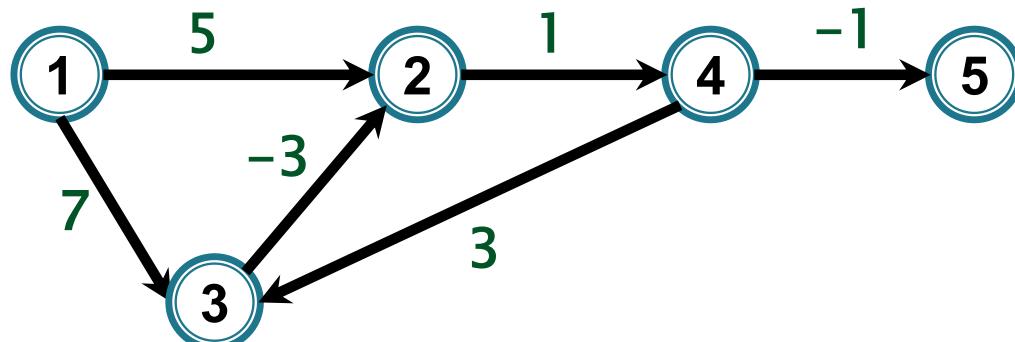


1 2  
1 3  
2 4  
4 3  
4 5

	1	2	3	4	5
d	0	5	7	6	5

# Etapa 1

Relaxăm

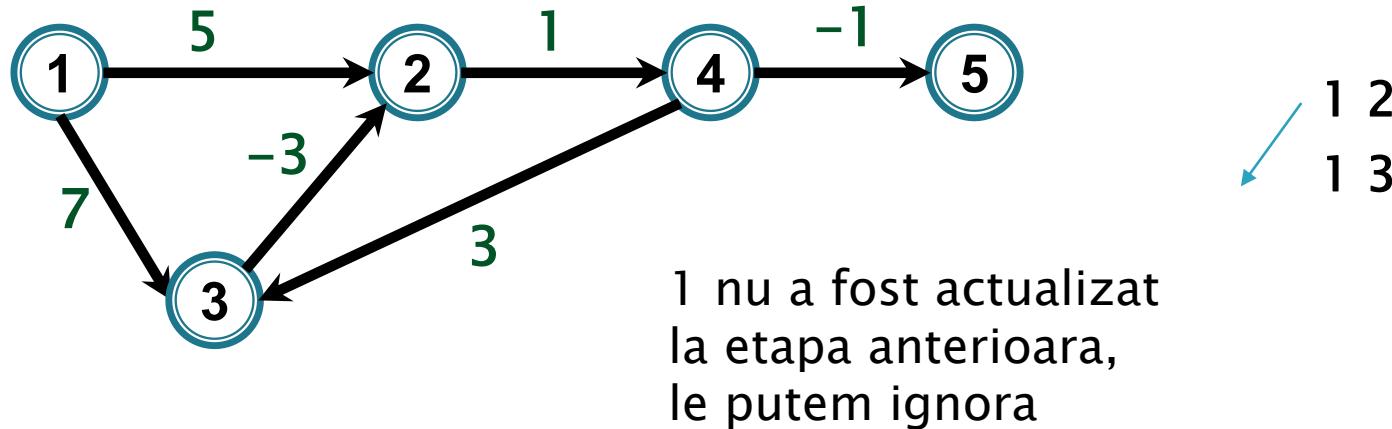


1 2  
1 3  
2 4  
4 3  
4 5  
3 2

	1	2	3	4	5
d	0	4	7	6	5

## Etapa 2

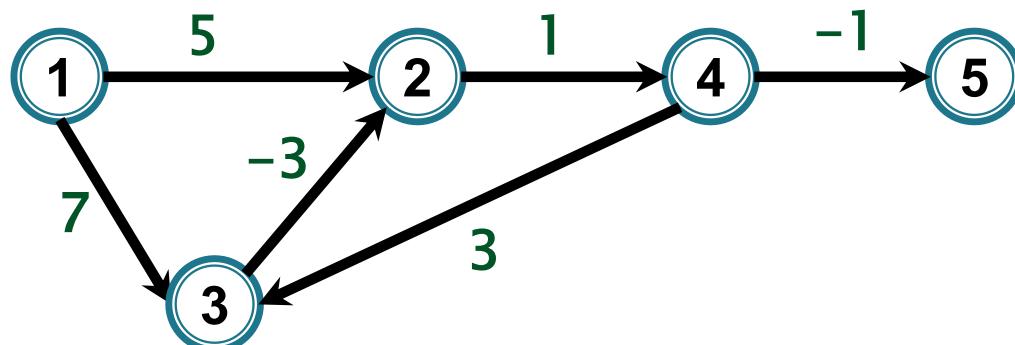
Relaxăm



	1	2	3	4	5
d	0	4	7	6	5

## Etapa 2

Relaxăm

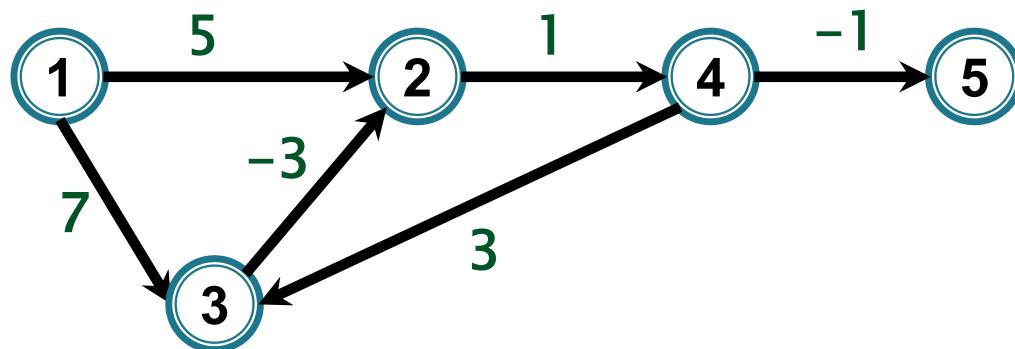


1 2  
1 3  
2 4

	1	2	3	4	5
d	0	4	7	5	5

## Etapa 2

Relaxăm

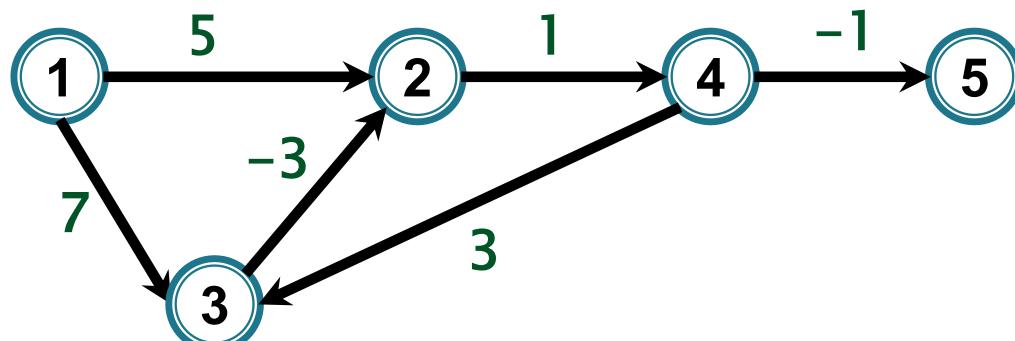


1 2  
1 3  
2 4  
4 3

	1	2	3	4	5
d	0	4	7	5	5

## Etapa 2

Relaxăm

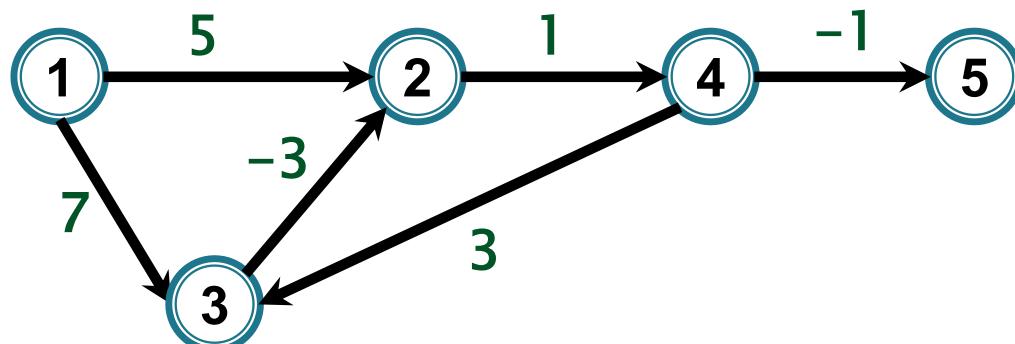


1 2  
1 3  
2 4  
4 3  
4 5

	1	2	3	4	5
d	0	4	7	5	4

## Etapa 2

Relaxăm

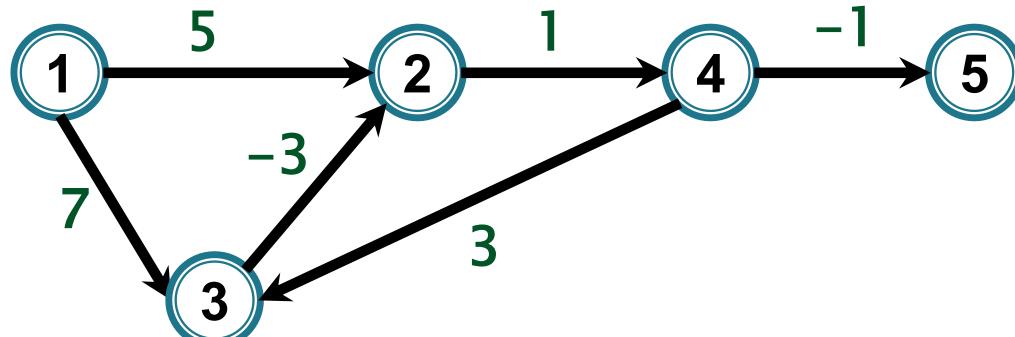


1 2  
1 3  
2 4  
4 3  
4 5  
3 2

	1	2	3	4	5
d	0	4	7	5	4

## Etapa 3

Relaxăm



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

Nu se mai actualizează nimic – stop

	1	2	3	4	5
d	0	4	7	5	4

# **Corectitudinea Algoritmului lui Bellman–Ford**

# Corectitudine

- ▶ **Lema 1 (comună).** Pentru orice  $u \in V$ , la orice pas al algoritmului avem:
  - a) 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:  
 $tata[u] =$  predecesorul lui  $u$  pe un drum de la  $s$  la  $u$  de cost  $d[u]$
  - b)  $d[u] \geq \delta(s, u)$

# Corectitudine

- ▶ **Demonstrație:** Inducție după numărul de etape (o etapa = relaxarea tuturor muchiilor):

După k iterării

$$d[x] \leq \delta_k(s, x)$$

= 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 iterată  $k$   
 $d[x] \leq \delta_{k-1}(s, x)$  pentru orice  $x$

Etiqueta unui vârf  $y$  la iterată  $k$  se actualizează astfel:

# Corectitudine

ipoteza de inducție

$$d[y] \leq \min\{d[y], \min\{d[x] + w(x,y) \mid xy \in E\} \leq$$



# Corectitudine

ipoteza de inducție

$$d[y] \leq \min\{d[y], \min\{d[x] + w(x,y) \mid xy \in E\} \leq$$

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

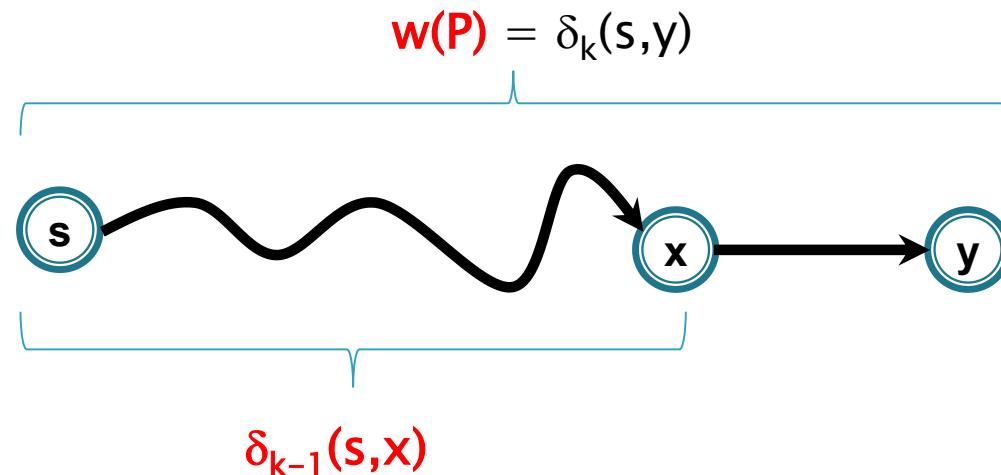
$$= \delta_k(s,y)$$

# Corectitudine

►  $k-1 \Rightarrow k$ . Presupunem că înainte de iterată  $k$

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

**Varianta 2.** Fie  $P$  un  $s-y$  drum cu cost minim printre cele cu cel mult  $k$  arce ( $w(P) = \delta_k(s, y)$  )



# Corectitudine

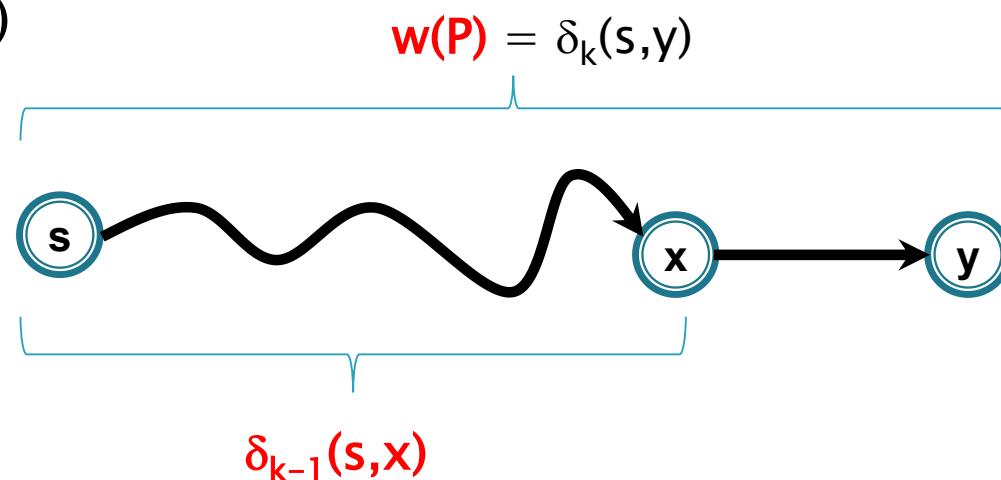
►  $k-1 \Rightarrow k$ . Presupunem că înainte de iterată  $k$

$$d[x] \leq \delta_{k-1}(s, x) \text{ pentru orice } x$$

**Varianta 2.** Fie  $P$  un  $s-y$  drum cu cost minim printre cele cu cel mult  $k$  arce ( $w(P) = \delta_k(s, y)$ )

$\Rightarrow [s \underline{P} x]$  este  $s-x$  drum cu cost minim printre cele cu cel mult  $k-1$  arce, deci are cost  $\delta_{k-1}(s, x)$

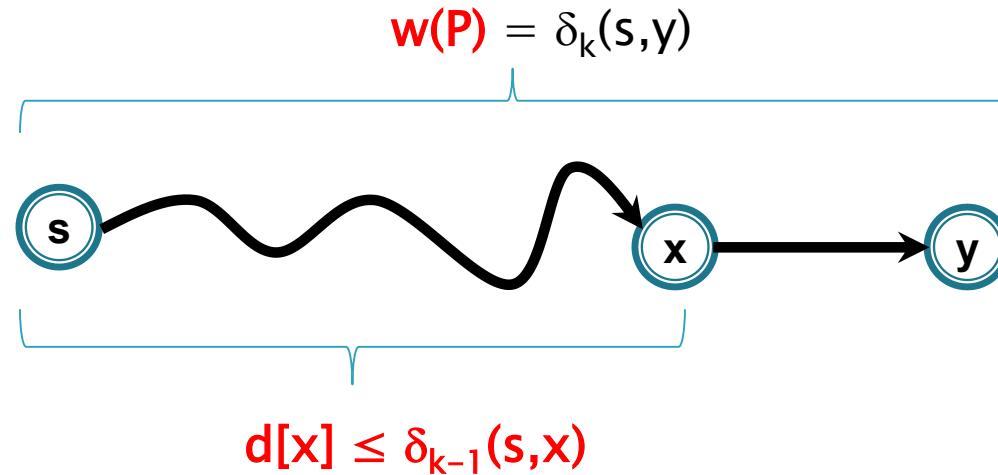
$\Rightarrow d[x] \leq \delta_{k-1}(s, x)$



# Corectitudine

După relaxarea arcului  $xy$ :

$$\begin{aligned} d[y] &\leq d[x] + w(xy) \leq \\ &\leq \delta_{k-1}(s, x) + w(xy) = \\ &= w([s \underline{P} x]) + w(xy) = w(P) = \delta_k(s, y) \end{aligned}$$



# **Detectarea de circuite negative**

# Detectarea de circuite negative

- ▶ Există un circuit negativ în  $G \Leftrightarrow$  dacă algoritmul ar mai face o iterare s-ar mai actualiza etichete de distanță  
 $\Leftrightarrow$  După  $n-1$  iterări 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  $C = [v_0, \dots, v_p, v_0] \Rightarrow d[v_i] \leq d[v_i] + w(v_i v_{i+1})$

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  $C = [v_0, \dots, v_p, v_0] \Rightarrow d[v_i] \leq d[v_i] + w(v_i v_{i+1})$

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 negative detectat – folosind tata:

# Detectarea de circuite negative

Afișarea ciclului negative detectat – folosind tata:

- ▶ Fie  $v$  un vârf al cărei etichetă s-a actualizat la pasul  $k$

# Detectarea de circuite negative

Afișarea ciclului negative 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 (fără circuite)**

# Drumuri minime de sursă unică în grafuri aciclice

## □ Ipoteze:

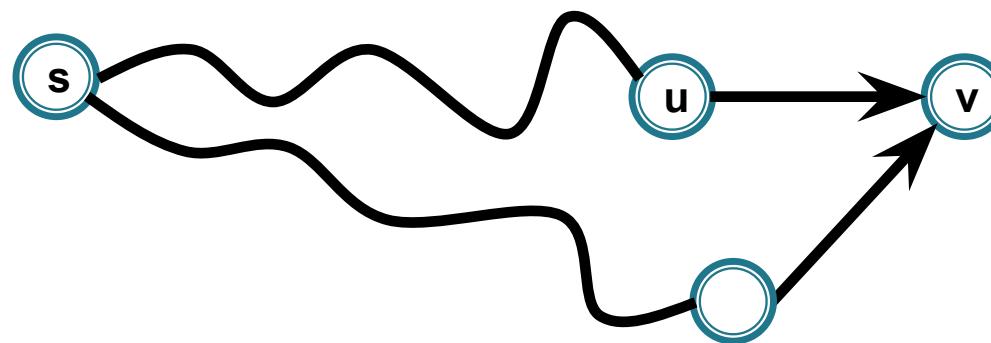
- **Graful nu conține circuite**
- **Arcele pot avea și cost negativ**

# 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  $uv \in E \Rightarrow$**

- 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:

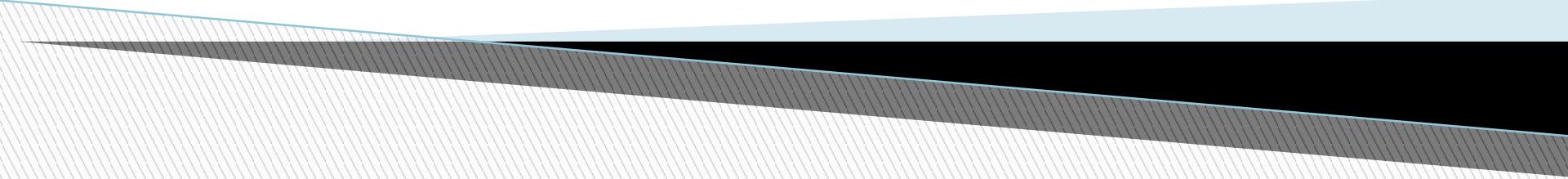
**Când considerăm un vârf  $v$ , pentru a calcula  $d(s,v)$  ar fi util să știm deja  $\delta(s,u)$  pentru orice  $u$  cu  $uv \in E \Rightarrow$**

- 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 vîrfurile în ordinea dată de sortarea topologică
  - Pentru fiecare vîrf  $u$  relaxăm arcele  $uv$  către vecinii săi (pentru a găsi drumuri noi către aceștia)

# Drumuri minime de sursă unică în grafuri aciclice

s – vârful de start

//initializam distante – ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

//este suficient sa pastrăm vârfurile din sortare începând cu s  
SortTop = sortare\_topologica(G, s)

pentru fiecare  $u \in \text{SortTop}$

pentru fiecare  $uv \in E$  executa

daca  $d[u] + w(u, v) < d[v]$  atunci //relaxam uv

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

$tata[v] = u$

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

```
//initializam distante - ca la Dijkstra
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  execută

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

SortTop = sortare\_topologica(G)

pentru fiecare  $u \in \text{SortTop}$

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

SortTop = sortare\_topologica(G)

pentru fiecare  $u \in \text{SortTop}$

    pentru fiecare  $uv \in E$  executa

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

SortTop = sortare\_topologica(G)

pentru fiecare  $u \in \text{SortTop}$

pentru fiecare  $uv \in E$  executa

daca  $d[u] + w(u, v) < d[v]$  atunci //relaxam  $uv$

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

$tata[v] = u$

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

SortTop = sortare\_topologica(G)

pentru fiecare  $u \in \text{SortTop}$

pentru fiecare  $uv \in E$  executa

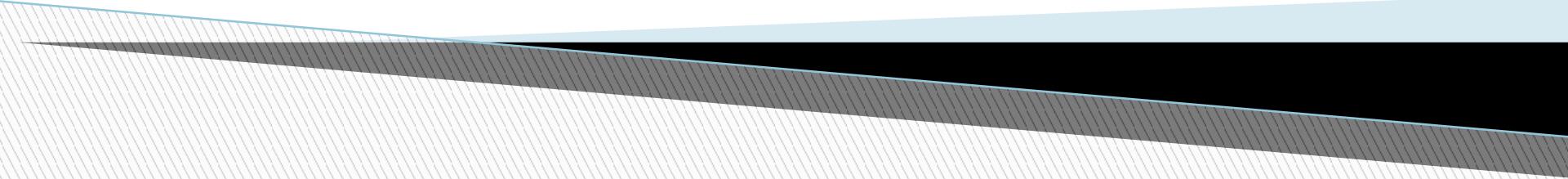
daca  $d[u] + w(u, v) < d[v]$  atunci //relaxam uv

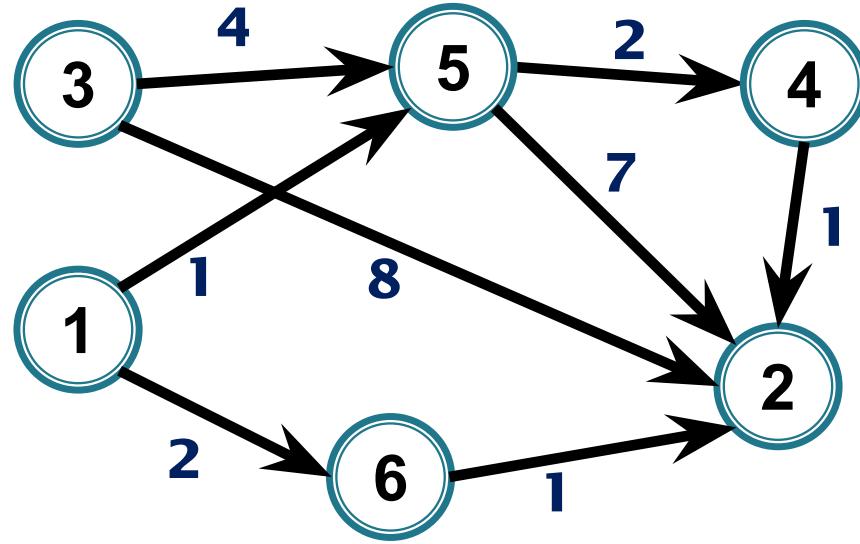
$d[v] = d[u] + w(u, v)$

$tata[v] = u$

scrie  $d, tata$

# **Exemplu**





# Drumuri minime de sursă unică în grafuri aciclice

## □ Etapa 1 – determinăm o ordonare topologică a vârfurilor

### □ Amintim algoritm

SortTop  $\leftarrow \emptyset$ ;

coada  $C \leftarrow \emptyset$ ;

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

cat timp  $C \neq \emptyset$  executa

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

    adauga  $i$  in SortTop

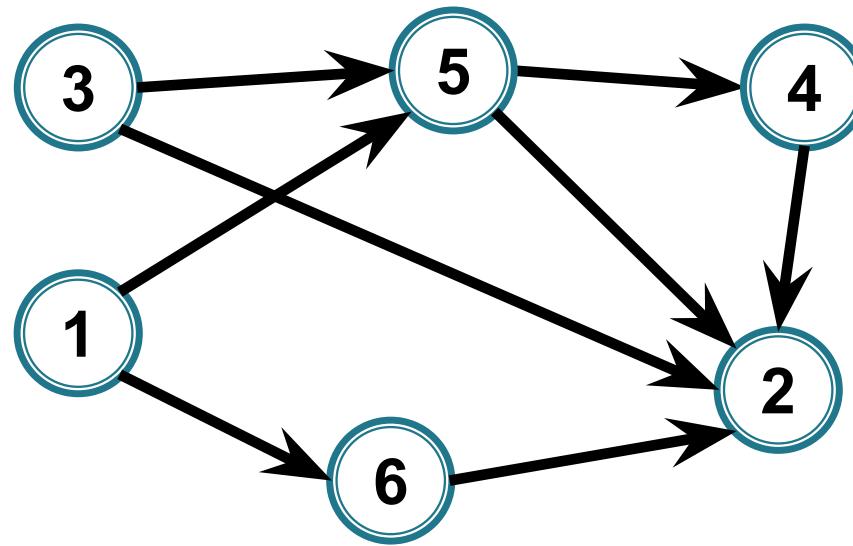
    pentru  $ij \in E$  executa

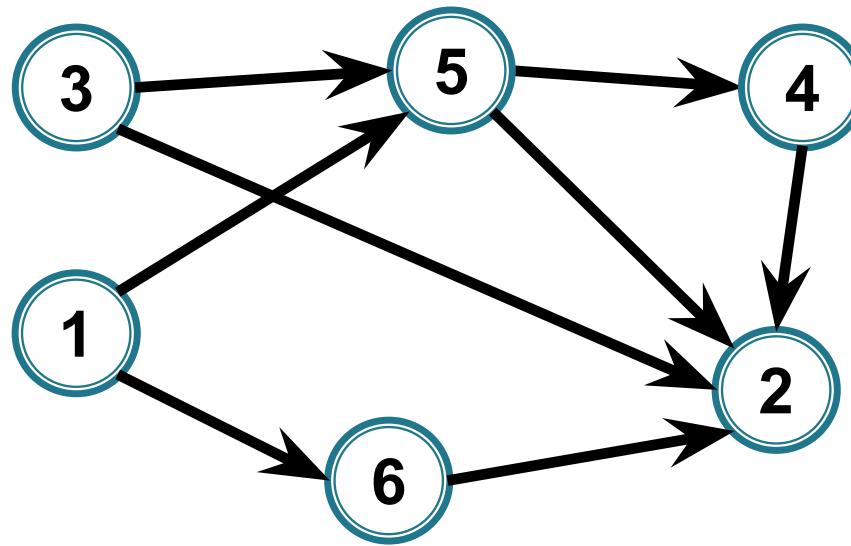
$d^-[j] = d^-[j] - 1$

        daca  $d^-[j]=0$  atunci

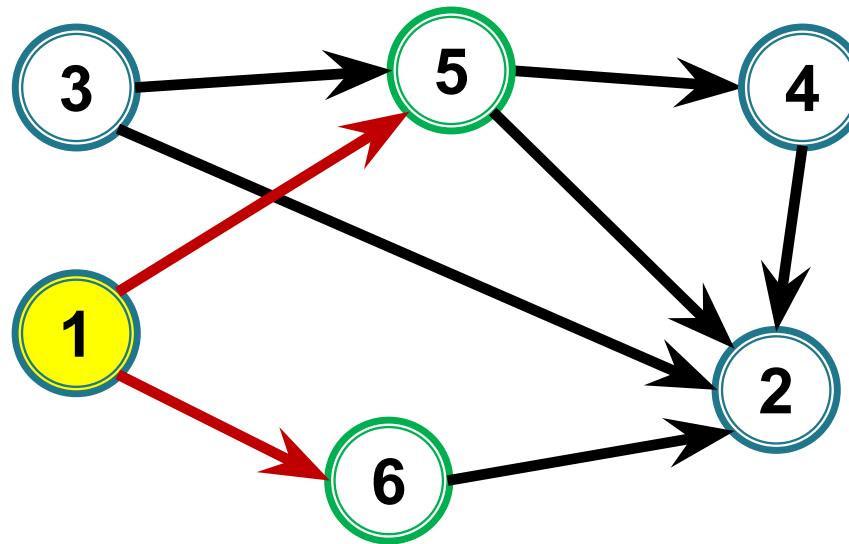
            adauga( $j$ ,  $C$ )

returneaza SortTop

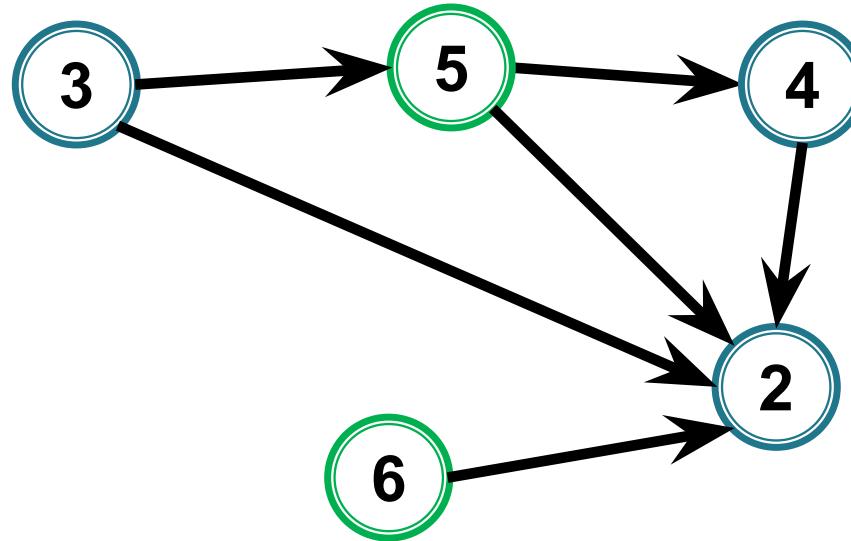




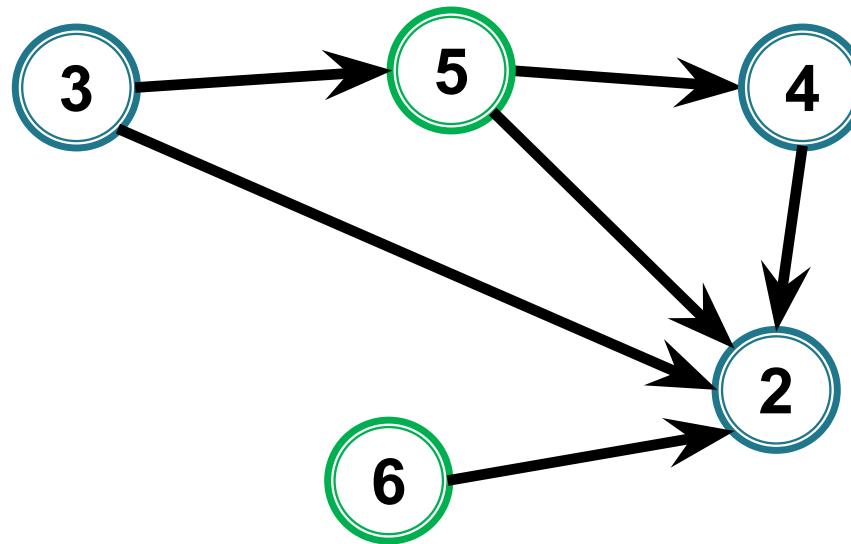
**C: 1 3**



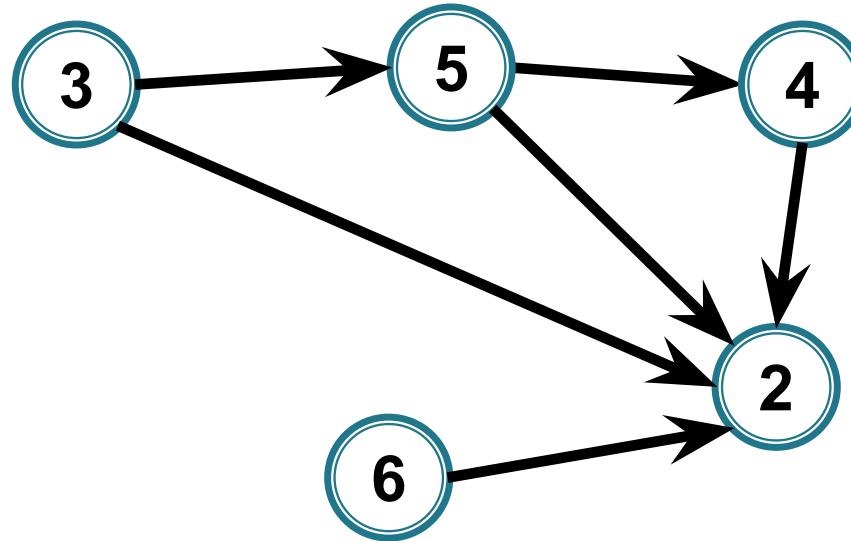
C: 1 3



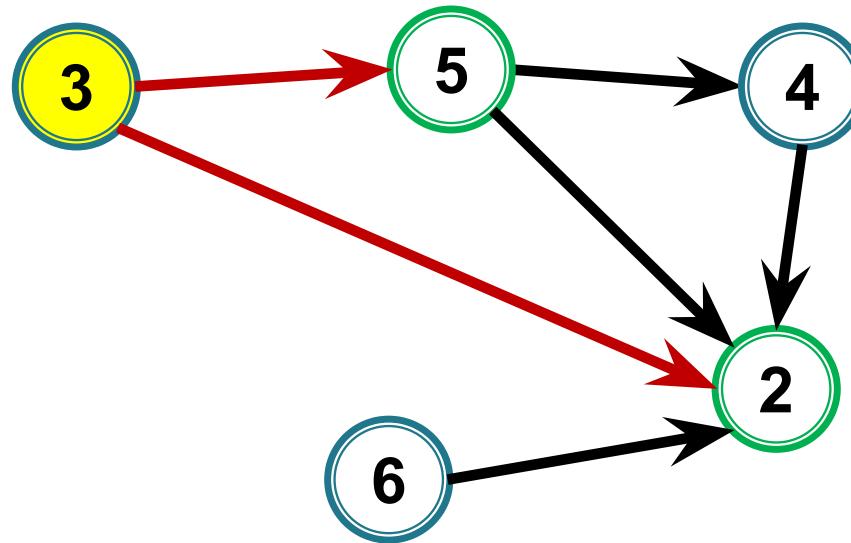
C: 1 3



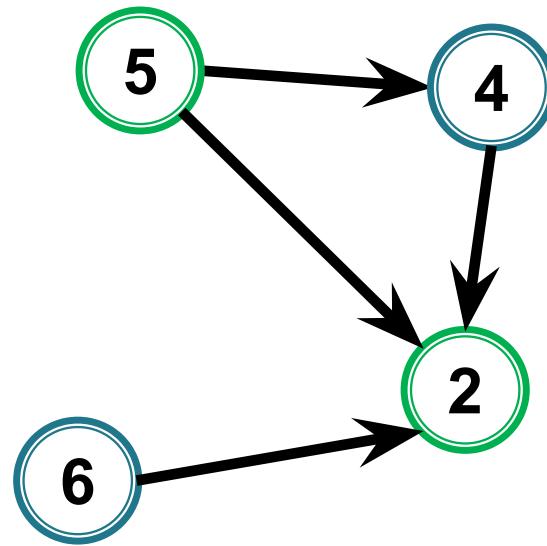
C: **1** 3 6



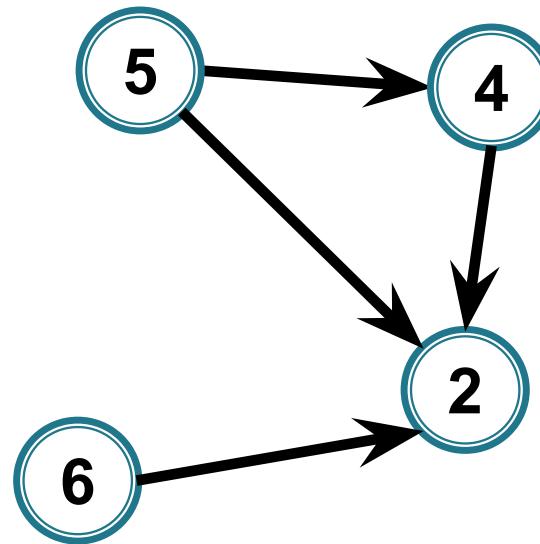
C: **1** 3 6



C: 1 3 6

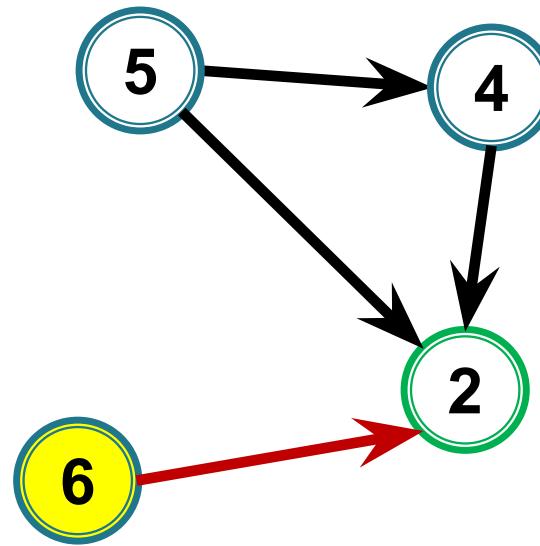


C: 1 3 6

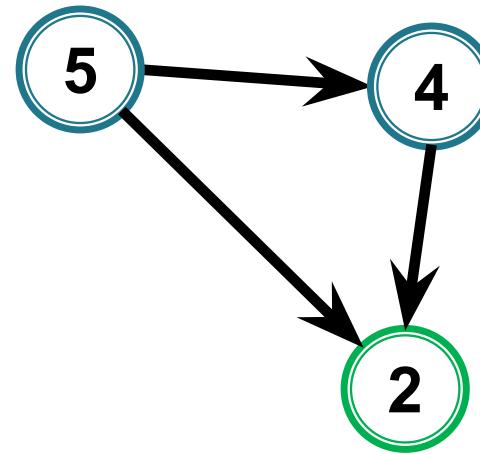


C: 1 3 6

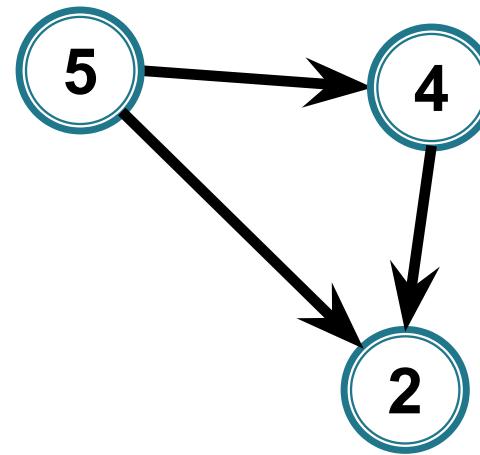
5



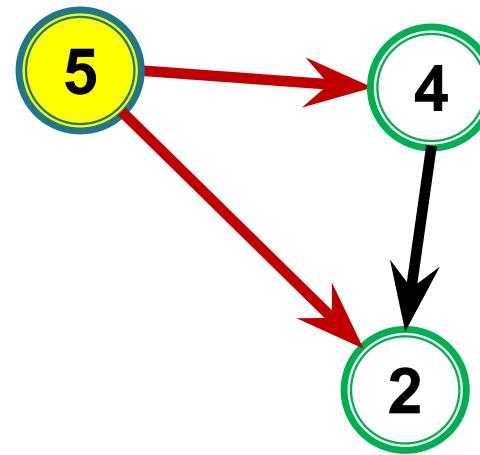
C: 1 3 6 5



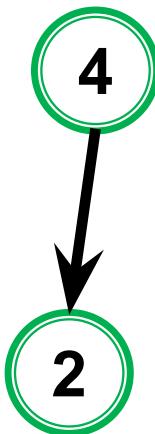
C: 1 3 6 5



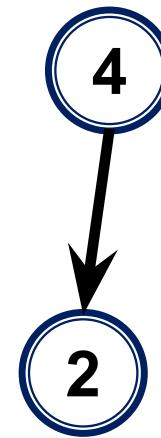
C: 1 3 6 5



C: 1 3 6 5



C: 1 3 6 5



C: 1 3 6 5 4



C: 1 3 6 5 4

2

C: 1 3 6 5 4

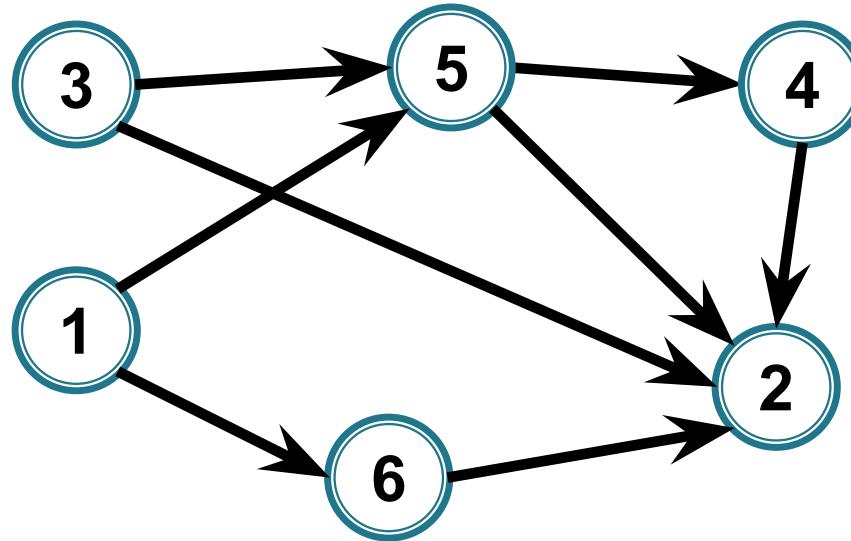
2

C: 1 3 6 5 4 2

2

C: 1 3 6 5 4 2

**C: 1 3 6 5 4 2**



**Sortare topologică: 1 3 6 5 4  
2**

# Sortare topologică - Algoritm

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

# Drumuri minime de sursă unică în grafuri aciclice

- **Etapa 2** – parcurgem vârfurile în ordinea dată de sortarea topologică și relaxăm pentru fiecare vârf arcele care ies din acesta

# Drumuri minime de sursă unică în grafuri aciclice

■ **Initial - determinăm o ordonare topologică a vârfurilor**

■ **Amintim algoritm**

$\text{SortTop} \leftarrow \emptyset;$

*coada C*  $\leftarrow \emptyset;$

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

cat timp  $C \neq \emptyset$  executa

*i*  $\leftarrow$  extrage(*C*) ;

    adauga *i* in SortTop

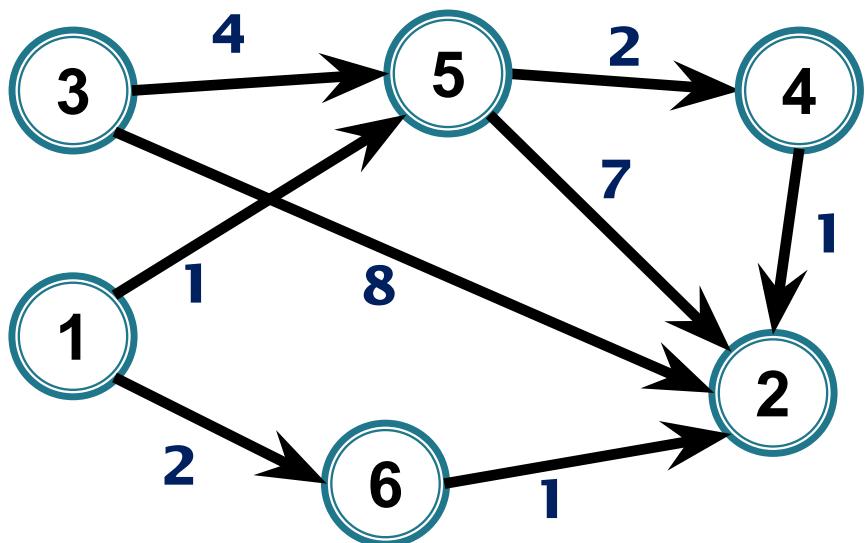
    pentru *ij*  $\in E$  executa

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

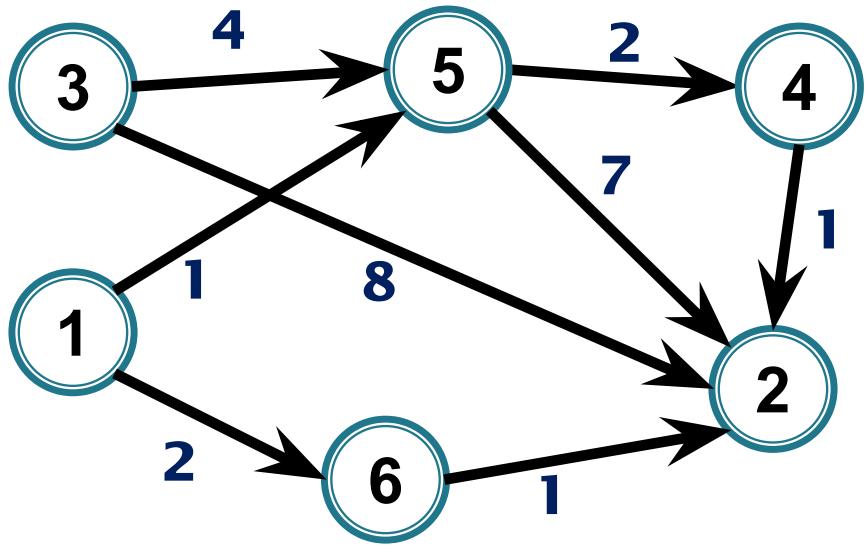
        daca  $d^-[j]=0$  atunci

            adauga (*j*, C)

returneaza SortTop



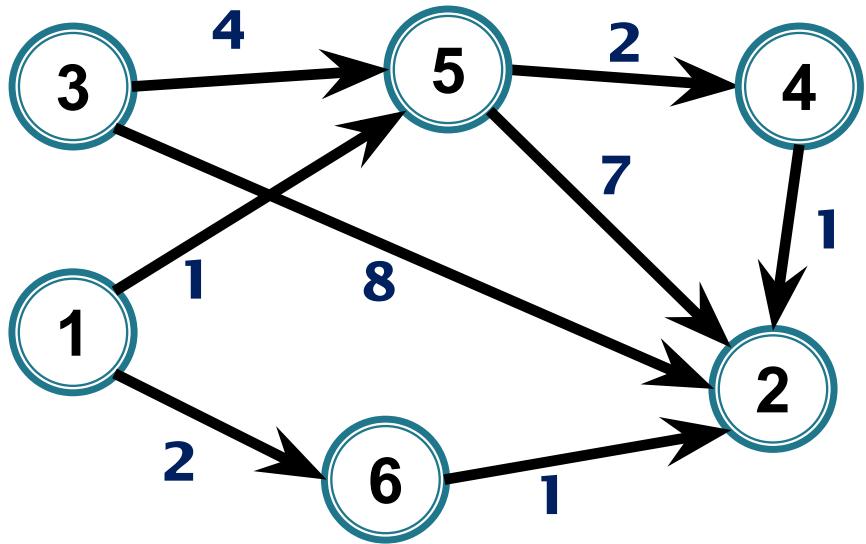
**Sortare topologică**  
1, 3, 6, 5, 4,  
2



**Sortare topologică**  
1, 3, 6, 5, 4,  
2

**s=3 – vârf de start**

**Ordine de calcul  
distanțe:**  
1, 3, 6, 5, 4,  
2



**Sortare topologică**  
**1, 3, 6, 5, 4,**  
**2**

**s=3 - vârf de start**

**Ordine de calcul**

**distanțe:**  
**1, 3, 6, 5, 4,**  
**2**

**d/tat<sub>a</sub>**

[ **1**/0, **∞**/0,

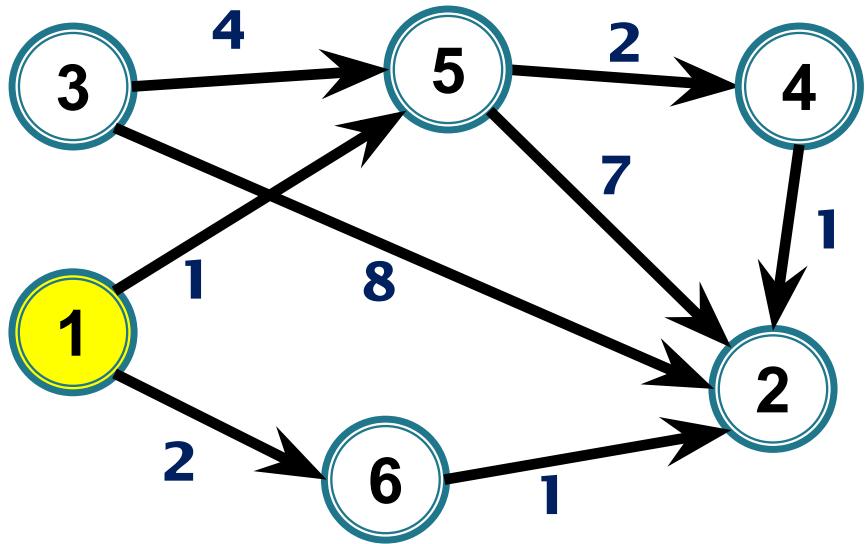
**2**/0, **∞**/0,

**3**/0,

**4**/0, **∞**/0,

**5**/0, **∞**/0,

**6**/0 ]



**Sortare topologică** 1, 3, 6, 5, 4,

2

**s=3 - vârf de start**

**Ordine de calcul distanțe:**

1, 3, 6, 5, 4,  
2

$d_{tat}$   
 $a$

$u = 1:$

[  $\infty/0$ ,

$2\infty/0$ ,

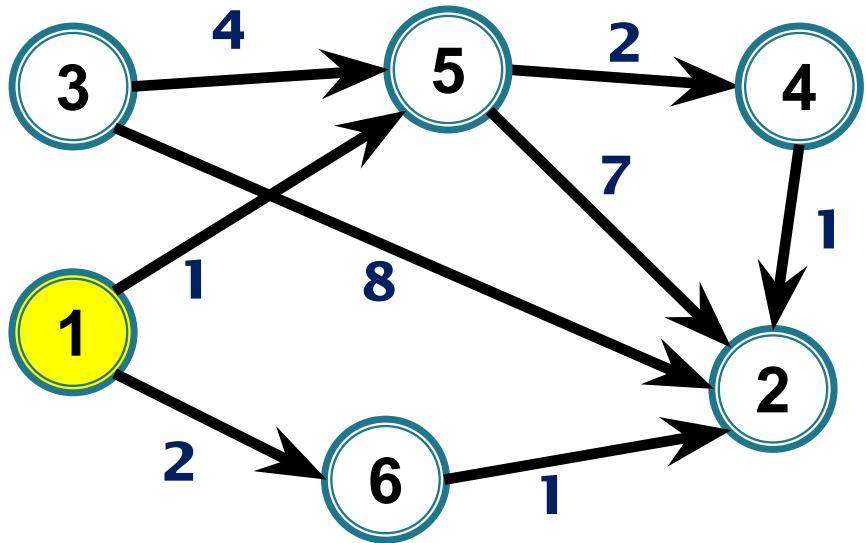
$3/0$ ,

$4\infty/0$ ,

$5\infty/0$ ,

$6\infty/0$  ]

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



**Sortare topologică** 1, 3, 6, 5, 4, 2

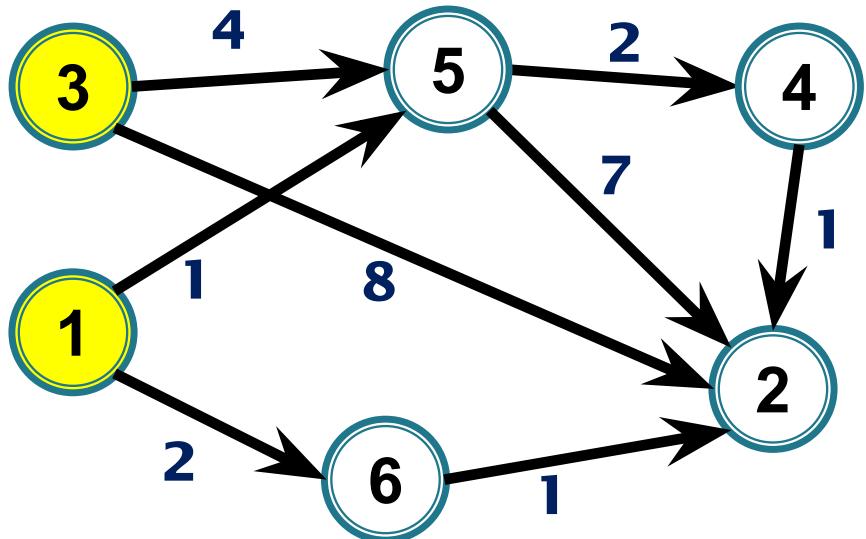
**s=3 - vârf de start**

**Ordine de calcul distanțe:** 1, 3, 6, 5, 4, 2

$d_{tat_a}$	[ $\infty/0$ , $1/0$ , $2\infty/0$ ,	$3/0$ ,	$4\infty/0$ ,	$5\infty/0$ ,	$6\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $\infty/0$ ,	$0/0$ ,	$\infty/0$ ,	$\infty/0$ ,	$\infty/0$ ]

1 nu este accesibil din s, puteam să nu îl considerăm  
(să ignorăm vâfurile din ordonare topologică aflate înaintea lui s)

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



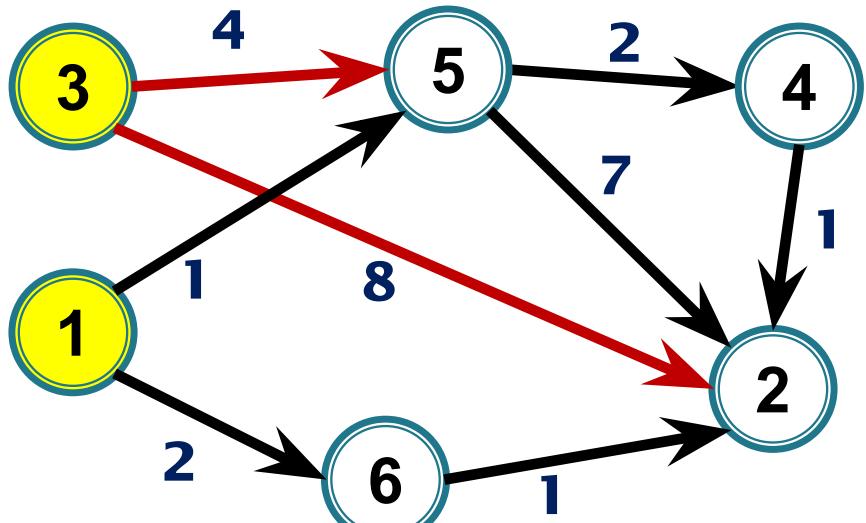
**Sortare topologică**  
1, 3, 6, 5, 4,  
2

**s=3 - vârf de start**

**Ordine de calcul distanțe:**  
1, 3, 6, 5, 4,  
2

$d_{tat}$	$[ \infty/0, \infty/0 ]$					
$u = 1:$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ 0/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$
$u = 3:$						

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



**Sortare topologică** 1, 3, 6, 5, 4,

2

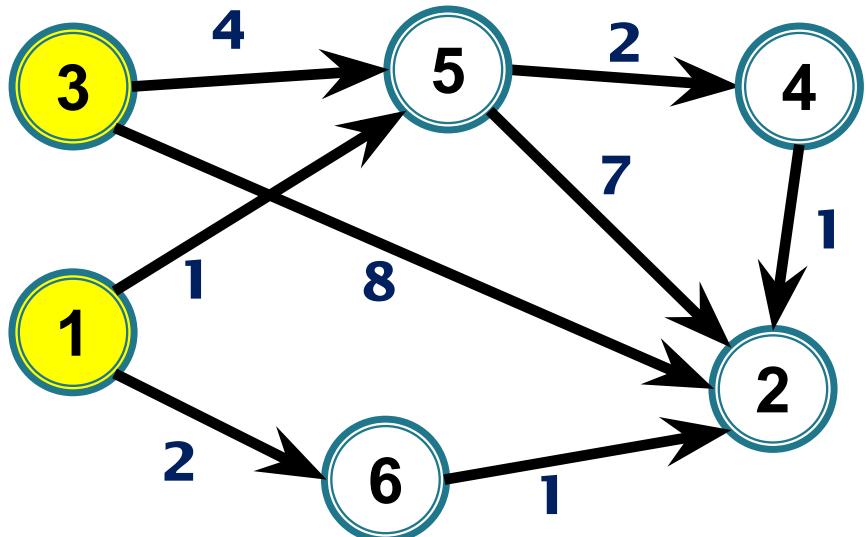
**s=3 - vârf de start**

**Ordine de calcul distanțe:**

1, 3, 6, 5, 4,  
2

$d_{tat}$	$[ \infty/0, \infty/0 ]$					
$u = 1:$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ 0/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$	$[ \infty/0, \infty/0 ]$
$u = 3:$						

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



**Sortare topologică** 1, 3, 6, 5, 4, 2

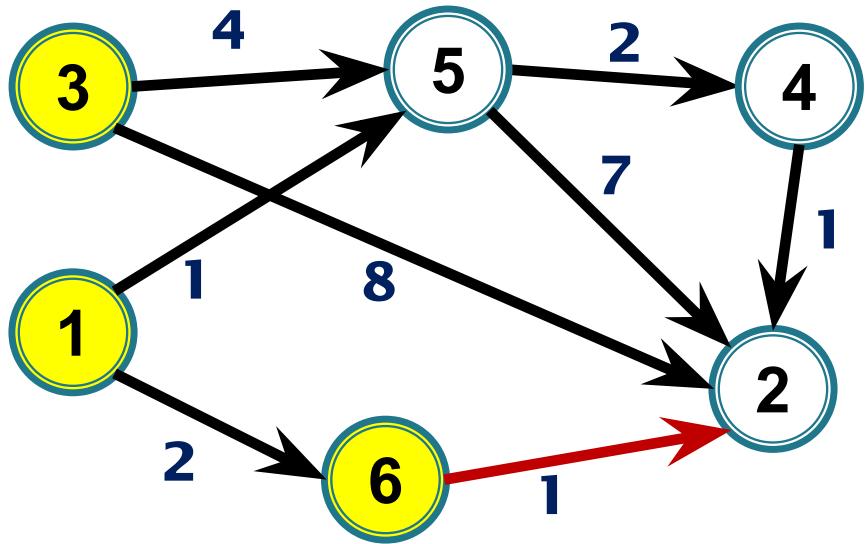
**s=3 - vârf de start**

**Ordine de calcul**

**distanțe:** 1, 3, 6, 5, 4, 2

<b>d/tat<sub>a</sub></b>	[ $\infty/0$ , $1/0$ , $2\infty/0$ , $30/0$ , $4\infty/0$ , $5\infty/0$ , $6\infty/0$ ]
<b>u = 1:</b>	[ $\infty/0$ , $\infty/0$ , $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
<b>u = 3:</b>	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]

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



**Sortare topologică** 1, 3, 6, 5, 4, 2

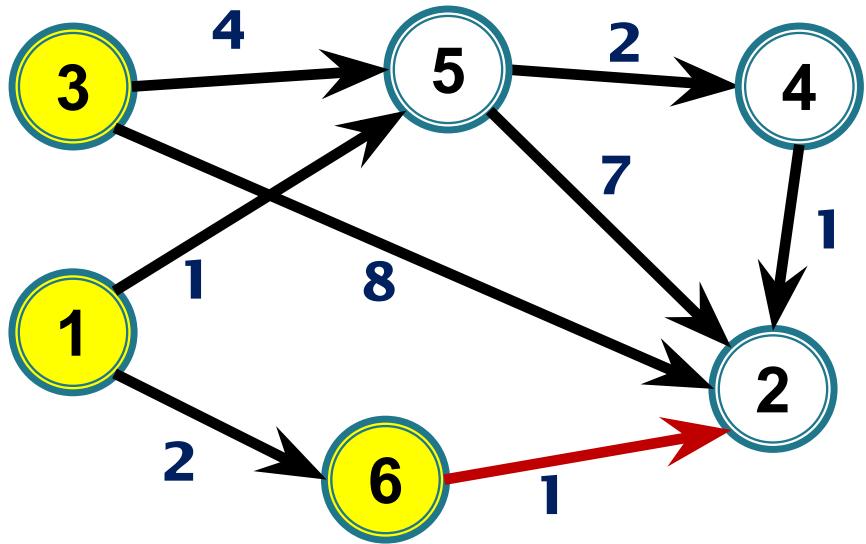
**s=3 - vârf de start**

**Ordine de calcul**

**distanțe:** 1, 3, 6, 5, 4, 2

$d_{tat}$	[ $\infty/0$ , $1/0$ , $2\infty/0$ , $3/0$ , $4\infty/0$ , $5\infty/0$ , $6\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $\infty/0$ , $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
$u = 3:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 6:$	

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



**Sortare topologică** 1, 3, 6, 5, 4, 2

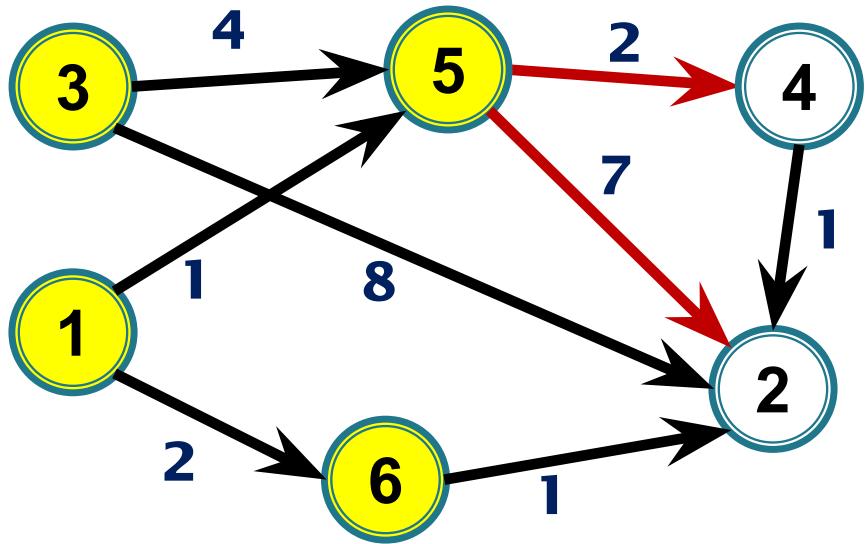
**s=3** – vârf de start

**Ordine de calcul**

distanțe: 1, 3, 6, 5, 4, 2

$d_{tat}$	1	2	3	4	5	6
$u = 1:$	$\infty/0$	$\infty/0$	$0/0$	$\infty/0$	$\infty/0$	$\infty/0$
$u = 3:$	$\infty/0$	$8/3$	$0/0$	$\infty/0$	$4/3$	$\infty/0$
$u = 6:$	$\infty/0$	$8/3$	$0/0$	$\infty/0$	$4/3$	$\infty/0$

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



**Sortare topologică** 1, 3, 6, 5, 4,

2

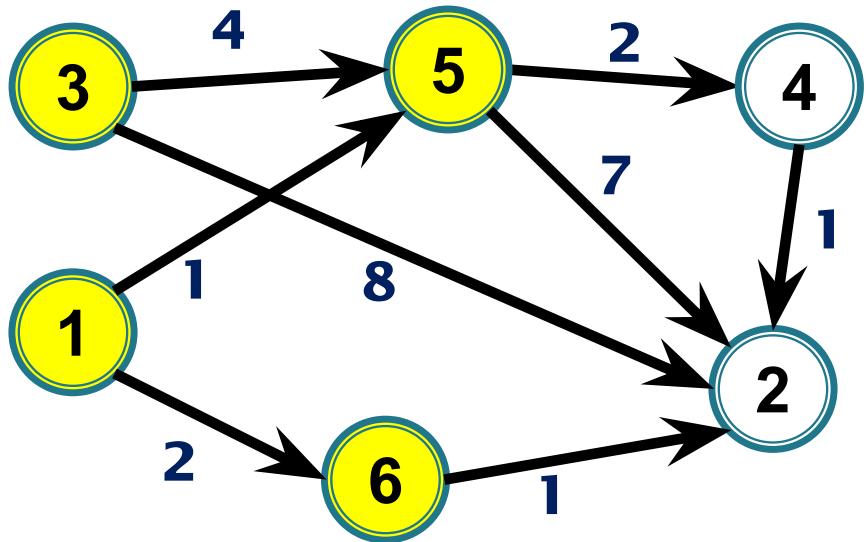
**s=3 - vârf de start**

**Ordine de calcul distanțe:**

1, 3, 6, 5, 4,  
2

$d_{tat}$	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

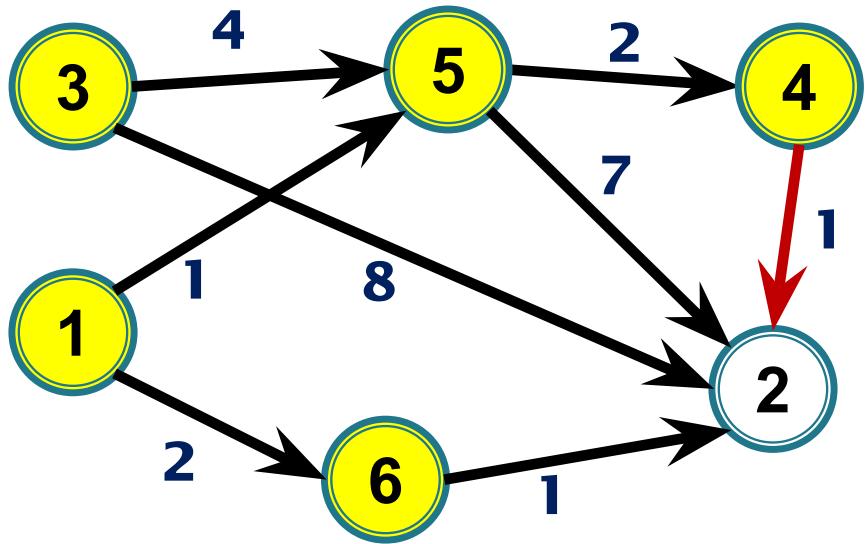
**s=3 - vârf de start**

**Ordine de calcul**

**distanțe:** 1, 3, 6, 5, 4, 2

$d_{tat}$ $a$	[ $\infty/0$ , $1/0$ , $2\infty/0$ , $30/0$ , $4\infty/0$ , $5\infty/0$ , $6\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $\infty/0$ , $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
$u = 3:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 6:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 5:$	[ $\infty/0$ , $8/3$ , $0/0$ , $6/5$ , $4/3$ , $\infty/0$ ]

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



**Sortare topologică** 1, 3, 6, 5, 4,

2

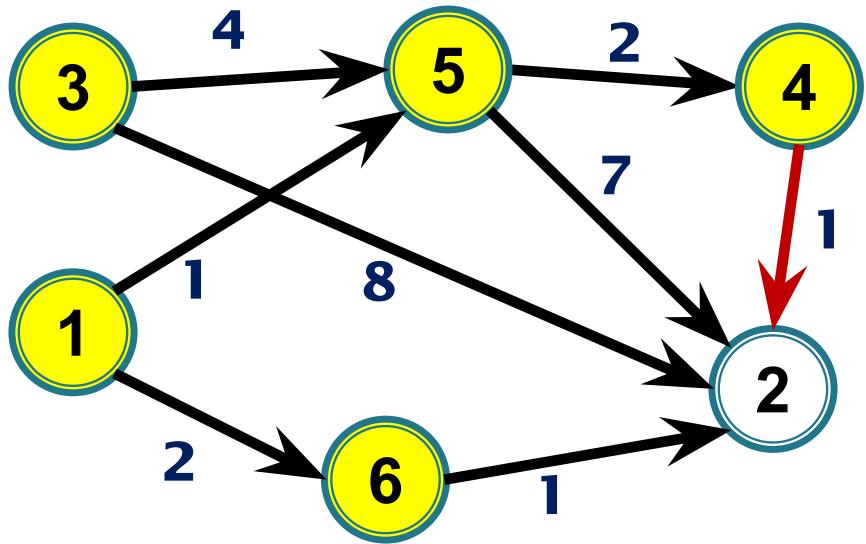
**s=3 - vârf de start**

**Ordine de calcul distanțe:**

1, 3, 6, 5, 4,  
2

$d_{tat}$ $a$	[ $\infty/0$ , $1/0$ ]	[ $\infty/0$ , $2/0$ ]	[ $3/0$ , $0/0$ ]	[ $4/0$ , $\infty/0$ ]	[ $5/0$ , $\infty/0$ ]	[ $6/0$ , $\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]	[ $0/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]
$u = 3:$	[ $\infty/0$ , $0/0$ ]	[ $8/3$ , $0/0$ ]	[ $0/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]	[ $4/3$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]
$u = 6:$	[ $\infty/0$ , $0/0$ ]	[ $8/3$ , $0/0$ ]	[ $0/0$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]	[ $4/3$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]
$u = 5:$	[ $\infty/0$ , $0/0$ ]	[ $8/3$ , $0/0$ ]	[ $0/0$ , $0/0$ ]	[ $6/5$ , $0/0$ ]	[ $4/3$ , $0/0$ ]	[ $\infty/0$ , $0/0$ ]
$u = 4:$						

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



**Sortare topologică** 1, 3, 6, 5, 4,

2

**s=3 - vârf de start**

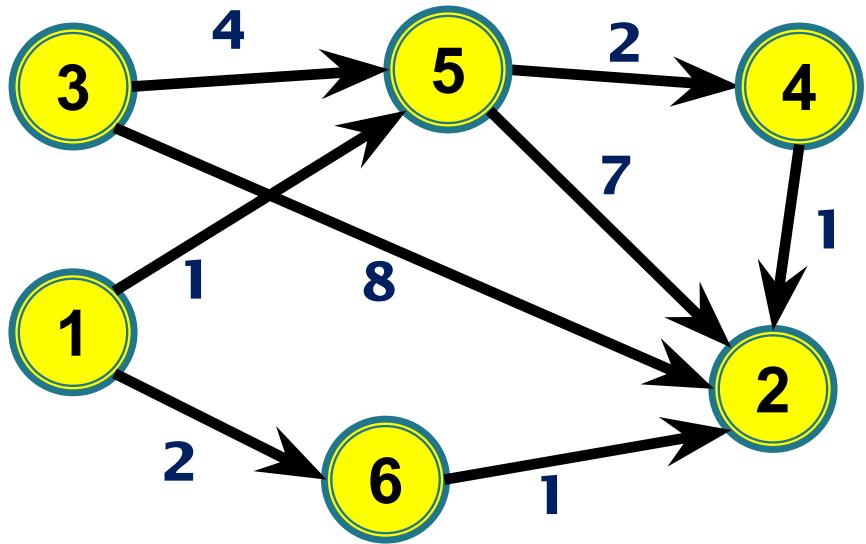
**Ordine de calcul distanțe:**

1, 3, 6, 5, 4,

2

$d_{tat_a}$	[ $\infty/0$ , $1/0$ , $2\infty/0$ , $30/0$ , $4\infty/0$ , $5\infty/0$ , $6\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $\infty/0$ , $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
$u = 3:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 6:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 5:$	[ $\infty/0$ , $8/3$ , $0/0$ , $6/5$ , $4/3$ , $\infty/0$ ]
$u = 4:$	[ $\infty/0$ , $7/4$ , $0/0$ , $6/5$ , $4/3$ , $\infty/0$ ]

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



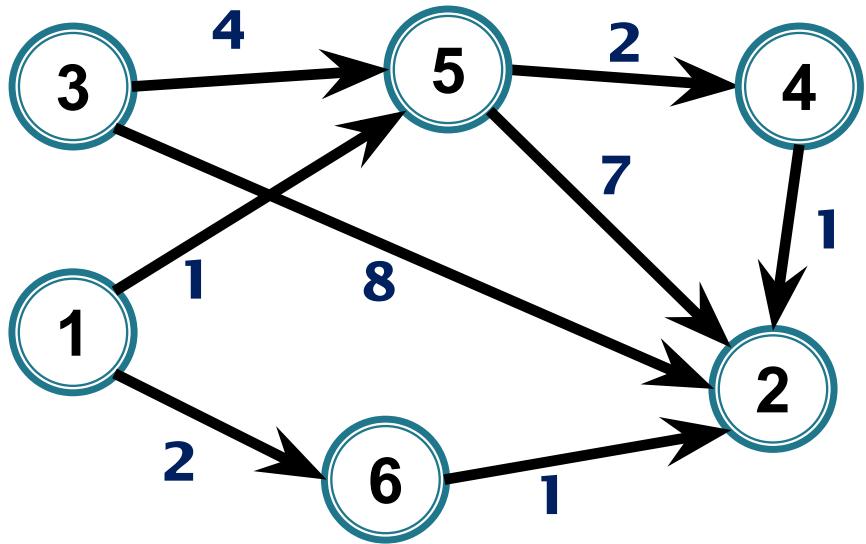
**Sortare topologică**  
1, 3, 6, 5, 4,  
2

**s=3 - vârf de start**

**Ordine de calcul distanțe:**  
1, 3, 6, 5, 4,  
2

$d_{tat_a}$	[ $\infty/0$ , $1/0$ , $2\infty/0$ , $30/0$ , $4\infty/0$ , $5\infty/0$ , $6\infty/0$ ]
$u = 1:$	[ $\infty/0$ , $\infty/0$ , $0/0$ , $\infty/0$ , $\infty/0$ , $\infty/0$ ]
$u = 3:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 6:$	[ $\infty/0$ , $8/3$ , $0/0$ , $\infty/0$ , $4/3$ , $\infty/0$ ]
$u = 5:$	[ $\infty/0$ , $8/3$ , $0/0$ , $6/5$ , $4/3$ , $\infty/0$ ]
$u = 4:$	[ $\infty/0$ , $7/4$ , $0/0$ , $6/5$ , $4/3$ , $\infty/0$ ]
$u = 2:$	

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



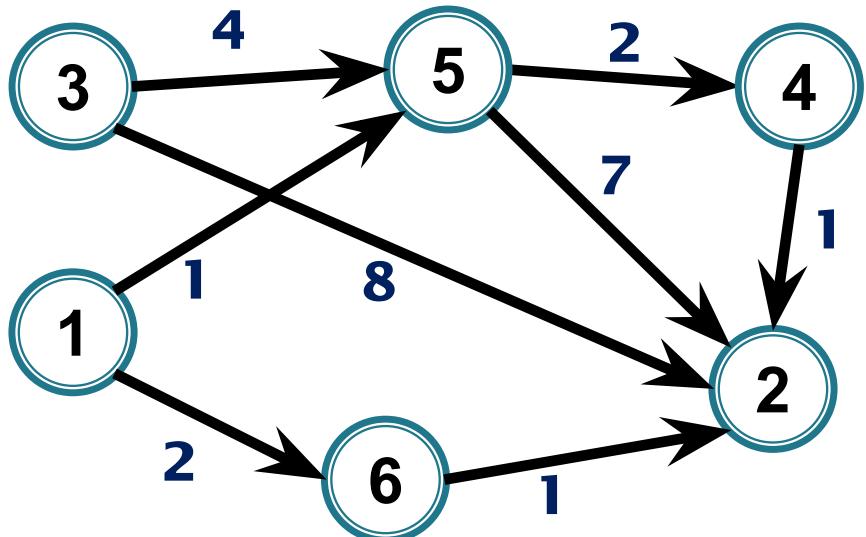
**Sortare topologică** 1, 3, 6, 5, 4, 2

**s=3** – vârf de start

**Ordine de calcul**

distanțe: 1, 3, 6, 5, 4, 2

$d_{tat}$	1	2	3	4	5	6
$u = 1:$	$\infty/0$	$\infty/0$	$0/0$	$\infty/0$	$\infty/0$	$\infty/0$
$u = 3:$	$\infty/0$	$8/3$	$0/0$	$\infty/0$	$4/3$	$\infty/0$
$u = 6:$	$\infty/0$	$8/3$	$0/0$	$\infty/0$	$4/3$	$\infty/0$
$u = 5:$	$\infty/0$	$8/3$	$0/0$	$6/5$	$4/3$	$\infty/0$
$u = 4:$	$\infty/0$	$7/4$	$0/0$	$6/5$	$4/3$	$\infty/0$
$u = 2:$	$\infty/0$	$7/4$	$0/0$	$6/5$	$4/3$	$\infty/0$



**Sortare topologică** 1, 3, 6, 5, 4, 2

**s=3** – vârf de start

**Ordine de calcul distanțe:** 1, 3, 6, 5, 4, 2

d/tat	1	2	3	4	5	6
<b>Soluție</b>	[ $\infty/0$ , $7/4$ ,	$0/0$ ,	$6/5$ ,	$4/3$ ,	$\infty/0$ ]	

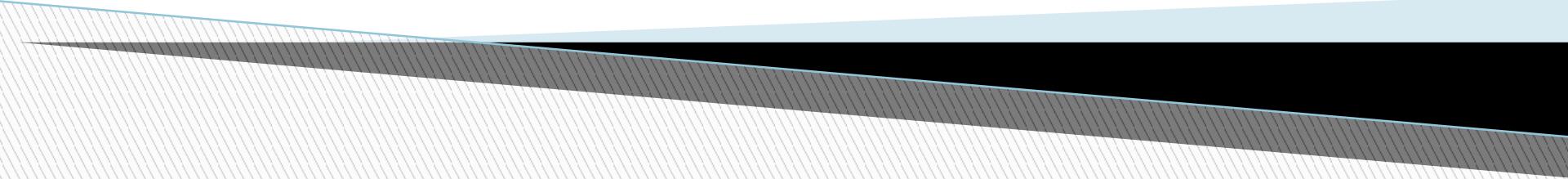
**Un drum minim de la 3 la 2?**

# 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();  
        adauga u in sortare  
    }
```

# Drumuri minime de sursă unică în grafuri aciclice

s - vârful de start

//initializam distante - ca la Dijkstra

pentru fiecare  $u \in V$  executa

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

$d[s] = 0$

//determinăm o sortare topologică a vârfurilor

SortTop = sortare\_topologica(G)

pentru fiecare  $u \in \text{SortTop}$

pentru fiecare  $uv \in E$  executa

daca  $d[u] + w(u, v) < d[v]$  atunci //relaxam  $uv$

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

$tata[v] = u$

scrie  $d, tata$

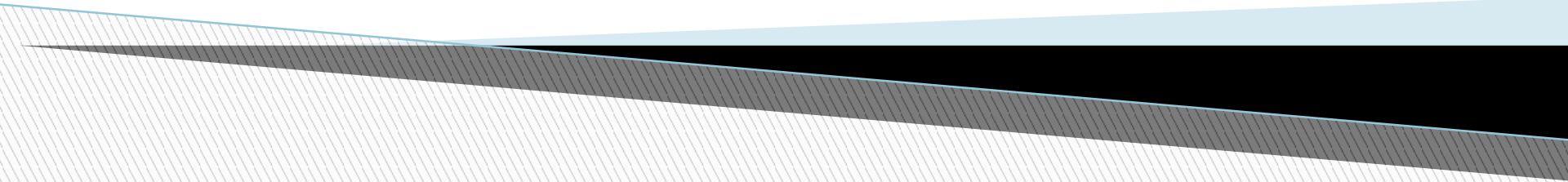
# Drumuri minime de sursă unică în grafuri aciclice

# Complexitate

- **Inițializare** →  $O(n)$
  - **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

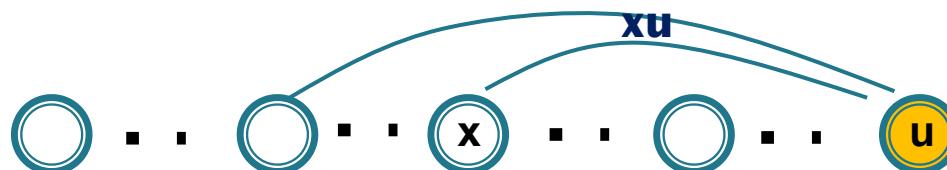


$$\delta(s, u) = \min\{\delta(s, x) + w(x, u) \mid xu \in E\}$$

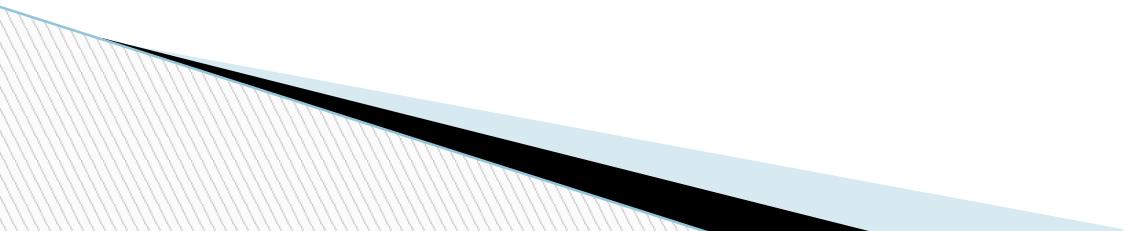
Când algoritmul ajunge la vârful **u**  
avem

$$d[u] = \min\{d[x] + w(x, u) \mid xu \in E\}$$

↑ relaxare arce



SortTo  
p



# Concluzii

Drumuri minime de sursă unică –  
algoritmi

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

$G$  – neponderat

Parcuregere lățime BF

BF(s)

```
coada C ← Ø;  
adauga(s, C)
```

$G$  – ponderat, ponderi  $> 0$

Algoritmul lui Dijkstra

Dijkstra(s)

```
(min-heap) Q ← V  
{se putea incepe doar cu Q ← {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 lățime BF

BF(s)

coada  $C \leftarrow \emptyset$ ;

adauga(s, C)

pentru fiecare  $u \in V$

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

$viz[s] \leftarrow 1$ ;  $d[s] \leftarrow 0$

$G$  – ponderat, ponderi  $> 0$

Algoritmul lui Dijkstra

Dijkstra(s)

(min-heap)  $Q \leftarrow V$   
{se putea incepe 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  $\leftarrow$  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

Parcuregere lățime BF

BF(s)

```
coada C ← Ø;  
adauga(s, C)
```

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

```
viz[s]← 1; d[s] ← 0
```

```
cat timp  $C \neq \emptyset$   
u ← extrage(C);
```

```
pentru fiecare uv∈E
```

$G$  – ponderat, ponderi  $>0$

Algoritmul lui Dijkstra

Dijkstra(s)

```
(min-heap) Q ← V  
{se putea incepe doar cu  $Q \leftarrow \{s\}$   
+vector viz;  $v \in Q \Leftrightarrow v$  nevizitat}
```

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

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

```
cat timp  $Q \neq \emptyset$   
u = extrage(Q) vârf cu eticheta  
d minimă
```

```
pentru fiecare uv∈E
```

$G$  – ponderat fără circuite

DAGS(s)

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

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

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

```
pentru fiecare  $u \in \text{SortTop}$ 
```

```
pentru fiecare uv∈E
```

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

$G$  – neponderat

Parcuregere lățime BF

BF(s)

```
coada C ← Ø;  
adauga(s, C)
```

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

```
viz[s]← 1; d[s] ← 0
```

```
cat timp  $C \neq \emptyset$   
    u ← extrage(C);
```

```
pentru fiecare uv∈E  
    daca viz[v]=0  
        d[v] ← d[v]+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 incepe doar cu  $Q \leftarrow \{s\}$   
+vector viz;  $v \in Q \Leftrightarrow v$  nevizitat}
```

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

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

```
cat timp  $Q \neq \emptyset$   
    u = extrage(Q) vârf cu eticheta  
        d minimă  
  
    pentru fiecare uv∈E  
        daca  $v \in Q$  si  $d[u]+w(u,v) < d[v]$   
            d[v] = d[u]+w(u,v)  
            tata[v] = u  
            repară(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  $u \in \text{SortTop}$ 
```

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

```
scrie d, tata
```

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

$G$  – neponderat

Parcuregere lățime BF

BF(s)

```
coada C ← Ø;
adauga(s, C)
```

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

```
viz[s] ← 1;  $d[s] \leftarrow 0$ 
```

```
cat timp  $C \neq \emptyset$ 
     $u \leftarrow \text{extrage}(C);$ 
```

```
pentru fiecare  $uv \in E$ 
    daca  $viz[v] = 0$ 
         $d[v] \leftarrow d[u] + 1$ 
         $tata[v] \leftarrow u$ 
        adauga(v, C)
         $viz[v] \leftarrow 1$ 
```

scrie  $d$ ,  $tata$

$O(n+m)$

$G$  – ponderat, ponderi  $> 0$

Algoritmul lui Dijkstra

Dijkstra(s)

```
(min-heap)  $Q \leftarrow V$ 
{se putea incepe doar cu  $Q \leftarrow \{s\}$ 
+vector viz;  $v \in Q \Leftrightarrow v$  nevizitat}
```

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

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

```
cat timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu eticheta
        d minimă
```

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

scrie  $d$ ,  $tata$

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

$G$  – ponderat fără circuite

DAGS(s)

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

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

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

```
pentru fiecare  $u \in \text{SortTop}$ 
```

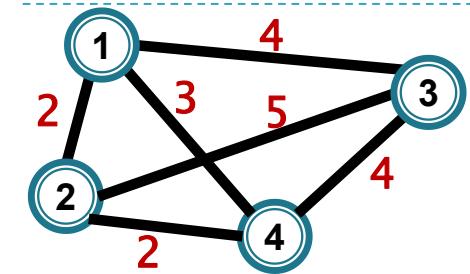
```
pentru fiecare  $uv \in E$ 
    daca  $d[u] + w(u,v) < d[v]$ 
         $d[v] = d[u] + w(u,v)$ 
         $tata[v] = u$ 
```

scrie  $d$ ,  $tata$

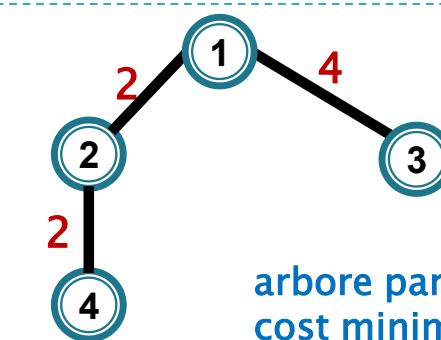
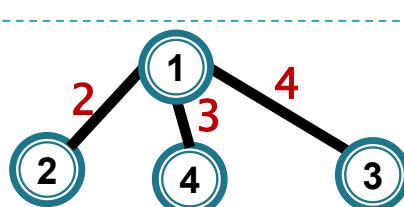
$O(n+m)$

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

$\neq$  arbore parțial de cost minim – minimizează costul total



arbore al drumurilor minime față de 1



arbore parțial de cost minim

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

$\neq$  arbore parțial de cost minim – minimizează costul total

G-(ne)orientat ponderat,  
ponderi  $>0$

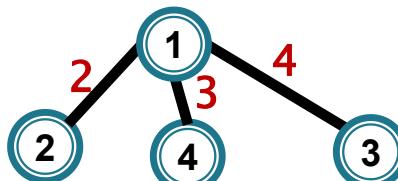
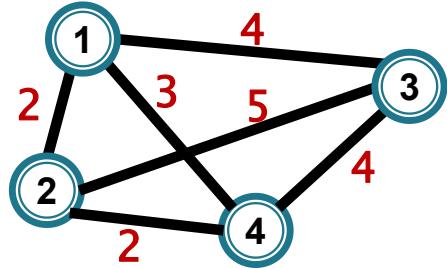
Drumuri minime din s

Algoritmul lui Dijkstra

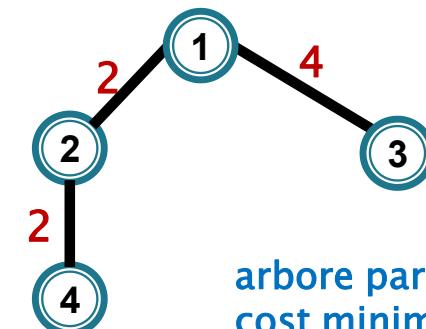
G- neorientat ponderat  
ponderi reale

Arbore parțial de cost minim

Algoritmul lui Prim



arbore al drumurilor minime față de 1



arbore parțial de  
cost minim

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

$\neq$  arbore parțial de cost minim – minimizează costul total

G-(ne)orientat ponderat,  
ponderi  $> 0$

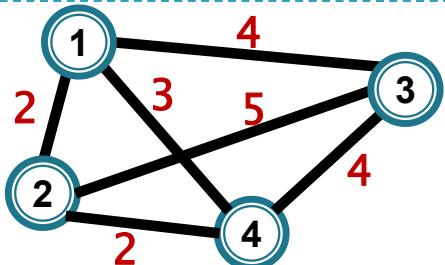
Drumuri minime din s

Algoritmul lui Dijkstra

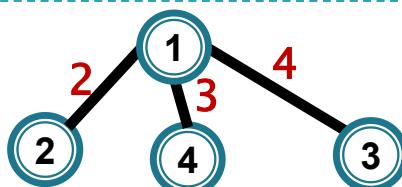
Dijkstra(s)

```
(min-heap) Q ← V
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ; tata[u]=0
 $d[s] = 0$ 
cat timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu eticheta
        d minimă
    pentru fiecare  $uv \in E$ 
        dacă  $v \in Q$  și  $d[u] + w(u, v) < d[v]$ 
             $d[v] = d[u] + w(u, v)$ 
            tata[v] = u
            repara(v, Q)
```

scrie d, tata



arbore al drumurilor minime față de 1



G- neorientat ponderat  
ponderi reale

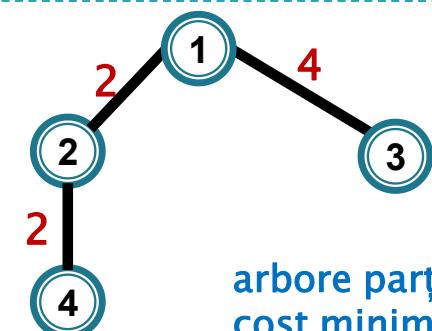
Arbore parțial de cost minim

Algoritmul lui Prim

Prim(s)

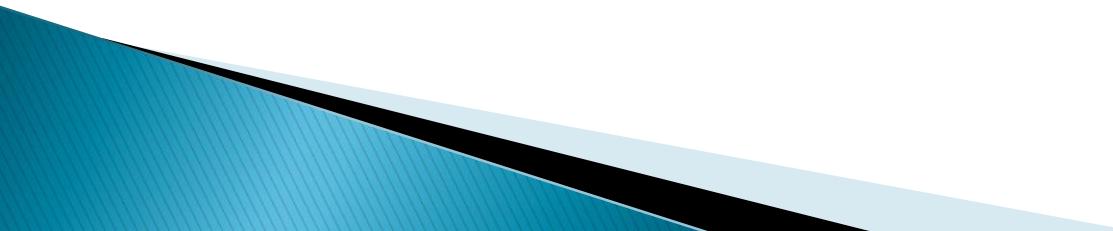
```
(min-heap) Q ← V
pentru fiecare  $u \in V$ 
     $d[u] = \infty$ ; tata[u]=0
 $d[s] = 0$ 
cat timp  $Q \neq \emptyset$ 
     $u = \text{extrage}(Q)$  vârf cu
        eticheta d minimă
    pentru fiecare  $uv \in E$ 
        dacă  $v \in Q$  și  $w(u, v) < d[v]$ 
             $d[v] = w(u, v)$ 
            tata[v] = u
            repara(v, Q)
```

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



arbore parțial de  
cost minim





**Drumuri minime între toate  
perechile de vârfuri**

**Algoritmul Floyd–Warshall**

# Problema drumurilor minime între toate perechile de vârfuri

Se dă:

- un graf **orientat** ponderat  $G = (V, E, w)$

Pentru oricare două vârfuri  $x$  și  $y$  al 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  $G$



## ▶ Soluția 1

Se aplică algoritmul lui Dijkstra pentru fiecare vârf  $x$

## ► Soluția 1

Se aplică algoritmul lui Dijkstra pentru fiecare vârf x  
!!!funcționează dacă ponderile sunt pozitive

Complexitate =  $n * \text{complexitate Dijkstra}$

## ➤ Soluția 2

# Algoritmul Floyd–Warshall

# Floyd–Warshall

- ▶ Fie  $W = (w_{ij})_{i,j=1..n}$  **matricea costurilor** grafului  $G$ :

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

- ▶ Vrem să calculăm **matricea distanțelor**  $D = (d_{ij})_{i,j=1..n}$ :

$$d_{ij} = \delta(i, j)$$

# Floyd–Warshall

- ▶ Fie  $W = (w_{ij})_{i,j=1..n}$  **matricea costurilor** grafului  $G$ :

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

- ▶ Vrem să calculăm **matricea distanțelor**  $D = (d_{ij})_{i,j=1..n}$ :

$$d_{ij} = \delta(i, j)$$

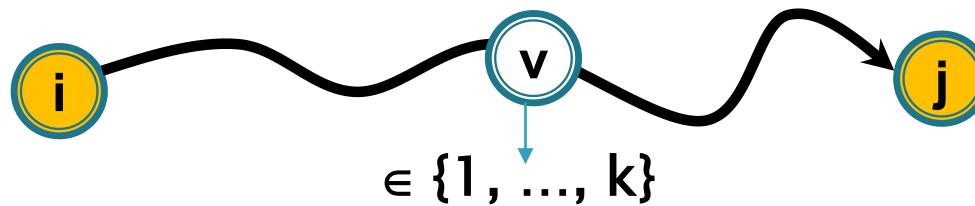
- ▶ **Observație:**  $w_{ij} = \text{costul minim al unui } i-j \text{ drum fără vârfuri intermediare (cu cel mult un arc)}$

# 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 intermediare doar vârfuri din multimea  $\{1, 2, \dots, k\}$**



# Floyd–Warshall

## ► Ideea algoritmului Floyd–Warshall:

Astfel, pentru  $k = 1, 2, \dots, n$  calculăm matricea

$$D^{(k)} = (d_{ij}^k)_{i,j=1..n} :$$

$d_{ij}^k$  = costul minim al unui drum de la  $i$  la  $j$  care are  
vârfurile intermediiare în  $\{1, 2, \dots, k\}$

- Inițializare:



# Floyd–Warshall

## ► Ideea algoritmului Floyd–Warshall:

Astfel, pentru  $k = 1, 2, \dots, n$  calculăm matricea

$$D^{(k)} = (d_{ij}^k)_{i,j=1..n} :$$

$d_{ij}^k$  = 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

$$D^{(k)} = (d_{ij}^k)_{i,j=1..n} :$$

$d_{ij}^k$  = costul minim al unui drum de la  $i$  la  $j$  care are  
vârfurile intermediiare 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  $P^{(k)} = (p_{ij}^k)_{i,j=1..n}$  :

$p_{ij}^k$  = 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  $D^{(k)}$ ?



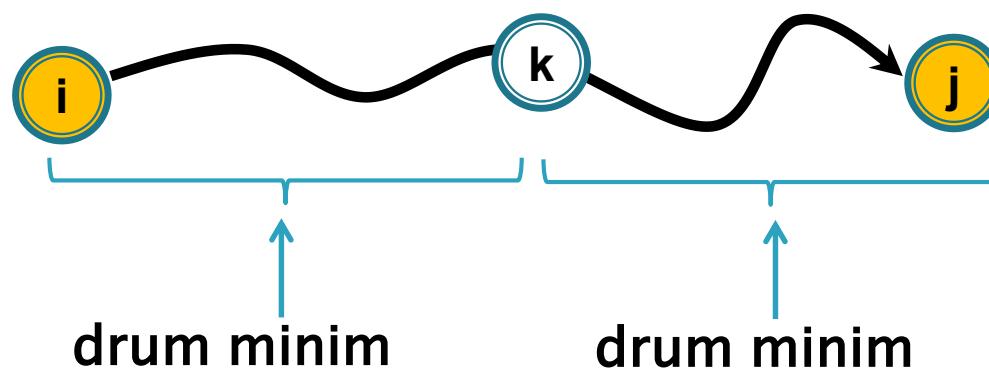
Relație de recurență

# Floyd–Warshall

- ▶ **Ideea** de calcul al matricei  $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$  cu vârfurile intermediare în mulțimea  $\{1, 2, \dots, k\}$

- Dacă vârful  $k$  este vârf intermediar al lui  $P$

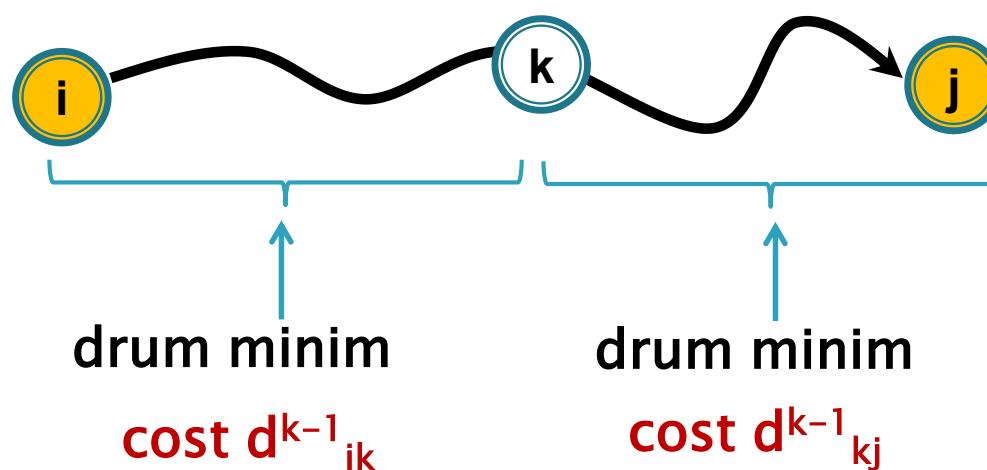


# Floyd–Warshall

## ► Ideea de calcul al matricei $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$  cu vârfurile intermediare în mulțimea  $\{1, 2, \dots, k\}$

- Dacă vârful  $k$  este vârf intermediar al lui  $P$

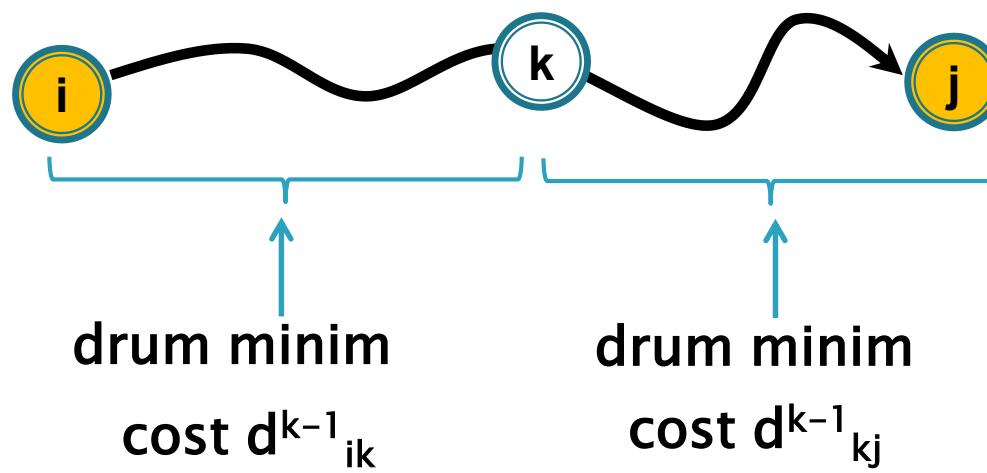


# Floyd–Warshall

## ► Ideea de calcul al matricei $D^{(k)}$ :

Fie  $P$  un drum de cost minim de la  $i$  la  $j$  cu vârfurile intermediare în mulțimea  $\{1, 2, \dots, k\}$

- 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

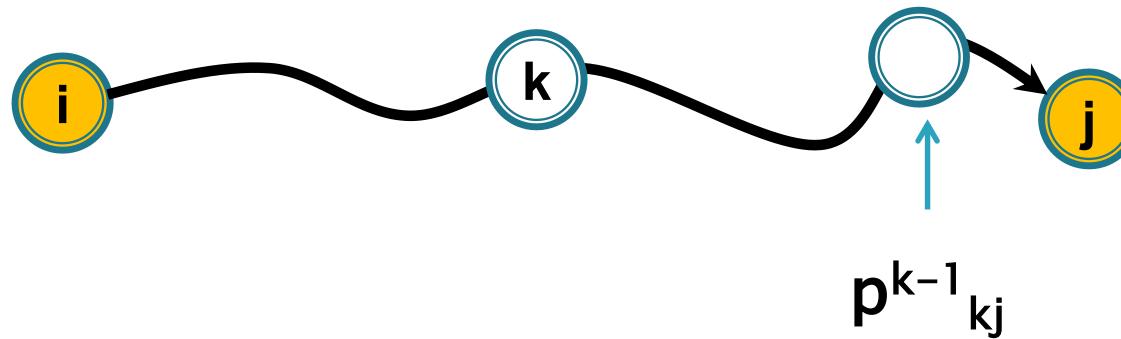
- ▶ Se obține astfel relația

$$d^k_{ij} = \min \{d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}\}$$

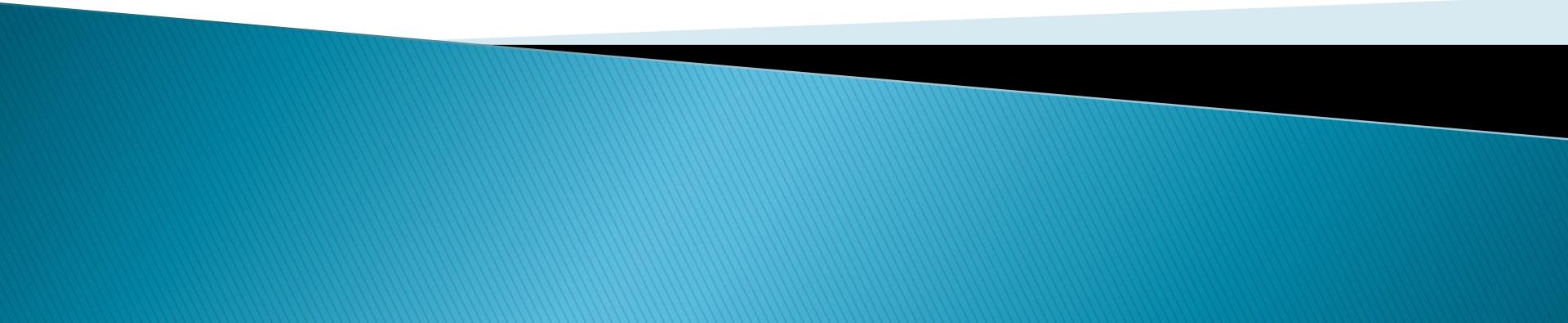
- ▶ Observații

Când de 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
- ▶ **Initializare**

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

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

Floyd(G, w)

```
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(G, w)

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

Floyd(G, w)

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

Floyd(G, w)

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

# Floyd–Warshall

- ▶ Ieșire: matricea  $d$  = matricea distanțelor minime
- ▶ Afisarea unui drum de la  $i$  la  $j$ , daca  $d[i][j] < \infty$ , se face folosind matricea  $p$

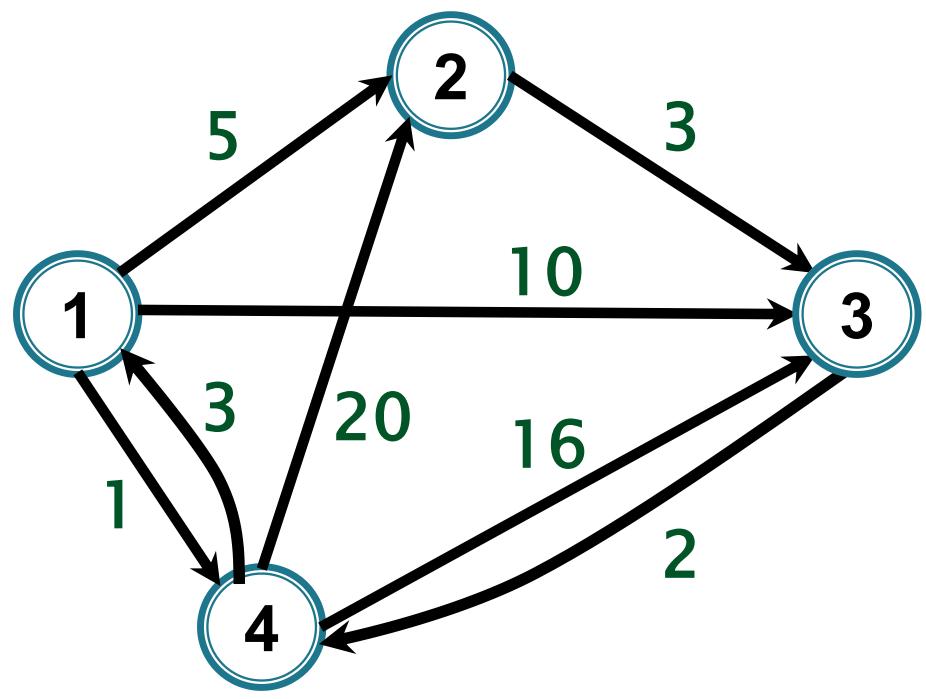
# Floyd-Warshall

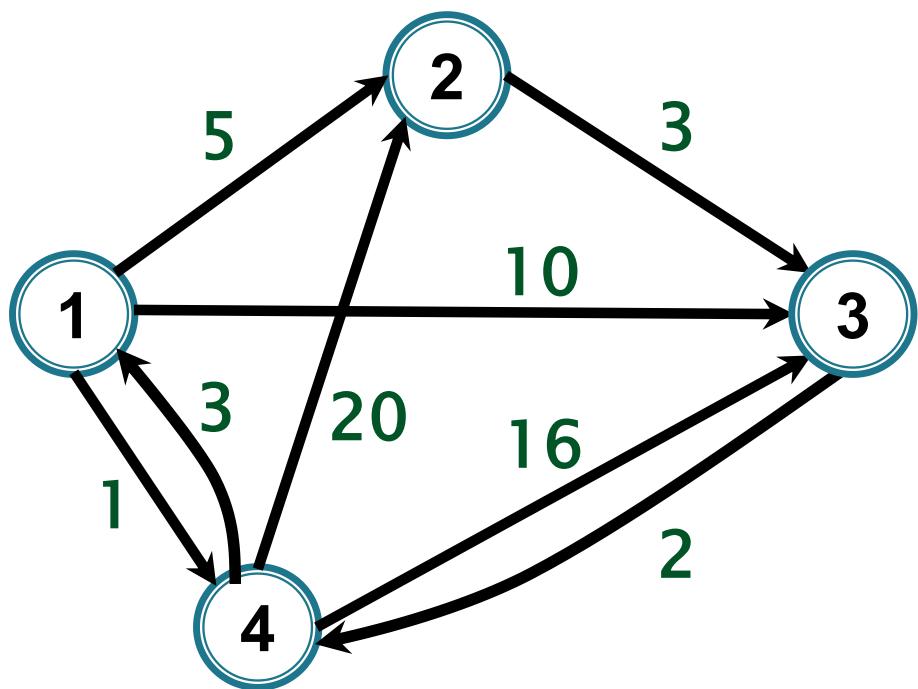
- ▶ Ieșire: matricea  $d = \text{matricea distanțelor minime}$
- ▶ Afisarea unui drum de la  $i$  la  $j$ , **daca  $d[i][j] < \infty$** , se face folosind matricea  $p$

```
void drum(int i,int j){  
    if(i!=j)  
        drum(i,p[i][j]);  
    cout<<j<<" ";  
}
```

# Floyd-Warshall

**Complexitate -  $O(n^3)$**



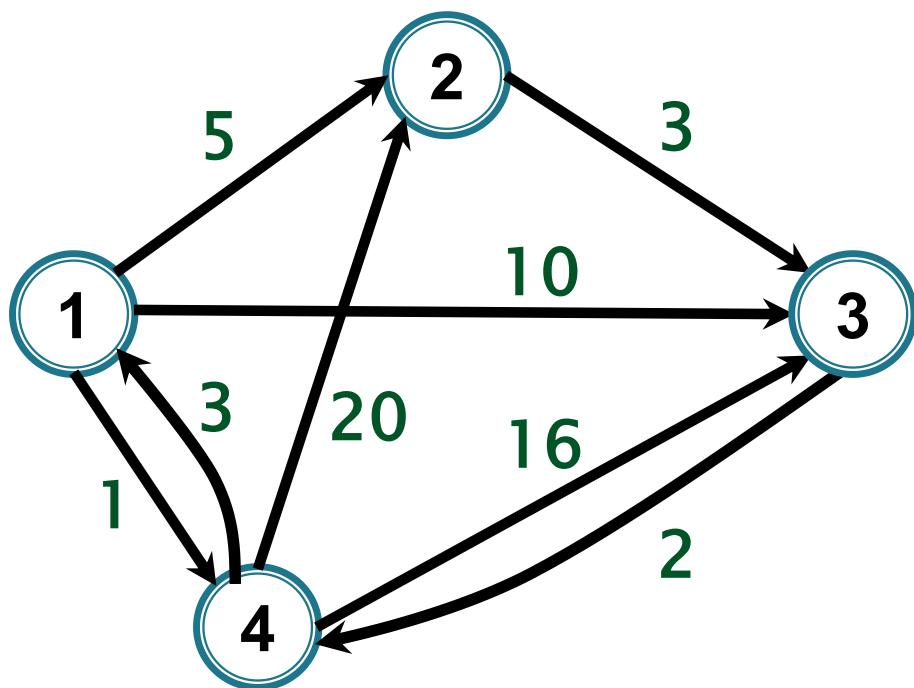


$w=d=$

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

$p =$

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

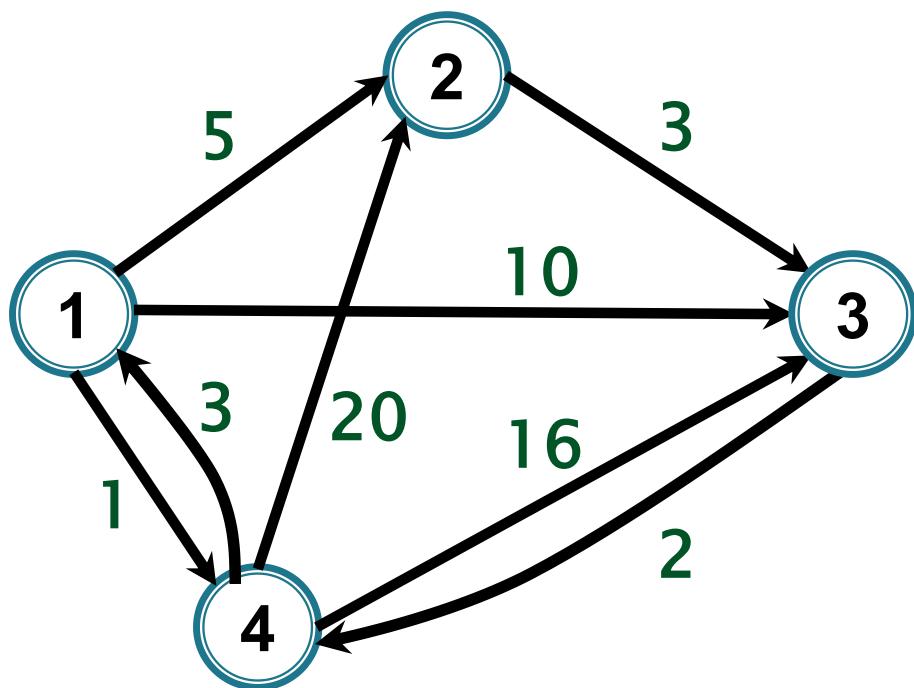
$d =$

$k=1$

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	20	16	0

$p=$

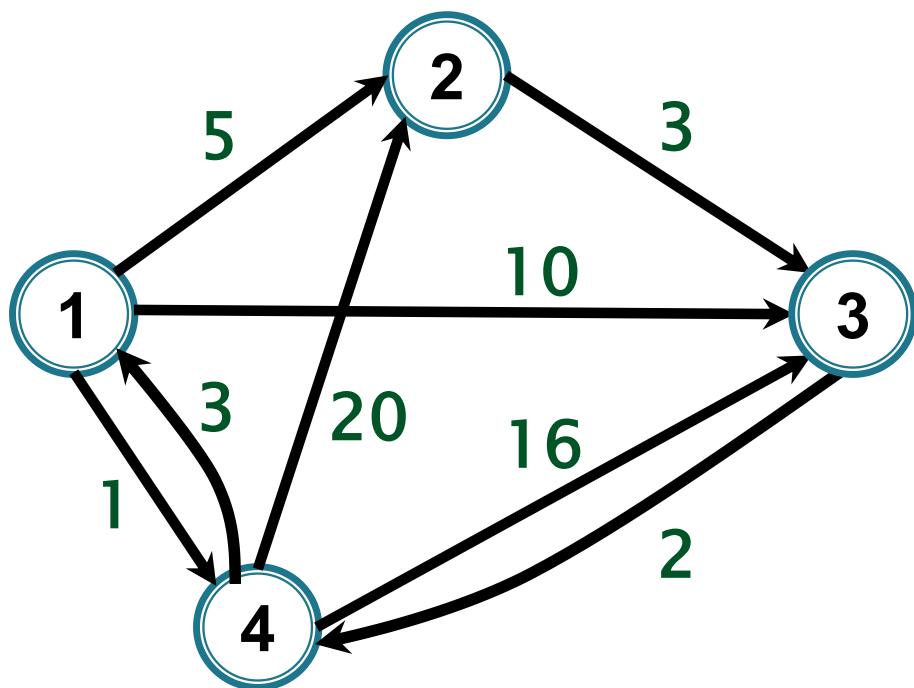
0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

$d=$

$k=1$	$k=2$
0 5 10 1	0 5 8 1
$\infty$ 0 3 $\infty$	$\infty$ 0 3 $\infty$
$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2
3 8 13 0	3 8 11 0

$p=$

$k=1$	$k=2$
0 1 1 1	0 1 2 1
0 0 2 0	0 0 2 0
0 0 0 3	0 0 0 3
4 1 1 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

$k=2$

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

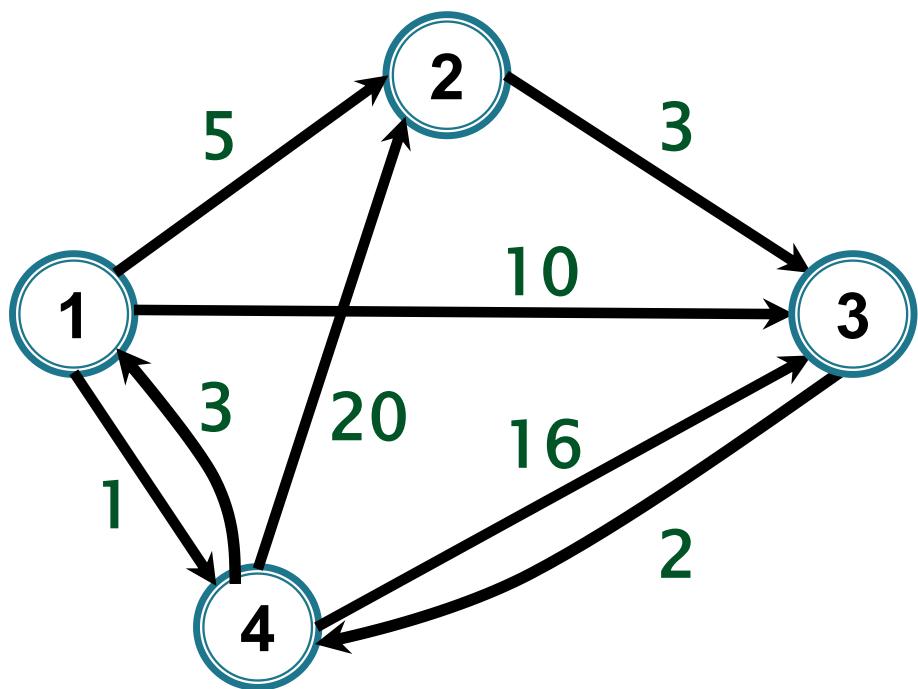
$d=$

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

$p=$

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



0	5	10	1
$\infty$	0	3	$\infty$
$\infty$	$\infty$	0	2
3	20	16	0

w=d=

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

p=

k=1	k=2	k=3	k=4
0 5 10 1	0 5 8 1	0 5 8 1	0 5 8 1
$\infty$ 0 3 $\infty$	$\infty$ 0 3 $\infty$	$\infty$ 0 3 5	8 0 3 5
$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2	5 10 0 2
3 8 13 0	3 8 11 0	3 8 11 0	3 8 11 0

d=	k=1	k=2	k=3	k=4
0 5 10 1	0 5 8 1	0 5 8 1	0 5 8 1	0 5 8 1
$\infty$ 0 3 $\infty$	$\infty$ 0 3 $\infty$	$\infty$ 0 3 5	8 0 3 5	8 0 3 5
$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2	$\infty$ $\infty$ 0 2	5 10 0 2	5 10 0 2
3 8 13 0	3 8 11 0	3 8 11 0	3 8 11 0	3 8 11 0

p=	k=1	k=2	k=3	k=4
0 1 1 1	0 1 2 1	0 1 2 1	0 1 2 1	0 1 2 1
0 0 2 0	0 0 2 0	0 0 2 3	4 0 2 3	4 0 2 3
0 0 0 3	0 0 0 3	0 0 0 3	4 1 0 3	4 1 0 3
4 1 1 0	4 1 2 0	4 1 2 0	4 1 2 0	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) ?

# **Aplicație**

## **Închiderea tranzitivă a unui graf orientat**

### **Algoritmul Roy–Warhsall**

# Roy–Warshall

- ▶ **Aplicație:** Închiderea tranzitivă a unui graf orientat  $G=(V, E)$  (**!!!neponderat**):

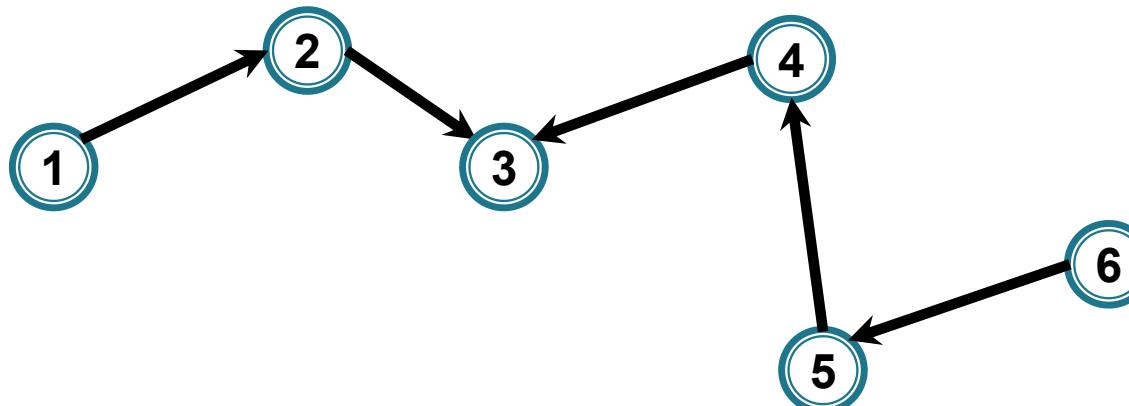
$G^* = (V, E^*)$ , unde

$E^* = \{(i, j) \mid \text{există drum (de lungime minim 1) de la } i \text{ la } j \text{ în } G\}$

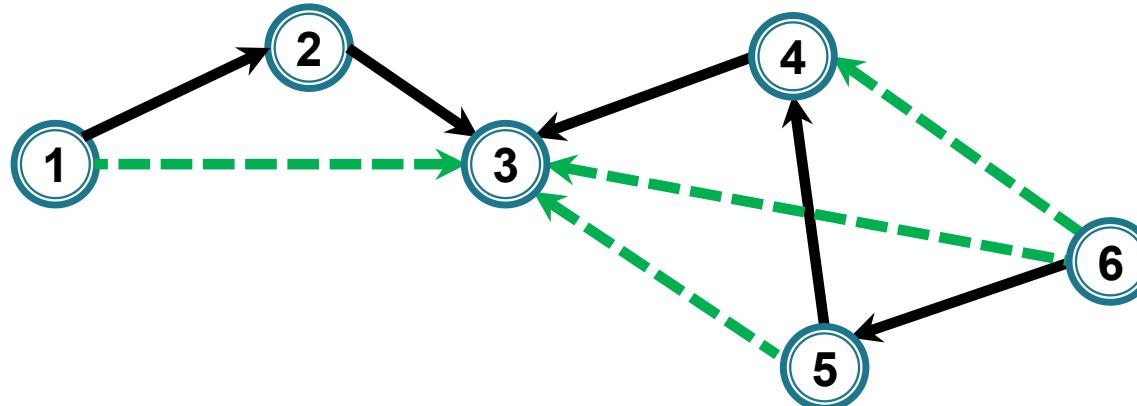
- **Utilitate:**
  - grupări de obiecte aflate în relație (directă sau indirectă): optimizări în baze de date, analize în rețele, logică

# Roy–Warshall

## Exemplul 1

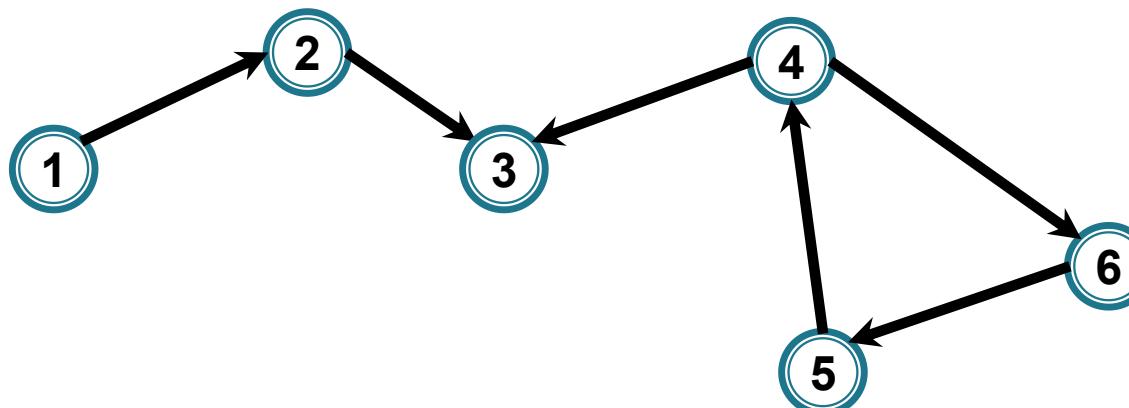


Închiderea tranzitivă

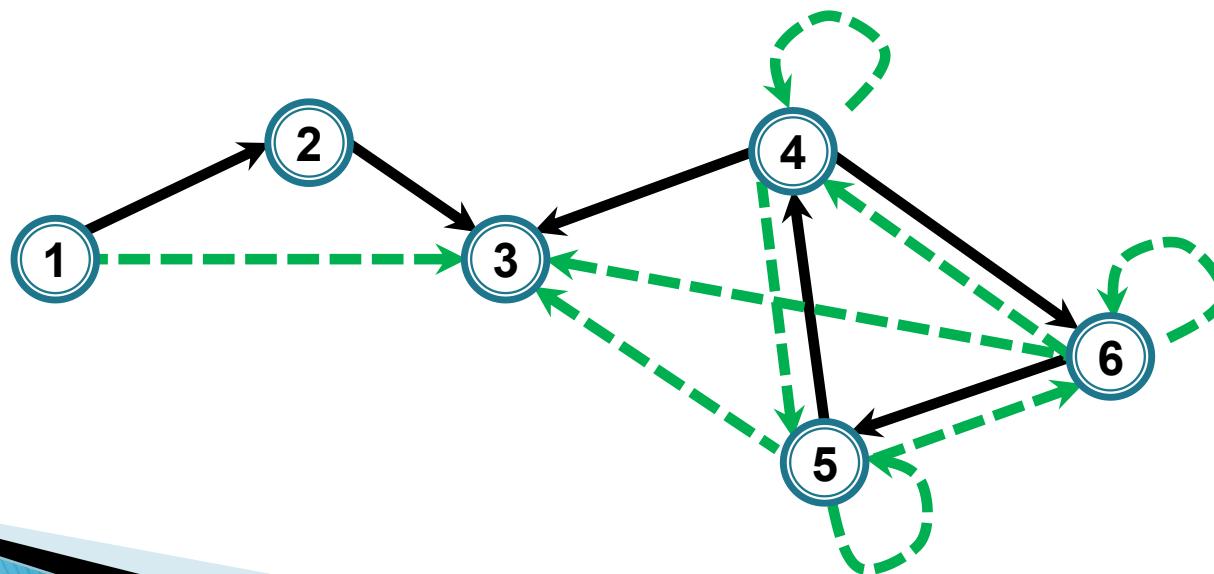


# Roy–Warshall

## Exemplul 2



Închiderea tranzitivă



# Roy–Warshall

- ▶ Închiderea tranzitivă  $\Leftrightarrow$  calculăm matricea existenței drumurilor (matricea de adiacență a înciderii tranzitive)

$D = (d_{ij})_{i,j=1..n} :$

$d_{ij} = 1$ , dacă există drum nevid de la i la j  
 $0$ , altfel

# Roy–Warshall

initial d = matricea de adiacenta

```
for (k=1 ; k<=n ; k++)
```

```
for (i=1 ; i<=n ; i++)
```

```
for (j=1 ; j<=n ; j++)
```

**d[i][j] = d[i][j] ∨ (d[i][k] ∧ d[k][j]);**

SAU

SII

# Roy-Warshall

## ▶ Observație

Dacă A este matricea de adiacență a unui graf și

$A^k = (a_{ij}^k)_{i,j=1..n}$  : puterea k a matricei ( $k < n$ )

atunci  $a_{ij}^k$  = numărul de drumuri distințe de lungime k de la i la j (!nu neapărat elementare)

**Demonstrație – Inducție. Temă**

# Roy-Warshall

## ▶ Observație

Dacă A este matricea de adiacență a unui graf și

$A^k = (a_{ij}^k)_{i,j=1..n}$  : puterea k a matricei ( $k < n$ )

atunci  $a_{ij}^k$  = numărul de drumuri distințe de lungime k de la i 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