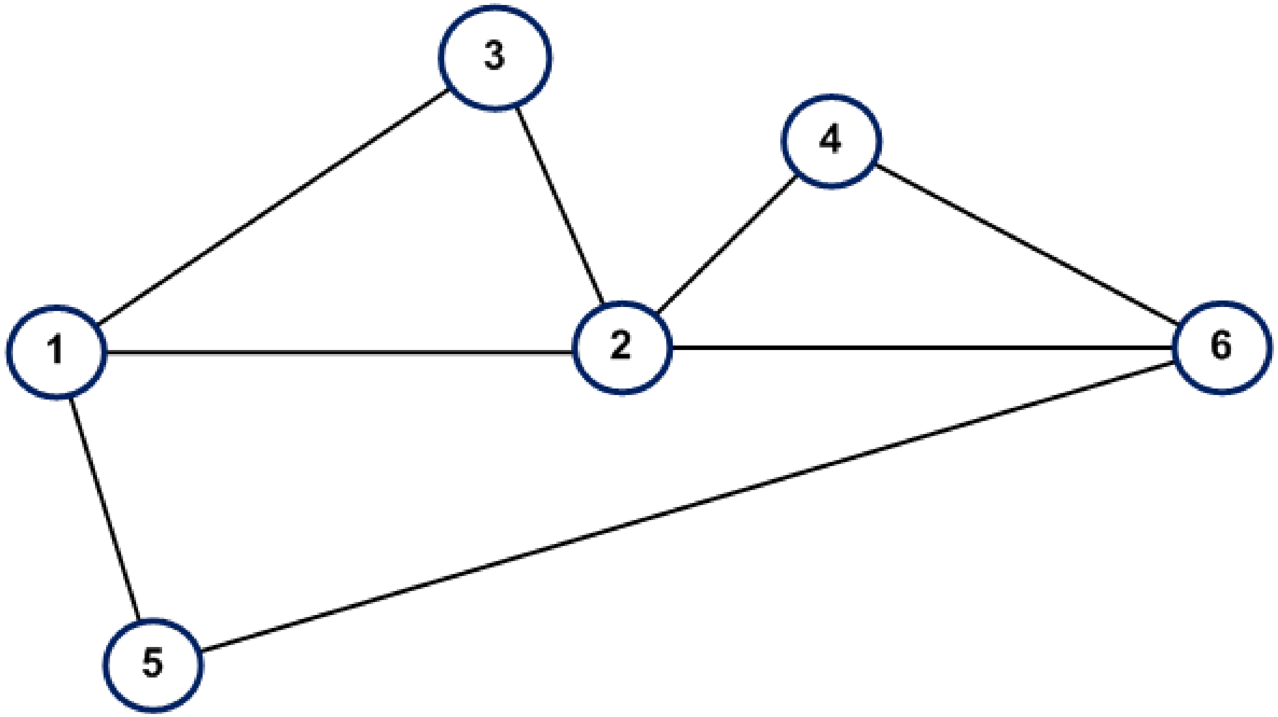
Modalităţi de reprezentare a grafurilor

Reprezentarea grafurilor

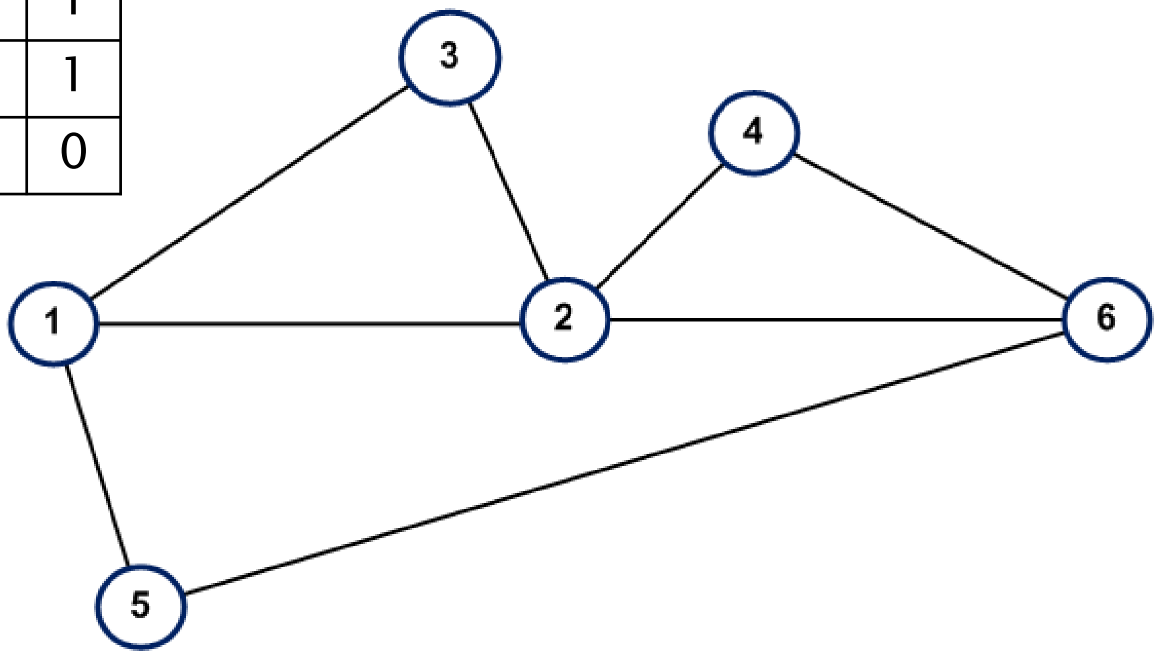
- Matrice de adiacenţă
- Liste de adiacenţă
- Listă de muchii/arce



Reprezentarea grafurilor

Matrice de adiacenţă

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	1	0	1	1	0	1
3	1	1	0	0	0	0
4	0	1	0	0	0	1
5	1	0	0	0	0	1
6	0	1	0	1	1	0

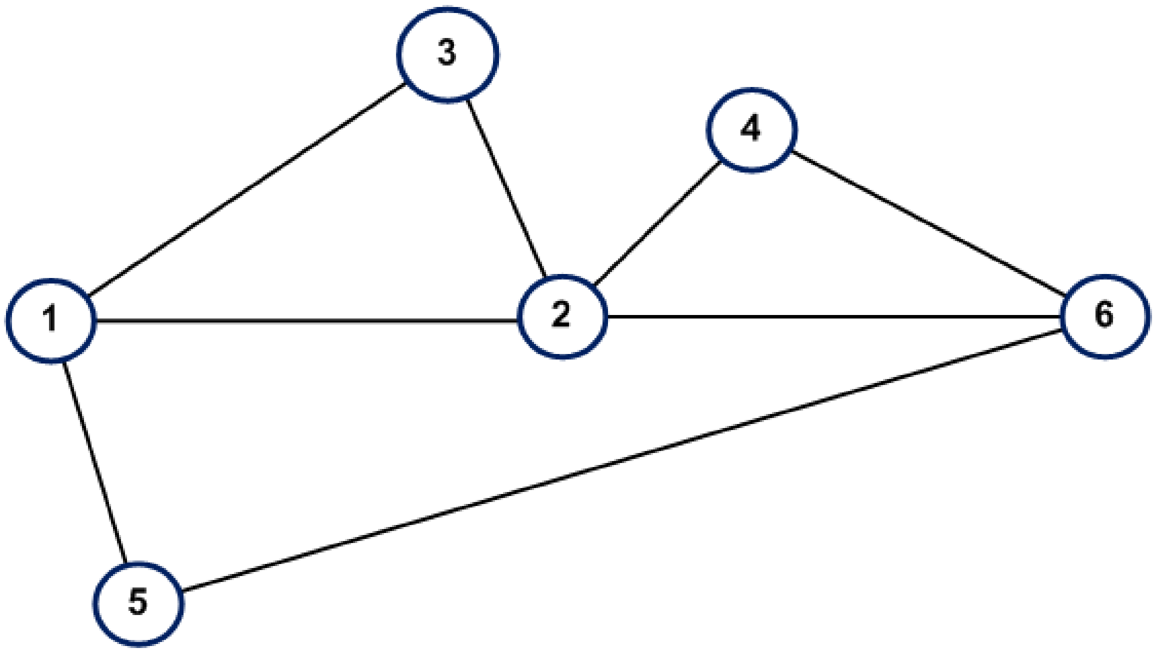


Reprezentarea grafurilor

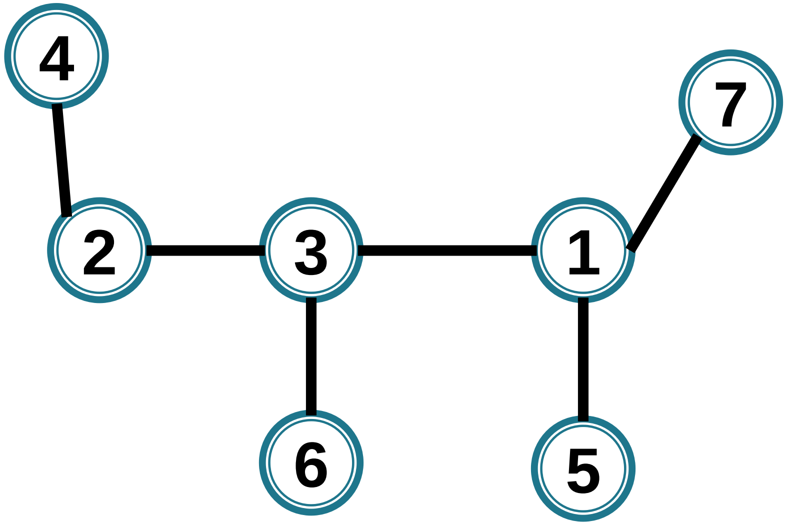
Lista de adiacenţă

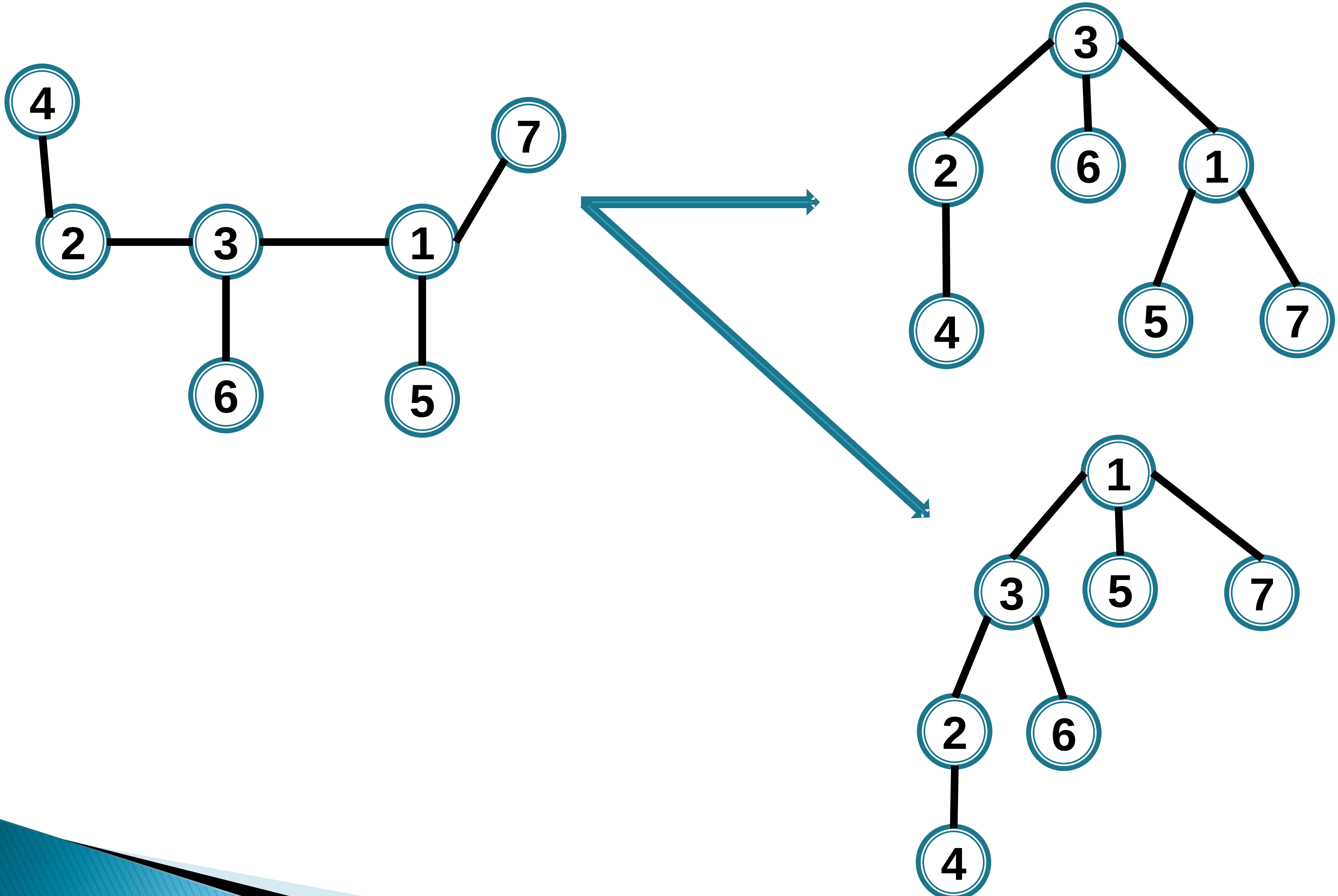
6 noduri:

- 1: 2, 3, 5
- 2: 1, 3, 4, 6
- 3: 1, 2
- 4: 2, 6
- 5: 1, 6
- 6: 2, 4, 5



Arbori cu rădăcină


4
7
2
3
1
6
5


4
7
2
3
1
6
5
3
2
6
1
4
5
7
1
3
5
7
2
6
4

- # Noţiuni
  - ## Arbore cu rădăcină
    - După fixarea unei rădăcini, arborele se aşează pe niveluri
    - Nivelul unui nod v,

      `niv[v]` = distanţa de la rădăcină la nodul v
    - În arborele cu rădăcină există muchii doar între niveluri consecutive

Noțiuni

- **Tată:** x este tată al lui y dacă există muchie de la x la y şi x se află în arbore pe un nivel cu 1 mai mic decât y

- **Fiu:** y este fiu al lui x $\Leftrightarrow$ x este tată al lui y

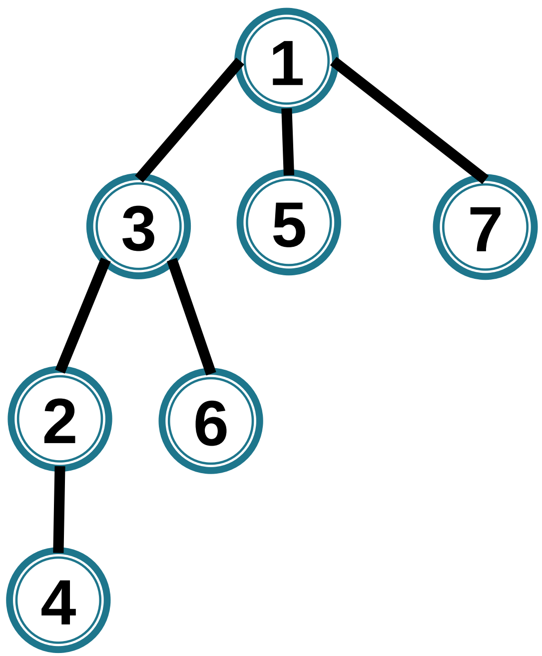
- **Ascendent (strămoș):** x este ascendent a lui y dacă x aparţine unicului lanţ elementar de la y la rădăcină (echivalent, dacă există un lanţ de la y la x care trece prin noduri situate pe niveluri din ce în ce mai mici)

- **Descendent (urmaș):** y este descendent al lui x $\Leftrightarrow$ x este ascendent a lui y

- **Frunză:** nod fără fii

Noțiuni

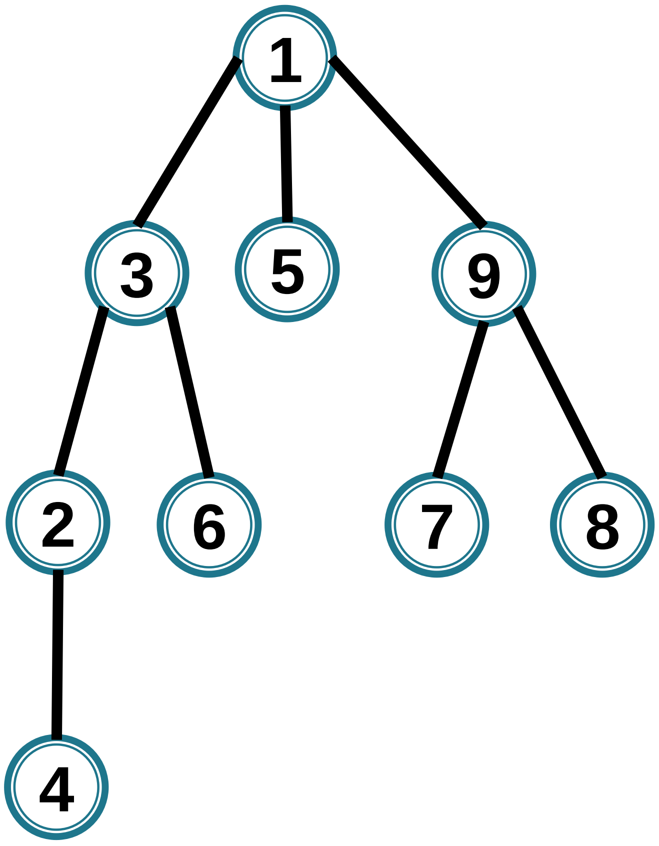
- **Fiu**: fii lui 3 sunt 2 şi 6
- **Tată**: 1 este tatăl lui 7
- **Ascendent**: ascendenţii lui 6 sunt 3 şi 1
- **Descendent**: descendenţii lui 3 sunt 2, 6 şi 4
- **Frunză**: frunzele arborelui sunt 4, 6, 5 şi 7



Modalităţi de reprezentare a arborilor cu rădăcină

Reprezentarea arborilor

- Vector tata
- Lista de fii



Vectorul tata

Folosind vectorul tata putem determina **lanțuri de la orice vârf x la rădăcină**, **urcând** în arbore de la x la rădăcină

```
lant(x)
   cat timp x!=0 executa
      scrie x
      x = tata[x]
```

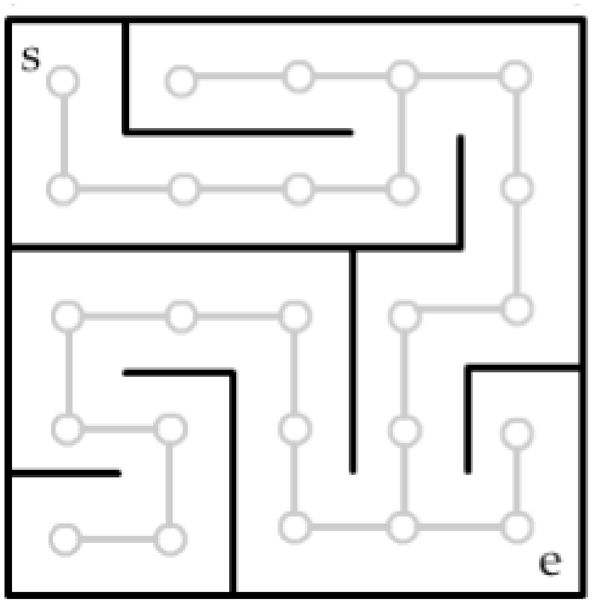
```
lantr(x)
   daca x!=0 atunci
        lantr(tata[x])
        scrie x
```

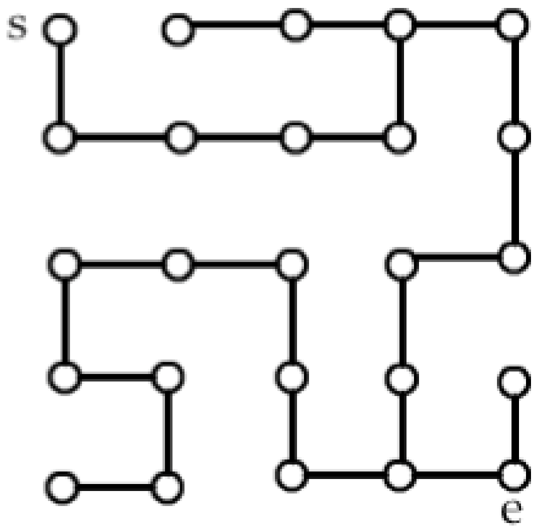
Parcurgerea grafurilor

Exemple de aplicații

Drumuri în labirint

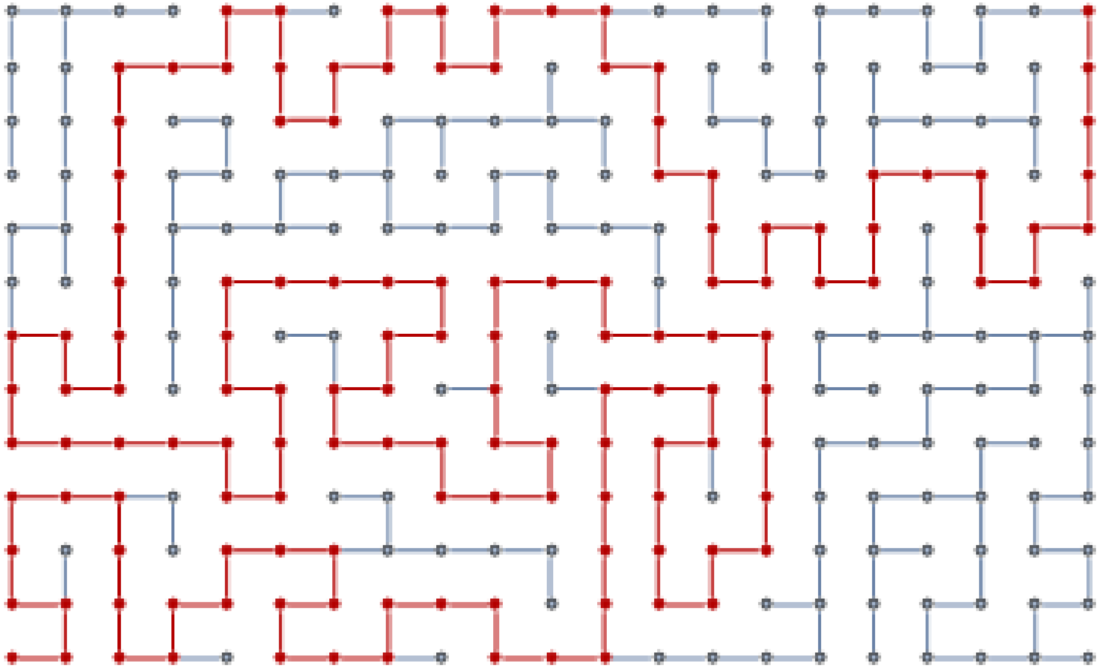
- Labirint – asociat un graf





http://www.cs.umd.edu/class/spring2019/cmsc132-020X-040X/Project8/proj8.html

Drumuri în labirint



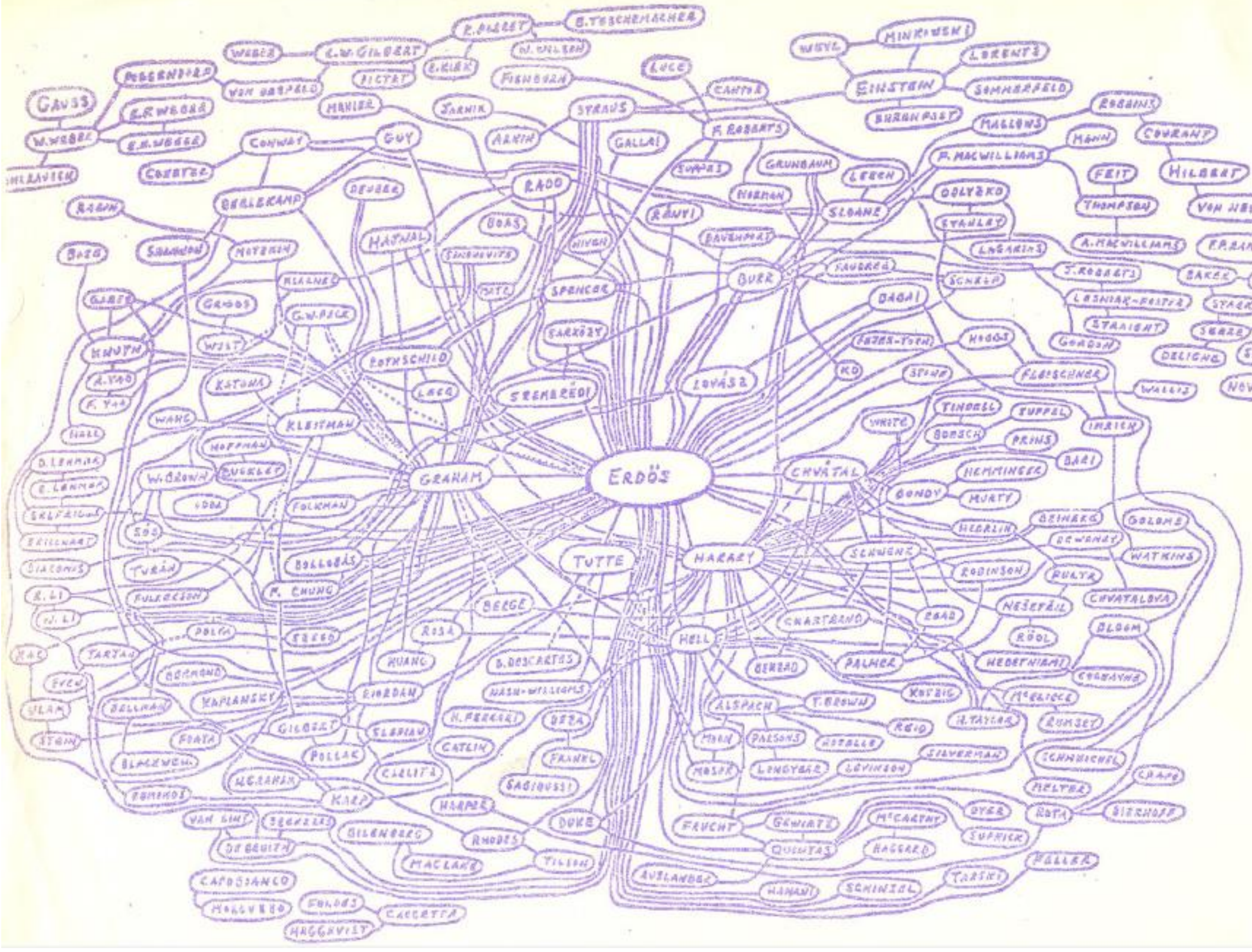
https://rosettacode.org/wiki/Maze_solving

Probleme de accesibilitate și distanțe

- Reteaua de colaborare cu Erdos
  - Muchii – a colaborat (publicat impreuna) cu
  - Noduri – matematicieni

- “Distanta” fata de Erdos? – Numărul Erdos

- Generalizare – distanța între 2 autori într-o rețea de colaborare

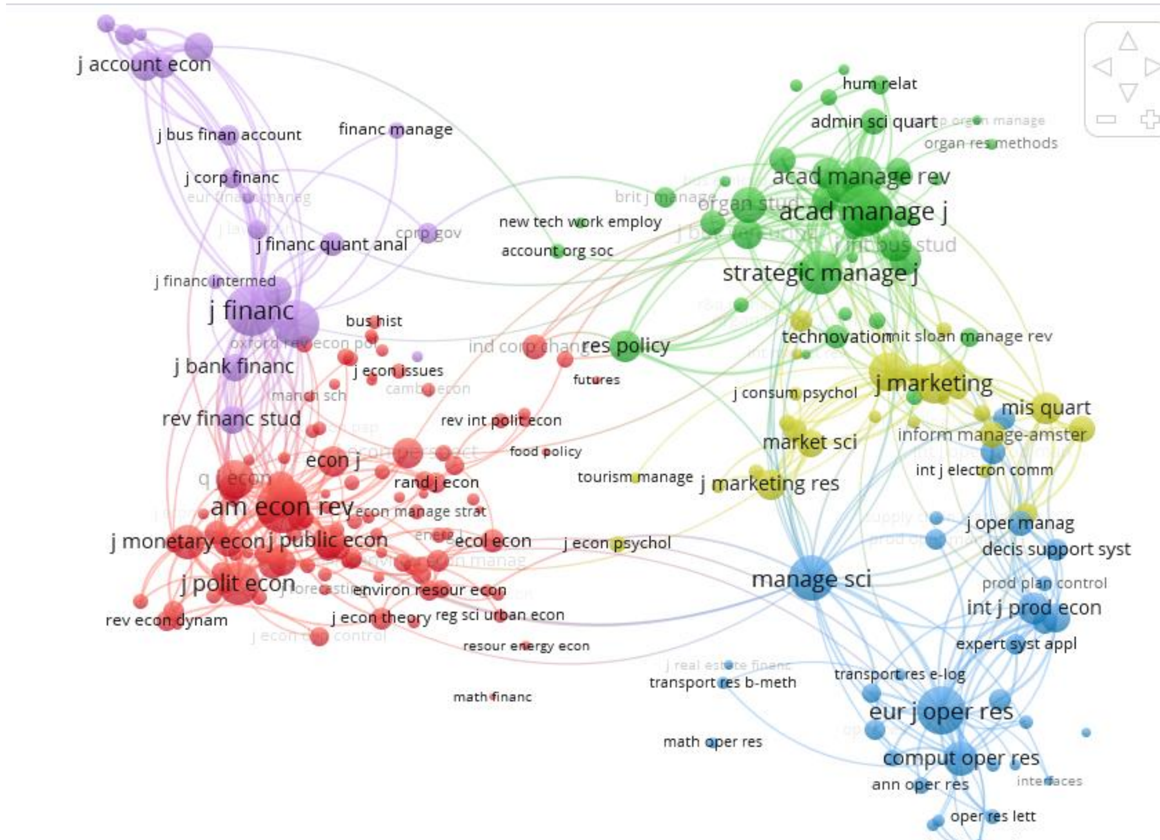


Ron Graham (alias Tom Odda).

Retea de colaborări, citări

Probleme relevante

- Distanțe
- Componenta unui autor/jurnal/pagina web
- componenta dominantă (de dimensiune maximă, gigant): conexa (neorientat), tare conexă (orientat) etc

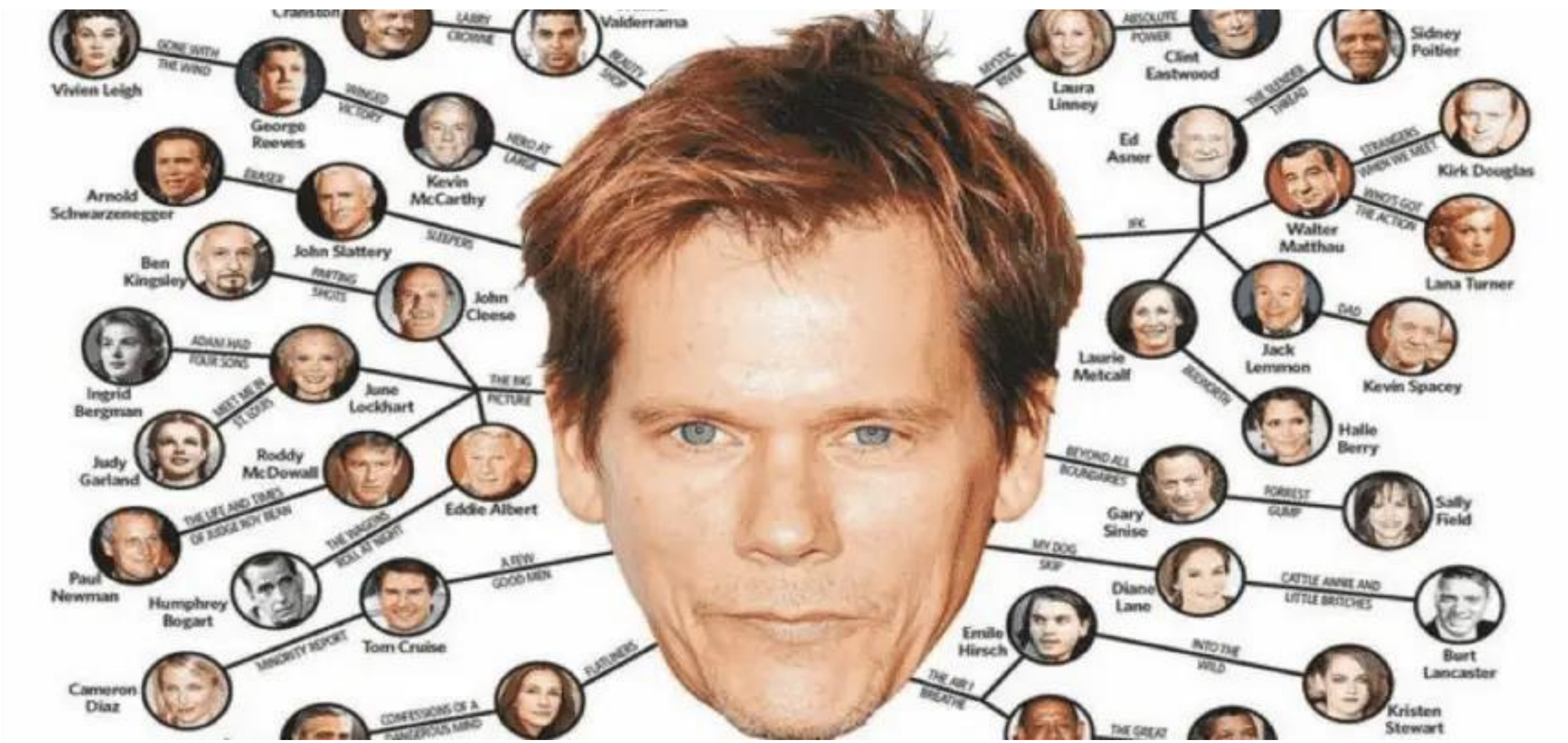


https://www.vosviewer.com/

Retea de colaborări, citări

Numarul Kavin Bacon

- Noduri – actori
- Muchii – au jucat impreuna
- Six Degrees of Kevin Bacon

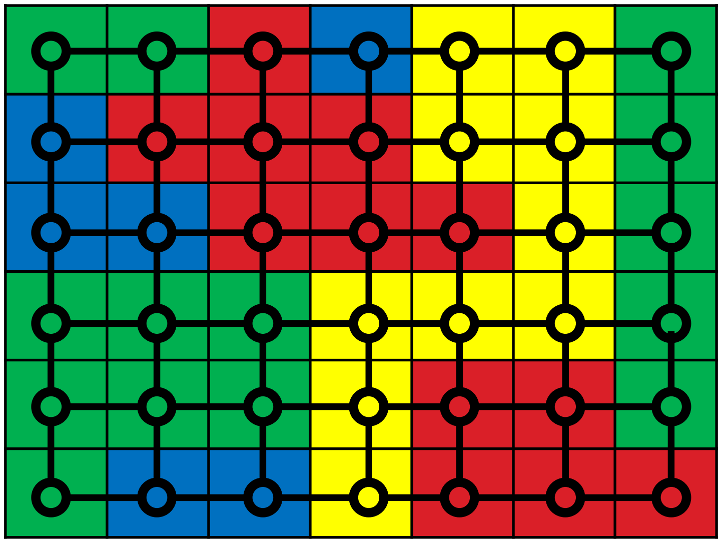


https://www.iceinstitute.org/blog/2019/4/1/six-degrees-of-kevin-bacon-education-edition

Procesarea imaginilor

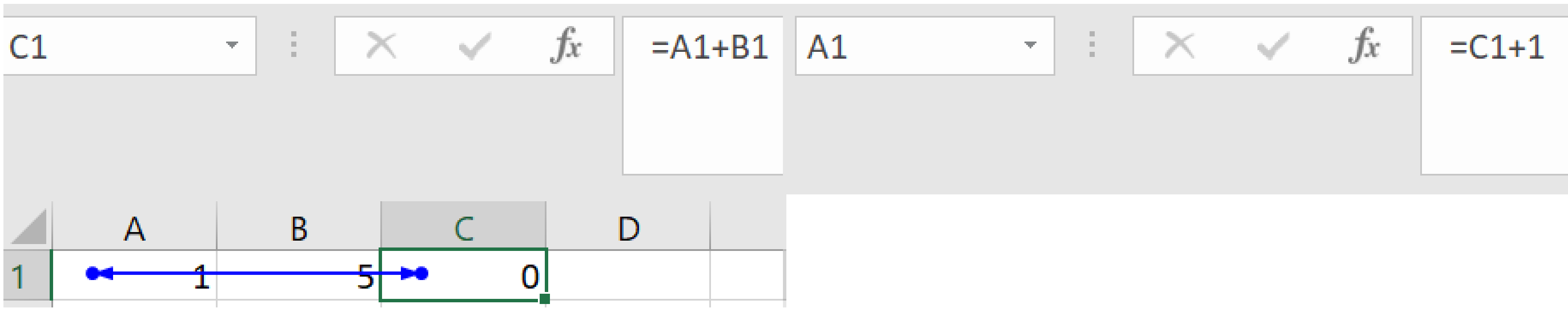
Algoritmi de umplere (fill)

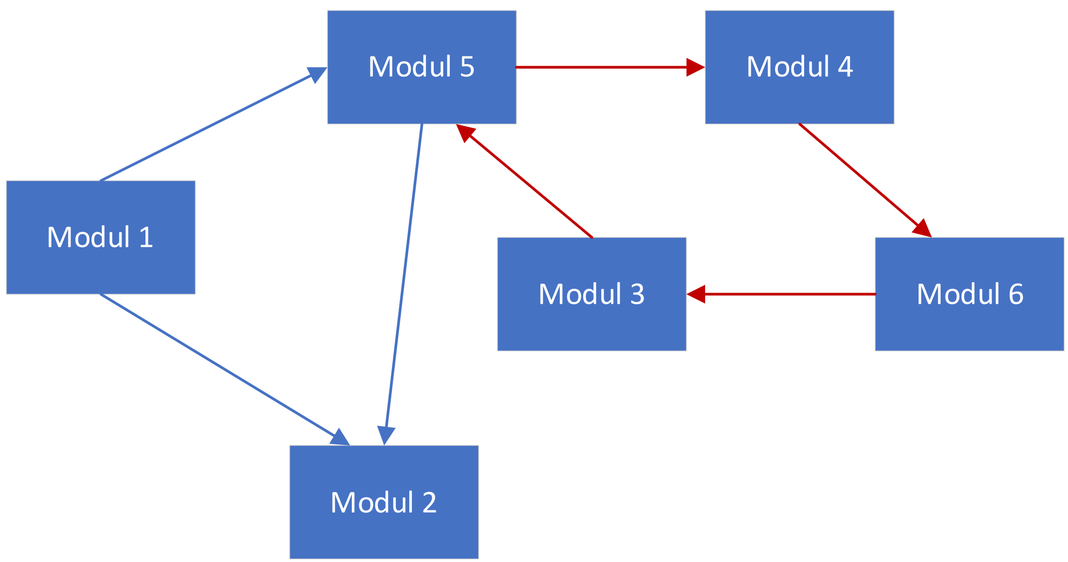
- Imagine –asociat graf grid
- Pornind de la un pixel de o anumita culoare c – determinăm componenta lui de culoare c (obiect)



Eroare formula in excel

Existența de dependențe circulare în formule, module:





Tipuri de parcurgeri ale unui graf

Parcurgerea grafurilor

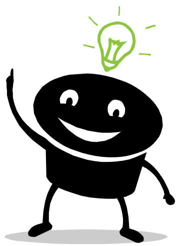
Dat un graf G și un vârf s, care sunt toate vârfurile accesibile din s?

- Un vârf v este **accesibil din s** dacă există un drum/lanț de la s la v în G.

Parcurgerea grafurilor

Parcurgere = o modalitate prin care, plecând de la un vârf de start și mergând pe arce/muchii să ajungem la toate vârfurile accesibile din s

Parcurgerea grafurilor



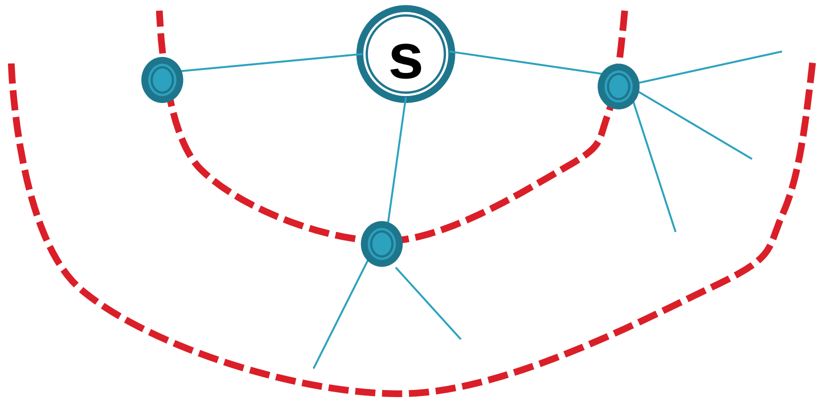
Idee: Dacă

- u este **accesibil din s**
- $uv \in E(G)$

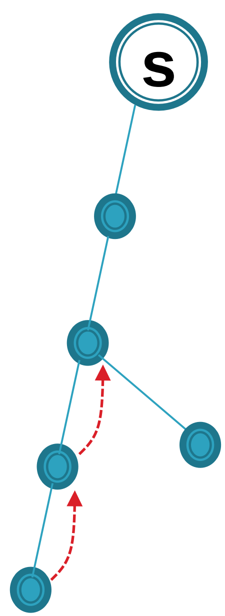
atunci v este accesibil din s.

Parcurgerea grafurilor

- Parcurgerea în lățime (BF = breadth first)



- Parcurgerea în adâncime (DF = depth first)

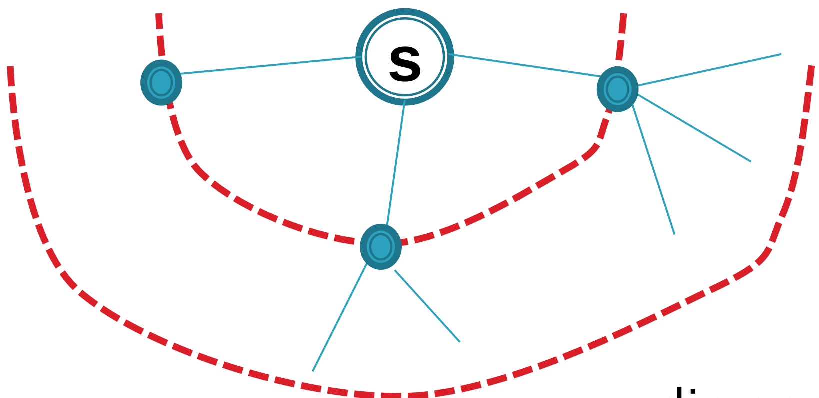


Parcurgerea în lăţime

Parcurgerea grafurilor

- **Parcurgerea în lățime:** se vizitează
  - vârful de start **s**
  - vecinii acestuia
  - vecinii nevizitați ai acestora

etc

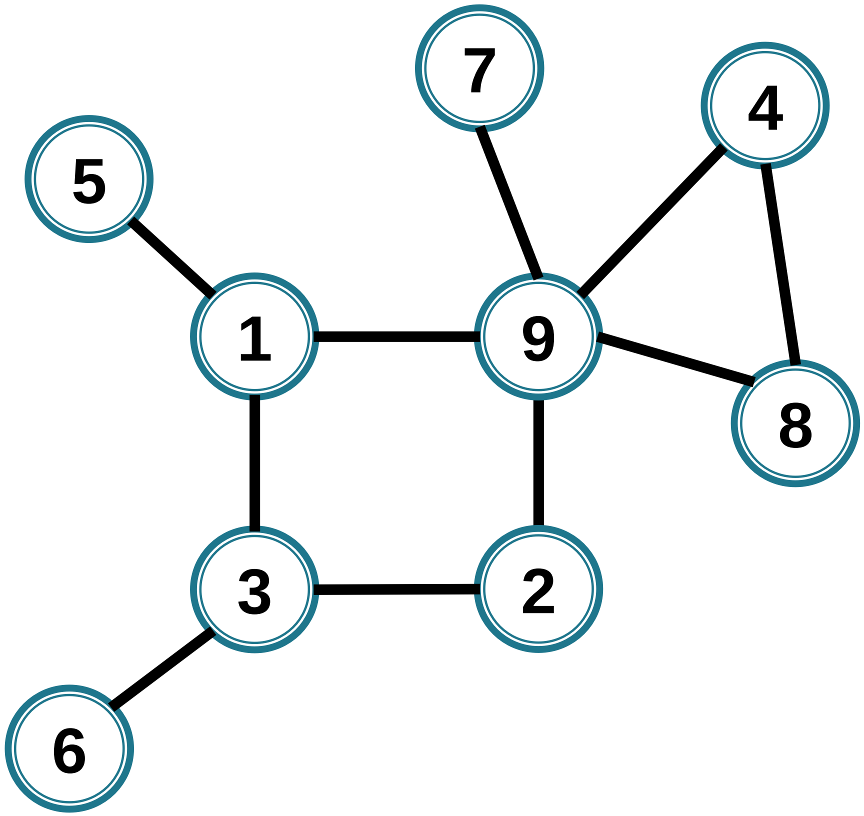


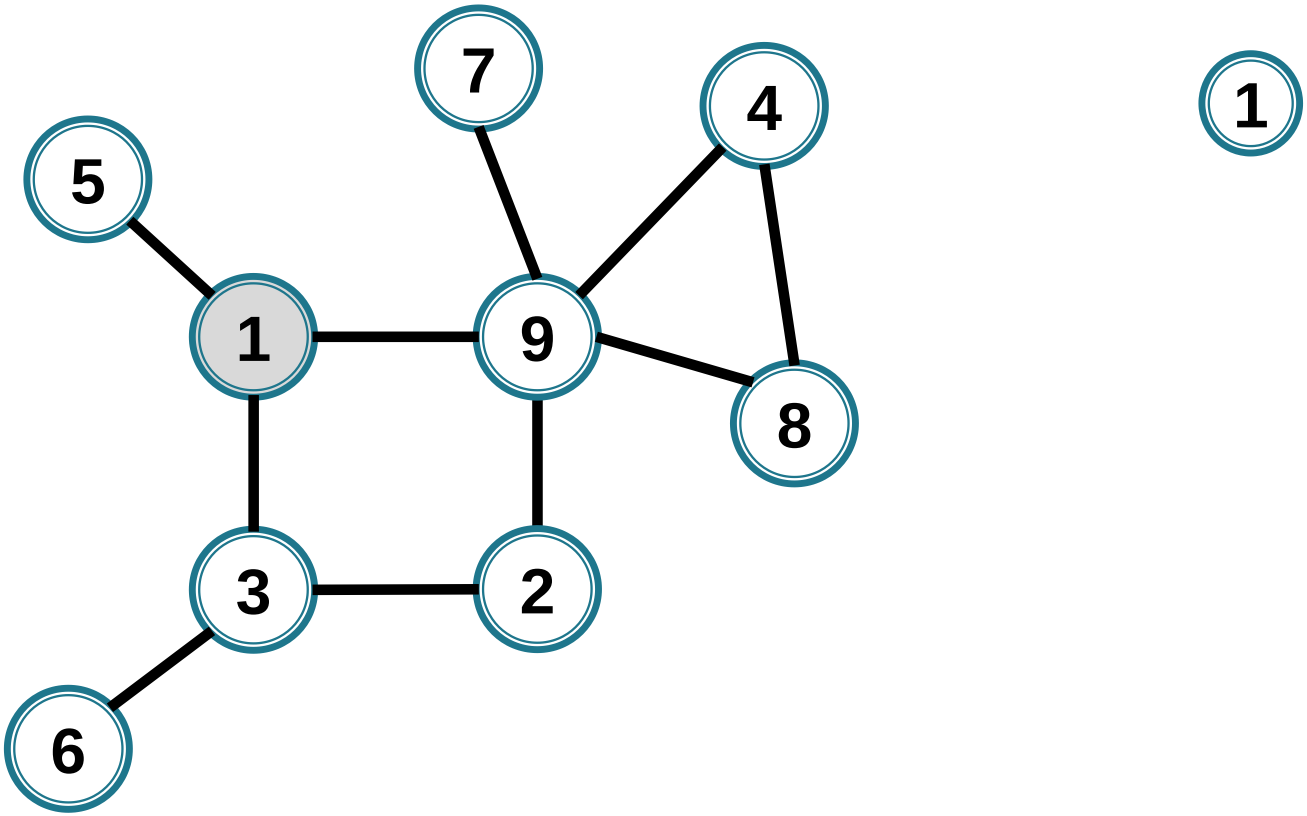
din aproape în aproape

Parcurgerea în lăţime

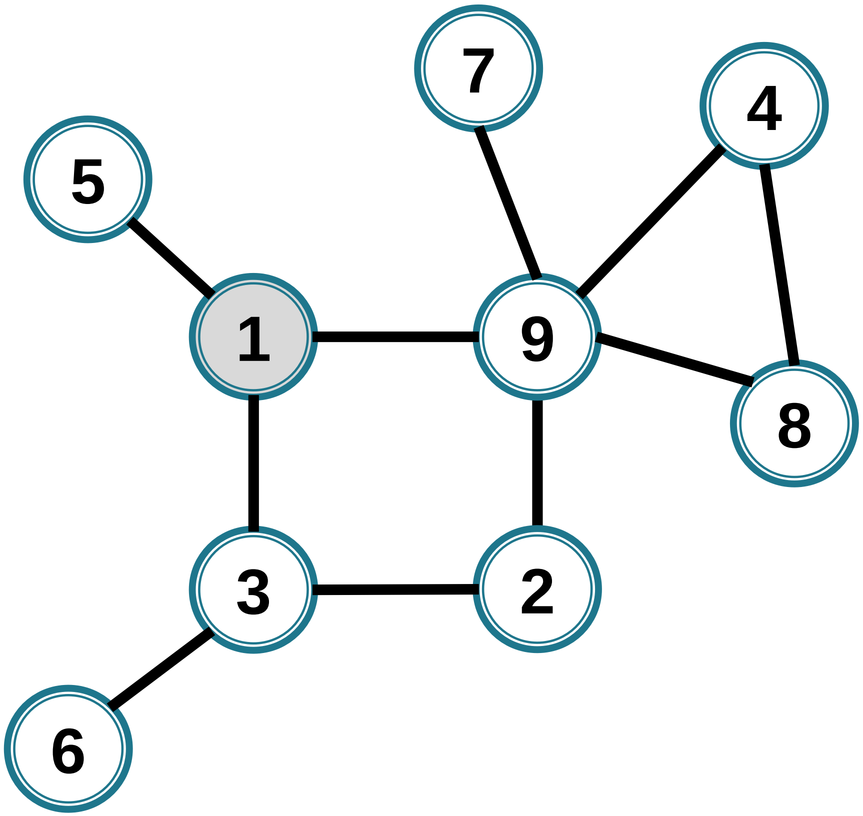
- Pentru gestionarea vârfurilor parcurse care mai pot avea vecini nevizitaţi – o structură de tip **coadă**

Exemplu
pentru graf neorientat

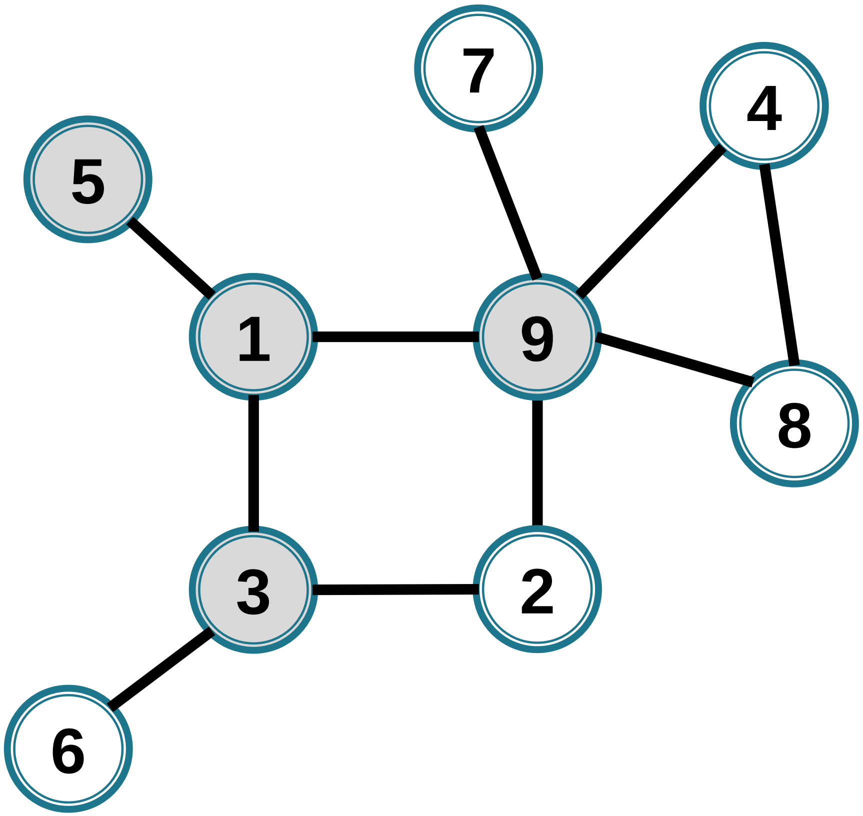

7
4
5
1
9
8
3
2
6

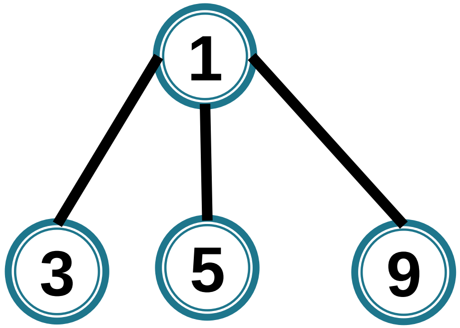


1

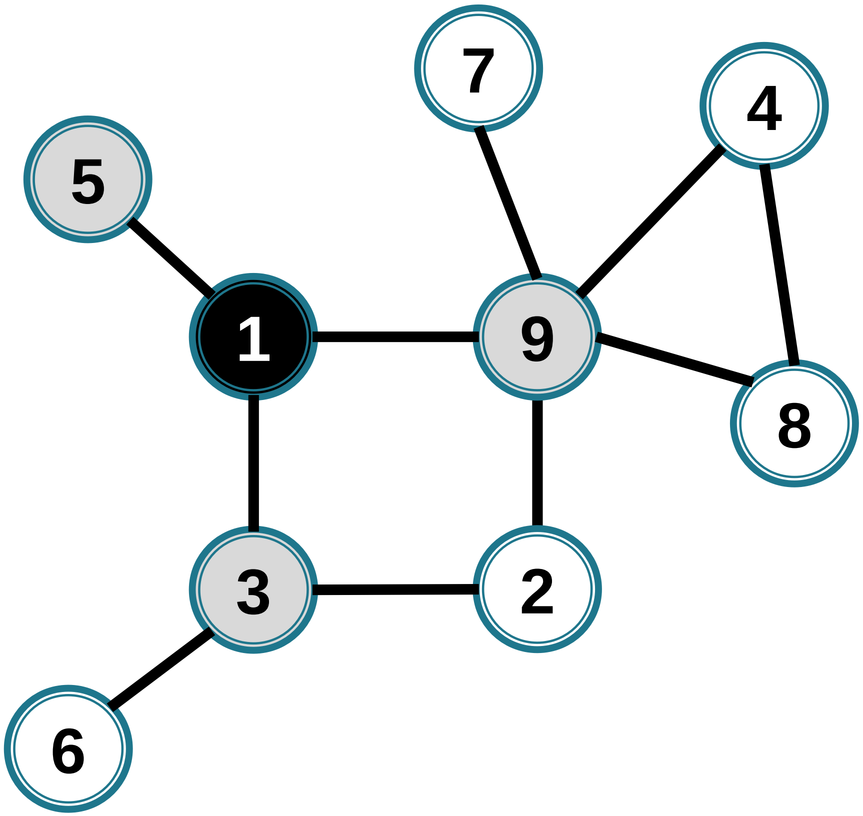


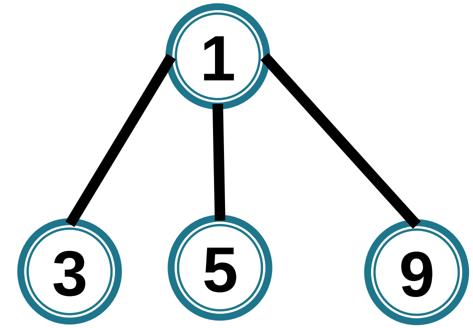
1



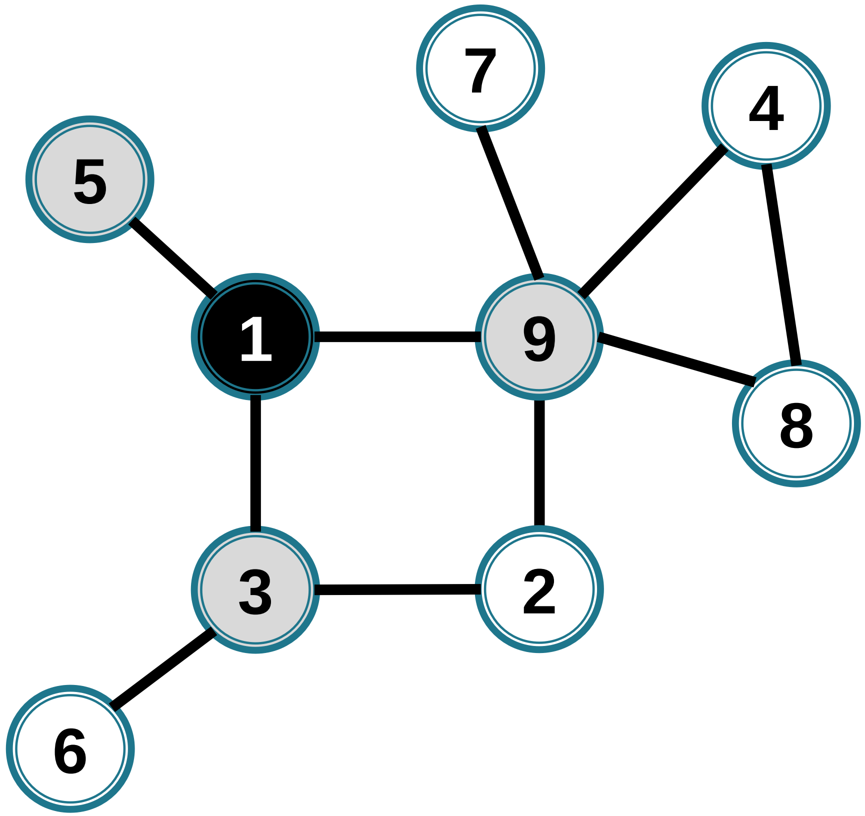


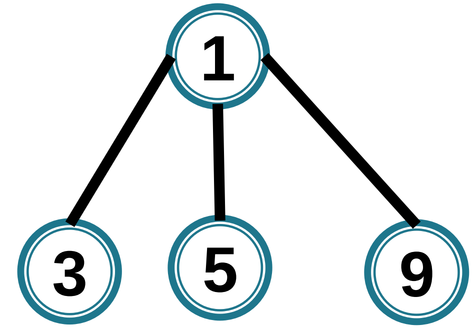
1 3 5 9



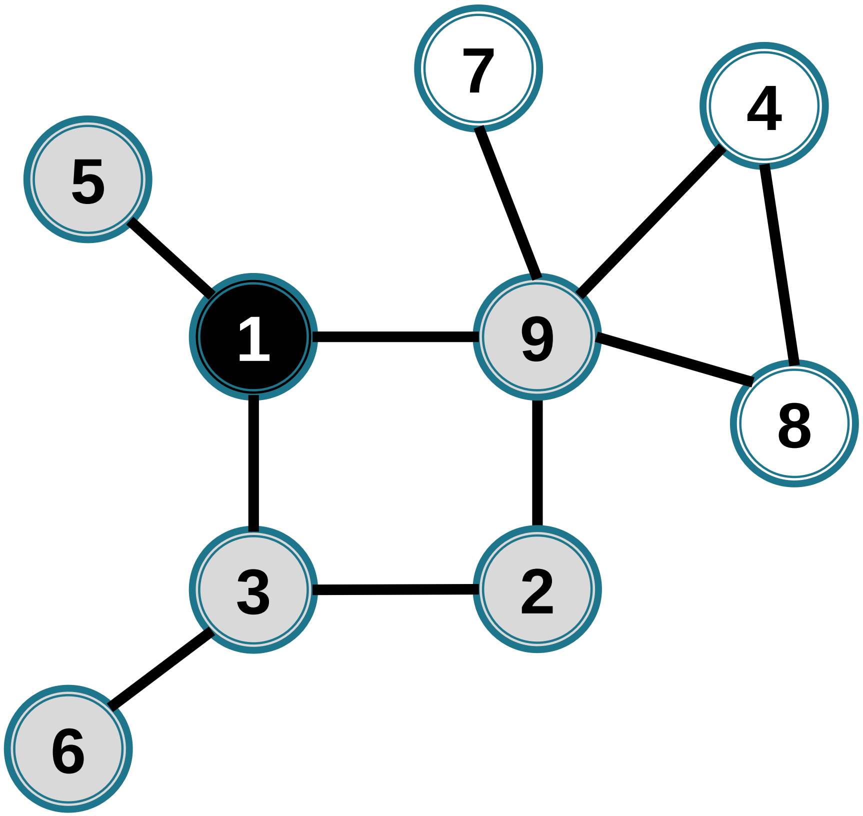


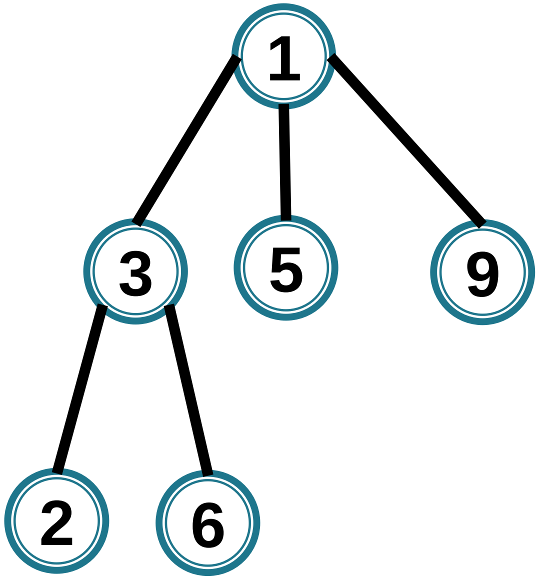
~~1~~ 3 5 9



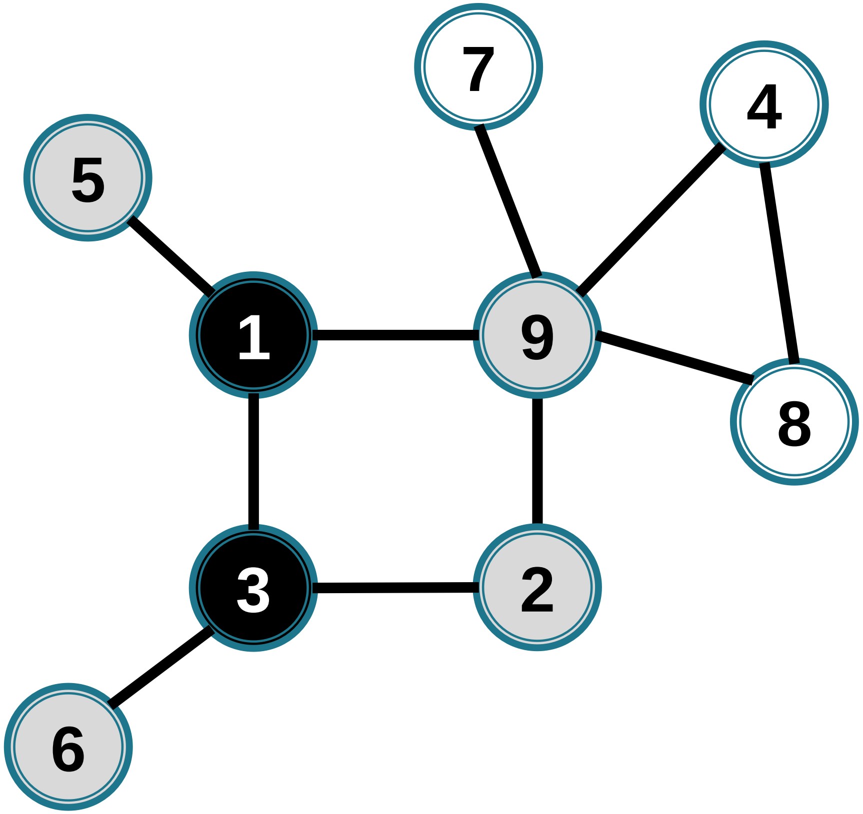


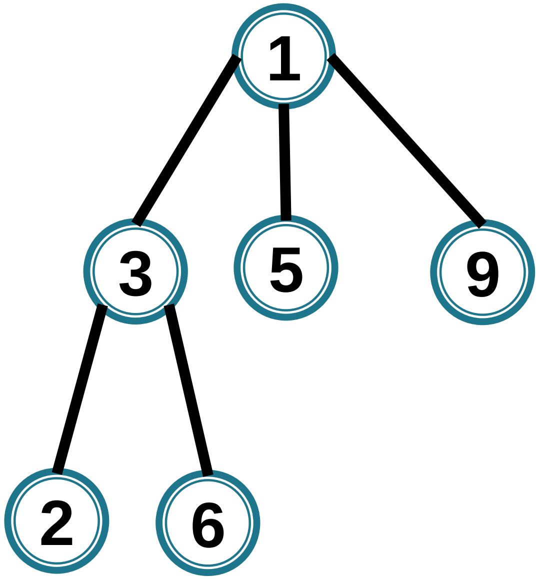
~~1~~ 3 5 9



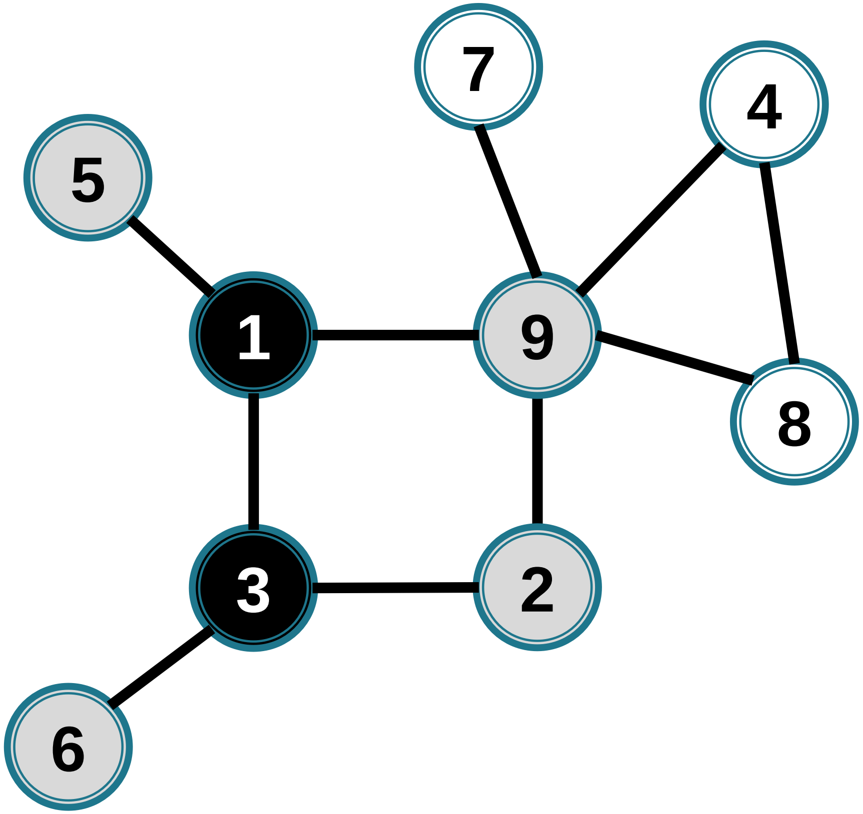


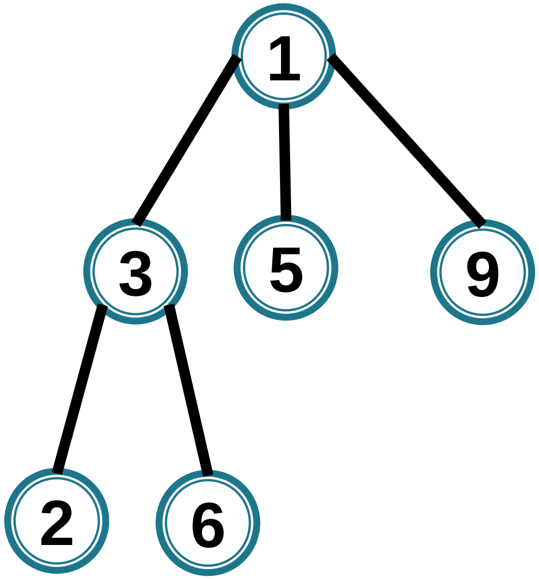
~~1~~ 3 5 9 2 6



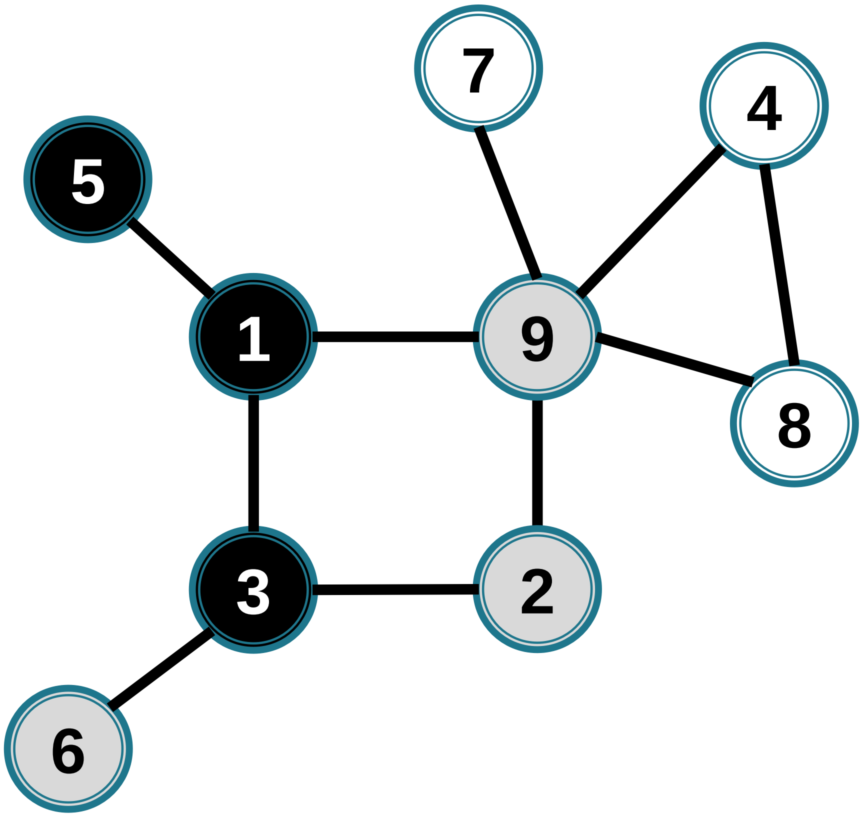


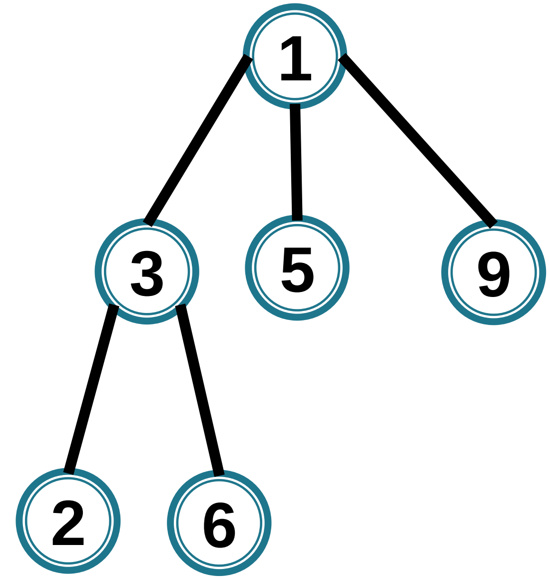
~~1~~ ~~3~~ 5 9 2 6



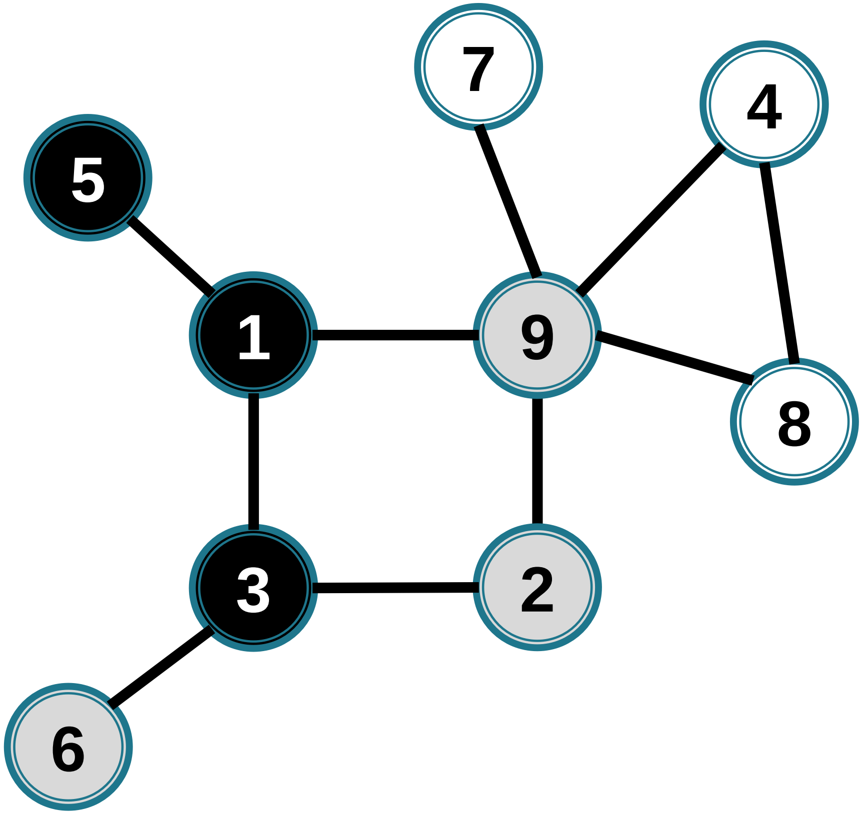


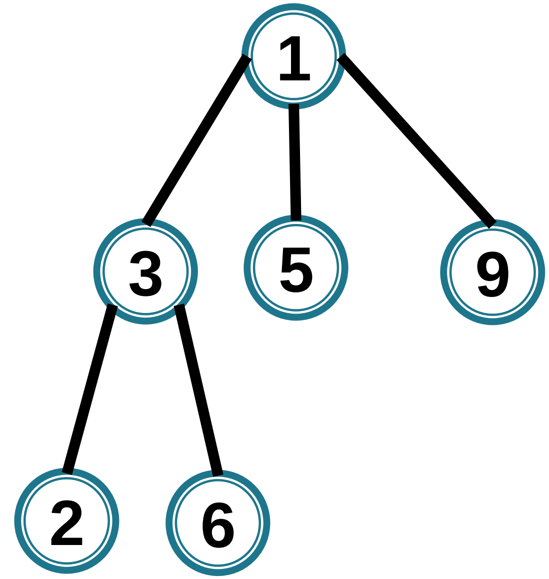
~~1~~ ~~3~~ 5 9 2 6



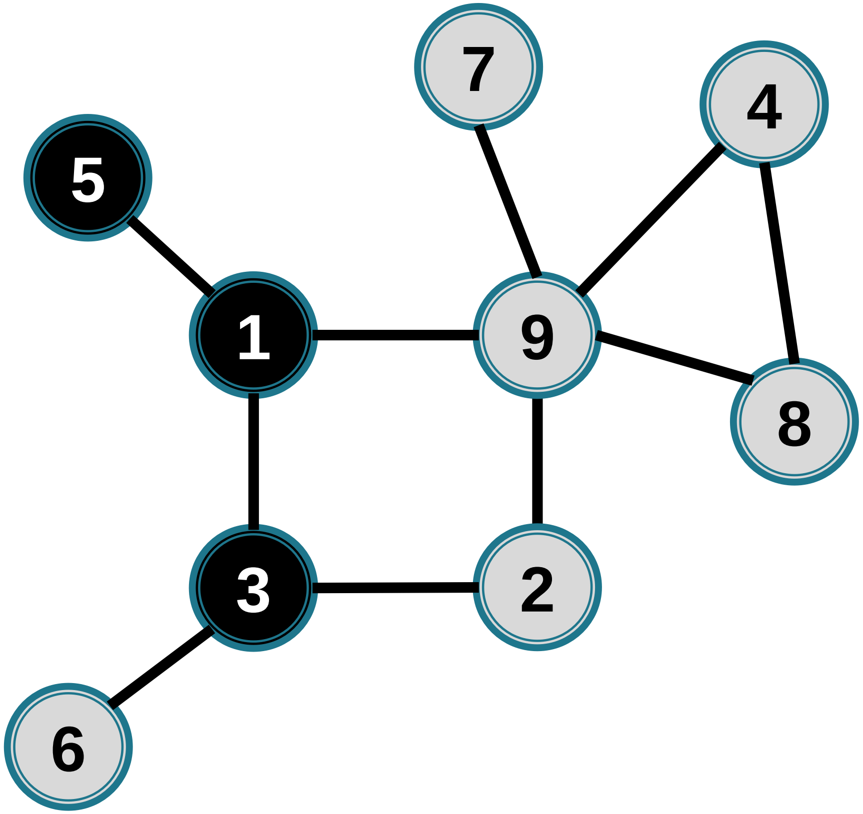


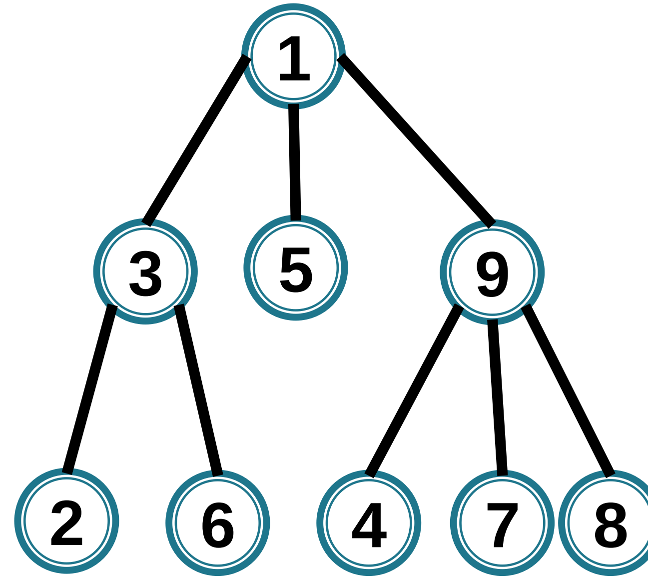
~~1~~ ~~3~~ ~~5~~ 9 2 6



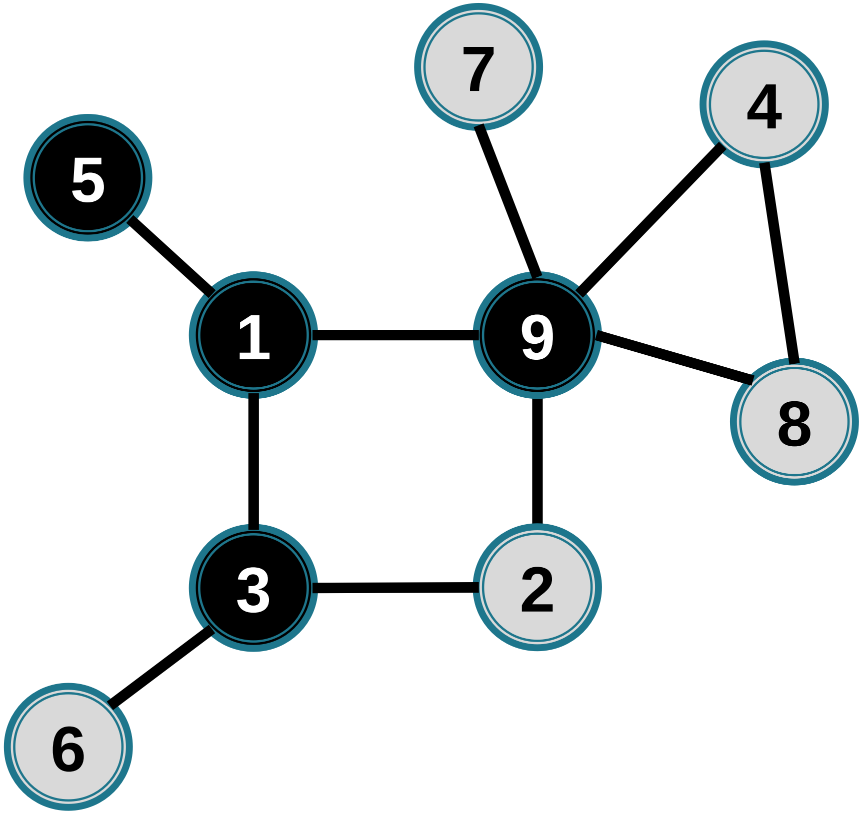


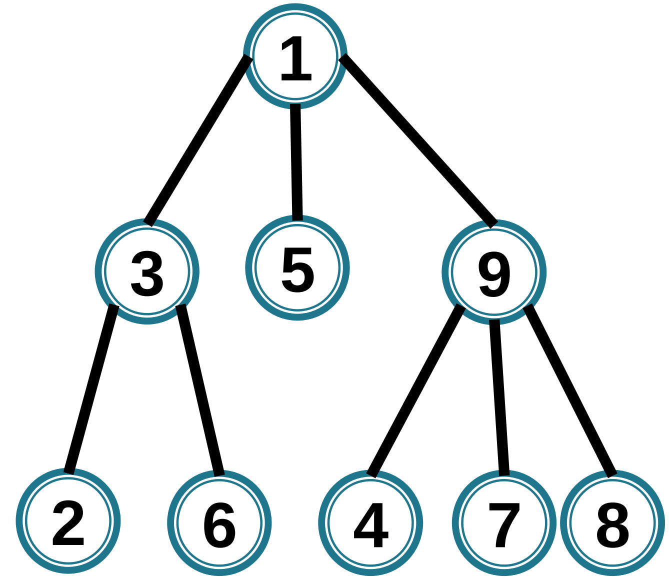
~~1~~ ~~3~~ ~~5~~ 9 2 6



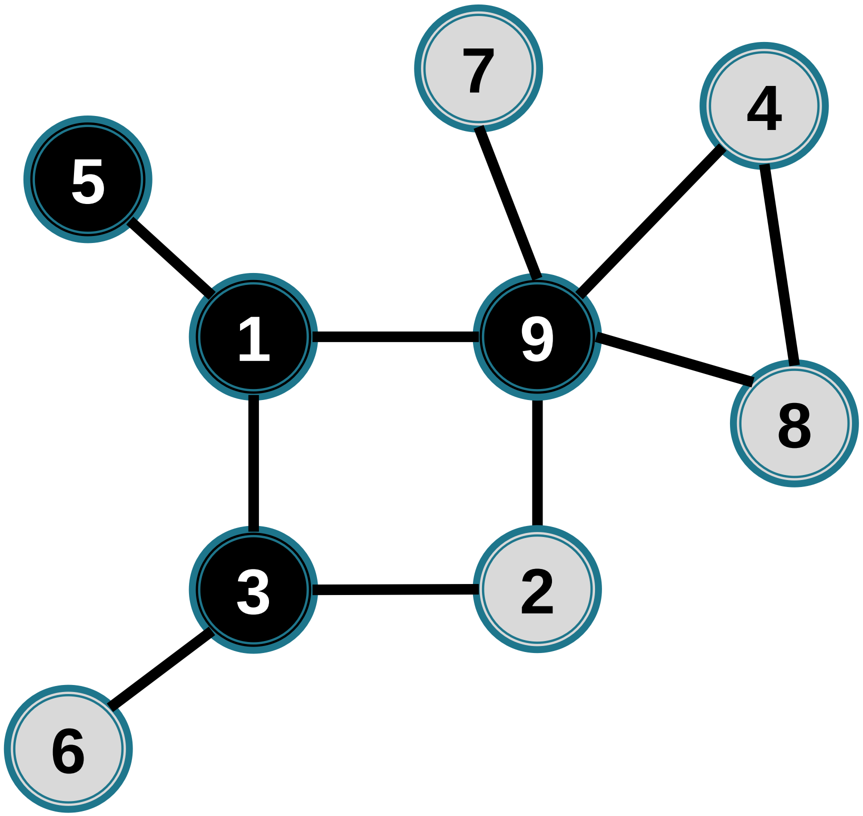


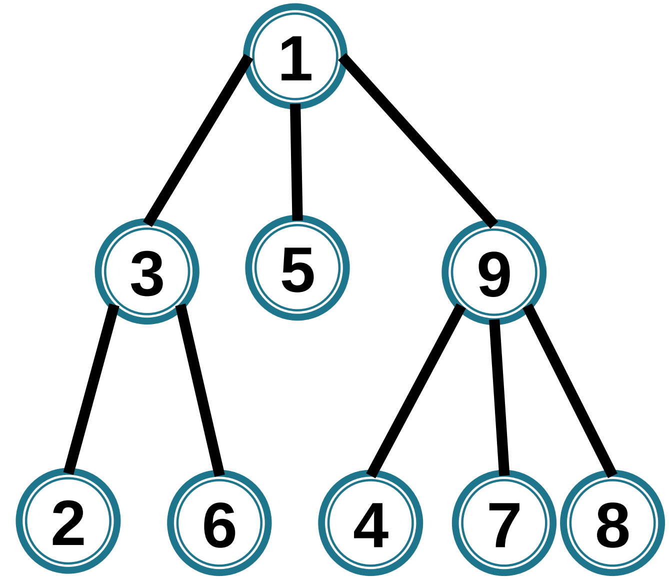
~~1~~ ~~3~~ ~~5~~ 9 2 6 4 7 8



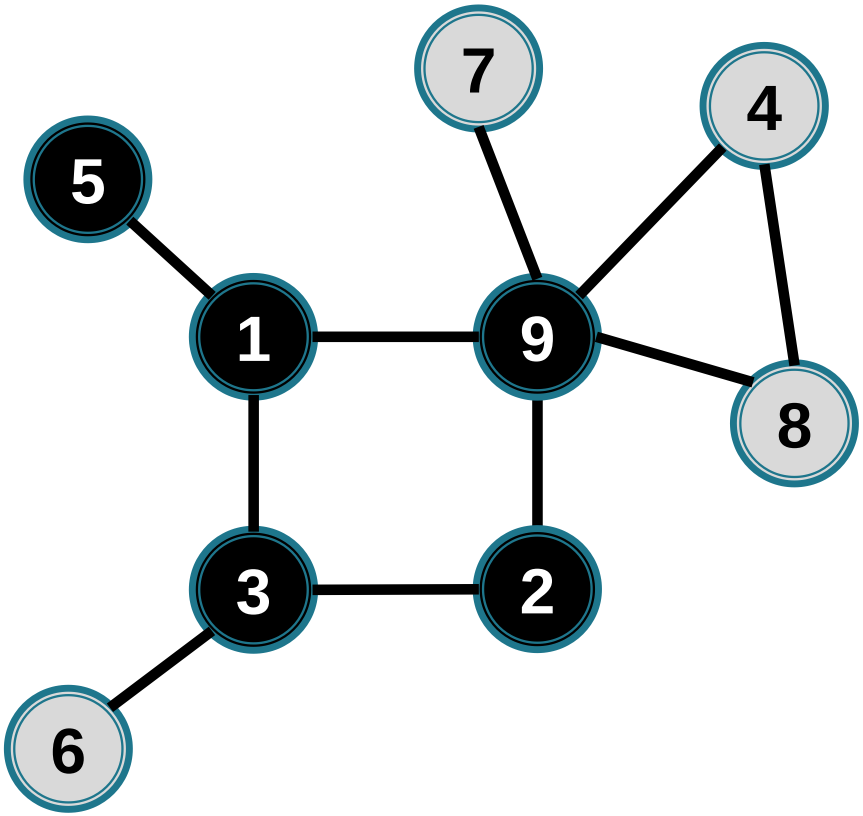


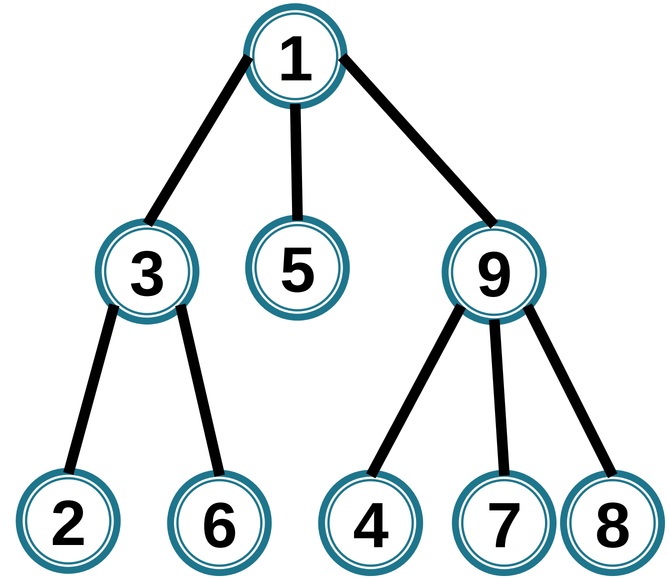
~~1~~ ~~3~~ ~~5~~ ~~9~~ 2 6 4 7 8



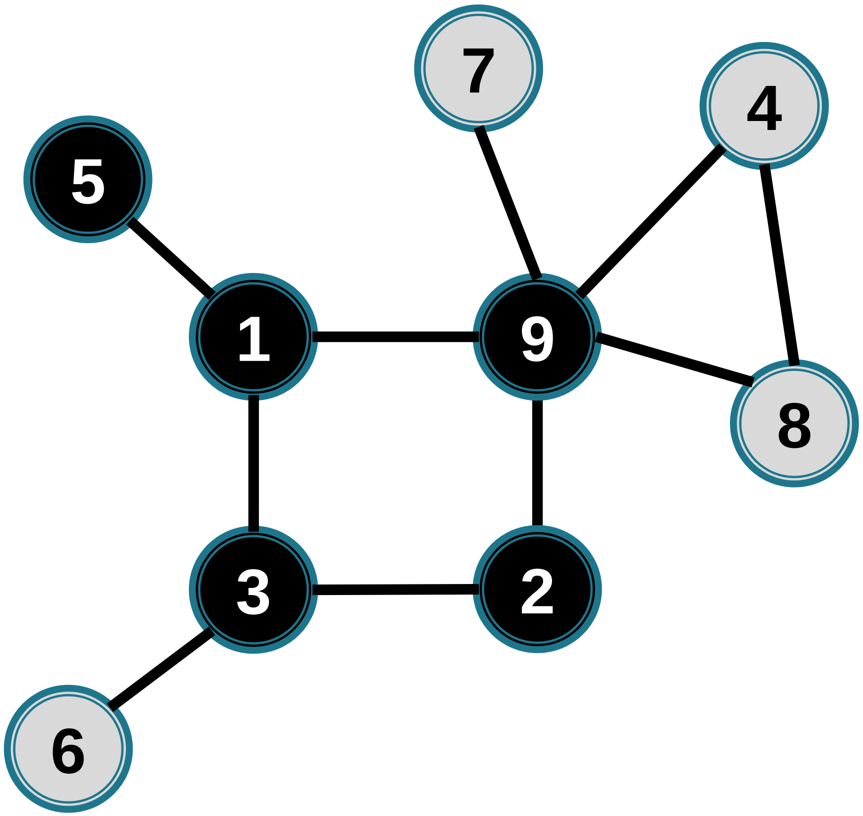


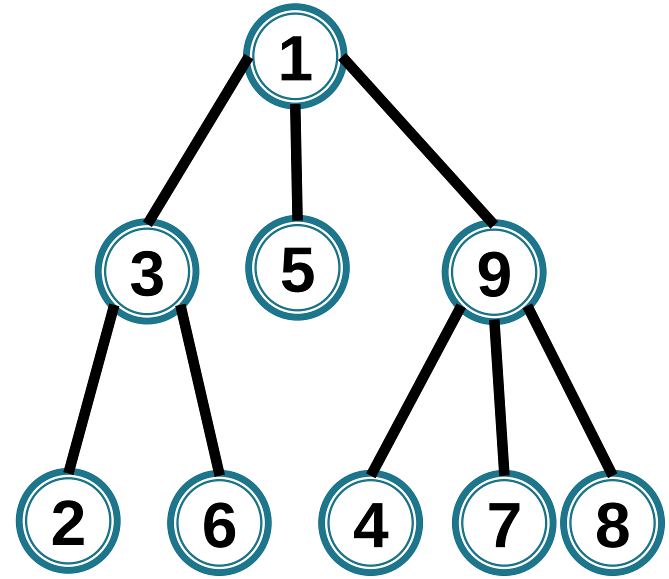
~~1~~ ~~3~~ ~~5~~ ~~9~~ 2 6 4 7 8



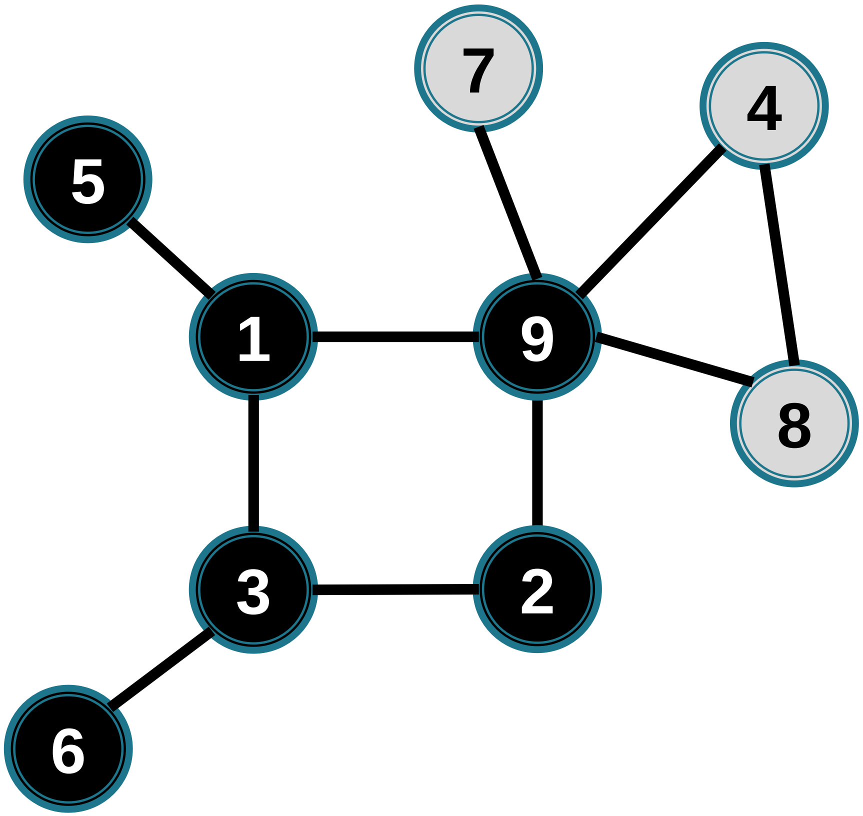


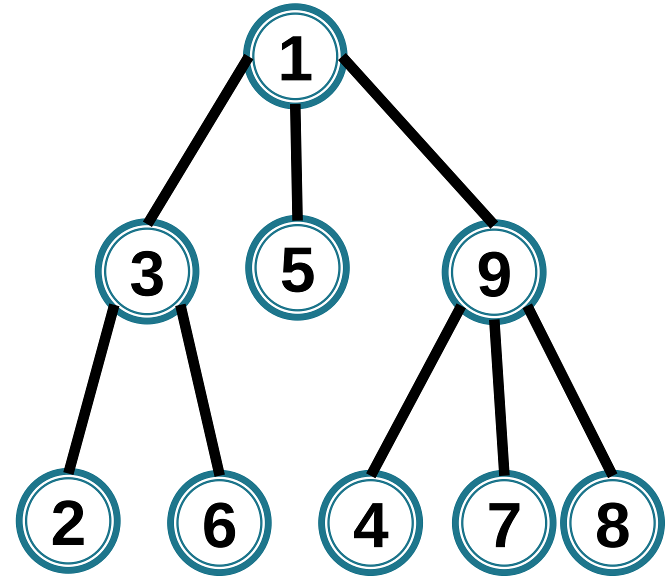
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ 6 4 7 8



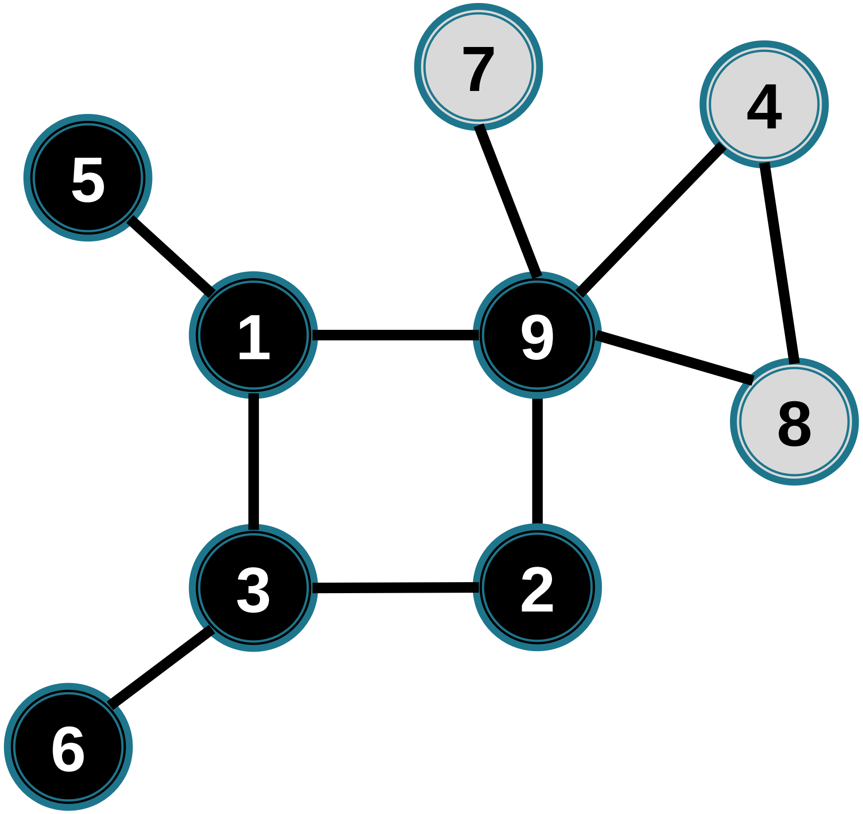


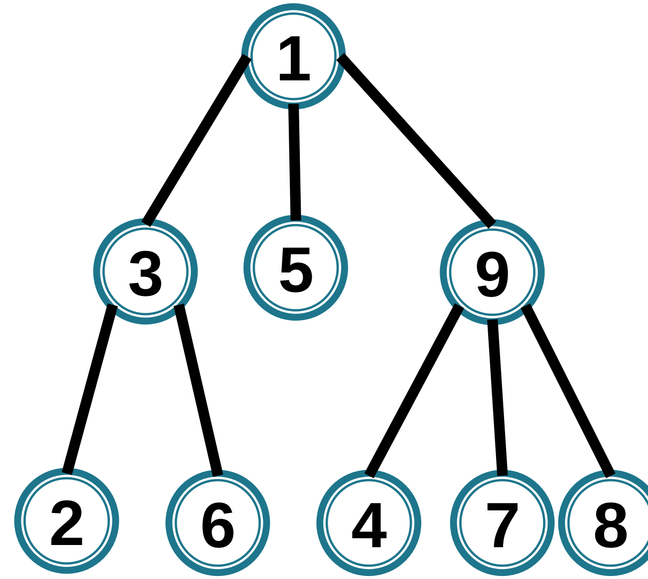
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ 6 4 7 8



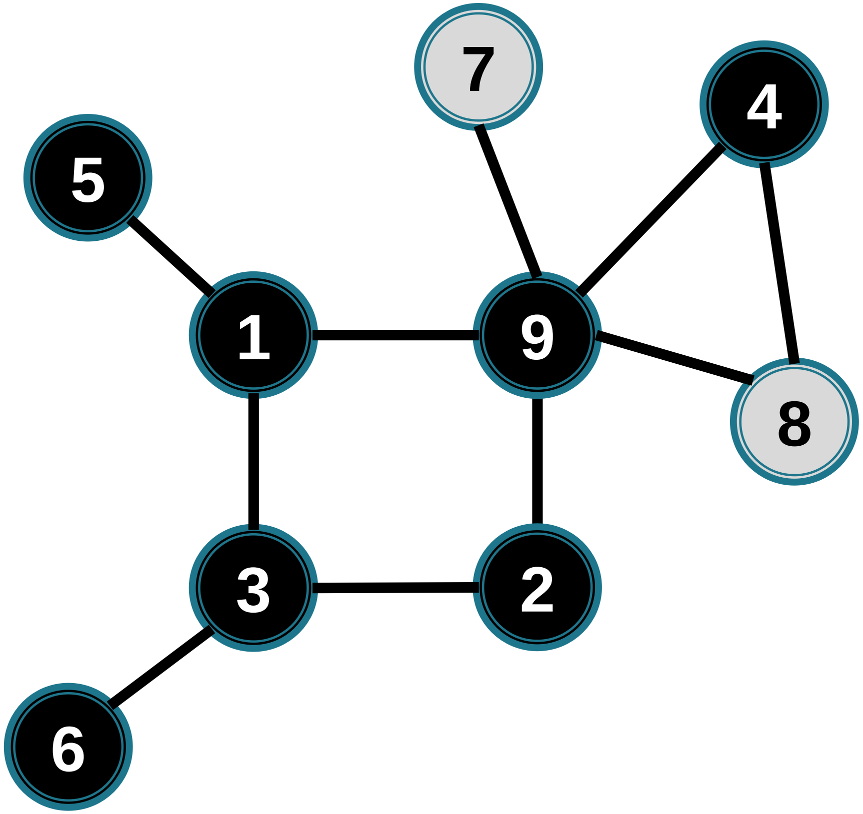


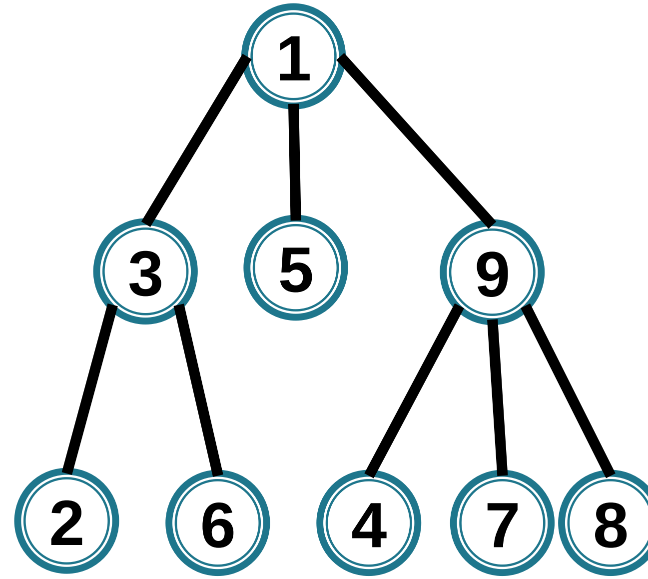
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ 4 7 8



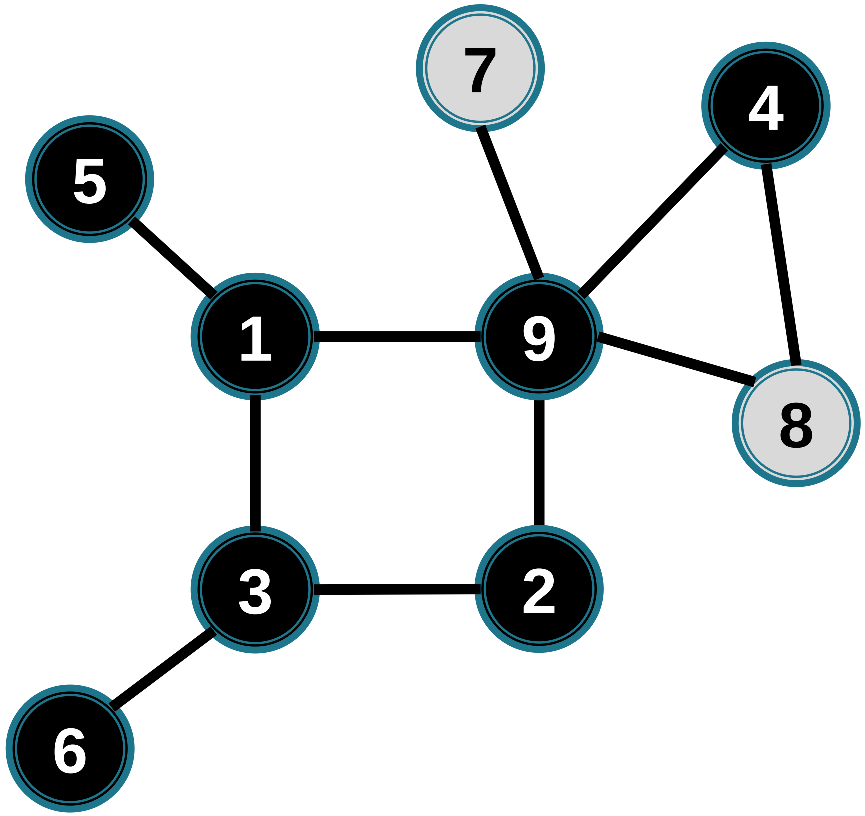


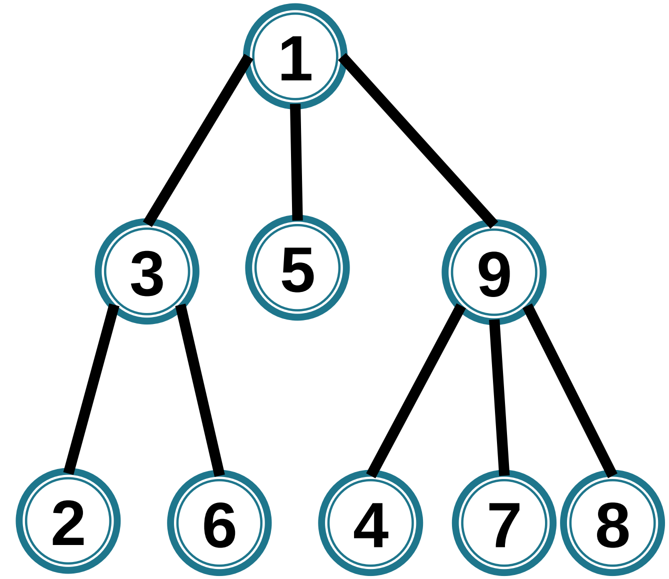
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ **4** 7 8



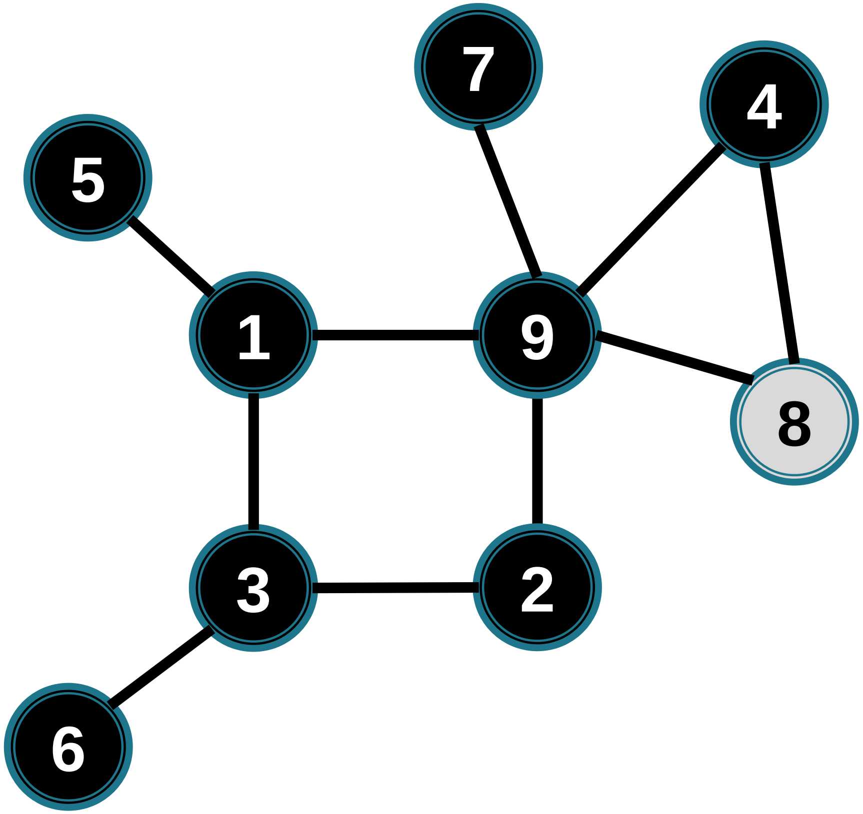


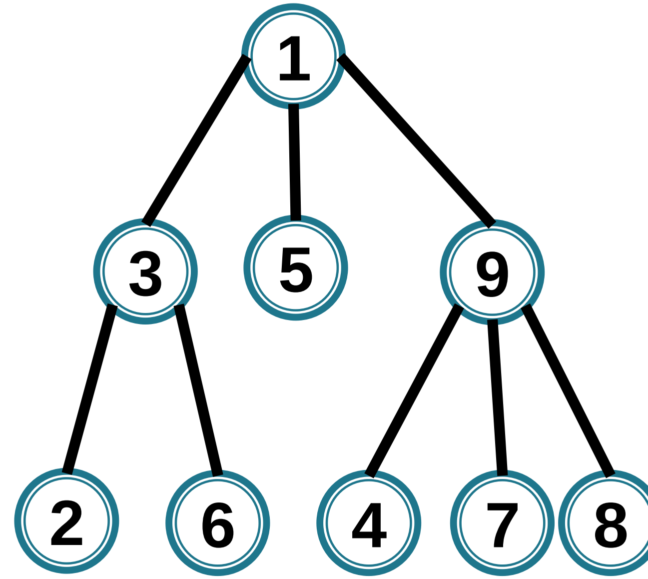
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ 7 8



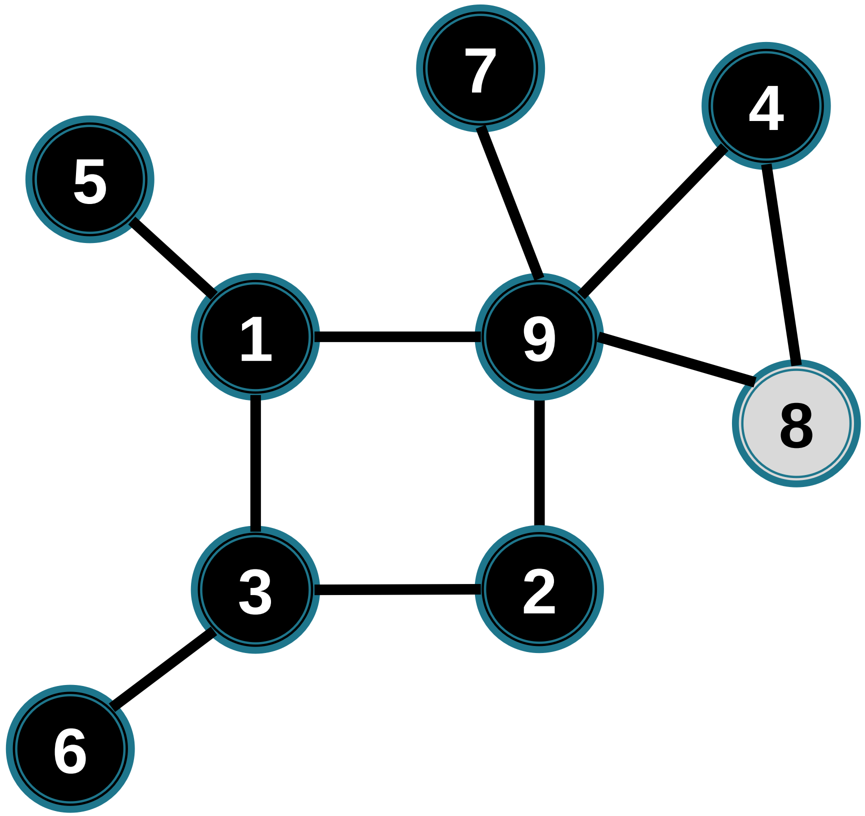


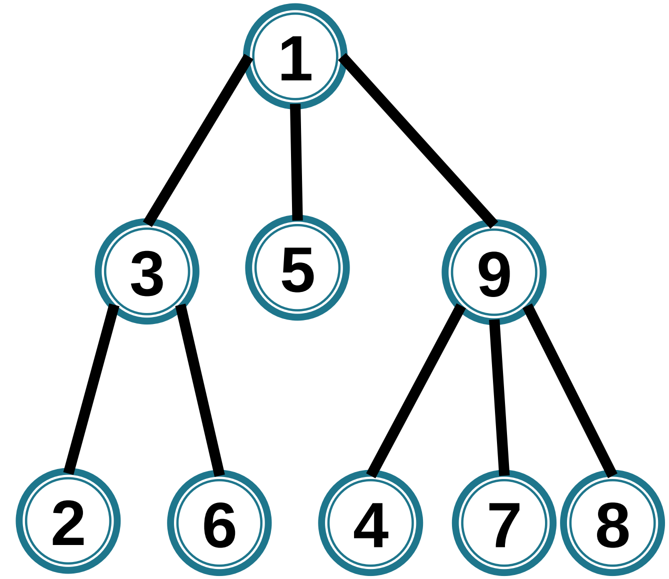
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ 7 8



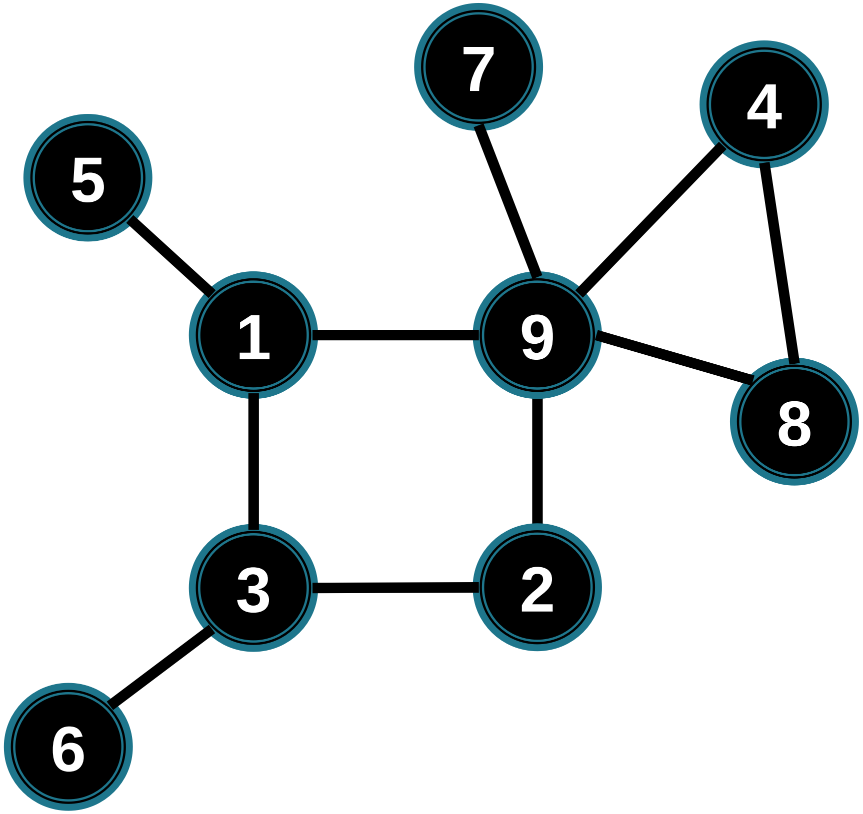


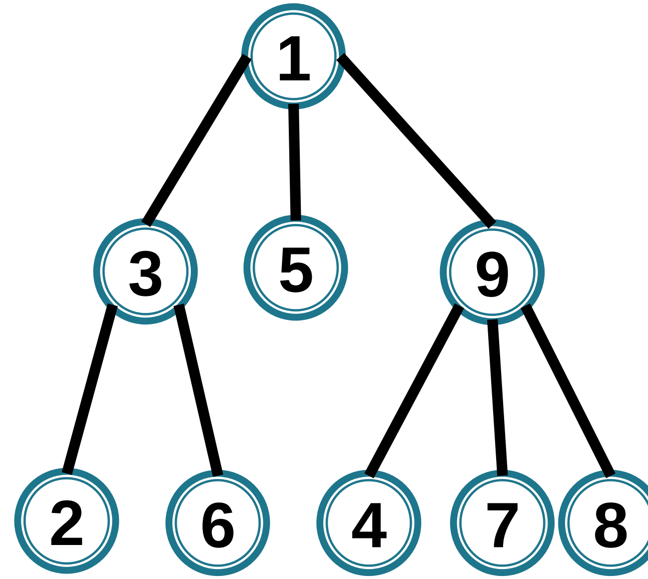
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ 8





~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ 8





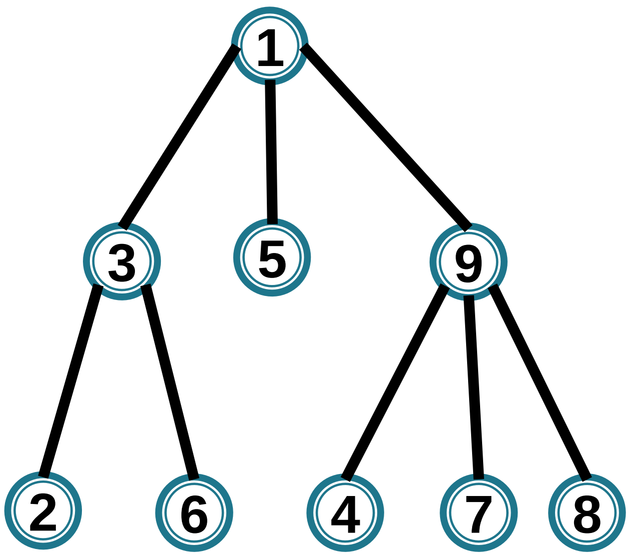
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ ~~8~~

Parcurgerea în lățime – graf neorientat

- Muchiile folosite pentru a descoperi vârfuri noi pornind din s formează un **arbore cu rădăcina s** (numit **arbore BF),** care este un arbore parțial al componentei conexe a lui s

- **Arborele se memorează în BF cu vector tata**

  tata[v] = vârful din care v a fost descoperit (vizitat)



tata = [0, 3, 1, 9, 1, 3, 9, 9, 1]

Pseudocod

Parcurgerea în lățime

- Informații necesare:

$$\textbf{viz[i]} = \begin{cases} 1, \text{ dacă i a fost vizitat} \\ 0, \text{ altfel} \end{cases}$$

Parcurgerea în lățime

- Informații necesare:

$$\textbf{viz[i]} = \begin{cases} 1, \text{ dacă i a fost vizitat} \\ 0, \text{ altfel} \end{cases}$$

Opţional

- **tata[j]** = acel vârf i din care este descoperit (vizitat) j => arborele BF
-

Parcurgerea în lățime

- Informații necesare:

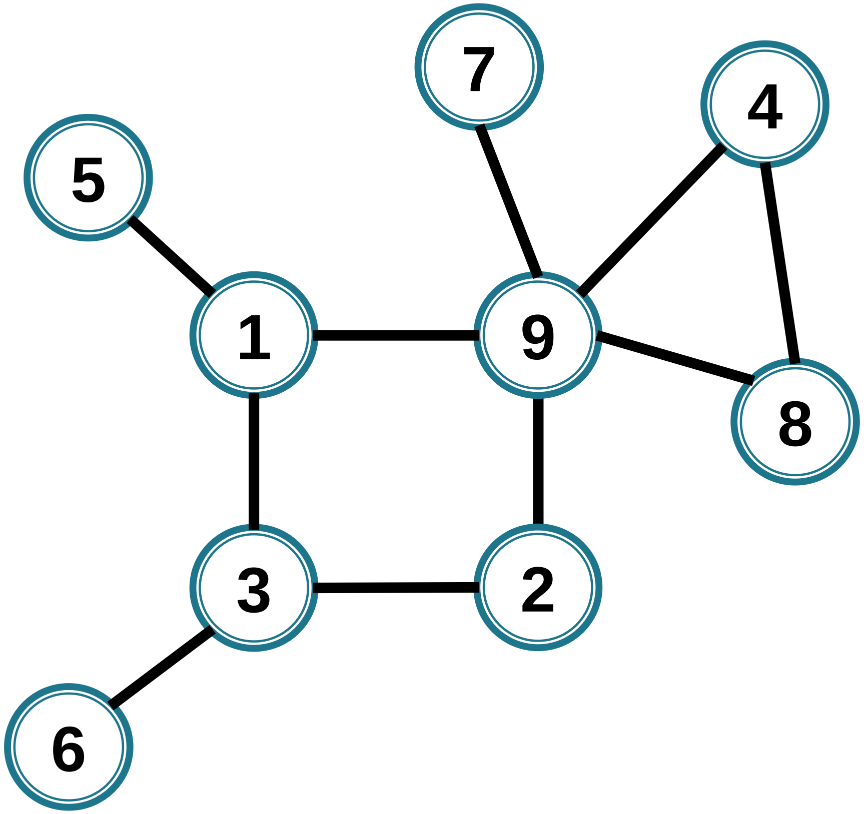
$$\textbf{viz[i]} = \begin{cases} 1, \text{ dacă i a fost vizitat} \\ 0, \text{ altfel} \end{cases}$$

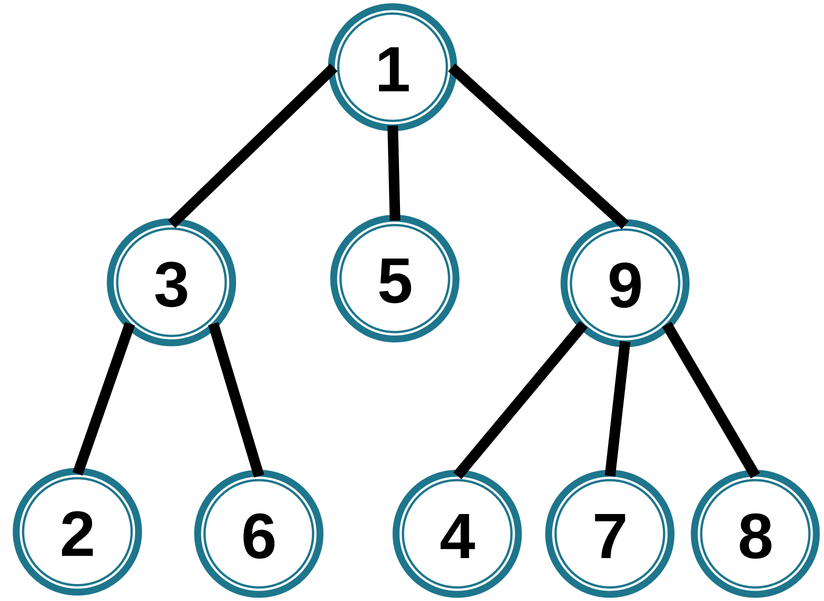
Opţional

- **tata[j]** = acel vârf i din care este descoperit (vizitat) j => arborele BF

- **d[j]** = lungimea drumului determinat de algoritm de la s la j =
  = **nivelul lui j în arborele asociat parcurgerii**

d[j] = d[tata[j]]+1


7
4
5
1
9
8
3
2
6


1
3
5
9
2
6
4
7
8

Inițializări

```
pentru fiecare x ∈ V executa
    viz[x] = 0
    tata[x] = 0
    d[x] = ∞
```

Toate vârfurile sunt albe

```
BFS(s)
    coada C = ∅
    adauga(s, C)
    viz[s] = 1; d[s] = 0
```

s devine gri

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

        pentru j vecin al lui i (ij ∈ E)
```

s devine gri

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

        pentru j vecin al lui i (ij ∈ E)

            daca viz[j]==0 atunci
                adauga(j, C)
                viz[j] = 1
                tata[j] = i
                d[j] = d[i]+1
```

s devine gri

j devine gri

i devine negru
(s-a finalizat explorarea sa)

Apel

- Pentru un vârf s procedura de parcurgere se apelează

```
BFS(s)
```

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

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

Complexitate

Complexitate

De câte ori se execută corpul instrucțiunii repetitive:

```
pentru j vecin al lui i
```

Complexitate

De câte ori se execută corpul instrucțiunii repetitive:

```
pentru j vecin al lui i
```

Depinde de modalitatea de reprezentare a grafului:

- Matrice de adiacență – j ia pe rând toate valorile lui n
- Liste de adiacență – j ia ca valori doar vecinii lui i (ia atâtea valori cât este gradul/gradul de ieșire al lui i)

Complexitate

- Matrice de adiacență

- Liste de adiacență

Complexitate

- Matrice de adiacență $O(|V|^2)$

- Liste de adiacență $O(|V|+|E|)$
  - **Suma gradelor într-un graf neorientat este 2|E|**
  - **Suma gradelor de ieșire într-un graf orientat este |E|**

Proprietăți

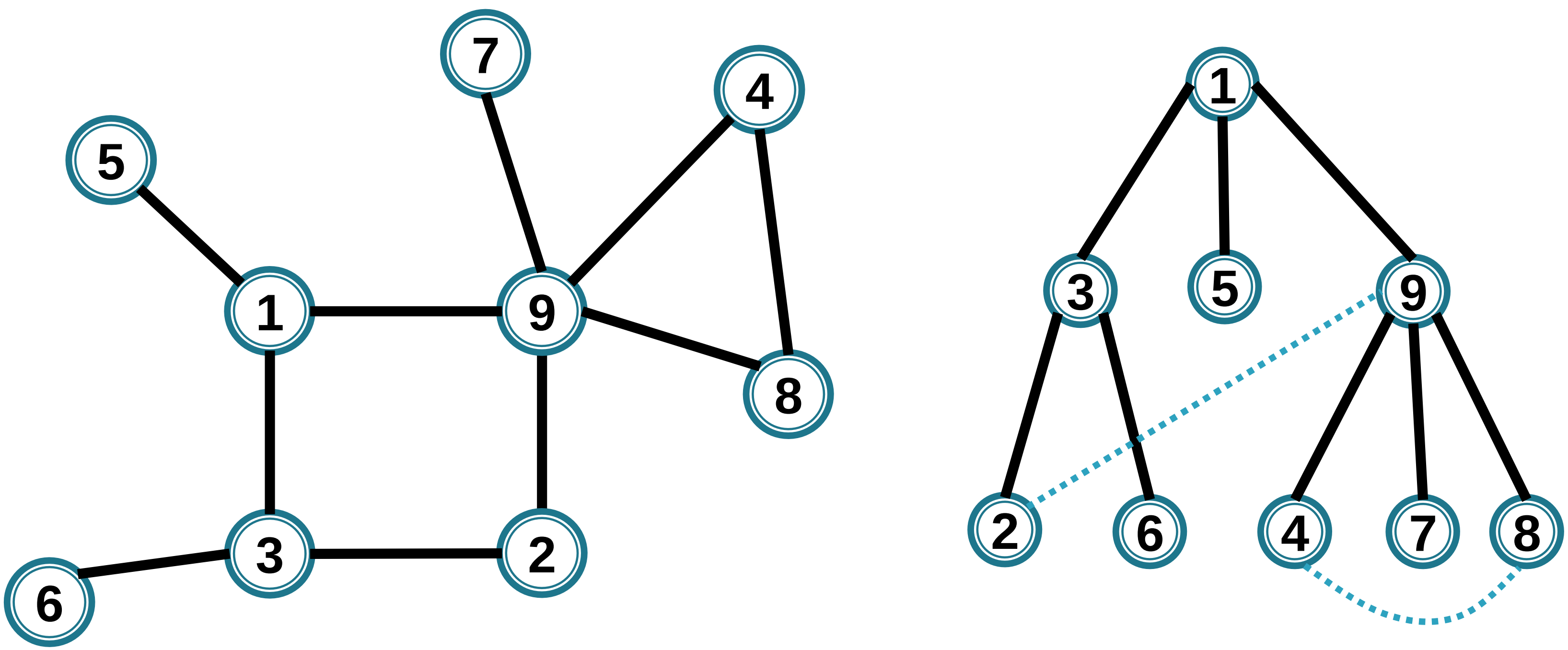
Parcurgerea în lățime - graf neorientat

- Muchiile folosite pentru a descoperi vârfuri noi pornind din s formează un **arbore cu rădăcina s** (numit **arbore BF)**

- Dacă parcurgem toate vârfurile => pădure BF (cu câte un arbore parțial pentru fiecare componentă)

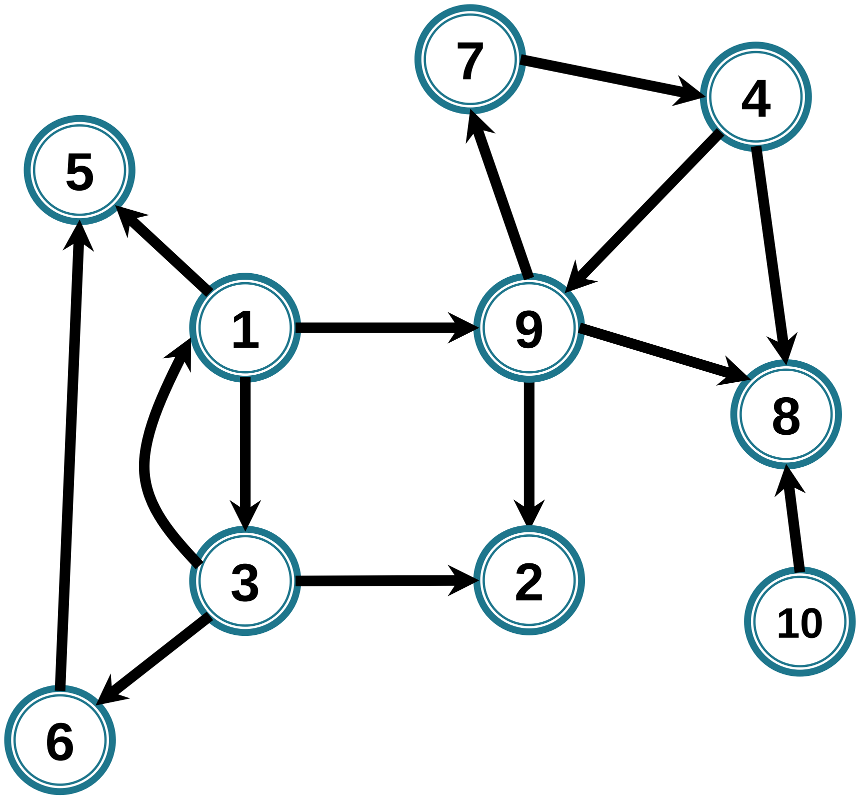
Parcurgerea în lățime - graf neorientat

- **Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore/pădure)**



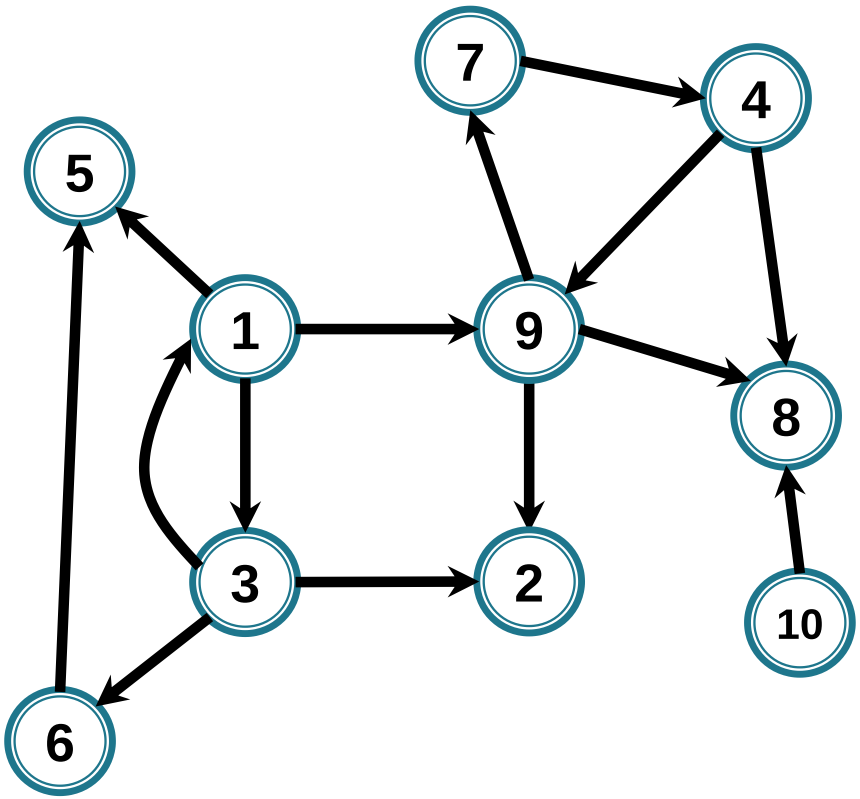
Parcurgerea în lățime - graf orientat

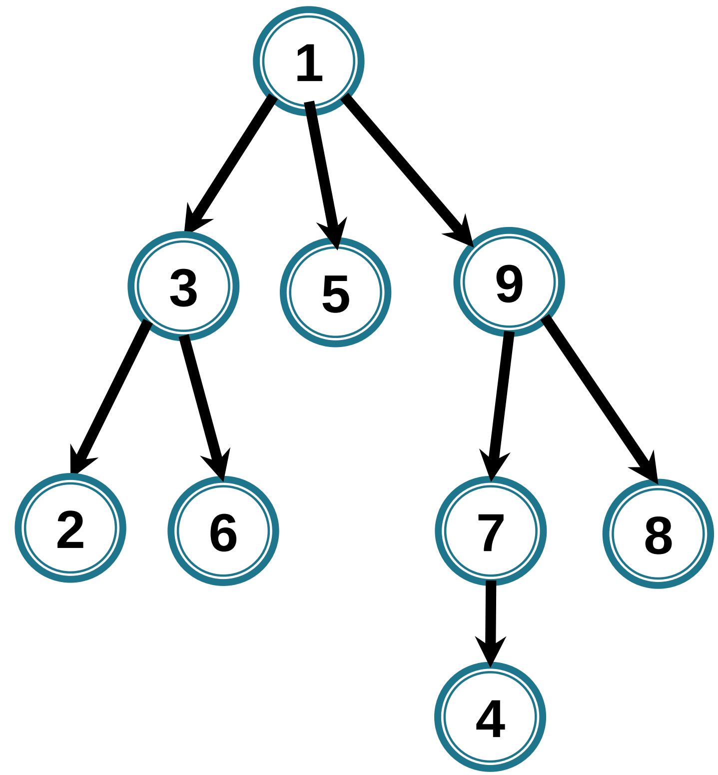
- **Exemplu - caz orientat:**



Parcurgerea în lățime – graf orientat

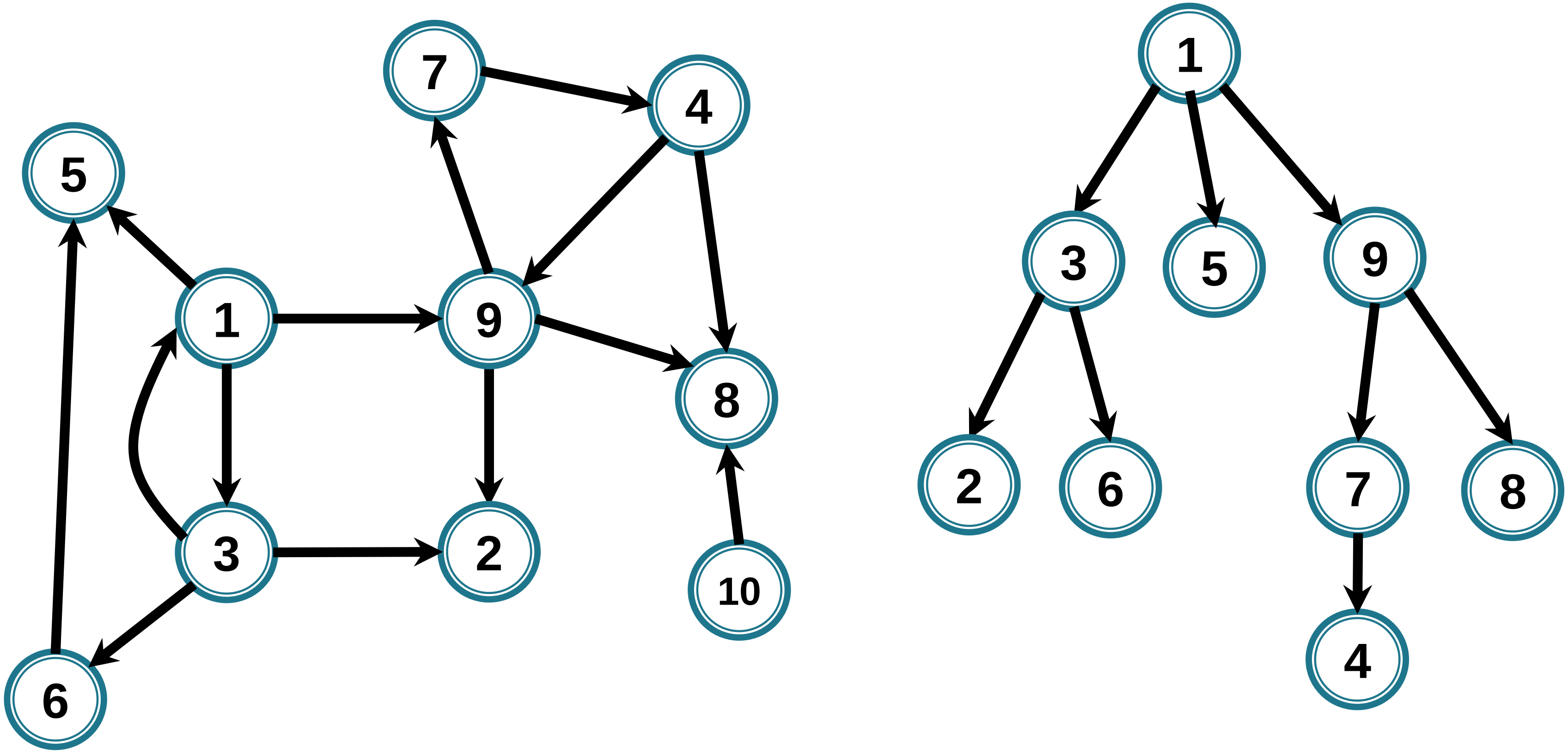
- **Exemplu – caz orientat:**





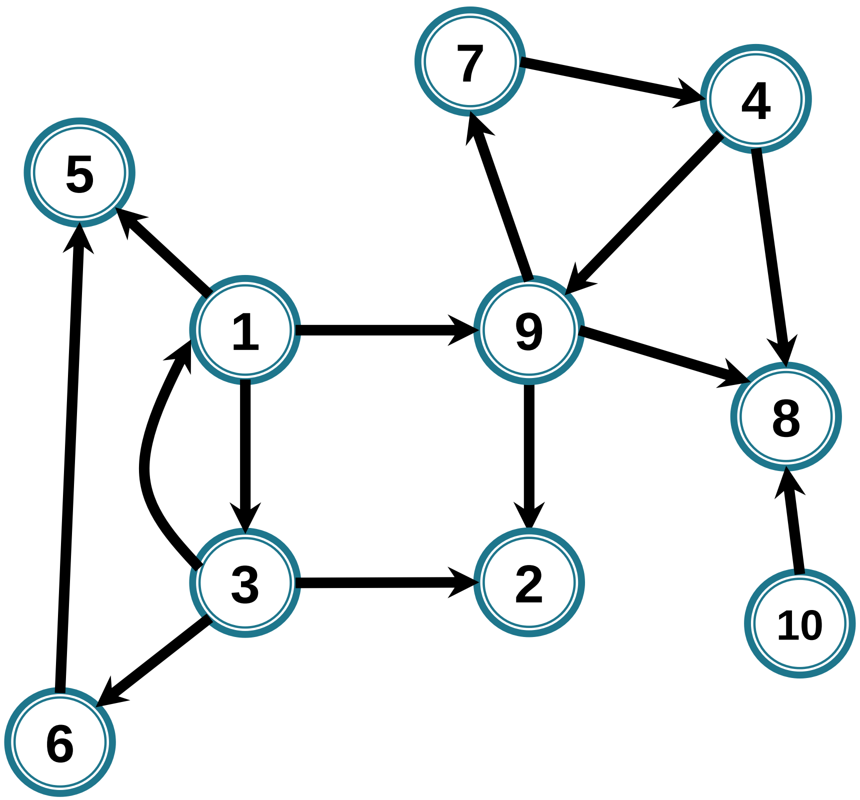
BF(1): 1, 3, 5, 9, 2, 6, 7, 8, 4

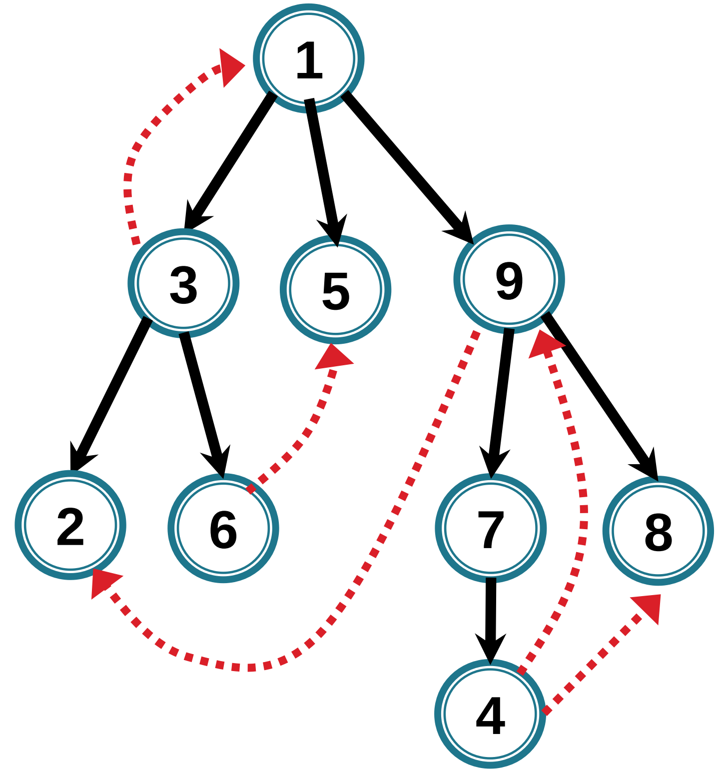
Parcurgerea în lățime - graf orientat



Arbore BF - tot arbore dacă ignorăm orientarea
- arcele corespunzătoare - orientate spre frunze

Parcurgerea în lățime - graf orientat





În arborele BF dacă adăugăm restul arcelor între vârfuri vizitate se închid cicluri, dar nu neapărat circuite

Parcurgerea în lăţime – Aplicații

Determinarea componentelor conexe

G = (V,E) graf neorientat

- BF(s) => arbore parțial pentru componenta conexă care
  conține s (cu rădăcina s) = arbore BF

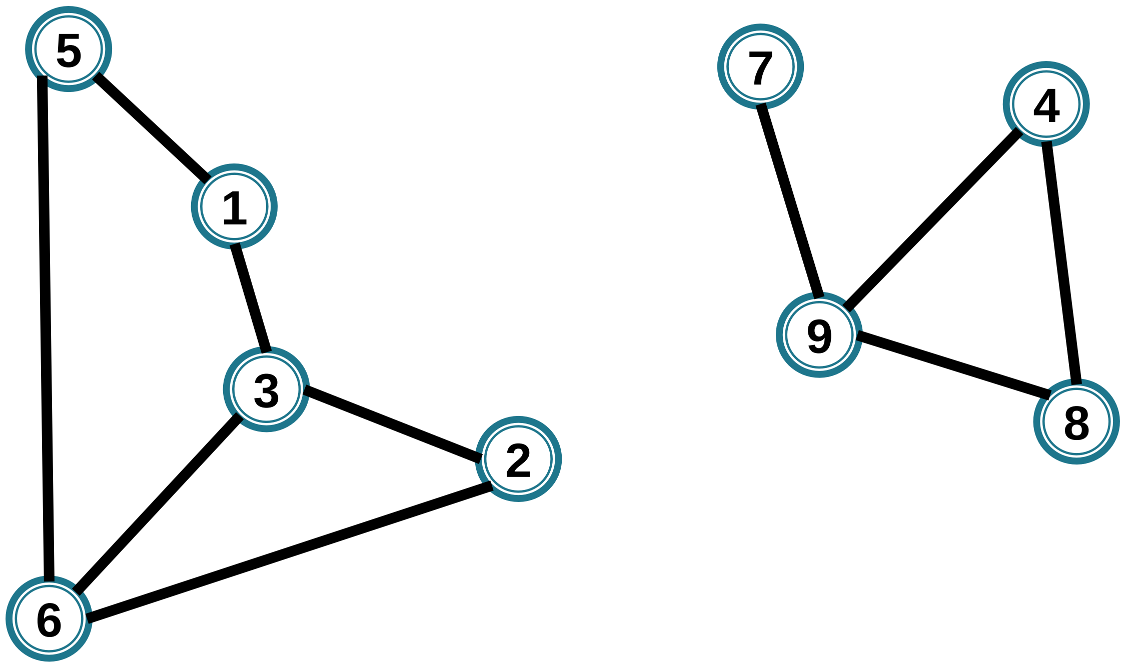
Determinarea componentelor conexe

G = (V,E) graf neorientat

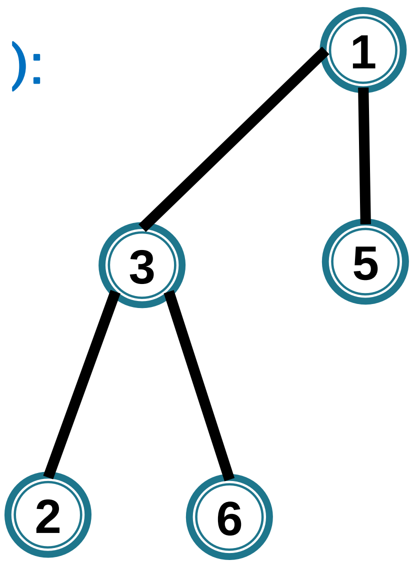
- BF(s) => arbore parțial pentru componenta conexă care conține s (cu rădăcina s) = arbore BF
- Toate componentele conexe - reluăm BFS din vârfuri nevizitate => pădure BF, cu arbori parțiali pentru fiecare componentă

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

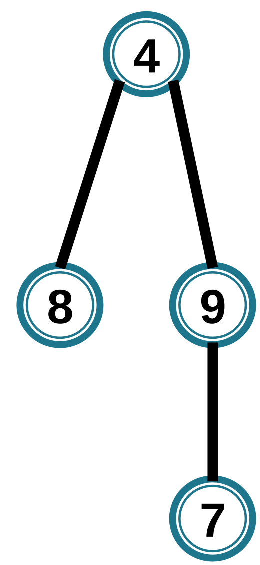
Pădure: muchiile `{tata[x], x}`, `tata[x]` $\neq 0$



BF(1):

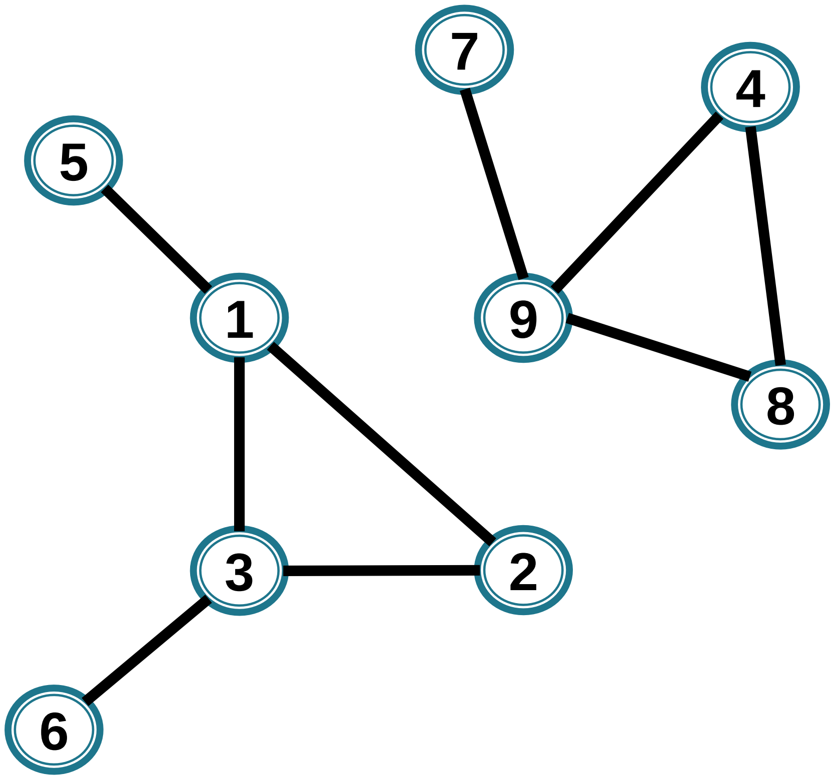


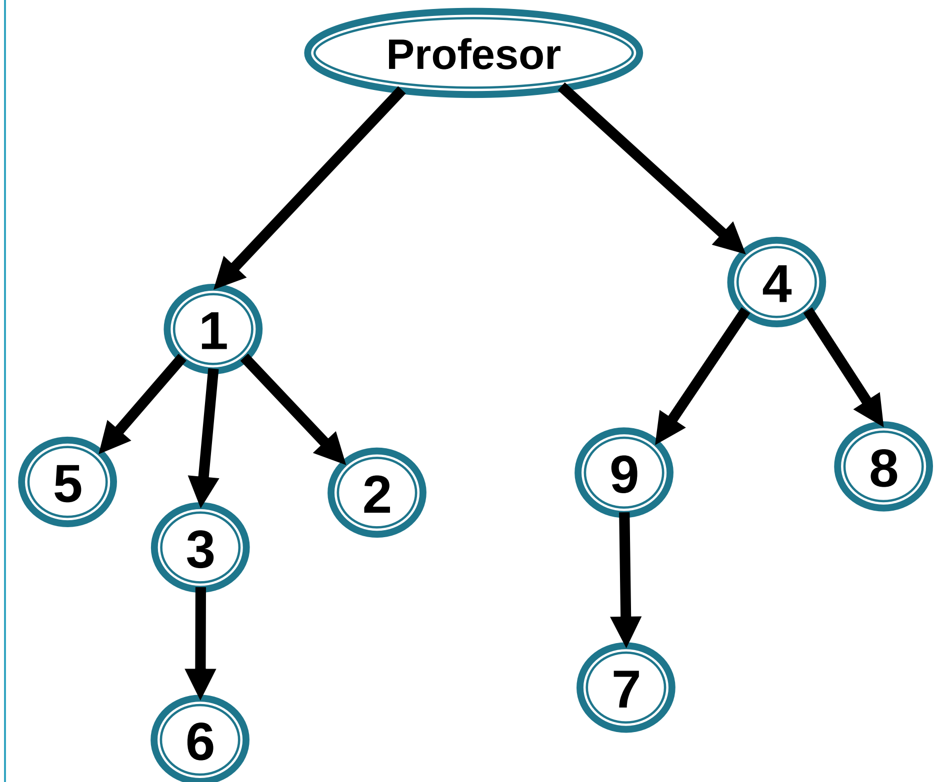
BF(4):



Aplicație - arbore parțial

- Determinarea unui arbore parțial al unui graf conex
- **Transmiterea unui mesaj în rețea**: Între participanții la un curs s-au legat relații de prietenie și comunică și în afara cursului. Profesorul vrea să transmită un mesaj participanților și știe ce relații de prietenie s-au stabilit între ei. El vrea să contacteze cât mai puțini participanți, urmând ca aceștia să transmită mesajul între ei. Ajutați-l pe profesor să decidă cui trebuie să transmită inițial mesajul și să atașeze la mesaj o listă în care să arate fiecărui participant către ce prieteni trebuie să trimită mai departe mesajul, astfel încât mesajul să ajungă la fiecare participant la curs o singură dată.


7
4
5
1
9
8
3
2
6


Profesor
1
4
5
3
2
9
8
6
7

Determinarea de distanțe și lanțuri/drumuri minime de la un vârf la celelalte

Amintim aplicații:

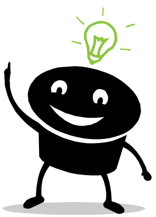
- Numărul Erdös
- Traseu între două puncte cu număr minim de stații intermediare
- Traseul minim în labirint la una dintre ieșiri
  - BFS în matrice - algoritmul lui Lee

Determinarea de distanțe și lanțuri/drumuri minime de la un vârf la celelalte

- Determinarea unui lanț/drum minim între două vârfuri date u și v

Determinarea de distanțe și lanțuri/drumuri minime de la un vârf la celelalte

- Determinarea unui lanț/drum minim între două vârfuri date u și v



Se apelează BFS(u), apoi se afişează drumul de la u la v folosind vectorul tata (ca la arbori), **dacă există**

```
BFS(u)
daca viz[v] == 1 atunci
    lant(v)
altfel
    scrie "nu exista"
```

Parcurgerea BFS(u) se poate opri atunci când este vizitat v

Corectitudine

Parcurgerea în lățime

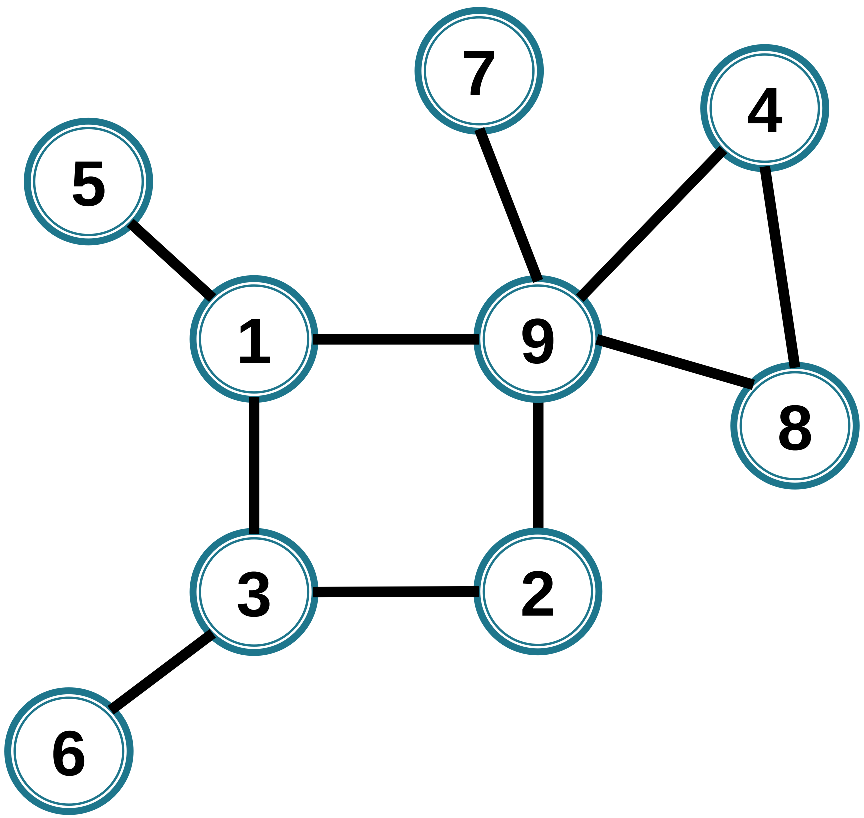
Propoziție – Corectitudinea BF

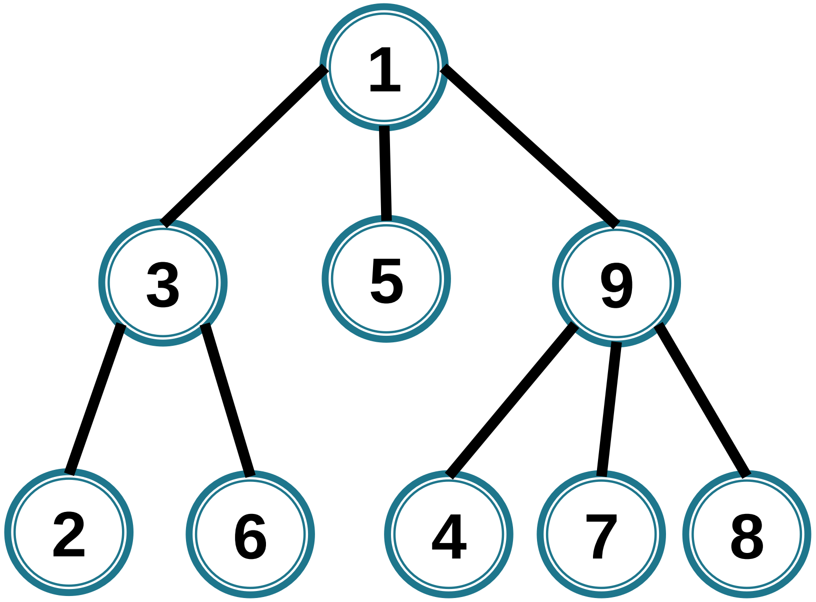
La finalul algoritmului BFS(s), pentru orice vârf v avem

$$d[v] = \delta_G(s, v)$$

În plus, arborele BF de rădăcină s (notat T) memorat în vectorul tata **conservă distanțele din graf de la s la celelalte vârfuri**, deci este un arbore de distanțe față de s:

$$\delta_T(s, v) = \delta_G(s, v), \text{ pentru orice vârf } v$$

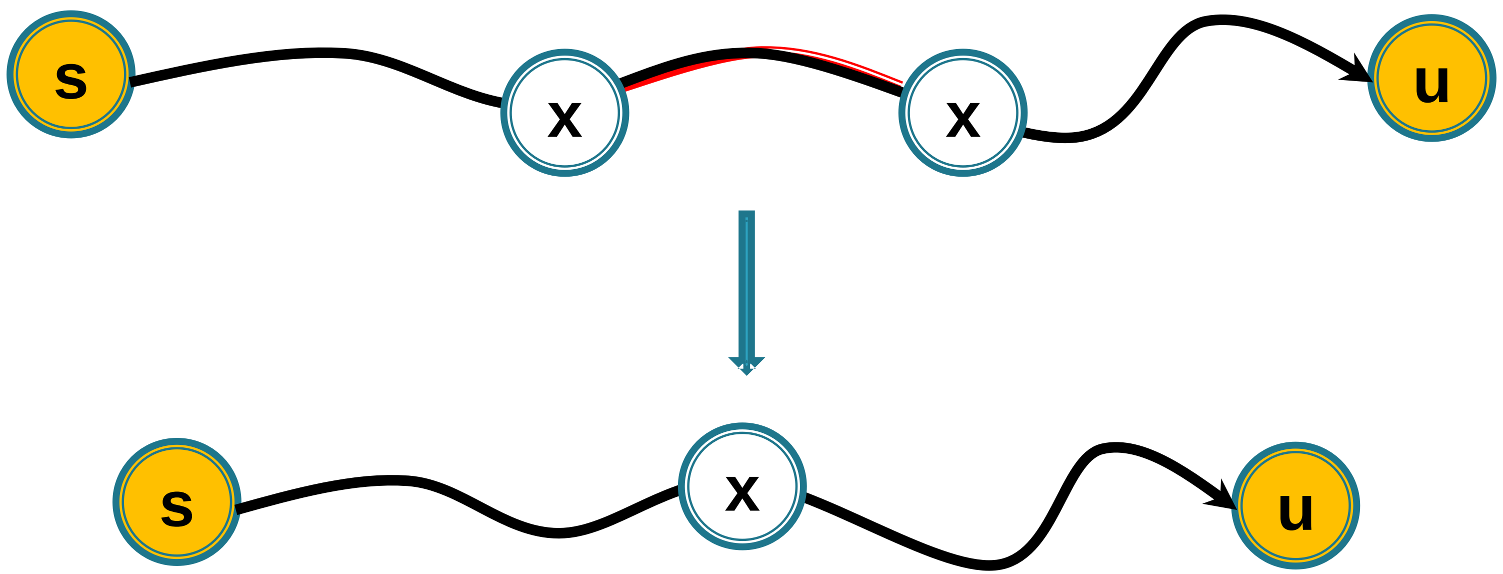




$$\delta_G(1,v) = \delta_T(1,v)$$

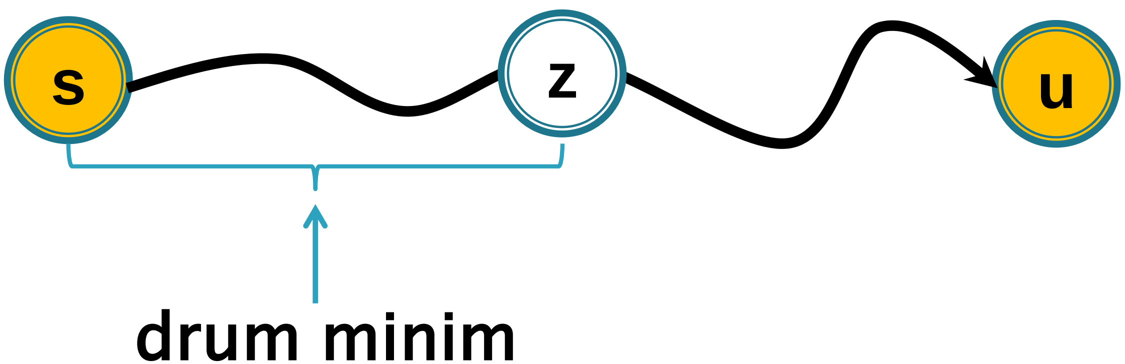
Corectitudine

- **Observația 1.** Dacă P este un drum/lanț minim de la s la u, atunci P este drum/lanț **elementar**.

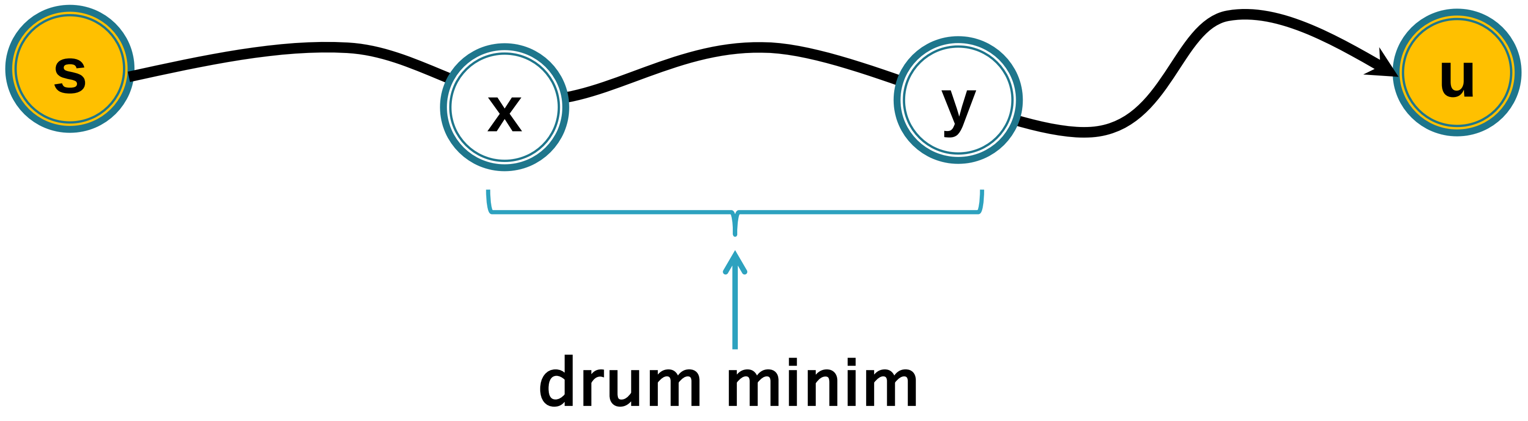


Corectitudine

- **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.



Corectitudine



Corectitudine

- **Lema 1.** Dacă în coada **C** avem: $v_1, v_2, \ldots, v_r$ (la un moment al execuției algoritmului), atunci

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$$

De ce?

Corectitudine

- **Lema 1**. Dacă în coada **C** avem: $v_1, v_2, \dots, v_r$ (la un moment al execuției algoritmului), atunci

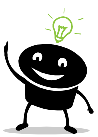
$$d[v_1] \le d[v_2] \le \dots \le d[v_r] \le d[v_1] + 1$$

Evidențiem operațiile - Inducție

Corectitudine

- **Lema 1**. Dacă în coada **C** avem: v_1, v_2,..., v_r (la un moment al execuției algoritmului), atunci

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$$



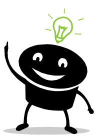
Evidențiem operațiile - Inducție

- Prima operație: $C = [s]$ și $d[s] = tata[s] = 0$ - se verifică

Corectitudine

- **Lema 1.** Dacă în coada **C** avem: v_1, v_2,…, v_r (la un moment al execuției algoritmului), atunci

$$d[v_1] \le d[v_2] \le \ldots \le d[v_r] \le d[v_1] + 1$$



Evidențiem operațiile – Inducție

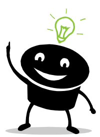
- Prima operație: C =[s] și d[s] = tata[s] = 0 – se verifică
- Presupunem adevărat la pasul curent (intrare in while)
  - extragem $i = v_1$

    $C = [v_2,\ldots,v_r]$ și $d[v_r] \le \mathbf{d[v_1] + 1 \le d[v_2] + 1}$

Corectitudine

- **Lema 1.** Dacă în coada **C** avem: v_1, v_2,…, v_r (la un moment al execuției algoritmului), atunci

$$d[v_1] \le d[v_2] \le \ldots \le d[v_r] \le d[v_1] + 1$$



Evidențiem operațiile – Inducție

- Prima operație: $C = [s]$ și $d[s] = tata[s] = 0$ – se verifică
- Presupunem adevărat la pasul curent (intrare in while)
  - ➢ extragem $i = v_1$

    $C = [v_2,\ldots,v_r]$ și $d[v_r] \le d[v_1] + 1 \le d[v_2] + 1$
  - ➢ Inserăm pe rând vecinii w_i ai lui v_1 nevizitați:

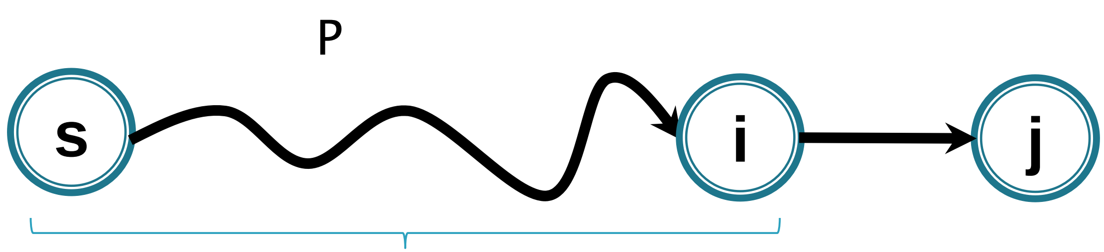
    $d[w_i] = d[v_1] + 1 \le d[v_2] + 1 \Rightarrow$

    $C = [v_2,\ldots,v_r, w_1,\ldots w_i]$ verifică relația

Corectitudine

- **Lema 2**. Dacă d[v] = k, atunci există în G un drum de la s la v de lungime k și acesta se poate determina din vectorul tata, mai exact tata[v] = predecesorul lui v pe un drum de la s la v de lungime k.

Inducție – adevărat pentru vârfurile deja vizitate



Lungime d[i]
(ipoteza de inducție)

d[j]= d[i] + 1 = l([s_P_i,j])
tata[j] = i

Corectitudine

- **Consecințe**.
  - Dacă x a fost extras din C înaintea lui y, avem
    $$d[x] \le d[y]$$
  - $d[v] \ge \delta(s,v)$ (d[v] este o "*supraestimare*")
  - $\delta(s,v) = \infty \Rightarrow d[v] = \infty$

Parcurgerea în lățime

Propoziție – Corectitudinea BF

La finalul algoritmului BFS(s), pentru orice vârf v avem

$$d[v] = \delta_G(s, v)$$

În plus, arborele BF de rădăcină s (notat T) memorat în vectorul tata **conservă distanțele din graf de la s la celelalte vârfuri**, deci este un arbore de distanțe față de s:

$$\delta_T(s, v) = \delta_G(s, v), \text{ pentru orice vârf } v$$