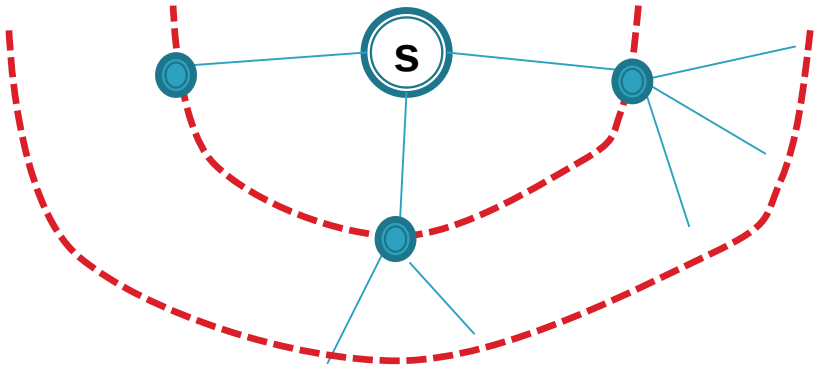
Parcurgerea în lăţime

Parcurgerea în lățime

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

etc



Parcurgerea în lățime

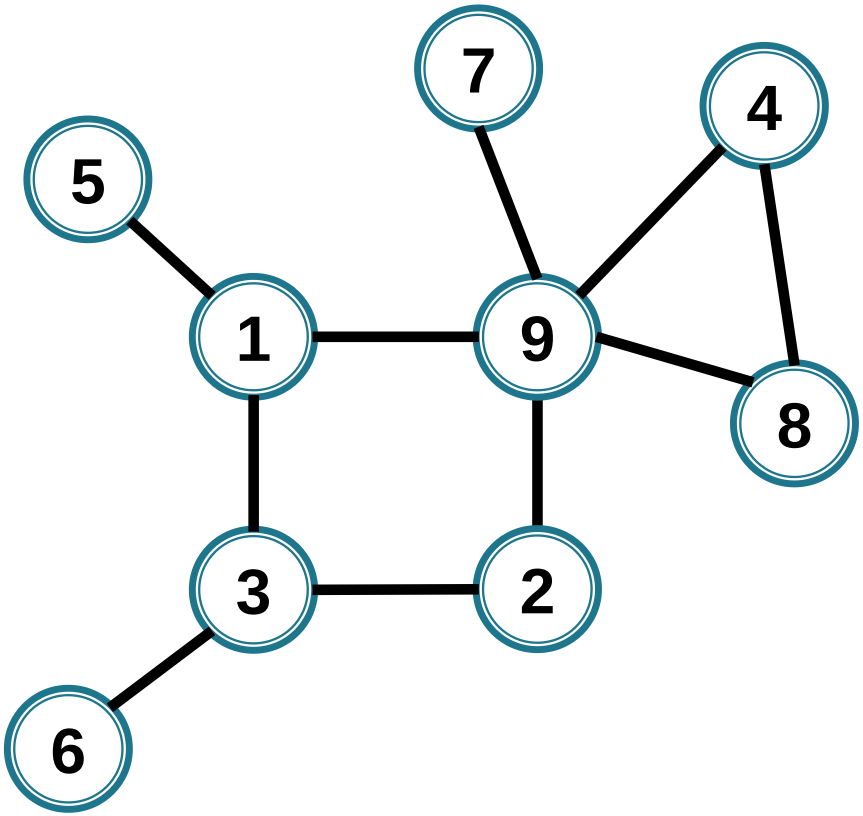
- Pentru gestionarea vârfurilor parcurse care mai pot avea vecini nevizitaţi – o structură de tip **coadă**

Parcurgerea în lățime

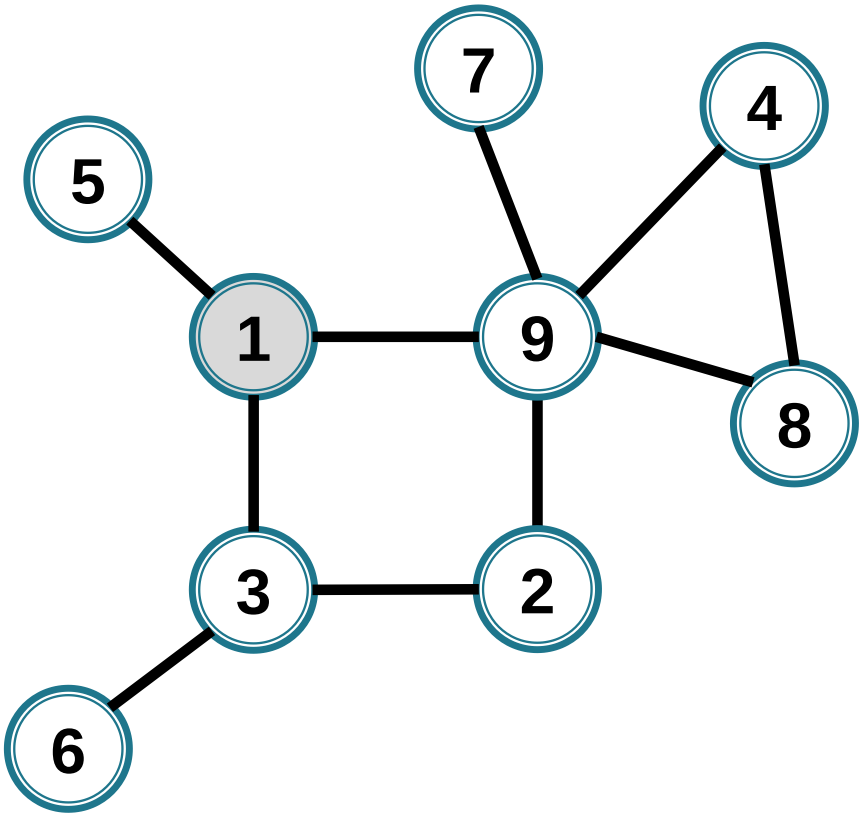
- Pentru gestionarea vârfurilor parcurse care mai pot avea vecini nevizitaţi – o structură de tip **coadă**

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

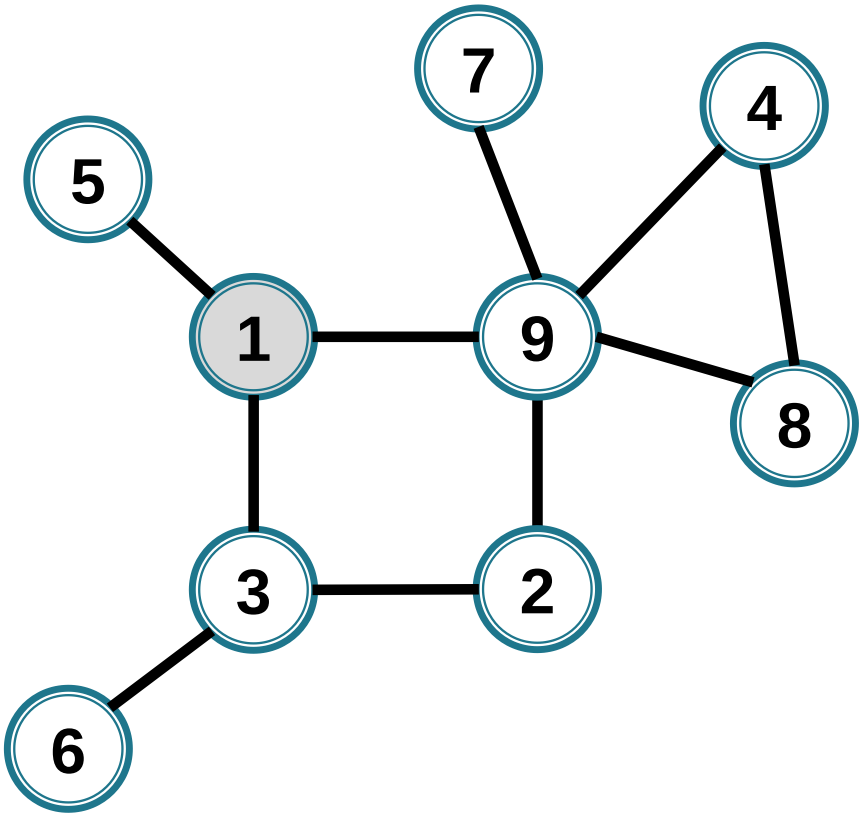
Exemplu
pentru graf neorientat



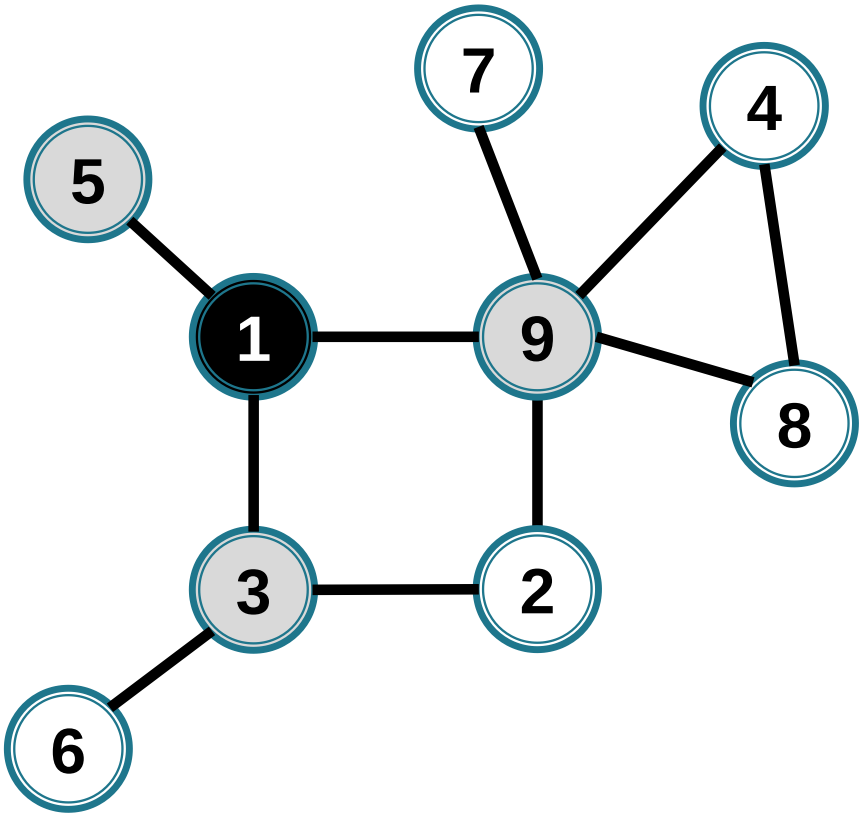
Vârf de start 1

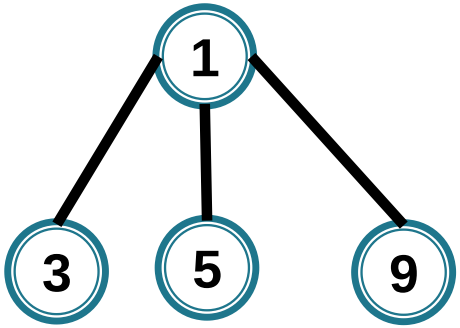


1

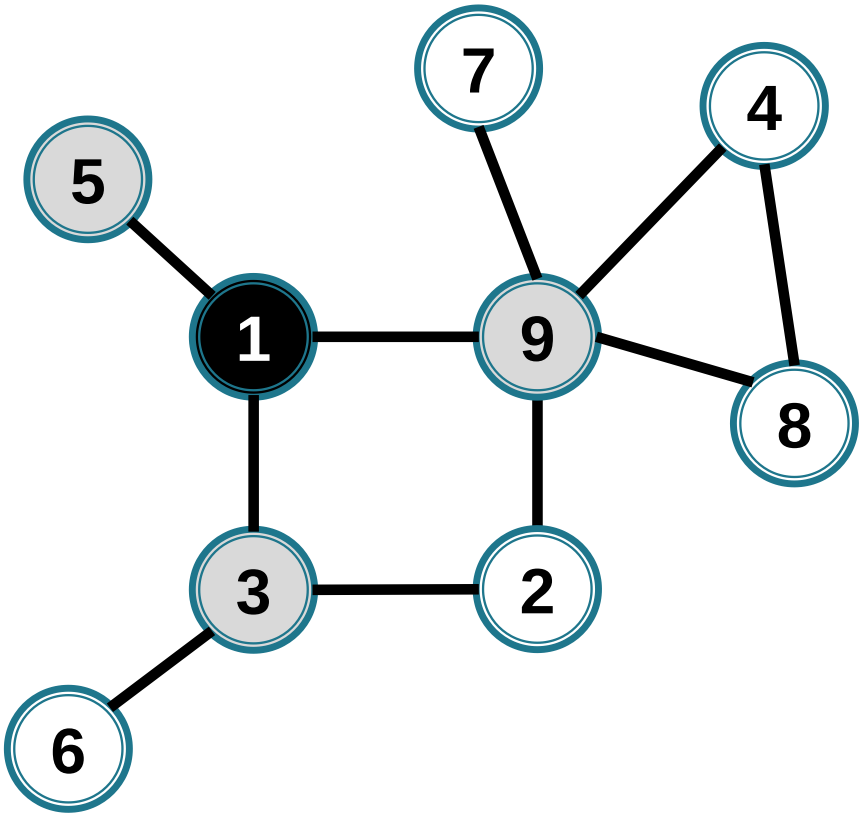


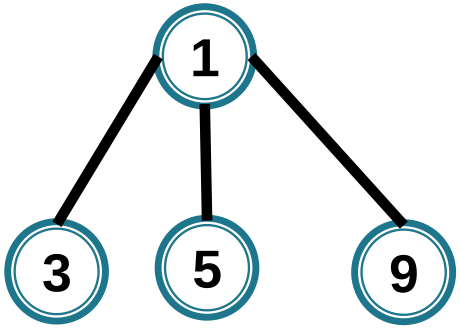
1



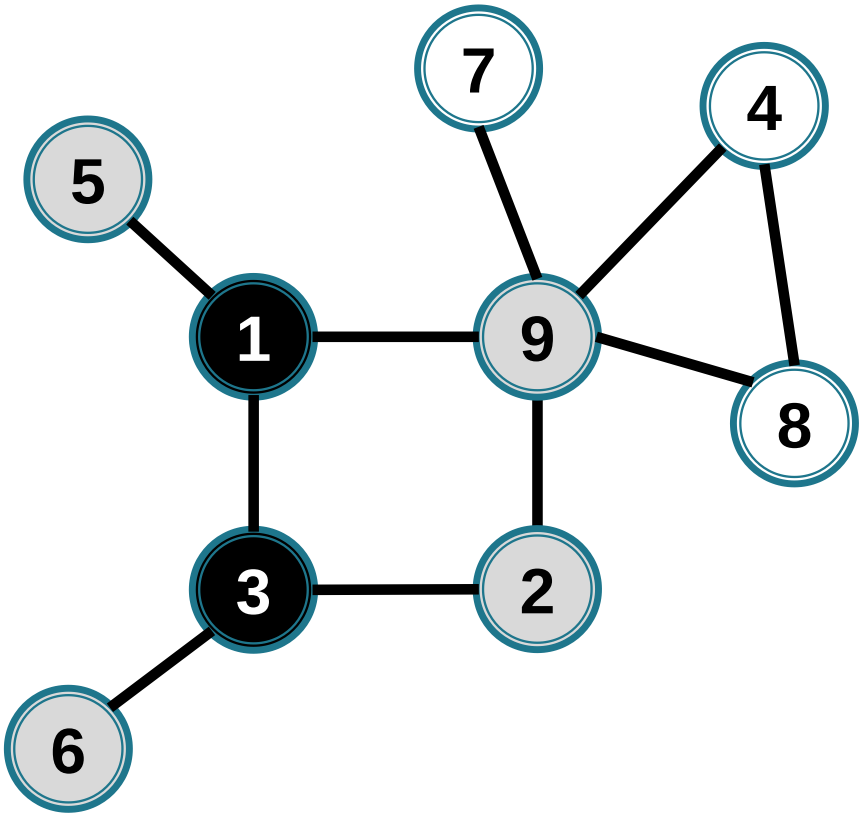


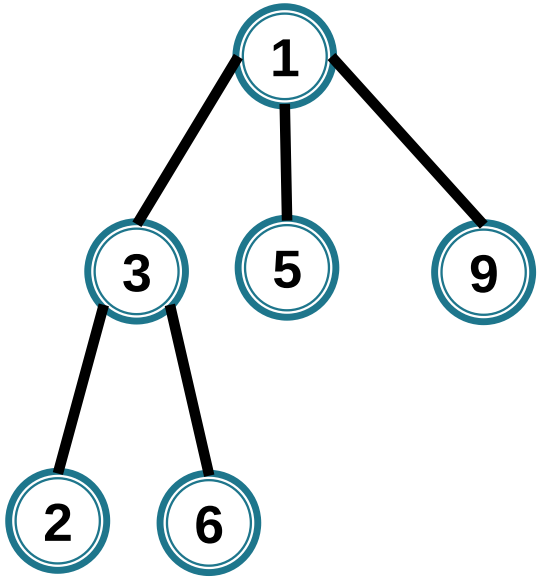
~~1~~ 3 5 9



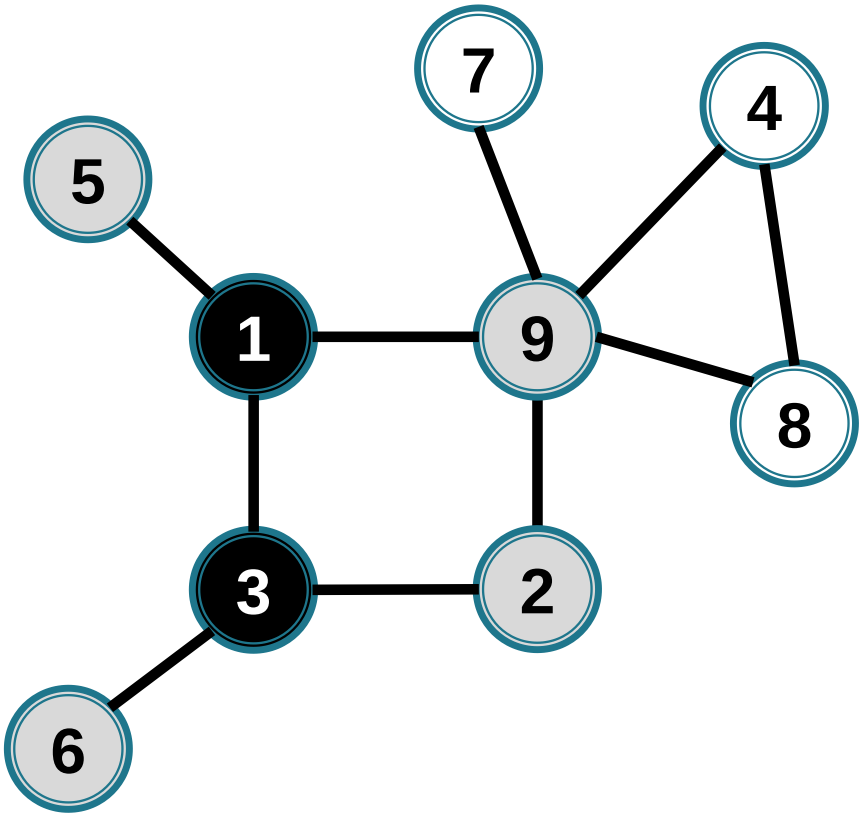


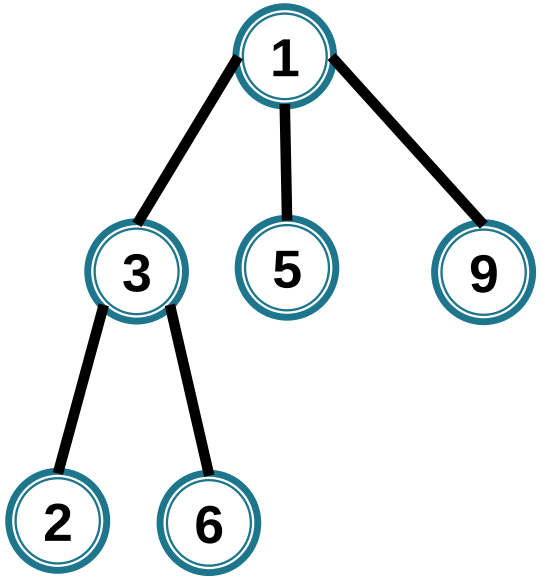
~~1~~ 3 5 9



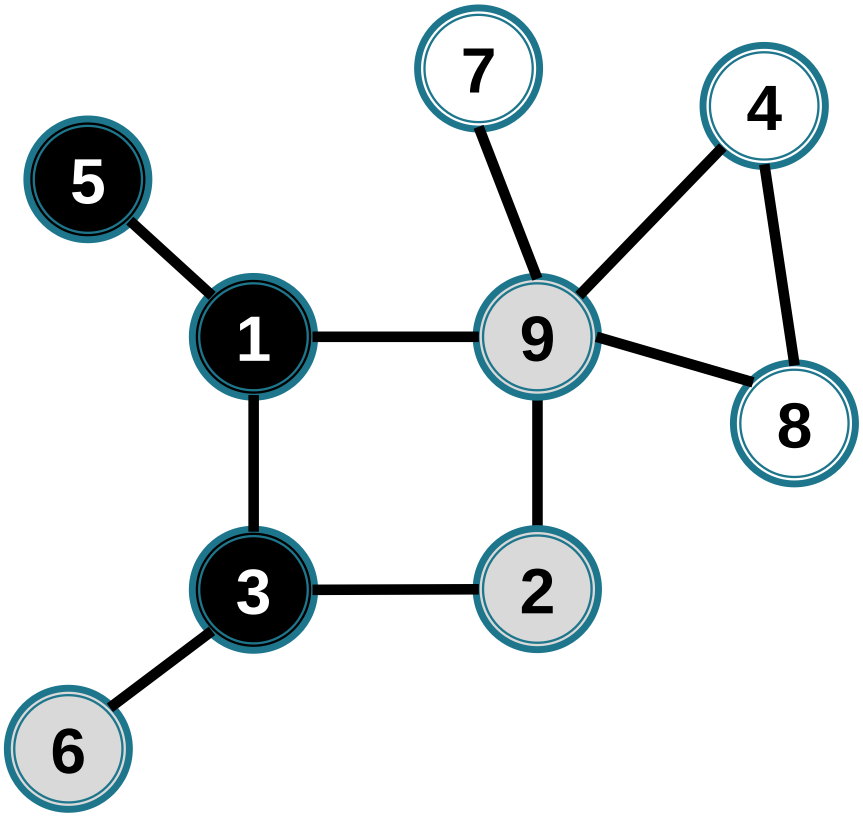


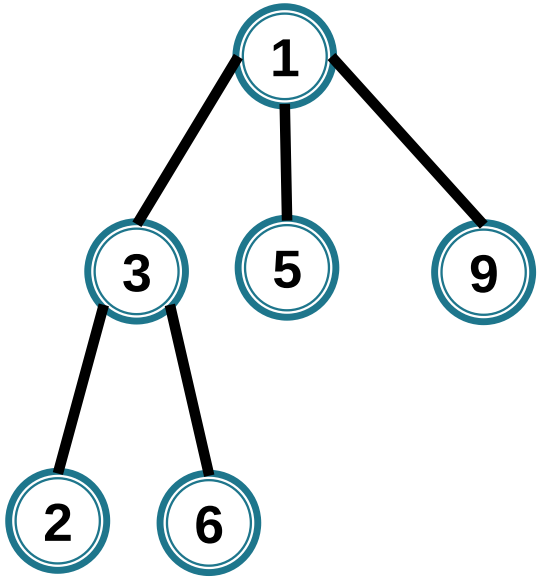
~~1~~ ~~3~~ 5 9 2 6



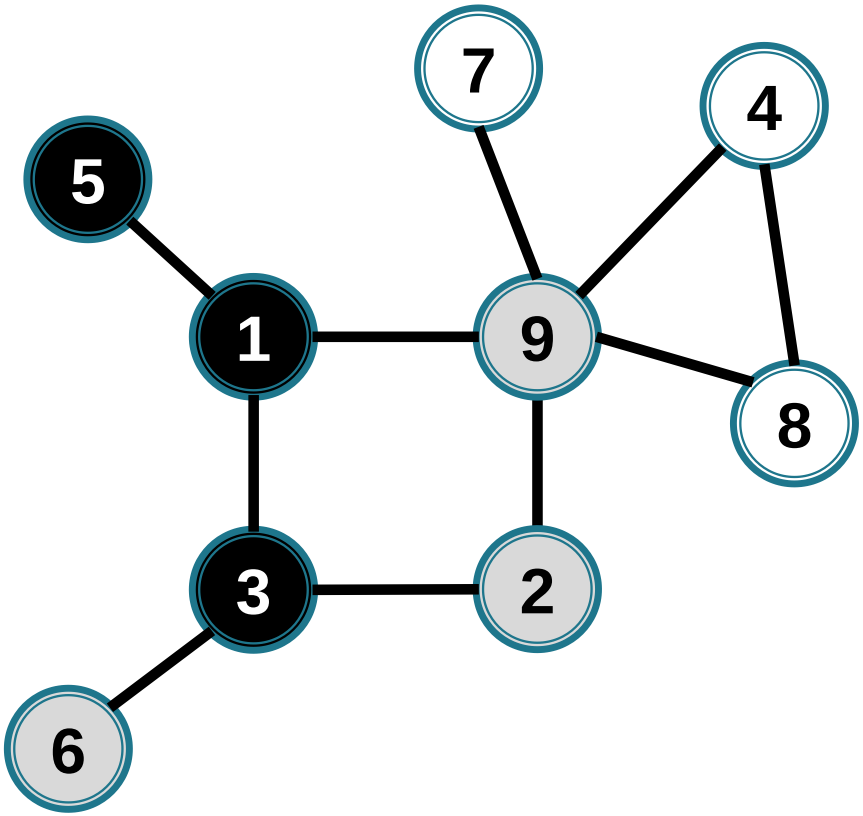


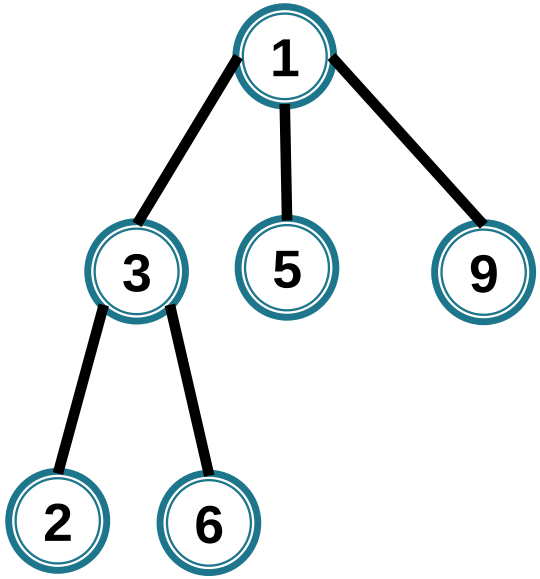
~~1~~ ~~3~~ 5 9 2 6



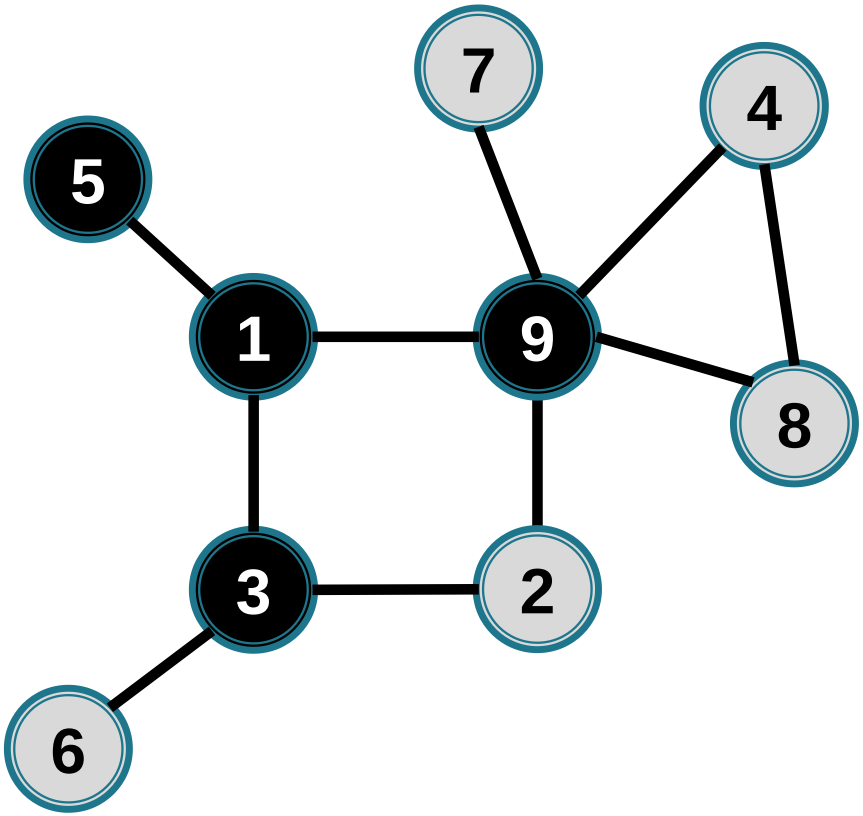


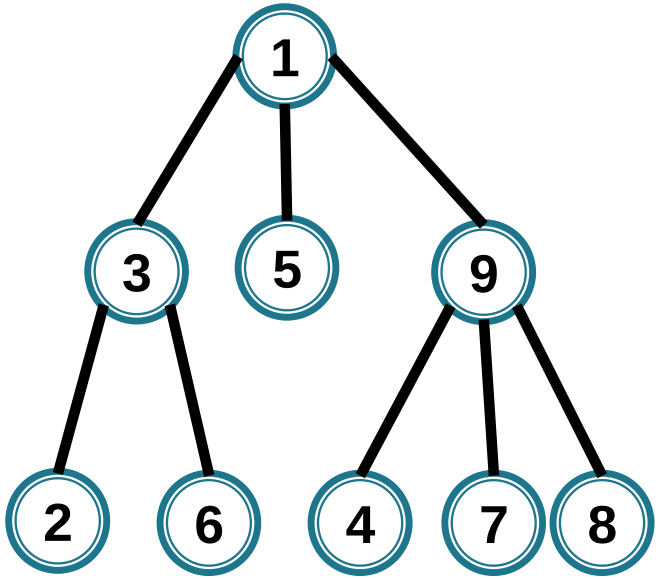
~~1~~ ~~3~~ ~~5~~ 9 2 6



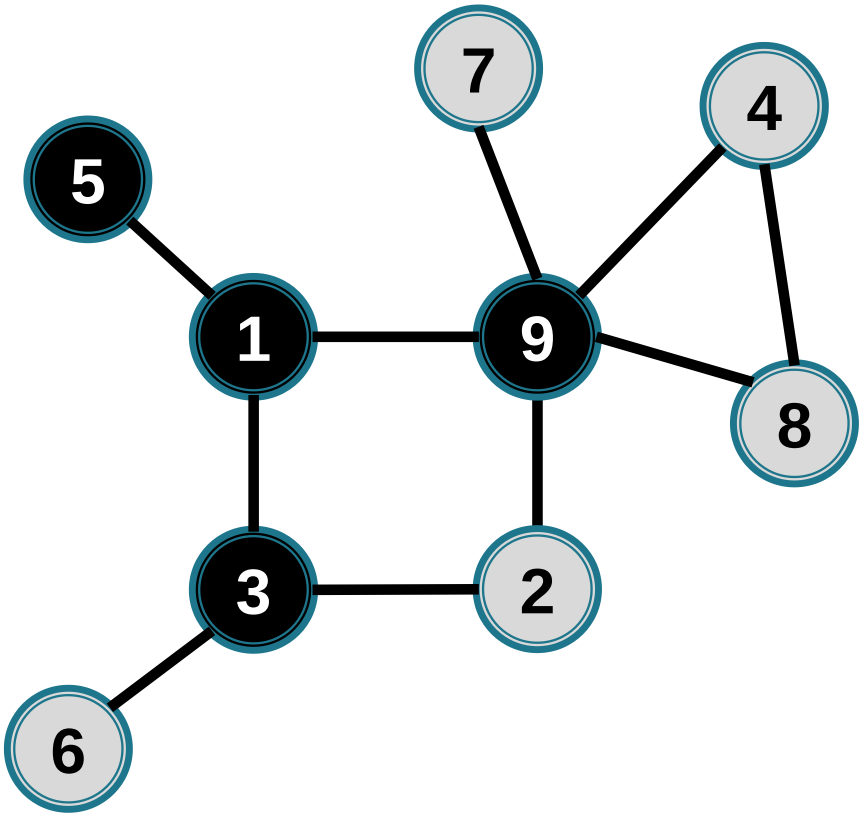


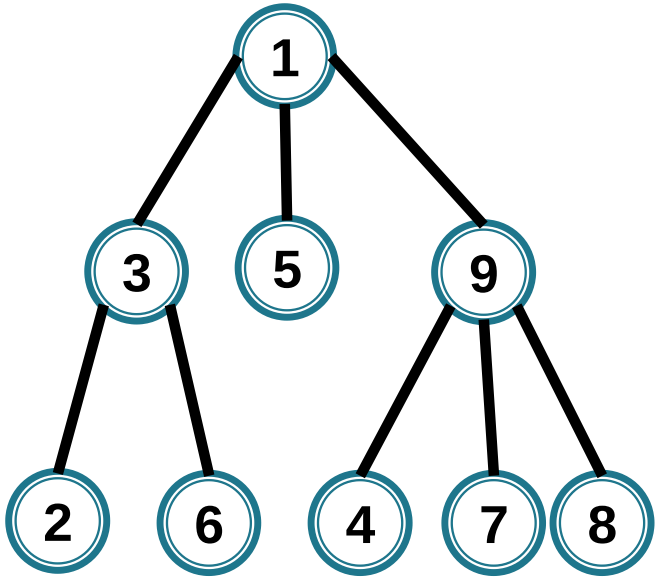
~~1~~ ~~3~~ ~~5~~ 9 2 6



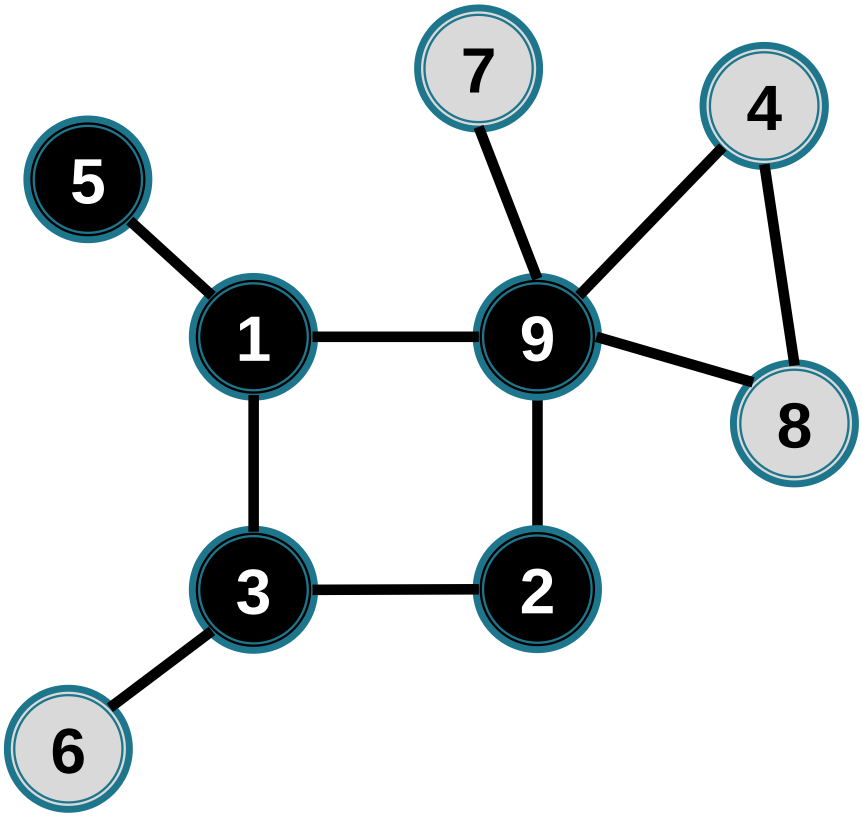


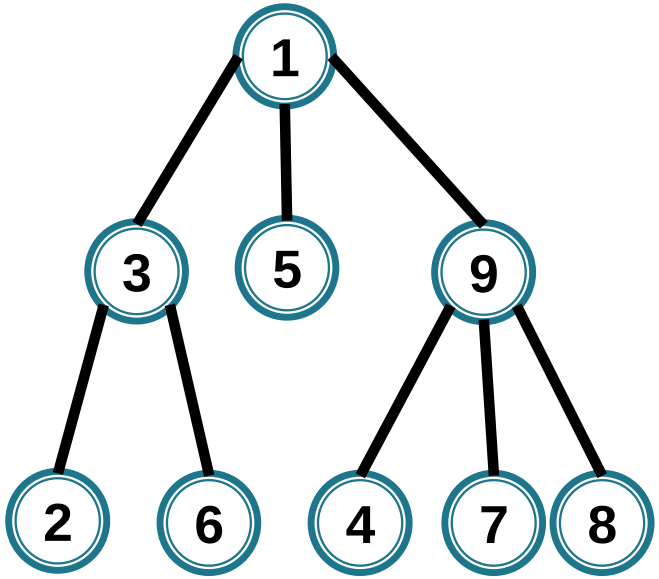
~~1~~ ~~3~~ ~~5~~ ~~9~~ 2 6 4 7 8



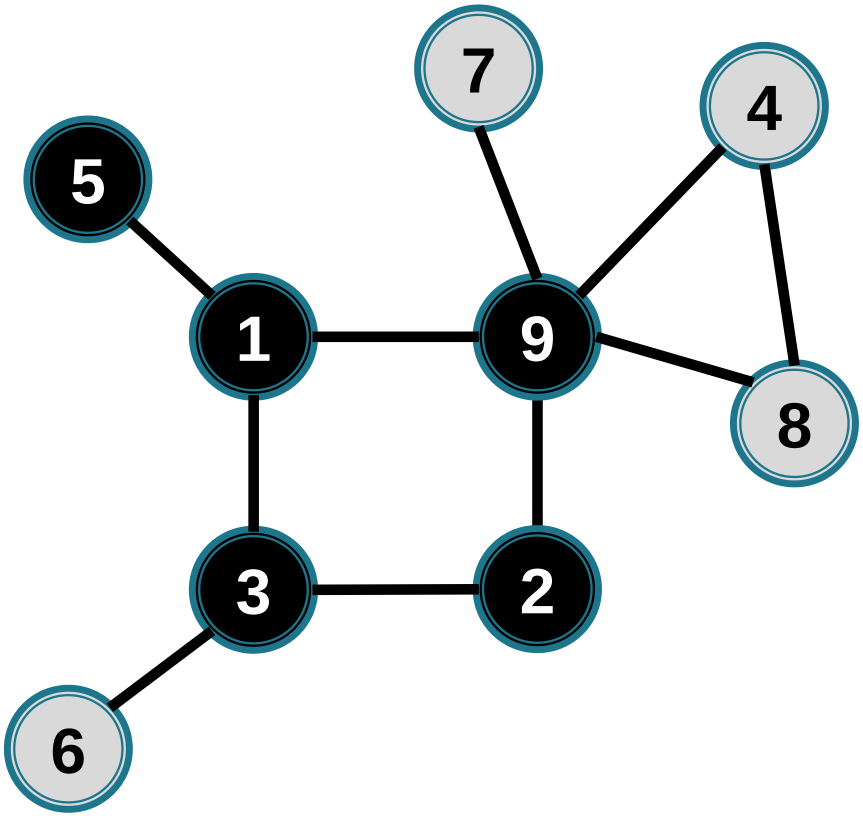


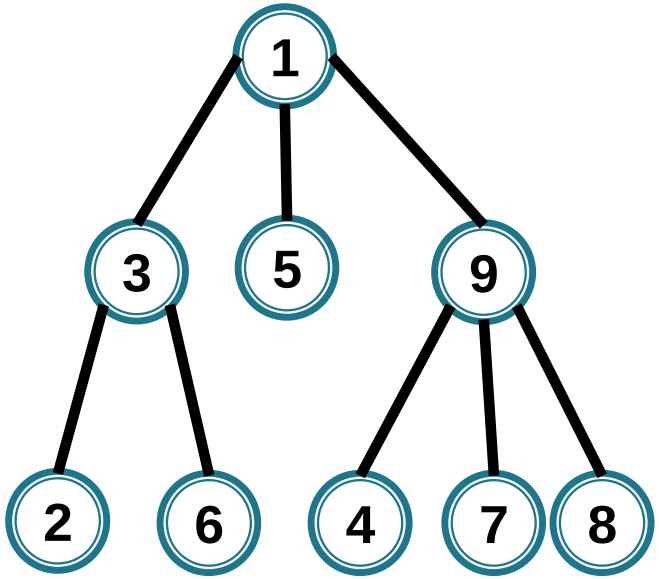
~~1~~ ~~3~~ ~~5~~ ~~9~~ 2 6 4 7 8



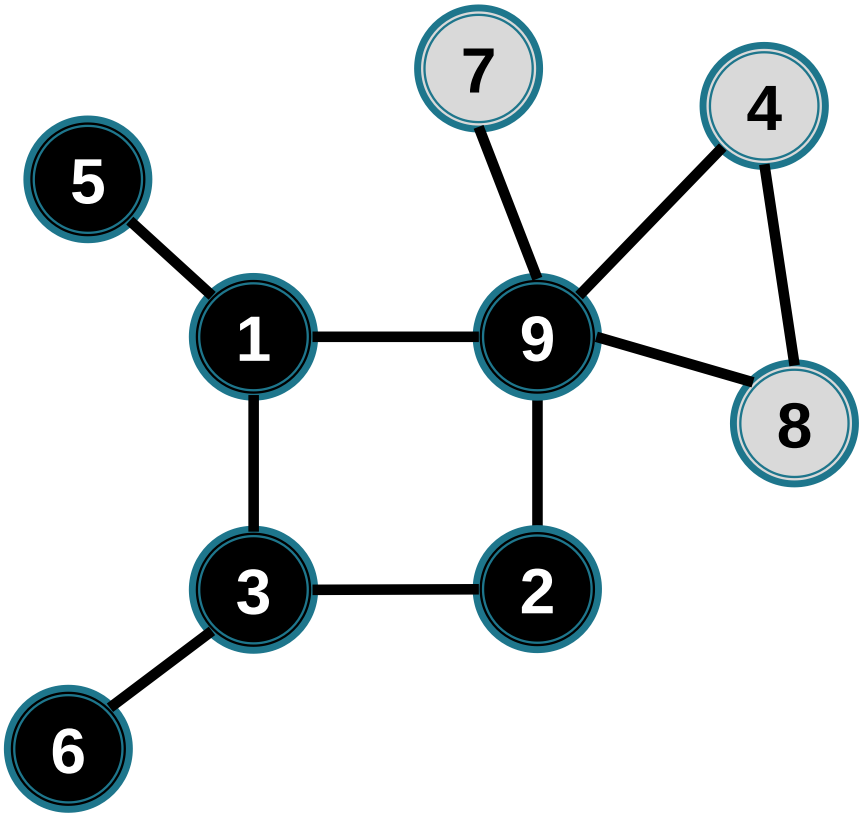


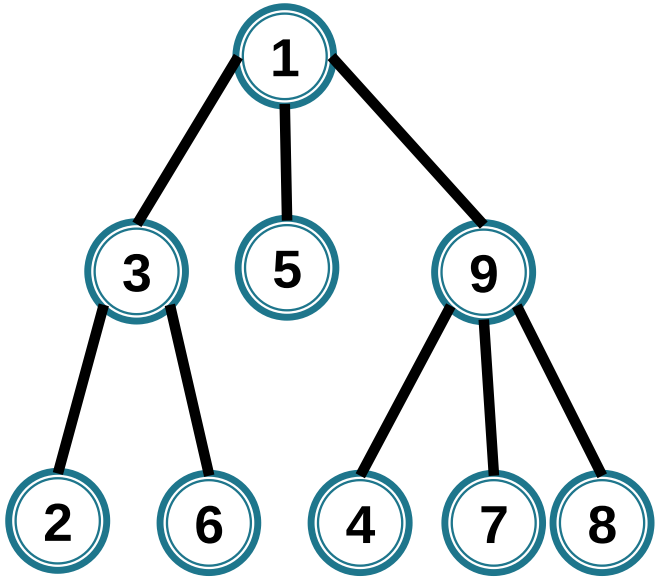
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ 6 4 7 8



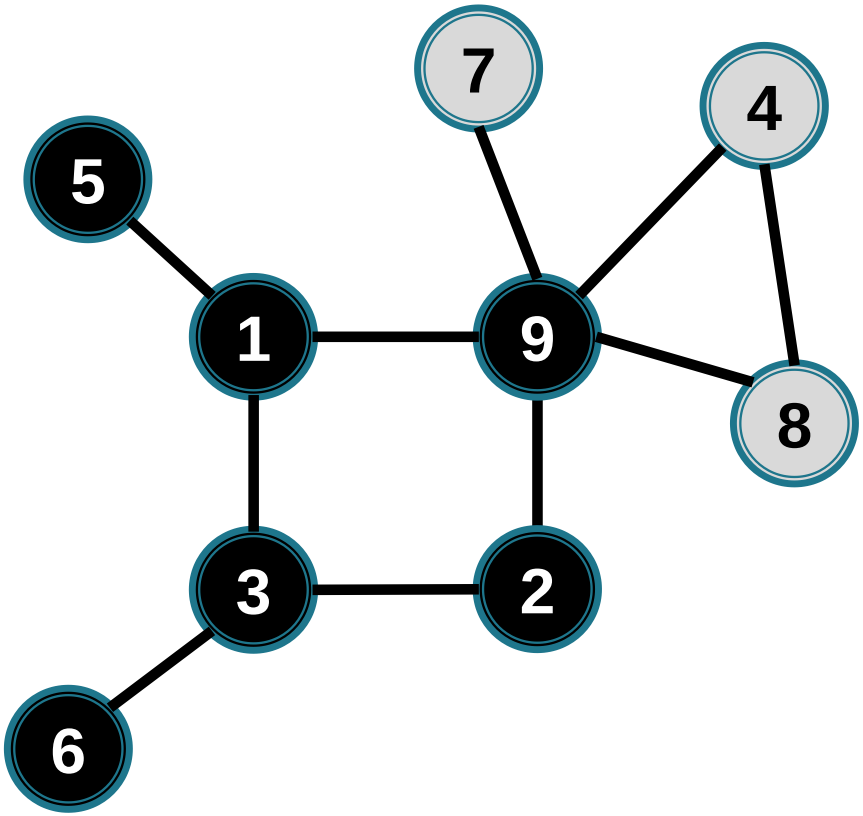


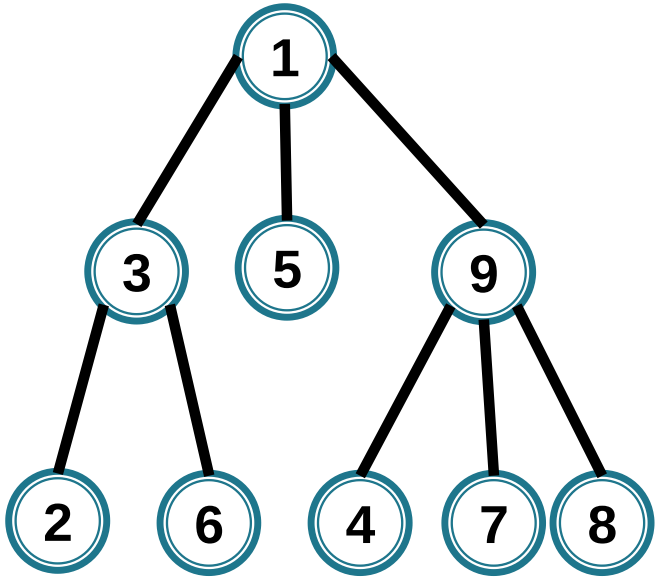
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ 6 4 7 8



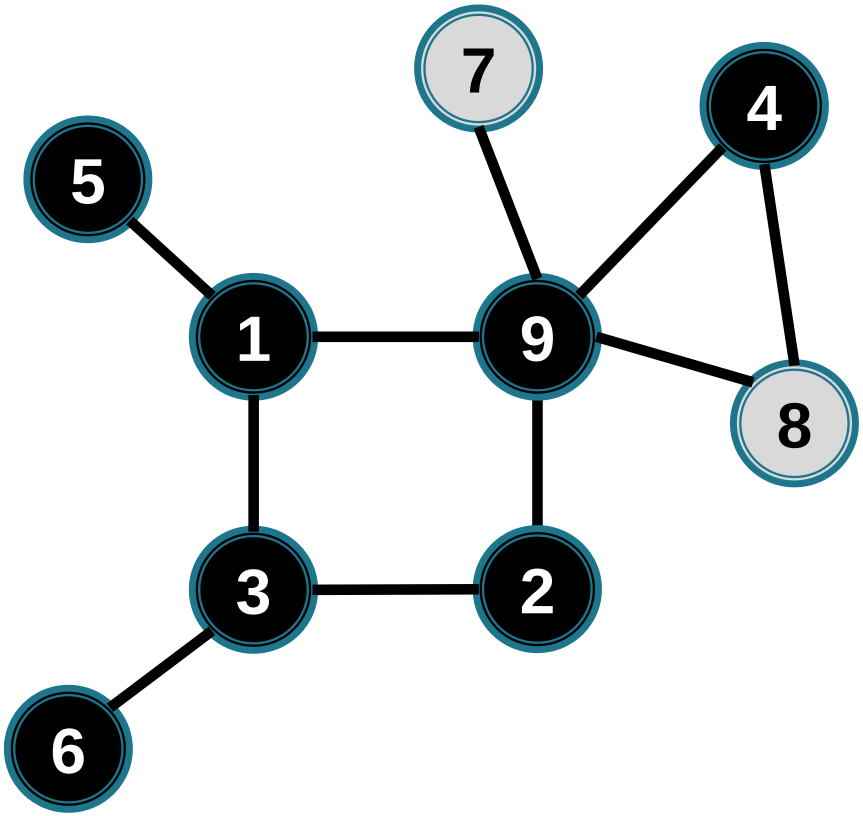


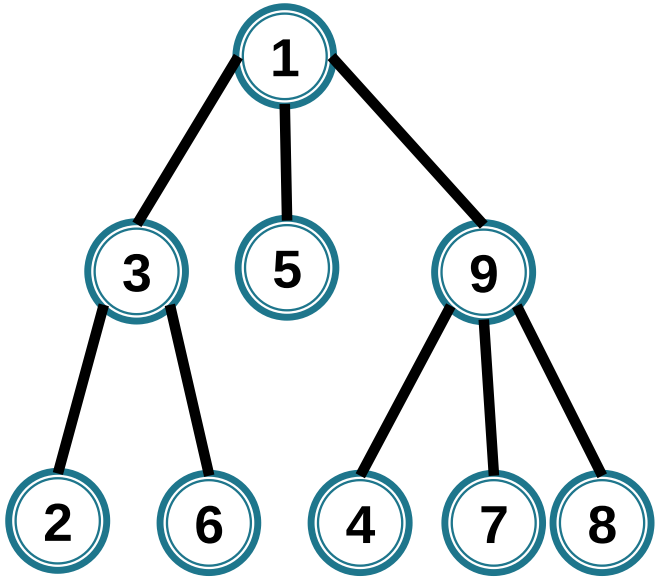
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ 4 7 8



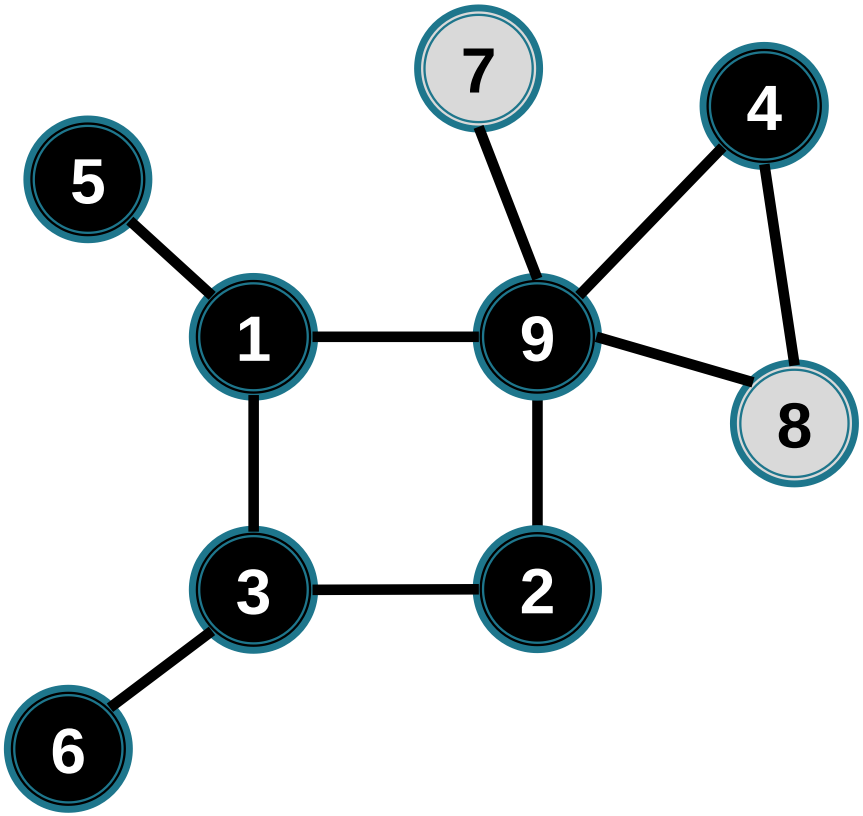


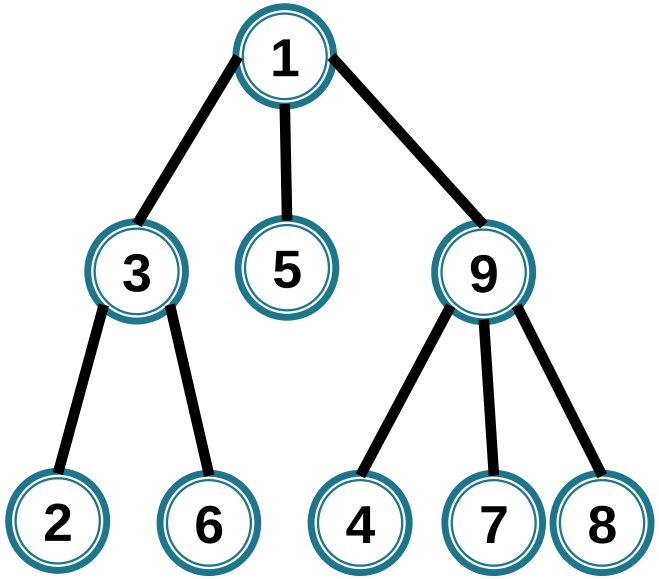
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ 4 7 8



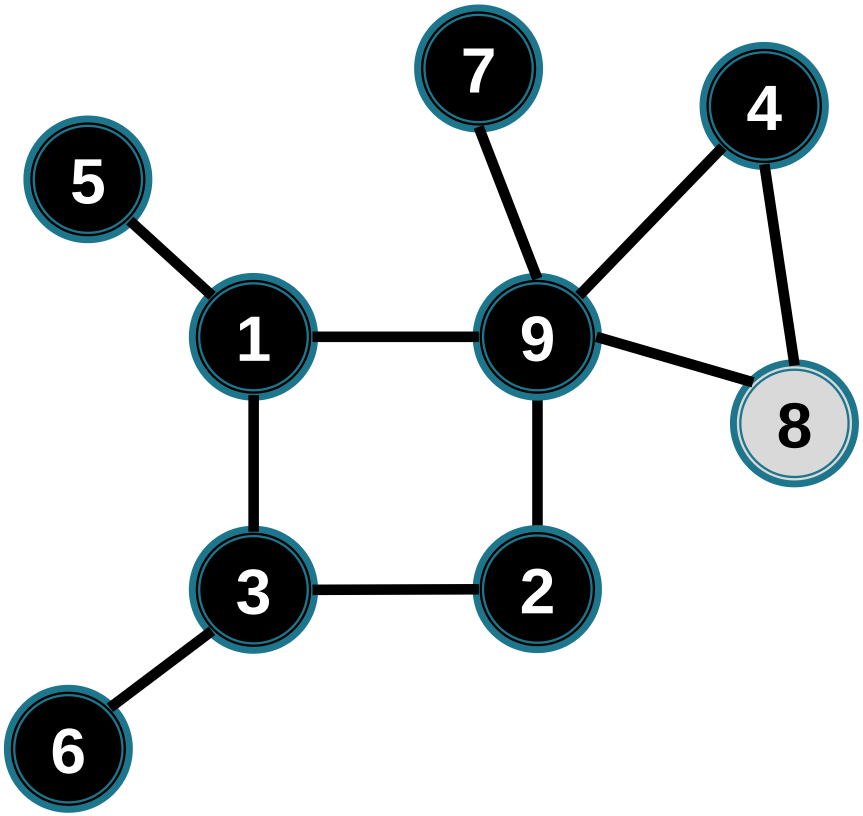


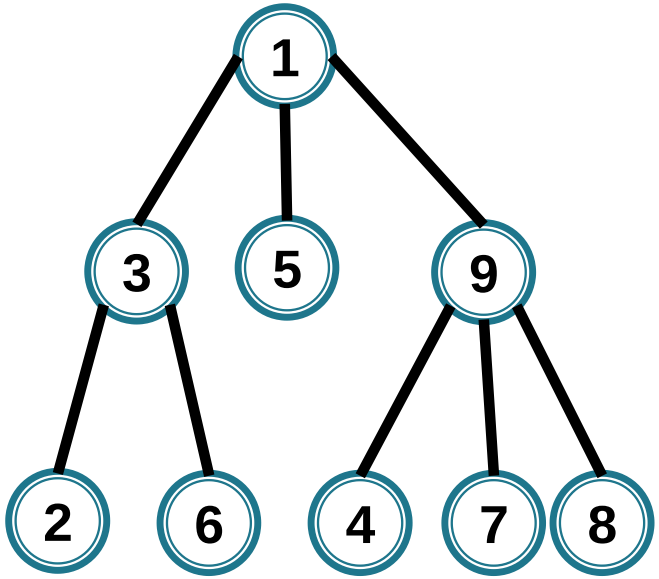
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ 7 8



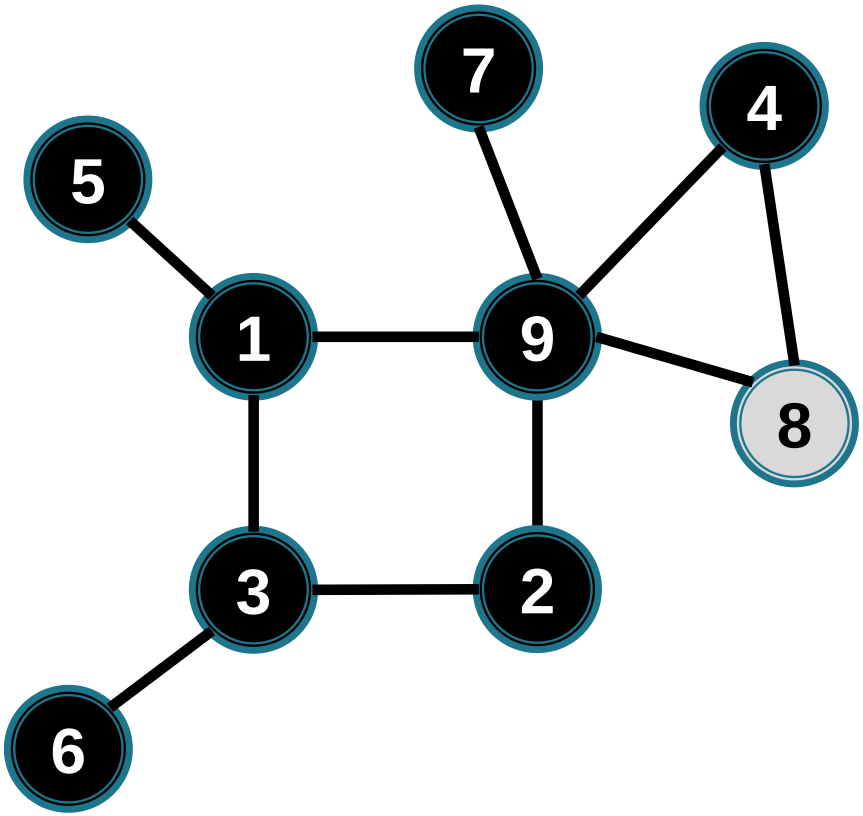


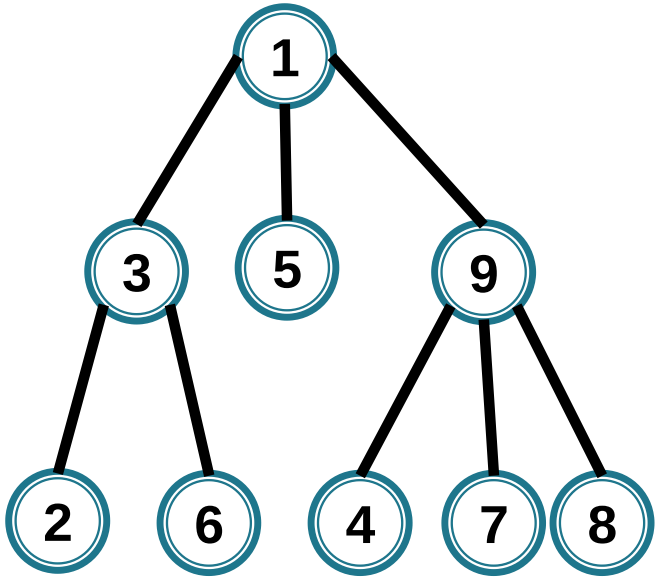
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ 7 8



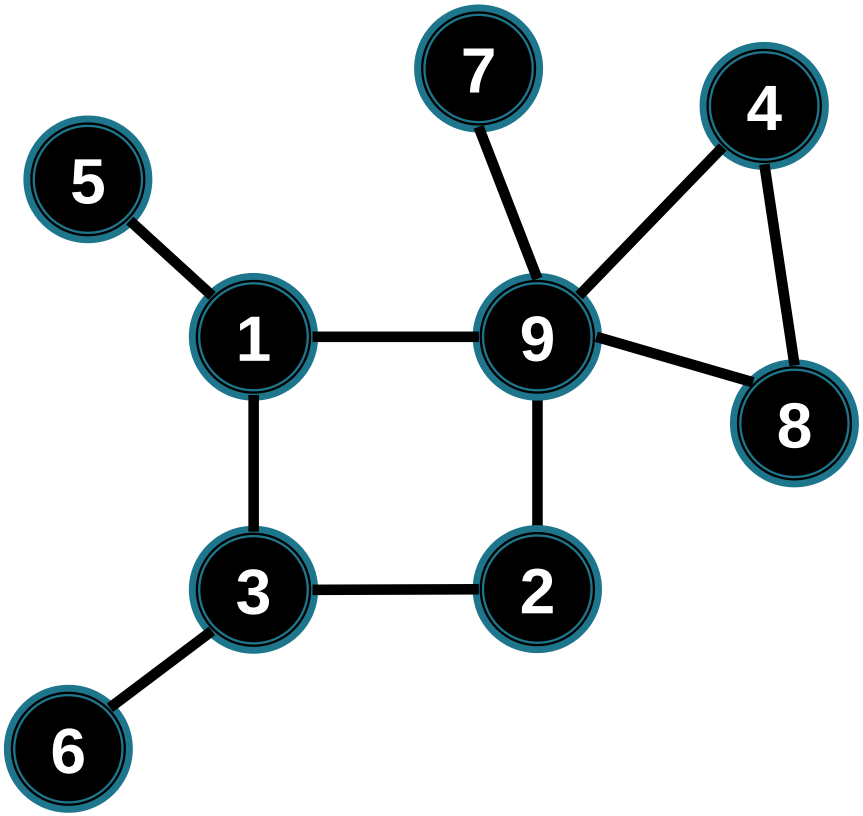


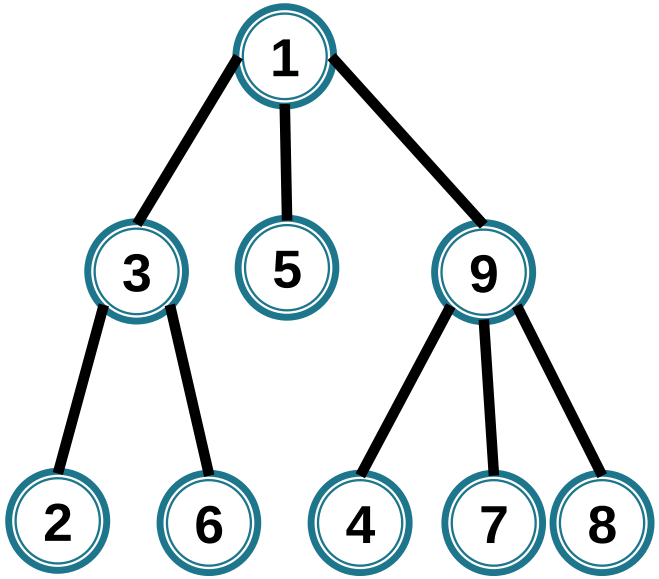
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ 8





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





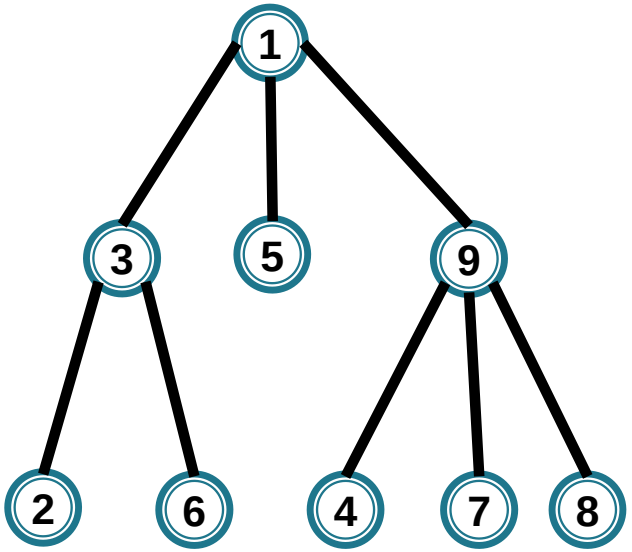
~~1~~ ~~3~~ ~~5~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ ~~8~~

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 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:

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

Parcurgerea în lățime

- Informații necesare:

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

Opţional

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

  => **arborele BF**

Parcurgerea în lățime

- Informații necesare:

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

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**
  = **distanța de la s la j (vom demonstra)**

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

Parcurgerea în lățime

- Informații necesare:

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

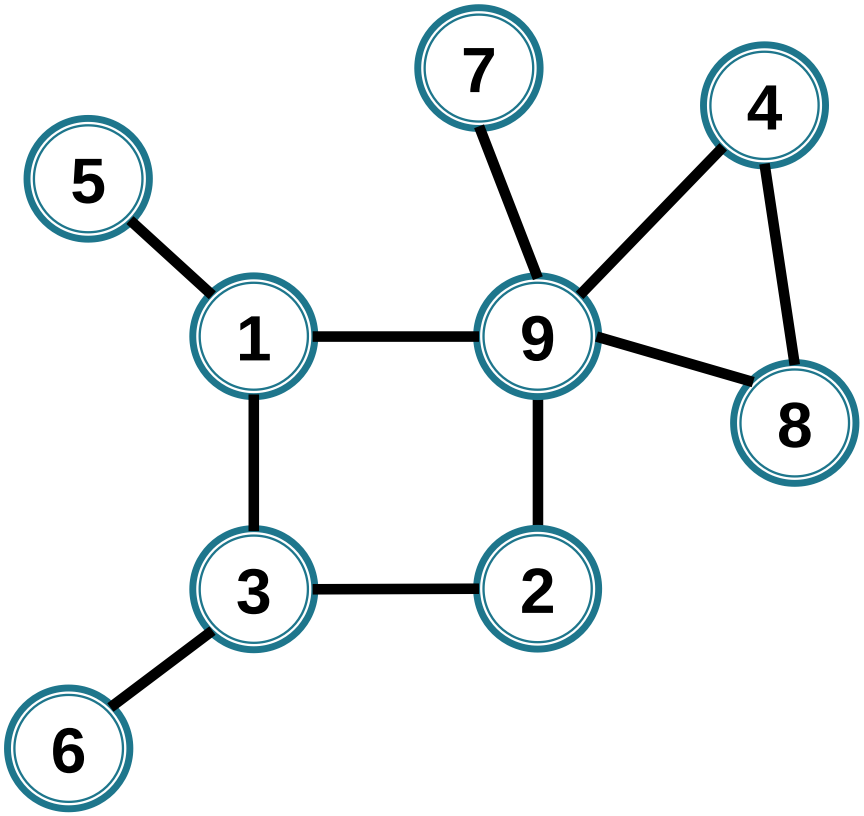
Se poate nuanța:

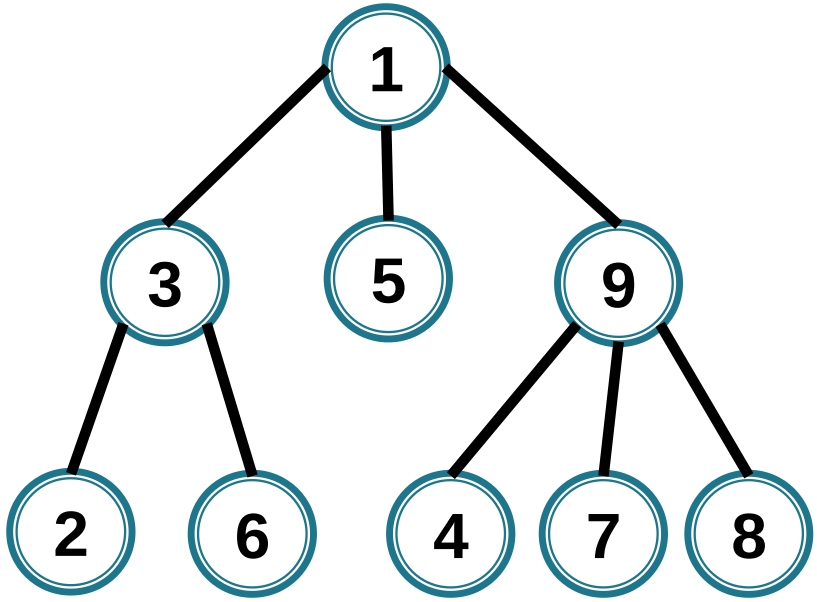
- **alb** –nevizitat
- **gri** – in explorare
- **negru** – finalizat

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**
  = **distanța de la s la j (vom demonstra)**

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)
```

Apel

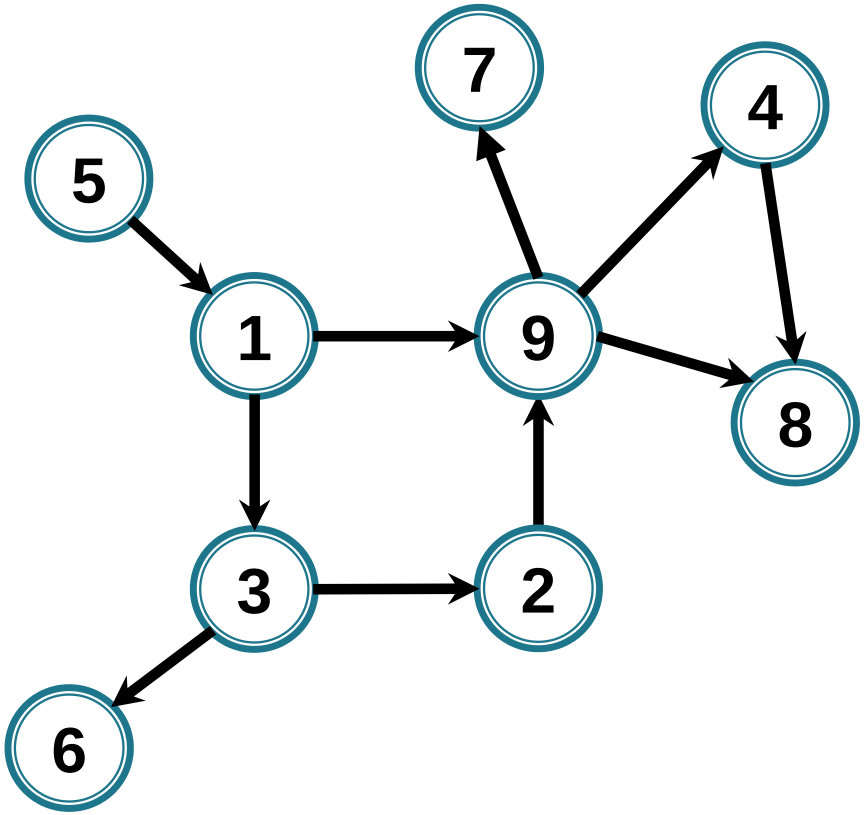
- Pentru un vârf s procedura de parcurgere se apelează

```
BFS(s)
```

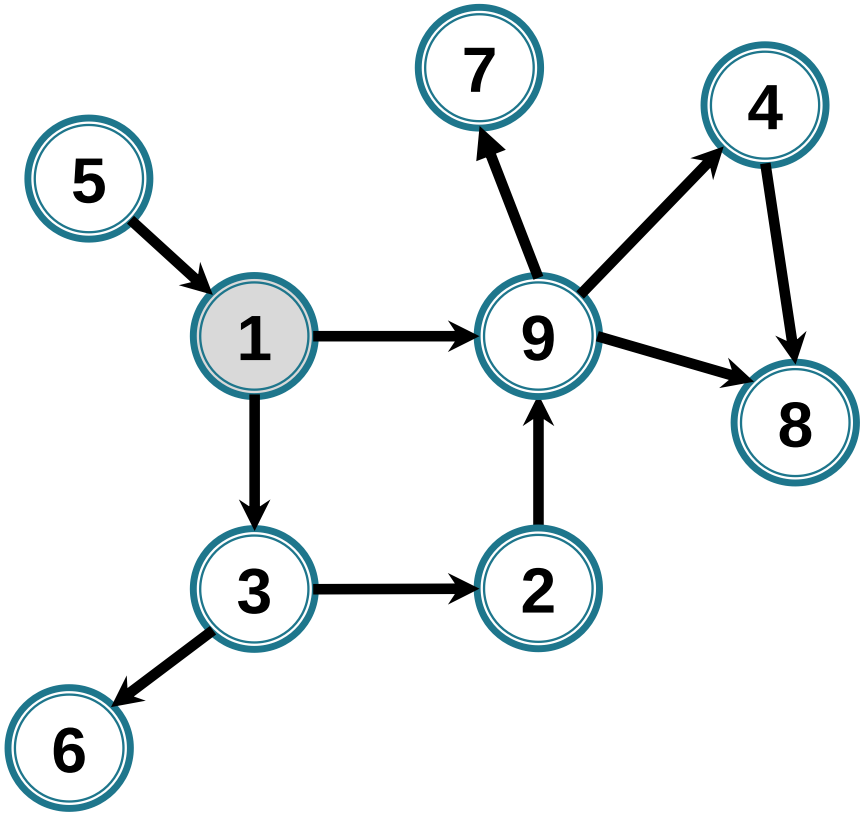
- Pentru a parcurge **toate vârfurile grafului** se reia apelul subprogramului BFS pentru vârfuri rămase nevizitate:

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

Exemplu pentru graf orientat cu calculul distanței

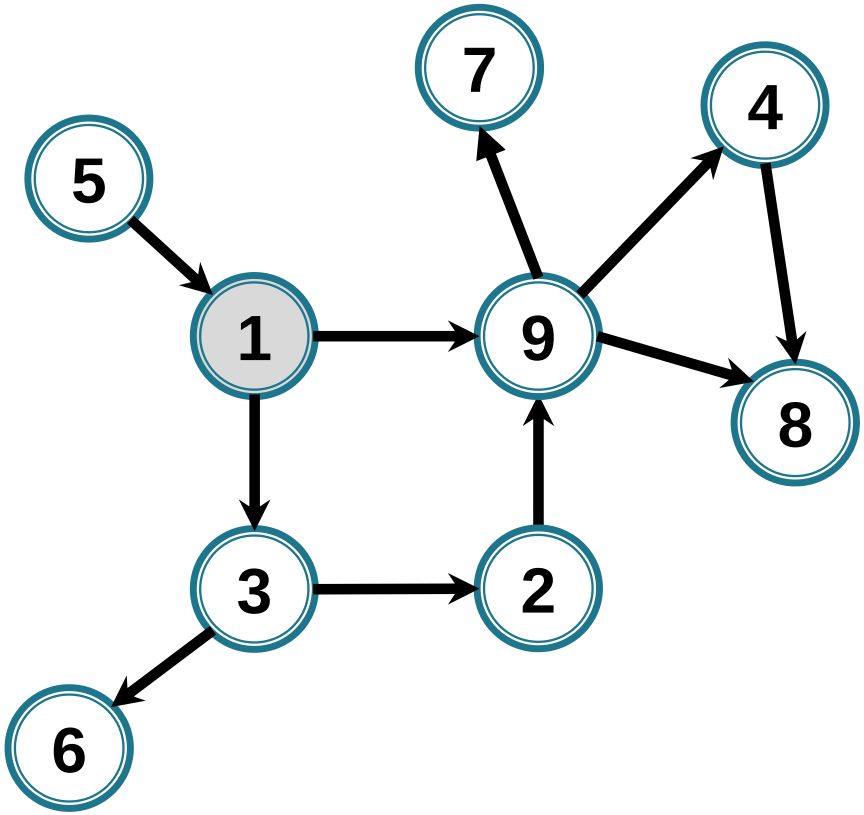


Vârf de start 1



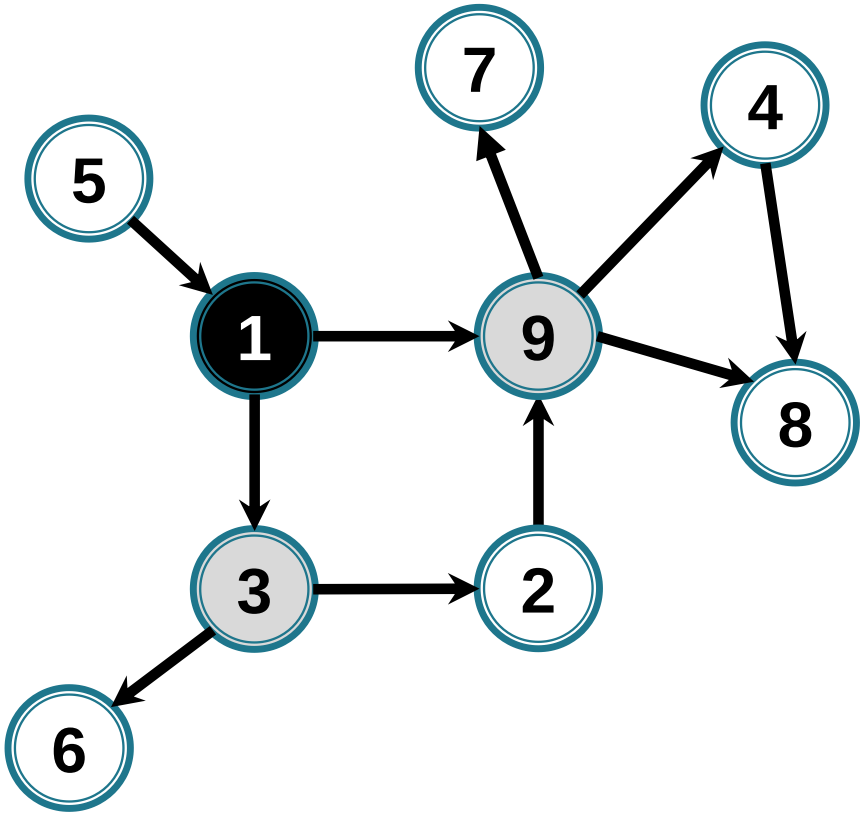
	1	2	3	4	5	6	7	8	9
tata	0	0	0	0	0	0	0	0	0
d	0	∞	∞	∞	∞	∞	∞	∞	∞

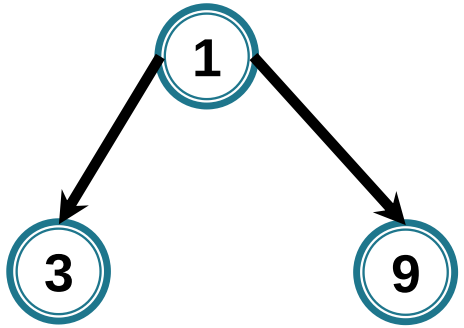
c: 1



	1	2	3	4	5	6	7	8	9
tata	0	0	0	0	0	0	0	0	0
d	0	∞	∞	∞	∞	∞	∞	∞	∞

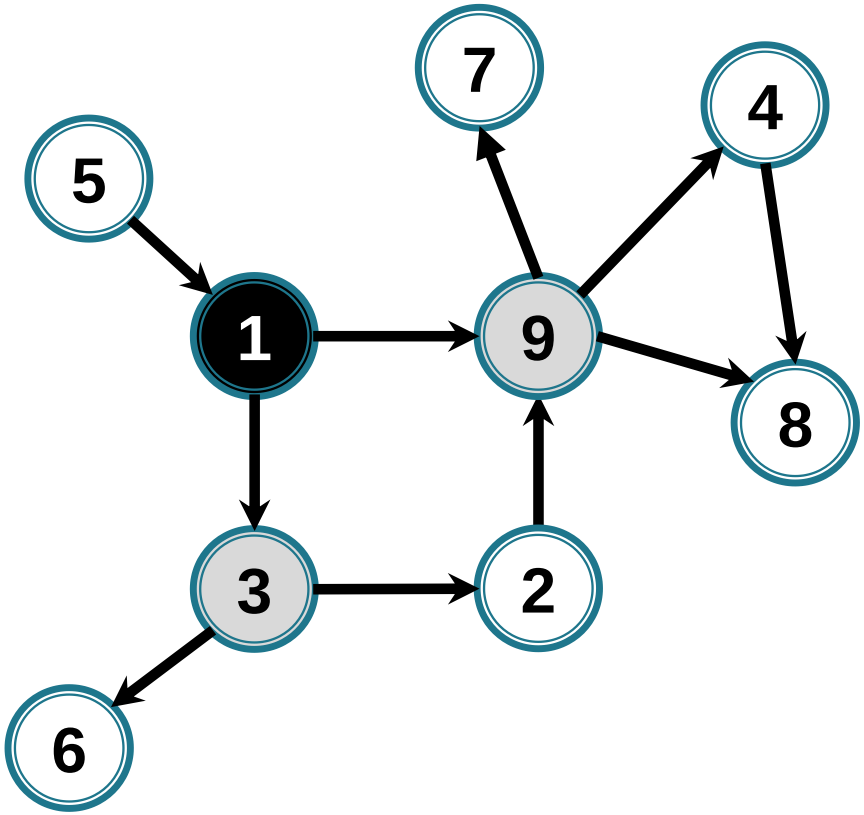
c: 1

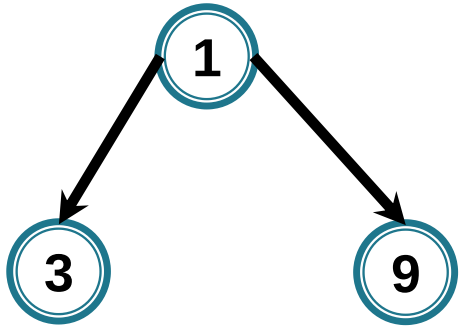




	1	2	3	4	5	6	7	8	9
tata	0	0	1	0	0	0	0	0	1
d	0	∞	1	∞	∞	∞	∞	∞	1

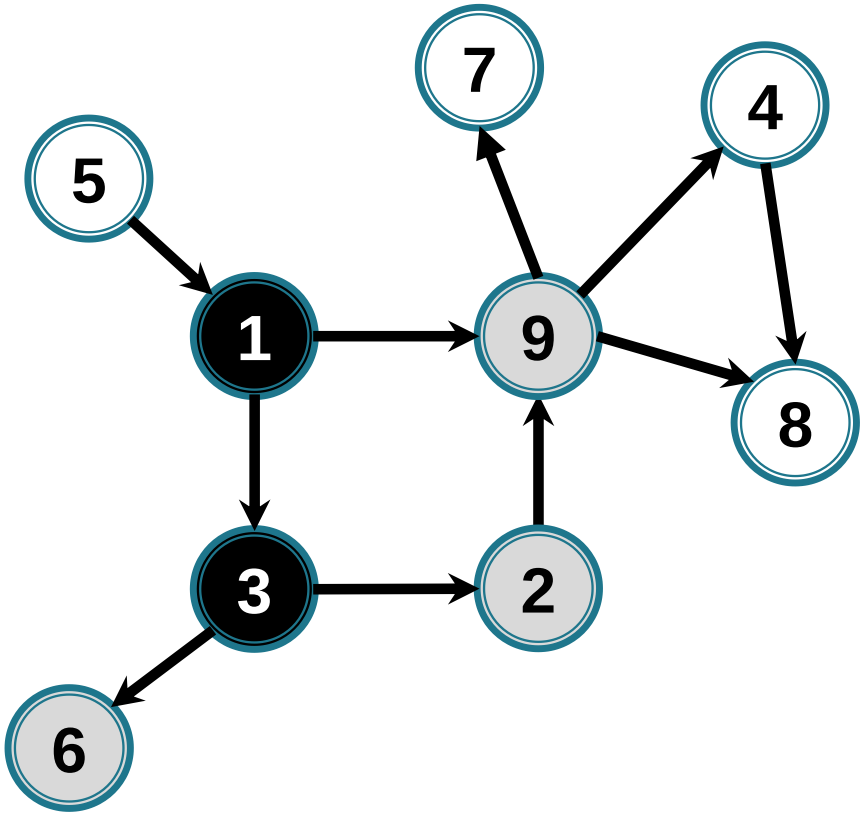
c: ~~1~~ 3 9

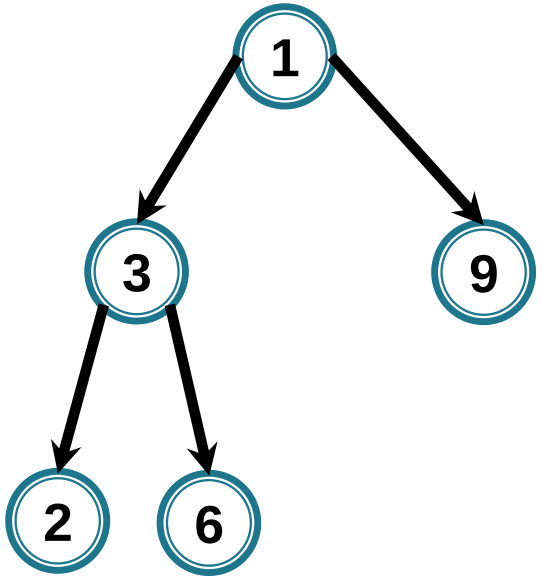




	1	2	3	4	5	6	7	8	9
tata	0	0	1	0	0	0	0	0	1
d	0	∞	1	∞	∞	∞	∞	∞	1

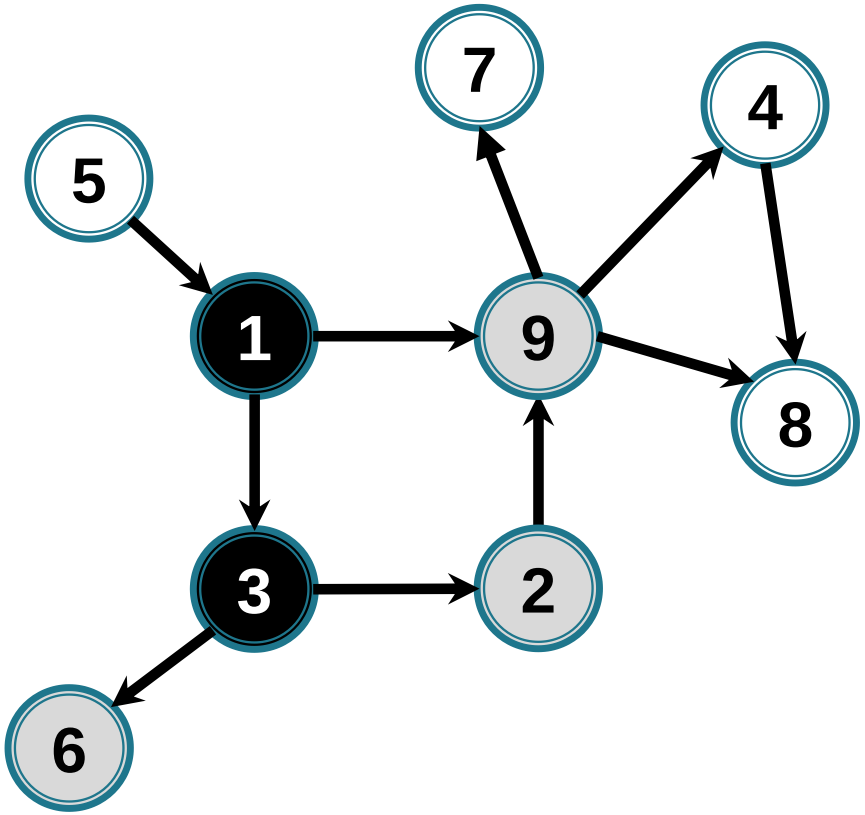
c: ~~1~~ 3 9

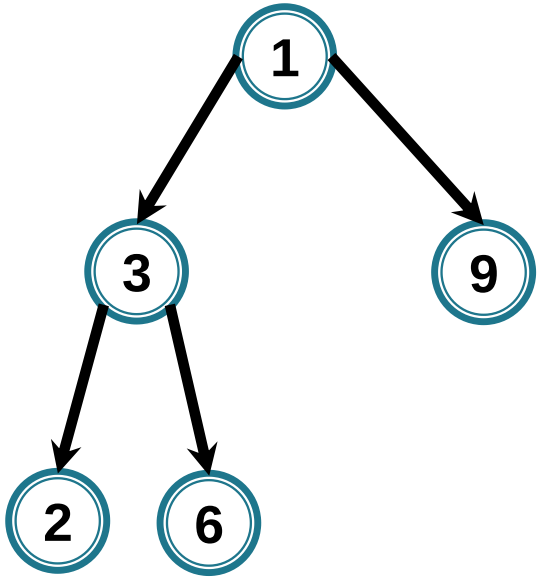




	1	2	3	4	5	6	7	8	9
tata	0	3	1	0	0	3	0	0	1
d	0	2	1	∞	∞	2	∞	∞	1

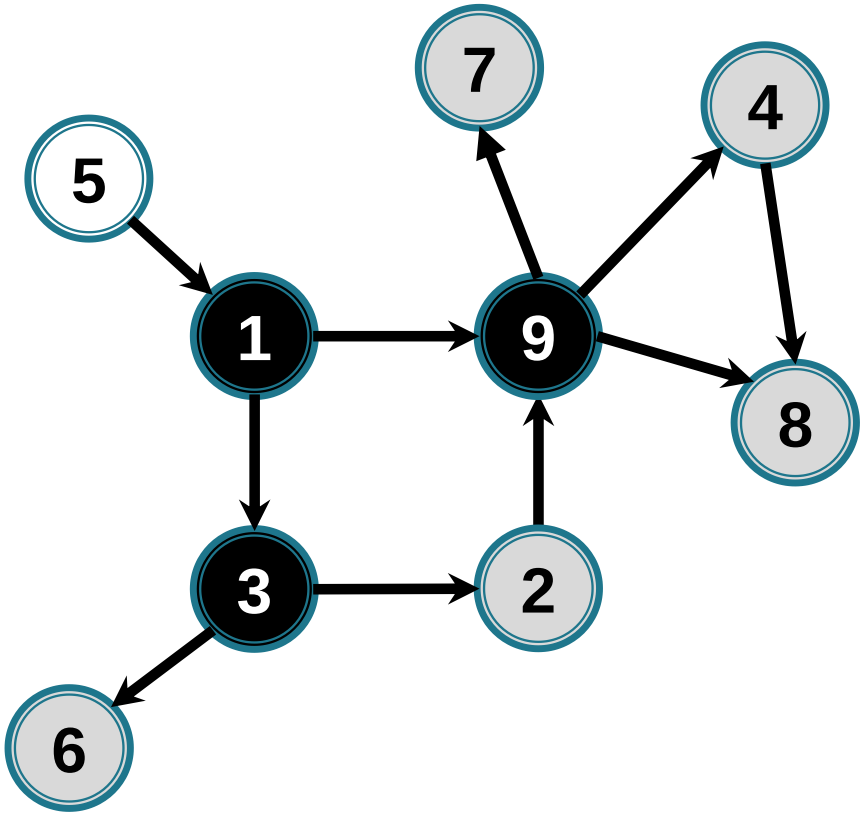
c: ~~1~~ ~~3~~ 9 2 6

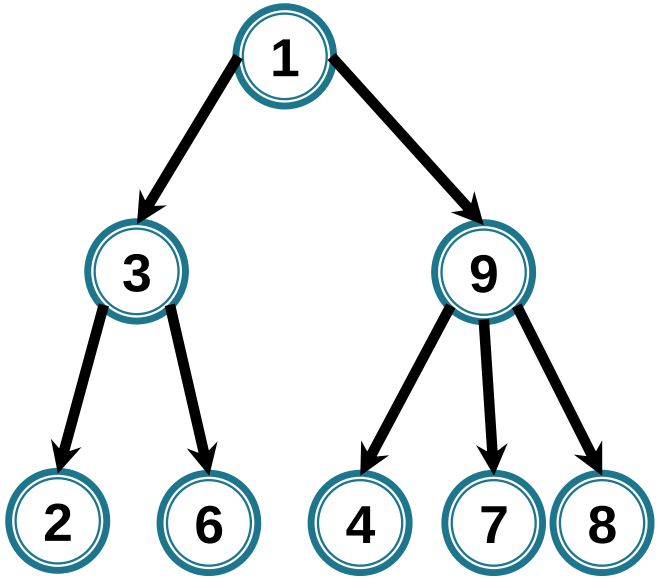




	1	2	3	4	5	6	7	8	9
tata	0	3	1	0	0	3	0	0	1
d	0	2	1	∞	∞	2	∞	∞	1

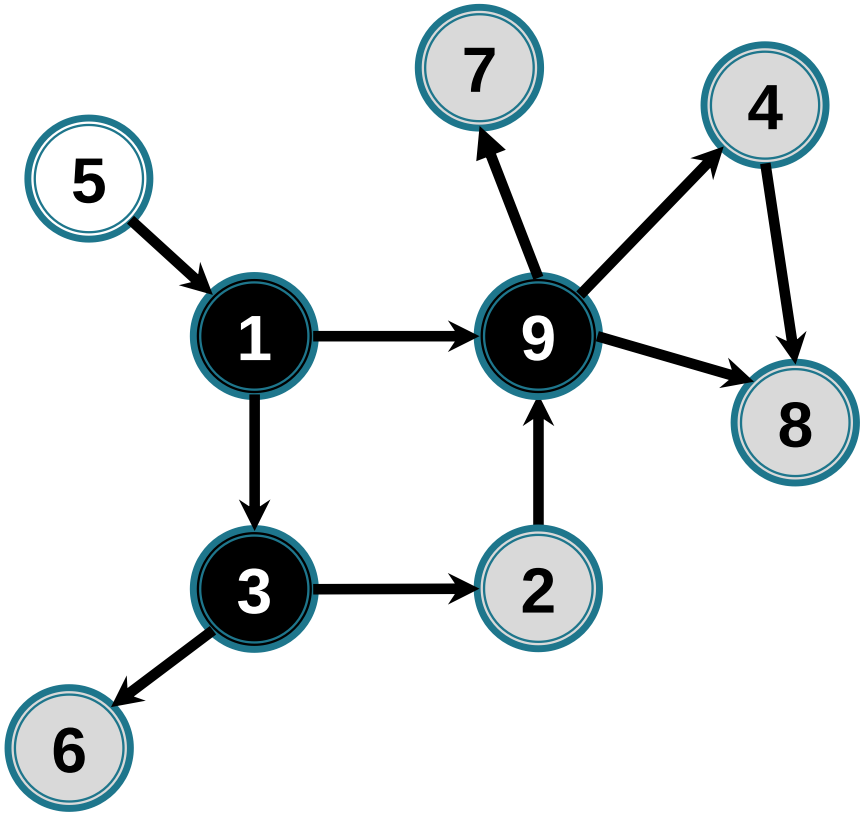
c: ~~1~~ ~~3~~ 9 2 6

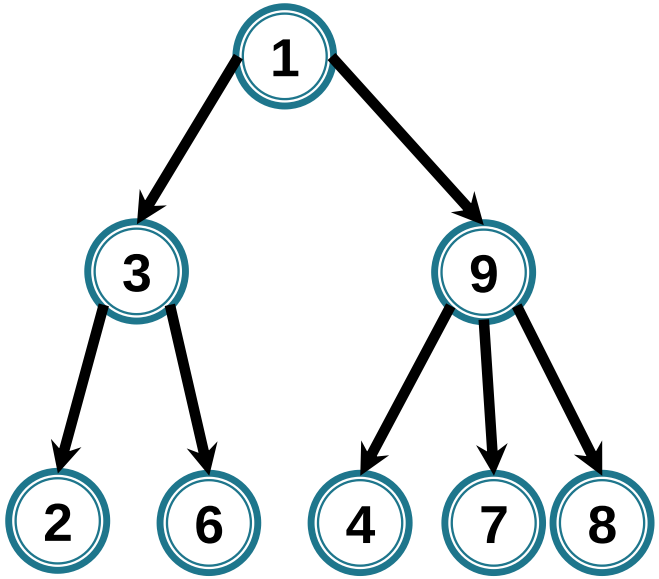




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

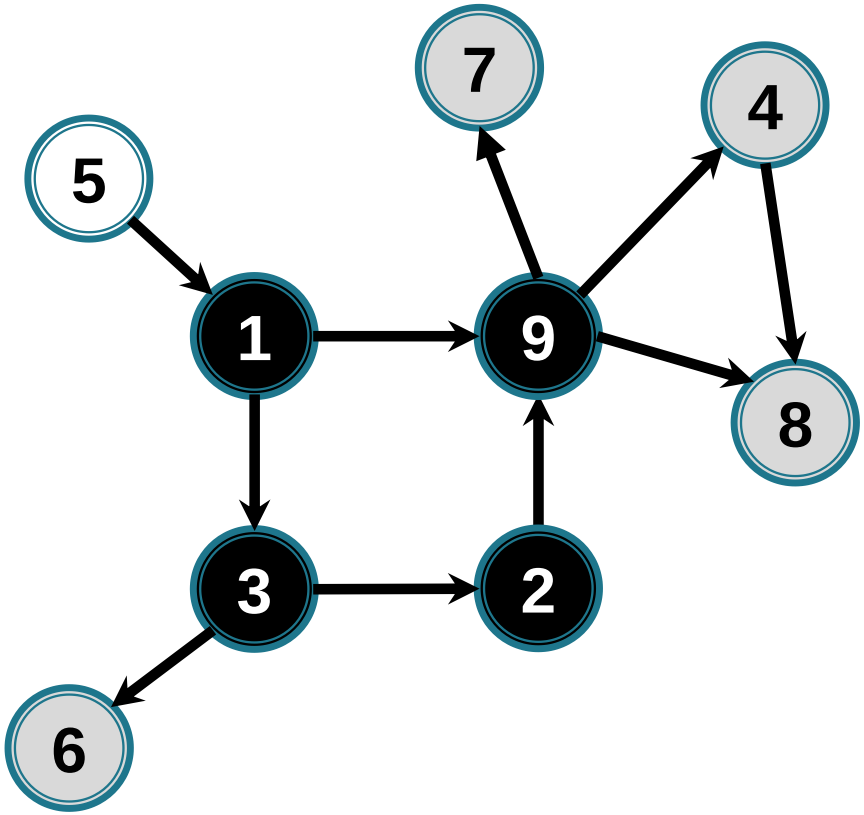
c: ~~1~~ ~~3~~ ~~9~~ 2 6 4 7 8

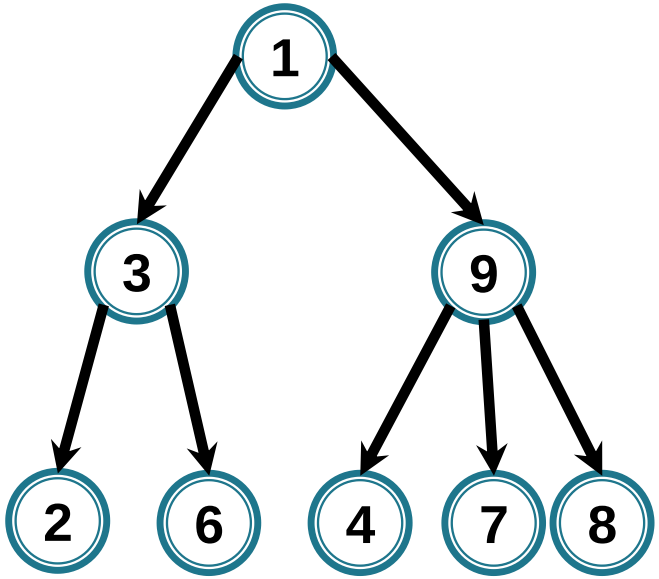




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

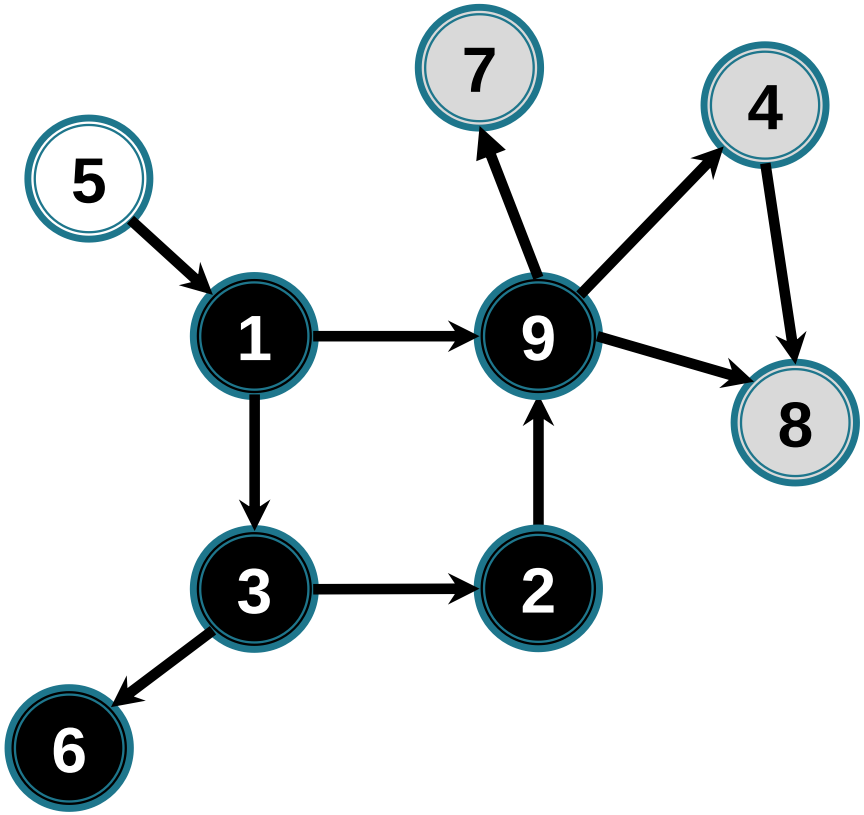
c: ~~1~~ ~~3~~ ~~9~~ 2 6 4 7 8

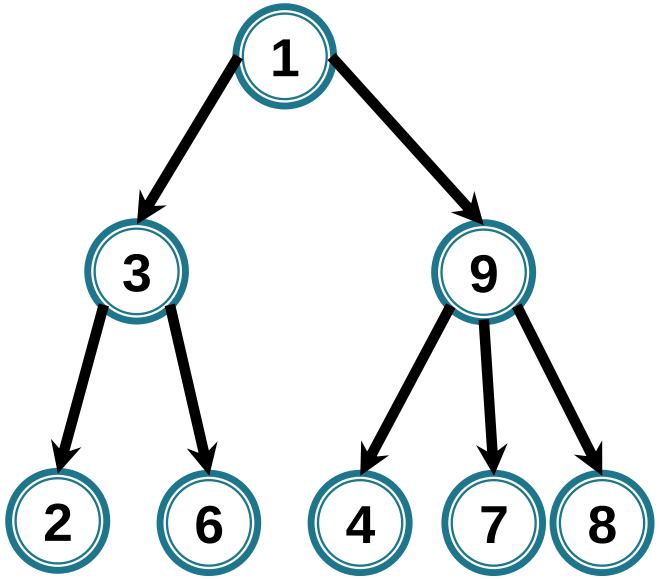




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

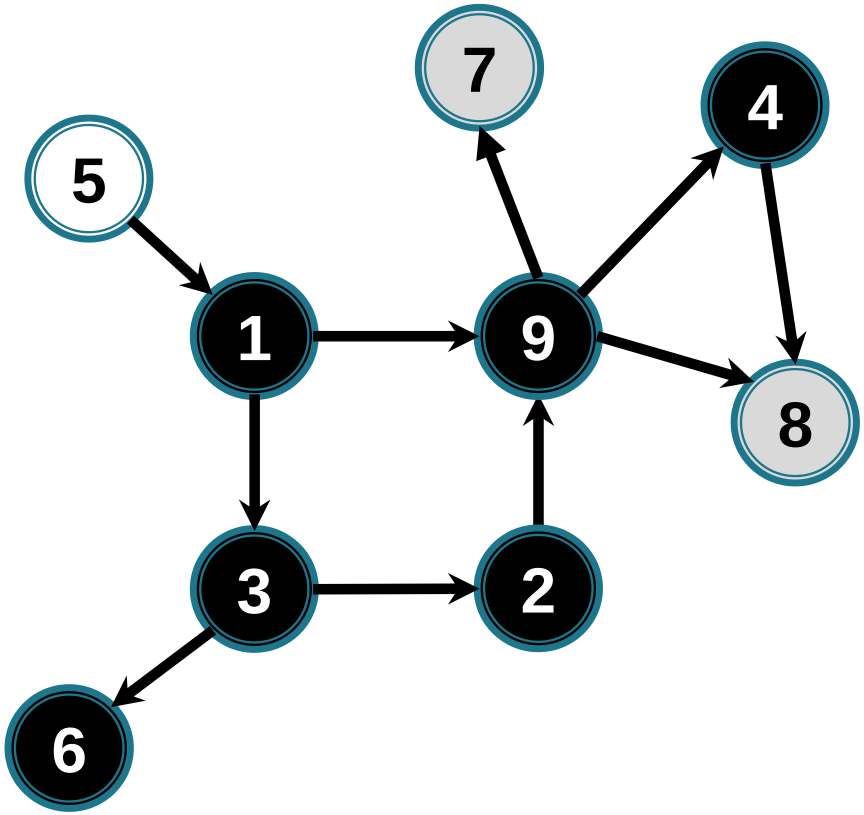
c: ~~1~~ ~~3~~ ~~9~~ ~~2~~ 6 4 7 8

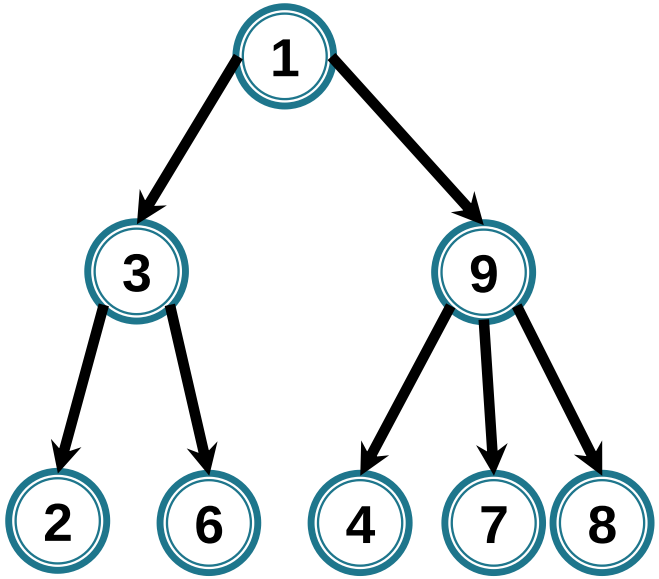




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

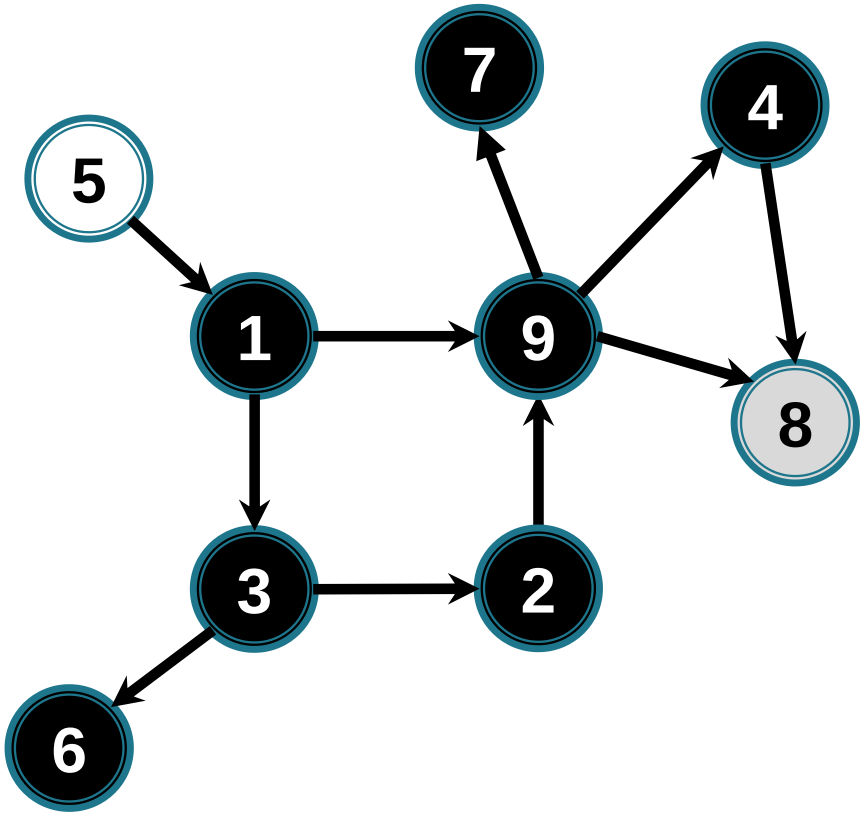
c: ~~1~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ 4 7 8

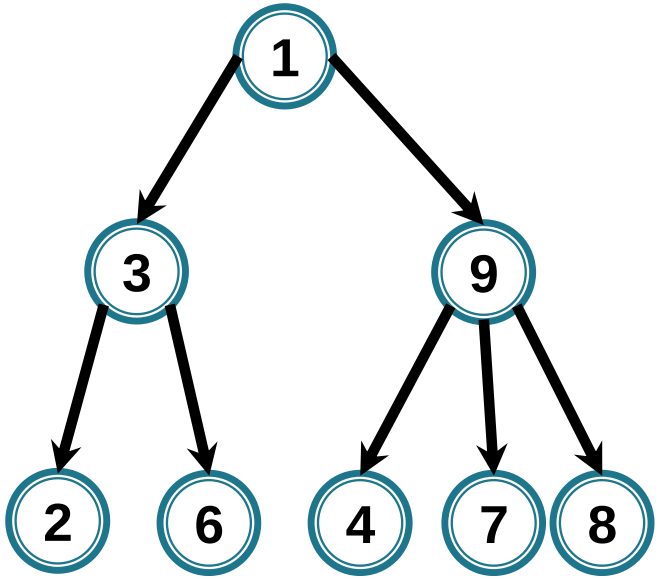




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

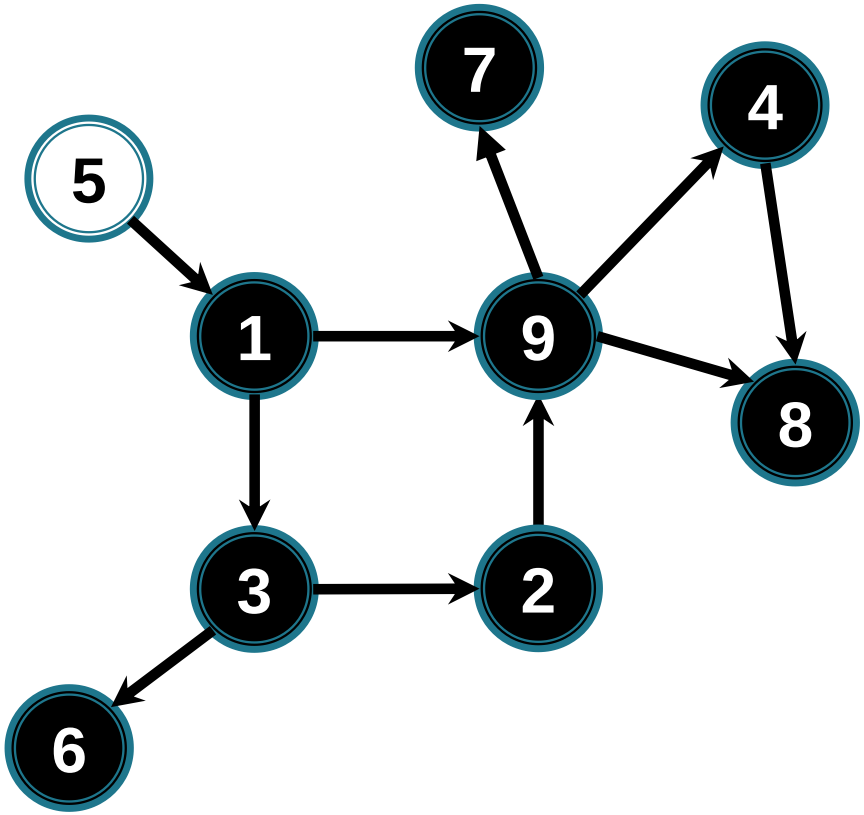
c: ~~1~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ 7 8

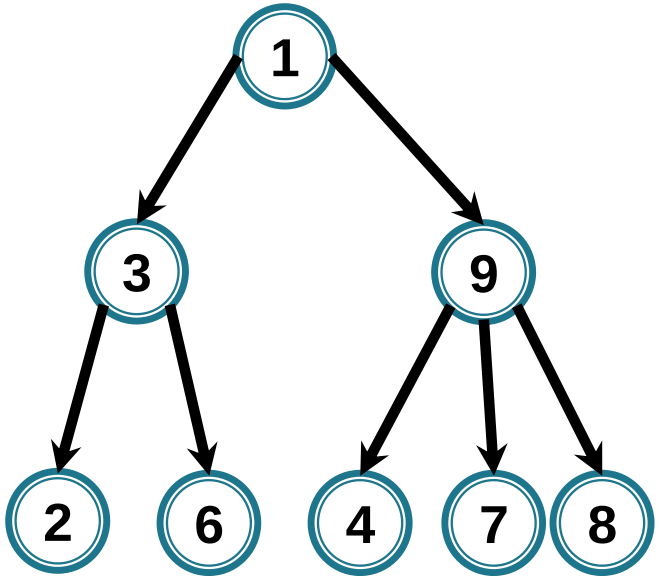




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

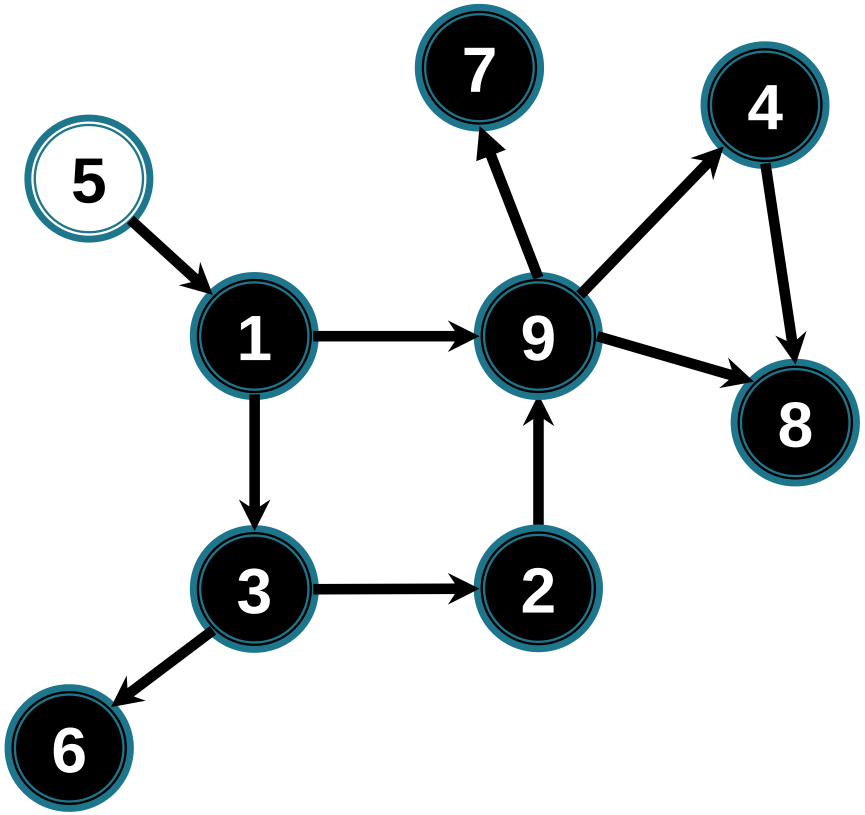
c: ~~1~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ ~~4~~ ~~7~~ 8

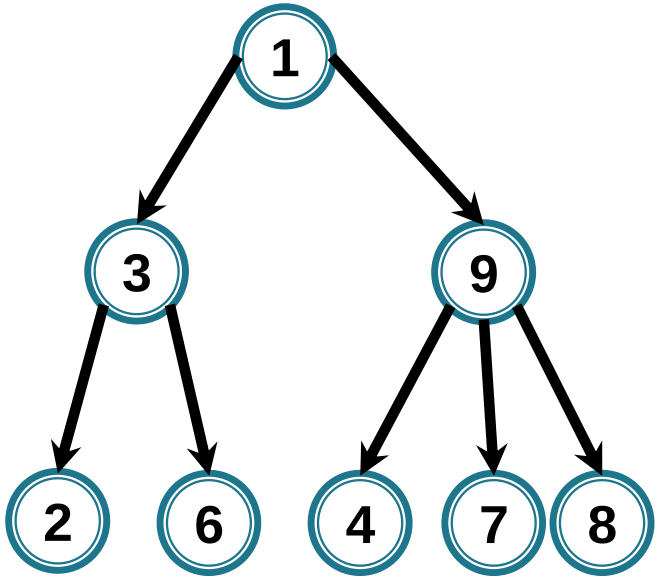




	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

c: ~~1~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ 4 ~~7~~ ~~8~~





	1	2	3	4	5	6	7	8	9
tata	0	3	1	9	0	3	9	9	1
d	0	2	1	2	∞	2	2	2	1

C: ~~1~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ 4 ~~7~~ ~~8~~

d: 0 1 1 2 2 2 2 2

Complexitate

Complexitate

- **Inițializări O(n)** + **Pentru fiecare vârf i** se execută cel mult o dată instrucțiunea pentru de parcurgere a vecinilor:

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

Complexitate

- **Inițializări O(n)** + **Pentru fiecare vârf i** se execută cel mult o dată instrucțiunea pentru de parcurgere a vecinilor:

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

De câte ori se execută corpul acestei instrucțiunii repetitive (care conține un număr constant de operații) ?

Complexitate

- **Inițializări O(n)** + **Pentru fiecare vârf i** se execută cel mult o dată instrucțiunea pentru de parcurgere a vecinilor:

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

- **De câte ori se execută corpul acestei instrucțiunii repetitive (care conține un număr constant de operații) ?**

Depinde de modalitatea de reprezentare a grafului:

- **Matrice de adiacență** – j ia pe rând toate valorile de la 1 la 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

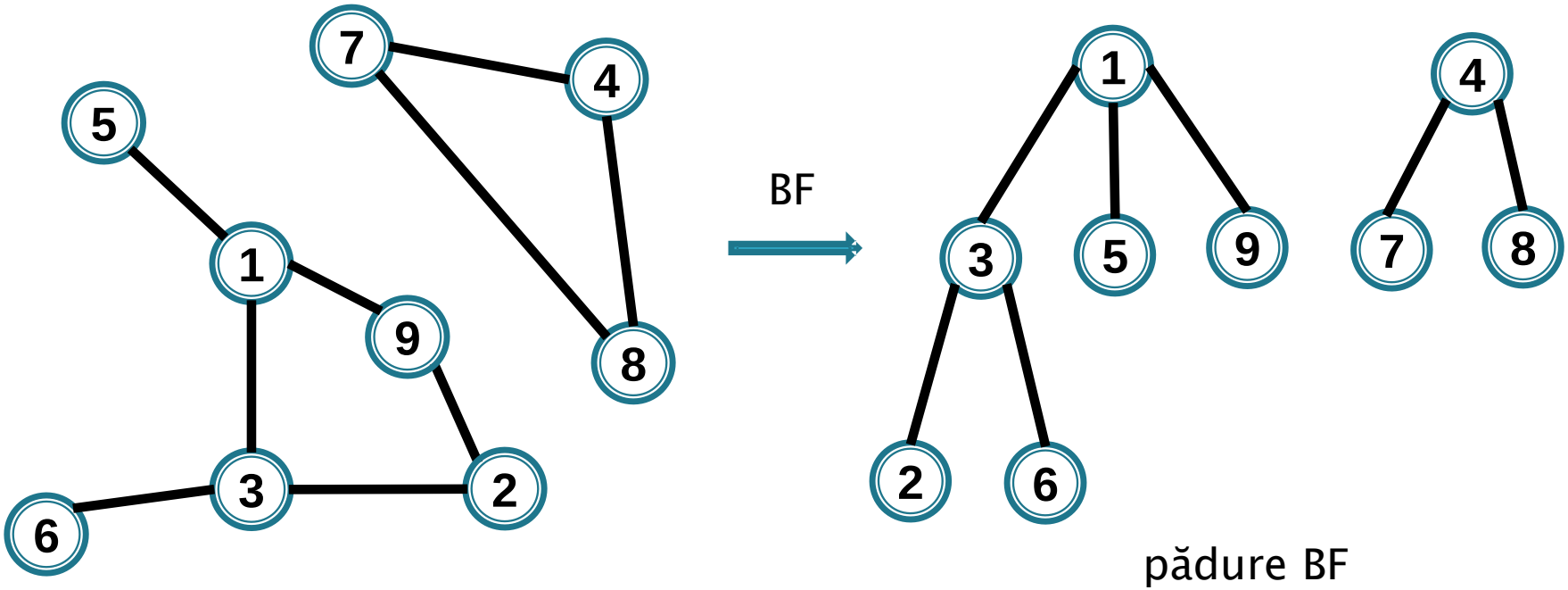
- **Matrice de adiacență** – $\boldsymbol{O(n + \sum_{i=1}^{n} n) = O(n^2)}$
- **Liste de adiacență** – $\boldsymbol{O(n + \sum_{i=1}^{n} d(i)) = O(n + m)}$

Proprietăți

Parcurgerea în lățime – graf neorientat

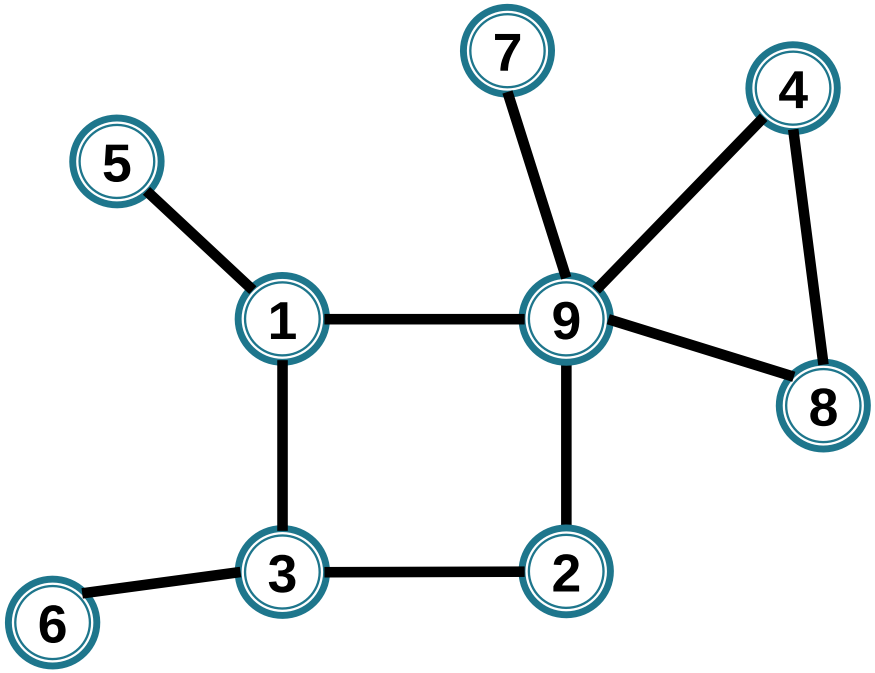
- Muchiile folosite pentru a descoperi vârfuri noi pornind din s formează un **arbore cu rădăcina s** (numit **arbore BF)**

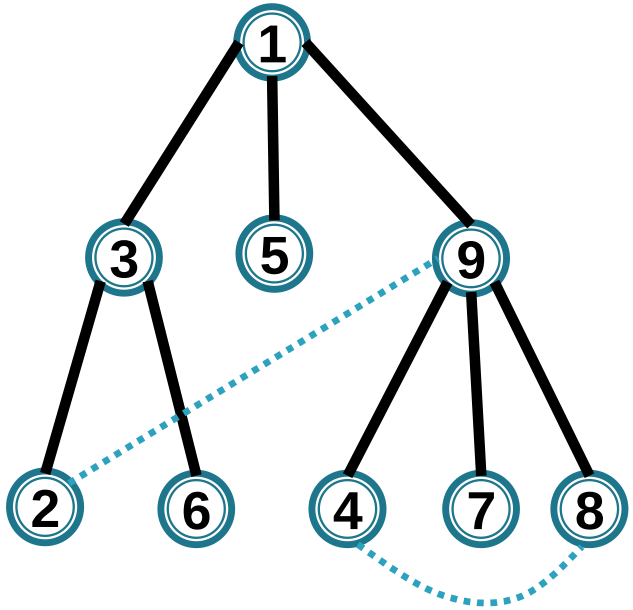
- Dacă parcurgem toate vârfurile => **pădure** BF (cu câte un arbore parțial pentru fiecare componentă)



Parcurgerea în lățime – graf neorientat

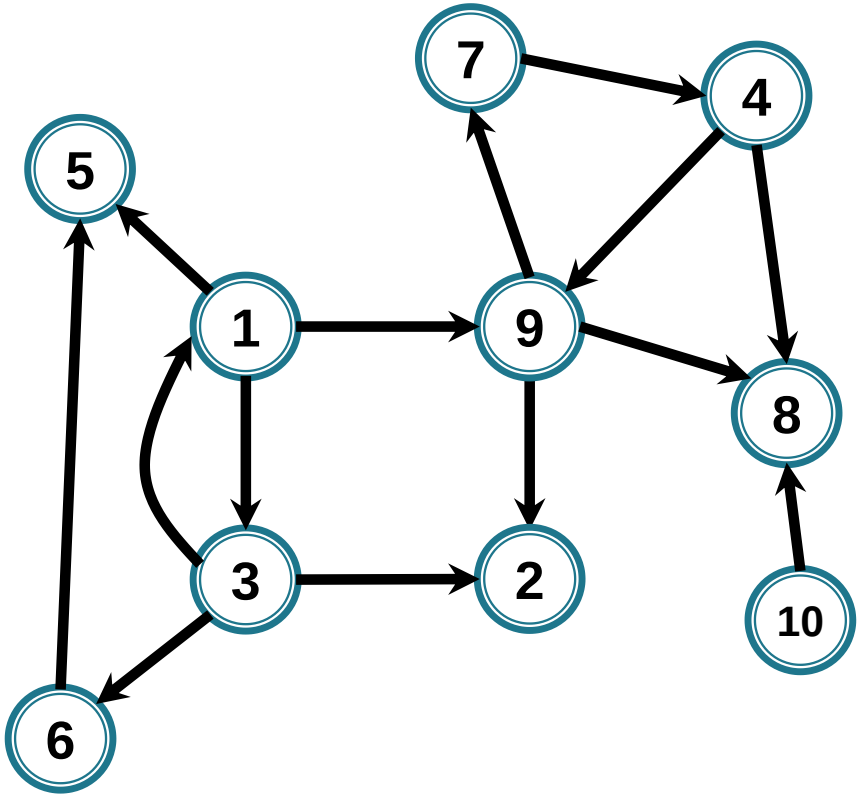
- Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore/pădure)





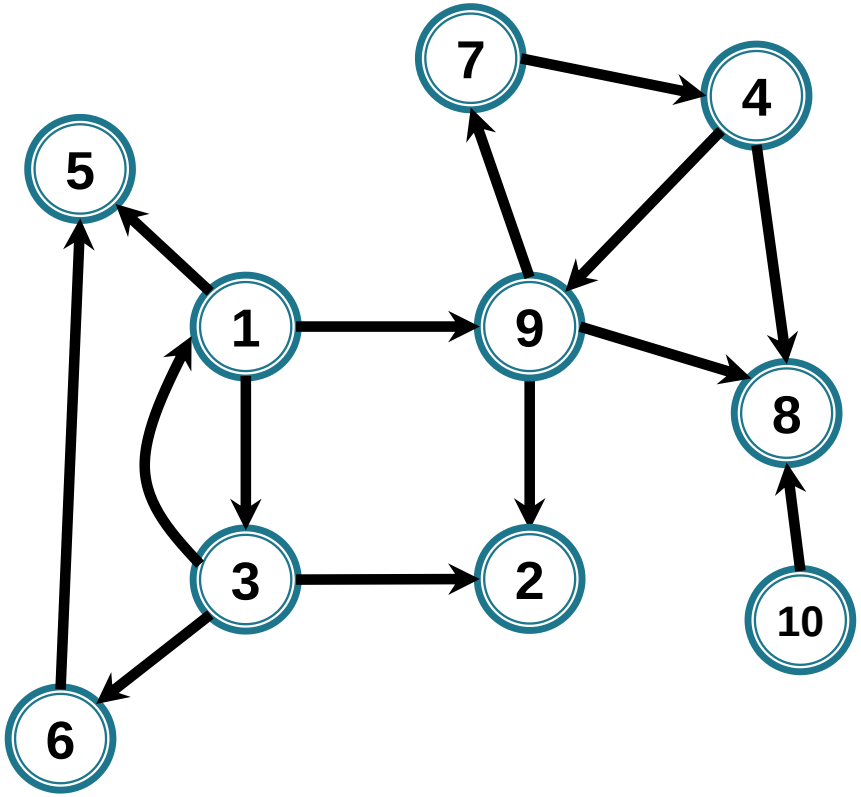
Parcurgerea în lățime – graf orientat

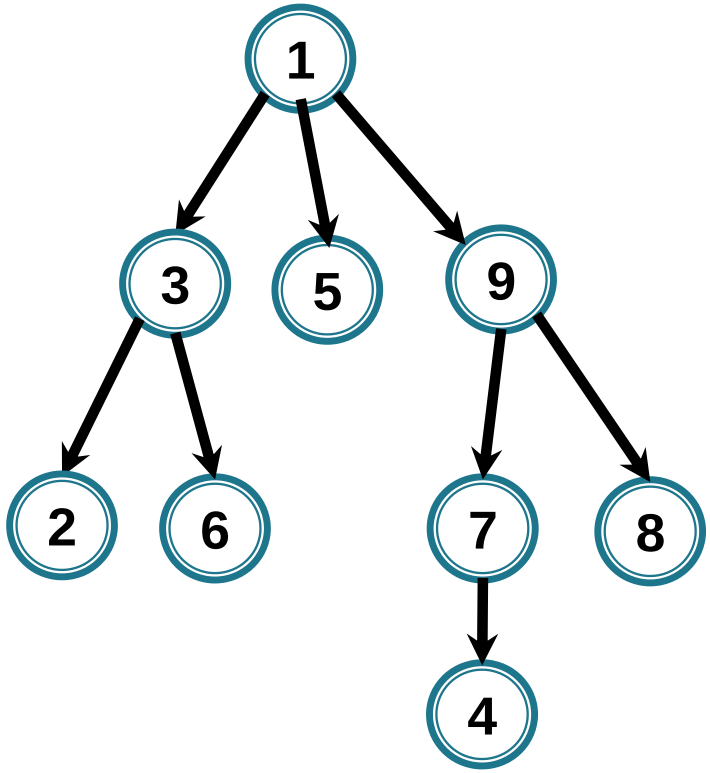
- Exemplu – caz orientat:



Parcurgerea în lățime – graf orientat

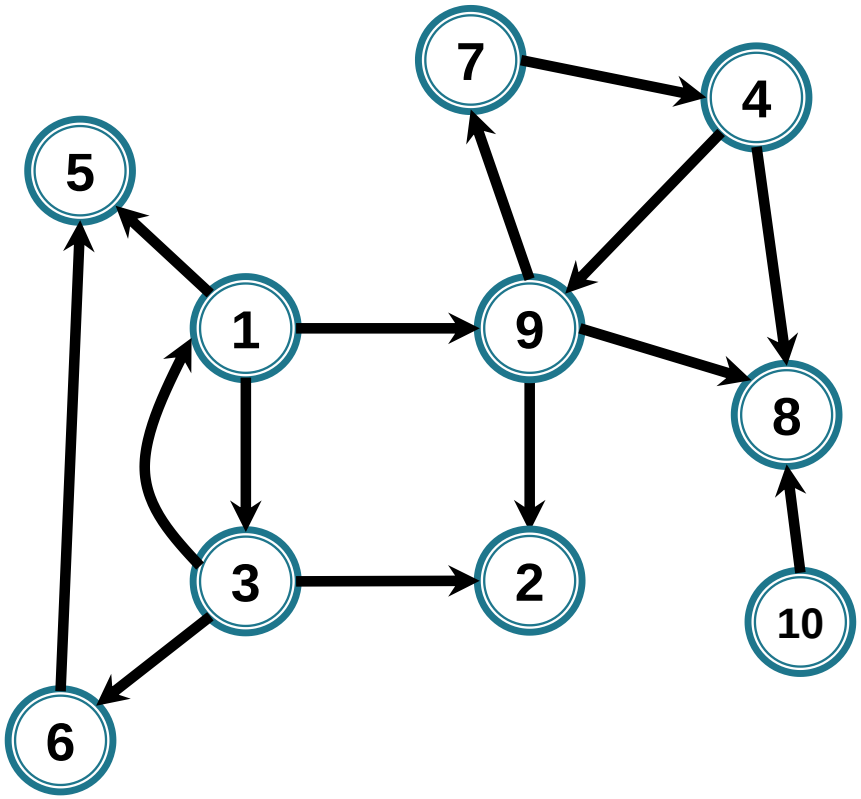
- Exemplu – caz orientat:

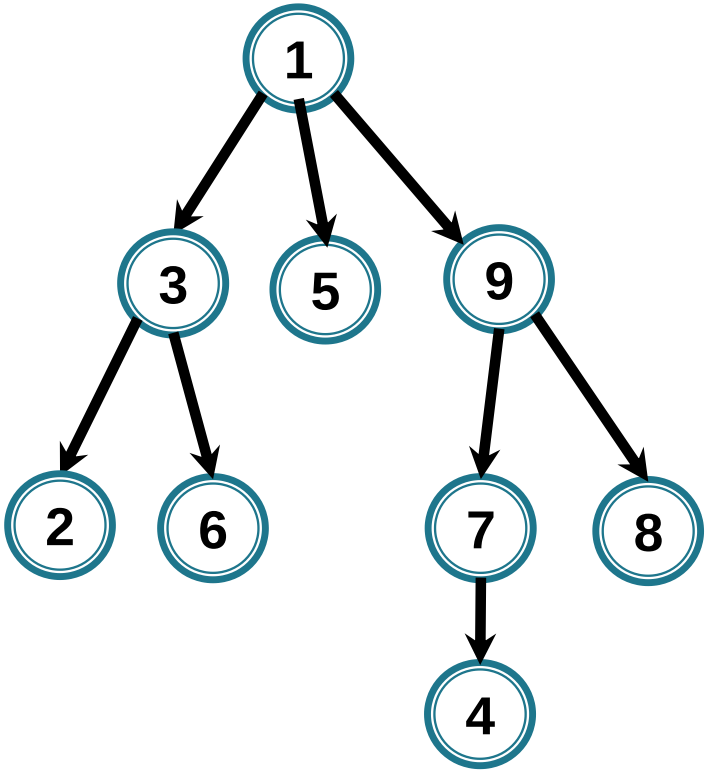




BF(1): 1, 3, 5, 9, 2, 6, 7, 8, 4

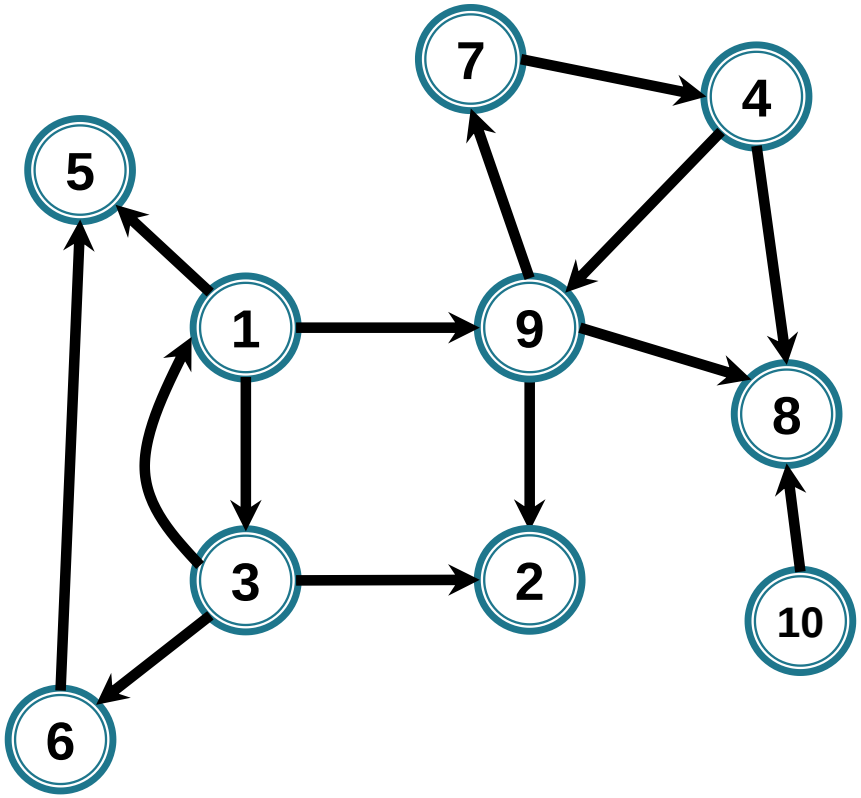
Parcurgerea în lățime – graf orientat

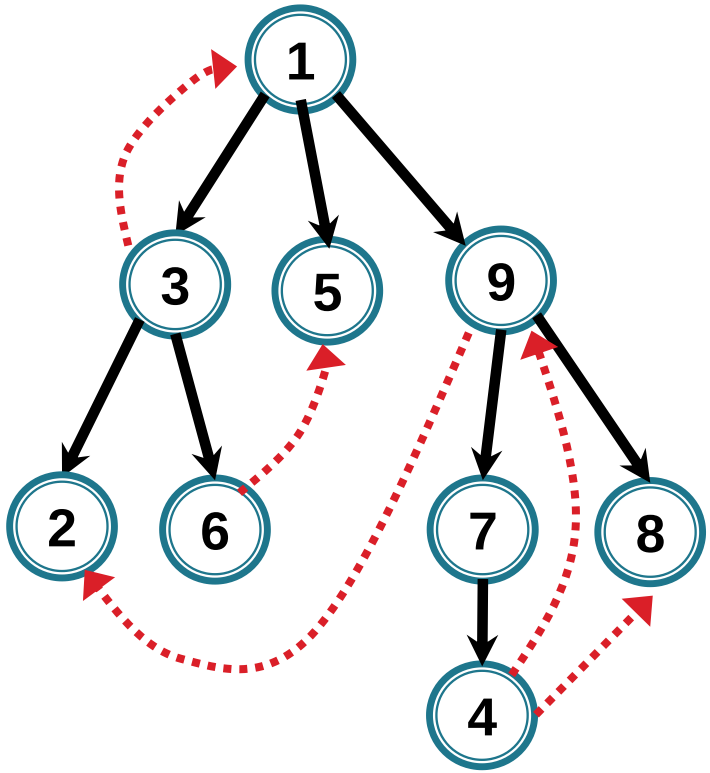




Arbore BF – tot arbore dacă ignorăm orientarea
– arcele corespunzătoare – orientate spre frunze

Parcurgerea în lățime – graf orientat





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

Parcurgerea în lăţime – Aplicaţii

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

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

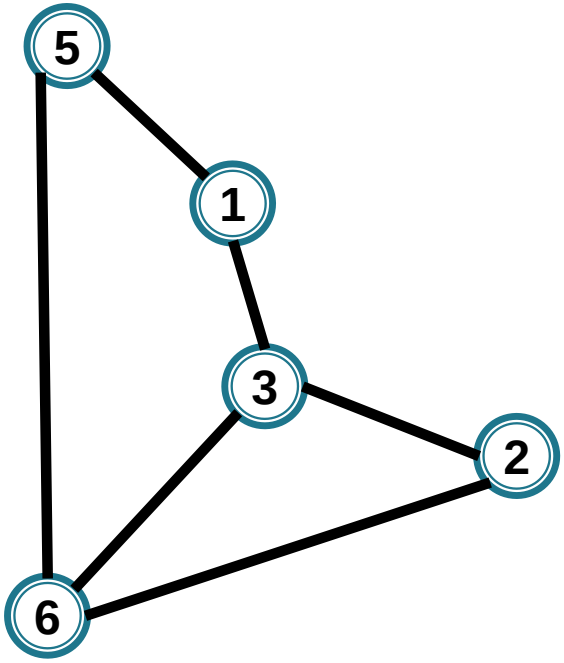
Determinarea componentelor conexe

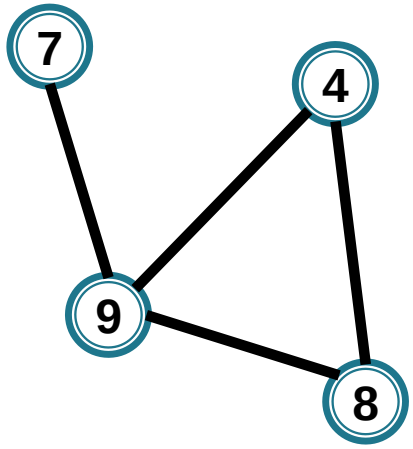
G = (V,E) graf neorientat

- 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

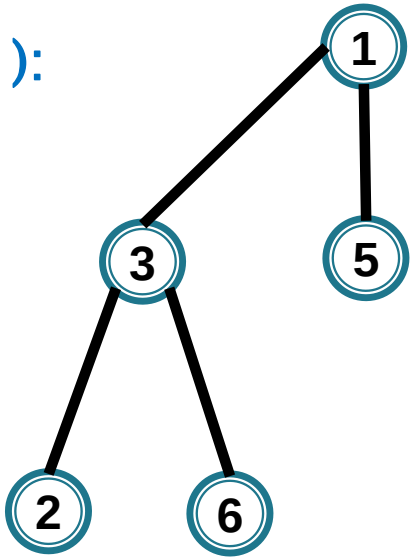
```
nr_componente = 0
pentru fiecare x ∈ V executa
    daca viz[x] == 0 atunci
        BFS(x)
        nr_componente += 1
```

- Pădure parțială / arbore parțial: muchiile {tata[x], x}, tata[x] ≠ 0

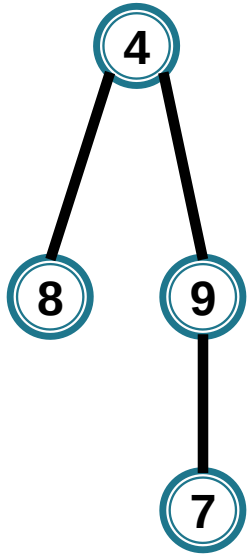




BF(1):



BF(4):

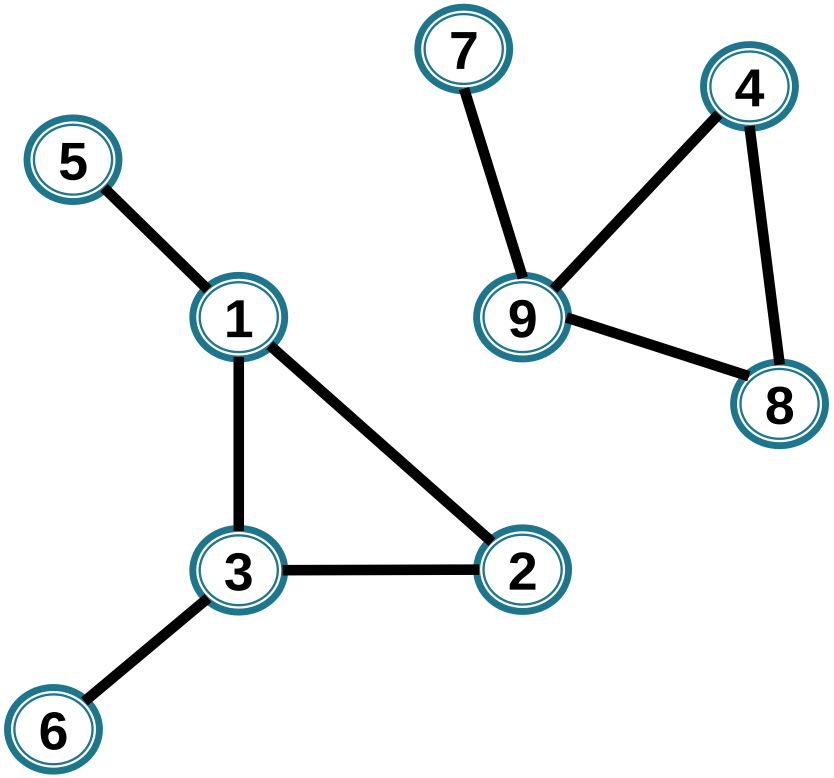


Aplicație – arbore parțial

- 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.

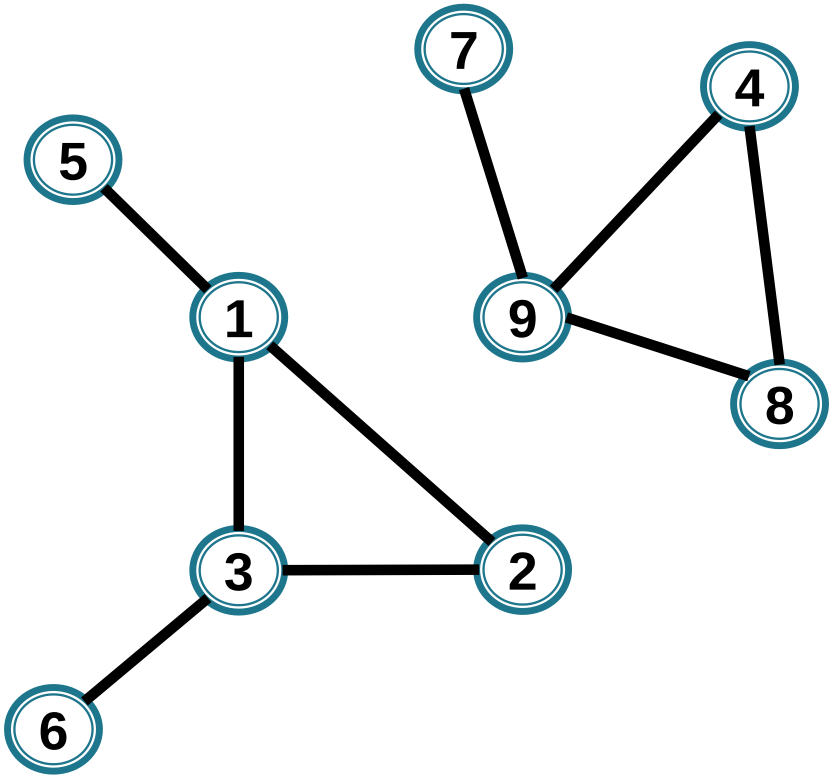
relații de prietenie/comunicare



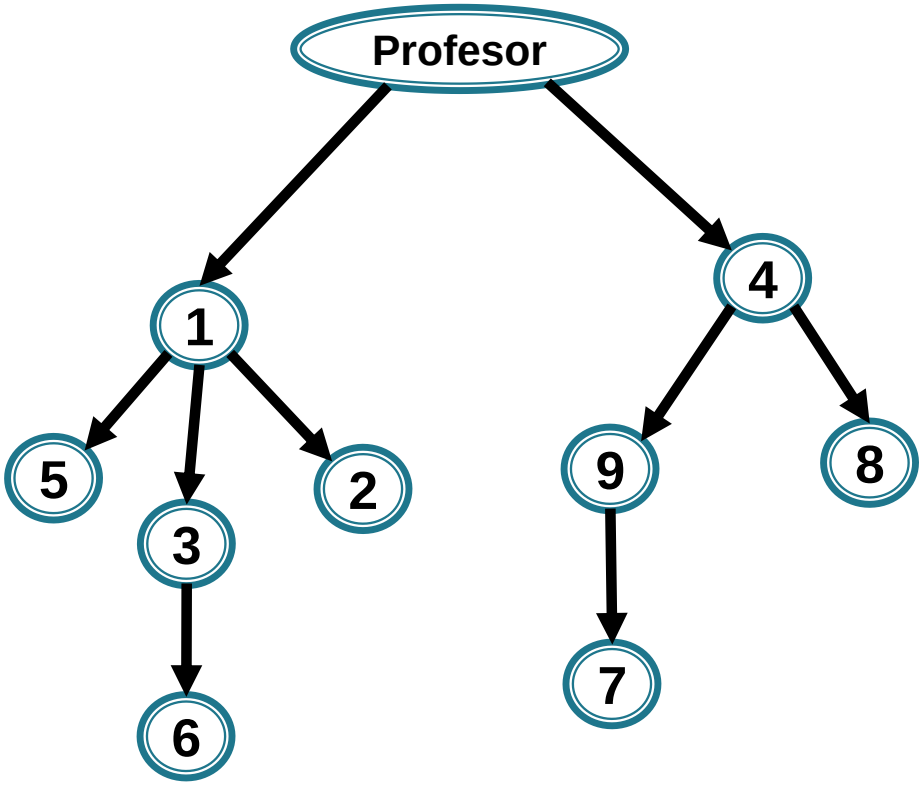
traseu de transmitere a unui mesaj

Profesor

relații de prietenie/comunicare



traseu de transmitere a unui mesaj



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

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

- **Ieșirea din labirint cu număr minim de pași**

Se dă un labirint sub forma unei matrice cu elemente 0 și 1, 1 semnificând perete (obstacol) iar 0 celula liberă. Prin labirint ne putem deplasa doar din celula curentă într-una din celulele vecine (N,S,E,V) care sunt libere. Dacă ajungem într-o celulă liberă de la periferia matricei (prima sau ultima linie/coloană) atunci am găsit o ieșire din labirint. Date două coordonate x și y, să se decidă dacă există un drum din celula (x,y) prin care se poate ieși din labirint. În caz afirmativ să se afișeze un drum minim către ieșire.

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

start = (3,3)

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

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

- **Ieșirea din labirint cu număr minim de pași**
  - Matrice labirint => **graf cu vârfuri** corespunzătoare celulelor și **muchii** corespunzătoare celulelor vecine libere (cu valoarea 0)
  - **Soluție**: Parcurgere BF din celula în care ne aflăm până găsim o ieșire (!!!prima ieșire vizitată în BF este și cea mai apropiată)
    - viz, d, tata – devin matrice

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

- **Drumuri minime ale calului pe tabla de șah**

Corectitudine

Parcurgerea în lățime

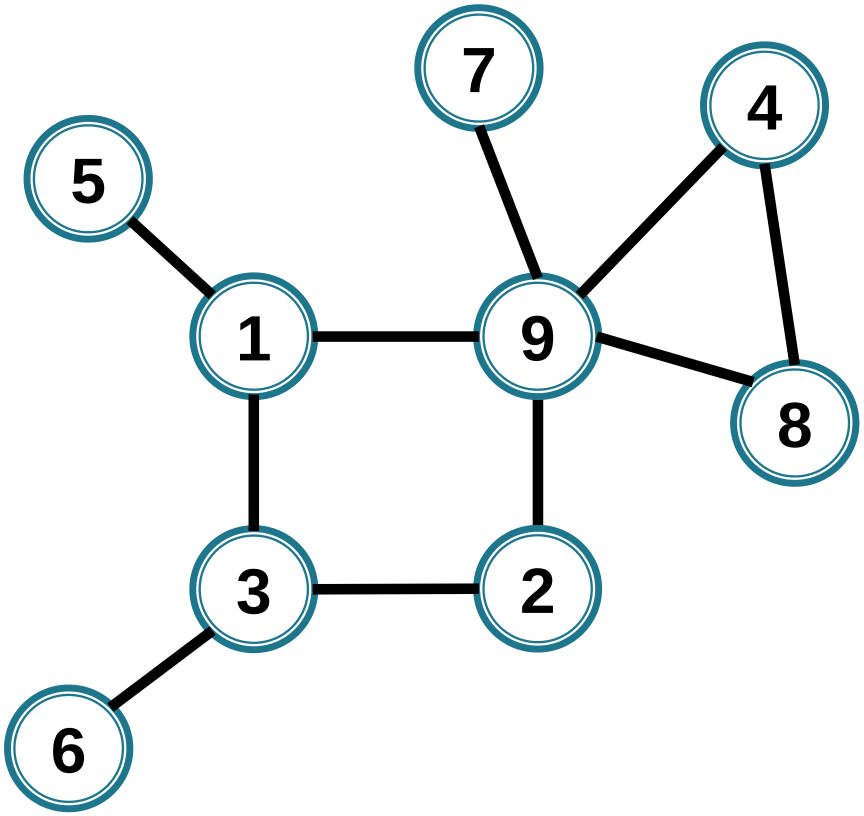
- **Propoziție – Corectitudinea BF**

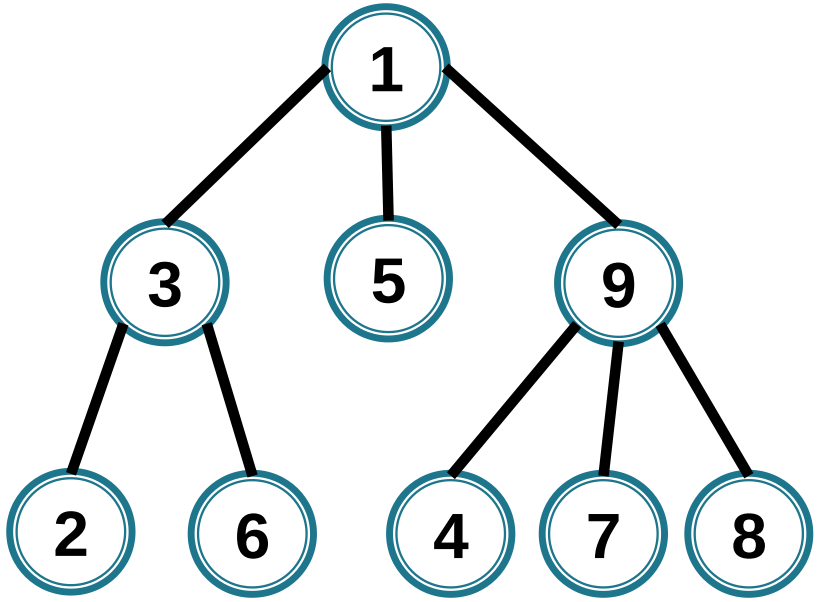
La finalul algoritmului BFS(s), pentru orice vârf v avem

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

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:

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

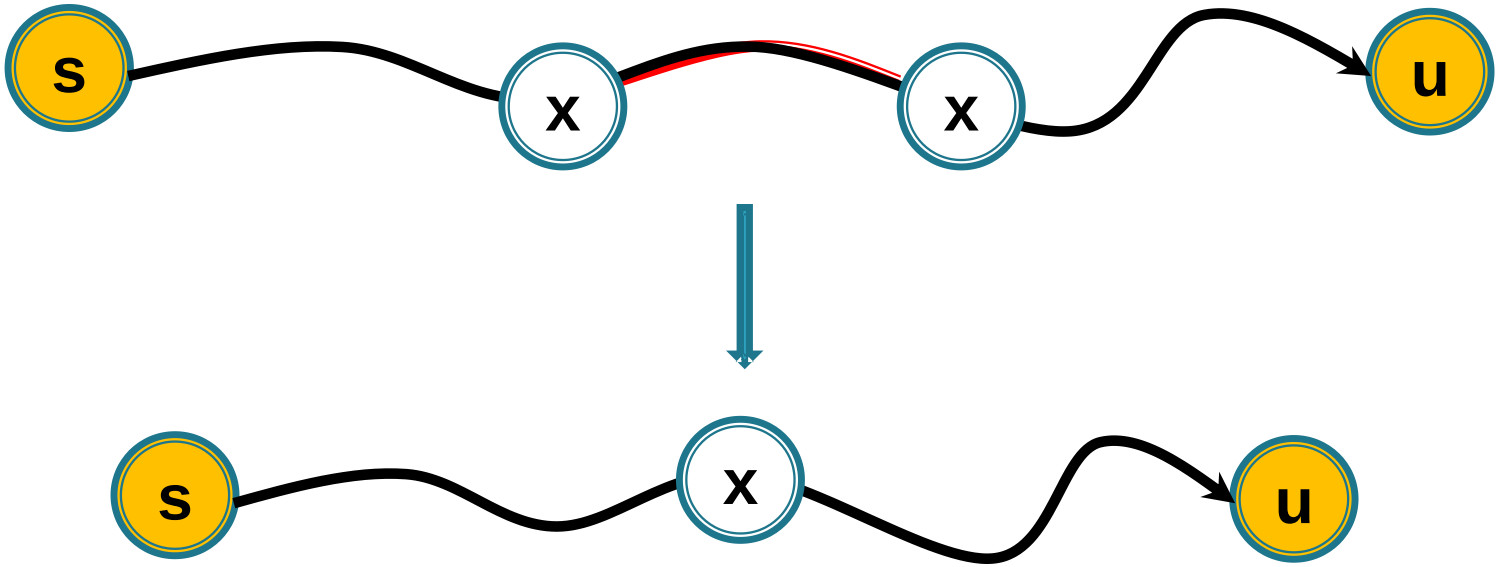




$$\delta_G(1,v) = \delta_T(1,v)$$

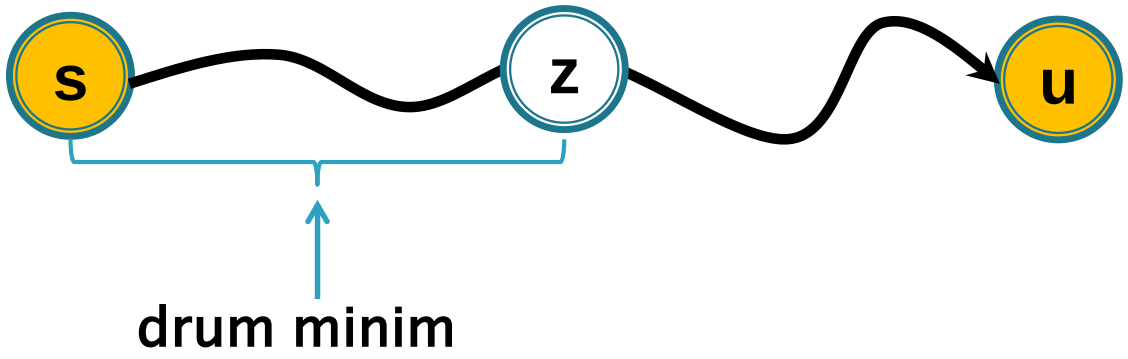
Corectitudine

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

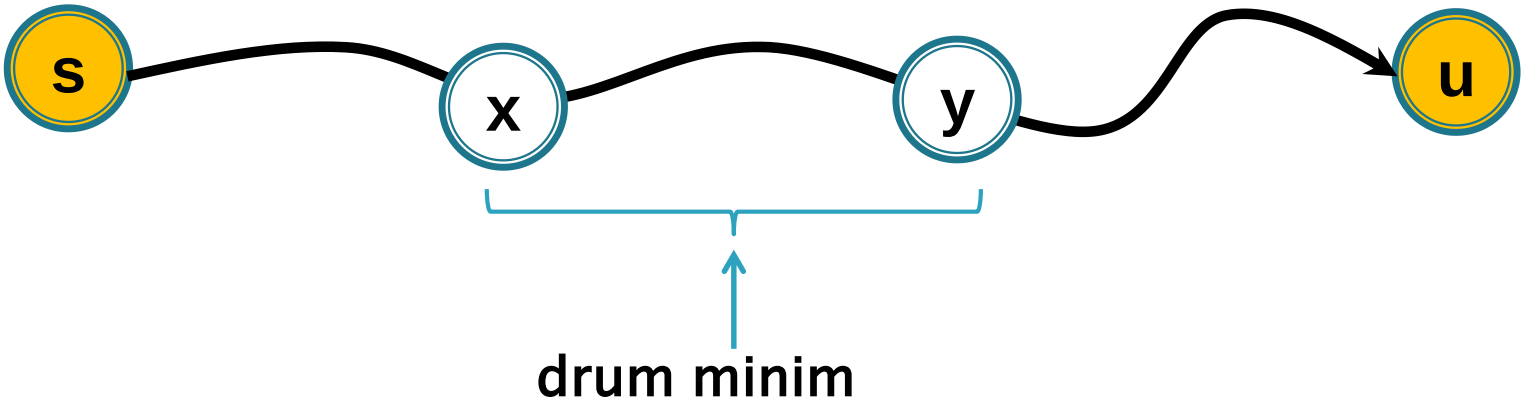


Corectitudine

- **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

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

(nodurile în coadă – crescător după d)

De ce?

Corectitudine

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

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

Evidențiem operațiile – Inducție

Corectitudine

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

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

Evidențiem operațiile – Inducție

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

Corectitudine

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

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

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 $d[v_r] \leq \mathbf{d[v_1] + 1 \leq d[v_2] + 1}$

Corectitudine

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

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

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 $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$
  - Inserăm pe rând vecinii w_i ai lui v_1 nevizitați:

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

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

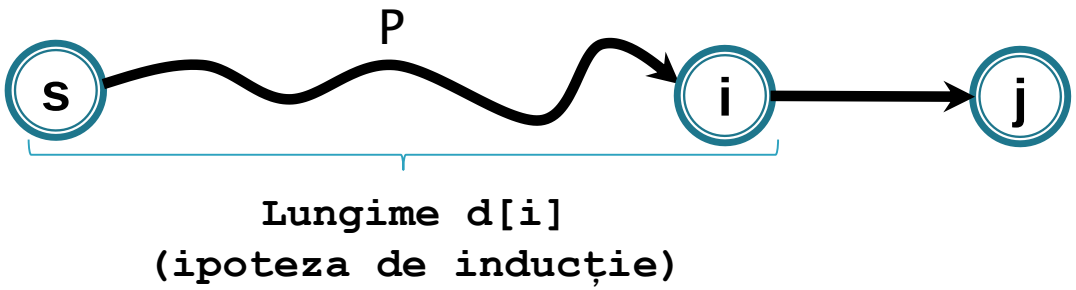
Corectitudine

- **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.

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




```
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

    $$d[x] \leq d[y]$$

  - $d[v] \geq \delta(s,v)$ (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:

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

Corectitudine

Demonstrație (**schița**)

Fie y vârful **cel mai apropiat** de s cu d[y] calculat **incorect**:

$d[y] > \delta(s, y)$ (consec. Lema 2)

Fie x predecesorul lui pe un drum minim P de la s la y

Corectitudine

Demonstrație (**schița**)

Fie y vârful **cel mai apropiat** de s cu d[y] calculat **incorect**:

$d[y] > \delta(s, y)$ (consec. Lema 2)

Fie x predecesorul lui pe un drum minim P de la s la y

Avem $l(P) = \delta(s,x) + 1 = \delta(s,y) < d[y]$

(subdrumul lui P de la s la x este tot drum minim iar $d[x] = \delta(s,x)$, deoarece x este mai apropiat de s decât y) $\Rightarrow$

Corectitudine

Demonstrație (**schița**)

Din modul de funcționare al BF, cand x este scos din coada, **y este în una din situațiile**:

- deja vizitat și extras din coadă (**negru**) $\Rightarrow$ $d[y] \leq d[x]$ (Lema 1)
- deja vizitat și încă în coadă (**gri**) $\Rightarrow$ $d[y] \leq d[x] + 1$ (Lema 1)
- este **nevizitat** încă (**alb**) $\Rightarrow$ va fi vizitat din x, deci $d[y]=d[x]+1$.
- Rezultă $d[y] \leq d[x]+1 = l(P) = \delta(s,y) < d[y]$, contradicție

Detalii – Cormen