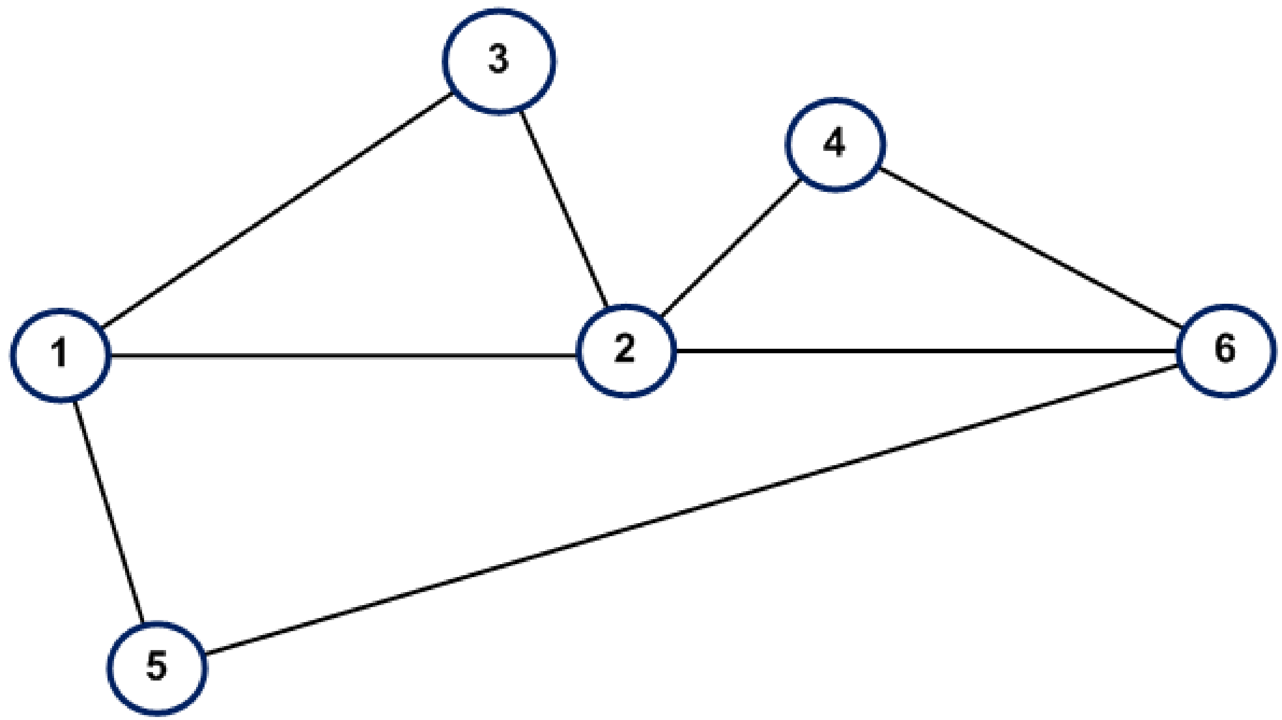
Modalităţi de reprezentare

Reprezentarea grafurilor

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



Matrice de adiacenţă

- Construcţia din lista de muchii

Intrare: n,m și muchiile

```
6 8
1 2
1 3
2 3
2 4
4 6
2 6
1 5
5 6
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
void citire(int &n, int**&a, int orientat=0, const char*
nume_fisier="graf.in"){
       int i,x,y,j,m;
       ifstream f(nume_fisier);
       f>>n>>m;
       a=new int*[n];
       for(i=0;i<n;i++)
           a[i]=new int[n];
       for(i=0;i<n;i++)
           for(j=0;j<n;j++)
                 a[i][j]=0;
       while(f>>x>>y){
              x--; y--;
              a[x][y]=1;
              if(not orientat)
                 a[y][x]=1;
       }
       f.close();
}
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
def citire(orientat=0,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:
        linie=f.readline()
        n,m=(int(z) for z in linie.split())
        #aux=linie.split(); x=int(aux[0]); y=int(aux[1])
        a=[[0 for i in range(n)] for j in range(n)]
        #a=[[0]*n for i in range(n)]
        for linie in f: #linie=f.readline(); while linie!='':
            x,y=(int(z) for z in linie.split())
            #x,y=map(int,linie.split())
            x-=1; y-=1
            a[x][y]=1
            if not orientat:
                a[y][x]=1
                    #linie=f.readline()

    return n,a
```

Liste de adiacenţă

- **Dinamic**
  - folosind tipul vector / list

Liste de adiacență - Construcție din lista de muchii

```
void citire(int &n,vector<int> *&la, int orientat=0,
            const char *nume_fisier="graf.in"){

       int i,j,x,y,m;
       ifstream f(nume_fisier);
       f>>n>>m;

       la=new vector<int>[n];

       while(f>>x>>y){ //mergea si cu for i=1,m
              x--;y--; //lucram de la 0

              la[x].push_back(y);
              if (not orientat)
                     la[y].push_back(x);
       }
       f.close();
}
```

Liste de adiacență - Construcție din lista de muchii

```
def citire(orientat=False,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:

        linie = f.readline()
        n, m=(int(z) for z in linie.split())

        la=[[] for i in range(n)]
        #la=n*[[]] #!!!NU

        for linie in f:
            x,y=(int(z) for z in linie.split())
            x-=1; y-=1

            la[x].append(y)
            if not orientat:
                la[y].append(x)


    return n,la
```

Liste de adiacenţă - Varianta 2

- **Liste implementate** – pointeri

Arbori cu rădăcină

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

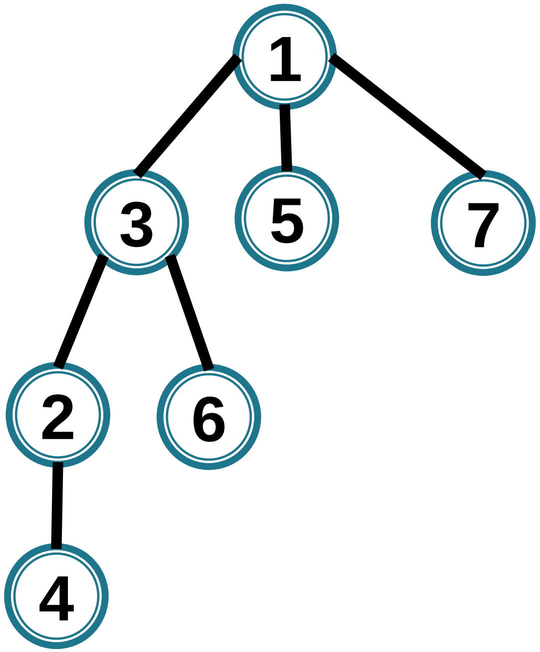
- În arborele cu rădăcină există muchii doar între niveluri consecutive

Noţiuni

- **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:** 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:** y este descendent al lui x $\Leftrightarrow$ x este ascendent a lui y
- **Frunză:** nod fără fii

Noțiuni

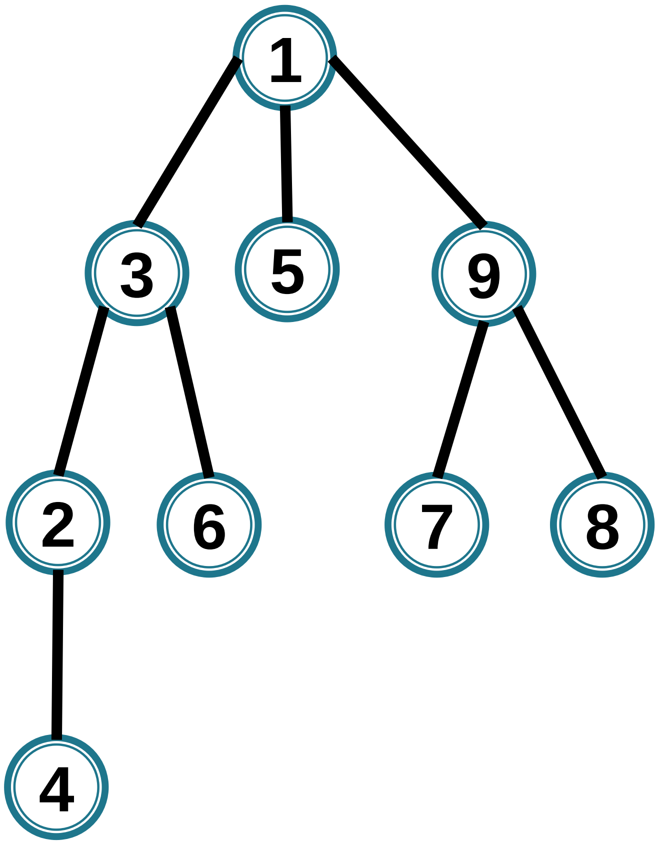
- **Fiu:** fii lui 3 sunt 2 şi 6
- **Tată:** 1 este tatăl lui 7
- **Ascendent:** ascendenţii lui 6 sunt 3 şi 1
- **Descendent:** descendenţii lui 3 sunt 2, 6 şi 4
- **Frunză:** frunzele arborelui sunt 4, 6, 5 şi 7



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

```
void lant(int x){
  while(x!=0){
    cout<<x<<" ";
    x=tata[x];
  }
}
```

```
void lantr(int x){
  if(x!=0){
      lantr(tata[x]);
      cout<<x<<" ";
  }
}
```

Parcurgerea Grafurilor

Parcurgerea grafurilor

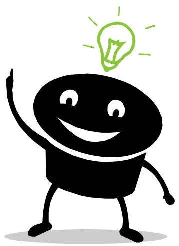
Dat un graf G și un vârf s, care sunt toate vârfurile accesibile din s?

- 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

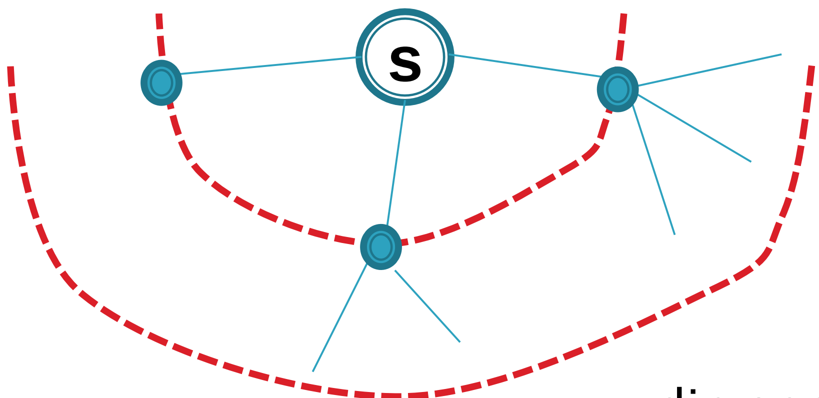
- Parcurgerea în lățime (BF = breadth first)
- Parcurgerea în adâncime (DF = depth first)

Parcurgerea în lăţime

Parcurgerea grafurilor

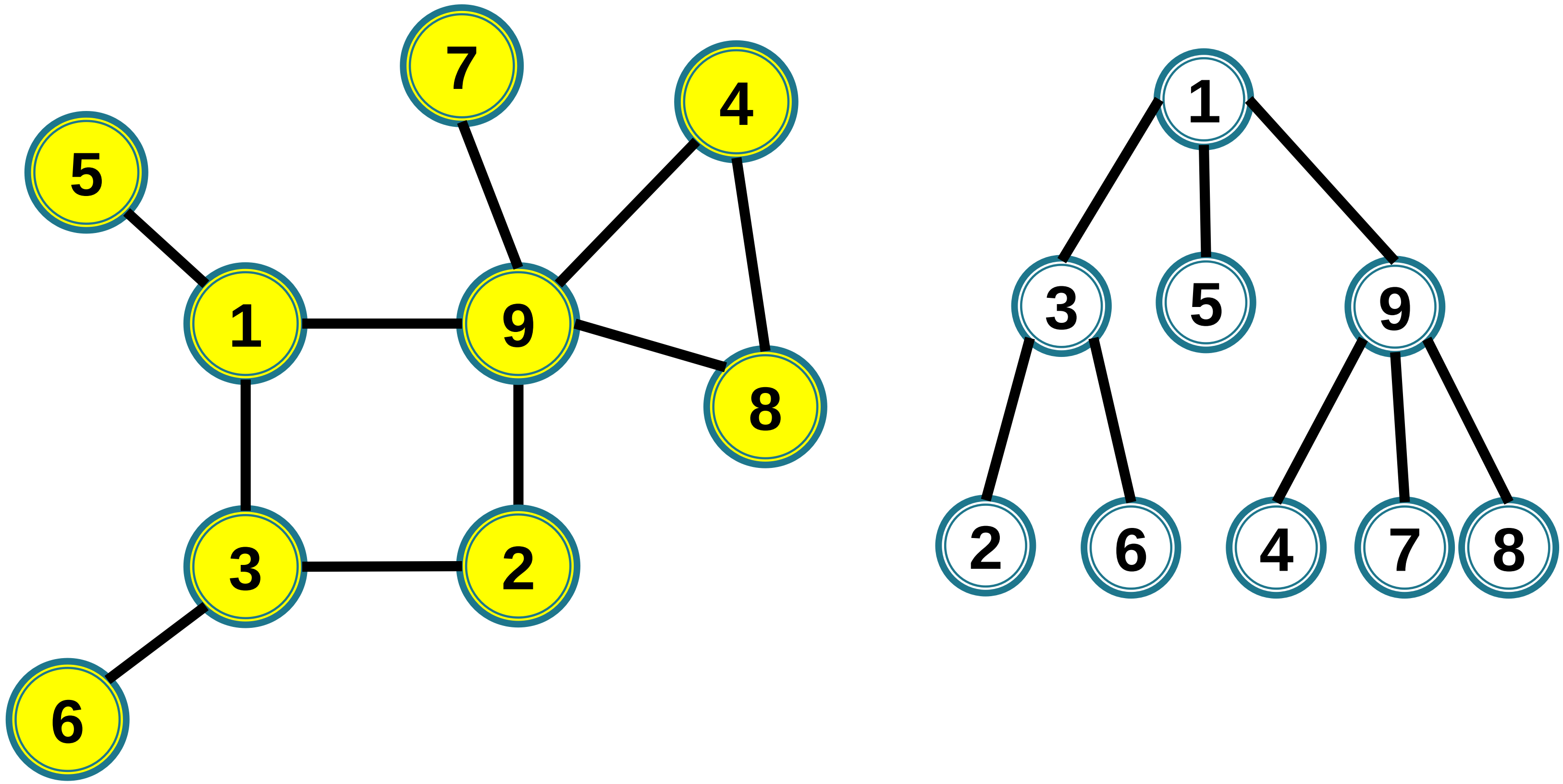
- **Parcurgerea în lățime:** se vizitează
  - vârful de start **s**
  - vecinii acestuia
  - vecinii nevizitați ai acestora

etc



din aproape în aproape

Exemplu
pentru graf neorientat



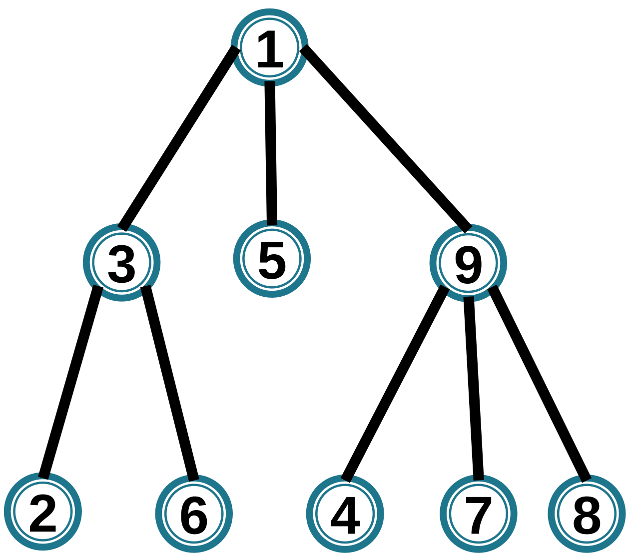
1 3 5 9 2 6 4 7 8

Parcurgerea în lățime – graf neorientat

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore BF**)

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

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

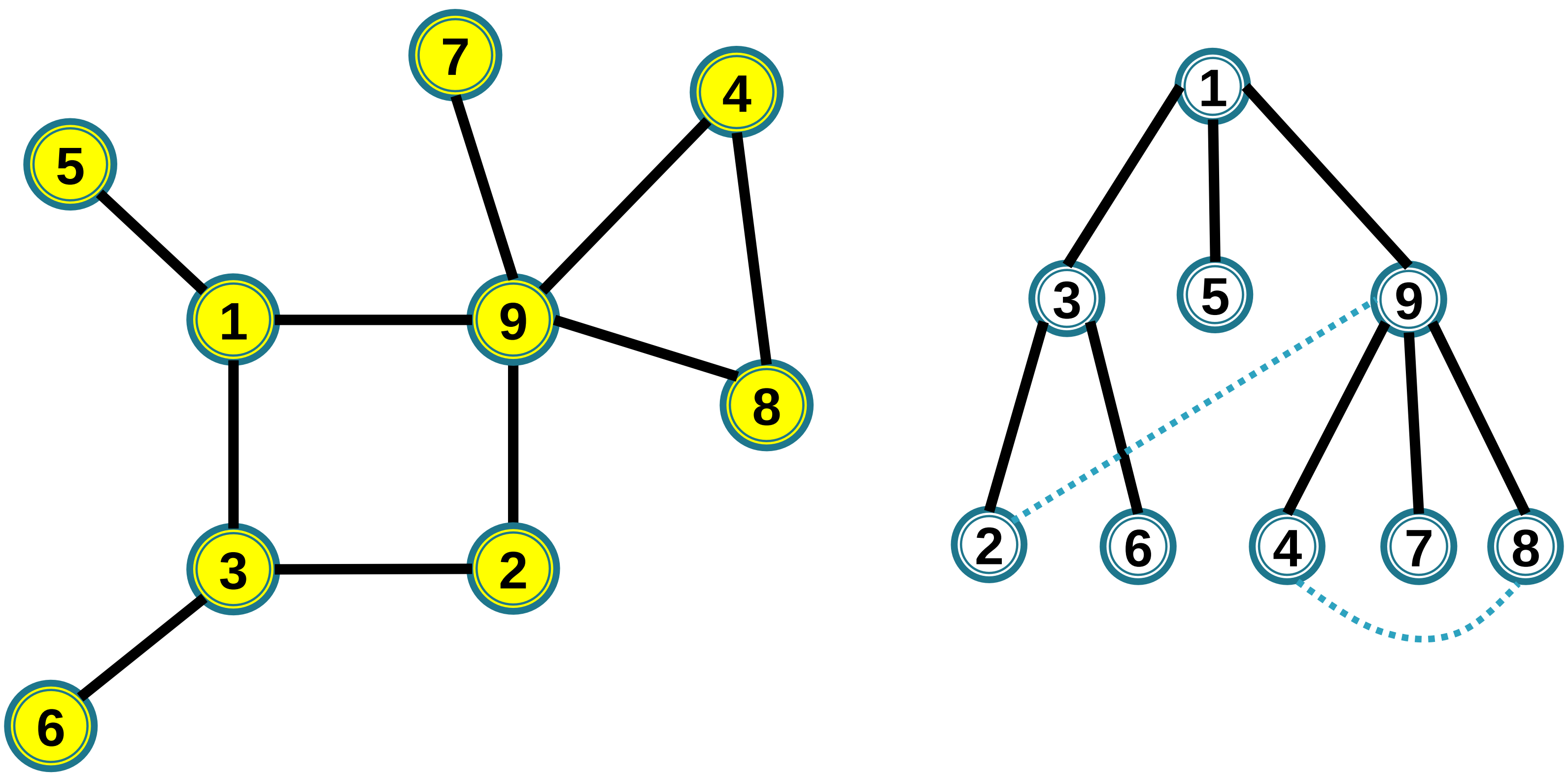


tata = [0, 3, 1, 9, 1, 3, 9, 9, 1]

Parcurgerea în lățime – graf neorientat

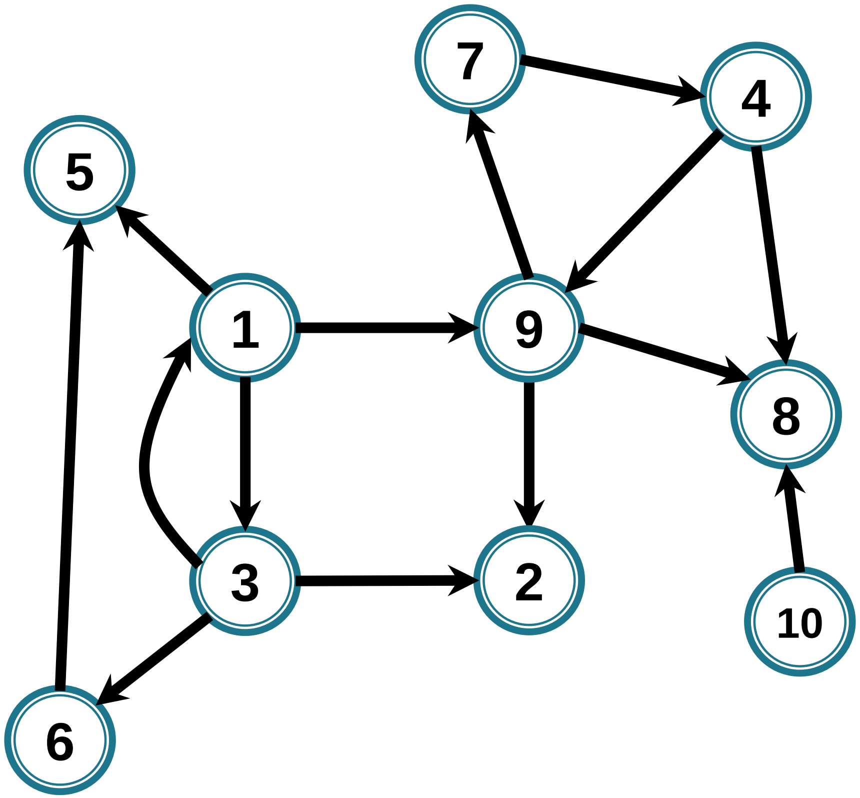
- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore BF**)

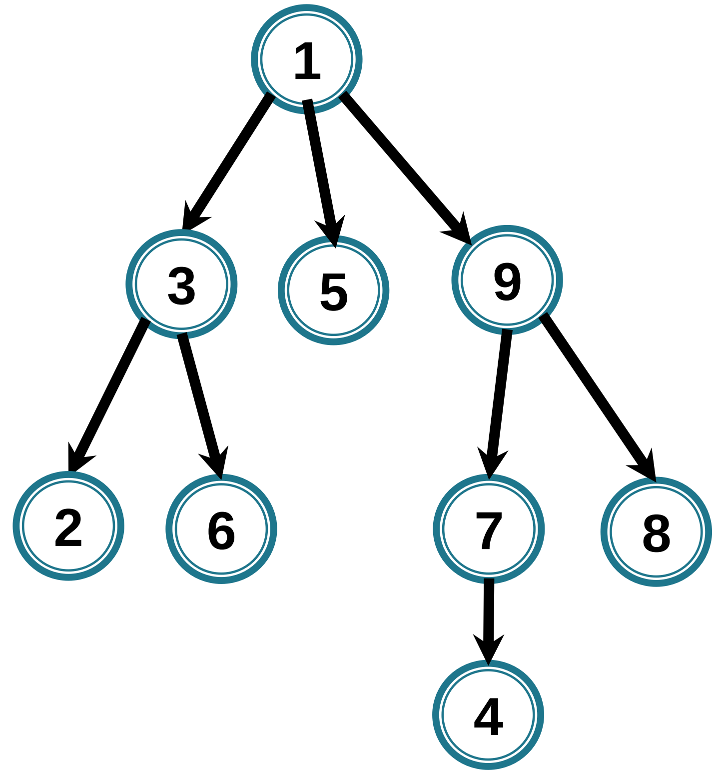
- **Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore)**



Parcurgerea în lățime

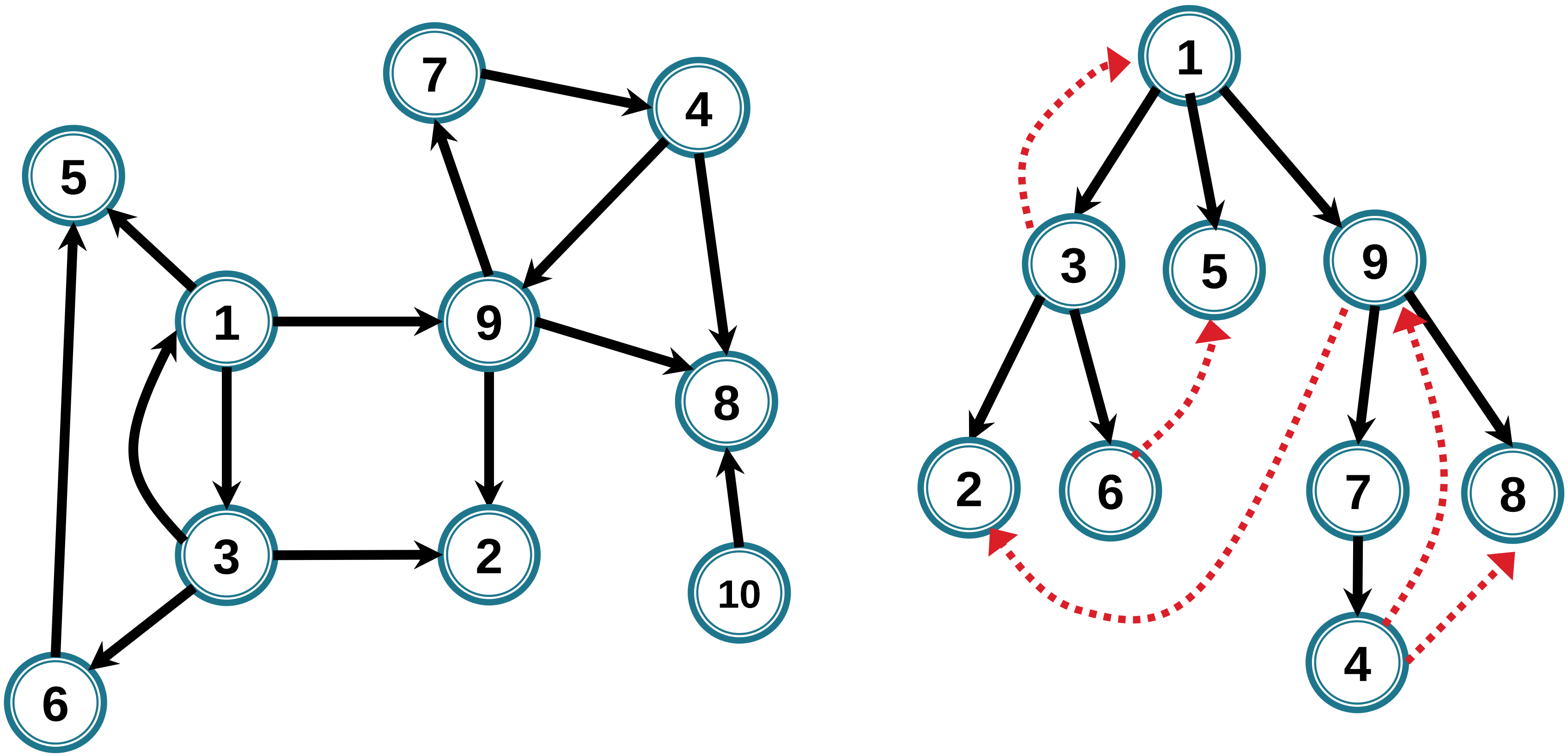
- Exemplu – caz orientat:





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

Parcurgerea în lățime



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

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

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

Parcurgerea în lățime

- **Propoziție – Corectitudinea BF**

    d[i] este chiar distanța de la s la i

Demonstraţia – după pseudocod

```
procedure BF(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

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

Implementare

Inițializări

```
pentru i=1,n executa
    viz[i]←0
    tata[i]←0
    d[i]← ∞
```

```
for(i=0;i<n;i++){
    viz[i]=0;
    tata[i]=d[i]=-1;//sau n+1
}
```

```
procedure BF(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

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

```
queue<int> c;
c.push(s);
viz[s]=1; d[s]=0;
while(c.size()>0){
      int x=c.front();c.pop();
      parc_bf.push_back(x+1);

      for (i=0;i<la[x].size();i++){
          int y=la[x][i];
          if(viz[y]==0) {
                c.push(y);
                viz[y]=1;
                tata[y]=x;
                d[y]=d[x]+1;
          }
      }
}
```

Inițializări

```
pentru i=1,n executa
    viz[i]←0                    viz=[0]*n
    tata[i]←0                   tata=[None]*n
    d[i]← ∞                     d=[None]*n
```

```
procedure BF(s)
   coada C ← ∅;                      q=[]
   adauga(s, C)                      q.append(s)
   viz[s]← 1; d[s] ← 0               viz[s]=1; d[s]=0
   cat timp C ≠ ∅ executa            while len(q)>0:
     i ←  extrage(C);                    x=q.pop(0)
     afiseaza(i);                        parc_bf.append(x+1)

     pentru j vecin al lui i
                                         for y in la[x]:
          daca viz[j]=0                      if viz[y]==0:
              adauga(j, C)                       q.append(y)
              viz[j] ← 1                         viz[y]=1
              tata[j] ← i                        tata[y]=x
              d[j] ← d[i]+1                      d[y]=d[x]+1
```

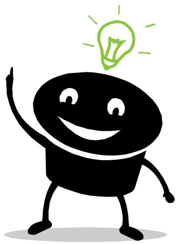
Complexitate

- Matrice de adiacență $O(|V|^2)$

- Liste de adiacență $O(|V|+|E|)$

Aplicații

- Test graf conex



```
bf(1)
testăm dacă toate vârfurile au fost vizitate
```

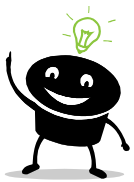
Aplicații

- Determinarea numărului de componente conexe

```
nrcomp = 0;
for(i=1;i<=n;i++)
   if(viz[i]==0){
      nrcomp++;
      bf(i);
   }
```

Aplicații

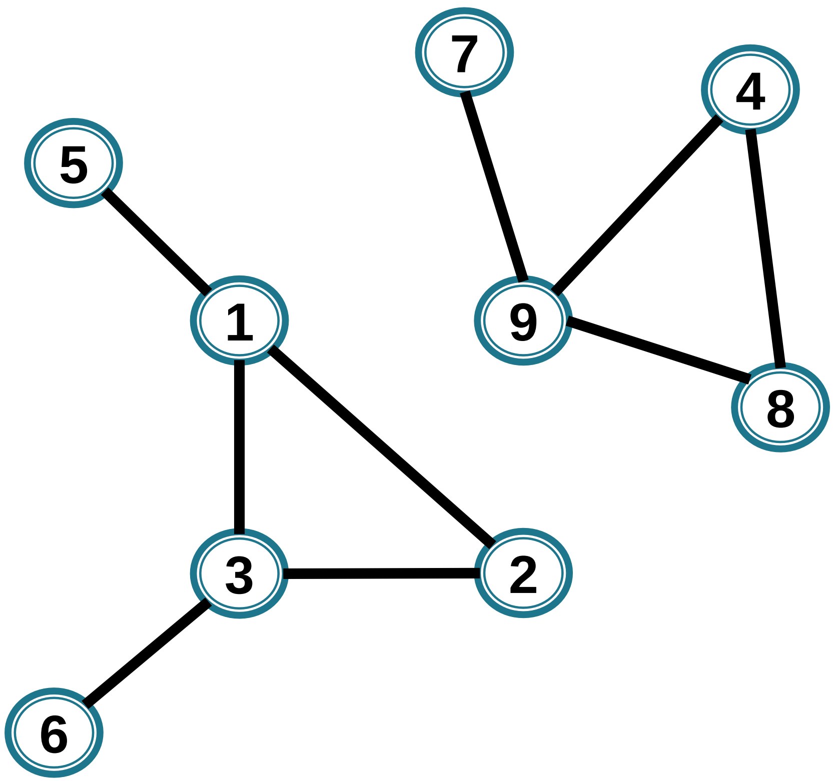
- Determinarea unui arbore parțial al unui graf conex

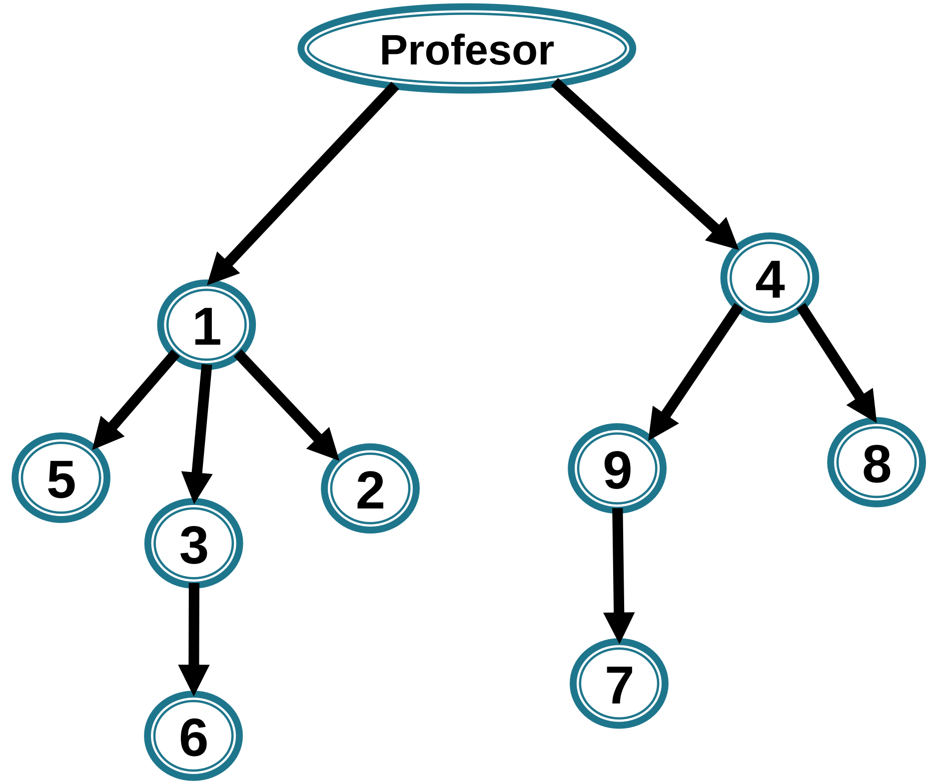


Muchiile `{tata[x], x}`, $x \neq s$

Aplicații

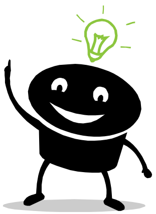
- Determinarea unui arbore parțial al unui graf conex
- **Transmiterea unui mesaj în rețea**: Între participanții la un curs s-au legat relații de prietenie și comunică și în afara cursului. Profesorul vrea să transmită un mesaj participanților și știe ce relații de prietenie s-au stabilit între ei. 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

Aplicații

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



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

```
bf(u);
if(viz[v] == 1)
     lant(v);
else
   cout<<"nu exista drum";
```

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

Corectitudine

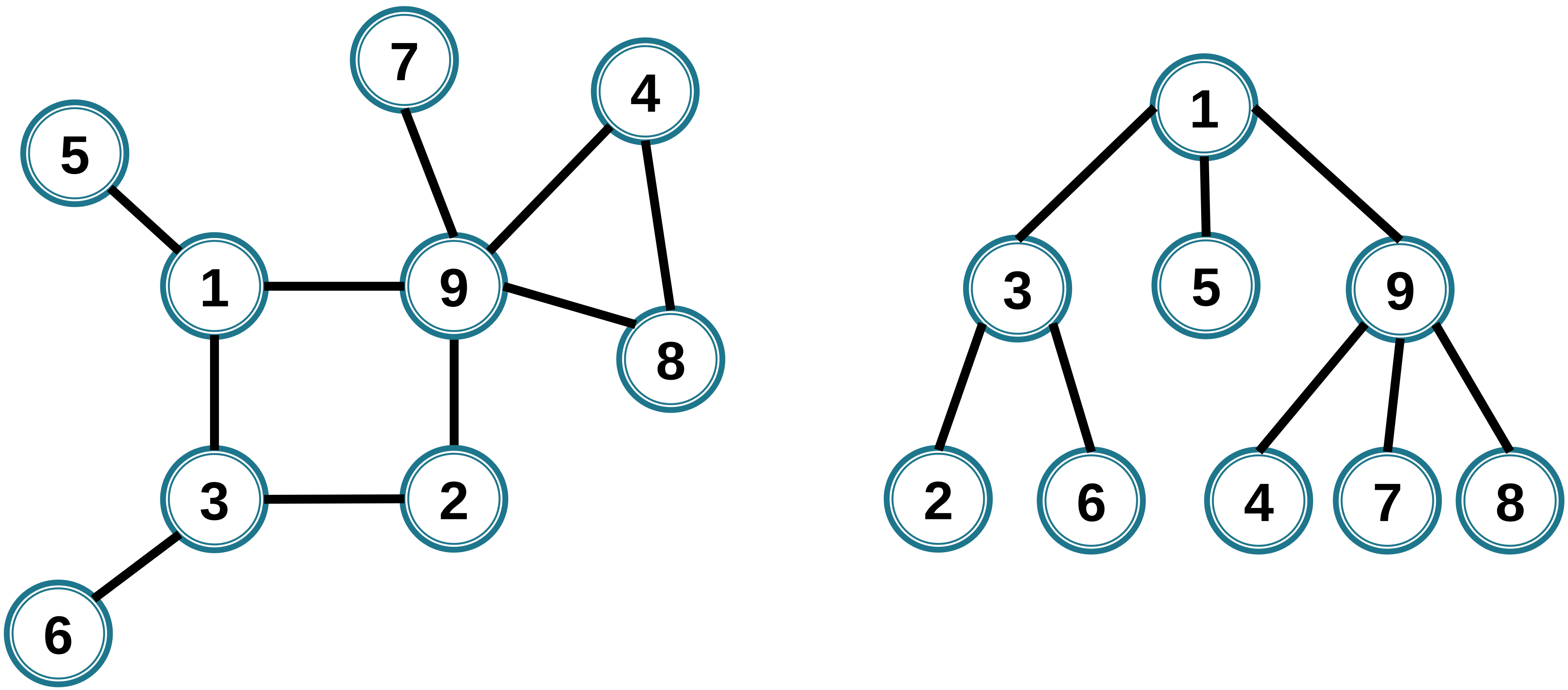
Parcurgerea în lățime

- **Propoziție – Corectitudinea BF**

  d[i] este chiar distanța de la s la i

$\Rightarrow$ Un arborele BF (notat T) al unui graf G care **conservă distanțele din graf de la s la celelalte vârfuri** – este un arbore de distanțe față de s:

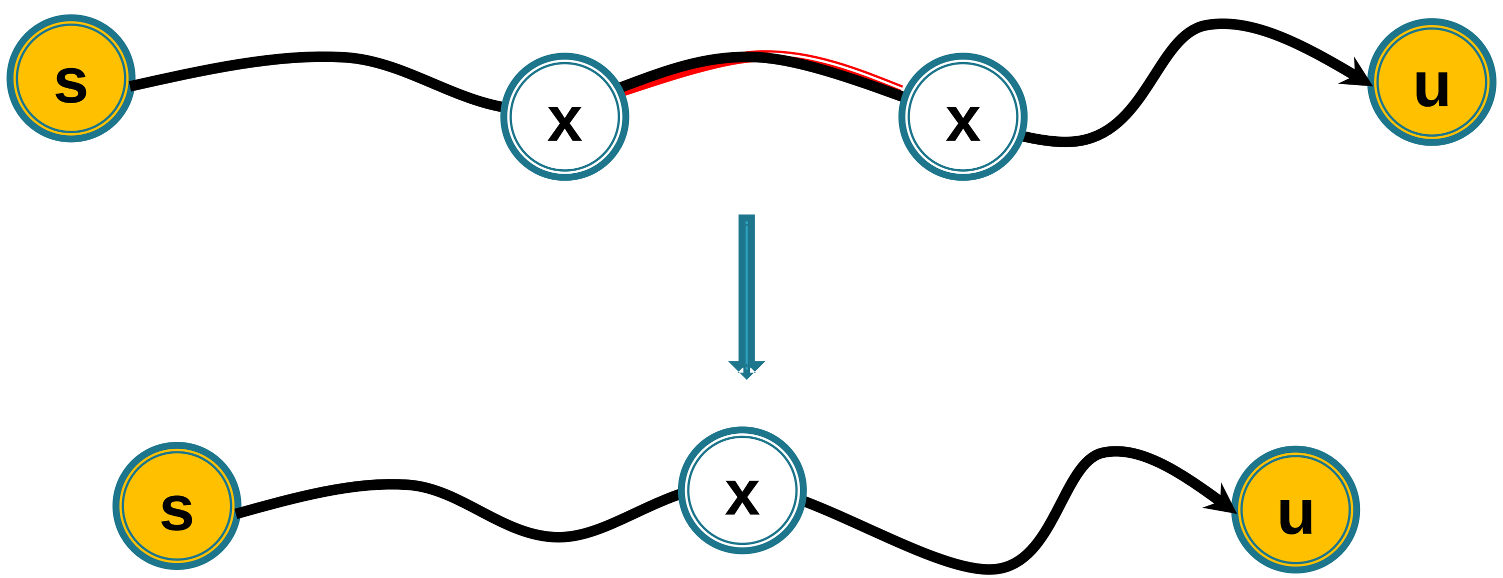
$$d_T(s, v) = d_G(s, v), \text{ pentru orice vârf } v$$



$$d_G(1,v) = d_T(1,v)$$

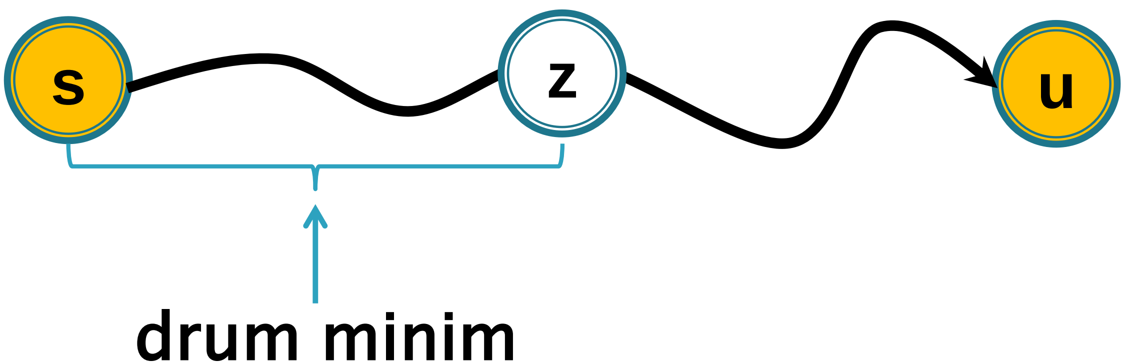
Corectitudine

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

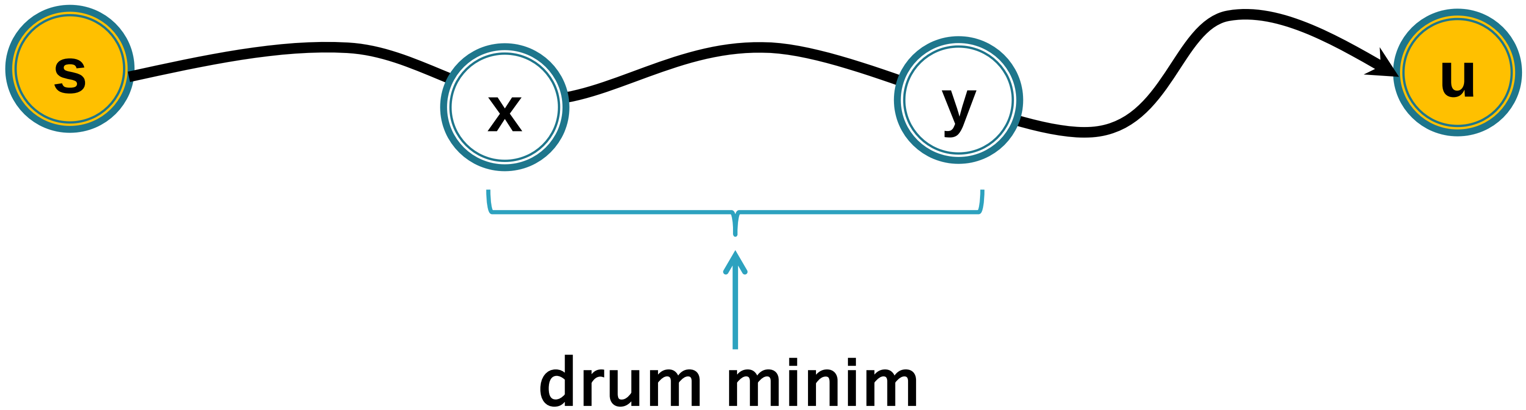


Corectitudine

- **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.



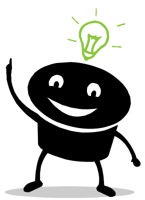
Corectitudine



Corectitudine

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

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



Evidențiem operațiile - Inducție

Corectitudine

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

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

- **Lema 2**. Dacă $d[v] = k$, atunci există în G un drum de la s la v de lungime k

- **Consecințe**.
  - Dacă x a fost extras din C inaintea lui y, avem

$$d[x] \le d[y]$$

  - $d[v] \ge d(s,v)$ (d[v] este o "*supraestimare*")
  - $d(s,v) = \infty \Rightarrow d[v] = \infty$

Corectitudine

- **Propoziție. Pentru orice vârf v avem**

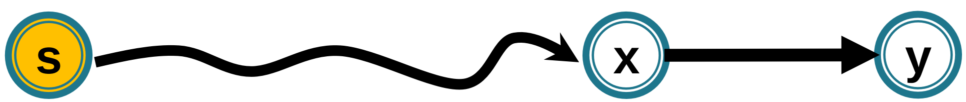
  d[v] = d(s, v) = distanța de la s la v

Demonstrație (**schița**)

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

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

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



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

(subdrumul lui P de la s la x este tot drum minim) $\Rightarrow$

x este mai aproape de s decât y, deci $d[x] = d(s,x)$ (este corect calculat) și

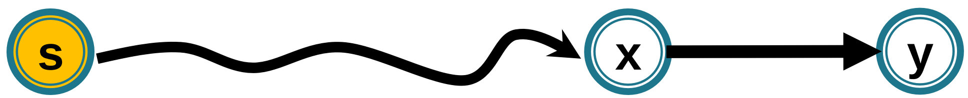
$$d[x] + 1 = l(P) < d[y]$$

Corectitudine

- **Propoziție. Pentru orice vârf v avem**

  d[v] = d(s, v) = distanța de la s la v

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 $\Rightarrow$ va fi vizitat din x, deci d[y]=d[x]+1.
- Rezultă d[y] $\leq$ d[x]+1 = l(P) = d(s,y)<d[y], contradicție

Detalii – Cormen

Parcurgerea în adâncime

Parcurgerea în adâncime

Se vizitează

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

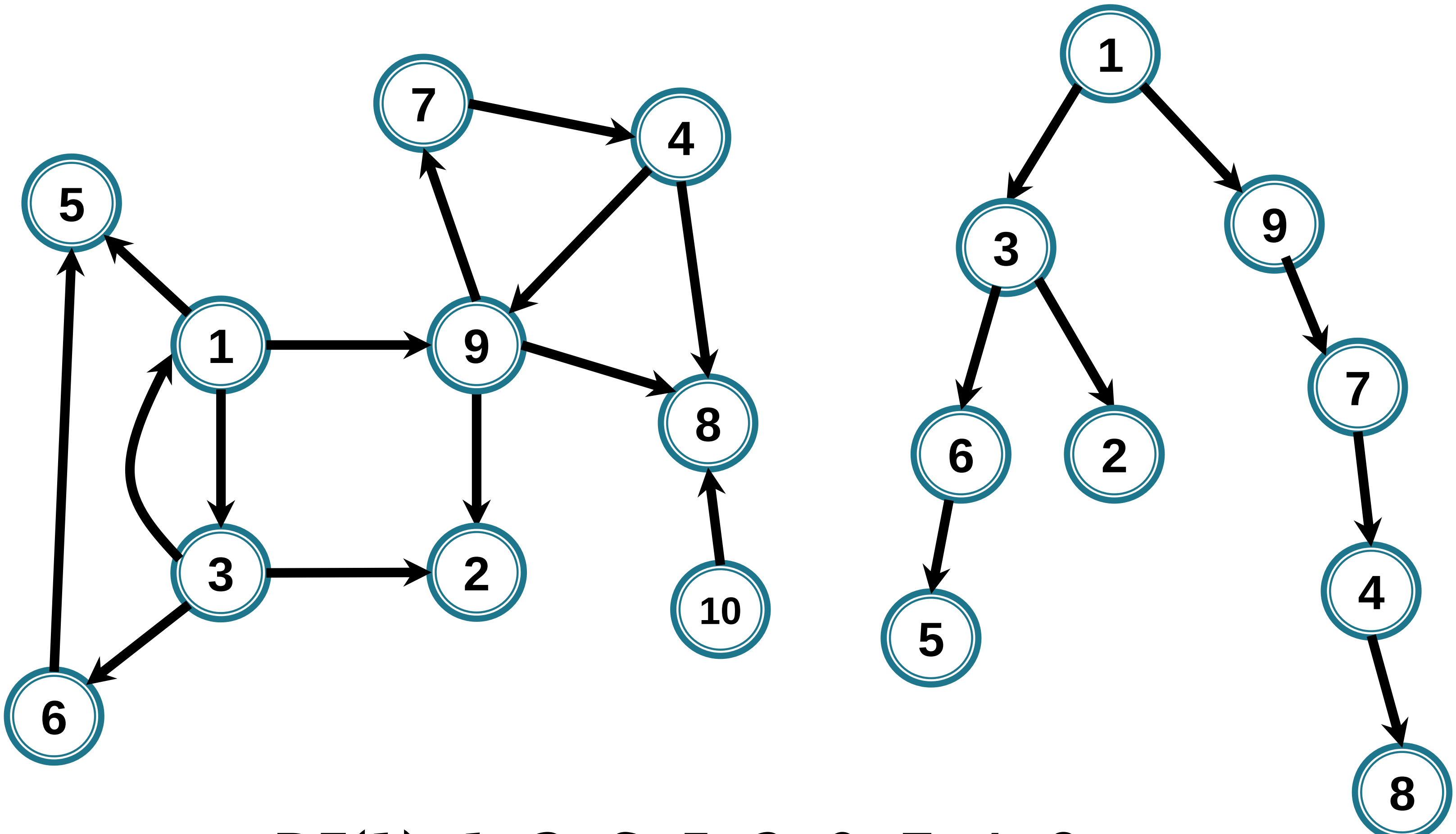
Parcurgerea în adâncime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore DF) -> se numesc** **muchii de avansare**

- Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore), mai exact **unesc un vârf cu un ascendent al lui în arborele DF -> se numesc** **muchii de întoarcere**

Parcurgerea în adâncime

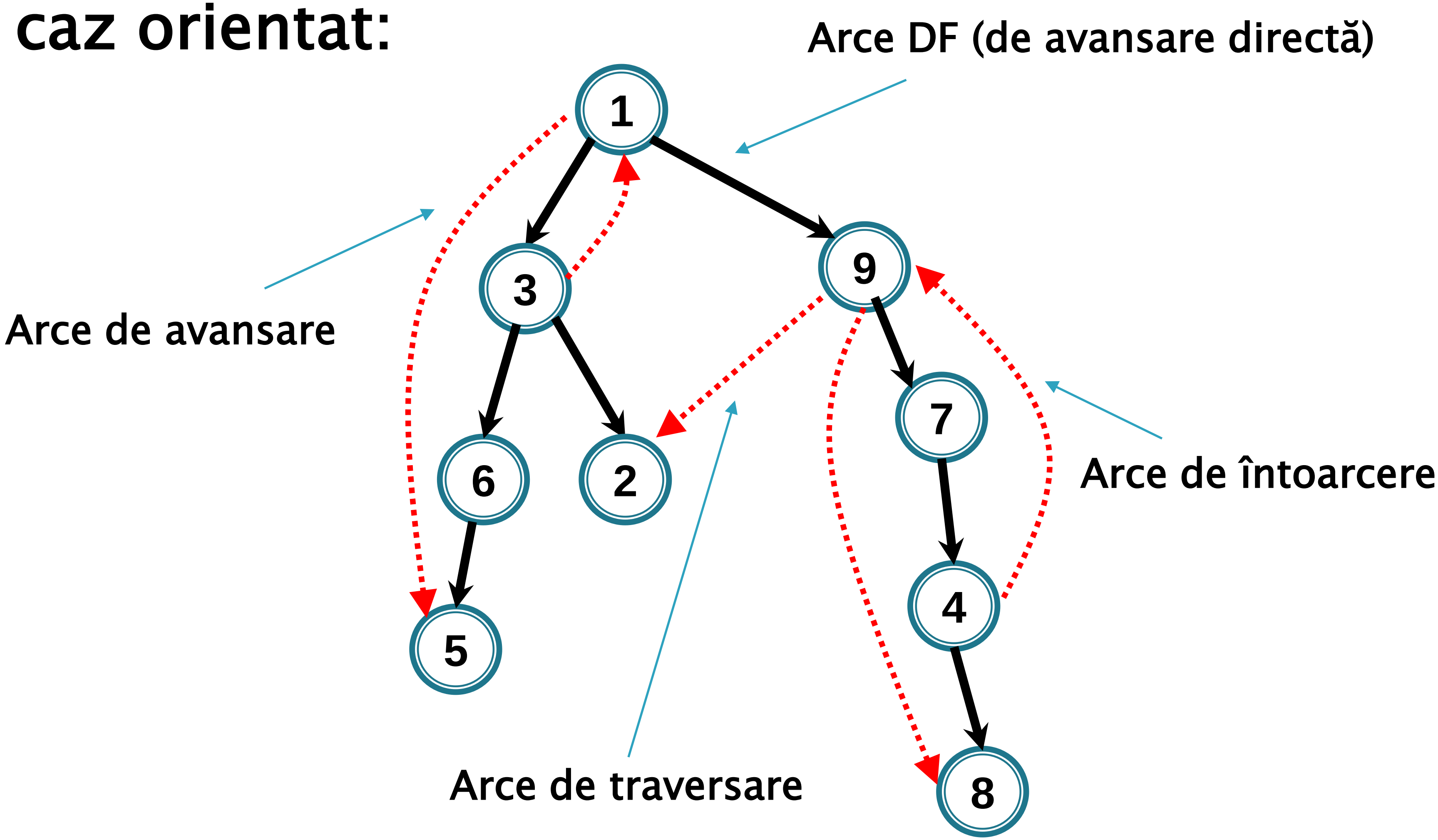
- Exemplu – caz orientat:



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

Parcurgerea în adâncime

caz orientat:



Doar arcele de întoarcere închid circuite

```
void df(int x){
    //incepe explorarea varfului x
    viz[x]=1;
    for(int i=0;i<la[x].size();i++){
        int y=la[x][i];
        if (viz[y]==0){
            tata[y]=x;
            d[y] = d[x]+1; //nivel, nu distanta
            df(y);
        }
    //s-a finalizat exlorarea varfului x
}
```

x alb

x gri

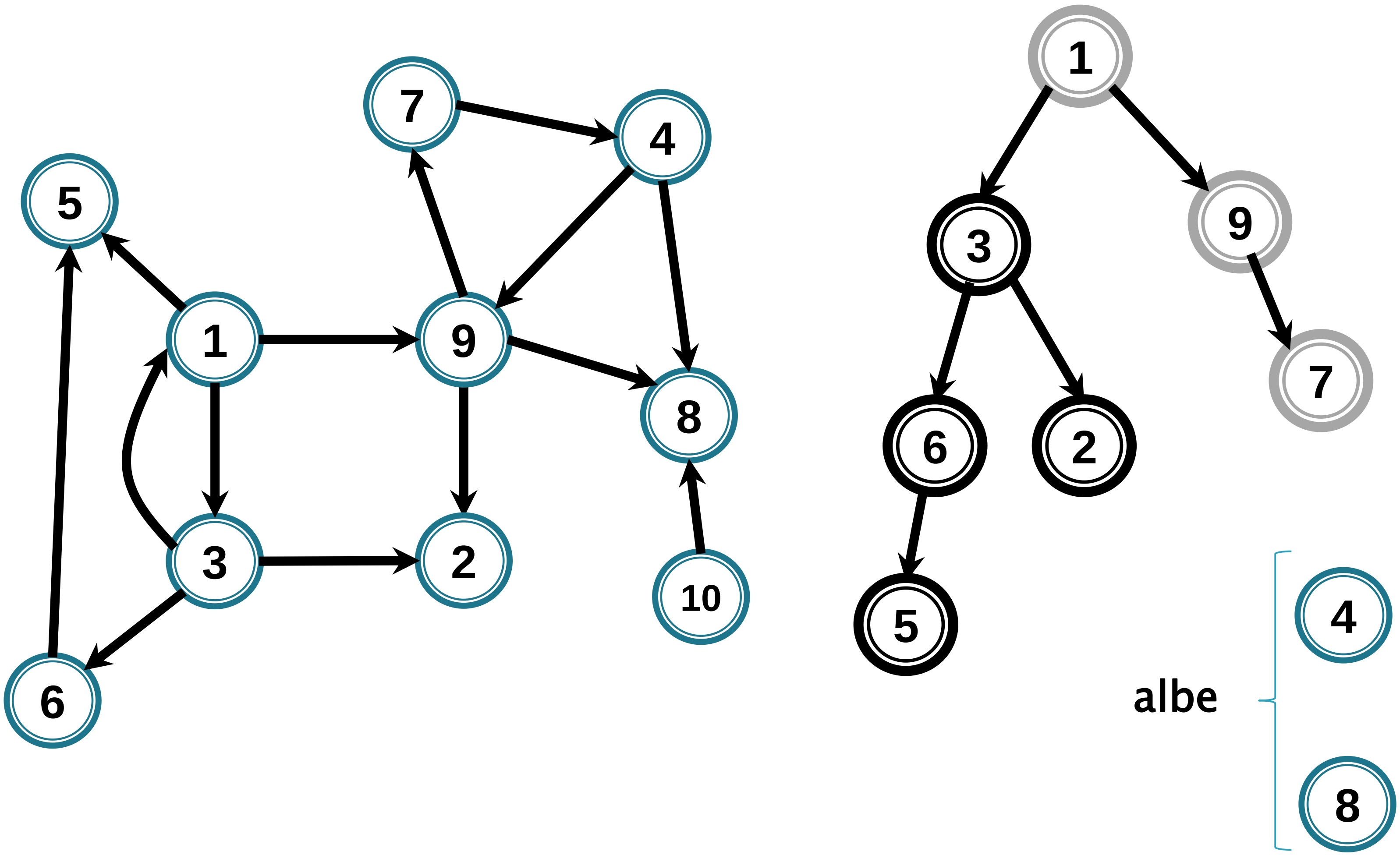
x negru

Apel:

```
    df(s)
```

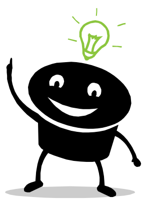
Parcurgerea în adâncime

- Culorile nodurilor după ce 7 devine vârf curent:



Alte aplicații

Dat un graf neorientat, să se verifice dacă graful conține cicluri și, în caz afirmativ, să se afișeze un ciclu al său



Un ciclu se închide în parcurge când vârful curent are un vecin deja vizitat, care nu este tatăl lui

Problema se poate rezolva folosind oricare dintre cele două parcurgeri?

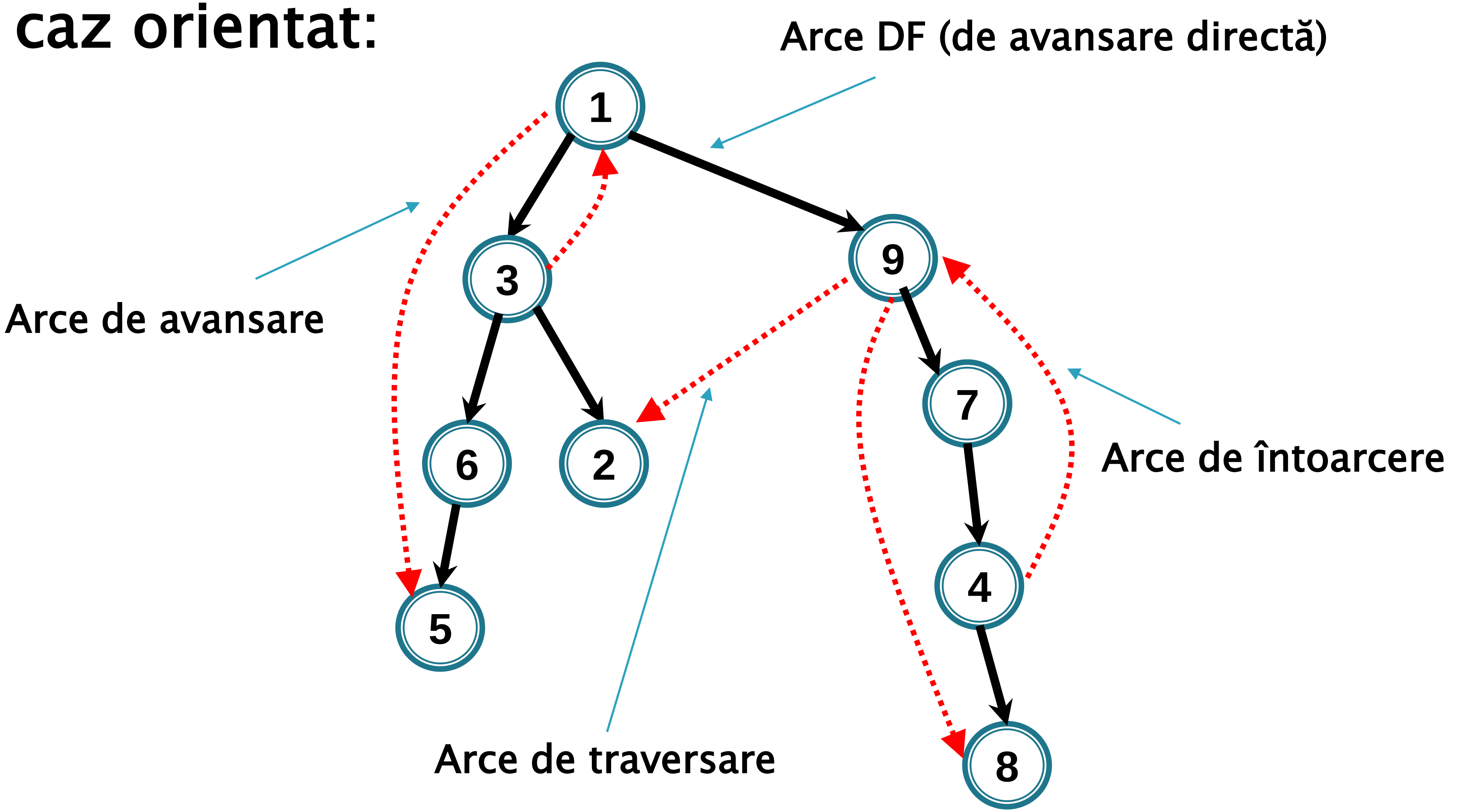
```
void dfs(int x, vector<int> *la, int *viz, int *tata, int &ok){
    viz[x]=1;
    for(int i=0;i<la[x].size() && ok==0 ;i++){
        int y=la[x][i];
         if (viz[y]==0){ //muchie de avansare
            tata[y]=x;
            dfs(y,la,viz,tata,ok);
         }
         else
            if(y!=tata[x] ){ //muchie de intoarcere
                cout<<"un ciclu elementar ";
                int v=x;
                while(v!=y){
                    cout<<v<<" ";
                    v=tata[v];
                }
                cout<<y<<" "<<x;
                ok=1;
            }
     }}
```

Alte aplicații

Dat un graf **orientat**, să se verifice dacă graful conține circuite și, în caz afirmativ, să se afișeze un circuit al său

Parcurgerea în adâncime

caz orientat:



Doar arcele de întoarcere închid circuite

Alte aplicații

Dat un graf **orientat**, să se verifice dacă graful conține circuite și, în caz afirmativ, să se afișeze un circuit al său

Un circuit este închis în arborele DF de **arce de întoarcere** = de la x la un **ascendent** al lui x = de la x la un **vârf aflat încă în explorare (gri)**

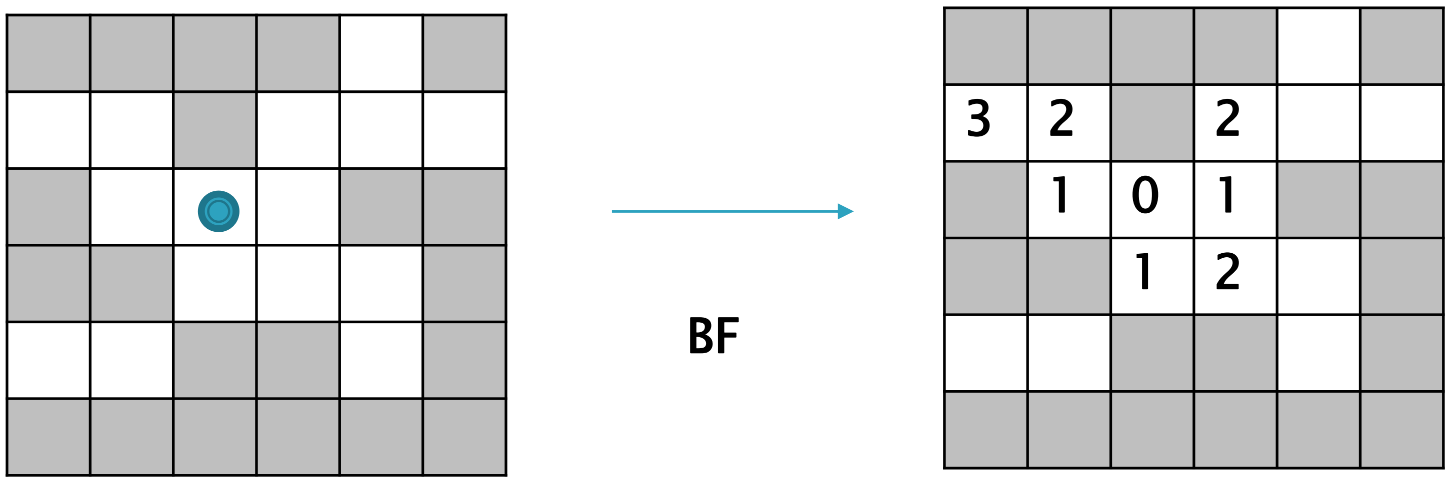
nefinalizat

```
void dfs(int x, vector<int> *la, int *viz, int *fin, int *tata, int &ok){
      viz[x]=1;
      for(int i=0;i<la[x].size() && ok==0 ;i++){
           int y=la[x][i];
           if (viz[y]==0){
                tata[y]=x;
                dfs(y,la,viz,fin,tata,ok);
           }
           else
                if(fin[y]!=1 ){ //xy de intoarcere <=>
x descendent al lui y <=> y nu a fost finalizat, este inca in
explorare
                       cout<<"un circuit elementar ";
                       lant(x,y,tata);
                       cout<<y;
                       ok=1;
                }
      }
      fin[x]=1; //finalizarea explorarii lui x
}
```

Parcurgere în lățime pe matrice

- Se dă un labirint sub forma unei matrice nxn cu elemente 0 și 1, 1 semnificând perete (obstacol) iar 0 celula liberă. Prin labirint ne putem deplasa doar din celula curenta într-una din celulele vecine care sunt libere (N,S,E,V). 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**.

Parcurgere în lățime pe matrice



matrice de distanțe

```
int deplx[]={-1,1,0,0};
int deply[]={0,0,-1,1,};
matrice de viz, tata, d....
queue<pereche> c;
viz[start.x][start.y]=1;
c.push(start);
if(iesire(start, n)) return start;
while(!c.empty()){
    pereche celula_curenta=c.front();  c.pop();
    x=celula_curenta.x; y=celula_curenta.y;
    for(int i=0;i<4;i++){
        pereche celula_vecina;
        celula_vecina.x = vx = x+deplx[i]; //vecinii celulei (x,y) vor fi (vx,vy)
        celula_vecina.y = vy = y+deply[i];
        if(lab[vx][vy]==0 && viz[vx][vy]==0){//celule libere nevizitate
            tata[vx][vy]=celula_curenta; //perechea curenta
            viz[vx][vy]=1; //marcam celula
            if(iesire(celula_vecina,n)) return celula_vecina;
            c.push(celula_vecina);
        }
    }
}
```

Alte aplicații

Să se verifice dacă un graf neorientat dat este bipartit

Graf bipartit

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

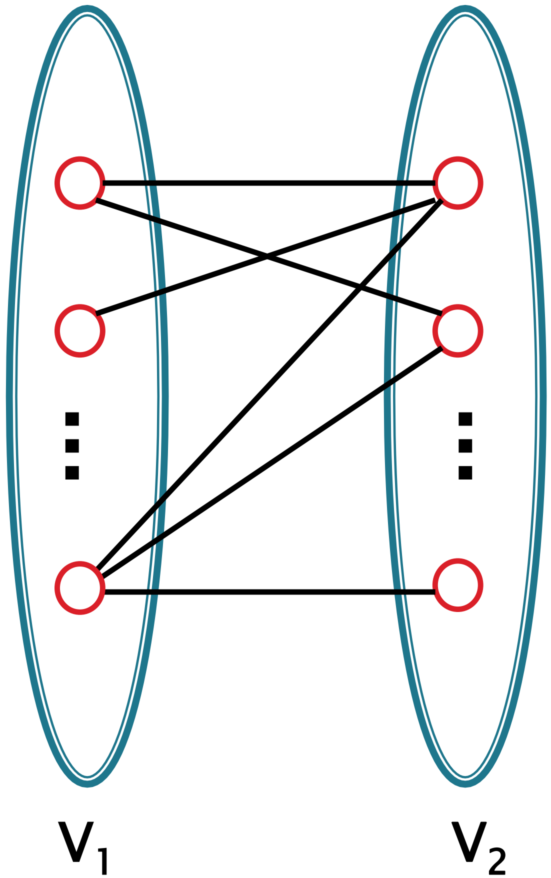
$$V = V_1 \cup V_2$$

$$V_1 \cap V_2 = \varnothing$$

astfel încât orice muchie $e \in E$ are o extremitate

în V_1 și cealaltă în V_2:

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


V_1
V_2

Graf bipartit

- $G = (V, E)$ **bipartit** $\Leftrightarrow$ există o colorare a vârfurilor cu două culori:

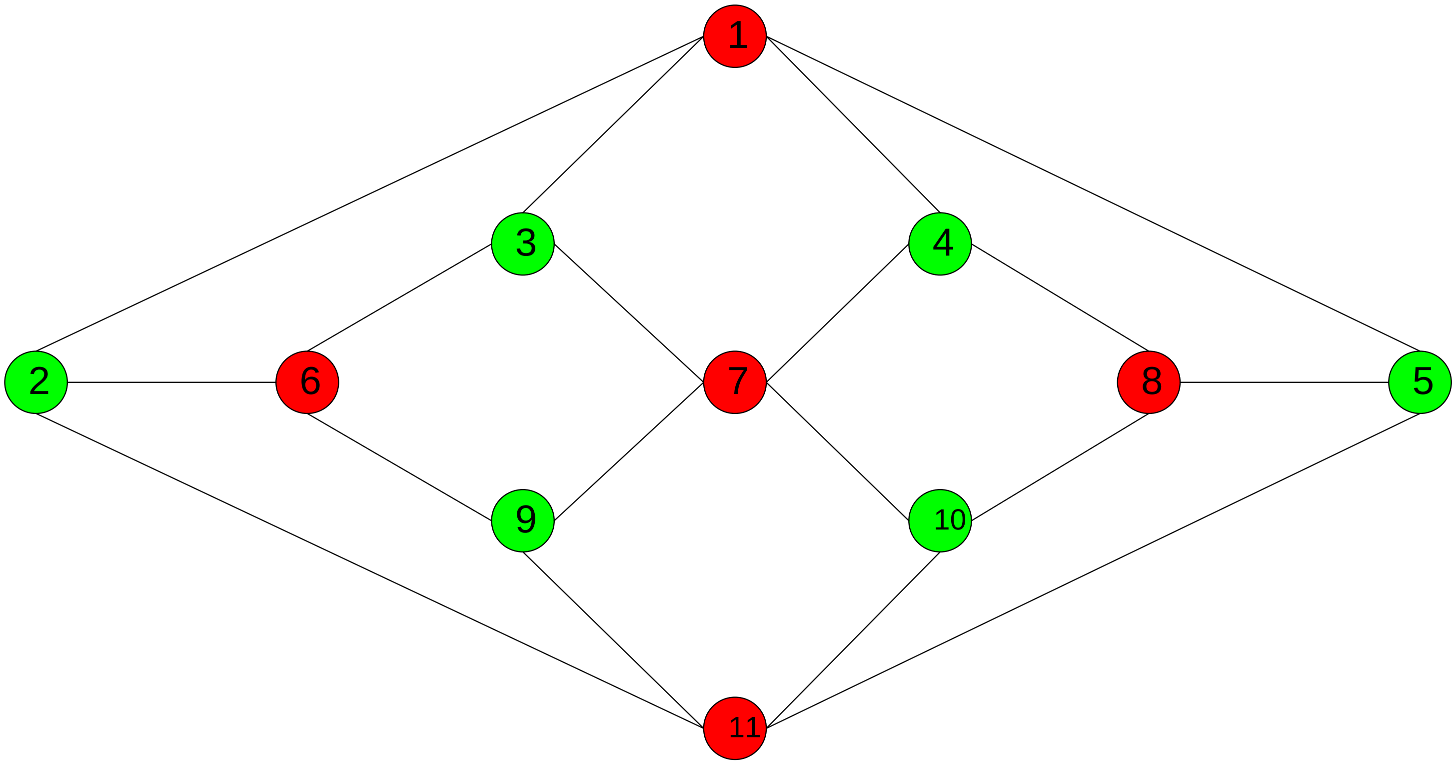
$$c : V \rightarrow \{1, 2\}$$

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

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

(**bicolorare**)

Graf bipartit



Graf bipartit