

# Parcurgerea grafurilor. Aplicații

# Aplicații

## ► Test graf conex



`bf(1) / df(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); //merge si df(i) ?  
    }
```

# Aplicații

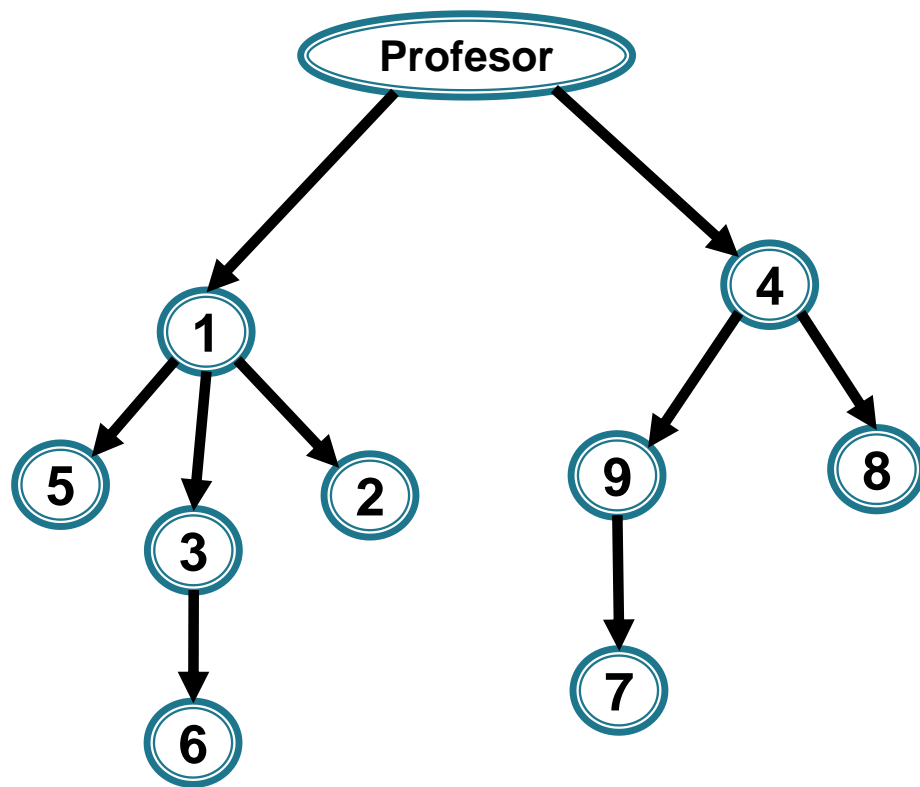
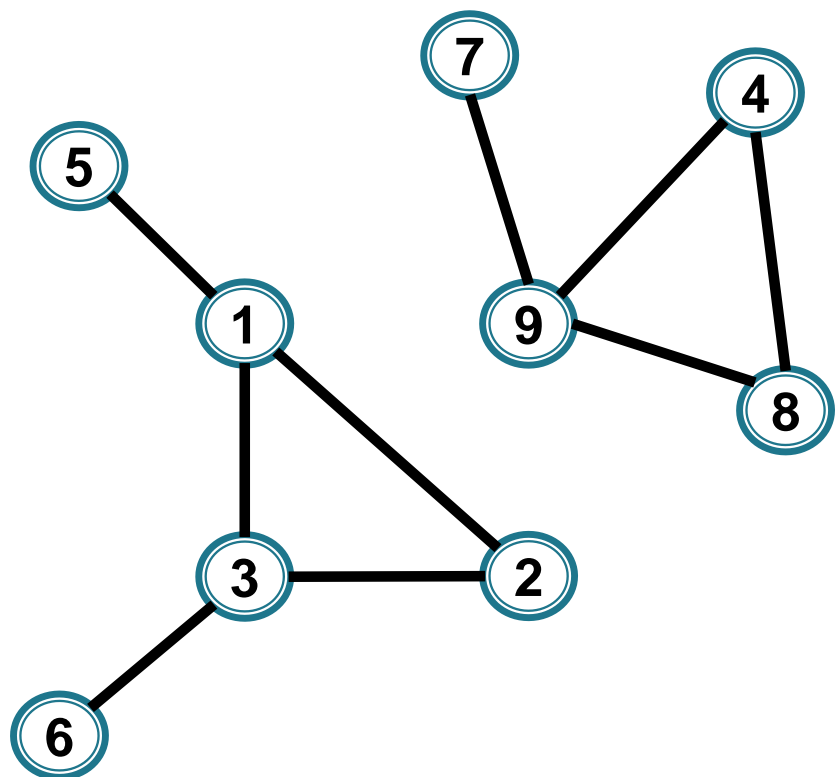
- ▶ Determinarea unui arbore parțial al unui graf conex



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

# 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.



# Aplicații

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



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

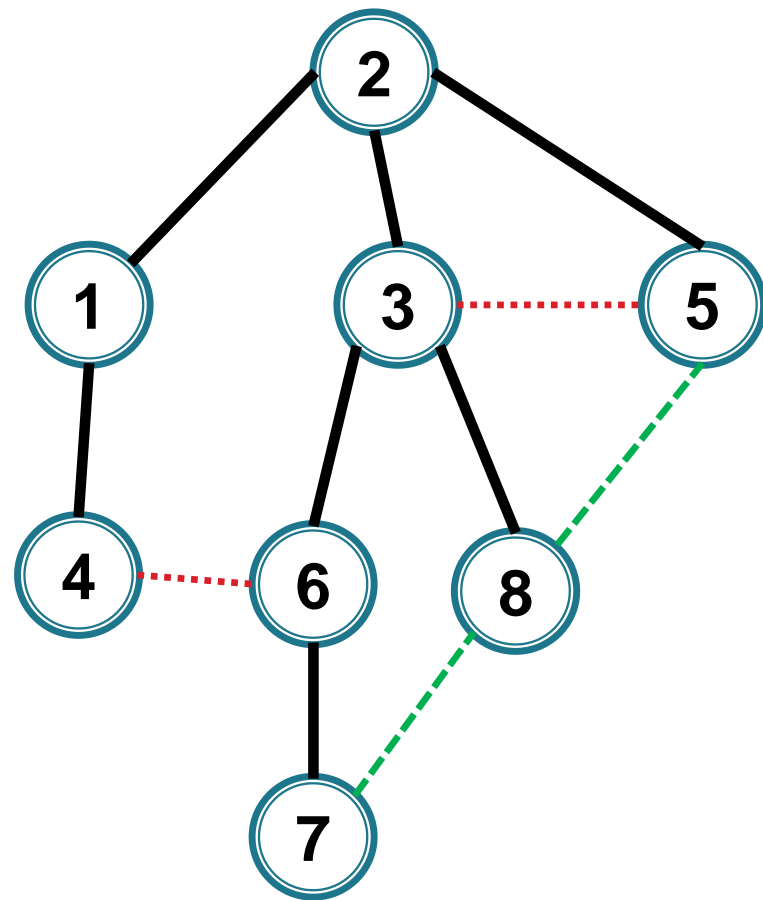
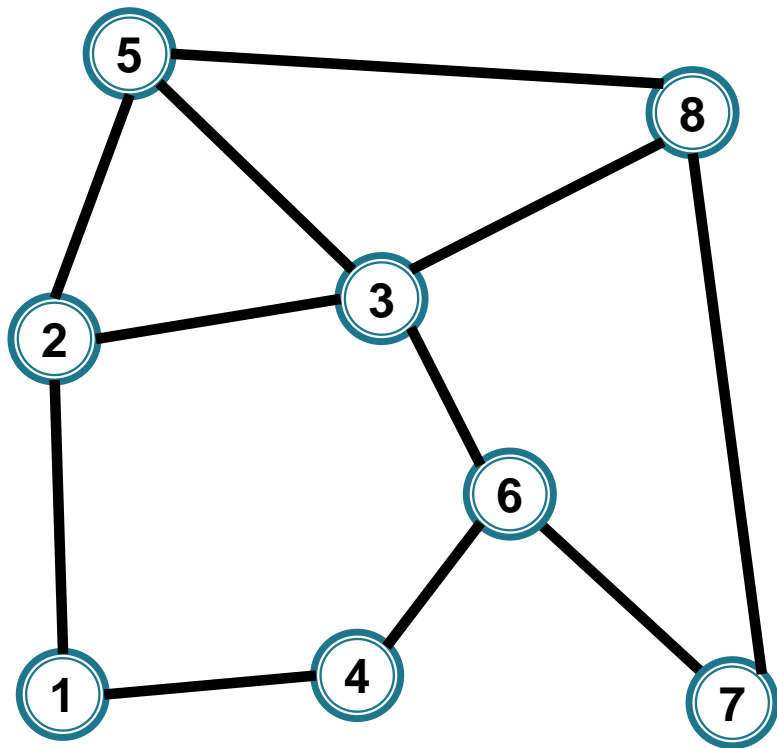
```
bf(u) ;  
if (viz[v] == 1)  
    lant(v) ;  
else  
    cout<<"nu exista drum" ;
```

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

# Aplicații

- ▶ Determinarea mai multor lanțuri minime între două vârfuri date  $u$  și  $v$



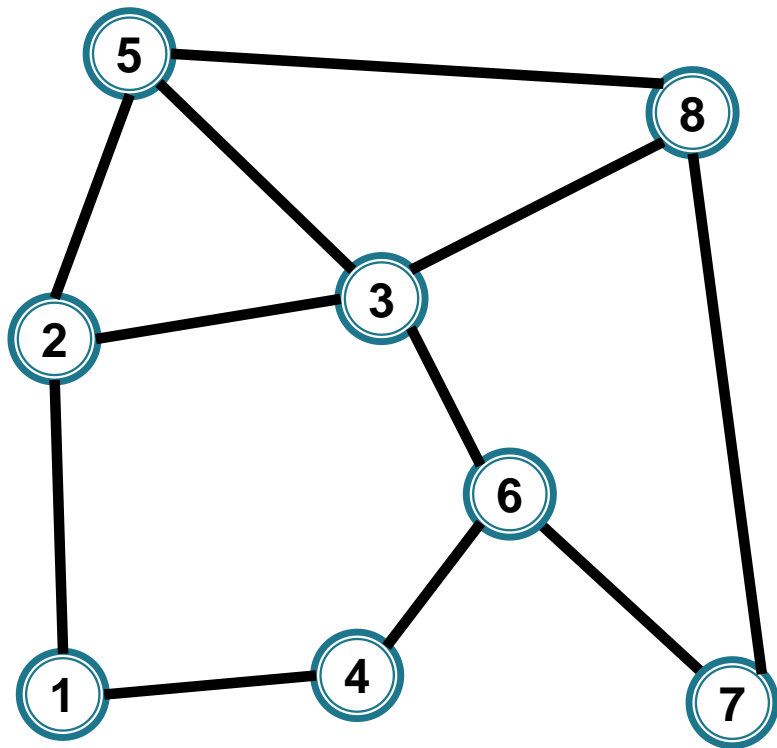


$u = 2$

$v = 7$



$tata[x]$  = lista tuturor vârfurilor care pot fi tata pentru  $x$  în parcurgerea BF



$u = 2$

$v = 7$

Tata:

1: 2

2:

3: 2

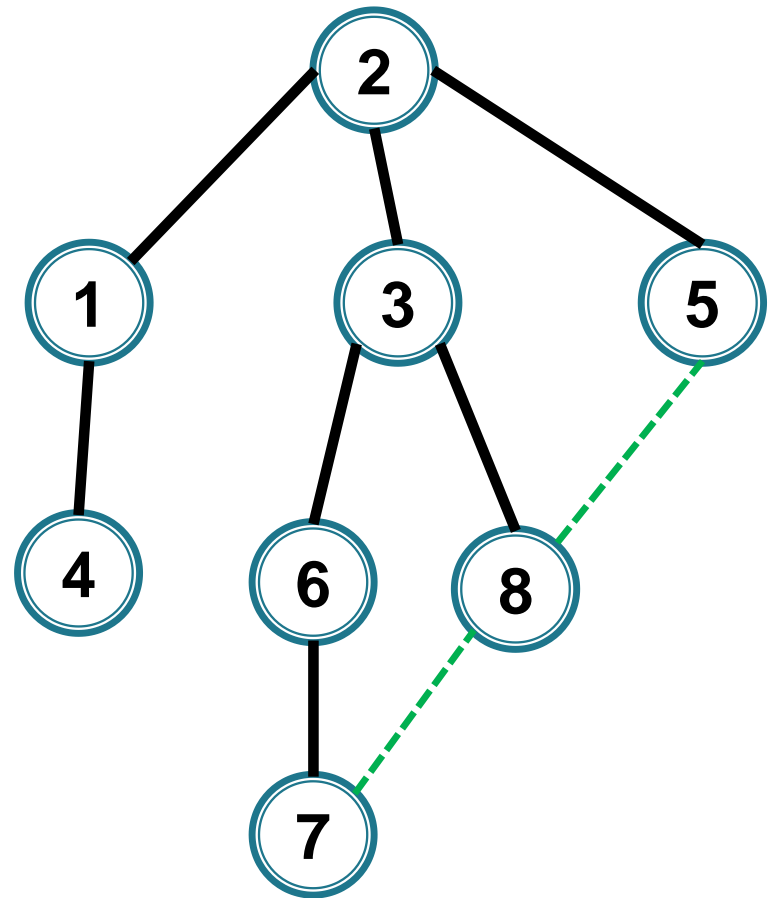
4: 1

5: 2

6: 3

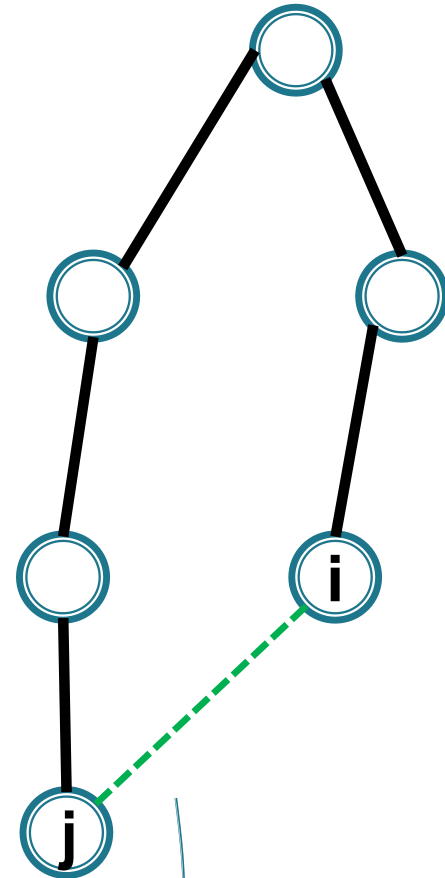
7: 6, 8

8: 3, 5



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



# Aplicații



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



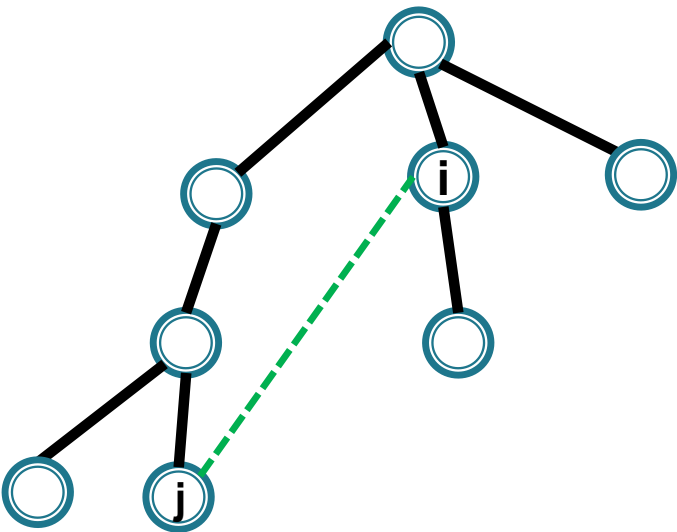
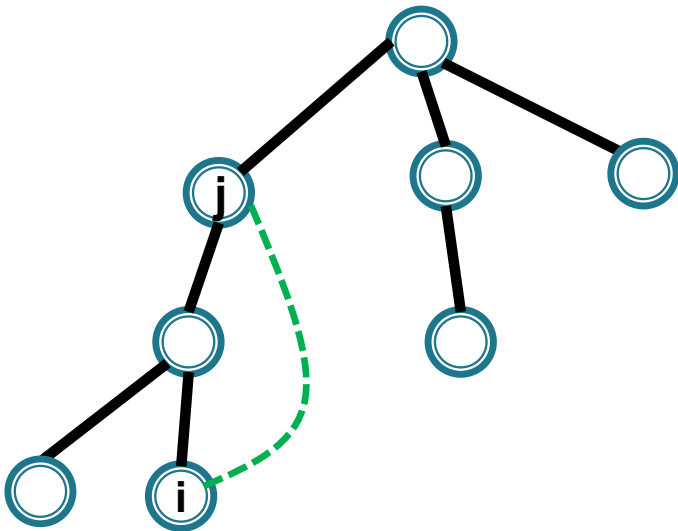
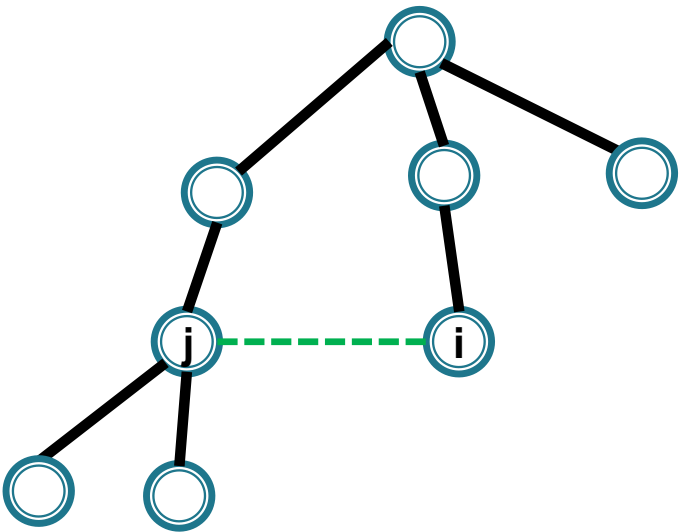
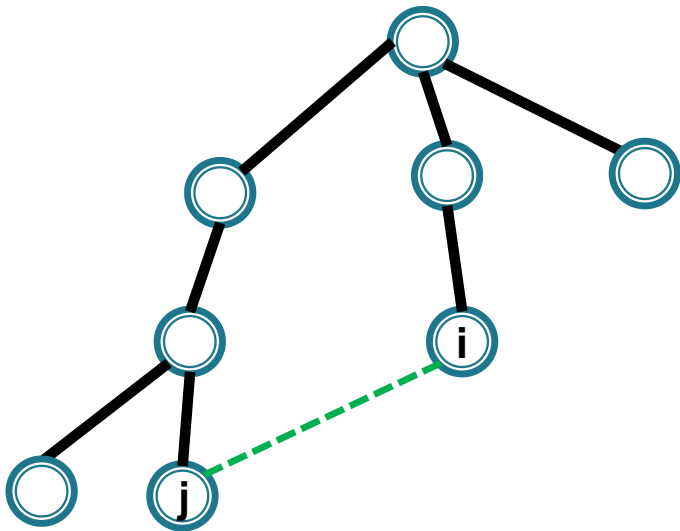
Un ciclu se închide în parcurge când vârful curent are un vecin deja vizitat, care nu este tatăl lui



Problema se poate rezolva folosind oricare dintre cele două parcurgeri?

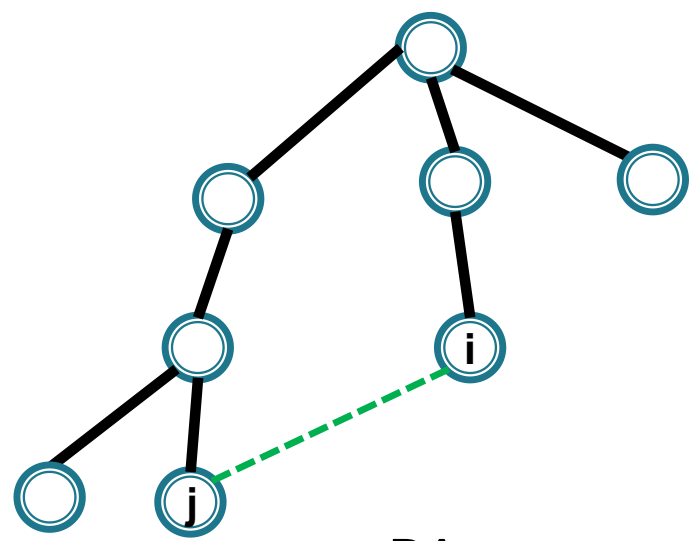
Care situație (muchie care închide ciclu cu muchiile din arborele BF) este posibilă?

BF

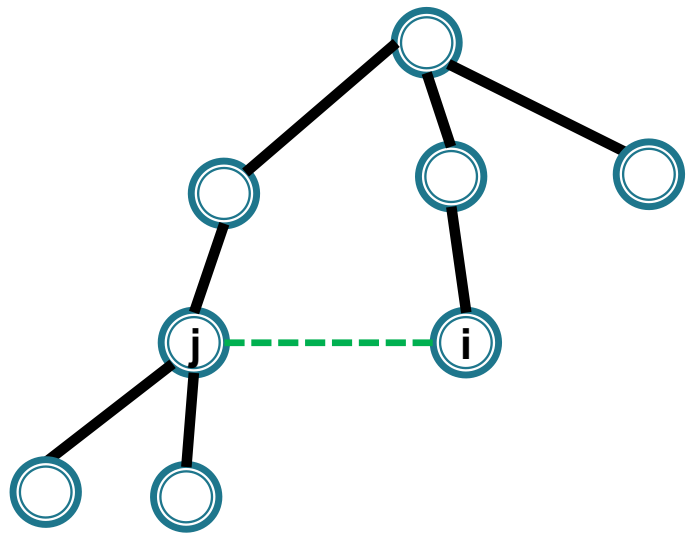


Care situație (muchie care închide ciclu cu muchiile din arborele BF) este posibilă?

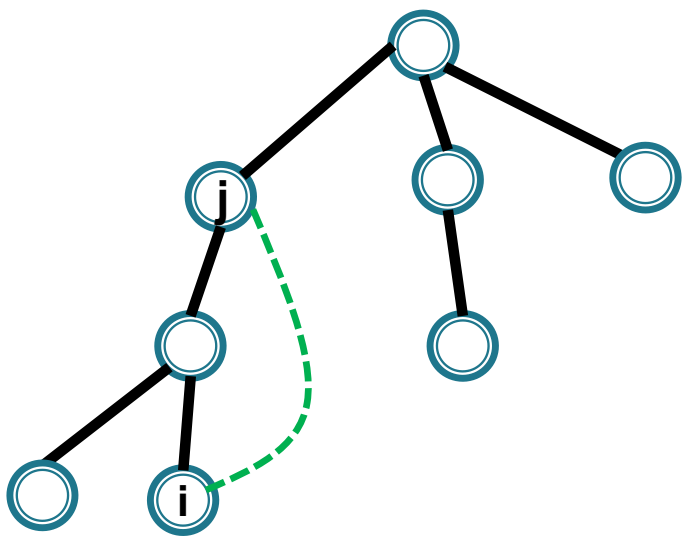
BF



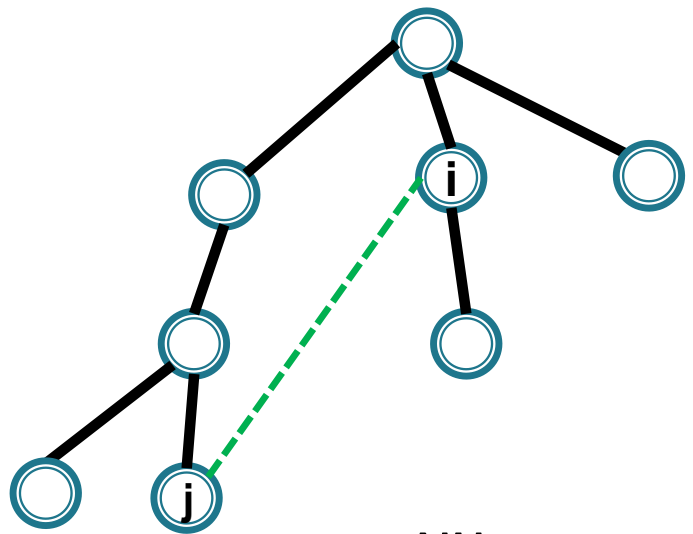
DA



DA



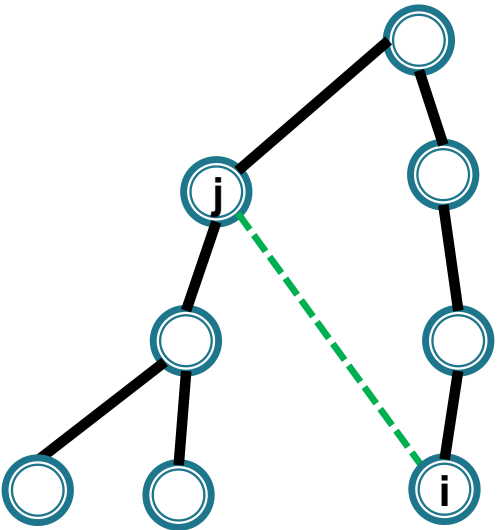
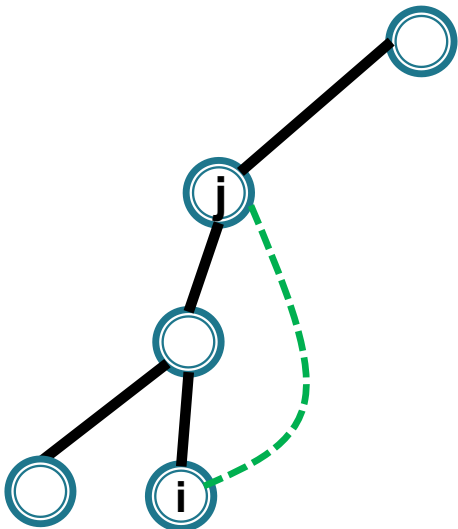
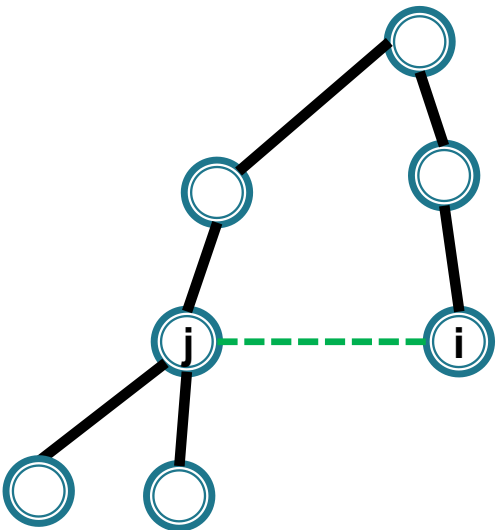
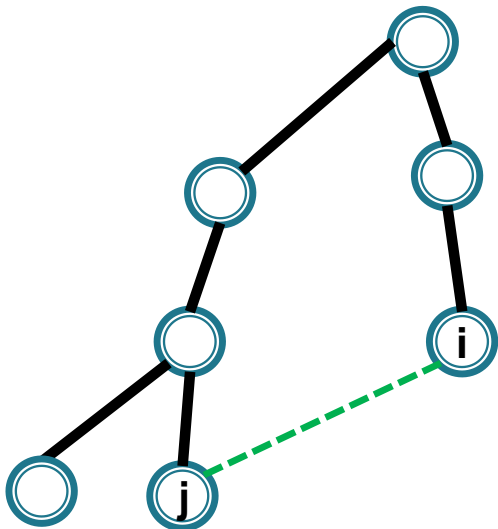
NU



NU

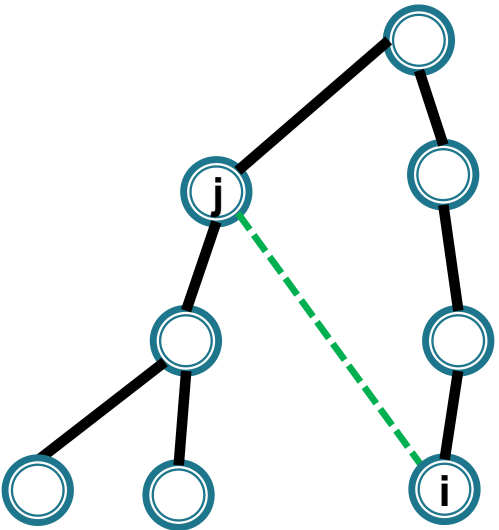
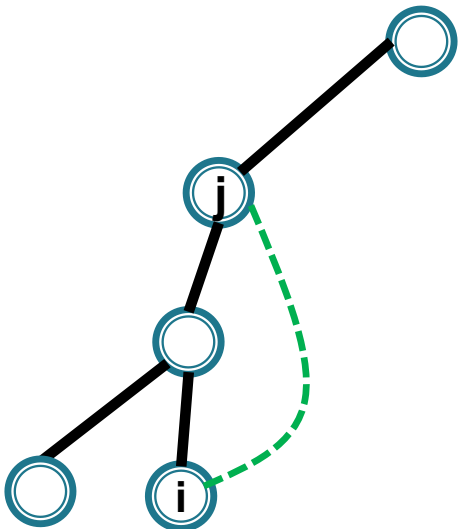
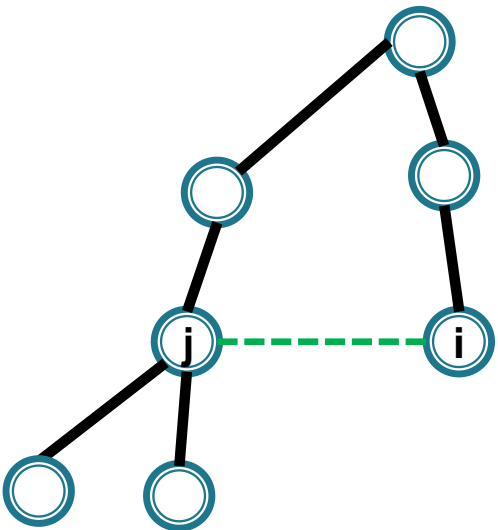
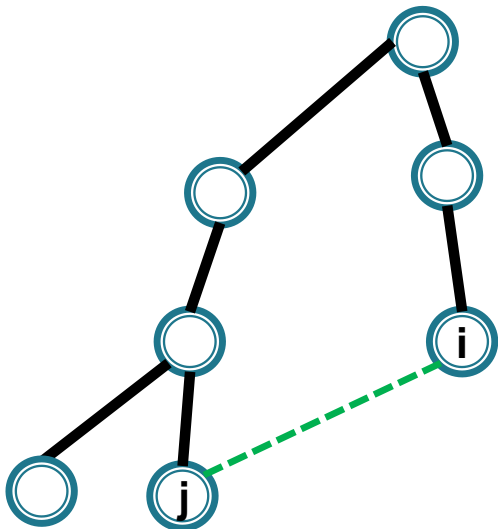
Care situație (muchie care închide ciclu cu muchiile din arborele DF) este posibilă?

DF



Care situație (muchie care închide ciclu cu muchiile din arborele DF) este posibilă?

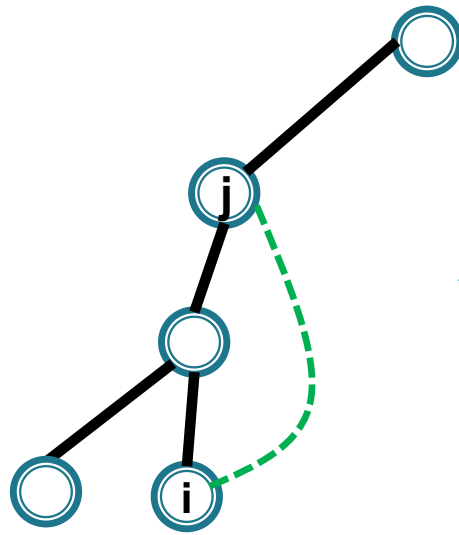
DF



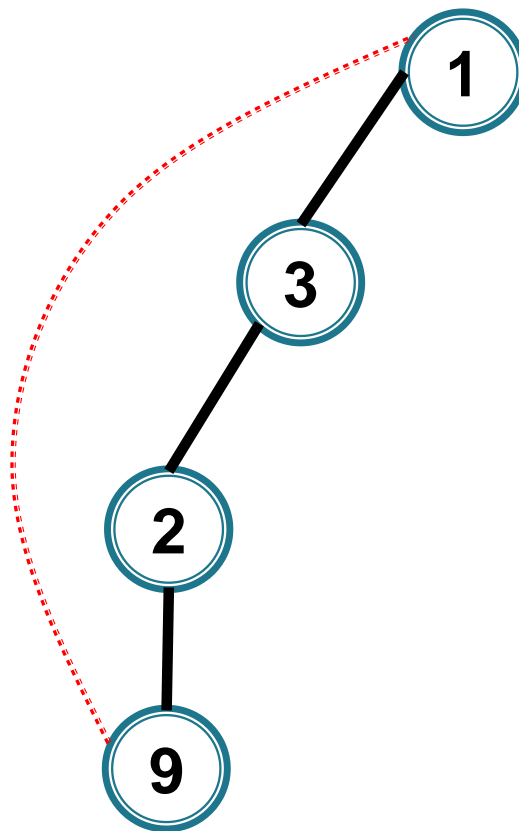
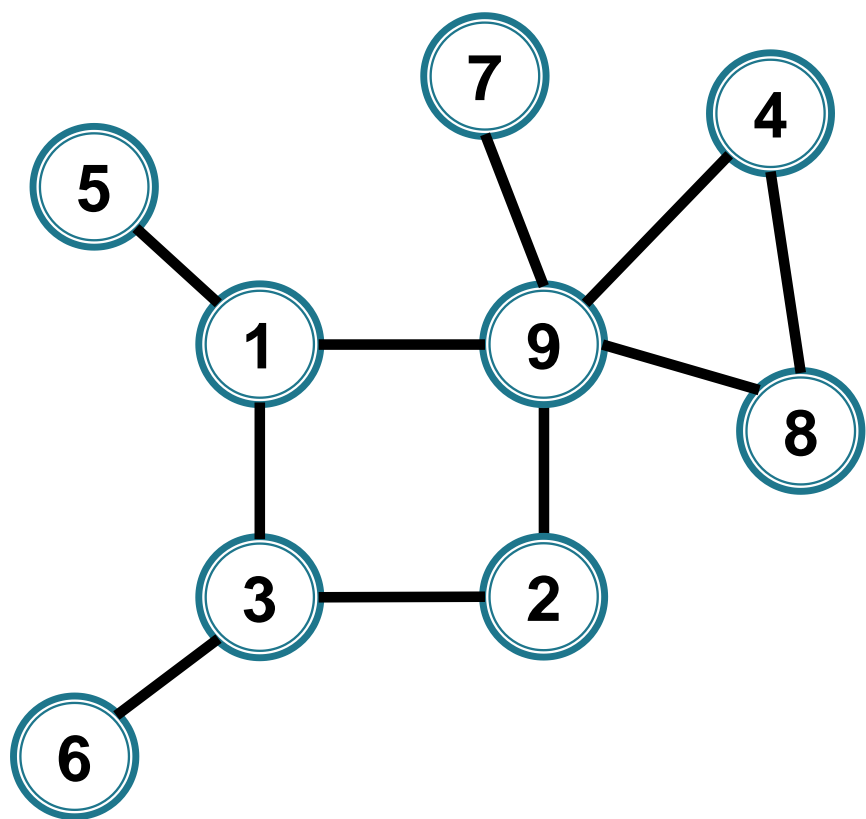
DA

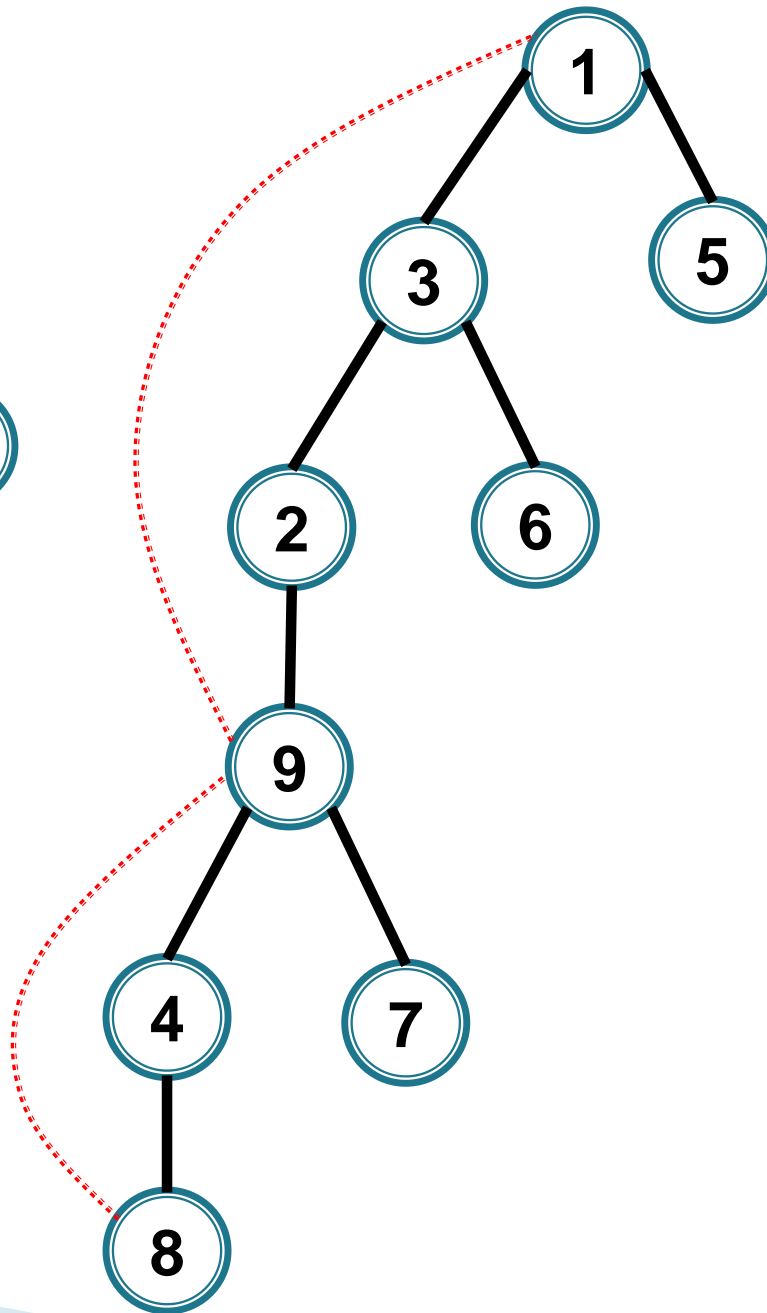
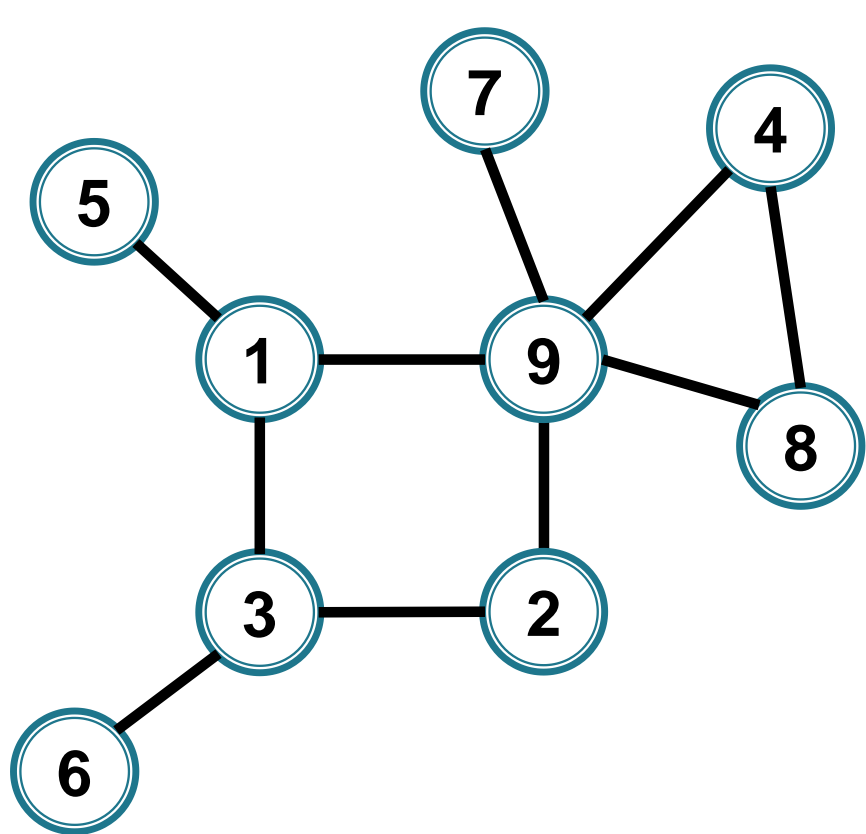


DF



← Muchie de întoarcere





```

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;
            }
    }
}

```

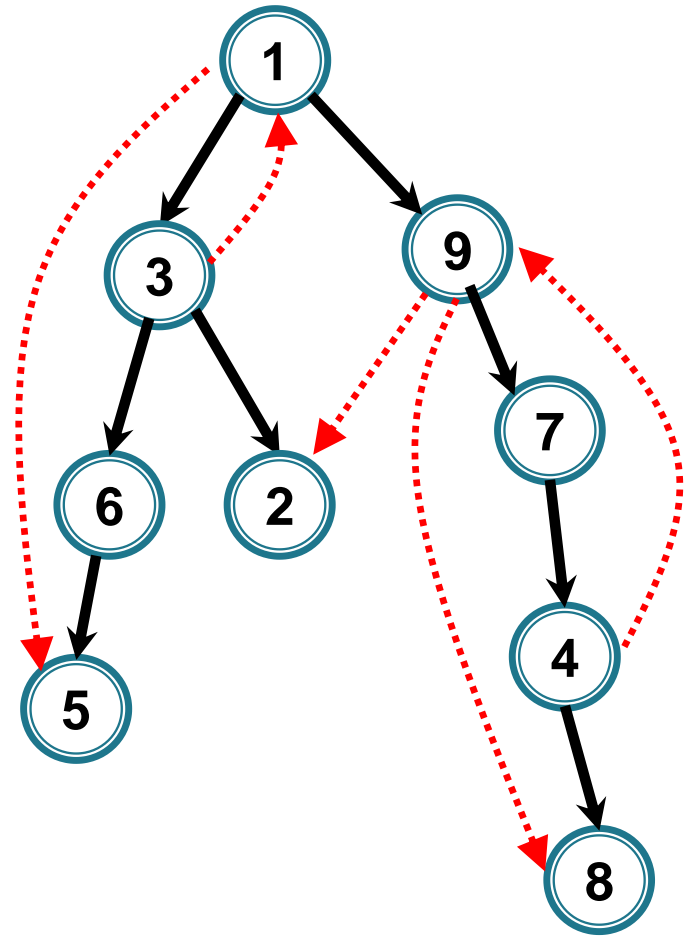
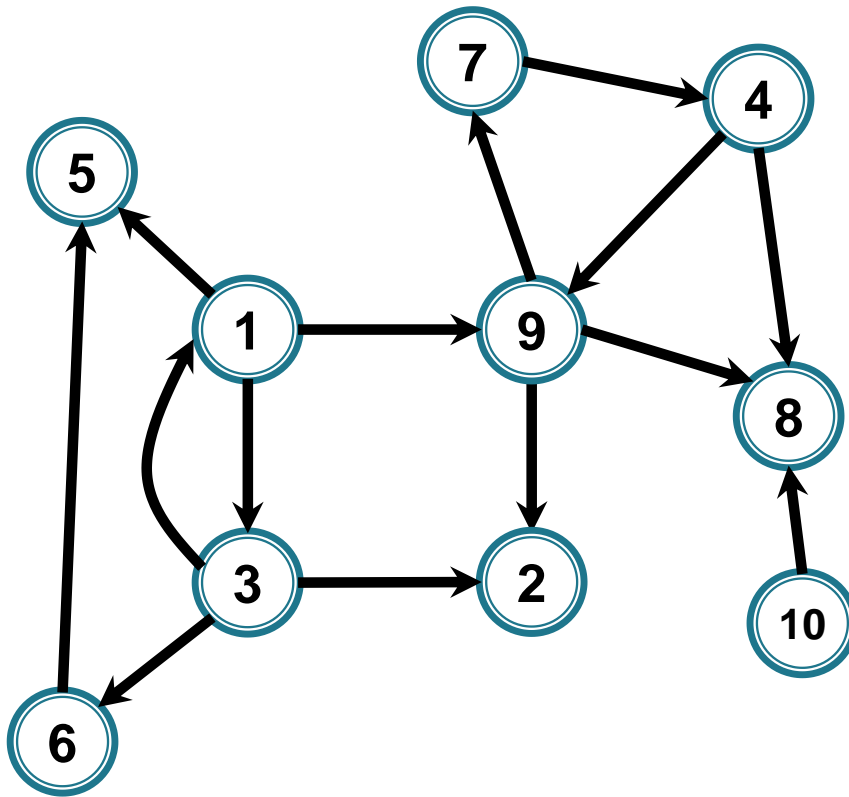
# Aplicații



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

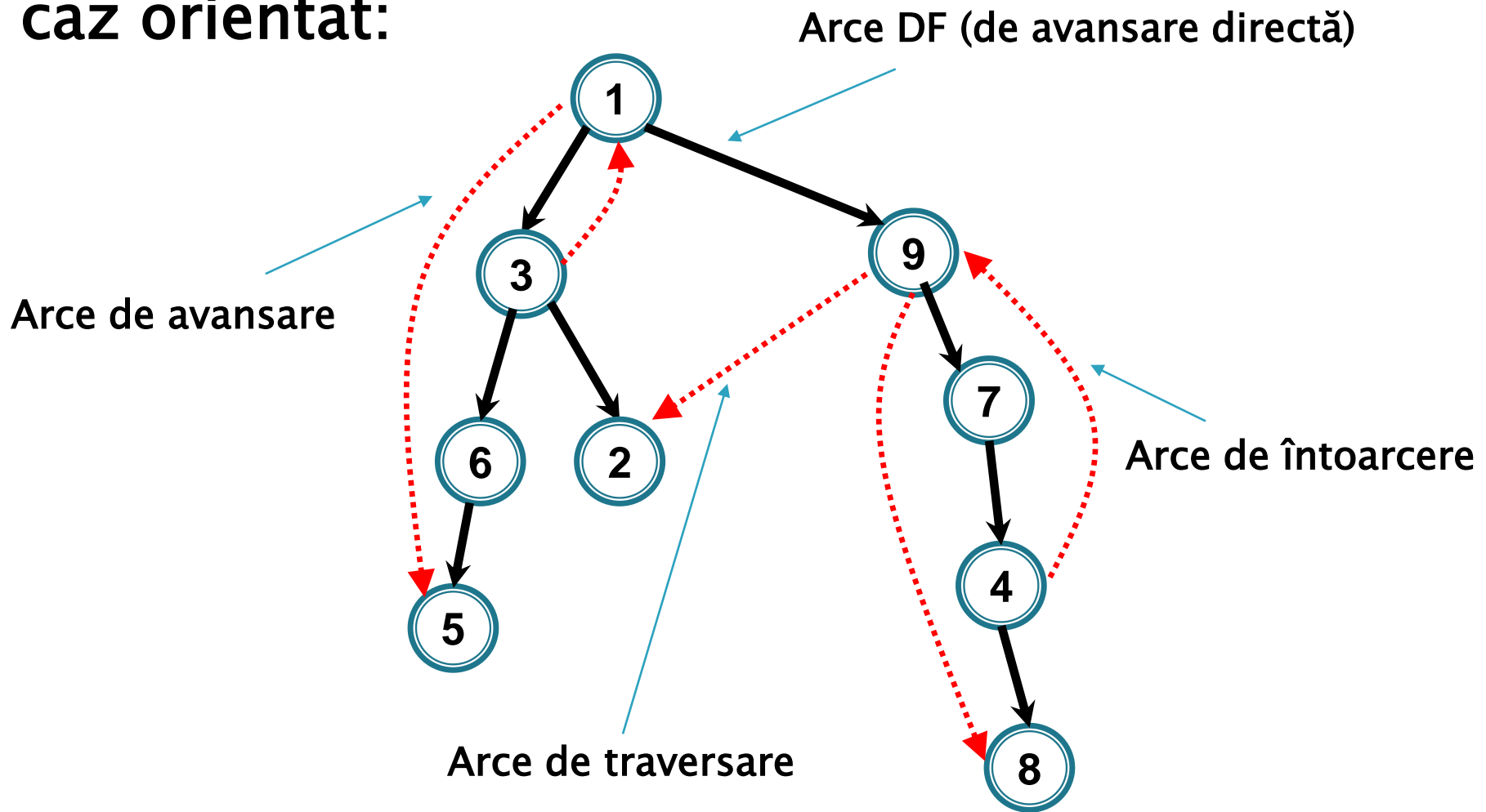
# Parcurgerea în adâncime

## ► Exemplu – caz orientat:



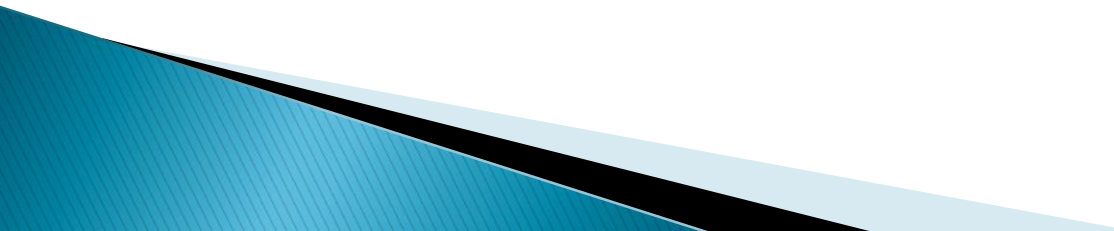
# Parcurgerea în adâncime

**caz orientat:**



**Doar arcele de întoarcere închid circuite**

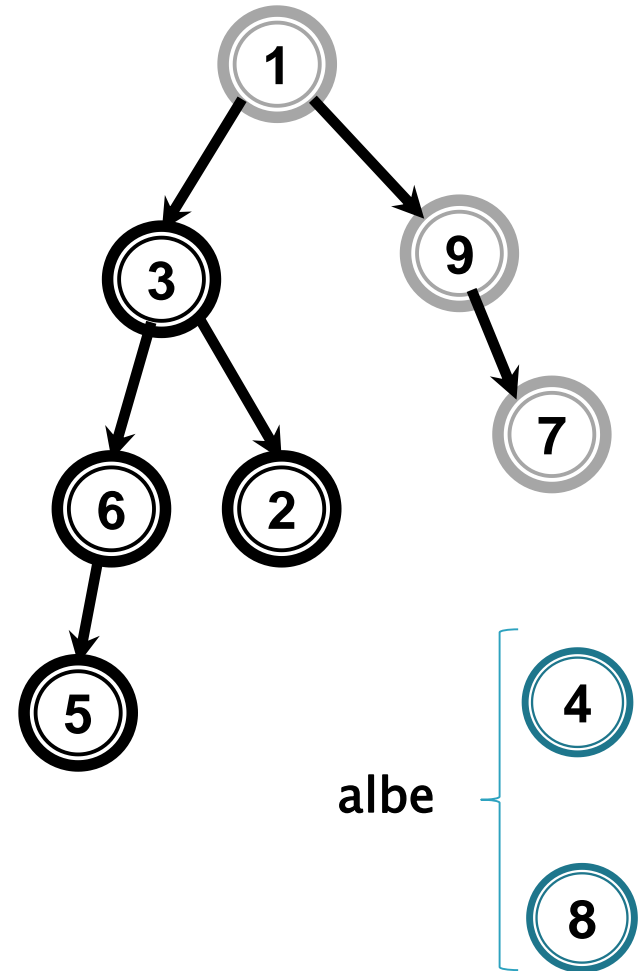
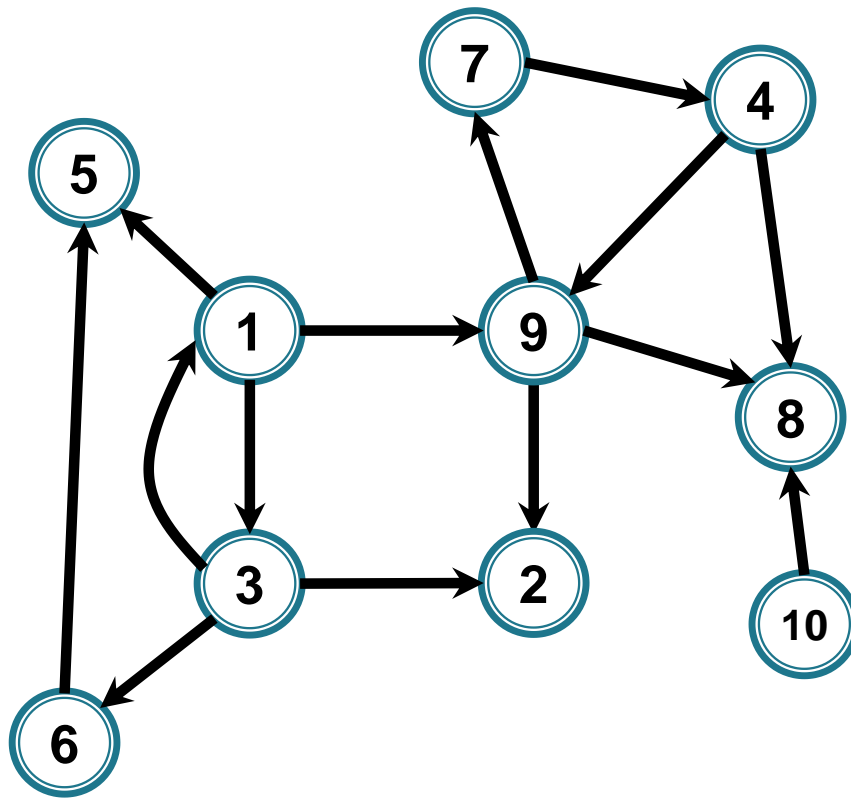
# Parcurgerea în adâncime

- ▶ Starea nodurilor în DF:
  - ▶ albe = nevizitate = încă nu s-a ajuns la ele
  - ▶ gri = vizitate, în curs de explorare
  - ▶ negre = finalizate
- 



# Parcurgerea în adâncime

- ▶ Starea nodurilor după ce 7 devine vârf curent:



# 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 lăţime pe matrice

- ▶ Se dă un labirint sub forma unei matrice  $n \times n$  cu elemente 0 şi 1, 1 semnificând perete (obstacol) iar 0 celula liberă. Prin labirint ne putem deplasa doar din celula curentă 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 ieşire din labirint. Date două coordonate  $x$  şi  $y$ , să se decidă dacă există un drum din celula  $(x,y)$  prin care se poate ieşi din labirint. În caz afirmativ să se afişeze **un drum minim către ieşire**.

# Parcurgere în lățime pe matrice




BF

3	2		2		
	1	0	1		
		1	2		

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);

        }

    }

}

```