# *Parcurgerea în lățime*

- Graf orientat sau neorientat, conex, ponderat sau neponderat
- Determinarea arborelui parțial de cost minim pentru grafuri ponderate
- Aplicații: determinarea componentelor conexe, drumuri minime

**Pseudocod**:

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

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
```

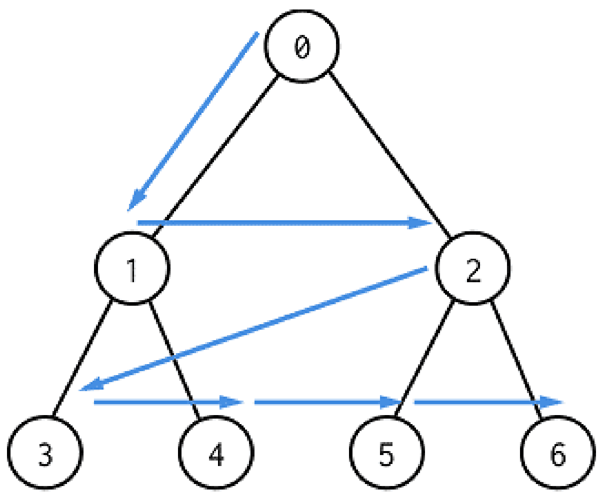
Pentru a parcurge toate vârfurile grafului se reia apelul subprogramului BFS pentru vârfuri rămase nevizitate:

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

**Complexitate:**

- Matrice de adiacență: $O(n^2)$
- Listă de adiacență: $O(n + m)$

**Exemplu:**



# *Parcurgerea în adâncime*

- Graf orientat sau neorientat, conex, ponderat sau neponderat
- Determinarea arborelui parțial de cost minim pentru grafuri ponderate
- Aplicații: determinarea componentelor conexe, arbore parțial, cicluri și circuite și componente tare conexe

**Pseudocod**:

```
DFS(x)
      culoare[x] = gri
      timp = timp + 1
      desc[x]= timp;//incepe explorarea varfului x
      pentru fiecare  xy ∈ E //y vecin al lui x
        daca culoare[y]==alb atunci
              tata[y] = x
              d[y] = d[x]+1 //nivel, nu distanta
              DFS(y)
      culoare[x] = negru
      timp = timp + 1
      fin[x] = timp //s-a finalizat explorarea varfului x
```

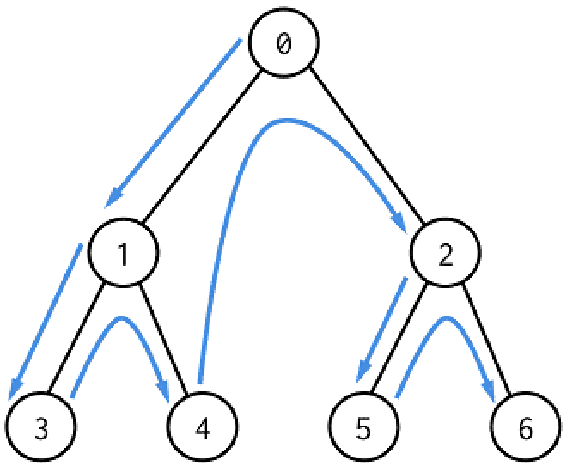
Apel:

```
pentru fiecare x in V executa
      culoare[x] = alb
timp = 0
pentru fiecare x in V executa
        daca culoare[x] == alb
            DFS(x)
```

**Complexitate:**

- Matrice de adiacență: $O(n^2)$
- Lista de adiacență: $O(n + m)$
- Traversare: O (n)

**Exemplu:**



## *Algoritmul lui Kosaraju*

- Determinarea componentelor tare conexe dintr-un graf orientat

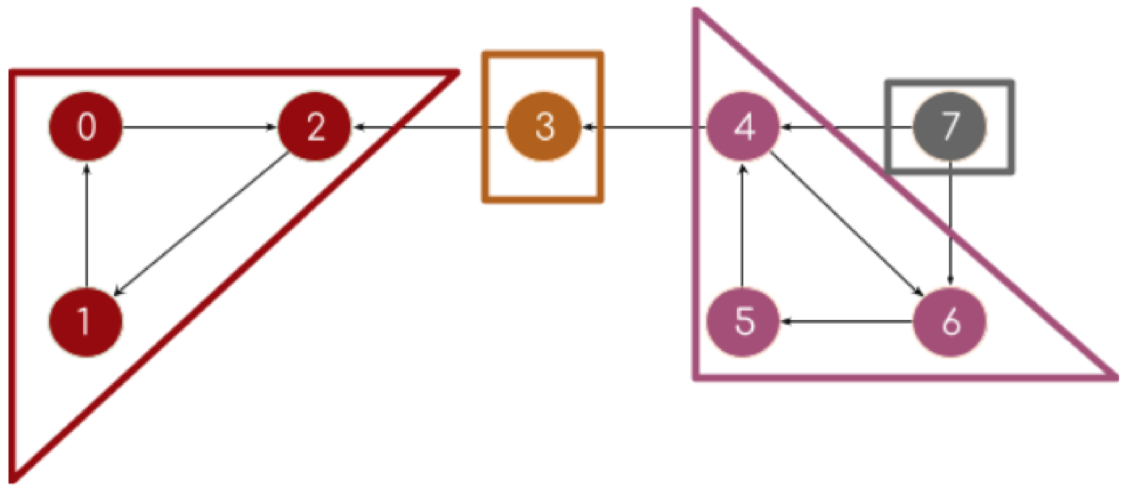
**Algoritm**:

1. Parcurgerea grafului cu DFS și introducerea vârfurilor într-o stivă
2. Inversarea sensurilor din graf și parcurgerea nodurilor cu DFS având ca nod de start elemetele din stivă (descrescator după timpul de finalizare)
3. Vârfurile vizitate în fiecare apel al DFS-ului în graful invers reprezintă o componentă tare conexă

**Complexitate:**

- Două parcurgeri și construirea grafului invers: O(n + m)

**Exemplu:**




*Componente tare conexe:*

- Componenta 1: 0, 2, 1
- Componenta 2: 3
- Componenta 3: 4, 6, 5
- Componenta 4: 7

# *Sortarea topologică*

- Graf orientat, aciclic, ponderat sau neponderat
- Ordonarea vârfurilor astfel încât dacă uv e muchie atunci u se află înaintea lui v în ordonare
- Aplicații: ordinea de calcul unde intervin dependențe, detecția de deadlock, determinarea drumurilor critice

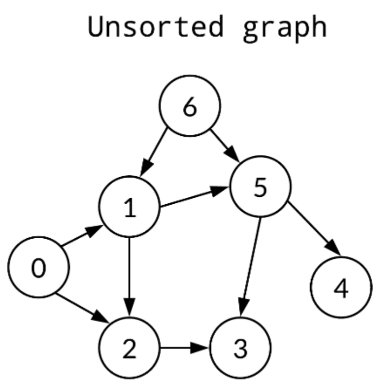
**Algoritm**:

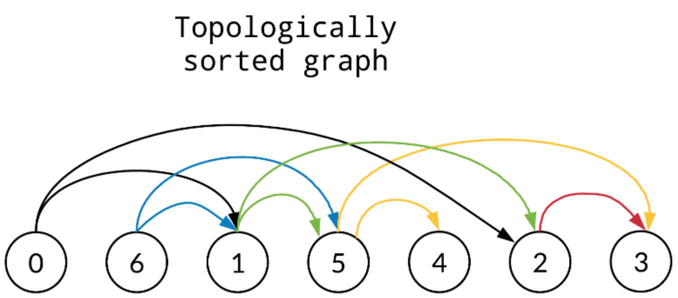
1. Adăugăm toate vârfurile cu grad intern 0 într-o coadă
2. Extragem un vârf din coadă și îl eliminăm din graf (nu din coadă) și scădem gradele interne ale vecinilor
3. Repetăm pasul 1 și 2 până când nu mai avem noduri rămase

**Pseudocod**:

```
coada C = ∅;
adauga in C toate vârfurile v cu d-[v]=0

cat timp C ≠ ∅ executa
   i ←  extrage(C);
   adauga i in sortare
   pentru ij ∈ E executa
      d-[j] = d-[j] - 1
      daca d-[j]==0 atunci
          adauga(j, C)
```

**Complexitate:** O(n + m)



# *Algoritmul lui Kruskal*

- Graf conex, ponderat, neorientat
- Determinarea arborelui parțial de cost minim: graf conex fără cicluri
- Aplicații: proiectarea de rețele, clustering, protocoale de rutare

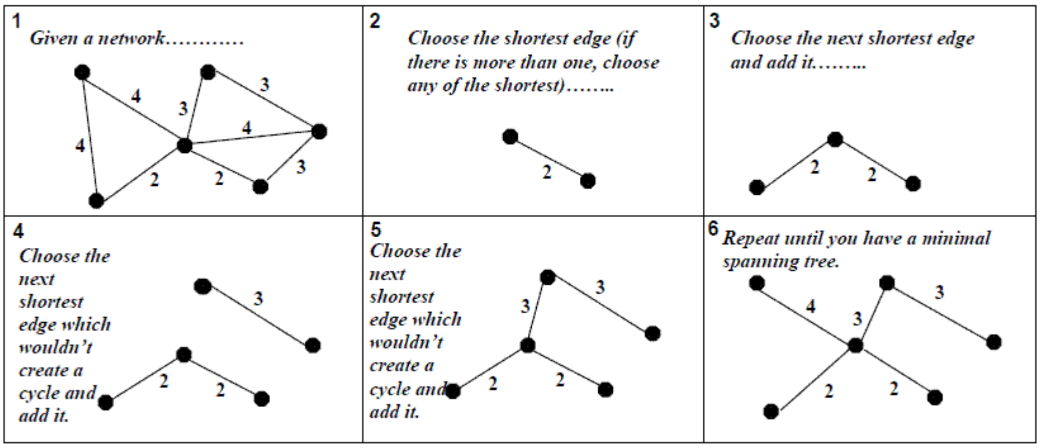
**Algoritm**:

1. Sortăm muchiile crescător după cost (algoritm Greedy)
2. Selectăm muchia de cost minim care nu formează cicluri cu muchiile deja selectate (care unește două componente conexe)
3. Pentru a testa dacă se formează un ciclu vom marca două noduri unite cu aceeași culoare și vom unii două noduri doar dacă au culori diferite
4. Oprim algoritmul când toate nodurile au aceeași culoare

**Pseudocod**:

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

**Complexitate:** O(m log n)



# *Algoritmul lui Prim*

- Graf conex, ponderat, neorientat
- Determinarea arborelui partial de cost minim: graf conex fara cicluri
- Aplicatii: proiectarea de retele, clustering, protocoale de rutare

**Algoritm**:

1. Selectăm o muchie de cost minim de la un vârf deja adăugat în arbore la unul neadăugat
2. Pentru fiecare vârf neselectat memorăm doar muchia de cost minim care îl unește de un vârf deja adăugat în arbore

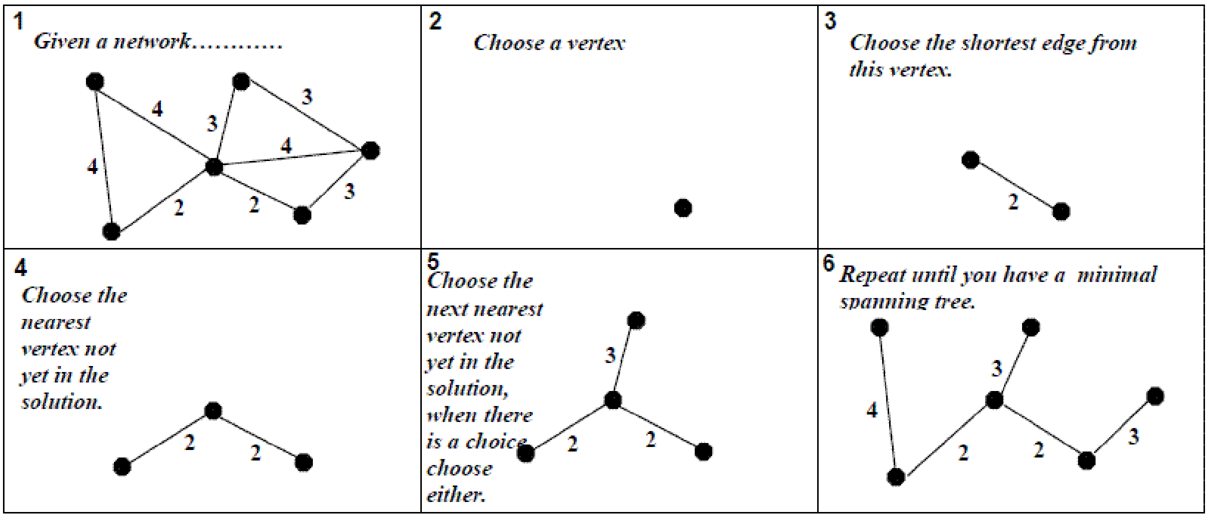
**Pseudocod**:

```
s– vârful de start
inițializează Q cu V
pentru fiecare u∈V executa
      d[u] = ∞; tata[u]=0
 d[s] = 0
cat timp Q ≠ ∅ executa
     extrage un vârf u∈Q cu eticheta d[u] minimă
     pentru fiecare uv∈E executa
                daca v∈Q si w(u,v)<d[v] atunci
                     d[v] = w(u,v)
                     tata[v] = u
scrie (u, tata[u]), pentru u≠ s
```

```
pentru fiecare u∈V executa
     d[u] = ∞; tata[u]=0
d[s] = 0
inițializează Q cu V
cat timp Q ≠ ∅ executa
       u=extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
              daca v∈Q si w(u,v)<d[v] atunci
                   d[v] = w(u,v)
                   tata[v] = u
                   //actualizeaza Q - pentru Q heap
scrie (u, tata[u]), pentru u≠ s
```

**Complexitate:**

- Implementare cu coadă: $O(n^2)$
- Implementare cu min-heap: $O(m \log n)$



# *Directed Acyclic Graph*

- Graf orientat, aciclic, ponderat și poate avea ponderi negative
- Determinarea drumului minim de la o sursă unică

**Algoritm**:

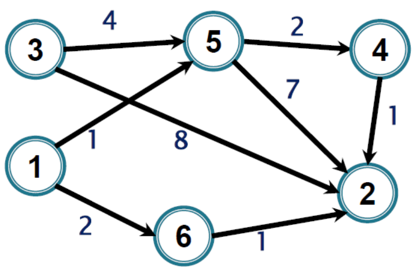
1. Considerăm vârfurile în ordinea dată de sortarea topologică
2. Pentru fiecare vârf relaxăm arcele către vecinii săi pentru a găsi posibile drumuri noi minime către aceștia

**Pseudocod**:

```
s – vârful de start

//initializam distante – ca la Dijkstra
pentru fiecare u∈V executa
       d[u] = ∞; tata[u]=0
d[s] = 0

//determinăm o sortare topologică a vârfurilor
SortTop = sortare_topologica(G)

pentru fiecare u ∈ SortTop
       pentru fiecare uv∈E executa
            daca d[u]+w(u,v)<d[v] atunci  //relaxam uv
                  d[v] = d[u]+w(u,v)
                  tata[v] = u
scrie d, tata
```

**Complexitate:** $O(m + n)$

- Inițializare: $O(n)$
- Sortare: $O(m + n)$
- Relaxări: $O(m)$




Sortare topologică
1, 3, 6, 5, 4, 2

s=3 – vârf de start

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

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| d/tata | [ ∞/0, | ∞/0, | 0/0, | ∞/0, | ∞/0, | ∞/0 ] |
| u = 1: | [ ∞/0, | ∞/0, | 0/0, | ∞/0, | ∞/0, | ∞/0 ] |
| u = 3: | [ ∞/0, | 8/3, | 0/0, | ∞/0, | 4/3, | ∞/0 ] |
| u = 6: | [ ∞/0, | 8/3, | 0/0, | ∞/0, | 4/3, | ∞/0 ] |
| u = 5: | [ ∞/0, | 8/3, | 0/0, | 6/5, | 4/3, | ∞/0 ] |
| u = 4: | [ ∞/0, | 7/4, | 0/0, | 6/5, | 4/3, | ∞/0 ] |
| u = 2: | [ ∞/0, | 7/4, | 0/0, | 6/5, | 4/3, | ∞/0 ] |

## *Dijkstra*

- Graf orientat, poate fi ciclic, ponderat cu valori pozitive
- Determinarea drumului minim de la o sursă unică

**Algoritm**:

1. Selectăm o vârful care are asociat costul minim cost minim
2. Pentru fiecare vârf relaxăm arcele către vecinii săi pentru a găsi posibile drumuri noi minime către aceștia

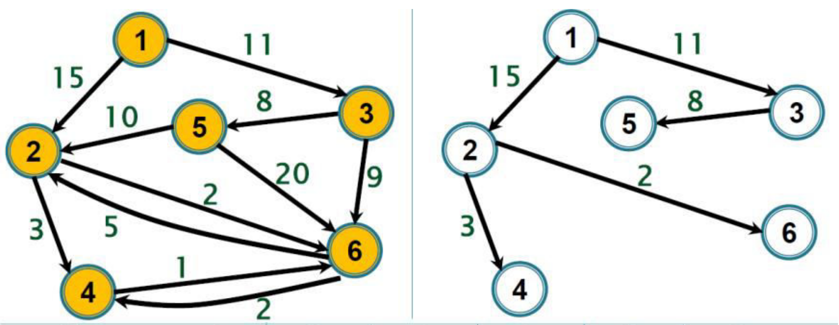
**Pseudocod**:

```
inițializează mulțimea vârfurilor neselectate Q cu V
pentru fiecare u∈V executa
     d[u] = ∞; tata[u]=0
d[s] = 0
cat timp Q ≠ ∅ executa
     u = extrage vârf cu eticheta d minimă din Q
     pentru fiecare uv∈E executa
          daca d[u]+w(u,v)<d[v] atunci
                d[v] = d[u]+w(u,v)
                tata[v] = u
scrie d, tata

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

```
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0
Q = V //creare heap cu cheile din d
cat timp Q ≠ ∅ executa
    u = extrage_min(Q)
    pentru fiecare uv∈E executa
         daca d[u]+w(u,v)<d[v] atunci
                d[v] = d[u]+w(u,v)
                repara(Q,v)
                tata[v] = u
scrie d, tata

//scrie drum minim de la s la t un varf t dat
```

**Complexitate:**

- Implementare cu coadă: $O(n^2)$
- Implementare cu min-heap: $O(m \log n)$




| d/tata | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | [ 0/0, | ∞/0, | ∞/0, | ∞/0, | ∞/0, | ∞/0 ] |
| Sel. 1: | [ – , | 15/1, | 11/1, | ∞/0, | ∞/0, | ∞/0 ] |
| Sel. 3: | [ – , | 15/1, | – , | ∞/0, | 19/3, | 20/3 ] |
| Sel. 2: | [ – , | – , | – , | 18/2, | 19/3, | 17/2 ] |
| Sel. 6: | [ – , | – , | – , | 18/2, | 19/3, | – ] |
| Sel. 4: | [ – , | – , | – , | – , | 19/3, | – ] |
| Sel. 5: | [ – , | – , | – , | – , | – , | – ] |

## *Bellman Ford*

- Graf orientat, ponderat si ponderile pot fi negative, fără cicluri negative
- În cazul în care există cicluri negative acestea pot fi detectate
- Determinarea drumului minim de la o sursă unică

**Algoritm**:

1. Relaxăm toate arcele din graf
2. Repetăm primul pas până când nu mai apar îmbunătățiri la distanțe: vom avea cel mult n-1 repetări
3. În cazul în care există un circuit negativ se va ajunge la pasul n

**Pseudocod**:

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

pentru i = 1,n-1 executa
   pentru fiecare uv∈E executa
          daca d[u]+w(u,v)<d[v] atunci
                 d[v] = d[u]+w(u,v)
                 tata[v] = u
```

**Complexitate:** O(nm)