Grafuri

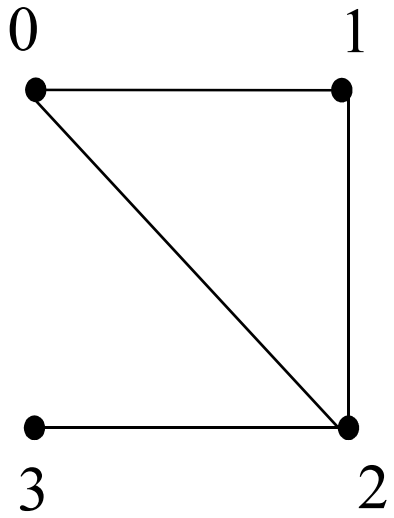
SD 2014/2015

Conținut

- Grafuri
  - tipul abstract Graf;
  - tipul abstract Digraf;
  - implementarea cu matrici de adiacenţă;
  - implementarea cu liste de adiacenţă înlănţuite;
  - algoritmi de parcurgere (DFS, BFS);
  - determinarea componentelor (tare) conexe.

Grafuri

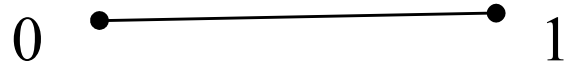
- G = (V, E)
  - V mulţime de vârfuri
  - E mulţime de muchii; o muchie = o pereche neordonată de vârfuri distincte



$V = \{0, 1, 2, 3\}$

$E = \{\{0,1\}, \{0,2\}, \{1,2\}, \{2, 3\}\}$

$u = \{0,1\} = \{1, 0\}$



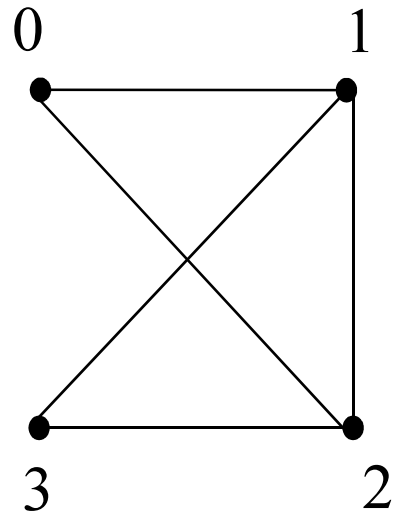
0, 1 – extremităţile lui u

u este incidentă în 0 şi 1

0 şi 1 sunt adiacente (vecine)

Grafuri

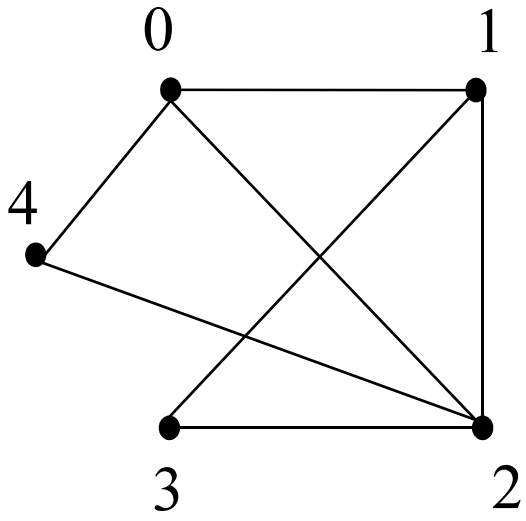
- Mers de la u la v: u = i_0, {i_0,i_1}, i_1,..., {i_{k-1},i_k}, i_k = v.
  3, {3,2}, 2, {2,0}, 0, {0,1}, 1, {1,3},3,{3,2},2
- parcurs: mers în care oricare două muchii sunt distincte.
- drum: mers în care oricare două vârfuri sunt distincte
- mers închis: $i_0 = i_k$
- circuit = mers închis în care oricare două vârfuri intermediare sunt distincte.



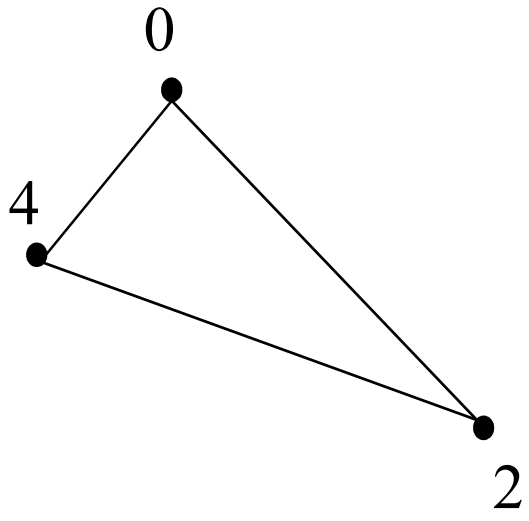
Subgraf indus

- G = (V, E) – graf
- W – submulțime a lui V
- Subgraf indus de W: G'(W,E'), unde E'={{i,j} | {i,j}∈E și i∈W, j∈W}

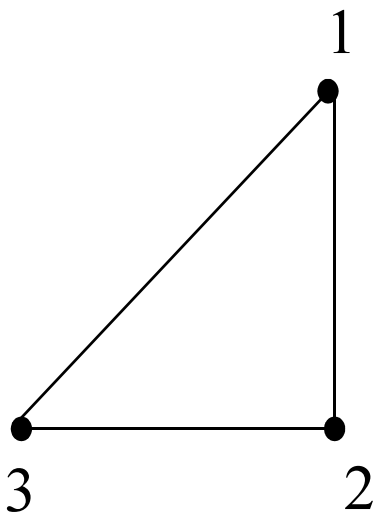
V = {0,1,2,3,4}



W = {0,2,4}

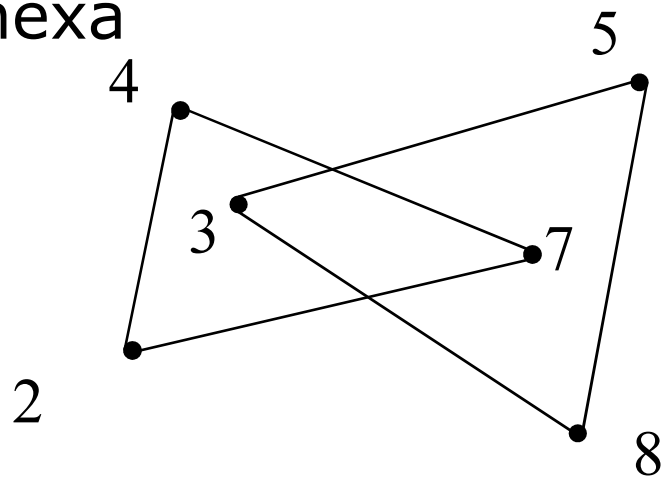


W = {1,2,3}



Grafuri - Conexitate

- i R j dacă şi numai dacă există drum de la i la j
- R este relaţie de echivalenţă
- V_1, ..., V_p clasele de echivalenţă
- $G_i = (V_i, E_i)$ subgraful indus de V_i
- G_1, ..., G_p – componente conexe
- graf conex = graf cu o singură componentă conexă



$V_1 = \{2, 4, 7\}$

$E_1 = \{\{2, 4\}, \{4, 7\}, \{2, 7\}\}$

$V_2 = \{3, 5, 8\}$

$E_2 = \{\{3, 5\}, \{5, 8\}, \{8, 3\}\}$

Tipul de date abstract **`Graf`**

- obiecte:
  - grafuri G = (V, E), V = {0, 1, ..., n-1}
- operaţii:
  - grafVid()
    - intrare: nimic
    - ieşire: graful vid (∅, ∅)
  - esteGrafVid()
    - intrare: G = (V, E),
    - ieşire: **`true`** daca G =(∅, ∅), **`false`** în caz contrar
  - insereazaMuchie()
    - intrare: G = (V, E), i, j ∈V
    - ieşire: G = (V, E ∪ {i, j})
  - insereazaVarf()
    - intrare: G = (V, E), V = {0, 1, ..., n-1}
    - ieşire: G = (V', E), V' = {0, 1, ..., n-1, n}

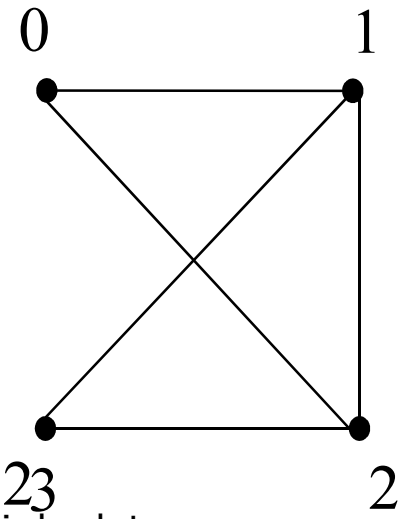
Tipul de date abstract **Graf**

- eliminaMuchie()
  - intrare: G = (V, E), i, j $\in$V
  - ieşire: G = (V, E – {i, j})
- eliminaVarf()
  - intrare: G = (V, E), V = {0, 1, ..., n-1}, k
  - ieşire: G = (V', E'), V' = {0, 1, ..., n-2}

$\{i', j'\} \in E' \Leftrightarrow (\exists \{i, j\} \in E)\ i \neq k, j \neq k$,

$i' =$ **if** $(i < k)$ **then** i **else** i-1,

$j' =$ **if** $(j < k)$ **then** j **else** j-1

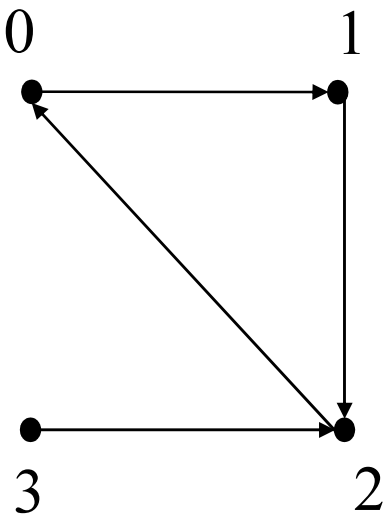


Tipul de date abstract **Graf**

- listaDeAdiacenta()
  - intrare: G = (V, E), i $\in$V
  - ieşire: lista vârfurilor adiacente cu i
- listaVarfurilorAccesibile()
  - intrare: G = (V, E), i $\in$V
  - ieşire: lista vârfurilor accesibile din i

Digraf (graf orientat)

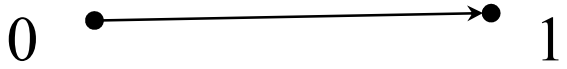
- D = (V, A)
  - V mulţime de vârfuri
  - A mulţime de arce; un arc = o pereche ordonată de vârfuri distincte



$V = \{0, 1, 2, 3\}$

$A = \{(0,1), (2,0), (1,2), (3, 2)\}$

$a = (0,1) \neq (1, 0)$

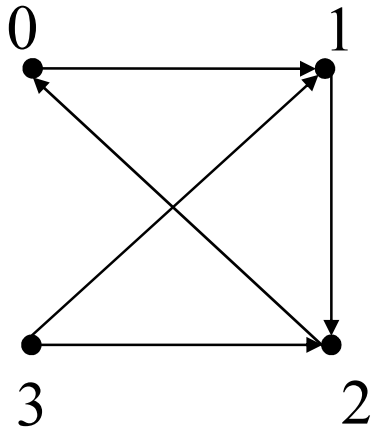


0 – sursa lui a

1 – destinația lui a

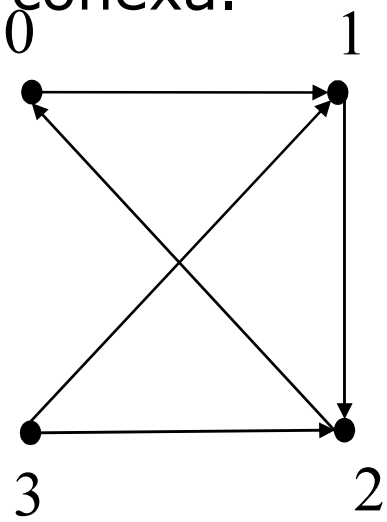
Digraf

- mers: i_0, (i_0,i_1), i_1, ..., (i_{k-1},i_k), i_k.
  3, (3,2), 2, (2,0), 0, (0,1), 1, (1,2), 2, (2,0), 0
- parcurs: mers în care oricare două arce sunt distincte.
- drum: mers în care oricare două vârfuri sunt distincte.
- mers închis: $i_0 = i_k$.
- circuit = mers închis în care oricare două vârfuri intermediare sunt distincte.



Digraf. Conexitate

- i R j dacă şi numai dacă există drum de la i la j şi drum de la j la i.
- R este relaţie de echivalenţă.
- V_1, ..., V_p clasele de echivalenţă.
- $G_i = (V_i, A_i)$ subdigraful indus de V_i.
- G_1, ..., G_p – componente tare conexe
- digraf tare conex = digraf cu o singură componentă tare conexă.



$V1 = \{0, 1, 2\}$

$A1 = \{(0, 1), (1, 2), (2, 0)\}$

$V2 = \{3\}$

$A2 = \emptyset$

Tipul de date abstract **`Digraf`**

- obiecte: digrafuri D = (V, A)
- operaţii:
  - digrafVid()
    - intrare: nimic
    - ieşire: digraful vid (∅, ∅)
  - esteDigrafVid()
    - intrare: D = (V, A),
    - ieşire: **`true`** dacă D =(∅, ∅), **`false`** în caz contrar
  - insereazaArc()
    - intrare: D = (V, A), i, j ∈V
    - ieşire: D = (V, A ∪ (i, j))
  - insereazaVarf()
    - intrare: D = (V, A), V = {0, 1, ..., n-1}
    - iesire: D = (V', A), V' = {0, 1, ..., n-1, n}

Tipul de date abstract **Digraf**

- eliminaArc()
  - intrare: D = (V, A), i, j $\in$V
  - ieşire: D = (V, A – (i, j))

- eliminaVarf()
  - intrare: D = (V, A), V = {0, 1, ..., n-1}, k
  - ieşire: D = (V', A'), V' = {0, 1, ..., n-2}

    (i', j') $\in$ A' $\Leftrightarrow$

    ($\exists$(i, j) $\in$ A) i $\neq$ k, j $\neq$ k ,

    i' = **if** (i < k) **then** i **else** i-1,

    j' = **if** (j < k) **then** j **else** j-1

Tipul de date abstract **Digraf**

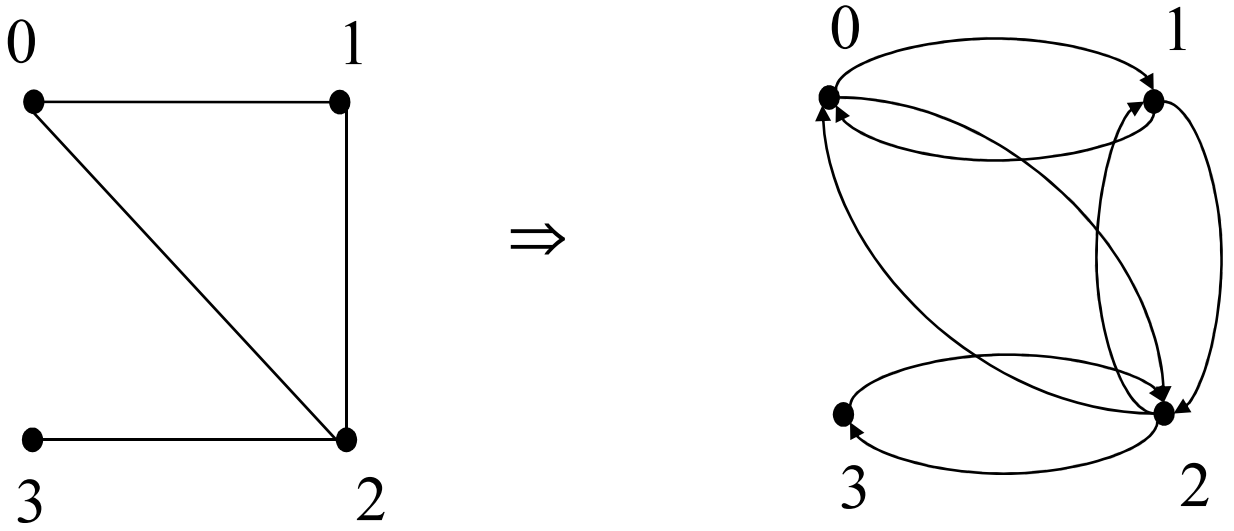
- listaDeAdiacentaExterioara()
  - intrare: D = (V, A), i $\in$V
  - ieşire: lista vârfurilor destinatare ale arcelor care pleacă din i
- listaDeAdiacentaInterioara()
  - intrare: D = (V, A), i $\in$V
  - ieşire: lista vârfurilor sursă ale arcelor care sosesc în i
- listaVarfurilorAccesibile()
  - intrare: D = (V, A), i $\in$V
  - ieşire: lista vârfurilor accesibile din i

Reprezentarea grafurilor ca digrafuri

$G = (V, E) \Rightarrow D(G) = (V, A)$

$\{i,j\} \in E \Rightarrow (i,j), (j,i) \in A$

- topologia este păstrată
  - lista de adiacenţă a lui i în G = lista de adiacenţă exterioară (=interioară) a lui i în D



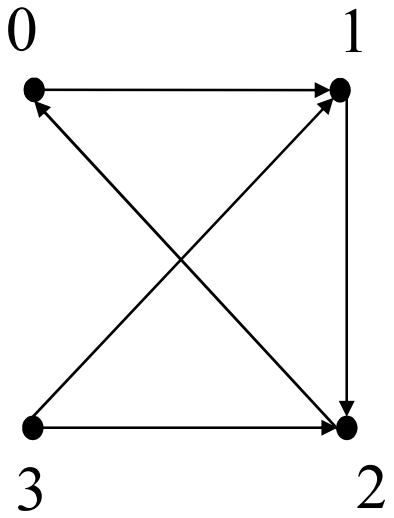
Implementarea cu matrici de adiacenţă a digrafurilor

- reprezentarea digrafurilor
  - **`n`** numărul de vârfuri
  - **`m`** numărul de arce (opţional)
  - o matrice (**`a[`**i,j**`]`**| 0 ≤ i, j < n)

    `a[`i,j**`] = if`** (i,j) ∈ A **`then 1 else 0`**
  - dacă digraful reprezintă un graf, atunci a[i,j] este simetrică
  - lista de adiacenţă exterioară a lui i ⊆ linia i
  - lista de adiacenţă interioară a lui i ⊆ coloana i

Implementarea cu matrici de adiacenţă

	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	1	1	0



Implementarea cu matrici de adiacenţă

- operaţii
  - digrafVid

    n ←0; m ← 0
  - insereazaVarf: O(n)
  - insereazaArc: O(1)
  - eliminaArc: O(1)

Implementarea cu matrici de adiacenţă

- eliminaVarf()

```
procedure eliminaVirf(a, n, k)
begin
   for i ← 0 to n-1 do
      for j ← 0 to n-1 do
         if (i > k) then a[i-1, j] ← a[i,j]
         if (j > k) then a[i, j-1] ← a[i,j]
   n ← n-1
end
```

- timp de execuţie: $O(n^2)$

Implementarea cu matrici de adiacenţă

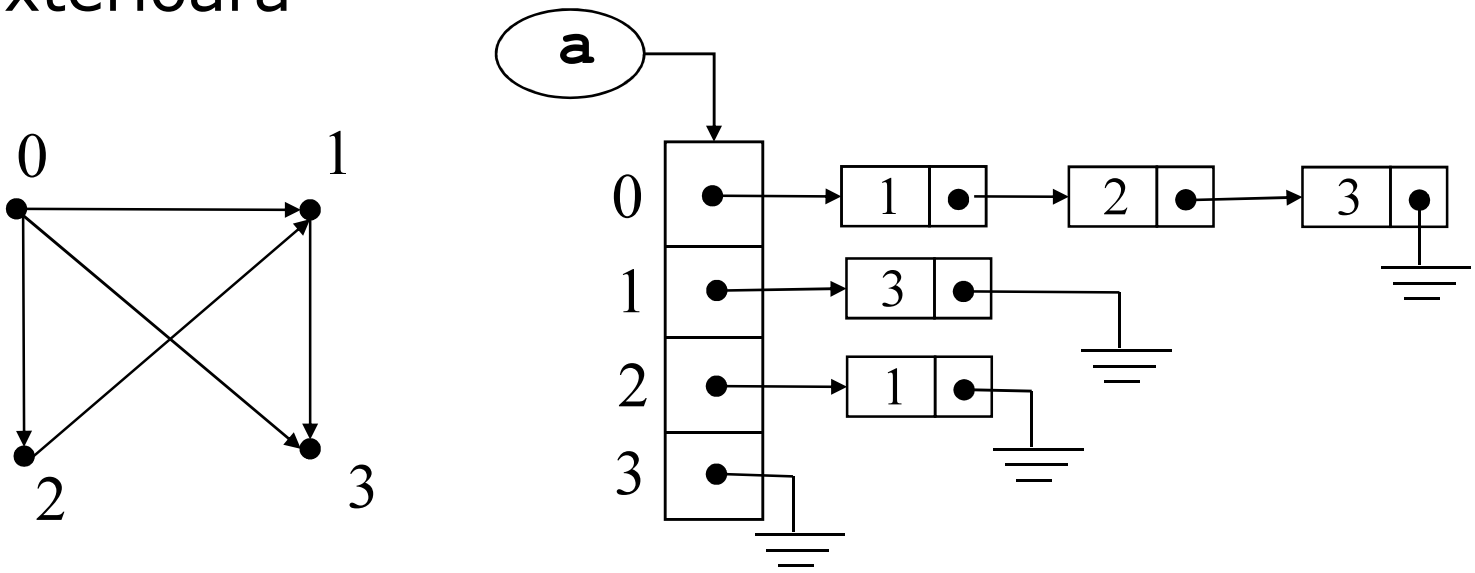
– listaVarfurilorAccesibile()

```
procedure inchReflTranz(a, n, b) // (Warshall, 1962)
  for i ← 0 to n-1 do
      for j ← 0 to n-1 do
         b[i,j] ← a[i,j]
         if (i == j) then b[i,j] ← 1
   for k ← 0 to n-1 do
      for i ← 0 to n-1 do
         if (b[i,k] == 1)
         then for j ← 0 to n-1 do
                 if (b[k,j] == 1)
                 then b[i,j] ← 1
end
```

- timp de executie: $O(n^3)$

Implementarea cu liste de adiacenţă

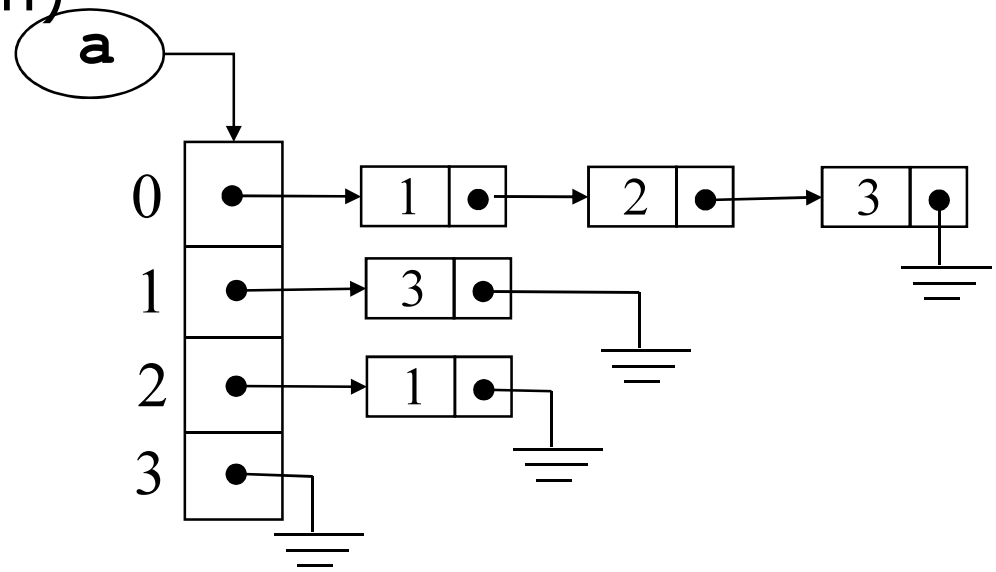
- reprezentarea digrafurilor cu liste de adiacenţă exterioară



- un tablou **`a[0..n-1]`** de liste înlănţuite (pointeri)
- **`a[i]`** este lista de adiacenţă exterioară corespunzătoare lui **`i`**

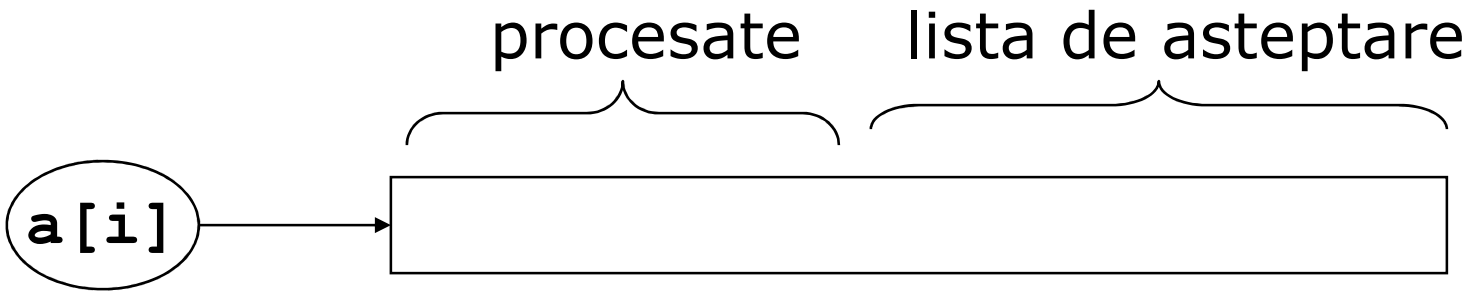
Implementarea cu liste de adiacenţă

- operaţii
  - digrafVid
  - insereazaVarf: O(1)
  - insereazaArc: O(1)
  - eliminaVarf: O(n+m)
  - eliminaArc: O(m)



Digrafuri: explorare sistematică

- se gestionează două mulţimi
  - S = mulţimea vârfurilor vizitate deja
  - SB $\subseteq$ S submulţimea vârfurilor pentru care există şanse să găsim vecini nevizitaţi încă
- lista de adiacenţă (exterioară) a lui i este divizată în două:



Digrafuri: explorare sistematică

- pasul curent
  - citeşte un vârf i din SB
  - extrage un j din lista de "aşteptare" a lui i (dacă este nevidă)
  - dacă j nu este în S, atunci îl adaugă la S şi la SB
  - dacă lista de "aşteptare" a lui i este vidă, atunci elimină i din SB
- iniţial
  - S = SB = $\{i_0\}$
  - lista de "aşteptare a lui i" = lista de adiacenta a lui i
- terminare SB = $\varnothing$

Digrafuri: explorare sistematică

```
procedure explorare(a, n, i0, S)
   for i ← 0 to n-1 do p[i] ← a[i]
   SB ← (i0); S ← (i0);
   viziteaza(i0)
   while (SB != ∅) do
      i ← citeste(SB)
      if (p[i] == NULL) then SB ← SB-{i}
      else j ← p[i]->varf
           p[i] ← p[i]->succ
           if (j ∉ S)
           then SB ← SB ∪ {j}
                S ← S ∪ {j}
                viziteaza(j)
end
```

Explorare sistematică: complexitate

Teoremă

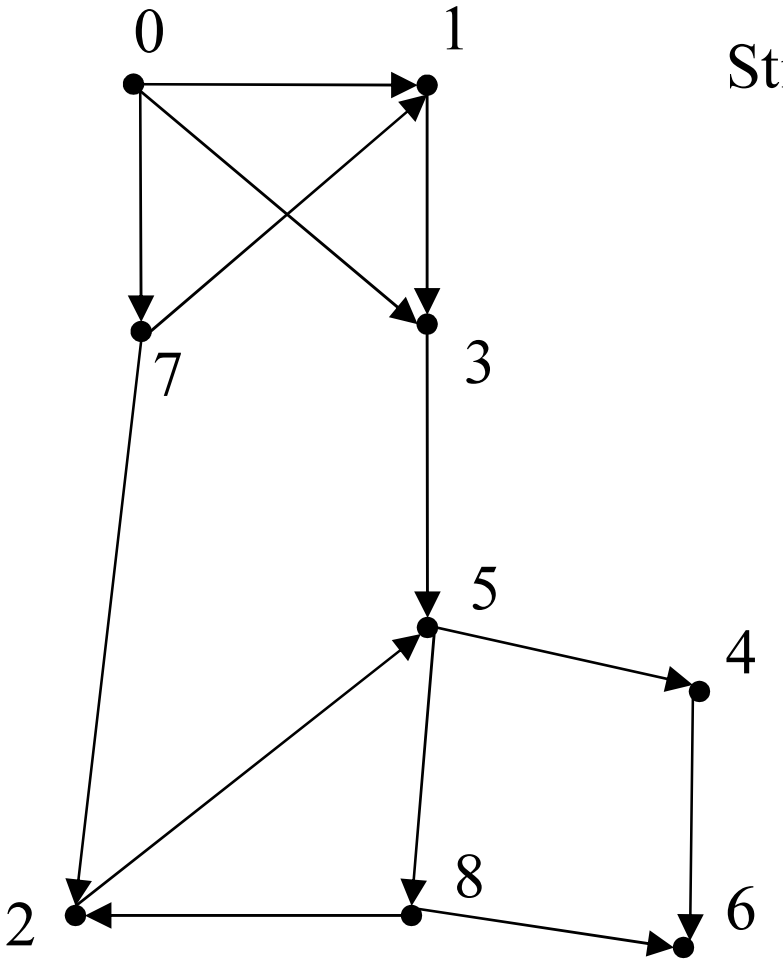
În ipoteza că operaţiile peste S şi SB precum şi **`viziteaza()`** se realizează în O(1), complexitatea timp, în cazul cel mai nefavorabil, a algoritmului **`explorare`** este O(n+m).

Explorarea DFS (Depth First Search)

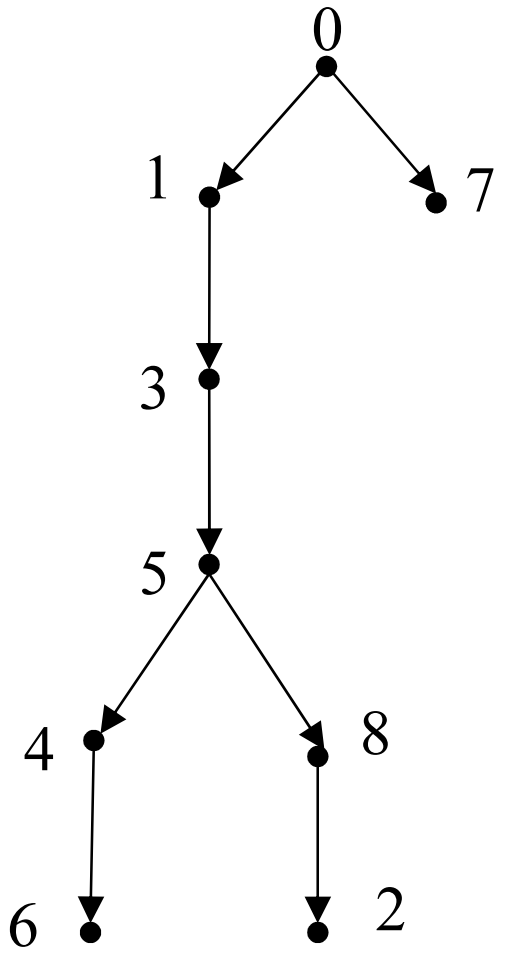
- SB este implementată ca stivă:

```
SB ← (i0) ⇔ SB ← stivaVida()
                 push(SB, i0)
i ← citeste(SB) ⇔ i ← top(SB)
SB ← SB-{i} ⇔ pop(SB)
SB ← SB ∪ {j} ⇔ push(SB, j)
```

Explorarea DFS: exemplu



Stiva



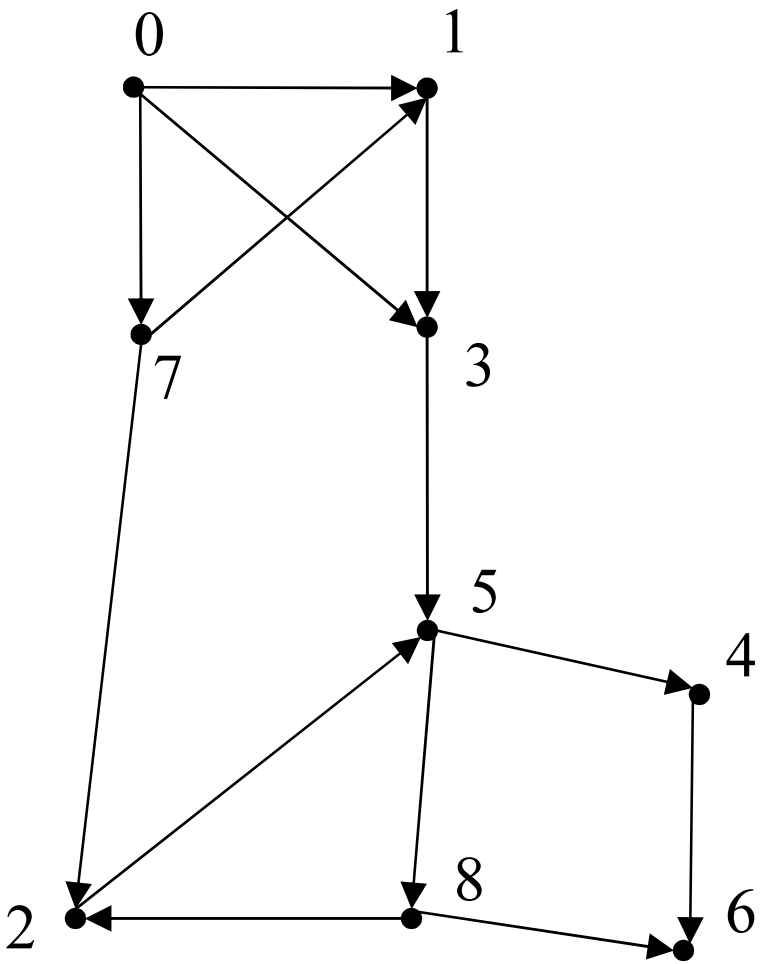
Arborele DFS

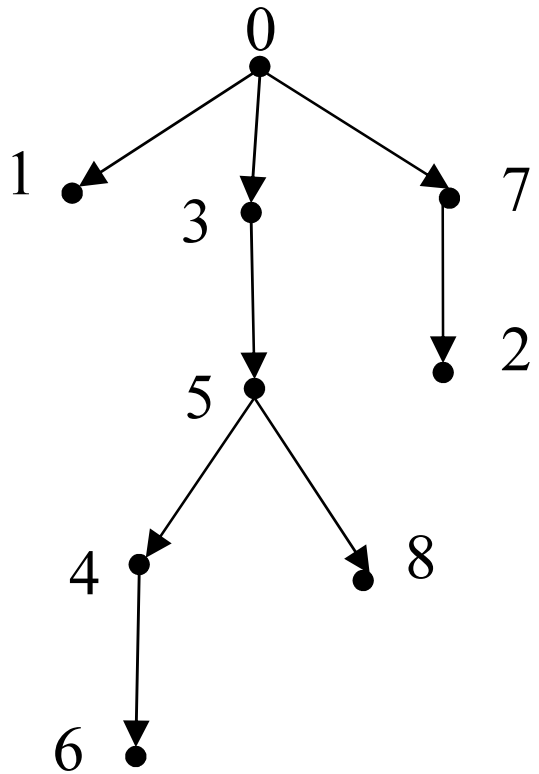
Explorarea BFS (Breadth First Search)

- SB este implementată ca o coadă

```
SB ← (i0) ⇔  SB ← coadaVida();
                  insereaza(SB, i0)
i ← citeste(SB) ⇔ citeste(SB, i)
SB ← SB-{i} ⇔ elimina(SB)
SB ← SB ∪ {j} ⇔ insereaza(SB, j)
```

Explorarea BFS: exemplu





Arborele BFS

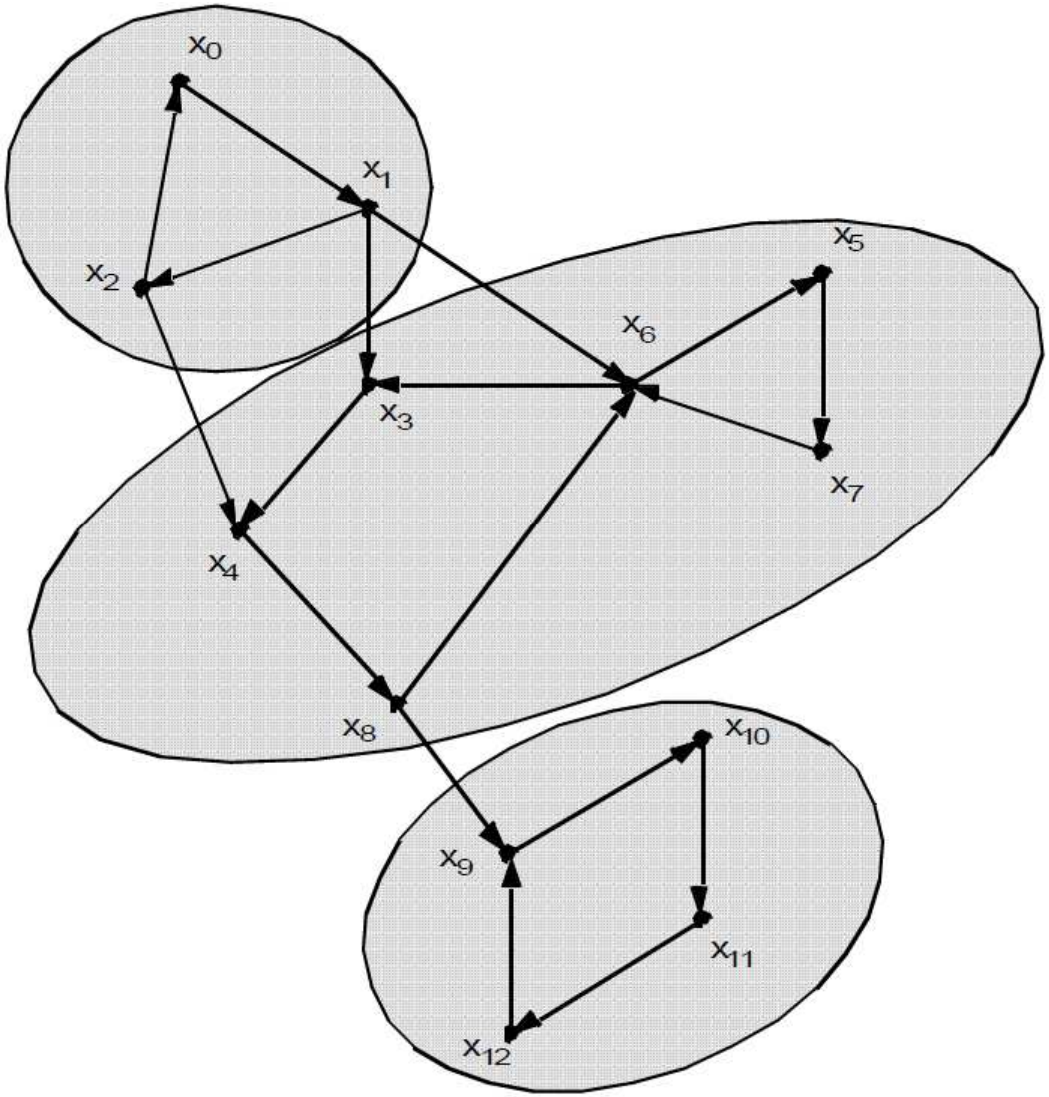
Determinarea componentelor conexe (grafuri neorientate)

```
function CompConexeDFS(G)
begin
   for i ← 0 to n-1 do culoare[i] ← 0
   k ← 0
   for i ← 0 to n-1 do
      if (culoare[i] == 0)
      then  k ← k+1
            DfsRecCompConexe(i, k)
   return k
end
```

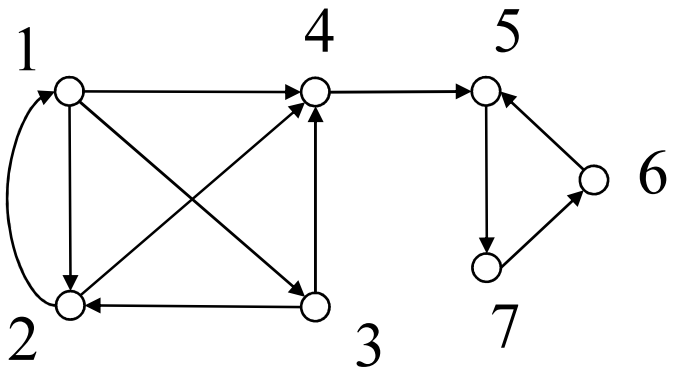
Determinarea componentelor conexe (grafuri neorientate)

```
procedure DfsRecCompConexe(i, k)
begin
   culoare[i] ← k
   for (fiecare vârf j în listaDeAdiac(i)) do
      if (culoare[j] == 0)
            then DfsRecCompConexe(j, k)
end
```

Componente tare conexe (digrafuri)

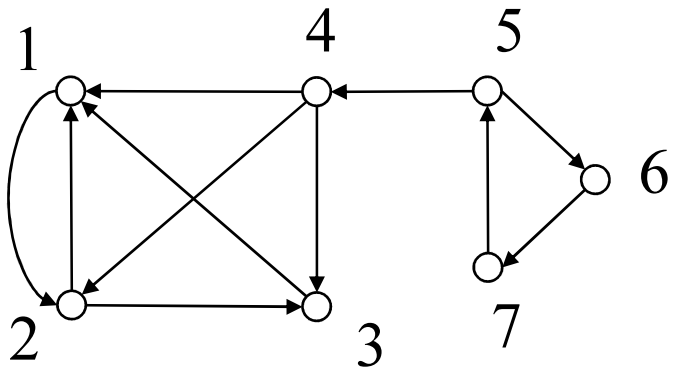


Componente tare conexe: exemplu

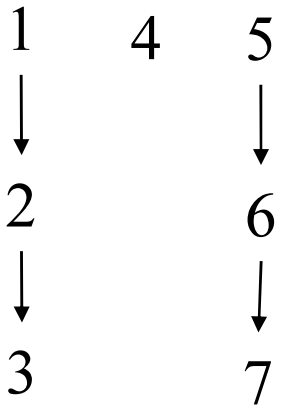


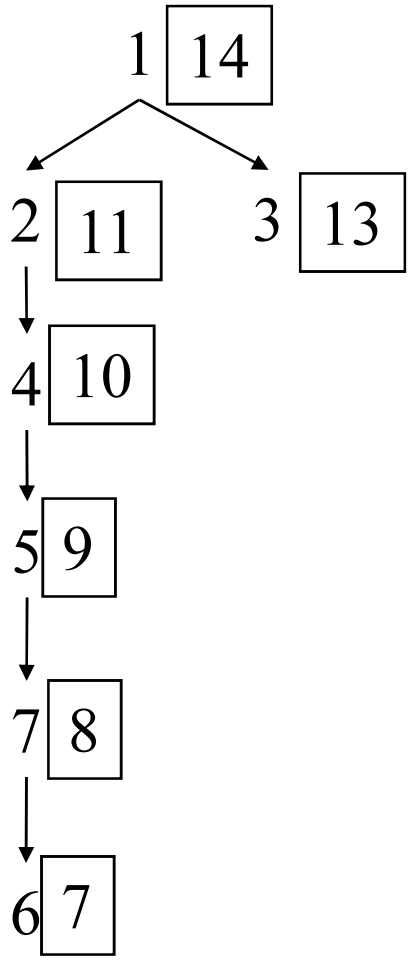
D

$1 \rightarrow (2, 3, 4)$
$2 \rightarrow (1, 4)$
$3 \rightarrow (2, 4)$
$4 \rightarrow (5)$
$5 \rightarrow (7)$
$6 \rightarrow (5)$
$7 \rightarrow (6)$



D^T





Determinarea componentelor tare conexe

```
procedure DfsCompTareConexe(D)
begin
   for i ← 0 to n-1 do
      culoare[i] ← 0
      tata[i] ← -1

   timp ← 0
   for i ← 0 to n-1 do
      if (culoare[i] == 0)
      then DfsRecCompTareConexe(i)
end
```

Determinarea componentelor tare conexe

```
procedure DfsRecCompTareConexe(i)
begin
    timp ← timp + 1
   culoare[i] ← 1
   for (fiecare virf j in listaDeAdiac(i)) do
      if (culoare[j] == 0)
      then tata[j] ← i
           DfsRecCompTareConexe(j)
   timp ← timp + 1
   timpFinal[i] ← timp
end
```

Determinarea componentelor tare conexe

Notatie:

$D^T = (V, A^T)$, $(i, j) \in A \Leftrightarrow (j, i) \in A^T$

procedure CompTareConexe(D)

begin

1. **DFSCompTareConexe(D)**
2. calculeaza **D^T**
3. **DFSCompTareConexe(D^T)** dar considerând în bucla **for** principală vârfurile în ordinea descrescătoare a timpilor finali de vizitare **timpFinal[i]**
4. returnează fiecare arbore calculat la pasul 3 ca fiind o componentă tare conexă separată

end

Determinarea componentelor tare conexe: complexitate

- **`DFSCompTareConexe(D)`**: O(n + m)
- calculeaza **D^T**: O(m)
- **`DFSCompTareConexe(`D^T`)`**: O(n + m)
- Total: O(n + m)