



GRAFURI

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare



Introducere

- Grafurile reprezintă unele dintre cele mai utile structuri de date
- Grafurile sunt structuri de date asemănătoare cu arborii
- Arborii reprezintă un caz particular de grafuri



Introducere

- Grafurile se utilizează într-un mod diferit față de arbori
- Structurile de date discutate anterior au o arhitectură determinată de algoritmi care le prelucrează
- De exemplu, un arbore binar are o formă adecvată unei căutări facile a datelor memorate și a unei inserări simple a elementelor noi



Introducere

- Grafurile au o formă determinată de o problemă concretă
- De exemplu, nodurile unui graf pot reprezenta orașe, iar arcele pot fi curse aeriene între acele orașe

Definiții

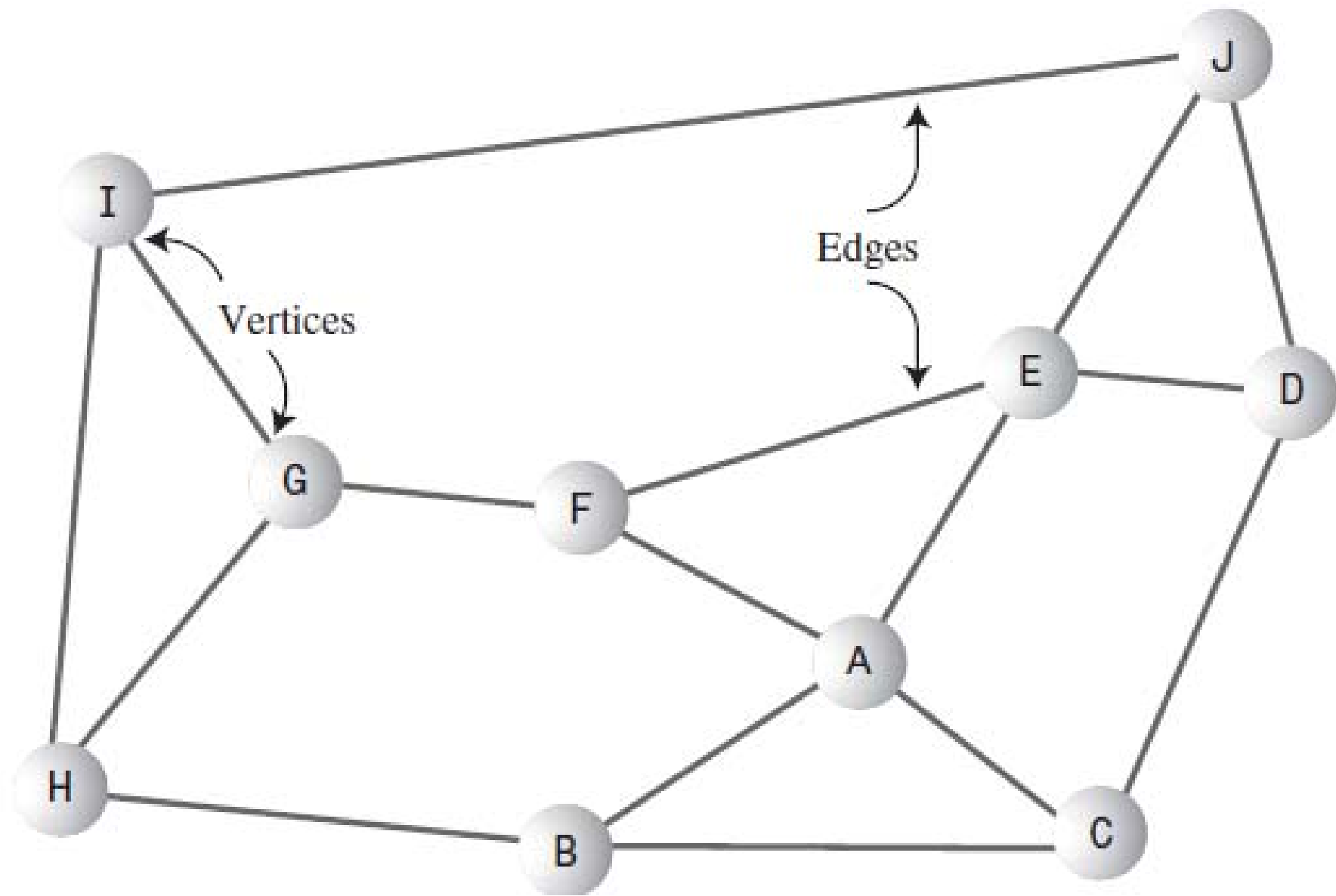
- Nodurile se numesc **vârfuri**
- Arcele se numesc **muchii**
- Două vârfuri se numesc **adiacente** dacă sunt conectate direct printr-o muchie
- Vârfurile adiacente cu un vârf se numesc **vecinii** vârfului dat
- Un **drum** reprezintă o secvență de muchii

Definiții

- Un graf este un model abstract (matematic) pentru multe probleme reale, concrete, a căror rezolvare necesită folosirea unui calculator
- În matematică, un graf este definit ca o pereche de două mulțimi $\mathbf{G} = (\mathbf{V}, \mathbf{M})$, unde \mathbf{V} este mulțimea (nevidă) a vârfurilor (nodurilor), iar \mathbf{M} este mulțimea muchiilor (arcelor)
- O muchie din \mathbf{M} unește o pereche de două vârfuri din \mathbf{V} și se notează (\mathbf{v}, \mathbf{w})

Definiții

- Nodurile unui graf se numerează începând cu **1** și mulțimea **V** este o submulțime a numerelor naturale
- Termenii “**vârf**” și “**muchie**” provin din analogia unui graf cu un poliedru și se folosesc mai ales pentru **grafuri neorientate**
- Termenii “**nod**” și “**arc**” se folosesc mai ales pentru **grafuri orientate**

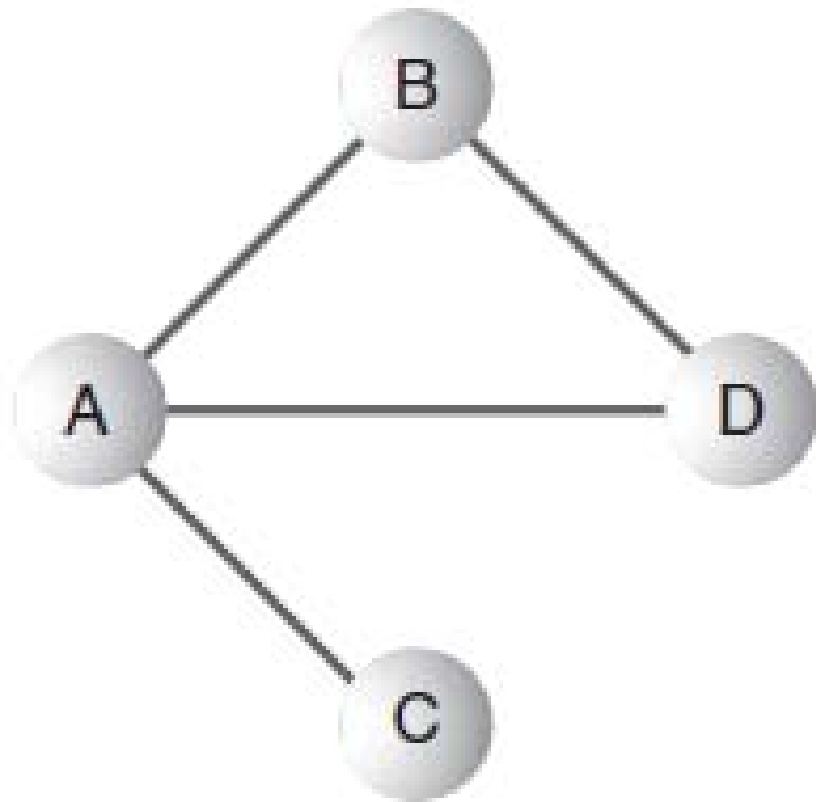


Observații

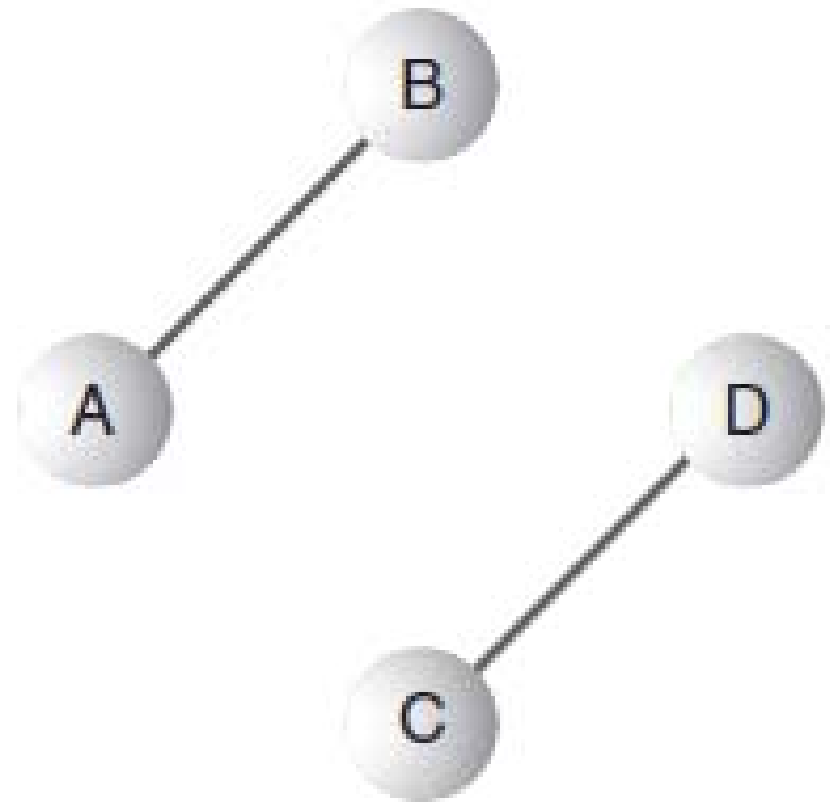
- Un **drum** (o **cale**) într-un graf unește o serie de noduri $v[1], v[2], \dots, v[n]$ printr-o secvență de arce $(v[1], v[2]), (v[2], v[3]), \dots$
- Între două noduri date poate să nu existe un arc, dar să existe o cale, ce trece prin alte noduri intermediare

Grafuri conexe și neconexe

- Un graf se numește **conex** dacă există cel puțin un drum de la fiecare nod până la fiecare alt nod
- Un graf este **conex** dacă, pentru orice pereche de noduri (v,w) , există cel puțin o cale de la v la w sau de la w la v
- Un graf neconex este alcătuit din mai multe **componente conexe**



a) Connected Graph



b) Non-connected Graph

Observații

- Vârfurile A și B alcătuiesc una din componentele conexe, iar C și D alcătuiesc a doua componentă conexă
- Grafurile prezentate sunt **grafuri neorientate**
- Aceasta înseamnă că muchiile nu au o **direcție**, ne putem deplasa pe ele în orice sens

Grafuri orientate

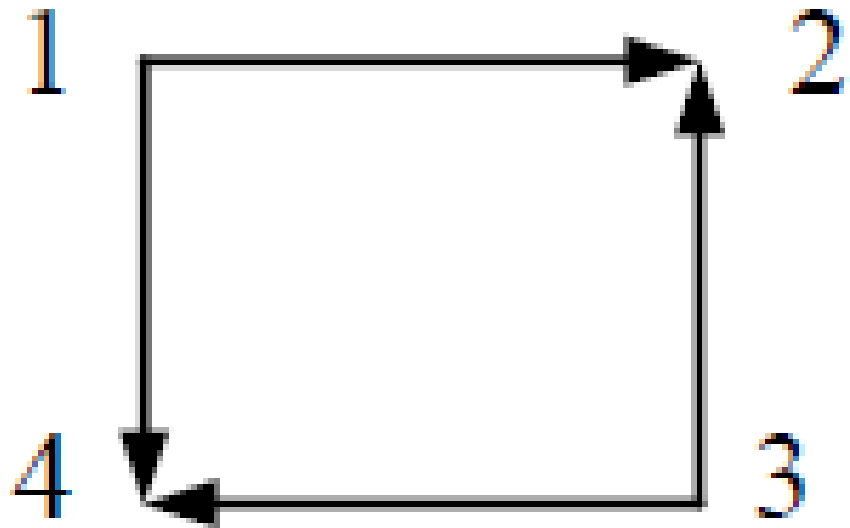
- Un graf în care ne putem deplasa într-o singură direcție de-a lungul unei muchii date se numește **graf orientat**
- Direcția de deplasare permisă este indicată printr-o săgeată plasată pe muchie

Componentă conexă

- O **componentă conexă** a unui graf (V, M) este un subgraf conex (V', M') , unde V' este o submulțime a lui V , iar M' este o submulțime a lui M
- Împărțirea unui **graf neorientat** în componente conexe este **unică**, dar un **graf orientat** poate fi partiționat în mai multe moduri în componente conexe

Exemplu

- Graful $(1,2), (1,4), (3,2), (3,4)$ poate avea componentele conexe:
 - $\{1,2,4\}$ și $\{3\}$
 - $\{3,2,4\}$ și $\{1\}$



Graf orientat tare conex

- Un graf orientat este **tare conex (puternic conectat)** dacă, pentru orice pereche de noduri (v,w) , există (cel puțin) o cale de la v la w și (cel puțin) o cale de la w la v
- Exemplu de graf tare conex - graf care conține un ciclu ce trece prin toate nodurile:
- $(1,2), (2,3), (3,4), (4,1)$

Observații

- Într-un graf orientat, numit și digraf (prescurtare din limba engleză de la “directed graph”), arcul (v,w) pleacă din nodul v și intră în nodul w
- Acesta este diferit de arcul (w,v) , care pleacă de la w la v
- Într-un graf neorientat poate exista o singură muchie între două vârfuri, notată (v,w) sau (w,v)

Observații

- Două noduri între care există un arc se numesc și noduri **vecine** sau **adiacente**
- Într-un graf orientat ne putem referi la **succesorii** și **predecesorii** unui nod, respectiv la arce care **ies** și la arce care **intră** într-un nod

Grafuri ponderate

- În unele grafuri, muchiile au asociate **ponderi**, adică numere care reprezintă distanțe fizice între două vârfuri, sau timpul necesar parcurgerii drumului dintre două vârfuri, sau costul drumului dintre cele două vârfuri
- Astfel de grafuri se numesc **grafuri ponderate**



Definiții

- Un **ciclu** într-un graf (un **circuit**) este o cale care pornește și se termină în același nod
- Un **ciclu hamiltonian** este un ciclu complet, care unește toate nodurile dintr-un graf



Definiții

- Un graf neorientat conex este **ciclic** dacă numărul de muchii este mai mare sau egal cu numărul de vârfuri
- Un **arbore liber** este un graf conex fără cicluri



Tipul de date abstract graf

- Un graf poate fi privit ca un tip de date abstract, care permite orice relații între componentele structurii
- Operațiile uzuale asociate tipului “graf” sunt:

Tipul de date abstract graf

- 1) Inițializare graf cu număr dat de noduri:
 - `initG (Graf & g, int n);`
- 2) Adăugare muchie (arc) la un graf:
 - `addArc (Graf & g, int x, int y);`
- 3) Verificare existența unui arc de la un nod x la un nod y:
 - `int arc(Graf g, int x, int y);`
- 4) Eliminare arc dintr-un graf:
 - `delArc (Graf & g, int x, int y);`
- 5) Eliminare nod dintr-un graf:
 - `delNod (Graf & g, int x);`

Observații

- Mai mulți algoritmi pe grafuri necesită parcurgerea vecinilor (succesorilor) unui nod dat, care poate folosi funcția “arc” într-un ciclu repetat pentru toți vecinii posibili (pentru toate nodurile din graf)
- Pentru grafuri reprezentate prin liste de vecini, este suficientă parcurgerea listei de vecini a unui nod (egală cu numărul de arce asociate acelui nod), mult mai mică decât numărul de noduri din graf

Vârfuri

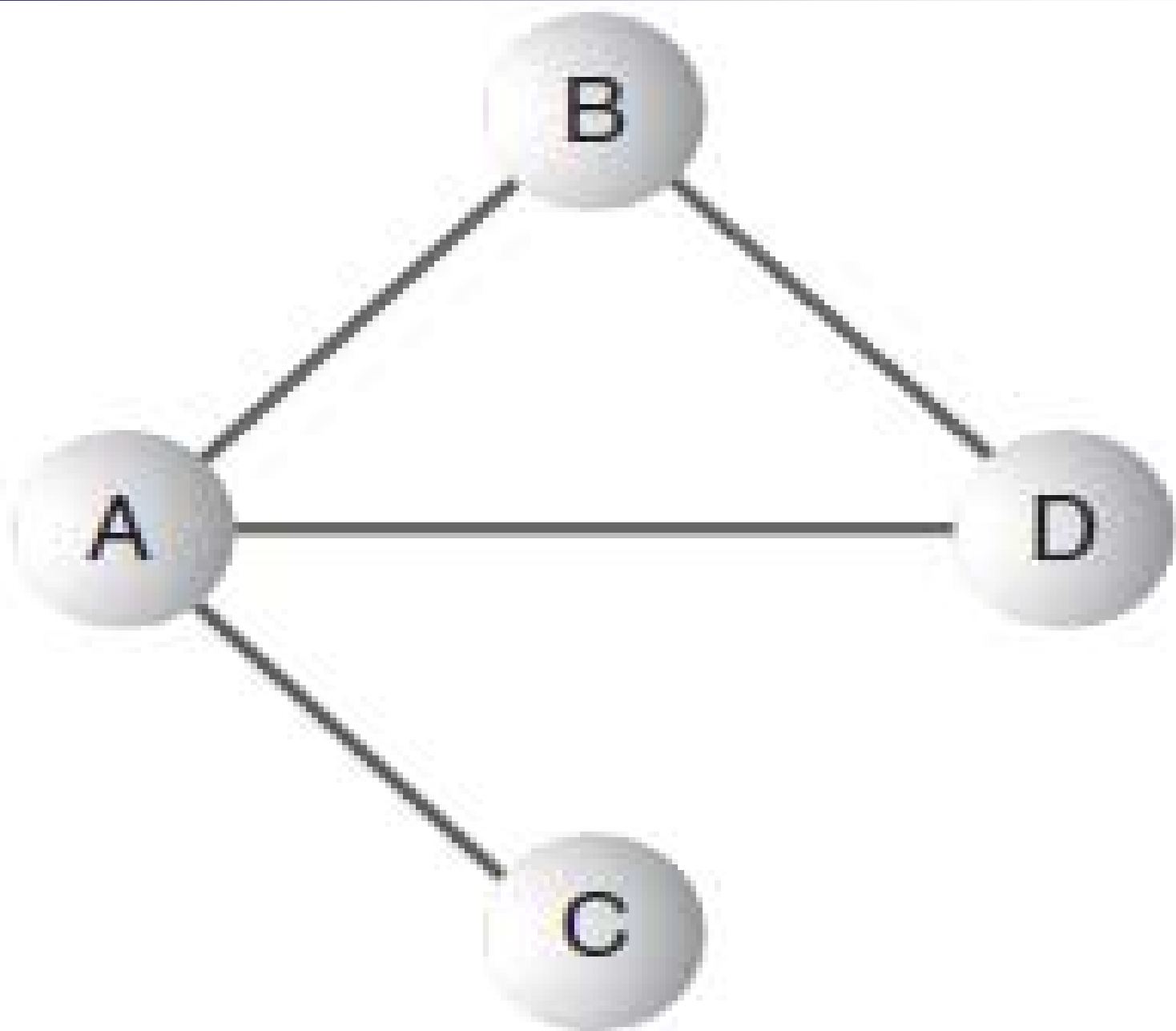
- Vârfurile sunt numerotate cu indici cuprinși între **1** și **N**, unde **N** este numărul lor
- Vârfurile pot fi memorate într-un tablou și referite utilizând indicele fiecăruia
- În locul tabloului, putem utiliza o listă înlănțuită sau o altă structură
- Indiferent de structura utilizată, aceasta este numai o convenție, care nu afectează modul în care vârfurile sunt conectate prin muchii


Muchii

- Într-un graf, fiecare vârf poate fi conectat cu un număr arbitrar de alte vârfuri
- Pentru modelarea conectării nodurilor, cele mai frecvente două metode sunt:
 - **Matricea de adiacență**
 - **Lista de adiacență**

Matricea de adiacență

- O matrice de adiacență este un tablou bidimensional, în care elementele indică prezența unei muchii între două vârfuri
- Dacă graful are N vârfuri, matricea de adiacență este un tablou cu $N \times N$ elemente

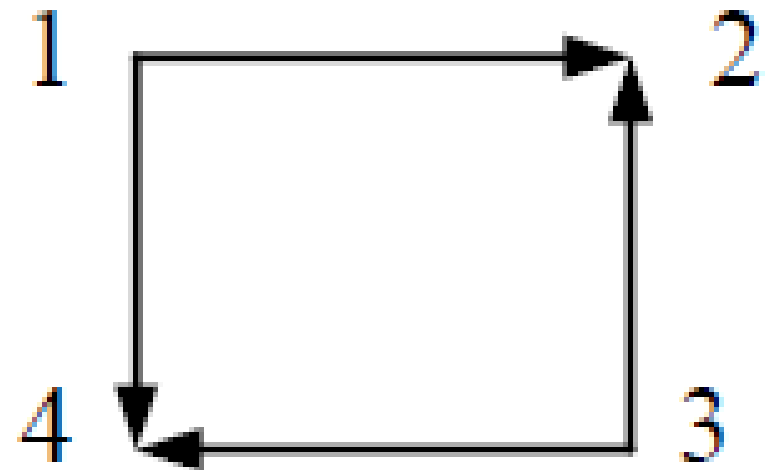




| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 |
| B | 1 | 0 | 0 | 1 |
| C | 1 | 0 | 0 | 0 |
| D | 1 | 1 | 0 | 0 |

Exemplu

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |



Observații

- Vârfurile sunt trecute atât la capetele liniilor, cât și ale coloanelor din tabel
- O muchie între două vârfuri este indicată printr-o valoare 1 în tabel
- Valoarea 0 înseamnă absența unei muchii
- Legătura dintre un vârf și el însuși este reprezentată prin 0, formând **diagonala principală** a matricei

Observații

- Triunghiul superior al matricei (situat deasupra diagonalei principale) reprezintă imaginea în oglindă a celui inferior
- Ambele triunghiuri conțin aceeași informație
- Această redundanță pare inefficientă, dar, în majoritatea limbajelor de programare, nu dispunem de o modalitate convenabilă de a crea un tablou triunghiular

Observații

- Reprezentarea matricială este preferată în determinarea drumurilor dintre oricare două vârfuri (tot sub formă de matrice), în determinarea drumurilor minime dintre oricare două vârfuri dintr-un graf ponderat, în determinarea componentelor conexe ale unui graf orientat (prin **transpunerea matricei** se obține graful cu arce inversate, numit și **graf dual** al grafului inițial), și în alte aplicații cu grafuri

Observații

- Dezavantajul matricei de adiacențe apare atunci când numărul de noduri din graf este mult mai mare ca numărul de arce, iar matricea este **rară** (cu peste jumătate din elemente nule)
- În aceste cazuri se preferă reprezentarea prin liste de adiacențe

Definirea structurii de graf

- Definirea structurii de graf printr-o matrice de adiacență alocată dinamic:
- `typedef struct {`
- `int n, m ;`
- `// n = număr de noduri, m = număr de arce`
- `int ** a; // adresa matrice de adiacență`
- `} Graf ;`

Observații

- **Succesorii** unui nod v sunt reprezentați de elementele nenule din **linia** v
- **Predecesorii** unui nod v sunt reprezentați de elementele nenule din **coloana** v
- În general, nu există arce de la un nod la el însuși, deci $a[i][i] = 0$

Funcții cu grafuri

```
// funcție de inițializare a grafului
void initG (Graf & g, int n) {
    int i;
    g.n = n;
    g.m = 0;
    g.a = (int**) malloc( (n+1)*sizeof(int*));
    // vârfuri numerotate 1...n
    for (i = 1; i <= n; i++)
        g.a[i] = (int*) calloc( (n+1), sizeof(int));
    // linia 0 și coloana 0 sunt nefolosite
}
```



Funcții cu grafuri

```
// funcție de adăugare a arcului (x,y) la graful g  
void addArc (Graf & g, int x, int y) {  
g.a[x][y]=1;  
g.m++;  
}
```

```
// funcție care întoarce arcul (x,y) din graful g  
int arc (Graf g, int x, int y) {  
return g.a[x][y];  
}
```

```
// funcție de eliminare a arcului (x,y) din graful g
void delArc (Graf & g, int x, int y) {
    g.a[x][y] = 0;
    g.m--;
}
```

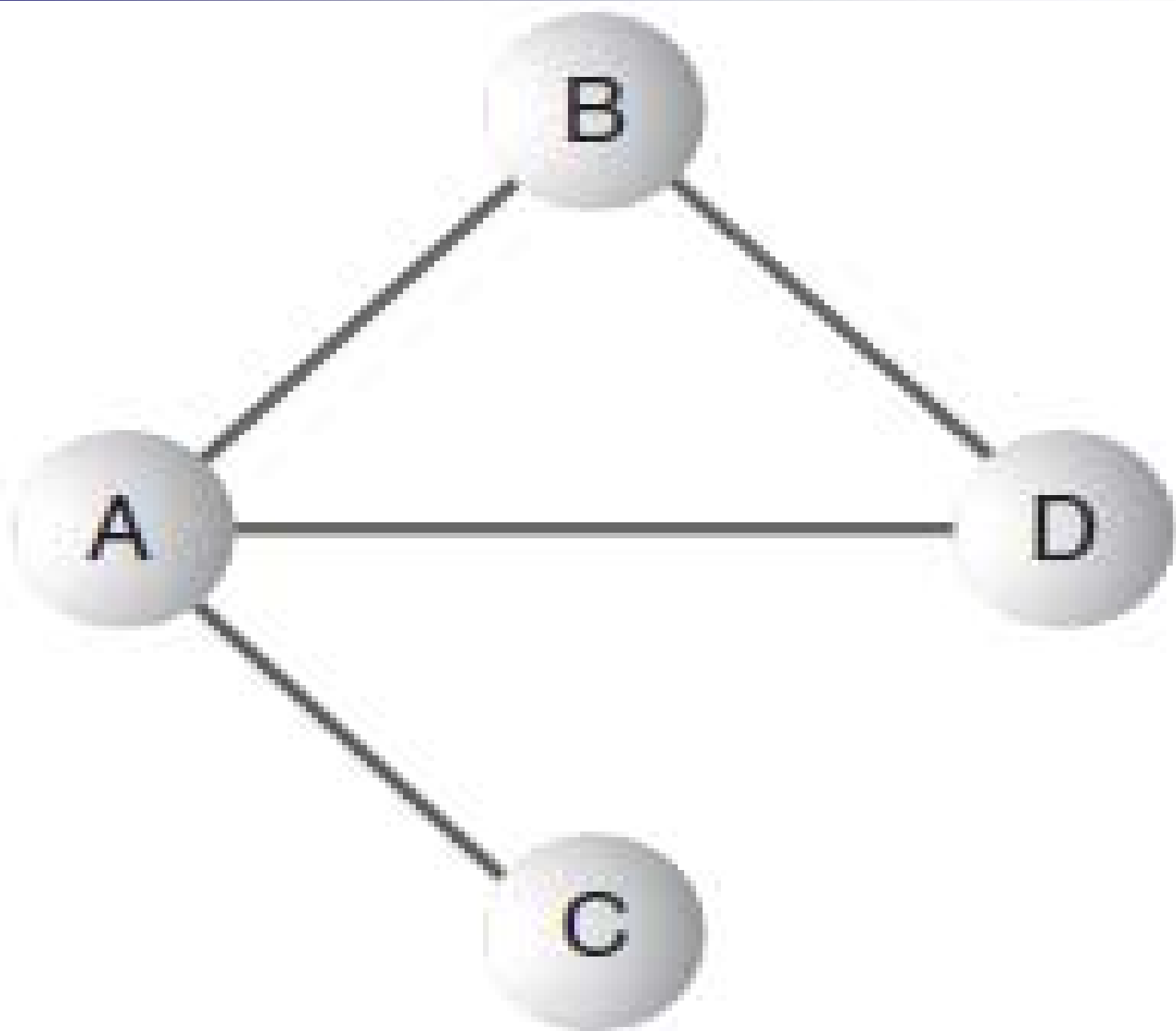
/*Eliminarea unui nod din graf ar trebui să modifice și dimensiunile matricei, dar se elimină doar arcele ce pleacă și vin din/în acel nod*/


```
// funcție de eliminare a nodului x din graful g
void delNod (Graf & g, int x) {
    int i;
    for (i = 1; i <= g.n; i++) {
        delArc(g,x,i);
        delArc(g,i,x);
    }
}
```



Lista de adiacență

- Cealaltă soluție de reprezentare a muchiilor este utilizând o **listă de adiacență**
- O listă de adiacență reprezintă un tablou de liste (sau o listă de liste)
- Fiecare listă individuală conține vârfurile adiacente unui vârf dat





| Vertex | List Containing Adjacent Vertices |
|--------|-----------------------------------|
| A | B—>C—>D |
| B | A—>D |
| C | A |
| D | A—>B |



Observații

- Fiecare legătură din listă reprezintă un vârf din graf
- În exemplu, vârfurile sunt ordonate alfabetic, deși această ordine nu este necesară
- Nu trebuie confundat conținutul listelor de adiacență cu drumurile din graf

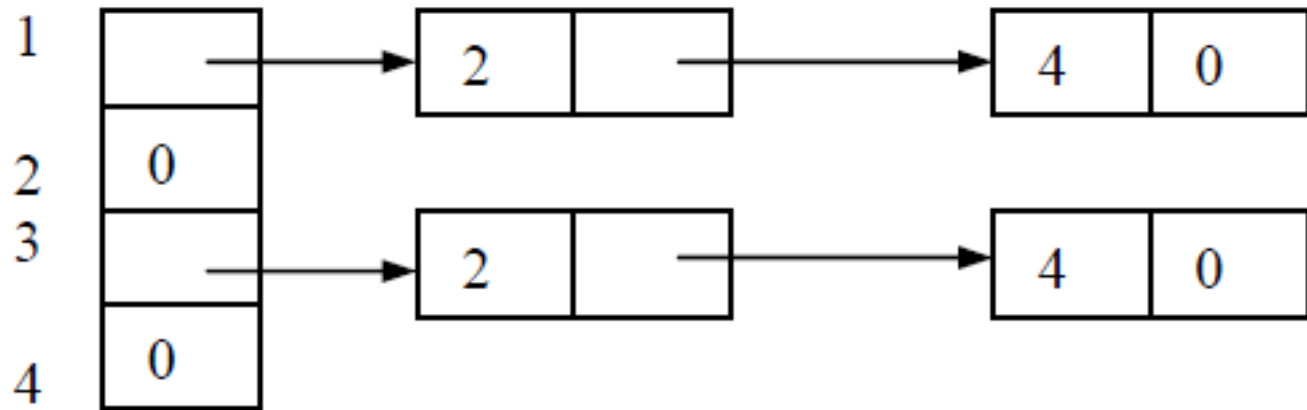
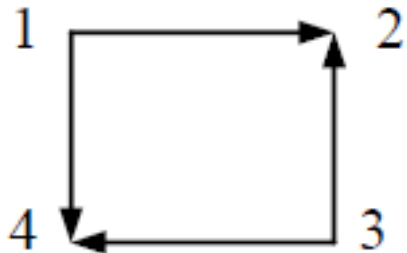


Observații

- Lista tuturor arcelor din graf este împărțită în mai multe subliste, câte una pentru fiecare nod din graf
- Listele de noduri vecine pot avea lungimi diferite și se preferă implementarea lor prin liste înlanțuite

Exemplu

- Reprezentarea grafului
(1,2),(1,4),(3,2),(3,4) printr-un tablou de pointeri la liste de adiacență





Observații

- Ordinea nodurilor într-o listă de adiacență nu este importantă și de aceea se poate adăuga mereu la începutul listei de noduri vecine

Definirea structurii de graf

```
■ typedef struct nod {  
■     int val;                // număr nod  
■     struct nod * leg;  
■     // adresa listei de succesori pentru un nod  
■ } * pnod ;                  // pnod este un tip pointer  
■ typedef struct {  
■     int n ;                  // număr de noduri în graf  
■     pnod * v;  
■     // tablou de pointeri la liste de succesori  
■ } Graf;
```



Funcții cu grafuri

```
// funcție de inițializare a grafului  
void initG (Graf & g, int n) {  
    g.n = n;    // număr de noduri  
    g→v = (pnod*) calloc(n + 1, sizeof(pnod));  
    // inițializare pointeri cu 0  
}
```


Funcții cu grafuri

```
// funcție de adăugare a arcului (x,y)
void addArc (Graf & g, int x, int y) {
    pnod nou = (pnod) malloc (sizeof(nod));
    nou→val = y;
    nou→leg = g→v[x];
    g→v[x] = nou;
    // se adaugă la începutul listei de adiacență
}
```

Funcții cu grafuri

```
// funcție care testează dacă există arcul (x,y) în graful g
int arc (Graf g, int x, int y) {
    pnod p;
    for (p = g→v[x]; p != NULL; p = p→leg)
        if ( y == p→val) return 1;
    return 0;
}
```

Observații

- Reprezentarea unui graf prin liste de adiacență pentru fiecare vârf asigură cel mai bun timp de explorare a grafurilor (timp proporțional cu suma dintre numărul de vârfuri și numărul de muchii din graf), iar explorarea apare ca operație în mai mulți algoritmi pe grafuri

Observații

- Pentru un graf neorientat, fiecare muchie (x,y) este memorată de două ori: y în lista de adiacență a lui x și x în lista de adiacență a lui y
- Pentru un graf orientat, listele de adiacență sunt de obicei liste de succesori, dar pentru unele aplicații ne interesează predecesorii unui nod
- Lipsa de simetrie poate fi un dezavantaj al listelor de adiacență pentru reprezentarea grafurilor orientate



Arbore liber

- Un **arbore liber** este un graf neorientat aciclic
- Într-un arbore liber nu există un nod special rădăcină
- Într-un arbore, fiecare vârf are un singur părinte (predecesor), deci se poate reprezenta arborele printr-un tablou de noduri părinte



Observații

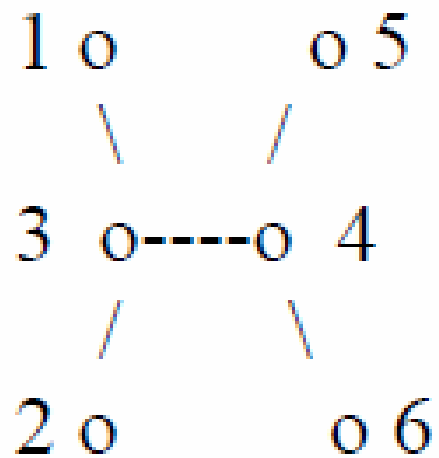
- Rezultatul mai multor algoritmi este un arbore liber și acesta se poate reprezenta compact printr-un singur tablou
- Exemple: arbori de acoperire de cost minim, arborele cu drumurile minime de la un nod la toate celelalte noduri (Dijkstra)



Observații

- Un graf conex se poate reprezenta printr-o singură listă - lista arcelor, iar numărul de noduri este valoarea maximă a unui nod prezent în lista de arce (toate nodurile din graf apar în lista de arce)
- Lista arcelor poate fi un tablou sau o listă de structuri, sau două tablouri de noduri

Exemplu



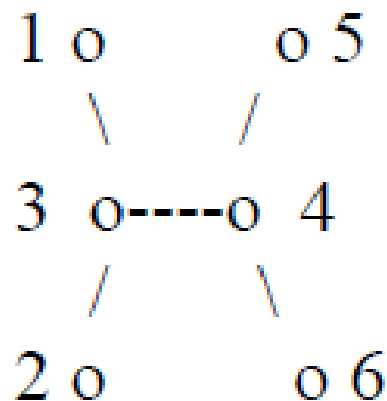
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| x | 1 | 2 | 3 | 4 | 4 |
| y | 3 | 3 | 4 | 5 | 6 |

Observații

- Pentru arbori liberi, această reprezentare poate fi simplificată și mai mult, dacă se impune ca poziția în tablou să fie egală cu unul dintre noduri
- Se folosește un singur tablou P , în care $P[k]$ este perechea (predecesorul) nodului k
- Este posibil să se noteze arcele din arbore astfel încât fiecare nod să aibă un singur predecesor (sau un singur succesor)

Exemplu

- Tabloul P este:

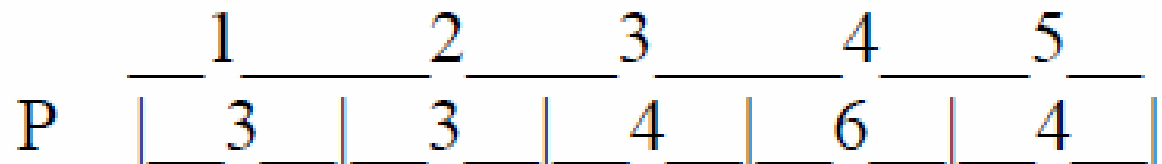
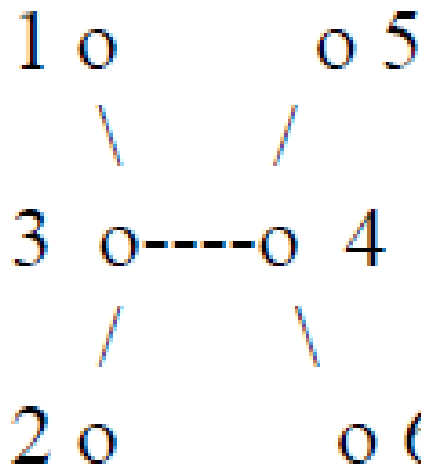


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| P | | 3 | 1 | 3 | 4 | 4 |

- Lista arcelor $(k, P[k])$ este:
 $(2,3), (3,1), (4,3), (5,4), (6,4)$

Observații

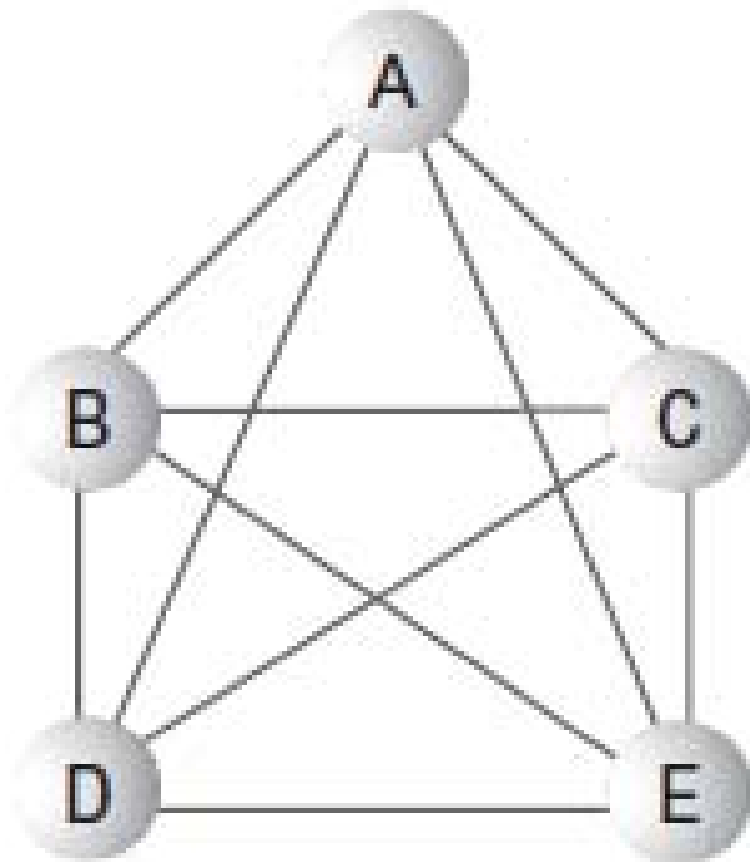
- Se consideră că nodul 1 nu are niciun predecesor, dar se poate considera că ultimul nod nu are niciun predecesor:



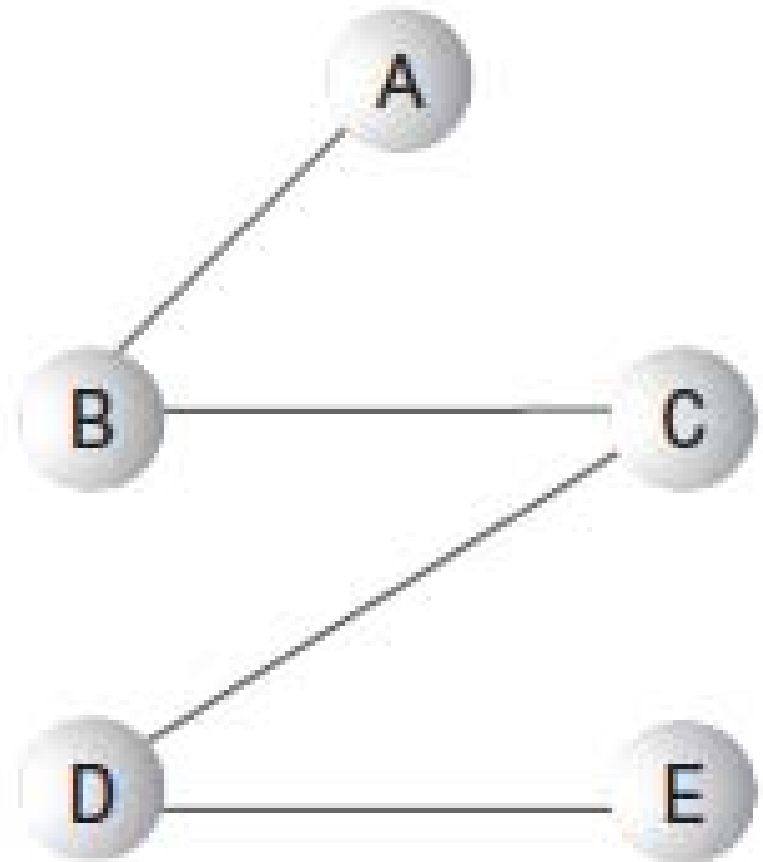


Arbori minimi de acoperire

- Un arbore minim de acoperire reprezintă un arbore cu numărul minim de muchii necesare pentru conectarea tuturor vârfurilor



a) Extra Edges



b) Minimum Number of Edges

Observații

- În prima figură, avem cinci vârfuri conectate printr-un număr excesiv de muchii
- Aceleași vârfuri sunt conectate în a doua figură printr-un număr minim de muchii
- Acesta reprezintă un **arbore minim de acoperire**

Observații

- Pentru o mulțime dată de vârfuri, este posibilă construcția mai multor arbori minimi de acoperire
- Arborele din a doua figură conține muchiile AB, BC, CD și DE
- Un alt arbore este cel care conține muchiile AC, CE, ED și DB

Arbori minimi de acoperire

- Pentru un graf conex neorientat $G = (V, E)$, se numește **arbore minim de acoperire** al lui G , un **subgraf** $G' = (V, E')$, care conține toate vârfurile grafului G și o submulțime **minimă** de muchii $E' \subseteq E$, cu proprietatea că unește toate vârfurile și nu conține cicluri
- Cum G' este conex și aciclic, el este arbore

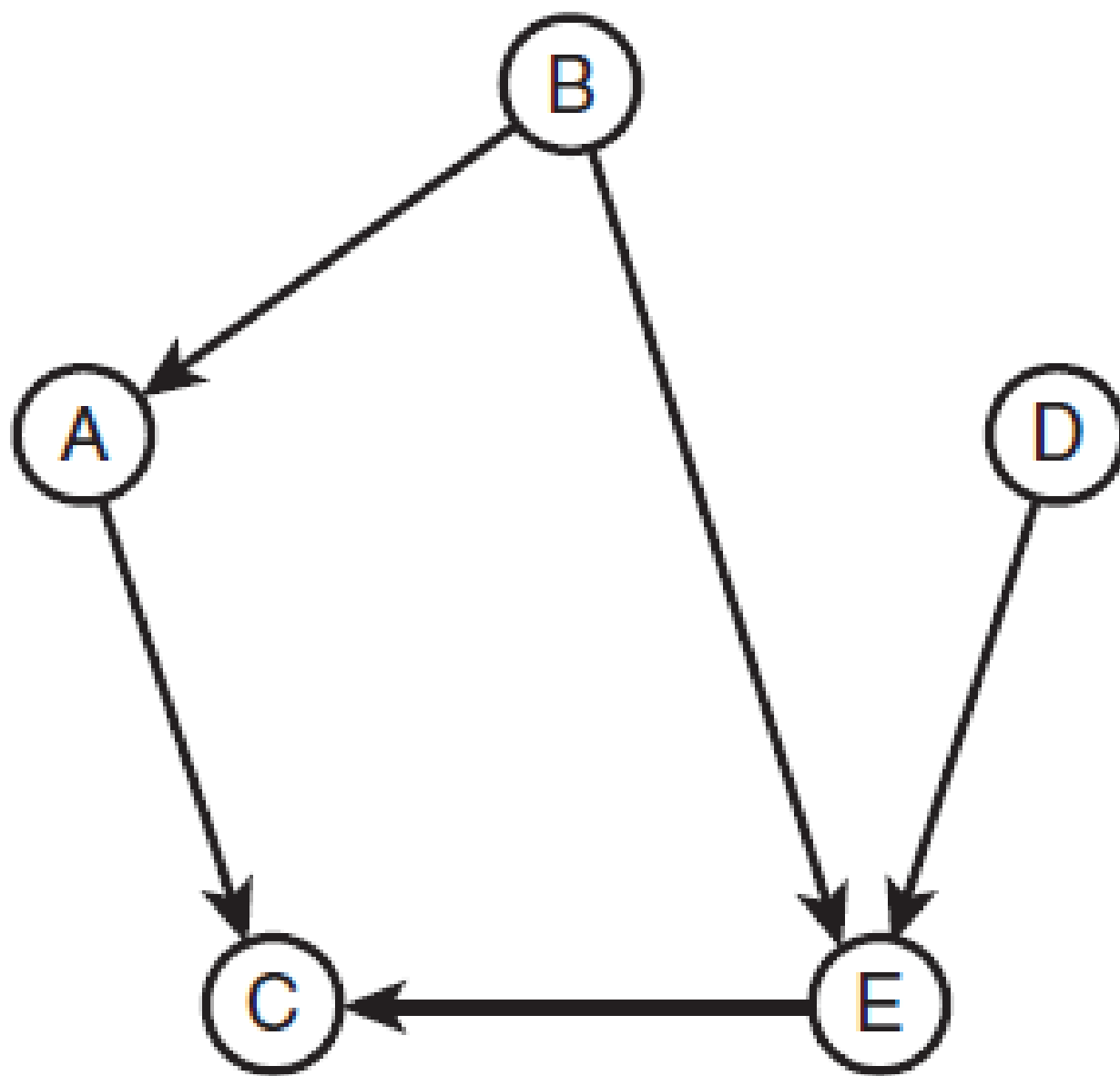
Observații

- Numărul **E** de muchii dintr-un arbore minim de acoperire este cu o unitate mai mic decât numărul **V** de vârfuri
- **$E = V - 1$**
- Executând o **parcursere în adâncime** și **notând muchiile traversate**, se obține un **arbore minim de acoperire**



Conectivitate în grafuri orientate

- Într-un graf neorientat, se pot găsi toate vârfurile care sunt conectate, utilizând parcurgerea în adâncime sau parcurgerea pe nivel
- Dacă se dorește găsirea tuturor vârfurilor conectate într-un graf orientat, nu se poate porni dintr-un vârf selectat aleator, pentru a ajunge la toate celelalte vârfuri conectate



Exemplu

- Dacă se pornește din A, se poate ajunge la C, dar nu și la alte vârfuri
- Dacă se pornește din B, nu se poate ajunge la D
- Dacă se pornește din C, nu se poate ajunge la niciun alt vârf
- La ce vârfuri se poate ajunge dacă se pornește dintr-un vârf anume ?

Observații

- Se poate modifica explorarea în adâncime, pentru a începe explorarea pe rând, din fiecare vârf
- Pentru graful anterior, se obține rezultatul:
 - ☐ AC
 - ☐ BACE
 - ☐ C
 - ☐ DEC
 - ☐ EC



Observații

- Prima literă indică vârful de pornire, iar literele următoare arată vârfurile la care se ajunge (fie direct, fie trecând prin alte vârfuri), pornind din vârful de start




Algoritmul lui Warshall

- Algoritmul află dacă **se poate ajunge într-un vârf**, pornind din **oricare alt vârf**
- Se creează un tabel care va indica dacă se poate ajunge într-un vârf, pornind din oricare alt vârf
- Acest tabel se obține prin modificarea matricei de adiacență a grafului

Algoritmul lui Warshall

- Graful reprezentat de această matrice de adiacență revizuită reprezintă **închiderea tranzitivă** a grafului inițial
- În matricea de adiacență, pentru o anumită muchie, linia indică vârful de început al muchiei, iar coloana vârful de sfârșit



| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 0 |
| B | 1 | 0 | 0 | 0 | 1 |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 1 | 0 | 0 |

Observații

- Se poate folosi algoritmul lui Warshall pentru a transforma matricea de adiacență în închiderea tranzitivă a grafului
- Algoritmul se bazează pe următoarea idee: Dacă se poate ajunge de la vârful **L** la vârful **M**, precum și de la vârful **M** la vârful **N**, atunci se poate ajunge de la vârful **L** la vârful **N**

Observații

- Matricea de adiacență arată toate căile posibile cu un singur pas, deci poate fi folosită pentru a obține căi cu doi pași
- Algoritmul construiește căi de lungime arbitrară, bazate pe căi cu mai multe muchii, descoperite anterior



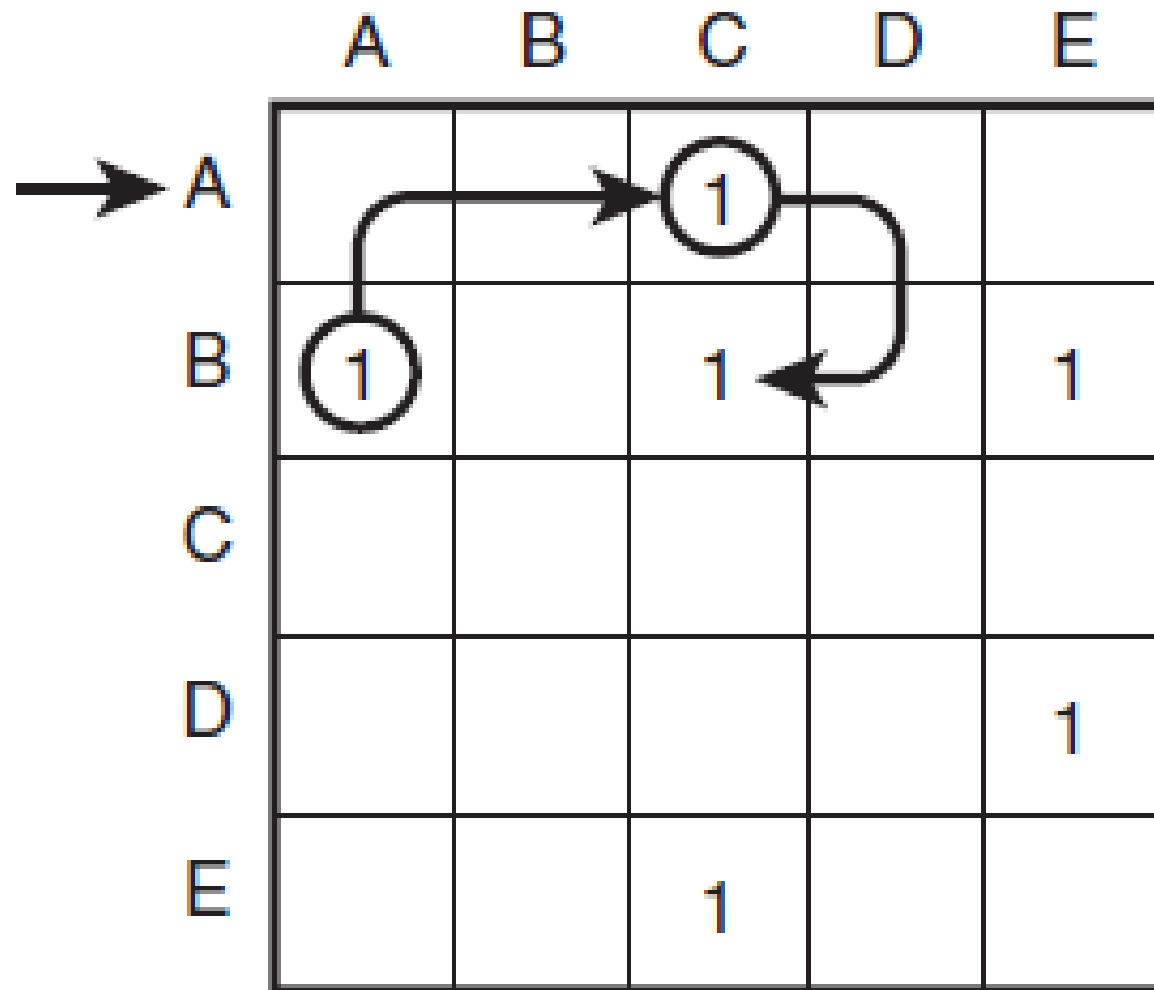
Linia A

- Există 1 pe coloana C, care arată că există o cale de la A la C
- Muchiile posibile care se termină în A se află în coloana A
- Se examinează toate elementele din coloana A
- Se observă că există o muchie de la B la A

Observații

- Există o muchie de la B la A și altă muchie de la A la C
- Se poate deduce că se poate ajunge de la B la C în doi pași
- Se pune 1 la intersecția liniei B cu coloana C

a) $y = 0$



A to C and B to A
so B to C

Linia B

- Se ajunge la linia B
- Prima celulă din coloana A indică existența unei muchii de la B la A
- Există muchii care se termină în B ?
- Deoarece coloana B conține doar 0-uri, se observă că nicio valoare 1 din linia B nu va duce la obținerea unei căi mai lungi, deoarece nicio muchie nu se termină în B

Liniiile C și D

- Linia C nu conține nicio valoare 1
- Se ajunge la linia D, unde se află o muchie de la D la E
- Deoarece coloana D conține numai 0-uri, nu există muchii care se termină în D

Linia E

- În linia E există o muchie de la E la C
- Din coloana E se observă că există o muchie de la B la E
- Se poate deduce că există o cale de la B la C, deoarece există muchii de la B la E și de la E la C
- Această cale a fost deja descoperită anterior

Observații

- Există 1 în coloana E, corespunzător liniei D
- Muchia de la D la E și de la E la C formează calea de la D la C
- Se adaugă două valori 1 la matricea de adiacență, care arată nodurile la care se poate ajunge dintr-un alt nod, într-un anumit număr de pași

Implementarea algoritmului

- Algoritmul folosește trei bucle imbricate
- Bucla exterioară parcurge fiecare linie
- Numim **y** variabila asociată
- Bucla interioară parcurge fiecare celulă de pe linie, folosind variabila **x**
- Dacă se găsește **1** în celula (**y,x**), atunci există o muchie de la **y** la **x**

Implementarea algoritmului

- Se activează a treia buclă, cea mai interioară, care are asociată variabila **z**
- A treia buclă examinează celulele din coloana **y**, căutând o muchie care se termină în **y**
- **y** este folosit pentru linii în prima buclă și pentru coloane în a treia buclă

Implementarea algoritmului

- Dacă există 1 la intersecția dintre coloana y și linia z , atunci există o muchie de la z la y
- Cu o muchie de la z la y și altă muchie de la y la x , se obține calea de la z la x
- Se pune 1 în celula (x,z)

Implementarea algoritmului

- Se consideră o matrice **A** cu valori logice (**0** sau **1**) și se aplică, la fiecare pas **k**, transformarea următoare:
- $A_k[i,j] = A_{k-1}[i,j] \vee (A_{k-1}[i,k] \wedge A_{k-1}[k,j])$
- Formula arată că există o cale între **i** și **j**, trecând prin vârfuri având numărul mai mic sau egal cu **k**, dacă:

Implementarea algoritmului

- a) Există o cale între i și j , trecând prin vârfuri numerotate cel mult $k-1$, sau
- b) Există o cale între i și k , trecând prin vârfuri numerotate cel mult $k-1$ și o cale între k și j similară
- Doarece $A_k[i,k] = A_{k-1}[i,k]$ și $A_k[k,j] = A_{k-1}[k,j]$, determinarea închiderii tranzitive se poate realiza utilizând o singură copie a lui A

Pseudocod algoritmul lui Warshall

```
AlgoritmWarshall() {  
    pentru toate liniile i execută  
        pentru toate coloanele j execută  
            dacă există un drum de la i la j atunci  $A[i,j] \leftarrow 1$   
                altfel  $A[i,j] \leftarrow 0$   
        pentru k de la 1 la n execută  
            pentru toate liniile i execută  
                pentru toate coloanele j execută  
                    dacă  $(A[i,j] == 0)$  atunci  
                         $A[i,j] \leftarrow A[i,k] \wedge A[k,j]$   
}
```