PROIECTAREA ALGORITMILOR

Lect. univ. dr. Adrian Runceanu

Curs 7

Elemente de teoria grafurilor (partea II)

Conţinutul cursului

6.1. Definiţii

6.2. Memorarea(reprezentarea) grafurilor

6.3. Parcurgerea grafurilor

- **6.3.1. Parcurgerea în lăţime (algoritmul BF)**
- **6.3.2. Parcurgerea în adâncime (algoritmul DF)**

6.4. Grafuri hamiltoniene şi grafuri euleriene

6.5. Aplicaţii ale grafurilor neorientate

6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.3. Parcurgerea grafurilor

- Prin *parcurgerea unui graf neorientat* se înţelege *examinarea în mod sistematic a nodurilor sale, plecând dintr-un vârf dat i, astfel încât fiecare nod accesibil din i pe muchii adiacente două câte două, să fie atins o singură dată*.
- Trecerea de la un nod x la altul se face prin explorarea, într-o anumită ordine, a vecinilor lui x, adică a vârfurilor cu care nodul x curent este adiacent.
- Această acţiune este numită şi **vizitare** sau **traversare** a vârfurilor grafului, scopul acestei vizitări fiind acela de prelucrare a informaţiei asociată nodurilor.
- Graful este o structură neliniară de organizare a datelor iar rolul traversării sale poate fi şi *determinarea unei aranjări lineare a nodurilor* în vederea trecerii de la unul la altul.

6.3. Parcurgerea grafurilor

Există două tipuri de parcurgere:

1. **Parcurgerea în lăţime (*Breadth First*)**

2. **Parcurgerea în adâncime (*Depth First*)**

Conţinutul cursului

6.1. Definiţii

6.2. Memorarea(reprezentarea) grafurilor

6.3. Parcurgerea grafurilor

6.3.1. Parcurgerea în lăţime (algoritmul BF)

6.3.2. Parcurgerea în adâncime (algoritmul DF)

6.4. Grafuri hamiltoniene şi grafuri euleriene

6.5. Aplicaţii ale grafurilor neorientate

6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.3.1. Parcurgerea în lăţime (algoritmul BF)

Prin algoritmul *BF* se realizeaza o parcurgere a grafului „*în lăţime*“ :

Se vizitează un vârf iniţial s, apoi vecinii săi (vârfurile adiacente cu s), după aceea vecinii vecinilor lui s (nevizitaţi încă), etc.

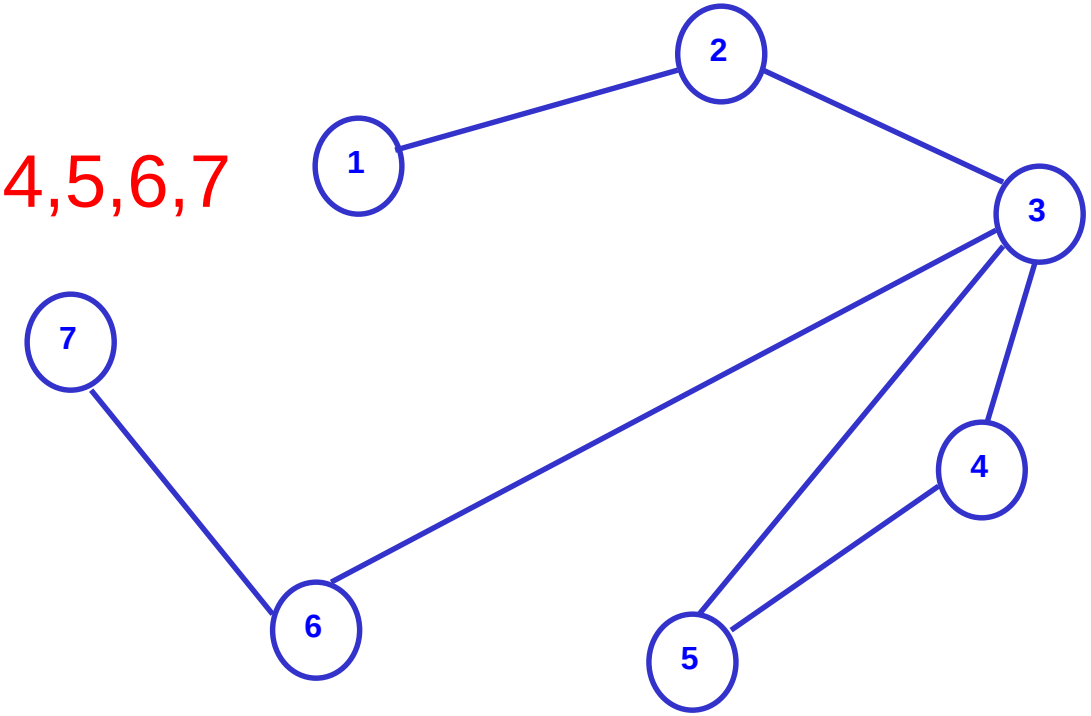
Observatie:

Dacă graful nu este conex nu se pot vizita toate vârfurile.

6.3.1. Parcurgerea în lăţime (algoritmul BF)

Exemplu
Pentru graful următor, pornind de la varful initial 1:

se obtine: 1,2,3,4,5,6,7



6.3.1. Parcurgerea în lăţime (algoritmul BF)

In constructia algoritmului *BF* trebuie, ca in fiecare moment, sa se poata face distinctie intre varfurile „vizitate“ si cele „nevizitate inca“.

De aceea se vor utiliza:

a) un tablou unidimensional *VIZ[ ]* cu *n* componente, definite astfel:

$$VIZ[k] = \begin{cases} 1 & \text{daca varful k a fost vizitat} \\ 0 & \text{daca varful k nu a fost vizitat} \end{cases}$$

$$(\forall k \in V = \{1,2,\ldots,n\})$$

6.3.1. Parcurgerea în lăţime (algoritmul BF)

b) o coada *C* in care se vor introduce *varfurile care au fost vizitate dar neprelucrate inca*, adica *nu au fost vizitati vecinii lor*.

*Algoritmul **BF** consta in scoaterea a cate unui varf din coada C si, in acelasi timp, introducerea vecinilor acestuia care nu au fost inca vizitati, vizitandu-i in acelasi timp.*

6.3.1. Parcurgerea în lăţime (algoritmul BF)

Pentru un varf oarecare *k* se pot intalni urmatoarele situatii:

- *VIZ[k]=0, k nu se afla in coada*, adica varful *k* nu a fost vizitat.
- *VIZ[k]=1, k nu se afla in coada*, adica varful *k* a fost vizitat si prelucrat.
- *VIZ[k]=1, k se afla in coada*, adica varful *k* a fost vizitat dar nu a fost prelucrat.

Orice varf introdus in coada va fi prelucrat.
Algoritmul se incheie atunci cand coada devine vida.

Descrierea algoritmului BF

Sa consideram graful neorientat G=(V,U) reprezentat prin matricea sa de adiacenta *A*.

In mod normal, *un astfel de algoritm nu este recursiv*.

Utilizam doua variabile de tip intreg:

1. ***p*** (care retine pozitia primului element din *C*)
2. ***u*** (care retine pozitia ultimului element din *C*)

Pasii algoritmului, in pseudocod:

```
C ← Ø                    // Initial coada C este vida
for k ← 1,n executa
   VIZ[k] ← 0;     // Initial toate varfurile se considera nevizitate

C[1] ← s;                // In coada C se memoreaza varful initial s
p ← 1;
u ← 1;
VIZ[s] ← 1;    // Se viziteaza varful s, fara a fi prelucrat
```

```
while (p ≤ u ) {
// Se executa un ciclu while cat timp coada C este nevida
// Se va scoate varful care urmeaza din C, indicat prin p
j ← C[p]; // La inceput se va scoate s, vizitandu-se vecinii sai
// Se prelucreaza toti vecinii k ai lui j, nevizitati inca,
// identificandu-i prin parcurgerea liniei j din matricea A.
for k ← 1,n executa
  if (a[j][k] ==1 and VIZ[k]== 0)
  {
      u ← u+1;    // Varful k, va deveni noul ultim element din C
      C[u] ← k;    // Se retine actualul ultim element in C
      VIZ[k] ← 1; // Se viziteaza varful k
   }
  p ← p+1; // Se va trece la urmatorul varf care va fi scos din C
}
```

Codul sursa al programului:

```
#include<iostream.h>

int viz[30],n,i,j,k,u,v,p,a[20][20],c[30];
int main(void)
{
   cout<<"Dati numarul de varfuri n = ";
   cin>>n;
   for(i=1; i<=n-1; i++)
    for(j=i+1; j<=n; j++)
    {
        cout<<"a["<<i<<","<<j<<"]= ";
        cin>>a[i][j];
        a[j][i] = a[i][j];
    }
```

```
cout<<"Dati varful de plecare ";    cin>>i;
for(j=1; j<=n; j++) viz[j]=0;
c[1]=i;
p=1;
u=1;
viz[i]=1;
while(p<=u)
{
     v=c[p];
     for(k=1; k<=n; k++)
     {
               if( (a[v][k]==1) && (viz[k]==0) )
               {
                         u++;
                         c[u]=k;
                         viz[k]=1;
               }
     }
     p++;
}
```

6.3.1. Parcurgerea în lăţime (algoritmul BF)

```
    cout<<"Lista varfurilor in parcugerea in
    latime: "<<endl;
    cout<<i<<" ";
    for(j=2; j<=u; j++) cout<<c[j]<<" ";
}
```

6.3.1. Parcurgerea în lăţime (algoritmul BF – varianta recursivă)

Implementarea se abordeaza astfel:

Se construieste o functie numita BF_recursiva cu un parametru formal - i, care reprezintă pozitia curentă la care s-a ajuns în coadă

Algoritmul este urmatorul:

- se parcurg nodurile grafului, cu j:
  - dacă j este adiacent cu nodul curent din coadă si j este nevizitat
  - atunci se adaugă la coadă;
  - si apoi se marchează ca fiind vizitat;
  - dacă mai sunt elemente în coadă se trece la următorul si se reapelează functia

6.3.1. Parcurgerea în lăţime (algoritmul BF – varianta recursivă)

```
#include <iostream.h>
#include <stdio.h>
int a[20][20];
int coada[20], viz[20];
int i, n , j, u, nod_plecare, m,x, y;
void BF_recursiva(int i)
{
        int j,v;
        for (j=1;j<=n;j++) { v=coada[i];
                if ((a[v][j]==1) && (viz[j]==0))
                {
                        u=u+1;
                        coada[u]=j;
                        viz[j]=1;
                } }
        if (i<=u) BF_recursiva(i+1);
}
```

6.3.1. Parcurgerea în lăţime (algoritmul BF – varianta recursivă)

```
int main()
{
        cout<<"n="; cin>>n;
        cout<<"m="; cin>>m;
        for (i=1;i<=m;i++)
        {
                cout<<"x y"; cin>>x>>y;
                a[x][y]=1; a[y][x]=1;
        }
        for (i=1;i<=n;i++) viz[i]=0;
        cout<<"dati nodul de plecare : "; cin>>nod_plecare;
        viz[nod_plecare]=1;
        coada[1]= nod_plecare;
        u=1;
        BF_recursiva(1);
        for (i=1; i<=u; i++) cout<<coada[i]<<" ";
}
```

Conţinutul cursului

6.1. Definiţii

6.2. Memorarea(reprezentarea) grafurilor

6.3. Parcurgerea grafurilor

- **6.3.1. Parcurgerea în lăţime (algoritmul BF)**
- **6.3.2. Parcurgerea în adâncime (algoritmul DF)**

6.4. Grafuri hamiltoniene şi grafuri euleriene

6.5. Aplicaţii ale grafurilor neorientate

6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.3.2. Parcurgerea în adâncime (algoritmul DF)

Algoritmul *DF (Depth First)* se caracterizează prin faptul că realizează o parcurgere a grafului „*în adâncime*“ atât cât este posibil.

Parcurgerea începe cu un vârf s ales inițial.

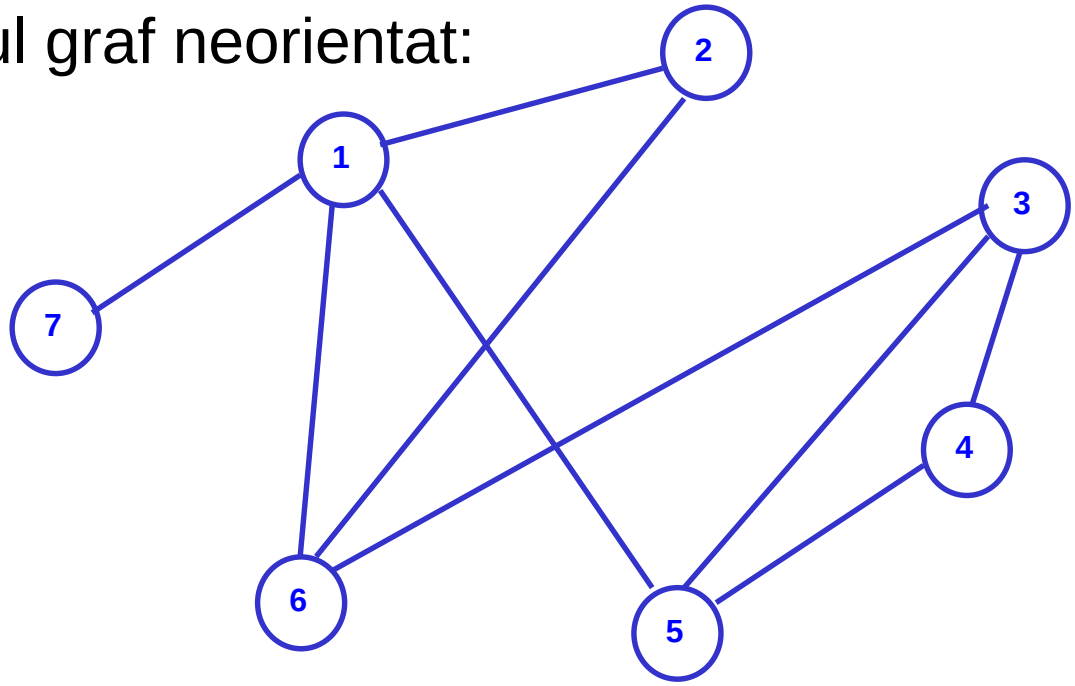
Prelucrarea unui vârf conduce la prelucrarea primului său vecin încă nevizitat, apoi se prelucrează primul vecin al acestuia care nu a fost încă vizitat, etc.

Se observă un procedeu recursiv de parcurgere a grafului.

Această tehnică de parcurgere conduce la efectuarea unui număr relativ mare de apeluri recursive înainte de a se întoarce dintr-un apel.

6.3.2. Parcurgerea în adâncime (algoritmul DF)

Să considerăm următorul graf neorientat:



Metoda *DF*, aplicată acestui graf, pornind de la *vârful inițial 1*, conduce la vizitarea varfurilor în următoarea ordine:

1,2,6,3,4,5,7

6.3.2. Parcurgerea în adâncime (algoritmul DF)

- Se observa ca de fiecare data, *atunci cand se ajunge la prelucrarea unui anumit varf, se cauta varful adiacent lui care nu a fost inca vizitat*.
- *Daca nu mai este posibil de a continua, se revine la varful de la care am plecat ultima data si cautam un alt varf adiacent cu el care nu a fost inca vizitat (daca exista).*
- Prin urmare, algoritmul respectiv este de tip backtracking (metoda ce va fi studiata in cursurile urmatoare).
- *Parcurgerea DF a unui graf nu este unica pentru ca ea depinde atat de alegerea varfului initial cat si de ordinea de vizitare a vecinilor.*

6.3.2. Parcurgerea în adâncime (algoritmul DF)

- Acum se poate face o comparatie intre cele doua tehnici de parcurgere *BF (varianta recursiva)* si *DF*.
- Spre deosebire de algoritmul *BF*, in algoritmul *DF*, alaturi de tabloul unidimensional *VIZ[ ]*, cu aceeasi semnificatie ca la metoda *BF*, se va utiliza *o stiva S in care se respecta ordinea de parcurgere mentionata*.
- *Primul varf adiacent cu cel curent, inca nevizitat, se va afla in varful stivei.*

6.3.2. Parcurgerea în adâncime (algoritmul DF)

Descrierea algoritmului DF

- Se va considera graful neorientat G=(V,U) reprezentat prin matricea sa de adiacenta *A* .
- Vom folosi un vector *viz[ ],* in care componenta *viz[k]* reprezinta varful *k* vizitat.
- *Metoda constă în a "vizita" vârful iniţial k şi a continua cu vecinii săi nevizitaţi j.*
- *Tot timpul mergem în adâncime, cât este posibil, cât nu, ne întoarcem şi plecăm, dacă este posibil, spre vecinii nevizitaţi încă.*

Codul sursa al algoritmului de parcurgere in adancime:

```
#include<iostream.h>

int n,m,i,j,p,a[20][20],viz[30];
void df(int k)
{
    int j;
    cout<<k<<" ";
    viz[k]=1;
    for(j=1; j<=n; j++)
      if( (a[k][j]==1) && (viz[j]==0) )
                                          df(j);
    return;
}
```

```
int main(void)
{
    cout<<"Dati numarul de varfuri n = ";cin>>n;
    for(i=1; i<=n-1; i++)
     for(j=i+1; j<=n; j++)
     {
         cout<<"a["<<i<<","<<j<<"]= ";
         cin>>a[i][j];
         a[j][i]=a[i][j];
     }
     cout<<"Dati varful de plecare ";
     cin>>p;
     df(p);
}
```

Conţinutul cursului

6.1. Definiţii
6.2. Memorarea(reprezentarea) grafurilor
6.3. Parcurgerea grafurilor
- **6.3.1. Parcurgerea în lăţime (algoritmul BF)**
- **6.3.2. Parcurgerea în adâncime (algoritmul DF)**

6.4. Grafuri hamiltoniene şi grafuri euleriene
6.5. Aplicaţii ale grafurilor neorientate
6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.4. Grafuri hamiltoniene si grafuri euleriene

Definitie

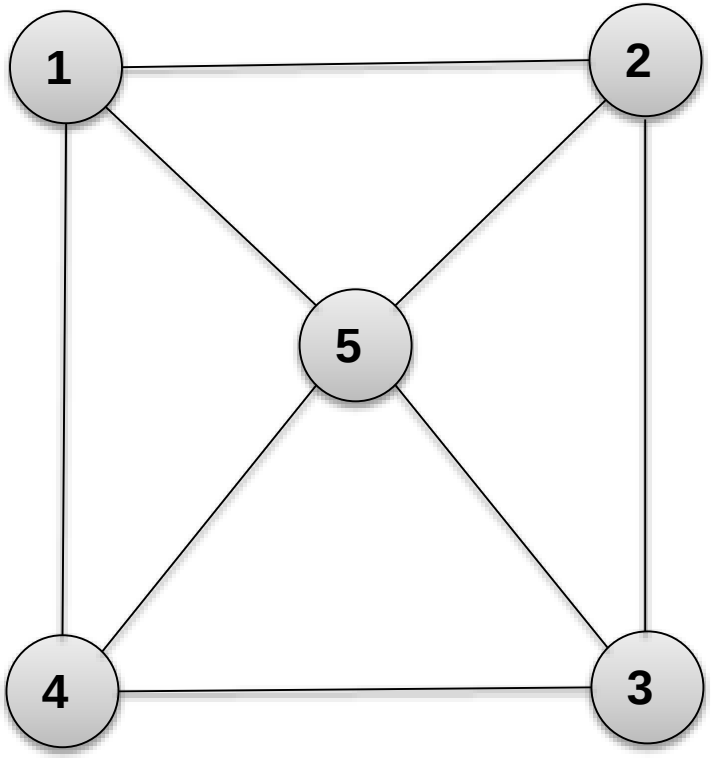
Se numeste **ciclu hamiltonian** un ciclu elementar care contine toate varfurile grafului.

Definitie

Un graf se numeste **hamiltonian** daca are cel putin un ciclu hamiltonian.

6.4. Grafuri hamiltoniene si grafuri euleriene

Graf hamiltonian



Ciclu hamiltonian: [1,2,3,5,4,1]

6.4. Grafuri hamiltoniene si grafuri euleriene

TEOREMA

Fie G=(X,U) un graf. Daca gradul fiecarui varf este >=n/2 atunci graful este hamiltonian.

Observație: Conditia din teorema este necesara, dar nu si suficienta.

Aplicatie:

Ciclurile hamiltoniene sunt legate de *problema comis-voiajorului* care pleaca dintr-un oras si trebuie sa treaca o singura data prin celelalte orase si sa se intoarca de unde a plecat.

6.4. Grafuri hamiltoniene si grafuri euleriene

Definitie

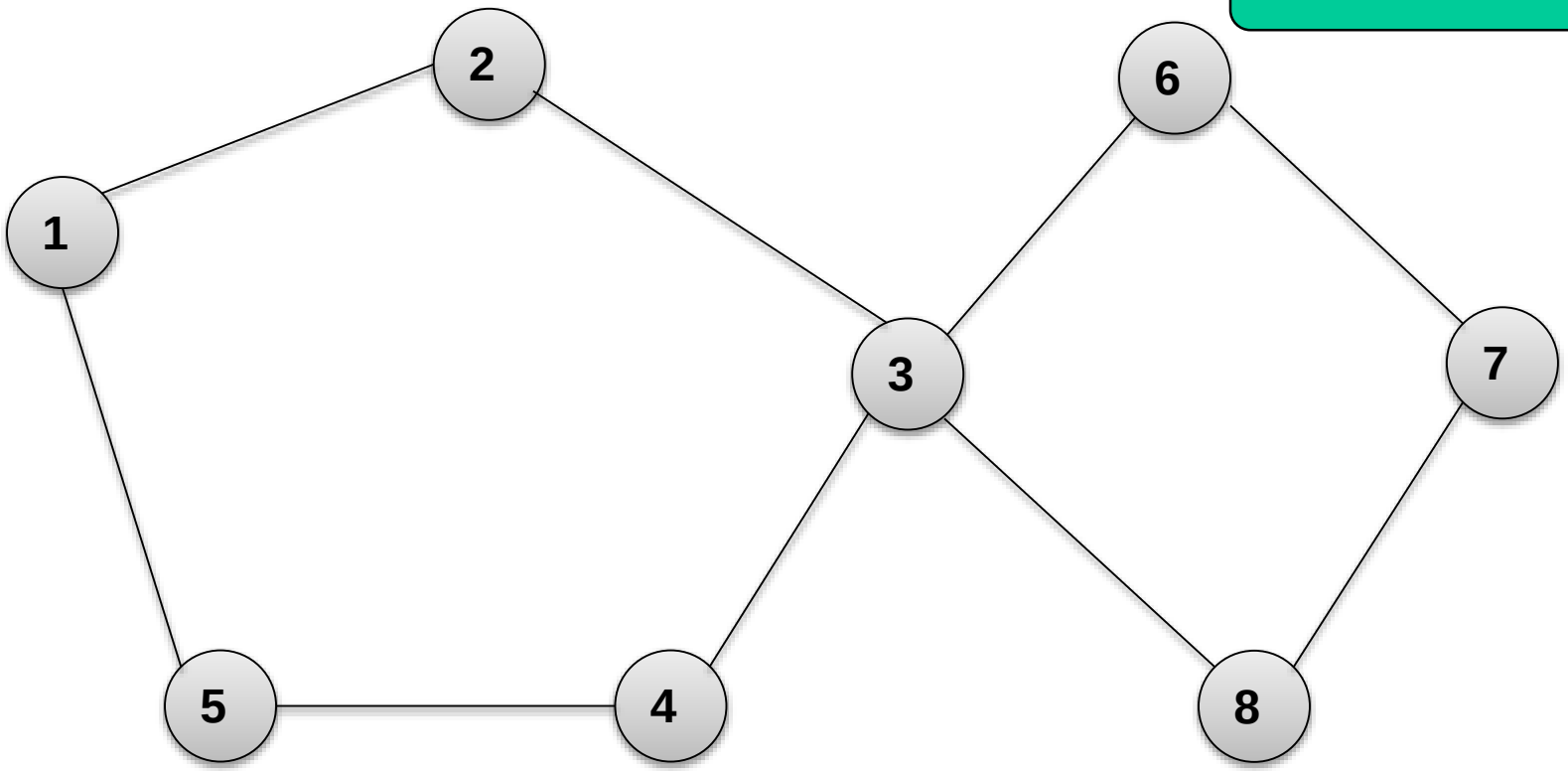
Un ciclu se numeste **eulerian** daca trece o singura data prin toate muchiile.

Definitie

Un graf se numeste **eulerian** daca are un ciclu eulerian.

6.4. Grafuri hamiltoniene si grafuri euleriene

Graf eulerian



Ciclu eulerian: [1,2,3,6,7,8,3,4,5,1]

6.4. Grafuri hamiltoniene si grafuri euleriene

TEOREMA

Un graf fara varfuri izolate se numeste eulerian daca si numai daca este conex si gradul fiecarui varf este par.

Observație

Dintre grafurile complete sunt euleriene cele cu numar impar de varfuri.

6.4. Grafuri hamiltoniene si grafuri euleriene

Definitie

O componenta conexa este un subgraf conex maximal cu aceasta proprietate.

Definitie

O componenta conexa a unui graf G=(X,U) este un subgraf S=(Y,Z) cu proprietatea ca nu exista un lant care sa uneasca un varf din Y cu un varf din X-Y.

Observație

Fiecare varf izolat constituie o componenta conexa.

Codul sursa: Determinarea componentelor conexe

```
#include<iostream.h>
int a[20][20],cc[30];
int n,ncc,i,j,v;        // ncc=nr. de componente conexe
void df(int v)
{
  int i;
  cc[v]=ncc;
  for(i=1;i<=n;i++)
          if( (a[v][i]==1) && (cc[i]==0) )
                                  df(i);
}
int main(void)
{
     cout<<"Dati numarul de varfuri n = ";          cin>>n;
     cout<<"Matricea de adiacenta "<<endl;
     for(i=1;i<=n-1;i++)
       for(j=i+1;j<=n;j++)
       {
            cout<<"a["<<i<<","<<j<<"]= ";
            cin>>a[i][j];
            a[j][i]=a[i][j];
       }
```

Parcurgere in adancime pentru un varf v

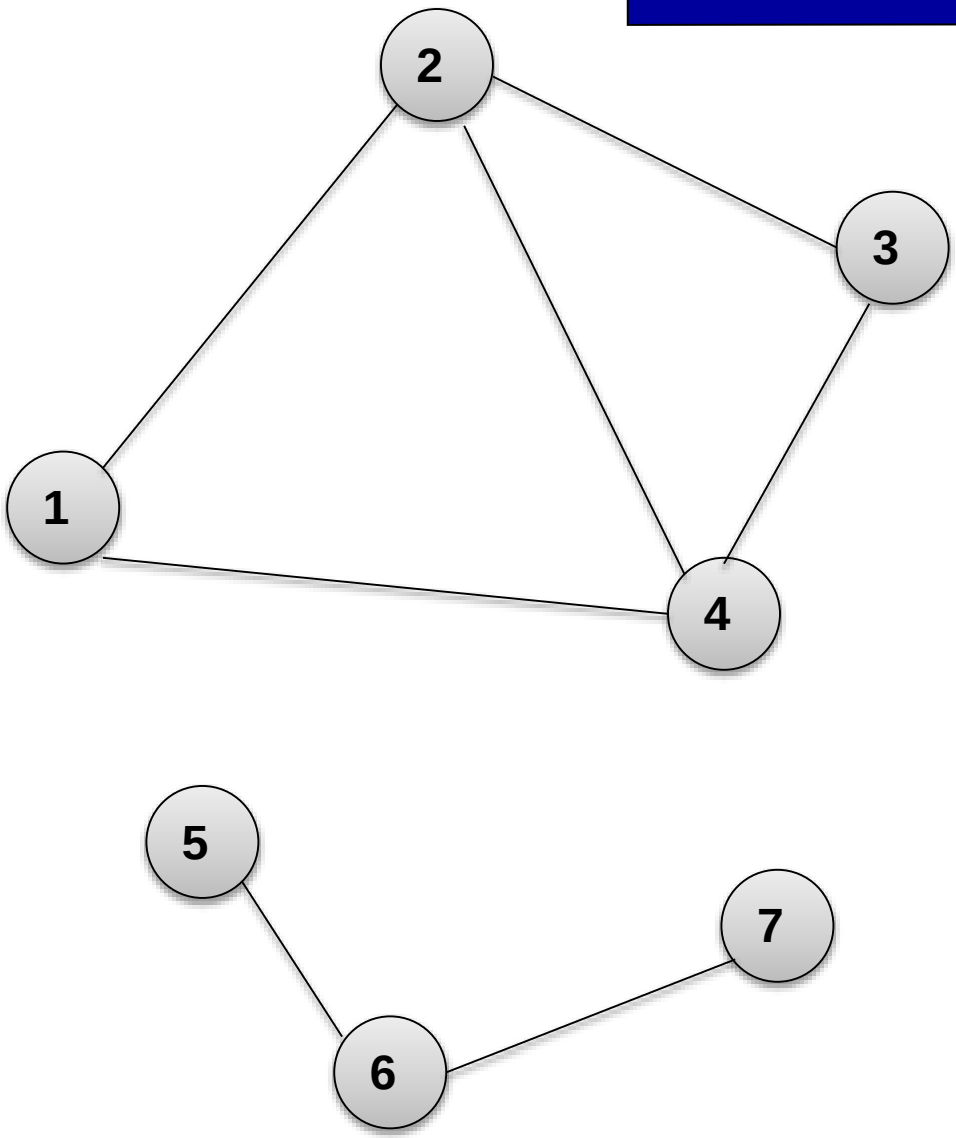
```
    ncc=0;
    for(i=1;i<=n;i++) cc[i]=0;
    for(i=1;i<=n;i++)
        if (cc[i]==0)
        {
                ncc=ncc+1;
                df(i);
        }
    for(i=1;i<=ncc;i++)          // ncc = nr. de comp. conexe
    {
        cout<<"componenta "<<i<<" ";
        for(j=1;j<=n;j++)
                if(cc[j]==i)
                        cout<<j<<" ";
                cout<<endl;
    }
}
```

Determinarea varfurilor pentru fiecare componenta conexa

Afisare elementelor pentru fiecare componenta conexa

```
E:\Universitate\2009-2010\Semestrul
Dati numarul de varfuri n = 7
Matricea de adiacenta
a[1,2]= 1
a[1,3]= 0
a[1,4]= 1
a[1,5]= 0
a[1,6]= 0
a[1,7]= 0
a[2,3]= 1
a[2,4]= 0
a[2,5]= 0
a[2,6]= 0
a[2,7]= 0
a[3,4]= 1
a[3,5]= 0
a[3,6]= 0
a[3,7]= 0
a[4,5]= 0
a[4,6]= 0
a[4,7]= 0
a[5,6]= 1
a[5,7]= 0
a[6,7]= 1
componenta 1 1 2 3 4
componenta 2 5 6 7
componenta 3
componenta 4
componenta 5
componenta 6
componenta 7


Terminated with return code 0
Press any key to continue ...
```



*Să se verifice dacă un graf dat prin matricea de adiacenţă este **graf conex** si să se afiseze un mesaj corespunzător*

```
#include<iostream.h>
int mat[10][10], n, viz[10];
void citire()
{
    int i,j;
    for(i=1;i<n;i++)
        for(j=i+1;j<=n;j++)
        {
            cout<<"mat["<<i<<"]["<<j<<"]=";
            cin>>mat[i][j];
            mat[j][i]=mat[i][j];
        }
}
void parcurg(int x)
{   int i;
    viz[x]=1;
    for(i=1;i<=n;i++)
        if(mat[x][i]&&viz[i]==0) parcurg(i);
}
```

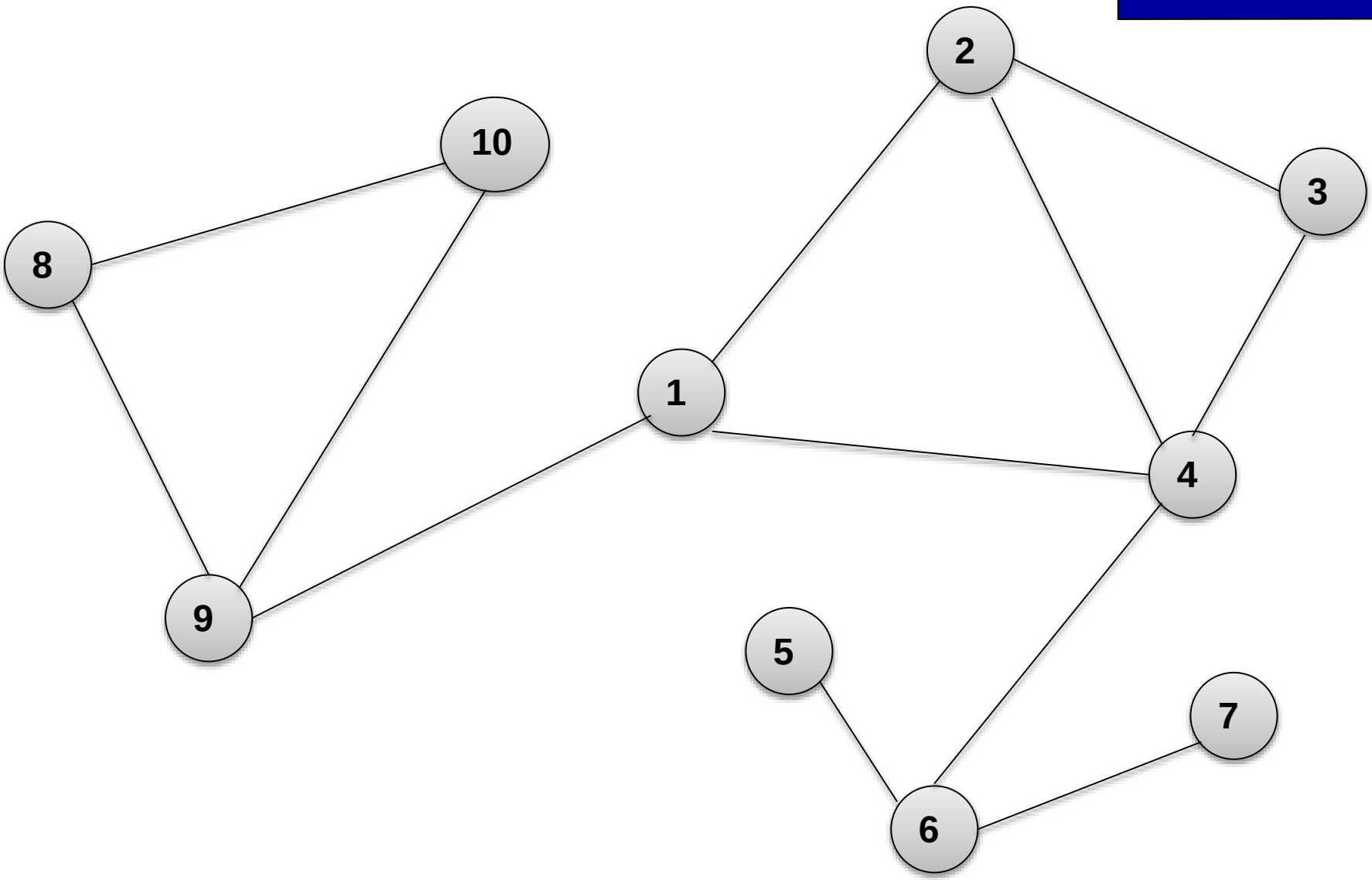
Functie de citire a datelor si memorare in matricea de adiacenta

Parcurgerea in adancime

```
int conex()
{
    int i;
    parcurg(1);
    for(i=1; i<=n; i++)
        if(viz[i]==0)      return 0;
    return 1;
}
int main()
{
    cout<<"n="; cin>>n;
    citire();
    if(conex()==1)
        cout<<"Graful dat este conex";
    else
        cout<<"Graful dat NU este conex";
}
```

Determinarea conexitatii grafului prin parcurgere in adancime si verificarea vectorului viz

Afisare mesajului corespunzator


2
10
3
8
1
4
9
5
7
6

Conţinutul cursului

6.1. Definiţii
6.2. Memorarea(reprezentarea) grafurilor
6.3. Parcurgerea grafurilor
- **6.3.1. Parcurgerea în lăţime (algoritmul BF)**
- **6.3.2. Parcurgerea în adâncime (algoritmul DF)**

6.4. Grafuri hamiltoniene şi grafuri euleriene
6.5. Aplicaţii ale grafurilor neorientate
6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.5. Aplicatii ale grafurilor neorientate

Aplicatii ale parcugerilor grafurilor neorientate:

1. Afisarea componentelor conexe ale unui graf neorientat

2. Determinarea ciclurilor care contin un varf specificat

```
#include<iostream.h>
int x[30],k,n,m,i,j,p;
int a[20][20];     // matricea de adiacenţă a grafului
int viz[30];       // arată dacă un nod a fost sau nu vizitat
int g,t,u,kk,r,tt;
int c[5][5];       // matrice ce conţine componentele conexe care sunt
                   // reprezentate sub forma de mulţimi
int main(void)
{
    cout<<"Dati numarul de varfuri n = ";cin>>n;
    cout<<"Matricea de adiacenta "<<endl;
    for(i=1; i<=n-1; i++)
      for(j=i+1; j<=n; j++)
      {
           cout<<"a["<<i<<","<<j<<"]= ";
           cin>>a[i][j];
           a[j][i]=a[i][j];
      }
```

```
k=0;kk=0;
for(i=1; i<=n; i++) viz[i]=0;
g=1;    // g ramane 1 atata timp cat mai sunt componente conexe de gasit in graf
while(g!=0)
{
      j=1;t=1;
      while( (t==1) && (j<=n) )
        if(viz[j]==0) t=0;
        else j++;
      if(t==1) g=0;
      else
      {
              k++; // s-a gasit o noua componenta conexa
              kk++;
```

```
r=1; // numar cate elem. are o componenta conexa
viz[j]=k;
c[kk][r++]=j;
p=1;u=1;      //  folosesc parcurgerea in latime
x[p]=j;
while(p<=u)
{
      for(i=1;i<=n;i++)
        if( (viz[i]==0) && (a[i][x[p]]==1) )
                              {
                                    u++;
                                    x[u]=i;
                                    viz[i]=k;
                                    c[kk][r++]=i;
                              }
                            p++;
                  }
                  tt=r;
    }
}
```

Parcurgere in latime pentru un varf v si completarea liniei kk din matricea de componente conexe, cu toate varfurile din acea componenta conexa

```
    if(k==1) cout<<"Graful este conex";
    else
    {
      cout<<"S-au gasit urmat. componente conexe"
    <<endl;
      for(i=1;i<=kk;i++)
      {
          cout<<"Componenta conexa : "<<i<<" = { ";
          for(j=1;j<=tt;j++)
                      cout<<c[i][j]<<",";
           cout<<"}"<<endl;
      }
  }
}
```

6.5. Aplicatii ale grafurilor neorientate

Aplicatii ale parcugerilor grafurilor neorientate:

1. Afisarea componentelor conexe ale unui graf neorientat

2. Determinarea ciclurilor care contin un varf specificat

```
#include <iostream.h>
#define n 6
int m[n][n]={    {0,1,0,0,0,1},
                 {0,0,1,0,0,0},
                 {0,1,0,1,0,1},
                 {1,0,0,0,0,1},
                 {0,0,1,0,0,0},
                 {0,1,0,0,1,0}
             };
int s[n];
int varf=0;
int verific(int c)    //se verifica daca nodul a fost inclus in stiva
  {
     for (int i=0;i<varf;i++)
            if (s[i]==(c)) return 1;
    return 0;
  }
```

Initializarea matricei de adiacenta

Daca nodul exista in stiva se returneaza 1, altfel 0

```
void afisare()
{
    cout<<"\n Solutie: ";
    for(int i=0; i<varf; i++) cout<<“ ”<<s[i];
}
int main()
{ int nod;     // nod = nodul de la care incepe parcurgerea
  int rand, col;
  int parcurs;        // parcurs=1 - s-a parcurs intreg graful
  int extrag;         // extrag - var. folosita pentru extragerea
    unui elem din stiva
```

```
cout<<"Introdu numarul nodului:";
cin>>nod;
s[varf]=nod;
varf++;                          // se depune prima valoare in stiva
rand=nod;
col=0;
parcurs=0;
while(parcurs==0)
 {
   while ((m[rand][col]==0)&&(col<n))
                        col++;
   if (col<n)
    { if (!verific(col))
       { s[varf]=col;            // se adauga nodul in lista de noduri (stiva)
         varf++;
         rand=col;
         col=0;  }
      else
       { if (col==nod)
         afisare();
         col++; }
    }
```

```
else // ptr nodul din varful stivei s-au verificat toate arcele, deci se extrage
     din stiva
      { extrag=s[varf-1];
      varf--;
      if (extrag==nod)
        parcurs=1; // s-au extras toate nodurile din stiva - se incheie
     parcurgerea grafului
      else            // se trateaza in continuare nodul ramas in varf
        {
          col=extrag+1;
         rand=s[varf-1];
        }
      }
  }
}
```

Conţinutul cursului

6.1. Definiţii
6.2. Memorarea(reprezentarea) grafurilor
6.3. Parcurgerea grafurilor
- **6.3.1. Parcurgerea în lăţime (algoritmul BF)**
- **6.3.2. Parcurgerea în adâncime (algoritmul DF)**

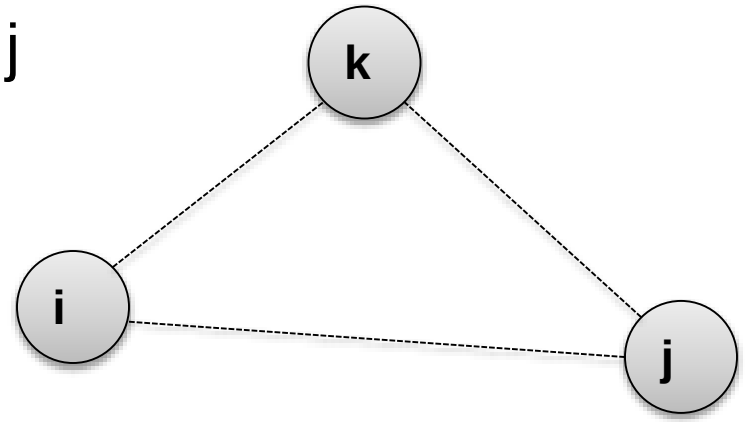
6.4. Grafuri hamiltoniene şi grafuri euleriene
6.5. Aplicaţii ale grafurilor neorientate
6.6. Matricea lanţurilor. Algoritmul Roy-Warshall

6.6. Matricea lanturilor. Algoritmul Roy-Warshall

Formarea unei matrici in care sa aiba lanturile dintr-un graf neorientat.

l[i,j]=1, daca exista lant de la i la j

l[i,j]=0, altfel



Algoritmul Roy-Warshall:

Initial **l=a;**

```
for(k=1; k<=n; k++)
    for(i=1; i<=n; i++)
     for(j=1; j<=n; j++)
          if( l[i][k] == 1 && l[k][j] == 1 ) l[i][j] = 1;
```

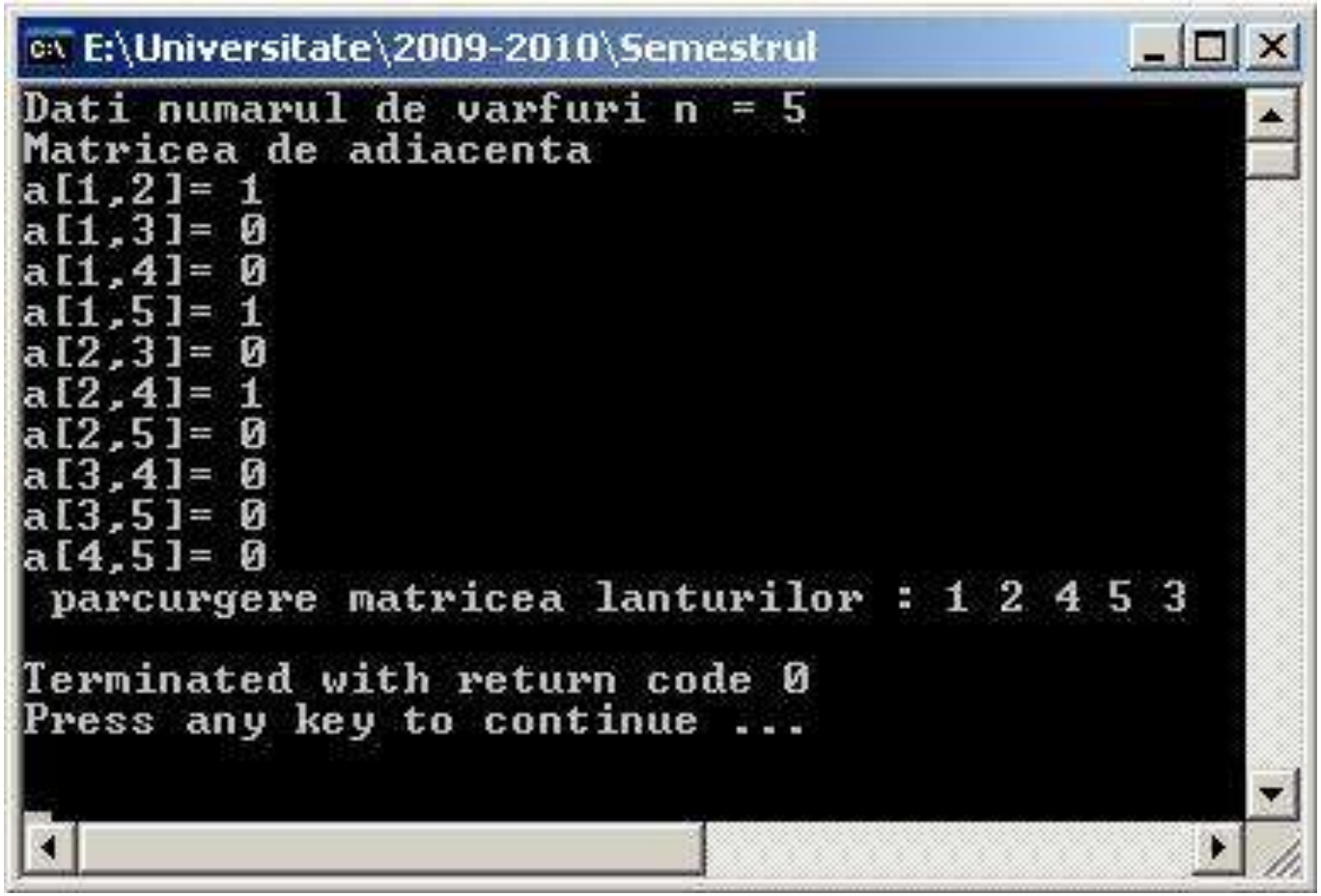
Initial matricea lanturilor este chiar matricea de adiacenta, si apoi daca exista lant de la i la k si lant de la k la j atunci exista lant de la i la j.

6.6. Matricea lanturilor. Algoritmul Roy-Warshall

Parcurgerea matricei lanturilor a unui graf neorientat.

```
#include<iostream.h>
int a[20][20],l[20][20];
int cc[30];
int n,ncc,i,j,k;
int main(void)
{
    cout<<"Dati numarul de varfuri n = ";        cin>>n;
    cout<<"Matricea de adiacenta "<<endl;
    for(i=1;i<=n-1;i++)
      for(j=i+1;j<=n;j++)
      {
            cout<<"a["<<i<<","<<j<<"]= ";
            cin>>a[i][j];
            a[j][i]=a[i][j];
      }
   for(i=1;i<=n;i++)
      for(j=i;j<=n;j++) l[i][j]=a[i][j];
```

```
for(k=1;k<=n;k++)
   for(i=1;i<=n;i++)
     for(j=1;j<=n;j++)
          if(l[i][k]==1 && l[k][j]==1)
                                  l[i][j]=1;
  ncc=0;
  for(i=1;i<=n;i++) cc[i]=0;
  for(i=1;i<=n;i++)
     if(cc[i]==0)
      {
         ncc=ncc+1;
         cc[i]=ncc;
         for(j=1;j<=n;j++)
               if(l[i][j]==1) cc[j]=ncc;
      }
  cout<<" parcurgere matricea lanturilor : ";
      for (i=1;i<=ncc;i++)
       for (j=1;j<=n;j++)
            if (cc[j]==i) cout<<j<<" ";
}
```


E:\Universitate\2009-2010\Semestrul
Dati numarul de varfuri n = 5
Matricea de adiacenta
a[1,2]= 1
a[1,3]= 0
a[1,4]= 0
a[1,5]= 1
a[2,3]= 0
a[2,4]= 1
a[2,5]= 0
a[3,4]= 0
a[3,5]= 0
a[4,5]= 0
parcurgere matricea lanturilor : 1 2 4 5 3
Terminated with return code 0
Press any key to continue ...

Întrebări?