Tabăra de pregătire a lotului de juniori
Măgurele
18-22 mai 2016

Teoria Grafurilor
Curs Introductiv

Prof Dana Lica
Centrul Județean de Excelență Prahova - Ploiești

Teoria grafurilor

1. *Noţiuni introductive*

1.1 Terminologie

Graf ⇔ orice mulţime finită *V*, prevăzută cu o relaţie binară internă *E*. Notăm graful cu *G=(V,E)*.

Graf neorientat ⇔ un graf *G=(V,E)*, în care relaţia binară este simetrică: dacă *(v,w)*$\in$*E*, atunci *(w,v)*$\in$*E*.

Graf orientat ⇔ un graf *G=(V,E)*, în care relaţia binară nu este simetrică.

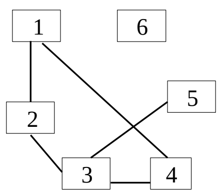
Nod ⇔ element al mulţimii *V*, unde *G=(V,E)* este un graf neorientat.

Vârf ⇔ element al mulţimii *V*, unde *G=(V,E)* este un graf orientat sau neorientat.

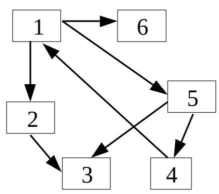
Muchie ⇔ element al mulţimii *E* ce descrie o relaţie existentă între două vârfuri din *V*, unde *G=(V,E)* este un graf neorientat;

Arc ⇔ element al mulţimii *E* ce descrie o relaţie existentă între două vârfuri din *V*, unde *G=(V,E)* este un graf orientat;
Arcele sunt parcurse în direcţia dată de relaţia vârf → succesor_direct. Muchiile unui graf neorientat sunt considerate ca neavând direcţie, deci pot fi parcurse în ambele sensuri.

Adiacenţă ⇔ Vârful *w* este adiacent cu *v* dacă perechea *(v,w)*$\in$*E*. Într-un graf neorientat, existenţa muchiei *(v,w)* presupune că *w* este adiacent cu *v* şi *v* adiacent cu *w*.



Graf neorientat



Graf orientat

În exemplele din figura de mai sus, vârful 1 este adiacent cu 4, dar 1 şi 3 nu reprezintă o pereche de vârfuri adiacente.

Incidenţă ⇔ o muchie este incidentă cu un nod dacă îl are pe acesta ca extremitate. Muchia *(v,w)* este incidentă în nodul *v*, respectiv *w*.

Incidenţă spre interior ⇔ Un arc este incident spre interior cu un vârf, dacă îl are pe acesta ca vârf terminal (arcul converge spre vârf). Arcul *(v,w)* este incident spre interior cu vârful *w*.

Incidenţă spre exterior ⇔ Un arc este incident spre exterior cu un vârf dacă îl are pe acesta ca vârf iniţial (arcul pleacă din vârf). Arcul *(v,w)* este incident spre exterior cu vârful *v*.

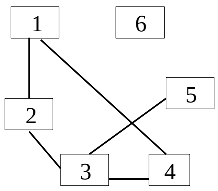
Grad ⇔ Gradul unui nod *v*, dintr-un graf neorientat, este un număr natural ce reprezintă numărul de noduri adiacente cu acesta.

Grad interior ⇔ În cazul unui graf orientat, fiecare nod *v* are asociat un număr numit grad interior şi care este egal cu numărul de arce care îl au pe *v* ca vârf terminal (numărul de arce incidente spre interior).

Grad exterior ⇔ În cazul unui graf orientat, fiecare nod *v* are asociat un număr numit grad exterior şi care este egal cu numărul de arce care îl au pe *v* ca vârf iniţial (numărul de arce incidente spre exterior).

Vârf izolat ⇔ Un vârf cu gradul 0.

Vârf terminal ⇔ Un vârf cu gradul 1.



Vârful 5 este terminal (gradul 1).

Vârful 6 este izolat (gradul 0).

Vârfurile 1, 2, 4 au gradele egale cu 2.

Lanţ ⇔ este o secvenţă de noduri ale unui graf neorientat $G=(V,E)$, cu proprietatea că oricare două noduri consecutive sunt adiacente:

$w_1, w_2, w_3,. . ,w_p$ cu proprietatea că $(w_i, w_{i+1})\in E$ pentru $1\leq i<p$.

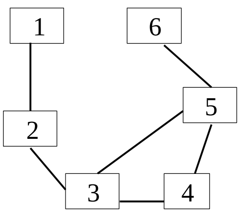
Lungimea unui lanţ ⇔ numărul de muchii din care este format.

Lanţ simplu ⇔ lanţul care conţine numai muchii distincte.

Lanţ compus ⇔ lanţul care nu este format numai din muchii distincte.

Lanţ elementar ⇔ lanţul care conţine numai noduri distincte.

Ciclu ⇔ Un lanţ în care primul nod coincide cu ultimul. Ciclul este elementar dacă este format doar din noduri distincte, excepţie făcând primul şi ultimul. Lungimea minimă a unui ciclu este 3.



Succesiunea de vârfuri 2, 3, 5, 6 reprezintă un lanţ simplu şi elementar de lungime 3.
Lanţul 5 3 4 5 6 este simplu dar nu este elementar.
Lanţul 5 3 4 5 3 2 este compus şi nu este elementar.
Lanţul 3 4 5 3 reprezintă un ciclu elementar.

Drum ⇔ este o secvenţă de vârfuri ale unui graf orientat $G=(V,E)$, cu proprietatea că oricare două vârfuri consecutive sunt adiacente:

$(w_1, w_2, w_3,. . ,w_p)$, cu proprietatea că $(w_i, w_{i+1})\in E$, pentru $1\leq i<p$.

Lungimea unui drum ⇔ numărul de arce din care este format.

Drum simplu ⇔ drumul care conţine numai arce distincte.

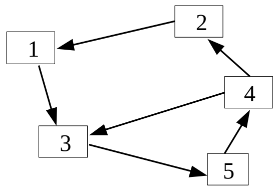
Drum compus ⇔ drumul care nu este format numai din arce distincte.

Drum elementar ⇔ drumul care conţine numai vârfuri distincte.

Circuit ⇔ Un drum în care primul vârf coincide cu ultimul. Circuitul este elementar dacă este format doar din vârfuri distincte, excepţie făcând primul şi ultimul.

Buclă ⇔ Circuit format dintr-un singur arc.

Ciclu elementar 3,6,4,5,1,3



Circuit elementar 1,3, 5,4,2,1

Graf parţial ⇔ Un graf *G'=(V,E')* reprezintă graf parţial al grafului *G=(V,E)* dacă *E'*$\subseteq$*E*. Cu alte cuvinte *G'* este graf parţial al lui *G*, dacă este identic, sau se obţine prin suprimarea unor muchii (respectiv arce) din *G*.

Subgraf ⇔ Un subgraf al lui *G=(V,E)* este un graf *G'=(V',E')* în care *V'*$\subseteq$*V*, iar *V'* conţine toate muchiile/arcele din *E* ce au ambele extremităţi în *V'*. Cu alte cuvinte *G'* este subgraf al lui *G*, dacă este identic, sau se obţine prin suprimarea unor noduri împreună cu muchiile/arcele incidente cu acestea.

Graf regulat ⇔ graf neorientat în care toate nodurile au acelaşi grad.

Graf complet ⇔ graf neorientat *G=(V,E)* în care există muchie între oricare două noduri. Numărul de muchii ale unui graf complet este $|V|*|V\text{-}1|/2$.

Graf conex ⇔ graf neorientat *G=(V,E)* în care, pentru orice pereche de noduri *(v,w)*, există un lanţ care le uneşte.

Graf tare conex ⇔ graf orientat *G=(V,E)* în care, pentru orice pereche de vârfuri *(v,w)*, există drum de la *v* la *w* şi un drum de la *w* la *v*.

Componentă conexă ⇔ subgraf al grafului de referinţă, maximal în raport cu proprietatea de conexitate (între oricare două vârfuri există lanţ);

Lanţ hamiltonian ⇔ un lanţ elementar care conţine toate nodurile unui graf.

Ciclu hamiltonian ⇔ un ciclu elementar care conţine toate nodurile grafului.

Graf hamiltonian ⇔ un graf *G* care conţine un ciclu hamiltonian.

Condiţie de suficienţă: Dacă *G* este un graf cu $n \geq 3$ vârfuri, astfel încât gradul oricărui vârf verifică inegalitatea: $gr(x) \geq n/2$, rezultă că *G* este graf hamiltonian.

Lanţ eulerian ⇔ un lanţ simplu care conţine toate muchiile unui graf.

Ciclu eulerian ⇔ un ciclu simplu care conţine toate muchiile grafului.

Graf eulerian ⇔ un graf care conţine un ciclu eulerian.

Condiţie necesară şi suficientă: Un graf este eulerian dacă şi numai dacă oricare vârf al său are gradul par.

1.2 Moduri de reprezentare la nivelul memoriei

Există mai multe modalităţi standard de reprezentare a unui graf *G=(V, E)*:

1. matricea de adiacenţă;
2. listele de adiacenţă;
3. matricea ponderilor (costurilor);
4. lista muchiilor.

• *Matricea de adiacenţă* este o matrice binară (cu elemente 0 sau 1) care codifică existenţa sau nu a muchiei/arcului între oricare pereche de vârfuri din graf. Este indicată ca mod de memorare a grafurilor, în special în cazul grafurilor dense, adică cu număr mare de muchii ($|V^2| \approx E$).

```
Creare_MA_neorientat(G=(V,E))       /*complexitate: O(V² + E)*/
┌pentru i←1, |V| executa
│ ┌pentru j ← 1 to |V| executa G[i][j] ← 0;
│ └■
└■
┌pentru e = 1,|E| executa
│ citeste i, j;
│ G[i][j] ← 1;
│ G[j][i] ← 1;     //instructiunea lipseste in cazul digrafului
└■
```

Implementarea în limbaj a subprogramului care creează matricea de adiacenţă a unui graf neorientat ia în considerare următoarele declaraţii:

```
const  MAX_N=101;
       MAX_M=1001;
var n,m:longint;
G:array[0..max_n,0..max_n] of byte;
```

```
#include <stdio.h>
#define MAX_N 101
#define MAX_M 1001
int N, M,; char G[MAX_N][MAX_N];
```

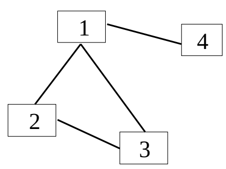
Subprogramul *citeste_graf()* preia informaţiile din fişierul text *graf.in*, în felul următor: de pe prima linie numărul *n* de vârfuri şi *m* de muchii, iar de pe următoarele *m* linii, perechi de numere reprezentând muchiile grafului. Matricea *G* de adiacenţă se consideră iniţializată cu valoarea 0.

```
void citeste_graf(void)
{
    int i, j;
    freopen("graf.in","r", stdin);
    scanf("%d %d", &N, &M);
    for (; M > 0; M--)
    {
      scanf("%d %d", &i, &j);
      G[i][j] = G[j][i] = 1;
    }
}
```

```
void citeste_graf(void)
{
    int i, j;
    freopen("graf.in","r", stdin);
    scanf("%d %d", &N, &M);
    for (; M > 0; M--)
    {
      scanf("%d %d", &i, &j);
      G[i][j] = 1;
    }
}
```

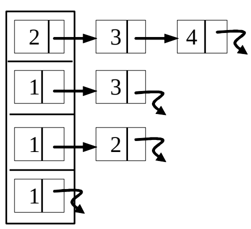
• *Listele de adiacenţă* reţin, pentru fiecare nod din graf, toate vârfurile adiacente cu acesta (vecine cu el). Ca modalitate de reprezentare se poate folosi un vector *LA* ale cărui elemente sunt adresele de început ale celor |*V*| liste simplu înlănţuite memorate, una pentru fiecare vârf din V. Lista simplu înlănţuită, a cărei adresă de început este reţinută de *LA*[*u*] memorează toate vârfurile din *G*, adiacente cu *u* şi stocate într-o ordine oarecare.

Pentru graful alăturat exemplificăm reprezentarea atât în liste de adiacenţă, cât şi în matrice de adiacenţă.



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Matricea de adiacenţă *(MA)*



Liste de adiacenţă *(LA)*

```
# include <vector>

using namespace std;

vector <int> G[MAXN];
```

```
#include <stdio.h>
#define MAXN 1001

struct lista
{ int nod;
  lista *urm;
} *G[MAXN];
```

Subprogramul *citeste_graf()* preia informaţiile din fişierul text *graf.in*, în felul următor: de pe prima linie, numărul *n* de vârfuri şi *m* de muchii, iar de pe următoarele *m* linii, perechi de numere reprezentând muchiile grafului. La apelul *adauga(i,j)* se realizează inserarea unui element de informaţie *j* în lista lui vecinilor nodului *i*. Tabloul *G*, reprezentând listele de adiacenţă, se consideră iniţializat cu constanta *NULL* (C++).

```

void load(){
   scanf("%d %d\n", &N, &M);
   for (i=1; i<= M; i++){
        scanf("%d %d\n",&x, &y);
        G[x].push_back(y);
        G[y].push_back(x);
 }
 return;
}
```

```
void adauga(int i, int j)
{
  lista *p;
  p = new lista; p->nod = j;
  p->urm = G[i]; G[i] = p;
}

void citeste_graf(void)
{
  int i, j;
  freopen("graf.in","r", stdin);
  scanf("%d %d", &N, &M);
  for (; M > 0; M--){
    scanf("%d %d", &i, &j);
    adauga(i, j); adauga(j, i);
   }
}
```

Un potenţial dezavantaj al reprezentării sub forma listelor de adiacenţă este modul oarecum anevoios de a determina dacă o muchie *(u,v)* este prezentă sau nu în graf. Eficienţa reprezentării unui graf este dată de densitatea acestuia, deci matricea de adiacenţă este viabilă dacă numărul muchiilor este aproximativ egal cu $|V^2|$.

• *Matricea costurilor* este o matrice cu $|V|$ linii şi $|V|$ coloane, care codifică existenţa sau nu a muchiei/arcului, între oricare pereche de vârfuri din graf, prin costul acesteia. Astfel:

$G[i][j] = \propto$, dacă $(i, j) \notin E$
$G[i,][j] = cost < \propto$, dacă $(i, j) \in E$
$G[i,][j] = 0$, dacă $i = j$

```
Creare_MC_neorientat(G=(V,E))       /*complexitate: O(V² + E)*/
┌pentru i←1, |V| executa
│ ┌pentru j ← 1 to |V| executa
│ │ ┌daca i = j atunci G[i][j] ← 0;
│ │ │altfel G[i][j] ← ∝
│ │ └■
│ └■
└■
┌pentru e = 1,|E| executa
│ citeste i, j, c;
│ G[i][j] ← c;
│ G[j][i] ← c;     //instructiunea lipseste in cazul digrafului
└■
```

Implementarea în limbaj a subprogramului care creează matricea costurilor unui graf neorientat ia în considerare următoarele declaraţii:

```
#include <algorithm>
#include <stdio.h>
#include <vector>
#define pb push_back
#define f first
#define s second
#define mp make_pair
#define MAXN 10001

using namespace std;

int N, M;
vector <pair <int, int> > L[MAXN];
```

```
#include <stdio.h>
#include <string.h>

#define MAXN 10001
#define INF 0x3f3f

int N, M, G[MAXN][MAXN];
```

Subprogramul *citeste_graf()* preia informaţiile din fişierul text *graf.in* în felul următor: de pe prima linie, numerele *n* de vârfuri şi *m* de muchii, iar de pe următoarele *m* linii, câte trei numere *i, j, c* având semnificaţia muchie între *i* şi *j* de cost *c*.

```
void load(){
  scanf("%d %d\n", &N, &M);
  for (i=1; i<= M; i++){
   scanf("%d %d %d\n",&x, &y,&c);
   G[x].pb(mp(y,c));
   G[y].pb(mp(x,c);
 }
 return;
}
```

```
void citeste_graf(void) {
  int i, j, c;
  freopen("graf.in","r",stdin);

  scanf("%d %d", &N, &M);
  for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++)
     G[i][j] = (i != j) * INF;
  for (; M > 0; M--) {
   scanf("%d %d %d", &i, &j, &c);
   G[i][j] = c;
   G[j][i] = c;
  }
}
```

• *Lista muchiilor* este utilizată în cadrul unor algoritmi, ea memorând, pentru fiecare muchie, cele două vârfuri incidente şi eventual, costul ei.

Implementarea în limbaj a subprogramului care creează lista muchiilor unui graf neorientat ponderat, ia în considerare următoarele declaraţii:

```
#include <algorithm>
#include <stdio.h>
#include <vector>
#define pb push_back
#define f first
#define s second
#define mp make_pair

using namespace std;

int N, M;
vector <pair <int, pair<int, int> > > L;
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_N 101
#define MAX_M 10001

int N, M, E[MAX_M][3];
```

Tabloul *E* reţine lista muchiilor. Subprogramul *citeste_graf()* preia informaţiile din fişierul text *graf.in* în felul următor: de pe prima linie, numărul *n* de vârfuri şi *m* de muchii, iar de pe următoarele *m* linii, câte trei numere *i, j, c* având semnificaţia muchie între *i* şi *j* de cost *c*.

```
void load(){

int x, y, c;

  scanf("%d %d", &N, &M);
  for (int i = 0; i < M; i++){
  scanf("%d %d %d",&x, &y, &c);
  L.pb (mp (c, mp(x, y) ) );

}
```

```
void citeste_graf(void) {
 int i, j, k, c;
 freopen("graf.in","r",stdin);
 scanf("%d %d", &N, &M);
 for (k = 0; k < M; k++) {
  scanf("%d %d %d", &i,&j,&c);
  E[k][0] = i; E[k][1] = j;
  E[k][2] = c;
 }
}
```

2. *Algoritmi pe grafuri*

1.2 Parcurgeri pe grafuri

1. *Parcurgerea grafurilor în adâncime DF - Depth First*

Fie graful *G*=(*V*,*E*), unde *V* este mulţimea nodurilor şi *E* mulţimea muchiilor. Realizaţi un subprogram care să permită afişarea nodurilor în cazul parcurgerii în adâncime-*DF*.

Soluţie:

Printre operaţiile fundamentale efectuate asupra structurilor de date se numără şi traversarea acestora. Această operaţie constă într-o căutarea efectuată asupra tuturor elementelor ei. Cu alte cuvinte, traversarea poate fi privită şi ca un caz special de căutare, deoarece constă în regăsirea tuturor elementelor structurii.

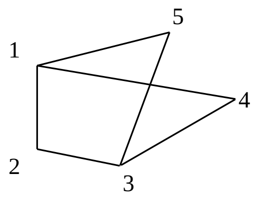
Strategia parcurgerii în adâncime a unui graf neorientat presupune traversarea unei muchii incidente în vârful curent, către un vârf adiacent nedescoperit. Când toate muchiile vârfului curent au fost explorate, se revine în vârful care a condus la explorarea nodului curent. Procesul se repetă până în momentul în care toate vârfurile au fost explorate.

Această strategie a parcurgerii *DF* funcţionează respectând mecanismul *LIFO*. Vârful care este eliminat din stivă nu mai are nici o muchie disponibilă pentru traversare. Aceleaşi reguli se respectă şi la parcurgerea în adâncime a grafurilor orientate (*digrafuri*).

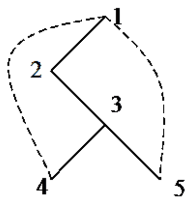
În procesul de parcurgere, muchiile unui graf neorientat se pot împărţi în:

- *muchii de arbore (avans),* folosite pentru explorarea nodurilor; ele constituie un graf parţial format din unul sau din mai mulţi arbori ai parcurgerii *DF*.

- *muchii de întoarcere (înapoi)*, care unesc un nod cu un strămoş al său în arborele *DF*.



Parcurgerea *DF*: 1 2 3 4 5



Muchii de întoarcere: (1,4), (1,5)

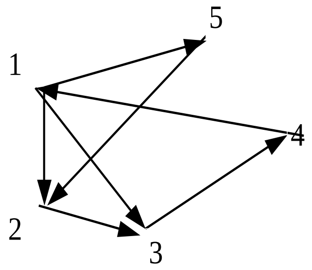
În cazul digrafurilor (grafurilor orientate), se disting următoarele grupe de arce:

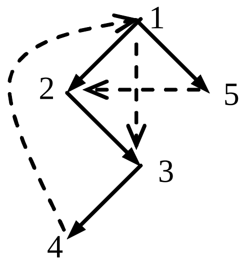
- *arce de arbore*, folosite pentru explorarea vârfurilor; ele constituie un graf parţial format din unul sau din mai mulţi arbori ai parcurgerii *DF*.

- *arce înapoi*, care unesc un nod cu un strămoş al său în arborele *DF*.

- *arce înainte*, sunt arce care nu aparţin arborelui *DF* şi care conectează un vârf cu un descendent al său.

- *arce transversale*, sunt arce care conectează două vârfuri ce nu se află în relaţia ascendent-descendent.

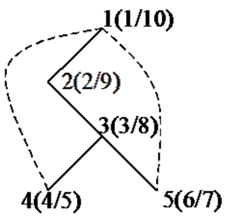




Arce transversale:
(5,2)
Arce înainte:(1,3)
Arce înapoi: (4,1)
Arce de arbore:
(1,2), (1,5) (2,3) (3,4)

În implementarea recursivă a parcurgerii *DF*, prezentată în continuare, se folosesc următoarele tablouri unidimensionale:

- *T*[*N*], în care, pentru fiecare vârf, se va reţine vârful predecesor (părinte) în parcurgere;
- *D*[*N*], în care, pentru fiecare vârf, se memorează momentul "descoperirii", moment care coincide cu momentul de introducere în stivă;
- *F*[*N*], vector în care va memora momentul de "finish" în explorare, care coincide cu momentul extragerii din stivă;
- *Use*[*N*], vector în care se codifică, în timpul parcurgerii, starea unui nod: vizitat sau nevizitat.




```
Valorile vectorilor:

    D=( 1,2,3,4,6)
    F=(10,9,8,5,7)
    T=( 0,1,2,3,3)
```

Fie graful *G=(V,E)*, unde *V* este mulţimea nodurilor şi *E* mulţimea muchiilor. Notăm cu *N* cardinalul lui *V* şi cu *M* cardinalul lui *E*.

```
Depth_First (G=(V,E), D[N], F[N], T[N])    //complexitate O(N+M)
 Use ← False;                             //nici un nod nu este vizitat
 timp ← 0;
 ┌pentru nod ← 1,N executa
 │     ┌daca Not(Use[nod]) atunci Parcurge_df(nod)
 │     └■
 └■

Parcurge_df(nod)
 Use[nod] ← TRUE; timp ← timp+1; D[nod]←timp;
 scrie nod;
 i ← prim_vecin(nod);
 ┌cat_timp lista_vecinilor(nod)≠∅ executa
 │     ┌daca not(Use[i]) atunci
 │     │      T[i]←nod;
 │     │      parcurge_df(i);
 │     └■
 │     i ← urmator_vecin(nod);
 └■
 timp ← timp+1;
 F[nod] ← timp;
```

Implementarea în limbaj a subprogramului ce realizează parcurgerea în adâncime a unui graf, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <vector>
#include <stdio.h>
#include <string.h>
#define MAX_N 1001
#define pb push_back

using namespace std;

vector <int> L[MAXN];
int n, m, i, x, y, nrc,timp;
int T[MAXN], F[MAXN];
bool U[100001];
```

```
#include <stdio.h>
#include <string.h>
#define MAXN 10001

struct lista
{   int nod;
    lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], D[MAX_N], F[MAX_N],
timp;
char U[MAX_N];
```

Ca şi în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacenţă (vezi 2.1.2).

```
void dfs( int x ){

  int i;
  U[x] = true;
  D[x] = ++ timp;
  for(i = 0; i< L[x].size();i++)
         if (!U[L[x][i]]){
                T[L[x][i]] = x;
                dfs(L[x][i]);
         }
  F[x] = ++timp;

}
```

```
void DF(int nod){
 lista *p;

  U[nod] = 1;
  D[nod] = ++timp;
  printf("%d ",nod);
  for (p = G[nod]; p != NULL;
       p = p->urm)
   if (!U[p->nod])
       {
          T[p->nod] = nod;
          DF(p->nod);
        }
    F[nod] = ++timp;
}
```

2. *Parcurgerea grafurilor în lăţime BF - Breadth First*

Fie graful $G=(V,E)$, unde V este mulţimea nodurilor şi E mulţimea muchiilor. Realizaţi un subprogram care să permită afişarea nodurilor în cazul parcurgerii în lăţime-*BF*.

<u>*Soluţie:*</u>

Strategia parcurgerii *BF* funcţionează respectând mecanismul de tip *FIFO*. Ea se bazează pe traversarea tuturor muchiilor disponibile din nodul curent către noduri adiacente nedescoperite, care vor fi astfel vizitate. După acest proces, nodul explorat este scos din coadă, prelucrându-se succesiv toate nodurile ajunse în vârful cozii.

Acest mecanism permite identificarea lanţurilor de lungime minimă de la nodul de start către toate vârfurile accesibile lui din graf. Arborele *BF*, ce cuprinde muchiile traversate în parcurgerea în lăţime, are proprietatea de a fi format doar din lanţuri de lungime minimă, lanţuri ce unesc nodul de start al parcurgerii cu toate vârfurile accesibile acestuia.

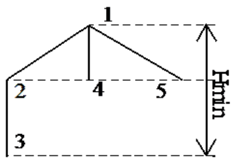
Aceleaşi reguli se respectă şi la parcurgerea în lăţime a grafurilor orientate (*digrafuri*).

Algoritmul lui *Lee* de determinare a lanţului minim dintre două vârfuri ale unui graf se bazează tocmai pe această strategie de traversare în lăţime, nodul de plecare fiind unul dintre cele două vârfuri date.

În implementarea iterativă a parcurgerii *BF*, prezentată în continuare, se folosesc următoarele tablouri unidimensionale:

- *T*[*N*], în care, pentru fiecare vârf, se va reţine vârful predecesor (părinte) în parcurgere;
- *D*[*N*], în care, pentru fiecare vârf, se memorează lungimea lanţului minim către nodul de start;
- *C*[*N*], vector (coadă) în care se va memora ordinea de parcurgere a nodurilor în *BF*;
- *Use*[*N*], vector în care se codifică, în timpul parcurgerii, starea unui nod: vizitat sau nevizitat.

Pentru graful considerat anterior ca exemplu, arborele *BF* este următorul:



Valorile vectorilor:

$C = (1, 2, 4, 5, 3)$
$D = (0, 1, 2, 1, 1)$
$T = (0, 1, 2, 1, 1)$

Fie graful $G=(V,E)$, unde V este mulţimea nodurilor şi E, mulţimea muchiilor. Notăm cu N cardinalul lui V şi cu M cardinalul lui E.

```
Breadth_First (G=(V,E), C[N] T[N], Nod_Start)
                                     //*complexitate: O(M+N)*//
Use ← FALSE;
Coada ⇐ {Nod_Start}   //nodul de start este introdus in coada
Use[nod] ← True;
┌cat_timp Coada ≠ ϕ executa
│     v ← varf_Coada; i ← prim_vecin(v);
│     ┌cat_timp lista_vecinilor(v) ≠ ϕ executa
│     │      ┌daca not(Use[i]) atunci
│     │      │ T[i] ← v; Use[i] ← TRUE;
│     │      │ D[i] ← D[v]+1;
│     │      │ Coada ⇐ {i};
│     │      └■
│     │      i ← urmator_vecin(v);
│     └■
│     COADA ⇒ v;
└■
```

Implementarea în limbaj a subprogramului ce realizează parcurgerea în lăţime a unui graf, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <queue>
#include <vector>
#include <stdio.h>
#include <cstring>
#define MAXN 10001
#define pb push_back

using namespace std;

vector <int> G[100001];
queue <int> Q;
int N, M, i,x, y, P, Lg[MAXN], T[MAXN];
bool U[MAXN];
```

```
#include <stdio.h>
#include <string.h>
#define MAX_N 1001
struct lista
{   int nod;
    lista *urm;
} *G[MAX_N];

int N, M;
int T[MAX_N], D[MAX_N], C[MAX_N];
char U[MAX_N];
```

Ca şi în pseudocod, s-a preferat implementarea în manieră iterativă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacenţă.

```

void bfs(int x){

  Q.push(x); Lg[x] = 0; U[x] = true;

  while (!Q.empty()){
    x = Q.front();
    for( i = 0; i < G[x].size(); i++)
        if (!U[G[x][i]])  {
             Q.push(G[x][i]);
             Lg[G[x][i]] = Lg[x]+1;
             U[G[x][i]] = true;
             T[G[x][i]] = x;
            }
    Q.pop();
    }
}
```

```
void bfs(int start) {
 lista *p;
 int nod, st, dr;
  st = dr = 1;
 C[st]=start; U[start] = 1;
 for (D[start]=0;st<=dr;st++)
 {nod=C[st];
  for (p = G[nod];p != NULL;
       p = p->urm)
    if (!U[p->nod])
     {
      U[C[++dr] = p->nod] = 1;
      D[p->nod] = D[nod]+1;
      T[p->nod] = nod;
     }
 }
}
```

3. *Ciclu eulerian*

Fie $G=(V,E)$, graf neorientat conex, în care fiecare nod are gradul par. Notăm cu N cardinalul lui V şi cu M, cardinalul lui E. Să se determine un ciclu eulerian al grafului G, altfel spus, un ciclu simplu de lungime M.

Exemplu:
Considerând graful G în care N=6, M=10 şi arcele: (1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5), (4,6), (5,6), se va afişa: 1, 3, 5, 6, 4, 5, 2, 4, 3, 2, 1.

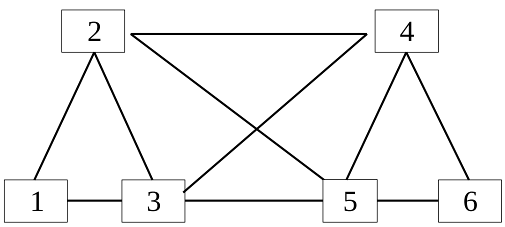
<u>Soluţie</u>:
Pentru a construi un ciclu eulerian se va parcurge graful folosind o strategie asemănătoare cu cea a parcurgerii în adâncime a unui graf, strategie care respectă mecanismul *LIFO*.

Înaitarea către un vârf adiacent vârfului curent se va face simultan cu eliminarea muchiei respective. În acest fel, nodurile nu vor mai fi marcate (ca la parcurgerea *DF*) pentru a putea fi vizitate de mai multe ori.

De fiecare dată când nodul curent este eliminat din stivă, acesta este afişat sau salvat într-o coadă. Această coadă va reţine, în ordine, nodurile ciclului eulerian. Procesul se repetă până în momentul în care stiva devine vidă, deci toate muchiile au fost traversate.

```
Ciclu_Euler (Nod)
                                                     //*complexitate: O(M+N)*//
i ← prim_vecin(nod);
┌cat_timp lista_vecinilor(nod)≠ϕ executa
│       elimin(i,nod)           //elimin pe i din vecinii lui nod
│       elimin(nod,i)           //elimin pe nod din vecinii lui i
│       ciclu_euler(i)
│        i ← urmator_vecin(nod);
└■
Coada ⇐ {nod};
```

Considerăm graful următor şi desemnăm ca nod de start pentru construirea ciclului eulerian, nodul 1:



Pasul 1:
Se parcurge începând cu nodul 1
Stiva=(1,2,3,1)
Coada = ϕ

Pasul 2:
Iese din stivă nodul 1, salvându-se în coadă:
Stiva = (1,2,3)
Coada = (1)

Pasul 3:
Se continuă parcurgerea:
Stiva = (1,2,3,4,2,5,3)
Coada = (1)

Pasul 4:
Iese din stivă nodul 3, salvându-se în coadă:
Stiva = (1,2,3,4,2,5)
Coada = (1,3)

Pasul 5:
Se continuă parcurgerea:
Stiva = (1,2,3,4,2,5,4,6,5)
Coada = (1,3)

Pasul 6:
Iese din stivă nodul 5, salvându-se în coadă:
Stiva = (1,2,3,4,2,5,4,6)
Coada = (1,3,5)

Pasul 7:
Toate nodurile sunt extrase din stivă şi introduse în coadă. La finalul algoritmului:
Stiva = ϕ
Coada = (1,3,5,6,4,5,2,4,3,2,1).

Implementarea în limbaj a subprogramului ce realizează determinara unui ciclu eulerian, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <cstdio>
#include <vector>
#include <list>
#include <cstring>
#define MAXN 1001

using namespace std;

vector<int> G[MAXN];
list <int> Q;
int nc, nr, i, k, N, M, x, y;
bool ok; char s[100000];
```

```
#include <stdio.h>

#define MAX_N 1001
#define MAX_M 100001

int N, M, C[MAX_M], nc;
char G[MAX_N][MAX_N];
```

Ca şi în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul matricei de adiacenţă. Datorită acestui mod de memorare, implementarea în limbaj conduce la o complexitate pătratică O(N^2).

```
void euler(int x) {
vector<int>::iterator it;

Q.pb(x);
while (!Q.empty()){
 x = Q.front();
 if (G[x].empty()){
   Q.pop_front();
   printf("%d ", x);
 }
 else {
  int i = G[x].back();
  Q.push_front(i);
  G[x].pop_back();
  for(it=G[i].begin();it!=G[i].end();++it)
    if(*it==x) {G[i].erase(it); break;}
  }
 }
}
```

```
void euler(int nod)
{
int urm;

 for (urm = 1;urm <= N;urm++)

  if (G[nod][urm])
     {
      G[nod][urm] = 0;
      G[urm][nod] = 0;
      euler(urm);
     }
    C[nc++] = nod;
}
```

În cazul în care se doreşte un lanţ eulerian, nu un ciclu, se poate aplica acelaşi algoritm începând dintr-un nod cu grad impar, dacă există vreunul. Un lanţ eulerian se poate forma numai dacă există zero sau două noduri cu grad impar.

4. *Matricea drumurilor - Algoritmul lui Warshall*

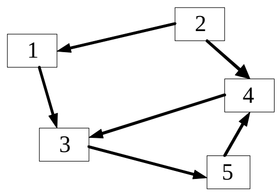
Fie $G=(V,E)$, graf orientat cu n vârfuri şi m arce. Să se determine, pentru orice pereche de vârfuri $x, y \in V$, dacă există sau nu un drum format din unul sau din mai multe arce între x şi y.

<u>*Soluţie*</u>:

Numim matrice a drumurilor sau a închiderii tranzitive, o matrice pătratică $D(N,N)$ în care elementul $D[i,j]$=1, dacă există drum între nodul i şi nodul j şi 0, în caz contrar.

De exemplu, pentru graful ilustrat alăturat, matricea drumurilor este următoarea:

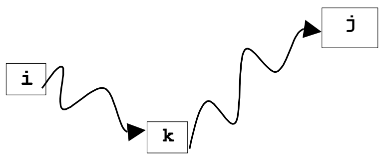
```
0 0 1 1 1
1 0 1 1 1
0 0 1 1 1
0 0 1 1 1
0 0 1 1 1
```



Algoritmul *Warshall* construieşte matricea închiderii tranzitive D, plecând de la matricea de adiacenţă G.

Iniţial, matricea *D* indică prezenţa drumurilor de lungime 1 între orice pereche de vârfuri adiacente.

Algoritmul parcurge de *n* ori matricea *D*, câte o dată pentru fiecare nod *k* al grafului. Astfel, pentru fiecare nod *k*, între orice pereche de noduri *i* şi *j* din graf, fie s-a descoperit deja un drum (*D*[*i*,*j*]=1), fie acesta poate fi construit prin concatenarea drumurilor de la *i* la *k* şi de la *k* la *j*, dacă acestea există.



Luând în considerare caracteristica logică a valorilor elementelor matricei *D*, atunci regula descrisă mai sus poate fi exprimată astfel:
D[i,j]=D[i,j] OR {D[i,k] AND D[k,j]}

```
Warshall (G=(V,E))
                                        //*complexitate: O(N³)*//
D ← G;                               //initializare matrice drumuri
┌pentru k ← 1,n executa
│      ┌ pentru i ← 1,n executa
│      │      ┌ pentru j ← 1,n executa
│      │      │ D[i,j] ← D[i,j] SAU {D[i,k] SI D[k,j]}
│      │      └■
│      └■
└■
```

Implementarea în limbaj a subprogramului ce realizează determinarea matricei închiderii tranzitive, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#define MAX_N 101
#define MAX_M 1001
int N, M, C[MAX_M], nc;
char G[MAX_N][MAX_N],D[MAX_N][MAX_N];
```

Ca şi în pseudocod, graful este considerat a fi memorat cu ajutorul matricei de adiacenţă. Complexitatea algoritmului lui *Warshall* este cubică $O(N^3)$.

```
void Warshall()
{
int i,j,k;
for (i = 1; i <= N; i++)
 for (j = 1; j <= N; j++)
    D[i][j] = G[i][j];
for (k = 1; k <= N; k++)
 for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++)
   if (!D[i][j])
       D[i][j] = D[i][k] && D[k][j];
}
```

5. *Componente conexe*

Fie *G*=(*V*,*E*) un graf neorientat. Se doreşte determinarea componentei conexe cu număr maxim de noduri din *G*. Componenta va fi identificată printr-unul dintre vârfurile sale şi numărul de vârfuri din care este formată.

Exemplu: Considerând graful *G* în care *N*=6, *M*=6 şi arcele: (1,2), (3,4), (3,5), (4,5), (4,6), (5,6) se va afişa: 3 4 (nodul 3 face parte din componenta conexă formată din 4 noduri).

Soluţie:

Problema va fi rezolvată prin determinarea tuturor componentelor conexe ale grafului şi identificarea componentei cu număr maxim de noduri. Ne reamintim că o componentă conexă este un subgraf maximal în raport cu proprietatea de conexitate.

Pentru a descompune graful în componente conexe, vom proceda în felul următor: vom realiza o parcurgere *DF* din nodul 1, determinându-se componenta conexă din care acestea fac parte. Vom continua algoritmul cu o nouă parcurgere efectuată dintr-un nod nevizitat anterior. Procedeul continuă până când s-au vizitat toate nodurile.

Algoritmul de descompunere în componente conexe a unui graf neorientat este prezentat în pseudocod, în continuare.

```
Componente_conexe (G=(V,E))          //*complexitate: O(M+N)*//
nrc ← 0;                             // nr de componente conexe
Use ← False;                         // nici un nod selectat
┌pentru i ← 1,n executa
│    ┌daca Not(Use[i]) atunci
│    │  nrc ← nrc + 1;
│    │  parcurge_df(i)
│    └■
└■
```

Pentru a identifica cea mai numeroasă componentă conexă, vom determina în cadrul fiecărei parcurgeri *DF,* numărul de noduri selectate.

6. ***Tare-conexitate***

Fie $G=(V,E)$ un graf orientat. Realizaţi un program care afişează vârfurile fiecărei componente tare conexe în care se descompune graful *G*.

Exemplu: Considerând digraful *G* în care $N=5$, $M=6$ şi arcele: (1,2), (1,5), (2,3), (3,4), (3,5), (4,1), se vor afişa două componente tare conexe:

1 4 3 2
5

<u>*Soluţie:*</u>

În cazul digrafurilor (grafurilor orientate), o componentă tare conexă reprezintă un subgraf maximal în care, oricare două vârfuri sunt accesibile unul din celălalt.

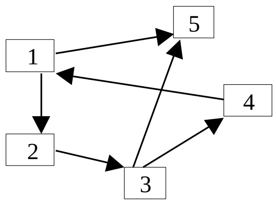
În cadrul unei componente tare conexe, inversarea sensului de deplasare nu implică blocarea accesului către vreunul din vârfurile sale.

Algoritmul propus identifică componenta tare conexă a unui vârf ca fiind intersecţia dintre mulţimea nodurilor accesibile din el în graful iniţial *G* şi mulţimea nodurilor accesibile din el în graful transpus *Gt* (obţinut prin inversarea sensurilor arcelor).

Acest algoritm se bazează pe parcurgerea *DF* a celor două grafuri, de aici şi complexitatea liniară a acestuia $O(N+M)$. Operaţiile efectuate sunt:

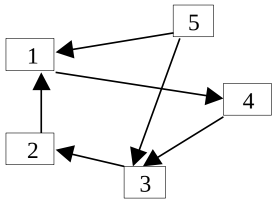
– Parcurgerea *DF* a grafului iniţial (*G*) pentru memorarea explicită a stivei ce conţine vârfurile grafului în ordinea crescătoare a timpilor de finish.
– Parcurgerea *DF* a grafului transpus (*Gt*) începând cu ultimul vârf reţinut în stivă către primul. Parcurgerea se reia din fiecare vârf rămas nevizitat la parcurgerile anterioare. Vârfurile fiecărui arbore *DF* obţinut la acest pas reprezintă câte o componentă tare conexă.

Pentru graful:



Pasul 1:
Stiva *DF* cuprinzând nodurile în ordinea crescătoare a timpilor de finish este:

St=(4, 5, 3, 2, 1).



Pasul 2:

Parcurgerea *DF* pe graful transpus începând cu *St*[*n*] accesează vârfurile din prima componentă tare conexă: {1,4,3,2}.

Parcurgerea *DF* din primul vârf rămas neselectat *St*[2] accesează vârfurile celei de a doua componente tare conexe: {5}.

Implementarea în limbaj a subprogramelor prezentate în continuare ia în considerare următoarele declaraţii:

```
#include <cstdio>
#include <vector>
#include <cstring>
#define pb push_back
#define MAXN 10001
using namespace std;

vector<int> G[MAXN],GT[MAXN];
int nc, nr, i, k, N, M, x, y, St[MAXN];
bool sel[MAXN];
```

```
#include <stdio.h>
#include <string.h>
#define MAX_N 10001

struct lista
{   int nod;
    lista *urm;
} *G[MAX_N], *GT[MAX_N];
int N, M, ST[MAX_N], nst;
char U[MAX_N];
```

Tabloul *GT* va reţine graful transpus asociat lui *G*. Subprogramul *citeste_graf* preia datele referitoare la graful *G* şi construieşte matricele de adiacenţă ale celor două grafuri *G* şi *GT*.

```
void DF(int x){
int i;
  sel[x]=true;
  for(i = 0; i < G[x].size(); ++i)
    if (!sel[G[x][i]]) DF(G[x][i]);
  St[++nr]=x;
}

void DFT(int x, bool k){
int i;
  sel[x]=true;
  if (k) printf("%d ", x);
  for(i = 0; i < GT[x].size(); ++i)
   if (!sel[GT[x][i]])
       DFT(GT[x][i], k);
}

int main(){
. . . . .
    scanf("%d %d\n", &N, &M);
    for (i=1; i<=M; ++i){
      scanf("%d %d\n", &x, &y);
      G[x].pb(y); GT[y].pb(x);
    }
    nr=0; nc=0;
    memset(sel, false,sizeof(sel));
    for(i=1; i<=N; i++)
        if (!sel[i]) DF(i);
    memset(sel, false,sizeof(sel));
    for(i =N; i>0; --i)
        if (!sel[St[i]]) {
            DFT(St[i], false);
            nc++;
        }
    printf("%d\n", nc);
    memset(sel, false,sizeof(sel));
    for(i=N; i>0; --i)
        if (!sel[St[i]]) {
            DFT(St[i], true);
            printf("\n");
        }
    return 0;
}
```

```
void DF1(int nod)
{
 lista *p;

 U[nod] = 1;
 for (p = G[nod]; p != NULL;
      p = p->urm)
   if (!U[p->nod])
          DF1(p->nod);
 ST[nst++] = nod;
}

void DF2(int nod)
{
 lista *p;

 U[nod] = 1;
 printf("%d ", nod);
 for (p = GT[nod]; p != NULL;
      p = p->urm)
   if (!U[p->nod])
          DF2(p->nod);
}

void citeste_graf(void)
{......}

int main(void)
{int i; citeste_graf();
 for (i = 1; i <= N; i++)
   if (!U[i]) DF1(i);
 memset(U, 0, sizeof(U));
 for (i = N-1; i >= 0; i--)
   if (!U[ST[i]]) {
            DF2(ST[i]);
            putchar('\n');
   }  return 0;
}
```

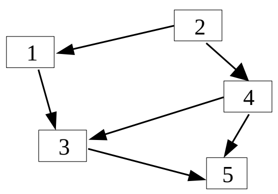
7. ***Sortarea topologică a unui digraf***

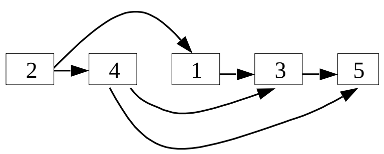
Fie *G*=(*V*,*E*) un graf orientat aciclic. Realizaţi un program care ordonează vârfurile grafului după următoarea regulă: pentru orice arc (*x*,*y*)∈ *E*, vârful *x* apare, în şirul ordonat, înaintea vârfului *y*.

Exemplu: Considerând digraful *G* în care *N*=5, *M*=6 şi arcele: (1,3), (2,1), (2,4), (3,5), (4,3), (4,5), se va afişa 2 4 1 3 5.

<u>*Soluţie:*</u>

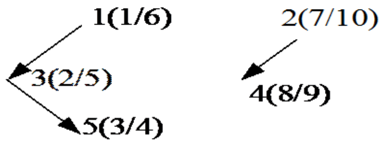
Pentru a înţelege mai bine modul de ordonare a vârfurilor, realizat de sortarea topologică, să urmărim graful din exemplu:





O ordonare corectă a vârfurilor este: 2 4 1 3 5

Presupunem că se realizează o parcurgere *DF* a grafului prezentat. Vectorii *D* şi *F* care reţin timpii de intrare, respectiv timpii de ieşire din stivă sunt:



D=(1, 7, 2, 8, 3)
F=(6, 10, 5, 9, 4)

Dacă vârfurile sunt reţinute într-o listă, în ordinea ieşirii lor din stivă, atunci această listă va conţine: *St*=(5, 3, 1, 4, 2). Afişarea conţinutului ei în ordine inversă, adică în ordinea inversă timpilor de finish, va reprezenta ordinea vârfurilor obţinută prin sortarea topologică 2, 4, 1, 3, 5.

Să presupunem vârfurile digrafului ca nişte evenimente şi fiecare arc (*x*,*y*) considerăm că indică faptul că evenimentul *x* trebuie să se execute înaintea evenimentului *y*. Plecând de la aceste considerente, deducem că sortarea topologică induce o ordine asupra evenimentelor, astfel încât acestea să poată fi executate.

Să asociem următoarele evenimente grafului anterior: Nodul 1 - Mă îmbrac; Nodul 2 - Mă trezesc; Nodul 3 - Servesc masa; Nodul 4 - Mă spăl; Nodul 5 - Plec din casă. Atunci ordinea evenimentelor dată de sortarea topologică este: Mă trezesc, mă spăl, mă îmbrac, servesc masa, plec din casă.Fie graful *G=(V,E)*, unde *V* este mulţimea vârfurilor şi *E*, mulţimea arcelor. Notăm cu *N* cardinalul lui *V* şi cu *M*, cardinalul lui *E*.

```
Sortare_topologică_DF (G=(V,E),St[N])  //complexitate O(N+M)
 nr ← 0;                  //nr de varfuri extrase din stiva
 Use ← False
  ┌pentru nod ← 1,N executa
  │     ┌daca Not(Use[nod]) atunci Parcurge_df(nod)
  │     └■
  └■
  ┌pentru i ← N,1,-1 executa scrie St[i]
  └■
Parcurge_df(nod)
 Use[nod] ← TRUE;  i ← prim_vecin(nod);
  ┌cat_timp lista_vecinilor(nod)≠∅ executa
  │     ┌daca not(Use[i]) atunci parcurge_df(i);
  │     └■
  │     i ← urmator_vecin(nod);
  └■
 nr ← nr+1; St[nr] ← nod;
```

Implementarea în limbaj a subprogramului ce realizează sortarea topologică a unui digraf aciclic, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <vector>
#include <stdio.h>
#include <string.h>
#include <algorithm>
#define pb push_back

using namespace std;
vector< int > L[50005], C;
vector <int> :: iterator it;
int n, m, i, x, y ; bool sel[50005];
```

```
#include <stdio.h>
#define MAX_N 1001
struct lista
{   int nod;
    lista *urm;
} *G[MAX_N];
int N, M, ST[MAX_N], nst;
char U[MAX_N];
```

Ca şi în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacenţă

```
void load(){
   scanf("%d %d\n", &n, &m);
   for (i=1; i<= m; i++){
   scanf("%d %d\n",&x, &y);
   L[x].pb(y);
  }
}

void dfs(int x){
 vector <int> :: iterator it;
 sel[x]=true;
 for(it=L[x].begin();it!=L[x].end();it++)
       if (!sel[*it]) dfs(*it);
 C.pb(x);
}

int main(){
...
 load();
 memset(sel, false, sizeof(sel));
 for (i=1;i<=n; i++)
       if (!sel[i]) dfs(i);

 reverse(C.begin(),C.end());

 for(it=C.begin(); it!=C.end(); it++)
   printf("%d ",*it);
printf("\n");
}
```

```
void DF(int nod)
{
  lista *p;
  U[nod] = 1;
  for (p = G[nod]; p != NULL;
       p = p->urm)
   if (!U[p->nod])
             DF(p->nod);
  ST[nst++] = nod;
}

int main(void)
{ int i;
  citeste_graf();
  for (i = 1; i <= N; i++)
    if (!U[i]) DF(i);
  for (i = N-1; i >= 0; i--)
      printf("%d ", ST[i]);
    return 0;
}
```

O altă modalitate de implementare a sortării topologice ţine cont de observaţia că, la un moment dat, un eveniment poate fi executat, dacă nu există nici un alt eveniment de care acesta depinde, care să nu fi fost executat anterior.

Revenind la modelarea pe digrafuri, gradul interior al unui vârf reprezintă tocmai numărul de evenimente (vârfuri) de care acesta depinde.

Iniţial, în graf trebuie indentificate vârfurile cu gradul interior nul, ele fiind plasate primele într-o coadă care va reţine, la finalul algoritmului, ordinea vârfurilor date de sortarea topologică. Vom parcurge graful în lăţime, pornind din primul vârf plasat în coadă. La fiecare pas, vom decrementa gradele tuturor vârfurilor adiacente spre interior cu acestea. În coadă vor fi adăugate doar vârfurile vecine cu cel curent, neselectate anterior şi care au gradul interior nul. Algoritmul se încheie când toate vârfurile au fost plasate în coadă.

Pentru o mai bună înţelegere, să privim exemplul următor în care vectorul *Deg(N)* reţine gradele interioare ale vârfurilor, iar coada *C(N)* va memora vârfurile în ordinea indusă de sortarea topologică:

Iniţial vectorii conţin:

Deg=(1,0,2,1,2)

C=(2)

a)Se parcurge în lăţime din nodul 2

Deg=(0,0,2,0,2)

C=(2,4,1)

b)Se parcurge în lăţime din nodul 4

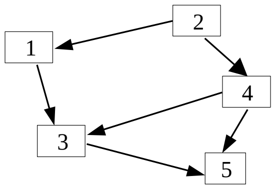
Deg=(0,0,1,0,1)

C=(2,4,1)

d)Se parcurge în lăţime din nodul 1

Deg=(0,0,0,0,1)

C=(2,4,1,3)



e) Se parcurge în lăţime din nodul 3

Deg=(0,0,0,0,0)

C=(2,4,1,3,5)

Implementarea în limbaj a subprogramului ce realizează sortarea topologică a unui digraf aciclic, folosind parcurgerea *BF*, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <vector>
#include <stdio.h>
#include <cstring>
#include <queue>
#include <algorithm>
#define pb push_back

using namespace std;
vector <int > G[50005];
queue <int > Q;
vector <int> :: iterator it;
int N, M, i, x, y, deg[50005];
bool U[50005];
```

```
#include <stdio.h>
#define MAX_N 10001
struct lista
{  int nod;
   lista *urm;
} *G[MAX_N];

int N, M, C[MAX_N], deg[MAX_N];
char U[MAX_N];
```

Graful este considerat a fi memorat cu ajutorul listelor de adiacenţă.

```
void load(){
   scanf("%d %d\n", &N, &M);
   for (i=1; i<= M; i++){
     scanf("%d %d\n",&x, &y);
     G[x].pb(y);
     deg[y]++;
  }
}

void sort_BF() {
 vector<int>:: iterator it;
 int i, x;

 for (i = 1; i <= N; i++)
  if (deg[i] == 0){
     Q.push(i); U[i]=true;
  }

while (!Q.empty()) {
 x=Q.front();
 printf("%d ", x);
 for(it=G[x].begin();it!= G[x].end(); it++)
  {
   deg[*it]--;
   if (!U[*it] && deg[*it] == 0){
     Q.push(*it);
     U[*it] = true;
   }
  }
  Q.pop();
 }
}
```

```
void sort_BF(void)
{lista *p;
 int i, st = 0, dr = -1;
 for (i = 1; i <= N; i++)
  if (deg[i] == 0){
   C[++dr] = i;
   U[i]=1;
  }
 for (; st <= dr; st++){
  p=G[C[st]];
  while (p != NULL){
   deg[p->nod]--;
   if (!U[p->nod] &&
       deg[p->nod] == 0){
     C[++dr] = p->nod;
     U[p->nod] = 1;
    }
    p = p->urm;
  }
 }
}
```

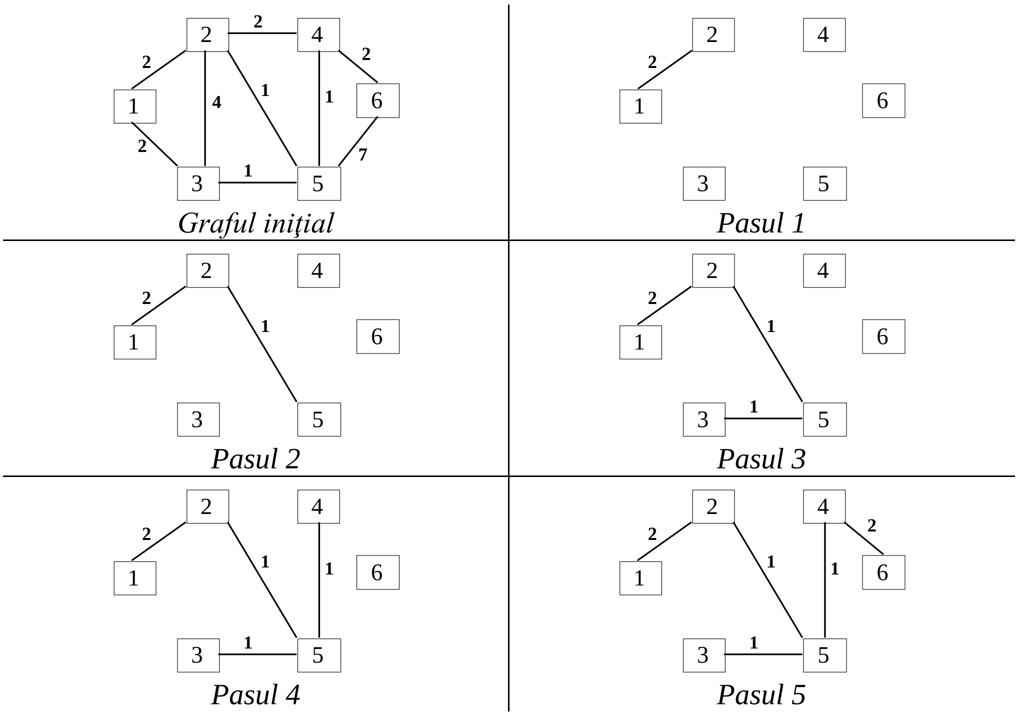
8. ***Arbore parţial de cost minim (A.P.M) – Algoritmul lui Prim***

Fie *G*=(*V*,*E*) un graf neorientat conex, cu costuri asociate muchiilor. Un arbore parţial al lui *G* este un graf parţial conex şi fără cicluri. Realizaţi un program care determină un arbore parţial de cost minim, adică un arbore parţial pentru care suma costurilor tuturor muchiilor sale este minimă.

Soluţie:

Algoritmul lui Prim construieşte arborele parţial de cost minim, pornind de la un nod oarecare considerat rădăcină. La fiecare pas al algoritmului, la arbore se va adăuga un nou vârf. Acesta are proprietatea că este conectat de unul dintre vârfurile prezente deja în arbore, printr-o muchie de cost minim.

Să privim modul de construcţie al *A.P.M.* cu ajutorul algoritmului lui *Prim* pe graful următor, considerând ca rădăcină nodul 1:



Graful iniţial | *Pasul 1* | *Pasul 2* | *Pasul 3* | *Pasul 4* | *Pasul 5*

S-a obţinut un A.P.M plecând de la nodul 1, costul acestuia fiind 7.

Transcrierea în pseudocod a algoritmului foloseşte următoarele structuri de date:

- tabloul *D*, în care elementul *D[x]* va reţine costul minim prin care putem „lega” nodul *x*, neconectat la arbore, de orice nod al arborelui.
- tabloul *T*, în care elementul *T[x]* va reţine nodul din arbore de care nodul *x* a fost conectat cu costul *D[x]*.
- Lista *Q* conţine pe tot parcursul algoritmului nodurile grafului *G* neconectate la arbore.

```
Prim (G=(V,E,cost), rad)                    //complexitate O(N*N)
  Q ← V;      //lista Q contine initial toate nodurile din G
  D ← ∞;
  D[rad] ← 0; T[rad] = 0;
  ┌cat_timp (Q ≠ ∅) executa
  │  x ← minim (Q) ;
  │  Q ⇒ {x}
  │  ┌pentru fiecare y ∈ Q, adiacent cu x executa
  │  │  ┌daca  (cost[x,y] < d[y] ) atunci
  │  │  │    T[y] = x;
  │  │  │    D[x] = cost[x,y];
  │  │  └■
  │  └■
  └■
```

Implementarea în limbaj a algoritmului lui *Prim*, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#include <stdio.h> //O(N*N)
#include <string.h>
#define MAXN 101
#define INF 3000000
int N, M, R, C[MAXN][MAXN], D[MAXN],
T[MAXN], cost;
bool U[MAXN];
```

Graful este memorat cu ajutorul matricei costurilor. Parametrul *x* indică nodul desemnat ca rădăcină a arborelui parţial de cost minim. Lista *Q* a fost codificată cu ajutorul vectorului *Use*, astfel *Use*[*i*]=*true*, dacă nodul *i* aparţine arborelui şi *false*, în caz contrar.

Vectorul *D* s-a iniţializat cu valorile plasate pe linia *x* a matricei ponderilor deoarece, la prima iteraţie a algoritmului, arcele minime prin care un vârf din graf poate fi conectat la rădăcina *x* sunt chiar costurile arcelor ce pleacă din *x*.

```
void prim(int x) {
 int i, min, nod;

 memset(U, false, sizeof(U));
 memset(T, 0, sizeof(T));

 for (i = 1; i <= N; i++)
   D[i]=C[x][i], T[i]=x;
 U[x] = 1;

 while (1)
 {
  min = INF; nod = -1;
  for (i = 1; i <= N; i++)
     if (!U[i] && min> D[i])
          min = D[i], nod = i;

  if (min == INF) break;

  U[nod] = 1;
  cost += D[nod];
  printf("%d %d\n", T[nod], nod);

  for (i = 1; i <= N; i++)
    if (D[i] > C[nod][i])
      {
        D[i] = C[nod][i];
        T[i] = nod;
       }
  } printf("%d\n", cost);
}
```

9. ***Arbore parţial de cost minim (A.P.M) – Algoritmul lui Kruskal***

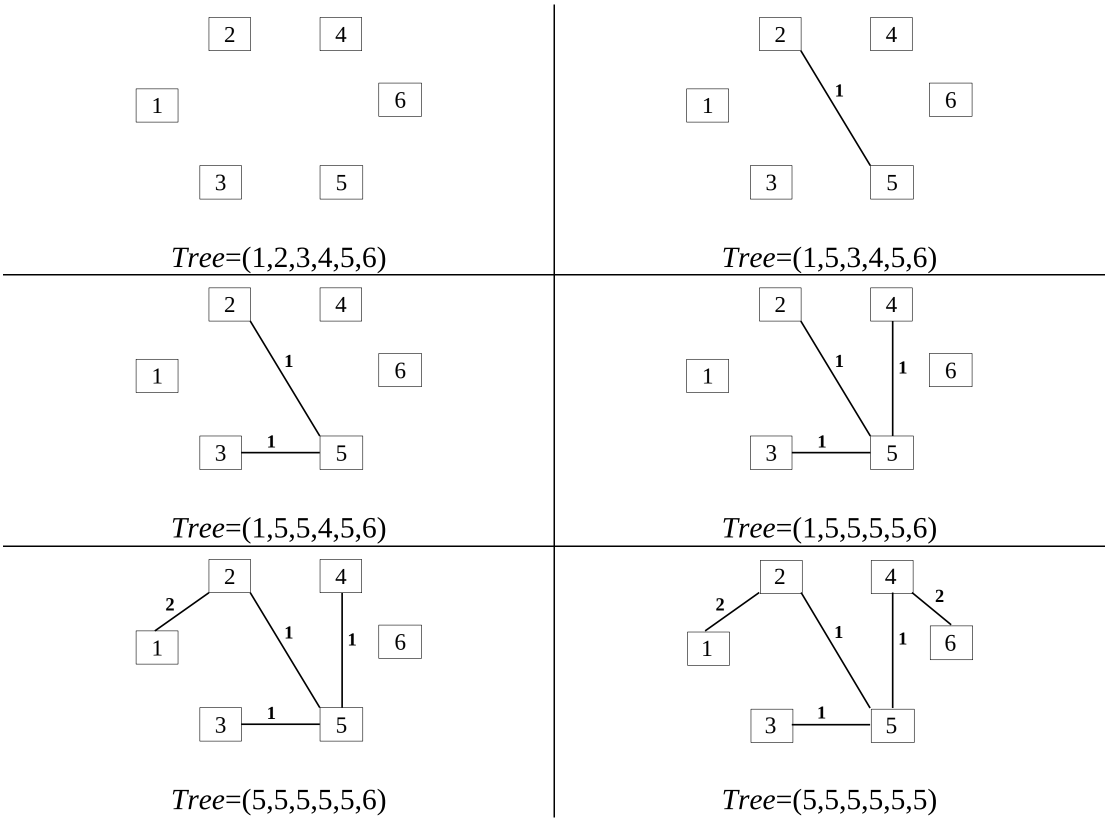
Fie *G*=(*V*,*E*) un graf neorientat conex, cu costuri asociate muchiilor. Un arbore parţial al lui *G* este un graf parţial conex şi fără cicluri. Realizaţi un program care determină un arbore parţial de cost minim, adică un arbore parţial pentru care suma costurilor tuturor muchiilor sale este minimă.

<u>Soluţie</u>:

Considerăm *N* cardinalul mulţimii *V* şi *M* cardinalul mulţimii *E*. Algoritmul lui Kruskal construieşte arborele parţial de cost minim, pornind de la *N* arbori disjuncţi, notaţi T_1, T_2... T_N. Fiecare vârf al grafului defineşte la momentul iniţial, câte un arbore. La fiecare pas al algoritmului vor fi aleşi doi arbori care se vor unifica. În finalul algoritmului se va obţine un singur arbore care va constitui un *APM* al grafului *G*. Proprietatea de *acoperire minimă* a arborelui rezultat este dată de modul de alegere a celor doi arbori care vor fi unificaţi la fiecare etapă a algoritmului. Regula respectată este următoarea:

Se identifică muchia de cost minim, nefolosită anterior, şi care are vârfurile extreme în doi arbori disjuncţi. În felul acesta, prin unirea celor doi arbori va rezulta tot un arbore (nu se vor crea cicluri).

Să privim modul de construcţie al *A.P.M.* cu ajutorul algoritmului lui *Kruskal* pe graful luat ca exemplu la prezentarea algoritmului lui *Prim*:
Vectorul *Tree* codifică arborii disjuncţi astfel *Tree*[*x*]=*y* semnifică faptul că vârful *x* se află în arborele cu numărul de ordine *y*.



S-a obţinut un A.P.M plecând de la nodul 1, costul acestuia fiind 7.

În transcrierea algoritmului în pseudocod, mulţimea *A* desemnează muchiile arborelui de acoperire minimă. Subprogramul *reuneste(x,y)* realizează unificarea arborilor disjuncţi în care se regăsesc nodurile *x* şi *y*.

```
Kruskal (G=(V,E,cost))          //complexitate O(N*M)
 A ← ∅;                      //lista muchilor din A.P.M
┌pentru orice varf i din V executa
│  Tree[i] ← i
└■
Sortare a listei muchiilor ∊ E, crescator dupa cost
┌pentru fiecare (x,y) ∊ E executa
│    ┌daca  Tree[x]≠Tree[y] atunci
│    │   A U {(x,y)};   reuneste(x,y);
│    └■
└■
  returneaza A
```

Implementarea în limbaj a algoritmului lui *Kruskal*, prezentată în continuare, ia în considerare următoarele declaraţii:

```
#define pb push_back    // O(MlogN + MlogM)
#define f first      //utilizand paduri disjuncte
#define s second
#define mp make_pair

using namespace std;

int N, M, T[200010], x, y, c, i;
vector <pair <int, int> > Sol;
vector <pair <int, pair <int, int> > > L;
```

```
#include <stdio.h> //O(M*N)
#include <stdlib.h>
#include <string.h>
#define MAX_N 101
#define MAX_M 1001
#define INF 0x3f3f
struct muchie
{ int x, y, c;} e[MAX_M];
int N, M, T[MAX_N], cost;
```

Vectorul *T* are aceeaşi semnificaţie ca a vectorului *Tree* prezentat anterior în exemplu şi în pseudocod.

```
int Rad(int nod)
{
   if (T[nod] != nod)
       T[nod] = Rad(T[nod]);
   return T[nod];
}

int main()
{
. . . .

scanf("%d %d", &N, &M);
for (i = 0; i < M; i++){
 int x, y, c;
 scanf("%d %d %d", &x,&y,&c);
 L.pb(mp(c, mp(x, y)));
}

sort(L.begin(), L.end());

for (int i = 0; i <= N; i++)
             T[i]=i;

int Cost = 0;

for (i = 0; i < M; i++){
  int n1 = L[i].s.f;
  int n2 = L[i].s.s;

if (Rad(n1) != Rad(n2)){
       Cost += L[i].f;
       Sol.pb(mp(n1, n2));
       T[T[n2]] = T[n1];
  }
}

printf("%d\n", Cost);
printf("%d\n", Sol.size());
for (i=0;i < Sol.size(); i++)
  printf("%d %d\n", Sol[i].f, Sol[i].s);

  return 0;
}
```

```
void reuneste(int i, int j)
{ int k;
  i = T[i];
  j = T[j];
  if (i == j) return;
  for (k = 1; k <= N; k++)
   if (T[k] == i) T[k] = j;}

int comp_muchie(const void *i,
                const void *j)
{  return ((int *) i)[2] -
          ((int *) j)[2];
}

void kruskal(void)
{
 int i, j, k, c;

qsort(e, M, sizeof(e[0]),
       comp_muchie);

 for (i=1;i<=N;i++) T[i]=i;

 for (k = 0; k < M; k++){
   i = e[k].x; j = e[k].y;
   c = e[k].c;
   if (T[i]==T[j]) continue;
   reuneste(i, j);
   cost += c;
   printf("%d %d %d\n",i,j,c);
  }

 printf("Cost minim = %d\n",
        cost);
}
```

10. *Puncte de articulaţie - critice*

Un nod dintr-un graf $G=(V, E)$ neorientat conex este punct de articulaţie (critic), dacă şi numai dacă prin eliminarea lui, împreună cu muchiile incidente acestuia, se pierde proprietatea de conexitate. Realizaţi un program care determină mulţimea punctelor critice dintr-un graf.

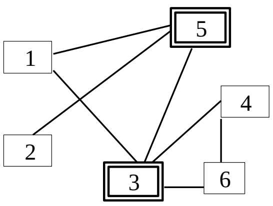
<u>*Soluţie:*</u>

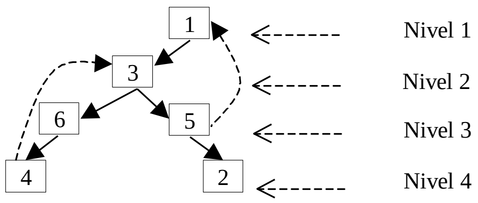
Determinarea mulţimii punctelor de articulaţie poate fi realizată printr-un algoritm liniar ($O(m+n)$). Acesta are la bază o parcurgere *DF* în care se reţin mai multe informaţii despre fiecare nod, informaţii care vor conduce, în final, la identificarea punctelor de articulaţie.

Pentru un nod $x \in V$ se vor identifica:

- numărul nivelului atins în parcurgerea *DF*, memorat în vectorul *nv* pe poziţia *x* ($nv[x]$);
- numărul minim al nivelului care poate fi atins din *x* folosind descendenţii săi şi cel mult o muchie de întoarcere. Intuitiv este vorba de "cel mai de sus" nivel care poate fi atins din *x* prin intermediul muchiilor de întoarcere accesibile din el sau din descendenţii lui. Acest număr va fi reţinut în vectorul *L* pe poziţia *x* ($L[x]$);
- vârful părinte în arborele *DF*, reţinut în vectorul *t* pe poziţia *x* ($t[x]$).

Dacă dintr-un nod *x* din graf nu se poate ajunge pe un nivel strict mai mic decât al tatălui său din arborele *DF* ($nv[t[x]] \leq L[x]$), atunci $t[x]$ este punct de articulaţie; eliminarea acestuia, împreună cu muchiile adiacente, ar duce la izolarea nodului x.





Considerăm graful din figura alăturată. El conţine două puncte de articulaţie: nodul 3 şi nodul 5.

Arborele *DF* al acestuia cu rădăcina în nodul 1 are patru nivele:

Vectorii:

t=(0,5,1,6,3,3)

nv=(1,4,2,4,3,3)

L=(1,4,1,2,1,2)

$L[3]=1$ deoarece nivelurile minime atinse de descendenţii săi sunt:

- nivelul 2 pentru nodul 4 ($L[4]=2$);
- nivelul 1 pentru nodul 5 ($L[5]=1$);
- nivelul 2 pentru nodul 6 ($L[6]=2$);
- nivelul 4 pentru nodul 2 ($L[2]=4$).

Nivelul minim atins din nodul 3 prin intermediul descendenţilor săi şi al unei muchii de întoarcere este nivelul 1 ($L[3]=1$).

Cum pentru nodul 3 există descendentul direct nodul 6, care nu poate atinge un nivel mai mic decât cel pe care este situat el, rezultă că 3 este punct de articulaţie. Analog pentru nodul 5.

De reţinut că nodul rădăcină al arborelui *DF* este punct de articulaţie dacă are mai mult de un singur descendent direct.

Implementarea în limbaj a subprogramului *df* ce identifică mulţimea punctelor critice, prezentată în continuare, ia în considerare următoarele declaraţii:

```pascal
const MAX_N=1001;
type plista=^lista;
     lista=record nod:integer;
       urm:Plista;
     end;

var G:array[0..max_n]of plista;
t,L,nv:array[0..max_n]of integer;
rad,nr,i,n,m:integer;
c,u:array[0..max_n]of byte;
```

```c
#include <stdio.h>
#define MAX_N 1001
struct lista
{ int nod;
  lista *urm;
} *G[MAX_N];

int N, M, T[MAX_N], L[MAX_N],
       nv[MAX_N], rad, nr;
char U[MAX_N], c[MAX_N];
```

Vectorul *C* va reţine pentru fiecare nod, valoarea 0 dacă nodul este critic şi 1, în caz contrar. Vectorul *U* codifică, în timpul parcurgerii *DF*, starea unui nod: vizitat sau nevizitat.

Variabila *nr* contorizează numărul de descendenţi ai nodului considerat rădăcină în parcurgerea *DF*.

Graful *G* se consideră a fi memorat cu ajutorul listelor de adiacenţă.

```pascal
...
procedure df(nod:integer);
var p:plista;
begin
 U[nod]:=1;
 L[nod]:=nv[nod];
 p:=G[nod];
 while p<>nil do begin
  if (U[p^.nod]=0) then begin
   nv[p^.nod]:= nv[nod]+1;
   T[p^.nod]:= nod;
   if (nod =rad) then inc(nr);
   DF(p^.nod);
   if (L[nod]>L[p^.nod]) then
      L[nod] := L[p^.nod];
   if (L[p^.nod]>=nv[nod])then
      c[nod] := 1;
  end
  else
   if (p^.nod <>T[nod])and
    (L[nod] > nv[p^.nod])then
         L[nod]:= nv[p^.nod];
  p:=p^.urm;
 end;
end;

begin
 citeste_graf;
 for i:=1 to n do
  if u[i]=0 then begin
   nv[i]:=1;
   rad:=i;
   nr:=0;
   DF(i);
   if nr>1 then c[rad]:=1
   else c[rad]:=0;
 end;
 for i:=1 to n do
   if c[i]<>0 then write(i,' ');
end.
```

```c
...
void DF(int nod)
{lista *p;
 U[nod] = 1;
 L[nod] = nv[nod];
 for (p = G[nod]; p != NULL;
      p = p->urm)
  if (!U[p->nod]) {
   nv[p->nod] = nv[nod]+1;
   T[p->nod] = nod;
   if (nod == rad) nr++;
   DF(p->nod);
   if (L[nod] > L[p->nod])
        L[nod] = L[p->nod];
   if (L[p->nod] >= nv[nod])
        c[nod] = 1;
    }
  else
   if (p->nod != T[nod] &&
       L[nod] > nv[p->nod])
    L[nod] = nv[p->nod];
}

int main(void)
{int i;
 citeste_graf();
 for (i = 1; i <= N; i++)
  if (!U[i])
   {nv[i] = 1;
    rad = i;
    nr = 0;
    DF(i);
    c[rad] = nr > 1;
  }
  for (i = 1; i <= N; i++)
   if (c[i]) printf("%d ", i);
    return 0;
}
```

2. ***Muchii critice – punţi***

O muchie dintr-un graf *G*=(*V*, *E*) neorientat conex este punte (muchie critică), dacă şi numai dacă, prin eliminarea sa, se pierde proprietatea de conexitate. Realizaţi un program care determină mulţimea muchiilor critice dintr-un graf.

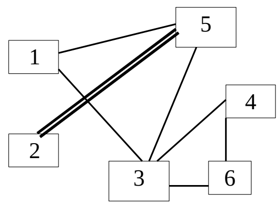
Soluţie:

Determinarea mulţimii punţilor poate fi realizată printr-un algoritm liniar ($O(m+n)$). Acesta are la bază o parcurgere *DF* în care se reţin mai multe informaţii despre fiecare nod, informaţii care vor conduce, în final, la identificarea punţilor.

Pentru un nod $x \in V$ se vor identifica:

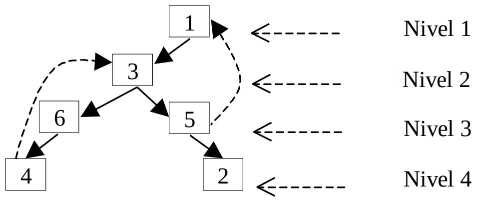
- numărul nivelului atins în parcurgerea *DF*, memorat în vectorul *nv* pe poziţia *x* (*nv*[*x*]);
- numărul minim al nivelului care poate fi atins din *x* folosind descendenţii săi şi cel mult o muchie de întoarcere. Intuitiv este vorba de “cel mai de sus” nivel care poate fi atins din *x* prin intermediul muchiilor de întoarcere accesibile din el sau din descendenţii lui. Acest număr va fi reţinut în vectorul *L* pe poziţia *x* (*L*[*x*]);
- vârful părinte în arborele *DF*, reţinut în vectorul *t* pe poziţia *x* (*t*[*x*]).

O observaţie necesară este că muchiile de întoarcere din arborele *DF* nu pot fi muchii critice, deoarece acestea închid un ciclu, iar eliminarea lor nu strică proprietatea de conexitate. Aşadar, singurele muchii care vor fi verificate sunt muchiile de arbore. Dacă dintr-un nod *x* din graf nu se poate ajunge pe un nivel mai mic sau egal decât al tatălui său din arborele *DF* (*nv*[*t*[*x*]] < *L*[*x*]), atunci muchia (*t*[*x*], *x*) este critică.



Considerăm graful din figura alăturată. El conţine o muchie critică între nodurile 2 şi 5.

Arborele *DF* al acestuia cu rădăcina în nodul 1 are patru nivele:



Vectorii:

t=(0,5,1,6,3,3)

nv=(1,4,2,4,3,3)

L=(1,4,1,2,1,2)

Nivelul minim atins din nodul 2 prin intermediul descendenţilor săi şi al unei muchii de întoarcere este nivelul 4 (*L*[2]=4), iar nivelul predecesorului său (nodul 5) este 3 (*nv*[*t*[2]] = 3).

Implementarea în limbaj a subprogramului *DF* ce identifică mulţimea muchiilor critice, prezentată în continuare, ia în considerare următoarele declaraţii:

```pascal
const MAX_N=1001;
type plista=^lista;
    lista=record
nod:integer;c:boolean;urm:Plista;
end;
var G:array[0..max_n]of plista;
t,L,nv:array[0..max_n]of integer;
i,n,m:integer; p:plista;
u:array[0..max_n]of byte;
```

```c
#include <stdio.h>
#define MAX_N 1001
struct lista {
 int nod; char c;
 lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], L[MAX_N],
nv[MAX_N];
char U[MAX_N];
```

Vectorul *U* codifică, în timpul parcurgerii *DF*, starea unui nod: vizitat sau nevizitat.

Graful *G* se consideră a fi memorat cu ajutorul listelor de adiacenţă, iar pentru fiecare element din listele de adiacenţă se va memora o variabilă de tip *boolean*(Pascal)/*char* (C/C++) care va fi *true/1* dacă muchia respectivă este critică.

```
...
procedure df(nod:integer);
var p:plista;
begin
 U[nod]:=1;
 L[nod]:=nv[nod];
 p:=G[nod];
 while p<>nil do begin
  if (U[p^.nod]=0) then begin
   nv[p^.nod]:= nv[nod]+1;
   T[p^.nod]:= nod;
   DF(p^.nod);
   if (L[nod]>L[p^.nod]) then
      L[nod] := L[p^.nod];
   if (L[p^.nod]>nv[nod])then
      p^.c := true;
  end
  else
   if (p^.nod <>T[nod])and
     (L[nod] > nv[p^.nod])then
         L[nod]:= nv[p^.nod];
  p:=p^.urm;
 end;
end;

begin
 citeste_graf;
 for i:=1 to n do
  if u[i]=0 then
   begin
    nv[i]:=1;
    DF(i);
   end;
 for i:=1 to n do
  begin
   p:=G[i];
   while p<>nil do
    begin
     if p^.c then
      write(i,' ',p^.nod);
     p:=p^.urm;
    end;
  end;
end.
```

```
...
void DF(int nod)
{lista *p;
 U[nod] = 1;
 L[nod] = nv[nod];
 for (p = G[nod]; p != NULL;
      p = p->urm)
   if (!U[p->nod]) {
    nv[p->nod] = nv[nod]+1;
    T[p->nod] = nod;
    DF(p->nod);
    if (L[nod] > L[p->nod])
         L[nod] = L[p->nod];
    if (L[p->nod] > nv[nod])
         p->c = 1;
    }
   else
    if (p->nod != T[nod] &&
        L[nod] > nv[p->nod])
     L[nod] = nv[p->nod];
}

int main(void)
{int i;
 lista *p;
 citeste_graf();
 for (i = 1; i <= N; i++)
  if (!U[i])
   { nv[i] = 1;
     DF(i);
   }
 for (i = 1; i <= N; i++)
  for (p = G[i]; p != NULL;
        p = p->urm) if (p->c)
 printf("%d %d\n", i, p->nod);
 return 0;
}
```

3. *Componente biconexe*

Prin definiţie, un graf *G*=(*V*, *E*) este biconex dacă nu conţine puncte de articulaţie. Prin componentă biconexă se înţelege un subgraf maximal în raport cu proprietatea de biconexitate. Realizaţi un program care determină muchiile fiecărei componente biconexe a unui graf.

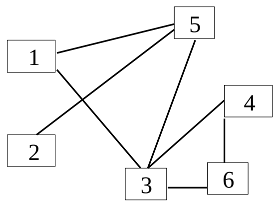
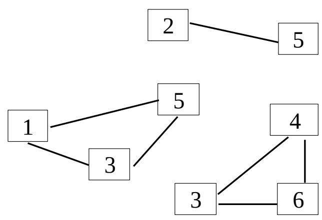
<u>*Soluţie:*</u>

Algoritmul de determinare a componentelor biconexe are la bază parcurgerea *DF* a grafului, de aici şi complexitatea liniară a acestuia ($O(m+n)$). De fapt, algoritmul este o extensie a algoritmului pentru determinarea punctelor de articulaţie:

- parcurgerea *DF* pentru determinarea numărului minim al nivelului care poate fi atins din fiecare nod folosind descendenţii acestuia şi cel mult o muchie de întoarcere. Aceste numere vor fi reţinute în vectorul L pe poziţia *x* ($L[x]$).
- în timpul parcurgerii *DF* se vor afişa muchiile din fiecare componentă biconexă. Această operaţie se va realiza memorându-se explicit o stivă cu muchiile parcurse. Când se determină un nod *x* din graf care nu poate ajunge pe un nivel strict mai mic decât al tatălui său din arborele *DF* ($nv[t[x]] \leq L[x]$), se va afişa o nouă componentă biconexă. Ea va fi formată din muchiile din stivă, operaţia de

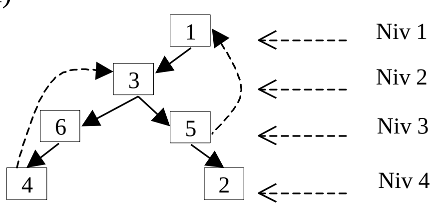
extragere din stivă se opreşte la găsirea muchiei (*t*[*x*], *x*) în vârful stivei.

Considerând ca exemplu graful următor, se vor obţine trei componente biconexe:

Etapele algoritmului:

a)



După parcurgerea *DF* se vor obţine vectorii:
t=(0,5,1,6,3,3)
nv=(1,4,2,4,3,3)
L=(1,4,1,2,1,2)

b)

Pasul 1:
În momentul găsirii nodului 6 care nu poate ajunge mai sus, stiva conţine:
st=((1,3),(3,6),(4,6),(3,4)).
Se elimină muchii din stivă până la găsirea muchiei (t[6],6)=(3, 6).
La final stiva va conţine:
st=((1,3))

Pasul 2:
Se găseşte nodul 2 care nu poate ajunge mai sus:
st=((1,3),(3,5),(2,5))
Se afişează componenta biconexă formată din succesiunea de muchii până la muchia (2,5).
La final stiva va conţine:
st=((1,3),(3,5))

Pasul 3:
Se găseşte nodul 3 care nu poate ajunge mai sus:
st=((1,3),(3,5),(1,5))

Se afişează componenta biconexă formată din succesiunea de muchii până la muchia (1,3).

Pasul 4:
Stiva vidă, algoritmul ia sfârşit.

Implementarea în limbaj a subprogramului *DF* care identifică componentele biconexe, prezentată în continuare, ia în considerare următoarele declaraţii:

```
const MAX_N=101; MAX_M=1001;
type plista=^lista;
      lista=record
      nod:integer; urm:plista;
     end;
var G:array[0..MAX_N]of Plista;
U,T,L,nv:array[0..MAX_N]of
integer;
st:array[0..MAX_M,0..2]of
integer;
lung,N,M:integer;
```

```
#include <stdio.h>
#define MAX_N 101
#define MAX_M 1001
struct lista
{ int nod;
  lista *urm; } *G[MAX_N];
int N, M, T[MAX_N], L[MAX_N],
nv[MAX_N], st[MAX_M][2], lung;
char U[MAX_N];
```

Vectorul *U* codifică, în timpul parcurgerii *DF*, starea unui nod: vizitat sau nevizitat. Subprogramele *push()* şi *pop()* implementează operaţiile de introducere, respectiv extragere din stivă a unei muchii.

```pascal
...
procedure push(i,j:integer);
begin
 st[lung][0]:=i;st[lung,1]:=j;
 inc(lung);
end;
procedure pop(var i,j:integer);
begin
 dec(lung); i:=st[lung,0];
 j:=st[lung][1];
end;
procedure DF(nod:integer);
var p:plista; x,y:integer;
begin
 U[nod]:=1;
 L[nod]:= nv[nod];
 p:=G[nod];
 while p<>nil do begin
  if (p^.nod<>T[nod])and
     (nv[nod]>nv[p^.nod]) then
       push(p^.nod,nod);
  if U[p^.nod]=0 then begin
   nv[p^.nod]:=nv[nod]+1;
   T[p^.nod]:= nod;
   DF(p^.nod);
   if (L[nod]>L[p^.nod]) then
     L[nod]:= L[p^.nod];
   if (L[p^.nod]>=nv[nod])then
    begin
    repeat
     pop(x, y);
     write('(',x,' ',y,') ');
    until not((x<>nod)or
              (y<>p^.nod))
          or not ((x<> p^.nod)
                 or (y<>nod));
    writeln;
   end;
  end else
  if (p^.nod<>T[nod])and
     (L[nod]>nv[p^.nod]) then
   L[nod]:= nv[p^.nod];
  p:=p^.urm;
 end;
end;
```

```c
...
void push(int i, int j)
{
  st[lung][0] = i;
  st[lung++][1] = j;
}
void pop(int *i, int *j)
{
 *i = st[--lung][0];
 *j = st[lung][1];
}
void DF(int nod)
{
 lista *p;
 int x, y;
 U[nod] = 1; L[nod] = nv[nod];
 for (p = G[nod]; p != NULL;
      p = p->urm)
 {if (p->nod != T[nod] &&
      nv[nod] > nv[p->nod])
         push(p->nod, nod);
  if (!U[p->nod])
  {nv[p->nod] = nv[nod]+1;
   T[p->nod] = nod;
   DF(p->nod);
   if (L[nod] > L[p->nod])
      L[nod] = L[p->nod];
   if (L[p->nod] >= nv[nod])
   {do
    {
     pop(&x, &y);
     printf("(%d %d) ", x, y);
    } while ((x != nod ||
              y != p->nod)
           && (x != p->nod ||
              y != nod));
      printf("\n");}
    }
   else
    if (p->nod != T[nod] &&
      L[nod] > nv[p->nod])
        L[nod] = nv[p->nod];
 }
}
```