

# Teoria grafurilor

May 8

# 2023

---

Curs introductiv lot național de informatică - juniori

Prof. Dana Lica

# Teoria grafurilor

## 1. Terminologie

*Graf*  $\Leftrightarrow$  orice mulțime finită  $V$ , prevăzută cu o relație binară internă  $E$ . Notăm graful cu  $G=(V,E)$ .

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

*Graf orientat*  $\Leftrightarrow$  un graf  $G=(V,E)$ , în care relația binară nu este simetrică.

*Nod*  $\Leftrightarrow$  element al mulțimii  $V$ , unde  $G=(V,E)$  este un graf neorientat.

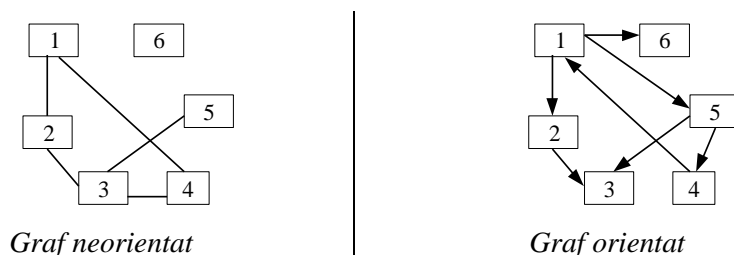
*Vârf*  $\Leftrightarrow$  element al mulțimii  $V$ , unde  $G=(V,E)$  este un graf orientat sau neorientat.

*Muchie*  $\Leftrightarrow$  element al mulțimii  $E$  ce descrie o relație existentă între două vârfuri din  $V$ , unde  $G=(V,E)$  este un graf neorientat;

*Arc*  $\Leftrightarrow$  element al mulțimii  $E$  ce descrie o relație existentă între două vârfuri din  $V$ , unde  $G=(V,E)$  este un graf orientat;

Arcele sunt parcurse în direcția dată de relația vârf  $\rightarrow$  succesor\_direct. Muchiile unui graf neorientat sunt considerate ca neavând direcție, deci pot fi parcurse în ambele sensuri.

*Adiacență*  $\Leftrightarrow$  Vârful  $w$  este adiacent cu  $v$  dacă perechea  $(v,w) \in E$ . Într-un graf neorientat, existența muchiei  $(v,w)$  presupune că  $w$  este adiacent cu  $v$  și  $v$  adiacent cu  $w$ .



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

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

*Incidență spre interior*  $\Leftrightarrow$  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  $w$ .

*Incidență spre exterior*  $\Leftrightarrow$  Un arc este incident spre exterior cu un vârf dacă îl are pe acesta ca vârf inițial (arcul pleacă din vârf). Arcul  $(v,w)$  este incident spre exterior cu vârful  $v$ .

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

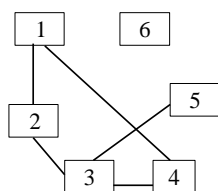
*Grad interior*  $\Leftrightarrow$  În cazul unui graf orientat, fiecare nod  $v$  are asociat un număr numit grad interior și care este egal cu numărul de arce care îl au pe  $v$  ca vârf terminal (numărul de arce incidente spre interior).

*Grad exterior*  $\Leftrightarrow$  În cazul unui graf orientat, fiecare nod  $v$  are asociat un număr numit grad exterior și care este egal cu numărul de arce care îl au pe  $v$  ca vârf inițial (numărul de arce

incidente spre exterior).

*Vârf izolat*  $\Leftrightarrow$  Un vârf cu gradul 0.

*Vârf terminal*  $\Leftrightarrow$  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.

*Lanț*  $\Leftrightarrow$  este o secvență de noduri ale unui graf neorientat  $G=(V,E)$ , cu proprietatea că oricare două noduri consecutive sunt adiacente:

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

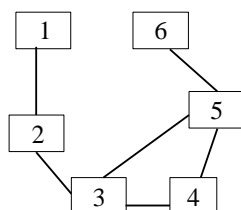
*Lungimea unui lanț*  $\Leftrightarrow$  numărul de muchii din care este format.

*Lanț simplu*  $\Leftrightarrow$  lanțul care conține numai muchii distincte.

*Lanț compus*  $\Leftrightarrow$  lanțul care nu este format numai din muchii distincte.

*Lanț elementar*  $\Leftrightarrow$  lanțul care conține numai noduri distincte.

*Ciclu*  $\Leftrightarrow$  Un lanț în care primul nod coincide cu ultimul. Ciclu este elementar dacă este format doar din noduri distincte, excepție făcând primul și ultimul. Lungimea minimă a unui ciclu este 3.



Sucesiunea de vârfuri 2, 3, 5, 6 reprezintă un lanț simplu și elementar de lungime 3.

Lanțul 5 3 4 5 6 este simplu dar nu este elementar.

Lanțul 5 3 4 5 3 2 este compus și nu este elementar.

Lanțul 3 4 5 3 reprezintă un ciclu elementar.

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

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

*Lungimea unui drum*  $\Leftrightarrow$  numărul de arce din care este format.

*Drum simplu*  $\Leftrightarrow$  drumul care conține numai arce distincte.

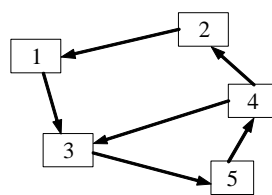
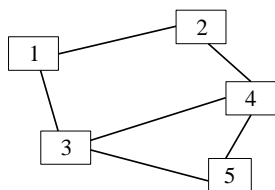
*Drum compus*  $\Leftrightarrow$  drumul care nu este format numai din arce distincte.

*Drum elementar*  $\Leftrightarrow$  drumul care conține numai vârfuri distincte.

*Circuit*  $\Leftrightarrow$  Un drum în care primul vârf coincide cu ultimul. Circuitul este elementar dacă este format doar din vârfuri distincte, excepție făcând primul și ultimul.

*Bucă*  $\Leftrightarrow$  Circuit format dintr-un singur arc.

Ciclu elementar 3,4, 2, 1



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

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

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

*Graf regulat*  $\Leftrightarrow$  graf neorientat în care toate nodurile au același grad.

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

*Graf conex*  $\Leftrightarrow$  graf neorientat  $G=(V,E)$  în care, pentru orice pereche de noduri  $(v,w)$ , există un lanț care le unește.

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

*Componentă conexă*  $\Leftrightarrow$  subgraf al grafului de referință, maximal în raport cu proprietatea de conexitate (între oricare două vârfuri există lanț);

*Lanț hamiltonian*  $\Leftrightarrow$  un lanț elementar care conține toate nodurile unui graf.

*Ciclu hamiltonian*  $\Leftrightarrow$  un ciclu elementar care conține toate nodurile grafului.

*Graf hamiltonian*  $\Leftrightarrow$  un graf  $G$  care conține un ciclu hamiltonian.

*Condiție de suficiență:*

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.

*Lanț eulerian*  $\Leftrightarrow$  un lanț simplu care conține toate muchiile unui graf.

*Ciclu eulerian*  $\Leftrightarrow$  un ciclu simplu care conține toate muchiile grafului.

*Graf eulerian*  $\Leftrightarrow$  un graf care conține un ciclu eulerian.

*Condiție necesară și suficientă:*

Un graf este eulerian dacă și numai dacă oricare vârf al său are gradul par.

## 2. Moduri de reprezentare la nivelul memoriei

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

1. matricea de adiacență;
2. listele de adiacență; (liste simplu înlănțuite și containeri)
3. matricea ponderilor (costurilor);
4. lista muchiilor.

• *Matricea de adiacență* este o matrice binară (cu elemente 0 sau 1) care codifică existența 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 ( $|V|^2 \approx E$ ).

```
1 Creare_MA_neorientat(G=(V,E))          /*complexitate:  $O(V^2 + E)$  */
2 pentru i ← 1, |V| executa
3   pentru j ← 1 to |V| executa G[i][j] ← 0;
4   ■
5   ■
6   pentru e = 1, |E| executa
7     citește i, j;
8     G[i][j] ← 1;
9     G[j][i] ← 1;          //instrucțiunea lipsește în cazul digrafului
10  ■
```

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

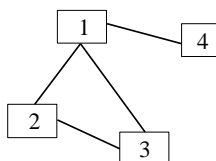
```
#include <fstream>
#define MAX_N 101
#define MAX_M 1001
int N, M; char G[MAX_N][MAX_N];
```

Subprogramul *citește\_graf()* preia informațiile din fișierul text *graf.in*, în felul următor: de pe prima linie numărul  $n$  de vârfuri și  $m$  de muchii, iar de pe următoarele  $m$  linii, perechi de numere reprezentând muchiile grafului. Matricea  $G$  de adiacență se consideră inițializată cu valoarea 0.

```
1 void citește_graf(void)
2 {
3     int i, j;
4     ifstream f("graf.in");
5     f >> N >> M;
6     for (; M > 0; M--)
7     {
8         f >> i >> j;
9         G[i][j] = G[j][i] = 1;
10    }
11 }
12
```

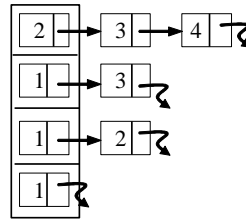
• *Listele de adiacență* rețin, pentru fiecare nod din graf, toate vârfurile adiacente cu acesta (vecine cu el). Ca modalitate de reprezentare se poate folosi un vector  $LA$  ale cărui elemente sunt adresele de început ale celor  $|V|$  liste simplu înlănțuite memorate, una pentru fiecare vârf din 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

Pentru graful alăturat exemplificăm reprezentarea atât în liste de adiacență, cât și în matrice de adiacență.



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

Matricea de adiacență (MA)



Liste de adiacență (LA)

Crearea listelor de adiacență este ilustrată în pseudocodul următor:

```

1  Creare_LA_neorientat(G=(V,E))          /*complexitate: O(V + E)*/
2  pentru i←1, |V| executa
3      G[i] ← NULL
4  ■
5  pentru e ← 1, |E| executa
6      citește i,j;
7      aloca(p);
8      p->inf ← j; p->leg ← G[i]; G[i] ← p
9      aloca(p);
10     p->inf ← i; p->leg ← G[j]; G[j] ← p
11 ■
12
```

Implementarea în limbaj a subprogramului care creează listele de adiacență a unui graf neorientat ia în considerare următoarele declarații:

```

#define MAX_N 101
struct lista
{ int nod;
  lista *urm;
} *G[MAX_N];
int N, M;
```

Subprogramul *citeste\_graf()* preia informațiile din fișierul text *graf.in*, în felul următor: de pe prima linie, numărul *n* de vârfuri și *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 informație *j* în lista lui vecinilor nodului *i*. Tabloul *G*, reprezentând listele de adiacență, se consideră inițializat cu constanta *NULL* (C++).

```

1  void adauga(int i, int j)
2  {
3      lista *p;
4      p = new lista; p->nod = j;
5      p->urm = G[i]; G[i] = p;
6  }
7
8  void citeste_graf(void)
9  {
10     int i, j;
11     ifstream f("graf.in");
12     f >> N >> M;
13     for (; M > 0; M--)
14     {f >> i >> j;
15      adauga(i, j); adauga(j, i);
16     }
17 }
```

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

Prezentăm și varianta de implementare C++ cu ajutorul containerului `<vector>`:

```

1  # include <vector>
2
3  using namespace std;
4  vector <int> G[MAXN];
5  ...
6  void load() {
7      f >> N >> M;
8      for (i=1; i<= M; i++) {
9          f >> x >> y;
10         G[x].push_back(y);
11         G[y].push_back(x);
12     }
13     return;
14 }

```

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

$$\begin{aligned}
 G[i, j] &= \infty, \text{ dacă } (i, j) \notin E \\
 G[i, j] &= \text{cost} < \infty, \text{ dacă } (i, j) \in E \\
 G[i, j] &= 0, \text{ dacă } i = j
 \end{aligned}$$

Pseudocodul care descrie modul de memorare cu ajutorul matricei costurilor:

```

1  Creare_MC_neorientat(G=(V,E))          /*complexitate: O(V^2 + E)*/
2  pentru i=1, |V| executa
3      pentru j = 1 to |V| executa
4          dacă i = j atunci G[i][j] ← 0;
5          altfel G[i][j] ← ∞
6      ■
7  ■
8  ■
9  pentru e = 1, |E| executa
10     citește i, j, c;
11     G[i][j] ← c;
12     G[j][i] ← c;      //instrucțiunea lipsește în cazul digrafului
13 ■

```

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

```

1  #define MAX_N 101
2  #define INF 0x3f3f3f3f
3  int N, M, G[MAX_N][MAX_N];

```

Subprogramul *citește\_graf()* preia informațiile din fișierul text *graf.in* în felul următor: de pe prima linie, numerele  $n$  de vârfuri și  $m$  de muchii, iar de pe următoarele  $m$  linii, câte trei numere  $i, j, c$  având semnificația muchie între  $i$  și  $j$  de cost  $c$ .

```

1  void citește_graf(void)
2  {
3      int i, j, c;
4      ifstream f("graf.in");
5
6      f >> N >> M;
7      for (i = 1; i <= N; i++)
8          for (j = 1; j <= N; j++)
9              G[i][j] = (i != j) * INF;
10     for (; M > 0; M--)
11     {
12         f >> i >> j >> c;
13         G[i][j] = c;
14         G[j][i] = c;
15     }
16 }

```

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

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

```
#define MAX_N 101
#define MAX_M 10001
int N, M, E[MAX_M][3];
```

Tabloul *E* reține lista muchiilor. Subprogramul *citeste\_graf()* preia informațiile din fișierul text *graf.in* în felul următor: de pe prima linie, numărul *n* de vârfuri și *m* de muchii, iar de pe următoarele *m* linii, câte trei numere *i, j, c* având semnificația muchie între *i* și *j* de cost *c*.

```
1 void citeste_graf(void)
2 {
3     int i, j, k, c;
4     ifstream f("graf.in");
5     f >> N >> M;
6     for (k = 0; k < M; k++) {
7         f >> i >> j >> c;
8         E[k][0] = i; E[k][1] = j;
9         E[k][2] = c;
10    }
11 }
```

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```
1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #define pb push_back
5 #define f first
6 #define s second
7
8 using namespace std;
9
10 int N, M, x, y, c;
11 vector <pair <int, pair<int, int> > > L;
12 void load(){
13
14     cin >> N >> M;
15     for (int i = 0; i < M; i++){
16         cin >> x >> y >> c;
17         L.pb({c, {x, y}});
18     }
19 }
```



### 3. Algoritmi pe grafuri

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

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Realizați un subprogram care să permită afișarea nodurilor în cazul parcurgerii în adâncime-DF.

##### Soluție:

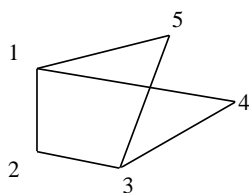
Printre operațiile fundamentale efectuate asupra structurilor de date se numără și traversarea acestora. Această operație constă într-o căutare efectuată asupra tuturor elementelor ei. Cu alte cuvinte, traversarea poate fi privită și ca un caz special de căutare, deoarece constă în regăsirea tuturor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

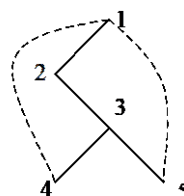
Această strategie a parcurgerii DF funcționează respectând mecanismul *LIFO*. Vârful care este eliminat din stivă nu mai are nici o muchie disponibilă pentru traversare. Aceleași reguli se respectă și la parcurgerea în adâncime a grafurilor orientate (*digrafuri*).

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

- *muchii de arbore (avans)*, folosite pentru explorarea nodurilor; ele constituie un graf parțial format din unul sau din mai mulți arbori ai parcurgerii DF.
- *muchii de întoarcere (înapoi)*, care unesc un nod cu un strămoș al său în arborele DF.



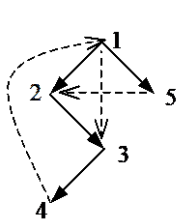
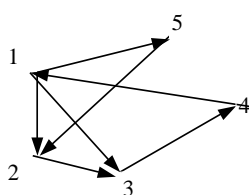
Parcurgerea DF: 1 2 3 4 5



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 parțial format din unul sau din mai mulți arbori ai parcurgerii DF.
- *arce înapoi*, care unesc un nod cu un strămoș al său în arborele DF.
- *arce înainte*, sunt arce care nu aparțin arborelui DF și care conectează un vârf cu un descendent al său.
- *arce transversale*, sunt arce care conectează două vârfuri ce nu se află în relația ascendent-descendent.



Arce transversale:

(5,2)

Arce înainte: (1,3)

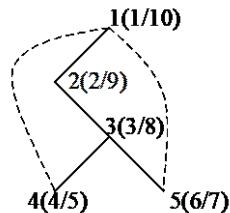
Arce înapoi: (4,1)

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 reține 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 mulțimea nodurilor și  $E$  mulțimea muchiilor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$  cardinalul lui  $E$ .

```

1  Depth_First (G=(V,E), D[N], F[N], T[N])    //complexitate O(N+M)
2
3  Use ← False;                                //nici un nod nu este vizitat
4  timp ← 0;
5  pentru nod ← 1,N executa
6      daca Not(Use[nod]) atunci Parcurge_df(nod)
7
8
9  Parcurge_df(nod)
10 Use[nod] ← TRUE; timp ← timp+1; D[nod]←timp;
11 scrie nod; i ← prim_vecin(nod);
12 cat_timp lista_vecinilor(nod)≠∅ executa
13     daca not(Use[i]) atunci
14         T[i]←nod;
15         parcurge_df(i);
16     ■
17     i ← urmator_vecin(nod);
18 ■
19 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 declarații:

```

#define MAX_N 1001
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 și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacență.

```

1  void DF(int nod) {
2      lista *p;
3      U[nod] = 1;
4      D[nod] = ++timp;
5      cout << nod << ' ';
6      for (p = G[nod]; p != NULL; p = p->urm)
7          if (!U[p->nod]) {
8              T[p->nod] = nod;
9              DF(p->nod);
10         }
11     F[nod] = ++timp;
12 }

```

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```
1  ...
2  #include <vector>
3  #define MAX_N 1001
4  #define pb push_back
5
6  using namespace std;
7
8  vector<int> L[MAXN];
9  int n, m, i, x, y, nrc, timp;
10 int T[MAXN], F[MAXN];
11 bool U[100001];
12
13 void dfs( int x ){
14     int i;
15     U[x] = true;
16     D[x] = ++ timp;
17     for(i = 0; i< L[x].size();i++)
18         if (!U[L[x][i]]){
19             T[L[x][i]] = x;
20             dfs(L[x][i]);
21         }
22     F[x] = ++timp;
23 }
```

## 2. Parcurgerea grafurilor în lățime BF - Breadth First

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Realizați un subprogram care să permită afișarea nodurilor în cazul parcurgerii în lățime-BF.

### Soluție:

Strategia parcurgerii BF funcționează 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 lanțurilor de lungime minimă de la nodul de start către toate vârfurile accesibile lui din graf. Arborele BF, ce cuprinde muchiile traversate în parcurgerea în lățime, are proprietatea de a fi format doar din lanțuri de lungime minimă, lanțuri ce unesc nodul de start al parcurgerii cu toate vârfurile accesibile acestuia.

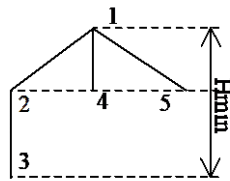
Aceleași reguli se respectă și la parcurgerea în lățime a grafurilor orientate (*digrafuri*).

Algoritmul lui Lee de determinare a lanțului minim dintre două vârfuri ale unui graf se bazează tocmai pe această strategie de traversare în lățime, 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 reține vârful predecesor (părinte) în parcurgere;
- $D[N]$ , în care, pentru fiecare vârf, se memorează lungimea lanțului 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 mulțimea nodurilor și  $E$ , mulțimea muchiilor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$  cardinalul lui  $E$ .

```

1  Breadth_First (G=(V,E), C[N] T[N], Nod_Start)
2                                     /*complexitate: O(M+N)*/
3  Use ← FALSE;
4  Coada ← {Nod_Start} //nodul de start este introdus in coada
5  Use[nod] ← True;
6  cat timp Coada ≠ ∅ executa
7      v ← varf_Coada; i ← prim_vecin(v);
8      cat timp lista_vecinilor(v) ≠ ∅ executa
9          daca not(Use[i]) atunci
10             T[i] ← v; Use[i] ← TRUE;
11             D[i] ← D[v]+1;
12             Coada ← {i};
13             i ← urmator_vecin(v);
14         ■
15     ■
16     COADA ⇒ v;
17     ■

```

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

```

#define MAX_N 1001
struct lista
{
    int nod;
    lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], D[MAX_N], C[MAX_N], timp; char U[MAX_N];

```

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

```

1  void BF(int start) {
2      lista *p;
3      int nod, st, dr;
4      memset(U, 0, sizeof(U));
5      st = dr = 1;
6      C[st]=start;
7      U[start] = 1;
8      for (D[start]=0;st<=dr;st++)
9          {nod=C[st];
10             for (p = G[nod];p != NULL;p = p->urm)
11                 if (!U[p->nod])
12                     {
13                         U[C[++dr] = p->nod] = 1;
14                         D[p->nod] = D[nod]+1;
15                         T[p->nod] = nod;
16                     }
17             }
18 }

```

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```

1  ...
2  #include <queue>
3  #include <vector>
4  #define MAXN 10001

```

```

5  using namespace std;
6  vector <int> G[100001];
7  queue <int> Q;
8  int N, M, i, x, y, P, Lg[MAXN], T[MAXN];
9  bool U[MAXN];
10
11
12 void bfs(int x){
13     Q.push(x); Lg[x] = 0; U[x] = true;
14     while (!Q.empty()){
15         x = Q.front();
16         for( i = 0; i < G[x].size(); i++)
17             if (!U[G[x][i]]) {
18                 Q.push(G[x][i]);
19                 Lg[G[x][i]] = Lg[x]+1;
20                 U[G[x][i]] = true;
21                 T[G[x][i]] = x;
22             }
23     Q.pop();
24 }
25 }

```

### 3. Ciclu eulerian

Fie  $G=(V,E)$ , graf neorientat conex, în care fiecare nod are gradul par. Notăm cu  $N$  cardinalul lui  $V$  și 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  $N=6$ ,  $M=10$  și arcele: (1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5), (4,6), (5,6), se va afișa: 1, 3, 5, 6, 4, 5, 2, 4, 3, 2, 1.

Soluție:

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*.

Înaintarea 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.

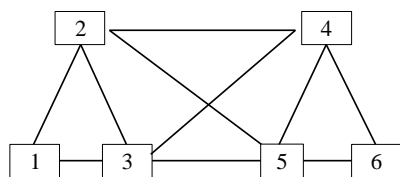
De fiecare dată când nodul curent este eliminat din stivă, acesta este afișat sau salvat într-o coadă. Această coadă va reține, în ordine, nodurile ciclului eulerian. Procesul se repetă până în momentul în care stiva devine vidă, deci toate muchiile au fost traversate.

```

1  Ciclu_Euler (Nod)
2                                     /*complexitate: O(M+N)*/
3  i ← prim_vecin(nod);
4  cat timp lista_vecinilor(nod) ≠ ∅ executa
5      elimin(i,nod)                  //elimin pe i din vecinii lui nod
6      elimin(nod,i)                  //elimin pe nod din vecinii lui i
7      ciclu_euler(i)
8      i ← urmator_vecin(nod);
9  Coadă ← {nod};
10

```

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



*Pasul 1:*

Se parcurge începând cu nodul 1

Stiva=(1,2,3,1)

Coadă =  $\phi$

*Pasul 2:*

Iese din stivă nodul 1, salvându-se în coadă:

Stiva = (1,2,3)

Coadă = (1)

*Pasul 3:*

Se continuă parcurgerea:

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

Coadă = (1)

*Pasul 4:*

Iese din stivă nodul 3, salvându-se în coadă:

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

Coadă = (1,3)

*Pasul 5:*

Se continuă parcurgerea:

Stiva = (1,2,3,4,2,5,4,6,5)

Coadă = (1,3)

*Pasul 6:*

Iese din stivă nodul 5, salvându-se în coadă:

Stiva = (1,2,3,4,2,5,4,6)

Coadă = (1,3,5)

*Pasul 7:*

Toate nodurile sunt extrase din stivă și introduse în coadă. La finalul algoritmului:

Stiva =  $\phi$

Coadă = (1,3,5,6,4,5,2,4,3,2,1).

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

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

Ca și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul matricei de adiacență. Datorită acestui mod de memorare, implementarea în limbaj conduce la o complexitate pătratică  $O(N^2)$ . Dacă graful este memorat cu ajutorul listelor de adiacență, complexitatea este redusă la  $O(N+M)$ .

```
1 void euler(int nod)
2 {int urm;
3   for (urm = 1; urm <= N; urm++)
4     if (G[nod][urm])
5     {
6       G[nod][urm] = 0;
7       G[urm][nod] = 0;
8       euler(urm);
9     }
10    C[nc++] = nod;
11 }
```

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

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```
1 #include <fstream>
2 #include <vector>
3 using namespace std;
4 ifstream f("ciclueuler.in");
5 ofstream g("ciclueuler.out");
6
7 typedef pair < int, int > PII;
8 const int NMAX = 1e5 + 1, MMAX = 5e5 + 1;
9 const int ROOT = 1;
```

```

10 int N, M;
11 vector < PII > G[NMAX];
12 bool Sel[MMAX];
13 vector < int > Sol;
14
15 void Read () {
16     f.tie(nullptr);
17     f >> N >> M;
18     for(int i = 1; i <= M; ++i){
19         int X = 0, Y = 0;
20         f >> X >> Y;
21
22         G[X].push_back({Y, i});
23         G[Y].push_back({X, i});
24     }
25 }
26
27 void DFS (int Node){
28     while(!G[Node].empty()){
29         auto it = G[Node].back();
30         G[Node].pop_back();
31
32         if(!Sel[it.second]){
33             Sel[it.second] = 1;
34
35             DFS(it.first);
36         }
37     }
38     Sol.push_back(Node);
39 }
...

```

#### 4. Matricea drumurilor - Algoritmul lui Warshall

Fie  $G=(V,E)$ , graf orientat cu  $n$  vârfuri și  $m$  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  $x$  și  $y$ .

##### Soluție:

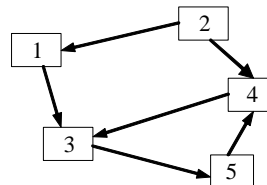
Numim matricea drumurilor sau a închiderii tranzitive, o matrice pătratică  $D(N,N)$  în care elementul  $D[i,j]=1$ , dacă există drum între nodul  $i$  și nodul  $j$  și 0, în caz contrar.

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

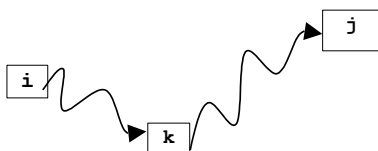
```



Algoritmul Warshall construiește matricea închiderii tranzitive  $D$ , plecând de la matricea de adiacență  $G$ .

Inițial, matricea  $D$  indică prezența 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  $i$  și  $j$  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  $k$  și 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:

$$D[i,j] = D[i,j] \text{ OR } \{D[i,k] \text{ AND } D[k,j]\}$$

```

1  Warshall (G=(V,E))
2
3  D ← G;                                     /*complexitate: O(N³)*/
4  pentru k ← 1,n executa                     //initializare matrice drumuri
5  |   pentru i ← 1,n executa
6  |   |   pentru j ← 1,n executa
7  |   |   |   D[i,j] ← D[i,j] OR {D[i,k] AND D[k,j]}
8  |   |   |   ■
9  |   |   ■
10  |   ■

```

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

```

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

```

1  void Warshall()
2  {
3  int i,j,k;
4  for (i = 1; i <= N; i++)
5  for (j = 1; j <= N; j++)
6  D[i][j] = G[i][j];
7  for (k = 1; k <= N; k++)
8  for (i = 1; i <= N; i++)
9  for (j = 1; j <= N; j++)
10 if (! D[i][j])
11 D[i][j]=D[i][k] && D[k][j];
12 }

```

## 5. Componente conexe

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

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

### Soluție:

Problema va fi rezolvată prin determinarea tuturor componentelor conexe ale grafului și 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. Procedeu continuă până când s-au vizitat toate nodurile.

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

```

1  Componente_conexe (G=(V,E))               /*complexitate: O(M+N)*/
2  nrc ← 0;                                   // nr de componente conexe
3  Use ← False;                               // nici un nod selectat
4  pentru i ← 1,n executa
5  |   daca Not(Use[i]) atunci
6  |   |   nrc ← nrc + 1;
7  |   |   parcurge_df(i)
8  |   ■

```



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

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

```
#include <iostream>
#define MAX_N 1001
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];
```

Graful este considerat a fi memorat cu ajutorul listelor de adiacență

```
1 void DF(int nod, int &x)
2 {
3     lista *p;
4     U[nod] = 1;
5     x++;
6     for (p = G[nod]; p != NULL; p = p->urm)
7         if (!U[p->nod])
8             DF(p->nod, x);
9 }
10
11 void comp_conex()
12 {
13     int i, max, v, x;
14     max=0;
15     memset(U, 0, sizeof(U));
16     for (i = 1; i <= N; i++)
17         if (!U[i])
18             {x=0;
19              DF(i, x);
20              if (x > max) {
21                  max=x; v=i;
22              }
23              cout << max << ' ' << v;
24 }
```

## 6. Tare-conexitate

Fie  $G=(V,E)$  un graf orientat. Realizați un program care afișează vârfurile fiecărei componente tare conexă în care se descompune graful  $G$ .

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

```
1 4 3 2
5
```

### Soluție:

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 conexă, inversarea sensului de deplasare nu implică blocarea accesului către vreunul din vârfurile sale.

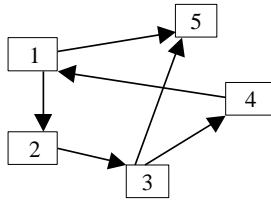
Algoritmul propus identifică componenta tare conexă a unui vârf ca fiind intersecția dintre mulțimea nodurilor accesibile din el în graful inițial  $G$  și mulțimea nodurilor accesibile din el în graful transpus  $G^t$  (obținut prin inversarea sensurilor arcelor).

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

- Parcurgerea *DF* a grafului inițial ( $G$ ) pentru memorarea explicită a stivei ce conține vârfurile

- grafului în ordinea crescătoare a timpilor de finish.
- Parcurgerea *DF* a grafului transpus (*Gt*) începând cu ultimul vârf reținut în stivă către primul. Parcurgerea se reia din fiecare vârf rămas nevizitat la parcurgerile anterioare. Vârfurile fiecărui arbore *DF* obținut la acest pas reprezintă câte o componentă tare conexă.

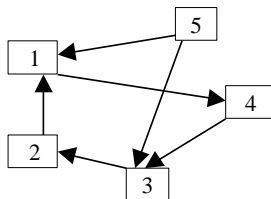
Pentru graf:



*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 declarații:

```

...
#include <iostream>
#define MAX_N 1001

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 reține graful transpus asociat lui *G*. Subprogramul *citeste\_graf* preia datele referitoare la graful *G* și construiește matricele de adiacență ale celor două grafuri *G* și *GT*.

```

1 void DF1(int nod)
2 {
3     lista *p;
4
5     U[nod] = 1;
6     for (p = G[nod]; p != NULL;
7         p = p->urm)
8         if (!U[p->nod])
9             DF1(p->nod);
10    ST[nst++] = nod;
11 }
12
13 void DF2(int nod)
14 {
15     lista *p;
16
17     U[nod] = 1;
18     cout << nod << ' ';
19     for (p = GT[nod]; p != NULL;
20         p = p->urm)
21         if (!U[p->nod])
22             DF2(p->nod);
23 }
24
25 void citeste_graf(void)
26 {.....}
27
  
```

```

28 int main()
29 {int i; citeste_graf();
30  for (i = 1; i <= N; i++)
31      if (!U[i])
32          DF1(i);
33
34  memset(U, 0, sizeof(U));
35
36  for (i = N-1; i >= 0; i--)
37      if (!U[ST[i]]) {
38          DF2(ST[i]);
39          cout << '\n';
40      } return 0;
41 }

```

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```

1  #include <vector>
2  #include <cstring>
3  #include <iostream>
4  #define pb push_back
5  #define MAXN 10001
6  using namespace std;
7
8  vector<int> G[MAXN],GT[MAXN];
9  int nc, nr, i, k, N, M, x, y, St[MAXN];
10 bool sel[MAXN];
11
12 void DF(int x){
13     int i;
14     sel[x]=true;
15     for(i = 0; i < G[x].size(); ++i)
16         if (!sel[G[x][i]]) DF(G[x][i]);
17     St[++nr]=x;
18 }
19
20 void DFT(int x, bool k){
21     int i;
22     sel[x]=true;
23     if (k) cout << x << ' ';
24     for(i = 0; i < GT[x].size(); ++i)
25         if (!sel[GT[x][i]])
26             DFT(GT[x][i], k);
27 }
28
29 int main(){
30     . . . . .
31     cin >> N >> M;
32     for (i=1; i<=M; ++i){
33         cin >> x >> y;
34         G[x].pb(y); GT[y].pb(x);
35     }
36     nr=0; nc=0;
37     memset(sel, false,sizeof(sel));
38     for(i=1; i<=N; i++)
39         if (!sel[i]) DF(i);
40     memset(sel, false,sizeof(sel));
41     for(i =N; i>0; --i)
42         if (!sel[St[i]]) {
43             DFT(St[i], false);
44             nc++;
45         }
46     cout << nc << '\n';
47     memset(sel, false,sizeof(sel));
48     for(i=N; i>0; --i)
49         if (!sel[St[i]]) {
50             DFT(St[i], true);
51             cout << '\n';

```

```

52     }
53     return 0;
54 }
55

```

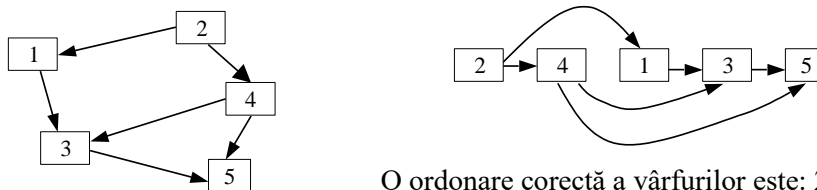
## 7. Sortarea topologică a unui digraf

Fie  $G=(V,E)$  un graf orientat aciclic. Realizați un program care ordonează vârfurile grafului după următoarea regulă: pentru orice arc  $(x,y) \in E$ , vârful  $x$  apare, în șirul ordonat, înaintea vârfului  $y$ .

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

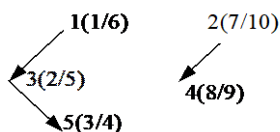
Soluție:

Pentru a înțelege 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  $D$  și  $F$  care rețin timpii de intrare, respectiv timpii de ieșire din stivă sunt:



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

Dacă vârfurile sunt reținute într-o listă, în ordinea ieșirii lor din stivă, atunci această listă va conține:  $St=(5, 3, 1, 4, 2)$ . Afișarea conținutului ei în ordine inversă, adică în ordinea inversă timpilor de finish, va reprezenta ordinea vârfurilor obținută prin sortarea topologică 2, 4, 1, 3, 5.

Să presupunem vârfurile digrafului ca niște evenimente și 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.

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea vârfurilor și  $E$ , mulțimea arcelor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$ , cardinalul lui  $E$ .

```

1  Sortare_topologică_DF (G=(V,E),St[N]) //complexitate O(N+M)
2  nr ← 0;                               //nr de varfuri extrase din stiva
3  Use ← False
4  pentru nod ← 1,N executa
5      dacă Not(Use[nod]) atunci Parcurge_df(nod)
6      ■
7  ■
8  pentru i ← N,1,-1 executa scrie St[i]
9  ■
10 Parcurge_df(nod)
11 Use[nod] ← TRUE; i ← prim_vecin(nod);
12 cat_timp lista_vecinilor(nod)≠∅ executa
13     dacă not(Use[i]) atunci parcurge_df(i);
14     ■
15     i ← urmator_vecin(nod);
16     ■
17 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 declarații:

```
#include <iostream>
#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 și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacență.

```
1 void DF(int nod)
2 {
3     lista *p;
4     U[nod] = 1;
5     for (p = G[nod]; p != NULL;
6         p = p->urm)
7         if (!U[p->nod])
8             DF(p->nod);
9     ST[nst++] = nod;
10 }
11
12 int main(void)
13 { int i;
14   citește_graf();
15   for (i = 1; i <= N; i++)
16       if (!U[i]) DF(i);
17   for (i = N-1; i >= 0; i--)
18       cout << ST[i] << ' ';
19   return 0;
20 }
```

O altă modalitate de implementare a sortării topologice ține cont de observația 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.

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

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

Inițial vectorii conțin:

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

$C=(2)$

a) Se parcurge în lățime din nodul 2

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

$C=(2,4,1)$

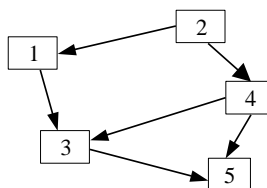
b) Se parcurge în lățime din nodul 4

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

$C=(2,4,1)$

d) Se parcurge în lățime din nodul 1

e) Se parcurge în lățime din nodul 3



$Deg=(0,0,0,0,1)$   
 $C=(2,4,1,3)$

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

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

```
#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 adiacență.

```
1 void sort_BF(void)
2 {lista *p;
3   int i, st = 0, dr = -1;
4   for (i = 1; i <= N; i++)
5     if (deg[i] == 0){
6       C[++dr] = i;
7       U[i]=1;
8     }
9   for (; st <= dr; st++){
10    p=G[C[st]];
11    while (p != NULL){
12      deg[p->nod]--;
13      if (!U[p->nod] &&
14          deg[p->nod] == 0){
15        C[++dr] = p->nod;
16        U[p->nod] = 1;
17      }
18      p = p->urm;
19    }
20  }
21 }
```

Prezentăm și varianta de implementare C++ cu ajutorul containerilor STL:

```
1 ...
2
3 #define pb push_back
4
5 using namespace std;
6 vector< int > L[50005], C;
7 vector<int> :: iterator it;
8 int n, m, i, x, y ; bool sel[50005];
9
10 void load(){
11   cin >> n >> m;
12   for (i=1; i<= m; i++){
13     cin >> x >> y;
14     L[x].pb(y);
15   }
16 }
17
18 void dfs(int x){
19   vector<int> :: iterator it;
20   sel[x]=true;
21   for(it=L[x].begin(); it!=L[x].end(); it++)
22     if (!sel[*it]) dfs(*it);
23   C.pb(x);
24 }
25
26 int main(){
```

```

27 ...
28 load();
29 memset(sel, false, sizeof(sel));
30 for (i=1; i<=n; i++)
31     if (!sel[i]) dfs(i);
32
33 reverse(C.begin(), C.end());
34
35 for(it=C.begin(); it!=C.end(); it++)
36     cout << *it << ' ';
37 cout << '\n';
38 }

```

## 8. Puncte de articulație – critice $O(N+M)$

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

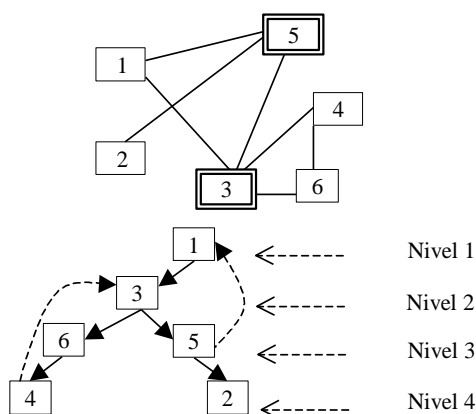
### Soluție:

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

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

- numărul nivelului atins în parcurgerea  $DF$ , memorat în vectorul  $nv$  pe poziția  $x$  ( $nv[x]$ );
- numărul minim al nivelului care poate fi atins din  $x$  folosind descendenții săi și 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 descendenții lui. Acest număr va fi reținut în vectorul  $L$  pe poziția  $x$  ( $L[x]$ );
- vârful părinte în arborele  $DF$ , reținut în vectorul  $t$  pe poziția  $x$  ( $t[x]$ ).

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 articulație; eliminarea acestuia, împreună cu muchiile adiacente, ar duce la izolarea nodului  $x$ .



Considerăm graful din figura alăturată. El conține două puncte de articulație: nodul 3 și 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 descendenții 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 descendenților săi și 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 articulație. Analog pentru nodul 5.

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

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

```
...
#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 reține pentru fiecare nod, valoarea 0 dacă nodul este critic și 1, în caz contrar. Vectorul *U* codifică, în timpul parcurgerii *DF*, starea unui nod: vizitat sau nevizitat.

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

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

```
1  ...
2  void DF(int nod)
3  {lista *p;
4    U[nod] = 1;
5    L[nod] = nv[nod];
6    for (p = G[nod]; p != NULL; p = p->urm)
7      if (!U[p->nod]) {
8        nv[p->nod] = nv[nod]+1;
9        T[p->nod] = nod;
10       if (nod == rad) nr++;
11       DF(p->nod);
12       if (L[nod] > L[p->nod])
13         L[nod] = L[p->nod];
14       if (L[p->nod] >= nv[nod])
15         c[nod] = 1;
16     }
17     else
18       if (p->nod != T[nod] && L[nod] > nv[p->nod])
19         L[nod] = nv[p->nod];
20   }
21
22   int main(void)
23   {int i;
24     citeste_graf();
25     for (i = 1; i <= N; i++)
26       if (!U[i])
27         {nv[i] = 1;
```



```

28     rad = i;
29     nr = 0;
30     DF(i);
31     c[rad] = nr > 1;
32 }
33 for (i = 1; i <= N; i++)
34     if (c[i]) cout << i << ' ';
35     return 0;
36 }
37
38

```

## 9. Componente biconexe $O(N+M)$

Prin definiție, un graf  $G=(V, E)$  este biconex dacă nu conține puncte de articulație. Prin componentă biconexă se înțelege un subgraf maximal în raport cu proprietatea de biconexitate. Realizați un program care determină muchiile fiecărei componente biconexe a unui graf.

### Soluție:

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

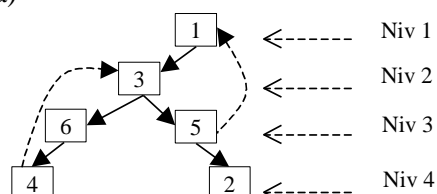
- parcurgerea  $DF$  pentru determinarea numărului minim al nivelului care poate fi atins din fiecare nod folosind descendenții acestuia și cel mult o muchie de întoarcere. Aceste numere vor fi reținute în vectorul  $L$  pe poziția  $x$  ( $L[x]$ ).
- în timpul parcurgerii  $DF$  se vor afișa muchiile din fiecare componentă biconexă. Această operație 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 afișa o nouă componentă biconexă. Ea va fi formată din muchiile din stivă, operația de extragere din stivă se oprește la găsirea muchiei ( $t[x], x$ ) în vârful stivei.

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



### Etapele algoritmului:

a)



După parcurgerea  $DF$  se vor obține 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 conține:  $st=((1,3),(3,6),(4,6),(3,4))$ . Se elimină muchii din stivă până la găsirea muchiei

#### Pasul 3:

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

(t[6],6)=(3, 6).

La final stiva va conține:  $st=((1,3))$

*Pasul 2:*

Se găsește nodul 2 care nu poate ajunge mai sus:

$st=((1,3),(3,5),(2,5))$

Se afișează componenta biconexă formată din succesiunea de muchii până la muchia (2,5).

La final stiva va conține:  $st=((1,3),(3,5))$

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

*Pasul 4:*

Stiva vidă, algoritmul ia sfârșit.

```
1  const int NMAX = 1e5 + 1;
2  const int ROOT = 1;
3
4  int N, M;
5  int Level[NMAX], Low[NMAX];
6  vector < int > G[NMAX];
7  stack < int > Stack;
8  vector < vector < int > > Sol;
9
10 void Read () {
11     f >> N >> M;
12     for(int i = 1; i <= M; ++i)
13     {
14         int X = 0, Y = 0;
15         f >> X >> Y;
16         G[X].push_back(Y), G[Y].push_back(X);
17     }
18 }
19
20 int my_min (int a, int b){
21     return ((a < b) ? a : b);
22 }
23
24 void DFS (int Node, int L = 1){
25     Level[Node] = Low[Node] = L;
26
27     Stack.push(Node);
28
29     for(auto it : G[Node])
30         if(Level[it]) //muchie de intoarcere
31             Low[Node] = my_min(Low[Node], Level[it]);
32     else {/// muchie de avans
33         DFS(it, L + 1);
34
35         Low[Node] = my_min(Low[Node], Low[it]);
36         if(Low[it] >= Level[Node])
37             { ///Node este punct de articulatie
38                 vector < int > sol;
39                 int Elem = 0;
40                 do
41                 {
42                     Elem = Stack.top(), sol.push_back(Elem);
43
44                     Stack.pop();
45                 }
46                 while(Elem != it);
47
48                 sol.push_back(Node); /// se adauga si Node
49
50                 Sol.push_back(sol);
51                 ///se adauga o noua componenta biconexa
52             }
53     }
54 }
55
56 void Solve () {
```

```

57     DFS(ROOT);
58
59     g << (int)Sol.size() << '\n';
60     for(auto it : Sol) {
61         for(int j = 0; j < (int)it.size(); ++j)
62             {
63                 g << it[j];
64
65                 if(j != (int)it.size() - 1)
66                     g << ' ';
67             }
68         g << '\n';
69     }
70 }
...

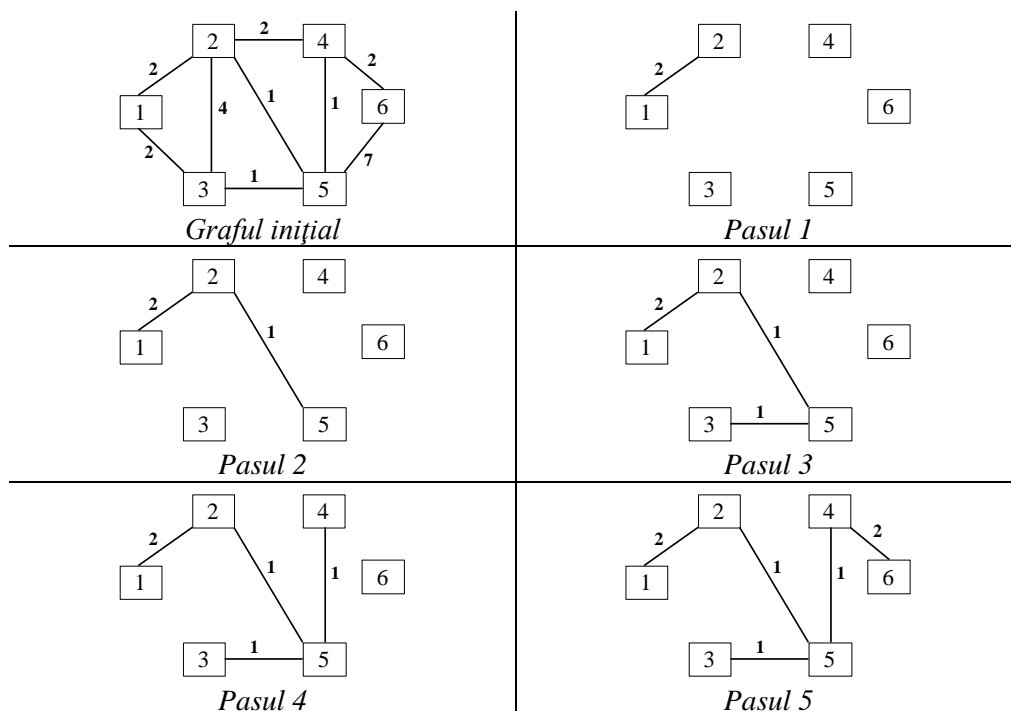
```

### 10. Arbore parțial de cost minim (A.P.M) – Algoritmul lui Prim /// $O(N^2)$

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

#### Soluție:

Algoritmul lui Prim construiește arborele parțial 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 construcție al A.P.M. cu ajutorul algoritmului lui Prim pe graful următor, considerând ca rădăcină nodul 1:



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

Transcrierea în pseudocod a algoritmului folosește următoarele structuri de date:

- tabloul  $D$ , în care elementul  $D[x]$  va reține costul minim prin care putem „lega” nodul  $x$ , neconectat la arbore, de orice nod al arborelui.
- tabloul  $T$ , în care elementul  $T[x]$  va reține nodul din arbore de care nodul  $x$  a fost conectat cu costul  $D[x]$ .

- Lista  $Q$  conține pe tot parcursul algoritmului nodurile grafului  $G$  neconectate la arbore.

```

1  Prim (G=(V,E,cost), rad) //complexitate O(N*N)
2  Q ← V; //lista Q contine initial toate nodurile din G
3  D ← ∞;
4  D[rad] ← 0; T[rad] = 0;
5  cat timp (Q ≠ ∅) executa
6  |   x ← minim (Q) ;
7  |   Q ⇒ {x}
8  |   pentru fiecare y ∈ Q, adiacent cu x executa
9  |   |   daca (cost[x,y] < d[y] ) atunci
10 |   |   |   T[y] = x;
11 |   |   |   D[x] = cost[x,y];
12 |   |   |
13 |   |   |
14 |   |   |

```

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

```

#include <iostream>
#define MAX_N 101
#define INF 30000
int N, M, R, C[MAX_N][MAX_N], D[MAX_N], T[MAX_N], cost;
char U[MAX_N];

```

Graful este memorat cu ajutorul matricei costurilor. Parametrul  $x$  indică nodul desemnat ca rădăcină a arborelui parțial de cost minim. Lista  $Q$  a fost codificată cu ajutorul vectorului  $Use$ , astfel  $Use[i]=1$ , dacă nodul  $i$  aparține arborelui și 0, în caz contrar.

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

```

1  void prim(int x)
2  {
3      int i, min, nod;
4
5      memset(U, 0, sizeof(U));
6      memset(T, 0, sizeof(T));
7
8      for (i = 1; i <= N; i++)
9          D[i]=C[x][i], T[i]=x;
10     U[x] = 1;
11
12     while (1)
13     {
14         min = INF; nod = -1;
15         for (i = 1; i <= N; i++)
16             if (!U[i] && min> D[i])
17                 min = D[i], nod = i;
18         if (min == INF) break;
19
20         U[nod] = 1;
21         cost += D[nod];
22         cout << T[nod] << ' ' << nod << '\n' ;
23
24         for (i = 1; i <= N; i++)
25             if (D[i] > C[nod][i])
26             {
27                 D[i] = C[nod][i];
28                 T[i] = nod;
29             }
30     } cout << cost << '\n';
31 }

```

## 11. Arbore parțial de cost minim (A.P.M) – Algoritmul lui Kruskal /// $O(N*M)$

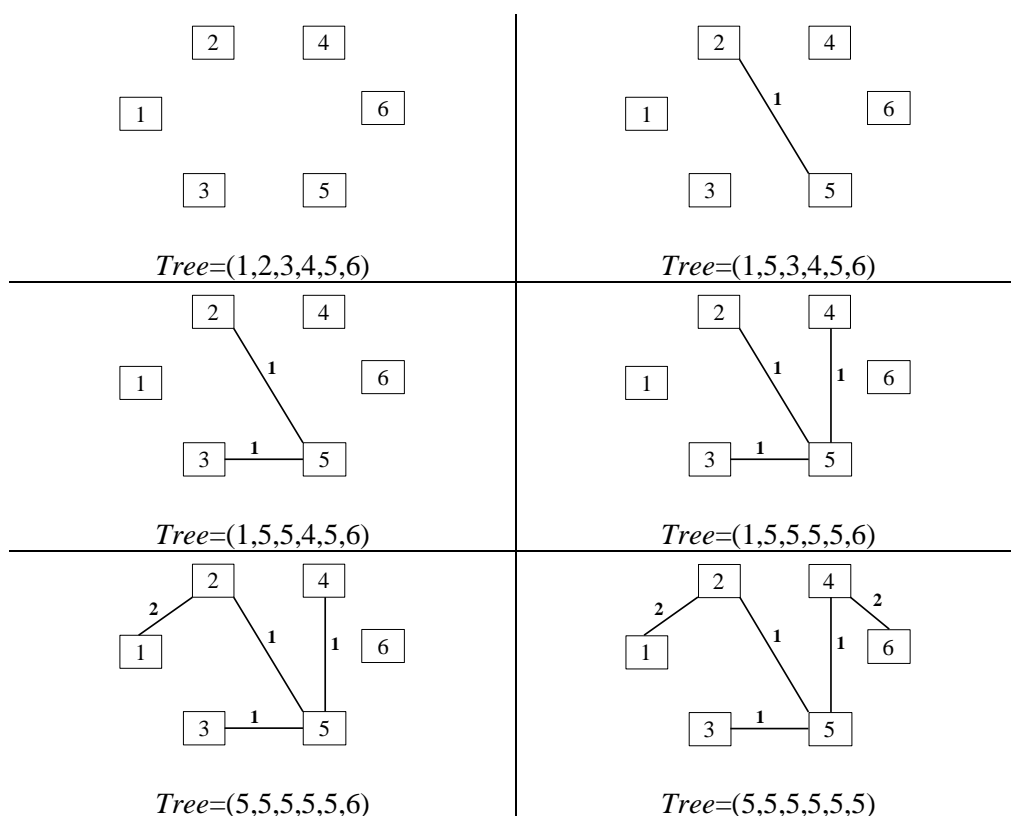
Fie  $G=(V,E)$  un graf neorientat conex, cu costuri asociate muchiilor. Un arbore parțial al lui  $G$  este un graf parțial conex și fără cicluri. Realizați un program care determină un arbore parțial de cost minim, adică un arbore parțial pentru care suma costurilor tuturor muchiilor sale este minimă.

Soluție:

Considerăm  $N$  cardinalul mulțimii  $V$  și  $M$  cardinalul mulțimii  $E$ . Algoritmul lui Kruskal construiește arborele parțial de cost minim, pornind de la  $N$  arbori disjuncți, notați  $T_1, T_2, \dots, T_N$ . Fiecare vârf al grafului definește la momentul inițial, câte un arbore. La fiecare pas al algoritmului vor fi aleși doi arbori care se vor unifica. În finalul algoritmului se va obține 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 unificați la fiecare etapă a algoritmului. Regula respectată este următoarea:

Se identifică muchia de cost minim, nefolosită anterior, și care are vârfurile extreme în doi arbori disjuncți. În felul acesta, prin unirea celor doi arbori va rezulta tot un arbore (nu se vor crea cicluri). Să privim modul de construcție al A.P.M. cu ajutorul algoritmului lui *Kruskal* pe graful luat ca exemplu la prezentarea algoritmului lui *Prim*:

Vectorul *Tree* codifică arborii disjuncți astfel  $Tree[x]=y$  semnifică faptul că vârfurile  $x$  și  $y$  sunt în același arbore.



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

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

```

1  Kruskal ( $G=(V,E,\text{cost})$ ) //complexitate  $O(N*M)$ 
2   $A \leftarrow \emptyset$ ; //lista muchiilor din A.P.M
3  pentru orice varf  $i$  din  $V$  executa
4   $Tree[i] \leftarrow i$ 
5  Sortare a listei muchiilor  $\in E$ , crescator dupa cost
6  pentru fiecare  $(x,y) \in E$  executa

```

```

9      |   daca Tree[x]≠Tree[y] atunci
10     |   |   A U {(x,y)};   reuneste(x,y);
11     |   |   ■
12     |   |   ■
13     |   returneaza A

```

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

```
#include <iostream>
#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 aceeași semnificație ca a vectorului  $Tree$  prezentat anterior în exemplu și în pseudocod.

```

1 void reuneste(int i, int j)
2 { int k;
3   i = T[i];
4   j = T[j];
5   if (i == j) return;
6   for (k = 1; k <= N; k++)
7     if (T[k] == i) T[k] = j;}
8
9 bool cmp(muchie a, muchie b)
10 {return a.c<b.c;}
11
12 void kruskal(void)
13 { int i, j, k, c;
14   sort(e, e + M, cmp);
15   for (i=1;i<=N;i++) T[i]=i;
16   for (k = 0; k < M; k++)
17     {i = e[k].x; j = e[k].y;
18      c = e[k].c;
19      if (T[i]==T[j]) continue;
20      reuneste(i, j);
21      cost += c;
22      cout << i << ' ' << j << ' ' << c << '\n';
23    }
24   cout << "Cost minim = " << cost << '\n';
25 }

```

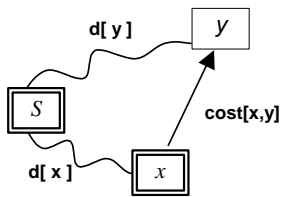
## 12. Drumuri minime de sursă unică - Algoritmul lui Dijkstra /// $O(N^2)$

Fie  $G=(V,E)$  un digraf cu costuri pozitive asociate arcelor. Fiind desemnat un vârf  $s$  ca punct de plecare - *sursă*, să se determine pentru orice pereche de vârfuri  $\{s,x\}$ , costul drumului minim care le unește.

*Soluție:*

Algoritmul lui *Dijkstra* rezolvă problema enunțată folosind o strategie tip *Greedy*. Notăm cu  $Q$  mulțimea vârfurilor digrafului și cu  $Use$ , mulțimea vârfurilor pentru care s-au determinat deja drumurile minime de la sursă. Algoritmul identifică, în cadrul fiecărei iterații, vârful  $x$  neselectat anterior ( $x \in Q - Use$ ), vârful ce are proprietatea că este cel mai “apropiat” de sursă.

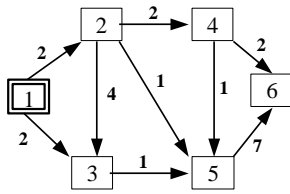
O dată cu identificarea și selectarea lui  $x$ , pentru toate vârfurile  $y$  adiacente cu el se încearcă o minimizare a costurilor drumurilor prin care acestea sunt legate de sursă. Această operație se efectuează cu ajutorul drumului minim ce trece prin  $x$ . Considerăm că vectorul  $D$  memorează costurile drumurilor minime de la sursă către orice vârf din graf.



Condiția ca distanța drumului de la  $s$  la  $y$  să poată fi minimizată prin vârful  $x$  este următoarea:

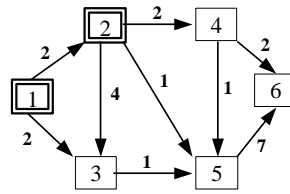
$$d[y] > d[x] + \text{cost}[x,y]$$

Să privim modul de operare al algoritmului lui *Dijkstra* pe graful următor, considerând ca sursă nodul 1:



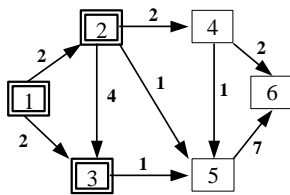
Iterația nr. 1

Vârful selectat  $x=1$ ,  $D=(0,2,2,\infty,\infty,\infty)$



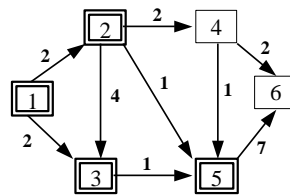
Iterația nr. 2

Vârful selectat  $x=2$ ,  $D=(0,2,2,4,3,\infty)$



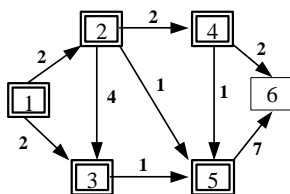
Iterația nr. 3

Vârful selectat  $x=3$ ,  $D=(0,2,2,4,3,\infty)$



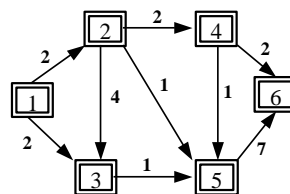
Iterația nr. 4

Vârful selectat  $x=5$ ,  $D=(0,2,2,4,3,10)$



Iterația nr. 5

Vârful selectat  $x=4$ ,  $D=(0,2,2,4,3,6)$



Iterația nr. 6

Vârful selectat  $x=6$ ,  $D=(0,2,2,4,3,6)$

Transcrierea în pseudocod a algoritmului folosește următoarele structuri de date:

- tabloul  $D$ , în care elementul  $D[x]$  va reține costul minim de la  $s$  la  $x$ .
- tabloul  $T$ , în care elementul  $T[x]$  va reține vârful precedent lui  $x$  pe drumul minim de la  $s$  la  $x$ .
- Lista  $Q$  conține, pe tot parcursul algoritmului, vârfurile digrafului  $G$  pentru care nu s-au determinat costurile minime ale drumurilor de la sursă către ele.

```

1  Dijkstra ( $G=(V,E,\text{cost}),s$ ) //complexitate  $O(N^2)$ 
2   $Q \leftarrow V$ ; //lista  $Q$  contine initial toate nodurile din  $G$ 
3   $Use \leftarrow \emptyset$ ;
4   $D \leftarrow \infty$ ; //distanțele minime de la  $s$  la varfurile din  $G$ 
5   $D[s] \leftarrow 0$ ;  $T[s] = 0$ ;
6  cat timp ( $Q \neq \emptyset$ ) executa
7  |    $x \leftarrow \text{minim}(Q)$ ;
8  |    $Q \Rightarrow \{x\}$ ;  $Use \leftarrow \{x\}$ ;
9  |   pentru fiecare  $y \in Q$ , adiacent cu  $x$  executa
10 | |   daca ( $d[x] + \text{cost}[x,y] < d[y]$ ) atunci
11 | | |    $T[y] = x$ ;
12 | | |    $D[y] = d[x] + \text{cost}[x,y]$ ;
13 | |
14 |
15 |

```

Implementarea în limbaj a algoritmului lui *Dijkstra*, prezentată în continuare, ia în considerare următoarele declarații:

```

...
#include <iostream>
#define MAX_N 101
#define INF 0x3f3f

```

```

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

```

Graful este memorat cu ajutorul matricei costurilor. Lista  $Q$  a fost codificată cu ajutorul vectorului  $Use$ , astfel  $Use[i]=1$ , dacă pentru vârful  $i$  s-a determinat costul drumului minim și 0, în caz contrar.

Subprogramul *scrie\_drum* poate fi apelat pentru afișarea vârfurilor de pe fiecare drum minim determinat cu ajutorul subprogramului *dijkstra*.

```

1 void dijkstra(int sursa)
2 {
3     int i, min, nod;
4     memset(U, 0, sizeof(U));
5     memset(T, 0, sizeof(T));
6     memset(D, 0x3f, sizeof(D));
7     D[sursa] = 0;
8
9     while (1)
10    {
11        min = INF;
12        nod = -1;
13        for (i = 1; i <= N; i++)
14            if (!U[i] && min > D[i])
15                min = D[i], nod = i;
16
17        if (min == INF) break;
18        U[nod] = 1;
19
20        for (i = 1; i <= N; i++)
21            if (D[i] > D[nod] + C[nod][i])
22            {
23                D[i] = D[nod] + C[nod][i];
24                T[i] = nod;
25            }
26    }
27 }
28
29 void scrie_drum(int i)
30 {
31     if (T[i])
32         scrie_drum(T[i]);
33     cout << i << ' ';
34 }

```

### 13. Drumuri minime între oricare vârfuri. Algoritmul lui Roy-Floyd /// $O(N^3)$

Fie  $G=(V,E)$ , graf orientat cu costuri reale asociate arcelor. Să se determine, pentru orice pereche de vârfuri  $x, y \in V$ , costul drumului minim de la  $x$  la  $y$ .

#### Soluție:

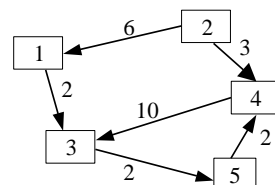
Numim matricea a drumurilor minime, o matrice pătratică  $D(N,N)$  în care valoarea elementului  $D[i,j]$  indică costul minim al unui drum ce leagă vârful  $i$  de vârful  $j$ .

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

```

0  ∞  2  6  4
6  0  8  3 10
∞  ∞  0  4  2
∞  ∞ 10  0 12
∞  ∞ 12  2  0

```

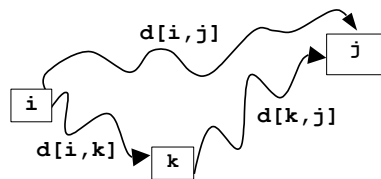


Algoritmul *Roy-Floyd* construiește matricea drumurilor minime, plecând de la matricea



ponderilor - *Cost*.

Inițial, matricea  $D$  reține costurile arcelor prezente în digraf. Algoritmul parcurge de  $n$  ori matricea  $D$ , câte o dată pentru fiecare vârf  $k$  al digrafului. Astfel, între orice pereche de vârfuri  $i$  și  $j$  se încearcă minimizarea costului drumului dintre ele prin concatenarea drumurilor de la  $i$  la  $k$  și de la  $k$  la  $j$ .



Regula descrisă mai sus poate fi exprimată astfel:

$$D[i,j] = \min(D[i,j], \{D[i,k] + D[k,j]\})$$

Pentru identificarea vârfurilor plasate pe drumurile minime, nu mai este folosită o informație de tip vârf “predecesor” (tată), deoarece minimizarea se face prin concatenarea a două drumuri și nu a unui drum cu un arc (ca în cazul algoritmului *Dijkstra*). Din această cauză se va construi matricea *Urm* care va reține, pentru orice pereche de vârfuri  $i$  și  $j$ , vârful care îi urmează lui  $i$  pe drumul minim către  $j$ .

Transcrierea în pseudocod a algoritmului lui *Roy-Floyd* este următoarea:

```

1  Roy-Floyd (G=(V,E,cost))          /*complexitate: O(N³)*/
2  D ← Cost;                          //initializare matrice drumuri minime
3  pentru i ← 1,n executa
4  | pentru j ← 1,n executa
5  | | daca (c[i,j]>0)AND (c[i,j]<∞) atunci Urm[i,j] ← j
6  | | altfel Urm[i,j] ← 0
7  | |
8  | |
9  | |
10 | pentru k ← 1,n executa
11 | | pentru i ← 1,n executa
12 | | | pentru j ← 1,n executa
13 | | | D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
14 | | | Urm[i,j] ← Urm[i,k]
15 | | |
16 | | |
17 | | |

```

Implementarea în limbaj a subprogramului ce realizează construirea matricei drumurilor minime, prezentată în continuare, ia în considerare următoarele declarații:

```

...
#define MAX_N 50
#define INF 0x3f3f
int N, M, C[MAX_N][MAX_N], D[MAX_N][MAX_N], Urm[MAX_N][MAX_N];

```

Ca și în pseudocod, digraful este considerat a fi memorat cu ajutorul matricei ponderilor.

```

1  void floyd()
2  {int i, j, k;
3  for (i = 1; i <= N; i++)
4  for (j = 1; j <= N; j++)
5  if (C[i][j]!=0 && C[i][j]!=INF)
6  Urm[i][j]=j;
7  else Urm[i][j]=0;
8
9  memcpy(D, C, sizeof(C));
10 for (k = 1; k <= N; k++)
11 for (i = 1; i <= N; i++)
12 for (j = 1; j <= N; j++)
13 if (D[i][j]>D[i][k]+D[k][j])
14 {
15 D[i][j]=D[i][k]+D[k][j];
16 Urm[i][j] =Urm[i][k];
17 }
18 }

```

```

19
20 void scrie_drum(int i, int j)
21 {
22     cout << i << ' ';
23     if (i!=j)
24         scrie_drum(Urm[i][j],j);
25     else return;
26 }

```

#### 14. Algoritmul lui Dijkstra – implementare folosind heap-uri $O(N \log N)$

Fie  $G=(V,E)$  un digraf cu costuri pozitive asociate arcelor. Fiind desemnat un vârf  $s$  ca punct de plecare - sursă, să se determine pentru orice pereche de vârfuri  $\{s,x\}$ , costul drumului minim care le unește.

##### Soluție:

În continuare vom arăta cum se poate folosi un min-heap pentru a obține complexitate  $O(N \log N)$ .

Vom implementa mulțimea  $Q$  folosind un min-heap, astfel extragerea minimului va avea complexitate  $O(\log N)$  (se șterge elementul din vârful heap-ului). Când se modifică distanțele în vectorul  $D$  este necesară și o ridicare în heap a nodului modificat. Pentru a găsi în  $O(1)$ , pe ce poziție se află fiecare nod în heap, se mai folosește un vector suplimentar  $poz[]$  cu aceste informații.

```

1  ...
2  #define MAX_N 101
3  #define INF 0x3f3f
4  struct lista
5  { int nod, c; lista *urm;
6  } *G[MAX_N];
7  int N, M, S, D[MAX_N],
8  T[MAX_N], H[MAX_N], poz[MAX_N], nh;
9  char U[MAX_N];
10 ...
11
12 void swap(int i, int j)
13 {
14     int t;
15     t=H[i]; H[i]=H[j]; H[j]=t;
16     poz[H[i]] = i; poz[H[j]] = j;
17 }
18
19 void HeapDw(int r, int k)
20 {
21     int St,Dr,i;
22     if (2*r+1<=k)
23     {
24         St=H[2*r+1];
25         if (2*r+2 <=k) Dr=H[2*r+2];
26         else Dr=St-1;
27         if (St>Dr) i=2*r+1;
28         else i=2*r+2;
29         if (D[H[r]]>D[H[i]]) {
30             swap(r,i);
31             HeapDw(i,k);
32         }
33     }
34 }
35
36 void HeapUp(int k)
37 {
38     int t;
39     if (k <= 0) return;
40     t = (k-1)/2;
41     if (D[H[k]]<D[H[t]])
42     {swap(k,t);
43       HeapUp(t);
44     }

```

```

45 }
46
47 void BuildH(int k)
48 {
49     int i;
50     for (i=1; i<k; i++) HeapUp(i);
51 }
52
53 int scoate_heap()
54 {
55     swap(0, nh-1);
56     poz[H[nh-1]]=0; nh--;
57     HeapDw(0, nh-1);
58     return H[nh];
59 }
60
61 void dijkstra(int sursa)
62 {
63     int i, nod; lista *p;
64     memset(U, 0, sizeof(U));
65     memset(T, 0, sizeof(T));
66     memset(D, 0x3f, sizeof(D));
67     for (i = 0; i < N; i++) {
68         H[i] = i+1; poz[i+1] = i;
69     }
70
71     D[sursa]=0; BuildH(N); nh=N;
72     while (nh > 0) {
73         nod = scoate_heap();
74         for (p=G[nod]; p; p=p->urm)
75             if (D[p->nod] > D[nod] + p->c) {
76                 D[p->nod] = D[nod] + p->c;
77                 T[p->nod] = nod;
78                 HeapUp(poz[p->nod]);
79             }
80     }
}

```

## 15. Algoritmul lui Bellman-Ford

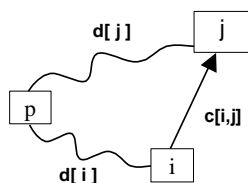
Se consideră un graf orientat  $G=(V, E)$  și o funcție  $c : E \rightarrow \mathbb{R}$ . Mulțimea  $V$  conține  $n$  vârfuri. Se desemnează un vârf  $p$  de plecare. Pentru orice vârf  $j \in V$  se cere determinarea drumului de cost minim de la  $p$  la  $j$ . Se vor detecta situațiile în care există circuite de cost negativ care includ nodul  $p$ .

### Soluție:

Algoritmul *Bellman-Ford* determină drumurile de cost minim dintre un nod desemnat ca sursă (plecare) și restul vârfurilor accesibile lui chiar dacă există costuri negative pe arce. Aceste rezultate sunt furnizate numai în situația în care nodul de plecare nu face parte dintr-un circuit de cost negativ.

Strategia algoritmului este aceea de minimizare succesivă a costului drumului de la  $p$  la orice nod  $j$  din graf ( $D[j]$ ) până la obținerea costului minim.

Această operație se face prin verificarea posibilității ca fiecare arc  $(i,j) \in E$  să participe la minimizarea distanței de la  $p$  la  $j$ . Operația va face o trecere completă prin toate arcele grafului.



Condiția ca distanța de la  $p$  la  $j$  să poată fi minimizată prin arcul  $(i,j)$  este ca:

$$d[j] > d[i] + c[i,j].$$

Notăm cu  $n$  numărul de vârfuri ale grafului. Algoritmul efectuează  $n-1$  treceri complete prin mulțimea arcelor grafului (orice drum elementar poate fi format din maximum  $n-1$  arce).

În final, existența unui circuit negativ face ca la o nouă trecere prin mulțimea arcelor să fie în continuare posibilă minimizarea costului unui drum. În acest caz algoritmul evidențiază prezența circuitului negativ ce cuprinde nodul sursă.

În situația în care graful este memorat în liste de adiacență, complexitatea algoritmului este  $O(N*M)$ .

```

1  ...
2  #define NMAX 50001
3  #define INF 0x3f3f3f3f
4  #define pb push_back
5  using namespace std;
6  ifstream in("bellmanford.in");
7  ofstream out("bellmanford.out");
8  vector< pair< int, int > > G[NMAX];
9  queue< int > Q;
10 int N, M, i, x, y, c, D[NMAX], It, ItNod[NMAX], Nod;
11 bool USED[NMAX];
12
13 int main()
14 {
15     in >> N >> M;
16     for( ; M--> 0; ){
17         in >> x >> y >> c;
18         G[x].pb( { y, c } );
19     }
20     memset( USED, false, sizeof(USED) ); //initial nu avem noduri in coada
21     memset( D, INF, sizeof(D) );        //toate distantele sunt infinit
22     D[1] = 0;                           //mai puțin cea până la sursa
23     memset( ItNod, 0, sizeof(ItNod) );  //nu s-a trecut niciodată prin niciun nod
24     Q.push( 1 );                        //introducem nodul 1 în coada
25     USED[1] = true;
26
27     while( !Q.empty() ){
28         Nod = Q.front();
29         Q.pop();
30         USED[Nod] = false;               // scoatem nodul din coada
31         for( auto it: G[Nod] )
32             if( D[it.first] > D[Nod] + it.second ) {
33                 D[it.first] = D[Nod] + it.second;
34                 if( !USED[it.first] )    // dacă nodul nu se afla în coada
35                     {
36                         Q.push( it.first ); // îl introducem
37                         USED[it.first] = true;
38                         if( ++ItNod[it.first] > N )
39                             // dacă s-a trecut de mai mult de N ori prin el
40                             {
41                                 out << "Ciclu negativ!\n";
42                                 return 0;
43                             }
44                     }
45             }
46     }
47
48     ...
49 }

```

## 16. Algoritmul lui Kruskal (A.P.M.) – implementare folosind păduri de mulțimi disjuncte

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

### Soluție:

Algoritmul lui Kruskal pentru determinarea arborelui parțial de cost minim a mai fost prezentat în capitolul 2.2.2. În continuare vom arăta cum se poate implementa acesta într-o complexitate  $O(M)$

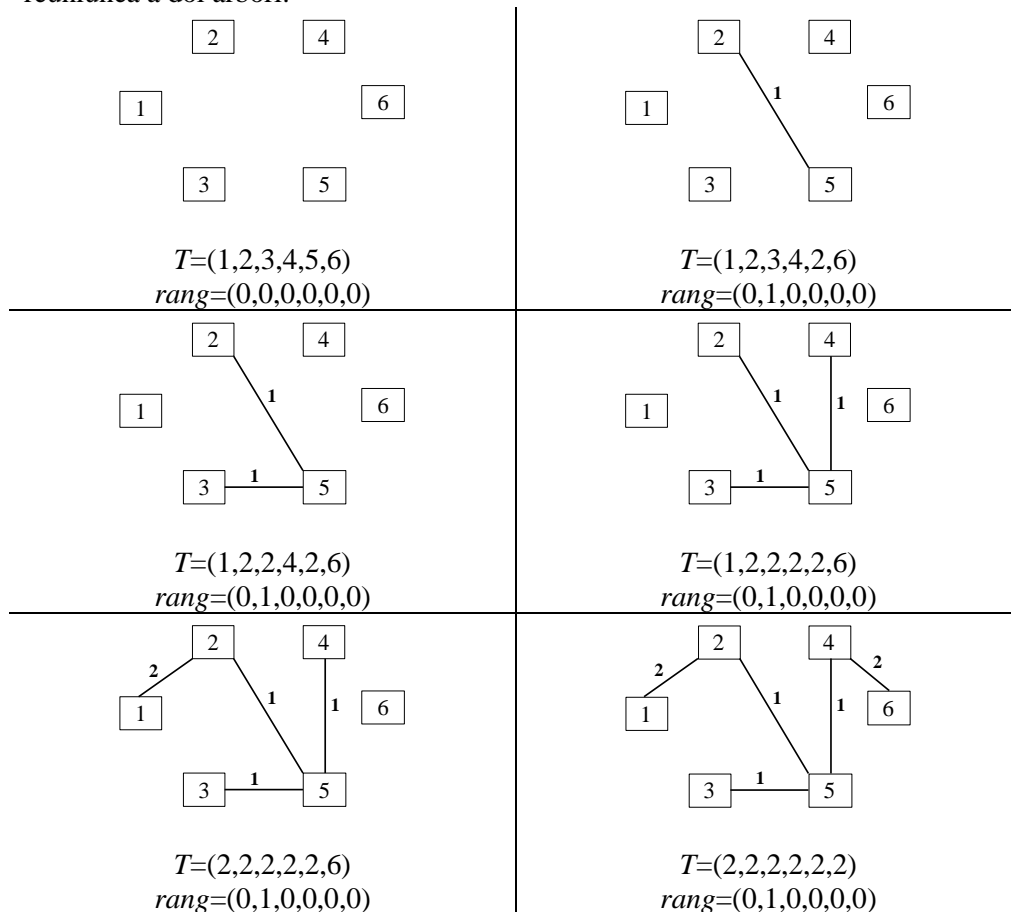
$\log^*N$ ) folosind o pădure de arbori pentru a menține mulțimile care se construiesc pe parcurs.

Fiecare nod reprezintă un element dintr-o mulțime și fiecare arbore reprezintă o mulțime. Fiecare element indică doar spre părintele lui, iar rădăcina fiecărui arbore conține reprezentantul mulțimii (care este propriul său părinte).

Pentru a uni două mulțimi se dorește ca rădăcina arborelui cu mai puține noduri să indice spre rădăcina arborelui cu mai multe noduri. Pentru fiecare nod se va menține un "rang" care aproximează logaritmul dimensiunii subarborelui și care este, de asemenea, o margine superioară a înălțimii nodului. Așadar, rădăcina cu rangul cel mai mic va indica spre rădăcina cu rang mai mare.

Pentru a verifica dacă două elemente fac parte din aceeași mulțime se determină, pentru fiecare, rădăcina arborelui din care fac parte, și se verifică dacă acestea coincid. O implementare directă a operațiilor descrise ar avea complexitatea  $O(\log N)$  pentru fiecare operație. Folosind o euristică de "comprimare a drumului" se poate reduce complexitatea fiecărei operații la  $O(\log^* N)$ . După ce se determină reprezentantul unui element (rădăcina arborelui din care face parte) se modifică fiecare nod parcurs în drumul către rădăcină astfel încât să aibă ca părinte rădăcina determinată. Comprimarea drumului nu modifică nici un rang.

Aplicarea acestei structuri de date în algoritmul lui *Kruskal* este evidentă: inițial fiecare nod face parte dintr-o mulțime distinctă, iar introducerea unei noi muchii în arborele de cost minim se face prin reuniunea a doi arbori.



Prezentăm varianta de implementare C++ cu ajutorul containerilor STL:

```

1 #define pb push_back // O(MlogN + MlogM)
2 #define f first //utilizand paduri disjuncte
3 #define s second
4
5 using namespace std;
6
7 int N, M, Rep[200010], x, y, c, i;
8 vector <pair <int, int> > Sol;
9 vector <pair <int, pair <int, int> > > L;
```

```

10
11 int Find(int nod)
12 {
13     if (Rep[nod] != nod)
14         Rep[nod] = Find(Rep[nod]);
15     return Rep[nod];
16 }
17
18 int main()
19 {
20     . . .
21     cin >> N >> M;
22     for (i = 0; i < M; i++){
23         cin >> x >> y >> c;
24         L.pb({c, {x, y}});
25     }
26
27     sort(L.begin(), L.end());
28
29     for (int i = 0; i <= N; i++) Rep[i]=i;
30
31     int Cost = 0;
32
33     for (i = 0; i < M; i++){
34         int n1 = L[i].s.f;
35         int n2 = L[i].s.s;
36
37         if (Find(n1) != Find(n2)){
38             Cost += L[i].f;
39             Sol.pb({n1, n2});
40             Rep[Rep[n2]] = Rep[n1];
41         }
42     }
43
44     cout << Cost << '\n';
45     cout << Sol.size() << '\n';
46     for (i=0; i < Sol.size(); i++)
47         cout << Sol[i].f << ' ' << Sol[i].s << '\n';
48
49     return 0;
50 }

```

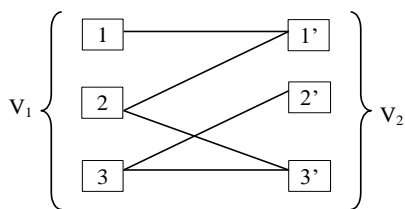
### 17. Cuplaj maxim în graf bipartit

Fie  $G=(V_1, V_2, E)$  un graf neorientat bipartit. Un cuplaj este o submulțime de muchii astfel încât pentru toate vârfurile din graf există cel mult o muchie din cuplaj incidentă vârfului. Realizați un program care determină un cuplaj de cardinalitate maximă.

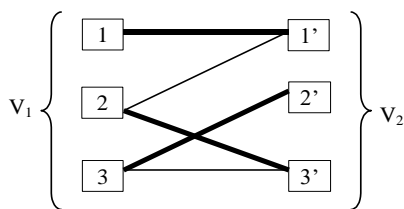
*Exemplu:* Pentru  $\text{card}(V_1)=\text{card}(V_2)=3$ ,  $m=5$  și muchiile  $(1,1')$   $(2,1')$   $(2,3')$   $(3,2')$   $(3,3')$  se va afișa un cuplaj maxim format din 3 muchii:  $(1,1')$   $(2,3')$   $(3,2')$ .

#### Soluție:

Un graf este bipartit dacă nodurile sale pot fi împărțite în două submulțimi  $V_1$ ,  $V_2$ , astfel încât oricare două noduri aparținând aceleiași submulțimi  $V_i$ , nu sunt adiacente: dacă  $\{x, y\} \in V_i \Rightarrow (x, y) \notin E$ . Vom considera în continuare că  $V_1$  conține  $N_1$  elemente, iar  $V_2$  conține  $N_2$  elemente. Problema găsirii unui cuplaj maxim într-un graf bipartit are numeroase aplicații practice. Să privim graful din exemplu și cuplajul maxim determinat:



*Graful inițial*



*Cuplajul maxim obținut*

Se poate obține un algoritm de rezolvare construind o rețea de transport și aplicând algoritmul lui *Ford-Fulkerson* de flux maxim, totuși implementarea fiind destul de dificilă. Vom prezenta în continuare un algoritm mult mai ușor de implementat, cu complexitate  $O(N_1^2 \cdot N_2)$  (se poate reduce complexitatea la  $O(N_1 \cdot |E|)$  folosind liste de adiacență).

Acest algoritm se folosește de următoarea proprietate: dacă la un moment dat un nod  $x$  din  $V_1$  a fost cuplat cu un nod  $y$  din  $V_2$ , nu se va decupla niciodată nodul  $x$  pentru a obține un cuplaj maxim, ci cel mult acesta va fi recuplat cu un alt nod  $z$ .

Această proprietate permite iterarea prin nodurile din  $V_1$  într-o ordine oarecare și încercarea de cuplare a fiecăruia. Pentru a cupla un nod  $x$  din  $V_1$  se folosește o procedură recursivă care încearcă să cupleze  $x$  cu un nod  $y$  din  $V_2$  dacă nodul  $y$  nu este deja cuplat, sau dacă nodul cu care  $y$  este cuplat poate fi recuplat cu un nod diferit de  $y$  (lucru care se verifică apelând aceeași procedură).

Pentru a nu cicla la infinit se folosește un vector  $U$  care menține informații despre nodurile care au fost deja apelate în procedura recursivă. În implementarea prezentată mai jos se folosește o optimizare care dă rezultate foarte bune în practică: se încearcă întâi apelarea procedurii de cuplare fără a mai reseta vectorul  $U$ . Funcția de cuplare este foarte asemănătoare cu o parcurgere *DF*; astfel, acest algoritm poate fi implementat și într-o manieră nerecursivă, folosind o coadă ca în parcurgerea *BF*.

Nodurile din submulțimea  $V_1$  sunt numerotate de la 1 la  $n1$ , iar cele din mulțimea  $V_2$ , cu numerele de la 1 la  $n2$ .

```

1  int cupleaza(int nod)
2  {
3      int i;
4      if (U[nod]) return 0;
5      U[nod] = 1;
6      for (i = 1; i <= N2; i++)
7          if (!G[nod][i]) continue;
8          if (!dr[i] || cupleaza(dr[i]))
9              {st[nod] = i;
10               dr[i] = nod;
11               return 1;
12              }
13      }
14      return 0;
15  }
16  void cuplaj()
17  {int i;
18   for (i = 1; i <= N1; i++)
19   {
20       if (st[i]) continue;
21       if (cupleaza(i)) nr++;
22       else {
23           memset(U, 0, sizeof(U));
24           if (cupleaza(i)) nr++;
25       }
26   }
27  }
```

## Aplicații

Arhiva Educațională:

[Floyd-Warshall/Roy-Floyd](#)

[Sortare Topologică](#)

[BFS](#)

[DFS](#)

[Cuplaj Maxim în Graf Bipartit](#)  
[Păduri de Mulțimi Disjuncte](#)  
[Heapuri](#)  
[Arbore Parțial de Cost Minim](#)  
[Componente Tare Conex](#)  
[Componente Biconexe](#)  
[Algoritmul lui Bellman-Ford](#)

*Suplimentar:*

[Police](#)  
[APM](#)  
[Tree](#)  
[Masina](#)  
[Tygyn](#)  
[Cangrena](#)  
[Restore Array](#)  
[Disconnect](#)  
[Connect and Disconnect](#)  
[Clepsidra](#)  
[Amici](#)  
[Dungeon](#)  
[Mine](#)  
[Episodul3](#)  
[Awesome Arrowland Adventure](#)  
[Catun](#)  
[No Prime Sum](#) (*Cuplaj Maxim în Graf Bipartit*)