Elemente de teoria grafurilor. Modalități de memorare. Parcurgerea în adâncime și parcurgerea în lățime

Grafurile pot fi aplicate cu succes pentru a modela probleme specifice analizei circuitelor electrice, găsirea celui mai scurt drum, analiza planificării proiectelor, critica textelor literare, rețele sociale, aplicații din domeniul chimiei și economiei.

I. Terminologie

Definiţii: Se numeşte **graf neorientat** o pereche ordonată de mulţimi G=(X, U), unde: X = {x_1, x_2, x_3, …, x_n}este o mulţime finită şi nevidă de elemente numite **noduri** sau **vârfuri**, iar

U o mulţime finită de perechi neordonate de forma (x_i, x_j), unde $i \neq j$ şi x_i, $x_j \in X$, numite **muchii**. O muchie uneşte două noduri.

X = **mulţimea nodurilor** sau **vârfurilor**, iar U = **mulţimea muchiilor** grafului G.

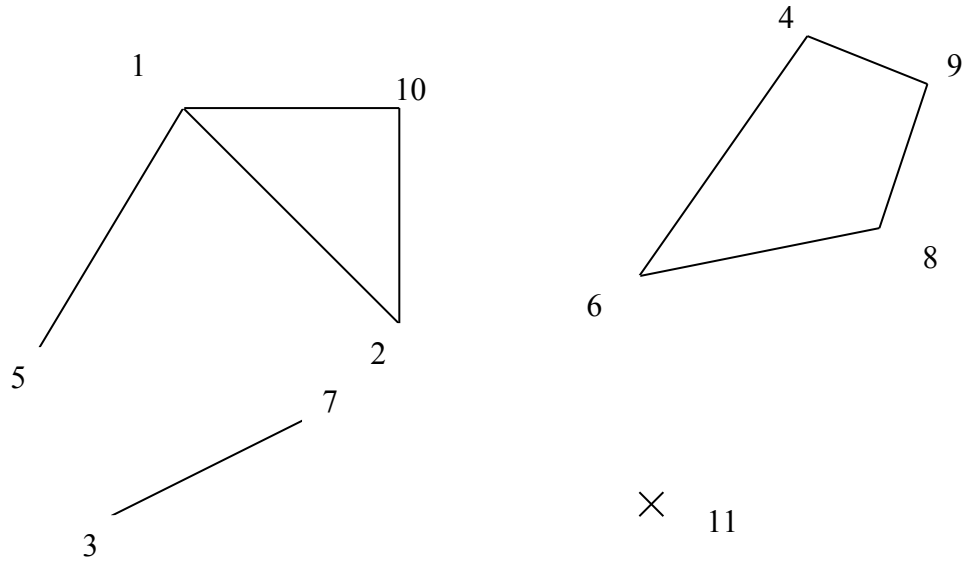
Un **vecin** al unui vârf este orice vârf care este adiacent lui în graf.

O muchie $u \in U$ este deci o submulţime {x, y} de vârfuri distincte din X şi se notează u = [x, y]. x şi y sunt **adiacente** în G, iar u şi x sunt **incidente** la fel ca u şi y. x şi y se numesc **extremităţile muchiei** u.

Dacă u_1 şi u_2 sunt 2 muchii care au o extremitate comună, ele se numesc **incidente**.

Mulţimea muchiilor are proprietatea de simetrie, deoarece $[x, y] \in U$ dacă şi numai dacă $[y,x] \in U$.

Exemplu: Fie G = (X, U), a.î. X = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}, iar
U = {[1, 5], [3, 7], [4, 6], [9, 8], [10, 2], [1, 2], [9, 4], [1, 10], [6, 8]}.



Definiţii: Gradul unui vârf x este numărul muchiilor incidente cu x. Se notează cu d (x) (d = degree).

Un vârf care are gradul 0 (deci pentru care nu există vreo muchie incidentă cu el) se numeşte **vârf izolat**. Un vârf care are gradul 1 (deci care este incident cu o singură muchie) se numeşte **vârf terminal**.

Exemplu: Pentru graful G definit anterior, d (11) = 0, deci vârful 11 este izolat.

Fie un graf neorientat cu **n** noduri şi **m** muchii. Dacă notăm cu d_1, d_2, d_3, …, d_n gradele celor n noduri, atunci avem relaţia:

$$d_1 + d_2 + d_3 + \ldots + d_n = 2m$$

Definiţii: Se numeşte **lanţ** $L = [x_0, x_1, \ldots, x_n]$ o succesiune de vârfuri cu proprietatea că oricare două vârfuri consecutive sunt adiacente.

Vârfurile x_0 şi x_n se numesc **extremităţile lanţului**. Numărul **n** se numeşte **lungimea lanţului** şi este numărul de muchii din care este format.

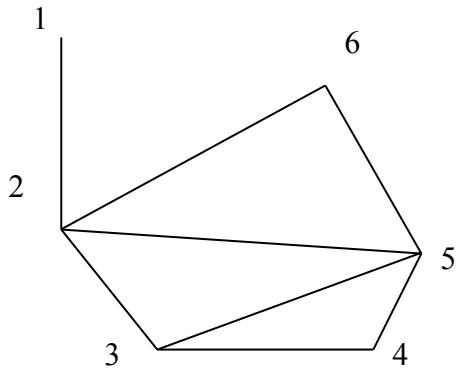
Lanţul care conţine numai vârfuri distincte, două câte două, este **lanţ elementar**.

Lanţul care conţine numai muchii distincte este **lanţ simplu**. Dacă muchiile unui lanţ nu sunt distincte se numeşte **lanţ compus**.

O matrice pătratică de ordin n se numeşte **matricea lanţurilor**, dacă:

$$l_{i,j} = \begin{cases} 1, \text{ dacă există lanţ de la i la j} \\ 0, \text{ în caz contrar} \end{cases}$$

Exemplu: Fie G = (X, U), a.î. X = {1, 2, 3, 4, 5, 6}, iar
U = {[1, 2], [2, 3], [3, 4], [3, 5], [4, 5], [5, 6]}.



Lanţul 2, 3, 5, 6 este simplu şi elementar de lungime 3.
Lanţul 5, 3, 4, 5, 6 este simplu, dar nu este elementar de lungime 4.
Lanţul 5, 3, 4, 5, 3, 2 este compus şi nu este elementar de lungime 5.

Definiţii: Un lanţ pentru care $x_0 = x_n$ (primul nod coincide cu ultimul) se numeşte **ciclu**. Dacă toate vârfurile unui ciclu, cu excepţia primului şi ultimului, sunt distincte, ciclul se numeşte **elementar**.

Lungimea unui ciclu nu poate fi 2.

Un ciclu se numeşte **par** dacă lungimea sa este pară şi **impar** în caz contrar.

Exemplu: În graful din figura anterioară lanţul 3, 4, 5, 3 este un ciclu elementar impar. Lanţul 2, 5, 3, 4, 5, 6, 2 este un ciclu par, dar nu este elementar.

II. Tipuri de grafuri

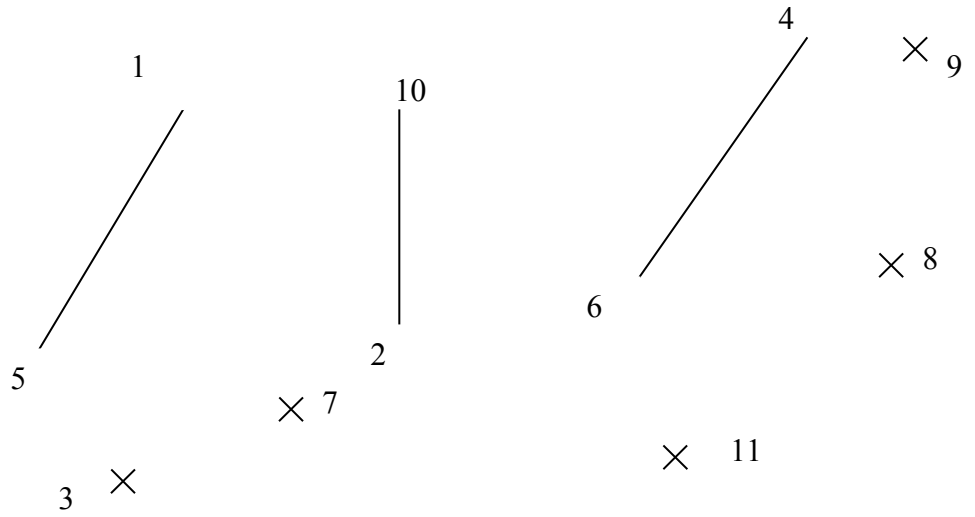
Dacă U = ∅ atunci graful G = (X, U) se numeşte **graf nul**, şi reprezentarea lui în plan se reduce la puncte izolate.

Definiţie: Un **graf parţial** al grafului G = (X, U) este un graf G_1 = (X, V) a.î. V⊆U, adică G_1 are aceeaşi mulţime de vârfuri ca G iar mulţimea de muchii V este chiar U sau o submulţime a acesteia.

Un graf parţial al unui graf se obţine păstrând aceeaşi mulţime de vârfuri şi eliminând o parte din muchii. Graful parţial G_1 este indus de mulţimea de muchii V.

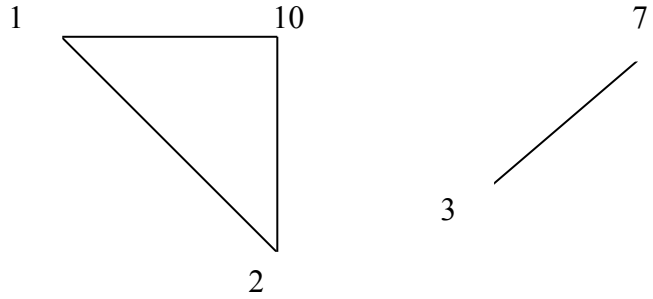
Un graf neorientat cu **n** noduri are 2^m grafuri parţiale şi anume numărul de submulţimi ale mulţimii muchiilor {1, 2, …, *m*},

Exemplu: Pentru graful G definit anterior G_1 = (X, V), unde
V = {[1, 5], [4, 6], [10, 2]}, este un graf parţial.



Definiţie: Un **subgraf** al grafului G = (X, U) este un graf H = (Y, V) a.î. Y⊆X, iar V conţine toate muchiile din U care au ambele extremităţi în Y. Subgraful H este indus sau generat de mulţimea de vârfuri Y.

Exemplu: Pentru graful G definit anterior, dacă Y = {1, 2, 3, 7, 10} şi
V = {[1, 2], [1, 10], [2, 10], [3, 7]}, atunci H = (Y, V) este un subgraf al grafului G.

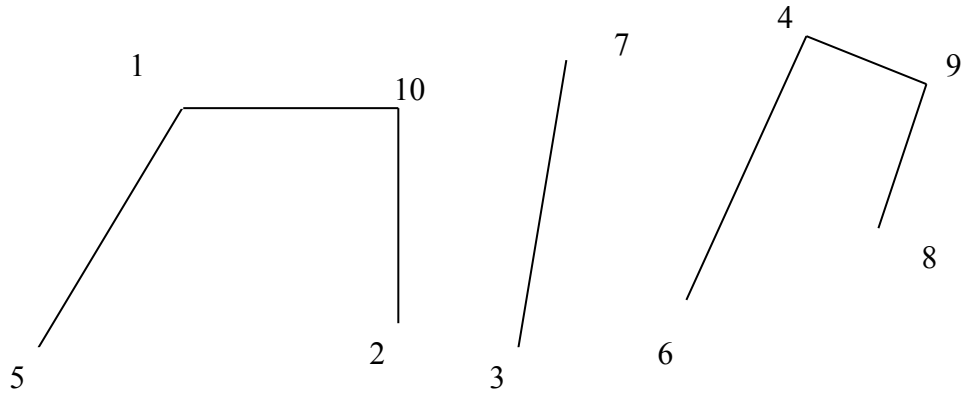


Un subgraf al unui graf se obţine eliminând o parte din vârfuri şi toate muchiile incidente cu acestea.

Un graf neorientat cu **n** noduri are $2^n - 1$ subgrafuri şi anume numărul de submulţimi ale mulţimii muchiilor {1, 2, …, *n*}, fără mulţimea vidă deoarece mulţimea nodurilor este nevidă.

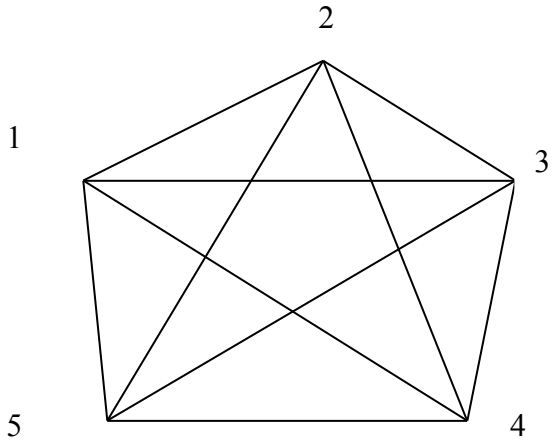
Definiţii: Un graf fără cilcuri se numeşte **graf aciclic**. Un graf aciclic şi conex este un **arbore**. În cazul grafului aciclic fiecare componentă conexă este un arbore. Un graf neorientat aciclic, care s-ar putea să nu fie conex, se numeşte **pădure**.

Exemplu: Graful de mai jos este aciclic şi formează o pădure. Cele 3 componente conexe sunt arbori.



Definiţie: Se numeşte **graf complet** cu n vârfuri un graf cu proprietatea că oricare două noduri distincte sunt adiacente. Un **graf complet** cu n vârfuri se notează K_n.

Exemplu: Graful definit mai jos este complet şi se notează cu K_5.



Într-un graf complet, gradul oricărui nod este **n - 1**, pentru că din/în fiecare nod pleacă/sosesc **n - 1** muchii.

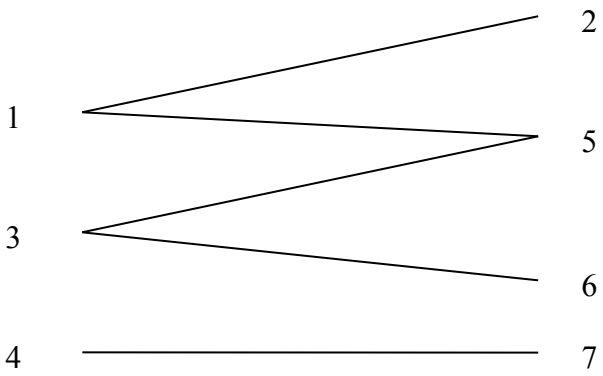
Într-un graf complet, avem relaţia: **m = n(n - 1)/2 =** C_n^2 (numărul de submulţimi cu 2 elemente ale mulţimii celor **n** noduri), unde **m** este numărul de muchii, iar **n** numărul de noduri.

Avem $2^{n(n-1)/2}$ grafuri neorientate cu **n** noduri.

Definiţie: Un graf G = (X, U) se numeşte **bipartit** dacă există 2 mulţimi nevide A şi B a.î. X = A ∪ B, A ∩ B = ∅ şi orice muchie u a lui G are o extremitate în A iar cealaltă în B. Mulţimile A şi B formează o partiţie a lui X.

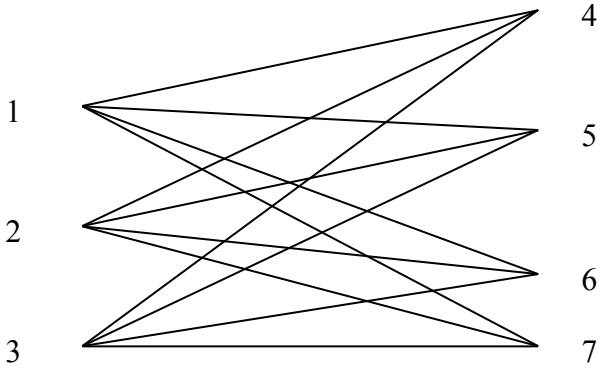
Un graf este **bipartit** dacă şi numai dacă nu conţine cicluri de lungime impară.

Exemplu: A = {1, 3, 4} şi B = { 2, 5, 6, 7}.



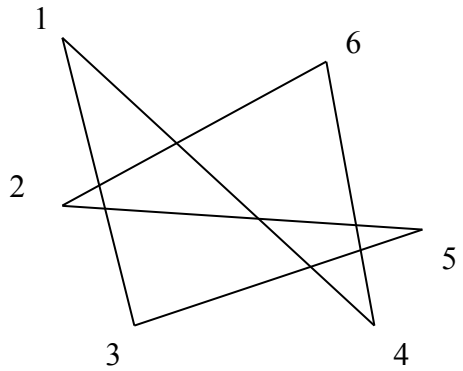
Definiţie: Un **graf bipartit** se numeşte **complet** dacă pentru $\forall x \in A$ şi $\forall y \in B$, $\exists$ în G muchia [x, y]. Notaţia este $K_{p,q}$ unde p şi q sunt elementele mulţimii A şi respectiv ale mulţimii B.

Exemplu: $K_{3,4}$ este:



Definiţie: Un graf în care toate nodurile au acelaşi grad se numeşte **graf regulat**.

Exemplu: Graful definit mai jos este regulat.



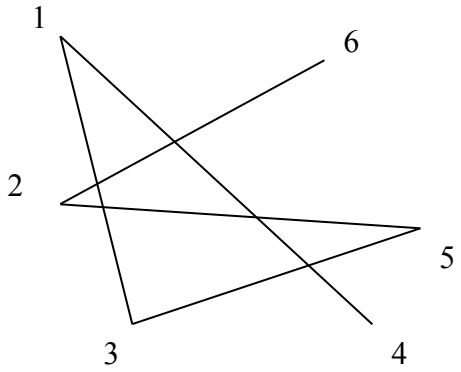
Definiţie: Un graf se numeşte **graf conex** dacă pentru oricare două vârfuri x şi y diferite ale sale, există un lanţ care le leagă, adică x este extremitatea iniţială şi y este extremitatea finală.

Un graf cu un singur nod este, prin definiţie, conex.

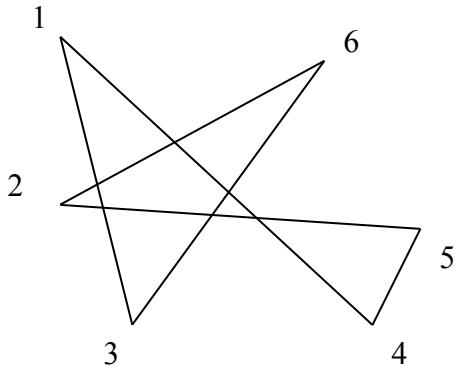
Se numeşte **componentă conexă** a unui graf G = (X,U) un subgraf H = (Y, V), conex, al lui G care are prorpietatea că nu există nici un lanţ în G care să lege un vârf din Y cu un vârf din X – Y.

Un graf este **conex** dacă admite o singură componetă conexă.

Exemplu: Graful definit mai jos are o singură componentă conexă şi în consecinţă este conex.

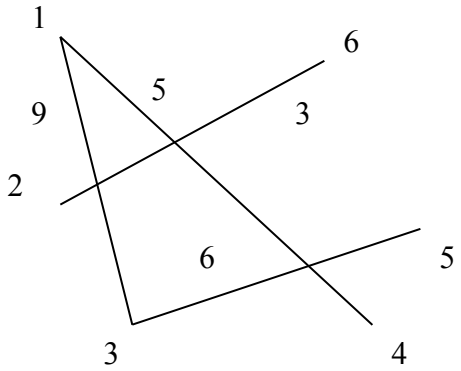


Definiţie: Un graf este **biconex** dacâ este conex şi pentru orice vârf eliminat subgraful generat îşi păstrează proprietatea de conexitate.
Exemplu: Graful definit mai jos este biconex.



Definiţii: Un **multigraf** seamănă cu un graf neorientat, însă poate avea atât muchii multiple între vârfuri cât şi autobucle. Un **hipergraf** seamănă cu un graf neorientat, dar fiecare **hipermuchie**, în loc de a conecta două vârfuri, conectează o submulţime arbitrară de vârfuri.

Definiţie: Costul unui graf este o funcţie definită pe mulţimea muchiilor grafului cu valori în mulţimea numerelor reale. Un graf notat cu o astfel de funcţie este notat sub forma: **G =** (X, U, c).
Exemplu: **G =** (X, U, c). este graful:



Definiţie: Se numeşte **ciclu hamiltonian** un ciclu elementar care trece prin toate vârfurile grafului. Un graf care admite un ciclu hamiltonian se numeşte **graf hamiltonian**.

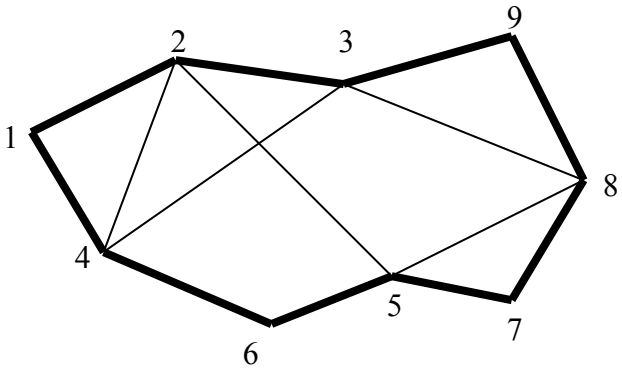
Fie G = (X,U) un graf neorientat şi un lanţ elementar care trece prin toate nodurile grafului [x_0, x_1, …, x_n]. Dacă $d(x_1) + d(x_n) \geq n$ atunci graful este hamiltonian.

Fie G = (X,U) un graf neorientat cu **n** noduri. Dacă pentru orice pereche de noduri neadiacente $x_i \neq x_j$, avem relaţia $d(x_i) + d(x_j) \geq n$ atunci graful este hamiltonian.

Dacă G = (X,U) este un graf neorientat cu **n** noduri şi gradul oricărui vârf este mai mare sau egal n/2, atunci G este hamiltonian.

Un graf G este hamiltonian dacă are n≥3 vârfuri şi gradul oricărui vârf verifică inegalitatea: d(x)≥n/2.

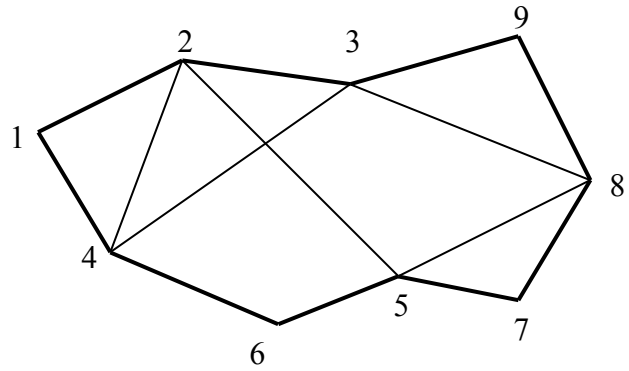
Exemplu: 1, 2, 3, 9, 8, 7, 5, 6, 4, 1 este ciclu hamiltonian.



Definiţie: Un lanţ al unui graf care conţine fiecare muchie o dată şi numai o dată se numeşte **lanţ eulerian**. Dacă $x_0 = x_n$ şi lanţul este eulerian atunci, ciclul respectiv se numeşte **ciclu eulerian**. Un graf care conţine un ciclu eulerian se numeşte **graf eulerian**.

Dacă este eulerian nu înseamnă că nu are vârfuri izolate.

Exemplu: 1, 2, 4, 6, 5, 7, 8, 3, 9, 8, 5, 2, 3, 4, 1 este ciclu eulerian.



Un graf G = (X,U), fără vârfuri izolate, este **eulerian** dacă şi numai dacă este conex şi gradele tuturor vârfurilor sale sunt numere pare.

III. METODE DE REPREZENTARE

1. Matrice de adiacenţă

O matrice pătratică şi simetrică de ordin n, cu:

$$a_{i,j} = \begin{cases} 1, \text{ pentru } [i, j] \in U \\ 0, \text{ pentru } [i, j] \notin U \end{cases}$$

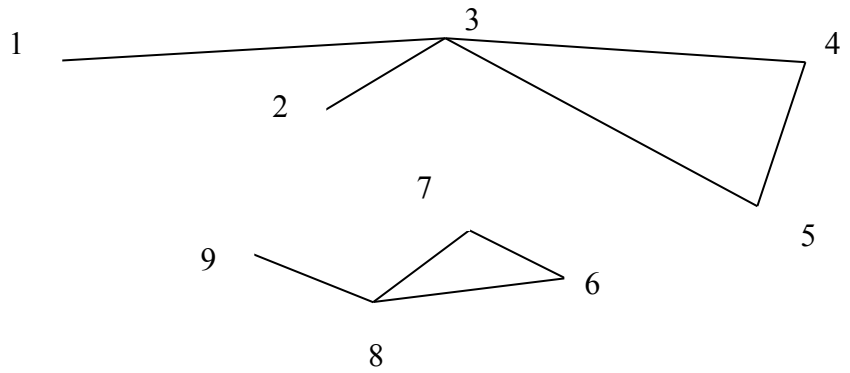
Elementele de pe diagonala principală reţin 0 şi anume: $a_{i,i} = 0$, $\forall i \in \{1, 2, \ldots, n\}$

Suma elementelor de pe linia **i** şi respectiv suma elementelor de pe coloana **j** au ca rezultat gradul nodului **i** respectiv **j**.

Suma tuturor elementelor matricei de adiacenţă este suma gradelor tuturor nodurilor , adică dublul numărului de muchii (2**m**).

Exemplu: Fie graful G = (X, U), de mai jos:

at



unci matricea lui de adiacenţă este:

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

2. Liste de adiacenţă definită dinamic

Pentru fiecare nod i, o listă liniară simplu înlănţuită va reţine toate nodurile j pentru care [i, j]$\in$ U, adică lista succesorilor.

Fiecare listă simplu înlănţuită conţine nodurile în ordinea inversă introducerii lor.

Exemplu: Pentru graful definit la reprezentarea prin matrice de adiacenţă avem:

1 → 3
2 → 3
3 → 1 → 2 → 4 → 5
4 → 5 → 3
5 → 4 → 3
6 → 8 → 7
7 → 8 → 6
8 → 9 → 6 → 7
9 → 8

Avem structurile următoare:

```
struct Nod  {
                int info;
                Nod * next; }
Nod* L[50];
```

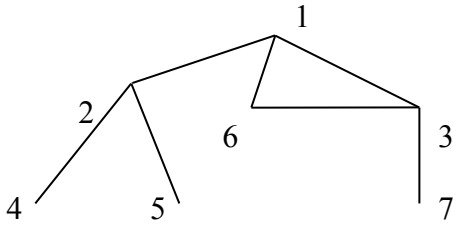
3. **Matricea costurilor**
4. **Lista muchiilor**

IV. METODE DE PARCURGERE

Parcurgerea în lătime sau pe nivele (BF – breath first)

Se face începând de la un anumit nod, parcurgem apoi toţi descendenţii săi, apoi toţi descendenţii nodurilor parcuse la pasul anterior, s.a.m.d. Un nod este parcurs o singură dată. Există mai multe soluţii ale unei astfel de parcurgeri pentru că ordinea de parcurgere a desendenţilor unui nod nu este impusă, ea depinde şi de modul în care a fost memorat graful.

Exemplu: Pentru graful din figura de mai jos:



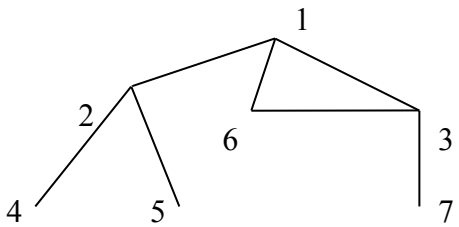
ordinea de vizitare a nodurilor este: 1 3 6 2 7 5 4 sau 1 2 6 3 4 5 7.

Parcurgerea **BF** se efectuează prin utilizarea structurii numită coadă, având grijă ca un nod să fie vizitat o singură dată. Coada va fi alocată static prin utilizarea unui vector. Graful poate fi memorat prin matrice de adiacenţă, respectiv liste de adiacenţă.

Parcurgerea în adâncime (DF – depth first)

Se face începând de la un anumit nod, parcurgem apoi primul dintre descendenţii săi, neparcurşi încă s.a.m.d. Un nod este parcurs o singură dată şi aici există mai multe soluţii de parcurgere.

Exemplu: Pentru graful din figura de mai jos:



ordinea de vizitare a nodurilor pornind de la nodul 1 este: 1 2 4 5 3 6 7 sau 1 3 7 6 2 5 4.

Parcurgerea **DF** se efectuează recursiv graful putând fi memorat atât prin matrice de adiacenţă cât şi liste de adiacenţă.

V. APLICAȚII CU GRAFURI NEORIENTATE

V.1. Cunoscând şi aplicând cele de mai sus realizaţi un program cu grafuri neorientate în care se citeşte dintr-un fişier pe câte o linie separat numărul de noduri şi perechile de noduri între care exista o muchie (**listă de muchii**); construiţi şi afisaţi pe ecran matricea de adiacenţă.

V.2. Se consideră un graf neorientat exprimat prin intermediul unei matrici de adiacenţă (analog cu cel din problema anterioară).

a) Scrieţi un program care calculează **gradul fiecărui nod** din graf, sortează descrescator şi afişează nodurile şi gradele aferente.
b) Scrieţi un program care determină **câte noduri sunt izolate în graf** şi afişează toate aceste noduri.
c) Să se construiască şi să se afişeze **pentru fiecare nod din graf lista de adiacenţă definită dinamic**.
d) Să se verifice dacă o secvenţă de noduri citită de la tastatură (încheiată cu -1) formează **un lanţ în graf**.
e) Să se verifice dacă o secvenţă de noduri citită de la tastatură (încheiată cu -1) formează **un ciclu în graf**.
f) Scrieţi un program care să verifice dacă un graf este **conex**. Dacă graful nu este conex să se determine (și afișeze) componentele conexe ale grafului.
g) Scrieţi un program care să verifice dacă un graf este **eulerian**[1].
h) Scrieţi un program care să verifice dacă un graf este **regulat**.
i) Scrieţi un program care să verifice dacă un graf este **complet**.
j) Realizați un o funcție care verifică dacă într-un graf neorientat, memorat cu ajutorul matricei de adiacență există un **ciclu elementar de lungime *k* (**unde *k* este citit de la tastatură**)**. Graful conține *n* noduri (n≤25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce 1 în caz afirmativ și 0, în caz contrar.
k) Scrieţi un program care determină şi afişează **cel mai lung lanţ** şi **cel mai lung ciclu din graf**.
l) Scrieţi un program care apelează două funcții, și anume cele prin care se **parcurge graful în lăţime** şi apoi în **adâncime**.
m) Scrieţi un program care să verifice dacă un graf este **hamiltonian**.

Se folosește matricea de numere întregi `int A[100][100]` și *numărul de noduri* `n`, ambele declarate global.
De asemenea, sunt necesare trei variabile ajutătoare declarate global: `int vizitat[100],ordine[100],nr_comp;` care rețin informații despre vizitarea sau nu a unui nod din graf, despre ordinea de parcurgere a nodurilor
Se vor implementa următoarele funcții având prototipurile:

[1] Un graf G = (X,U), fără vârfuri izolate, este eulerian dacă și numai dacă este conex și gradele tuturor vârfurilor sale sunt numere pare.
//**Observație**: graful poate fi eulerian și dacă conține noduri izolate. Dacă un graf este compus 1) dintr-o componentă conexă care este un subgraf conex cu gradele tuturor vârfurilor componente numere pare și 2) din noduri izolate, atunci graful este **eulerian**.

void **citesteMA**(char[100],int[100][100],int &); - citirea din fișierul primit ca parametru a listei de muchii și transformarea în matrice de adiacență și obținerea numărului de noduri al grafului.
void **BFMatrix_subgraf**(int nod); - parcurgerea în lățime a unui subgraf (a unei componente conexe care include nodul primit ca parametru)
void **BFMatrix**(); - parcurgerea pe nivele a grafului
void **DFmatrix_subgraf**(int); - parcurgerea în adâncime a unui subgraf (a unei componente conexe care include nodul primit ca parametru)
void **DFmatrix**(int); - parcurgerea în adâncime a unui graf. Folosește funcția **DFmatrix_subgraf** însă trece la primul nod nevizitat în cazul în care graful nu e conex.
int **conex**(); - verifică dacă un graf e conex. Practic dacă la parcurgerea lui în adâncime cu ajutorul **DFmatrix_subgraf** nu au mai rămas noduri nevizitate.
int **comp_conexe**(); - reprezintă o extindere a funcției **Dfmatrix** care afișează numărul componentei conexe și nodurile aparținătoare acesteia.
void **grad_toate**(int MA[100][100],int &noduri); - determină gradele tuturor nodurilor din graf.
int **grad_nod**(int nod); - returnează gradul nodului primit ca parametru.
void **grad_max**(int MA[100][100],int &noduri); - calculează gradul maxim al nodurilor din graf și afișează toate nodurile care au grad maxim.
int **Euler**(); - determină dacă un graf este eulerian și afișează un mesaj corespunzător.

```
void BFMatrix_subgraf(int nod){
int k,root,prim, ultim;

      vizitat[nod]=1;
      prim=1;
      ultim=1;
      ordine[ultim++]=nod;
      cout<<nod<<" ";
      while (prim != ultim)
      {
            root = ordine[prim];
            for (k = 1; k <= n; k++)
                  if (A[ordine[prim]][k] && (vizitat[k]==0))
                  {
                        vizitat[k] = 1;
                        ordine[ultim++] = k;
                        cout << k << " ";
                  }
            prim++;
      }
}
```

```
void DFmatrix_subgraf(int nod){
     int k;

     cout<<nod<<" ";
     vizitat[nod]=1;
     for(k=1;k<=n;k++)
          if((A[nod][k]==1)&&(vizitat[k]==0))
               DFmatrix_subgraf(k);
}
```

ARBORI PARTIALI DE COST MINIM

V.3. Realizaţi un program cu grafuri neorientate care implementează **algoritmul lui Prim**. Se citeşte dintr-un fişier pe câte o linie separat numărul de noduri şi perechile de noduri între care exista o muchie precum şi costul aferent. Algoritmul lui Prim de complexitate patratica ($O(n^2)$) este un algoritm din teoria grafurilor care găsește **arborele parțial de cost minim** al unui **graf conex ponderat**.

a) Să se reprezentinte graful prin matricea costurilor
b) Să se determine pornind de la un nod arbitrar submulțimea muchiilor care formează un arbore care include toate vârfurile și al cărui cost este minim.

Algoritmul lui Prim se bazează pe tehnica **Greedy**[2] ("*a lacomului*" – alegerea soluției optime local) și urmărește să realizeze un arbore (graful să nu aibă cicluri) incrementând mărimea arborelui, pornind de la un nod, până când sunt incluse toate nodurile. Inițial se pleacă de la un arbore format dintr-un singur vârf (ales arbitrar). La fiecare pas se selectează muchia cu costul minim pornind din respectivul vârf și care să ducă într-un nod neselectat anterior. Trebuie ca mulţimea E_{nou} a muchiilor selectate şi mulţimea V_{nou} a vârfurilor unite de acestea să formeze un arbore.

- **Intrare:** Un graf conex ponderat cu mulțimea nodurilor V și a muchiilor E.
- **Ieșire**: V_{nou} și E_{nou} descriu arborele parțial de cost minim

(1) **Initializare:** V_{nou} = {x}, unde x este un nod arbitrar (punct de plecare) din V, E_{nou}= {}
(2) **repetă până când** V_{nou}=V
    (3) Alege muchia (u,v) din E de cost minim astfel încât u este în V_{nou} și v nu e (dacă există mai multe astfel de muchii, se alege arbitrar)
    (4) Se adaugă v la V_{nou}, (u,v) la E_{nou}

Pentru rezolvarea problemei folosim matricea ponderilor (sau costurilor) în Forma 1 (∞), și anume: o **matrice pătratică şi simetrică de ordin n**, cu:

[2] Algoritmii ***greedy*** sunt aplicaţi în rezolvarea problemelor de optimizare; sunt compuşi dintr-o secvenţă de paşi, la fiecare pas existând mai multe alegeri posibile; pot fi priviţi ca o particularizare a tehnicii ***backtracking*** în care se renunţă la mecanismul de întoarcere. **Algoritmii greedy aleg la fiecare moment de timp soluţia ce pare a fi cea mai bună la momentul respectiv: o alegere optimă, făcută local, cu speranţa că va conduce la un optim global. Cu toate acestea nu întotdeauna conduc la soluţia optimă**. Timpul de calcul este polinomial (de cele mai multe ori este necesară o sortare descrescătoare a datelor de intrare în funcţie de prioritatea sau ponderea acestora la soluţia globală - optimă). Tehnica greedy este destul de puternică şi se aplică cu succes unui spectru larg de probleme de optimizare [Cor90].

$$\text{cost}([i, j]) = \begin{cases} c_{i,j}, \text{ pentru } [i, j] \in U \text{ şi } i \neq j \\ \infty, \text{ pentru } [i, j] \notin U \text{ şi } i \neq j \\ 0, \text{ pentru } i = j \end{cases}$$

Secvența de cod de mai jos exemplifică modul de determinare a vârfului care va fi selectat datorită costului minim a muchiei care pleacă dintr-un nod anterior selectat.

```
/*****************************************************************/
      min = INT_MAX;
      for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
               if(selectat[i]==1 && selectat[j]==0 && min>c[i][j])
                  {
                        min=c[i][j];
                        linie=i; coloana=j;
                  }
      cout<<"   "<<linie<<" "<<coloana<<" \t\t -> \t"<<min<<endl;
      selectat[coloana]=1;
/*****************************************************************/
```

Fişierele de intrare pot fi:

```
5
1 2 2
1 4 2
1 5 3
2 3 7
2 4 1
2 5 5
3 4 4
4 5 4
```

```
5
1 2 2
1 3 1
1 4 1
2 5 3
3 4 1
4 5 4
```

V.4. Realizaţi un program cu grafuri neorientate care implementează **algoritmul lui Kruskal**. Pe prima linie a fişierului de intrare se află numărul de noduri *n* şi numărul de muchii *m* din graf. Se citesc apoi de pe câte o linie separat perechile de noduri între care exista o muchie precum și costul aferent. Algoritmul lui Kruskal de complexitate logaritmică cu numărul de muchii ($O(m \cdot \log_2 m)$) este un algoritm din teoria grafurilor care găsește **arborele parțial de cost minim** al unui **graf conex ponderat** (găsește submulţimea muchiilor care formează un arbore, incluzând toate vârfurile şi care este minimizat din punct de vedere al costului).

a) Să se reprezentinte graful prin matricea costurilor
b) Să se determine submulțimea muchiilor care formează un arbore care include toate vârfurile și al cărui cost este minim.

La fel ca şi în cazul algoritmului Prim și algoritmul lui Kruskal se bazează pe tehnica **Greedy**. Se pornește de la lista de muchii cu costul asociat lor și se ordonează această listă în funcţie de cost. Se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Diferența dintre

cei doi algoritmi constă în faptul că algoritmul lui Kruskal poate crea câţiva arbori mici până când arborele este complet, pe când algoritmul lui Prim dezvoltă un arbore parţial pentru a deveni arborele căutat. De asemenea, se recomandă folosirea algoritmului Prim (care este mai rapid) când graful are foarte multe muchii (graf dens), mult mai multe muchii decât noduri. Algoritmul Kruskal se comportă mai bine (rapid) în cazul grafurilor "aerisite" (*sparse*) deoarece foloseşte structuri de date mai simple.

- **Intrare:** Un graf conex ponderat cu mulţimea nodurilor V şi a muchiilor E.
- **Ieşire**: E_{nou} descrie arborele parţial de cost minim

(1) Arborele căutat va conţine *n*-1 muchii (şi evident nu conţine cicluri)
(2) Se creează o mulţime de arbori, unde fiecare vârf din graf este un arbore separate
(3) **Se creează o mulţime E_{nou} care conţine toate muchiile din graf**
(4) **Sortează** lista de muchii crescător după cost
(5) **Cât timp** E_{nou} este nevidă şi nu s-au ales cele n-1 muchii **execută**:
    (6) Elimină o muchie de cost minim din E_{nou} cu extremităţile nemarcate sau marcate diferit
    (7) Incrementează la costul arborelui costul respectivei muchii
    (8) Dacă acea muchie unifică doi arbori distincţi, atunci adaugă muchia, unificand cei doi arbori într-unul singur
(9) La final, va rezulta arborele parţial de cost minim al grafului.

Mai jos se prezintă cele mai importante variabile, structuri de date precum şi secvenţa de cod care exemplifică modul de generare a arborelui parţial de cost minim.

```
int viz[20],m,n,c=0;
//vectorul viz spune carui arbore ii corespund muchiile, in final toate
muchiile trebuind sa apartina unuia singur

struct {
     int x,y,cost;
}v[20];
/* se considera un graf v format din maximum 20 de muchii pentru
usurinta procesului de debug-ing; fiecare muchie are doua extremitati
(varfuri) si un cost */

void kruskal()
{
 int i,j,k;
 i=1;
 for(k=1;k<=n-1;k++)
                    /*algoritmul se incheie in momentul in care am un
                    arbore care leaga toate nodurile; acest arbore
                    trebuie sa aiba n-1 muchii*/
  {
    while(viz[v[i].x]==viz[v[i].y] && viz[v[i].x]!=0)
          i++; /*muchiile avand capete in x (v[i].x) si y (v[i].y) sunt
                deja incluse intr-un arbore viz[v[i].x]!=0 ➔ trec la
                urmatoarea muchie in ordinea crescatoare a costurilor*/
          c+=v[i].cost; /*cand ajung pe prima muchie neasignata unui
                        arbore ii adaug costul la costul global
```

```
      out<<v[i].x<<"  "<<v[i].y<<'\n';      //afisez muchia selectata

        if(viz[v[i].x]+viz[v[i].y]==0)  /*Daca este adevarata conditia
                                        inseamna ca muchia nu este asignata
                                        vreunui arbore (nici unul dintre
                                        cele doua varfuri ale muchiei)*/
            viz[v[i].x]=viz[v[i].y]=v[i].x; /*si atunci se asigneaza
                                          arborelui aferent unui capat al
                                          muchiei*/
    else
       if(viz[v[i].x]*viz[v[i].y]==0)/*in acest moment cu siguranta unul
                                       din capetele muchiei a fost vizitat
                                       iar celalalt nu*/
         viz[v[i].x]=viz[v[i].y]=viz[v[i].x]+viz[v[i].y]; /*si ca urmare
                                                          se asigneaza
                                                          arborelui de
                                                          care apartine
                                                          celalalt nod
                                                          (cel care este
                                                          deja intr-un
                                                          arbore)*/
      else /*in acest moment cu siguranta au fost vizitate ambele capete
           ale muchiei si daca apartin la arbori diferiti*/
      {    /*atunci adaug muchia, unificand cei doi arbori intr-unul
           singur*/
        for(j=1;j<=n;j++) /*daca gasesc doua noduri diferite care apartin
                          aceluiasi arbore atunci adaug la arbore si
                          nodul din celalalt capat al muchiei*/
            if((viz[j]==viz[v[i].x]) && (j!=v[i].x)) /*si impreuna cu el
                                                     toate nodurile din
                                                     arborele care
                                                     cuprinde acest
                                                     nod*/
                viz[j]=viz[v[i].y];
      viz[v[i].x]=viz[v[i].y]; /*adaug cel de-al doilea varf al muchiei
                               sa apartina aceluiasi arbore*/
      }
    i++;
  }
  out<<c;
  in.close();
  out.close();
}
```

Fişierul de intrare poate fi:

```
6 8
1 2 3
2 3 1
3 1 2
1 4 4
4 6 2
6 1 5
4 5 1
5 1 6
```

Fişierul de ieşire arată astfel:
lista de muchii care compun arborele şi costul

```
2  3
4  5
4  6
3  1
1  4
10
```

BIBLIOGRAFIE

[Ior05] **Iorga V., Opincaru C., Stratan C., Chiriță A.** – „*Structuri de date şi algoritmi. Aplicații în C++ folosind STL*”, Editura Polirom, 2005.

[Tud08] **Tudor S.** – „*Informatica. Curs pentru clasele a IX-a şi a X-a*”, Editura L&S INFO-MAT, 2008.

[Hut06] **Huţanu V., Tudor S.** – „*Manual de Informatică intensiv, clasa a XI-a, Varianta C++*”, Editura L&S Soft, Bucureşti 2006 .

[Lic06] **Lica D., Paşoi M.** – „*Fundamentele programării – Culegere de probleme pentru clasa a XI-a*”, Editura L&S Soft, Bucureşti, 2006.

[Iva98] **Ivaşc C., Prună M.** – „*Bazele informaticii (Grafuri şi elemente de combinatorică), Manual pentru clasa a X-a*”, Editura Petrion, 1998.

[Iva98] **Ivaşc C., Prună M., Mateescu E.** – „*Bazele informaticii (Grafuri şi elemente de combinatorică), caiet de laborator*”, Editura Petrion, 1998.

[Cad98] **Cadar C., Stroe M.** – „*Culegere de probleme şi programe*”, Editura Petrion, 1998.

[Cor98] **Cormen T.H., Leiserson C.E., Rivest R.R.** – „*Introducere în algoritmi*”, Editura Agora, 1998.

[Flo10] **Florea D.** – „*Metode vizuale de e-Learning centrate pe algoritmi şi tehnici de programare*”, Lucrare ştiinţifică pentru conferirea gradului didactic I, Universitatea “Lucian Blaga” Sibiu, 2010.

[Cor90] Cormen, T. H., Leiserson, C. E., Rivest, R. L. – *Introduction to Algorithms*. McGraw-Hill, New York, 1990.