Grafuri

1. Obiectivele lucrării

În această lucrare se vor studia aspecte legate de grafuri.

2. Breviar teoretic

Grafurile sunt structuri de date complexe, utilizate in modelarea problemelor legate de activități întâlnite în realitatea de zi cu zi. Un **graf** G este o structura ce consta din două mulțimi: G = (V,E), unde V este o mulțime finita de ***noduri (vârfuri)*** ale grafului iar $E \subseteq V^2$ este o relație pe mulțimea V. Mulțimea E se mai numeşte si **mulțimea de muchii** (**arce** sau **laturi**) ale grafului. Daca fiecare element al mulțimii E este o pereche ordonata <v,w> graful se zice **orientat** sau **digraf**. Se spune ca vârful w este **adiacent** lui v daca si numai daca $(v,w) \in E$. v este **nodul sursa** iar w este **nodul destinație** al arcului <v,w>.

Un graf este **neorientat** daca oricare ar fi o muchie $(u,v) \in E$ se considera ca atât u este adiacent lui v cat si v este adiacent lui u.

Un graf **neorientat** G = (V,E) se zice **conex** daca pentru orice $u,v \in V$ exista o cale de la u la v.

Un graf **orientat** cu proprietatea de mai sus se zice **tare conex.**

Intr-un graf orientat **muchiile** se mai numesc **arce**.

Un graf **neorientat** G = (V,E) se zice **complet** daca si numai daca exista cate o muchie intre oricare doua vârfuri ale lui G. Un astfel de graf va avea in total n(n-1)/2 muchii.

In cazul unui graf **neorientat** prin **gradul** unui nod se înțelege numărul de muchii care sunt conectate la nodul respectiv.

Un nod izolat este un nod al cărui grad este 0.

Un **arbore** este un graf conex şi fără cicluri.

Modalități de reprezentare a grafurilor:

1. Reprezentarea prin matrice de adiacenta

Pentru un graf ce are N vârfuri, matricea de adiacenta este o matrice pătrată ce are N linii si N coloane.

$$A[i][j] = \begin{cases} 1 \ daca \ exista \ muchie \ intre \ i \ si \ j \\ 0 \ daca \ i, j \notin E \end{cases}$$

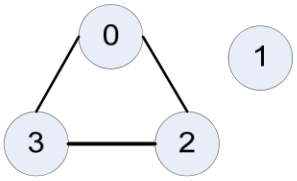
$$A[i][i] = 0$$

In unele aplicații, arcele pot avea ataşate **ponderi** (sau **costuri**) asociate. In acest caz, matricea de adiacenta devine **matrice de costuri**, spre ex. A[u][v] va ave ca valoare costul asociat arcului <u,v> iar daca nu exista arc intre u si v, A[u][v] va capătă o valoare foarte mica sau foarte mare in funcție de problema care trebuie rezolvata.

Spre exemplu intr-o problema de determinare a drumului de cost minim intre doua noduri ale unui graf, matricea de costuri asociata grafului va avea valori c[u,v] = ∞ (unde in loc de infinit se poate folosi o valoare foarte mare) daca intre vârfurile u si v nu exista legătură directă.

Un graf ale cărui arce au ataşate costuri se numeşte si **graf etichetat**.

Observație: Pentru un graf neorientat matricea de adiacenta A este simetrica fata de diagonala principala, deoarece legătură intre două noduri este bidirecțională. Acest gen de reprezentare este adecvata in cazul problemelor in care se doreşte sa se afle rapid daca exista un arc intre doua vârfuri oarecare.



Matricea de adiacenta:

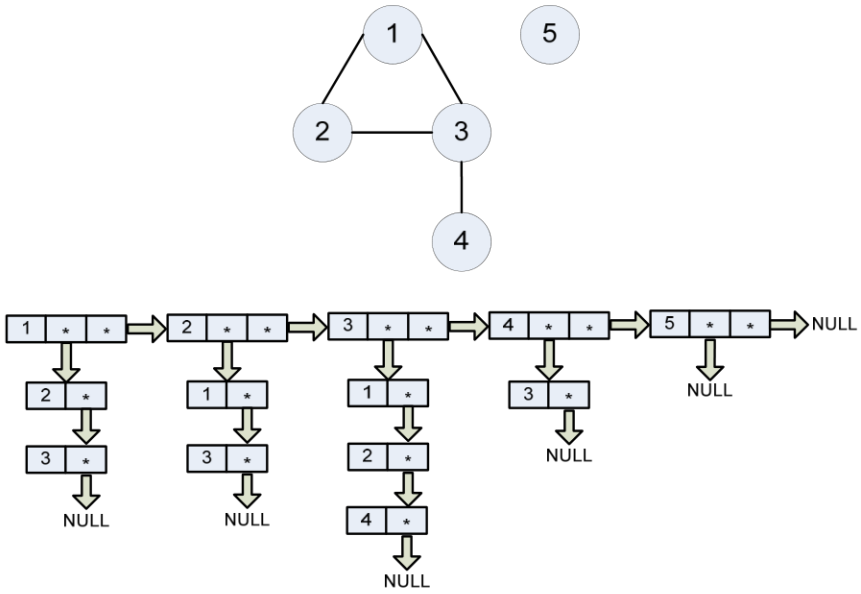
$$A[n][m] = \begin{pmatrix} 0\,0\,1\,1 \\ 0\,0\,0\,0 \\ 1\,0\,0\,1 \\ 1\,0\,1\,0 \end{pmatrix}$$

Observație:

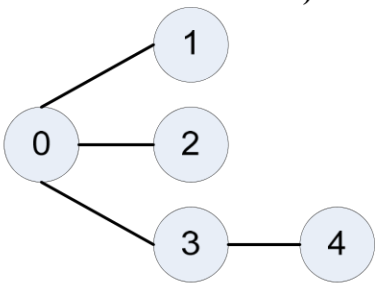
Pentru un graf orientat (digraf) matricea A nu mai este simetrică.

2. Reprezentarea prin liste de adiacenta

Aceasta reprezentare foloseşte o lista ce cuprinde toate vârfurile grafului, fiecare nod din lista are o legătură (pointer) către o alta lista ce conține vecinii nodului respectiv.



3. **Reprezentarea tabelara (graful se reprezintă sub forma unui vector de muchii)**



{(0,1), (0,2), (0,3), (3,4)}

O muchie se reprezintă ca o structura cu 2 câmpuri. Pentru ca reprezentarea să fie completă trebuie să citim:

- Numărul total de noduri ale grafului
- Numărul de muchii
- Vectorul de muchii.

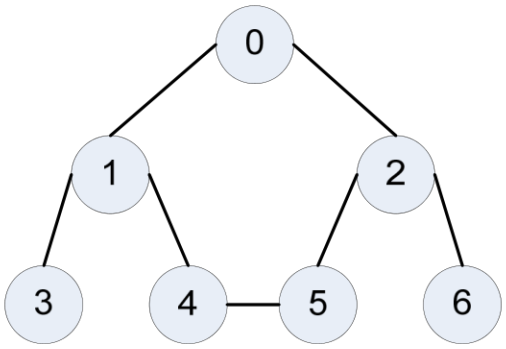
Modalități de parcurgere a grafurilor:

Exista 2 modalități fundamentale de a traversa vârfurile unui graf si a prelucra informația din aceste vârfuri:

1. **Parcurgerea în adâncime (Depth-First Traversal) -** conform acestui procedeu se porneşte dintr-un nod v, se prelucrează informaţia din acest nod (spre ex. se vizitează nodul), iar apoi se parcurg recursiv toate nodurile adiacente lui v. Este posibil ca graful sa conţină cicluri; in acest caz execuţia algoritmului de parcurgere prezentat anterior ar putea sa conţină o bucla infinita de instrucţiuni. Pentru a evita acest lucru trebuie marcat faptul ca algoritmul a prelucrat un anumit nod al grafului. In acest scop se foloseşte un tablou vizitat[] cu elemente booleene. Fiecare vârf al grafului are asociat un element al vectorului care va fi marcat (i se va atribui valoarea adevărat) in momentul prelucrării informaţiei asociate acelui vârf. In pseudocod, algoritmul de parcurgere in adâncime a unui graf este:

```
procedura DFT(v) este
    vizitat[v]←adevarat
    pentru *w adiacent cu v repeta
    daca not vizitat[w] atunci
    DFT(w)
   ¤
  ¤
sfarsit
```

Iniţial elementele tabloului vizitat[] au valoarea fals. Daca graful este neorientat si neconex (sau orientat si nu este tare conex) aceasta metoda ar putea sa nu parcurgă toate nodurile grafului. De aceea după terminarea execuţiei apelului DFT se va caută un nod nemarcat si se va începe din acel nod o noua parcurgere in adâncime. Aceasta strategie garantează ca fiecare muchie va fi parcursa o singura data. Evident, procesul de determinare a unui nod nevizitat încă va începe cu testarea vârfului ce urmează celui de unde a început ultima parcurgere DF (se foloseşte o structura de tip **stiva**).

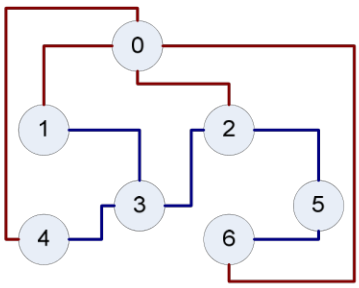


Ordinea de vizitare:

Nod start = 0: 0, 1, 3, 4, 5, 2, 6.
Nod start = 4: 4, 1, 0, 2, 5, 6, 3.

2. **Parcurgerea in lățime (Breadth-First Traversal) –** acest algoritm de parcurgere vizitează mai întâi nodul de start v, apoi toți vecinii (nevizitați încă) ai primului vecin al lui v, etc. procedura este asemănătoare cu cea prezentata la parcurgerea in lățime a arborilor. Parcurgerea BF foloseşte ca structura auxiliara o **coada**. Ordinea de vizitare este memorata in vectorul coada si se afişează la final, nu pe parcurs ca la metoda DF.

```
procedura BFT(v) este
    InitQ(q),
    AddQ(q,v),
    Cat timp *q nu este vida repeta
      RemQ(q,v)
      vizitat[v]←adevarat
      pentru *toti vecinii w ai lui v
repeta
        daca not vizitat[w] atunci
           AddQ(q,w)
          ¤
        ¤
      ¤
sfarsit
```



Ordinea de vizitare:

Nod start = 0: 0, 1, 2, 4, 6, 3, 5.

3. Probleme rezolvate

Se vor edita şi apoi executa programele descrise în continuare.

1. Se citeşte de la tastatura matricea de adiacenta a unui graf (înainte se vor citi numărul de vârfuri).
   - Se citeşte numărul unui nod al grafului. Sa se calculeze si afişeze gradul acestuia.
   - Sa se calculeze si afişeze care este nodul de grad maxim.

Sursa programului:

```
#include <stdio.h>
#include <conio.h>
#include<iostream.h>
void main()
{
      int N; //nr. varfuri
      cout<<"N="; cin>>N;
      int a[30][30]; //dimens. acoperitoare
      //graf neorientat - matrice de adiacenta
//simetrica
      //deci o citim doar pe jumatate
      int i,j;
      for(i=0;i<N;i++)
            a[i][i]=0;
      for(i=0;i<N-1;i++)
            for(j=i+1;j<N;j++)
            {
                  cout<<"a["<<i<<"]["<<j<<"]=";
                  cin>>a[i][j];
                  a[j][i]=a[i][j];
      }
```

```
	//citire nod
	int x;
	cout<<"x="; cin>>x;
	//aflare grad nod x: suma valorilor de 1 din
//linia x
	int grad=0;
	for(i=0;i<N;i++)
		grad = grad+a[x][i];
	cout<<"grad nod "<<x<<"="<<grad;

	//cautare nod de grad maxim din graf
	int gradMax=-1; int nodMax;
	//parcurgem liniile matricii
	for(i=0;i<N;i++)
	{
		//calcul grad nod i
		grad=0;
		for(j=0;j<N;j++)
			grad = grad+a[i][j];
		if(grad>gradMax)
		{
			gradMax = grad;
			nodMax = i;
		}
	}
	cout<<endl<<"Nr. nod = "<<nodMax;
	getch();
}
```

2. Se citeşte de la tastatura un graf sub forma tabelara: N – numărul de noduri, M – numărul de muchii si V_m[] – vectorul de muchii. Sa se realizeze si afişeze matricea de adiacenta a grafului.

Sursa programului:

```
#include <stdio.h>
#include <conio.h>
#include<iostream.h>

#define N 5 //nr. noduri
#define M 4 //nr. muchii

typedef struct
{
	int n1;
	int n2;
```

```
}muchie;

void main()
{
      clrscr();
      muchie Vm[M];
      int i,j,k;
      for(i=0;i<M;i++)
      {
        cout<<"Nod stanga pentru muchia "<<i<<"=";
            cin>>Vm[i].n1;
            cout<<"Nod dreapta pentru muchia
"<<i<<"=";
            cin>>Vm[i].n2;
       }
       int a[N][N];
       for(i=0;i<N;i++)
            a[i][i]=0;
       for(i=0;i<N-1;i++)
            for(j=i+1;j<N;j++)
            {
                  int exista=0;
                  for(k=0;k<M;k++)

      if(((Vm[k].n1==i)&&(Vm[k].n2==j))||((Vm[k].n1=
=j)&&

(Vm[k].n2==i)))
                        {
                              exista=1;
                              break;
                        }
                        if(exista)
                        {
                              a[i][j]=1;
                              a[j][i]=1;
                        }
                        else
                        {
                              a[i][j]=0;
                              a[j][i]=0;
                        }
            }
       //afisare tabelara a matricii de adiacenta
```

```
        for(i=0;i<N;i++)
        {
             for(j=0;j<N;j++)
                   cout<<a[i][j]<<"  ";
             cout<<endl;
        }
        getch();
}
```

3. Se citeşte de la tastatura matricea de adiacenta a unui graf (numărul de vârfuri sunt date cu #define). Sa se parcurgă in lățime (**Breadth-First Traversal**) graful construit.

Sursa programului:

```
//parcurgere BF
#include<stdio.h>
#include<conio.h>
#include<iostream.h>

#define N 5 //nr. noduri

void main()
{
      clrscr();
      int i,j;
      int a[N][N];
      for(i=0;i<N;i++)
            a[i][i]=0;
      for(i=0;i<N-1;i++)
            for(j=i+1;j<N;j++)
            {
                  cout<<"a["<<i<<"]["<<j<<"]=";
                  cin>>a[i][j];
                  a[j][i]=a[i][j];
            }
      int c[N]; //vectorul coada
      int esteVizitat[N];
      for(i=0;i<N;i++)
            esteVizitat[i]=0; //nici un nod vizitat
      int prim=0; int ultim=0;
      int nodStart;
      cout<<"Nod start = ";
      cin>>nodStart;
      c[0]=nodStart;
      esteVizitat[nodStart]=1;
```

```
	while(prim<=ultim)
	{
		int nodCrt = c[prim]; //nodul din capul
//cozii
		for(i=0;i<N;i++)

	if((a[i][nodCrt]==1)&&(esteVizitat[i]==0))
			{
				//il adaugam in coada
				ultim++;
				c[ultim]=i;
				esteVizitat[i]=1;
			}
		prim++;
	}
	//afisare ordine BF
	for(i=0;i<N;i++)
		cout<<c[i]<<" ";
	getch();
}
```

4. Se citeşte de la tastatura matricea de adiacenta a unui graf (numărul de vârfuri sunt date cu #define). Să se parcurgă in adâncime (**Depth-First Traversal**) graful construit.

Sursa programului:

```
//parcurgere DF
#include<stdio.h>
#include<conio.h>
#include<iostream.h>

//var globale pentru stiva
int st[100];
int iV; //index vf stiva
int nMax; //nr maxim de elemente
//pentru parcurgere DF
int N; //nr. noduri
int a[30][30]; //matricea de adiacenta
int esteVizitat[30];

void init(int dim)
{
	nMax=dim;
	iV=-1;
}
```

```
void push(int nr)
{
      //presupunem ca are loc
      iV++;
      st[iV]=nr;
}

int pop()
{
      int x = st[iV];
      iV--;
      return x;
}

int esteVida()
{
      if(iV==-1)
            return 1;
      return 0;
}

int calculVecin(int nodCrt)
{
      //parcurgem toate nodurile
      int i;
      for(i=0;i<N;i++)

      if((a[i][nodCrt]==1)&&(esteVizitat[i]==0))
                  return i;
      return -1;
}

void main()
{
      clrscr();
      init(50); //initializare stiva
      //citire graf
      cout<<"Nr. de noduri = ";
      cin>>N;
      //citire matrice de adiacenta
      int i,j;
      for(i=0;i<N;i++)
            a[i][i]=0;
```

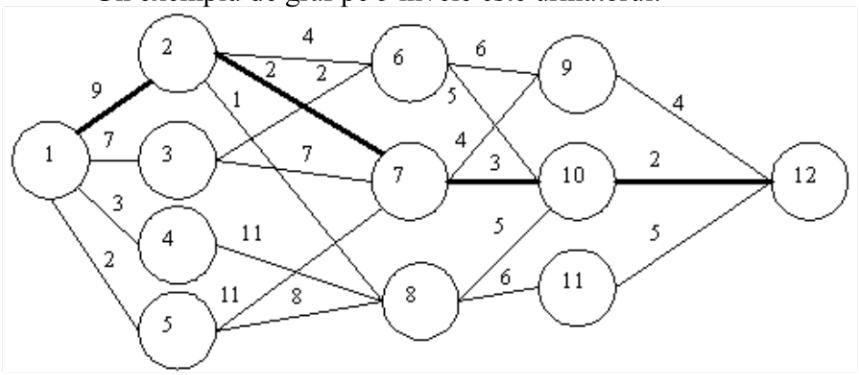
```
	for(i=0;i<N-1;i++)
		for(j=i+1;j<N;j++)
		{
			cout<<"a["<<i<<"]["<<j<<"]=";
			cin>>a[i][j];
			a[j][i]=a[i][j];
		}

	int nodStart;
	cout<<"Nod start = ";
	cin>>nodStart;
	//la inceput nici un nod nu este vizitat
	for(i=0;i<N;i++)
		esteVizitat[i]=0; //nici un nod vizitat
	//algoritm DF
	push(nodStart);
	cout<<nodStart<<" ";
	esteVizitat[nodStart]=1;
	int nodCrt = nodStart;
	for(;;)
	{
		int nrVecin = calculVecin(nodCrt);
		if(nrVecin!=-1)
		{
			nodCrt = nrVecin;
			push(nodCrt);
			cout<<nodCrt<<" ";
			esteVizitat[nodCrt]=1;
		}
		else //nu am gasit vecini
		{
			if(esteVida())
				break; //iesire din for(;;)
			else
				nodCrt = pop();
		}
	}
	getch();
}
```

5. Aplicație pentru grafuri pe mai multe nivele. Numim graf pe mai multe nivele, un graf orientat G = (X,A) unde X este mulțimea vârfurilor şi A mulțimea arcelor, în care:

- Vârfurile grafului pot fi împărțite în submulțimi disjuncte V1, V2, ..., Vn, n >= 2 numite nivele. Fiecare nivel conține numai vârfuri la care se poate ajunge din vârfurile nivelului anterior şi/sau din care se poate pleca spre vârfurile nivelului următor.
- Vârfurile primului nivel (din mulțimea V1) nu au ascendenți, vârfurile ultimului nivel nu au descendenți.
- Vârfurile aceluiaşi nivel nu pot fi unite între ele.
- Fiecărui arc (i,j) i se ataşează un cost C(i,j) > 0.

Problema pe care ne-o punem este de a **determina un drum de cost minim de la un vârf inițial s∈V1 la un vârf final t∈Vn.**

Un exemplu de graf pe 5 nivele este următorul:



Drumul de cost minim de la s la t este indicat de linia îngroşată. Costul unui drum fiind suma costurilor arcelor din care este format, rezultă că principiul optimalității este satisfăcut, deci se poate aplica metoda programării dinamice.

O formulare a programării dinamice este obținută prin observarea că orice drum de la s la t este rezultatul a n-2 decizii. Decizia cu numărul k implică determinarea unui nod al nivelului k+1 (din mulțimea Vk+1, 1 <= k <= n-2) care să fie în drumul optim. Pentru nivelul k luăm toate drumurile de cost minim de la vârfurile v∈Vk la t.

Fie cost (k,v) reprezentând costul drumului minim de la v la t. Folosind *accesul inainte* vom obține:

Cost (k,v) = min {C(v,w) + cost(k+1,w | w din Vk+1, (v,w)∈A}

unde cost(n-1,v) = C(v,t) dacă (v,t)∈A şi cost (n-1,v) = +∞ în caz contrar.

Calculul decurge astfel:

Pentru nivelul 4:

cost(4,9) = 4; cost(4,10) = 2; cost(4,11) = 5

Pentru nivelul 3:

cost(3,6) = min{6+cost(4,9),5+cost(4,10)} = 7

cost(3,7) = min{4+cost(4,9),3+cost(4,10)} = 5

cost(3,8) = min{5+cost(4,10),6+cost(4,11)}= 7

Pentru nivelul 2:

cost(2,2) = min{4+cost(3,6),2+cost(3,7),1+cost(3,8)}=7

cost(2,3) = min{2+cost(3,6),7+cost(3,7)}=9

cost(2,4) = min{11+cost(3,8)}=18

cost(2,5) = min{11+cost(3,7),8+cost(3,8)}=15

Pentru nivelul 1:

cost(1,1) = min{9 + cost(2, 2), 7 + cost(2,3), 3+ cost(2,4), 2+ cost(2,5) }=16

Astfel un cost minim al drumului de la s la t este 16. Acest drum poate fi determinat uşor dacă vom înregistra decizia făcută în cazul fiecărui nivel şi vârf. Luând D(k,v) reprezentând valoarea lui w astfel încât:

cost(k,v) = min{C(v,w)+cost(k+1,w) | w∈Vk+1, (v,w)∈A}

Vom avea:

D(4,9) = D(4,10) = D(4,11) = 12

D(3,6) = 10; D(3,7) = 10; D(3,8) = 10;

D(2,2) = 7; D(2,3) = 6; D(2,4) = 8; D(2,5) = 8;

D(1,1) = 2;

Fie drumul s, v1, v2, …, vn-1, t de cost minim. Este uşor de văzut că:

v2 = D(1,1) = 2; v3 = D(2,D(1,1)) = 7; v4 = D(3,D(2,(D(1,1))) = D(3,7) = 10.

Înainte de a scrie programul în C de rezolvare, impunem o ordine a vârfurilor grafului. Dacă graful are *nv* vârfuri, vom folosi pentru indexare valorile 1 ,…, *nv*. Astfel, vârful s va avea indicele 1 şi indicii pentru vârfurile nivelului k sunt mai mari decât indicii vârfurilor nivelului k-1. Ca rezultat al acestei scheme de indexare, cost şi D pot fi calculate în ordinea n-1, n-2, .., 1. Primul indice din cost şi D care identifică doar numărul nivelului este omis în program. Programul care urmează foloseşte pentru graful G reprezentarea care

asociază fiecărui vârf I lista Ld[i] a descendenților săi şi lista C[i] a costurilor ataşate arcelor respective.

Sursa programului:

```
#include <stdio.h>
#include <conio.h>
#include<iostream.h>

const float infinit = 1.e10;

void graf(int& nv, int**& Ld,float**& C)
{
      cout<<" Nr. De varfuri in arbore:" ;
      cin>>nv;
      Ld=new int*[nv-1]-1;
      C=new float*[nv-1]-1;
      for(int i=1;i<nv;i++)
      {
            int nd;
            cout<<"Varful "<<i<<". Nr descendenti:";
            cin>>nd;
            Ld[i]=new int[nd+1];
            C[i]=new float[nd]-1;
            Ld[i][0]=nd;   //numarul   descendentilor
//varfului i
            for(int j=1;j<=nd;j++)
            {
                  cout<<"arcul    "    <<i<<"--->    ";
                  cin>>Ld[i][j];
                  cout<<"cost ";
                  cin>>C[i][j];
            }
      }
}

float  DrumMin(int  nv,int**  Ld,float**  C,int*&  x,
int& n)
{
      int j;
      float* cost=new float[nv]-1; //alocare vector
//cost[1],…,cost[nv]
      int*   D=new   int[nv-1]-1;   //alocare   vector
//D[1],…,D[nv-1]
      cout<<" Numar nivele: ";
      cin>>n;
```

```
      x=new int[n]-1; // vector solutie x[1],…,x[n]
      cost[nv]=0;
      for(int  k=nv-1;k>=1;k--)
      {
            cost[k]=infinit;
            for(j=1;j<=Ld[k][0]; j++)
            {
                  int r=Ld[k][j]; //descendent
                  float L=C[k][j]+cost[r];
                  if  (L<cost[k])  {  cost[k]=L;D[k]=r;
}
            }
      }
      x[1]=1;
      x[n]=nv;
      for(j=2;j<n;j++)
            x[j]=D[x[j-1]];
      return cost[1];
 }

void main()
{
      int nv,n;
      int** Ld;
      float** C;
      int* x;
      clrscr();
      graf(nv,Ld,C);
      float cost=DrumMin(nv,Ld,C,x,n);
      cout<<"Drum minim:";
      for(int i=1;i<=n;i++)
      cout<<x[i]<<" , ";
      cout<<"\b cost:"<<cost<<endl;
}
```

4. Probleme propuse

1. *Se citeşte de la tastatura matricea de adiacenta a unui graf (înainte se vor citi numărul de vârfuri). Să se calculeze şi afişeze numărul de noduri izolate.*
2. *Dându-se un graf reprezentat tabelar, să se calculeze şi să se afişeze numărul de vârfuri izolate.*