Grafuri - memorare, probleme de început

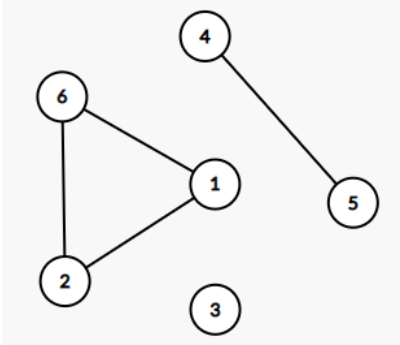
Pe scurt, un graf este o rețea de noduri unite între ele prin muchii. Putem avea grafuri neorientate (orice muchie se consideră în ambele sensuri) sau orientate. Putem avea costuri asociate muchiilor.
Pentru început analizăm câteva modalități uzuale de a stoca un graf neorientat fără costuri pe muchii.
Convenim ca în paragrafele ce urmează să notăm cu `n` numărul de noduri și cu `m` numărul de muchii.

Matricea de adiacență

Este o matrice pătratică `nXn`, binară:

`a[i][j] =`	`1`, dacă există muchie între nodul `i` și nodul `j`
	`0`, dacă nu există muchie între nodul `i` și nodul `j`

Exemplu:



Pentru graful neorientat desenat mai sus matricea de adiacență este:

0	1	0	0	0	1
1	0	0	0	0	1
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
1	1	0	0	0	0

Proprietăți
Elementele de pe diagonala principală sunt toate `0` deoarece se consideră că nu avem muchie de la un nod la el însuși.

Matricea este simetrică față de diagonala principală deoarece o muchie între `i` și `j` este considerată în ambele sensuri. Astfel, pentru a obține toate muchiile, este suficient să traversăm doar unul dintre triunghiurile delimitate de diagonala principală (cum s-a hașurat și în figură).
Gradul unui nod se obține numărând valorile nenule de pe linia sau de pe coloana sa iar vecinii săi se obțin ca fiind indicii coloanelor unde sunt elemente nenule pe linia nodului (sau indicii liniilor unde sunt elemente nenule pe coloana nodului).
Observăm că în cazul unui număr mic de muchii majoritatea elementelor matricei sunt nule. Acesta este un prim semn că pentru memorarea în practică a unui graf matricea de adiacență nu este cea mai fericită soluție.
Algoritmii de parcurgere a unui graf (în lățime sau în adâncime) necesită obținerea vecinilor fiecăruia dintre cele `n` noduri. Astfel va trebui să parcurgem toate liniile matricei, deci obținem timp de calcul de ordin n^2.
Din aceste motive în general se folosesc alte modalități de a stoca un graf. Totuși, matricea de adiacență și forme derivate ale ei (cum ar fi matricea costurilor - adică la o muchie nu notăm în matrice `1` ci costul ei) sunt utile nu numai din punct de vedere didactic ci si pentru câțiva algoritmi foarte importanțe precum Roy-Walshall (de determinare dacă există drum între oricare două noduri) sau Roy-Floyd (de găsire a drumurilor minime între oricare două noduri).

Listele de vecini

Dacă graful este complet numărul său de muchii este de ordin n^2, deci exact câte elemente are și matricea de adiacență.
Sunt însă extrem de multe situații în practică unde nu avem de-a face cu graf complet. Iată două exemple

- Rețeaua legăturilor directe prin drumuri între orașele dintr-o țară. Orașele sunt nodurile și legăturile directe dintre ele sunt drumurile. Am putea să ne imaginăm că în România sunt în jur de `20000` de localități și fiecare dintre ele este legată prin drum direct cu doar câteva să zicem în medie `10`. Memorarea acestui graf folosind matricea de adiacență ar necesita o structură cu `200` milioane de componente (20000^2) dintre care maxim `400000` ar fi nenule. Evident că ar fi mare risipă de memorie (dacă nu chiar imposibilitatea de a aloca).
- Legătutile de prietenie de la o rețea de socializare. Cred că e evident că o rețea în care sunt câteva sute de milioane de useri nu poate fi stocată printr-o matrice de adiacență. Un lucru care întărește ideea este că numărul de regulă prieteni este limitat la rețelele cu mulți useri (tocmai din acest motiv).

În astfel de cazuri și în general în practică, un graf trebuie memorat cu liste de vecini. Astfel pentru fiecare nod se vor stoca într-o structură, doar vecinii săi, unul lângă altul.
În C/C++ se folosește de obicei structura vector din **stl**.

Prezentăm în continuare câteva elemente de limbaj specifice ei apoi vom exemplifica utilizarea în programe concrete.

Declarare:

```
vector <tip> v;
```

Din multe puncte de vedere care țin de sintaxă, declararea este echivalentă cu:
`tip v[dimensiune];`

Iată diferențele semnificative:

- Imediat după declararea unui vector din `stl` nu se rezervă memorie, aceasta urmează să se rezerve pe măsură ce se adaugă elemente cu metodele (funcțiile) specifice structurii;
- Funcția membru `size()` oferă numărul de componente existente în structură;
- Funcția membru `push_back()` permite adăugarea unui element la structură (având ca efect actualizarea dimensiunii).

Acestea sunt funcțiile necesare în majoritatea cazurilor dar vector admite o multitudine de funcții membru, inclusiv cele pentru ștergere.
Precizăm însă că funcția de adăugare la final are timp de calcul constant iar cea de ștergere are timp de calcul liniar.

- Vector permite accesul direct la elementele sale folosind operatorii paranreze drepte, `[]` ca la un tablou unidimensional obișnuit. Accesul rapid în timp constant se datorează faptului că elementele unui vector sunt stocate în locații vecine de memorie.

Iată un detaliu de implementare a structurii care asigură cele de mai sus. Există riscul ca la adăugarea unui element să nu mai fie locații disponibile așa încât să rămână componentele compact în memorie. Pentru asta, structrura are mereu alocate în memorie număr de componente putere de `2` (până la următoarea putere de `2` mai mare sau egală cu size-ul curent). Astfel, dacă la adăugare se trece cu size peste o putere de `2`, în acel moment se alocă deja elemente până la următoarea putere de `2` (deci doar la trecerile peste o putere de `2`).
Astfel riscul de mutare în memorie este de ordin logaritmic.

- Structura admite și o metodă care permite dimensionarea de la început la o anume valoare (dacă știm de la început câte elemente urmează să adăugăm). Această metodă se numește resize.

O simplă căutare pe google oferă locuri de unde se găsesc informații detaliate despre cele de mai sus dar și despre alte proprietăți.

De exemplu: https://www.cplusplus.com/reference/vector/vector/

Memoria necesară pentru stocarea unui astfel de graf este de ordin `n+m` (spunem așa pentru că și în situația în care numărul de muchii este neglijabil, trebuie să stocăm pentru fiecare nod măcar obiectul listei, fie el și cu size nul).
Iată costurile de timp în cazul memorării unui graf cu liste de vecini implememtate în modul descris:

- Determinarea gradului unui nod se face în timp constant (accesăm metoda size), ca și determinarea vecinilor unui nod (în lista nodului sunt chiar ei nu suntem obligați să facem oricum n pași ca la matricea de adiacență);
- Parcurgerea întregului graf cu metodele cunosciute se face cu timp de calul de ordin `n+m` (la fiecare nod parcurgem direct vecinii săi, dar numărul total de vecini este același cu numărul de muchii).

- Într-adevăr, dacă dorim să testăm doar dacă este muchie între două noduri anume, timpul de calcul este liniar (trebuie căutat un nod în lista celuilalt). În practică însă avem cel mai des de făcut parcurgeri deci ne trebuie toți vecinii și atunci oricum parcurgem toată lista. Totuși pentru situații când chiar dorim test repetat doar dacă sunt prezente muchii, în loc de vector putem folosi set (pentru lista vecinilor unui nod). Costul construirii unui set ca listă de vecini adaugă un factor de timp logaritmic, dar la accesul direct putem face căutare binară și în loc de timp de ordin liniar avem și aici timp de ordin logaritmic (o transformare cumva echivalentă este să sortăm vectorii de vecini după crearea lor).

Iată mai jos exemple de memorare a unui graf atât cu matrice de adiacență cât și cu liste de vecin.

Probleme rezolvate

Adiacența (pbinfo, #412)
Se dă lista muchiilor unui graf. Să se afișeze matricea de adiacență a sa.

Soluție
În fișierul de intrare se dau numărul de noduri, numărul de muchii și muchiile. La citirea fiecărei muchii marcăm în matricea de adiacență (pe care o vedem ca pe o matrice de frecvență) cu `1`. Ținem cont că graful este neorientat deci marcăm atât (`i,j`) cât și (`j,i`).

```
#include <fstream>
#define DIM 101
using namespace std;
int a[DIM][DIM];
int n, m, x, y, i, j;
ifstream fin ("adiacenta.in");
ofstream fout("adiacenta.out");
int main () {
    fin>>n>>m;
    for (i=1;i<=m;i++) {
        fin>>x>>y;
        a[x][y] = a[y][x] = 1;
    }
    for (i=1;i<=n;i++) {
        for (j=1;j<=n;j++) {
            fout<<a[i][j]<<" ";
        }
        fout<<"\n";
    }
}
```

Lista vecini (pbinfo #414)
Se dă lista muchiilor unui graf neorientat. Să se afișeze, pentru fiecare vârf al grafului, lista vecinilor săi.

Soluție

```
#include <fstream>
using namespace std;
int a[101][101], d[101];
int n, m, i, j;
int main () {
    ifstream fin ("listavecini.in");
    ofstream fout("listavecini.out");
    fin>>n;
    while (fin>>i>>j) {
        a[i][j] = 1;
        a[j][i] = 1;
    }
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (a[i][j] == 1)
                d[i]++;
    /// d[i] = numarul de vecini ai lui i (gradul lui i)
    for (i=1;i<=n;i++) {
        fout<<d[i]<<" ";
        for (j=1;j<=n;j++)
            if (a[i][j] == 1)
                fout<<j<<" ";
        fout<<"\n";
    }
}
```

Pentru problema de pe pbinfo la restricții se specifică faptul că în fișierul de intrare o muchie se poate repeta. Dar noi trebuie să considerăm acest lucru doar o dată. Așa că în matricea de adiacență am notat cu `1` la perechea de indici corespunzători unei muchii iar la altă apariție suprascriem tot cu `1`. Problema mai cere să afișăm la fiecare nod mai întâi gradul lui. Deoarece muchiile se pot repeta la intrare ar fi fost mai costisitor ca efort de implementare să facem acest lucru de la citire. Așa că am preferat să parcurgem matricea de adiacență ulterior momentului creării ei.

Graf complet (pbinfo #431).
Se dau mai multe grafuri neorientate, prin matricea de adiacență. Să se verifice despre fiecare dintre ele dacă este complet. Se va afișa `1` dacă graful este complet și `0` în caz contrar.

Soluție
Efectiv este suficient să verificăm că matricea de adiacență este completată doar cu valori `1`, cu excepția diagonalei principale. Dacă ne bazăm pe faptul că datele de intrare sunt

corecte, este suficient să testăm foar valorile din unul dintre triunghiurile delimitate de diagonale.

```
#include <fstream>
using namespace std;
ifstream fin("graf_complet.in");
ofstream fout("graf_complet.out");
int n, m, a[51][51], x, y, i, j;
int main(){
    fin>>n;
    for(int k=1;k<=n;k++){
        fin>>m;
        int ok = 1;
        for(i=1;i<=m;i++){
            for(j=1;j<=m;j++){
                fin>>x;
                if (i != j && x == 0){
                    ok = 0;
                }
                if (i == j && x == 1)
                    ok = 0;
            }
        }
        fout<< (ok == 1 ? "DA\n" : "NU\n");
    }
}
```

Gen graf (pbinfo #466)
Se dă un număr natural `n`. Construiți toate grafurile neorientate cu `n` vârfuri. Pentru fiecare se afișează matricea de adiacență.

Soluție
Trebuie să determinăm toate configurațiile posibile pentru matricea de adiacență. Întrucât ea este simetrică față de diagonala principală este suficient să ne ocupăm de generarea în toate modurile a triunghiului de sub această diagonală. Acolo sunt `n(n-1)/2` elemente și vor trebui generate toate șirurile de `0` și `1` de această lungime și apoi plasarea elementelor fiecărei astfel de soluții în matrice. Sunt $2^{\frac{n(n-1)}{2}}$ astfel de soluții așa cum știm că acesta este numărul de grafuri neorientate cu `n` noduri.
Pentru generare vom asocia vectorului de construit scrierea binară a fiecărui număr cuprins între `0` și $2^{\frac{n(n-1)}{2}} - 1$. Am folosit operații la nivel de bit.

Sursa

```
#include <fstream>
using namespace std;
int a[7][7], v[17], n, m, i, j, k, b, c;
int main () {
    ifstream fin ("gengraf.in");
    ofstream fout("gengraf.out");
    fin>>n;
    m = n*(n-1)/2;
    fout<<(1<<m)<<"\n";
    for (c=0;c < (1<<m); c++ ) {
        for (b = 0; b<m; b++)
            v[b] = ((c >> b) & 1);
        k = 0;
        for (i=2;i<=n;i++)
            for (j=1;j<i; j++) {
                a[i][j] = v[k++];
                a[j][i] = a[i][j];
            }
        for (i=1;i<=n;i++) {
            for (j=1;j<=n;j++)
                fout<<a[i][j]<<" ";
            fout<<"\n";
        }
        fout<<"\n";
    }
    return 0;
}
```

Verif Lanț (pbinfo #474)

Se dă lista muchiilor unui graf neorientat cu `n` noduri și mai multe șiruri de noduri din graf. Să se verifice despre fiecare șir dacă reprezintă un lanț. În caz afirmativ, să se precizeze dacă este elementar.

Soluție
Memorăm graful dat cu matrice de adiacență deoarece este necesar să testăm dacă există muchie între două noduri și astfel facem operația mai ușor.
Pentru a testa apoi dacă un șir este lanț este suficient să analizăm fiecare două valori aflate una lângă alta și să verificăm în matricea de adiacență valoarea de la cei doi indici (codul:`if ( a[ v[i-1] ][ v[i] ] == 0) ...`).
Pentru a verifica apoi dacă lanțul este elementar folosim un vector de frecvență `f` în care notăm pentru fiecare nod dacă a apărut sau nu pe lanț.

```
#include <fstream>
#define DIM 101
```

```
using namespace std;
int a[DIM][DIM], v[DIM], f[DIM];
int n, m, x, y, i, j, k, t;
ifstream fin ("veriflant.in");
ofstream fout("veriflant.out");
int main () {
    fin>>n>>m;
    for (i=1;i<=m;i++) {
        fin>>x>>y;
        a[x][y] = a[y][x] = 1;
    }
    fin>>k;
    while (k--) {
        fin>>t;
        for (i=1;i<=t;i++)
            fin>>v[i];
        int lant = 1;
        for (i=2;i<=t;i++)
            if ( a[ v[i-1] ][ v[i] ] == 0) {
                lant = 0;
                break;
            }
        if (lant == 0) {
            fout<<"NU\n";
            continue;
        }
        for (i=1;i<=n;i++)
            f[i] = 0;
        int elementar = 1;
        for (i=1;i<=t;i++) {
            f[ v[i] ]++;
            if (f[ v[i] ] == 2) {
                elementar = 0;
                break;
            }
        }
        if (elementar)
            fout<<"ELEMENTAR\n";
        else
            fout<<"NEELEMENTAR\n";
    }
}
```

Lanț maxim (pbinfo, #479)
Se dă lista muchiilor unui graf neorientat cu `n` noduri și două noduri `p` și `q`. Să se determine cel mai lung lanț elementar cu extremitățile `p` și `q`.

Soluție
Vom folosi backtracking pentru a genera toate lanțurile cu extremitatea inițială într-un nod ales, `p`.

De câte ori alegem în soluție nodul `q` verificăm dacă aceasta este soluția de lungime maximă. Ca detaliu de implementare, folosim vectorul de frecvență `f` în care avem notate nodurile care apar deja în prefix întrucât noi trebuie să generăm un lanț elementar.
Un alt detaliu de implementare care merită subliniat este că fixăm pe poziția `1` nodul `p` (marcându-l ca vizitat) și apoi începem generarea efectivă de la nivelul doi al stivei, câștigând astfel timp de executare.

```
#include <fstream>
using namespace std;
int n, m, z, y, p, q, maxim, i;
ifstream fin ("lantmaxim.in");
ofstream fout("lantmaxim.out");
int x[21], t[21], f[21];
int a[21][21];

void backtrack(int pas) {
    for (int i=1;i<=n;i++) {
        if (f[i] == 0 && a[i][x[pas-1]]) {
            x[pas] = i;
            f[i] = 1;
            if (i!=q)
                backtrack(pas+1);
            else {
                if (pas > maxim) {
                    maxim = pas;
                    for (int j=1;j<=pas;j++)
                        t[j] = x[j];
                }
            }
            f[i] = 0;
        }
    }
}

int main ()
{
    fin>>n>>m;
    for (i=1;i<=m;i++) {
        fin>>z>>y;
        a[z][y] = 1;
        a[y][z] = 1;
    }
    fin>>p>>q;
    x[1] = p;
    f[p] = 1;

    backtrack(2);
    for (int i=1;i<=maxim;i++)
        fout<<t[i]<<" ";
```

```
    return 0;
}
```

Bipartit (pbinfo, #471)
Se dă lista muchiilor unui graf neorientat cu n noduri, etichetate de la 1 la n, precum si o mulțime A de noduri ale grafului. Considerăm mulțimea B formată din nodurile care nu aparțin lui A. Să se verifice dacă graful este bipartit peste partiția formată din mulțimile A și B.

Soluție
Folosim un vector de frecvență în care marcăm cu `1` elementele care se află în mulțimea `A`. Cele care rămân marcate cu `0` vor face deci parte din mulțimea `B`.
Este suficient apoi ca pentru fiecare muchie să testăm dacă are o extremitate marcată cu o valoare și o extremitate marcată cu cealaltă valoare.
Observație: Pentru a testa dacă este bipartit un graf la care nu se dă înaine partiția, se folosește un algoritm de parcurgere a grafului în care la avansarea din nodul curent pe o muchie, nodul nou se marchează cu cealaltă valoare decât cel din care este accesat, iar dacă nodul vecin al celui curent este deja marcat cu aceeași valoare ca și cel curent, atunci concluzionăm că graful nu este bipartit.

```
#include <fstream>
using namespace std;
int n,m,x,y,f[101][101],u[101],i,j,a;
int main () {
    ifstream fin("bipartit.in");
    ofstream fout("bipartit.out");
    fin>>n>>m;
    for (i=1;i<=m;i++){
        fin>>x>>y;
        f[x][y]=f[y][x]=1;
    }
    fin>>a;
    for (i=1;i<=a;i++){
        fin>>x;
        u[x] = 1;
    }
    for (i=2;i<=n;i++){
        for (j=1;j<i;j++){
            if (f[i][j]==1 && u[i] == u[j]){
                fout<<"NU";
                return 0;
            }
        }
    }
    fout<<"DA";
    return 0;
```

```
}
```

În rezolvarea problemelor anterioare s-a folosit matricea de adiacență ca modalitate de reprezentare a grafurilor. În cele mai multe probleme care necesită parcurgeri complexe precum și pentru cazurile când grafurile sunt mai mari, modalitatea de reprezentare folosită este cea a listelor de vecini. Iată în continuare câteva exemple de utilizare a acestora. Pentru fiecare exemplu vom enunța și rezolva o problemă.

Problema 1 cu liste de vecini.
Din fișierul de intrare se citesc datele unui graf neorientat: `n` - numărul de noduri, `m` - numărul de muchii și apoi `m` perechi de câte două valori cuprinse între `1` și `n` reprezentând extremitățile câte unei muchii. Considerăm că muchiile nu se repetă în fișierul de intrare. Să se scrie în fișierul de ieșire `n` linii, câte una pentru fiecare nod. Fiecate dintre aceastea conține mai întâi gradul nodului apoi vecinii nodului într-o ordine oarecare. Liniile se scriu în ordinea crescătoare a numerotării nodurilor, de la `1` la `n`.
Considerăm că numărul de noduri este maxim `100000` și numărul de muchii este maxim `200000`.

Exemplu

date.in	date.out
5 7 1 2 3 4 5 4 1 3 3 5 1 5 4 1	4 2 3 5 4 1 1 3 4 1 5 3 3 5 1 3 4 3 1

Soluție
Reprezentarea cu o matrice de adiacență nu mai este practică întrucât numărul de noduri este foare mare. Vom folosi așadar câte un vector (de tipul din biblioteca standard) pentru fiecare nod. Vom avea așadar un tablou unidimensional de astfel de vectori, cu maxim `100000` de componente,
La fiecare muchie citită, cu extremități `x` și `y`, vom adăuga nodul `x` în lista `y` dar și nodul `y` în lista `x` (graful fiind neorientat). Faptul că muchiile se dau distincte în fișierul de intrare ne ajută, altfel în lista de vecini a unui nod ar fi apărut dubluri.
Odată construite listele de vecini obținem gradul unui nod prin apelul metodei `size()` a vectorului său (cu timp de executare constant) iar prin parcurgerea vectorului obținem

vecinii săi (în ordinea în care au fost adăugați, adică cel mai probabil în ordinea din fișierul de intrare).

```
#include <fstream>
#include <vector>
using namespace std;
ifstream fin ("date.in");
ofstream fout("date.out");
int n, m, x, y;
vector<int> L[100001];
int main () {
    fin>>n>>m;
    for (int i=1;i<=m;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
    }
    for (int i=1;i<=n;i++) {
        fout<<L[i].size()<<" ";
        for (int j=0;j<L[i].size();j++)
            fout<<L[i][j]<<" ";
        fout<<"\n";
    }
    return 0;
}
```

Problema 2 cu liste de vecini
În contextul problemei anterioare, se cere să afișăm lista de vecini sortată.

Soluție
Există două abordări pe care le vom prezenta și analiza pe rând.

1. Construim listele ca mai sus apoi le ordonăm crescător.

```
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
ifstream fin ("date.in");
ofstream fout("date.out");
int n, m, x, y;
vector<int> L[100001];
int main () {
    fin>>n>>m;
    for (int i=1;i<=m;i++) {
        fin>>x>>y;
        L[x].push_back(y);
        L[y].push_back(x);
```

```
    }

    for (int i=1;i<=n;i++)
        sort(L[i].begin(), L[i].end());

    for (int i=1;i<=n;i++) {
        fout<<L[i].size()<<" ";
        for (int j=0;j<L[i].size();j++)
            fout<<L[i][j]<<" ";
        fout<<"\n";
    }
    return 0;
}
```

Ca detalii de implementare observăm folosirea funcției de bibliotecă (`sort`) căreia i se transmit drept parametri adresa pentru primul element din zona de sortat și adresa de la finalul zonei de sortat. Timpul de rulare al acestei funcții este de ordinul $L*\log_2 L$ (am notat cu `L` lungimea zonei de sortat). Așadar avem în plus un factor logaritmic la estimarea timpului de rulare pe cazul cel mai defavorabil.

2. În loc să folosim `vector` pentru a stoca vecinii unui element, vom folosi `set`. Aceasta este tot o structură preimplementată în biblioteca standard, deci noi o folosim direct (implementarea internă se bazează pe arbori binari de căutare, echilibrați și dispunem de timp de calcul logaritmic pentru fiecare dintre operațiile importante: adăugare, eliminare, ștergere; în schimb structura consumă de regulă ceva mai multă memorie decât vector întrucât pentru fiecare nod al arborelui se țin pointeri către cei doi fii și către tată).

Elementele unui set sunt automat distincte (dacă se încearcă inserarea unui element care deja există, structura rămâne nemodificată).

```
#include <fstream>
#include <set>
using namespace std;
ifstream fin ("date.in");
ofstream fout("date.out");
int n, m, x, y;
set<int> S[100001];
int main () {
    fin>>n>>m;
    for (int i=1;i<=m;i++) {
        fin>>x>>y;
        S[x].insert(y);
        S[y].insert(x);
    }
    for (int i=1;i<=n;i++) {
```

```
        fout<<S[i].size()<<" ";
        for (set<int>::iterator it=S[i].begin();it!=S[i].end();it++)
            fout<<*it<<" ";
        fout<<"\n";
    }
    return 0;
}
```

Iată câteva observații.
Dacă accesul la un anume element este de ordin logaritmic, traversarea întregului set are timp de calcul liniar (se face o parcurgere a arborelui de căutare echilibrat folosind pointerii stocați la mutarea dintr-un nod în altul).
Traversarea se realizează cu un iterator. Acesta este o variabilă care poate lua la un moment dat ca valoare adresa unui nod al structurii. Iteratorii au fost gândiți încât pentru utilizator traversarea unei structuri să se scrie în cod în mod asemănător, chiar dacă modul intern de implementare este diferit de la o structură la alta.
Declararea unui iterator se face indicând ca tip de date o combinație între structura de folosit și cuvântul iterator, legate prin operatorul de rezoluție.
`set<int>::iterator it;`
`it` este variabila iar `set<int>::iterator` este tipul de date.
Iteratorii admit instrucțiuni de atribuire și, mai cu seamă, operatorul `++` care permit avansarea conform modului de implementare a structurii la care se aplică (de exemplu la set elementele se traversează în ordine crescătoare).
Ca la orice pointer, folosind construcția `*it` avem acces la datele efective de la adresa referită de iterator. De asemenea, operațiile de comparare sunt des utilizate pentru iteratori.
La prima problemă, cea rezolvată folosind vectori, am ales să folosesc modalitatea clasică de parcurgere, ca pe un tablou unidimensional.
Iată și varianta cu iteratori, la fel de eficientă:

```
for (int i=1;i<=n;i++) {
  fout<<L[i].size()<<" ";
  for (vector<int>::iterator it = L[i].begin(); it!=L[i].end();
it++)
    fout<<*it<<" ";
  fout<<"\n";
}
```

Problema 3 cu liste de vecini.
Cum stocăm datele dacă muchiile au asociate costuri?
Prin folosirea matricei de adiacență de regulă se pune în loc de `1` costul muchiei corespunzătoare.
Când se lucrează cu liste de vecini, în acestea, în loc să se țină doar valorile vecinilor, se vor păstra perechi (`vecin`, `cost`).
Astfel, dacă lista nodului nod este formată din perechile ($vecin_1$, $cost_1$), ($vecin_2$, $cost_2$) ... ($vecin_k$, $cost_k$), atunci avem muchiile: `nod` $vecin_1$ de cost $cost_1$, `nod` $vecin_2$ de cost $cost_2$ ... `nod` $vecin_k$ de cost $cost_k$.

Codul de mai jos consideră că datele de intrare au formatul descris la prima problemă doar că la fiecare muchie, în loc de perechea extremităților, se dă un triplet care conține și costul muchiei.

```
#include <fstream>
#include <vector>
using namespace std;
ifstream fin ("date.in");
ofstream fout("date.out");
int n, m, x, y, k, cost;
vector< pair<int, int> > L[100001];
int main () {
    fin>>n>>m;
    for (int i=1;i<=m;i++) {
        fin>>x>>y>>cost;
        L[x].push_back( make_pair(y,cost) );
        L[y].push_back( make_pair(x,cost) );
    }
    fin>>k;
    fout<<"Vecinii nodului "<<k<<" si costurile muchiilor la
ei:\n";
    for (vector<pair<int, int> >::iterator it = L[k].begin();
                          it!=L[k].end(); it++)
        fout<<it->first<<" "<<it->second<<"\n";

    return 0;
}
```

Codul de mai sus face să se memoreze datele conform modului descris anterior și apoi să se afișeze perechile (`vecin`,`cost`) pentru fiecare vecin al unui nod dat `k`.
Aici am folosit tipul `pair` care este o structură cu două câmpuri, primul se cheamă `first` și al doilea `second`. E bine de știut că datele de tip `pair` se pot sorta, având definit implicit comparator (se compară mai întâi după câmpul `first` și în caz de egalitate după `second` - deseori este necesară sortarea listei de vecini după cost și pentru asta de regulă se trece costul ca și `first`).
Remarcăm modul de accesare a câmpurilor unei structuri atunci când avem adresa ei (operatorul `->`).
De asemenea, se observă folositea funcției make_pair, care returnează un pair format din cele două valori date ca parametri.