UNIVERSITATEA DE STAT DIN MOLDOVA
FACULTATEA MATEMATICĂ ŞI INFORMATICĂ
DEPARTAMENTUL INFORMATICĂ

CEMÎRTAN CRISTIAN

Lucrare de laborator nr. 1

La disciplina **Algoritmica grafurilor**

Coordonator: Țurcanu Călin, lector universitar

Chişinău, 2022

CUPRINS

I. CERINȚE DE REALIZAT 3

II. SARCINA 1 4

2. 1. Comentariu 4

2. 2. Cod sursă 6

2. 3. Rezultatele programului 8

III. SARCINA 2 9

3. 1. Comentariu 9

3. 2. Cod sursă 11

3. 3. Rezultatele programului 12

IV. SARCINA 3 13

4. 1. Prima subsarcină 13

4. 2. A doua subsarcină 13

CONCLUZII 15

I. CERINȚE DE REALIZAT

1. Să se introducă un graf prin matricea de adiacență, formați din ea matricea de incidență, apoi, din ea, formați matricea de adiacență a grafului muchiilor.
2. Se dă un graf neorientat conex. Utilizând parcurgerea in lățime găsiți raza și diametrul grafului.
3. De rezolvat analitic următoarele probleme:
   I. Fie V o mulțime cu n elemente. Să se determine:
      a) numărul grafurilor neorientate, a căror mulțime de vârfuri este V;
      b) numărul grafurilor neorientate cu bucle, a căror mulțime de vârfuri este V;
      c) numărul grafurilor orientate, a căror mulțime de vârfuri este V.
   II. Fie G un graf conex. Este adevărat oare că graful complementar nu este conex? Cum ar fi graful complementar, dacă graful G nu ar fi conex?

II. SARCINA 1

2. 1. Comentariu

Înainte să realizăm sarcina, facem cunoștință cu următoarele noțiuni:

- **Graf (neorientat)** – perechea (V, E), unde V este o mulțime de vârfuri (elemente) unice, unde $|V| \geq 1$, iar E este o mulțime formată din muchii (perechi neordonate) ce leagă vârfurile din V, unde $|E| \geq 0$;
- **Graf conex** – un graf ce are o singură componentă conexă. În general, se satisface condiția: pentru fiecare două vârfuri din graf, există un lanț ce le leagă;
- **Matricea de adiacență** – notată prin $A = ||a_{ij}||$, este o reprezentarea matricială a grafului $G = (V, E)$, unde $a_{ij} = \begin{cases} 1, & dacă\ v_i \sim v_j \\ 0, & altfel \end{cases}$
- **Matricea de incidență** – notată prin $B = ||b_{ij}||$, este o reprezentare matricială a grafului $G = (V, E)$, unde $b_{ij} = \begin{cases} 1, & dacă\ v_j\ este\ incident\ cu\ e_i \\ 0, & altfel \end{cases}$
- **Graf al muchiilor grafului** $G = (V, E)$ – notată prin L(G), este graful unde vârfurile căruia reprezintă muchiile grafului G, moștenind și adiacențele corespunzătoare.

Transformarea matricei de adiacență în cea de incidență, a avut loc utilizând următorul algoritm:

1. Pregătim matricea de incidență B. Fie $n = |V|$;
2. $k \leftarrow 0$ și $i \leftarrow 0$;
3. Dacă $i >= n$, atunci returnează matricea B;
4. $j \leftarrow i + 1$;
5. Dacă $A_{ij} \neq A_{ji}$, algoritmul eșuează, pentru că graful e orientat;
6. Dacă $A_{ij} = 1$, atunci $B_{ik} \leftarrow 1$, $B_{jk} \leftarrow 1$, și $k \leftarrow k + 1$;
7. $j \leftarrow j + 1$;
8. Dacă $j < n$, atunci se revine la pasul 5;
9. $i \leftarrow i + 1$;
10. Se revine la pasul 3.

Transformarea matricei de incidență în matricea de adiacență a grafului muchiilor, a avut loc utilizând următorul algoritm:

1. Se știe că matricea de incidență B nu e mereu pătrată, de aceea vom avea dimensiunile $n \times m$.

2. Pregătim matricea de adiacență A;
3. $j \leftarrow 0$;
4. Dacă $j >= m$, atunci returnează matricea A;
5. $i_1 \leftarrow 0$;
6. Dacă $i_1 < n$ și $B_{i1j} \neq 1$, atunci $i_1 \leftarrow i_1 + 1$, și se revine la pasul 6;
7. Dacă $i_1 = n$, algoritmul eșuează;
8. $i_2 \leftarrow 0$;
9. Dacă $i_2 < n$ și $B_{i2j} \neq 1$, atunci $i_2 \leftarrow i_2 + 1$, și se revine la pasul 6;
10. Dacă $i_2 = n$, algoritmul eșuează;
11. $k \leftarrow j + 1$;
12. Dacă $B_{i1j} = 1$ și $B_{i1k} = 1$, sau $B_{i2j} = 1$ și $B_{i2k} = 1$, atunci $A_{jk} \leftarrow 1$, $A_{kj} \leftarrow 1$;
13. $k \leftarrow k + 1$;
14. Dacă $k < m$, atunci se revine la pasul 12;
15. $j \leftarrow j + 1$;
16. Se revine la pasul 4.

2. 2. Cod sursă

```
#include <iostream>
#include <iostream>
#include <deque>
#include <set>

using Matrix = std::deque<std::deque<bool>>;

static auto InputAdjMtx()
{
    size_t n;

    std::cout << "n = ";
    std::cin >> n;

    Matrix mtx(n, Matrix::value_type(n));

    for (size_t j, i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            std::cin >> mtx[i][j];

    return mtx;
}

static auto AdjToInc(const Matrix& adj)
{
    const auto n = adj.size();
    Matrix mtx(n);

    for (size_t j, i = 0; i < n; ++i)
    {
        if (adj[i][i])
            throw std::invalid_argument("Graful trebuie sa nu fie pseudograf.");

        for (j = i + 1; j < n; ++j)
        {
            if (adj[i][j] != adj[j][i])
                throw std::invalid_argument("Graful trebuie sa nu fie orientat.");

            if (adj[i][j])
            {
                for (auto& r : mtx)
                    r.resize(r.size() + 1);

                mtx[i].back() = mtx[j].back() = true;
            }
        }
    }

    return mtx;
}

static auto IncToAdjEdgeMtx(const Matrix& inc)
{
    auto n = inc.size(), m = inc.at(0).size();
    Matrix mtx(m, Matrix::value_type(m));

    for (size_t j = 0; j < m; ++j)
```

```
    {
        size_t i1, i2;

        {
            for (i1 = 0; i1 < n; ++i1)
                if (inc[i1][j])
                    break;

            if (i1 == n)
                throw std::invalid_argument("Graful are o muchie invalida.");
        }

        {
            for (i2 = i1 + 1; i2 < n; ++i2)
                if (inc[i2][j])
                    break;

            if (i2 == n)
                throw std::invalid_argument("Graful are o muchie invalida.");
        }

        for (size_t k = j + 1; k < m; ++k)
            if ((inc[i1][j] && inc[i1][k]) || (inc[i2][j] && inc[i2][k]))
                mtx[j][k] = mtx[k][j] = true;
    }

    return mtx;
}

static auto& operator<<(std::ostream& out, const Matrix& mtx)
{
    for (auto& r : mtx)
    {
        for (auto& v : r)
            out << v << ' ';

        out << '\n';
    }

    return out << '\n';
}

int main()
{
    try
    {
        auto mtx = InputAdjMtx();
        std::cout << '\n';

        mtx = AdjToInc(mtx);
        std::cout << mtx;

        mtx = IncToAdjEdgeMtx(mtx);
        std::cout << mtx;
    }
    catch (std::exception& e)
    {
        std::cerr << e.what();
    }
```

```
}
```

2. 3. Rezultatele programului

```
n = 9
0 0 0 1 1 0 1 0 0
0 0 0 1 0 1 0 0 0
0 0 0 0 1 1 1 0 0
1 1 0 0 0 1 1 0 0
1 0 1 0 0 1 0 0 1
0 1 1 1 1 0 0 0 0
1 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 1 0

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

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

Figura 2. 1

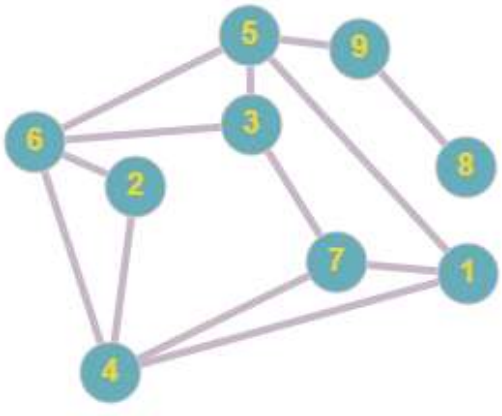

Figura 2. 2. Reprezentarea grafică a grafului

III. SARCINA 2

3. 1. Comentariu

Înainte să realizăm sarcina, facem cunoștință cu următoarele noțiuni:

- **Lanț** – $L = (v_1, ..., v_k)$. O mulțime de vârfuri ce aparțin mulțimii V, unde se satisface condiția: $v_i \sim v_{i+1}$, unde $i \in \mathbb{N} \cap [1, k)$.
- **$\Gamma(v_i)$**, unde $v_i \in V$ – o mulțime de vârfuri adiacente (vecini) cu v_i;
- **$\deg(v_i)$** – grad al vârfului v_i, sinonim pentru $|\Gamma(v_i)|$;
- **Distanța dintre două vârfuri u și v** – $d(u, v)$. Lungimea minimă a lanțului cu extremitățile u și v;
- **$e(v)$** – excentricitatea vârfului v. Distanța maximă dintre v și alte vârfuri;
- **r(G)** – raza grafului. Excentricitatea minimă a unui vârf din graf;
- **D(G)** – diametrul grafului. Excentricitatea maximă a unui vârf din graf;
- **Parcurgerea în lățime** – un algoritm ce parcurge un graf, începând cu vârful de pornire, apoi alte vârfuri adiacente cu vârful de pornire, și apoi această operațiune se extinde pentru alte vârfuri adiacente, până când au fost parcurse toate vârfurile dintr-o componentă conexă, din care face parte vârful de pornire. Acest algoritm este utilizat pentru a găsi lanțuri minime între vârfuri, cum a spus d-nul Țurcanu la o lecție de laborator: *„torni o găleată de apă într-un labirint, și drumul unde apa se scurge primul la sfârșitul labirintului – este cel mai scurt”*.

Eu am implementat algoritmul de parcurgere în lățime, în felul următor:

- Algoritmul ia doi parametri: harta de adiacență și vârful de pornire.
- **Observație**: am implementat algoritmul sub stilul orientat pe obiecte.

1. Verificăm dacă vârful de pornire face parte din harta de adiacență;
2. Dacă vârful dat nu există în hartă, atunci algoritmul eșuează;
3. Se inițializează o hartă de distanță, unde cheia este vârful parcurs, iar valoarea – distanță de la vârful de pornire;
4. Se inițializează lista cu iteratori la vârfuri din harta de adiacență, primul element fiind iteratorul la vârful de pornire;
5. Pentru fiecare iterator-element din listă (lista se extinde dinamic):
   - ✓ Dacă nu mai sunt elementele recent adăugate în listă, atunci algoritmul returnează excentricitatea vârfului de pornire (distanța ultimului vârf parcurs din listă). Dacă

|V| nu coincide cu mărimea listei, atunci algoritmul eșuează, deoarece graful nu e conex.

- Pentru fiecare vârf din mulțimea Γ a vârfului obținut prin iteratorul curent:
  - Dacă se introduce cu succes în harta de distanță, următoarea înregistrare: cheia – vârful curent, valoarea se adaugă cu 1;
    - ✓ Se adaugă cu succes, dacă și numai dacă inițial cheia nu e prezentă în hartă.
    - Se anexează în listă, vârful curent din mulțimea Γ.

Vorbind despre datele de intrare, graful va fi introdus sub forma algebrică a muchiilor, care apoi va fi convertit în harta de adiacență.

3. 2. Cod sursă

```
#include <iostream>
#include <algorithm>
#include <map>
#include <set>
#include <list>

using Vertex = size_t; // varf
using Set = std::set<Vertex>; // multimea gamma
using Graph = std::map<Vertex, Set>; // harta de adiacenta

static auto Eccentricity(const Graph& graph, Vertex start)
{
    if (auto startIt = graph.find(start); startIt != graph.end())
    {
        std::map<Vertex, size_t> depth{ { start, 0 } };
        std::list<Graph::const_iterator> seq{ startIt };

        for (auto& it : seq)
        {
            auto& [vertex, gamma] = *it;
            for (auto& v : gamma)
                if (depth.emplace(v, depth[vertex] + 1).second)
                    seq.emplace_back(graph.find(v));
        }

        if (depth.size() != graph.size())
            throw std::invalid_argument("Graful nu este conex.");

        return depth[seq.back()->first];
    }

    throw std::invalid_argument("Varful de pornire dat nu exista in graf.");
}

static auto InputGraph()
{
    Graph graph;
    size_t m;

    std::cout << "m = ";
    std::cin >> m;

    if (m != 0)
        for (size_t i = 0; i < m; )
        {
            std::cout << "e" << i + 1 << " = ";
            Vertex a, b;

            if (std::cin >> a >> b; a != b)
            {
                graph[a].insert(b);
                graph[b].insert(a);
                ++i;
            }
            else
                std::cout << "Nu permitem bucle.\n";
        }
```

```
    else
        graph[0];

    return graph;
}

int main()
{
    auto graph = InputGraph();

    try
    {
        auto it = graph.begin();
        auto r = Eccentricity(graph, it->first), d = r;

        while (++it != graph.end())
            std::tie(r, d) = std::minmax({ Eccentricity(graph, it->first), r, d });

        std::cout << "r(G) = " << r << "; D(G) = " << d;
    }
    catch (std::invalid_argument& e)
    {
        std::cerr << e.what();
    }
}
```

3. 3. Rezultatele programului

```
m = 13
e1 = 1 4
e2 = 1 5
e3 = 1 7
e4 = 2 4
e5 = 2 6
e6 = 3 5
e7 = 3 6
e8 = 3 7
e9 = 4 6
e10 = 4 7
e11 = 5 6
e12 = 5 10
e13 = 9 10
r(G) = 2; D(G) = 4
```

Figura 3. 1

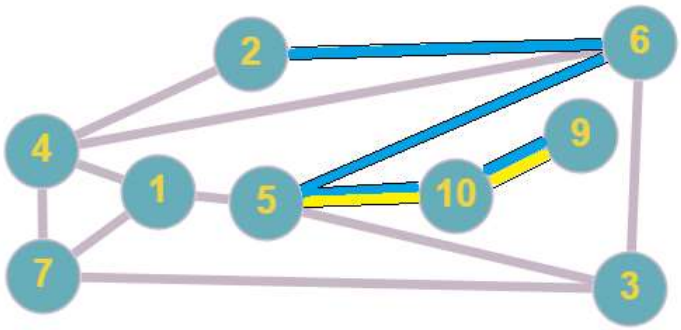

Figura 3. 2. Reprezentarea grafică a grafului. Lungimea lanțului albastru este diametrul grafului, iar cea galbenă - raza

IV. SARCINA 3

4. 1. Prima subsarcină

Reprezentăm un graf $G = (V, E)$ printr-o matrice de adiacență A:

$$\begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix}$$

Având rezolvate primele două sarcini de laborator, se știe profund despre această matrice că:

- Pentru grafuri neorientate: matricea este simetrică, înseamnă că obligatoriu $A_{ij} = A_{ji}$;
- Pentru grafuri orientate: Egalitatea între A_{ij} și A_{ji} e opțională;
- Pentru pseudografuri (grafuri cu bucle): unde cel puțin un element A_{ii}, din diagonala principală a matricei, este egal cu 1.
- Numărul de elemente deasupra/sub diagonalul principal, sau numărul maxim de muchii pentru un graf neorientat cu n vârfuri, este egal cu:

$$\frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

Numărul de grafuri neorientate, cu n vârfuri, este egal cu:

$$2^{\frac{n^2-n}{2}}$$

Numărul de pseudografe neorientate (obligatoriu cu bucle), cu n vârfuri, este egal cu:

$$(2^n - 1) \cdot 2^{\frac{n^2-n}{2}}$$

Numărul de grafuri orientate (fără bucle), cu n vârfuri, este egal cu:

$$\left(2^{\frac{n^2-n}{2}}\right)^2 = 2^{n^2-n}$$

Numărul tuturor grafurilor orientate, cu n vârfuri, este egal cu:

$$2^{n^2}$$

■

4. 2. A doua subsarcină

Înainte să rezolvăm această subsarcina, facem cunoștință cu definiția grafului complementar, a grafului $G = (V, E)$. Notat prin $\overline{G}$, graful complementar are mulțimea V copiată de la V_G, iar mulțimea E = $E_{K_n} \setminus E_G$, unde K_n este graful complet de ordin n. Putem spune că $G \cup \overline{G} = K_n$, sau $G \cap \overline{G} = \emptyset$, și ne reamintim că n = |V|. Reprezentarea matricei de adiacență a grafului $\overline{G}$ va fi în felul următor:

$$\begin{pmatrix} A_{11} & 1-A_{12} & \cdots & 1-A_{1(n-1)} & 1-A_{1n} \\ 1-A_{21} & A_{22} & \cdots & 1-A_{2(n-1)} & 1-A_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1-A_{(n-1)1} & 1-A_{(n-1)2} & \cdots & A_{(n-1)(n-1)} & 1-A_{(n-1)n} \\ 1-A_{n1} & 1-A_{n2} & \cdots & 1-A_{n(n-1)} & A_{nn} \end{pmatrix}$$

Ca consecință, gradul unui vârf din graful complementar, va fi:

$$deg_{\overline{G}}(v_i) = |V_G| - 1 - deg_G(v_i)$$

■

Fie G un graf conex. Este adevărat oare că graful complementar nu este conex?

În cazul grafurilor cu gradul maxim $\Delta(G) = n - 1$, afirmația este adevărată. Vârfurile cu acest grad devin izolate în graful complementar, împărțind graful în componente conexe.

În general, afirmația este fi *semi-adevărată*, sau nu este în întregime adevărată. Vom demonstra prin inducție: fie P_n un graf lanț cu n = 4. Observând figura 3. 2, complementul grafului lanț tot rămâne conex, și că vârfurile cu gradul $\Delta(P_n)$ și celelalte cu gradul $\delta(P_n)$, s-au mutat cu locul.

- ✓ Graf lanț – un graf unde două vârfuri au gradul 1 (sunt terminale), iar *celelalte vârfuri au gradul 2 (dacă $|V| > 2$).*

Figura 4. 1. Graful Pn

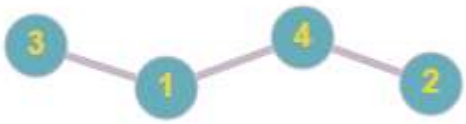

Figura 4. 2. Graful complementar a lui Pn

Cum ar fi graful complementar, dacă graful G nu ar fi conex?

Numaidecât conex, deoarece „*starea adiacenței" (există sau nu)* a fiecărei două vârfuri distincte (nu vorbim despre pseudografe), din graful G, devine opusă în graful complementar, unde vârfurile neadiacente devin adiacente, și ca rezultat se „fuzionează" componentele conexe într-o una conexă, obținând un graf (complementar) conex.

Aceasta duce la următoarea afirmație: *dacă este dat un graf și complementul său, atunci cel puțin unul din doi este conex.*

■

CONCLUZII

Primele două programe au fost elaborate în limbajul de programare C++17, utilizând doar bibliotecile prevăzute de membrii comitetului C++. Pentru a rezolva sarcinile practice în privința grafurilor, am utilizat următoarele tipuri de date: vectori dinamici, seturi și hărți. De asemenea, am implementat și comportamentele definite (aruncarea excepțiilor) pentru datele de intrare care nu corespund sarcinilor date.

În opinia mea, harta de adiacență este mai eficientă decât matricea de adiacență, din punctul de vedere a consumului de memorie. Harta de adiacență stochează mulțimea Γ pentru orice vârf, iar matricea de adiacență stochează informațiile de prisos, de ex. vârfurile neadiacente, diagonala principală deseori populată cu 0.

Rezolvarea sarcinilor analitice au fost realizate în majoritate utilizând conceptul matricei de adiacență, deoarece este cea mai intuitivă formă de a reprezenta un graf – orice element a matricei egal cu 1 înseamnă că există o muchie în care sunt incidente două vârfuri respective.