

UNIVERSITATEA DE STAT DIN MOLDOVA
FACULTATEA MATEMATICĂ ȘI INFORMATICĂ
DEPARTAMENTUL INFORMATICĂ

CEMÎRTAN CRISTIAN

Lucrare de laborator nr. 2
La disciplina **Algoritmica grafurilor**

Coordonator: Țurcanu Călin, lector universitar

Chișinău, 2022

CUPRINS

I. CERINȚA DE REALIZAT.....	3
II. ALGORITMUL LUI BRON ȘI KERBOSCH	4
2. 1. Comentariu	4
2. 2. Cod sursă	5
2. 2. 1. BronKerbosch.h.....	5
2. 2. 2. BronKerbosch.cpp	6
III. COLORAREA GRAFULUI UTILIZÂND PROGRAMAREA DINAMICĂ.....	9
3. 1. Comentariu	9
3. 2. Cod sursă	11
3. 2. 1. ColoringDP.h.....	11
3. 2. 2. ColoringDP.cpp	11
IV. PROGRAM	13
4. 1. Comentariu	13
4. 2. Cod sursă	13
4. 2. Rularea programului	15
CONCLUZII.....	16

I. CERINȚA DE REALIZAT

De determinat o colorare optimă a grafului prin metoda Programării Dinamice.

- ✓ Colorarea optimă implică algoritmul lui Bron și Kerbosch, care generează mulțimile stabile interior maximale.

II. ALGORITMUL LUI BRON ȘI KERBOSCH

2. 1. Comentariu

Înainte să realizăm algoritmul, facem cunoștință cu următoarele noțiuni:

- **Mulțime stabilă interior** – numită și anticică (complementul unei cliki), este o submulțime $S \subseteq V$, unde fiecare două vârfuri din S sunt neadiacente în graful $G = (V, E)$.
- **Mulțime maximală** – o mulțime care nu mai poate fi extinsă.

Scopul algoritmului lui Bron și Kerbosch este de a determina toate mulțimile stabile interioare *maximale* într-un graf $G = (V, E)$.

Descrierea algoritmului:

1. $k = 1$, $S = \emptyset$, $Q^- = \{\emptyset, \emptyset, \emptyset, \dots\}$, $Q^+ = \{V, \emptyset, \emptyset, \dots\}$;
2. $v_i \leftarrow Q^+_{k_1}$;
3. $S \leftarrow S \cup v_i$;
4. $k \leftarrow k + 1$;
5. $Q^+_k \leftarrow Q^+_{k-1} \setminus (\Gamma(v_i) \cup \{v_i\})$, $Q^-_k \leftarrow Q^-_{k-1} \setminus \Gamma(v_i)$;
6. Dacă $|Q^-_k| = 0$:
Dacă $|Q^+_k| = 0$, atunci se anexează S în lista mulțimilor stabile interior (și se trece la pasul 7);
Altfel, se revine la pasul 2.
Altfel, dacă $\forall v \in Q^-_k, \forall v' \in \Gamma(v), \exists v' \in Q^+_k$, atunci se revine la pasul 2.
7. $k \leftarrow k - 1$;
8. Dacă $k = 0$ sau $k = 1$, $|Q^+_1| = 1$, atunci algoritmul își termină execuția.
9. $v_i \leftarrow Q^+_{k_1}$;
10. $Q^+_k \leftarrow Q^+_k \setminus \{v_i\}$, $Q^-_k \leftarrow Q^-_{k-1} \setminus \{v_i\}$, $S \leftarrow S \setminus \{v_i\}$;
11. Se revine la pasul 6.

2. 2. Cod sursă

2. 2. 1. BronKerbosch.h

```
#pragma once
#include <set>
#include <map>
#include <list>

using Vertex = size_t;
using Set = std::set<Vertex>;
using Graph = std::map<Vertex, Set>;

inline auto operator-(Graph g, const Set& s)
{
    for (auto& v : s)
        g.erase(v);

    for (auto& [_, gamma] : g)
        for (auto& v : s)
            gamma.erase(v);

    return g;
}

inline auto VertexSet(Graph g)
{
    Set s;

    for (auto& [v, _] : g)
        s.emplace(v);

    return s;
}

class BronKerbosch
{
    class Iterator
    {
    public:
        friend class BronKerbosch;
        const BronKerbosch* bk;
        std::list<std::pair<Set, Set>> q;
        Set s;
        bool end;

        Iterator(const BronKerbosch* bk, bool end) : bk(bk), end(end) {}
        Iterator(const BronKerbosch* bk) : Iterator(bk, bk->g.empty()) {}

        auto& operator*() const { return s; }
        Iterator& operator++();
        friend bool operator==(const Iterator& lhs, const Iterator& rhs) =
        default;
    };

    Graph g;

public:
    BronKerbosch(const Graph& g) : g(g) {}
    BronKerbosch(Graph&& g) : g(std::move(g)) {}
};
```

```

    auto begin() const { return ++Iterator(this); }
    auto end() const { return Iterator(this, true); }
};

```

2. 2. 2. BronKerbosch.cpp

```

#include "BronKerbosch.h"
#include <algorithm>

auto BronKerbosch::Iterator::operator++() -> Iterator&
{
    if (end)
        return *this;

    auto& g = bk->g;

    if (q.size() != 0)
        goto nextS;

#define QM q.back().first
#define QP q.back().second

    q.resize(1);

    for (auto& [v, _] : g)
        QP.insert(v);

    for (; ; )
    {
        {
            auto vi = *QP.begin();
            s.insert(vi);

            q.emplace_back(q.back());
            QP.erase(QP.begin());

            for (auto& v : g.at(vi))
            {
                QM.erase(v);
                QP.erase(v);
            }
        }

        for (; ; )
        {
            if (QM.empty())
            {
                if (QP.empty())
                    return *this;

                break;
            }
            else if (std::ranges::all_of(QM, [&](auto& v)
            {
                auto& gamma = g.at(v);
                return std::ranges::find_first_of(gamma, QP) !=
gamma.end();
            })))

```

```

        break;

    nextS:
        if (q.pop_back(); q.empty() || (q.size() == 1 && QP.size() == 1))
        {
            q.clear();
            s.clear();
            end = true;
            return *this;
        }

        auto vi = *QP.begin();
        QP.erase(QP.begin());
        QM.insert(vi);
        s.erase(vi);
    }
}
#undef QP
#undef QM
#include "BronKerbosch.h"
#include <algorithm>

auto BronKerbosch::Iterator::operator++() -> Iterator&
{
    if (end)
        return *this;

    auto& g = bk->g;

    if (q.size() != 0)
        goto nextS;

#define QM q.back().first
#define QP q.back().second

    q.resize(1);

    for (auto v : VertexSet(g))
        QP.insert(v);

    for (; ; )
    {
        {
            auto vi = *QP.begin();
            s.insert(vi);

            q.emplace_back(q.back());
            QP.erase(QP.begin());

            for (auto& v : g.at(vi))
            {
                QM.erase(v);
                QP.erase(v);
            }
        }

        for (; ; )
        {
            if (QM.empty())

```

```

        {
            if (QP.empty())
                return *this;

            break;
        }
    else if (std::ranges::all_of(QM, [&](auto& v)
    {
        auto& gamma = g.at(v);
        return std::ranges::find_first_of(gamma, QP) !=
gamma.end());
        )))
        break;

    nextS:
    if (q.pop_back(); q.empty() || (q.size() == 1 && QP.size() == 1))
    {
        q.clear();
        s.clear();
        end = true;
        return *this;
    }

    auto vi = *QP.begin();
    QP.erase(QP.begin());
    QM.insert(vi);
    s.erase(vi);
}

}
#undef QP
#undef QM
}

```


III. COLORAREA GRAFULUI UTILIZÂND PROGRAMAREA DINAMICĂ

3. 1. Comentariu

Înainte să realizăm sarcina, facem cunoștință cu următoarele noțiuni:

- **Colorare** – Atribuirea culorilor pentru orice vârf din V , în așa fel ca nici o pereche din două vârfuri adiacente să aibă culoarea identică.
- **Numărul cromatic** – numărul minim/optim de culori necesare pentru a colora un graf.

Descrierea algoritmului (parcurea pe lățime):

1. Dacă $|V| = 0$, atunci algoritmul returnează 0.
2. $Q_2 = \emptyset$;
3. Pentru $\forall S \in \{\text{toate mulțimile stabile interior maximale din graful } G\}$:
Dacă $|S| = |V|$, atunci algoritmul returnează 1. Altfel, $Q_2 \leftarrow Q_2 \cup \{S'\}$.
4. $r \leftarrow 2$;
5. $Q_1 \leftarrow Q_2$, $Q_2 \leftarrow \emptyset$;
6. Pentru $\forall Q' \in Q_1$, $\forall S \in \{\text{toate mulțimile stabile interior maximale din graful } G - Q'\}$:
 - a. $S' \leftarrow S \cup Q'$;
 - b. Dacă $|S'| = |V|$, atunci algoritmul returnează valoarea r .
 - c. Dacă $\nexists q_1 \in Q_2$, $S' \subseteq q_1$:
 - i. $Q_2 \leftarrow Q_2 \setminus \{q_2 \subset S' \mid \forall q_2 \in Q_2\}$.
 - ii. $Q_2 \leftarrow Q_2 \cup \{S'\}$.
7. $r \leftarrow r + 1$;
8. Se revine la pasul 5.

Descrierea algoritmului (parcurea în adâncime):

- Fie M – hartă de tip cheie-valoare, unde cheia – mulțimea vârfurilor unui graf și valoarea – numărul cromatic a grafului.
1. $map_{\emptyset} \leftarrow 0, bound \leftarrow \infty, h \leftarrow 0$;
 2. Algoritmul returnează $\lambda(G)$.
- Fie aplicația $\lambda : G \rightarrow \{0\} \cup \mathbb{N}$:
 - a. Dacă $V \in M$:
 - i. Dacă $V = \emptyset$, atunci $bound \leftarrow \min\{h - 1, bound\}$;

- ii. λ returnează M_V .
- b. Dacă $h = bound$, atunci λ returnează ∞ .
- c. $M_V \leftarrow \infty$;
- d. Pentru $\forall S \in \{\text{toate mulțimile stabile interior maximale din graful } G\}$:
 - i. $h \leftarrow h + 1$;
 - ii. $M_V \leftarrow \min\{M_V, \lambda(G - S) + 1\}$;
 - iii. $h \leftarrow h - 1$.
- e. λ returnează M_V .

3. 2. Cod sursă

3. 2. 1. ColoringDP.h

```
#pragma once
#include "BronKerbosch.h"
size_t ChromaticBFS(const Graph& g);
size_t ChromaticDFS(const Graph& g);
```

3. 2. 2. ColoringDP.cpp

```
#include "ColoringDP.h"
#include <iterator>
#include <algorithm>
#include <functional>

size_t ChromaticBFS(const Graph& g)
{
    if (g.empty())
        return 0;

    std::list<Set> nextQ;

    for (auto& s : BronKerbosch(g))
    {
        if (s.size() == g.size())
            return 1;

        nextQ.emplace_back(s);
    }

    for (size_t r = 2; ; ++r)
    {
        auto q = std::exchange(nextQ, {});

        for (auto it = q.begin(); it != q.end(); it = q.erase(it))
            for (Set s : BronKerbosch(g - *it))
            {
                if (std::ranges::copy(*it, std::inserter(s, s.end())); s.size() ==
g.size())
                    return r;

                if (std::ranges::none_of(nextQ, [&](auto& sq) { return
std::ranges::includes(sq, s); }))
                {
                    std::erase_if(nextQ, [&](auto& sq) { return
std::ranges::includes(s, sq); });
                    nextQ.emplace_back(std::move(s));
                }
            }
    }
}

size_t ChromaticDFS(const Graph& g)
{
    std::map<Set, size_t> m({{}});

    std::function<size_t(const Graph&)> lambda =
```

```

[&,
    bound = SIZE_MAX,
    h = size_t{}
](auto& g) mutable
{
    auto set = VertexSet(g);

    if (auto it = m.find(set); it != m.end())
    {
        if (set.empty())
            bound = std::min(h - 1, bound);

        return it->second;
    }

    auto min = SIZE_MAX;

    if (h == bound)
        return min;

    for (auto& s : BronKerbosch(g))
    {
        ++h;

        if (auto temp = lambda(g - s); --h, temp != SIZE_MAX)
            min = std::min(min, temp + 1);
        else
            break;
    }

    return m[std::move(set)] = min;
};

return lambda(g);
}

```

IV. PROGRAM

4. 1. Comentariu

Programul citește de la tastatură un graf „neorientat”, sub formatul setului de adiacență, și apoi determină timpul estimat de executare pentru două implementări al algoritmului menționat în capitolul precedent. Dacă ambele implementări au returnat aceeași valoare, atunci se afișează culoarea cromatică a grafului. Prima implementare care va fi executată – parcurgere în adâncime, a doua – pe lățime.

4. 2. Cod sursă

```
#include "ColoringDP.h"
#include <iostream>
#include <string>
#include <sstream>
#include <chrono>
#include <cassert>

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

    std::cout << "n = ";
    (std::cin >> n).ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    for (size_t i = 1; i <= n; ++i)
    {
        std::cout << "v" << i << ": ";
        auto ptr = &graph[i];

        std::string temp;
        std::getline(std::cin, temp);
        std::istringstream istr(temp);

        for (Vertex v; istr >> v; ptr->insert(v))
            graph[v];
    }

    return graph;
}

static void Stopwatch()
{
    static bool status;
    auto now = std::chrono::high_resolution_clock::now();
    static decltype(now) start;

    if (status)
        std::cout << "Timpul estimat: " << now - start << '\n';
    else
        status = true;

    start = now;
}
```

```

}

int main()
{
    auto g = InputGraph();
    size_t hi[2]{};

    Stopwatch();
    hi[0] = ChromaticDFS(g);
    Stopwatch();

    hi[1] = ChromaticBFS(g);
    Stopwatch();

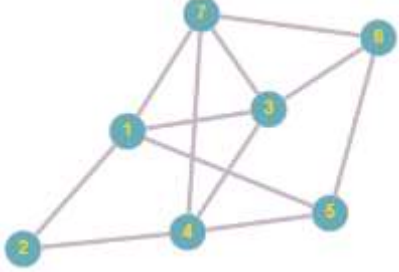
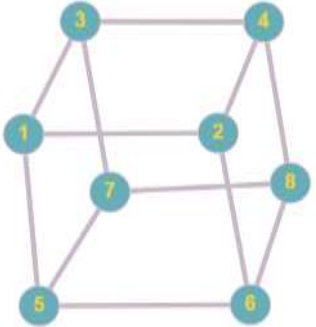
    assert(hi[0] == hi[1]);
    std::cout << "Numarul cromatic: " << hi[0];
}

```

4. 2. Rularea programului

Avem următoarea tabelă:

Tabela 4. 1

Graf			Graf complet K_{14}	Graf nul O_5
Rulare	<p> $n = 7$ $v1: 2\ 3\ 5\ 7$ $v2: 1\ 4$ $v3: 1\ 4\ 6\ 7$ $v4: 2\ 3\ 5\ 7$ $v5: 1\ 4\ 6$ $v6: 3\ 5\ 7$ $v7: 1\ 3\ 4\ 6$ Timpul estimat: 95400ns Timpul estimat: 160000ns Numarul cromatic: 3 </p>	<p> $n = 8$ $v1: 2\ 3\ 5$ $v2: 1\ 4\ 6$ $v3: 1\ 4\ 7$ $v4: 2\ 3\ 8$ $v5: 1\ 6\ 7$ $v6: 2\ 5\ 8$ $v7: 3\ 5\ 8$ $v8: 4\ 6\ 7$ Timpul estimat: 64200ns Timpul estimat: 119100ns Numarul cromatic: 2 </p>	<p> $n = 14$... Timpul estimat: 1585708300ns Timpul estimat: 4034710000ns Numarul cromatic: 14 </p>	<p> $n = 5$ $v1:$ $v2:$ $v3:$ $v4:$ $v5:$ Timpul estimat: 40300ns Timpul estimat: 263000ns Numarul cromatic: 1 </p>

CONCLUZII

Bibliotecile și programul principal au fost elaborate în limbajul de programare C++20, utilizând doar bibliotecile prevăzute de membrii comitetului C++. Pentru a rezolva sarcinile practice în privința grafurilor, am utilizat următoarele tipuri de date: liste înlănțuite, seturi și hărți.

Observând tabela 4. 1, putem spune că varianta parcurgerii în adâncime este mai rapidă decât cea de parcurgere pe lățime, dar utilizează mai multă memorie de calcul, deoarece utilizează o hartă ce memorizează numerele cromatice pentru fiecare subgraf cercetat. Pentru graful complet K_{14} , varianta parcurgerii în adâncime a utilizat 8MB de memorie, iar cealaltă – 3MB.