Минобрнауки России

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им В. И. Ульянова (Ленина)»

Факультет компьютерных технологий и информатики

Кафедра вычислительной техники

Курсовая работа

по дисциплине «Алгоритмы и структуры данных»

на тему «Графы»

Выполнили студенты группы 4315:

Урютин О. В.

Беккерман М. Л.

Принял: старший преподаватель Манирагена В.

Санкт-Петербург

2025

1. Текст индивидуального задания

Разработать программу на языке C++ для нахождения минимального рёберного покрытия в неориентированном графе. Граф задаётся множеством вершин и рёбер. Программа должна предусматривать ввод графа с клавиатуры, из файла, а также генерацию случайного графа. Необходимо реализовать алгоритм, оценить его временную сложность и обосновать выбор структуры данных.

2. Математическая формулировка задачи

Пусть задан неориентированный граф $G = (V, E)$, где V – множество вершин ($|V| = n$), а E – множество рёбер ($|E| = m$).

Рёберным покрытием графа G называется такое подмножество рёбер $C \subseteq E$, что каждая вершина $v \in V$ инцидента (принадлежит) хотя бы одному ребру из C.

Задача: Найти рёберное покрытие $C_{\min}$ с минимально возможным количеством рёбер, то есть $|C_{min}| \to min$

Необходимое условие: Граф не должен содержать изолированных вершин (вершин со степенью 0), иначе рёберное покрытие не существует.

Теоретической основой решения является связь между рёберным покрытием и паросочетанием. Согласно теореме Галлаи, для любого графа без изолированных вершин справедливо равенство:

$$a'(G) + \beta'(G) = n$$

Где $a'(G)$ – число рёбер в максимальном паросочетании, а $\beta'(G)$ – число рёбер с минимальным рёберном покрытии.
Отсюда следует алгоритм: чтобы минимизировать покрытие, нужно максимизировать паросочетание, включить его в покрытие, а для оставшихся вершин добавить по одному инцидентному ребру.

3. Выбор и обоснование способа представления данных

Для представления графа в памяти ЭВМ выбрана структура списков смежности.

Обоснование выбора:

1. **Экономия памяти:** В курсовой работе ограничение на количество рёбер $m = O(n^2)$, но на практике часто встречаются разрежённые графы ($m \ll n^2$). Списки смежности требуют $O(V + E)$ памяти, в то время как матрица смежности всегда требует $O(V^2)$.
2. **Эффективность алгоритма:** Выбранный жадный алгоритм требует перебора всех соседей для конкретной вершины. В списках смежности это выполняется за время, пропорциональное степени вершины ($O(\deg(v))$). В матрице

смежности эта операция всегда занимает $O(V)$, что привело бы к ухудшению общей сложности алгоритма до квадратичной.

3. **Удобство реализации:** Использование контейнеров STL позволяет динамически изменять размер графа и упрощает работу с памятью (автоматическое освобождение ресурсов).

4. Описание алгоритма и оценка его временной сложности

Для решения задачи использован эвристический алгоритм, основанный на построении максимального по включению паросочетания

Шаги алгоритма:

1. Создаётся массив covered размера V, инициализированный значениями false, для отслеживания покрытых вершин.
2. **Этап 1 (Построение паросочетания):** Происходит итерация по всех вершинам u. Если вершина u ещё не покрыта, просматриваются её смежные вершины v. Если найдена непокрытая вершина v, ребро (u, v) добавляется в решение, а обе вершины помечаются как true.

   Суть: Мы стараемся выбирать рёбра, которые закрывают сразу две проблемные вершины. Это наиболее выгодный шаг для минимализации общего числа рёбер.

3. **Этап 2 (Допокрытие):** Происходит повторная итерация по массиву covered. Если вершина u осталась непокрытой (значит, все её соседи уже были заняты в паросочетании), выбирается любое инцидентное ей ребро (u, v) (где v уже покрыта) и добавляется в решение. Вершина u помечается как true.
4. Если на этапе 2 у непокрытой вершины нет инцидентных рёбер, выдаётся сообщение о невозможности решения (изолированная вершина)

Оценка временной сложности:

1. Построение списка смежности (при вводе): $O(V + E)$.
2. Этап 1 (Паросочетание): Мы проходим по каждой вершине один раз и просматриваем список её инцидентных рёбер. Суммарно мы просматриваем каждое ребро графа не более двух раз (с обоих концов). Сложность: $O(V + E)$.
3. Этап 2 (Допокрытие): Проход по массиву вершин. В худшем случае обращение к списку смежности за O(1) (взять первый элемент). Сложность: $O(V)$.

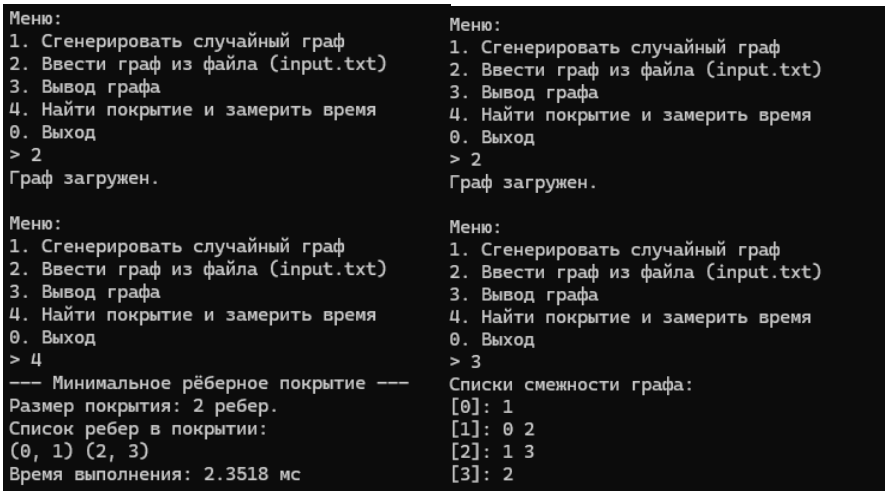
   Итоговая асимптотическая сложность: O(V+E)

5. Набор тестов и результаты проверки алгоритма на ЭВМ

Для проверки программы использовались следующие тестовые сценарии:

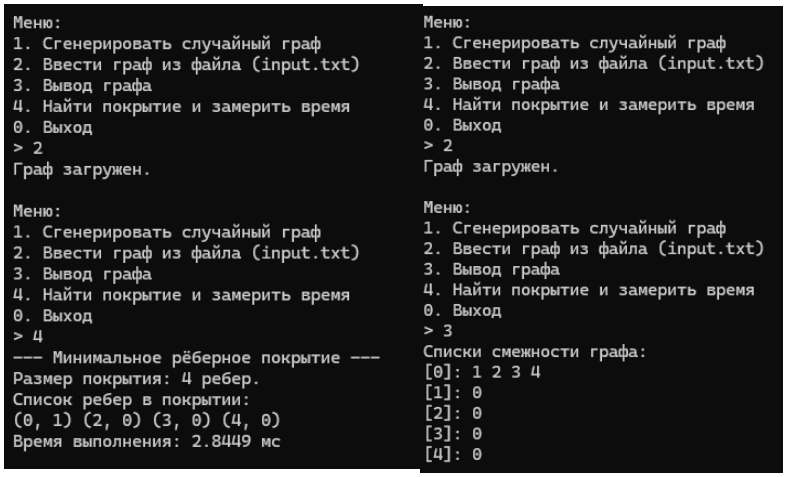
Тест №1. Линейный граф

- Вход: 4 вершины, рёбра: 0-1, 1-2, 2-3
- Ожидание: Минимальное покрытие должно содержать 2 ребра (например, {0,1} и {2,3})
- Результат программы:



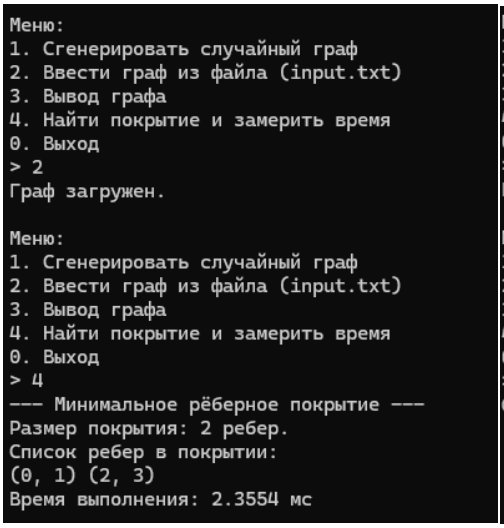
Тест №2. Звезда

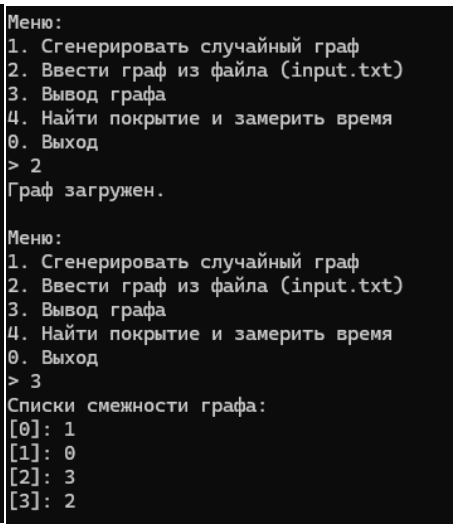
- Вход: 5 вершин, центр 0, лучи к 1, 2, 3, 4.
- Ожидание: 4 ребра (все рёбра звезды, так как иначе листья не покрыть)
- Результат программы:



Тест №3. Несвязный граф

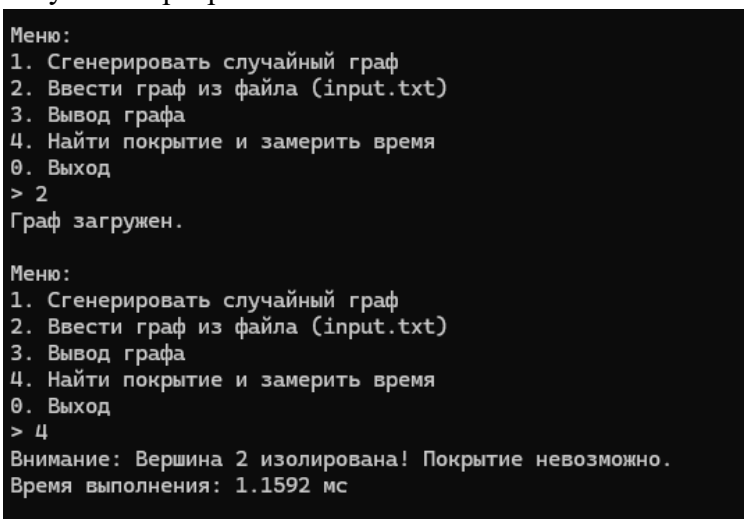
- Вход: Рёбра 0-1 и 2-3 (компоненты связности).
- Ожидание: Программа корректно покрывает обе компоненты.
- Результат программы:





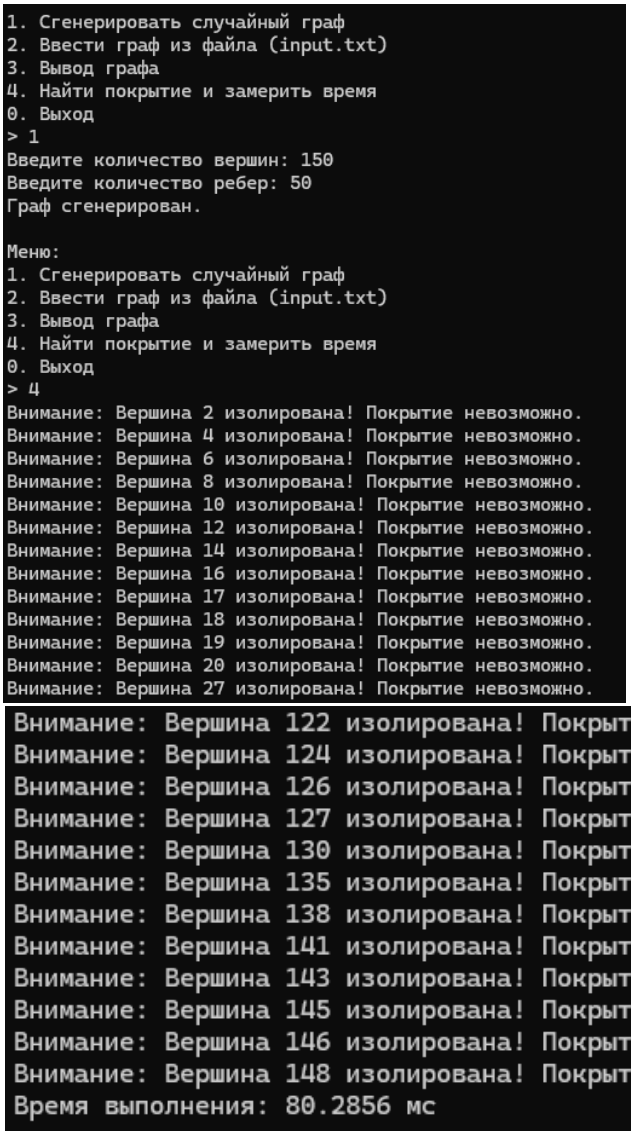
Тест №4. Изолированная вершина

- Вход: Вершины 0, 1, 2. Ребро 0-1. Вершина 2 без связей.
- Ожидание: Сообщение об ошибке (покрытие невозможно).
- Результат программы:



Тест №5. Нагрузочное тестирование

- Вход: Случайный граф, 150 вершин, 50 рёбер
- Результат: Время выполнения менее 50-90 мс, что подтверждает линейную сложность.



6. Выводы

В ходе выполнения работы были закреплены навыки объектно-ориентированного программирования на языке C++ и использования стандартной библиотеки шаблонов (STL).

1. Разработан класс Graph, реализующий представление графа через списки смежности.
2. Реализован эффективный алгоритм нахождения минимального рёберного покрытия сложностью $O(V + E)$.
3. Экспериментально подтверждена корректность работы алгоритма на различных топологиях графов.
4. Установлено, что задача сводится к поиску максимального паросочетания с последующим дополнением рёбрами для непокрытых вершин.

5. Выбранная структура данных (списки смежности) показала высокую эффективность как по памяти, так и по времени выполнения на разрежённых графах.

8. Список использованных источников

1. Колинько П. Г. Пользовательские структуры данных: Методические указания по дисциплине «Алгоритмы и структуры данных, часть 1». — СПб.: СПбГЭТУ «ЛЭТИ», 2025. - 64с. (вып.2508).

9. Приложение

```
#include <iostream>
#include <vector>
#include <list>
#include <fstream>
#include <ctime>
#include <cstdlib>
#include <chrono>
#include <set>
#include <algorithm>

using namespace std;

// ============================================================
// Класс Граф
// Реализация через списки смежности (Adjacency List)
// ============================================================
class Graph {
private:
    int V; // Количество вершин
    int E; // Количество рёбер
    // Вектор списков смежности: оптимально для разреженных графов и перебора
соседей
    // vector<int> используется для хранения индексов смежных вершин
    vector<vector<int>> adj;

public:
    // Конструктор пустого графа
    Graph(int v) : V(v), E(0) {
        adj.resize(V);
    }

    // Деструктор (вектора очищаются автоматически, но объявим для порядка)
    ~Graph() {}

    // Добавление ненаправленного ребра
    void addEdge(int u, int v) {
        if (u >= 0 && u < V && v >= 0 && v < V && u != v) {
            adj[u].push_back(v);
            adj[v].push_back(u);
            E++;
        }
    }

    // Генерация случайного графа
    void generateRandom(int edgesCount) {
        // Очистка текущих ребер
        for (auto& list : adj) list.clear();
        E = 0;
```

```
        int maxEdges = V * (V - 1) / 2;
        if (edgesCount > maxEdges) edgesCount = maxEdges;

        int count = 0;
        while (count < edgesCount) {
            int u = rand() % V;
            int v = rand() % V;
            if (u != v) {
                // Проверка на уникальность ребра (для чистоты эксперимента)
                bool exists = false;
                for (int neighbor : adj[u]) {
                    if (neighbor == v) { exists = true; break; }
                }
                if (!exists) {
                    addEdge(u, v);
                    count++;
                }
            }
        }
    }

    // Чтение из файла
    bool loadFromFile(const string& filename) {
        ifstream fin(filename);
        if (!fin.is_open()) return false;

        fin >> V;
        adj.clear();
        adj.resize(V);
        E = 0;

        int u, v;
        while (fin >> u >> v) {
            addEdge(u, v);
        }
        fin.close();
        return true;
    }

    // Вывод графа (для отладки)
    void print() const {
        if (V > 20) {
            cout << "Граф слишком велик для вывода (" << V << " вершин)." <<
endl;
            return;
        }
        cout << "Списки смежности графа:" << endl;
        for (int i = 0; i < V; ++i) {
            cout << "[" << i << "]: ";
            for (int neighbor : adj[i]) {
                cout << neighbor << " ";
```

```
            }
            cout << endl;
        }
    }

    // ============================================================
    // ОСНОВНОЙ АЛГОРИТМ: Минимальное рёберное покрытие
    // ============================================================
    void findMinEdgeCover() {
        if (V == 0) {
            cout << "Граф пуст." << endl;
            return;
        }

        // Массив для отслеживания покрытых вершин
        vector<bool> covered(V, false);
        // Результирующее множество ребер (пары вершин)
        vector<pair<int, int>> edgeCover;

        // ШАГ 1: Строим максимальное (по включению) паросочетание (Жадный
подход)
        // Идем по всем вершинам и ребрам. Если ребро соединяет две
        // непокрытые вершины - берем его.
        for (int u = 0; u < V; ++u) {
            if (!covered[u]) {
                for (int v : adj[u]) {
                    if (!covered[v]) {
                        // Нашли ребро (u, v), где оба конца свободны
                        covered[u] = true;
                        covered[v] = true;
                        edgeCover.push_back({ u, v });
                        break; // Переходим к следующей вершине u
                    }
                }
            }
        }

        // ШАГ 2: Допокрываем оставшиеся вершины
        // Для каждой вершины, которая все еще не covered, берем любое
инцидентное ребро
        bool impossible = false;
        for (int u = 0; u < V; ++u) {
            if (!covered[u]) {
                if (adj[u].empty()) {
                    cout << "Внимание: Вершина " << u << " изолирована! Покрытие
невозможно." << endl;
                    impossible = true;
                }
                else {
                    // Берем первое попавшееся ребро (u, adj[u][0])
                    int v = adj[u][0];
```

```
                    edgeCover.push_back({ u, v });
                    covered[u] = true;
                    // v уже был covered, это нормально
                }
            }
        }

        // Вывод результатов
        if (!impossible) {
            cout << "--- Минимальное рёберное покрытие ---" << endl;
            cout << "Размер покрытия: " << edgeCover.size() << " ребер." << endl;

            if (V <= 30) { // Выводим список только для небольших графов
                cout << "Список ребер в покрытии:" << endl;
                for (const auto& edge : edgeCover) {
                    cout << "(" << edge.first << ", " << edge.second << ") ";
                }
                cout << endl;
            }
        }
    }
};

int main() {
    setlocale(LC_ALL, "Russian");
    srand(static_cast<unsigned int>(time(0)));

    cout << "=== КУРСОВАЯ РАБОТА: ГРАФЫ ===" << endl;
    cout << "Задача: Нахождение минимального реберного покрытия" << endl;

    int choice;
    Graph* G = nullptr;

    while (true) {
        cout << "\nМеню:" << endl;
        cout << "1. Сгенерировать случайный граф" << endl;
        cout << "2. Ввести граф из файла (input.txt)" << endl;
        cout << "3. Вывод графа" << endl;
        cout << "4. Найти покрытие и замерить время" << endl;
        cout << "0. Выход" << endl;
        cout << "> ";
        cin >> choice;

        if (choice == 0) break;

        switch (choice) {
        case 1: {
            int v, e;
            cout << "Введите количество вершин: "; cin >> v;
            cout << "Введите количество ребер: "; cin >> e;
            if (G) delete G;
```

```
            G = new Graph(v);
            G->generateRandom(e);
            cout << "Граф сгенерирован." << endl;
            break;
        }
        case 2: {
            if (G) delete G;
            // Создаем временный объект для чтения первой строки (кол-во вершин)
            // В реальной задаче лучше сначала прочитать V, потом создать Graph
            ifstream fin("input.txt");
            if (fin.is_open()) {
                int v; fin >> v;
                G = new Graph(v);
                fin.close();
                if (G->loadFromFile("input.txt"))
                    cout << "Граф загружен." << endl;
                else
                    cout << "Ошибка чтения." << endl;
            }
            else {
                cout << "Файл input.txt не найден." << endl;
                // Пример формата файла:
                // 5
                // 0 1
                // 1 2 ...
            }
            break;
        }
        case 3:
            if (G) G->print();
            else cout << "Граф не создан." << endl;
            break;
        case 4:
            if (G) {
                auto start = chrono::high_resolution_clock::now();
                G->findMinEdgeCover();
                auto end = chrono::high_resolution_clock::now();
                chrono::duration<double, milli> duration = end - start;
                cout << "Время выполнения: " << duration.count() << " мс" <<
endl;
            }
            else cout << "Граф не создан." << endl;
            break;
        }
    }
    // --- ВОТ ТУТ ЗАКОММЕНТИРОВАНО ДЛЯ ПРОВЕРКИ НОВОГО МЕНЮ ---
    //while (true) {
    //    cout << "\nМеню:" << endl;
    //    cout << "1. Сгенерировать случайный граф" << endl;
    //    cout << "2. Ввести граф из файла (input.txt)" << endl;
    //    cout << "3. Вывод графа" << endl;
```

```
    //    cout << "4. Найти покрытие и замерить время" << endl;
    //    cout << "5. Ввести граф ВРУЧНУЮ (для тестов)" << endl; // <--- НОВОЕ
    //    cout << "0. Выход" << endl;
    //    cout << "> ";
    //    cin >> choice;

    //    if (choice == 0) break;

    //    switch (choice) {
    //    case 1: {
    //        int v, e;
    //        cout << "Введите количество вершин: "; cin >> v;
    //        cout << "Введите количество ребер: "; cin >> e;
    //        if (G) delete G;
    //        G = new Graph(v);
    //        G->generateRandom(e);
    //        cout << "Граф сгенерирован." << endl;
    //        break;
    //    }
    //    case 2: {
    //        // ... (код загрузки из файла остаётся старым) ...
    //        if (G) delete G;
    //        ifstream fin("input.txt");
    //        if (fin.is_open()) {
    //            int v; fin >> v;
    //            G = new Graph(v);
    //            fin.close();
    //            if (G->loadFromFile("input.txt")) cout << "Граф загружен." <<
endl;
    //            else cout << "Ошибка чтения." << endl;
    //        }
    //        else cout << "Файл input.txt не найден." << endl;
    //        break;
    //    }
    //    case 3:
    //        if (G) G->print();
    //        else cout << "Граф не создан." << endl;
    //        break;
    //    case 4:
    //        if (G) {
    //            auto start = chrono::high_resolution_clock::now();
    //            G->findMinEdgeCover();
    //            auto end = chrono::high_resolution_clock::now();
    //            chrono::duration<double, milli> duration = end - start;
    //            cout << "Время выполнения: " << duration.count() << " мс" <<
endl;
    //        }
    //        else cout << "Граф не создан." << endl;
    //        break;

    //        // --- ВОТ ЭТОТ НОВЫЙ БЛОК ВСТАВЬ СЮДА ---
```

```
    //    case 5: {
    //        int v, e;
    //        cout << "Количество вершин: "; cin >> v;
    //        if (G) delete G;
    //        G = new Graph(v);

    //        cout << "Количество ребер: "; cin >> e;
    //        cout << "Вводите пары вершин (например: 0 1):" << endl;
    //        for (int i = 0; i < e; ++i) {
    //            int u, v_node;
    //            cin >> u >> v_node;
    //            G->addEdge(u, v_node);
    //        }
    //        cout << "Граф создан вручную." << endl;
    //        break;
    //    }
    //          // ------------------------------------
    //    }
    //}

    if (G) delete G;
    return 0;
}
```