Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%205

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 26.03.2025

Оглавление

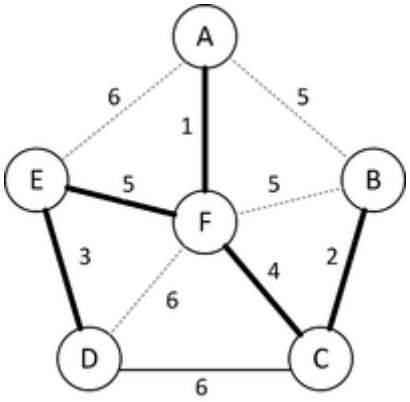
Описание задачи.

1. Создайте взвешенный граф, состоящий из [10, 20, 50, 100] вершин.
   - Каждая вершина графа связана со случайным количеством вершин, минимум с [3, 4, 10, 20].
   - Веса ребер задаются случайным значением от 1 до 20.
   - Каждая вершина графа должна быть доступна, т.е. до каждой вершины графа должен обязательно существовать путь до каждой вершины, не обязательно прямой.
2. Выведите получившийся граф в виде матрицы смежности.
3. Для каждого графа требуется провести серию из 5 - 10 тестов, в зависимости от времени, затраченного на выполнение одного теста, необходимо построить минимальное остовное дерево взвешенного связного неориентированного графа с помощью алгоритма Прима.
4. В рамках каждого теста, необходимо замерить потребовавшееся время на выполнение задания из пункта 3 для каждого набора вершин. По окончанию всех тестов необходимо построить график используя полученные замеры времени, где на ось абсцисс (X) нанести N – количество вершин, а на ось ординат(Y) - значения затраченного времени.

Описание взвешенного графа и остовного дерева.

Взвешенный граф — это граф, в котором рёбра имеют "вес", то есть каждому ребру приписано какое-то числовое значение (чаще всего, это стоимость или длина ребра). Например, если ребра графа представляют дороги между городами, то вес ребра может означать расстояние или время, необходимое для того, чтобы пройти этот путь.

Остовное дерево (или дерево покрытия) — это подмножество рёбер связанного графа, которое соединяет все вершины графа и при этом не образует циклов. То есть, остовное дерево — это дерево, в котором все вершины графа соединены, и ни одна вершина не имеет более одного пути до других вершин.



Алгоритм Прима — это один из алгоритмов для нахождения минимального остовного дерева взвешенного графа. Он работает по принципу жадного алгоритма, то есть на каждом шаге выбирает рёбра, которые приводят к минимальному увеличению стоимости, чтобы получить дерево, соединяющее все вершины.

Алгоритм:

1. Начинаем с произвольной вершины и ищем минимальное ребро, которое соединяет эту вершину с другой.
2. Добавляем это ребро в остовное дерево.
3. Затем повторяем процесс: выбираем минимальное ребро, которое соединяет уже выбранные вершины с остальными невыбранными вершинами.
4. Повторяем, пока все вершины не окажутся в остовном дереве.

Алгоритм Прима для поиска минимального остовного дерева имеет сложность порядка $O(E\log V)$, где V — количество вершин в графе, E — количество рёбер.

При худшем случае может деградировать до $O(V^2)$

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <climits>
#include <chrono>
#include <fstream>
#include <queue>

using namespace std;
```

- Подключаемые библиотеки
    - <iostream> — вывод результатов в консоль.
    - <vector> — хранение и работа с матрицы смежности графа.
    - <cstdlib> — генерация случайных чисел
    - <ctime> — инициализация генератора случайных чисел с использованием текущего времени.
    - <climits> — определение максимальных значений
    - <chrono> — измерение времени.
    - <fstream> — работа с файлом
    - <queue> — очередь для поиска в ширину.
- using namespace — упрощение кода, конфликта в данном проекте нет.

```
// Массив с возможными значениями связей для каждой вершины
vector<int> possibleConnections = {3, 4, 10, 20};

// Функция для проверки связности графа с использованием BFS
bool isConnected(const vector<vector<int>>& graph, int N) {
    vector<bool> visited(N, false);
    queue<int> q;
    q.push(0);
    visited[0] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < N; ++v) {
            if (graph[u][v] != 0 && !visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
```

```
    }

    // Если все вершины были посещены, то граф связан
    for (bool v : visited) {
        if (!v) return false;
    }

    return true;
}
```

- Функция isConnected
    - **Вход**: Матрица смежности graph (двумерный вектор) и количество вершин N.
    - **Что делает**: Проверяет, является ли граф связным, используя алгоритм поиска в ширину (BFS). Если все вершины можно достичь друг от друга, граф связан.
    - **Выход**: Возвращает true, если граф связан, иначе false.

```
// Функция для добавления ребер, чтобы граф стал связным
void ensureConnectedGraph(vector<vector<int>>& graph, int N) {
    // Если граф не связан, добавляем дополнительные ребра
    if (!isConnected(graph, N)) {
        for (int i = 0; i < N; ++i) {
            for (int j = i + 1; j < N; ++j) {
                if (graph[i][j] == 0) {
                    int weight = rand() % 20 + 1;  // случайный вес ребра от 1 до 20
                    graph[i][j] = weight;
                    graph[j][i] = weight;
                }
            }
        }
    }
}
```

- Функция ensureConnectedGraph
    - Вход: Матрица смежности graph и количество вершин N.
    - Что делает: Если граф не связан, добавляет случайные рёбра, чтобы граф стал связным. Использует isConnected для проверки связности.
    - Выход: Модифицирует граф, добавляя рёбра, чтобы сделать его связным.

```
// Функция для создания графа с N вершинами
vector<vector<int>> generateGraph(int N) {
    vector<vector<int>> graph(N, vector<int>(N, 0));

    // Сначала создаем случайные ребра для связности
    for (int i = 0; i < N; ++i) {
        // Для каждой вершины выбираем случайное количество связей из массива
        int numConnections = possibleConnections[rand() % possibleConnections.size()];
```

```
        // Для отслеживания уже связанных вершин
        vector<int> connected(N, 0);

        //Вершина не соединяется с самой собой
        if (numConnections >= N - 1) numConnections = N - 1;  // Не можем
соединить с количеством вершин больше, чем есть

        for (int j = 0; j < numConnections; ++j) {
            int randVertex = rand() % N;
            if (randVertex != i && !connected[randVertex]) {
                connected[randVertex] = 1;
                int weight = rand() % 20 + 1;  // случайный вес ребра от 1 до
20
                graph[i][randVertex] = weight;
                graph[randVertex][i] = weight;  // граф неориентированный
            }
        }
    }

    // Теперь проверим, является ли граф связным, и если нет, добавим ребра
для связности
    ensureConnectedGraph(graph, N);

    return graph;
}
```

- Функция generateGraph
  - **Вход**: Количество вершин N.
  - **Что делает**: Генерирует случайный граф с N вершинами, где для каждой вершины случайным образом выбирается количество рёбер из возможных значений, заданных в possibleConnections. Также проверяет, является ли граф связным, и если нет, добавляет рёбра.
  - **Выход**: Возвращает матрицу смежности для сгенерированного графа.

```
// Алгоритм Прима для поиска минимального остовного дерева
int primAlgorithm(const vector<vector<int>>& graph) {
    int N = graph.size();
    vector<bool> inMST(N, false);  // Вершины в минимальном остовном дереве
    vector<int> key(N, INT_MAX);    // Ключи для вершин
    key[0] = 0;
    int totalWeight = 0;

    for (int count = 0; count < N; ++count) {
        // Выбираем вершину с минимальным ключом, которая еще не включена
        int u = -1;
        for (int i = 0; i < N; ++i) {
            if (!inMST[i] && (u == -1 || key[i] < key[u])) {
                u = i;
            }
        }
```

```
        inMST[u] = true;
        totalWeight += key[u];

        // Обновляем ключи для соседей выбранной вершины
        for (int v = 0; v < N; ++v) {
            if (graph[u][v] != 0 && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
            }
        }
    }

    return totalWeight;
}
```

- Функция primAlgorithm
    - **Вход**: Матрица смежности graph для графа.
    - **Что делает**: Реализует алгоритм Прима для нахождения минимального остовного дерева в графе. Алгоритм выбирает рёбра с минимальными весами, чтобы соединить все вершины в единую связную компоненты.
    - **Выход**: Возвращает сумму весов рёбер в минимальном остовном дереве.

```
// Функция для вывода матрицы смежности
void printAdjacencyMatrix(const vector<vector<int>>& graph, int N) {
    cout << "Матрица смежности для графа с " << N << " вершинами:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cout << graph[i][j] << " ";
        }
        cout << "\n";
    }
    cout << "\n";
}
```

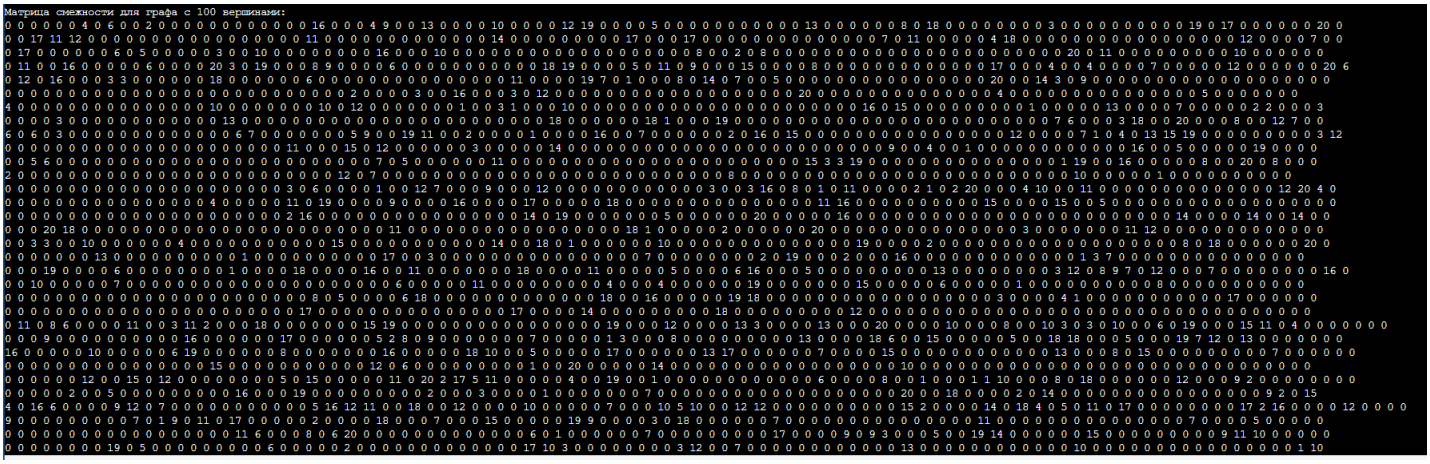
- Функция printAdjacencyMatrix
    - **Вход**: Матрица смежности graph и количество вершин N.
    - **Что делает**: Выводит матрицу смежности на экран.
    - **Выход**: Ничего не возвращает, только выводит данные в консоль.

```
int main() {
    srand(time(0));  // Инициализация генератора случайных чисел

    vector<int> vertexCounts = {10, 20, 50, 100};  // Количество вершин
    ofstream outputFile("results.txt");  // Открытие файла для записи

    // Заголовок для файла
    outputFile << "Vertices|TRY|TIME(ns)|Weight\n";

    // Тестируем для разных размеров графа
```

```
    for (int N : vertexCounts) {
        // Проводим серию из 5-10 тестов
        int numTests = 10;  // Вы можете увеличить до 10 для более точных результатов
        for (int test = 1; test <= numTests; ++test) {
            // Создаем новый граф для каждого теста
            vector<vector<int>> graph = generateGraph(N);
            // Выводим матрицу смежности в консоль
            printAdjacencyMatrix(graph, N);
            // Запуск алгоритма Прима и замер времени
            auto start = chrono::high_resolution_clock::now();
            int totalWeight = primAlgorithm(graph);
            auto end = chrono::high_resolution_clock::now();
            // Замер времени в наносекундах
            chrono::duration<double> duration = end - start;
            auto nanoseconds = chrono::duration_cast<chrono::nanoseconds>(duration).count();
            // Записываем результаты в файл
            outputFile << N << "|" << test << "|" << nanoseconds << "|" << totalWeight << "\n";
        }
    }

    outputFile.close();  // Закрытие файла

    cout << "Результаты записаны в файл results.txt.\n";

    return 0;
}
```

- Функция main
    - Инициализирует генератор случайных чисел с текущим временем.
    - Создаёт графы с различным количеством вершин (от 10 до 100) и для каждого размера графа проводит несколько тестов.
    - Для каждого теста генерирует новый граф, выводит его матрицу смежности, запускает алгоритм Прима для нахождения минимального остовного дерева и измеряет время его выполнения.
    - Результаты тестов записываются в файл results.txt в формате "Количество вершин | Номер теста | Время (нс) | Сумма весов остовного дерева".

Пример вывода матрицы смежности


Матрица смежности для графа с 100 вершинами:

Заключение.

Результаты работы были представлены в виде таблицы значений.

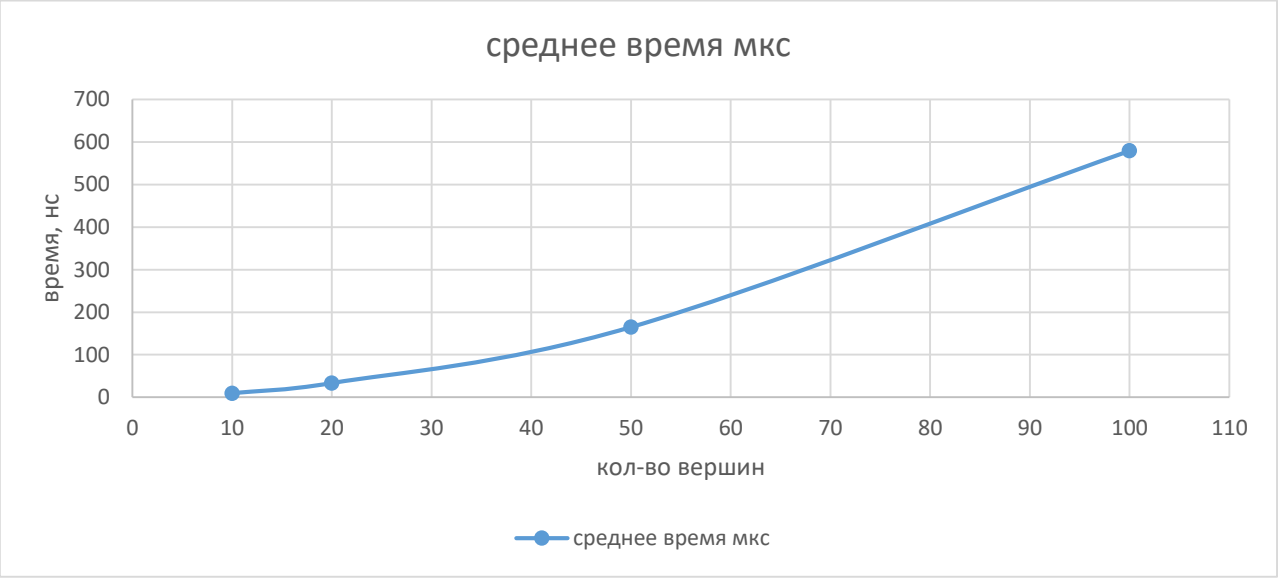
кол-во вершин	среднее время мкс
10	9,3518
20	33,1187
50	164,7343
100	579,5282

А затем по ним построен график. По нему можно сделать следующие выводы:

При увеличении количества вершин в графе:

- Время работы будет увеличиваться с увеличением числа вершин.
- В зависимости от плотности графа и случайных связей, графы с большим количеством вершин будут иметь больше рёбер, что увеличит сложность поиска минимального остовного дерева, а следовательно, и время работы.

Таким образом, если граф более плотный (с большими числами рёбер), то время работы алгоритма будет расти. Экспериментально подтверждена сложность алгоритма.



Поведение графика позволяет заключить, что алгоритм Прима достаточно эффективен для средних размеров графов, но можно предположить, что для очень крупных и плотных графов время работы может стать более значительным.