Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

Задание

1. Создайте взвешенный граф, состоящий из [10, 20, 50, 100] вершин.
    a. Каждая вершина графа связана со случайным количеством вершин, минимум с [3, 4, 10, 20].
    b. Веса ребер задаются случайным значением от 1 до 20.
    c. 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. Пример вывода данных: Матрица 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 время, потребовавшееся на выполнение задания из пункта 3, для каждого набора вершин. По окончании 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 алгоритма

Под **графом** в математике понимается абстракция реальной системы объектов безотносительно их природы, обладающих парными связями. **Вершина графа** — это некоторая точка, связанная с другими точками. **Ребро графа** — это линия, соединяющая две точки и олицетворяющая связь между ними.

Граф — это множество вершин, соединенных друг с другом произвольным образом множеством ребер

Что описывает граф:

- Взаимоотношение между людьми (социальные связи).

- Иерархические отношения (подчиненность людей, подразделений и прочего).
- Пути перемещения в любой местности (карта метро, сеть дорог).
- Взаимозависимости поставщиков услуг или товаров (поставщики для сборки одного автомобиля).
- Распределенные системы (любая микросервисная архитектура).
- Распределенные данные (реляционные базы данных).

Ориентированный граф — это граф, в котором каждое ребро имеет направление движения и, как правило, не предполагает возможности обратного перемещения. Направление ребер указывается стрелками.

Неориентированный граф — это граф, в котором ребра не имеют направления. Каждое ребро соединяет две вершины без указания порядка, и перемещение между ними возможно в обе стороны.

Путь (или маршрут) в графе — это последовательность вершин и ребер от начальной точки до конечной точки графа. Количество ребер, входящих в последовательность, задающую этот путь, называется **длиной пути**. **Циклический путь** — это путь, у которого начало совпадает с концом, и при этом его длина не равна нулю.

Матрица смежности — это двумерная таблица, в которой строки и столбцы соответствуют вершинам, а значения в таблице соответствуют ребрам. Для невзвешенного графа значения могут быть равны 1 (если связь есть и направлена в нужную сторону) или 0 (если связи нет). Для взвешенного графа в ячейках указываются конкретные значения весов.

Связный граф — это граф, в котором существует путь между любой парой вершин. Формально, для неориентированного графа $G = (V, E)$, где V — множество вершин, а E — множество ребер, граф называется связным, если для любых двух вершин $u, v \in V$ существует последовательность ребер (путь), соединяющая u и v. Иными словами, граф состоит из одной компоненты связности, и не существует

вершин, изолированных от остальных. Для проверки на связность в программе используется BFS.

BFS (Breadth-First Search, поиск в ширину) следует концепции «расширяйся, поднимаясь на высоту птичьего полета» («go wide, bird's eye-view»). Вместо того чтобы двигаться по одному пути до конца, BFS предполагает посещение ближайших соседей начальной вершины за один шаг, затем посещение соседей соседей и так далее, пока не будет найдена искомая вершина.

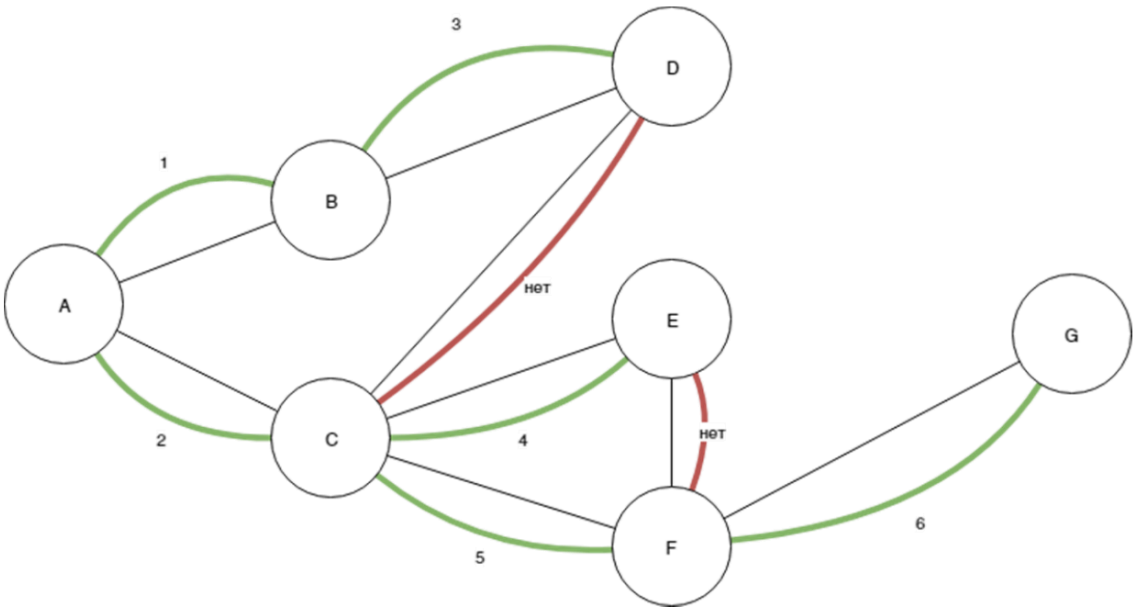

Рисунок 2 - Алгоритм поиска в ширину BFS

Алгоритм поиска в ширину (BFS):

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развернутый.
    a. Если узел u является целевым узлом, завершить поиск с результатом «успех».
    b. В противном случае в конец очереди добавляются все преемники узла u, которые еще не развернуты и не находятся в очереди.
3. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».

4. Вернуться к шагу 2.

Сложность алгоритма также определяется количеством вершин и ребер в графе. Процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

Минимальное остовное дерево (или минимальное покрывающее дерево) в (неориентированном) связном взвешенном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него ребер. Пример минимального остовного дерева представлен на рисунке 1.

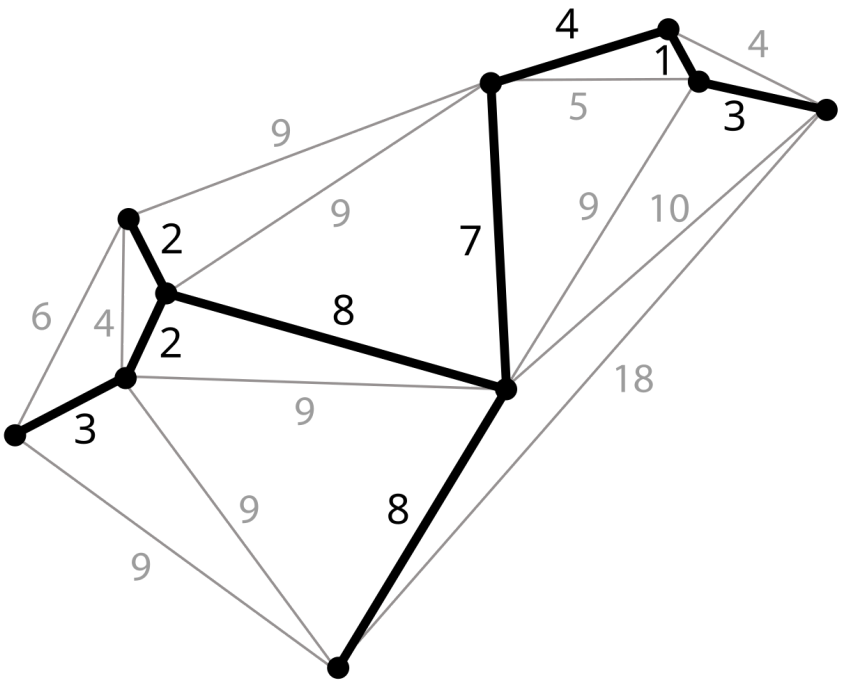

Рис. 1 - Пример минимального остовного дерева

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

Алгоритм:

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

Алгоритм Прима для поиска минимального остова имеет сложность порядка $O(E \cdot logV)$, где V — количество вершин в графе, E — количество ребер. Или в худшем случае может деградировать до $O(V^2)$.

Описание выполнения задачи

Для реализации задачи была написана программа на C++:

```
#include <iostream>
#include <vector>
#include <random>
#include <queue>
#include <algorithm>
#include <iomanip>
#include <chrono>
#include <climits>
#include <fstream>

using namespace std;

// Класс WeightedGraph представляет неориентированный взвешенный граф
// и реализует алгоритм Прима для построения минимального остовного дерева (MST).
class WeightedGraph {
private:
    int num_vertices; // Количество вершин в графе
    int min_edges;    // Минимальное количество ребер, исходящих из каждой вершины
    vector<vector<int>> adj_matrix; // Матрица смежности для хранения весов ребер
    mt19937 rng;      // Генератор случайных чисел (Mersenne Twister)
```

```
// Метод ensure_connectivity гарантирует, что граф связный.
// Использует BFS для проверки достижимости всех вершин и добавляет ребра, если граф несвязный.
void ensure_connectivity() {
    while (true) {
        // Инициализируем массив посещенных вершин
        vector<bool> visited(num_vertices, false);
        queue<int> q; // Очередь для BFS
        q.push(0);    // Начинаем обход с вершины 0
        visited[0] = true;

        // Выполняем BFS для проверки достижимости вершин
        while (!q.empty()) {
            int vertex = q.front(); // Берем вершину из начала очереди
            q.pop();
            // Проверяем все соседние вершины
            for (int i = 0; i < num_vertices; ++i) {
                // Если есть ребро (вес > 0) и вершина не посещена, добавляем ее в очередь
                if (adj_matrix[vertex][i] > 0 && !visited[i]) {
                    visited[i] = true;
                    q.push(i);
                }
            }
        }

        // Проверяем, все ли вершины были посещены
        bool all_reachable = true;
        for (int i = 0; i < num_vertices; ++i) {
            if (!visited[i]) { // Если вершина i недостижима
                all_reachable = false;
                // Собираем список достижимых вершин
                vector<int> reachable;
                for (int j = 0; j < num_vertices; ++j) {
                    if (visited[j]) reachable.push_back(j);
```

```
                }
                // Случайно выбираем достижимую вершину для соединения с недостижимой
                uniform_int_distribution<int> dist(0, reachable.size() - 1);
                int connect_to = reachable[dist(rng)];
                // Генерируем случайный вес ребра от 1 до 20
                uniform_int_distribution<int> weight_dist(1, 20);
                int weight = weight_dist(rng);
                // Добавляем ребро между вершинами connect_to и i
                adj_matrix[connect_to][i] = weight;
                    adj_matrix[i][connect_to] = weight; // Граф неориентированный, добавляем
симметрично
                break; // Прерываем цикл, чтобы заново запустить BFS
              }
            }

            // Если все вершины достижимы, выходим из цикла
            if (all_reachable) break;
          }
        }

        // Метод count_edges подсчитывает количество ребер в графе.
        // Возвращает число ребер, учитывая, что граф неориентированный (каждое ребро учтено
дважды в матрице).
        [[nodiscard]] int count_edges() const {
          int edges = 0;
          // Проходим по всей матрице смежности
          for (int i = 0; i < num_vertices; ++i) {
            for (int j = 0; j < num_vertices; ++j) {
              if (adj_matrix[i][j] > 0) { // Если есть ребро
                edges++;
              }
            }
          }
          return edges / 2; // Делим на 2, так как каждое ребро учтено дважды (i, j и j, i)
        }
```

```
    // Метод is_connected проверяет, является ли граф связным.
    // Использует BFS для обхода графа и проверки достижимости всех вершин.
    [[nodiscard]] bool is_connected() const {
        vector<bool> visited(num_vertices, false); // Массив посещенных вершин
        queue<int> q; // Очередь для BFS
        q.push(0);    // Начинаем с вершины 0
        visited[0] = true;

        // Выполняем BFS
        while (!q.empty()) {
            int vertex = q.front();
            q.pop();
            // Проверяем соседей текущей вершины
            for (int i = 0; i < num_vertices; ++i) {
                if (adj_matrix[vertex][i] > 0 && !visited[i]) {
                    visited[i] = true;
                    q.push(i);
                }
            }
        }

        // Проверяем, все ли вершины были посещены
        return all_of(visited.begin(), visited.end(), [](bool v) { return v; });
    }

public:
    // Конструктор класса WeightedGraph.
    // Принимает количество вершин и минимальное количество ребер на вершину.
    WeightedGraph(int vertices, int min_e)
        : num_vertices(vertices), min_edges(min_e), rng(random_device{}()) {
        // Инициализируем матрицу смежности нулями (нет ребер)
        adj_matrix = vector<vector<int>>(num_vertices, vector<int>(num_vertices, 0));
        generate_graph();    // Генерируем случайный граф
        ensure_connectivity(); // Убеждаемся, что граф связный
```

```
}

// Метод generate_graph генерирует случайный неориентированный взвешенный граф.
 // Каждая вершина соединяется со случайным количеством других вершин (от min_edges до 2*min_edges).
void generate_graph() {
    for (int i = 0; i < num_vertices; ++i) {
        // Собираем список возможных вершин для соединения (все, кроме i)
        vector<int> possible_connections;
        for (int j = 0; j < num_vertices; ++j) {
            if (j != i) possible_connections.push_back(j);
        }
        // Определяем случайное количество ребер для вершины i
            uniform_int_distribution<int> conn_dist(min_edges, min(num_vertices-1, min_edges * 2));
        int num_connections = conn_dist(rng);
        // Перемешиваем возможные соединения, чтобы выбирать случайные вершины
        shuffle(possible_connections.begin(), possible_connections.end(), rng);

        // Добавляем ребра к выбранным вершинам
        for (int j = 0; j < num_connections && j < possible_connections.size(); ++j) {
            int target = possible_connections[j];
            if (adj_matrix[i][target] == 0) { // Если ребра еще нет
                // Генерируем случайный вес от 1 до 20
                uniform_int_distribution<int> weight_dist(1, 20);
                int weight = weight_dist(rng);
                adj_matrix[i][target] = weight;
                adj_matrix[target][i] = weight; // Граф неориентированный
            }
        }
    }
}

// Метод print_adj_matrix выводит матрицу смежности графа в консоль.
// Также выводит отладочную информацию: количество ребер и связность графа.
```

```
    void print_adj_matrix() const {
        cout << "Adjacency matrix for graph with " << num_vertices << " vertices:\n";
        cout << "Debug info: Number of edges = " << count_edges()
             << ", Connected: " << (is_connected() ? "Yes" : "No") << "\n";

        // Выводим заголовок с номерами столбцов
        cout << "    ";
        for (int j = 0; j < num_vertices; ++j) {
            cout << setw(3) << j << " ";
        }
        cout << "\n";
        // Выводим разделительную линию
        cout << "   " << string(num_vertices * 4, '-') << "\n";

        // Выводим строки матрицы
        for (int i = 0; i < num_vertices; ++i) {
            cout << setw(2) << i << " |";
            for (int j = 0; j < num_vertices; ++j) {
                cout << setw(3) << adj_matrix[i][j] << " ";
            }
            cout << "\n";
        }
    }
```

// Метод prim_mst реализует алгоритм Прима для построения минимального остовного дерева (MST).

```
    // Возвращает вектор ребер MST в формате пар (u, v).
    // Сложность: O(V^2), где V — количество вершин.
    [[nodiscard]] vector<pair<int, int>> prim_mst() const {
        vector<bool> in_mst(num_vertices, false); // Флаг: включена ли вершина в MST
         vector<int> key(num_vertices, INT_MAX);   // Минимальный вес ребра, соединяющего вершину с MST
        vector<int> parent(num_vertices, -1);     // Родитель вершины в MST
        vector<pair<int, int>> mst_edges;         // Список ребер MST
```

```
    key[0] = 0; // Начинаем с вершины 0, ее ключ — 0

    // Проходим по всем вершинам, чтобы включить их в MST
    for (int count = 0; count < num_vertices; ++count) {
        // Ищем вершину с минимальным ключом, которая еще не в MST
        int u = -1;
        for (int v = 0; v < num_vertices; ++v) {
            if (!in_mst[v] && (u == -1 || key[v] < key[u])) {
                u = v;
            }
        }

        in_mst[u] = true; // Добавляем вершину u в MST

        // Если у вершины есть родитель, добавляем ребро (parent[u], u) в MST
        if (parent[u] != -1) {
            mst_edges.emplace_back(parent[u], u);
        }

        // Обновляем ключи и родителей для соседних вершин
        for (int v = 0; v < num_vertices; ++v) {
            // Если есть ребро (u, v), вершина v не в MST и вес ребра меньше текущего ключа
v
            if (adj_matrix[u][v] > 0 && !in_mst[v] && adj_matrix[u][v] < key[v]) {
                key[v] = adj_matrix[u][v]; // Обновляем ключ
                parent[v] = u;             // Устанавливаем родителя
            }
        }
    }

    return mst_edges; // Возвращаем ребра MST
}

// Метод print_mst выводит ребра минимального остовного дерева и его общий вес.
void print_mst() const {
```

```
        vector<pair<int, int>> mst = prim_mst(); // Получаем ребра MST
        cout << "\nMinimum Spanning Tree edges:\n";
        int total_weight = 0; // Общий вес MST
        // Проходим по всем ребрам MST
        for (const auto& edge : mst) {
            int u = edge.first;
            int v = edge.second;
            cout << "(" << u << ", " << v << ") weight: " << adj_matrix[u][v] << "\n";
            total_weight += adj_matrix[u][v]; // Суммируем веса ребер
        }
        cout << "Total MST weight: " << total_weight << "\n";
    }

    // Метод measure_prim_time измеряет время выполнения алгоритма Прима.
    // Возвращает время в микросекундах.
    [[nodiscard]] double measure_prim_time() const {
        auto start = chrono::high_resolution_clock::now(); // Засекаем начальное время
            static_cast<void>(prim_mst()); // Вызываем prim_mst, игнорируем результат (нужен только для замера времени)
        auto end = chrono::high_resolution_clock::now(); // Засекаем конечное время
        // Вычисляем разницу времени в микросекундах
        return chrono::duration<double, micro>(end - start).count();
    }
};

// Главная функция программы
int main() {
    // Массивы с параметрами для тестов: количество вершин и минимальное количество ребер
    int vertex_counts[] = {10, 20, 50, 100};
    int min_edges[] = {3, 4, 10, 20};
    const int num_tests = 10; // Количество тестов для каждого графа

    cout << "Undirected graphs - Prim's MST Performance Tests:\n";
    vector<vector<double>> test_results(4); // Массив для хранения результатов тестов
```

```
// Проходим по всем размерам графа
for (int i = 0; i < 4; ++i) {
    cout << "\nGraph " << i+1 << " (" << vertex_counts[i] << " vertices):\n";
    // Создаем граф с заданным количеством вершин и минимальным количеством ребер
    WeightedGraph graph(vertex_counts[i], min_edges[i]);
    graph.print_adj_matrix(); // Выводим матрицу смежности
    graph.print_mst();        // Выводим MST

    // Проводим тесты производительности
    cout << "\nRunning " << num_tests << " tests for Prim's MST...\n";
    test_results[i].resize(num_tests); // Инициализируем массив для результатов
    for (int t = 0; t < num_tests; ++t) {
        test_results[i][t] = graph.measure_prim_time(); // Замеряем время
        // Выводим результат теста с точностью до 3 знаков после запятой
        cout << "Test " << t+1 << ": " << fixed << setprecision(3)
             << test_results[i][t] << " mks\n";
    }

    // Вычисляем среднее время выполнения
    double avg_time = 0;
    for (double time : test_results[i]) {
        avg_time += time;
    }
    avg_time /= num_tests;
    cout << "Average time: " << fixed << setprecision(3) << avg_time << " mks\n";
}

// Сохраняем результаты в CSV-файл
ofstream csv_file("prim_performance.csv");
if (!csv_file.is_open()) {
    cerr << "Error opening CSV file!" << endl;
    return 1;
}
```

```
    // Записываем заголовок CSV
    csv_file << "N,Average Time (mks)\n";
    for (int i = 0; i < 4; ++i) {
        double avg_time = 0;
        for (double time : test_results[i]) {
            avg_time += time;
        }
        avg_time /= num_tests;
        // Записываем среднее время для каждого размера графа
        csv_file << vertex_counts[i] << "," << fixed << setprecision(3) << avg_time << "\n";
    }
    csv_file.close();
    cout << "\nResults saved to prim_performance.csv\n";

    return 0;
}
```

Этот код представляет собой реализацию на C++ программы для генерации случайного неориентированного взвешенного графа и построения его минимального остовного дерева (MST) с использованием алгоритма Прима. Программа также проводит тесты производительности алгоритма и сохраняет результаты в CSV-файл для дальнейшего анализа.

Основные функции программы:

- Генерация графа: создается случайный неориентированный граф с заданным количеством вершин и минимальным количеством ребер на вершину. Граф представлен в виде матрицы смежности.
- Обеспечение связности: проверяется, является ли граф связным, и при необходимости добавляются ребра для обеспечения связности.
- Построение MST: используется алгоритм Прима для построения минимального остовного дерева.
- Вывод информации: выводится матрица смежности графа, ребра MST и общий вес дерева.

- Измерение производительности: проводятся тесты для измерения времени выполнения алгоритма Прима, результаты сохраняются в CSV-файл.

Специфические элементы реализации:

- Представление графа. Матрица смежности (adj_matrix): Граф представлен в виде матрицы смежности, где adj_matrix[i][j] — это вес ребра между вершинами i и j. Если ребра нет, значение равно 0. Поскольку граф неориентированный, матрица симметрична: adj_matrix[i][j] == adj_matrix[j][i].
- Генерация случайного графа.
    - Метод generate_graph: Для каждой вершины создается случайное количество ребер (от min_edges до 2 * min_edges), соединяющих ее с другими вершинами.
        - Используется std::shuffle для случайного выбора вершин для соединения.
        - Вес ребер генерируется случайным образом в диапазоне [1, 20] с помощью uniform_int_distribution.
    - Генератор случайных чисел. Используется mt19937 (Mersenne Twister) для генерации случайных чисел, что обеспечивает высокое качество случайности.
- Обеспечение связности графа.
    - Метод ensure_connectivity: Использует алгоритм поиска в ширину (BFS) для проверки, является ли граф связным.
        - Начинает обход с вершины 0.
        - Если какая-то вершина недостижима, случайным образом выбирается достижимая вершина, и между ними добавляется ребро с случайным весом.
        - Процесс повторяется до тех пор, пока граф не станет связным.
- Реализация алгоритма Прима. Метод prim_mst: Реализует классический алгоритм Прима для построения минимального остовного дерева.

- Сложность: $O(V^2)$, так как для поиска вершины с минимальным ключом используется линейный поиск.
- Данные:
  - in_mst: Массив флагов, указывающий, включена ли вершина в MST.
  - key: Массив минимальных весов ребер, соединяющих вершину с MST.
  - parent: Массив для хранения родителя каждой вершины в MST.
- Процесс:
  - Начинает с вершины 0, устанавливая ее ключ в 0.
  - На каждой итерации выбирает вершину с минимальным ключом, добавляет ее в MST и обновляет ключи для соседних вершин.
  - Результат: Возвращает вектор ребер MST в формате vector<pair<int, int>>.
- Текущая реализация не использует приоритетную очередь, что делает ее менее эффективной для больших графов. Более эффективная реализация с std::priority_queue имела бы сложность $O(E \cdot logV)$.

- Измерение производительности. Метод measure_prim_time: Измеряет время выполнения алгоритма Прима в микросекундах.
  - Использует chrono::high_resolution_clock для точного замера времени.
  - Возвращает время в микросекундах с помощью chrono::duration<double, micro>.
- Вывод и форматирование.
  - Метод print_adj_matrix: Выводит матрицу смежности в читаемом формате.
    - Выводит заголовок с номерами столбцов.
    - Использует setw из <iomanip> для выравнивания чисел.
    - Выводит отладочную информацию: количество ребер и связность графа.

    - Метод print_mst: Выводит ребра MST и их общий вес. Особенность: Для красивого форматирования используется string(num_vertices * 4, '-') для создания разделительной линии в матрице смежности.
- Сохранение результатов. Функция main: Сохраняет результаты тестов производительности в файл prim_performance.csv.
    - Формат: N,Average Time (ms) — количество вершин и среднее время выполнения.
    - Использует ofstream для записи в файл.
- Тестирование. Функция main: Проводит тесты для графов с разным количеством вершин (10, 20, 50, 100).
    - Для каждого графа выполняется 10 тестов, чтобы получить среднее время.
    - Результаты выводятся в консоль с точностью до 3 знаков после запятой (setprecision(3)).
- Генерация случайных чисел. Генератор mt19937: Используется для генерации случайных чисел высокого качества.
    - Распределения: uniform_int_distribution для выбора количества ребер, весов и вершин для соединения.
    - Инициализация генератора через random_device для получения истинно случайного начального значения.

Выводы

В ходе выполнения лабораторной работы была реализована генерация случайных неориентированных взвешенных графов с заданным количеством вершин и минимальным количеством ребер на вершину. Была обеспечена связность графа с использованием алгоритма поиска в ширину (BFS), что позволило гарантировать корректность работы алгоритма Прима. Реализована возможность вывода матрицы смежности графа, а также ребер минимального остовного дерева (MST) и его общего веса.

На основании генерации четырех графов с количеством вершин 10, 20, 50 и 100 был применен алгоритм Прима для построения минимального остовного дерева. Для каждого графа было проведено 10 тестов производительности, в результате чего были измерены времена выполнения алгоритма в микросекундах (табл. 1).

N	Average Time (μs)
10	17,38
20	49,43
50	178,76
100	686,26

Таблица 1 - Полученные в результате работы программы данные о времени выполнения алгоритма Прима для построения минимального остовного дерева

Был построен графики зависимости времени выполнения алгоритма Прима для построения минимального остовного дерева от вершин для ненаправленных (рис. 2) графов при помощи полученных в результате работы программы данных, представленных в таблице 1.

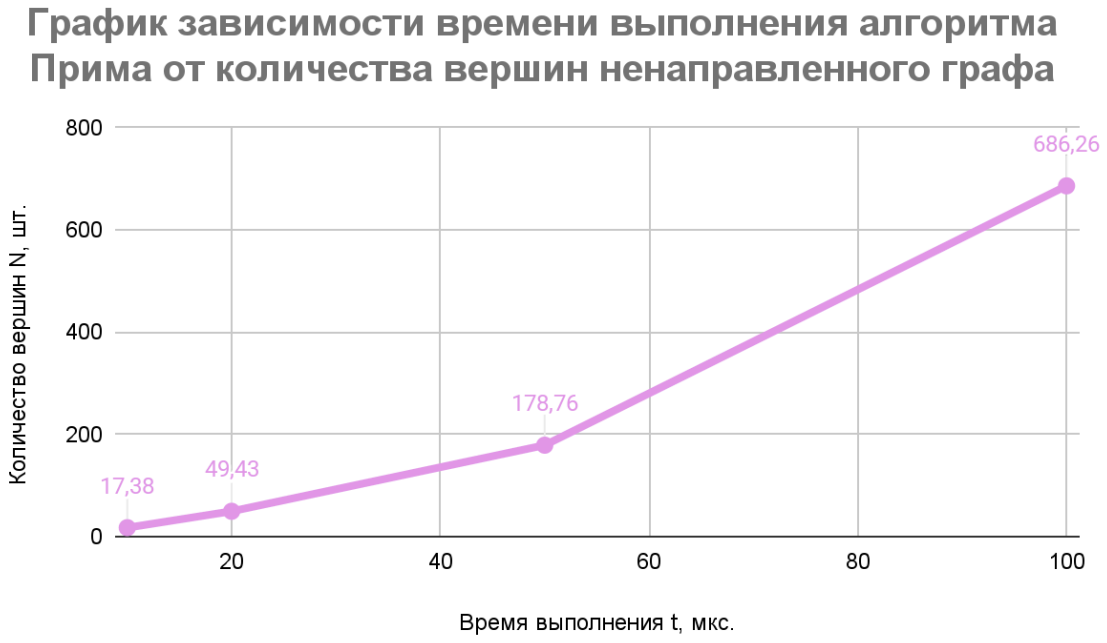

Рисунок 2 - График зависимости времени выполнения алгоритма Прима от вершин для ненаправленных графов

График демонстрирует квадратичный рост времени выполнения, что соответствует теоретической сложности текущей реализации алгоритма Прима $O(V^2)$, где V — количество вершин.

Теоретически текущая реализация алгоритма Прима имеет сложность $O(V^2)$, так как для выбора вершины с минимальным ключом используется линейный поиск. Однако фактический рост времени выполнения оказался ближе к $O(E \cdot logV)$, где E — количество ребер, а V — количество вершин.

Для дальнейшего улучшения производительности стоит использовать приоритетную очередь в реализации алгоритма Прима, что позволит достичь сложности $O(E \cdot logV)$. Это может значительно уменьшить время выполнения, особенно для графов с большим числом вершин и ребер.