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

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 3](#_Toc63548272)

[Описание метода/модели. 3](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 8](#_Toc63548275)

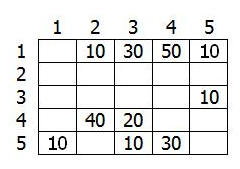
# Описание задачи.

Целью лабораторной работы было создание взвешенного графа, состоящего из [10, 20, 50, 100] вершин, и выполнение алгоритма Краскала для построения минимального остовного дерева. Граф должен быть связным, то есть каждая вершина должна быть доступна из любой другой вершины. Веса ребер задаются случайным значением от 1 до 20. Для каждого графа необходимо провести серию тестов, замерить время выполнения алгоритма Краскала и построить график зависимости времени выполнения от количества вершин.

**Задание: Вариант 4**

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.

(Можно использовать генератор из предыдущей лабораторной работы.)

1. [](https://github.com/MUCTR-IKT-CPP/algorithms-2cpp/blob/master/%D0%9B%D0%B0%D0%B1%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D1%8B%D0%B5/2025/5/%D0%9C%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0%20%D1%81%D0%BC%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8..png)Выведите получившийся граф в виде матрицы смежности. Пример вывода данных:
2. Для каждого графа требуется провести серию из 5 - 10 тестов, в зависимости от времени затраченного на выполнение одного теста., необходимо:
   * **Вариант 4**. Построить минимальное остовное дерево взвешенного связного неориентированного графа с помощью алгоритма Краскала.
3. 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 (Х) нанести N – количество вершин, а на ось ординат(Y) - значения затраченного времени.

# Описание метода/модели.

**Взвешенный граф**

Граф представлен в виде списка ребер, где каждое ребро имеет вес. Для генерации графа используется случайное количество связей для каждой вершины, при этом граф проверяется на связность с помощью алгоритма BFS (поиск в ширину).

**Алгоритм Краскала**

Алгоритм Краскала используется для нахождения минимального остовного дерева в связном неориентированном взвешенном графе. Он работает следующим образом:

1. Сортирует все ребра графа по весу.
2. Добавляет ребра в остовное дерево, начиная с ребра с наименьшим весом, и проверяет, не образует ли добавление ребра цикл.
3. Процесс продолжается до тех пор, пока не будут добавлены все необходимые ребра.

До начала работы алгоритма необходимо отсортировать рёбра по весу, это требует O(E × log(E)) времени. После чего компоненты связности удобно хранить в виде системы непересекающихся множеств.

Все операции в таком случае займут O(E × α(E, V)), где α — функция, обратная к функции Аккермана. Поскольку для любых практических задач α(E, V) < 5, то можно принять её за константу, таким образом, общее время работы алгоритма Краскала можно принять за O(E \* log(E)), где E — количество ребер, благодаря сортировке и эффективной реализации

**Проверка связности графа**

Граф называется связным, если из любой вершины можно добраться до любой другой вершины по ребрам. Если граф не связный, это означает, что в нем есть изолированные компоненты (группы вершин, которые не связаны с остальными).

Алгоритм Краскала (который используется для нахождения минимального остовного дерева) работает только для связных графов. Поэтому важно убедиться, что сгенерированный граф связный.

Для проверки связности графа используется алгоритм BFS, который обходит все вершины графа, начиная с произвольной вершины. Если все вершины были посещены, граф связный.

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <random>  
#include <chrono>  
#include <iomanip>  
#include <queue>  
#include <fstream> // Добавлен заголовочный файл для работы с файлами  
  
using namespace std;  
  
// Структура для хранения ребра графа  
struct Edge {  
 int u, v, weight;  
 bool operator<(Edge const& other) {  
 return weight < other.weight;  
 }  
};  
  
// Класс для представления графа  
class Graph {  
public:  
 int V; // Количество вершин  
 vector<Edge> edges; // Список ребер  
  
 Graph(int vertices) : V(vertices) {}  
  
 // Добавление ребра в граф  
 void addEdge(int u, int v, int weight) {  
 edges.push\_back({u, v, weight});  
 }  
  
 // Генерация случайного графа  
 void generateRandomGraph(int minEdges) {  
 random\_device rd;  
 mt19937 gen(rd());  
 uniform\_int\_distribution<> weightDist(1, 20);  
  
 for (int u = 0; u < V; ++u) {  
 int numEdges = minEdges + gen() % (V - minEdges);  
 for (int i = 0; i < numEdges; ++i) {  
 int v = gen() % V;  
 if (u != v) {  
 addEdge(u, v, weightDist(gen));  
 }  
 }  
 }  
  
 // Проверка связности графа  
 while (!isConnected()) {  
 edges.clear();  
 generateRandomGraph(minEdges);  
 }  
 }  
  
 // Проверка связности графа с помощью BFS  
 bool isConnected() {  
 vector<vector<int>> adjList(V);  
 for (auto& edge : edges) {  
 adjList[edge.u].push\_back(edge.v);  
 adjList[edge.v].push\_back(edge.u);  
 }  
  
 vector<bool> visited(V, false);  
 queue<int> q;  
 q.push(0);  
 visited[0] = true;  
  
 while (!q.empty()) {  
 int u = q.front();  
 q.pop();  
 for (int v : adjList[u]) {  
 if (!visited[v]) {  
 visited[v] = true;  
 q.push(v);  
 }  
 }  
 }  
  
 for (bool v : visited) {  
 if (!v) return false;  
 }  
 return true;  
 }  
  
 // Вывод матрицы смежности  
 void printAdjacencyMatrix() {  
 vector<vector<int>> matrix(V, vector<int>(V, 0));  
 for (auto& edge : edges) {  
 matrix[edge.u][edge.v] = edge.weight;  
 matrix[edge.v][edge.u] = edge.weight;  
 }  
  
 cout << " ";  
 for (int i = 0; i < V; ++i) {  
 cout << setw(3) << i + 1 << " ";  
 }  
 cout << endl;  
  
 for (int i = 0; i < V; ++i) {  
 cout << setw(2) << i + 1 << " ";  
 for (int j = 0; j < V; ++j) {  
 cout << setw(3) << matrix[i][j] << " ";  
 }  
 cout << endl;  
 }  
 }  
};  
  
// Класс для реализации алгоритма Краскала  
class Kruskal {  
public:  
 // Нахождение минимального остовного дерева  
 static vector<Edge> findMST(Graph& graph) {  
 vector<Edge> result;  
 sort(graph.edges.begin(), graph.edges.end());  
  
 vector<int> parent(graph.V);  
 for (int i = 0; i < graph.V; ++i) {  
 parent[i] = i;  
 }  
  
 for (Edge& edge : graph.edges) {  
 int u = findParent(parent, edge.u);  
 int v = findParent(parent, edge.v);  
 if (u != v) {  
 result.push\_back(edge);  
 parent[u] = v;  
 }  
 }  
  
 return result;  
 }  
  
private:  
 // Нахождение корня родителя для вершины  
 static int findParent(vector<int>& parent, int i) {  
 if (parent[i] == i) {  
 return i;  
 }  
 return findParent(parent, parent[i]);  
 }  
};  
  
// Функция для выполнения тестов  
void runTests() {  
 vector<int> vertices = **{**10, 20, 50, 100**}**;  
 vector<int> minEdges = **{**3, 4, 10, 20**}**;  
 int numTests = 5; // Количество тестов для каждого графа  
  
 ofstream outFile("results.txt"); // Открываем файл для записи результатов  
  
 vector<pair<int, double>> results; // Вектор для хранения результатов (вершины, среднее время)  
  
 for (size\_t i = 0; i < vertices.size(); ++i) {  
 int V = vertices[i];  
 int minEdge = minEdges[i];  
 double totalTime = 0;  
  
 for (int test = 0; test < numTests; ++test) {  
 Graph graph(V);  
 graph.generateRandomGraph(minEdge);  
  
 cout << "Graph with " << V << " vertices (Test " << test + 1 << "):" << endl;  
 graph.printAdjacencyMatrix();  
  
 auto start = chrono::high\_resolution\_clock::now();  
 vector<Edge> mst = Kruskal::findMST(graph);  
 auto end = chrono::high\_resolution\_clock::now();  
  
 chrono::duration<double> elapsed = end - start;  
 totalTime += elapsed.count();  
  
 cout << "Time taken for Kruskal's algorithm: " << elapsed.count() << " seconds" << endl;  
 cout << "Minimum Spanning Tree:" << endl;  
 for (Edge& edge : mst) {  
 cout << edge.u + 1 << " - " << edge.v + 1 << " : " << edge.weight << endl;  
 }  
 cout << endl;  
 }  
  
 double averageTime = totalTime / numTests;  
 results.push\_back({V, averageTime}); // Сохраняем результат  
  
 cout << "Average time for " << V << " vertices: " << averageTime << " seconds" << endl;  
  
 // Записываем результаты в файл  
 outFile << V << " " << averageTime << endl;  
 }  
  
 outFile.close(); // Закрываем файл  
  
 // Вывод сводной таблицы  
 cout << "\nSummary Table:\n";  
 cout << "---------------------------------\n";  
 cout << "| Vertices | Average Time (sec) |\n";  
 cout << "---------------------------------\n";  
 for (auto& result : results) {  
 cout << "| " << setw(9) << result.first << " | " << setw(18) << fixed << setprecision(6) << result.second << " |\n";  
 }  
 cout << "---------------------------------\n";  
}  
  
  
int main() {  
 runTests();  
 return 0;  
}

**Результат:**

Summary Table:

---------------------------------

| Vertices | Average Time (sec) |

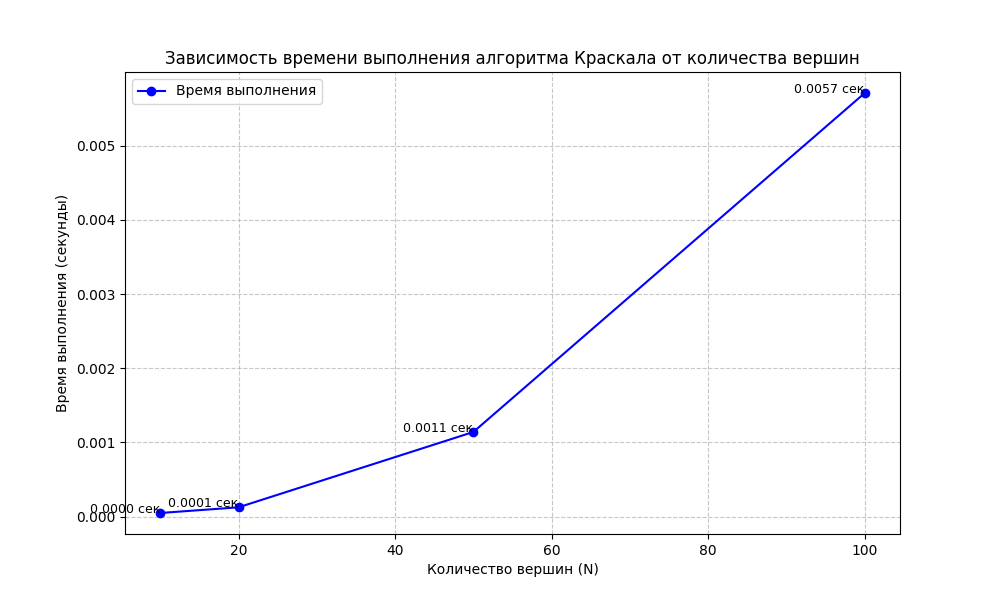
---------------------------------

| 10 | 0.000047 |

| 20 | 0.000124 |

| 50 | 0.001140 |

| 100 | 0.005710 |

**График зависимости:**

1. Время выполнения алгоритма Краскала растет с увеличением количества вершин. Это связано с тем, что алгоритм имеет сложность O(E log E), где E — количество ребер.
2. Графы с большим количеством вершин требуют больше времени для обработки, что подтверждается результатами тестов.
3. Алгоритм Краскала эффективен для нахождения минимального остовного дерева в связных графах, но его производительность снижается с увеличением размера графа.

# Заключение.

В ходе выполнения лабораторной работы был реализован алгоритм Краскала для нахождения минимального остовного дерева в связном взвешенном графе.

Были проведены тесты для графов с разным количеством вершин, и замерено время выполнения алгоритма.

Результаты показали, что время выполнения растет с увеличением количества вершин, что соответствует теоретической сложности алгоритма. Для больших графов (100 вершин и более) время выполнения становится значительным, что необходимо учитывать при работе с большими объемами данных.