МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ИС

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Графы. Алгоритм Краскала

Студент гр. 4372 _____ Климаш И.В.

Преподаватель _____ Пелевин М.С.

Санкт-Петербург

2025

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Климаш И.В.

Группа 4372

Тема работы: Графы. Алгоритм Краскала

Исходные данные:

Любой текстовый файл, содержащий матрицу смежности графа.

Содержание пояснительной записки:

“Содержание”, “Введение”, “Сортировка”, “Обходы графа”, “Построение СНМ”, “Заключение”, “Список использованных источников”, “Приложение А. Код программы”

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 01.12.2025

Дата сдачи реферата: 5.12.2025

Дата защиты реферата: 5.12.2025

Студент _____ Климаш И.В.

Преподаватель _____ Пелевин М.С.

АННОТАЦИЯ

Содержание работы:

В курсовой работе реализован алгоритм Краскала для поиска минимального остова графа. Граф задаётся матрицей смежности из текстового файла. Используются сортировка рёбер по весу, система непересекающихся множеств и методы обхода графов. Программа находит остов с минимальной суммой весов, что подтверждено тестами на различных данных.

Методы исследования:

1. Представление графа в виде матрицы смежности, загружаемой из текстового файла.

2. Сортировка рёбер графа по весам.

3. Построение системы непересекающихся множеств.

4. Пошаговое добавление рёбер в остов, следуя критерию минимального веса и избегая циклов.

Результаты:

Разработан программный модуль, который эффективно реализует алгоритм Краскала. Итогом является минимальный остов графа, представленный списком рёбер с указанием их весов.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача

Реализовать алгоритм поиска минимального остова на основе алгоритма Краскала (Крускала).

Цель

Продемонстрировать знания следующих вопросов:

1. сортировка
2. обход графов (в глубину и в ширину)
3. хранение графов (списки смежности, матрицы смежности, инцидентности)
4. построение системы непересекающихся множеств

Методы решения:

1. Написание алгоритмов сортировки
2. Написание алгоритмов обхода
3. Реализация алгоритма Краскала
4. Реализация структуры “Graph”

1. СОРТИРОВКА

1.1. Выбор алгоритма сортировки

Сортировка вставками была применена для упорядочивания рёбер по имени (для вывода) и по весу (для алгоритма Краскала). Данный выбор обусловлен её эффективностью на небольших наборах данных, что соответствует условиям задачи, где число вершин не превышает 50.

1.2. Описание алгоритма сортировки

Алгоритм сортировки вставками:

Сортировка вставками — это простой алгоритм, который сортирует массив, постепенно выстраивая отсортированную последовательность.

Шаги алгоритма:

1. Начать с первого элемента, считая его уже отсортированным.
2. Для каждого следующего элемента массива:
3. Сравнивать его с элементами отсортированной части.
4. Найти его правильное место, сдвигая элементы вправо.
5. Вставить текущий элемент на найденную позицию.
6. Повторять до конца массива.

2. ОБХОДЫ ГРАФОВ

2.1. Обход в глубину

Обход графа в глубину (DFS) — это алгоритм, который посещает все вершины графа, начиная с указанной вершины и исследуя как можно дальше вдоль каждого пути перед возвратом.

Шаги алгоритма:

1. Начать с выбранной вершины и пометить её как посещённую.
2. Для каждой соседней вершины, которая ещё не посещена, рекурсивно выполнить обход.
3. Повторять шаги 1 и 2 для всех смежных вершин, пока не будут посещены все достижимые вершины.

2.2. Обход в ширину

Обход графа в ширину (BFS) — это алгоритм, который посещает все вершины графа, начиная с указанной вершины, и исследует соседей уровня за уровнем.

Шаги алгоритма:

1. Начать с выбранной вершины и пометить её как посещённую.
2. Поместить начальную вершину в очередь.
3. Пока очередь не пуста:
    a. Извлечь вершину из очереди.
    b.Посетить всех её непосещённых соседей, пометить их как посещённые и добавить в очередь.
4. Повторять шаг 3, пока не будут посещены все вершины.

3. ПОСТРОЕНИЕ СНМ

3.1. Алгоритм Краскала

Алгоритм Краскала относится к классическим методам построения минимального остовного дерева. Его принцип действия основан на жадной стратегии: итеративно выбирая ребро минимального веса из оставшихся, которое не образует цикла с уже выбранными рёбрами, алгоритм формирует искомое дерево, начиная с пустого множества.

Процесс алгоритма Краскала можно описать следующим образом:

1. Сортировка всех ребер графа в порядке возрастания их весов.
2. Создание пустого остовного дерева.
3. Последовательный выбор ребер с наименьшими весами.
4. Добавление выбранного ребра к остовному дереву, при условии, что оно не создаст цикл.
5. Повторение шага 4 до тех пор, пока остовное дерево не будет содержать все вершины графа.

Функция возвращает список рёбер входящих в минимальное остовное дерево.

Код алгоритма:

```
void Graph::FindMinSpanningTree()
{
    Edge edgesList[MAX_EDGES];
    int edgesCount = CollectEdges(edgesList, MAX_EDGES);

    if (edgesCount == 0)
    {
        cout << "No edges in graph\n";
        return;
    }

    InsertionSortByWeight(edgesList, edgesCount);
```

```
int parent[MAX_VERTICES];
for (int i = 0; i < V; i++) parent[i] = -1;

Edge mst[MAX_EDGES];
int mstCount = 0;
int totalWeight = 0;

for (int i = 0; i < edgesCount && mstCount < V - 1; i++)
{
    int x = FindParent(parent, edgesList[i].src);
    int y = FindParent(parent, edgesList[i].dest);

    if (x != y) {
        mst[mstCount++] = edgesList[i];
        totalWeight += edgesList[i].weight;
        UnionParents(parent, x, y);
    }
}

cout << "Edges in MST are:\n";

// сортировка по именам вершин
for (int i = 1; i < mstCount; ++i)
{
    Edge key = mst[i];
    int j = i - 1;

    auto compare = [&](const Edge& a, const Edge& b)
    {
        if (names[a.src] != names[b.src])
            return names[a.src] > names[b.src];
        return names[a.dest] > names[b.dest];
    };

    while (j >= 0 && compare(mst[j], key))
    {
        mst[j + 1] = mst[j];
        j--;
```

```
            }
            mst[j + 1] = key;
        }

        for (int i = 0; i < mstCount; i++)
        {
            cout << names[mst[i].src] << " -- "
                      << names[mst[i].dest] << " ("
                      << mst[i].weight << ")\n";
        }

        cout << "Average Weight = " << totalWeight << endl;
    }
```

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы была разработана и протестирована программная реализация алгоритма Краскала для построения минимального остовного дерева. Алгоритм корректно обрабатывает графы, заданные матрицей смежности, и демонстрирует эффективную работу за счёт применения сортировки рёбер по весу и системы непересекающихся множеств (СНМ) для предотвращения циклов. Все ключевые этапы — от хранения структуры графа и сортировки до реализации СНМ — выполнены в полном объёме.

Таким образом, цель работы — практическое освоение алгоритмов на графах и структур данных — достигнута, что подтверждается успешным тестированием на разнообразных наборах данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кормен Т. Х., Лейзерсон Ч. Э., Ривест Р. Л., Штайн Клиффорд. Введение в алгоритмы. М.: Издательство "Вильямс", 2010. 1312 с.

2. Эпштейн М. С., Анисимов В. С., Шеин И. В., Морозов А. В. Алгоритмы и структуры данных. СПб.: БХВ-Петербург, 2011. 640 с.

3. Седов И. Н., Гусев В. А., Кравчук Ю. В. и др. Алгоритмы и структуры данных: Теория и практика. М.: Наука, 2008. 528 с.

4. Тараненко В. А. Алгоритмы и структуры данных. СПб.: Издательство Питер, 2017. 480 с.

5. Ли Ю. С. Алгоритмы и структуры данных: Введение в анализ и проектирование. СПб.: Питер, 2010. 384 с.

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ

Файл Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <string>

using namespace std;

const int MAX_VERTICES = 50;
const int MAX_EDGES = 50;
const int MAX_EDGES_PER_VERTEX = 10; // макс кол-во ребер из одной
вершины

struct Edge {
    int src;    // источник
    int dest;   // приемник
    int weight;
};

class Graph {
public:

    int V;

    string names[MAX_VERTICES];

    // у каждой вершины есть массив рёбер adj[i]
    // рёбра хранятся в Edge-массивах
    // adjCount[i] — сколько рёбер реально есть
    Edge adj[MAX_VERTICES][MAX_EDGES_PER_VERTEX];
    int adjCount[MAX_VERTICES];

    Graph(int v = 0);

    void SetVertexName(int idx, const string& name);
    int FindVertexIndex(const string& name) const;

    void AddEdgeByIndex(int vi, int vj, int weight);

    void DFS();
    void DFSFun(int v, bool visited[]);
    void BFS(const string& startName);
    void Print();
    // крускал
    void FindMinSpanningTree();

private:

    int CollectEdges(Edge outEdges[], int maxEdges);
```

```
    void InsertionSortByWeight(Edge arr[], int n);

    int FindParent(int parent[], int i);
    void UnionParents(int parent[], int x, int y);
};

#endif // GRAPH_H
```

Файл Graph.cpp

```
#include "Graph.h"
#include <iostream>

Graph::Graph(int v)
{
    V = v;

    for (int i = 0; i < MAX_VERTICES; ++i)
    {
        names[i] = "";
        adjCount[i] = 0;

        for (int j = 0; j < MAX_EDGES_PER_VERTEX; ++j)
        {
            adj[i][j].src = -1;
            adj[i][j].dest = -1;
            adj[i][j].weight = 0;
        }
    }
}

void Graph::SetVertexName(int idx, const string& name)
{
    if (idx < 0 || idx >= MAX_VERTICES) return;
    names[idx] = name;
}

int Graph::FindVertexIndex(const string& name) const
{
    for (int i = 0; i < V; ++i)
    {
        if (names[i] == name)
            return i;
    }
    return -1;
}

void Graph::AddEdgeByIndex(int vi, int vj, int weight)
{
    if (vi < 0 || vi >= V || vj < 0 || vj >= V) return;

    for (int k = 0; k < adjCount[vi]; ++k)
    {
        if (adj[vi][k].dest == vj && adj[vi][k].weight == weight)
            return;
    }
    if (adjCount[vi] < MAX_EDGES_PER_VERTEX)
```

```
    {
        adj[vi][adjCount[vi]] = {vi, vj, weight};
        adjCount[vi]++;
    }
    for (int k = 0; k < adjCount[vj]; ++k)
    {
        if (adj[vj][k].dest == vi && adj[vj][k].weight == weight)
            return;
    }
    if (adjCount[vj] < MAX_EDGES_PER_VERTEX)
    {
        adj[vj][adjCount[vj]] = {vj, vi, weight};
        adjCount[vj]++;
    }
}

void Graph::DFS()
{
    bool visited[MAX_VERTICES];

    for (int i = 0; i < V; ++i) visited[i] = false;

    for (int i = 0; i < V; ++i)
    {
        if (!visited[i])
            DFSFun(i, visited);
    }

    cout << endl;
}

void Graph::DFSFun(int v, bool visited[])
{
    visited[v] = true;

    cout << names[v] << " ";

    for (int k = 0; k < adjCount[v]; ++k)
    {
        int to = adj[v][k].dest;
        if (!visited[to])
            DFSFun(to, visited);
    }
}

void Graph::BFS(const string& startName)
{
    int start = FindVertexIndex(startName);
    if (start == -1)
    {
        cout << "Start vertex not found\n";
        return;
    }

    bool visited[MAX_VERTICES];
    for (int i = 0; i < V; i++) visited[i] = false;
```

```
    int q[MAX_VERTICES];
    int front = 0, back = 0;

    auto qpush = [&](int x) { q[back++] = x; };
    auto qpop = [&]() { return q[front++]; };
    auto qempty = [&]() { return front == back; };

    visited[start] = true;
    qpush(start);

    while (!qempty())
    {
        int u = qpop();
        cout << names[u] << " ";

        for (int k = 0; k < adjCount[u]; ++k)
        {
            int to = adj[u][k].dest;
            if (!visited[to])
            {
                visited[to] = true;
                qpush(to);
            }
        }
    }
    cout << endl;
}

void Graph::Print()
{
    for (int i = 0; i < V; ++i)
    {
        cout << names[i] << " -> ";

        if (adjCount[i] == 0)
        {
            cout << "(no edges)";
        }
        else
        {
            for (int k = 0; k < adjCount[i]; ++k)
            {
                int d = adj[i][k].dest;
                int w = adj[i][k].weight;
                cout << "(" << names[d] << ", " << w << ") ";
            }
        }

        cout << endl;
    }
}

int Graph::CollectEdges(Edge outEdges[], int maxEdges)
{
    int count = 0;

    for (int i = 0; i < V; ++i)
```

```
    {
        for (int k = 0; k < adjCount[i]; ++k)
        {
            int j = adj[i][k].dest;
            int w = adj[i][k].weight;

            if (i < j)
            {
                if (count < maxEdges)
                {
                    outEdges[count++] = {i, j, w};
                }
            }
        }
    }
    return count;
}

void Graph::InsertionSortByWeight(Edge arr[], int n)
{
    for (int i = 1; i < n; ++i)
    {
        Edge key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j].weight > key.weight)
        {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

int Graph::FindParent(int parent[], int i)
{
    while (parent[i] != -1)
        i = parent[i];
    return i;
}

void Graph::UnionParents(int parent[], int x, int y)
{
    parent[y] = x;
}

void Graph::FindMinSpanningTree()
{
    Edge edgesList[MAX_EDGES];
    int edgesCount = CollectEdges(edgesList, MAX_EDGES);

    if (edgesCount == 0)
    {
        cout << "No edges in graph\n";
        return;
    }
```

```
    InsertionSortByWeight(edgesList, edgesCount);

    int parent[MAX_VERTICES];
    for (int i = 0; i < V; i++) parent[i] = -1;

    Edge mst[MAX_EDGES];
    int mstCount = 0;
    int totalWeight = 0;

    for (int i = 0; i < edgesCount && mstCount < V - 1; i++)
    {
        int x = FindParent(parent, edgesList[i].src);
        int y = FindParent(parent, edgesList[i].dest);

        if (x != y) {
            mst[mstCount++] = edgesList[i];
            totalWeight += edgesList[i].weight;
            UnionParents(parent, x, y);
        }
    }

    cout << "Edges in MST are:\n";

    // сортировка по именам вершин
    for (int i = 1; i < mstCount; ++i)
    {
        Edge key = mst[i];
        int j = i - 1;

        auto compare = [&](const Edge& a, const Edge& b)
        {
            if (names[a.src] != names[b.src])
                return names[a.src] > names[b.src];
            return names[a.dest] > names[b.dest];
        };

        while (j >= 0 && compare(mst[j], key))
        {
            mst[j + 1] = mst[j];
            j--;
        }
        mst[j + 1] = key;
    }

    for (int i = 0; i < mstCount; i++)
    {
        cout << names[mst[i].src] << " -- "
                  << names[mst[i].dest] << " ("
                  << mst[i].weight << ")\n";
    }

    cout << "Average Weight = " << totalWeight << endl;
}
```

Файл application.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include "Graph.h"

// Разбивает строку на слова
int SplitTokens(const string& line, string tokens[], int maxTokens)
{
    istringstream iss(line);
    string temp;
    int count = 0;

    while (iss >> temp && count < maxTokens)
    {
        tokens[count++] = temp;
    }

    return count;
}

// Читаем матрицу смежности и строим граф
Graph* ParseAdjacencyMatrix(const string& filePath)
{
    ifstream in(filePath);
    if (!in.is_open())
    {
        cout << "Error: cannot open file " << filePath << "\n";
        return nullptr;
    }

    string line;

    // Читаем первую строку — имена вершин
    if (!getline(in, line))
    {
        cout << "Error: file is empty\n";
        return nullptr;
    }

    string tokens[MAX_VERTICES];
    int numVertices = SplitTokens(line, tokens, MAX_VERTICES);

    if (numVertices <= 0)
    {
        cout << "Error: no vertex names found\n";
        return nullptr;
    }

    Graph* g = new Graph(numVertices);

    for (int i = 0; i < numVertices; ++i) g->SetVertexName(i, tokens[i]);

    // Читаем строки матрицы
```

```
    int row = 0;
    while (getline(in, line) && row < numVertices)
    {
        istringstream iss(line);
        int weight;
        int col = 0;

        while (iss >> weight && col < numVertices)
        {
            if (weight > 0 && row <= col) g->AddEdgeByIndex(row, col,
weight); // диогонали учитываем A - A
            col++;
        }
        row++;
    }
    return g;
}

void printMenu()
{
    cout << "Main menu:\n";
    cout << "1. Initialization Graph\n";
    cout << "2. Graph traversal\n";
    cout << "3. Find Min Spanning Tree\n";
    cout << "4. Print Graph\n";
    cout << "0. Exit\n";
}

void application()
{
    Graph* graph = nullptr;
    int choice = 0;

    printMenu();

    while (true)
    {
        cout << "Enter your choice: ";

        if (!(cin >> choice))
        {
            cin.clear();
            cin.ignore(10000, '\n');
            cout << "Invalid input, try again\n";
            continue;
        }

        switch (choice)
        {
        case 1:
        {
            string filename;
            cout << "Enter filename: ";
            cin >> filename;

            Graph* g = ParseAdjacencyMatrix(filename);
            if (g != nullptr)
```

```
        {
            if (graph != nullptr) delete graph;
            graph = g;
        }
        break;
    }

    case 2:
    {
        if (!graph)
        {
            cout << "Graph not loaded\n";
            break;
        }

        cout << "1) DFS\n2) BFS\nEnter choice: ";
        int t;
        cin >> t;

        if (t == 1)
        {
            cout << "DFS: ";
            graph->DFS();
        }
        else if (t == 2)
        {
            string start = "A";
            cout << "BFS from \"" << start << "\": ";
            graph->BFS(start);
        }
        break;
    }

    case 3:
    {
        if (!graph)
        {
            cout << "Graph not loaded\n";
            break;
        }
        graph->FindMinSpanningTree();
        break;
    }

    case 4:
    {
        if (!graph)
        {
            cout << "Graph not loaded\n";
            break;
        }
        cout << "Graph:\n";
        graph->Print();
        break;
    }

    case 0:
```

```
        {
            if (graph) delete graph;
            return;
        }

        default:
            cout << "Unknown choice, try again\n";
        }
    }
}
```

Файл main.cpp

```
void application();
int main() {
    application();
    return 0;
}
```