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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: “Алгоритм поиска минимального остова на основе алгоритма Краскала”

Студент гр. 2372 _____ Васильев Ю.А.

Преподаватель _____ Пелевин М.С.

Санкт-Петербург

2023

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Васильев Ю.А.

Группа 2372

Тема работы: Алгоритм поиска минимального остова на основе алгоритма Краскала

Студент _____ Васильев Ю.А.

Преподаватель _____ Пелевин М.С.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы:

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

На вход подаётся текстовый файл, содержащий матрицу смежности графа, где первая строка – названия вершин графов, а остальные строки – квадратная матрица, представляющая собой вес ребра между вершинами. В матрице значение 0 стоит, если ребра между вершинами нет, и положительное число, которое соответствует весу, когда ребро между вершинами существует. Необходимо из найти минимальное остовое дерево из данного графа.

Результат в виде отсортированных по имени пар и их суммарный вес. Максимальный размер входных данных: 50 вершин. Вершины могут быть заданы любой текстовой последовательностью без пробелов. Вес ребра ограничен интервалом от 1 до 1023 включительно.

Методы решения:

Из представленного графа, ребра выписываются в отсортированном порядке, после чего они объединяются с помощью системы непересекающихся множеств (СНМ).

1. Граф

1.1. Что такое граф и как его представить

Граф – это абстрактная математическая структура, представляющая собой множество вершин (или узлов), соединенных рёбрами (или дугами). Графы используются для моделирования различных отношений и взаимосвязей в различных областях, таких как компьютерные науки, социология, транспортная логистика и многие другие.

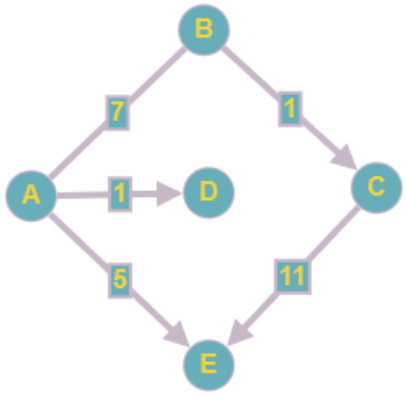

Рис.1 Граф с пятью вершинами

Существует несколько способов представления графа в программировании, включая матрицы смежности, матрицы инцидентности и списки смежности.

Рассмотрим возможности представления графов:

1.1.1 Матрица смежности:

Матрица смежности – это квадратная матрица, где строки и столбцы соответствуют вершинам графа, а элемент (i, j) равен весу ребра, если между вершинами i и j есть ребро, и 0 в противном случае, если ребра нет.

Пример матрицы смежности, представляющей граф на картинке выше:

	A	***B***	***C***	***D***	***E***
A	0	7	0	1	5
B	7	0	1	0	0
C	0	0	0	0	11
D	0	0	0	0	0
E	0	0	0	0	0

Табл.1 Пример матрицы смежности

1.1.2 Матрица инцидентности

Матрица инцидентности имеет вид представления графа в виде матрицы, в которой каждый столбец задаёт отдельное ребро. Строки матрицы при этом задают вершины. Положительное число в столбце задаёт вершину, из которой выходит ребро, а отрицательное - в которое входит. Если оба числа положительные, то ребро является двусторонним.

Пример матрицы инцидентности, представляющей граф на картинке выше:

	1	***2***	***3***	***4***	***5***
A	1	5	7	0	0
B	0	0	7	1	0
C	0	0	0	-1	11
D	-1	0	0	0	0
E	0	-5	0	0	-11

Табл.2 Пример матрицы инцидентности

1.1.3 Списки смежности:

В списке смежности каждой вершине сопоставляется список соседних с ней вершин.

Пример списка смежности, представляющей граф на картинке выше:

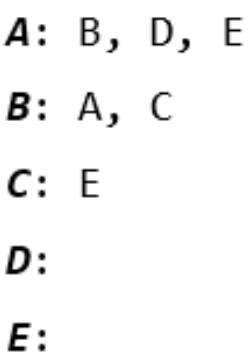

Рис.2 Пример списка смежности

1.2. Обходы графов

Алгоритмы обхода графов в глубину (Depth-First Search, DFS) и в ширину (Breadth-First Search, BFS) используются для поиска и обхода вершин в графе. Оба алгоритма помогают определить связность графа, находить пути, обнаруживать циклы и выполнять другие операции.

Оба алгоритма гарантируют, что каждая вершина будет посещена ровно один раз. Выбор между DFS и BFS зависит от конкретных требований задачи. DFS обычно используется для поиска в глубину, например, при топологической сортировке, а BFS - для поиска в ширину, такого как нахождение кратчайшего пути между двумя вершинами.

1.2.1 Обход в глубину (DFS)

Алгоритм работает так

1. Выбирается начальная вершина.

2. Вершина отмечается как посещённая.

3. Рекурсивное повторение этого же процесса для всех соседних вершин, которые ещё не были посещены.

1.2.2 Обход в ширину (BFS)

1. Выбирается начальная вершина и она же помещается в очередь.

2. Пока очередь не пуста, извлекается вершина из начала очереди.

3. Вершина отмечается как посещённая, после чего все её не посещённые соседние вершины добавляются в конец очереди.

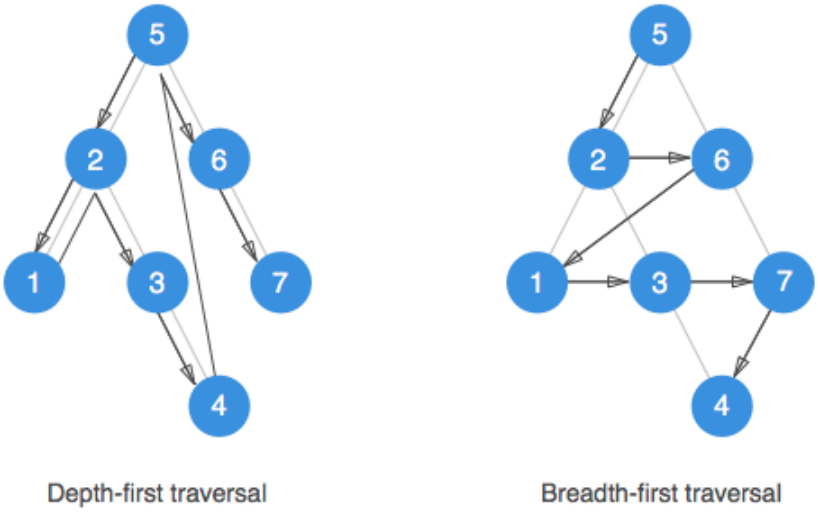

Рис.3 Иллюстрирование работы алгоритмов DFT и BFT

2. Система непересекающихся множеств

Система непересекающихся множеств – это структура данных, которая управляет множеством элементов, разбитых на несколько непересекающихся подмножеств. Она предоставляет около-константное время выполнения операций по добавлению новых множеств, слиянию существующих множеств и определению, относятся ли элементы к одному и тому же множеству.

Основные методы системы непересекающихся множеств:

- make_set(x): создает новое множество, содержащее элемент x. Каждый элемент изначально является представителем своего собственного множества.

- find(x): возвращает представителя множества, к которому принадлежит элемент x. Этот метод позволяет определить, принадлежат ли два элемента одному множеству.

- union(x, y): объединяет два множества, к которым принадлежат элементы x и y. Этот метод позволяет объединять множества и создавать более крупные множества, объединяя их представителей.

3. Минимальное остовое дерево

3.1. Что такое МОД и как его строить

Минимальное остовое дерево (Minimum Spanning Tree, MST) в графе — это подграф, содержащий все вершины исходного графа и обладающий следующими свойствами:

1. Связность: Все вершины связаны между собой рёбрами. То есть, минимальное остовое дерево должно быть связным.
2. Ацикличность: Все рёбра остового дерева не образуют циклы.
3. Минимальный вес: Сумма весов рёбер остового дерева минимальна. Каждый раз, когда добавляется ребро, выбирается ребро с минимальным весом из доступных.

Различают два алгоритма поиска такого дерева, такие как Алгоритм Прима и Алгоритм Краскала (в этой работе разобран именно он).

3.2. Алгоритм Краскала (Крускала)

Механизм, по которому работает данный алгоритм, очень прост. На входе имеется пустой подграф, который и будем достраивать до потенциального минимального остового дерева.

1. Вначале мы производим сортировку рёбер по неубыванию по их весам.
2. Добавляем i-ое ребро в наш подграф только в том случае, если данное ребро соединяет две разные компоненты связности, одним из которых является наш подграф. То есть, на каждом шаге добавляется минимальное по весу ребро, один конец которого содержится в нашем подграфе, а другой - еще нет. (В работе на этом этапе используется СНМ)
3. Алгоритм завершит свою работу после того, как множество вершин нашего подграфа совпадет с множеством вершин исходного графа.

Пример МОД, на основании графа на картинке выше:

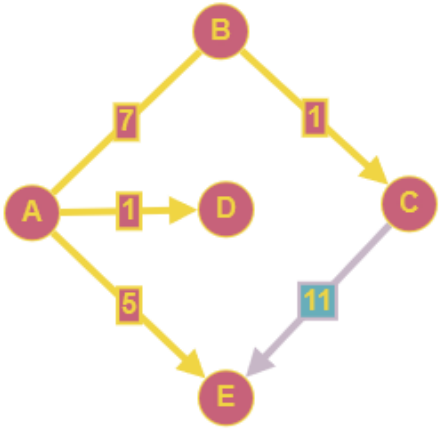

Рис.4 Минимальное остовое дерево (отмечено жёлтым)

ЗАКЛЮЧЕНИЕ

В результате выполненной работы было успешно построено минимальное остовое дерево на основе алгоритма Краскала с использованием системы непересекающихся множеств (СНМ). Алгоритм Краскала, в сочетании с СНМ, обеспечил эффективное нахождение минимального остового дерева во взвешенном графе, сохраняя свойства связности и ацикличности.

Применение СНМ позволило эффективно управлять структурой множеств, обеспечивая оптимальное объединение вершин и предотвращение образования циклов.

ПРИЛОЖЕНИЕ А. КОД ПРОГРАММЫ

dsu.h:

```
#pragma once
#include <vector>

// Disjoint-Set-Union Structure
class DSU {
    std::vector<size_t> parent, rank;
public:
    DSU(size_t n) : parent(n, 0), rank(n, 1) {
        for (size_t i = 0; i < n; i++)
            make(i);
    }

    // Поиск элемента v в множествах
    size_t find(size_t v) {
        return (v == parent[v]) ? v : (parent[v] = find(parent[v]));
    }

    // Создать множество из одного элемента v
    void make(size_t v) {
        parent[v] = v;
        rank[v] = 1;
    }

    // Объединение двух множеств a и b
    void unite(size_t a, size_t b) {
        a = find(a);
        b = find(b);
        if (a != b) {
            if (rank[a] < rank[b])
                std::swap(a, b);
            parent[b] = a;
            rank[a] += rank[b];
        }
    }
};
```

graph.h:

```
#pragma once
#include <string>
#include <vector>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <stack>
#include <unordered_set>
#include <queue>
#include "dsu.h"

// Edge Structure
struct Edge {
    std::string a, b;
    int u, v, weight;

    // Простой конструктор
    Edge(std::string x, std::string y, int w = 0) {
        a = x;
```

```
        b = y;
        weight = w;
    }

    // Именованный конструктор
    Edge(std::vector<std::string>& named, int x, int y, int w = 0) : Edge(named[x],
named[y], w) {
        u = x;
        v = y;
    }

    bool operator< (Edge const& other) {
        return weight < other.weight;
    }
};

class Graph {
    std::vector<Edge> edges;
    unsigned mass;

public:
    Graph();
    Graph(std::vector<Edge>);

    static Graph fromFile(std::string);

    std::string first();
    Graph mst() const;
    std::vector<std::string> dfs(const std::string&);
    std::vector<std::string> bfs(const std::string&);
    const std::vector<Edge> get_edges() const;
    unsigned get_mass() const;
};

Graph::Graph() {
    mass = 0;
}

Graph::Graph(std::vector<Edge> e) {
    edges = e;
    mass = 0;
    for (Edge edge : edges) {
        mass += edge.weight;
    }
}

// Создать граф из файла
Graph Graph::fromFile(std::string path) {
    std::vector<std::string> names;
    std::string next;

    std::ifstream ifs(path);
    if (!ifs.is_open()) throw std::runtime_error("File not found: " + path);

    std::string line;
    getline(ifs, line);
    std::istringstream ss(line);

    while (ss >> next) {
        names.push_back(next);
```

```
    }

    unsigned m = 0;
    std::vector<Edge> edges;
    for (unsigned i = 0; i < names.size(); i++) {
        for (unsigned j = 0; j < names.size(); j++) {
            int w;
            ifs >> w;
            if (i < j) {
                if (w < 0) throw std::runtime_error("Invalid weights in file: " +
std::to_string(w));
                else if (w > 0) {
                    Edge e(names, i, j, w);
                    m += w;
                    edges.push_back(e);
                }
            }
        }
    }

    return Graph(edges);
}

std::string Graph::first() {
    return edges.empty() ? "" : edges.front().a;
}

std::vector<std::string> Graph::dfs(const std::string& start) {
    std::stack<std::string> stack;
    std::unordered_set<std::string> visited;
    std::vector<std::string> out;

    stack.push(start);
    while (!stack.empty()) {
        std::string current = stack.top();
        stack.pop();

        if (visited.find(current) == visited.end()) {
            out.push_back(current);
            visited.insert(current);

            for (const Edge& edge : edges) {
                if (edge.a == current && visited.find(edge.b) == visited.end()) {
                    stack.push(edge.b);
                }
            }
        }
    }

    return out;
}

std::vector<std::string> Graph::bfs(const std::string& start) {
    std::queue<std::string> queue;
    std::unordered_set<std::string> visited;
    std::vector<std::string> out;

    queue.push(start);
    while (!queue.empty()) {
        std::string current = queue.front();
```

```
        queue.pop();

        if (visited.find(current) == visited.end()) {
            std::cout << current << " ";
            visited.insert(current);

            for (const Edge& edge : edges) {
                if (edge.a == current && visited.find(edge.b) == visited.end()) {
                    queue.push(edge.b);
                }
            }
        }
    }

    return out;
}

// Поиск минимального остового дерева текущего графа
Graph Graph::mst() const {
    DSU dsu(edges.size());

    unsigned n = 0;
    std::vector<Edge> mst;
    std::vector<Edge> edges_sorted(edges);
    std::sort(edges_sorted.begin(), edges_sorted.end());
    for (Edge e : edges_sorted) {
        if (dsu.find(e.u) != dsu.find(e.v)) {
            n += e.weight;
            mst.push_back(e);
            dsu.unite(e.u, e.v);
        }
    }

    return Graph(mst);
}

// Получить вершины графа
const std::vector<Edge> Graph::get_edges() const
{
    return edges;
}

// Получить массу графа
unsigned Graph::get_mass() const
{
    return mass;
}
```

T1_Coursework.cpp:

```
#include <iostream>
#include <fstream>
#include "dsu.h"
#include "graph.h"

using namespace std;

/*
*   Функция для ввода данных в терминал
*   При вызове функции нужно указать получаемые данные в скобках,
```

```
*   т.е. readValue<int>() - получить число.
*   prompt - текст перед вводом
*   value - значение для заполнения
*/
template <typename T>
T readValue(const char* prompt = "") {
    T value = 0;
    cout << prompt;
    while (true) {
        cin >> value;
        if (cin.fail()) {
            cout << "Incorrect input. Enter new value: ";
            cin.clear();
            // numeric_limits<streamsize> это предел количества знаков в streamsize
(вернёт число)
            // нужно чтобы очистить максимальное количество оставшихся символов в буфере
до новой строки
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
        else {
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            return value;
        }
    }
}

void loadGraph(Graph& g) {
    while (true) {
        try {
            system("cls");
            g = Graph::fromFile("input.txt");
            cout << "Loaded successfully\n";
            return;
        } catch (runtime_error e) {
            cout << "Loading error: " << e.what() << endl;
            cout << "Press anything to load again\n";
            system("pause");
        }
    }
}

int main()
{
    Graph graph;
    loadGraph(graph);
    while (true) {
        system("cls");
        cout << "Graph loaded\n";
        cout <<
            "\nChoose a category from below:\n"
            "0. Exit\n"
            "1. Reload graph\n"
            "2. MST to console\n"
            "3. MST to file\n"
            "4. Depth First Search\n"
            "5. Breadth First Search\n\n";
        int choice = readValue<int>("Type a number to continue: ");
        cout << endl;
        switch (choice) {
            case 0:
```

```
                return 0;
            case 1: {
                loadGraph(graph);
                break;
            }
            case 2: {
                auto mst = graph.mst();
                for (Edge e : mst.get_edges()) {
                    cout << e.a << ' ' << e.b << endl;
                }
                cout << mst.get_mass() << endl;
                break;
            }
            case 3: {
                ofstream ofs("output.txt");
                auto mst = graph.mst();
                for (Edge e : mst.get_edges())
                    ofs << e.a << ' ' << e.b << endl;
                ofs << mst.get_mass();
                break;
            }
            case 4: {
                auto w = graph.dfs(graph.first());
                for(std::string s : w) {
                    cout << s << ' ';
                }
                cout << endl;
                break;
            }
            case 5: {
                auto w = graph.bfs(graph.first());
                for(std::string s : w) {
                    cout << s << ' ';
                }
                cout << endl;
                break;
            }
            default:
                cout << "\nCategory with number " << choice << " does not exist." <<
endl;
                break;
        }
        system("pause");
    }
}
```