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 _____ Гейдарбеков Р.Т.

Преподаватель _____ Пелевин М.С.

Санкт-Петербург

2025

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Гейдарбеков Р.Т.

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: 19.12.2025

Дата защиты реферата: 22.12.2025

Студент _____ Гейдарбеков Р.Т.

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

1.2. Описание алгоритма сортировки

Алгоритм: модифицированный TimSort — адаптивная гибридная сортировка, использующая обнаружение run, insertion sort для коротких run и поэтапные слияния с галопированием:

1)Вычислить minRun функцией calculateMinRun(lenArr).

2)Идти по массиву слева направо и выделять run:

- если неубывающая последовательность — продолжать;
- если убывающая — накопить и затем развернуть (reverseRun) в возрастающую.

3)Если длина run < minRun, дополнять его до minRun сортировкой вставками (insertionSort).

4)Помещать каждый run в стек как пару (начало, длина).

5)После добавления run применять правила сворачивания стека (mergeRuns):

- если верхние три run X,Y,Z нарушают инварианты, сливать X+Y или Y+Z по правилам X.len <= Y.len + Z.len и Y.len <= Z.len.

6)Для слияния двух run выделять временные буферы и выполнять стандартное слияние (merge).

7)При слиянии применять галопирование (gallopingModernizated + binarySearch) для быстрого пропуска длинных последовательностей из одного буфера.

8)Повторять слияния по правилам стека до получения одного отсортированного run (всего 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 kruskal(DynamicArray<Edge>& edges, int vertexCount) {

    timSort(edges, edges.size);

    SDS sds(vertexCount);

    DynamicArray<Edge> mst(vertexCount - 1);

    for (int i = 0; i < edges.size; i++) {
```

```
            int u = edges[i].u;
            int v = edges[i].v;

            if (u < 0 || u >= vertexCount || v < 0 || v >= vertexCount) {
                cout << "Пропущено некорректное ребро: "
                    << u << " - " << v << " вес: " << edges[i].weight << endl;
                continue;
            }

            if (sds.find(u) != sds.find(v)) {
                mst.insert(mst.size, edges[i]);
                sds.unionSets(u, v);
            }

            if (mst.size == vertexCount - 1) break;
        }


        cout << "Минимальное остовное дерево:" << endl;
        for (int i = 0; i < mst.size; i++) {
            cout << mst[i].u_name << " - " << mst[i].v_name
                << " : " << mst[i].weight << 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 cursa4ik.cpp

```
#include <iostream>
#include <climits>
#include <fstream>
#include <sstream>
#include <string>

using namespace std;



struct Edge {
    int u, v, weight;
    string u_name, v_name;

    Edge() : u(-1), v(-1), weight(0) {}
    Edge(int _u, int _v, int _w, string _un, string _vn)
        : u(_u), v(_v), weight(_w), u_name(_un), v_name(_vn) {
    }

    bool operator<(const Edge& other) { return weight < other.weight; }
    bool operator>(const Edge& other) { return weight > other.weight; }
    bool operator<=(const Edge& other) { return weight <= other.weight; }
    bool operator>=(const Edge& other) { return weight >= other.weight; }
    bool operator==(const Edge& other) { return weight == other.weight; }
    bool operator!=(const Edge& other) { return weight != other.weight; }
};
```

```
template <typename T>
class StackNode {
public:
    T value;
    StackNode<T>* next = nullptr;
public:
    StackNode(T value) {
        this->value = value;
    }
};

template <typename T>
class Stack {
public:
    StackNode<T>* top;

public:
    Stack<T>() {
        top = nullptr;
    }
    ~Stack() {
        while (!isEmpty()) {
            pop();
        }
    }
    void push(T value) {
        StackNode<T>* temp = new StackNode<T>(value);
        temp->next = top;
        top = temp;
    }
```

```
void pop() {
    if (!isEmpty()) {
        StackNode<T>* temp = top;
        top = top->next;
        delete temp;
    }
    else {
        cout << "Стэк и так пуст";
        return;
    }
}

bool isEmpty() {
    return (top == nullptr);
}

T get() {
    if (!isEmpty()) {
        return top->value;
    }
    else {
        cout << "Стэк пуст";
        return T();
    }
}

int size() {
    StackNode<T>* cur = top;
    int count(0);
```

```
        while (cur) {
            cur = cur->next;
            count++;
        }
        return count;
    }

};

template <typename T>
class DynamicArray {
public:
    T* data;
    int size;
    int capacity;

public:
    DynamicArray(int capacity) {
        this->capacity = capacity;
        data = new T[capacity];
        this->size = 0;
    }
    ~DynamicArray() {
        delete[] data;
    }
    void resize(int newSize) {
        T* temp = new T[newSize];
        for (int i = 0; i < size; i++) {
            temp[i] = data[i];
        }
```

```
    for (int i = size; i < newSize; i++) {
        temp[i] = T();
    }
    this->size = newSize;
    this->capacity = newSize;
    delete[] data;
    this->data = temp;
}

void reverse() {
    T* temp = new T[capacity * 2];
    for (int i = 0; i < size; i++) {
        temp[i] = data[i];
    }
    delete[] data;
    this->capacity *= 2;
    this->data = temp;
}

void reverse(int capacity) {
    T* temp = new T[capacity];
    for (int i = 0; i < size; i++) {
        temp[i] = data[i];
    }
    delete[] data;
    this->capacity = capacity;
    this->data = temp;
}

void remove(int index) {
```

```
        while (index >= size || index < 0) {
            std::cout << "Индекс либо больше размера массива, либо меньше нуля, попробуйте снова: ";
            std::cin >> index;
        }
        for (int i = index; i < size - 1; i++) {
            data[i] = data[i + 1];
        }
        size--;
    }

    void insert(int index, T value) {
        while (index > size || index < 0) {
            std::cout << "Индекс либо больше размера массива, либо меньше нуля, попробуйте снова: ";
            std::cin >> index;
        }
        if (size == capacity) {
            reverse();
        }
        for (int i = size; i > index; i--) {
            data[i] = data[i - 1];
        }
        data[index] = value;
        size++;
    }

    T& operator[](int index) {
        if (index < 0 || index >= size) {
```

```
            throw std::out_of_range("Индекс либо больше размера массива, либо меньше нуля");
        }
        return data[index];
    }
};

int binarySearch(DynamicArray<Edge>& arr, Edge target, int left, int right) {
    int mid = (left + right) / 2;
    if (target == arr[mid]) {
        return mid;
    }
    if (target > arr[mid]) {
        return binarySearch(arr, target, mid + 1, right);
    }
    else {
        return binarySearch(arr, target, left, mid - 1);
    }
}
void insertionSort(DynamicArray<Edge>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        Edge value = arr[i];
        for (int j = i - 1; j >= left && arr[j] > value; j--) {
            arr[j + 1] = arr[j];
            arr[j] = value;
        }
    }
}
int calculateMinRun(int lenArr) {
    int reminder = 0;
```

```
    while (lenArr >= 64) {
        if (lenArr & 1) {
            reminder++;
        }
        lenArr >>= 1;
    }
    return lenArr + reminder;
}

void reverseRun(DynamicArray<Edge>& arr, int left, int right) {
    while (left < right) {
        Edge tmp = arr[left];
        arr[left] = arr[right];
        arr[right] = tmp;
        left++;
        right--;
    }
}
int gallopingModernizated(DynamicArray<Edge>& arr, int start, int end, Edge value)
{
    int limit = min(start + 7, end);
    int i = start;
    while (i < limit && arr[i] < value) {
        i++;
    }

    if (i < limit) {
        return i;
    }
```

```
    int step = 1;
    int posAfterFirstCheck = start + 7;

    while (posAfterFirstCheck < end && arr[posAfterFirstCheck] < value) {
        posAfterFirstCheck = start + 7 + (1 << step);
        step++;
    }
    int leftBound = start + 7 + (1 << (step - 1));
    int rightBound = min(posAfterFirstCheck, end);
    leftBound = min(leftBound, end);
    rightBound = min(rightBound, end);
    return binarySearch(arr, value, leftBound, rightBound);
}
void merge(DynamicArray<Edge>& arr, int left, int mid, int right) {
    int i = 0, j = 0;
    int len1 = mid - left + 1;
    int len2 = right - mid;

    Edge* a = new Edge[len1 + 1];
    Edge* b = new Edge[len2 + 1];
    a[len1] = Edge(-1, -1, INT_MAX, "", "");
    b[len2] = Edge(-1, -1, INT_MAX, "", "");
    for (int i = 0; i < len1; i++) {
        a[i] = arr[left + i];
    }
    for (int i = mid + 1; i < right + 1; i++) {
        b[j++] = arr[i];
    }
    i = 0;
    j = 0;
```

```
    for (int k = left; k <= right; k++) {
        if (a[i] <= b[j]) {
            arr[k] = a[i++];
        }
        else {
            if (len2 - j > 7) {
                int gallopingJ = gallopingModernizated(arr, mid + 1 + j, right, a[i]);
                if (gallopingJ > mid + 1 + j) {
                    while (mid + 1 + j < gallopingJ && k <= right) {
                        arr[k++] = b[j++];
                    }
                    k--;
                }
            }
            arr[k] = b[j++];
        }
    }
    delete[] a;
    delete[] b;
}
void mergeRuns(DynamicArray<Edge>& arr, Stack <pair<int, int>>& stack) {
    while (stack.size() > 1) {
        pair<int, int> X, Y, Z;
        Z = stack.get();
        stack.pop();
        Y = stack.get();
        stack.pop();
        bool hasX = false;
        if (!stack.isEmpty()) {
            X = stack.get();
```

```
            stack.pop();
            hasX = true;
        }
        bool merged = false;
        if (hasX && X.second <= Y.second + Z.second) {
            merge(arr, X.first, X.first + X.second - 1, Y.first + Y.second - 1);
            stack.push({ X.first, X.second + Y.second });
            stack.push(Z);
            merged = true;
        }
        else if (Y.second <= Z.second) {
            merge(arr, Y.first, Y.first + Y.second - 1, Z.first + Z.second - 1);
            stack.push({ Y.first, Y.second + Z.second });
            if (hasX) stack.push(X);
            merged = true;
        }
        if (!merged) {
            if (hasX) stack.push(X);
            stack.push(Y);
            stack.push(Z);
            break;
        }
    }
    while (stack.size() > 1) {
        pair<int, int> Z = stack.get(); stack.pop();
        pair<int, int> Y = stack.get(); stack.pop();
        merge(arr, Y.first, Y.first + Y.second - 1, Z.first + Z.second - 1);
        stack.push({ Y.first, Y.second + Z.second });
    }
}
```

```
void timSort(DynamicArray<Edge>& arr, int lenArr) {
    int minRun = calculateMinRun(lenArr);
    Stack<pair<int, int>> stack;
    int counter;
    int startRun;
    for (int i = 0; i < lenArr; i++) {
        counter = 1;
        startRun = i;
        if (lenArr == i + 1) {
            stack.push({ i,counter });
            mergeRuns(arr, stack);
            break;
        }
        startRun = i;
        if (arr[i] <= arr[i + 1]) {
            while (i + 1 < lenArr && arr[i] <= arr[i + 1]) {
                counter++;
                i++;
            }
        }
        else {
            while (i + 1 < lenArr && arr[i] > arr[i + 1]) {
                counter++;
                i++;
            }
            reverseRun(arr, startRun, startRun + counter - 1);
        }
        if (counter < minRun) {
            int minimal = min(lenArr - startRun, minRun);
            insertionSort(arr, startRun, startRun + minimal - 1);
```

```
            counter = minimal;
            i = startRun + counter - 1;
        }
        stack.push({ startRun, counter });
        mergeRuns(arr, stack);
    }
    mergeRuns(arr, stack);
}
class SDS {
public:
    DynamicArray<int> parents;
    SDS(int countEl) : parents(countEl) {
        parents.resize(countEl);
        for (int i = 0; i < countEl; i++) {
            parents[i] = i;
        }
    }
    int find(int index) {
        if (index < 0 || index >= parents.size) {
            throw std::out_of_range("Индекс в find() вне диапазона");
        }
        if (parents[index] == index) {
            return index;
        }
        else {
            parents[index] = find(parents[index]);
            return parents[index];
        }
    }
    void unionSets(int firstEl, int secondEl) {
```

```
        int root1 = find(firstEl);
        int root2 = find(secondEl);
        if (root1 != root2) {
            parents[root1] = root2;
        }
    }
};
void DFS(int current, DynamicArray<Edge>& edges, bool visited[], string
vertexNames[], int vertexCount) {
    visited[current] = true;
    cout << vertexNames[current] << " ";
    for (int i = 0; i < edges.size; i++) {
        Edge edge = edges[i];
        int neighbor = -1;
        if (edge.u == current) {
            neighbor = edge.v;
        }
        else if (edge.v == current) {
            neighbor = edge.u;
        }
        if (neighbor != -1 && !visited[neighbor]) {
            DFS(neighbor, edges, visited, vertexNames, vertexCount);
        }
    }
}
void BFS(int start, DynamicArray<Edge>& edges, int vertexCount, string
vertexNames[]) {
    bool* visited = new bool[vertexCount];
    for (int i = 0; i < vertexCount; i++)
        visited[i] = false;
```

```
    int* queue = new int[vertexCount];
    int front = 0, rear = 0;
    queue[rear++] = start;
    visited[start] = true;
    while (front < rear) {
        int current = queue[front++];
        cout << vertexNames[current] << " ";
        for (int i = 0; i < edges.size; i++) {
            int neighbor = -1;
            if (edges[i].u == current) neighbor = edges[i].v;
            else if (edges[i].v == current) neighbor = edges[i].u;
            if (neighbor != -1 && !visited[neighbor]) {
                queue[rear++] = neighbor;
                visited[neighbor] = true;
            }
        }
    }
    delete[] visited;
    delete[] queue;
}
bool edgeExists(DynamicArray<Edge>& edges, int u, int v) {
    for (int i = 0; i < edges.size; i++) {
        if ((edges[i].u == u && edges[i].v == v) || (edges[i].u == v && edges[i].v == u))
{
            return true;
        }
    }
    return false;
}
void kruskal(DynamicArray<Edge>& edges, int vertexCount) {
```

```
    timSort(edges, edges.size);
    SDS sds(vertexCount);
    DynamicArray<Edge> mst(vertexCount - 1);
    for (int i = 0; i < edges.size; i++) {
        int u = edges[i].u;
        int v = edges[i].v;
        if (u < 0 || u >= vertexCount || v < 0 || v >= vertexCount) {
            cout << "Пропущено некорректное ребро: "
                << u << " - " << v << " вес: " << edges[i].weight << endl;
            continue;
        }
        if (sds.find(u) != sds.find(v)) {
            mst.insert(mst.size, edges[i]);
            sds.unionSets(u, v);
        }

        if (mst.size == vertexCount - 1) break;
    }



    cout << "Минимальное остовное дерево:" << endl;
    for (int i = 0; i < mst.size; i++) {
        cout << mst[i].u_name << " - " << mst[i].v_name
            << " : " << mst[i].weight << endl;
    }
}

int main() {
    ifstream file("C:\\Users\\tolop\\Desktop\\Graph.txt");
    if (!file.is_open()) {
```

```
        cout << "Такого файла не найдено" << endl;
        return 1;
    }
    string line;
    string vertexNames[20];
    int vertexCount = 0;
    if (getline(file, line)) {
        stringstream ss(line);
        string name;
        while (ss >> name) {
            vertexNames[vertexCount++] = name;
        }
    }
    DynamicArray<Edge> edges(10);
    for (int i = 0; i < vertexCount; i++) {
        if (!getline(file, line)) {
            break;
        }
  stringstream ss(line);

        int weight;
        for (int j = 0; j < vertexCount; j++) {
            ss >> weight;
            if (weight != 0 && !edgeExists(edges, i, j)) {
                edges.insert(edges.size, Edge(i, j, weight, vertexNames[i],
vertexNames[j]));
            }
        }
    }
    file.close();
```

```
    if (vertexCount > 0) {
        bool* visited = new bool[vertexCount];
        for (int i = 0; i < vertexCount; i++) {
            visited[i] = false;
        }
        cout << "DFS: ";
        DFS(0, edges, visited, vertexNames, vertexCount);
        cout << "BFS: ";
        BFS(0, edges, vertexCount, vertexNames);

        kruskal(edges, vertexCount);
    }

}
```