**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ»** **им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

Курсовая РАБОТА

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Реализация алгоритма поиска минимального остова на основе алгоритма Краскала

|  |  |  |
| --- | --- | --- |
| Студент гр. 8363 |  | Нерсисян А.С. |
| Преподаватель |  | Ушаков Р. |

Санкт-Петербург

2019

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Нерсисян А.С. | | |
| Группа 8363 | | |
| Тема работы: Реализация алгоритма поиска минимального остова на основе алгоритма Краскала. | | |
| Исходные данные:  Разработать на языке C++ приложение для поиска минимального остова графа. Входные данные: любой текстовый файл или введённый через консоль набор троек: {A B 3, B C 2, A C 1; }, выходные данные: результат в виде отсортированных по имени пар и суммарный вес: {A C, B C, 3; }. Максимальный размер входных данных: 50 вершин. | | |
| Содержание пояснительной записки:  Введение, способ представления данных в памяти, инициализация графа, сортировка ребер графа по возрастанию весов, описание алгоритма Краскала, сортировка ребер в лексикографическом порядке, оценка временной сложности, результат выполнения задачи, заключение, список использованных источников, приложение А ̶ исходный код. | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страниц. | | |
| Дата выдачи задания: 12.11.2019 | | |
| Дата сдачи курсового проекта: 20.12.2019 | | |
| Дата защиты курсового проекта: 20.12.2019 | | |
| Студент |  | Нерсисян А.С. |
| Преподаватель |  | Ушаков Р. |

**Аннотация**

Алгоритма Краскала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

Задача: разработать на языке C++ консольное приложение для нахождения минимального остова графа. В проекте описывается теоретическая составляющая: реализация консольного приложения для построение системы непересекающихся множеств (СНМ), а также описание выбранных алгоритмов сортировки, обхода графов и СНМ и их предназначение в данном приложении.

В отчете присутствует исходный код программы.

**Summary**

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

The task is to develop a console application in C ++ for finding the minimum spanning tree of a graph. The project describes the theoretical component: the implementation of a console application for constructing a disjoint-set data structure, as well as a description of the selected algorithms for sorting, traversing graphs and а disjoint-set data structure and their purpose in this application.

This report contains the source code of the program.

**содержание**

[Введение 5](#_Toc25023157)

[1. Способ представления данных в памяти 6](#_Toc25023158)

[1.1 Класс ребра: class Edge 6](#_Toc25023159)

[1.2 Класс графа: class Graph 7](#_Toc25023160)

[2. Инициализация графа: void initGraphFromFileAngry() 7](#_Toc25023161)

[3. Сортировка ребер графа по возрастанию весов: void QuickSort() 9](#_Toc25023162)

[4. Описание алгоритма Краскала: void Kruskal() 10](#_Toc25023163)

[4.1 Обход графа 10](#_Toc25023164)

[4.2 Оценка сложности алгоритма 11](#_Toc25023165)

[5. Сортировка ребер в лексикографическом порядке: void sortEdgesОfDisjointSetAtLex() 11](#_Toc25023166)

[6. Оценка общей временной сложности 12](#_Toc25023167)

[7. Результаты решения задачи 12](#_Toc25023168)

[Заключение 14](#_Toc25023169)

[Список использованных источников 15](#_Toc25023170)

[Приложение А: Исходный код программы 16](#_Toc25023171)

# Введение

Цель работы: Реализовать алгоритм поиска минимального остова на основе алгоритма Краскала (Крускала). Продемонстрировать знания следующих вопросов:

- сортировка,

- обход графов,

- хранение графов,

- построение системы непересекающихся множеств.

Входные данные:

Любой текстовый файл или введённый через консоль набор троек:

A B 3

B C 2

A C 1

Результат в виде отсортированных по имени пар и суммарный вес:

A C

B C

3

Максимальный размер входных данных: 50 вершин

Вершины могут быть заданы любой текстовой последовательностью без пробелов. Вес ребра ограничен интервалом от 0 до 1023 включительно**.**

# Способ представления данных в памяти

При выполнении задания для хранения информации о неориентированном графе был выбран метод хранения массива рёбер графа для исходных данных и результата. Матрица смежности в данной работе не подходит, так как рёбра имеют веса. Из предыдущего логично предположить, что подойдёт весовая матрица, но, во-первых, она будет иметь пустые ячейки, вдобавок к этому веса будут повторяться, т. к. граф неориентированный. Во-вторых, для одного обхода всей матрицы временная сложность будет порядка Матрица инцидентности имеет схожую проблему, обход всей матрицы займёт порядка времени.

Вывод: отсортированный массив рёбер, по моему мнению, лучше всего подходит для хранения результата ввода и вывода.

Для выполнения обработки были использованы список рёбер, создаваемый из массива рёбер, а также, чтобы иметь возможность ограничить количество вершин (максимально 50) было использована список вершин (динамический массив).

## **1.1 Класс ребра:** class Edge

Класс ребра (class Edge) в себе содержит два строковых и три числовых переменных: имена двух вершин между которыми связь (string FirstV, string SecondV), вес ребра (unsigned short dist), цвет первой (int FirstVcolor) и второй вершины (int SecondVcolor) соответственно. Более подробная схема класса Edge показана на схеме.

Цвет второй вершины

Имя второй вершины

Вес ребра

Цвет первой вершины

Имя первой вершины

|  |
| --- |
| Edge |
| **unsigned short dist** |
| **string FirstV** |
| **string SecondV** |
| **int** **FirstVcolor** |
| **int** **SecondVcolor** |

## 1.2 Класс графа: class Graph

Класс графа в себе содержит три динамических массива: первый – std::vector<Edge> edges для хранения графа, второй –std::vector<Edge> disjointSet для построения системы непересекающихся множеств и третий –std::vector<std::string> collection для хранения вершин графа.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vector<Edge> edges | | | | |
| **Edge** | **Edge** | **Edge** | **...** | **Edge** |

# Инициализация графа: void initGraphFromFileAngry()

Входными данныими для построения графа являються набор троек: вершина, вершина и вес ребра из текстового файла "mattrix.txt". Алгоритм получения данных из файла получает и анализирует их по следующей логике:

**ЦИКЛ** while (пока не дошли до конца файла)

получить из файла очередную строку

**ЦИКЛ** for по i (пока не дошли до конца строки)

**ЦИКЛ** (пока очередной i символ пробел) i++

ЕСЛИ (очередной i символ буква) то первая вешина это i символ

**ЦИКЛ** (пока очередной i+1 символ пробел) i++

ЕСЛИ (очередной i+1 символ буква) то вторая вешина это i символ

**ЦИКЛ** (пока очередной i+2 символ пробел) i++

ЕСЛИ (очередной i+2 символ не буква) то число начинается с i символа

**ЦИКЛ** for по i (пока не встретили букву, пробел, и пока не дошли до

конца строки)

добавить каждый i символ в буфферную строку

**КЦ**

i+= длина буфферной строки

Операции доступа к символам строки с помощью арифметики указателей (i+1 и i+2) безопасны, имея в виду следующее: если существует первая вершина, то вторая точно должна существовать, а если существуют две вершины, следовательно следующий символ (за исключением пробелов, если они есть) очевидно должна быть весом ребра, связующей эти две вершины, которую нужно собрать посимвольно. Таким образом входная последоватеьность ограниена только одним параметром: две вершины и числовое значение веса ребра, связующей эти вершины, должны быть на одной строке, а число написанно без пробелов.

После получения всех необходимых данных для описания одного ребра (имена первого и второго вершин, вес ребра), создаеться новый экземпляр (объект) класса Edge, заполняется полученными данными (для получения информацци о цвете вершин используется вспомогательная функция-член клааса Graph int colorOf(std::string str), которая принимает строку и возвращает ASCII-код первого символа. Раскраска вершин выполняеться по данному алгоритму, чтобы все вершины с одинаковым именем, красились в один и тот же цвет), добавляеться в конец массива ребер (std::vector<Edge> edges) – в граф.

Для того, чтобы иметь возможность ограничить количество вершин графа используеться коллекция вешин (std::vector<std::string> collection). При последнем шаге, после добавления ребра в граф, оба вершины проверяються на наличие их имен в коллекции, если их там нет – добавляються. Для проверки очередной вершины на наличие в коллекции используеться вспомогательная финкция-член класса Graph bool isInCollection(std::string str). Она принимает строку (символ) и возвращает true если она уже есть в колекции и false в противном случае. После каждого добавления ребер в граф и вершин в колекцию проверяються их размеры: ребра до 1225 шт. и вершин до 50 шт. включительно. Если условия не выполняються программа выдает ошибку и останавливает дальнейшее выполнение.

# Сортировка ребер графа по возрастанию весов: void QuickSort()

Для сортировки ребер графа по возрастанию (не убыванию) весов ребер изначалньно был выбран алгоритм пузирьковой сортировки (BubbleSort). Это простейший алгоритм для понимания и реализации, но эффективен он лишь для небольших массивов. Сложность алгоритма в любом случае. Казалось бы, 50 вершин не так уж и много, но нужно сортировать ребра, а их максимальное количество при исключении петель (ребро из А в А) по формуле получаеться , что приводит к мыслю отказаться от пузырьковой сортировки в пользу быстрой сортировки (qsort, quickSort).

Алгоритм быстрой сортировки один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем случае обменов при упорядочении элементов. Алгоритм быстрой сортировки использует стратегию «разделяй и властвуй», общая идея алгоритма состоит в следующем:

Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива, в данной работе используется средний элемент. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.

Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на две непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного» и «больше или равные».

Для обоих отрезков значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Таким образом имеем следующее:

|  |  |  |
| --- | --- | --- |
| Лучший случай  (отсортированный массив) | Средний случай | Худший случай  (по убыванию  отсортированный массив) |
|  |  |  |

# Описание алгоритма Краскала: void Kruskal()

Алгоритм Краскала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа, общая идея алгоритма состоит в следующем:

1. Упорядочить все ребра по возрастанию (неубыванию) весов (функция сортировки: void QuickSort()).
2. Присвоить вершинам графа различные номера (раскраска вершин в разные цвета, выполняется на этапе инициализации графа в функции void initGraphFromFileAngry()).
3. ЦИКЛ ПОКА не кончились ребра

Рассмотреть очередное ребро

ЕСЛИ концы ребра окрашены в разные цвета, ТО

добавить ребро в стягивающее дерево и перекрасить все вершины (в старом графе), номер которых совпадает с большим номером из концов этого ребра

КОНЕЦ ЕСЛИ

КОНЕЦ ЦИКЛА

## Обход графа

**Алгоритм поиск в ширину**. Пусть зафиксирована начальная вершина . Рассматриваем все смежные с ней вершины ,,...,. Затем рассматриваем смежные вершины каждой из рассмотренных вершин ,,...,, и т.д. Так будут перебраны все вершины графа и поиск закончится.

**Алгоритм поиск в глубину**. Пусть зафиксирована начальная вершина . Выберем смежную с ней вершину . Затем для вершины  выбираем смежную с ней вершину из числа еще не выбранных вершин и т.д.: если мы уже выбрали вершины ,,...,, то следующая вершина выбирается смежной с вершиной  из числа невыбранных. Если для вершины  такой вершины не нашлось, то возвращаемся к вершине  и для нее ищем смежную среди невыбранных. При необходимости возвращаемся назад и т.д. Так будут перебраны все вершины графа и поиск закончится.

В данной реализации используется обход графа в глубину: в каждой итерации цикла рассматривая и перекрашивая ребро, переходим к следующему ребру номер одного из вершин которого совпадает с большим номером из концов предыдущего ребра.

## 4.2 Оценка сложности алгоритма

До начала работы алгоритма необходимо читать граф из файла, что требует времени, где – число ребер в графе. После нужно отсортировать рёбра по весу, это требует времени. После чего компоненты связности хранятся в виде системы непересекающихся множеств. Все операции в таком случае займут константное время , таким образом, общее время работы алгоритма Краскала можно принять за .

# 5. Сортировка ребер в лексикографическом порядке: void sortEdgesОfDisjointSetAtLex()

Лексикографическая сортировка ребер минимального остовного дерева в функии void sortEdgesОfDisjointSetAtLex() проходит в два этапа:

1. Сортировка вершинам внутри класса ребра (class Edge) (чтобы сортировать лексикографически две смежные вершины между собой).
2. Сортировка пузырьком, по первым вершинам класса ребра в массиве std::vector<Edge> disjointSet .

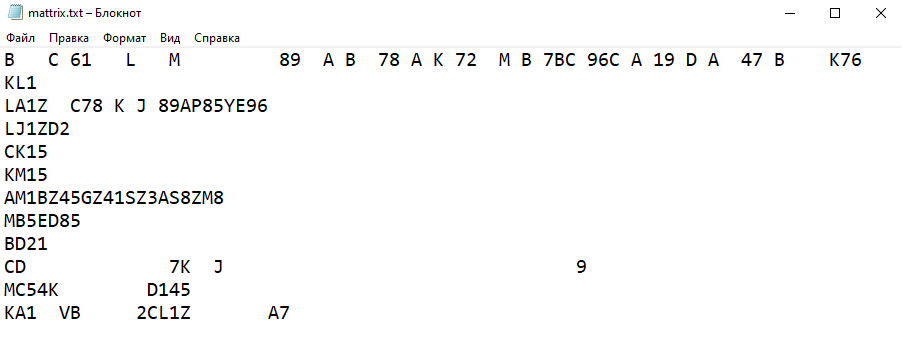
# 6. Оценка общей временной сложности

Временная сложность работы всей программы представлена в таблице.

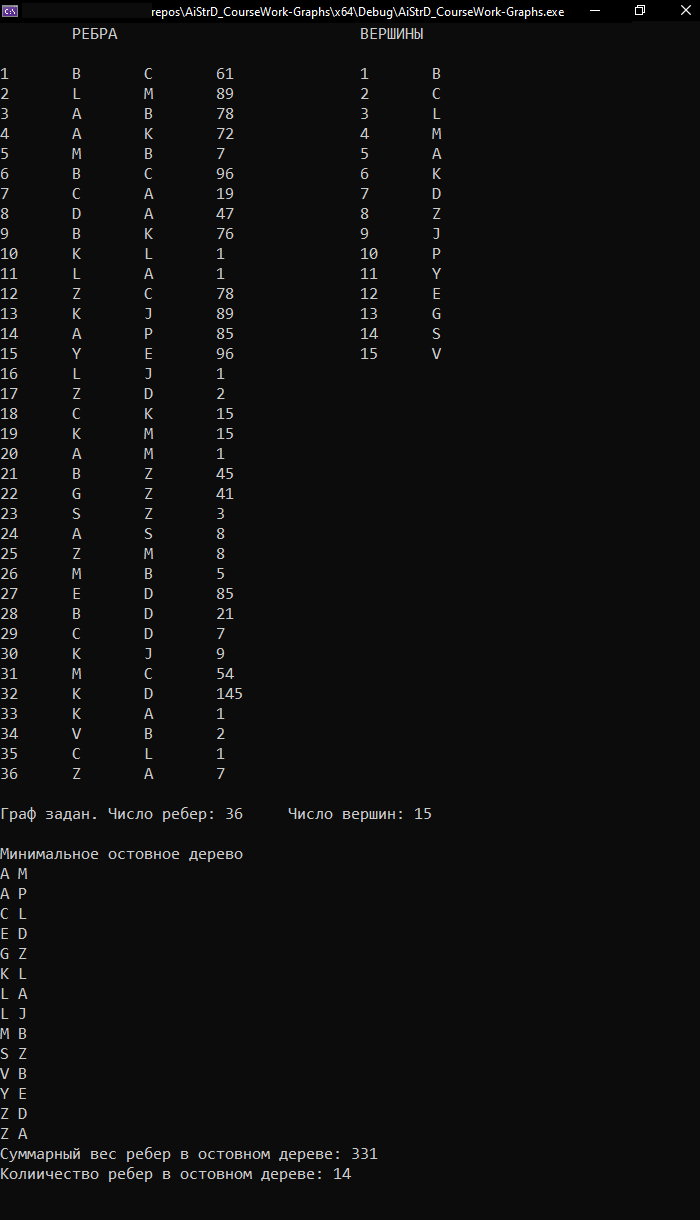
|  |  |  |
| --- | --- | --- |
| Функция | | Временная сложность |
| Создание графа (массив рёбер)  void initGraphFromFileAngry() | |  |
| Обработка | Сортировка ребер по возрастанию весов  void QuickSort() |  |
| Создание минимального остовного дерева  (Disjoint-set / Union-find Forest)  void Kruskal() |  |
| Сортировка ребер в лексикографическом порядке  void sortEdgesОfDisjointSetAtLex() |  |
| Вывод минимального остовного дерева  void printUnionFindForest() | |  |

# 7. Результаты решения задачи

Программа выполняет перед собой ставленную задачу. Результаты выполнения программы с произвольными входными данными показанна на рисунках ниже (см. рис. 1, рис. 2).



*Рис. 1. Пример произвольных входных данных*



*Рис. 2. Результат выполнения программы с произвольными входными данными*

# Заключение

При выполнении курсовой работы были получены практические навыки работы с графами на языке программирования C++, была продемонстрировна знание следующих вопросов: сортировка, обход графов, хранение графов и   
построение системы непересекающихся множеств. Также был закодирован оптимальный алгоритм, для поиска минимального остовного дерева. Временная сложность обработки алгоритма равна .

# Список использованных источников

1. Дискретная математика: учебник для студентов вузов / С. Н. Поздняков, С. В. Рыбин. Москва: Издательский центр «Академия», 2008. 448 с.
2. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих./ А.Бхаргава. СПб.: Питер, 2019. 288 с.
3. Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е издание / Р.Лафоре СПб., Питер, 2016. 928 с.

# Приложение А: Исходный код программы

* main.cpp – исходный код программы
* graph.h – заголовочный файл с классами
* graph.cpp – функии-члены класса
* mattrix.txt – файл с тестом

===================== main.cpp =========================

/\*

This program was created for a course project for taking credit

for the subject of Algorithms and Data Structures in LETI at CyberSecurity department

(C) Arthur Nersisyan 11/2019

vk @arthurnersisyan

\*/

// this file contains only main function for this program

#include "graph.h"

int main() {

setlocale(LC\_ALL, "Russian");

Graph graf;

graf.initGraphFromFileAngry();

graf.QuickSort();

//graf.BubbleSort();

graf.Kruskal();

graf.sortEdgesОfDisjointSetAtLex();

graf.printUnionFindForest();

std::cin.get();

return 0;

}

===================== graph.h =========================

#pragma once

#include <iostream>

#include <string>

#include <vector>

#include <fstream>

const unsigned short MIN\_DIST = 0; // Мин дистанция (вес) ребера

const unsigned short MAX\_DIST = 1023;

const int MAX\_NODES = 50; // Максимальное кол-во вершин

const int MAX\_EDGES = 1225; // Полный граф с N вершинами имеет N(N-1)/2 рёбер

// класс ребра

class Edge

{

private:

std::string FirstV, SecondV; // вершины между которыми связь

int FirstVcolor, SecondVcolor; // цвета вершин

unsigned short dist; // вес ребра с диапазоном 0-1023, провер. при вводе

public:

Edge() : dist(0), FirstVcolor(0), SecondVcolor(0), FirstV(""), SecondV("") {}

void setDist(unsigned short dist) {

this->dist = dist;

}

void setFirstV(std::string FirstV) {

this->FirstV = FirstV;

}

void setSecondV(std::string SecondV) {

this->SecondV = SecondV;

}

void setFirstVcolor(int FirstVcolor) {

this->FirstVcolor = FirstVcolor;

}

void setSecondVcolor(int SecondVcolor) {

this->SecondVcolor = SecondVcolor;

}

unsigned short getDist() const {

return dist;

}

std::string getFirstV() const {

return FirstV;

}

std::string getSecondV() const {

return SecondV;

}

int getFirstVcolor() const {

return FirstVcolor;

}

int getSecondVcolor() const {

return SecondVcolor;

}

};

// класс Графа

class Graph

{

private:

std::vector<Edge> edges; // ГРАФ - по сути массив объектов класса Edge (ребер с вершинами)

std::vector<Edge> disjointSet; // Минимальное остовное дерево Disjoint-set/Union-find Forest

std::vector<std::string> collection; // Колекция вершин

// вспомогательные функции-члены

int colorOf(std::string str); // принимает строку возвращает ASCII код первого элемента

int maxOf(int a, int b); // возвращает максимум из аргументов

int minOf(int a, int b); // возвращает минимум из аргументов

bool isABC(char str); // определяет являеться ли символ буквой

bool isInCollection(std::string str); // принимает символ (строку) определяет есть

ли такое в коллекции вершин

void quickSort(std::vector<Edge>& arr, size\_t first, size\_t last);

public:

Graph() : edges(0), disjointSet(0) {}

void initGraphFromFileAngry(); // Инициализация ГРАФА, чтение из файла

void QuickSort(); // сортировка ребер по возрастанию весов

void Kruskal(); // алгоритм Краскала

void sortEdgesОfDisjointSetAtLex(); // Сортировка вершины в лекиикографичеком порядке

void printUnionFindForest(); // Вывод минимального остогого дерева в консоль

};

===================== graph.cpp =========================

#include "graph.h"

/\*

А. Инициализация ГРАФА +

Б. Краскал +

1. сортировка по возрастанию весов ребер //quickSort +

2. раскраска вешин разными цветами +

3. Крутится по циклу в методе Kruskal() +

В. Сортировать вершины в лекиикографичеком порядке +

Г. Считать сумму вес Минимального остового дерева +

Д. Вывести МОД и общий вес +

\*/

//=============================== ИНИЦИАЛИЗАЦИЯ ГРАФА =======================

/\* Входные данные: любая текстовая последовательность -- без пробелов, с пробелами, как желайте

ОГРАНИЧЕНИЕ вершины и вес должны быть хотя бы на одной строке

открвает файл, считает граф, записывает в класс ребер, каждое созданное ребро добавляет в граф

\*/

void Graph::initGraphFromFileAngry() {

std::string filePath = "mattrix.txt";

std::string str, getA, getB, getWeight;

int tempWeight = 0;

std::ifstream file;

file.open(filePath);

if (!file.is\_open())

{

std::cout << "Oшибка! Не могу открыть файл..." << std::endl;

}

else {

while (!file.eof())

{

str = "";

std::getline(file, str);

for (size\_t i = 0; i < str.length(); i++) {

getA = ""; getB = ""; getWeight = "";

while (str[i] == ' ') {

i++; // перешагаем пробелы если они есть

}

if (isABC(str[i])) {

getA = str[i];

while (str[i + 1] == ' ') {

i++;

}

if (isABC(str[i + 1])) { // i+1 безопасно, т.к. если есть 1-ая вершина, то 2-ая точно есть

getB = str[i + 1];

while (str[i + 2] == ' ') {

i++;

}

if (!isABC(str[i + 2])) {

// число нужно проверить, т.к. может быть больше одного знака

// цикл пока не встретили букву, пробел и пока не кончилась строка

for (size\_t j = i + 2; !isABC(str[j]) && str[j] != ' ' && j < str.length(); j++) {

getWeight += str[j]; // очередной знак должен быть цифрой, собираем число по символьно

}

i += getWeight.length(); // перешагаем длину веса ребра //все число // тоже безопасно

if (getA.length() && getB.length() && getWeight.length()) { // если все нашли

tempWeight = std::stoi(getWeight); // извлекаем число из строки в тип int\_32

if (MIN\_DIST <= tempWeight && tempWeight <= MAX\_DIST) {

Edge temporay; // создаем, задаем, добавлем в граф

temporay.setFirstV(getA);

temporay.setSecondV(getB);

temporay.setFirstVcolor(colorOf(getA)); // цвет вершин ASCII-код символа данной вершины

temporay.setSecondVcolor(colorOf(getB));

temporay.setDist((unsigned short)tempWeight); //безопасно т.к tempWeight из [0;1023] см if

edges.push\_back(temporay);

if (!isInCollection(temporay.getFirstV())) {

collection.push\_back(temporay.getFirstV());

}

if (!isInCollection(temporay.getSecondV())) {

collection.push\_back(temporay.getSecondV());

}

if (edges.size() > MAX\_EDGES) {

std::cout << "Ошибка! Число ребер должна быть меньше либо равно 1225 !" << std::endl;

throw;

}

if (collection.size() > MAX\_NODES) {

std::cout << "Ошибка! Число ребер вершин должна быть меньше либо равно 50 !" << std::endl;

throw;

}

}

else {

std::cout << "Ошибка! Значение веса ребра не входит в диапазон [0; 1023]." << std::endl;

throw;

}

}

}

}

}

}

}

file.close();

std::cout << "\tРЕБРА" << "\t\t\t\tВЕРШИНЫ" << std::endl << std::endl;

for (size\_t i = 0; i < edges.size(); i++) {

std::cout << i + 1 << "\t" << edges[i].getFirstV() << "\t" /\*<< edges[i].getFirstVcolor() << "\t"\*/ << edges[i].getSecondV() << "\t" /\*<< edges[i].getSecondVcolor() << "\t"\*/ << edges[i].getDist();

if (i < collection.size()) {

std::cout << "\t\t" << i+1 << "\t" << collection[i] << std::endl;

}

else {

std::cout << std::endl;

}

}

}

std::cout << std::endl << "Граф задан. Число ребер: " << edges.size()

<< "\tЧисло вершин: " << collection.size() << std::endl;

}

/\*

вспомогающая функция, отвечает за количество вершин графа, пр каждом добавлении новых вершин

ищем в колекции если есть, не включаем, если нету вкулючаем в колекцию

\*/

bool Graph::isInCollection(std::string str) {

for (size\_t i = 0; i < collection.size(); i++) {

if (collection[i] == str || collection[i] == str) {

return true;

}

}

return false;

}

/\*

вспомогающая функция, определяет являеться ли символ буквой

чтобы не писать такое в if-else конструкции

\*/

bool Graph::isABC(char str) {

return (str >= 'A' && str <= 'Z') || (str >= 'a' && str <= 'z');

}

/\* ракраска графа -- вспомогательная функиия-метод инициализации графа

принимает строку, возвращает ASCII-код первого символа

\*/

int Graph::colorOf(std::string str) {

int color;

color = (int)str[0];

return color;

}

// СОРТИРОВКА ПО ВОЗРАТТАНИЮ весов ребер Графа

void Graph::QuickSort() {

quickSort(edges, 0, edges.size() - 1);

/\*

std::cout << edges.size() << std::endl;

for (size\_t i = 0; i < edges.size(); i++)

{

std::cout << i << " " << edges[i].getFirstV() << " " << edges[i].getFirstVcolor() << " " << edges[i].getSecondV() << " " << edges[i].getSecondVcolor() << " " << edges[i].getDist() << std::endl;

}

\*/

//std::cout << std::endl << "Сортировка ребер по возрастанию весов успешно завершена!" << std::endl;

}

void Graph::quickSort(std::vector<Edge> &arr, size\_t first, size\_t last) {

if (first < last)

{

int left = first, right = last;

Edge middle = arr[(left + right) / 2];

do

{

while (arr[left].getDist() < middle.getDist()) left++;

while (arr[right].getDist() > middle.getDist()) right--;

if (left <= right)

{

Edge tmp = arr[left];

arr[left] = arr[right];

arr[right] = tmp;

left++;

right--;

}

} while (left <= right);

quickSort(arr, first, right);

quickSort(arr, left, last);

}

}

/\*

вспомогательные функции к Алгортму Краскала

Возвращают соответственно максимум и минимум из двух значений

\*/

int Graph::maxOf(int a, int b) {

if (a >= b) {

return a;

}

else {

return b;

}

}

int Graph::minOf(int a, int b) {

if (a < b) {

return a;

}

else {

return b;

}

}

// РАЛИЗАЦИЯ АЛГОРИТМА КРАСКАЛА

void Graph::Kruskal() {

for (size\_t i = 0; i < edges.size(); i++) { // ЦИКЛ-ПОКА не кончились ребра, рассматреть очередное ребро

if (edges[i].getFirstVcolor() != edges[i].getSecondVcolor()) {

// ЕСЛИ концы ребра окрашены в разные цвета, ТО

disjointSet.push\_back(edges[i]);//добавить ребро в стягивающее дерево

int maxcolor = maxOf(edges[i].getFirstVcolor(), edges[i].getSecondVcolor());

int mincolor = minOf(edges[i].getFirstVcolor(), edges[i].getSecondVcolor());

edges[i].setFirstVcolor(maxcolor);

edges[i].setSecondVcolor(maxcolor);

for (size\_t j = 0; j < edges.size(); j++) { // и перекрасить все вершины,

if (edges[j].getFirstVcolor() == mincolor) {

// цвет которых совпадает с цветoм

edges[j].setFirstVcolor(maxcolor); // одного конца ребра (берем большую)

}

if (edges[j].getSecondVcolor() == mincolor) {

// перекрасить все отстальное дерево

edges[j].setSecondVcolor(maxcolor);

}

}

}

}

}

// СОРТИРОВКА РЕБЕР В ЛЕКСИКОГРАФИЧЕСКОМ ПОРЯДКЕ

/\*

сортировка вершин каждого ребра лексикографически

все ребра, которые описаны как "D A" перевести в "A D"

\*/

void Graph::sortEdgesОfDisjointSetAtLex() {

for (size\_t i = 0; i < disjointSet.size(); i++) {

if (colorOf(disjointSet[i].getFirstV()) > colorOf(disjointSet[i].getSecondV())) {

std::string temp; //swap

temp = disjointSet[i].getFirstV();

disjointSet[i].getFirstV() = disjointSet[i].getSecondV();

disjointSet[i].getSecondV() = temp;

}

}

for (size\_t i = 0; i < disjointSet.size(); i++) {

for (size\_t j = disjointSet.size() - 1; j > i; j--) {

if (colorOf(disjointSet[j].getFirstV()) < colorOf(disjointSet[j - 1].getFirstV())) {

Edge temp; //swap

temp = disjointSet[j];

disjointSet[j] = disjointSet[j - 1];

disjointSet[j - 1] = temp;

}

}

}

/\*

std::cout << edges.size() << std::endl;

for (size\_t i = 0; i < edges.size(); i++)

{

std::cout << i << " " << edges[i].FirstV << " " << edges[i].FirstVcolor << " "

<< edges[i].SecondV << " " << edges[i].SecondVcolor << " "

<< edges[i].getDist() << std::endl;

}

\*/

//std::cout << std::endl << "Лексикографическая сортировка ребер успешно завершена!" << std::endl;

};

// ВЫВОД МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА и общего веса МОД

void Graph::printUnionFindForest()

{

std::cout << std::endl << "Минимальное остовное дерево" << std::endl;

int sumOfEdgeWeight = 0;

for (size\_t i = 0; i < disjointSet.size(); i++)

{

std::cout << disjointSet[i].getFirstV() << " "

<< disjointSet[i].getSecondV() << " "

<< /\*disjointSet[i].getDist() <<\*/ std::endl;

sumOfEdgeWeight += disjointSet[i].getDist();

}

std::cout << "Суммарный вес ребер в остовном дереве: " << sumOfEdgeWeight << std::endl;;

std::cout << "Колиичество ребер в остовном дереве: " << disjointSet.size() << std::endl;;

/\*

std::cout << std::endl << "edges...." << std::endl;

for (size\_t i = 0; i < edges.size(); i++)

{

std::cout << i << " " << edges[i].FirstV << " " << edges[i].FirstVcolor << " "

<< edges[i].SecondV << " " << edges[i].SecondVcolor << " "

<< edges[i].getDist() << std::endl;

}

\*/

}

==================== mattrix.txt ========================

B C 61 L M 89 A B 78 A K 72 M B 7BC 96C A 19 D A 47 B K76

KL1

LA1Z C78 K J 89AP85YE96

LJ1ZD2

CK15

KM15

AM1BZ45GZ41SZ3AS8ZM8

MB5ED85

BD21

CD 7K J 9

MC54K D145

KA1 VB 2CL1Z A7