**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: **Нахождение остовного дерева с помощью алгоритма Краскала (Крускала)**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1374 |  | Дюков Н. В. |
| Преподаватель |  | Пелевин М. С. |

Санкт-Петербург

2022

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Дюков Николай | | |
| Группа 1374 | | |
| ***Тема работы:*** Реализация алгоритма поиска минимального остова на основе Алгоритма Краскала (Крускала).  Исходные данные:  Разработать приложение для поиска минимального остова дерева с использованием алгоритма Краскала. | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Основная часть», «Заключение», «Список использованных источников», «Приложение A ̶ исходный код». | | |
| Предполагаемый объем пояснительной записки:  Не менее 10 страниц. | | |
| Дата выдачи задания: 04.12.2022 | | |
| Дата сдачи реферата: 22.12.2022 | | |
| Дата защиты реферата: 22.12.2022 | | |
| Студент |  | Дюков Н. В. |
| Преподаватель |  | Пелевин М.С. |

**Аннотация**

***Алгоритм Краскала*** — это алгоритм минимального остовного дерева, что принимает граф в качестве входных данных и находит подмножество ребер этого графа, который формирует дерево, включающее в себя каждую вершину, а также имеет минимальную сумму весов среди всех деревьев, которые могут быть сформированы из графа.

***Задача:*** разработать программу для нахождения минимального остова графа. В проекте описывается теоретическая составляющая: реализация программы для построения системы непересекающихся множеств (СНМ), а также описание выбранных алгоритмов сортировки, обхода графов и СНМ.

В отчете присутствует исходный код программы.

**SUMMARY**

***Krascal's algorithm*** is a minimal spanning tree algorithm that takes a graph as input and finds a subset of the edges of this graph that forms a tree that includes each vertex, and also has the minimum sum of weights among all trees that can be formed from the graph.

**Task**: to develop a program for finding the minimum backbone of the graph. The project describes the theoretical component: the implementation of a program for building a system of disjoint sets (SNM), as well as a description of the selected sorting algorithms, graph traversal and SNM.

The report contains the source code of the program.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Алгоритм Краскала | 6 |
| 1.1. | Понятие остовного дерева | 6 |
| 1.2. | Описание алгоритма Краскала | 6 |
| 2. | Реализация алгоритма | 8 |
| 2.1. | Описание класса edgeArray | 8 |
| 2.2. | Описание класса DynArray | 8 |
|  |
| 3.  4. | Пузырьковая сортировка  Обход графа | 9  11 |
| 5. | Система непересекающихся множеств | 14 |
|  | Заключение | 15 |
|  | Список использованных источников | 16 |
|  | Приложение А. Исходный код | 17 |
|  |  |  |

**введение**

**Задача** для данной курсовой работы: реализовать алгоритм поиска минимального остова на основе алгоритма Краскала (Крускала).

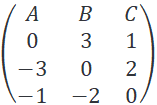
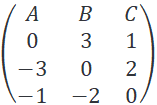
**Цель работы**

Продемонстрировать знания следующих вопросов:

* сортировка
* обход графов (в глубину и в ширину)
* хранение графов (списки смежности, матрицы смежности, инцидентности)
* построение системы непересекающихся множеств

Исходные данные:

Любой текстовый файл, содержащий матрицу смежности графа в виде



где первая строка содержит через пробел список всех рёбер, за которым следует матрица смежности. В матрице значение 0 стоит, если ребра между вершинами нет, положительный вес, если ребро идёт из вершины для этой строки, и отрицательное, когда ребро входит в эту вершину.

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 1 до 1023 включительно.

* 1. **Алгоритм Краскала**

**1.1. Понятие остовного дерева**

**Остовное дерево (англ. spanning tree) графа G=(V,E)** — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины. Неформально говоря, остовное дерево получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Остовное дерево включает в себя все n вершин исходного графа и содержит n -1 ребро.

**Остовное дерево (англ. spanning tree) графа G=(V,E) —** ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

**1.2. Описание Алгоритма Краскала.**

**Алгоритм Краскала**— это алгоритм минимального остовного дерева, что принимает граф в качестве входных данных и находит подмножество ребер этого графа, который формирует дерево, включающее в себя каждую вершину, а также имеет минимальную сумму весов среди всех деревьев, которые могут быть сформированы из графа.

В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появления в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса.

Программа на вход принимает список ребер. Но есть ещё множество вариантов хранения графа:

● Списки ребер

● Списки смежности

● Матрица смежности

● Матрица инцидентности

Алгоритм реализации алгоритма Краскала:

1. Сортируем имеющиеся рёбра по весу.
2. Создаём новое множество и добавляем в него первое ребро.
3. Затем пытаемся добавить каждое новое ребро в имеющееся множество, но если возникает цикл то это ребро пропускаем, поскольку циклов не может быть.
4. Итоговое множество рёбер и есть искомое минимальное остовное дерево.

Сложность алгоритма: O(n\*log(n)), где n – количество рёбер в графе

* 1. **РЕАЛИЗАЦИЯ АЛГОРИТМА**

**2.1. Описание класса edgeArray**

Класс edgeArray - представляет собой массив элементов типе edge (ребра графа и информация об их вершинах), основной класс приложения.

Операции с массивом вершин

void add(edge v) - добавление нового элемента в массив

edge get(int ind) - функция возвращает элемент по индексу

int getf(int ind) - функция возвращает начальную вершину ребра по индексу

int gett(int ind) - функция возвращает конечную вершину ребра по индексу

int getw(int ind) - функция возвращает вес ребра по индексу

**2.2.** **Описание класса** **DynArray**.

Класс DynArray представляет собой реализацию динамического массива. Поля класса: массив, на котором основывается реализация списка, длина списка, вместительность массива.

void add(const int s); - добавить вершину

int size(); - размер массива

void clear(); - очистить массив

int findset(int ind); - возвращает идентификатор множества, которому принадлежит элемент

void unionset(int f, int t); - объединение двух множества

* 1. **Пузырьковая сортировка**

Для реализации алгоритма была выбрана сортировка пузырьком, главным образом из-за простоты её реализации.

Время работы алгоритма:

* Худшее: O(n^2)
* Лучшее: O(n)
* В среднем: О(n^2)

Сортировка простыми обменами, сортировка пузырьком (англ. bubble sort) — один из квадратичных алгоритмов сортировки. Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами. За каждый проход по массиву как минимум один элемент встает на свое место, поэтому необходимо совершить не более n−1 проходов, где n размер массива, чтобы отсортировать массив.

for (int i = 0; i < edgearr.\_size(); i++) // пузырьковая сортировка весов рёбер графа

{

bool flag = true;

for (int j = 0; j < edgearr.\_size() - i - 1; j++)

{

if (edgearr.getw(j) > edgearr.getw(j + 1))

{

flag = false;

tmp = edgearr.get(j);

edgearr.set(j, edgearr.get(j + 1));

edgearr.set(j + 1, tmp);

}

}

if (flag) {

break;

}

}

int ParF, ParS;

for (int i = 0; ((i < edgearr.\_size()) or (MOD.\_size() != (nodes.size() - 1))); i++) {

ParF = parents.findset(edgearr.getf(i));

ParS = parents.findset(edgearr.gett(i));

if (ParF != ParS)

{

MOD.add(edgearr.get(i));

parents.unionset(ParF, ParS);

MODweight += edgearr.getw(i);

}

}

for (int i = 0; i < MOD.\_size(); i++) // пузырьковая сортровка рёбер в минимальном остовном дереве по возрастанию первых вершин

{

bool flag = true;

for (int j = 0; j < MOD.\_size() - i - 1; j++)

{

if (MOD.getf(j) > MOD.getf(j + 1))

{

flag = false;

edge tmp = MOD.get(j);

MOD.set(j, MOD.get(j + 1));

MOD.set(j + 1, tmp);

}

}

}

* 1. **ОБХОД ГРАФА**

***Обход графа*** — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами

**Обход графа в глубину**

*Поиск в глубину (англ. depth-first search, DFS)* – это рекурсивный алгоритм обхода вершин графа. Если метод поиска в ширину производился симметрично (вершины графа просматривались по уровням), то данный метод предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность дальнейшего продвижения, означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина). Отсутствие последнего свидетельствует об одной из двух возможных ситуаций: либо все вершины графа уже просмотрены, либо просмотрены все те, что доступны из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Сложность: О(V + n), V- кол-во вершин, n – кол-во ребер

// Обход в глубину depth-first search

void App::dfsAlg() {

if (matrix != NULL) {

visited = new int[msize + 1];

// Обнуляем массив посещений

for (int i = 0; i < msize; i++) {

visited[i] = 0;

}

cout << "DFS обход: ";

dfs(0);

cout << endl;

}

}

void App::dfs(const int v) {

// Посетили саму вершину

visited[v] = 1;

cout << vertices[v] << " ";

for (int j = 0; j < msize; j++) {

// Если есть наследник и он её не посещен

if ((matrix[v][j] > 0) && (!visited[j])) {

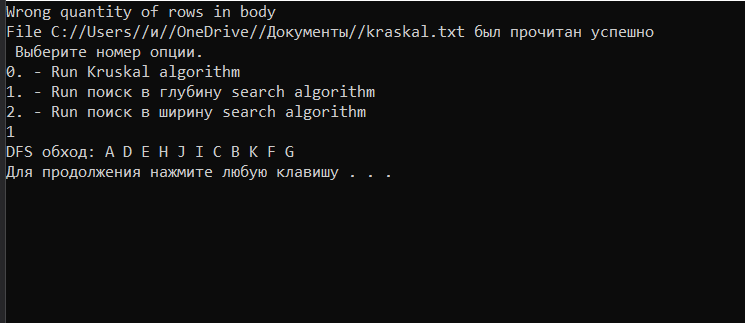
// посещаем наследника

dfs(j);

}

}

}



**Обход графа в ширину**

*Поиск в ширину (обход по уровням)* – один из алгоритмов обхода графа. Метод лежит в основе некоторых других алгоритмов близкой тематики. Поиск в ширину подразумевает поуровневое исследование графа: вначале посещается корень – произвольно выбранный узел, затем – все потомки данного узла, после этого посещаются потомки потомков и т.д. Вершины просматриваются в порядке возрастания их расстояния от корня.

Пусть задан граф G=(V, E) и корень s, с которого начинается обход. После посещения узла s, следующими за ним будут посещены смежные с s узлы (множество смежных с s узлов обозначим как q; очевидно, что q⊆V, то есть q – некоторое подмножество V). Далее, эта процедура повториться для вершин смежных с вершинами из множества q, за исключением вершины s, т. к. она уже была посещена. Так, продолжая обходить уровень за уровнем, алгоритм обойдет все доступные из s вершины множества V. Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения наличествующего условия.

Сложность: О(V + n), V- кол-во вершин, n – кол-во ребер

// Обход в ширину breadth-first search

void App::bfsAlg() {

if (matrix != NULL) {

visited = new int[msize + 1];

// Обнуляем массив посещений

for (int i = 0; i < msize; i++) {

visited[i] = 0;

}

cout << "BFS обход: ";

bfs(0);

cout << endl;

}

}

void App::bfs(const int v) {

// Очередь пройденных вершин

QueueArray\* q = new QueueArray();

// Индекс вершины

int v\_ind;

// Посетили саму вершину

q->push(v);

cout << vertices[v] << " ";

visited[v] = 1;

// пока очеред не пуста

while (!q->empty()) {

// извлечь вершину из гловы очереди

v\_ind = q->front();

q->pop();

// всех связный первого уровня непосещенных посещаем и в очередь

for (int j = 0; j < msize; j++) {

if ((matrix[v\_ind][j] > 0) && (!visited[j])) {

visited[j] = 1;

q->push(j);

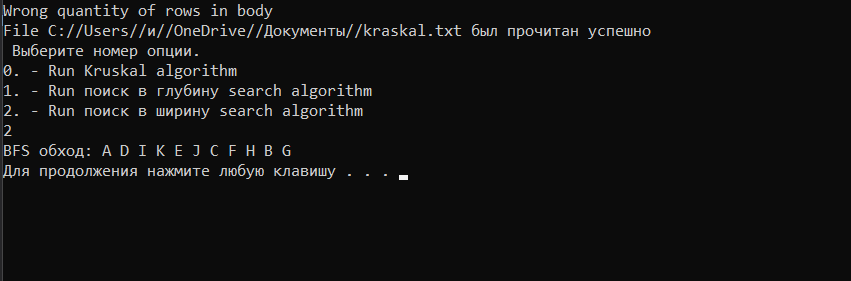
cout << vertices[j] << " ";

}

}

}

}



* 1. **Система непересекающихся множеств**

**Система непересекающихся множеств** (DSU, *disjoint set unio*n) — структура данных, которая позволяет администрировать множество элементов, разбитое на непересекающиеся подмножества. При этом каждому подмножеству назначается его представитель — элемент этого подмножества. Абстрактная структура данных определяется множеством двух операций: { Uni\displaystyle \{\mathrm {Union} ,\mathrm {Find} ,\mathrm {MakeSet} \}}Union, Find.

**findset (int ind)** — возвращает идентификатор множества, которому принадлежит элемент v. В качестве идентификатора мы будем выбирать один элемент из этого множества — представителя множества. Гарантируется, что для одного и того же множества представитель будет возвращаться один и тот же.

**unionset (int f, int t)** — объединяет два множества, в которых лежат элементы, в одно новое.

**заключение**

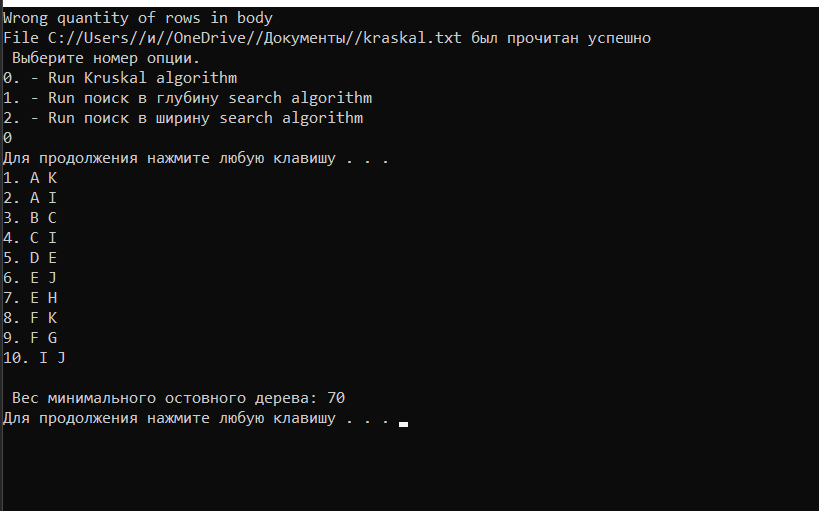
При выполнении курсовой работы были получены практические навыки работы с графами, используя язык программирования, было продемонстрировано знание следующих вопросов: сортировка, обход графов, хранение графов и   
построение системы непересекающихся множеств. Также был закодирован оптимальный алгоритм, для поиска минимального остовного дерева.

**список использованных источников**

1. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. / А.Бхаргава. СПб.: Питер, 2019. 288 с.
2. Электронный источник, Хабр, URL: <https://habr.com/ru/post/204600/>
3. Электронный источник, proginfo, URL: <https://proginfo.ru/>
4. Электронный источник vitmo, URL: [Остовные деревья: определения, лемма о безопасном ребре — Викиконспекты (ifmo.ru)](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D1%81%D1%82%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D1%8C%D1%8F:_%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F,_%D0%BB%D0%B5%D0%BC%D0%BC%D0%B0_%D0%BE_%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D0%BC_%D1%80%D0%B5%D0%B1%D1%80%D0%B5)

**приложение А**

**Пример работы алгоритма крускала**



**Исходный код**

**app.cpp**

#include "App.h"

// Конструктор

App::App(const string path) {

filepath = path;

}

// Деструктор

App::~App() {

}

// Меню пользователя

void App::showMenu() {

cout << " Выберите номер опции." << endl;

cout << "0. - Run Kruskal algorithm" << endl;

cout << "1. - Run поиск в глубину search algorithm" << endl;

cout << "2. - Run поиск в ширину search algorithm" << endl;

int option;

cin >> option;

switch (option) {

case 0:

run\_proc = false;

break;

case 1:

dfsAlg();

break;

case 2:

bfsAlg();

break;

case 3:

printMatrix();

break;

default:

cout << "Неизвестный вариант" << endl;

break;

}

}

// Основной цикл приложения

int App::run() {

while (run\_proc) {

readMatrixFromFile();

showMenu();

system("pause");

}

return 0;

}

// Создание массива для матрицы

void App::initMatrix() {

matrix = new int\* [msize];

for (int i = 0; i < msize; i++) {

matrix[i] = new int[msize];

}

for (int i = 0; i < msize; i++) {

for (int j = 0; j < msize; j++) {

matrix[i][j] = 0;

}

}

}

// Очистка массивов вершин и матрицы

void App::initArrays() {

delete[] vertices;

vertices = NULL;

for (int i = 0; i < msize; i++) {

delete[] matrix[i];

}

delete[] matrix;

matrix = NULL;

msize = 0;

}

// Добавить вершину

void App::addVertex(const string val) {

string\* newVert = new string[msize + 1];

for (int i = 0; i < msize; i++) {

newVert[i] = vertices[i];

}

newVert[msize] = val;

delete[] vertices;

vertices = newVert;

}

// Заполнить значение в матрице

void App::setMatrix(const int row, const int col, const int val) {

matrix[row][col] = val;

}

// Чтение матрицы из файла

void App::readMatrixFromFile() {

ifstream f\_in;

string s, token;

int i, j, len, val, row, col;

bool err;

// Инит массивов

initArrays();

// Считать строку с заголовками

f\_in.open(filepath);

getline(f\_in, s);

len = s.length();

//Обработать вершины

err = false;

i = 0;

token = "";

while (!err && i < len) {

if ((s[i] >= 'A' && s[i] <= 'z') || (s[i] >= '0' && s[i] <= '9')) {

// Новая вершина

token += s[i];

}

else if (s[i] == ' ' || '\t') {

if (token != "") {

// Записать в массив вершин

addVertex(token);

msize++;

token = "";

if (msize > TOP\_COUNT\_MAX) {

err = true;

cout << "Неправильное количество вершин" << endl;

}

}

}

else {

err = true;

cout << "Неправильные символы в заголовке" << endl;

}

i++;

if (i == len && token != "") {

// Последняя вершина

addVertex(token);

msize++;

token = "";

if (msize > TOP\_COUNT\_MAX) {

err = true;

cout << "Неправильное количество вершин" << endl;

}

}

}

// Обработка матрицы

if (!err) {

initMatrix();

}

i = 0;

token = "";

row = 0;

col = 0;

while (!err && getline(f\_in, s)) {

len = s.length();

token = "";

i = 0;

val = 0;

col = 0;

if (row == msize) {

err = false;

cout << "Wrong quantity of rows in body" << endl;

}

while (!err && i < len) {

if (col == msize) {

err = true;

cout << "Wrong quantity of cols in body" << endl;

}

if (s[i] >= '0' && s[i] <= '9') {

// Новое значение

token += s[i];

val \*= 10;

val += s[i] - '0';

}

else if (s[i] == ' ' || '\t') {

if (token != "") {

// Записать значение

if (val >= EDGE\_VAL\_MIN && val <= EDGE\_VAL\_MAX) {

setMatrix(row, col, val);

if ((row > col) && (matrix[row][col] != matrix[col][row])) {

//err = true;

//cout << "Value row: " << row << " col: " << col << " val: " << val << " isn't symmetrical" << endl;

}

token = "";

col++;

val = 0;

}

else {

err = true;

cout << "Строка с неправильным значением: " << row << " col: " << col << " val: " << val << endl;

}

}

}

else {

err = true;

cout << "Неправильные символы в теле" << endl;

}

i++;

if (i == len && token != "") {

// последнее значение

if (val >= EDGE\_VAL\_MIN && val <= EDGE\_VAL\_MAX) {

setMatrix(row, col, val);

if ((row > col) && (matrix[row][col] != matrix[col][row])) {

//err = true;

//cout << "Value row: " << row << " col: " << col << " val: " << val << " isn't symmetrical" << endl;

}

token = "";

col++;

val = 0;

}

else {

err = true;

cout << "Строка с неправильным значением: " << row << " col: " << col << " val: " << val << endl;

}

}

}

row++;

};

if (!err) {

cout << "File " << filepath << " был прочитан успешно" << endl;

}

f\_in.close();

}

void App::printMatrix() {

if (vertices != NULL && matrix != NULL) {

for (int i = 0; i < msize; i++) {

/\*if (vertices[i].length() > 1) {

cout << vertices[i] << " ";

}

else{

cout << vertices[i] << " ";

}\*/

cout << vertices[i] << '\t';

}

cout << endl;

for (int i = 0; i < msize; i++) {

for (int j = 0; j < msize; j++) {

/\*if (matrix[i][j] > 9) {

cout << matrix[i][j] << " ";

}

else {

cout << matrix[i][j] << " ";

}\*/

cout << matrix[i][j] << '\t';

}

cout << endl;

}

}

};

// Обход в глубину depth-first search

void App::dfsAlg() {

if (matrix != NULL) {

visited = new int[msize + 1];

// Обнуляем массив посещений

for (int i = 0; i < msize; i++) {

visited[i] = 0;

}

cout << "DFS обход: ";

dfs(0);

cout << endl;

}

}

void App::dfs(const int v) {

// Посетили саму вершину

visited[v] = 1;

cout << vertices[v] << " ";

for (int j = 0; j < msize; j++) {

// Если есть наследник и он её не посещен

if ((matrix[v][j] > 0) && (!visited[j])) {

// посещаем наследника

dfs(j);

}

}

}

// Обход в ширину breadth-first search

void App::bfsAlg() {

if (matrix != NULL) {

visited = new int[msize + 1];

// Обнуляем массив посещений

for (int i = 0; i < msize; i++) {

visited[i] = 0;

}

cout << "BFS обход: ";

bfs(0);

cout << endl;

}

}

void App::bfs(const int v) {

// Очередь пройденных вершин

QueueArray\* q = new QueueArray();

// Индекс вершины

int v\_ind;

// Посетили саму вершину

q->push(v);

cout << vertices[v] << " ";

visited[v] = 1;

// пока очеред не пуста

while (!q->empty()) {

// извлечь вершину из гловы очереди

v\_ind = q->front();

q->pop();

// всех связный первого уровня непосещенных посещаем и в очередь

for (int j = 0; j < msize; j++) {

if ((matrix[v\_ind][j] > 0) && (!visited[j])) {

visited[j] = 1;

q->push(j);

cout << vertices[j] << " ";

}

}

}

}

**dynArray.cpp**

#include <iostream>

#include "DynArray.h"

DynArray::DynArray()

{

count = 0;

data = NULL;

}

DynArray::~DynArray()

{

delete[] data;

}

void DynArray::add(const int s) {

int\* newData = new int[count + 1];

for (int i = 0; i < count; i++) {

newData[i] = data[i];

}

newData[count] = s;

delete[] data;

data = newData;

count++;

}

void DynArray::set(const int index, const int s) {

data[index] = s;

}

int DynArray::read(const int index) {

return data[index];

}

int DynArray::size() {

return count;

}

void DynArray::clear() {

count = 0;

delete[] data;

data = NULL;

}

int DynArray::findset(int ind) // возвращает идентификатор множества, которому принадлежит элемент

{

if (ind == data[ind])

{

return ind;

}

else

{

return findset(data[ind]);

}

}

void DynArray::unionset(int f, int t) // объединение двух множеств

{

data[f] = data[t];

}

**Main.cpp**

#include <iostream>

#include <fstream>

#include <string>

#include "DynArray.h"

#include "edgeArray.h"

#include "edge.h"

#include "App.h"

#include "queueArray.h"

using namespace std;

string deletespaces(string sub) // удаляет все пробелы из строки, сдвигая все символы, не являющиеся пробелами, влево, а затем стирая лишние

{

string::iterator end\_pos = remove(sub.begin(), sub.end(), '\t');

sub.erase(end\_pos, sub.end());

return sub;

}

int main()

{

setlocale(0, "RUS");

App app("C://Users//и//OneDrive//Документы//kraskal.txt");

app.run();

setlocale(0, "");

string nodes;

ifstream file("C://Users//и//OneDrive//Документы//kraskal.txt");

if (file.is\_open())

{

getline(file, nodes); // извлекаем первую строку из файла и записывем её в nodes

nodes = deletespaces(nodes);

edgeArray edgearr;

edgeArray MOD;

edgearr.clear();

MOD.clear();

int num;

int x = 0;

int maxx = 0;

int y = 0;

int MODweight = 0;

edge temp;

while (!file.eof()) // объекты класса edge будут заноситься в edgeArray

{

file >> num;

if (x > y)

{

if (num > 0) {

temp.from = y;

temp.to = x;

temp.weight = abs(num);

edgearr.add(temp);

}

}

x++;

if (x == nodes.length()) {

if (maxx < x) maxx = x;

x = 0;

y++;

}

}

file.close();

DynArray parents;

parents.clear();

for (int i = 0; i < nodes.size(); i++)

{

parents.add(i);

}

edge tmp;

for (int i = 0; i < edgearr.\_size(); i++) // пузырьковая сортировка весов рёбер графа

{

bool flag = true;

for (int j = 0; j < edgearr.\_size() - i - 1; j++)

{

if (edgearr.getw(j) > edgearr.getw(j + 1))

{

flag = false;

tmp = edgearr.get(j);

edgearr.set(j, edgearr.get(j + 1));

edgearr.set(j + 1, tmp);

}

}

if (flag) {

break;

}

}

int ParF, ParS;

for (int i = 0; ((i < edgearr.\_size()) or (MOD.\_size() != (nodes.size() - 1))); i++) {

ParF = parents.findset(edgearr.getf(i));

ParS = parents.findset(edgearr.gett(i));

if (ParF != ParS)

{

MOD.add(edgearr.get(i));

parents.unionset(ParF, ParS);

MODweight += edgearr.getw(i);

}

}

for (int i = 0; i < MOD.\_size(); i++) // пузырьковая сортровка рёбер в минимальном остовном дереве по возрастанию первых вершин

{

bool flag = true;

for (int j = 0; j < MOD.\_size() - i - 1; j++)

{

if (MOD.getf(j) > MOD.getf(j + 1))

{

flag = false;

edge tmp = MOD.get(j);

MOD.set(j, MOD.get(j + 1));

MOD.set(j + 1, tmp);

}

}

}

for (int i = 0; i < MOD.\_size(); i++)

{

cout << i + 1 << ". " << nodes[MOD.getf(i)] << " " << nodes[MOD.gett(i)] << endl;

}

cout << "\n Вес минимального остовного дерева: " << MODweight << endl;

}

system("pause");

return 0;

}

**queueArray.cpp**

#include "queuearray.h";

QueueArray::QueueArray() {

count = 0;

data = NULL;

}

QueueArray::~QueueArray() {

delete[] data;

}

bool QueueArray::empty() {

return (count == 0);

}

int QueueArray::size() {

return count;

}

int QueueArray::front() {

if (count > 0)

return (data[0]);

else

return -1;

}

void QueueArray::push(const int& val) {

int\* newData = new int[count + 1];

for (int i = 0; i < count; i++) {

newData[i] = data[i];

}

newData[count] = val;

delete[] data;

data = newData;

count++;

}

void QueueArray::pop() {

int\* newData = new int[count - 1];

if (count > 1) {

for (int i = 0; i < count - 1; i++) {

newData[i] = data[i + 1];

}

}

delete[] data;

data = newData;

count--;

}

void QueueArray::print() {

for (int i = 0; i < count; i++) {

cout << data[i];

}

cout << endl;

}

**App.h**

#pragma once

#include <iostream>

#include <fstream>

#include <string>

#include "queuearray.h";

using namespace std;

// Макс кол-во вершин

const int TOP\_COUNT\_MAX = 50;

// Диапазоны значений

const int EDGE\_VAL\_MIN = 0;

const int EDGE\_VAL\_MAX = 1023;

// Основной класс приложения

class App {

public:

App(const string path);

~App();

int run();

private:

// Переменная цикла ввода

bool run\_proc = true;

// Путь к файлу матрицы смежности

string filepath;

// Размер матрицы (симметричная)

int msize = 0;

// Массив вершин (имена)

string\* vertices = NULL;

// Матрица смежности msize \* msize

int\*\* matrix = NULL;

// Массив посещений для обхода

int\* visited = NULL;

// Операции с массивом вершин

void initArrays();

void addVertex(const string val);

// Операции с матрицей смежности

void setMatrix(const int row, const int col, const int val);

void initMatrix();

void readMatrixFromFile();

void printMatrix();

// Меню пользователя

void showMenu();

// Обход в глубину depth-first search

void dfsAlg();

void dfs(const int v);

// Обход в ширину breadth-first search

void bfsAlg();

void bfs(const int v);

};

**dynArray.h**

#pragma once

using namespace std;

class DynArray

{

public:

DynArray();

~DynArray();

void add(const int s);

int read(const int index);

void set(const int index, const int s);

int size();

void clear();

int findset(int ind);

void unionset(int f, int t);

private:

int count;

int\* data;

};

**edge.h**

#pragma once

struct edge

{

int from;

int to;

int weight;

};

**edgeArray.h**

#pragma once

#include <iostream>

#include "edge.h"

class edgeArray // класс представляет собой массив элементов типа edge

{

public:

edgeArray() // конструктор класса

{

size = 0;

arr = NULL;

}

~edgeArray() {

delete[] arr;

}

void add(edge v) // добавление нового элемента в массив

{

edge\* newArr = new edge[size + 1];

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

newArr[i] = arr[i];

}

newArr[size] = v;

delete[] arr;

arr = newArr;

size++;

}

edge get(int ind) // функция возвращает элемент по индексу

{

return arr[ind];

}

int getf(int ind) // функция возвращает начальную вершину ребра по индексу

{

return arr[ind].from;

}

int gett(int ind) // функция возвращает конечную вершину ребра по индексу

{

return arr[ind].to;

}

int getw(int ind) // функция возвращает вес ребра по индексу

{

return arr[ind].weight;

}

void set(int ind, edge e)

{

arr[ind] = e;

}

int \_size() {

return size;

}

void clear() {

delete[] arr;

arr = NULL;

size = 0;

}

void print() {

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

cout << arr[i].from << " - " << arr[i].to << endl;

}

}

private:

int size;

edge\* arr;

};

**queueArray.h**

#pragma once

#include <iostream>

using namespace std;

class QueueArray {

public:

QueueArray();

~QueueArray();

int size();

void push(const int& val);

void pop();

bool empty();

int front();

void print();

private:

int count;

int\* data;

};