**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**Курсовая работа**

**по дисциплине** «Построение и анализ алгоритмов»

**Тема:** «Алгоритм Прима (Исследование)»

Студент гр.6382 \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Воропаев А.О.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фирсов М.А.

Санкт-Петербург

2018

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Воропаев А.О. | | |
| Группа 6382 | | |
| Тема работы: Алгоритм Прима (Исследование) | | |
| Исходные данные:  Случайно сгенерированные или введенные самостоятельно графы. | | |
| Содержание пояснительной записки:  Содержание, введение, заключение, список используемой литературы, приложение (исходный код программы) | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страниц. | | |
| Дата выдачи задания: 03.05.2018 | | |
| Дата сдачи реферата: 04.06.2018 | | |
| Дата защиты реферата: 04.06.2018 | | |
| Студент |  | Воропаев А.О, |
| Преподаватель |  | Фирсов М.А. |

**Аннотация**

В данной работе исследовалась эффективность и скорость работы алгоритма Прима, который строит минимальное остовное дерево в графе. Была разработана программа на языке C++, которая справляется с данной задачей. Также в этой работе проведен анализ алгоритма

Оглавление

[ВВЕДЕНИЕ 5](#_Toc515856544)

[Цель работы. 6](#_Toc515856545)

[Псевдокод: 7](#_Toc515856546)

[Пример работы алгоритма. 8](#_Toc515856547)

[Спецификация. 10](#_Toc515856548)

[Тестирование. 12](#_Toc515856549)

[Исследование. 13](#_Toc515856550)

[Заключение. 16](#_Toc515856551)

[Список литературы. 17](#_Toc515856552)

[ПРИЛОЖЕНИЕ А 18](#_Toc515856553)

[ИСХОДНЫЙ КОД 18](#_Toc515856554)

### ВВЕДЕНИЕ

Целью работы было исследовать алгоритм Прима построения минимального остовного дерева. Для решения этой задачи была написана программа на языке C++, которая анализирует время работы алгоритма на графах с различным количеством вершин. Были построены графики зависимости и таблицы, которые позволяют наглядно оценить скорость работы алгоритма.

### Цель работы.

Изучить алгоритм ЯПД (алгоритм Прима).

**Постановка задачи.**

Реализовать алгоритм ЯПД (алгоритм Прима) и исследовать его на случайно сгенерированных графах.

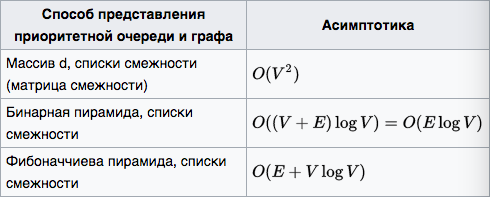
**Теория.**

**Алгоритм Прима (ЯПД->Ярник-Прим-Дейкстра)** — алгоритм поиска [минимального остовного дерева](https://neerc.ifmo.ru/wiki/index.php?title=Лемма_о_безопасном_ребре#.D0.9C.D0.B8.D0.BD.D0.B8.D0.BC.D0.B0.D0.BB.D1.8C.D0.BD.D0.BE.D0.B5_.D0.BE.D1.81.D1.82.D0.BE.D0.B2.D0.BD.D0.BE.D0.B5_.D0.B4.D0.B5.D1.80.D0.B5.D0.B2.D0.BE) во взвешенном [неориентированном связном графе](https://neerc.ifmo.ru/wiki/index.php?title=Основные_определения_теории_графов#.D0.9D.D0.B5.D0.BE.D1.80.D0.B8.D0.B5.D0.BD.D1.82.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.BD.D1.8B.D0.B5_.D0.B3.D1.80.D0.B0.D1.84.D1.8B).

Данный алгоритм очень похож на [алгоритм Дейкстры](https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Дейкстры). Будем последовательно строить поддерево Fответа в графе G, поддерживая [приоритетную очередь](https://neerc.ifmo.ru/wiki/index.php?title=Дискретная_математика,_алгоритмы_и_структуры_данных#.D0.9F.D1.80.D0.B8.D0.BE.D1.80.D0.B8.D1.82.D0.B5.D1.82.D0.BD.D1.8B.D0.B5_.D0.BE.D1.87.D0.B5.D1.80.D0.B5.D0.B4.D0.B8) Qиз вершин G \setminus F, в которой ключом для вершины vявляется \min\limits_{u \in V(F), uv \in E(G)}w(uv)— вес минимального ребра из вершин Fв вершины G \setminus F. Также для каждой вершины в очереди будем хранить p(v)— вершину u, на которой достигается минимум в определении ключа. Дерево Fподдерживается неявно, и его ребра — это пары \left(v,p(v)\right), где v \in G \setminus \{r\} \setminus Q, а r— корень F. Изначально Fпусто и значения ключей у всех вершин равны +\infty. Выберем произвольную вершину rи присвоим её ключу значение 0. На каждом шаге будем извлекать минимальную вершину vиз приоритетной очереди и релаксировать все ребра vu, такие что u \in Q, выполняя при этом операцию \text{decreaseKey}над очередью и обновление p(v). Ребро \left(v,p(v)\right)при этом добавляется к ответу. Дерево восстанавливается по окончательным записанным результатам u,p(v) и \left(v,p(v)\right).

Оценка времени работы алгоритма зависит от способа представления приоритетной очереди и графа. (таблица 1).

*Таблица 1 - Оценка времени алгоритма.*



### Псевдокод:

// G — исходный граф

// w — весовая функция

function \mathtt{primFindMST}():

for v \in V(G)

\mathtt{key}[v]\ =\ \infty

\mathtt{p}[v]\ = null

r\ = произвольная вершина графа G

\mathtt{key}[r]\ =\ \mathtt{0}

Q.\mathtt{push}(V(G))

while not Q.\mathtt{isEmpty()}

v\ =\ Q.\mathtt{extractMin}()

for vu \in E(G)

if u \in Q and \mathtt{key}[u] > w(v, u)

\mathtt{p}[u]\ =\ v

\mathtt{key}[u]\ =\ w(v, u)

Q.\mathtt{decreaseKey}(u, \mathtt{key}[u])

### Пример работы алгоритма.

Пример работы алгоритма представлен в таблице 2.

*Таблица 2. Пример работы алгоритма Прима*

|  |  |  |
| --- | --- | --- |
| Изображение | Множество вершин | Описание |
| Mst prima 1.png | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a | b | c | d | e | | 0 | \infty | \infty | \infty | \infty | | Извлечём из множества вершину a, так как её приоритет минимален. Рассмотрим смежные с ней вершины b, c, и e.  Обновим их приоритеты, как веса соответствующих рёбер ab, ac и ae, которые будут добавлены в ответ. |
| Mst prima 2.png | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a | b | c | d | e | | 0 | 3 | 4 | \infty | 1 | | Теперь минимальный приоритет у вершины е. Извлечём её и рассмотрим смежные с ней вершины a, c, и d. Изменим приоритет только у вершины d, так как приоритеты вершин a и с меньше, чем веса у соответствующих  рёбер ea и ec, и установим приоритет вершины d равный весу ребра ed, которое будет добавлено в ответ. |
| Mst prima 3.png | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a | b | c | d | e | | 0 | 3 | 4 | 7 | 1 | | После извлечения вершины b ничего не изменится, так как приоритеты вершин a и с меньше, чем веса у соответствующих рёбер ba и bc. Однако, после извлечения следующей вершины — c, будет обновлён приоритет у вершины d на более низкий (равный весу ребра cd) и в ответе ребро ed будет заменено на cd. |
| Mst prima 4.png | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a | b | c | d | e | | 0 | 3 | 4 | 2 | 1 | | Далее будет рассмотрена следующая вершина — d, но ничего не изменится, так как приоритеты вершин e и с меньше, чем веса у соответствующих рёбер de и dc. После этого алгоритм завершит работу, так как в заданном множестве не останется вершин, которые не были бы рассмотрены |

### Спецификация.

Программа предназначена для исследования алгоритма ЯПД на случайно сгенерированных графах инаписана на языке С++ с использованием компилятора МVC++2017.

Входными данными является количество генерируемых вершин и матрица смежности введенной с клавиатуры, на выходе которых выводятся вершины и их родители, а также их соединяющие веса ребер, или количество тестов, вершин и количество обходов для каждого теста , после которых на выходе получим время выполнения работы алгоритма для каждого из этих тестов.

**Реализация.**

**Описание функций и структур данных.**

Вершина графа представлена следующей структурой:

*template <class T>*

*struct* Vertex{

size\_t top\_;*// имя вершины*

T priority\_; *//вес-приоритет*

*int* parent\_;*//родитель*

*};*

Граф представлен следующим классом:

*template* <*class* T>

*class* Graph {

vector<vector<T>> graph\_; //матрица смежности - представление графа

size\_t V; //название вершины (порядковый номер)

};

void prim () - метод, выполняющий алгоритм прима для данного графа. Описание реализации находится в описание алгоритма Прима (см. выше).

В качестве контейнера реализации очереди с приоритетом была использована двоичная куча, которая представляется в виде бинарного дерева, корень которой является минимальным элементом. Скорость выполнения метода равна O(\log n).

А скорость выполнения алгоритма для V вершин и E ребер равна O(E\log{V}).

Двоичная куча представлена следующим классом:

*template* <*typename* T>

*class* BinTree{

*private:*

size\_t heap\_size; - размер кучи

std::vector<Vertex<T>> heap; - куча из вершин с приоритетом типа Т

};

Функции:

size\_t getLeft(size\_t cur) const- возвращение адреса левого сына

size\_t getRight(size\_t cur) const- возвращение адреса правого сына

void siftDown(int cur) – восстановление свойств кучи , если приоритет элемента стал больше

void siftUp(int cur)- восстановление свойств кучи , если приоритет элемента стал меньше.

void insert(SV<T> key) – вставка элемента

size\_t extractMin();- извлечение минимального элемента (фактически, корень дерева)

void reduceKey(size\_t top, T change) -уменьшение приоритета.

Генератор графов позволяет выбрать количество тестов, количество вершин в тести и количество проходов алгоритма по графу

### Тестирование.

Примеры выполнения работы алгоритма программой отображены в таблице 3.

*Таблица 3 - Примеры работы программы.*

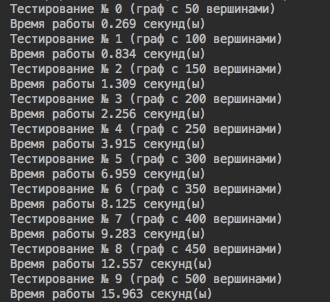
|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| Матрица смежности  0 3 4 0 1  3 0 5 0 0  4 5 0 2 6  0 0 2 0 7  1 0 6 7 0 | Результат    Рисунок  Mst prima 4.png |
| Матрица смежности  0 4 0 0 0 0 0 8 0  4 0 8 0 0 0  0 11 0  0 8 0 7 0 4 0 0 2  0 0 7 0 9 14 0 0 0  0 0 4 14 10 0 2 0 0  0 0 0 0 0 2 0 1 6  8 11 0 0 0 0 1 0 7  0 0 2 0 0 0 6 7 0 | Результат    Рисунок |

### Исследование.

Так как мы генерируем полный граф, то E=V\*(V-1)/2.

Запускаем тестирование для графов с 50, 100, 150, 200, 250, 300, 350, 400, 450 и 500 вершинами.

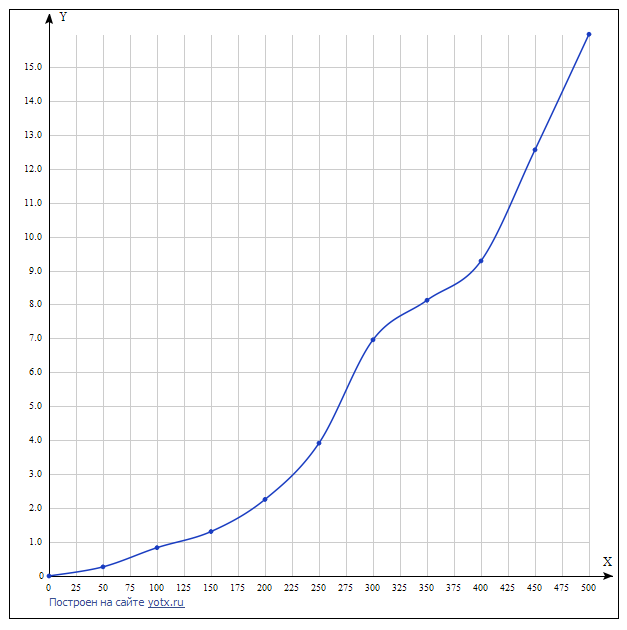
Результаты работы программы изображен на рисунке 2.



***Рис.2****.* *- Результат работы программы.*

По полученным данным можно построить график (ось OY - время, OX -

количество вершин) (график 1).

****

***График 1 -*** *Результирующий график.*

Пусть время работы алгоритма f. f=O(ElogV)=O(V\*(V-1)/2 \* logV).

Это означает, что f/(V\*(V-1)/2 \* log2V) = c (c - некоторая константа).

Построим таблицу зависимости c(f(V)). (Таблица 4)

*Таблица 4 - Зависимость время работы алгоритма Прима от количества вершин в графе*

|  |  |  |
| --- | --- | --- |
| **V - количество вершин** | **f(V) - время** | **c - константа** |
| 50 | 0,269 | 0,00003891 |
| 100 | 0,834 | 0,00002536 |
| 150 | 1,309 | 0,00001620 |
| 200 | 2,256 | 0,00001483 |
| 250 | 3,915 | 0,00001579 |
| 300 | 6,959 | 0,00001886 |
| 350 | 8,125 | 0,00001574 |
| 400 | 9,283 | 0,00001346 |
| 450 | 12,557 | 0,00001410 |
| 500 | 15,963 | 0,00001427 |

Из таблицы видно, что предположительная константа “c” с учетом некоторой погрешности действительно является константой.

Следовательно, практические и теоретические данные совпали и сложность алгоритма O(ElogV).

### Заключение.

В этой курсовой работе на практике был освоен один из алгоритмов поиска минимального остовного дерева- алгоритм ЯПД. Практические и теоретические данные работы алгоритма совпали и сложность алгоритма, если в качестве очереди с приоритетом использовать двоичную кучу, O(ElogV) .

### Список литературы.

1) Томас Х. *Кормен*, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ, 3-е издание = Introduction to Algorithms,

2) https://neerc.ifmo.ru/wiki/.

### ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД

MAIN.CPP

#include <iostream>

#include "graph.h"

#include <ctime>

using namespace std;

void input\_by\_hands();

void test();

int main(int argc, char \*argv[])

{

int c;

bool finish = false;

while(!finish){

std::cout <<"Введите пункт меню" << std::endl <<

"1. Ввести граф вручную" << std::endl <<

"2. Запустить тестирование" << std::endl <<

"3. Выход" << std::endl;

cin >> c;

switch (c) {

case 1:

input\_by\_hands();

break;

case 2:

test();

break;

case 3:

finish = true;

break;

default:

std::cout << "Неверный ввод" << std::endl;

cin.ignore();

cin.clear();

cin.get();

break;

}

}

return 0;

}

void input\_by\_hands() {

cout << "Введите количество вершин" << endl;

int n;

cin >> n;

vector<vector<int>> distance(n);

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

distance[j].resize(n);

}

int count = 0;

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

for (int q = 0; (q + count)<n; q++) {

if (i != (q + count)) {

cout << "Введите расстояние между " << i + 1 << " и " << count + q + 1 << " вершинами" << endl;

cin >> distance[i][count + q];

distance[count + q][i] = distance[i][count + q];

cout << "В матрице на позиции (" << i << ", " << count + q << ") и симметрично ей записывается: "<< distance[i][count + q] << endl << endl;

}

else

distance[i][count + q] = 0;

}

++count;

}

cout << "Матрица смежности" << endl;

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

for (int q = 0; q<n; ++q) {

cout << distance[i][q] << " ";

}

cout << endl;

}

cout << endl;

Graph<int> gr(distance);

cout << "Алгоритм Прима:\n";

gr.prim();

}

void test() {

std::cout << "Введите количество тестов" << std::endl;

int tn;

std::cin >> tn;

//int a[tn];

std::vector<int> a(tn);

std::cout << "Введите количество запусков алгоритма на каждом графе (чем больше, чем точнее значение времени работы)" << std::endl;

int rn;

std::cin >> rn;

time\_t alltime = 0;

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

std::cout << "Введите количество вершин для графа теста № " << i << std::endl;

std::cin >> a[i];

}

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

std::cout << "Сгенерирован тестовый файл № " << i << " с " << a[i] << " вершинами" << std::endl;

one\_test\_generator(a[i]);

}

std::ifstream fin;

std::string name = "";

for (int z = 0; z<tn; z++) {

name = "";

name += "test";

name.append(std::to\_string(z));

name += ".txt";

fin.open(name);

std::cout << "Тестирование № " << z << " (граф с " << a[z] << " вершинами)" << std::endl;

int n = a[z];

vector <vector<double>> distance(n);

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

distance[j].resize(n);

}

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

for (int q = 0; q<n; q++) {

fin >> distance[i][q];

}

}

Graph<double> gr(distance);

clock\_t t;

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

t = clock();

gr.silent\_prim();

t = clock() - t;

alltime = alltime + t;

//std::cout << "Время работы " << t / 1000.0 << " секунд(ы)" << std::endl;

}

std::cout << "Время работы " << alltime / rn / 1000.0 << " секунд(ы)" << std::endl;

alltime = 0;

fin.close();

}

}

BINTREE.cpp

#pragma once

#include <vector>

#include <iostream>

using namespace std;

//----------------------------------------------------------------------------------------------------

template <class T>

struct Edge

{

size\_t to\_;// имя вершины

T weight\_; //вес-приоритет

int from\_;//родитель

Edge() :to\_(0), weight\_(0){}

Edge(int to, T priority) :to\_(to), weight\_(priority), from\_(-1){}

template <class U>

Edge& operator=(const Edge<U> &other){

if (this != &other)

{

to\_ = other.to\_;

weight\_ = other.weight\_;

from\_ = other.from\_;

}

return \*this;

}

void add(int to, T priority, int parent){

to\_ = to;

weight\_ = priority;

from\_ = parent;

}

void show(){

std::cout << weight\_ << std::endl;

}

bool operator==(const Edge<T>& v2){

return weight\_ == v2.weight\_;

}

bool operator<(const Edge<T> &v2){

return weight\_ <v2.weight\_;

}

bool operator>(const Edge<T> &v2){

return v2.weight\_ <weight\_;

}

};

//----------------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------------

template <typename T> // https://habr.com/post/112222/

class BinTree

{

private:

size\_t heap\_size;

std::vector<Edge<T>> heap;

public:

BinTree(std::vector<Edge<T>> array);

size\_t get\_left\_vertex(size\_t cur) const;

size\_t get\_right\_vertex(size\_t cur) const;

size\_t extract\_vertex\_min\_path();

void sift\_down(int cur);

void sift\_up(int cur);

void insert(Edge<T> key);

void reduce\_key(size\_t to,T change);

void swap\_vertex(Edge<T> &v1, Edge<T> &v2);

bool is\_empty() const;

void show() const;

bool real(size\_t to); //проверка на существование вершины

size\_t get\_heap\_size(){ return heap\_size;}

std::vector<Edge<T>> get\_heap(){ return heap;}

};

template<typename T>

BinTree<T>::BinTree(std::vector<Edge<T>> array){

heap\_size = 0;

for (size\_t i = 0; i < array.size(); i++){ // fill with edges (0 ... len of amount edges)

insert(array[i]);

}

}

template<typename T>

void BinTree<T>::swap\_vertex(Edge<T> &v1, Edge<T> &v2){

Edge<T> buffer = v1;

v1 = v2;

v2 = buffer;

}

template<typename T>

bool BinTree<T>::is\_empty() const{

return !heap\_size;

}

template<typename T>

void BinTree<T>::show() const{

for (size\_t i = 0; i < heap\_size; i++)

std::cout << heap[i].from\_ << " " << heap[i].weight\_ << " " << heap[i].to\_ << endl;

}

template<typename T>

bool BinTree<T>::real(size\_t to){ //проверка на существование вершины

for (size\_t count = 0; count<heap\_size; ++count) {

if (to == heap[count].to\_)

return true;

}

// cout << " Закончено рассмотрение вершины " << to << ". Все ребра (входящие и исходящие) для данной вершины построены. " << endl;

return false;

}

template<typename T>

void BinTree<T>::sift\_down(int cur){ //восстановление свойств кучи , если приоритет элемента стал больше

while (2 \* cur + 1 < heap\_size) { // heap\_size — количество элементов в куче

int left = 2 \* cur + 1; // left — левый сын

int right = 2 \* cur + 2; // right — правый сын

int j = left;

if ((right < heap\_size) && (heap[right].weight\_ < heap[left].weight\_))

j = right;

if (heap[cur].weight\_ <= heap[j].weight\_)

break;

swap\_vertex(heap[cur], heap[j]);

cur = j;

}

}

template<typename T>

void BinTree<T>::sift\_up(int cur){

while ((heap[cur].weight\_ < heap[(cur - 1) / 2].weight\_)) { // cur == 0 — мы в корне

swap\_vertex(heap[cur], heap[(cur - 1) / 2]);

cur = (cur - 1) / 2;

}

}

template<typename T>

void BinTree<T>::insert(Edge<T> key){

heap\_size++;

heap.push\_back(key);

sift\_up(heap\_size - 1);

}

template<typename T> // оставить минимальное количество ребер для поддержания пути из вершины to

void BinTree<T>::reduce\_key(size\_t to, T change){ // находим вершину, в которую придем из вершины to

for (int i = 0; i < heap\_size; i++) {

if (to == heap[i].to\_) {

cout << " Меняем вес между вершинами " << heap[i].from\_ << " и " << heap[i].to\_

<< " c " << heap[i].weight\_ << " на "<< change <<endl;

heap[i].weight\_ = change;

sift\_up(i);

return;

}

}

return;

}

template<typename T>

size\_t BinTree<T>::get\_left\_vertex(size\_t cur) const{

return 2 \* cur + 1;

}

template<typename T>

size\_t BinTree<T>::get\_right\_vertex(size\_t cur) const{

2 \* cur + 2;

}

template<typename T>

size\_t BinTree<T>::extract\_vertex\_min\_path(){

cout << "Нерассмотренных до конца вершин " << heap\_size << ". Переход по самому дешевому пути из вершины : "<< heap[0].from\_ << endl;

cout << " Рассматриваемая вершина " << heap[0].to\_ << endl;

size\_t min = heap[0].to\_;

heap[0] = heap[heap\_size - 1];

heap\_size = heap\_size - 1;

sift\_down(0);

return min;

}

GRAPH.H

#ifndef Graph\_H

#define Graph\_H

#include <algorithm>

#include <vector>

#include <iostream>

#include <queue>

#include <random>

#include <fstream>

#include <functional>

#include <limits>

#include <string>

#include "bintree.h"

//----------------------------------------------------------------------------------------------------

///Реализация графа///

template <class T>

class Graph {

private:

std::vector<std::vector<T>> graph\_; // статья про векторы http://www.prog.org.ru/topic\_27860\_0.html

size\_t edge\_amount;

public:

void prim(); //алгоритм Прима (Я-Прима-Дейкстры)

void silent\_prim();

Graph() :graph\_(nullptr), edge\_amount(0){}

Graph(std::vector<std::vector<T>> buffer) :graph\_(buffer), edge\_amount(buffer.size()){}

};

//----------------------------------------------------------------------------------------------------

template <class T>

void Graph<T>::prim()

{

std::vector<Edge<T>> var(edge\_amount); // vector with vertexes

for (size\_t i = 0; i<edge\_amount; i++) { // граф, где из стартовой вершины проведены ребра во все остальные с макс. весом

if (i == 0)

var.at(i).add(i, 0, 0); // mark the first Edge,

else

var.at(i).add(i, std::numeric\_limits<T>::max(), 0); // max value of variable with type T

}

BinTree <T> Q(var); // очередь с приоритетами

while (!Q.is\_empty()) {

size\_t vertex\_min\_path = Q.extract\_vertex\_min\_path();

for (size\_t count = 0; count<edge\_amount; count++) {

if ((graph\_[vertex\_min\_path][count] != 0) && (var[count].weight\_ > graph\_[vertex\_min\_path][count]) && (Q.real(count)))

{

cout << " Т.к. из вершины " << var.at(count).from\_ << " есть путь в вершину " << count

<< " с весом " << graph\_[vertex\_min\_path][count]

/\* << graph\_[vertex\_min\_path][count] << " != 0 и "

<< var[count].weight\_ << " > " <<graph\_[vertex\_min\_path][count] << ", и "\*/

<< ", и вершина " << count << " не рассматривалась" << endl;

cout << " Строим ребро между вершинами " << var.at(count).from\_ << " и " << var.at(count).to\_ << endl;

var.at(count).weight\_ = graph\_[vertex\_min\_path][count];

var.at(count).from\_ = vertex\_min\_path;

Q.reduce\_key(var.at(count).to\_, var.at(count).weight\_);

}

}

}

cout << "\nМинимальное остовное дерево построено: " << endl;

for (size\_t count = 0; count<edge\_amount; count++)

std::cout << " Ребро(" << var[count].from\_ << ", " << var[count].to\_ << ") с весом " << var[count].weight\_ << std::endl;

cout << endl;

}

template <class T>

void Graph<T>::silent\_prim()

{

std::vector<Edge<T>> var(edge\_amount);

for (size\_t i = 0; i < edge\_amount; i++) {

if (i == 0)

var.at(i).add(i, 0, 0);

else

var.at(i).add(i, std::numeric\_limits<T>::max(), 0);

}

BinTree <T> Q(var);

while (!Q.is\_empty()) {

size\_t vertex\_min\_path= Q.extract\_vertex\_min\_path();

for (size\_t count = 0; count < edge\_amount; count++) {

if ((graph\_[vertex\_min\_path][count] != 0) && (var[count].weight\_>graph\_[vertex\_min\_path][count]) && (Q.real(count)))

{

var.at(count).weight\_ = graph\_[vertex\_min\_path][count];

var.at(count).from\_ = vertex\_min\_path;

Q.reduce\_key(var.at(count).to\_, var.at(count).weight\_);

}

}

}

}

void graph\_generator(size\_t Vnumber, std::vector<std::vector<double>> &matrix); //принимает размер графа и матрицу

void test\_generator(int t, int Vnumber); //принимает количество тестовых файлов и размер графа

void graph\_generator(size\_t Vnumber, std::vector<std::vector<double>> &matrix) {

std::random\_device generator; // генератор настоящих случайных чисел

std::uniform\_real\_distribution<double> distr(0, 10); // равномерное распределение в диапазоне от 1 до 10

auto real\_random = std::bind(distr, std::ref(generator)); // обертка для удобного вызова функции

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

matrix[j].resize(Vnumber);

}

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

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

matrix[i][j] = matrix[j][i] = real\_random();

}

}

}

int test\_num = 0;

void test\_generator(int t, int Vnumber) {

for (int i = test\_num; i<t + test\_num; i++) {

std::ofstream fout;

std::string name = "";

name += "test";

name.append(std::to\_string(i));

name += ".txt";

fout.open(name);

std::vector<std::vector<double>> edge\_amount(Vnumber);

graph\_generator(Vnumber, edge\_amount);

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

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

fout << edge\_amount[i][j] << " ";

}

fout << '\n';

}

test\_num++;

}

}

void one\_test\_generator(int Vnumber) {

// Vnumber - количество вершин

std::ofstream fout;

std::string name = "";

name += "test";

name.append(std::to\_string(test\_num));

name += ".txt";

fout.open(name);

std::vector<std::vector<double>> edge\_amount(Vnumber);

graph\_generator(Vnumber, edge\_amount);

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

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

fout << edge\_amount[i][j] << " ";

}

fout << '\n';

}

test\_num++;

}

#endif // Graph\_H