Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Нижегородский государственный университет  
им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики (ИИТММ)

Отчёт по лабораторной работе

**Построение минимального остовного дерева**

Выполнил:  
студент ИИТММ гр. 381503-1  
 Красикова Е.А.

Проверил:

к.т.н., ассистент каф. ПРИН  
ИИТММ

Сиднев А.А.

Нижний Новгород  
2017г.

Содержание

[Введение 3](#_Toc483155675)

[Постановка учебно-практической задачи 4](#_Toc483155676)

[Руководство пользователя 5](#_Toc483155677)

[Руководство программиста 7](#_Toc483155678)

[Общее описание структуры программного комплекса 7](#_Toc483155679)

[Описание структур данных 7](#_Toc483155680)

[Описание алгоритмов 8](#_Toc483155681)

[Алгоритм Краскала 8](#_Toc483155682)

[Алгоритм Прима 9](#_Toc483155683)

[Алгоритмы реализации методов класс UnionFind 9](#_Toc483155684)

[Алгоритмы реализации методов класса FiveHeap 10](#_Toc483155685)

[Заключение 12](#_Toc483155686)

[Список литературы 13](#_Toc483155687)

[Приложение 1. SpanningTree.h 14](#_Toc483155688)

[Приложение 2. CreateGraph.cpp 14](#_Toc483155689)

[Приложение 3. PrintGraph.cpp 15](#_Toc483155690)

[Приложение 4. STKruskal.cpp 16](#_Toc483155691)

[Приложение 5. STPrim.cpp 16](#_Toc483155692)

[Приложение 6. UnionFind.h 17](#_Toc483155693)

[Приложение 7. UnionFind.cpp 17](#_Toc483155694)

[Приложение 8. FiveHeap.h 18](#_Toc483155695)

[Приложение 9. test\_UnionFind.cpp 19](#_Toc483155696)

[Приложение 10. test\_FiveHeap.cpp 20](#_Toc483155697)

[Приложение 11. test\_STKruskal.cpp 20](#_Toc483155698)

[Приложение 12. test\_STPrim.cpp 22](#_Toc483155699)

# Введение

Задача построения минимального остовного дерева возникает в повседневной жизни довольно часто. К примеру, при разработке сетей, когда требуется соединить n городов в единую телефонную сеть с минимальной суммарной стоимостью соединений; в производство печатных плат, когда требуется соединить n контактов проводами с минимальной суммарной стоимостью. Минимальное остовное дерево может использоваться для визуализации многоаспектных, многомерных данных, например, для отображения их взаимосвязи. Наука, и в частности биология, используют многомерные данные для группировки объектов, растений, животных. Минимальное остовное дерево позволяет разбивать их на взаимосвязанные классы, четко отслеживая близкие по строению и характеристикам группы. В данной лабораторной работе будут рассмотрены алгоритмы Краскала и Прима построения минимального остовного дерева.[1]

# Постановка учебно-практической задачи

Необходимо разработать программу, реализующую алгоритмы Краскала и Прима для построения минимального остовного дерева. Программа должна позволять задавать граф и визуализировать его. При реализации алгоритма Краскала использовать представление графа в виде матрицы инцидентности и представление разделенных множеств с помощью массива. При реализации алгоритма Прима использовать представление графа в виде списка смежности и 5-кучи. Реализовать тесты, доказывающие корректность работы алгоритма и используемых структур данных.

# Руководство пользователя

Чтобы построить минимальное остовное дерево при помощи алгоритма Краскала, необходимо:

1. Задать граф при помощи матрицы инцидентности в следующем формате (в отдельном файле):

[количество вершин]

[количество рёбер]

[матрица инцидентности]

[строка весов ребер]

1. Создать пустую матрицу и массив, необходимые для представления взвешенного графа при помощи матрицы инцидентности:

vector<vector<int>> gr; // матрица инцидентности

vector<int> w; // массив весов рёбер

1. Считать граф из файла при помощи функции:

void CreateGraphMatrixOfIncidence(char filename[], vector<vector<int>> \*gr, vector<int> \*w);

1. Создать матрицу и массив, необходимые для представления минимального остовного дерева (количество строк матрицы равно количеству вершин графа, количество столбцов на единицу меньше количества вершин; количество вершин можно узнать при помощи метода gr.size())
2. Воспользоваться функцией

void STKruskal(vector<vector<int>> gr, vector<int> w, vector<vector<int>> \*ST,

vector<int> \*wST);

Чтобы построить минимальное остовное дерево при помощи алгоритма Краскала, необходимо:

1. Задать граф при помощи списка смежности в следующем формате (в отдельном файле):

[количество вершин]

[количество вершин, смежных с вершиной] [номер смежной вершины] [вес ребра]

1. Считать граф из файла при помощи функции:

vector < vector < pair<int, int> > > CreateGraphAdjacencyList(char filename[]);

1. Воспользоваться функцией

vector<vector<pair<int, int>>> STPrim(vector<vector<pair<int, int>>> gr);

Чтобы визуализировать граф, необходимо:

1. Если граф задан матрицей инцидентности, воспользоваться функцией:

void PrintGraphMatrixOfIncidence(const vector<vector<int>> &gr, const vector<int> &w, char filename[]),

либо функцией:

void PrintGraphAdjacencyList(vector < vector < pair<int, int> > > gr, char filename[]),

если граф задан списком смежности.

Эти функции создают файл с описанием графа на языке DOT.

1. Установить graphviz-2.38.msi
2. Открыть файл с описанием графа на языке DOT в этой программе

# Руководство программиста

## Общее описание структуры программного комплекса

Разработанная программа включает:

1. заголовочный файл «SpanningTree.h» (содержит прототипы функций CreateGraphAdjacencyList, CreateGraphMatrixOfIncidence, STKruskal, STPrim, PrintGraphAdjacencyList, PrintGraphMatrixOfIncidence)
2. файл исходного кода «CreateGraph.cpp» (содержит реализации функций CreateGraphAdjacencyList и CreateGraphMatrixOfIncidence)
3. файл исходного кода «PrintGraph.cpp» (содержит реализации функций PrintGraphAdjacencyList и PrintGraphMatrixOfIncidence)
4. файл исходного кода «STKruskal.cpp» (содержит реализацию функции STKruskal)
5. файл исходного кода «STPrim.cpp» (содержит реализацию функции STPrim)
6. заголовочный файл «UnionFind.h» (содержит интерфейс класса UnionFind)
7. файл исходного кода «UnionFind.cpp» (содержит реализацию методов класса UnionFind)
8. заголовочный файл «FiveHeap.h» (содержит интерфейс шаблонного класса FiveHeap и реализацию его методов)
9. файл исходного кода «test\_UnionFind.cpp» (содержит тесты, проверяющие корректность работы методов класса UnionFind)
10. файл исходного кода «test\_FiveHeap.cpp» (содержит тесты, проверяющие корректность работы методов класса FiveHeap)
11. файл исходного кода «test\_STKruskal.cpp» (содержит тесты, проверяющие корректность работы функции STKruskal)
12. файл исходного кода «test\_STPrim.cpp» (содержит тесты, проверяющие корректность работы функции STPrim)

## Описание структур данных

class UnionFind { // класс для представления разделенных множеств при  
 помощи массива  
 vector<int> a; // массив, в i-й ячейке которого лежит номер  
 множества i-го элемента  
 public:  
 explicit UnionFind(int n); // конструктор, создающий коллекцию из  
 n одноэлементных множеств  
 void Union(int x, int y); // объединений двух множеств  
 int Find(int x); // поиск множества, к которому принадлежит  
 элемент x  
};

template <class T>

class FourHeap { //шаблонный класс для представления 5-кучи

vector<T> heap; // вектор элементов

int child(int i); // метод, определяющий индекс ребенка по индексу   
 родителя

int parent(int i); // метод, определяющий индекс родителя по   
 индексу ребенка

public:

T Min(); // метод, определяющий минимальный элемент

void DeleteMin(); // удаление минимального элемента

void Insert(T key); // вставка нового элемента

bool empty(); // проверка на пустоту

};

## Описание алгоритмов

Дан связный, неориентированный граф с весами на ребрах G(V, E), в котором V — множество вершин (контактов), а E — множество их возможных попарных соединений (ребер). Пусть для каждого ребра (u,v) однозначно определено некоторое вещественное число w(u,v) — его вес (длина или стоимость соединения). Задача состоит в нахождении такого связного ациклического подграфа T ⊂ G, содержащего все вершины, что суммарный вес его ребер будет минимален. [1]

### Алгоритм Краскала

Рассмотрим алгоритм построения минимального остовного дерева, который был предложен Джозефом Краскалом в 1957 г.

gr – матрица инцидентности исходного графа

w – массив весов исходного графа

ST – матрица инцидентности остовного дерева

wST – массив весов остовного дерева

h – 5-куча для сортировки рёбер, её элементами являются пары (вес ребра, номер ребра)

uf - объект класса UnionFind, элементами являются вершины

Начало алгоритма

1. Добавляем в h все ребра графа (метод Insert)
2. Выполняем, пока куча не пуста и количество ребер (размер wST) меньше количества вершин без единицы (gr.size()-1):
3. Выбираем из кучи ребро с наименьшим весом (при помощи метода Min) и удаляем его из кучи (метод DeleteMin)
4. Если вершины, соединенные этим ребром, принадлежат разным множествам:
5. Объединяем вершины (при помощи метода Union)
6. Добавляем ребро в ST и его вес в wST
7. Проверяем все вершины: если среди вершин с номерами от 1 до n нашлась вершина, не лежащая в одном множестве с 0-вершиной, значит исходный граф был несвязный и нужно вывести сообщение об ошибке

Конец алгоритма

### Алгоритм Прима

Рассмотрим алгоритм, описанный в статье Роберта Прима, опубликованной в 1956 г.

gr – список смежности исходного графа

ST – список смежности остовного дерева

h – 5-куча для сортировки рёбер, элементами которой являются пары (вес ребра, пара смежных вершин)

f - массив вершин, добавленных в остовное дерево с элементами типа bool

mST – количество ребер в остовном дереве

Начало алгоритма

1. Добавляем в кучу все ребра, инцидентные первой вершине (метод Insert)
2. Отмечаем первую вершину как добавленную
3. Выполняем, пока куча не пуста:
4. Выбираем из кучи ребро с минимальным весом (при помощи метода Min) и удаляем его из кучи (метод DeleteMin)
5. Если вершина, в которую ведет это ребро (из той вершины, которая была добавлена в остовное дерево на предыдущих шагах), не отмечена как добавленная:
6. Увеличиваем mST на единицу
7. Добавляем ребро и вершину, в которую оно ведет, в список смежности остовного дерева
8. Вставляем в кучу все ребра, выходящие из новой вершины
9. Отмечаем эту вершину как добавленную
10. Если mST меньше количества ребер исходного графа без единицы, то исходный граф несвязный и будет сгенерировано исключение

Конец алгоритма

### Алгоритмы реализации методов класс UnionFind

Разделенные множества – это абстрактный тип данных, предназначенный для представления коллекции, состоящей из некоторого числа k попарно непересекающихся подмножеств U1, U2, ..., Uk заданного множества U. Для простоты в качестве U будем рассматривать множество {0, 2, …, n-1}. В качестве имени подмножества будем использовать один из его элементов (главный элемент), выбираемый по определенному правилу. Поскольку в коллекции всегда будут находиться попарно непересекающиеся подмножества множества *U*, такое имя будет однозначно определять требуемое подмножество. Одним из очевидных способов представления коллекции является представление ее с помощью массива. При таком способе для каждого элемента i в соответствующей (i-й) ячейке массива помещаем имя (канонический элемент) того подмножества, которому принадлежит элемент i. [2]

#### Поиск

Необходимо определить, к какому множеству принадлежит i-й элемент. Для этого достаточно вернуть содержимое i-й ячейки массива. Время выполнения операции –O(n).

#### Объединение

Необходимо объединить два множества, к которым принадлежат элементы x и y. Время выполнения операции –O(n).

Начало алгоритма

1. Если x и y принадлежат разным множествам:
2. Присваиваем s номер множества, к которому принадлежит элемент x
3. В каждую ячейку, в которой лежал номер множества s, записываем номер множества, к которому принадлежит элемент y

Конец алгоритма

### Алгоритмы реализации методов класса FiveHeap

5-heap (пирамида, сортирующее дерево, binary heap) – это 5-дерево, удовлетворяющее следующим условиям:

1. Приоритет любой вершины не больше приоритета ее потомков
2. Дерево является полным 5-деревом – все уровни заполнены слева направо

(возможно за исключением последнего) [4]

Каждая вершина дерева соответствует элементу массива. Если вершина имеет индекс i, то её родитель имеет индекс (i-1)/5, а её дети 5\*i+1, 5\*i+2, 5\*i+3, 5\*i+4. [3]

#### Поиск минимального элемента

Сложность операции – О(1). Минимальный элемент в 5-куче – первый элемент в массиве.

#### Вставка нового элемента

Сложность операции – О(logn)

Начало алгоритма

1. Добавляем новый элемент в конец массива
2. tmp – индекс добавленного элемента
3. Пока tmp больше нуля и родитель элемента tmp больше элемента tmp, выполнять:
4. Поменять местами элементы массива с индексами родителя tmp и tmp
5. Присвоить tmp индекс родителя tmp

Конец алгоритма

#### Удаление минимального элемента

Сложность операции – О(logn)

Начало алгоритма

1. Записать на первое место в массиве последний элемент
2. Удалить из массива последний элемент
3. tmp = 0
4. Пока индекс потомка tmp меньше размера массива, выполнять:
5. Найти минимальный элемент среди tmp и его потомков
6. Если минимальный элемент имеет индекс tmp, выйти из цикла
7. В противном случае поменять местами tmp и минимального потомка tmp
8. Присвоить tmp индекс минимального потомка

Конец алгоритма

# Заключение

Реализованы алгоритмы построение минимального остовного дерева Краскала и Прима. Реализованы функции чтения графа из файла для представления его в виде матрицы инцидентности и списка смежности, а также соответствующие функции для визуализации графа. Реализованы классы 5-кучи и разделенных множеств. Реализованы необходимые тесты.

# Список литературы

1. Минимальное остовное дерево - [http://rain.ifmo.ru/cat/view.php/theory/graph-spanning-trees/mst-2005
2. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород: Изд-во ННГУ, 2005. 307 с.
3. Свойства кучи – [http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall2014-lec6.pdf]
4. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн "Алгоритмы. Построение и анализ" Вильямс, 2013 год, 1324 стр. Издание 3-е
5. Бьёрн Страуструп.  Язык программирования C++ = TheC++ ProgrammingLanguage / Пер. с англ. — 3-е изд. — СПб.; М.: Невский диалект — Бином, 1999.

# Приложение 1. SpanningTree.h

#ifndef INCLUDE\_SPANNINGTREE\_H\_

#define INCLUDE\_SPANNINGTREE\_H\_

#include <vector>

#include <utility>

#define pair std::pair

#define vector std::vector

vector < vector < pair<int, int> > > CreateGraphAdjacencyList(char filename[]);

void CreateGraphMatrixOfIncidence(char filename[], vector<vector<int>> \*gr,

vector<int> \*w);

void STKruskal(vector<vector<int>> gr, vector<int> w, vector<vector<int>> \*ST,

vector<int> \*wST);

vector<vector<pair<int, int>>> STPrim(vector<vector<pair<int, int>>> gr);

void PrintGraphAdjacencyList(vector < vector < pair<int, int> > > gr,

char filename[]);

void PrintGraphMatrixOfIncidence(const vector<vector<int>> &gr, const vector<int> &w,

char filename[]);

#endif // INCLUDE\_SPANNINGTREE\_H\_

# Приложение 2. CreateGraph.cpp

#include <vector>

#include <fstream>

#include <utility>

#define pair std::pair

#define vector std::vector

vector < vector < pair<int, int> > > CreateGraphAdjacencyList(char filename[]) {

std::ifstream file(filename);

int N, n;

file >> N;

vector < vector < pair<int, int> > > g(N);

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

file >> n;

g[i] = vector< pair<int, int>>(n);

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

file >> g[i][j].first >> g[i][j].second;

}

}

return g;

}

void CreateGraphMatrixOfIncidence(char filename[], vector<vector<int>> \*gr, vector<int> \*w) {

std::ifstream file(filename);

int n, m;

file >> n >> m;

(\*gr) = vector<vector<int>>(n, vector<int>(m));

(\*w) = vector<int>(m);

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

for (int j = 0; j < m; ++j)

file >> (\*gr)[i][j];

for (int j = 0; j < m; ++j)

file >> (\*w)[j];

}

# Приложение 3. PrintGraph.cpp

#include <vector>

#include <fstream>

#include <utility>

#define pair std::pair

#define vector std::vector

void PrintGraphAdjacencyList(vector < vector < pair<int, int> > > gr,

char filename[]) {

std::ofstream file;

file.open(filename);

vector<vector<bool>> m(gr.size(), vector<bool>(gr.size(), 0));

file << "digraph MyGraph {" << std::endl <<

"node [shape=\"circle\", style=\"filled\", fillcolor=\"blue\", " <<

"fontcolor=\"#FFFFFF\", margin=\"0.01\"];"

<< std::endl << "edge [dir=\"both\"];" << std::endl;

for (int i = 0; i < static\_cast<int>(gr.size()); ++i)

for (int j = 0; j < static\_cast<int>(gr[i].size()); ++j)

if (m[i][gr[i][j].first] == 0) {

m[i][gr[i][j].first] = 1;

m[gr[i][j].first][i] = 1;

file << i << "->" << gr[i][j].first << "[label=\""<<

gr[i][j].second << "\"]" << std::endl;

}

file << "}";

}

void PrintGraphMatrixOfIncidence(const vector<vector<int>> &gr,

const vector<int> &w, char filename[]) {

std::ofstream file;

file.open(filename);

file << "digraph MyGraph {" << std::endl << "node [shape=\"circle\", " <<

"style=\"filled\", fillcolor=\"blue\", fontcolor=\"#FFFFFF\", " <<

"margin=\"0.01\"];"

<< std::endl << "edge [dir=\"both\"];" << std::endl;

for (int j = 0; j < static\_cast<int>(w.size()); ++j) {

int x = -1;

for (int i = 0; i < static\_cast<int>(gr.size()); ++i) {

if ((x != -1) && (gr[i][j] == 1))

file << x << "->" << i << "[label=\""<< w[j] << "\"]" <<

std::endl;

if (gr[i][j] == 1) x = i;

}

}

file << "}";

}

# Приложение 4. STKruskal.cpp

#include <vector>

#include <utility>

#include "FiveHeap.h"

#include "UnionFind.h"

#define pair std::pair

#define vector std::vector

void STKruskal(vector<vector<int>> gr, vector<int> w, vector<vector<int>> \*ST,

vector<int> \*wST) {

FiveHeap<pair<int, int>> h;

UnionFind uf(gr.size());

for (int i = 0; i < static\_cast<int>(w.size()); ++i)

h.Insert(pair<int, int>(w[i], i));

while ( (!h.empty()) && ((\*wST).size() < gr.size() - 1) ) {

pair<int, int> tmp = h.Min();

h.DeleteMin();

int x = -1;

for (int i = 0; i < static\_cast<int>(gr.size()); ++i) {

if ((x != -1) && (gr[i][tmp.second] == 1)) {

if (uf.Find(x) != uf.Find(i)) {

uf.Union(x, i);

(\*ST)[x][(\*wST).size()] = 1;

(\*ST)[i][(\*wST).size()] = 1;

(\*wST).push\_back(tmp.first);

}

}

if (gr[i][tmp.second] == 1) x = i;

}

}

for (int i = 0; i < static\_cast<int>(gr.size()); i++)

if (uf.Find(i) != uf.Find(0))

throw std::logic\_error("graph is disconnected");

}

# Приложение 5. STPrim.cpp

#include <utility>

#include "FiveHeap.h"

#define pair std::pair

#define vector std::vector

vector<vector<pair<int, int>>> STPrim(vector<vector<pair<int, int>>> gr) {

vector<vector<pair<int, int>>> ST(gr.size());

FiveHeap<pair<int, pair<int, int>>> h;

vector<bool> f(gr.size(), 0);

int mST = 0;

for (int i = 0; i < static\_cast<int>(gr[0].size()); ++i)

h.Insert(pair<int, pair<int, int>>(gr[0][i].second,

pair<int, int>(0, gr[0][i].first)));

f[0] = 1;

while (!h.empty()) {

pair<int, pair<int, int>> tmp = h.Min();

h.DeleteMin();

if (f[tmp.second.second] == 0) {

mST++;

ST[tmp.second.first].push\_back(pair<int, int>(tmp.second.second,

tmp.first));

ST[tmp.second.second].push\_back(pair<int, int>(tmp.second.first,

tmp.first));

for (int i = 0; i < static\_cast<int>(gr[tmp.second.second].size());

++i)

h.Insert(pair<int, pair<int, int>>(

gr[tmp.second.second][i].second, pair<int, int>(

tmp.second.second, gr[tmp.second.second][i].first)));

f[tmp.second.second] = 1;

}

}

if (mST < static\_cast<int>(gr.size())-1)

throw std::logic\_error("graph is disconnected");

return ST;

}

# Приложение 6. UnionFind.h

#ifndef INCLUDE\_UNIONFIND\_H\_

#define INCLUDE\_UNIONFIND\_H\_

#include <vector>

#define vector std::vector

class UnionFind {

vector<int> a;

public:

explicit UnionFind(int n);

void Union(int x, int y);

int Find(int x);

};

#endif // INCLUDE\_UNIONFIND\_H\_

# Приложение 7. UnionFind.cpp

#include "UnionFind.h"

UnionFind::UnionFind(int n) {

a = vector<int>(n);

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

a[i] = i;

}

int UnionFind::Find(int x) {

return a[x];

}

void UnionFind::Union(int x, int y) {

if (Find(x) != Find(y)) {

int s = Find(x);

for (int i = 0; i < static\_cast<int>(a.size()); ++i)

if (a[i] == s) a[i] = a[y];

}

}

# Приложение 8. FiveHeap.h

#ifndef INCLUDE\_FIVEHEAP\_H\_

#define INCLUDE\_FIVEHEAP\_H\_

#include <vector>

#include <stdexcept>

#define vector std::vector

template <class T>

class FiveHeap {

vector<T> heap;

int child(int i);

int parent(int i);

public:

T Min();

void DeleteMin();

void Insert(T key);

bool empty();

};

template <class T>

T FiveHeap<T>::Min() {

return heap[0];

}

template <class T>

int FiveHeap<T>::child(int i) {

if ( (i >= 0) && (5\*i+1 < static\_cast<int>(heap.size()))) return 5\*i+1;

throw std::logic\_error("incorrect index");

}

template <class T>

int FiveHeap<T>::parent(int i) {

if ( (i > 0) && (i < static\_cast<int>(heap.size()))) return (i-1)/5;

throw std::logic\_error("incorrect index");

}

template <class T>

void FiveHeap<T>::Insert(T key) {

heap.push\_back(key);

int tmp = static\_cast<int>(heap.size())-1;

while ((tmp > 0) && (heap[parent(tmp)] > heap[tmp])) {

T temp = heap[parent(tmp)];

heap[parent(tmp)] = heap[tmp];

heap[tmp] = temp;

tmp = parent(tmp);

}

}

template <class T>

void FiveHeap<T>::DeleteMin() {

heap[0] = heap[static\_cast<int>(heap.size())-1];

heap.pop\_back();

int tmp = 0;

while (5\*tmp+1 < static\_cast<int>(heap.size())) {

int iofmin = tmp;

for (int j = 0; (j < 5) && (child(tmp)+j <

static\_cast<int>(heap.size())); j++)

if ( heap[child(tmp) + j] < heap[iofmin]) iofmin = child(tmp) + j;

if (tmp == iofmin) break;

T temp = heap[tmp];

heap[tmp] = heap[iofmin];

heap[iofmin] = temp;

tmp = iofmin;

}

}

template <class T>

bool FiveHeap<T>::empty() {

return heap.empty();

}

#endif // INCLUDE\_FIVEHEAP\_H\_

# Приложение 9. test\_UnionFind.cpp

#include <gtest/gtest.h>

#include "UnionFind.h"

TEST(UnionFind, can\_create\_collection) {

ASSERT\_NO\_THROW(UnionFind h(5));

}

TEST(UnionFind, can\_find\_set\_of\_elem) {

UnionFind h(5);

ASSERT\_NO\_THROW(h.Find(3));

EXPECT\_EQ(3, h.Find(3));

}

TEST(UnionFind, can\_union\_singleton\_sets) {

UnionFind h(5);

ASSERT\_NO\_THROW(h.Union(1, 2));

EXPECT\_EQ(h.Find(1), h.Find(2));

}

TEST(UnionFind, can\_union\_multi\_item\_set\_with\_singleton\_set) {

UnionFind h(5);

h.Union(1, 2);

ASSERT\_NO\_THROW(h.Union(2, 3));

EXPECT\_EQ(h.Find(2), h.Find(3));

}

TEST(UnionFind, can\_union\_singleton\_set\_with\_multi\_item\_set) {

UnionFind h(5);

h.Union(1, 2);

ASSERT\_NO\_THROW(h.Union(3, 1));

EXPECT\_EQ(h.Find(3), h.Find(1));

}

TEST(UnionFind, can\_union\_multi\_item\_sets) {

UnionFind h(5);

h.Union(1, 2);

h.Union(3, 4);

ASSERT\_NO\_THROW(h.Union(2, 3));

EXPECT\_EQ(h.Find(2), h.Find(4));

}

# Приложение 10. test\_FiveHeap.cpp

#include <gtest/gtest.h>

#include "FiveHeap.h"

TEST(FourHeap, can\_create\_heap) {

ASSERT\_NO\_THROW(FiveHeap<int> h);

}

TEST(FiveHeap, can\_insert\_in\_heap) {

FiveHeap<int> h;

ASSERT\_NO\_THROW(h.Insert(2));

ASSERT\_NO\_THROW(h.Insert(5));

ASSERT\_NO\_THROW(h.Insert(1));

}

TEST(FiveHeap, can\_find\_min) {

FiveHeap<int> h;

h.Insert(2);

h.Insert(5);

h.Insert(1);

EXPECT\_EQ(1, h.Min());

}

TEST(FiveHeap, can\_delete\_min) {

FiveHeap<int> h;

h.Insert(2);

h.Insert(5);

h.Insert(1);

ASSERT\_NO\_THROW(h.DeleteMin());

EXPECT\_EQ(2, h.Min());

}

TEST(FiveHeap, can\_check\_empty) {

FiveHeap<int> h;

EXPECT\_TRUE(h.empty());

}

# Приложение 11. test\_STKruskal.cpp

#include <gtest/gtest.h>

#include "SpanningTree.h"

TEST(STKruskal, work\_right\_for\_graph\_with\_one\_vertex) {

vector < vector<int> > gr(1);

vector<int> w;

vector < vector<int> > res(1);

vector<int> wres;

STKruskal(gr, w, &res, &wres);

EXPECT\_EQ(static\_cast<int>(res.size()), 1);

EXPECT\_EQ(static\_cast<int>(res[0].size()), 0);

EXPECT\_EQ(static\_cast<int>(wres.size()), 0);

}

TEST(STKruskal, work\_right\_for\_graph\_which\_equel\_his\_spanning\_tree) {

vector < vector<int> > gr(3, vector<int>(2, 1));

gr[0][1] = 0;

gr[2][0] = 0;

vector<int> w(2);

w[0] = 5;

w[1] = 7;

vector < vector<int> > res(3, vector<int>(2));

vector<int> wres;

STKruskal(gr, w, &res, &wres);

for (int i = 0; i < static\_cast<int>(gr.size()); ++i)

for (int j = 0; j < static\_cast<int>(gr[i].size()); ++j)

EXPECT\_EQ(gr[i][j], res[i][j]);

for (int i = 0; i < static\_cast<int>(w.size()); ++i)

EXPECT\_EQ(w[i], wres[i]);

}

TEST(STKruskal, throw\_when\_graph\_is\_disconnected) {

vector < vector<int> > gr(4, vector<int>(2, 1));

gr[0][1] = 0;

gr[2][0] = 0;

gr[3][0] = gr[3][1] = 0;

vector<int> w(2);

w[0] = 5;

w[1] = 7;

vector < vector<int> > res(4, vector<int>(3));

vector<int> wres;

ASSERT\_ANY\_THROW(STKruskal(gr, w, &res, &wres));

}

TEST(STKruskal, work\_right\_for\_complete\_graph\_with\_three\_vertexes) {

vector < vector<int> > gr(3, vector<int>(3, 1));

gr[0][1] = 0;

gr[1][2] = 0;

gr[2][0] = 0;

vector<int> w(3);

w[0] = 5;

w[1] = 7;

w[2] = 11;

vector < vector<int> > exp(3, vector<int>(2, 1));

exp[0][1] = 0;

exp[2][0] = 0;

vector<int> wexp(2);

wexp[0] = 5;

wexp[1] = 7;

vector < vector<int> > res(3, vector<int>(2));

vector<int> wres;

STKruskal(gr, w, &res, &wres);

for (int i = 0; i < static\_cast<int>(exp.size()); ++i)

for (int j = 0; j < static\_cast<int>(exp[i].size()); ++j)

EXPECT\_EQ(exp[i][j], res[i][j]);

for (int i = 0; i < static\_cast<int>(wexp.size()); ++i)

EXPECT\_EQ(wexp[i], wres[i]);

}

TEST(STKruskal, work\_right\_for\_connected\_graph\_with\_five\_vertexes) {

char filename1[] = "GraphMatrixOfIncidence";

vector<vector<int>> gr;

vector<int> w;

CreateGraphMatrixOfIncidence(filename1, &gr, &w);

char filename2[] = "connected\_graph\_with\_five\_vertexes\_m\_of\_i.txt";

PrintGraphMatrixOfIncidence(gr, w, filename2);

vector < vector<int> > res(5, vector<int>(4));

vector<int> wres;

STKruskal(gr, w, &res, &wres);

char filename3[] = "spanning\_tree\_with\_five\_vertexes\_Kruskal.txt";

PrintGraphMatrixOfIncidence(res, wres, filename3);

vector < vector<int> > exp(5, vector<int>(4, 0));

exp[0][2] = exp[1][0] = exp[2][3] = exp[3][1] =

exp[3][2] = exp[3][3] = exp[4][0] = exp[4][1] = 1;

vector<int> wexp(4, 1);

wexp[2] = 2;

wexp[3] = 5;

for (int i = 0; i < static\_cast<int>(exp.size()); ++i)

for (int j = 0; j < static\_cast<int>(exp[i].size()); ++j)

EXPECT\_EQ(exp[i][j], res[i][j]);

for (int i = 0; i < static\_cast<int>(wexp.size()); ++i)

EXPECT\_EQ(wexp[i], wres[i]);

}

# Приложение 12. test\_STPrim.cpp

#include <gtest/gtest.h>

#include "SpanningTree.h"

TEST(STPrim, work\_right\_for\_graph\_with\_one\_vertex) {

vector < vector < pair<int, int> > > gr(1);

vector < vector < pair<int, int> > > res = STPrim(gr);

EXPECT\_EQ(static\_cast<int>(res.size()), 1);

EXPECT\_EQ(static\_cast<int>(res[0].size()), 0);

}

TEST(STPrim, work\_right\_for\_graph\_which\_equel\_his\_spanning\_tree) {

vector < vector < pair<int, int> > > gr(3);

gr[0] = vector<pair<int, int>>(1, pair<int, int>(1, 1));

gr[1] = vector<pair<int, int>>(2);

gr[1][0] = pair<int, int>(0, 1);

gr[1][1] = pair<int, int>(2, 3);

gr[2] = vector<pair<int, int>>(1, pair<int, int>(1, 3));

vector < vector < pair<int, int> > > res = STPrim(gr);

for (int i = 0; i < static\_cast<int>(gr.size()); ++i)

for (int j = 0; j < static\_cast<int>(gr[i].size()); ++j)

EXPECT\_EQ(gr[i][j], res[i][j]);

}

TEST(STPrim, throw\_when\_graph\_is\_disconnected) {

vector < vector < pair<int, int> > > gr(5);

gr[0] = vector<pair<int, int>>(1, pair<int, int>(1, 1));

gr[1] = vector<pair<int, int>>(2);

gr[1][0] = pair<int, int>(0, 1);

gr[1][1] = pair<int, int>(2, 3);

gr[2] = vector<pair<int, int>>(1, pair<int, int>(1, 3));

gr[3] = vector<pair<int, int>>(1, pair<int, int>(4, 5));

gr[4] = vector<pair<int, int>>(1, pair<int, int>(3, 5));

ASSERT\_ANY\_THROW(STPrim(gr));

}

TEST(STPrim, work\_right\_for\_complete\_graph\_with\_three\_vertexes) {

vector < vector < pair<int, int> > > gr(3);

gr[0] = vector<pair<int, int>>(2);

gr[0][0] = pair<int, int>(1, 1);

gr[0][1] = pair<int, int>(2, 4);

gr[1] = vector<pair<int, int>>(2);

gr[1][0] = pair<int, int>(0, 1);

gr[1][1] = pair<int, int>(2, 3);

gr[2] = vector<pair<int, int>>(2);

gr[2][0] = pair<int, int>(1, 3);

gr[2][1] = pair<int, int>(0, 4);

vector < vector < pair<int, int> > > res = STPrim(gr);

vector < vector < pair<int, int> > > exp = gr;

exp[0].pop\_back();

exp[2].pop\_back();

for (int i = 0; i < static\_cast<int>(exp.size()); ++i)

for (int j = 0; j < static\_cast<int>(exp[i].size()); ++j)

EXPECT\_EQ(exp[i][j], res[i][j]);

}

TEST(STPrim, work\_right\_for\_connected\_graph\_with\_five\_vertexes) {

char filename1[] = "GraphAdjacencyList";

vector < vector < pair<int, int> > > gr = CreateGraphAdjacencyList(filename1);

char filename2[] = "connected\_graph\_with\_five\_vertexes\_a\_l.txt";

PrintGraphAdjacencyList(gr, filename2);

vector<vector<pair<int, int>>> res = STPrim(gr);

char filename3[] = "spanning\_tree\_with\_five\_vertexes\_Prim.txt";

PrintGraphAdjacencyList(res, filename3);

EXPECT\_EQ(res[0][0].first, 3);

EXPECT\_EQ(res[0][0].second, 2);

EXPECT\_EQ(res[1][0].first, 4);

EXPECT\_EQ(res[1][0].second, 1);

EXPECT\_EQ(res[2][0].first, 3);

EXPECT\_EQ(res[2][0].second, 5);

EXPECT\_EQ(static\_cast<int>(res[3].size()), 3);

bool f1 = 0, f2 = 0, f3 = 0;

for (int i = 0; i < 3; ++i) {

if (res[3][i] == pair<int, int>(0, 2)) f1 = 1;

if (res[3][i] == pair<int, int>(4, 1)) f2 = 1;

if (res[3][i] == pair<int, int>(2, 5)) f3 = 1;

}

EXPECT\_EQ(f1\*f2\*f3, 1);

EXPECT\_EQ(static\_cast<int>(res[4].size()), 2);

f1 = 0;

f2 = 0;

for (int i = 0; i < 2; ++i) {

if (res[4][i] == pair<int, int>(1, 1)) f1 = 1;

if (res[4][i] == pair<int, int>(3, 1)) f2 = 1;

}

EXPECT\_EQ(f1\*f2, 1);

}