Федеральное государственное автономное образовательное учреждение

высшего образования

“Нижегородский государственный университет

им. Н.И. Лобачевского”

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Поиск минимального остовного дерева

Выполнила:

студент группы 0823-1

Ермакова Е.А.

Проверил:

к.т.н., ассистент каф. пр. инж. ИИТММ

Сиднев А.А.

Нижний Новгород

2017 г.

Содержание

[Введение 3](#_Toc483127963)

[1 Постановка задачи 4](#_Toc483127964)

[2 Руководство пользователя 5](#_Toc483127965)

[3 Руководство программиста 9](#_Toc483127966)

[3.1 Описание структуры программы 9](#_Toc483127967)

[3.2 Описание модульной структуры программы 9](#_Toc483127968)

[3.3 Описание структур данных 13](#_Toc483127969)

[3.4 Описание алгоритмов 14](#_Toc483127970)

[Алгоритмы 4-куч. 14](#_Toc483127971)

[Алгоритмы системы разделенных множеств. 16](#_Toc483127972)

[Алгоритм Краскала 16](#_Toc483127973)

[Алгоритм Прима 16](#_Toc483127974)

[Алгоритмы чтения, печати и преобразования графов 17](#_Toc483127975)

[Заключение 20](#_Toc483127976)

[Литература 21](#_Toc483127977)

[Приложения 22](#_Toc483127978)

[Приложение1 (файл main.cpp) 22](#_Toc483127979)

[Приложение 2 (файл four\_heap.h) 22](#_Toc483127980)

[Приложение 3 (файл graph.h) 25](#_Toc483127981)

[Приложение 4 (файл priority\_queue.h) 28](#_Toc483127982)

[Приложение 5 (файл spanning\_tree.h) 29](#_Toc483127983)

[Приложение 6 (файл uf.h) 29](#_Toc483127984)

[Приложение 7 (файл kruskal.cpp) 30](#_Toc483127985)

[Приложение 8 (файл prim.cpp) 30](#_Toc483127986)

[Приложение 9 (файл test\_kruskal.cpp) 31](#_Toc483127987)

[Приложение 10 (файл test\_prim.cpp) 32](#_Toc483127988)

[Приложение 11 (файл main\_test.cpp) 33](#_Toc483127989)

# Введение

Модели графов, в которых с каждым ребром связаны веса или стоимости, используются во многих приложениях. В картах авиалиний, в которых ребрами отмечены авиарейсы, такие веса означают расстояния или стоимости билетов. В электронных схемах, где ребра представляют проводники, веса могут означать длину проводника, его стоимость или время прохода сигнала. В задачах календарного планирования веса могут представлять время или трудоемкость либо выполнения задачи, либо ожидания ее завершения.

В таких ситуациях естественно возникают вопросы, касающиеся минимизации затрат, например, поиск пути наименьшей стоимости, соединяющего все точки. Эта задача на языке графов называется задачей поиска минимального остовного дерева

# 1 Постановка задачи

Требуется реализовать поиск минимального остовного дерева в неориентированном взвешенном графе с помощью: а) алгоритма Краскала; б) алгоритма Прима.

Программа принимает на входе текстовый файл, содержащий описание графа. На выходе программа возвращает текстовый файл, содержащий описание самого графа и описание найденного остова на языке DOT с целью дальнейшей визуализации работы программы.

Для реализации алгоритма Краскала необходимо использовать представление разделенных множеств с помощью древовидной структуры и использованием рангов вершин. Граф для алгоритма Краскала представляется в виде списка смежности.

Для реализации алгоритма Прима необходимо использовать приоритетную очередь, реализованную на 4-куче. Граф для алгоритма Прима представляется в виде матрицы смежности.

Исходные данные:

s1 – имя файла с описанием графа с виде списка смежности для демонстрации работы алгоритма Краскала;

s2 – имя файла с описанием графа с виде матрицы смежности для демонстрации работы алгоритма Прима;

Требуемый рез-тат:

Kruskal.txt – файл, содержащий описание самого графа и описание найденного с помощью алгоритма Краскала остова на языке DOT.

Prim.txt – файл, содержащий описание самого графа и описание найденного с помощью алгоритма Прима остова на языке DOT.

# 2 Руководство пользователя

Для нахождения минимального остовного графа пользователю доступны две функции:

1. Kruskal – поиск минимального остова алгоритмом Краскала.
2. Prim – поиск минимального остова алгоритмом Прима.

Для получения текстового файла Kruskal\_spanning\_tree.txt необходимо ввести путь до файла, содержащего описание исходного графа в виде списка смежности.

KRUSKAL`S ALGORITHM

ENTER THE PATH TO THE FILE: C:\Users\qq\Kate\Uni\Git\Laba\_ostov\infrastructure\Kruskal.txt

Сам файл Kruskal.txt выглядит следующим образом:

0: 1 5 3 2 4 7

1: 2 8 4 1

2: 3 5 4 8

3: 4 4

4:

В результате в папке на компьютере будет создан файл Kruskal\_spanning\_tree.txt, содержащий в себе описание графа на языке DOT с выделенным остовом.

graph {

edge[color = "#2EB8E6"]

1--4[style = "bold", label="1"]

0--3[style = "bold", label="2"]

3--4[style = "bold", label="4"]

2--3[style = "bold", label="5"]

0--1[color = black, label="5"]

0 [shape = "rect", color = black]

1 [shape = "rect", color = black]

1--2[color = black, label="8"]

1 [shape = "rect", color = black]

2 [shape = "rect", color = black]

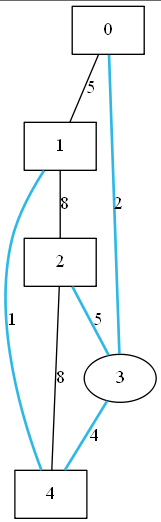
2--4[color = black, label="8"]

2 [shape = "rect", color = black]

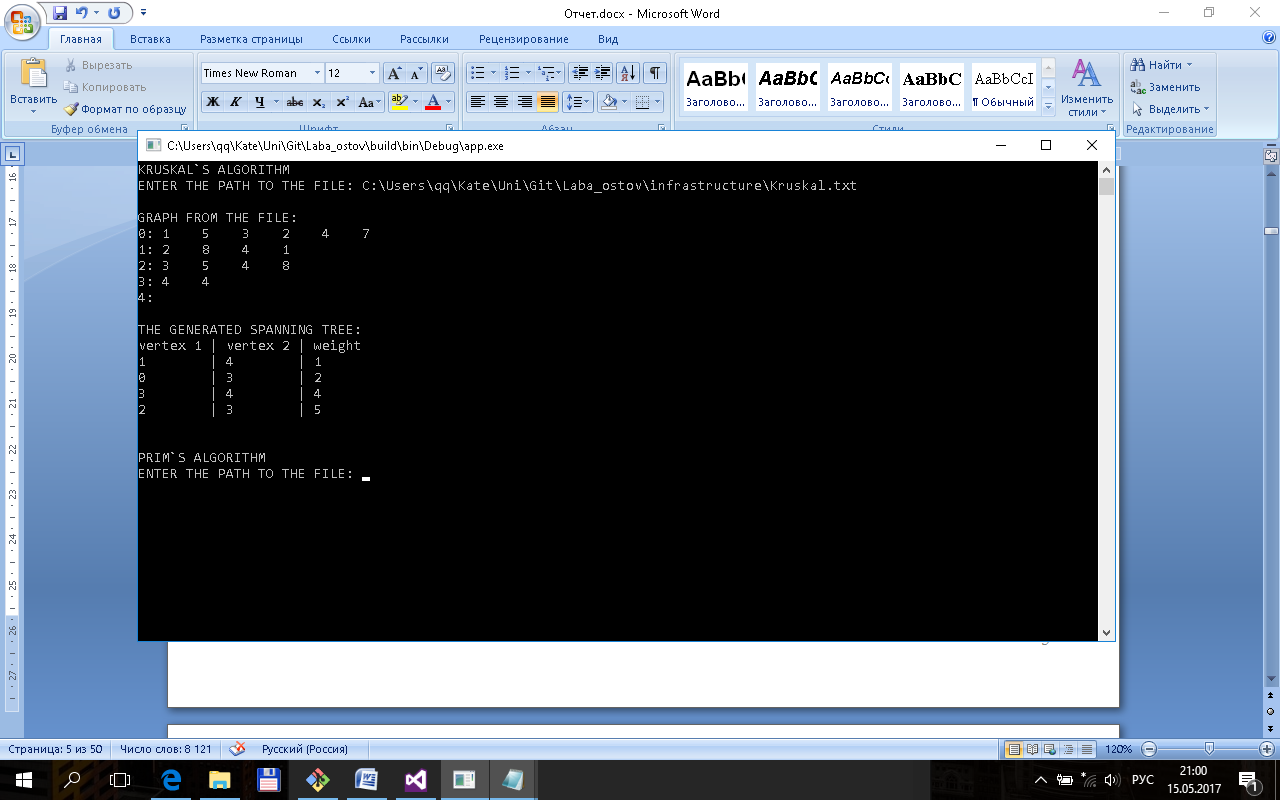
4 [shape = "rect", color = black]

}

Используя программу graphviz, можно получить следующее изображение графа (розовым цветом выделен остов)



Более того, результаты работы алгоритма будут выведены на консоль:



Для получения текстового файла Prim\_spanning\_tree.txt необходимо ввести путь до файла, содержащего описание исходного графа в виде матрицы смежности.

PRIM`S ALGORITHM

ENTER THE PATH TO THE FILE: C:\Users\qq\Kate\Uni\Git\Laba\_ostov\infrastructure\Prim.txt

Файл Prim.txt выглядит следующим образом:

0 5 - 2 7

5 0 8 - 1

- 8 0 5 8

2 - 5 0 4

7 1 8 4 0

В результате в папке на компьютере будет создан файл Prim\_spanning\_tree.txt, содержащий в себе описание графа на языке DOT с выделенным остовом.

graph {

edge[color = "#0DBD1C"]

1--4[style = "bold", label="1"]

3--4[style = "bold", label="4"]

0--3[style = "bold", label="2"]

2--3[style = "bold", label="5"]

0--1[color = black, label="5"]

0 [shape = "rect", color = black]

1 [shape = "rect", color = black]

0--4[color = black, label="7"]

0 [shape = "rect", color = black]

4 [shape = "rect", color = black]

1--2[color = black, label="8"]

1 [shape = "rect", color = black]

2 [shape = "rect", color = black]

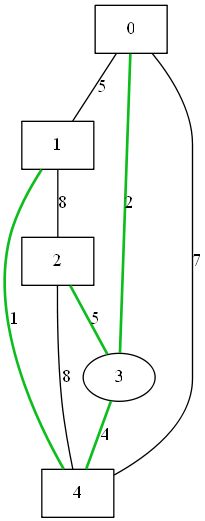
2--4[color = black, label="8"]

2 [shape = "rect", color = black]

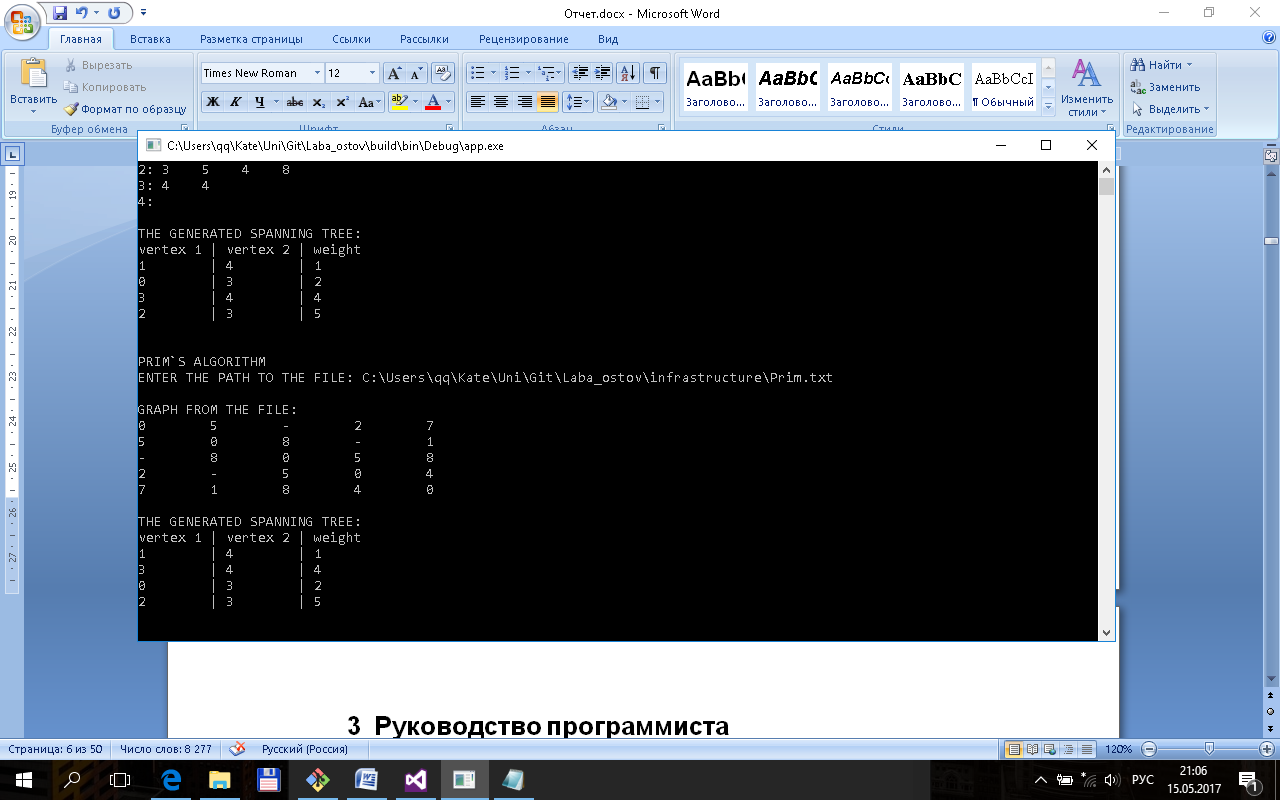
4 [shape = "rect", color = black]

}

Результат визуализации в программе graphviz



Результаты работы алгоритма будут также выведены на консоль:



# 3 Руководство программиста

## 3.1 Описание структуры программы

Разработано решение, состоящее из следующих проектов:

1. *code.lib –* статическая библиотека, содержащая реализации алгоритмов Краскала и Прима, реализации вспомогательных структур (система разделенных множеств, приоритетная очередь, 4-кучи) и реализации вспомогательных к основной программе функций чтения из файла и записи в файл. Содержит заголовочные файлы four\_heap.h, priority\_queue.h, graph.h, spanning\_tree.h, uf.h, файлы kruskal.cpp и prim.cpp.
2. *gtest.lib –* статическая библиотека. Содержит заголовочные файлы, а также файлы-реализации, необходимые для работы GoogleTests, с помощью которых происходит проверка корректности программы.
3. *app.exe –* консольное приложение. Содержит файл main.cpp.
4. *mytest.exe -* консольное приложение, тестирующее работоспособность программы. Содержит файлы main\_test.cpp, test\_kruskal.cpp, test\_prim.cpp.

## 3.2 Описание модульной структуры программы

* Файл main.cpp содержит функцию int main(), в которой осуществляется ввод пути до файлов, содержащих описание исходных графов, запуск алгоритмов, запись в конечный файл и вывод результатов на консоль.
* Файл four\_heap.h содержит:
  1. Директивы предпроцессора и объявление констант:

#ifndef INCLUDE\_FOUR\_HEAP\_H\_

#define INCLUDE\_FOUR\_HEAP\_H\_

#include<stdexcept>

constint d\_ = 4;

constint MAX\_SIZE = 1000;

using std::logic\_error;

#endif// INCLUDE\_FOUR\_HEAP\_H\_

* 1. Объявление шаблонного класса FOURheap:

template<classType>

class FOURheap{

int size;

int d;

public:

Type\* val;

FOURheap();

~FOURheap();

int GetSize() { return size; }

bool operator == (const FOURheap<Type>& B);

bool operator != (const FOURheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

* 1. Реализацию методов класса FOURheap:
     + Конструктор:

FOURheap<Type>::FOURheap();

Метод создает пустой массив для элементов кучи и устанавливает значение размера, равное 0.

* + - Деструктор:

FOURheap<Type>::~FOURheap();

Метод удаляет кучу.

* + - Перегрузка операции ==:

bool FOURheap<Type>::operator==(const FOURheap<Type>&B);

Функция получает ссылку на кучу и возвращает true, если элементы кучи и полученной кучи полностью совпадают. В противном случае метод возвращает false.

* + - Перегрузка операции !=:

bool FOURheap<Type>::operator!=(const FOURheap<Type>&B);

Функция получает ссылку на кучу и возвращает true, кучи не совпадают. В противном случае метод возвращает false.

* + - Перегрузка []:

Type& FOURheap<Type>::operator[](int i);

Функция получает ключ и возвращает значение элемента, лежащему по этому ключу.

* + - Уменьшение значения на константу:

void FOURheap<Type>::Reduction(int i, const Type& con);

Функция получает ключ и константу, находит элемент, лежащий по этому ключу, и уменьшает его значение на указанную константу, после чего производится «всплытие» элемента.

* + - Транспонирование:

void FOURheap<Type>::Swap(int i, int j);

Функция получает два ключа и меняет местами значения элементов, лежащих по этим ключам.

* + - Поиск минимального потомка:

int FOURheap<Type>::MinChild(int i);

Функция получает ключ и возвращает ключ минимального по значению потомка, принадлежащего элементу с данным ключом.

* + - Вставка:

void FOURheap<Type>::Insert(const Type& val\_);

Функция получает значение и добавляет элемент с этим значением в кучу.

* + - Удаление:

void FOURheap<Type>::Erase(int i);

Функция получает ключ и удаляет элемент, лежащий по этому ключу, из кучи.

* + - Удаление минимального элемента:

void FOURheap<Type>::EraseMin();

Функция находит минимальный элемент в куче и удаляет его.

* + - Погружение:

void FOURheap<Type>::Immertion(int i);

Функция получает ключ и «опускает» соответствующий элемент на нужное для восстановления структуры кучи место.

* + - Всплытие:

void FOURheap<Type>::Emertion(int i);

Функция получает ключ и «поднимает» соответствующий элемент на нужное для восстановления структуры кучи место.

* + - Окучивание:

void FOURheap<Type>::DoingHeaps();

Функция превращает массив вершин в 5-кучу.

* Файл graph.h содержит:
  1. Директивы предпроцессора

#ifndef INCLUDE\_GRAPH\_H\_

#define INCLUDE\_GRAPH\_H\_

#include "spanning\_tree.h"

#include <stdio.h>

#include <stdexcept>

using std::logic\_error;

#endif // INCLUDE\_GRAPH\_H\_

* 1. Реализацию метода записи графа из текстового файла name в список смежности

vector<vector<pair<int, int>>> Read\_To\_Adjacency\_List(char\* name)

* 1. Реализацию метода печати графа в виде списка смежности из текстового файла name на консоль

void Print\_Adjacency\_List(char\* name)

* 1. Реализацию метода записи графа из текстового файла name в матрицу смежности

vector<vector<int>> Read\_To\_Adjacency\_Matrix(char\* name)

* 1. Реализацию метода печати графа в виде матрицы смежности из текстового файла name на консоль

void Print\_Adjacency\_Matrix (char\* name)

* 1. Реализацию метода печати остовного графа на консоль

void Print\_Spanning\_Tree(vector<pair<int, pair<int, int>>> graph)

* 1. Реализацию записи графа с выделенным алгоритмом Краскала остовом на языке DOT в файл

void Write\_To\_Kruskal\_Gv(vector<vector<pair<int, int>>> graph, vector<pair<int, pair<int, int>>> spanning\_tree)

* 1. Реализацию записи графа с выделенным алгоритмом Прима остовом на языке DOT в файл

void Write\_To\_Prim\_Gv(vector<vector<int>> graph, vector<pair<int, pair<int, int>>> spanning\_tree)

* Файл priority\_queue.h содержит:
  1. Директивы предпроцессора

#ifndef INCLUDE\_PRIORITY\_QUEUE\_H\_

#define INCLUDE\_PRIORITY\_QUEUE\_H\_

#include "four\_heap.h"

template<classType>

#endif// INCLUDE\_PRIORITY\_QUEUE\_H\_

* 1. Описание шаблонного класса Queue\_on\_heap:

template<classType>

class Queue\_on\_heap {

FOURheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

void Erase(const Type& val);

Type Top();

};

* 1. Реализацию методов класса Queue\_on\_heap:
     + Конструктор:

Queue\_on\_heap<Type>::Queue\_on\_heap();

Метод создает вершину кучи.

* + - Деструктор:

Queue\_on\_heap<Type>::~Queue\_on\_heap()

Метод удаляет кучу.

* + - Вставка:

void Queue\_on\_heap<Type>::Push(const Type& val);

Функция получает значение элемента и добавляет его в очередь в соответствие с величиной значения.

* + - Удаление первого в очереди:

void Queue\_on\_heap<Type>::Pop();

Функция удаляет элемент, стоящий вначале очереди.

* + - Удаление по значению:

void Queue\_on\_heap<Type>::Erase(const Type& val);

Функция удаляет элемент, значение которого было получено в параметрах.

* + - Получение первого в очереди:

Type Queue\_on\_heap<Type>::Top();

Функция возвращает значение первого в очереди элемента.

* Файл spanning\_tree.h содержит:
  1. Директивы предпроцессора и объявления констант

#ifndef INCLUDE\_SPANNING\_TREE\_H\_

#define INCLUDE\_SPANNING\_TREE\_H\_

#include <algorithm>

#include <vector>

#include <utility>

using std::vector;

using std::pair;

using std::make\_pair;

const int INF = 100000000;

#endif // INCLUDE\_SPANNING\_TREE\_H\_

* 1. Прототип функции поиска минимального остовного дерева с помощью алгоритма Краскала.

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

* 1. Прототип функции поиска минимального остовного дерева с помощью алгоритма Прима.

vector<pair<int, pair<int, int>>> Prim(vector<vector<int>> g, int st);

* Файл uf.h содержит:
  1. Директивы предпроцессора

#ifndef INCLUDE\_UF\_H\_

#define INCLUDE\_UF\_H\_

#include "spanning\_tree.h"

#endif // INCLUDE\_UF\_H\_

* 1. Реализацию функции поиска множества, которому принадлежит элемент.

int Find(vector<int> parent, int p)

* 1. Реализацию функции объединения двух элементов в одно множество.

void Union(vector<int>\* parent, vector<int>\* count, int p, int q)

* Файл Kruskal.cpp содержит реализацию алгоритма Краскала поиска минимального остовного дерева.
* Файл Prim.cpp содержит реализацию алгоритма Прима поиска минимального остовного дерева.
* Файл main\_test.cpp содержит функцию для запусков всех тестов:

int main(intac, char\* av[]);

* Файл test\_kruskal.cpp содержит тесты для проверки корректности реализации алгоритма Краскала.
* Файл test\_prim.cpp содержит тесты для проверки корректности реализации алгоритма Прима.

## 3.3 Описание структур данных

В разработанной программе использованы следующие структуры данных:

1. Класс 4-кучи

template <class Type>

class FOURheap {

int size;

int d;

public:

Type\* val;

FOURheap();

~FOURheap();

int GetSize() { return size; }

bool operator == (const FOURheap<Type>& B);

bool operator != (const FOURheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

1. Класс приоритетной очереди на 4-куче:

template<classType>

class Queue\_on\_heap {

FOURheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

void Erase(const Type& val);

Type Top();

};

1. Вектор g с данными типа vector<pair<int, int>>> из стандартной библиотеки std для представления графа в виде списка смежности:

vector<vector<pair<int, int>>> g

1. Вектор g с данными типа vector<int> из стандартной библиотеки std для представления графа в виде матрицы смежности:

vector<vector<int>> g

1. Вектор res с данными типа pair<int, <pair<int, int>>> для представления остовного графа.

vector<pair<int, pair<int, int>>> res

1. Вектор parent с данными типа int для реализации древовидной структуры в системе разделенных множеств

vector<int> parent

1. Вектор count с данными типа int для реализации рангов вершин в системе разделенных множеств.

vector<int> count

1. Векторы min\_e, sel\_e с данными типа int для хранения веса минимального ребра для каждой вершины графа и номера вершины, к которой ведет это ребро.

vector<int> min\_e, sel\_е;

1. Вектор used с данными типа bool для хранения посещенных вершин.

vector<bool> used

## 3.4 Описание алгоритмов

В лабораторной работе для поиска минимального остова были использованы следующие алгоритмы:

### Алгоритмы 4-куч.

1. Алгоритм вставки элемента по значению в кучу.

Функция получает на входе значение элемента, требующего вставки.

* 1. Функция добавляет элемент в конец кучи.
  2. Функция увеличивает размер кучи.
  3. Функция производит «всплытие» добавленного элемента.

1. Алгоритм удаления элемента по ключу.

Функция получает ключ элемента, который нужно удалить.

* 1. На место элемента, который требуется удалить, функция вставляет последний элемент кучи.
  2. Если вставленный элемент меньше своего родителя, функция проводит «всплытие» этого элемента. В противном случае происходит «погружение» элемента.

1. Алгоритм удаления минимального элемента кучи.
   1. На место первого элемента кучи функция вставляет последний элемент и производит «погружение» этого элемента.
2. Алгоритм поиска минимального потомка элемента.

Функция получает ключ элемента, минимального ребенка которого нужно найти.

* 1. Если ключ первого потомка больше размера кучи – значит, у элемента нет детей.
  2. Функция получает ключ первого ребенка элемента и сохраняет значение этого потомка как минимальное.
  3. Функция получает ключ последнего потомка элемента.
  4. Функция перебирает всех детей элемента от первого до последнего и сравнивает их значение с минимальным. Если был найден потомок с меньшим значением, функция сохраняет его ключ, а значение сохраняет как новое минимальное.

Функция возвращает ключ минимального потомка.

1. Алгоритм «погружения»

Функция получает на входе ключ элемента, который нужно «погрузить».

* 1. Функция находит минимального потомка элемента.
  2. Функция меняет элемент и его минимального потомка местами.
  3. Находим элемент, который является новым минимальным потомком для исходного элемента.
  4. Функция повторяет пункты 2 и 3 до тех пор, пока значение элемента больше значение минимального потомка.

1. Алгоритм «всплытия».

Функция получает на входе ключ элемента, который нужно «поднять».

* 1. Функция находит родителя элемента.
  2. Производится транспозиция элемента и его родителя.
  3. Функция находит нового родителя элемента.
  4. Пункты 2 и 3 повторяются до тех пор, пока значение элемента меньше значения его родителя.

1. Алгоритм «окучивания».

Функция проводит последовательное «погружение» узлов size-1…0.

### Алгоритмы системы разделенных множеств.

1. Алгоритм поиска множества, которому принадлежит элемент.

Функция получает на входе массив, в котором хранятся номера родителей всех элементов, и номер вершины.

* 1. Функция сохраняет номер вершины.
  2. Пока функция не дошла до корня, она ищет родителя текущего элемента и переходит к нему, делая текущим элементом.

Функция возвращает номер множества, в котором лежит корень дерева, содержащего полученный в параметрах элемент.

1. Алгоритм объединения элементов.

Функция получает на входе два элемента, которые нужно объединить, указатель на массив, хранящий номера родителей для каждого элемента, и указатель на массив рангов вершин.

* 1. Функция определяет, каким множествам принадлежат полученные элементы.
  2. Если ранг корня дерева, содержащего первый элемент, больше ранга дерева, содержащего второй элемент, то второй элемент добавляется во множество с первым элементом, и ранг первого корня увеличивается.
  3. В противном случае первый элемент добавляется во множество со вторым элементом, и ранг второго корня увеличивается.

### Алгоритм Краскала

Функция получает на вход граф в виде списка смежности.

1. Функция помещает каждую вершину графа в свое множество.
2. Функция берет вершину графа и проверяет:
   1. Если в паре с i-ой вершиной, ребро равно -1 (уже учитывалось ранее или не существует), функция переходит к следующей паре.
   2. В противном случае, функция добавляет в массив ребер пару «вес ребра – текущая вершина – второй конец ребра»
   3. пункты a-b повторяются для всех вершин, смежных с текущей.
3. Пункт 2 повторяется для всех вершин графа.
4. Функция сортирует массив ребер в порядке возрастания веса ребер.
5. Функция берет ребро из массива ребер. Если вершины, соединяемые этим ребром, принадлежат разным множествам, тогда ребро добавляется в результирующий массив, а вершины объединяются в одно множество.
6. Пункт 5 повторяется для всех ребер.

На выходе функция выдает массив ребер в виде пар (вес, (вершина 1, вершина 2)).

### Алгоритм Прима

Функция получает на входе граф в виде матрицы смежности и стартовую вершину.

1. Функция заполняет веса всех ребер начальным значением, равным большому константному значению.
2. Вес минимального ребра для стартовой вершины кладется равным 0.
3. Функция добавляет в очередь пару (вес минимального ребра для стартовой вершины, стартовая вершина)
4. Из очереди извлекается пара. Второй элемент пары сохраняется как вершина, до которой ведет ребро минимального веса для текущей вершины.
5. Функция отмечает вершину из пункта 4 как посещенную.
6. Функция проверяет: если для вершины из пункта 4 вершина, соединенная с ней ребром минимального веса, существует, тогда функция кладет в результирующий массив ребро минимального веса для вершины из пункта 4 и вершины, которые оно соединяет.
7. Из матрицы смежности функция получает расстояние от вершины из пункта 4 до j-ой вершины графа. Если это расстояние меньше, чем вес минимального ребра для j-ой вершины и j-ая вершина еще не посещена, тогда пара (минимальное ребро j-ой вершины, j-ая вершина) удаляется из очереди, обновляется значение веса минимального ребра для этой вершины и значение вершины, к которой ведет ребро минимального веса (вершина из пункта 4). После этого в очередь добавляется новая пара (минимальное ребро j-ой вершины, j-ая вершина)
8. Пункт 7 повторяется для всех вершин графа.
9. Пункты 4-7 повторяются n раз, где n – количество вершин графа, так как остовное дерево по определению должно содержать все вершины исходного графа. Если прошло меньше итераций, чем n, а очередь уже пуста, значит, был передан лес (несвязный граф), работать с которым алгоритм не может.

Функция возвращает массив ребер в формате пар вида (вес ребра, (вершина 1, вершина 2)).

### Алгоритмы чтения, печати и преобразования графов

1. Алгоритм записи графа из файла в список смежности.

Функция получает на входе строку, содержащую путь до файла.

* 1. Функция открывает файл и прочитывает несколько первых элементов, требуемых для оформления текстового документа, но не несущих полезной информации для программы.
  2. Функция читает символ и определяет:
     1. Если прочитанный символ – символ перехода на новую строку, тогда:
        1. Если массив, в который кладутся пары (вершина 2, вес ребра) для i-ой вершины, пуст, значит, i-ая вершина не смежна ни с какими другими вершинами или связи с этой вершиной были усчитаны ранее, и в результирующий список смежности на i-ую позицию добавляется массив, состоящий из пары (-1, -1).
        2. В противном случае в результирующий список смежности на i-ую позицию добавляется вектор, содержащий пары смежных с i-ой вершиной вершин и вес ребра между ними.
        3. Функция прочитывает вспомогательные символы.
     2. Иначе функция сохраняет прочитанный символ как номер смежной вершины.
     3. Функция проверяет следующий значимый символ.
        1. Если это символ перехода на новую строку, значит, пользователь допустил ошибку при создании текстового файла и не указал вес ребра. Функция выдает ошибку.
        2. В противном случае функция сохраняет символ как вес ребра между текущей и смежной вершиной и добавляет в массив смежных с текущей вершиной вершин новую пару.
  3. Пункт 3 повторяется до тех пор, пока не будет получен символ конца файла.
  4. Функция добавляет в результирующий список последний созданный массив пар и закрывает файл.

Функция возвращает список смежности.

1. Алгоритм записи в матрицу смежности.

Функция получает на входе строку, содержащую путь до файла.

* 1. Функция открывает файл.
  2. Функция читает символ:
     1. Если это символ перехода на новую строку, функция кладет в результирующую матрицу i-ую строку, которая есть вектор весов ребер от i-ой вершины до всех остальных.
     2. В случае если это « - », что значит отсутствие ребра между вершинами, то в i-ую строку на соответствующую позицию кладется константа INF, означающая отсутствие ребра.
     3. Если это число, то оно добавляется в i-ую строку.
  3. Пункт 3 повторяется до тех пор, пока не встретится знак конца файла.
  4. Функция добавляет в результирующую матрицу последнюю созданную строку и закрывает файл.

Функция возвращает матрицу смежности.

1. Алгоритм записи в файл графа с выделением остова, полученного алгоритмом Краскала, на языке DOT.

Функция получает на входе граф в виде списка смежности и найденный в нем остов в виде массива ребер.

* 1. Функция открывает для записи файл, в который требуется сохранить графы. Если файла по указанному пути не существует, он будет создан.
  2. Функция печатает в файл строку, определяющую цвет ребер остовного дерева.
  3. Для всех вершин остова функция повторяет следующие действия:
     1. Сохранить номера первой и второй вершин и вес ребра между ними.
     2. Напечатать в файл номер первой вершины.
     3. Напечатать строку « -- », которая на языке DOT означает неориентированное ребро.
     4. Напечатать в файл номер второй вершины.
     5. Напечатать строку, определяющую стиль ребра между текущими вершинами и подпись к ребру, отражающую его вес.
     6. Функция ищет в списке смежных с первой вершиной вершин пару (вершина 2, вес ребра между вершинами 1 и 2) и удаляет ее (чтобы не добавить ребро как принадлежащее основном графу).
  4. Функция повторяет все действия пункта 3 для вершин основного графа.
  5. Функция закрывает файл.

1. Алгоритм записи в файл графа с выделением остова, полученного алгоритмом Прима, на языке DOT.

Функция получает на входе граф в виде матрицы смежности и найденный в нем остов в виде массива ребер.

* 1. Функция открывает для записи файл, в который требуется сохранить графы. Если файла по указанному пути не существует, он будет создан.
  2. Функция печатает в файл строку, определяющую цвет ребер остовного дерева.
  3. Для всех вершин остова функция повторяет следующие действия:
     1. Сохранить номера первой и второй вершин и вес ребра между ними.
     2. Напечатать в файл номер первой вершины.
     3. Напечатать строку « -- », которая на языке DOT означает неориентированное ребро.
     4. Напечатать в файл номер второй вершины.
     5. Напечатать строку, определяющую стиль ребра между текущими вершинами и подпись к ребру, отражающую его вес.
     6. Функция кладет в матрицу в ячейки с номерами [вершина1][вершина2] и [вершина2][вершина1] значение INF, чтобы далее не добавить ребро как принадлежащее основному дереву).
  4. Функция повторяет все действия пункта 3 для вершин основного графа при условии, если вес ребра между двумя вершинами не равен INF (то есть ребро существует и не принадлежит остову).
  5. Функция закрывает файл.

# Заключение

В ходе лабораторной работы были получены файлы Kruskal\_spanning\_tree.txt и Prim\_spanning\_tree.txt. Результаты визуализации этих файлов в программе graphviz показаны на рисунках 1 и 2.

Для получения указанных файлов были реализованы алгоритм Краскала с использованием представления разделенных множеств с помощью древовидной структуры и использованием рангов вершин для графа в виде списка смежности. Также был реализован алгоритм Прима с использованием приоритетной очереди, реализованной на 4-куче для графа в виде матрицы смежности.

|  |  |
| --- | --- |
| kruskal.png | prim.png |
| Рис.1 Граф с выделенным остовом, найденным с помощью алгоритма Краскала. | Рис.2 Граф с выделенным остовом, найденным с помощью алгоритма Прима. |

Полученные результаты полностью удовлетворяют поставленным учебно-практическим задачам.

# Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е издание. – М.: Вильямс, 2010.
2. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород ННГУ, 2005.

# 

# Приложения

## Приложение1 (файл main.cpp)

#include <cstdlib>

#include <fstream>

#include "graph.h"

int main() {

printf("KRUSKAL`S ALGORITHM\n");

printf("ENTER THE PATH TO THE FILE: ");

char s1[150];

scanf("%s", s1);

printf("\n");

Print\_Adjacency\_List(s1);

vector<vector<pair<int, int>>> graph1 = Read\_To\_Adjacency\_List(s1);

printf("\n");

vector<pair<int, pair<int, int>>> res\_Kruskal = Kruskal(graph1);

Print\_Spanning\_Tree(res\_Kruskal);

Write\_To\_Kruskal\_Gv(graph1, res\_Kruskal);

printf("\n");

printf("PRIM`S ALGORITHM\n");

printf("ENTER THE PATH TO THE FILE: ");

char s2[150];

scanf("%s", s2);

printf("\n");

Print\_Adjacency\_Matrix(s2);

vector<vector<int>> graph2 = Read\_To\_Adjacency\_Matrix(s2);

printf("\n");

vector<pair<int, pair<int, int>>> res\_Prim = Prim(graph2, 1);

Print\_Spanning\_Tree(res\_Prim);

Write\_To\_Prim\_Gv(graph2, res\_Prim);

return 0;

}

## Приложение 2 (файл four\_heap.h)

#ifndef INCLUDE\_FOUR\_HEAP\_H\_

#define INCLUDE\_FOUR\_HEAP\_H\_

#include <stdexcept>

const int d\_ = 4;

const int MAX\_SIZE = 1000;

using std::logic\_error;

template <class Type>

class FOURheap {

int size;

int d;

public:

Type\* val;

FOURheap();

~FOURheap();

int GetSize() { return size; }

bool operator == (const FOURheap<Type>& B);

bool operator != (const FOURheap<Type>& B);

Type& operator[] (int i);

void Reduction(int i, const Type& con);

void Swap(int i, int j);

int MinChild(int i);

void Insert(const Type& val\_);

void Erase(int i);

void EraseMin();

void Immertion(int i); // down

void Emertion(int i); // up

void DoingHeaps();

bool IsFull() { return (size >= MAX\_SIZE); }

bool IsEmpty() { return size == 0; }

};

template<class Type>

FOURheap<Type>::FOURheap() {

d = d\_;

size = 0;

val = new Type[MAX\_SIZE];

}

template<class Type>

FOURheap<Type>::~FOURheap() {

delete[] val;

}

template<class Type>

bool FOURheap<Type>::operator==(const FOURheap<Type>& B) {

if (size != B.size)

return false;

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

if (val[i] != B.val[i])

return false;

return true;

}

template<class Type>

bool FOURheap<Type>::operator!=(const FOURheap<Type>& B) {

return !((\*this) == B);

}

template<class Type>

Type& FOURheap<Type>::operator[](int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

return val[i];

}

template<class Type>

void FOURheap<Type>::Reduction(int i, const Type & con) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

val[i] -= con;

Emertion(i);

}

template<class Type>

void FOURheap<Type>::Swap(int i, int j) {

if ((i < 0) || (j < 0) || (i >= size) || (j >= size))

throw logic\_error("Out of range");

Type tmp = val[i];

val[i] = val[j];

val[j] = tmp;

}

template<class Type>

void FOURheap<Type>::Insert(const Type& val\_) {

if (size >= MAX\_SIZE)

throw logic\_error("Out of range");

val[size] = val\_;

size++;

Emertion(size - 1);

}

template<class Type>

void FOURheap<Type>::Erase(int i) {

if (size == 0)

throw logic\_error("Heap is empty");

if (i< 0 || i >= size)

throw logic\_error("Out of range");

val[i] = val[size - 1];

size--;

if (size > 1) {

if ((i != 0) && (val[i] < val[(i - 1) / d]))

Emertion(i);

else

Immertion(i);

}

}

template<class Type>

void FOURheap<Type>::EraseMin() {

if (size == 0)

throw logic\_error("Heap is empty");

val[0] = val[size - 1];

size--;

if (size != 0)

Immertion(0);

}

template<class Type>

int FOURheap<Type>::MinChild(int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

if (d\*i + 1 >= size)

return 0;

int first = d\*i + 1;

Type min\_val = val[first];

int last = (i + 1) \*d;

if (last >= size)

last = size - 1;

for (int k = first + 1; k <= last; ++k)

if (val[k] < min\_val) {

min\_val = val[k];

first = k;

}

return first;

}

template<class Type>

void FOURheap<Type>::Immertion(int i) {

if (i< 0 || i >= size)

throw logic\_error("Out of range");

int s = MinChild(i);

while ((s != 0) && (val[i] > val[s])) {

Swap(i, s);

i = s;

s = MinChild(i);

}

}

template<class Type>

void FOURheap<Type>::Emertion(int i) {

if (i < 0 || i >= size)

throw logic\_error("Out of range");

int parent = (i - 1) / d;

while ((i != 0) && (val[parent] > val[i])) {

Swap(i, parent);

i = parent;

parent = (i - 1) / d;

}

}

template<class Type>

void FOURheap<Type>::DoingHeaps() {

if (size > 0) {

for (int i = size - 1; size >= 0; --i)

Immertion(i);

}

}

#endif // INCLUDE\_FOUR\_HEAP\_H\_

## Приложение 3 (файл graph.h)

#ifndef INCLUDE\_GRAPH\_H\_

#define INCLUDE\_GRAPH\_H\_

#include "spanning\_tree.h"

#include <stdio.h>

#include <stdexcept>

using std::logic\_error;

vector<vector<pair<int, int>>> Read\_To\_Adjacency\_List(char\* name) {

FILE\* f = fopen(name, "r");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

vector<vector<pair<int, int>>> graph;

int ch = 0;

vector<pair<int, int>>\* v = new vector<pair<int, int>>;

ch = fgetc(f);

while (ch != ':')

ch = fgetc(f);

while ((ch = fgetc(f)) != EOF) {

if (ch != ' ') {

if (ch == '\n') {

if ((\*v).size() == 0)

(\*v).push\_back(make\_pair(-1, -1));

graph.push\_back(\*v);

v = new vector<pair<int, int>>;

while (ch != ':')

ch = fgetc(f);

} else {

int vertex = ch;

ch = fgetc(f);

if ((ch = fgetc(f)) == '\n')

throw logic\_error("Error! Incorrect list!");

(\*v).push\_back(make\_pair(vertex - 48, ch - 48));

}

}

}

graph.push\_back(\*v);

fclose(f);

return graph;

}

void Print\_Adjacency\_List(char\* name) {

FILE\* f = fopen(name, "r");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

printf("GRAPH FROM THE FILE:\n");

int ch = 0;

ch = fgetc(f);

while (ch != ':')

ch = fgetc(f);

int i = 0;

printf("%d: ", i);

++i;

while ((ch = fgetc(f)) != EOF) {

if (ch != ' ') {

if (ch == '\n') {

printf("\n");

while (ch != ':')

ch = fgetc(f);

printf("%d: ", i);

++i;

} else {

printf("%-5d", ch - 48);

ch = fgetc(f);

if ((ch = fgetc(f)) == '\n')

throw logic\_error("Error! Incorrect list!");

printf("%-5d", ch - 48);

}

}

}

printf("\n");

}

void Print\_Adjacency\_Matrix(char\* name) {

FILE\* f = fopen(name, "r");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

printf("GRAPH FROM THE FILE:\n");

int ch = 0;

while ((ch = fgetc(f)) != EOF) {

if (ch != ' ') {

if (ch == '\n') {

printf("\n");

} else {

if (ch == '-')

printf("%-9s", "-");

else

printf("%-9d", ch - 48);

}

}

}

printf("\n");

}

vector<vector<int>> Read\_To\_Adjacency\_Matrix(char\* name) {

FILE\* f = fopen(name, "r");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

vector<vector<int>> graph;

int ch = 0;

vector<int>\* v = new vector<int>;

while ((ch = fgetc(f)) != EOF) {

if (ch != ' ') {

if (ch == '\n') {

graph.push\_back(\*v);

v = new vector<int>;

} else {

if (ch == '-')

v->push\_back(INF);

else

v->push\_back(ch - 48);

}

}

}

graph.push\_back(\*v);

fclose(f);

return graph;

}

void Print\_Spanning\_Tree(vector<pair<int, pair<int, int>>> graph) {

int n = graph.size();

printf("THE GENERATED SPANNING TREE:\n");

printf("%-9s", "vertex 1 ");

printf("| ");

printf("%-9s", "vertex 2 ");

printf("| ");

printf("%-9s", "weight");

printf("\n");

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

printf("%-9d| ", graph[i].second.first);

printf("%-9d| ", graph[i].second.second);

printf("%-6d\n", graph[i].first);

}

printf("\n");

}

void Write\_To\_Kruskal\_Gv(vector<vector<pair<int, int>>> graph,

vector<pair<int, pair<int, int>>> spanning\_tree) {

FILE\* f = fopen("C:/Users/qq/Kate/Uni/Graphs/Kruskal\_spanning\_tree.txt", "w");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

fputs("graph { \nedge[color = \"#2EB8E6\"]\n", f);

int n = spanning\_tree.size();

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

int vertex1 = spanning\_tree[i].second.first;

int vertex2 = spanning\_tree[i].second.second;

int weight = spanning\_tree[i].first;

fputc(vertex1 + 48, f);

fputs("--", f);

fputc(vertex2 + 48, f);

fputs("[style = \"bold\", label=\"", f);

fputc(weight + 48, f);

fputs("\"]\n", f);

vector<pair<int, int>>::iterator it = graph[vertex1].begin();

for (int i = 0; i < static\_cast<int>(graph[vertex1].size()); ++i) {

if (graph[vertex1][i] == (make\_pair(vertex2, weight))) {

graph[vertex1].erase(it);

continue;

}

++it;

}

}

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

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

int vertex1 = i;

int vertex2 = graph[i][j].first;

int weight = graph[i][j].second;;

fputc(vertex1 + 48, f);

fputs("--", f);

fputc(vertex2 + 48, f);

fputs("[color = black, label=\"", f);

fputc(weight + 48, f);

fputs("\"]\n", f);

fputc(vertex1 + 48, f);

fputs(" [shape = \"rect\", color = black]\n", f);

fputc(vertex2 + 48, f);

fputs(" [shape = \"rect\", color = black]\n", f);

}

fputs("}\n", f);

fclose(f);

}

void Write\_To\_Prim\_Gv(vector<vector<int>> graph, vector<pair<int, pair<int, int>>> spanning\_tree) {

FILE\* f = fopen("C:/Users/qq/Kate/Uni/Graphs/Prim\_spanning\_tree.txt", "w");

if (f == 0)

throw logic\_error("Error! Can`t open the file!");

fputs("graph { \nedge[color = \"#0DBD1C\"]\n", f);

int n = spanning\_tree.size();

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

int vertex1 = spanning\_tree[i].second.first;

int vertex2 = spanning\_tree[i].second.second;

int weight = spanning\_tree[i].first;

fputc(vertex1 + 48, f);

fputs("--", f);

fputc(vertex2 + 48, f);

fputs("[style = \"bold\", label=\"", f);

fputc(weight + 48, f);

fputs("\"]\n", f);

graph[vertex1][vertex2] = INF;

graph[vertex2][vertex1] = INF;

}

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

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

if ((graph[i][j] != INF) && (i != j)) {

fputc(i + 48, f);

fputs("--", f);

fputc(j + 48, f);

fputs("[color = black, label=\"", f);

fputc(graph[i][j] + 48, f);

fputs("\"]\n", f);

fputc(i + 48, f);

fputs(" [shape = \"rect\", color = black]\n", f);

fputc(j + 48, f);

fputs(" [shape = \"rect\", color = black]\n", f);

graph[j][i] = INF;

}

fputs("}\n", f);

fclose(f);

}

#endif // INCLUDE\_GRAPH\_H\_

## Приложение 4 (файл priority\_queue.h)

#ifndef INCLUDE\_PRIORITY\_QUEUE\_H\_

#define INCLUDE\_PRIORITY\_QUEUE\_H\_

#include "four\_heap.h"

template <class Type>

class Queue\_on\_heap {

FOURheap<Type>\* heap;

public:

Queue\_on\_heap();

~Queue\_on\_heap();

int GetSize() { return heap->GetSize(); }

bool IsFull() { return heap->IsFull(); }

bool IsEmpty() { return heap->IsEmpty(); }

void Push(const Type& val);

void Pop();

void Erase(const Type& val);

Type Top();

};

template<class Type>

Queue\_on\_heap<Type>::Queue\_on\_heap() {

heap = new FOURheap<Type>();

}

template<class Type>

Queue\_on\_heap<Type>::~Queue\_on\_heap() {

delete heap;

}

template<class Type>

void Queue\_on\_heap<Type>::Push(const Type & val) {

if (IsFull())

throw logic\_error("Heap is full");

heap->Insert(val);

}

template<class Type>

void Queue\_on\_heap<Type>::Pop() {

if (IsEmpty())

throw logic\_error("Heap is empty");

heap->EraseMin();

}

template<class Type>

void Queue\_on\_heap<Type>::Erase(const Type& val) {

for (int i = 0; i < heap->GetSize(); ++i)

if (heap->val[i] == val)

heap->Erase(i);

}

template<class Type>

Type Queue\_on\_heap<Type>::Top() {

if (IsEmpty())

throw logic\_error("Heap is empty");

return (\*heap)[0];

}

#endif // INCLUDE\_PRIORITY\_QUEUE\_H\_

## Приложение 5 (файл spanning\_tree.h)

#ifndef INCLUDE\_SPANNING\_TREE\_H\_

#define INCLUDE\_SPANNING\_TREE\_H\_

#include <algorithm>

#include <vector>

#include <utility>

using std::vector;

using std::pair;

using std::make\_pair;

const int INF = 100000000;

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

vector<pair<int, pair<int, int>>> Prim(vector<vector<int>> g, int st);

#endif // INCLUDE\_SPANNING\_TREE\_H\_

## Приложение 6 (файл uf.h)

#ifndef INCLUDE\_UF\_H\_

#define INCLUDE\_UF\_H\_

#include "spanning\_tree.h"

int Find(vector<int> parent, int p) {

int res = p;

while (parent[res] != res) {

res = parent[res];

}

return res;

}

void Union(vector<int>\* parent, vector<int>\* count, int p, int q) {

int x = Find(\*parent, p);

int y = Find(\*parent, q);

if ((\*count)[x] > (\*count)[y]) {

(\*parent)[y] = x;

(\*count)[x] ++;

} else {

(\*parent)[x] = y;

(\*count)[y] ++;

}

}

#endif // INCLUDE\_UF\_H\_

## Приложение 7 (файл kruskal.cpp)

#include "spanning\_tree.h"

#include "uf.h"

vector<pair<int, pair<int, int>>> Kruskal(vector<vector<pair<int, int>>> g) {

int n = g.size();

vector<pair<int, pair<int, int>>> res;

vector<int> parent(n);

vector<int> count(n, 1);

vector<pair<int, int>>::iterator it;

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

parent[i] = i;

vector<pair<int, pair<int, int>>> edges; // weight - vertex1 - vertex2

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

for (it = g[i].begin(); it != g[i].end(); ++it) {

if ((\*it).second == -1)

continue;

edges.push\_back(make\_pair((\*it).second, make\_pair(i, (\*it).first)));

}

sort(edges.begin(), edges.end());

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

int vertex1 = edges[i].second.first;

int vertex2 = edges[i].second.second;

int x = Find(parent, vertex1);

int y = Find(parent, vertex2);

if (x != y) {

res.push\_back(edges[i]);

Union(&parent, &count, vertex1, vertex2);

}

}

return res;

}

## Приложение 8 (файл prim.cpp)

#include "spanning\_tree.h"

#include "priority\_queue.h"

#include <stdio.h>

using std::min;

using std::max;

vector<pair<int, pair<int, int>>> Prim(vector<vector<int>> g, int st) {

int n = g.size();

vector<pair<int, pair<int, int>>> res;

if (n == 0)

return res;

vector<int> min\_e(n, INF), sel\_e(n, -1);

min\_e[st] = 0;

Queue\_on\_heap<pair<int, int>> q;

vector<bool> used(n + 1, false);

q.Push(make\_pair(min\_e[st], st));

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

if (q.IsEmpty())

throw logic\_error("No MST");

int vertex = q.Top().second;

q.Pop();

used[vertex] = true;

if (sel\_e[vertex] != -1) {

int mi = min(sel\_e[vertex], vertex);

int ma = max(sel\_e[vertex], vertex);

res.push\_back(make\_pair(min\_e[vertex], make\_pair(mi, ma)));

}

for (size\_t j = 0; j < g[vertex].size(); ++j) {

int to = j;

int cost = g[vertex][j];

if (cost < min\_e[to] && !used[to]) {

q.Erase(make\_pair(min\_e[to], to));

min\_e[to] = cost;

sel\_e[to] = vertex;

q.Push(make\_pair(min\_e[to], to));

}

}

}

return res;

}

## Приложение 9 (файл test\_kruskal.cpp)

#include <gtest/gtest.h>

#include "spanning\_tree.h"

TEST(Kruscal, can\_work\_with\_empty\_graph) {

vector<vector<pair<int, int>>> g;

vector<pair<int, pair<int, int>>> res = Kruskal(g);

EXPECT\_TRUE(res.empty());

}

TEST(Kruskal, can\_work\_with\_already\_spanning\_tree) {

vector<pair<int, int>> v0 = { make\_pair(1, 10) };

vector<pair<int, int>> v1 = { make\_pair(2, 2), make\_pair(3, 3) };

vector<pair<int, int>> v2 = { make\_pair(-1, -1) };

vector<pair<int, int>> v3 = { make\_pair(4, 7), make\_pair(5, 8) };

vector<pair<int, int>> v4 = { make\_pair(-1, -1) };

vector<pair<int, int>> v5 = { make\_pair(-1, -1) };

vector<vector<pair<int, int>>> g = { v0, v1, v2, v3, v4, v5 };

vector<pair<int, pair<int, int>>> res = Kruskal(g);

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

{ make\_pair(2, make\_pair(1, 2)),

make\_pair(3, make\_pair(1, 3)),

make\_pair(7, make\_pair(3, 4)),

make\_pair(8, make\_pair(3, 5)),

make\_pair(10, make\_pair(0, 1))

};

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

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

}

TEST(Kruskal, correct\_work1) {

vector<pair<int, int>> v0 = { make\_pair(1, 1), make\_pair(3, 5), make\_pair(2, 4) };

vector<pair<int, int>> v1 = { make\_pair(3, 2) };

vector<pair<int, int>> v2 = { make\_pair(3, 3) };

vector<pair<int, int>> v3 = { make\_pair(-1, -1) };

vector<vector<pair<int, int>>> g = { v0, v1, v2, v3 };

vector<pair<int, pair<int, int>>> res = Kruskal(g);

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

{ make\_pair(1, make\_pair(0, 1)),

make\_pair(2, make\_pair(1, 3)),

make\_pair(3, make\_pair(2, 3)),

};

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

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

}

TEST(Kruskal, correct\_work2) {

vector<pair<int, int>> v0 = { make\_pair(1, 5), make\_pair(3, 2), make\_pair(4, 7) };

vector<pair<int, int>> v1 = { make\_pair(2, 8), make\_pair(4, 1) };

vector<pair<int, int>> v2 = { make\_pair(3, 5), make\_pair(4, 8) };

vector<pair<int, int>> v3 = { make\_pair(4, 4) };

vector<pair<int, int>> v4 = { make\_pair(-1, -1) };

vector<vector<pair<int, int>>> g = { v0, v1, v2, v3, v4 };

vector<pair<int, pair<int, int>>> res = Kruskal(g);

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

{ make\_pair(1, make\_pair(1, 4)),

make\_pair(2, make\_pair(0, 3)),

make\_pair(4, make\_pair(3, 4)),

make\_pair(5, make\_pair(2, 3)),

};

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

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

}

TEST(Kruskul, can\_work\_with\_forest) {

vector<pair<int, int>> v0 = { make\_pair(1, 10), make\_pair(2, 8)};

vector<pair<int, int>> v1 = { make\_pair(-1, -1) };

vector<pair<int, int>> v2 = { make\_pair(-1, -1) };

vector<pair<int, int>> v3 = { make\_pair(4, 11), make\_pair(5, 9) };

vector<pair<int, int>> v4 = { make\_pair(-1, -1) };

vector<pair<int, int>> v5 = { make\_pair(-1, -1) };

vector<vector<pair<int, int>>> g = { v0, v1, v2, v3, v4, v5 };

vector<pair<int, pair<int, int>>> res = Kruskal(g);

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

{ make\_pair(8, make\_pair(0, 2)),

make\_pair(9, make\_pair(3, 5)),

make\_pair(10, make\_pair(0, 1)),

make\_pair(11, make\_pair(3, 4)),

};

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

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

}

## Приложение 10 (файл test\_prim.cpp)

#include <gtest/gtest.h>

#include "spanning\_tree.h"

TEST(Prim, Can\_work\_with\_empty\_graph) {

vector<vector<int>> g;

vector<pair<int, pair<int, int>>> res = Prim(g, 0);

EXPECT\_TRUE(res.empty());

}

TEST(Prim, Can\_work\_with\_already\_spanning\_tree) {

vector<vector<int>> g =

{{ 0, 10, INF, INF, INF },

{ 10, 0, 5, INF, INF },

{ INF, 5, 0, 8, 3 },

{ INF, INF, 8, 0, INF },

{ INF, INF, 3, INF, 8 } };

vector<pair<int, pair<int, int>>> res = Prim(g, 0);

vector<pair<int, pair<int, int>>> exp = {

make\_pair(10, make\_pair(0, 1)),

make\_pair(5, make\_pair(1, 2)),

make\_pair(3, make\_pair(2, 4)),

make\_pair(8, make\_pair(2, 3))

};

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

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

}

TEST(Prim, Correct\_work) {

vector<vector<int>> g =

{ { 0, 5, INF, 2, 7 },

{ 5, 0, 8, INF, 1 },

{ INF, 8, 0, 5, 8 },

{ 2, INF, 5, 0, 4 },

{ 7, 1, 8, 4, 0 } };

vector<pair<int, pair<int, int>>> res = Prim(g, 0);

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

{ make\_pair(2, make\_pair(0, 3)),

make\_pair(4, make\_pair(3, 4)),

make\_pair(1, make\_pair(1, 4)),

make\_pair(5, make\_pair(2, 3)),

};

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

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

}

TEST(Prim, Cant\_work\_with\_forest) {

vector<vector<int>> g =

{ { 0, INF, 15, INF, 9 },

{ INF, 0, INF, 10, INF },

{ 15, INF, 0, INF, 1 },

{ INF, 10, INF, 0, INF },

{ 9, INF, 1, INF, 0 } };

EXPECT\_ANY\_THROW(Prim(g, 0));

}

## Приложение 11 (файл main\_test.cpp)

#include<gtest/gtest.h>

int main(intac, char\* av[]) {

testing::InitGoogleTest(&ac, av);

return RUN\_ALL\_TESTS();

}