Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе:

**Алгоритмы Краскала и Прима поиска**

**минимального остовного дерева**

Выполнил:

студент ИИТММ

группы 0823-1  
Ермолаев М.А.

Проверил:

к.т.н., ассистент каф. ПрИнж ИИТММ

Сиднев А.А.

Нижний Новгород

2017 г.

**Содержание**

[Введение 3](#_Toc486172685)

[Постановка задачи 4](#_Toc486172686)

[Руководство пользователя 5](#_Toc486172687)

[Руководство программиста 7](#_Toc486172688)

[Описание структуры программы 7](#_Toc486172689)

[Описание структур данных 8](#_Toc486172690)

[Описание алгоритмов 9](#_Toc486172691)

[Алгоритмы методов класса UnionFind (Разделённые множества) 9](#_Toc486172692)

[Алгоритмы методов класса Five\_Heap (5-куча) 10](#_Toc486172693)

[Алгоритм Краскала 11](#_Toc486172694)

[Алгоритм Прима 11](#_Toc486172695)

[Заключение 12](#_Toc486172696)

[Литература 13](#_Toc486172697)

# **Введение**

**Минимальное остовное дерево** в связанном взвешенном неориентированном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

Существует несколько алгоритмов для нахождения минимального остовного дерева. В данной лабораторной работе будут рассмотрены следующие алгоритмы:

* **Алгоритм Краскала** - впервые описан Джозефом Краскалом в 1956 году.
* **Алгоритм Прима** - был открыт в 1930 году чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 году, и, независимо от них, Э. Дейкстрой в 1959 году.

# **Постановка задачи**

**Алгоритм Краскала** необходимо реализовать с использованием разделённых множеств.

* Для работы с алгоритмом необходимо использовать представление графа в виде матрицы смежности;
* Для работы с разделёнными множествами необходимо реализовать класс, использующий представление разделённых множеств с помощью древовидной структур;
* Для сортировки рёбер необходимо использовать сортировку с помощью 5-кучи.

**Алгоритма Прима** необходимо реализовать с использованием приоритетной очереди.

* Для работы с алгоритмом необходимо использовать представление графа в виде матрицы инцидентности;
* Для реализации приоритетной очереди необходимо использовать 5-кучу.

**Визуализация графов**

При визуализации графов все ребра должны быть подписаны их весами, а вершины - соответствующими номерами вершин. Минимальное остовное дерево должно быть выделено отдельным цветом.

# **Руководство пользователя**

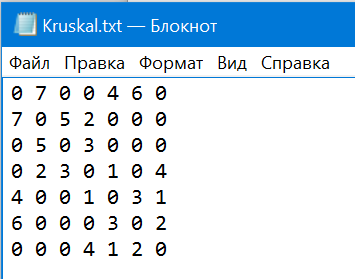
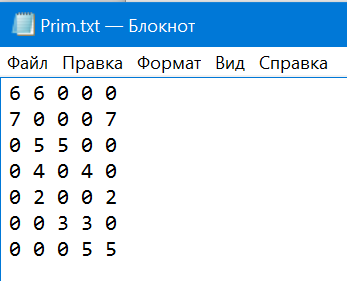
До работы с программой необходимо создать 3 текстовых документа:

1. Для работы с алгоритмом Краскала;
2. Для работы с алгоритмом Прима;
3. Для визуализации графа.

Текстовые документы следует заполнить следующим образом:

* Для алгоритма Краскала необходимо записать граф в виде матрицы смежности;
* Для алгоритма Прима необходимо записать граф в виде матрицы инцидентности;
* Текстовый документ для визуализации графа не заполнять.

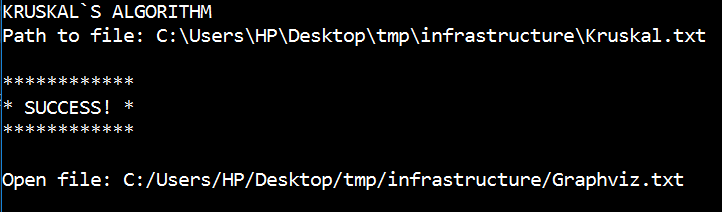
*Контрольный пример:*

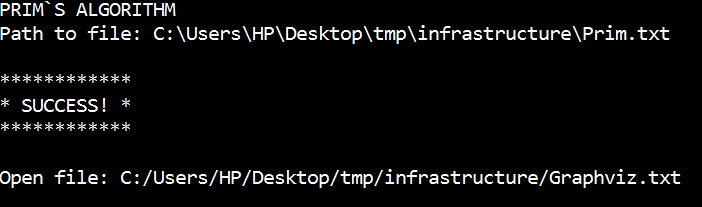
 

В консоль вводятся следующие данные:

* Алгоритм, с помощью которого будет реализован поиска минимального остовного дерева;
* Путь к ранее созданному текстовому документу с описанием графа;
* Путь к текстовому документу для записи результата работа программы

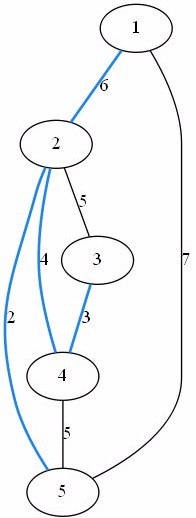
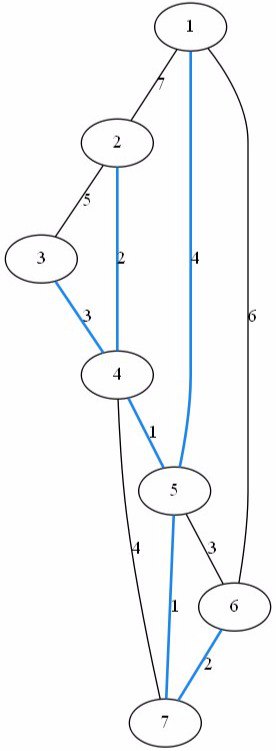
Результат работы программы записывается в текстовый файл для последующей визуализации графа. Для визуализации графа необходимо открыть программу Graphviz и выбрать текстовый файл, заполненный в результате работы программы.

*Контрольный пример:*



***Результаты:***

***Алгоритм Краскала: Алгоритм Прима:***



# **Руководство программиста**

## **Описание структуры программы**

Заголовочные файлы:

* **5-Heap.h** – содержит описание класса *Five\_Heap* 5-кучи и реализацию реализацию;
* **PriorityQueue** – содержит описание класса *PriorityQueue* приоритетной очереди, реализованный на куче;
* **Kruskal\_and\_Prim.h** – содержит прототипы функций *Kruskal* и *Prim*;
* **UnionFind** – содержит описание класс*а UnionFind* разделённых множеств;
* **Visual.h** – содержит прототипы функций *Reading\_the\_matrix, Init\_graph\_from\_adjacency\_matrix, Init\_graph\_from\_incidence\_matrix, Graphviz* необходимых для визуализации графов.

Файлы исходного кода:

* **Kruskal\_and\_Prim.cpp** - содержит реализацию функций *Kruskal* и *Prim*;
* **UnionFind.cpp** – содержит реализацию методов класса *UnionFind*;
* **Visual.cpp** – содержит реализацию функций *Reading\_the\_matrix, Init\_graph\_from\_adjacency\_matrix, Init\_graph\_from\_incidence\_matrix, Graphviz*;
* **main.cpp** – содержит интерфейс работы с пользователем.

Тесты:

* **test\_Kruskal\_and\_Prim.cpp** – cодержит тесты, проверяющие корректность реализации алгоритмов Краскала и Прима;
* **test\_Heap.cpp -** cодержит тесты, проверяющие корректность работы методов класса *Five\_Heap*;
* **test\_UnionFind.cpp -** cодержит тесты, проверяющие корректность работы методов класса *UnionFind*.

## **Описание структур данных**

- Класс **UnionFind**

class UnionFind {

vector<int> v;

public:

UnionFind();

UnionFind(int);

~UnionFind();

void Union(int, int);

int Find(int);

int operator[](int index) {

return v[index];

}

};

- Класс **Five\_Heap**

template <typename T>

class Five\_Heap {

private:

vector<T> array;

void ShiftUp(int index);

void ShiftDown();

int MinSon(int index\_parent);

public:

void InsertElem(T e);

void DeleteMin();

T GetMin();

bool IsEmpty() {

return array.empty();

}

int GetSize() {

return array.size();

}

T operator[](int index) {

return array[index];

}

};

- Класс **PriorityQueue**

template <typename T>

class PriorityQueue: public Five\_Heap<T> {

};

## **Описание алгоритмов**

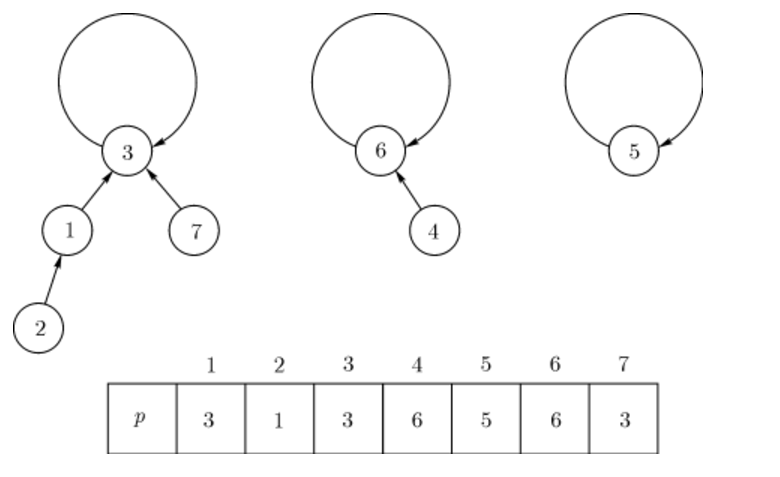
### **Алгоритмы методов класса UnionFind (Разделённые множества)**

**Теория:**

Пусть **U={1,2, .., N}** — множество, из элементов которого будет строиться коллекция. Каждое *подмножество* коллекции представляется *корневым деревом*, узлы которого являются элементами этого подмножества, то есть отождествляются с номерами из *множества* **{1,2, ..., N}.** Корень дерева используется в качестве имени соответствующего подмножества. Для каждого узла дерева определяется узел **v(x)**, являющийся его родителем в дереве. Если же **x** — корень, то полагаем **v(x) = x**.

В памяти компьютера это *дерево* будем представлять массивом **v[1,2, …,N]** так, что **v[x]** будет предком узла **x**, если **x** не является корнем, и **v[x] = x**, если **x** — корень.

Пример:



**Операции:**

***Поиск множества, которому принадлежит заданный элемент***

1. Если заданный элемент превышает размер массива – ошибка выхода за диапазон значений;
2. Создаём новую переменную res, которой присваиваем значение заданного элемента;
3. Пока v[res] не равно переменной res присваиваем res значение v[res];
4. Возвращаем полученное значение переменной res.

***Объединение множеств***

1. Ищем образующие элементы множеств, к которым принадлежат поданные на вход элементы и присваиваем их значения новым переменным, одну из которых назовём родителем;
2. Присваиваем элементу массива с индексом значения родителя значение элемента образующего множества, к которому принадлежит второй поданный на вход элемент.

### **Алгоритмы методов класса Five\_Heap (5-куча)**

5-Heap – пятиричное дерево, удовлетворяющее двум условиям:

1. Приоритет любой вершины не больше приоритета ее потомков
2. Дерево является полным 5-деревом, т.е. все уровни заполнены слева направо (за исключением может быть последнего)

Каждая вершина дерева – элемент массива. Если, например, рассматриваемая вершина имеет индекс i, то её родитель имеет индекс (i-1)/5, а её потомки: 5\*i+1, 5\*i+2, 5\*i+3, 5\*i+4, 5\*i+5;

#### *Поиск минимального элемента*

Минимальный элемент в куче – первый элемент массива.

#### *Вставка нового элемента*

1. Вставляем новый элемент в конец массива
2. Выполняем просеивание вверх

#### *Просеивание вверх*

1. На вход подается индекс i нового вставленного элемента
2. Пока i больше нуля и родитель элемента i больше элемента по индексу i, выполнять:
3. Поменять местами элементы массива с индексом i и индексом родителя i
4. Индексу i присвоить значение индекса родителя элемента, лежащего по текущему i.

#### *Удаление минимального элемента*

1. Меняем местами первый и последний элемент в массиве
2. Удаляем последний элемент в массиве
3. Выполняем просеивание вниз

#### *Просеивание вниз*

Индекс parent = 0, index\_min\_son равен индексу минимального сына нулевого элемента

Пока индекс минимального сына меньше индекса последнего и не меньше индекса первого элемента и значение родителя больше чем значение минимального сына

* Меняем местами значения родителя и минимального сына
* Индексу parent присваиваем значение index\_min\_son
* index\_min\_son присваиваем значение минимального сына для элемента с индексом parent

### **Алгоритм Краскала**

На вход подаётся вектор graph содержащий своими элементами кортежи, описывающие каждое ребро графа: вес, первая вершина, вторая вершина, а также число вершин графа count\_v.

spanning\_tree – вектор, содержащий рёбра остовного дерева;  
pq – очередь с приоритетом;  
min\_edge – хранит очередное минимальное ребро графа в очереди;  
uf – хранит множество вершин, входящих в минимальное остовное дерево.

1. Добавляем все рёбра в приоритетную очередь (далее, просто очередь). Рёбра сортируются по возрастанию весов.
2. Пока в остовном дереве не будет (count\_v – 1) ребро и очередь не пуста, выполняем:
   1. Извлекаем из очереди минимальное ребро и присваиваем его значение переменной min\_edge;
   2. Удаляем извлечённое минимальное ребро из очереди;
   3. Если первая и вторая вершина не принадлежат множеству вершин, входящих в остовное дерево:
      1. Добавляем их в это множество;
      2. Пушим в вектор остовных рёбер min\_edge;
3. Если по окончании цикла рёбер остовного дерева меньше, чем (count\_v – 1), значит заданное дерево не является связным – ошибка;
4. Вернуть минимальное остовное дерево.

### **Алгоритм Прима**

На вход подаётся вектор graph содержащий своими элементами кортежи, описывающие каждое ребро графа: вес, первая вершина, вторая вершина, а также число вершин графа count\_v

spanning\_tree – вектор, содержащий рёбра остовного дерева;  
pq – очередь с приоритетом;  
min\_edge – хранит очередное минимальное ребро графа в очереди;  
used[count\_v] – массив вершин, входящих в минимальное остовное дерево на данный момент.

1. Пока число рёбер не равно (count\_v – 1), выполняем:
   1. Отмечаем вершину v (изначально равна 0) использованной;
   2. Добавляем в очередь все рёбра, для которых одна из вершин равна v, а вторая не использована;
   3. Если очередь не пуста, выполняем:
2. Извлекаем из очереди минимальное ребро, присваиваем его значение переменной min\_edge и удаляем извлечённое минимальное ребро из очереди пока обе вершины ребра использованы;
3. Пушим в вектор остовных рёбер min\_edge;
4. Если использована первая вершина ребра min\_egde, присваиваем v значение второй вершины, иначе – значение первой вершины.

Иначе граф не является связным – ошибка;

1. Вернуть минимальное остовное дерево.

# **Заключение**

В данной работе реализованы алгоритмы Краскала и Прима поиска минимального остовного дерева. Программа позволяет загружать граф из файла и визуализировать полученный граф с минимальным остовным деревом другого цвета посредством программы Graphviz.

# **Литература**

T.A. Павловская – Программирование на языке высокого уровня

Разделённые множества

1. [http://www.intuit.ru/studies/courses/100/100/lecture/2925?page=3]

Алгоритм Краскала

1. [http://www.lotos-khv.narod.ru/algoritm/krascal.htm]
2. [https://ru.wikipedia.org/wiki/Алгоритм\_Краскала]

Алгоритм Прима

1. [http://www.lotos-khv.narod.ru/algoritm/prim.htm]
2. [https://ru.wikipedia.org/wiki/Алгоритм\_Прима]