Оглавление

[**Введение** 2](#_Toc122463581)

[Глава 1. Построение минимального остовного дерева 3](#_Toc122463582)

[Глава 2.1. Остовное дерево. Постановка задачи 3](#_Toc122463583)

[Глава 2.2. Алгоритм Дейкстры 3](#_Toc122463584)

[Глава 2.3. Алгоритм Прима 5](#_Toc122463585)

[Глава 2.4. Алгоритм Крускала 6](#_Toc122463586)

[Глава 2.5. Различия в скорости работы алгоритмов 8](#_Toc122463587)

[Глава 2. Реализация алгоритмов нахождения минимального остова 9](#_Toc122463588)

[Глава 2.1. Реализация алгоритма Прима 9](#_Toc122463589)

[**Список литературы** 10](#_Toc122463590)

# **Введение**

Одной из задач в области теории графов является задача о нахождении минимального остовного дерева. В данной работе будут разобраны популярные алгоритмы для решения этой проблемы, а конкретно алгоритмы Прима и Крускала (Краскала).

Для алгоритмов будет произведена их асимптотическая оценка, так же будет выбрана оптимальная структура для решения заданной проблемы о нахождении минимального остовного дерева.

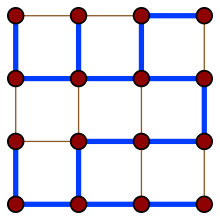
Задача по нахождению минимального остовного дерева может быть сформулирована по-разному, так, например, в терминах теории графов она определена как задача о нахождении минимального остовного дерева в графе, вершины которого представляют города, рёбра — это пары городов, между которыми можно проложить прямую дорогу, а вес ребра равен стоимости строительства соответствующей дороги.

1. Построение минимального остовного дерева
   1. Остовное дерево. Постановка задачи

Остовным деревом графа называется дерево, которое можно получить из него путём удаления некоторых рёбер. У графа может существовать несколько остовных деревьев, и чаще всего их достаточно много.

Для взвешенных графов существует понятие веса остовного дерева, которое определено как сумма весов всех рёбер, входящих в остовное дерево. Минимальное остовное дерево - остовное дерево с минимальным возможным весом.

Любое остовное дерево в графе с n вершинами содержит ровно n-1 ребро.  
Число остовных деревьев в полном графе на n вершинах равно — это утверждение называется формулой Кэли.



1. Пример остовного дерева: синими полужирными линиями выделены ребра остовного дерева, красными точками помечены вершины дерева.

Постановка задачи по поиску минимального остовного дерева выглядит так: Дан взвешенный неориентированный граф G с n вершинами и m рёбрами. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим возможным весом (т. е. суммой весов рёбер). Поддерево — это набор рёбер, соединяющих все вершины, причём из любой вершины можно добраться до любой другой ровно одним простым путём.

Для нахождения минимального остовного дерева графа существуют два основных алгоритма: алгоритм Прима и алгоритм Крускала.

* 1. Алгоритм Дейкстры

Алгоритм Прима есть реализация алгоритма Дейкстры, поэтому для начала разберем основополагающий алгоритм.

Задача определена таким образом: для заданного взвешенного графа *G*=(*V*, *E*) найти кратчайшие пути из заданной вершины s до всех остальных вершин. Веса всех рёбер неотрицательны.

Алгоритм для решения поставленной задачи реализован следующим образом: в ориентированном взвешенном графе *G*=(*V*, *E*), вес рёбер которых неотрицателен и определяется весовой функцией , алгоритм Дейкстры находит длины кратчайших путей из заданной вершины *s* до всех остальных.

В алгоритме поддерживается множество вершин *U*, для которых уже вычислены длины кратчайших путей до них из *s*. На каждой итерации основного цикла выбирается вершина *u*∉*U*, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина *u* добавляется в множество *U* и производится релаксация всех исходящих из неё рёбер.

Убедиться в представленном алгоритме позволяет следующая теорема: пусть *G*=(*V*, *E*) — ориентированный взвешенный граф, вес рёбер которого неотрицателен, *s* — стартовая вершина. Тогда после выполнения алгоритма Дейкстры для всех u, где — длина кратчайшего пути из вершины *s* в вершину *u*.

В реализации алгоритма присутствует функция выбора вершины с минимальным значением *d* и релаксация по всем рёбрам для данной вершины. Асимптотика работы зависит от реализации.

Пусть *n* — количество вершин в графе, *m* — количество рёбер в графе. Оценим сложность алгоритма для некоторых структур в таблице 1.

1. Оценка сложности следующих операций: поиск минимума, релаксация, а также оценка сложности общего времени работы для массива, двоичной кучи, фибоначчиевой кучи.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Поиск минимума | Релаксация | Общее |
| Массив |  |  |  |
| Двоичная куча |  |  |  |
| Фибоначчиева куча |  |  |  |

Наиболее привлекательными являются Фибоначчиевы кучи, которые позволяют производить операцию поиска минимума за , а второго — за *O*(1). Поэтому при использовании Фибоначчиевых куч время работы алгоритма Дейкстры составит , что является практически теоретическим минимумом для алгоритма поиска кратчайшего пути. Кстати говоря, эта оценка является оптимальной для алгоритмов, основанных на алгоритме Дейкстры, т. е. Фибоначчиевы кучи являются оптимальными с этой точки зрения.

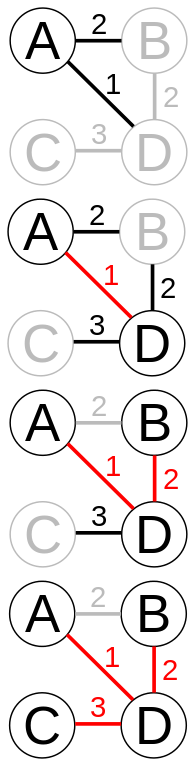
* 1. Алгоритм Прима

Немного об истории разработки алгоритма: этот алгоритм назван в честь американского математика Роберта Прима (Robert Prim), который открыл этот алгоритм в 1957 г. Впрочем, ещё в 1930 г. этот алгоритм был открыт чешским математиком Войтеком Ярником (Vojtěch Jarník). Кроме того, Эдгар Дейкстра (Edsger Dijkstra) в 1959 г. также изобрёл этот алгоритм, независимо от них.

Как упоминалось ранее, алгоритм Прима в идее и реализации очень похож на алгоритм Дейкстры. Как и в алгоритме Дейкстры, мы поддерживаем уже обработанную часть графа (минимального остовного дерева), и постепенно её расширяем за счёт ближайших вершин.

Утверждается, что если разделить вершины графа на два множества (обработанные и необработанные), первое из которых составляет связную часть минимального остовного дерева, то ребро минимальной длины, связывающее эти два множества гарантированно будет входить в минимальное остовное дерево.

Таким образом, для нахождения минимального остовного дерева начнём с произвольной вершины и будем постепенно добавлять ближайшие к уже имеющимся.



1. На рисунке красным цветом выделены рёбра, уже вошедшие в минимальный остов, а чёрным - текущие кандидаты, из которых выбирается ребро с минимальным весом.

В реализации алгоритма будем искать вес минимального остовного дерева. Для нахождения ближайшей вершины воспользуемся очередью с приоритетом (аналогично алгоритму Дейкстры), в которой будем хранить пары (расстояние от остова до вершины, номер вершины).

Оценим сложность работы алгоритма для некоторых структур в таблице 2. Производительность алгоритма Прима зависит от выбранной реализации приоритетной очереди, как и в алгоритме Дейкстры. Извлечение минимума выполняется V раз, релаксация — O(E) раз.

1. Оценка сложности следующих операций: поиск минимума, релаксация, а также оценка сложности общего времени работы для массива, двоичной кучи, фибоначчиевой кучи.

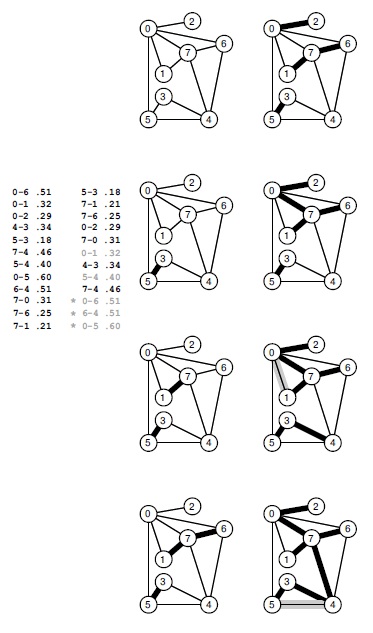
|  |  |
| --- | --- |
| Структура данных для приоритетной очереди | Асимптотика времени работы |
| Массив |  |
| Двоичная куча |  |
| Фибоначчиева куча |  |

* 1. Алгоритм Крускала

Алгоритм Крускала (или алгоритм Краскала) — еще один алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые описан Джозефом Крускалом в 1956 году.

Алгоритм Крускала объединяет вершины графа в несколько связных компонент, каждая из которых является деревом. На каждом шаге из всех ребер, соединяющих вершины из различных компонент связности, выбирается ребро с наименьшим весом. Для этого в алгоритме Крускала все ребра графа *G* перебираются по возрастанию веса. Для очередного ребра проверяется, не лежат ли концы ребра в разных компонентах связности, и, если это так, ребро добавляется, и компоненты объединяются.

Построение начинается с выбора кратчайшего ребра *е1* в *G*. На каждом последующем шаге будем выбирать из оставшихся ребро минимальной длины, не образующее цикла с ранее выбранными ребрами. Если имеется несколько таких ребер одинаковой длины, то можно выбрать любое из них. Процесс заканчивается построением остовного дерева.



1. Пример пошагового выполнения алгоритма Крускала.

Для реализации алгоритма Крускала необходимо уметь сортировать рёбра по возрастанию длины (для этого воспользуемся собственным типом данных) и проверять, соединяет ли ребро две различных компоненты связности. Реализовать представленный алгоритм проще всего с помощью СНМ (система непересекающихся отрезков), поэтому разберем ее устройство.

Система непересекающихся множеств — структура данных, которая позволяет администрировать множество элементов, разбитое на непересекающиеся подмножества. Изначально имеется несколько элементов, каждый из которых находится в отдельном (своём собственном) множестве. За одну операцию можно объединить два каких-либо множества, а также можно запросить, в каком множестве сейчас находится указанный элемент. Также вводится ещё одна операция — создание нового элемента, который помещается в отдельное множество.

Таким образом, базовый интерфейс данной структуры данных состоит всего из трёх операций:

* *make\_set(*x*)* - добавляет новый элемент x, помещая его в новое множество, состоящее из одного него.
* *union\_sets*(x, y) - объединяет два указанных множества (множество, в котором находится элемент x, и множество, в котором находится элемент y).
* *find\_set*(x) - возвращает, в каком множестве находится указанный элемент x. На самом деле при этом возвращается один из элементов множества (называемый представителем). Этот представитель выбирается в каждом множестве самой структурой данных (и может меняться с течением времени, а именно, после вызовов *union\_sets*.

Хранить структуру данных удобно в виде леса, то есть превратим СНМ в систему непересекающихся деревьев. Все элементы одного множества лежат в одном соответствующем дереве, представитель дерева — его корень, слияние множеств - объединение двух деревьев в одно.

Рассмотрим оценки операций для система непересекающихся отрезков. Количество вершин – V , количество ребер – E.

1. Оценка сложности некоторых операций при реализации алгоритма Крускала. - обратная функция Аккермана, которая не превосходит 4 во всех практических приложениях и которую можно принять за константу.

|  |  |
| --- | --- |
| Операции | Асимптотика времени работы операции |
| Сортировка *E* |  |
| Работа с СНМ |  |
| Общее время работы |  |

* 1. Различия в скорости работы алгоритмов

Оба алгоритма работают за , однако существуют константные различия в скорости их работы. На разреженных графах (количество рёбер примерно равно количеству вершин) быстрее работает алгоритм Крускала, а на насыщенных (количество рёбер примерно равно квадрату количеству вершин) - алгоритм Прима (при использовании матрицы смежности).

1. Реализация алгоритмов нахождения минимального остова
   1. Реализация алгоритма Прима

Для реализации алгоритма Прима были написаны следующие классы: *BinaryHeap*, *Edge*. *Edge* представляет собой класс ребра графа, который хранит информацию о конкретном ребре, и имеет поля *from*, *to*, *weight*, отвечающие за вершину, из которой идет ребро, вершину, в которую входит ребро, и вес ребра соответственно. *BinaryHeap* – класс бинарной кучи (структуры выбранной для алгоритма Прима). Поля класса включают в себя следующие переменные: *arr* (массив ребер), *len* (количество добавленных элементов), *size* (объем выделенной памяти). Основные методы класса, используемые для работы алгоритма: *ExtractMin* (в полученном алгоритме нам необходимо быстро извлекать минимальное ребро и этот метод используется в нем), *Swap* (меняет местами два элемента массива *arr*), *GetLeftChildIndex* и *GetRightChildIndex*, *Heapify* (просеивание вниз элемента кучи), *SiftUp* (просеивание вверх элемента кучи), *push* (добавление нового элемента в массив ребер).

Также для реализации алгоритма Прима, была построена функция *prima*, в которой и реализован данный алгоритм. Рассмотрим ее подробнее.

В самой функции определены переменные типа *BinaryHeap<Edge> edges* (куча, состоящая из рассматриваемых ребер)*, mst\_edges* (ребра минимального остова). Также нам понадобится массив *used* для сохранения информации о пройденных вершинах.

Начинаем рассматривать подграф с любой вершины (для удобства начнем с нулевой вершины). Затем из рёбер, инцидентных этой вершине, выбирается минимальное ребро, которое связало бы две абсолютно разные компоненты связности, одной из которых и является наш подграф, с помощью метода *ExtractMin*, если оно еще не рассматривалось. Таким образом, как только у нас появляется возможность добавить новую вершину в наш подграф, мы тут же включаем ее по минимальмально возможному весу. Далее будем добавлять в массив *edges* новые ребра для рассмотрения.

Продолжаем выполнять этот цикл до тех пор, пока не рассмотрим все возможные ребра. В результате функция выводит построенное минимальное остовное дерево и возвращет его вес. В листинге 1 приведен код основного цикла функции *prima*.

|  |
| --- |
| **while** (!edges.empty()) {  Edge edge = edges.ExtractMin();  **int** to = edge.to;  **int** from = edge.from;  **int** w = edge.weight;  **if** (used[to]) **continue**;  used[to] = 1;  mst\_weight += w;  mst\_edges.push(Edge(from, to, w));  **for** ( **int** i = 0; i < n ; i++ ) {  **if** ( graph[to][i] > 0 && !used[i] ) {  edges.push(Edge(to, i, graph[to][i]));  }  }  } |

1. Основной цикл *while* функции *prima*.
   1. Реализация алгоритма Крускала

Для реализации алгоритма Крускала необходимо разработать класс *DissjointSetUnion*, который представляет собой систему непересекающихся отрезков. Данный класс содержит в себе два атрибута-массива *parent* и *rank*. *Parent* хранит в себе информацию о множествах элементов, *rank* хранит в себе глубину вершин, что позволяет ускорить работу объединения множеств.

Рассмотрим метод *union\_sets* этого класса, поскольку это одна из важнейших функций для реализации алгоритма. Для начала находим в каких множествах находятся вершины, далее будем присоединять дерево с меньшим рангом к дереву с большим рангом.

|  |
| --- |
| **void** union\_sets(**int** a, **int** b) {  a = find\_set(a);  b = find\_set(b);  **if** (a != b) {  **if** (rank[a] < rank[b])  swap(a, b);  parent[b] = a;  **if** (rank[a] == rank[b])  rank[a]++;  }  } |

1. Метод класса *DissjointSetUnion union\_sets()*, реализующий объединение двух множеств.

Ход алгоритма. Вначале мы производим сортировку рёбер по неубыванию по их весам. Добавляем i-ое ребро в наш подграф только в том случае, если данное ребро соединяет две разные компоненты связности, одним из которых является наш подграф. То есть, на каждом шаге добавляется минимальное по весу ребро, один конец которого содержится в нашем подграфе, а другой - еще нет. Алгоритм завершит свою работу после того, как множество вершин нашего подграфа совпадет с множеством вершин исходного графа.

Разберем основное тело алгоритма, которое представлено в листинге 3. Метод *make*\_*set()* помещает каждую вершину в свое собственное дерево то есть, создает некоторое множество подграфов. Дальше проходимся по всем ребрам в отсортированном порядке и смотрим, принадлежат ли инцидентные вершины текущего ребра разным подграфам с помощью функции *find*\_*set()* или нет, если оба конца лежат в разных компонентах, то объединяем два разных подграфа в один с помощью функции *union\_sets().*

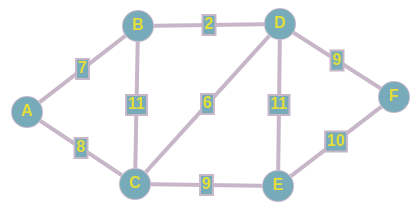
|  |
| --- |
| **for** (**int** i = 0; i < n; i++)  dsu.make\_set(i);  sort(edges.begin(), edges.end());  **for** (Edge e : edges) {  **if** (dsu.find\_set(e.from) != dsu.find\_set(e.to)) {  kruskala\_weight += e.weight;  result.push(e);  dsu.union\_sets(e.from, e.to);  }  } |

1. Реализация алгоритма Крускала.
   1. Тестирование алгоритмов Прима и Крускала

Рассмотрим некоторый граф (его визуализация представлена на рисунке 4). Выпишем все его ребра в отсортированном порядке в таблице 4.

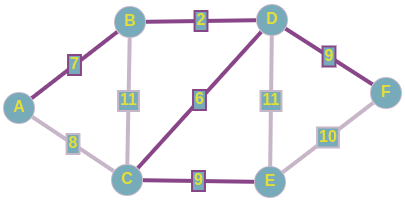
1. Ребра и веса выбранного графа.

|  |  |
| --- | --- |
| Ребра | Вес ребер |
| DB | 2 |
| DC | 6 |
| AB | 7 |
| AC | 8 |
| CE | 9 |
| DF | 9 |
| FE | 10 |
| BC | 11 |
| DE | 11 |



1. Рассматриваемый в ходе теста граф.

Начнем по списку добавлять эти ребра в наш остов: DB, DC, BA, DF, CE. Полученный остов представлен на рисунке 5. Вес найденного минимального остова равен 33.



1. Исходный граф и выделенное на нем минимальное остовное дерево.

Проверим работу написанных алгоритмов на данном графе.

1. Результат работы алгоритма Прима на данном графе.

|  |  |  |
| --- | --- | --- |
| Исток | Сток | Вес |
| 3 | 2 | 6 |
| 1 | 3 | 2 |
| 0 | 1 | 7 |
| 2 | 4 | 9 |
| 3 | 5 | 9 |

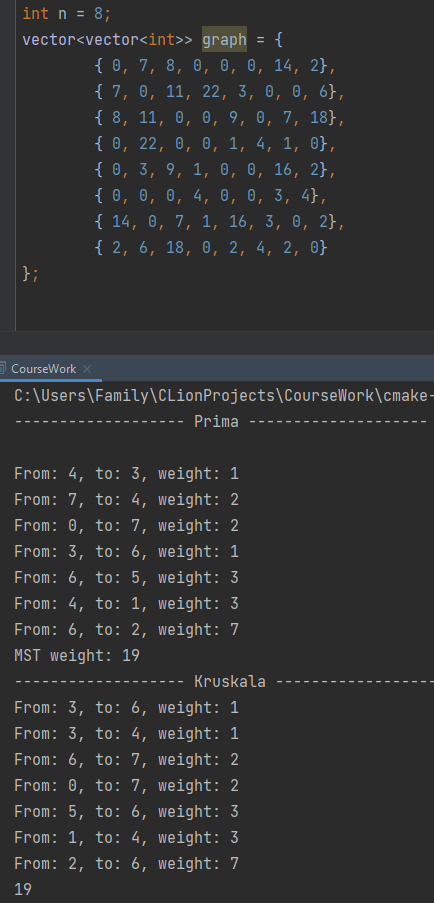
1. Результат работы алгоритма Крускала на данном графе.

|  |  |  |
| --- | --- | --- |
| Исток | Сток | Вес |
| 1 | 3 | 2 |
| 2 | 3 | 6 |
| 0 | 1 | 7 |
| 2 | 4 | 9 |
| 3 | 5 | 9 |

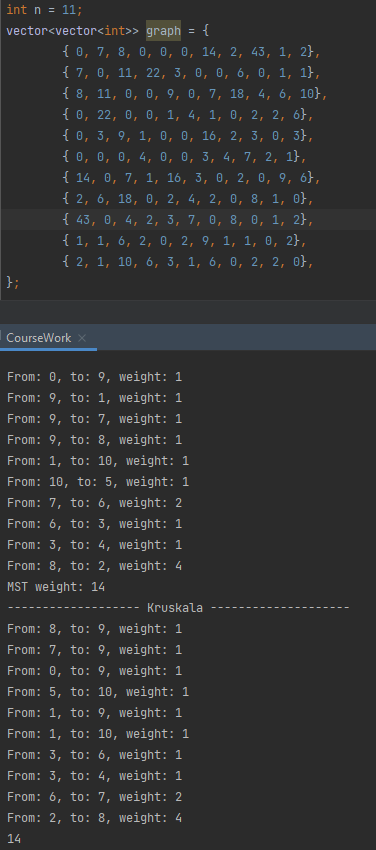
В результате работы алгоритмов Прима и Крускала для данного нам графа, мы получили необходимый минимальный остов и его вес.

* 1. Результат запуска кода

Приведем примеры запуска кода для разных графов.



1. Результат работы первого теста.



1. Результат работы второго теста.

# **Заключение**

В ходе работы были написаны два наиболее популярных алгоритма по поиску минимального остовного дерева. Также стоит упомянуть о некоторых других алгоритмах решения этой задачи: алгоритм Борувки и алгоритм обратного удаления.

Алгоритмы Прима и Крускала, засчет различий в алгоритмах подходят для определенных данных. Так, в зависимости от того, плотный граф или разреженный один из алгоритмов может подойти лучше по сравнению с другим в скорости работы.

# **Список литературы**

* Харари Фрэнк Теория графов: Пер. с англ./ Предисл. В. П. Козырева; Под ред. Г.П.Гаврилова. Изд. 4-е. — М.: Книжный дом "ЛИБРОКОМ", 2009. — 296 с. — ISBN 978-5-397-00622-4.
* Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы: Построение и анализ. «Вильямс» [2005] – 1328 с. - ISBN 978-5-8459-2016-4.
* Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани; Пер. с англ. под ред. А. Шеня. –– М.: МЦНМО, 2014. –– 320 с.   
  ISBN 978-5-4439-0236-4
* Уильям Топп, Уильям Форд. Структуры данных в C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 1999. – 816 с.: ил.