МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования

**«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»**
(ННГУ)

**Институт информационных технологий, математики и механики**

**Центр прикладных информационных технологий**

ЛАБОРАТОРНАЯ
на тему:

# «Задача о минимальном остовном дереве»

**Выполнил:**
студент группы 3821Б1ФИ2
_______________ Казанцев Е. А.

**Проверил:**
Перподаватель
_______________Уткин Г. В.

Нижний Новгород
2023 г.

# Содержание

**1 Введение** **2**

**2 Цель работы** **2**

**3 Постановка задач** **2**

**4 Руководство программиста** **3**

**5 Описание алгоритмов** **4**

**6 Эксперементы** **9**

**7 Результаты экспериментов** **14**

**8 Вывод** **14**

**Список литературы** **15**

# 1 Введение

В этой работе реализованы два алгоритма поиска минимального остовного дерева в полном, ориентированном, взвешенном графе. Алгоритм Прима на 2-куче и алгоритм Краскалла на массиве.
Минимальным остовным деревом (МОД) связного взвешенного графа называется его связный подграф, состоящий из всех вершин исходного дерева и некоторых его ребер, причем сумма весов ребер минимально возможная. Если исходный граф несвязен, то описываемую ниже процедуру можно применять поочередно к каждой его компоненте связности, получая тем самым минимальные остовные деревья для этих компонент.

# 2 Цель работы

Целью данной работы является сравнение двух алгоритмов поиска минимального остовного дерева во взвешенном неориентированном графе, реализованных на различных структурах данных. Сравнение асимптотической сложности и замеры времени работы. Для сравнения будут проведены эксперименты с различным количеством вершин в графе и различными весами, задаваемыми вручную, в функции генерации графов.

# 3 Постановка задач

## Написать классы бинарной кучи и графов.

*В бинарной куче длжны быть реализованы:*

1. Метод всплытия элемента
2. Погружениz элемента
3. Добавление элемента
4. Удаление элемента
5. Возвращение значения элемента (weight)

*Класс графа и дополнительные структуры:*

1. Вспомогательная структура Edge
2. Вспомогательная структура UnionFind

3. Добавление ребер в граф
4. Генератор случайного, связного, полного графа
5. Алгоритм поиска минимального остовного дерева Prima
6. Алгоритм поиска минимального остовного дерева Kruskal

# 4 Руководство программиста

## Описание структуры программы

Программа состоит из одного решения.

В решении min_spanning_tree определено 4 модуля main.cpp, Graph.h, UnionFind.h, 2_heap.h, Timer.h

- В модуле main.cpp определена стандартная функция int main(), где идет работа с остальными модулями.
- В модуле Graph.h определен класс Graph, а также объявлены все его методы, определения которых, вынесены в отдельный файл Graph.cpp.
- В модуле 2_heap.h определен класс 2_heap, и также, как в Graph все его методы вынесены в отдельный файл 2_heap.cpp.
- В модуле Timer.h определен класс Timer, в котором реализован простой секундомер, для замеров времени работы алгоритмов.

## Описание структур данных

- list: Вектор векторов, используется для представления списка смежности графа.
- Edge: Структура, представляющая ребро графа, с полями from (начальная вершина), to (конечная вершина) и weight (вес ребра).

**Методы:**

- Edge: Структура, представляющая ребро графа, с полями from (начальная вершина), to (конечная вершина) и weight (вес ребра).
- Graph::addEdge(int from, int to, int weight): Добавляет ребро в граф.
- Graph::printGraph(): Выводит граф в консоль.
- Graph::PrimMST(): Реализация алгоритма Прима для поиска минимального остовного дерева.

- Graph::Kruskal(): Реализация алгоритма Краскала для поиска минимального остовного дерева.

- Graph::generateRandGraph(int _size, int minWeight, int maxWeight): Генерирует случайный граф.

# 5 Описание алгоритмов

## Алгоритм Прима

На вход алгоритма подаётся связный неориентированный граф. Для каждого ребра задаётся его стоимость.

Сначала берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет; из этих рёбер выбирается ребро наименьшей стоимости. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости.

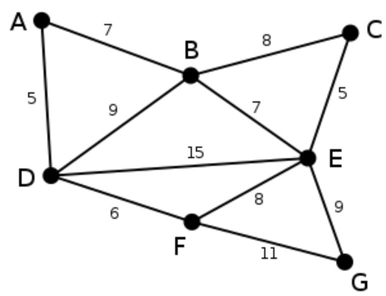
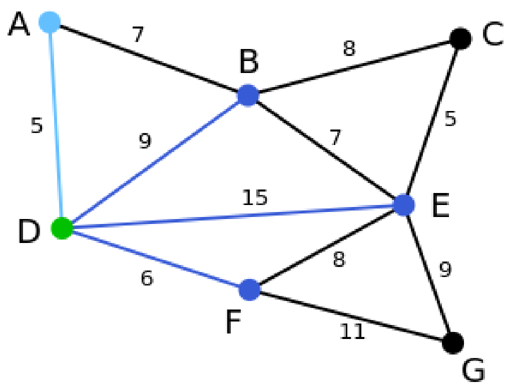
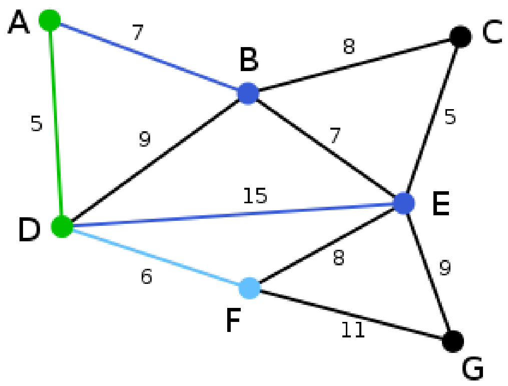



Рис. 1: Неориентированный взвешенный граф

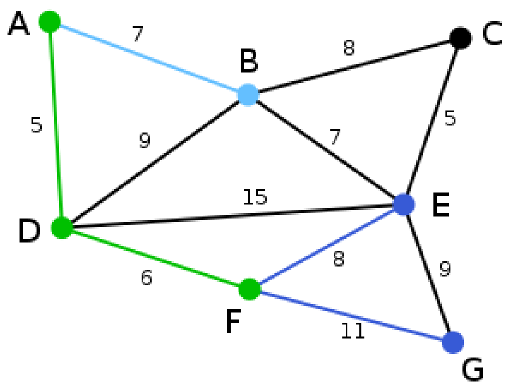
В качестве начальной произвольно выбирается вершина D. Каждая из вершин A, B, E и F соединена с D единственным ребром. Вершина A — ближайшая к D, и выбирается как вторая вершина вместе с ребром AD.



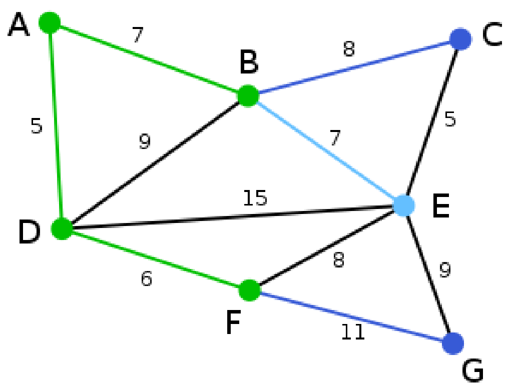
Следующая вершина — ближайшая к любой из выбранных вершин D или A. B удалена от D на 9 от A — на 7. Расстояние до E равно 15, а до F — 6. F является ближайшей вершиной, поэтому она включается в дерево F вместе с ребром DF.



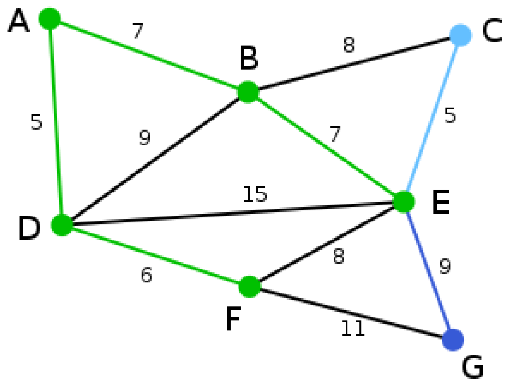

Аналогичным образом выбирается вершина B, удаленная от A на 7.



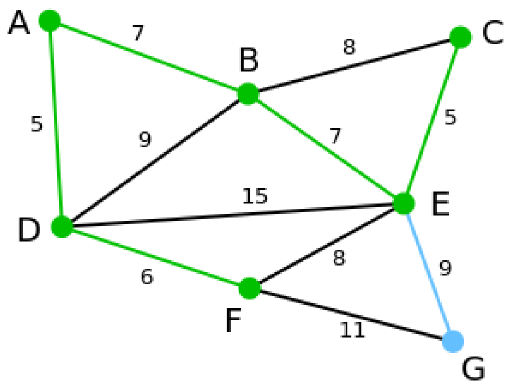

В этом случае есть возможность выбрать либо C, либо E, либо G. C удалена от B на 8, E удалена от B на 7, а G удалена от F на 11. E — ближайшая вершина, поэтому выбирается E и ребро BE.



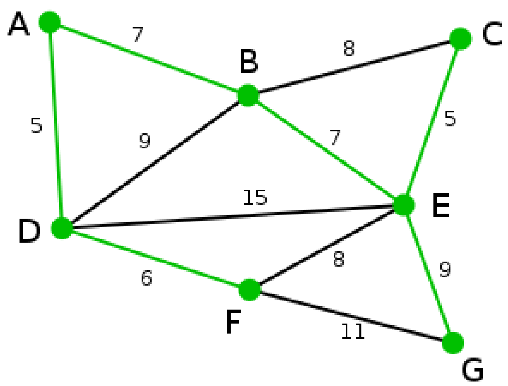

Здесь доступны только вершины C и G. Расстояние от E до C равно 5, а до G — 9. Выбирается вершина C и ребро EC.




Единственная оставшаяся вершина — G. Расстояние от F до неё равно 11, от E — 9. E ближе, поэтому выбирается вершина G и ребро EG.




Выбраны все вершины, минимальное остовное дерево построено (выделено зелёным). В этом случае его вес равен 39.



## Псевдокод

**Algorithm 1:** PrimMST($G$, $w$, $r$)

1 **for** *каждой вершины* $u \in V[G]$ **do**
2     $key[u] \leftarrow \infty$;
3     $\pi[u] \leftarrow$ NIL;
4 $key[r] \leftarrow 0$;
5 $Q \leftarrow V[G]$;
6 **while** $Q \neq \emptyset$ **do**
7     $u \leftarrow \mathrm{pop}(Q)$;
8     **for** *каждой вершины* $v \in$ *Adj[u]* **do**
9         **if** $v \in Q$ *и* $w(u,v) < key[v]$ **then**
10             $\pi[v] \leftarrow u$;
11             $key[v] \leftarrow w(u,v)$;

В нашей реализации, вместо приоритетной очереди будет использоваться двоичная, минимальная куча.

Производительность алгоритма Прима зависит от выбранной реализации оче- реди с приоритетами Q.В нашей реализации, вместо приоритетной очереди будет использоваться двоичная, минимальная куча minHeap. Тело цикла while выполняется |V | раз, а поскольку каждая операция pop занимает время O (lg V ), об- щее время всех вызовов процедур pop составляет O (V lg V ). Цикл for в строках 8–11 выполняется всего O (E) раз, поскольку сумма длин всех спис- ков смежности равна 2 |E|. Внутри цикла for проверка на принадлежность minHeap в строке 9 может быть реализована за постоянное время, если воспользоваться для каждой вершины битом, указывающим, находится ли она в minHeap, и обновлять этот бит при удалении вершины из minHeap. Присвоение в строке 11 неявно вклю- чает операцию elem_down над пирамидой. Время выполнения этой опера- ции — O (lg V ). Таким образом, общее время работы алгоритма Прима составляет O (V lg V + E lg V ) = O (E lg V ), что асимптотически совпадает со временем ра- боты алгоритма Крускала, который будет описан позже.

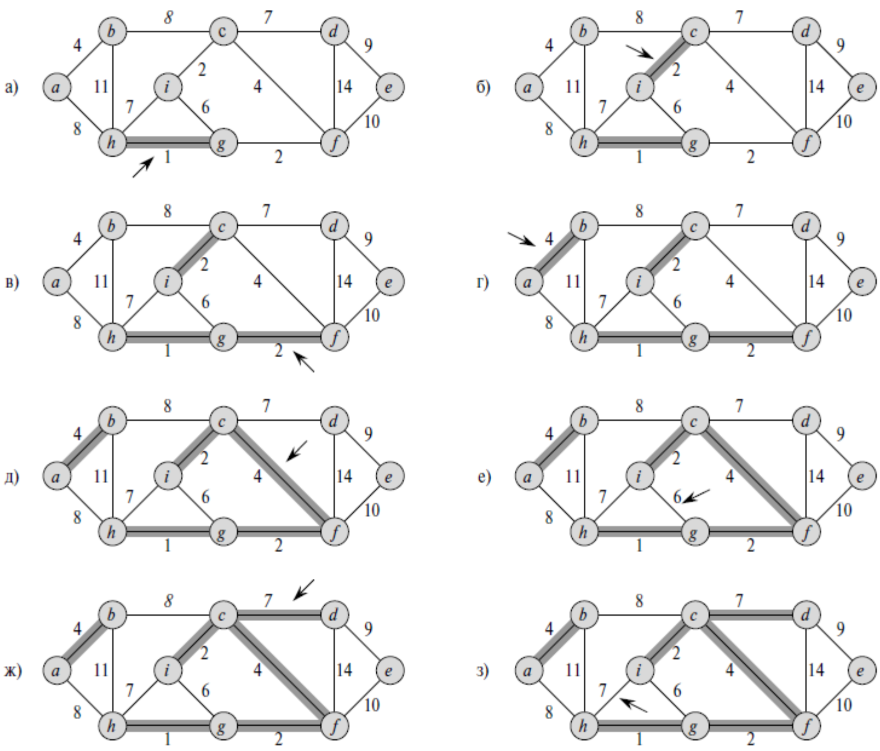
## Алгоритм Краскала

Алгоритм Крускала непосредственно основан на обобщенном алгоритме по- иска минимального остовного дерева. Он находит безопасное ребро для добавления в растущий лес путем поиска ребра (u, v) с ми- нимальным весом среди всех ребер, соединяющих два дерева в лесу. Обозначим два дерева, соединяемые ребром (u, v), как C1 и C2. Поскольку (u, v) должно быть легким ребром, соединяющим C1 с некоторым другим деревом, (u, v) — безопасное для C1 ребро. Алгоритм Крускала является жадным, поскольку на каждом шаге он добавляет к лесу ребро с минимально возможным весом. Наша реализация алгоритма Крускала напоминает алгоритм для вычисления связных компонентов. Она использует структуру для представ- ления непересекающихся множеств. Каждое множество содержит вершины де- рева в текущем лесу. Операция find_root(u) возвращает представительный эле- мент множества, содержащего u. Таким образом, мы можем определить, принадлежат ли две вершины u и v одному и тому же дереву, проверяя равенство find_root(u) и find_root(u). Объединение деревьев выполняется при помощи про- цедуры union_sets.

### Псевдокод

**Algorithm 2:** Kruskal($G$, $w$)

```
1 A ← ∅;
2 for каждой вершины v ∈ V[G] do
3     MAKE_SET(v);
4 Сортируем рёбра из E в неубывающем порядке их весов w;
5 for каждого (u, v) ∈ E (в порядке возрастания веса) do
6     if FIND_SET(u) ≠ FIND_SET(v) then
7         A ← A ∪ {(u, v)};
8         UNION(u, v);
9 return A;
```




Время работы алгоритма Крускала для графа G = (V,E) зависит от реализа- ции структуры данных для непересекающихся множеств. Инициализация множества A в строке 1 занимает вермя O(1), а время, необходимое для сортировки множества в строке 4, равно $\mathcal{O}(E\log(E))$ (стоимость |V| операций MAKE_SET в цикле for в строках 2–3 мы учтем чуть позже). Цикл for в строках 5–8 выполняет $\mathcal{O}((V+E)\alpha(V))$ где $\alpha-$ очень медленно растущая функция. Поскольку мы пред- полагаем, что $G$ — связный граф, справедливо соотношение ($|E| \geq |V|-1$), так что операции над непересекающимися множествами требуют $\mathcal{O}(E\alpha(V))$ времени. Кроме того, поскольку $\alpha(|V|) = \mathcal{O}(\lg(V)) = \mathcal{O}(\lg(E))$, общее время работы алгоритма Крускала равно $\mathcal{O}(E\lg(E))$. Заметим, что $|E| < |V|^2$, поэтому $\lg|E| = \mathcal{O}(\lg(V))$ и время работы алгоритма Крускала можно записать как $\mathcal{O}(E\lg(V))$.

# 6 Эксперементы

## Замеры

*Проведем эксперементы на основе седующих данных:*

1. Вершины от $n = 2$ до $n = 10^4 + 1$
2. Количество ребер $m = \frac{n^2}{10}$, $\quad m \approx \frac{n^2}{10}$
3. Веса ребер от 1 до $10^6$
4. Шаг - 100 вершин

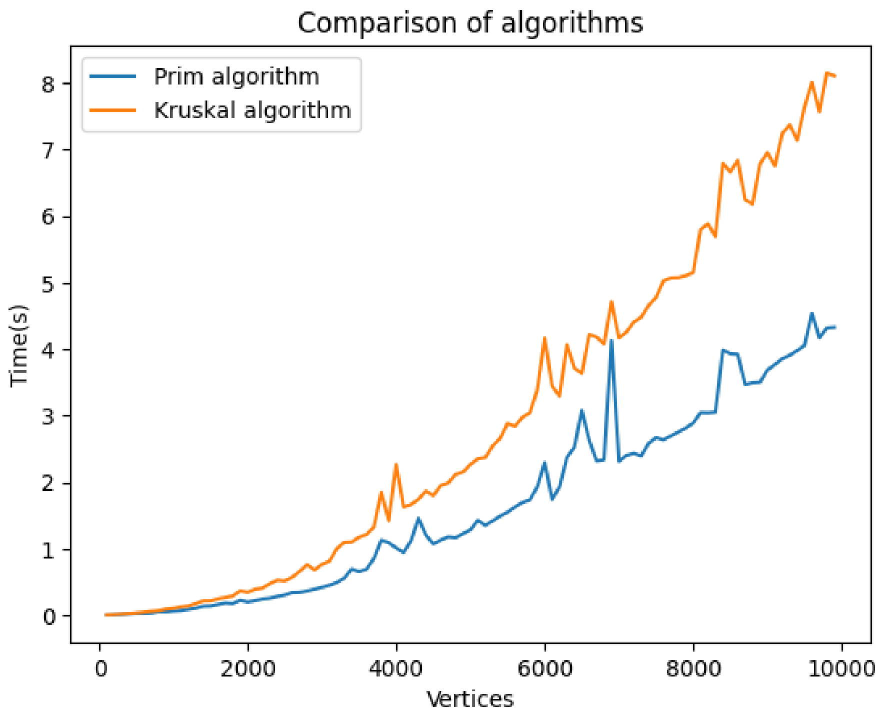



Рис. 2: Time

Среднее время отклонения работы двух алгоритмов в этих замерах составило $1.25060789c$ которое рассчитывалось по формуле:

$$mean\_time = \left| \frac{T(Prima) - T(Kruskal)}{count\_iter} \right|$$

Где $T(Prima)$ -время работы алгоритма Прима, $T(Kruskal)$ - время работы алгоритма Краскала, $count\_iter$ - количество итераций.

Как видно на графке, время работы алгоритмов расходится с увеличением вершин. С увеличением вершин до $m \approx n^2$, средняя разница во времмени работы заметно увеличилась и составила $6.167380556c$ (см. Рис. 3).

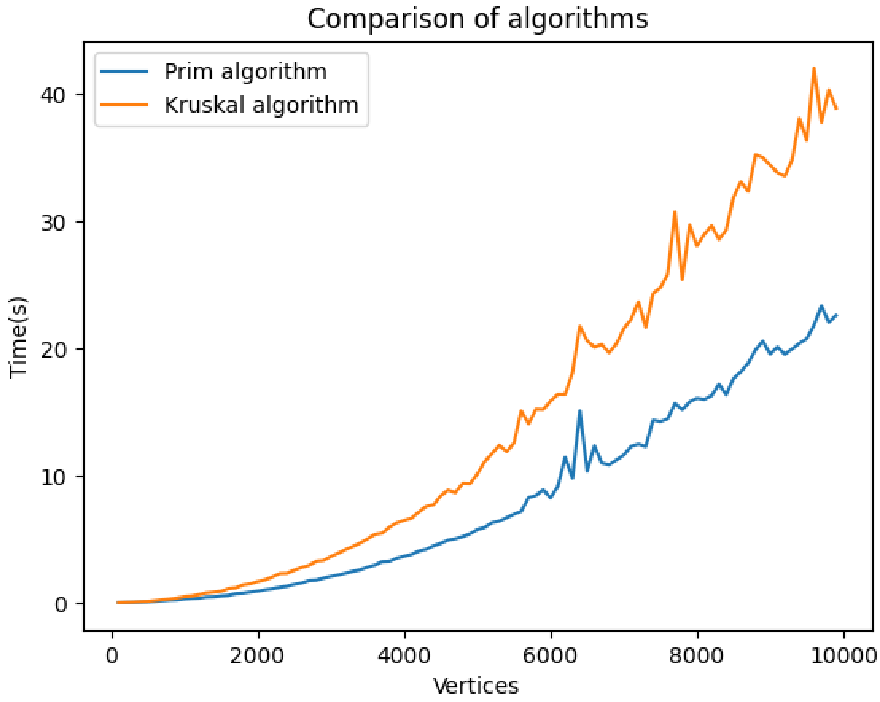



Рис. 3: Time

*Данные второго замера*

1. Вершины от $n = 101$ до $n = 10^4 + 1$
2. Количество ребер $m = 100n$ (см. Рис. 4) , $m = 1000n$ (см. Рис. 5)
3. Веса ребер от 1 до $10^6$
4. Шаг - 100 вершин

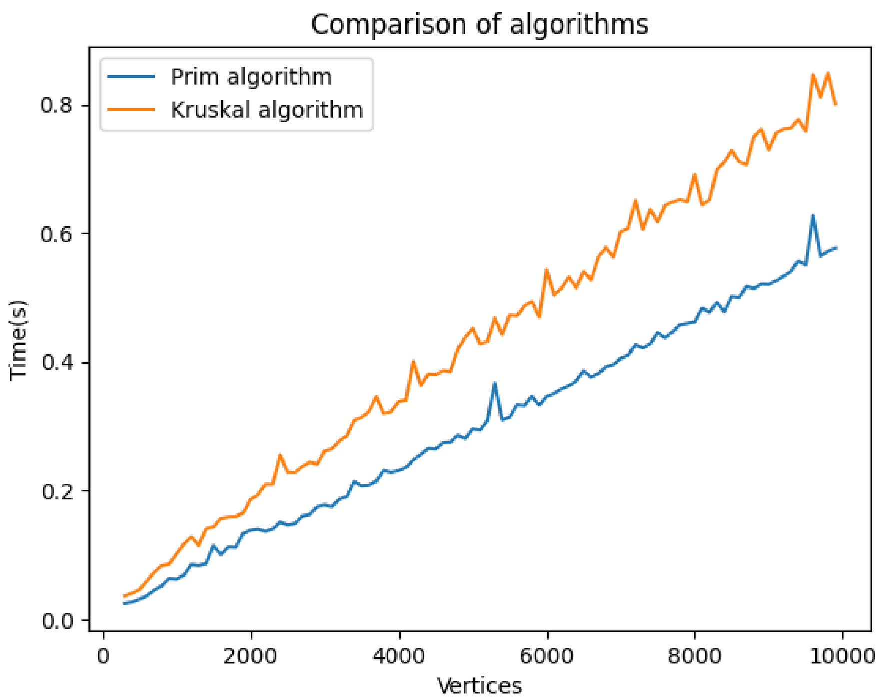



Рис. 4: Time

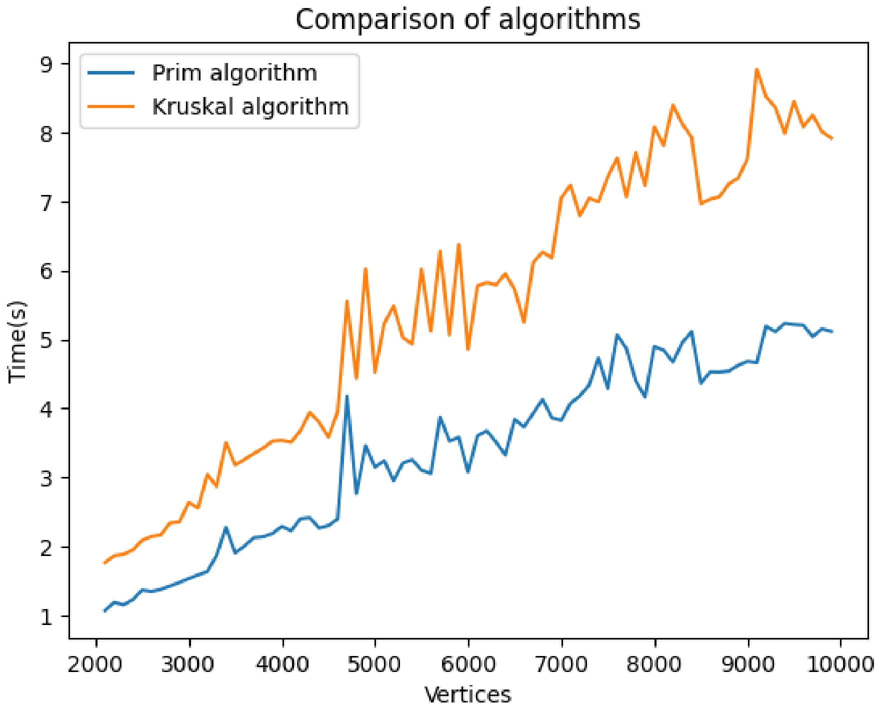



Рис. 5: Time

Как видно из графиков, при количестве ребер равным $m = 100n$, разница во времени работы достаточно мала и не превышает секунды $0.1292988c$. При $m = 1000n$ среднее отклонение $1.6487672c$.

*Данные третьего замера*

1. Вершины от $n = 101$ до $n = 10^4 + 1$
2. Количество ребер от $m = 2000$ до $10^7$ (см. Рис. 6)
3. Веса ребер от 1 до $10^6$
4. Шаг - 100000 вершин

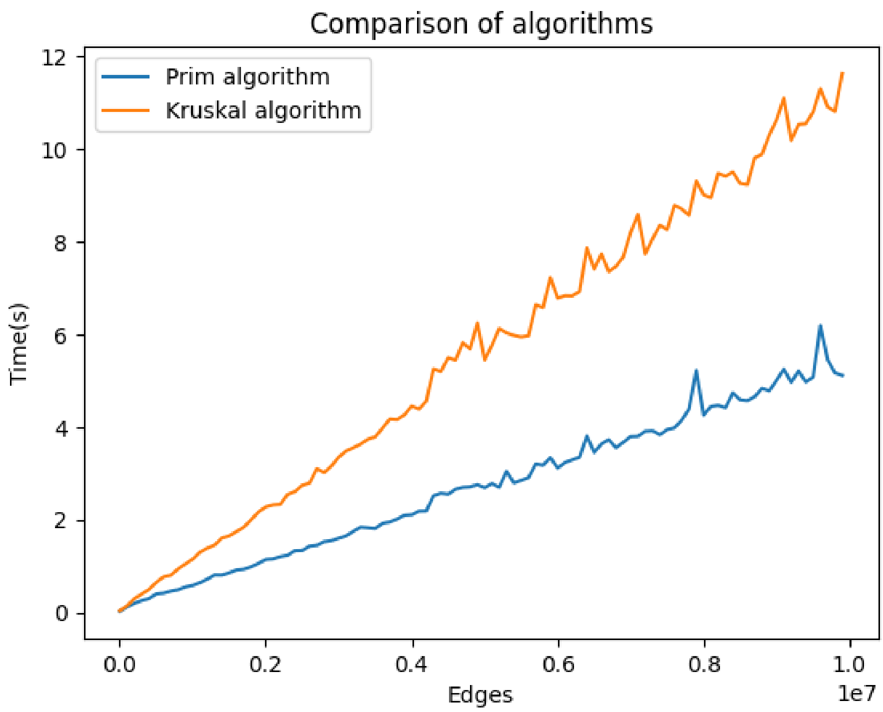



Рис. 6: Time

Среднее отклонение составило 2.90050744$c$, максимальное 6.51433$c$.

*Данные последнего замера*

1. Вершины $n = 10^4 + 1$
2. Количество ребер $m = n^2$ и $m = 1000n$
3. Веса ребер от 1 до 200
4. Шаг веса - 1

Удалось провести только частичный замер на 55 шагов, при $m = 1000n$ (см. Рис. 7) так как эксперемент продолжался довольно долгое время.

Дело в том, что генерация подобных графов, занимает намного больше времени, чем алгоритмы которые исследуются в данной лабораторной работе.

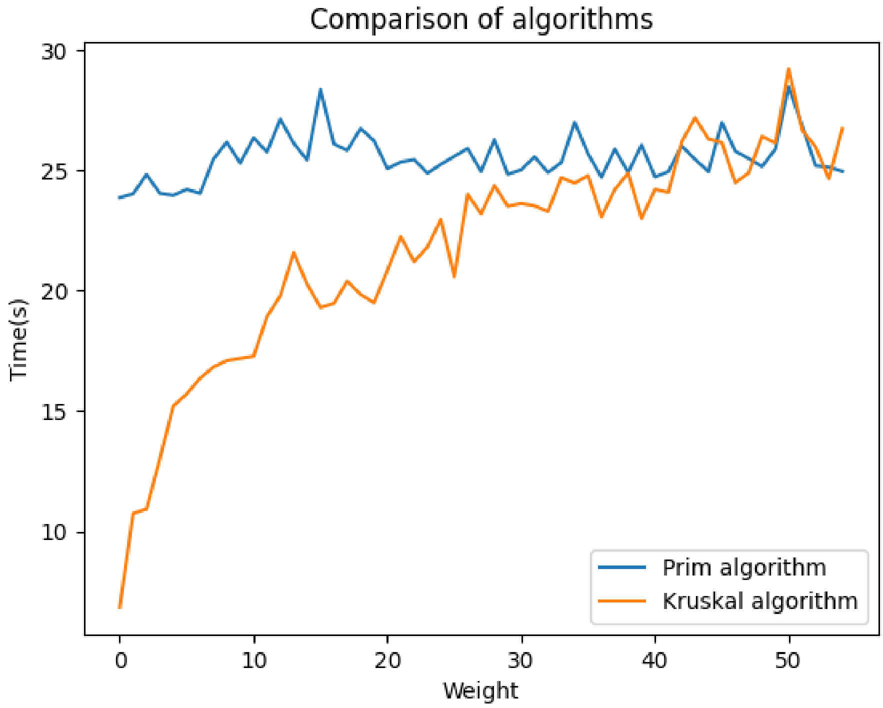



Рис. 7: Time

Как видно из замеров, впервые, вперед, вырвался алгоритм Краскала на начальных итерациях. После, время их работы выровнялось.

# 7 Результаты экспериментов

В результате экспериментов было установлено, что алгоритм Прима проявляет свою преимущественную эффективность на полных графах, особенно на графах с большим количеством вершин. Заметное ускорение работы алгоритма Прима начинается при количестве вершин, равном или превышающем 10000.

Для более общих графов, не являющихся полными, алгоритм Краскала может быть более эффективным в плане времени выполнения лишь на первых итеррациях и при среднем количестве вершин.

# 8 Вывод

Исходя из результатов экспериментов, можно сделать вывод, что выбор алгоритма для нахождения минимального остовного дерева зависит от конкретных характеристик задачи и графа. Если граф полный и содержит много вершин (более 10 000), то алгоритм Прима предпочтителен из-за своей более высокой скорости выполнения. В других случаях, алгоритм Краскала может оказаться более подходящим выбором.

Рекомендуется выбирать алгоритм нахождения минимального остовного дерева в зависимости от конкретных требований задачи и характеристик графа.

# Список литературы

[1] Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. // Алгоритмы: построение и анализ // 2-е издание.

[2] Майкл Солтис. // Введение в анализ алгоритмов. // Год издания: 2019.

[3] Алгоритм Краскала `https://ru.wikipedia.org/wiki/`

[4] Бинарные кучи `https://habr.com/ru/articles/112222/`