

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по курсовой работе
по дисциплине «Программирование»
Тема: Реализация и анализ работы
алгоритма поиска минимального
остовного дерева.

Студентка гр. 0324

Сотина Е.А.

Преподаватель

Глущенко А.Г.

Санкт-Петербург

2021

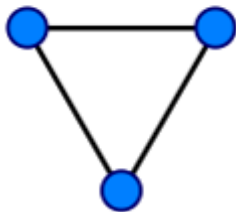
Цель работы.

Реализация алгоритма поиска минимального остовного дерева предложенным алгоритмом. Выбор самой быстрой реализации алгоритма поиска минимального остовного дерева. Привести аргументы за и против выбранной реализации.

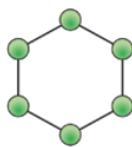
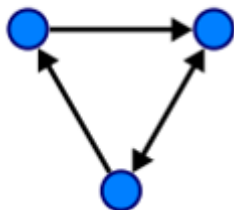
Основные теоретические положения.

Граф — абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин. Например, за множество вершин можно взять множество аэропортов, обслуживаемых некоторой авиакомпанией, а за множество рёбер взять регулярные рейсы этой авиакомпании между городами.

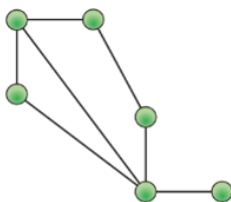
Граф



Ориентированный граф



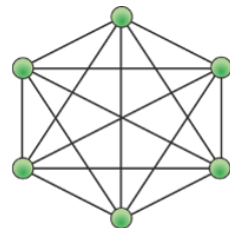
Кольцо



Связный граф



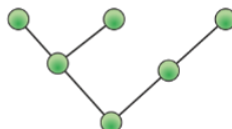
Звезда



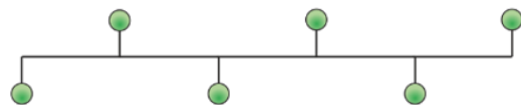
Полный граф



Линия



Дерево

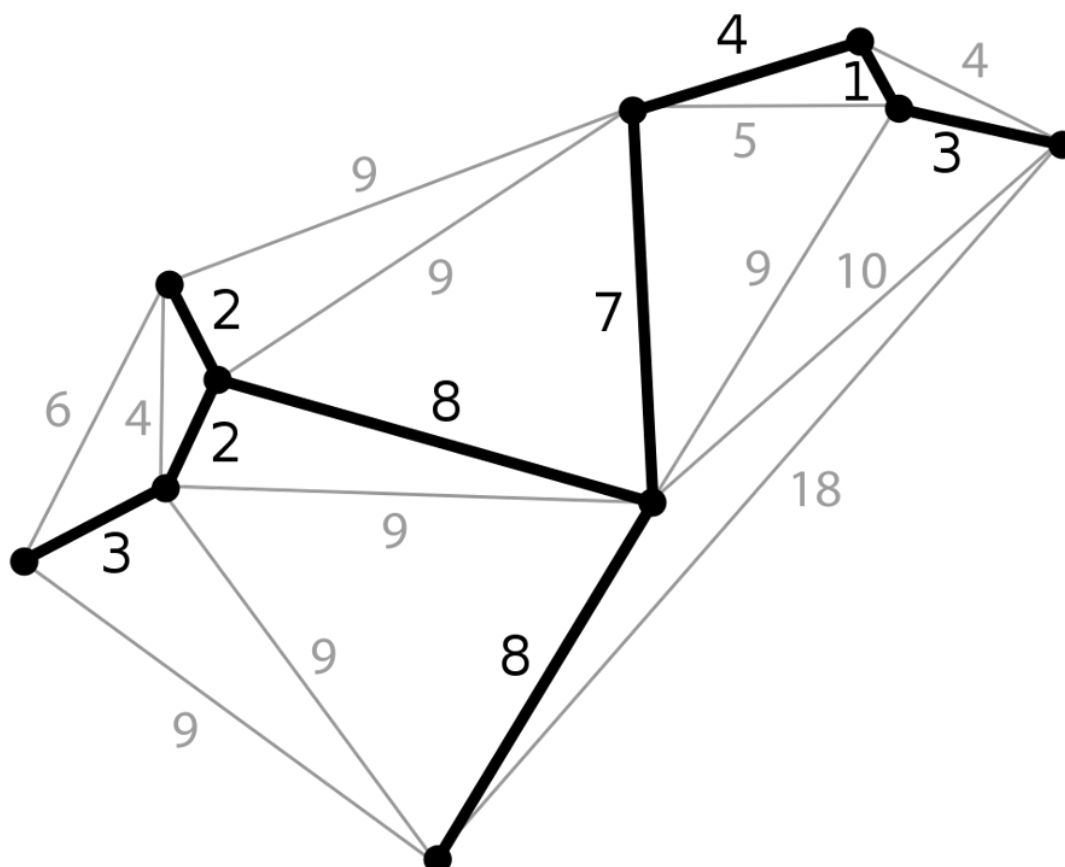


Автобус

Входные и выходные данные:

INPUT	OUTPUT
$\begin{pmatrix} 0 \end{pmatrix}$	$\begin{pmatrix} 0 \end{pmatrix}$
$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	Graph not connected
$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	Graph not connected
$\begin{pmatrix} 0 & 4 & 5 & 6 \\ 4 & 0 & 8 & 5 \\ 5 & 8 & 0 & 3 \\ 6 & 5 & 3 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 5 & 0 \\ 4 & 0 & 0 & 0 \\ 5 & 0 & 0 & 3 \\ 0 & 0 & 3 & 0 \end{pmatrix}$

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



Задача о нахождении минимального остовного дерева часто встречается в подобной постановке: допустим, есть n городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города). Разрешается строить дороги между заданными парами городов и известна стоимость строительства каждой такой дороги.

Требуется решить, какие именно дороги нужно строить, чтобы минимизировать общую стоимость строительства.

Эта задача может быть сформулирована в терминах теории графов 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.

Существует несколько алгоритмов для нахождения минимального остовного дерева. Наиболее известные из них:

- 1)Алгоритм Прима
- 2)Алгоритм Краскала
- 3)Алгоритм Борувки

Алгоритм Краскала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Также алгоритм используется для нахождения некоторых приближений для задачи Штейнера.

В алгоритме Краскала весь единый список ребер упорядочивается по возрастанию весов ребра. Далее ребра перебираются от ребер с меньшим весом к большему, и очередное ребро добавляется к каркасу, если оно не образует цикла с ранее выбранными ребрами. В частности, первым всегда выбирается одно из ребер минимального веса в графе.

В самом начале, когда ни одно ребро графа не выбрано, каждая вершина является отдельной компонентой связности.

По мере добавления новых ребер компоненты связности будут объединяться, пока не получится одна общая компонента связности.

Пронумеруем все компоненты связности и для каждой вершины будем хранить номер ее компоненты связности, таким образом, в самом начале для каждой вершины номер ее компоненты связности будет равен номеру самой вершины, а в конце у всех вершин будут одинаковые номера компоненты связности, которой они принадлежал.

Затем, пока это возможно, проводится следующая операция: из всех ребер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких ребер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество ребер, является его остовным деревом минимального веса.

При рассмотрении очередного ребра посмотрим номера компонент связности, соответствующих концам этого ребра. Если эти номера совпадают, то ребро соединяет две вершины, уже лежащие в одной компоненте связности, поэтому добавление этого ребра образует цикл.

Если же ребро соединяет две разные компоненты связности, например, с номерами a и b , то ребро добавляется к части основного дерева, а эти две компоненты связности объединяются вместе. Для этого можно, например, всем вершинам, которые раньше находились в компоненте b изменить номер компоненты на a .

Алгоритм Прима - это алгоритм минимального остовного дерева, что принимает граф в качестве входных данных и находит подмножество ребер этого графа, который формирует дерево, включающее в себя каждую вершину, а также

имеет минимальную сумму весов среди всех деревьев, которые могут быть сформированы из графа.

Шаги для реализации алгоритма Прима следующие:

1. Инициализируйте минимальное остовное дерево с произвольно выбранной вершиной.

2. Найдите все ребра, которые соединяют дерево с новыми вершинами, найдите минимум и добавьте его в дерево.

3. Продолжайте повторять шаг 2, пока не получите минимальное остовное дерево.

Постановка задачи.

Необходимо реализовать программу, которая выполняет следующие действия:

1) Реализация алгоритма (Прима или Краскала) поиска минимального остовного дерева для графов с большим числом вершин и количеством рёбер, не менее $3 \cdot N$, где N должно быть более 10000. Генерация графа случайна. Если была выбрана линейная структура для реализации, то реализовать алгоритм с использованием вектора.

2) Определение скорости сортировки графа. Отсортируйте граф различными сортировками в том числе встроенной сортировкой вектора. Определите самую быструю сортировку.

3) Проверка на связность графа и поиск минимального остовного дерева для небольших графов. Пользователь может построить граф двумя способами: ввести граф вручную, считать граф с файла.

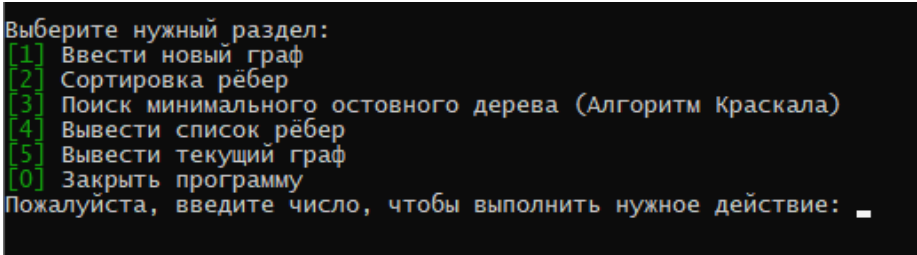
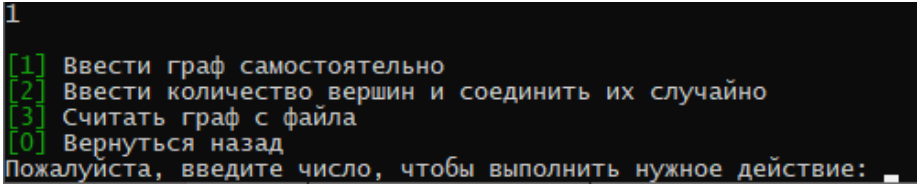
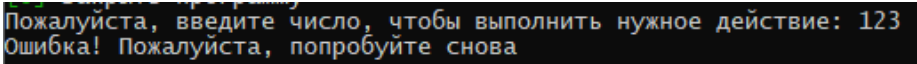
Произвести сравнительный анализ алгоритма Прима и Краскала. Определить лучший алгоритм. Выделить плюсы и минусы каждого алгоритма.

Студент сам определяет способ задания графа. Инструкция задания графа должна быть предоставлена пользователю. Корректность работы алгоритма производится на заранее заготовленных вариантах, где граф строится по матрице смежности.

Должна присутствовать возможность запуска каждого пункта многократно, если есть возможность (если в дереве нет элементов, то нельзя ничего удалить и об этом нужно сообщить пользователю).

Выполнение работы.

Код программы представлен в приложении А.

Ввод пользователем и обработка данных	Работа алгоритма и вывод на экран
Меню	
При запуске программы перед пользователем появляется окно с главным меню, где он может перейти к интересующему его действию	<p>Главное меню:</p>  <p>Подменю:</p>  <p>Проверка неправильного ввода:</p>  <p>Примеры ввода, вызывающие сообщение об ошибке: 123, 7, a, a1, 2b</p>

Ввод графа пользователем	
Пользователь имеет возможность создать граф самостоятельно, выбрав соответствующее действие.	<p>Сначала пользователю требуется ввести количество вершин:</p> <pre>[1] Ввести граф самостоятельно [2] Ввести количество вершин и соединить их случайно [3] Считать граф с файла [0] Вернуться назад Пожалуйста, введите число, чтобы выполнить нужное действие: 1 Введите количество вершин:</pre> <p>Проверка неправильного ввода:</p> <pre>Введите количество вершин: fd Ошибка! Количество вершин должно быть числом больше нуля.</pre> <p>Примеры ввода, вызывающие сообщение об ошибке: -123, а, а1</p> <p>Если число было всё-таки введено, но при этом присутствуют лишние символы, число считывается, а лишняя информация игнорируется:</p> <pre>[0] Вернуться назад Пожалуйста, введите число, чтобы выполнить нужное действие: 1 Введите количество вершин: 2b323673ddhsgj 0 0 0 0</pre> <p>Далее пользователю требуется вводить номера вершин (нумерация начинается с 1), между которыми нужно построить ребро:</p> <pre>Введите две вершины, которые ребро соединяет (например, 1 2):</pre> <p>При неправильном вводе, пользователю предлагается завершить создание графа:</p> <pre>Введите две вершины, которые ребро соединяет (например, 1 2): q Ошибка! Вершина должна быть числом больше нуля. Введите 'Y', если хотите завершить создание графа</pre> <p>Примеры ввода, вызывающие сообщение об ошибке: q, 1n2, 1 1, 3 2 (если кол-во вершин не больше 2), 0 1, -1 2</p>

Как только были корректно введены вершины, пользователь может ввести вес ребра между ними:

```
Введите две вершины, которые ребро соединяет (например, 1 2): 1 2
Введите вес ребра: _
```

Проверка некорректного ввода:

```
Введите вес ребра: q
Ошибка! Вводимое значение должно являться числом.
Введите 'Y', если хотите завершить создание графа
```

Примеры ввода, вызывающие сообщение об ошибке: q, q2

При попытке ввести вес существующего ребра, пользователь будет об этом уведомлён, а также будет предложено выбрать значение, которое должно принимать ребро:

```
Введите две вершины, которые ребро соединяет (например, 1 2): 1 2
Введите вес ребра: 5
0 5
5 0
Введите 'Y', если хотите завершить создание графа
n
Введите две вершины, которые ребро соединяет (например, 1 2): 1 2
Введите вес ребра: 3
Встречены повторяющиеся рёбра между вершинами 1 и 2. Какое значение удалить?
[1] 5
[2] 3
1
Значение оставшегося ребра: 3
0 3
3 0
Введите 'Y', если хотите завершить создание графа
```

При некорректном вводе, у пользователя он запрашивается повторно до тех пор, пока не будет получено корректное значение:

```
Встречены повторяющиеся рёбра между вершинами 1 и 2. Какое значение удалить?
[1] 3
[2] 1
g
Некорректный ввод! Пожалуйста, попробуйте снова
Встречены повторяющиеся рёбра между вершинами 1 и 2. Какое значение удалить?
[1] 3
[2] 1
```

Примеры ввода, вызывающие сообщение об ошибке: q, q2, 3, 2q, -1

Если пользователю требуется удалить имеющееся ребро, он может присвоить ему нулевой вес:

```
0 3
3 0
Введите 'Y', если хотите завершить создание графа
n
Введите две вершины, которые ребро соединяет (например, 1 2): 1 2
Введите вес ребра: 0
Ребро было удалено
0 0
0 0
```

Продолжение Таблицы

	<p>При попытке присвоить нулевой вес ребру, которого не существует, пользователю будет выведена ошибка:</p> <pre>Введите две вершины, которые ребро соединяет (например, 1 2): 3 5 Введите вес ребра: 0 Ребро не сохранено, так как имеет нулевой вес.</pre> <p>После работы с графом, происходит проверка на связность:</p> <pre>Введите 'Y', если хотите завершить создание графа Y Граф не прошёл проверку на связность, поэтому был удалён</pre>
Случайная генерация графа	
Если нужно сгенерировать случайный граф, пользователю предоставляется такая возможность	<p>Чтобы начать генерацию графа, пользователь должен ввести количество вершин:</p> <pre>[1] Ввести граф самостоятельно [2] Ввести количество вершин и соединить их случайно [3] Считать граф с файла [0] Вернуться назад Пожалуйста, введите число, чтобы выполнить нужное действие: 2 Введите количество вершин, которое должно быть в сгенерированном графе:</pre> <p>В случае некорректного ввода на экран выводится ошибка и запрашивается повторный ввод:</p> <pre>Введите количество вершин, которое должно быть в сгенерированном графе: q Ошибка! Количество вершин должно быть числом больше нуля.</pre> <p>Примеры ввода, вызывающие сообщение об ошибке: -123, а, а1</p> <p>После генерации графа, пользователь может вывести граф на экран и сохранить сгенерированный граф в файл:</p> <pre>Введите количество вершин, которое должно быть в сгенерированном графе: 15 Приступаем к генерации графа... [1] Вывести получившийся граф на экран [2] Вывести получившийся граф в файл [0] Выйти</pre>

Продолжение Таблицы

Считывание графа с файла	
Выбрав соответствующее действие, пользователь может считать граф с файла	<p>Прибегнув к считыванию графа с файла, пользователю выводится предполагаемое количество вершин. Если считанный граф прошёл проверку на связность, пользователь будет об этом уведомлён. У пользователя имеется возможность вывести граф на экран, чтобы убедиться, что он считался правильно:</p> <pre>Предполагаемое количество вершин: 10 Граф был успешно создан [1] Вывести получившийся граф на экран [0] Выйти</pre> <p>Если файл пуст, об этом сообщается пользователю:</p> <pre>Пожалуйста, введите число, чтобы выполнить нужное действие: 3 Предполагаемое количество вершин: 0 Файл пуст.</pre> <p>Если граф неправильно описан, об этом также будет сообщено:</p> <pre>Ошибка! Файл содержит некорректные значения. Ошибка! Вес одного ребра имеет разные значения.</pre>
Сортировка рёбер	
Имеется несколько способов сортировок (как по методу, так и по целям), которые предоставляются пользователю	<p>Способы сортировок:</p> <pre>[1] Сортировать граф по весу рёбер [2] Сортировать граф по вершинам [3] Сравнение различных сортировок по скорости [0] Вернуться назад Пожалуйста, введите число, чтобы выполнить нужное действие:</pre> <p>Перейдя в сравнение сортировок по скорости, пользователь должен ввести количество их повторений. Если ввод будет некорректен, установится значение по умолчанию:</p> <pre>Сколько раз повторять сортировку? q Ошибка! Количество повторений не должно быть меньше 1. Было установлено значение по умолчанию. Сортировка графа по весу рёбер Сортировка вектором: 0 Пузырьковая сортировка: 0.001 Сортировка вставками: 0 Шейкер сортировка: 0.001 Сортировка графа по вершинам Сортировка вектором: 0 Пузырьковая сортировка: 0.001 Сортировка вставками: 0 Шейкер сортировка: 0.001</pre> <p>Примеры ввода, вызывающие ошибку: q, q1, 0, -2</p> <p>Если граф пуст, об этом также будет сообщено пользователю:</p> <pre>Пожалуйста, введите число, чтобы выполнить нужное действие: 3 Ошибка! Создайте граф, прежде чем сортировать его.</pre>

Поиск минимального остовного дерева (Алгоритм Краскала)	
Как только граф был создан, пользователь может найти минимальное остовное дерево	<p>У пользователя есть две возможности работы с алгоритмом Краскала:</p> <pre>[1] Найти минимальное остовное дерево и сохранить его [2] Определить время нахождения минимального остовного дерева без сохранения [0] Вернуться назад Пожалуйста, введите число, чтобы выполнить нужное действие: _</pre> <p>Если граф пуст, об этом сообщается пользователю:</p> <pre>Пожалуйста, введите число, чтобы выполнить нужное действие: 3 Ошибка! Граф пуст.</pre> <p>Если пользователю необходимо сохранить получившийся в результате граф, он может воспользоваться соответствующим действием:</p> <pre>Пожалуйста, введите число, чтобы выполнить нужное действие: 1 Минимальное остовное дерево: 0 23 0 0 49 0 0 48 0 0 23 0 18 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 35 0 17 0 0 0 49 0 0 35 0 0 0 0 0 0 0 0 0 0 0 0 0 49 0 28 0 0 0 17 0 0 0 0 0 0 48 0 0 0 0 49 0 0 0 0 0 0 0 0 0 0 0 0 0 68 0 0 0 0 0 28 0 0 68 0</pre> <p>Если же необходимо посмотреть на продуктивность выполнения поиска, пользователь может указать количество повторений, которые ему необходимы:</p>

Продолжение Таблицы

	<p>Пожалуйста, введите число, чтобы выполнить нужное действие: 2 Сколько раз повторять поиск? 60000 Минимальное остовное дерево: 0 23 0 0 49 0 0 48 0 0 23 0 18 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 35 0 17 0 0 0 49 0 0 35 0 0 0 0 0 0 0 0 0 0 0 0 49 0 28 0 0 0 17 0 0 0 0 0 0 48 0 0 0 0 49 0 0 0 0 0 0 0 0 0 0 0 0 0 68 0 0 0 0 0 28 0 0 68 0 Найдено за: 2.118</p> <p>Если ввод будет некорректен, установится значение по умолчанию:</p> <p>Пожалуйста, введите число, чтобы выполнить нужное действие: 2 Сколько раз повторять поиск? q Ошибка! Количество повторений не должно быть меньше 1. Было установлено значение по умолчанию. Минимальное остовное дерево: 0 23 0 0 49 0 0 48 0 0 23 0 18 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 35 0 17 0 0 0 49 0 0 35 0 0 0 0 0 0 0 0 0 0 0 0 49 0 28 0 0 0 17 0 0 0 0 0 0 48 0 0 0 0 49 0 0 0 0 0 0 0 0 0 0 0 0 0 68 0 0 0 0 0 28 0 0 68 0 Найдено за: 0</p> <p>Примеры ввода, вызывающие ошибку: q, q1, 0, -2</p>
Вывод текущего графа	
<p>Пользователю доступны два способа вывода текущего графа: списком рёбер и в виде матрицы</p>	<p>Список рёбер:</p> <p>[4] Вывести список рёбер [5] Вывести текущий граф [0] Закрыть программу Пожалуйста, введите число, чтобы выполнить нужное действие: 4 1. Ребро между 1 и 2 имеет вес 23 2. Ребро между 1 и 4 имеет вес 73 3. Ребро между 1 и 5 имеет вес 49 4. Ребро между 1 и 7 имеет вес 57 5. Ребро между 1 и 8 имеет вес 48 6. Ребро между 2 и 3 имеет вес 18</p> <p>Матрица:</p> <p>[5] Вывести текущий граф [0] Закрыть программу Пожалуйста, введите число, чтобы выполнить нужное действие: 5 0 23 0 73 49 0 57 48 0 0 23 0 18 0 53 0 0 0 0 0 0 18 0 59 0 0 0 0 0 82 73 0 59 0 35 0 17 0 0 0 49 53 0 35 0 86 0 0 0 0 0 0 0 0 86 0 94 49 0 28 57 0 0 17 0 94 0 68 98 0 48 0 0 0 0 49 68 0 95 0 0 0 0 0 0 98 95 0 68 0 0 82 0 0 28 0 0 68 0</p>

Продолжение Таблицы

Тестовые данные для алгоритма Краскала	
Исходная матрица	Результат
0 23 0 73 49 0 57 48 0 0 23 0 18 0 53 0 0 0 0 0 0 18 0 59 0 0 0 0 0 82 73 0 59 0 35 0 17 0 0 0 49 53 0 35 0 86 0 0 0 0 0 0 0 0 86 0 94 49 0 28 57 0 0 17 0 94 0 68 98 0 48 0 0 0 0 49 68 0 95 0 0 0 0 0 0 0 98 95 0 68 0 0 82 0 0 28 0 0 68 0	0 23 0 0 49 0 0 48 0 0 23 0 18 0 0 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 35 0 17 0 0 0 49 0 0 35 0 0 0 0 0 0 0 0 0 0 0 0 0 49 0 28 0 0 0 17 0 0 0 0 0 0 48 0 0 0 0 49 0 0 0 0 0 0 0 0 0 0 0 0 0 68 0 0 0 0 0 28 0 0 68 0
0 3 6 0 0 9 0 0 0 0 3 0 4 0 9 9 0 0 0 0 6 4 0 2 0 0 9 0 0 0 0 0 2 0 8 0 9 0 0 0 0 9 0 8 0 8 7 0 9 10 9 9 0 0 8 0 0 0 0 18 0 0 9 9 7 0 0 4 5 0 0 0 0 0 0 0 4 0 1 4 0 0 0 0 9 0 5 1 0 3 0 0 0 0 10 18 0 4 3 0	0 3 0 0 0 0 0 0 0 0 3 0 4 0 0 0 0 0 0 0 0 4 0 2 0 0 0 0 0 0 0 0 2 0 8 0 0 0 0 0 0 0 0 8 0 8 7 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 7 0 0 4 0 0 0 0 0 0 0 0 4 0 1 0 0 0 0 0 0 0 0 1 0 3 0 0 0 0 0 0 0 0 3 0
0 7 0 5 0 0 0 7 0 8 9 7 0 0 0 8 0 0 5 0 0 5 9 0 0 15 6 0 0 7 5 15 0 8 9 0 0 0 6 8 0 11 0 0 0 0 9 11 0	0 7 0 5 0 0 0 7 0 0 0 7 0 0 0 0 0 0 5 0 0 5 0 0 0 0 6 0 0 7 5 0 0 0 9 0 0 0 6 0 0 0 0 0 0 0 9 0 0
0 52 0 0 0 0 0 52 0 49 45 0 0 0 0 49 0 60 28 0 0 0 45 60 0 73 79 95 0 0 28 73 0 41 39 0 0 0 79 41 0 73 0 0 0 95 39 73 0	0 52 0 0 0 0 0 52 0 49 45 0 0 0 0 49 0 0 28 0 0 0 45 0 0 0 0 0 0 0 28 0 0 41 39 0 0 0 0 41 0 0 0 0 0 0 39 0 0

Сравнение сортировок графа		
Сортировка графа по весу рёбер	Сортировка вектором: 0.205 Пузырьковая сортировка: 28.386 Сортировка вставками: 2.86 Шейкер сортировка: 18.682	
Сортировка графа по вершинам	Сортировка вектором: 0.222 Пузырьковая сортировка: 39.095 Сортировка вставками: 3.002 Шейкер сортировка: 28.599	
Сравнительный анализ алгоритма Прима и Краскала		
	Алгоритм Прима	Алгоритм Краскала
Временная сложность	$O(V^2)$	$O(\log V)$
Вид графа	Только связные	Связные и несвязные
Работа алгоритма	Работает с вершинами, следующий шаг зависит от текущей вершины	Работает с предварительно отсортированными по весу рёбрами, следующий шаг не зависит от предыдущего
Представление графа	Список смежности	Список рёбер

Выводы.

Реализован алгоритм поиска минимального остовного дерева одним из предложенных алгоритмов. Был выбран алгоритм Краскала, как самая быстрая реализации алгоритма поиска минимального остовного дерева.

Разработана программа, способная записывать данные о графах и работать с ними, выводить данные. Также программа способна осуществлять сортировку введенных данных по параметрам, определяемые пользователем.

Были получены практические навыки работы с векторами, итераторами; изучены способы реализации поиска минимального остовного дерева; проведён их сравнительный анализ.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ

Название файла: cw2.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <time.h>
#include <algorithm> // sort
using namespace std;

struct edge
{
    size_t first; // первая вершина (всегда меньше в
    // второй и не равна ей)
    size_t second; // вторая вершина
    int size; // вес ребра
};

// Создаём новый элемент списка рёбер
edge* createEdge(size_t frst, size_t scnd, int s)
{
    if (frst == 0 || scnd == 0 || s == 0)
    {
        cout << "Ребро не было создано, так как к
        // акой-то из элементов имеет нулевое значени
        // е\n";
        return NULL;
    }
    edge* newEdge = new edge;
    newEdge->first = frst;
    newEdge->second = scnd;
    newEdge->size = s;
    return newEdge;
}

// Вывод графа в виде матрицы, используя спис
// ок рёбер
void outputGraph(vector<edge> &graph, size_t N)
{
    if (N <= 0) { cout << "Граф пуст.\n"; }

    if (graph.empty())
    {
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            { cout << "0 "; }
        }
    }
}
```



```

        cout << "\n";
    }
    return;
}

int **Arr = new int* [N];
for (size_t i = 0; i < N; ++i)
{ Arr[i] = new int[N](); } //создаём двумерный массив

//задаём значения для вывода матрицы
vector<edge>::iterator iter = graph.begin(); //итератор, который перебирает весь список
do
{
    if (Arr[iter->first - 1][iter->second - 1] != 0) //если записываемое ребро уже встречалось - ошибка
    {
        char sw = '\0'; //для команды пользоваться
        bool check = true; //для выхода из меню с удалением
        vector<edge>::iterator del = graph.begin(); //итератор, ссылающийся на удаляемое ребро
        do
        {
            cout << "Встречены повторяющиеся рёбра между вершинами " << iter->first << " и " << iter->second << ". Какое значение удалить?\n";
            cout << "\x1b[32m[1]\x1b[0m " << Arr[iter->first - 1][iter->second - 1] << "\n"; //значение старого ребра
            cout << "\x1b[32m[2]\x1b[0m " << iter->size << "\n"; //значение нового ребра

            cin >> sw;
            while (cin.get() != '\n') { sw = ' '; }; //если строка содержит более одного символа, возвращается ошибка
            switch (sw)
            {
            case '1':
            {
                while ((del->first != iter->first) && (del->second != iter->second)) //пока не найден первый элемент с нужными координатами
                { ++del; } //итератор, указывающий на старое ребро
            }
            }
        }
    }
}

```

```

        del->size = iter->size; //"удаляемое"
ребро присваивает значение нового
        Arr[iter->first - 1][iter->second - 1] = iter-
>size; //запоминается новое значение
        Arr[iter->second - 1][iter->first - 1] = iter-
>size;

        iter = graph.erase(iter);
        cout << "Значение оставшегося
ребра: " << del->size << "\n";
        check = false;
        break;
    }
    case '2':
        while ((del->first != iter->first) && (del-
>second != iter->second)) //пока не найден первый элем
ент с нужными координатами
        {
            ++del;
        } //итератор, указывающий на
старое ребро
        Arr[iter->first - 1][iter->second - 1] = del-
>size; //запоминается новое значение
        Arr[iter->second - 1][iter->first - 1] = del-
>size;

        iter = graph.erase(iter);
        cout << "Значение оставшегося
ребра: " << del->size << "\n";
        check = false;
        break;
    default:
        cout << "Некорректный ввод! По
жалуйста, попробуйте снова\n";
        break;
    }
} while (check);
if (iter == graph.end()) { break; }
}
else
{
    Arr[iter->first - 1][iter->second - 1] = iter->size;
    Arr[iter->second - 1][iter->first - 1] = iter->size;
}
++iter;
} while (iter != graph.end());

//ВЫВОД
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)

```

```

        {
            cout << Arr[i][j] << " ";
        }
        cout << "\n";
    }

    for (size_t i = 0; i < N; ++i)
    { delete[] Arr[i]; }
    delete[] Arr; //удаляем двумерный массив
}

//Меняем местами первую и вторую вершину, если первая содержит значение больше
void swapVertex(size_t &frst, size_t &scnd)
{
    if (frst < scnd) { return; } //если первая вершина меньше второй - всё ок
    size_t c = frst;
    frst = scnd;
    scnd = c;
}

//Проверка графа на связность
bool connectivity(vector<edge>& graph, size_t& N)
{
    if (N <= 1 || graph.empty()) { return 0; }
    if (N == 2 && !(graph.empty())) { return 1; } //если имеется всего две вершины и одно ребро между ними

    //далее идёт алгоритм, если есть хотя бы 3 вершины
    bool check = true; //проверяет, есть ли в массиве хотя бы один элемент с пометкой "1"
    int* Arr = new int[N](); //по умолчанию все вершины имеют пометку "0"
    size_t curr = 0; //рассматриваемая вершина - 1
    vector<edge>::iterator iter = graph.begin(); //итератор графа
    Arr[0] = 1;
    do
    {
        Arr[curr] = 2;
        iter = graph.begin();
        do
        {
            //если текущая вершина меньше соединяемой
            if ((iter->first - 1 == curr)&&(Arr[iter->second - 1] == 0)) //если найдена нужная вершина

```

```

        { //проверяем, с кем она связана и п
омечаем вторую, если она ещё не помечена
            Arr[iter->second - 1] = 1;
        }

        //если текущая вершина больше соединяемой
        if ((iter->second - 1 == curr) && (Arr[iter->first - 1]
== 0)) //если найдена нужная вершина
        { //проверяем, с кем она связана и п
омечаем вторую, если она ещё не помечена
            Arr[iter->first - 1] = 1;
        }

        ++iter;
    } while (iter != graph.end()); //пока не просмотре
ны все рёбра

    check = false;
    for (size_t i = 0; i < N; i++)
    {
        if (Arr[i] == 1)
        {
            curr = i;
            check = true;
            break;
        }
    }
    } while (check);

    //если остались вершин, помеченные "0", то
граф несвязный
    for (size_t i = 0; i < N; i++)
    {
        if (Arr[i] == 0) { delete[] Arr; return 0; }
    }
    delete[] Arr;
    return 1;
}

//выход из создания ребра
void endInputEdge(char &sw)
{
    cout << "Введите 'Y', если хотите завершить с
оздание графа\n";
    cin >> sw;
    while (cin.get() != '\n') { sw = ' '; }; //если строка со
держит более одного символа, присваивается
пробел

```

```

}

//обработка неправильного ввода
void errorInput(char &sw)
{
    cin.clear();
    cin.sync();
    while (cin.get() != '\n');
    endInputEdge(sw);
}

bool delEdge(vector<edge>& graph, size_t first, size_t second)
{
    if (graph.empty()) { return 0; }
    vector<edge>::iterator iter = graph.end(); //итератор, кот
орый перебирает весь список
    do
    {
        iter--;
        if (iter->first == first && iter->second == second)
        {
            graph.erase(iter);
            return 1; //удаление успешно
        }
    } while (iter != graph.begin());

    return 0;
}

//Создаём граф пользователем
void inputUser(vector<edge>& graph, size_t& N)
{
    edge* newEdge = NULL; //создаваемое ребро
    size_t first = 0, second = 0; //вершины
    int x; //вес ребра
    string input; //ввод пользователем
    char sw = '\0';
    cout << "Введите количество вершин: ";
    while (!(cin >> x) || (x <= 0)) //проверка на коррект
ность ввода
    {
        cout << "Ошибка! Количество вершин должн
о быть числом больше нуля.\n";
        cin.clear();
        cin.sync();
        while (cin.get() != '\n');
        cout << "Введите количество вершин: ";
    }
}

```

```

    cin.ignore(32767, '\n'); //игнор лишних символов п
о с л е ч и с л а , е с л и о н и е с т ь
    N = x;

    outputGraph(graph, N); //вывод пустого графа

    while (sw != 'Y') {
        cout << "Введите две вершины, которые ре
б р о с о е д и н я е т ( н а п р и м е р , 1 2): ";
        cin >> x;
        if (!cin || x <= 0) //к о р р е к т н о с т ь в в о д а
        {
            cout << "О ш и б к а ! В е р ш и н а д о л ж н а б ы т ь
ч и с л о м б о л ь ш е н у л я .\n";
            errorInput(sw);
            if (sw == 'Y') { break; }
            else { continue; }
        }
        else { first = x; }

        cin >> x;
        if (!cin || x <= 0) //к о р р е к т н о с т ь в в о д а
        {
            cout << "О ш и б к а ! В е р ш и н а д о л ж н а б ы т ь
ч и с л о м б о л ь ш е н у л я .\n";
            errorInput(sw);
            if (sw == 'Y') { break; }
            else { continue; }
        }
        else { second = x; }
        cin.ignore(32767, '\n'); //игнор лишних символо
в п о с л е ч и с л а , е с л и о н и е с т ь

        if (first <= N && second <= N && first != second)
        {
            cout << "Введите вес ребра: ";
            cin >> x;
            if (!cin) //к о р р е к т н о с т ь в в о д а
            {
                cout << "О ш и б к а ! В в о д и м о е з н а ч е н и
е д о л ж н о я в л я т ь с я ч и с л о м .\n";
                errorInput(sw);
                if (sw == 'Y') { break; }
                else { continue; }
            }
            if (x != 0)
            {
                swapVertex(first, second); //проверяем, чт
о б ы п е р в а я в е р ш и н а б ы л а м е н ь ш е в т о р о е

```

```

        newEdge = createEdge(first, second, x); //присв
аиваем указателю новое ребро
        if (newEdge != NULL) { graph.push_back(*newEdge); }
//сохраняем его
    }
    else
    {
        if (delEdge(graph, first, second))
        {
            cout << "Р е б р о  б ы л о  у д а л е н о .\n";
        }
        else
        {
            cout << "Р е б р о  н е  с о х р а н е н о ,  т а
к  к а к  и м е е т  н у л е в о й  в е с .\n";
        }
    }
}
else //корректность ввода
{
    cout << "О ш и б к а !  З н а ч е н и е  в е р ш и н ы  д о л
жно  я в л я т ь с я  ч и с л о м  б о л ь ш е  н у л я ,  а  т а к ж е  н е
р а в н я т ь с я  д р у г  д р у г у .\n";
    endInputEdge(sw);
    if (sw == 'Y') { break; }
    else { continue; }
}
outputGraph(graph, N);
endInputEdge(sw);
}

//проверка на связность
if (!graph.empty() && connectivity(graph, N)) //если граф п
о л у ч и л с я  с в я з н ы м
{
    cout << "Г р а ф  б ы л  у с п е ш н о  с о з д а н \n";
}
else
{
    cout << "Г р а ф  н е  п р о ш ё л  п р о в е р к у  н а  с в я з
н о с т ь ,  п о э т о м у  б ы л  у д а л ё н \n";
    if (!graph.empty()) { graph.clear(); N = 0; }
}
}

//Функция подсчёта количества строк
size_t countRows()
{
    ifstream fin("graph.txt");

```

```

    if (fin.is_open())
    {
        size_t temp = 0; //количество строк
        string data;
        while (!fin.eof()) //пока указатель потока не
достигнет конца файла
        {
            getline(fin, data); //считывается строка
            if (data != "\0") { temp++; } //в счётчик не по
падают пустые строки
        }
        fin.close();
        return temp;
    }
    else return 0;
}

//Считываем граф с файла
void inputFile(vector<edge>& graph, size_t& N)
{
    ifstream fin("graph.txt");
    if (!fin.is_open()) // если файл не открыт
        cout << "Файл не был открыт.\n"; // сообщит
ь об этом
    else
    {
        N = countRows();
        cout << "Предполагаемое количество верш
ин: " << N << "\n";
        if (N == 0) { cout << "Файл пуст.\n"; }
        else
        {
            edge* newEdge = NULL; //создаваемое ребро
            //string data; // буфер промежуточного х
ранения считываемого из файла текста
            int** gr = new int* [N];
            for (size_t i = 0; i < N; ++i) { gr[i] = new int[N](); }
            //создаём двумерный массив

            for (size_t i = 0; i < N; i++)
            {
                //getline(fin, data); // Считываем очере
дную строку
                for (size_t j = 0; j < N; j++)
                {
                    fin >> gr[i][j];
                    if (!fin) //корректность ввода
                    {

```



```

        cout << "\nОшибка! Файл содержит некорректные значения.\n";
        return;
    }
    if (gr[i][i] != 0) { cout << "Ошибка! Главная диагональ матрицы должна содержать только нулевые значения.\n"; return; }
}

for (size_t i = 0; i < N; i++)
{
    for (size_t j = 1 + i; j < N; j++)
    {
        if (gr[i][j] != gr[j][i]) { cout << "\nОшибка! Вес одного ребра имеет разные значения.\n"; return; }

        else
        {
            if (gr[i][j] != 0)
            {
                newEdge = createEdge(i + 1, j + 1, gr[i][j]); //присваиваем указателю новое ребро
                if (newEdge != NULL) {
                    graph.push_back(*newEdge); } //сохраняем его
            }
        }
    }
}

if (!graph.empty() && connectivity(graph, N)) //если граф получился связным
{
    cout << "Граф был успешно создан\n";
}
else
{
    cout << "Граф не прошёл проверку на связность, поэтому был удалён\n";
    if (!graph.empty()) { graph.clear(); N = 0; }
}

bool chk = false;
do
{
    chk = true;
    char sw = '\0';
    cout << "\x1b[32m[1]\x1b[0m Вывести полученный граф на экран\n";
}

```

```

        cout << "\x1b[32m[0]\x1b[0m Выйти\n";
        cin >> sw;
        while (cin.get() != '\n') { sw = ' '; }; //если
строка содержит более одного символа, возвра
щается ошибка

        switch (sw)
        {
        case '0':
            chk = false;
            break;
        case '1':
            outputGraph(graph, N);
            break;
        default:
            cout << "Вы ввели некорректное
значение. Повторите снова\n";
            break;
        }
    } while (chk);

    for (size_t i = 0; i < N; ++i) { delete[] gr[i]; }
    delete[] gr; //удаляем двумерный массив
В
    }
    fin.close();
}
}

//Вывести граф в файл
void outputFile(vector<edge>& graph, size_t N)
{
    if (graph.empty())
    {
        cout << "Ошибка! Граф пуст\n";
        return;
    }

    ofstream fout("graph.txt");
    if (!fout.is_open()) { cout << "\nОшибка сохранения
!\n"; }
    else
    {
        int** Arr = new int* [N];
        for (size_t i = 0; i < N; ++i)
        {
            Arr[i] = new int[N]();
        } //создаём двумерный массив

        //задаём значения для вывода матрицы

```

```

        vector<edge>::iterator iter = graph.begin(); //и т е р а т о р
, к о т о р ы й  п е р е б и р а е т  в е с ь  с п и с о к
        do
        {
            Arr[iter->first - 1][iter->second - 1] = iter->size;
            Arr[iter->second - 1][iter->first - 1] = iter->size;
            ++iter;
        } while (iter != graph.end());

//в ы в о д
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        fout << Arr[i][j] << " ";
    }
    fout << "\n";
}

for (size_t i = 0; i < N; ++i)
{
    delete[] Arr[i];
}
delete[] Arr; //у д а л я е м  д в у м е р н ы й  м а с с и в
fout.close();
}

/*п р о в е р к а ,  и м е е т с я  л и  т а к а я  к о м б и н а ц и я  в е р
шин. 0 - е с л и  н е т
bool vertexCombination(vector<edge>& graph, size_t frst, size_t scnd)
{
    if (graph.empty()) { return 0; }

    vector<edge>::iterator iter = graph.begin(); //и т е р а т о р ,  к о
т о р ы й  п е р е б и р а е т  в е с ь  с п и с о к
    do
    {
        if (iter->first == frst && iter->second == scnd) { return 1; }
        ++iter;
    } while (iter != graph.end());

    return 0;
}

//Р а н д о м н а я  г е н е р а ц и я  г р а ф а
void generateGraph(vector<edge>& graph, size_t& N)
{
    edge* newEdge = NULL; //с о з д а в а е м о е  р е б р о

```

```

        //size_t count = 0; //сколько попыток ушло на ге
н е р а ц и ю   г р а ф а

        cout << "Введите количество вершин, которое
должно быть в сгенерированном графе: ";
        while (!(cin >> N) || (N == 0)) //проверка на коррект
ность ввода
        {
            cout << "Ошибка! Количество вершин должн
о быть числом больше нуля.\n";
            cin.clear();
            cin.sync();
            while (cin.get() != '\n');
            cout << "Введите количество вершин: ";
        }
        cout << "Приступаем к генерации графа...\n";

        if (!graph.empty()) { graph.clear(); } //очищаем предва
р и т е л ь н о   г р а ф
        for (size_t i = 0; i < N; i++)
        {
            size_t max = rand() % (N); //количество вершин,
с которыми будет соединено i. Не может превы
шать число оставшихся вершин
            for (size_t j = i + 1; j <= max; j++) //вторая верши
на не должна быть меньше или равно текущей
            {
                if (!vertexCombination(graph, i + 1, j + 1)) //если
такой комбинации нет
                {
                    newEdge = createEdge(i + 1, j + 1, rand() % 88 +
11); //!! МЕНЯЕМОЕ ЗНАЧЕНИЕ. Влияет на диапазо
н веса рёбер
                    if (newEdge != NULL) { graph.push_back(*newEdge); }
//сохраняем его
                }
            }
        }

        bool chk = false;
        do
        {
            chk = true;
            char sw = '\0';
            cout << "\x1b[32m[1]\x1b[0m Вывести получившийс
я граф на экран\n";
            cout << "\x1b[32m[2]\x1b[0m Вывести получившийс
я граф в файл\n";

```

```

        cout << "\x1b[32m[0]\x1b[0m В ы й т и\n";
        cin >> sw;
        while (cin.get() != '\n') { sw = ' '; }; //если строка
содержит более одного символа, возвращаетс
я ошибка

        switch (sw)
        {
        case '0':
            chk = false;
            break;
        case '1':
            outputGraph(graph, N);
            break;
        case '2':
            outputFile(graph, N);
            break;
        default:
            cout << "Вы ввели некорректное значе
ние. Повторите снова\n";
            break;
        }
    } while (chk);
}
*/

//Рандомная генерация графа v2.0
void generateGraphNew (vector<edge>& graph, size_t& N)
{
    int x;
    cout << "Введите количество вершин, которое
должно быть в сгенерированном графе: ";
    while (!(cin >> x) || (x <= 0)) //проверка на коррект
ность ввода
    {
        cout << "Ошибка! Количество вершин должн
о быть числом больше нуля.\n";
        cin.clear();
        cin.sync();
        while (cin.get() != '\n');
        cout << "Введите количество вершин: ";
    }
    N = x;
    cout << "Приступаем к генерации графа...\n";

    edge* newEdge = NULL; //создаваемое ребро
    if (!graph.empty()) { graph.clear(); } //очищаем предва р
ительно граф
    for (size_t i = 0; i < N; i++)

```

```

    {
        if (i != N - 1)
        {
            newEdge = createEdge(i + 1, i + 2, rand() % 88 + 11);
            ///!! МЕНЯЕМОЕ ЗНАЧЕНИЕ. Влияет на диапазон ве
            са рёбер
            if (newEdge != NULL) { graph.push_back(*newEdge); } //с
            о х р а н я е м е г о
        }
        size_t max = i + 2000;
        if (max > N) { max = N; }
        for (size_t j = i + 2; j < max; j++) //вторая вершин
        а не должна быть меньше или равно текущей
        {
            if (!(rand() % 3))
            {
                newEdge = createEdge(i + 1, j + 1, rand() % 88 +
            11); ///!! МЕНЯЕМОЕ ЗНАЧЕНИЕ. Влияет на диапазо
            н веса рёбер
                if (newEdge != NULL) { graph.push_back(*newEdge); }
            //с о х р а н я е м е г о
            }
        }
        if ((N > 1000) && i % 1000 == 0)
        {
            cout << i / 1000 << " тыс. вершина были сге
            н е р и р о в а н ы\n";
        }
    }

    bool chk = false;
    do
    {
        chk = true;
        char sw = '\0';
        cout << "\x1b[32m[1]\x1b[0m Вывести получивший с
        я г р а ф на экран\n";
        cout << "\x1b[32m[2]\x1b[0m Вывести получивший с
        я г р а ф в файл\n";
        cout << "\x1b[32m[0]\x1b[0m Выйти\n";
        cin >> sw;
        while (cin.get() != '\n') { sw = ' '; }; //если строка
        с о д е р ж и т более одного символа, возвращаетс
        я о ш и б к а

        switch (sw)
        {
            case '0':
                chk = false;

```

```

        break;
    case '1':
        outputGraph(graph, N);
        break;
    case '2':
        outputFile(graph, N);
        break;
    default:
        cout << "Вы ввели некорректное значение. Повторите снова\n";
        break;
    }
} while (chk);
}

//Вывод списка рёбер
void outputEdgeList(vector<edge>& graph)
{
    if (graph.empty()) { cout << "Список рёбер пуст\n"; return; }
    vector<edge>::iterator iter = graph.begin(); //итератор, который перебирает весь список
    size_t i = 0;
    do
    {
        cout << i + 1 << ". Рёбро между " << iter->first << " и " << iter->second << " имеет вес " << iter->size << "\n";
        i++;
        ++iter;
    } while (iter != graph.end());
}

//== СОРТИРОВКИ ==
//== Сортировка графа по весу рёбер ==
//Сортировка вектором
bool compareSize(edge item1, edge item2)
{
    return (item1.size < item2.size);
}

void vectorSizeSort(vector<edge>& graph)
{
    sort(graph.begin(), graph.end(), compareSize);
}

//Пузырьковая сортировка
void bubbleSizeSort(vector<edge>& graph)
{
    vector<edge>::iterator iter = graph.begin(); //итератор, который перебирает весь список

```

```

vector<edge>::iterator jter = graph.begin();
edge tmp;
for (iter = graph.begin(); iter < graph.end(); iter++) {
    for (jter = (graph.end() - 1); jter >= (iter + 1); jter--) {
        if (jter->size < (jter - 1)->size) {
            tmp = *jter;
            *jter = *(jter - 1);
            *(jter - 1) = tmp;
        }
    }
}
}

```

//Сортировка вставками

```

void insertSizeSort(vector<edge>& graph)
{

```

```

    vector<edge>::iterator iter = graph.begin(); //итератор, ко
    торый перебирает весь список
    vector<edge>::iterator jter = graph.begin();
    edge key;
    for (jter = (graph.begin() + 1); jter < graph.end(); jter++) {
        key = *jter;
        iter = jter - 1;
        while (iter >= graph.begin() && iter->size > key.size) {
            *(iter + 1) = *iter;
            *(iter) = key;
            if (iter != graph.begin()) { iter--; }
            else { break; }
        }
    }
}

```

//Быстрая сортировка

```

void shakerSizeSort(vector<edge>& graph)
{

```

```

    vector<edge>::iterator iter = graph.begin(); //итератор, ко
    торый перебирает весь список
    edge tmp;
    vector<edge>::iterator right = graph.end() - 1;
    vector<edge>::iterator left = graph.begin();
    while (left <= right)
    {
        for (iter = right; iter > left; iter--)
        {
            if ((iter - 1)->size > iter->size)
            {
                tmp = *iter;
                *iter = *(iter - 1);
                *(iter - 1) = tmp;
            }
        }
    }
}

```



```

    }
    left++;
    for (iter = left; iter < right; iter++)
    {
        if (iter->size > (iter + 1)->size)
        {
            tmp = *iter;
            *iter = *(iter + 1);
            *(iter + 1) = tmp;
        }
    }
    right--;
}

}

//== С о р т и р о в к а   г р а ф а   п о   в е р ш и н а м ==
//С о р т и р о в к а   в е к т о р о м
bool compareVert(edge item1, edge item2)
{
    size_t vertex1, vertex2;
    vertex1 = item1.first * 10 + item1.second;
    vertex2 = item2.first * 10 + item2.second;
    return (vertex1 < vertex2);
}

void vectorVertSort(vector<edge>& graph)
{
    sort(graph.begin(), graph.end(), compareVert);
}

//П у з ы р ь к о в а я   с о р т и р о в к а
void bubbleVertSort(vector<edge>& graph)
{
    vector<edge>::iterator iter = graph.begin(); //и т е р а т о р ,   к о
    т о р ы й   п е р е б и р а е т   в е с ь   с п и с о к
    vector<edge>::iterator jter = graph.begin();
    edge tmp;
    for (iter = graph.begin(); iter < graph.end(); iter++) {
        for (jter = (graph.end() - 1); jter >= (iter + 1); jter--) {
            if ((jter->first * 10 + jter->second) < ((jter - 1)-
>first * 10 + (jter - 1)->second)) {
                tmp = *jter;
                *jter = *(jter - 1);
                *(jter - 1) = tmp;
            }
        }
    }
}

//С о р т и р о в к а   в с т а в к а м и

```

```

void insertVertSort(vector<edge>& graph)
{
    vector<edge>::iterator iter = graph.begin(); //и т е р а т о р , к о
    т о р ы й  п е р е б и р а е т  в е с ь  с п и с о к
    vector<edge>::iterator jter = graph.begin();
    edge key;
    for (jter = (graph.begin() + 1); jter < graph.end(); jter++) {
        key = *jter;
        iter = jter - 1;
        while (iter >= graph.begin() && ((iter->first * 10 + iter-
>second) > (key.first * 10 + key.second))) {
            *(iter + 1) = *iter;
            *(iter) = key;
            if (iter != graph.begin()) { iter--; }
            else { break; }
        }
    }
}

```

//Б ы с т р а я с о р т и р о в к а

```

void shakerVertSort(vector<edge>& graph)
{
    vector<edge>::iterator iter = graph.begin(); //и т е р а т о р , к о
    т о р ы й  п е р е б и р а е т  в е с ь  с п и с о к
    edge tmp;
    vector<edge>::iterator right = graph.end() - 1;
    vector<edge>::iterator left = graph.begin();
    while (left <= right)
    {
        for (iter = right; iter > left; iter--)
        {
            if (((iter - 1)->first * 10 + (iter - 1)->second) >
(iter->first * 10 + iter->second))
            {
                tmp = *iter;
                *iter = *(iter - 1);
                *(iter - 1) = tmp;
            }
        }
        left++;
        for (iter = left; iter < right; iter++)
        {
            if ((iter->first * 10 + iter->second) > ((iter + 1)-
>first * 10 + (iter + 1)->second))
            {
                tmp = *iter;
                *iter = *(iter + 1);
                *(iter + 1) = tmp;
            }
        }
        right--;
    }
}

```

```

    }
}

void compareSort(vector<edge> graph, size_t N)
{
    int x;
    cout << "Сколько раз повторять сортировку
?\\n";
    cin >> x;
    if (!cin || x <= 0)
    {
        cout << "Ошибка! Количество повторений н
е должно быть меньше 1. Было установлено зна
чение по умолчанию.\\n";
        cin.clear();
        cin.sync();
        while (cin.get() != '\\n');
        x = 1;
    }

    clock_t start, end;
    cout << "\\x1b[36mСортировка графа по весу рёбе
р\\x1b[0m\\n";
    start = clock();
    for (int i = 0; i <= x; i++) { vectorSizeSort(graph); }
    end = clock();
    cout << "Сортировка вектором: " << ((double)end -
start) / ((double)CLOCKS_PER_SEC) << "\\n";

    start = clock();
    for (int i = 0; i <= x; i++) { bubbleSizeSort(graph); }
    end = clock();
    cout << "Пузырьковая сортировка: " << ((double)end
- start) / ((double)CLOCKS_PER_SEC) << "\\n";

    start = clock();
    for (int i = 0; i <= x; i++) { insertSizeSort(graph); }
    end = clock();
    cout << "Сортировка вставками: " << ((double)end -
start) / ((double)CLOCKS_PER_SEC) << "\\n";

    start = clock();
    for (int i = 0; i <= x; i++) { shakerSizeSort(graph); }
    end = clock();
    cout << "Шейкер сортировка: " << ((double)end - start)
/ ((double)CLOCKS_PER_SEC) << "\\n";

    cout << "\\x1b[36mСортировка графа по вершинам
\\x1b[0m\\n";
    start = clock();

```

```

    for (int i = 0; i <= x; i++) { vectorVertSort(graph); }
    end = clock();
    cout << "Сортировка вектором: " << ((double)end -
start) / ((double)CLOCKS_PER_SEC) << "\n";

    start = clock();
    for (int i = 0; i <= x; i++) { bubbleVertSort(graph); }
    end = clock();
    cout << "Пузырьковая сортировка: " << ((double)end
- start) / ((double)CLOCKS_PER_SEC) << "\n";

    start = clock();
    for (int i = 0; i <= x; i++) { insertVertSort(graph); }
    end = clock();
    cout << "Сортировка вставками: " << ((double)end -
start) / ((double)CLOCKS_PER_SEC) << "\n";

    start = clock();
    for (int i = 0; i <= x; i++) { shakerVertSort(graph); }
    end = clock();
    cout << "Шейкер сортировка: " << ((double)end - start)
/ ((double)CLOCKS_PER_SEC) << "\n";
}

void marksChange(int min, int max, int& m, int*& Arr, size_t& N)
{
    m = 0; //находим новый максимум
    for (size_t i = 0; i < N; i++) //просматриваем все с
ущестствующие метки
    {
        if (max == Arr[i]) //метки, равные большей
        {
            Arr[i] = min; //заменяем на метки, равны
е меньше
        }
        if (Arr[i] > m)
        {
            m = Arr[i]; //запоминаем новый миниму
м
        }
    }
}

//Поиск минимального остовного дерева
void searchTree(vector<edge>& graph, size_t N)
{
    vector<edge> tree(NULL); //список рёбер итогового
дерева
    if (N <= 1 || graph.empty()) { return; }

```

```

    if (N == 2 && !(graph.empty())) { tree = graph; return; } //если
имеется всего две вершины и одно ребро между
у ними

    //далее идёт алгоритм, если есть хотя бы 3
вершины
    vectorSizeSort(graph); //сортировка рёбер по весу
    int* Arr = new int[N](); //по умолчанию все вершин
ы имеют пометку "0"
    int m = 0; //самая большая существующая поме
тка
    for (vector<edge>::iterator iter = graph.begin(); iter <
graph.end(); iter++) //проходим каждое ребро
    {
        if ((Arr[iter->first - 1] == 0) && (Arr[iter->second - 1] ==
0)) //если вершины не помечены
        {
            tree.push_back(*iter); //запоминаем ребро в
дереве
            m++; //присваиваем метку, больше сущ
ествующих
            Arr[iter->first - 1] = m; //помечаем, что вер
шина в дереве
            Arr[iter->second - 1] = m;
        }
        else
        {
            if (Arr[iter->first - 1] != Arr[iter->second - 1]) //ес
ли вершины не принадлежат одной группе рёб
ер
            {
                tree.push_back(*iter); //запоминаем ребр
о в дереве
                if ((Arr[iter->first - 1] == 0) || (Arr[iter-
>second - 1] == 0)) //если одна из вершин не помечен
а
                {
                    if (Arr[iter->first - 1] == 0) //выясняе
м, какая из вершин равна нулю
                    {
                        Arr[iter->first - 1] = Arr[iter->second
- 1]; //присваивается метка другого ребра
                    }
                    else
                    {
                        Arr[iter->second - 1] = Arr[iter->first
- 1]; //присваивается метка другого ребра
                    }
                }
            }
        }
    }

```

```

        }
        else
        {
            if (Arr[iter->first - 1] < Arr[iter->second -
1]) //выясняем наименьшую метку
            {
                marksChange(Arr[iter->first - 1],
Arr[iter->second - 1], m, Arr, N); //присваивается наимен
ьшая метка
            }
            else
            {
                marksChange(Arr[iter->second - 1],
Arr[iter->first - 1], m, Arr, N); //присваивается наимен
ьшая метка
            }
        }
    }
}
delete[] Arr;
graph = tree;
return;
}

```

```

//Поиск минимального остовного дерева
void searchTreeTime(vector<edge> graph, size_t N)
{
    int x;
    cout << "Сколько раз повторять поиск?\n";
    cin >> x;
    if (!cin || x <= 0)
    {
        cout << "Ошибка! Количество повторений н
е должно быть меньше 1. Было установлено зна
чение по умолчанию.\n";
        cin.clear();
        cin.sync();
        while (cin.get() != '\n');
        x = 1;
    }

    clock_t start, end;
    start = clock();
    for (int i = 0; i <= x; i++) { searchTree(graph, N); }
    end = clock();

    cout << "Минимальное остовное дерево: \n";
    outputGraph(graph, N);
}

```

```

        cout << "Найденно за: " << ((double)end - start) / ((double)CLOCKS_PER_SEC) << "\n";
        return;
    }

int main()
{
    setlocale(0, "");
    vector<edge> graph(NULL); //список рёбер графа
    size_t N = 0; //количество вершин графа

    bool check = true; //выход из меню
    bool check1 = false; //выход из подменю
    //false - заканчивает цикл, приводя непосредственно к выходу
    do {
        //system("cls");
        char sw = ' '; //переключатель главного меню
        char sw1 = ' '; //переключатель саб-меню
        cout << "\nВыберите нужный раздел: \n";
        cout << "\x1b[32m[1]\x1b[0m Ввести новый граф\n";
        cout << "\x1b[32m[2]\x1b[0m Сортировка рёбер\n";
        cout << "\x1b[32m[3]\x1b[0m Поиск минимального
остовного дерева (Алгоритм Краскала)\n";
        cout << "\x1b[32m[4]\x1b[0m Вывести список рёбер\n";
        cout << "\x1b[32m[5]\x1b[0m Вывести текущий граф\n";
        cout << "\x1b[32m[0]\x1b[0m Закрыть программу\n";
        cout << "Пожалуйста, введите число, чтобы
выполнить нужное действие: ";

        cin >> sw;
        while (cin.get() != '\n') { sw = ' '; }; //если строка
содержит более одного символа, возвращаетс
я ошибка

        switch (sw)
        {

        case '1': //[1] Ввести новый граф
            do {
                check1 = false;
                sw1 = ' ';
                cout << "\n\x1b[32m[1]\x1b[0m Ввести граф
самостоятельно\n";

```

```

        cout << "\x1b[32m[2]\x1b[0m Ввести количе
с т в о в е р ш и н и с о е д и н и т ь и х с л у ч а й н о\n";
        cout << "\x1b[32m[3]\x1b[0m Считать граф
с ф а й л а\n";
        cout << "\x1b[32m[0]\x1b[0m Вернуться на з
а д\n";

        cout << "Пожалуйста, введите числ
о, чтобы выполнить нужное действие: ";

        cin >> sw1;
        while (cin.get() != '\n') { sw1 = ' '; };

        switch (sw1)
        {
        case '1': //[1] Ввести граф самостоя
т е л ь н о
                if (!graph.empty()) { graph.clear(); N = 0; }
                inputUser(graph, N);
                break;
        case '2': //[2] Ввести количество ве
р ш и н и с о е д и н и т ь и х с л у ч а й н о
                generateGraphNew(graph, N);
                break;
        case '3': //[3] Считать граф с файла
                if (!graph.empty()) { graph.clear(); N = 0; }
                inputFile(graph, N);
                break;
        case '0': //[0] Назад
                break;
        default:
                cout << "Ошибка! Пожалуйста, по
п р о б у й т е с н о в а\n";
                check1 = true; //цикл пойдёт заново
                break;
        }
    } while (check1);
    break;

    case '2': //[2] Сортировка рёбер
    do {
        check1 = false;
        sw1 = ' ';
        cout << "\n\x1b[32m[1]\x1b[0m Сортировать
г р а ф п о в е с у р ё б е р\n";
        cout << "\x1b[32m[2]\x1b[0m Сортировать г
р а ф п о в е р ш и н а м\n";

```



```

        cout << "\x1b[32m[3]\x1b[0m Сравнение раз
личных сортировок по скорости\n";
        cout << "\x1b[32m[0]\x1b[0m Вернуться на з
ад\n";

        cout << "Пожалуйста, введите числ
о, чтобы выполнить нужное действие: ";

        cin >> sw1;
        while (cin.get() != '\n') { sw1 = ' '; };

        switch (sw1)
        {
        case '1': //[1] Сортировать рёбра по
весе
            if (!graph.empty()) { vectorSizeSort(graph); }
            else { cout << "Ошибка! Создайте
граф, прежде чем сортировать его.\n"; }
            break;
        case '2': //[2] Сортировать рёбра по
вершинам
            if (!graph.empty()) { vectorVertSort(graph); }
            else { cout << "Ошибка! Создайте
граф, прежде чем сортировать его.\n"; }
            break;
        case '3': //[3] Сравнение различных
сортировок по скорости
            if (!graph.empty()) { compareSort(graph, N); }
            else { cout << "Ошибка! Создайте
граф, прежде чем сортировать его.\n"; }
            break;
        case '0': //[0] Назад
            break;
        default:
            cout << "Ошибка! Пожалуйста, по
пробуйте снова\n";
            check1 = true; //цикл пойдёт заново
            break;
        }
    } while (check1);
    break;

    case '3': //[3] Поиск минимального остовно
го дерева
        do {
            check1 = false;
            sw1 = ' ';

```

```

        cout << "\n\x1b[32m[1]\x1b[0m Н а й т и м и н и м а
льное о с т о в н о е д е р е в о и с о х р а н и т ь е г о\n";
        cout << "\x1b[32m[2]\x1b[0m О п р е д е л и т ь в р
е м я н а х о ж д е н и я м и н и м а л ь н о г о о с т о в н о г о д е р
е в а б е з с о х р а н е н и я\n";
        cout << "\x1b[32m[0]\x1b[0m В е р н у т ь с я н а з
а д\n";

        cout << "П о ж а л у й с т а , в в е д и т е ч и с л
о , ч т о б ы в ы п о л н и т ь н у ж н о е д е й с т в и е : ";

        cin >> sw1;
        while (cin.get() != '\n') { sw1 = ' '; };

        switch (sw1)
        {
        case '1': //[1] Н а й т и м и н и м а л ь н о е о с
товное д е р е в о и с о х р а н и т ь е г о
            if (!graph.empty())
            {
                searchTree(graph, N);
                cout << "М и н и м а л ь н о е о с т о в
ное д е р е в о : \n";

                outputGraph(graph, N);
            }
            else { cout << "О ш и б к а ! Г р а ф п у с т
.\n"; }

            break;
        case '2': //[2] О п р е д е л и т ь в р е м я н а х
о ж д е н и я
            if (!graph.empty()) { searchTreeTime(graph,
N); }

            else { cout << "О ш и б к а ! Г р а ф п у с т
.\n"; }

            break;
        case '0': //[0] Н а з а д
            break;
        default:
            cout << "О ш и б к а ! П о ж а л у й с т а , п о
п р о б у й т е с н о в а\n";
            check1 = true; //цикл пойдёт заново

            break;
        }
    } while (check1);
    break;

    case '4': //[4] В ы в е с т и с п и с о к р ё б е р
        if (!graph.empty()) { outputEdgeList(graph); }
        else { cout << "О ш и б к а ! Г р а ф п у с т .\n"; }

```

```

        break;

    case '5': //[5] Вывод текущего графа
        if (!graph.empty()) { outputGraph(graph, N); }
        else { cout << "Ошибка! Граф пуст.\n"; }
        break;

    case '0': //[0] Закрывать программу
        cout << "Выход из программы...\n";
        check = false; //выход из цикла
        break;
    default: //в случае, если введено что-то и
но е
        cout << "Ошибка! Пожалуйста, попробуйте
те снова\n";
        break;
    }

} while (check);

return 0;

}

```