**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Российский химико-технологический университет имени Д.И. Менделеева»**

**Кафедра информационных компьютерных технологий**

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5**

Выполнил студент группы КС-30 Лобачев Дмитрий Сергеевич

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/DSLobachev\_30/blob/main/Algorithms/Laba5/Laba5.cpp

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 03.04.2023

**Оглавление**

[Описание задачи 3](#_Toc131331759)

[Описание метода/модели 4](#_Toc131331760)

[Выполнение задачи. 5](#_Toc131331761)

[Заключение 13](#_Toc131331762)

# Описание задачи

1. Создайте взвешенный граф, состоящий из [10, 20, 50, 100] вершин.

* Каждая вершина графа связана со случайным количеством вершин, минимум с [3, 4, 10, 20].
* Веса ребер задаются случайным значением от 1 до 20.
* Каждая вершина графа должна быть доступна, т.е. до каждой вершины графа должен обязательно существовать путь до каждой вершины, не обязательно прямой.

1. Выведите получившийся граф в виде матрицы смежности. Пример вывода данных: Матрица смежности
2. Для каждого графа требуется провести серию из 5 - 10 тестов, в зависимости от времени затраченного на выполнение одного теста, необходимо найти кратчайшие пути между всеми вершинами графа и их длину с помощью алгоритма Краскала.
3. В рамках каждого теста, необходимо замерить потребовавшееся время на выполнение задания из пункта 3 для каждого набора вершин. По окончанию всех тестов необходимо построить график используя полученные замеры времени, где на ось абсцисс (Х) нанести N – количество вершин, а на ось ординат(Y) - значения затраченного времени.

# Описание метода/модели

Алгоритм Краскала - эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа, результатом которого является набор ребер, которые позволяют пройтись по всему графу с минимальным суммарным весом.

Работа алгоритма:

1) Для алгоритма требуется получить вектор ребер, в котором каждое ребро будет описано своими вершинами и весом. Данный вектор сортируется по весам в порядке возрастания.

2) Осуществляется проход по всем ребрам таким образом, что если в нашем остовном дереве еще не была добавлена вершина, то мы добавляем данное ребро в остовное дерево. Таким образом мы получаем остовное дерево в виде нескольких подграфов.

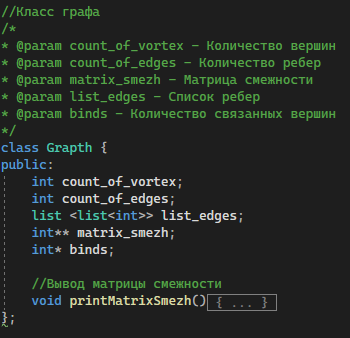
3)Последним пунктом мы соединяем все подграфы в единое остовное дерево таким образом, чтобы использовать ребро с наименьшим возможным весом.

# Выполнение задачи.

Для реализации данного метода сортировки использовался язык программирования C++.

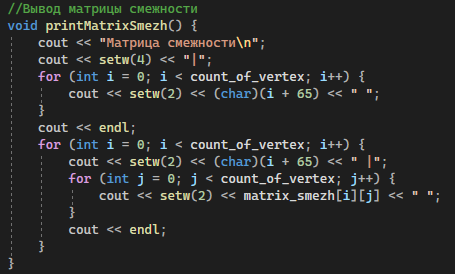
**Граф**

1. Класс стека имеет поля: количество вершин(count\_of\_vertex), количество ребер(count\_of\_edges), матрица смежности(matrix\_smezh), список ребер(list\_edges) и количество связей(binds).



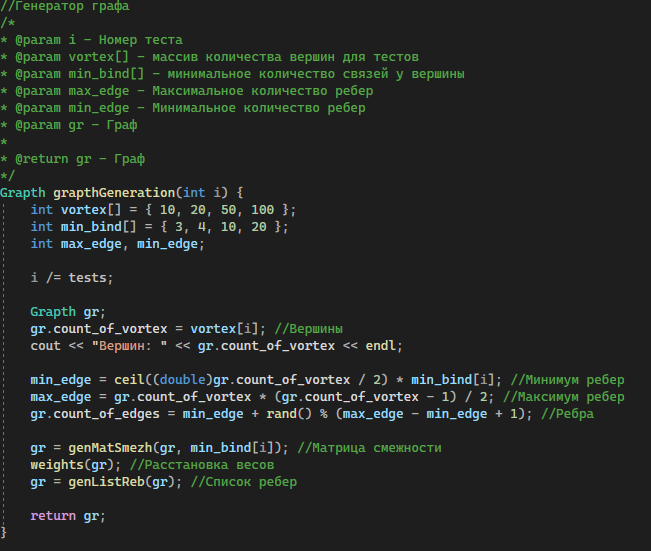
1. Класс графа имеет функцию:

* Функция вывода матрицы смежности printMatrixSmezh()

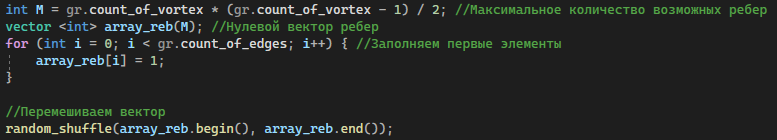


**Генерация графа**

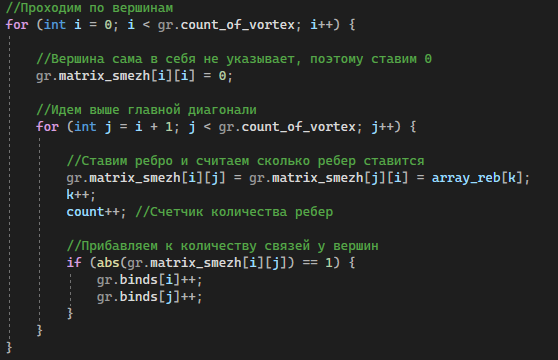
1. Был реализован генератор графов grapthGeneration(), в котором генерируются количество вершин, ребер, максимальное количество связей, максимальное количество входящих и выходящих ребер.



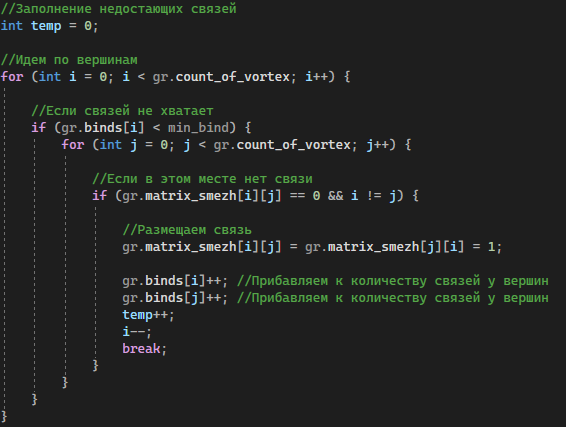
1. Функция genMatSmezh() генерирует матрицу смежности для определенного графа. Сперва генерируется вектор из M элементов (где М – максимальное количество ребер), первые N элементов которого равны 1 (где N – количество ребер в графе), а остальные равны 0. Далее этот вектор перемешивается для случайной расстановки функцией random\_shuffle



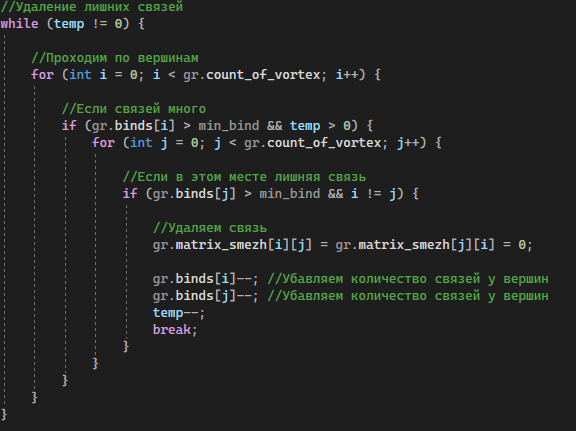
Следующим шагом заполняется матрица смежности так, что элементы вектора располагались выше главной диагонали (в исследуемых графах не существует ребер, входящие в исходящую вершину). Ниже главной диагонали зеркально отображаются эти же элементы. Параллельно этому процессу идет проверка на максимальное количество связей у вершины.



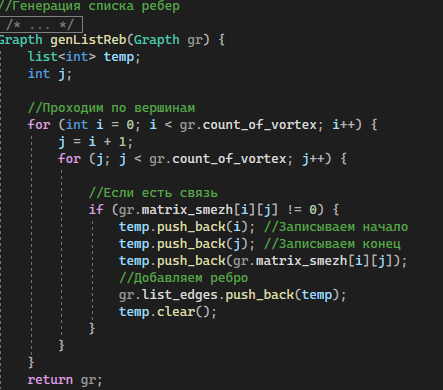
После расстановки всех ребер, выставляются те ребра, которые нужны для выполнения условия о минимальном количестве связей у ребра



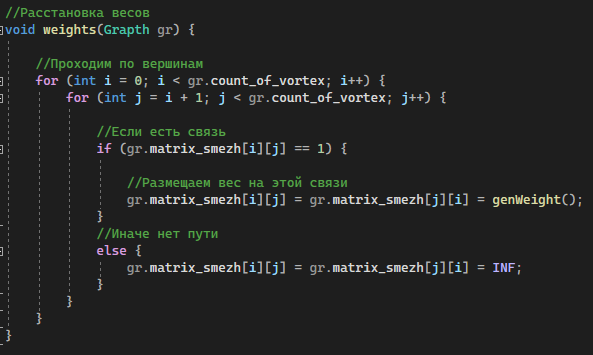
Следующим шагом мы удаляем лишние связи у других вершин.

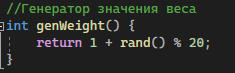


1. Функция genListReb() генерирует список ребер на основе матрицы смежности. В буферный список temp записываются начальная(i) и конечная(j) вершины ребра.



1. Функция weights() проставляет случайный вес всем существующим связям.

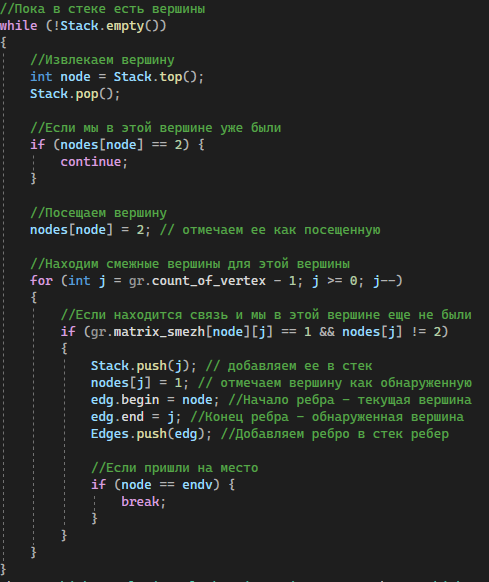


****

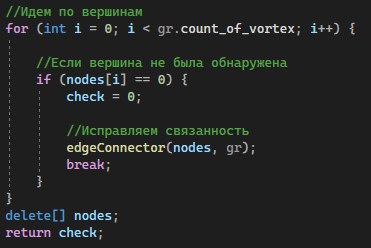
1. Функция edgeConnector() делает граф связным, а именно связывает вершины. Находятся 2 смежные и 2 не смежные вершины и они связываются.



1. Функция DFS() осуществляет поиск в глубину.
2. Создается массив размером равным количеству вершин. Изначально все элементы равны 0. Данный массив будет показывать нам посетили ли мы ту или иную вершину или нет.
3. Генерируем начальную и конечную вершины.
4. Помещаем начальную вершину в стек.
5. Извлекаем верхний элемент из стека и отмечаем его как посещенный.
6. Проверяем для взятой вершины все смежные и не обнаруженные вершины. Если такие имеются, то добавляем ее в стек и помечаем как обнаруженную.
7. Повторяем с 4 пункта, пока стек не будет пустым.

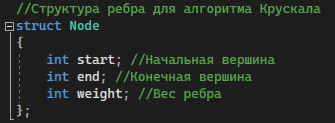


Если какая то из вершин не была обнаружена, то мы связываем её с другими.

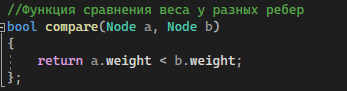


**Выполнение работы**

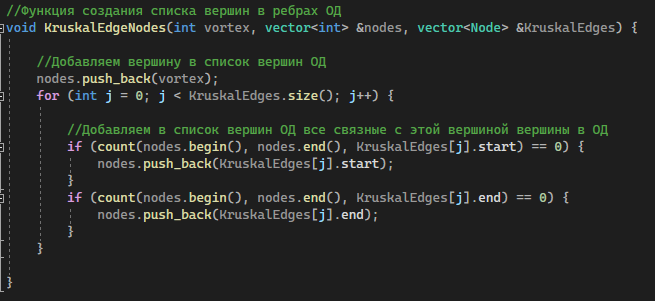
* + - 1. Была написана структура Node для хранения ребер



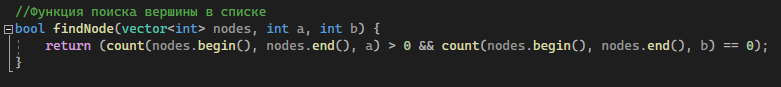
* + - 1. Была написана функция compare(), позволяющая сравнивать значения веса у ребер



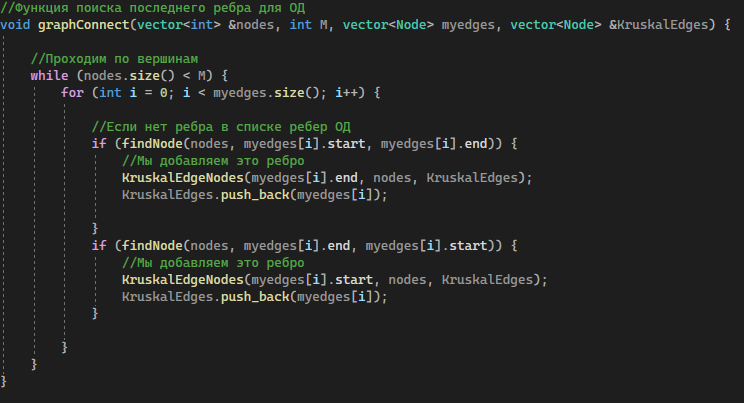
* + - 1. Была написана функция KruskalEdgeNodes(), которая заполняет вектор вершин, которые уже попали в остовное дерево по алгоритму Крускала



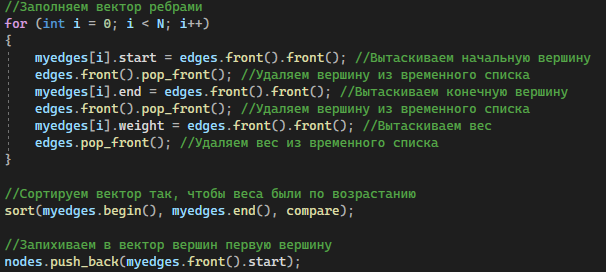
* + - 1. Была написана функция поиска вершин в списке вершин



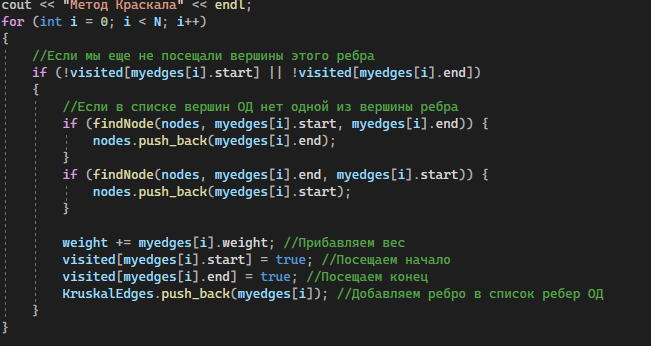
* + - 1. Была написана функция graphConnect(), связывающая все полученные подграфы в одно единое остовное дерево



* + - 1. Была написана функция Kruskal(), которая выполняет сам алгоритм Крускала для графа. Сперва мы заполняем вектор всеми ребрами графа и сортируем его в порядке увеличения веса ребра. Далее мы добавляем первую вершину верхнего ребра в вектор вершин, уже описанных в ОД.



Следующим шагом мы проходимся по всем ребрам и смотрим, просматривали ли мы вершины данного ребра, и если мы еще не смотрели одну из этих вершин, то добавляем данное ребро в список ребер ОД и помечаем вершины как просмотренные.



Таким образом мы получаем несколько подграфов, которые необходимо связать для получения полного остовного дерева. Дальнейшим шагом мы как раз ищем ребра, которыми могли бы связать эти подграфы.



**Тестирование**

Было проведено 10 тестов с разным количеством вершин (10, 20, 50, 100) и разным количеством минимальных связей у вершины (3, 4, 10, 20).

Для каждого теста было замерено время на выполнение алгоритма Крускала в миллисекундах. Таким образом было получены следующие результаты:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тесты | 10 | 20 | 50 | 100 |
| 1 | 1,9654 | 3,0567 | 8,3797 | 32,5458 |
| 2 | 2,2071 | 2,1592 | 11,7196 | 38,1789 |
| 3 | 1,38 | 2,3049 | 10,7006 | 44,1824 |
| 4 | 1,7464 | 2,9131 | 7,9992 | 39,342 |
| 5 | 1,4978 | 3,2259 | 13,1946 | 37,8154 |
| 6 | 1,4262 | 2,7844 | 9,3369 | 34,332 |
| 7 | 1,6397 | 3,1554 | 13,4637 | 33,5331 |
| 8 | 1,7285 | 2,1457 | 13,274 | 36,4638 |
| 9 | 1,4681 | 2,6238 | 12,5904 | 40,3085 |
| 10 | 1,5073 | 2,9433 | 14,068 | 40,4645 |
| Худшее | 2,2071 | 3,2259 | 14,068 | 44,1824 |
| Лучшее | 1,38 | 2,1457 | 7,9992 | 32,5458 |
| Среднее | 1,65665 | 2,73124 | 11,47267 | 37,71664 |

Был построен график времени от количества вершин для каждого теста:

А также был построен график лучшего, среднего и худшего случаев в зависимости от количества вершин у графа:

# Заключение

Проанализировав графики, можно заметит, что на скорость выполнения алгоритма Крускала влияет количество вершин в графе. Данный алгоритм позволяет нам определить минимальные затраты на полный обход графа. Таким образом, в результате мы имеем наименьшие затраты для прохода по этому графу, что может использоваться во многих разных задачах.

