МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики

Кафедра проблем управления

КУРСОВАЯ РАБОТА

по дисциплине
«Объектно-ориентированное программирование»

Тема курсовой работы
«Построение минимального остовного дерева»

Студент группы КМБО-04-19 *Пашшоев Б.*

Руководитель курсовой работы *Петрусевич Д.А.*

Работа представлена к защите «__»_______20___ г. *(подпись студента)*

«Допущен к защите» «__»_______20___ г. *(подпись руководителя)*

МОСКВА — 2020

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт кибернетики

Кафедра проблем управления

Утверждаю

Заведующий
кафедрой_____________*М.П.Романов*

«____» __________20__г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Пашшоев Б.* Группа *КМБО-04-19*

1. **Тема: «Построение минимального остовного дерева»**
2. **Исходные данные:**

Реализовать необходимый набор классов для представления ребра, чтения информации о ребрах из файла

Произвести выбор эффективной структуры данных для хранения ребер / вершин

Реализовать необходимый набор классов и методов для минимального остовного дерева

Реализовать перебор ребер и поиск минимального ребра, подходящего для добавления в дерево

Реализовать процесс построения минимального остовного дерева

Минимальный набор рассматриваемых алгоритмов: алгоритм Прима, алгоритм Крускала

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:

Продемонстрировать работу алгоритма Прима, построив минимальное остовное дерево
Продемонстрировать работу алгоритма Крускала, построив минимальное остовное дерево
Обосновать выбор структуры данных для хранения ребер / вершин
По построенному дереву построить путь между двумя произвольными пунктами

4. Срок представления к защите курсовой работы: до « 31» декабря 2020 г.

Задание на курсовую работу выдал	«01» октября 2020 г.	________________	*(____________)*
Задание на курсовую работу получил	«01» октября 2020 г.	________________	*(____________)*

Оглавление

Введение

Минимальным остовным деревом (MST) связного взвешенного графа называется его связный подграф, состоящий из всех вершин исходного графа и некоторых его ребер, причем сумма весов ребер минимально возможная. Нахождение такого дерева встречается в многих прикладных задачах и решение этой задачи позволяет минимизировать затраты на связь компонент принятых за вершины графа.

Примером такой задачи может являться поиск способа соединения городов дорогами так чтобы их общая длинна или стоимость была минимальной. Также задача поиска минимального островного дерева возникает в компьютерных сетях, а именно в протоколе STP, который устраняет петли в сети выбирая при этом лучшие по скорости соединения.

Актуальность данной работы заключается в том, что появляются новые области, где возникает задача поиска MST и сложно определить по получившемуся графу какой алгоритм решит эту задачу оптимально. Объектом исследования процесс выбора оптимального алгоритма поиска минимального остовного дерева для графа определенного вида.

Предметом исследования являются различные алгоритмы поиска минимального остовного дерева

Цель работы: оценка производительности на различных графах однопоточных и многопоточные алгоритмы поиска минимального остовного дерева.

Работа состоит из введения, четырех глав, заключения и списка литературы и приложения. Введение отражает основные характеристики работы определяет тему работы и ее актуальность, описывает объект и предмет исследования, цели и задачи, которые необходимо рассмотреть в настоящем документе

В первой главе дан необходимый минимум знаний про графы и способы работы с ним

Во второй главе рассмотрена задача поиска минимального остовного дерева и алгоритмы решения этой задачи

В третей главе рассказан как работает программа и обзор функций и классов

В четвертой главе показан несколько примеров работы алгоритмов

1.Теоретический обзор. Графы.MST

1.1 Что такое граф?

Граф, или неориентированный граф G это упорядоченная пара G=(V,E) ,где V— непустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых рёбрами. А с точки зрения компьютерных наук и дискретной математики, графы—это абстрактный способ представления типов отношений, например дорог, соединяющих города, и других видов сетей

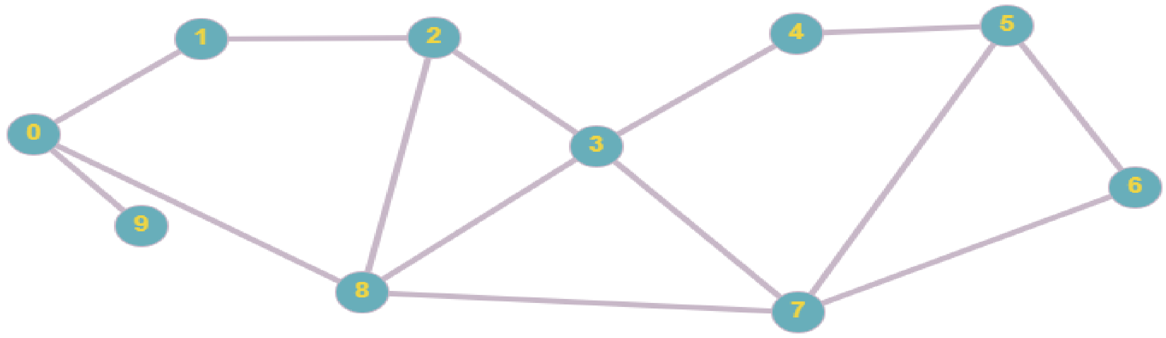


Рисунок 1. Неориентированный граф

Важные определения:

- **Вершины и рёбра** графа называются также элементами графа, число вершин в графе **|V| — порядком**, число рёбер **|E| — размером графа**.
- Вершины **u** и **v** называются **концевыми вершинами** (или просто концами) ребра **e={u,v}**. Ребро, в свою очередь, соединяет эти вершины. Две концевые вершины одного и того же ребра называются **соседними**.
- Две вершины называются **смежными**, если они являются концами одного ребра
- Два ребра называются **кратными**, если множества их концевых вершин совпадают.
- Ребро называется **петлёй**, если его концы совпадают, то есть **e={v,v}**.
- Граф без петель и кратных рёбер называется **простым**.
- **Степенью degV** вершины V называют количество инцидентных ей рёбер (при этом петли считают дважды).
- Вершина называется **изолированной**, если она не является концом ни для одного ребра;
- **Висячей** (или листом), если она является концом ровно одного ребра.

- **Путь**—последовательность рёбер, соединяющая разные (неповторяющиеся) вершины;
- **Маршруты**—это те же пути, только они не требуют последовательности разных вершин;
- **Цикл**—группа вершин, связанных вместе в замкнутую цепь. На рисунке выше вершины [1,2,4] составляют цикл;
- **Связный граф**—граф, в котором между любой парой вершин имеется один путь;
- **Дерево**—связный граф, не содержащий цикла;

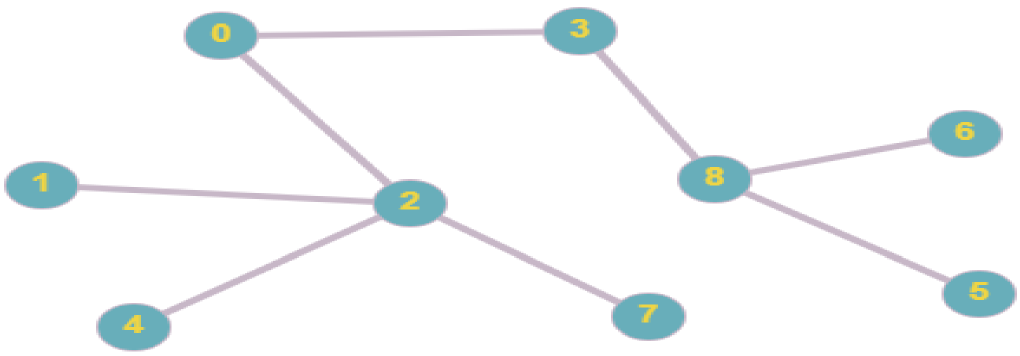


Рисунок 2. Дерево

- **Ориентированный граф**—граф, в котором рёбра имеют направления и обозначаются стрелками. В таком ориентированном графе можно перемещаться вдоль ребра только в указанном направлении.

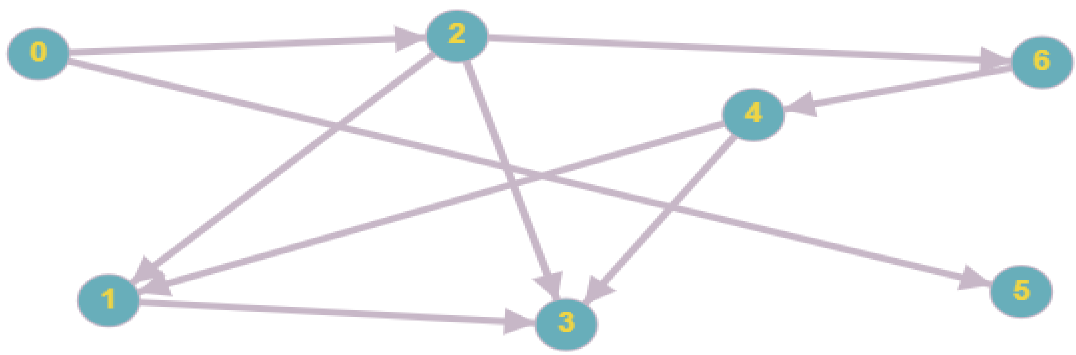


Рисунок 3. Ориентированный связанный граф

- **Взвешенный граф** — это граф, дугам которого поставлены в соответствие веса, так что дуге *(x_{if} xj)* сопоставлено некоторое число *c(x_{jy} Xj)* = называемое *длиной* (или *весом*, или *стоимостью*) дуги. Обычный (не взвешенный) граф можно интерпретировать как взвешенный, все ребра которого имеют одинаковый вес 1.

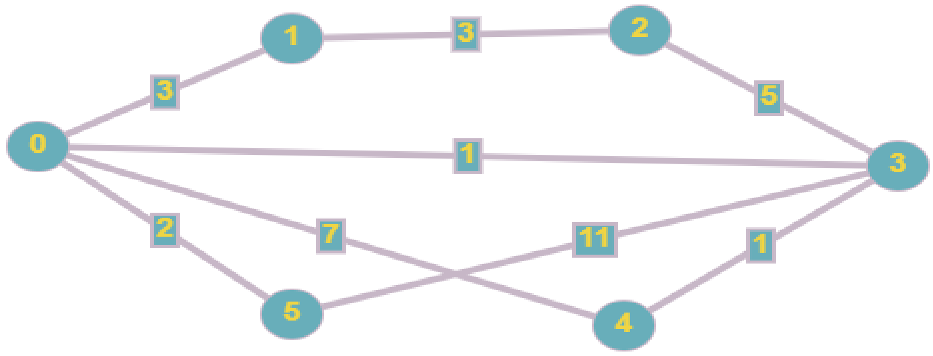


Рисунок 4. Взвешенный связанный граф

- **Длина пути** во взвешенном графе — это сумма длин (весов) тех ребер, из которых состоит путь.
- **Расстояние между вершинами** — это длина кратчайшего пути.

1.1. Способы хранения и работы с графом

Граф может быть представлен (сохранен) несколькими способами:

- матрица смежности;
- матрица инцидентности;
- список смежности (инцидентности);
- список ребер.

Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Размер массива зависит от количества вершин и/или ребер в конкретном графе

Матрица смежности графа — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа.

- 0 – соответствует отсутствию ребра,
- 1 – соответствует наличию ребра

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

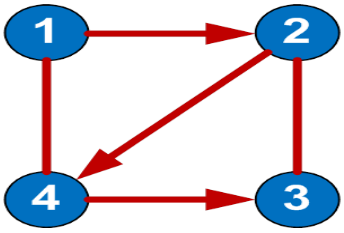


Рисунок 5. Матрица смежности и соответствующий граф

Матрица инцидентности (инциденции) графа — это матрица, количество строк в которой соответствует числу вершин, а количество столбцов – числу рёбер. В ней указываются связи между инцидентными элементами графа (ребро(дуга) и вершина). В неориентированном графе если вершина инцидентна ребру то соответствующий элемент равен 1, в противном случае элемент равен 0.В ориентированном графе если ребро выходит из вершины, то соответствующий элемент равен 1, если ребро входит в вершину, то соответствующий элемент равен -1, если ребро отсутствует, то элемент равен 0.

	1	2	3	4	5
1	1	0	0	1	0
2	-1	1	0	0	1
3	0	1	-1	0	0
4	0	0	1	1	-1

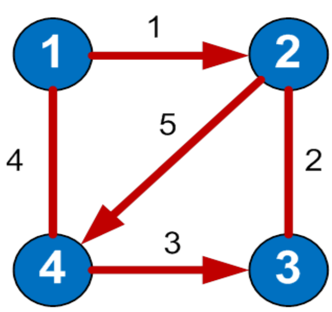


Рисунок 6. Матрица инцидентности и соответствующий граф

Список смежности (инцидентности). Если количество ребер графа по сравнению с количеством вершин невелико, то значения большинства элементов матрицы смежности будут равны 0. При этом использование данного метода нецелесообразно. Для подобных графов имеются более оптимальные способы их представления. По отношению к памяти списки смежности менее требовательны, чем матрицы смежности. Такой список можно представить в виде таблицы, столбцов в которой – 2, а строк — не больше, чем вершин в графе. В каждой строке в первом столбце указана вершина выхода, а во второмстолбце – список вершин, в которые входят ребра из текущей вершины.

1	2, 4
2	3, 4
3	2
4	1, 3

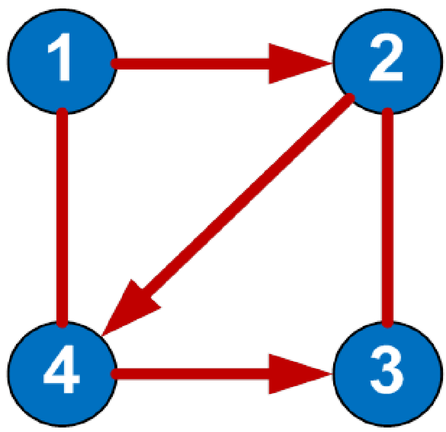


Рисунок 7. Список смежности и соответствующий граф

Преимущества списка смежности:

- Рациональное использование памяти.
- Позволяет быстро перебирать соседей вершины.
- Позволяет проверять наличие ребра и удалять его.

Недостатки списка смежности:

- При работе с насыщенными графами (с большим количеством рёбер) скорости может не хватать.
- Нет быстрого способа проверить, существует ли ребро между двумя вершинами.
- Количество вершин графа должно быть известно заранее.

- Для взвешенных графов приходится хранить список, элементы которого должны содержать два значащих поля, что усложняет код:
    - номер вершины, с которой соединяется текущая;
    - вес ребра.

Список рёбер. В списке рёбер в каждой строке записываются две смежные вершины и вес соединяющего их ребра (для взвешенного графа). Количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.

	Начало	Конец	Вес
1	1	2	
2	1	4	
3	2	3	
4	2	4	
5	3	2	
6	4	1	
7	4	3	

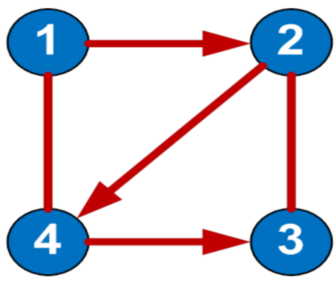


Рисунок 8. Список ребер и соответствующий граф

Какой способ представления графа лучше? Ответ зависит от отношения между числом вершин и числом рёбер. Число ребер может быть довольно малым (такого же порядка, как и количество вершин) или довольно большим (если граф является полным). Графы с большим числом рёбер называют ***плотными***, с малым — ***разреженными***. Плотные графы удобнее хранить в виде матрицы смежности, разреженные — в виде списка смежности.

1.2. Построения минимального остовного дерева

1.2.1 Постановка задачи

Представим, что есть некая компьютерная сеть, соединяющая различное вычислительное оборудование. Каждое соединение имеет свою скорость, с которой по нему могут передаваться данные. Для корректной работы этой сети требуется отсутствие в ней циклов. Наша задача состоит в том, чтобы каждый вычислительный узел был доступен в сети, и суммарная скорость была максимальной. Такую задачу можно описать как задачу поиска максимального остовного дерева, где узлы — это вычислительное оборудование, а ребра соединение между оборудованием с весом равному скорости соединения. Знак веса ребра мы заменим на противоположный и получим задачу поиска минимального остовного дерева. Далее мы увидим почему мы можем так сделать.

Сформулируем задачу в общем виде. Пусть дан связанный, взвешенный, неориентированный граф G=(V,E) , где V – множество вершин, а E – множество ребер этого графа. Для каждого ребра задан вес w(u,v), задающий стоимость соединения вершины u и v . Задача состоит в нахождении такого подграфа T , который соединяет все вершины и общий вес ребер минимален.

Свойства минимального остова

- Минимальный остов **уникален, если веса всех рёбер различны**. В противном случае, может существовать несколько минимальных остовов (конкретные алгоритмы обычно получают один из возможных остовов).
- Минимальный остов является также и **остовом с минимальным произведением** весов рёбер.(доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)
- Минимальный остов является также и **остовом с минимальным весом самого тяжелого ребра**.(это утверждение следует из справедливости алгоритма Крускала)
- **Остов максимального веса** ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова

Общая схема работы алгоритмов построения минимального остовного дерева имеет следующий вид. Существует связанный неориентированный взвешенный граф G и мы хотим найти минимальное остовное дерево для этого графа. Искомый граф получается из этого графа каждый раз добавлением одного ребра пока не получится Минимальное остовное дерево.

Далее рассмотрим пару конкретных алгоритмов построения минимального остовного дерева.

1.2.2 Обзор алгоритма Прима

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.Алгоритм Прима обладает тем свойством, что выбранные ребра всегда образуют связанное дерево.

Сам **алгоритм** имеет очень простой вид. Искомый минимальный остов строится постепенно, добавлением в него рёбер по одному. Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно). Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов. После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины (или, что то же самое, $n-1$ ребро).

В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше $n-1$).

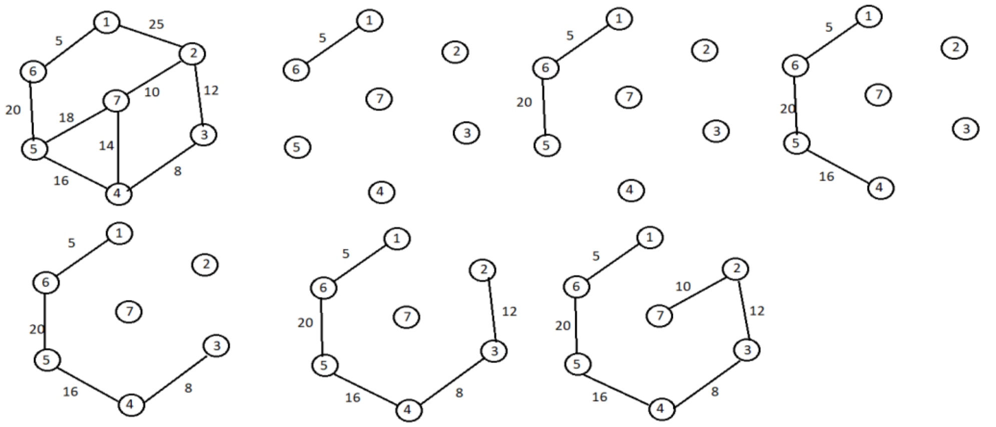


Рисунок 9. Исходный граф и результаты итерации алгоритма Прима

1.2.3 Обзор алгоритма Крускала

Алгоритм Крускала - 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. Данный алгоритм был описан Крускалом (Kruskal) в 1956 г.

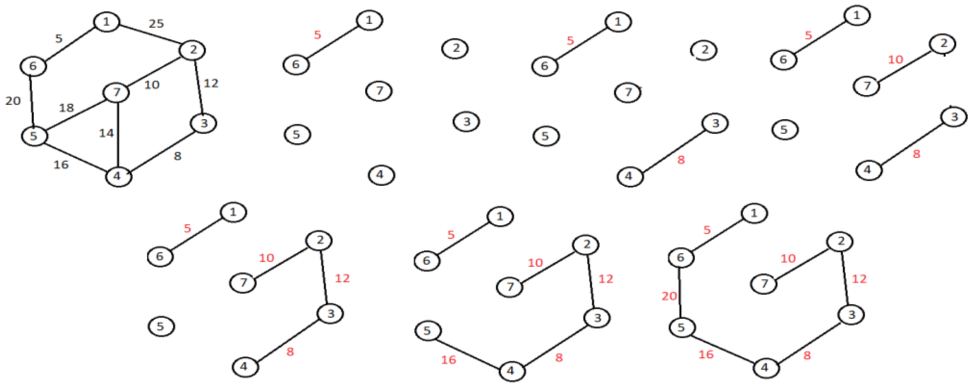


Рисунок 10. Исходный граф и результаты итерации алгоритма Крускала

1.2.4 Обзор алгоритма Борувки

Алгоритм Борувки – это алгоритм поиска минимального остовного дерева в взвешенном связанном неориентированном графе. Впервые был опубликован в 1926 году Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии.

Алгоритм Борувки представляет граф как лес поддеревьев MST. На первом этапе каждая вершина принадлежит отдельному дереву. Далее каждое дерево выбирает минимальное ребро соединяющие текущее дерево с другим. Если минимальных ребер несколько выбирается ребро с наименьшим порядковым номером это очень важно для правильной работы алгоритма. На следующем этапе все выбранные ребра добавляются в MST и алгоритм повторяется до тех пор, пока не останется одно дерево. Оставшееся дерево и будет искомым минимальным остовным деревом [5].

Опишем модель алгоритма. Пусть дан связный неориентированный граф G=(V,E), где– V множество вершин, а E– множество ребер. Вес ребра обозначим w(e) , а искомое дерево T=(R,A).

1.Изначально имеем множество поддеревьев MST Comp={C1,C2,…Cn}, где Ci содержит вершину i .

2. Для каждого поддерева из множества Comp выбирается ребро соединяющие текущую компоненту с другой и имеющие наименьший вес. Если веса равны, то выбираем ребро с наименьшим номером.

3. Компоненты объединяются и из оставшихся компонент формируется новое множество Comp.

4. Если |Comp| > 1 перейти к шагу 2 иначе к шагу 5.

5. Минимальный остовным деревом будет являть дерево C1.

На каждой итерации алгоритма Борувки число поддеревьев сокращается как минимум вдвое. Следовательно алгоритм в худшем случае выполнит O(log V) итераций. На каждой итерации мы в худшем случае просмотрим все ребра. Получается итоговая оценка алгоритма Борувки O(E*log V) .

2.Реализация алгоритмов построение MST

2.1 Подходы к реализации алгоритма Прима

Время работы алгоритма существенно зависит от того, каким образом мы производим поиск очередного минимального ребра среди подходящих рёбер. Здесь могут быть разные подходы, приводящие к разным асимптотикам и разным реализациям.

Тривиальная реализация: алгоритмы за O(n*m) и O(n^2*logn)

Если искать каждый раз ребро простым просмотром среди всех возможных вариантов, то асимптотически будет требоваться просмотр O(m) рёбер, чтобы найти среди всех допустимых ребро с наименьшим весом. Суммарная асимптотика алгоритма составит в таком случае O(n*m)

Этот алгоритм можно улучшить, если просматривать каждый раз не все рёбра, а только по одному ребру из каждой уже выбранной вершины. Для этого, например, можно отсортировать рёбра из каждой вершины в порядке возрастания весов, и хранить указатель на первое допустимое ребро (напомним, допустимы только те рёбра, которые ведут в множество ещё не выбранных вершин). Тогда, если пересчитывать эти указатели при каждом добавлении ребра в остов, суммарная асимптотика алгоритма будет O(n^2+m), но предварительно потребуется выполнить сортировку всех рёбер за O(m*logn), что в худшем случае (для плотных графов) даёт асимптотику O(n^2*logn).

Ниже мы рассмотрим два немного других алгоритма: для плотных и для разреженных графов, получив в итоге заметно лучшую асимптотику.

Случай плотных графов: алгоритм за $O(n^2)$

Подойдём к вопросу поиска наименьшего ребра с другой стороны: для каждой ещё не выбранной будем хранить минимальное ребро, ведущее в уже выбранную вершину.

Тогда, чтобы на текущем шаге произвести выбор минимального ребра, надо просто просмотреть эти минимальные рёбра у каждой не выбранной ещё вершины — асимптотика составит O(n).

Но теперь при добавлении в остов очередного ребра и вершины эти указатели надо пересчитывать. Заметим, что эти указатели могут только уменьшаться, т.е. у каждой не просмотренной ещё вершины надо либо оставить её указатель без изменения, либо присвоить ему вес ребра в только что добавленную вершину. Следовательно, эту фазу можно сделать также за O(n).

Таким образом, мы получили вариант алгоритма Прима с асимптотикой $O(n^2)$.

Случай разреженных графов: алгоритм за O(m*logn)

В описанном выше алгоритме можно увидеть стандартные операции нахождения минимума в множестве и изменение значений в этом множестве. Эти две операции являются классическими, и выполняются многими структурами данных, например, реализованным в языке C++ красно-чёрным деревом set.

По смыслу алгоритм остаётся точно таким же, однако теперь мы можем найти минимальное ребро за время O(logn). С другой стороны, время на пересчёт n указателей теперь составит O(n*logn), что хуже, чем в вышеописанном алгоритме.

Если учесть, что всего будет O(m) пересчётов указателей и O(n) поисков минимального ребра, то суммарная асимптотика составит O(m*logn) — для разреженных графов это лучше, чем оба вышеописанных алгоритма, но на плотных графах этот алгоритм будет медленнее предыдущего.

Код алгоритма смотреть на «Приложения – Код алгоритма Прима»

2.2 Подходы к реализации алгоритма Крускала

Мы начинаем с ребер с наименьшим весом, проверяем не получится ли цикл с помощью системы не пересекающихся множеств – если нет, то добавляем ребро в ответ. Так продолжаем добавлять ребра, пока не достигнем нашей цели.

Тривиальная реализация: алгоритмы за O(n*m)

Если как и в алгоритме Прима каждый раз искать ребро простым просмотром среди всех возможных вариантов, то асимптотически будет требоваться просмотр O(m) рёбер, чтобы найти среди всех допустимых ребро с наименьшим весом. Суммарная асимптотика алгоритма составит в таком случае O(n*m)

Алгоритм за O (m*log n)

Если не искать каждый раз минимальное ребро с нуля а хранить ребра в структурах как Heap то можно улучшить алгоритм до O (m*log n).Код алгоритма смотреть на «Приложения – Код алгоритма Крускала»

2.3 Основное меню программы

2.3.1 Класс Graph

Я реализовал класс Graph чтобы было удобнее работать с матрицей смежности графа. Перегрузил необходимые операции: Индексация [], оператор Ввода из консоли/из файла, оператор Вывода в консоль/ в файл. Чтобы ввести матрицу смежности через консоль или файл нужно сначала ввести количество вершин N, потом ввести N*N чисел.

2.3.2 Класс Edge

Класс Edge реализован чтобы было удобнее хранить ребра графа, он хранит номера двух конечных вершин и вес самого ребра. Перегружены оператор сравнения <, также оператор Ввода из консоли/из файла и оператор Вывод в консоль/в файл. Чтобы ввести данные через консоль или файл нужно последовательно ввести 3 числа: начальная вершина, конечная вершина, весь ребра

2.4 Алгоритм нахождения пути между двумя вершинами

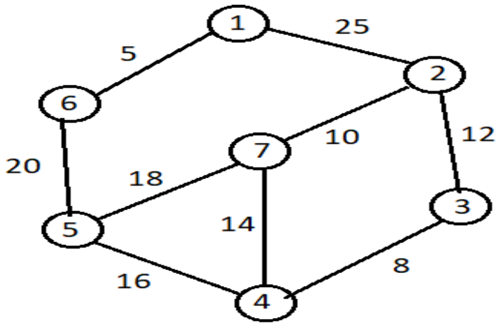
GetPathBFS(u,v) - Алгоритм построение пути между вершинами в дереве. Алгоритм основан на алгоритме Поиска в ширину. Начинается с начальной вершины и перебирает вершины пока не дойдет до конечной, при этом он хранит путь. Если он дойдет до конечной вершины Поиск в ширину заканчивается и возвращает путь который сохранил.

2.5. Примеры работы алгоритмов

В данной главе представлены примеры работы алгоритмов построение минимальное остовного дерева и построение пути между двумя произвольными вершинами в дереве

Дан граф и его матрица смежности построить минимальное остовное дерево и построить между двумя вершинами **u**, **v**

Пример 1.



0	**25**	**0**	**0**	**0**	**5**	**0**
25	**0**	**12**	**0**	**0**	**0**	**10**
0	**12**	**0**	**8**	**0**	**0**	**0**
0	**0**	**8**	**0**	**16**	**0**	**14**
0	**0**	**0**	**16**	**0**	**20**	**18**
5	**0**	**0**	**0**	**20**	**0**	**0**
0	**10**	**0**	**14**	**18**	**0**	**0**

Рисунок 11. Граф 1 и его матрица смежности

Результат 1.

Ниже приведен MST. Сумма весов равен 71. И также таблицы – результаты алгоритмов. Первые два столбца – номера вершин, третий столбец – веса между ними.

Таблица 2.1 – Результат алгоритма Прима. Мы видим что алгоритм последовательно добавляет минимальное ребро инцидентное к уже добавленным вершинам.

Таблица 2.2 – Результат алгоритма Крускала. Алгоритм последовательно добавляет минимальное ребро, но при условии, что добавленное ребро не образует цикл и если получится цикл – то пропускаем

Таблица 2.3 – Путь между вершинами 7,5.

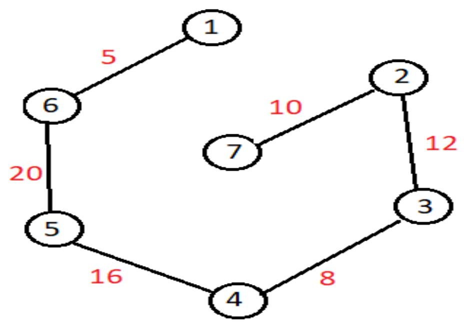


Рисунок 12. MST 1.

u	v	w
1	6	5
6	5	20
5	4	16
4	3	8
3	2	12
2	7	10

Таблица 2.1

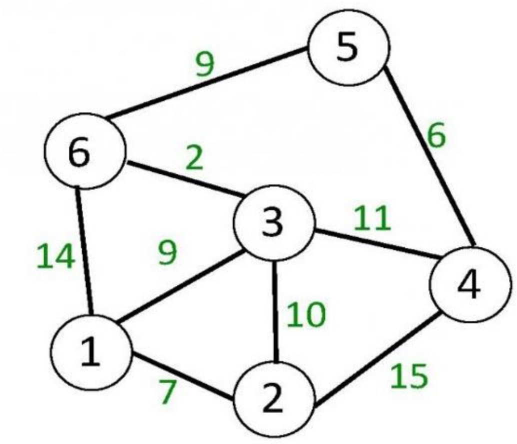
u	v	w
1	6	5
3	4	8
2	7	10
2	3	12
4	5	16
5	6	20

Таблица 2.2

7	5	
7	2	10
2	3	12
3	4	8
4	5	16

Таблица 2.3

Пример 2.



0	7	9	0	0	14
7	0	10	15	0	0
9	10	0	11	0	2
0	15	11	0	6	0
0	0	0	6	0	9
14	0	2	0	9	0

Рисунок 13. Граф 2 и его матрица смежности

Результат 2.

Ниже приведен MST 2. Сумма весов равен 33И также таблицы – результаты алгоритмов. Первые два столбца – номера вершин, третий столбец – веса между ними.

Таблица 2.4 – Результат алгоритма Прима. Мы видим что алгоритм последовательно добавляет минимальное ребро инцидентное к уже добавленным вершинам.

Таблица 2.5 – Результат алгоритма Крускала. Алгоритм последовательно добавляет минимальное ребро, но при условии, что добавленное ребро не образует цикл и если получится цикл – то пропускаем.

Таблица 2.6 – Путь между вершинами 4,2.

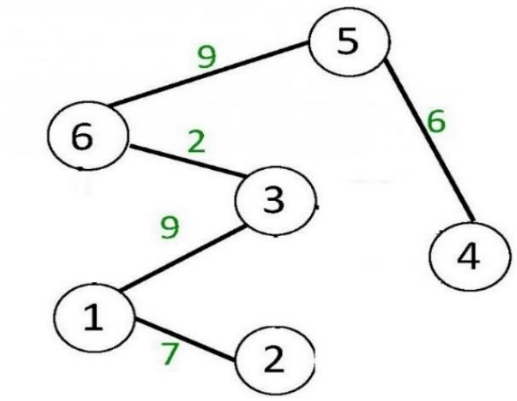


Рисунок 14. MST 2.

u	v	w
3	6	2
1	3	9
2	1	7
5	6	9
4	5	6

Таблица 2.4

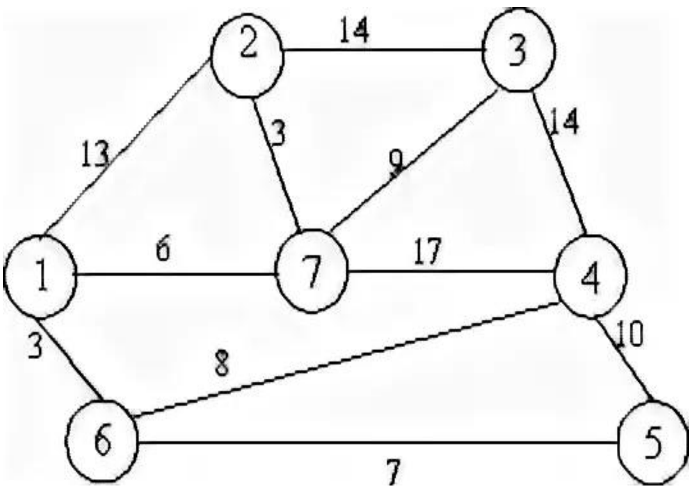
u	v	w
3	6	2
4	5	6
1	2	7
1	3	9
5	6	9

Таблица 2.5

4	2	
4	5	6
5	6	9
6	3	2
3	1	9
1	2	7

Таблица 2.6

Пример 3.



0	13	0	0	0	3	6
13	0	14	0	0	0	3
0	14	0	14	0	0	9
0	0	14	0	10	8	17
0	0	0	10	0	7	0
3	0	0	8	7	0	0
6	3	9	17	0	0	0

Рисунок 15. Граф 3 и его матрица смежности

Результат 3.

Ниже приведен MST. Сумма весов равен 36. И также таблицы – результаты алгоритмов. Первые два столбца – номера вершин, третий столбец – веса между ними.

Таблица 2.7 – Результат алгоритма Прима. Мы видим что алгоритм последовательно добавляет минимальное ребро инцидентное к уже добавленным вершинам.

Таблица 2.8 – Результат алгоритма Крускала. Алгоритм последовательно добавляет минимальное ребро, но при условии, что добавленное ребро не образует цикл и если получится цикл – то пропускаем

Таблица 2.9 – Путь между вершинами 2,4

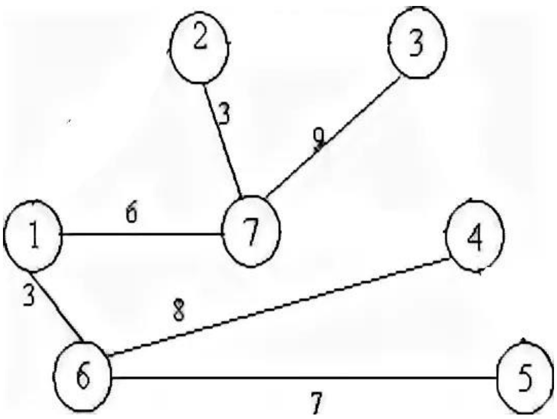


Рисунок 16. MST 3

u	v	w
1	6	3
7	1	6
2	7	3
5	6	7
4	6	8
3	7	9

Таблица 2.7

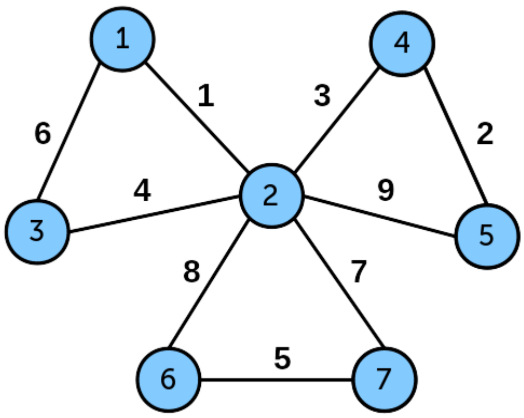
u	v	w
1	6	3
2	7	3
1	7	6
5	6	7
4	6	8
3	7	9

Таблица 2.8

2	4	
2	7	3
7	1	6
1	6	3
6	4	8

Таблица 2.9

Пример 4.



0	1	6	0	0	0	0
1	0	4	3	9	8	7
6	4	0	0	0	0	0
0	3	0	0	2	0	0
0	9	0	2	0	0	0
0	8	0	0	0	0	5
0	7	0	0	0	5	0

Рисунок 17. Граф 4 и его матрица смежности

Результат 4.

Ниже приведен MST. Сумма весов равен 22.И также таблицы – результаты алгоритмов. Первые два столбца – номера вершин, третий столбец – веса между ними.

Таблица 2.10 – Результат алгоритма Прима. Мы видим что алгоритм последовательно добавляет минимальное ребро инцидентное к уже добавленным вершинам.

Таблица 2.11 – Результат алгоритма Крускала. Алгоритм последовательно добавляет минимальное ребро, но при условии, что добавленное ребро не образует цикл и если получится цикл – то пропускаем.

Таблица 2.12 – Путь между вершинами 6,5.

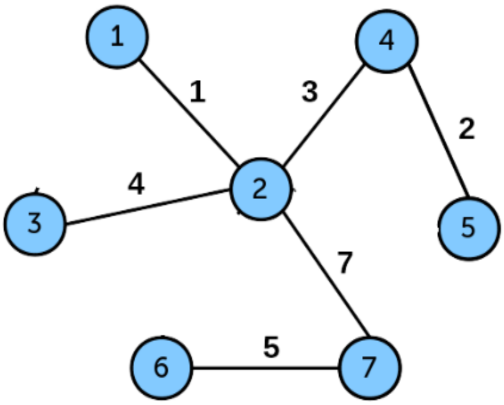


Рисунок 18. MST 4.

u	v	w
1	2	1
4	2	3
5	4	2
3	2	4
7	2	7
6	7	5

Таблица 2.10

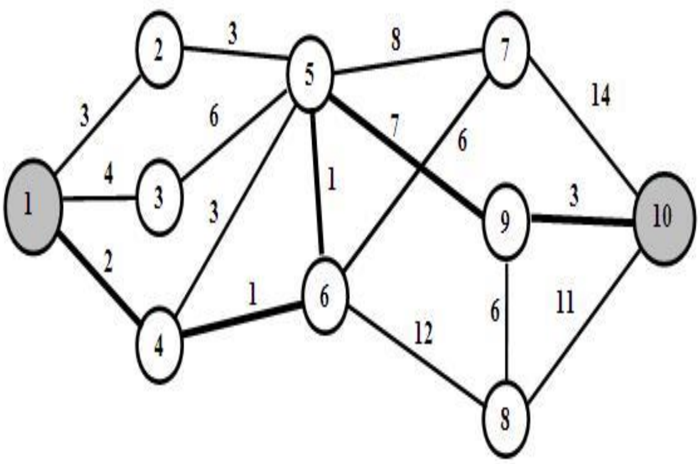
u	v	w
1	2	1
4	5	2
2	4	3
2	3	4
6	7	5
2	7	7

Таблица 2.11

6	5	
6	7	5
7	2	7
2	4	3
4	5	2

Таблица 2.12

Пример 5.



0	3	4	2	0	0	0	0	0	0
3	0	0	0	3	0	0	0	0	0
4	0	0	0	6	0	0	0	0	0
2	0	0	0	3	1	0	0	0	0
0	3	6	3	0	1	8	0	7	0
0	0	0	1	1	0	6	12	0	0
0	0	0	0	8	6	0	0	0	14
0	0	0	0	0	12	0	0	6	11
0	0	0	0	7	0	0	6	0	3
0	0	0	0	0	0	14	11	3	0

Рисунок 19. Граф 5 и его матрица смежности

Результат 5.

Ниже приведен MST. Сумма весов равен 33И также таблицы – результаты алгоритмов. Первые два столбца – номера вершин, третий столбец – веса между ними.

Таблица 2.13 – Результат алгоритма Прима. Мы видим что алгоритм последовательно добавляет минимальное ребро инцидентное к уже добавленным вершинам.

Таблица 2.14 – Результат алгоритма Крускала. Алгоритм последовательно добавляет минимальное ребро, но при условии, что добавленное ребро не образует цикл и если получится цикл – то пропускаем.

Таблица 2.15 – Путь между вершинами 10,1.

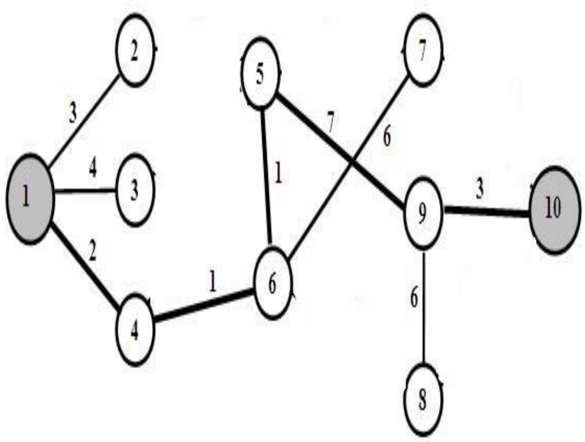


Рисунок 20. MST 5.

u	v	w
4	6	1
5	6	1
1	4	2
1	2	3
3	1	4
7	6	6
9	5	7
10	9	3
8	9	6

Таблица 2.13

u	v	w
4	6	1
5	6	1
1	4	2
1	2	3
9	10	3
1	3	4
6	7	6
8	9	6
5	9	7

Таблица 2.14

10	1	
10	9	3
9	5	7
5	6	1
6	4	1
4	1	2

Таблица 2.15

Заключение

В данный курсовой работе произведен анализ основных алгоритмов поиска минимального остовного дерева и рассмотрен алгоритм построение пути между двумя вершинами в дереве

Были проанализированы такие алгоритмы как алгоритм Крускала и алгоритм Прима. Каждый алгоритм имеет свои особенности и по-разному ведет себя на различных графах.

Так было выяснено, что алгоритм Крускала по сравнению с другими алгоритмами потребляет меньше памяти и показывает на разреженных графах лучший результат. Асимптотика работы алгоритма значительно зависит от используемой сортировки.

Алгоритм Прима отлично подходит для полных графов и выбирать его следует если граф хранится в виде списка смежности или матрицы смежности. Минусом алгоритма является высокое потребление памяти

Список использованной литературы

1. Кормен Т. Х. Алгоритмы. Построение и анализ /Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест. – М.: Вильямс, 2018. – 1328 с.

2. Берцун В.Н. Математическое моделирование на графах. / В.Н. Берцун. – Часть 2. – Томск: Изд-во Том. ун-та, 2014. – 86 с.

3. Скиена С. Алгоритмы. Руководство по разработке. / С. Скиена. – СПБ.: БХВ-Петербург, 2015. – 720 с.

4. Седжвик Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах. / Р. Седжвик. – СПБ.: ООО «ДиаСофтЮП». 2014. – 496 с.

5. Белоусов А.И. Дискретная математика. / А.И. Белоусов, С.Б. Ткачев. – М.: МГТУ, 2016. – 744с

Приложения

Код алгоритма Прима

```
vector<Edge> PrimsMST(Graph& G)
{
    int u, v, V = G.getN(), min = I;
    vector<int> near(V);
    vector<Edge> T(V - 1, 0);

    for (int i = 0; i < V; i++)// Находим мин ребро
    {
        near[i] = I;
        for (int j = i; j < V; j++)
        {
            if (G[i][j] < min)
            {
                min = G[i][j];
                u = i;
                v = j;
            }
        }
    }
    Edge e(u, v, min);
    T[0] = e;
    near[u] = near[v] = -1;
    // Иницализация массива ближайшим к i-той вершине вершиной
    // т.е.  near[i] - ближайшее к  i  вершина из тех которую мы уже
добавили в решение
    for (int i = 0; i < V; i++)
    {
        if (near[i] != -1)
        {
            if (G[i][u] < G[i][v])
            {
                near[i] = u;
            }
            else
            {
                near[i] = v;
            }
        }
    }

    for (int i = 1; i < V - 1; i++)
    {
        int k;
        min = I;
        for (int j = 0; j < V; j++)
        {
            if (near[j] != -1 && G[j][near[j]] < min)
            {
                k = j;
                min = G[j][near[j]];
            }
        }
        e.from = k;
```

```
        e.to = near[k];
        e.weight = min;
        near[k] = -1;
        T[i] = e;
        // Обновим массив near
        for (int j = 0; j < V; j++)
        {
            if (near[j] != -1 && G[j][k] < G[j][near[j]])
            {
                near[j] = k;
            }
        }
    }
    return T;
}
```

Код алгоритма Крускала

```
void Union(int u, int v, vector<int>& s)
{
    if (s[u] < s[v])
    {
        s[u] += s[v];
        s[v] = u;
    }
    else
    {
        s[v] += s[u];
        s[u] = v;
    }
}

int Find(int u, vector<int>& s)
{
    int x = u;
    int v = 0;

    while (s[x] > 0)
    {
        x = s[x];
    }

    while (u != x)
    {
        v = s[u];
        s[u] = x;
        u = v;
    }
    return x;
}
vector<Edge> KruskalsMCST(vector<Edge> &G, int V)
{
    int E = G.size();// Количество ребер
    vector <Edge> T(V - 1);
    vector<int> track(E);  // Чтобы проверить добавили ли мы вершину
или нет
```

```
    vector<int> set(V + 1, -1); // Для того чтобы проверить не получится ли цикл
    int i = 0;
    while (i < V - 1)
    {
        int p1, p2, k = 0;
        Edge min(0, 0, I);
        for (int j = 0; j < E; j++)
        {
            if (track[j] == 0 && G[j] < min)
            {
                min = G[j];
                k = j;
            }
        }
        p1 = Find(min.from, set);
        p2 = Find(min.to, set);
        if (p1 != p2)
        {
            T[i] = min;
            Union(p1, p2, set);
            i++;
        }
        track[k] = 1;
    }
    return T;
}
```

Код алгоритма построение пути между вершинами

```
vector<Edge> getPathBFS(Graph &edges, int v1, int v2 )
{
    int done =0,V = edges.getN();
    queue<int> q;
    vector<int> visited(V);
    unordered_map<int, int> t;

    q.push(v1);
    visited[v1] = 1;
    while (!q.empty() && done == 0)
    {
        int front = q.front();
        q.pop();
        for (int i = 0; i < V; i++)
        {
            if (edges[front][i] != 0 && visited[i] != 1)
            {
                q.push(i);
                t[i] = front;
                visited[i] = 1;
                if (i == v2)
                {
                    done = 1;
                    break;
                }
            }
        }
    }
    vector<Edge> a;
    if (done == 0)
        return a;
    else
    {
        int k = v2;
        int from = k,to=t[k];
        while (k != v1)
        {
            from = k;
            k = t[k];
            to = k;
            Edge e(from, to, edges[from][to]);
            a.push_back(e);
        }
        return a;
    }
}
```

Код всей программы

```
// Kursovaya_sem33.cpp : Этот файл содержит функцию "main". Здесь
начинается и заканчивается выполнение программы.
#include <iostream>
#include <fstream>
#include<vector>
```

```
#include<queue>
#include<unordered_map>
#define I INT_MAX
using namespace std;

class Edge
{
public:
    int from, to, weight;
    Edge(int from = -1, int to = -1, int weight = 0) : from(from),
to(to), weight(weight) {}
    Edge(const Edge& Ed)
    {
        from = Ed.from;
        to = Ed.to;
        weight = Ed.weight;
    }
    int operator<(Edge& Ed)
    {
        return (weight < Ed.weight);
    }
    friend ostream& operator<<(ostream& s, Edge& e);
    friend istream& operator>>(istream& s, Edge& e);
    friend ofstream& operator<<(ofstream& s, Edge& e);
    friend ifstream& operator>>(ifstream& s, Edge& e);
};
ostream& operator<<(ostream& s, Edge& e)
{
    s << "[" << e.from << "] --- [" << e.to << "]    " << e.weight;
    return s;
}
istream& operator>>(istream& s, Edge& e)
{
    s >> e.from >> e.to >> e.weight;
    return s;
}
ofstream& operator<<(ofstream& s, Edge& e)
{
    s << e.from << " " << e.to << " " << e.weight << endl;
    return s;
}
ifstream& operator>>(ifstream& s, Edge& e)
{
    s >> e.from >> e.to >> e.weight;
    return s;

}

class Graph
{
private:
    class Row
    {
    public:
        int* row;
        Row()
```

```
        {
            row = nullptr;
        }
        Row(int N)
        {
            row = new int[N];
        }
        int& operator [](const int i)
        {
            return row[i];
        }
    };
    int n;
    Row* mat;
public:
    Graph()
    {
        n = 0;
    }
    Graph(int N)
    {
        mat = new Row[N];
        n = N;
        for (int i = 0; i < N; i++)
        {
            mat[i] = Row(N);
        }
    }
    Graph(int N,int num)
    {
        mat = new Row[N];
        n = N;
        for (int i = 0; i < N; i++)
        {
            mat[i] = Row(N);
        }
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                mat[i][j] = num;
            }
        }
    }
    Graph(const Graph& M)
    {
        mat = new Row[M.n];
        n = M.n;
        for (int i = 0; i < n; i++)
        {
            mat[i] = Row(n);
        }

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
```

```
            {
                mat[i][j] = M.mat[i][j];
            }
        }
    }
    int getN() { return n; }
    Row& operator [](const int i)
    {
        return mat[i];
    }
    friend ostream& operator <<(ostream& stream, Graph& M);
    friend istream& operator >>(istream& stream, Graph& M);
    friend ofstream& operator <<(ofstream& stream, Graph& M);
    friend ifstream& operator >>(ifstream& stream, Graph& M);
};

ostream& operator << (ostream& stream, Graph& M)
{
    stream << "Size: " << M.n << "*" << M.n << endl;

    for (int i = 0; i < M.n; i++)
    {
        for (int j = 0; j < M.n; j++)
        {
            if (M[i][j] == I)
                stream << "0 ";
            else
                stream << M[i][j] << " ";
        }
        cout << endl;
    }
    return stream;
}
istream& operator >>(istream& stream, Graph& M)
{
    cout << "Выведите " << M.n << "x" << M.n << " чисел\n";
    int n;
    for (int i = 0; i < M.n; i++)
    {
        for (int j = 0; j < M.n; j++)
        {
            stream >> n;
            (n == 0) ? n = I : n = n;
            M[i][j] = n;
        }
    }
    return stream;
}
ofstream& operator <<(ofstream& stream, Graph& M)
{
    stream << M.n << endl;
    int n;
    for (int i = 0; i < M.n; i++)
    {
        for (int j = 0; j < M.n; j++)
        {
```

```
            n = M[i][j];
            (n == I) ? n = 0 : n = n;

            stream << n << " ";
        }
        stream << endl;
    }
    return stream;

}
ifstream& operator >>(ifstream& stream, Graph& M)
{
    int n;
    stream >> n;
    M = Graph(n);
    for (int i = 0; i < M.n; i++)
    {
        for (int j = 0; j < M.n; j++)
        {
            stream >> n;
            (n == 0) ? n = I : n = n;// Инициализируем INT_MAX-ом для
Алгоритма Прима
            M[i][j] = n;
        }
    }
    return stream;
}

vector<Edge> PrimsMST(Graph& G)
{
    int u, v, V = G.getN(), min = I;
    vector<int> near(V);
    vector<Edge> T(V - 1, 0);

    for (int i = 0; i < V; i++)// Находим мин ребро
    {
        near[i] = I;
        for (int j = i; j < V; j++)
        {
            if (G[i][j] < min)
            {
                min = G[i][j];
                u = i;
                v = j;
            }
        }
    }
    Edge e(u, v, min);
    T[0] = e;
    near[u] = near[v] = -1;
    // Иницализация массива ближайшим к i-той вершине вершиной
    // т.е.  near[i] - ближайшее к  i  вершина из тех которую мы уже
добавили в решение
    for (int i = 0; i < V; i++)
    {
        if (near[i] != -1)
```

```
        {
            if (G[i][u] < G[i][v])
            {
                near[i] = u;
            }
            else
            {
                near[i] = v;
            }
        }
    }

    for (int i = 1; i < V - 1; i++)
    {
        int k;
        min = I;
        for (int j = 0; j < V; j++)
        {
            if (near[j] != -1 && G[j][near[j]] < min)
            {
                k = j;
                min = G[j][near[j]];
            }
        }
        e.from = k;
        e.to = near[k];
        e.weight = min;
        near[k] = -1;
        T[i] = e;
        // Обновим массив near
        for (int j = 0; j < V; j++)
        {
            if (near[j] != -1 && G[j][k] < G[j][near[j]])
            {
                near[j] = k;
            }
        }
    }
    return T;
}

// Find & Union функции для непересками непересекающихся множеств чтобы обнаружить циклы в Алгоритме Крускала
void Union(int u, int v, vector<int>& s)
{
    if (s[u] < s[v])
    {
        s[u] += s[v];
        s[v] = u;
    }
    else
    {
        s[v] += s[u];
        s[u] = v;
    }
}
```

```
int Find(int u, vector<int>& s)
{
    int x = u;
    int v = 0;

    while (s[x] > 0)
    {
        x = s[x];
    }

    while (u != x)
    {
        v = s[u];
        s[u] = x;
        u = v;
    }
    return x;
}

vector<Edge> KruskalsMCST(vector<Edge> &G, int V)
{

    int E = G.size();// Количество ребер
    vector <Edge> T(V - 1);
    vector<int> track(E);  // Чтобы проверить добавили ли мы вершину
или нет
    vector<int> set(V + 1, -1); // Для того чтобы проверить не
получится ли цикл

    int i = 0;
    while (i < V - 1)
    {
        int p1, p2, k = 0;
        Edge min(0, 0, I);
        for (int j = 0; j < E; j++)
        {
            if (track[j] == 0 && G[j] < min)
            {
                min = G[j];
                k = j;
            }
        }
        p1 = Find(min.from, set);
        p2 = Find(min.to, set);
        if (p1 != p2)
        {
            T[i] = min;
            Union(p1, p2, set);
            i++;
        }
        track[k] = 1;
    }
    return T;
}

```

```
vector<Edge> getPathBFS(Graph &edges, int v1, int v2 )
{
    int done =0,V = edges.getN();
    queue<int> q;
    vector<int> visited(V);
    unordered_map<int, int> t;

    q.push(v1);
    visited[v1] = 1;
    while (!q.empty() && done == 0)
    {
        int front = q.front();
        q.pop();
        for (int i = 0; i < V; i++)
        {
            if (edges[front][i] != 0 && visited[i] != 1)
            {
                q.push(i);
                t[i] = front;
                visited[i] = 1;
                if (i == v2)
                {
                    done = 1;
                    break;
                }
            }
        }
    }
    vector<Edge> a;
    if (done == 0)
        return a;
    else
    {
        int k = v2;
        int from = k,to=t[k];
        while (k != v1)
        {
            from = k;
            k = t[k];
            to = k;
            Edge e(from, to, edges[from][to]);
            a.push_back(e);
        }
        return a;
    }
}

int main()
{
    setlocale(LC_ALL, "");
    Graph G1;
    ifstream f1("my.txt");
    if (f1.is_open())
    {
        f1 >> G1;
        f1.close();
```

```
    }
    vector<Edge> T = PrimsMST(G1);
    int sum = 0;
    for (Edge& e : T)
    {
        sum += e.weight;
        cout << e << endl;
    }
    cout << endl << "Min cost (Prims) " << sum << endl;
    vector<Edge> Edges;
    for (int i = 0; i < G1.getN(); i++)
    {
        for (int j = i; j < G1.getN(); j++)
        {
            if (G1[i][j] != I)
            {
                Edge e(i, j, G1[i][j]);
                Edges.push_back(e);
            }
        }
    }
    T = KruskalsMCST(Edges, G1.getN());
    sum = 0;
    for (Edge& e : T)
    {
        sum += e.weight;
        cout << e << endl;
    }
    cout << endl << "Min cost (Kruskals) " << sum << endl;
    // Запишу Вектор Еджес в виде матрицы смежностей для демонстрации
нахождения пути
    Graph G(G1.getN(), 0);
    for (int i = 0; i < T.size(); i++)
    {
        G[T[i].from][T[i].to] = T[i].weight;
        G[T[i].to][T[i].from] = T[i].weight;
    }
    vector<Edge> res = getPathBFS(G, 2, 0);
    for (int i=0; i<res.size();i++)
    {
        cout << res[i]<<endl;
    }
    return 0;
}
```