

**Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное
образовательное учреждение высшего образования
«Российский экономический университет имени Г.В. Плеханова»
Институт математики, информационных систем и цифровой экономики
Базовая кафедра цифровой экономики института развития
информационного общества**

КУРСОВАЯ РАБОТА

**по дисциплине «Структуры и алгоритмы компьютерной обработки
данных»
на тему «Реализация алгоритма Краскала в списковом представлении
графа»**

Выполнил :
обучающийся группы
15.11Д-МО11/196
очной формы обучения
института математики,
информационных систем и
цифровой экономики Бабурян
Арменак Каренович

Научный руководитель:
старший преподаватель
Чапкин Н.С.,

Москва – 2021

Содержание:

Введение.....	3
Глава 1. Граф.....	5
1.1.Граф.....	5
1.2.Представление графа в ИС.....	7
1.3.Алгоритмы на графах.....	10
1.4.Алгоритм нахождения остовного дерева.....	12
Глава 2. Проектирование и разработка программы.....	14
2.1. Состав и назначение отдельных компонент программы.....	14
2.2. Алгоритмы реализации задач.....	14
2.3. Описание машинного эксперимента.....	17
Заключение.....	19
Список использованных источников.....	20
Приложение 1.....	22
Приложение 2.....	23
Приложение 3.....	25
Приложение 4.....	26
Приложение 5.....	28

Введение

Графы интересны не только своим огромным спектром возможных применений в совершенно разных сферах (как научных ,химия ,экономика, так и сугубо практических электротехника ,строительство) , но и фактически бесконечной глубиной сложности .Именно поэтому , я выбрал их в качестве темы для своей курсовой работы.Таким образом я определил для себя невыполнимую задачу написать в двух словах (примерно 30 страниц) о такой обширной теме , для специалистов в которой написать монографию в несколько сот страниц является обыденностью .Поэтому , для упрощения и уменьшения своего труда , в данной работе будет объяснено и рассказано лишь об основных вещах (будут рассмотрены лишь основные алгоритмы обработки графов).

Актуальность данной темы состоит в бесчисленном множестве применений и открытий(или переоткрытий) графов в разных сферах

Целью данной курсовой работы является исследование и систематизация знаний о графах и реализация одного из представлений для практического применения в прикладных программах.

Задание на курсовую работу

Для достижения поставленной цели необходимо решить следующие задачи:

- рассмотреть понятие и сущность графов
- изучить возможные представления графов

- изучить понятие и сущность алгоритма Краскала
- спроектировать и разработать программу сложной структуры (интерфейс-реализация-клиент), которая будет псевдо-случайным образом проектировать граф со псевдо-случайным весом ребра и выполнять над ним операция , для нахождения остовного дерева.
- реализовать индивидуальную задачу: найти и вывести остовное дерево графа.

Объектом данного исследования являются граф

Предметом исследования является представление графа и алгоритм Краскала

Метод исследования. В данной курсовой работе применяются такие общенаучные методы исследования, как анализ, эксперимент и моделирование.

Глава 1.Граф

1.1. Граф

Теория графов — раздел дискретной математики, изучающий графы. В общем и целом граф — это множество точек (вершин, узлов), которые соединяются множеством линий (рёбер, дуг). Теория графов включена в большинство учебных программ для начинающих специалистов в математике и информатике (или Computer Science) , так как имеет много общего с другими теориями и направлениями математики :

- Также наглядна как и геометрия
- Проста в объяснении , но при этом имеет множество сложнейших нерешённых задач , как теория чисел
- Выраженный прикладной характер

Для понимания специфики и терминологий необходимо обозначить определения и некоторые характеристики и выходящие из них виды графов:

- Цикл.Циклом называют путь, в котором первая и последняя вершины совпадают.
- Путь вершины.Путём или цепью в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.
- Степень вершины.Степенью вершины это количество рёбер, для которых она является концевой (при этом петли считают дважды).
- Ориентированность.Существуют ориентированные(ОрГраф) (у каждого ребра есть направление) , неориентированные (все дуги не

имеют направления) и смещенные (некоторые линии имеют направления некоторые нет).Очевидно , что смещенный и неориентированный граф можно представить в виде ориентированного , а вот наоборот нет.

- Взвешенность.Графы у которых рёбра имеют вес - взвешенные.
- Кратность рёбер и существование петель.Граф который имеет кратные рёбра и/или петли(ребро соединяет один и тот же узел) называется мультиграфом или псевдографом.
- Связность.Если каждый узел имеет общее ребро с хотя бы одним другим ребром ,то это связный граф.Также существует такое понятие , как полносвязный граф , это граф у которого каждая вершина соединена с каждой другой вершиной.

Основателем теории графов считается Леонард Эйлер (1707-1882), решивший в 1736 году известную в то время задачу о кёнигсбергских мостах.На протяжении более ста лет развитие теории графов определялось в основном проблемой четырёх красок. Решение этой задачи в 1976 году оказалось поворотным моментом истории теории графов, после которого произошло её развитие как основы современной прикладной математики. Универсальность графов незаменима при проектировании и анализе коммуникационных сетей.

Несмотря на разнообразные, усложнённые, малопонятные и специализированный термины многие модельные (схемные, структурные) и конфигурационные проблемы переформулируются на языке теории графов, что позволяет значительно проще выявить их концептуальные трудности.В разных областях знаний понятие «граф» может встречаться под следующими названиями:

- структура (гражданское строительство);

- сеть (электротехника);
- социограмма (социология и экономика);
- молекулярная структура (химия);
- навигационная карта (картография);
- распределительная сеть (энергетика)

В этом смысле теория графов как направление дискретной математики, является своего рода математическим оружием, которым могут воспользоваться специалисты вышеперечисленного спектра наук о поведении, так и абстрактных дисциплин (теории множеств, теории матриц, теории групп и так далее). Таким образом автоматизация процессов и алгоритмов в графах и улучшение их эффективности может повлиять и я уверен повлияет на науку и обыденную жизнь всего человечества.

1.2. Представление графа в ИС

Как и многие другие сущности из математики и информатики не имею однозначной реализации в ЭВМ, так и для графа существует некоторые возможные представления, такие как:

- Матрица смежности. Матрица смежности графа — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 (нет ребра) или 1 (ребро есть). Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа. Если это обычный граф, то есть не псевдограф, то элементы главной диагонали равны нулю.

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

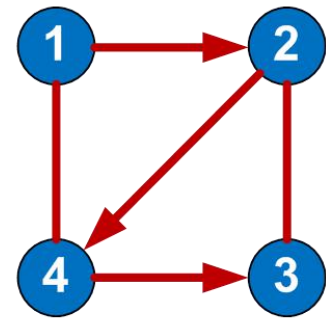


Рисунок 1- Матрица смежности

- Матрица инцидентности. 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 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

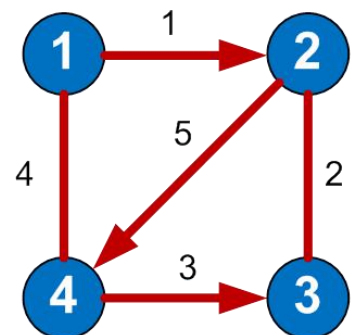


Рисунок 2- Матрица инцидентности

Матрица инцидентности для своего представления требует нумерации рёбер, что не всегда удобно.

- 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	

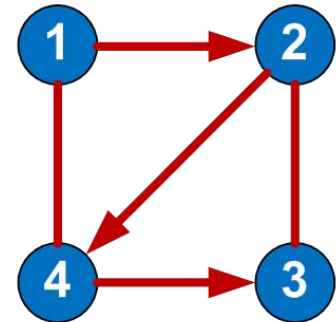


Рисунок 3- Список рёбер

- Список смежности. Выявим преимущества и недостатки и более подробно остановимся именно на этом представлении так как именно оно реализован в качестве машинного эксперимента.

Наиболее эффективно реализовать список смежности с помощью массива из списков, узлами которого в узлах которого будут размещаться указатели на вершины графа, и опционально вес ребра.

Также вместо указателей может быть идентификационный номер вершины и в паре с статическим массивом (то есть копия которого храниться в каждом объекте класса) хранящем указатели на каждую вершину, это может быть эффективнее первого варианта.

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

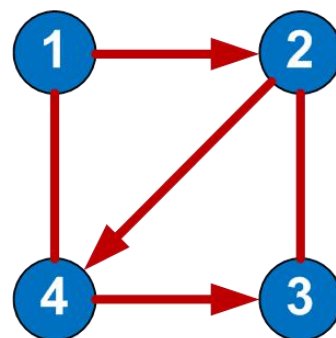


Рисунок 4- Список смежности

Преимущества :

- Быстрый перебор соседей вершины.
- Быстрая проверка наличия и удаление ребра.
- Рациональное использование памяти.

Недостатки:

- Чем больше рёбер тем больше занимает памяти , и тем более матрица смежности становится эффективнее в некоторых алгоритмах.
- Слишком долгая проверка связности двух вершин.
- Для взвешенных графов приходится хранить список, элементы которого должны содержать два значащих поля, что усложняет код:
 - ◆ номер вершины, с которой соединяется текущая;
 - ◆ вес ребра.

1.3. Алгоритмы на графах

Поиск в ширину

Обход или поиск — это одна из фундаментальных операций, выполняемых на графах. Поиск в ширину 1)начинается с конкретной вершины, далее 2)проходить по всем смежным вершинам на этой глубине , затем 3)увеличение глубины , обратно к пункту 2. В графах, в отличие от деревьев, могут быть циклы ,поэтому для предотвращения бесконечного цикла , необходимо отслеживать посещённые вершины. При реализации алгоритма обычно используется «очередь».

Применяется для определения кратчайших путей и минимальных остовных деревьев.

Поиск в глубину

Поиск в глубину начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек».

Применяется:

- для нахождения пути между двумя вершинами;
- для обнаружения циклов на графе;
- в топологической сортировке;
- в головоломках с единственным решением (например, лабиринтах).

Кратчайший путь

Кратчайший путь от одной вершины графа к другой — это путь, при котором сумма весов рёбер, его составляющих, должна быть минимальна.

Алгоритмы нахождения кратчайшего пути:

- Алгоритм Дейкстры.
- Алгоритм Беллмана-Форда.

Применяются в:

- сетях для решения проблемы минимальной задержки пути;
- абстрактных автоматах для определения через переход между различными состояниями возможных вариантов достижения некоторого целевого состояния, например минимально возможного количества ходов, необходимого для победы в игре.

Обнаружение циклов

Обнаружение циклов — это процесс выявления таких циклов.

Алгоритмы обнаружения цикла:

- Алгоритм Флойда.
- Алгоритм Брента.

Применяются:

- в распределённых алгоритмах, использующих сообщения;
- для обработки крупных графов с использованием распределённой системы обработки в кластере;

1.4. Алгоритм нахождения остовного дерева

Минимальное остовное дерево — это подмножество рёбер графа, которое соединяет все вершины, имеющие минимальную сумму весов рёбер, и без циклов.

Алгоритмы :

- Алгоритм Прима.
- Алгоритм Краскала.

Применяются:

- для создания деревьев для распределения данных в компьютерных сетях;
- в кластерном анализе с использованием графов;
- при сегментации изображений;
- при социально-географическом районировании, когда смежные регионы объединяются.

Алгоритм Прима.

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

Результатом работы алгоритма является остовное дерево минимальной стоимости.

Так как , в машинном эксперименте используется Алгоритм Краскала нахождения минимального остовного дерева , рассмотрим его подробнее.

Алгоритм Краскала.

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

Глава 2. Проектирование и разработка программы

2.1. Состав и назначение отдельных компонент программы

При запуске программы пользователю предлагается ввести количество вершин и способ заполнения весов рёбер.

```
C:\Users\armen\CLionProjects\FinalKR\cmake-build-debug\FinalKR.exe
Enter size of graph:1000
Do you want to enter numbers(1) or made it randomly(2)?2
```

Рисунок 5– указание количества вершин и способ заполнения рёбер

Так как программа реализована в системе интерфейс-реализация-клиент , следовательно для её работы необходимо подключить к файлу main.cpp , где размещается точка входа , два файла с классами (list.h и matrixadj.h) и два файла с уже реализованными методами(list.cpp и matrixadj.cpp).

2.2 Алгоритмы реализации задач

Так как вся программа и некоторые её компоненты имеют большой объём , то в данном пункте будет рассмотрена лишь реализация алгоритма Краскала.

Так как алгоритм в общем виде уже рассмотрен и вся программа и некоторые её компоненты имеют большой объём , то в данном пункте будет рассмотрена лишь реализация алгоритма Краскала и класса graph.

Описание Класа graph: Рисунок 6

```
class graph {
private:
    struct node{
```

```

    graph* pointer;
    int weight;
    node* next;
};

node* Node=new node;
int weight;
int id;static int statnum;
static graph ** Graph;static int gsize;

public:
    static graph* copyarr(graph* A ,int size);

    static bool fullarr(bool asd[], int size);
    static graph* KruskalAlg(graph asd[] , int size);
    graph();
    static graph* grapharray(matrixadj* mat);
    static void Outarray(graph* arr ,int size);
    void Out();
    void deletefirst();
    void deletethis(int id ,graph * B);
    ~graph();
    void deletegraph();
    static void connect(graph *,graph * , int weight);
    static void connectSort(graph *,graph * , int weight);
    void newNodeSort(int weight,int size ,graph* pointer);

};

```

В приватном поле данных содержится статический указатель на массив Graph , в котором размещены указатели на все существующие в данный момент объекты , и его размер. Также в каждом объекте есть поле

веса(weight) , идентификатор (id) , общее количество созданных объектов(statnum) и список в котором также храниться вес ребра , указатель на другую вершину и указатель на следующий узел списка.

Описание Алгоритма KruskalAlg: Рисунок 7

```
graph* graph::KruskalAlg(graph asd[] , int size){
    graph* tempgraph=new graph[size];
    bool* tree= new bool[size];
    tempgraph = graph::copyarr(asd,size);graph* final=new graph[size];
    int diffid= tempgraph[0].id;
    int min=2147483647;int iterofgraph;int id;
    while(!graph::fullarr(tree,size)){
        min=2147483647;
        for(int i=0;i<size;i++){
            node *tempnode = tempgraph[i].Node;
            if(tempnode->weight < min){min=tempnode->weight;
                iterofgraph=i;id=tempnode->pointer->id;
            }
        }
        if((!tree[iterofgraph]) || (!tree[id-diffid])){
            tree[iterofgraph]=1;tree[id-diffid]=1;
            graph::connectSort(&final[iterofgraph],&final[id-diffid],min);
        }tempgraph[iterofgraph].deletethis(id,&tempgraph[iterofgraph]);
        tempgraph[iterofgraph].deletethis(iterofgraph+diffid,&tempgraph[id-diffid]);
        delete []tree;delete[] tempgraph;
        return final;}
}
```


В этот статический метод передаётся массив графов из которого необходимо сделать оставное дерево , и его размер , передаёт этот метод , ссылку на остовное дерев.Сначала создаётся временный граф tempgraph по которому будет проходить основной цикл , массив bool который хранит

уже использованные вершины , tempgraph заполняется массивом asd то есть копируется без изменения id (важно) , так как id разные но они необходимы при работе цикла , создаётся число diffid , которое нивелирует эту разницу. Основной цикл , пока есть хотя бы одна вершина которая не соединена с другими , делать искать наименьшее ребро , и если оно соединяет не соединённую ни разу вершину , то соединить эти вершины в финальном графе , и в конце удалить это звено из временного графа.

2.3 Описание машинного эксперимента

Программа запрашивает размер графа:

Рисунок 8



```
Enter size of graph:8
```

Предлагает выбрать способ заполнения весов рёбер:

1) Набрать их вручную или 2) Заполнить псевдо-случайными числами.

Рисунок 9



```
Do you want to enter numbers(1) or made it randomly(2)?2
```

Выводиться изначальный граф, сразу после основного дерева:

1)8) -4827	- 2) -6334	- 5) -15350	- 3) -15724	- 6) -16827	- 4) -26962	-
2)1) -6334	- 7) -11538	- 4) -12382	- 5) -19718	- 6) -21538	- 3) -23811	- 8) -25667
-						
3)5) -7711	- 1) -15724	- 4) -17673	- 2) -23811	- 7) -32662	-	
4)5) -1842	- 6) -9040	- 8) -15890	- 3) -17673	- 1) -26962	- 2) -12382	- 7) -22648
-						
5)4) -1842	- 7) -7376	- 3) -7711	- 1) -15350	- 8) -16944	- 6) -18756	- 2) -19718
-						
6)8) -2306	- 4) -9040	- 1) -16827	- 5) -18756	- 2) -21538	- 7) -22704	-
7)5) -7376	- 2) -11538	- 4) -22648	- 6) -22704	- 3) -32662	-	
8)6) -2306	- 1) -4827	- 4) -15890	- 5) -16944	- 2) -25667	-	
25)32) -4827	- 26) -6334	-				
26)25) -6334	-					
27)29) -7711	-					
28)29) -1842	-					
29)28) -1842	- 31) -7376	- 27) -7711	-			
30)32) -2306	-					
31)29) -7376	-					
32)30) -2306	- 25) -4827	-				

Рисунок 10

Заключение

В данной курсовой работе были проведены исследования графов , алгоритмов на графах, алгоритма остовного дерева Краскала их классификация и анализ, их основные свойства, область их применения, а также приведены примеры их использования.

Как пример практической работы с алгоритмом Краскала была разработана сложная программа, которая выполняет ряд задач: ввод значений или их псевдо-случайная генерация , вывод созданного графа , вывод остовного дерева.

Исследование и программная реализация основных действий с графами позволили систематизировать и расширить теоретические знания в данной области и применить их на практике.

Список использованных источников

1. ГОСТ Р 54593-2011. Информационные технологии. Свободное программное обеспечение. Общие положения.
2. ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.
3. ГОСТ Р 56920-2016. Системная и программная инженерия. Тестирование программного обеспечения. Часть 1. Понятия и определения.
4. Алгоритмизация и программирование : Учебное пособие / С.А. Канцедал. - М.: ИД ФОРУМ: НИЦ ИНФРА-М, 2013. — 352 с.
5. Воронцова Е.А. Программирование на С++ с погружением: практические задания и примеры кода - М.:НИЦ ИНФРА-М, 2016. — 80 с.
6. Программирование, Python, С++, Часть 1, Поляков К.Ю., 2019. — 140 с.
7. Стивен Прата. Язык программирования С. Лекции и упражнения, 6-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2015. — 928 с.
8. Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. — К.:Издательство «ДиаСофт», 2001.- 688 с.
9. Википедия Режим доступа:
https://ru.wikipedia.org/wiki/Алгоритм_Краскала (дата обращения 15.10.2021)
10. МАХimal. Режим доступа:
https://e-maxx.ru/algo/mst_kruskal (дата обращения 15.10.2021)
11. МАХimal. Режим доступа:
https://e-maxx.ru/algo/mst_prim (дата обращения 15.10.2021)
12. Prog-cpp.ru
<https://prog-cpp.ru/data-graph/> (дата обращения 15.10.2021)

13. Habr

<https://habr.com/ru/company/otus/blog/568026/> (дата обращения 15.10.2021)

```
#include <iostream>

#include "list.h"

#include <random>

using namespace std;

int main() {

    int size=13;

    std::cout<<"Enter size of graph:";std::cin>>size;

    matrixadj* asd1 = new matrixadj(size);asd1->In();

    graph* asd2 = graph::grapharray(asd1);

    graph* KRUSKAL = graph::KruskalAlg(asd2,size);

    graph::Outarray(asd2 , asd1->size);

    graph::Outarray(KRUSKAL,size);

    delete []asd2;

    delete asd1;

    delete []KRUSKAL;

    return 0;

}
```

```

#ifndef FINALKR_LIST_H
#define FINALKR_LIST_H
#include "matrixadj.h"
class graph {
private:
    struct node{
        graph* pointer;
        int weight;
        node* next;
    };
    node* Node=new node;
    int weight;
    int id;static int statnum;
    static graph ** Graph;static int gsize;

public:
    static graph* copyarr(graph* A ,int size);
    static bool fullarr(bool asd[], int size);
    static graph* KruskalAlg(graph asd[] , int size);
    graph();
    static graph* grapharray(matrixadj* mat);
    static void Outarray(graph* arr ,int size);
    void Out();
    void deletefirst();
    void deletethis(int id ,graph * B);
    ~graph();
    void deletegraph();
    static void connect(graph *,graph * , int weight);

```

```
static void connectSort(graph *,graph * , int weight);  
void newNodeSort(int weight,int size ,graph* pointer);  
  
};  
  
#endif FINALKR_LIST_H
```



```
#ifndef FINALKR_MATRIXADJ_H
#define FINALKR_MATRIXADJ_H

class matrixadj{
private:
public:
    int ** matrix;
    int size;
    matrixadj(int a);
    ~matrixadj();
    void In();
};
#endif //FINALKR_MATRIXADJ_H
```

```
#include "matrixadj.h"
#include "iostream"

matrixadj::matrixadj(int a) {
    matrix = new int*[a];size=a;
    for (int i=0;i<a;i++){
        matrix[i]=new int[a];
        for(int j=0;j<a;j++){
            matrix[i][j]=0;
        }
    }
}

void matrixadj::In(){
    std::cout <<"Do you want to enter numbers(1) or made it randomly(2)?";
    int temp;std::cin >> temp;
    switch (temp){
        case 1:
            for (int i =0 ; i < size; i++){
                for (int j =0 ; j < size; j++){
                    std::cout<<"Enter lenght" << i <<" between " << j << ":";
                    std::cin >>matrix[i][j];
                }
            }
            break;
        case 2:
            for (int i =0 ; i < size; i++){
                for (int j =0 ; j < size; j++){
                    if ((i!=j) && (rand()-7500 < rand())){
                        if(matrix[i][j]==0){matrix[i][j]=
```

```

        rand();matrix[j][i]=matrix[i][j];}
        }
    }
    break;
}

}

matrixadj::~~matrixadj(){
    for(int i=0; i < size; i++){
        delete matrix[i];
    }delete matrix;matrix=NULL;
}

```

```
#include "iostream"
#include "list.h"
#include "matrixadj.h"
#include <iomanip>
#include <math.h>

int graph::gsize=0;graph** graph::Graph=NULL;int graph::statnum=0;
graph::graph(){
    statnum++;gsize++;id=statnum;
    weight=0;this->Node->weight=0;
    Node->next=NULL;Node->pointer=NULL;
    if(Graph){
        graph * asd[gsize-1];
        for(int i=0;i<gsize-1;i++){
            asd[i]=Graph[i];
        }
        delete Graph;
        Graph =new graph*[gsize];
        for(int i=0;i<gsize-1;i++){
            Graph[i]=asd[i];
        }Graph[gsize-1]=this;
    }else {Graph=new graph*[gsize];Graph[gsize-1]=this;}
    //std::cout <<num;
}

graph * graph::grapharray(matrixadj* mat) {
    graph* asd = new graph[mat->size];
    for (int i =0;i<mat->size;i++){
        for (int j =0;j<mat->size;j++){
            if (mat->matrix[i][j] != 0){
```

```

        graph::connectSort(&asd[i],&asd[j],mat->matrix[i][j]);
    }

    }

}

return asd;
}

void graph::Outarray(graph * arr , int size){
    std::cout <<std::endl;
    for (int i =0;i<size;i++){
        arr[i].Out();
    }
}

void graph::Out(){
    std::cout<<std::setw( 2)<<std::left <<id<<"");
    node* asd = this->Node;int i= 0;
    while((asd->weight != NULL)){
        std::cout <<std::setw(2)<<asd->pointer->id<<"");
        std::cout<< "|-"<<std::setw( 10) <<std::left <<asd->weight<<"-|";i++;
        asd = asd->next;if(!asd){break;}
    }
    std::cout<<"\n";
}

graph::~~graph(){
    deletegraph();
    this->Node=NULL;
    graph** asd = new graph*[gsize-1];
    int temp=0;
    for (int i =0;i<gsize-1;i++){
        if(this->id == Graph[i]->id){

```

```

        temp=i;break;
    }else{asd[i]= Graph[i];}
}
for(int i=temp;i<gsize-1;i++){
    asd[i]=Graph[i+1];
}
delete Graph;Graph = new graph*[gsize-1];Graph=asd;
}

void graph::deletagraph() {
    node* TempNode = this->Node;
    while(TempNode){
        if (TempNode->pointer){
            deletethis(id ,TempNode->pointer);
        }
        TempNode= TempNode->next;
    }
    delete this->Node;
}

void graph::deletethis(int id,graph * B) {
    if (B->Node->pointer->id == id) {B->deletefirst();} else {
        node* TempNode = B->Node , *LastTempNode = B->Node;
        while(TempNode){
            if(TempNode->pointer->id == id){
                LastTempNode->next = TempNode->next;
                delete TempNode;TempNode = NULL;
            }else{LastTempNode = TempNode;TempNode =
TempNode->next;}
        }
    }
}

```

```

}

graph* graph::KruskalAlg(graph asd[] , int size){
    graph* tempgraph=new graph[size];
    bool* tree= new bool[size];
    tempgraph = graph::copyarr(asd,size);graph* final=new graph[size];
    int diffid= tempgraph[0].id;
    int min=2147483647;int iterofgraph;int id;
    while(!graph::fullarr(tree,size)){
        min=2147483647;
        for(int i=0;i<size;i++){
            node *tempnode = tempgraph[i].Node;
            if(tempnode->weight < min){min=tempnode->weight;
                iterofgraph=i;id=tempnode->pointer->id;
            }}
        if((!tree[iterofgraph]) || (!tree[id-diffid])){
            tree[iterofgraph]=1;tree[id-diffid]=1;
            graph::connectSort(&final[iterofgraph],&final[id-diffid],min);
        }tempgraph[iterofgraph].deletethis(id,&tempgraph[iterofgraph]);

tempgraph[iterofgraph].deletethis(iterofgraph+diffid,&tempgraph[id-diffid]);
        }delete []tree;delete[] tempgraph;
        return final;
    }

graph* graph::copyarr(graph A[] ,int size){
    graph* B=new graph[size];
    //int* idA = new int[size]; int* idB = new int[size];
    int diffId=B[0].id - A[0].id;
    //for (int i=0;i<size;i++){idA[i]=A[i]->id;idB[i]=B[i].id;}

```

```

    for(int i=0;i<size;i++){
        node* tempB=B[i].Node;node* tempA=A[i].Node;
        while(tempA->pointer){
            tempB->pointer=&B[(tempA->pointer->id + diffId)-B[0].id];
            tempB->weight=tempA->weight;
            tempA=tempA->next;
            if(!tempA){tempB->next=NULL;break;}tempB->next=new
node;tempB=tempB->next;
        }
    }
    return B;
}

bool graph::fullarr(bool asd[], int size){
    for(int i=0;i<size;i++){
        if(asd[i]==0){return 0;}
    }
    return 1;
}

void graph::deletefirst(){
    if(Node){node * asd=Node->next;
        delete this->Node;
        Node = new node;Node=asd;}
}

void graph::connect(graph *A,graph *B,int weight){
    if (A->id != B->id){
        node * asd=A->Node;bool flag=1;node *lastasd=asd;
        while((asd->pointer)){
            if (asd->pointer->id == B->id){
                flag=0;break;
            }
        }
    }
}

```



```

        }lastasd=asd;asd=asd->next;if    (!asd){break;}    //БМЕСТО    if
(!asd){break;} МОЖНО В while(asd->pointer) && asd
    }
    if (flag){
        if (asd){
            if (!asd->pointer){asd->pointer=B;}asd->weight=weight;
        }else{
            asd= new node ;
            asd->next=NULL;asd->weight=weight;
            asd->pointer=B;lastasd->next=asd;
        }
        asd=B->Node;lastasd =asd;
        while(asd->pointer){
            lastasd=asd;asd=asd->next;if (!asd){break;}
        }
        if (asd){
            if (!asd->pointer){asd->pointer=A;}asd->weight=weight;
        }else {asd=new node;
            asd->next=NULL;asd->weight=weight;
            asd->pointer=A;lastasd->next=asd;}
        }
    }

}

void graph::connectSort(graph *A,graph *B,int weight){
    if (A->id != B->id){
        node * asd=A->Node;bool flag=1;node *lastasd=asd;int temp=0;
        while((asd->pointer)){
            if (asd->pointer->id == B->id){
                flag=0;break;
            }
        }
    }
}

```

```

        }lastasd=asd;asd=asd->next;if (!asd){break;}temp++; //Вместо if
(!asd){break;} можно в while(asd->pointer) && asd
    }
    int temp1=0;
    asd=B->Node;
    while(asd->pointer){
        if (asd->pointer->id== A->id){
            flag=0;break;
        }lastasd=asd;asd=asd->next;if(!asd){break;}temp1++;
    }
    if (flag){
        A->newNodeSort(weight,temp,B);
        B->newNodeSort(weight,temp1,A);
    }
}

}

void graph::newNodeSort(int weight,int size, graph* pointer ){
    if (size == 0){
        if (this->Node->pointer){
            this->Node->next=new node;this->Node->next->pointer= pointer;
            this->Node->next->weight=
weight;this->Node->next->next=NULL;
        }else{this->Node->pointer= pointer;this->Node->weight= weight;}

    }else{bool flag = false;
        node* asd = this->Node;
        if (asd->weight>=weight){
            node* lastasd= asd;this->Node = new node;
            this->Node->next=lastasd;this->Node->weight=weight;

```

```

        this->Node->pointer=pointer;flag =true;
    }else{node *lastasd= asd;
        for(int i =0;i<(size);i++){
            asd=asd->next;
        }if (asd->weight<weight){
            asd->next=new node;
            asd->next->weight=weight;asd->next->next=NULL;
            asd->next->pointer=pointer;flag =true;
        }
    }if (!flag){
        int i = (size)/2;
        int step =i;
        while(true){asd=this->Node;
            for (int j =0;j<i;j++){
                asd=asd->next;
            }int temp = asd->weight;
            if ((temp <= weight)&& (weight <= asd->next->weight)){
                node* nextasd =asd->next;
                asd->next=new node;asd->next->weight=weight;

asd->next->pointer=pointer;asd->next->next=nextasd;break;
                }else if(temp  < weight) {
                    i+=ceil(float(step)/2);step=ceil(float(step/2));
                }else {i-=ceil(float(step)/2);step=ceil(float (step/2));}
            }
        }
    }
}

```

