数据结构课程设计报告

设计题目：图的算法实现

班　　级：计科 182

学　　号：16817206

姓　　名：朱嘉程

南京农业大学计算机系

数据结构课程设计报告内容

一．课程设计题目

图的算法实现

基本要求:

（1）建立一文件，将图的信息存在此文件中；
（2）从此文件读入图的信息，建立邻接矩阵或邻接表；
（3）实现 Prim、Kruskal、Dijkstra 和拓扑排序算法。

二．算法设计思想

1.Prim 算法

假设G=（V，{E}）是连通网，TE是G上最小生成树中边的集合。算法从U={u0}(u0∈V),TE={}开始，重复执行以下操作：在所有u∈U，v∈V-U的边(u,v)∈E中找一条代价最小的边(u0,v0)并入集合TE，同时v0并入U，直至U=V为止。此时TE中必有n-1条边，则T=(V,{TE})为G的最小生成树。

2.Kruskal 算法

假设G=（V，{E}）是连通网，最小生成树的初始状态为只有n个顶点而无边的非连通图T=(V,{})，图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边。依此类推，直至T中所有顶点都落在同一连通分量上为止。

3.Dijkstra 算法

假设G=(V,{E})是有向图。

(1)用带权的邻接矩阵arcs来表示带权有向图，arcs[i][j]表示弧<vi,vj>上的权值。若<vi,vj>不存在，则置arcs[i][j]为∞。S为已找到从v出发的最短路径的终点的集合，它的初始状态为空集。那么，从v出发到图上其余各顶点(终点)vi可能到达的最短路径长度的初值为:D[i]=arcs[LocateVex(G,v)][i]，vi∈V。

(2)选择vj，使得D[j]=Min{D[i]|vi∈V-S}，vj就是当前求得的一条从v出发的最短路径的终点，令S=S∪{j}。

(3)修改从v出发到集合V-S上任一顶点vk可达的最短路径长度。如果D[j]+arcs[j][k]<D[k]，则D[k]=D[j]+arcs[j][k]。

(4)重复操作(2)(3)共n-1次(n为G的顶点数)。由此求得从v到图上其余各顶点的最短路径是依路径长度递增的序列。

4.拓扑排序算法

(1)求出有向图G中所有顶点的入度，将入度为0的顶点压入栈S

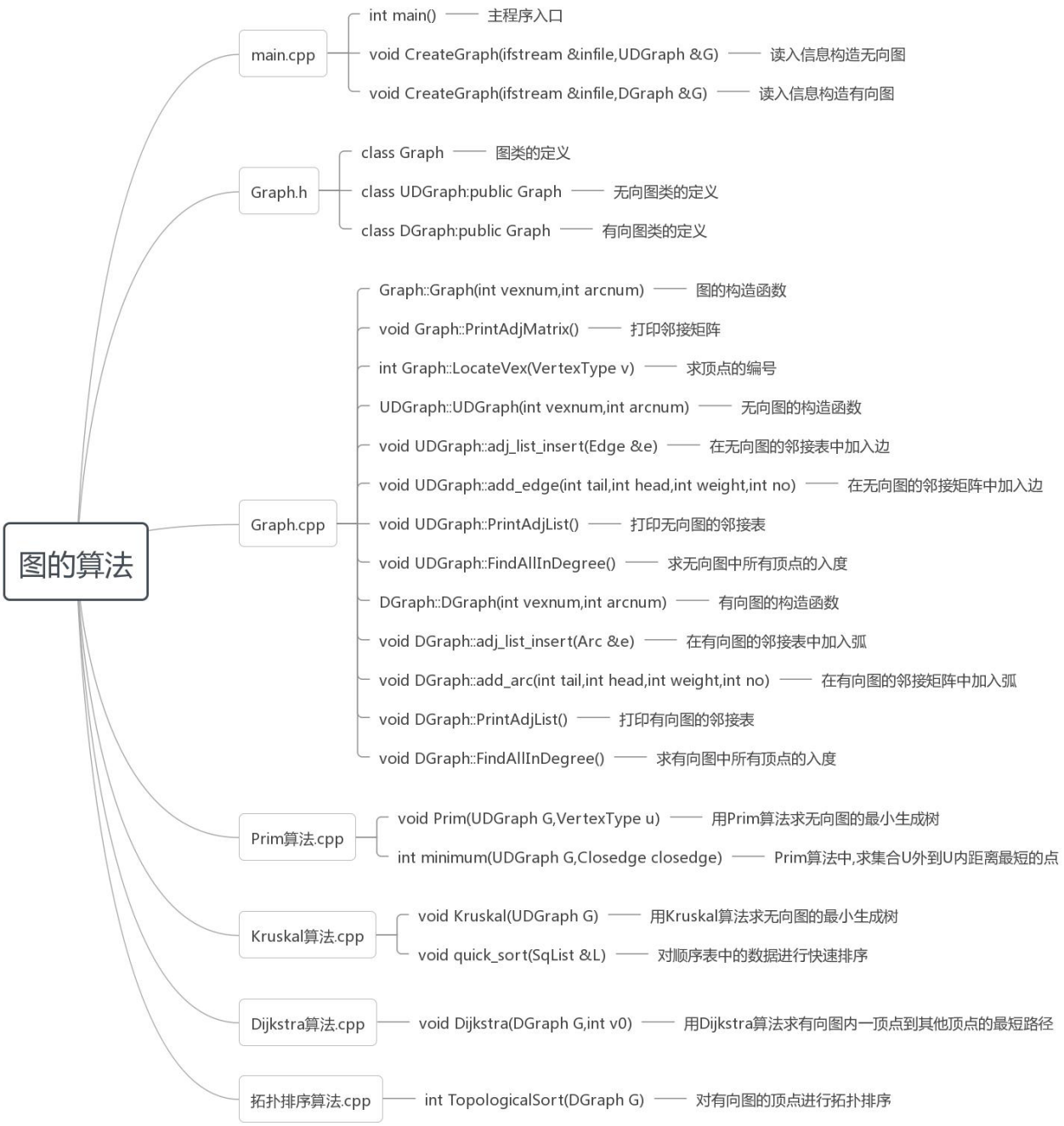
(2)执行以下步骤，直至S为空：

①S的栈顶元素vi出栈，加入拓扑序列的尾端。

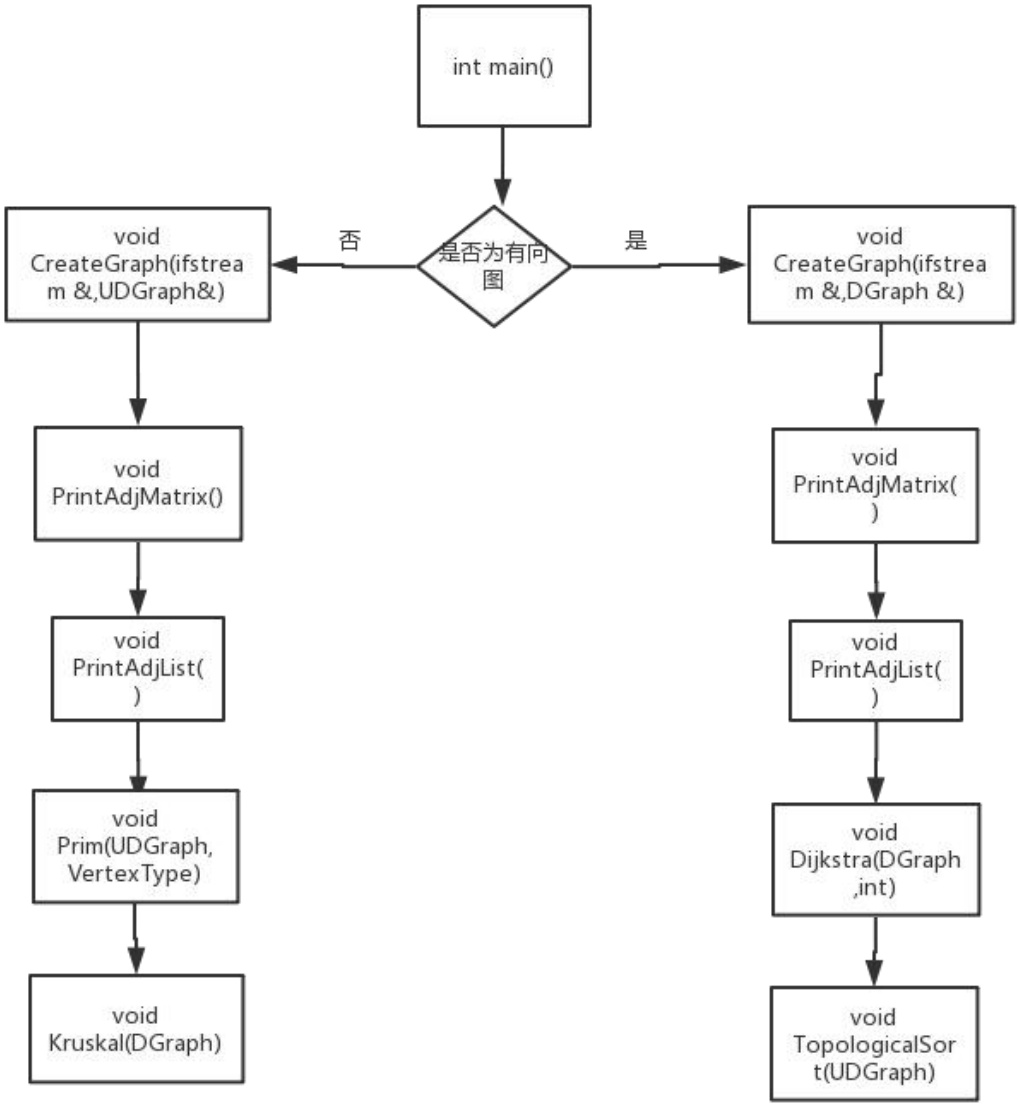
②以vi为弧尾的所有弧的弧头顶点入度减一，若其入度为0则压入栈S。

三．程序结构

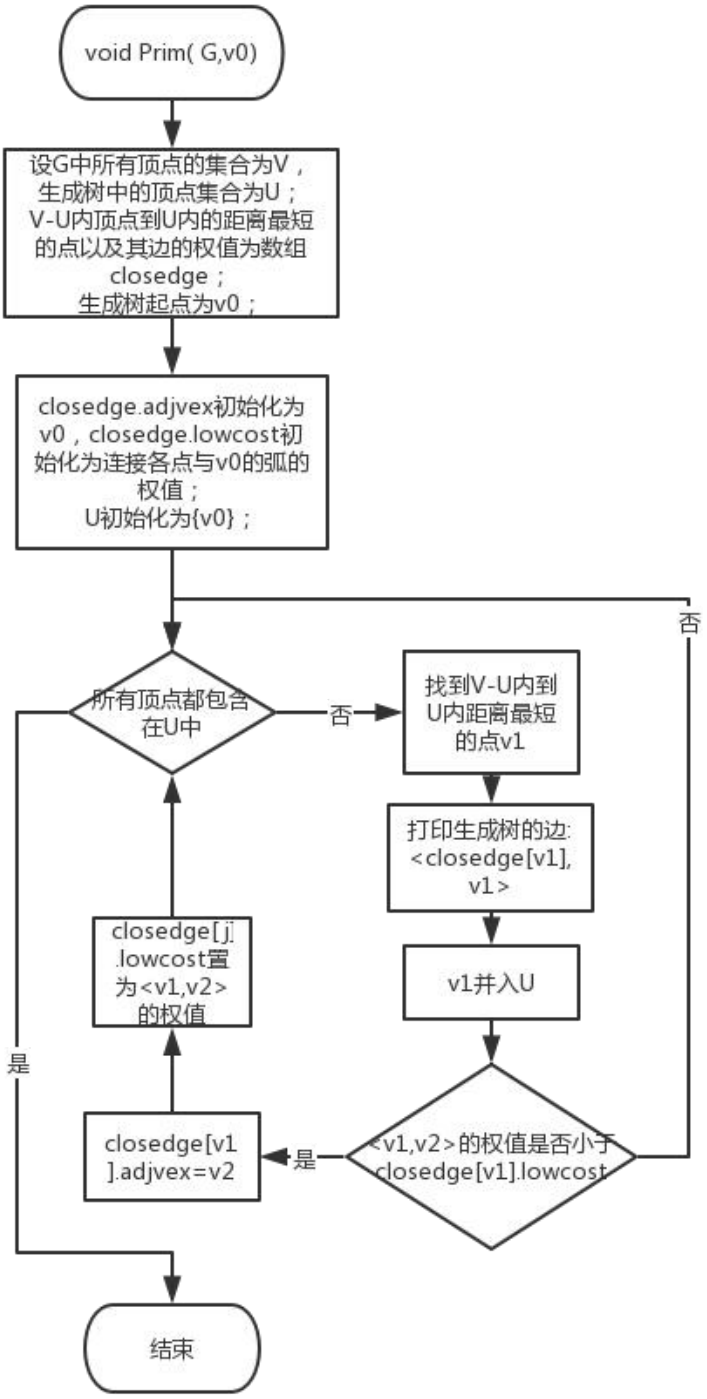
1.各程序模块之间的层次(调用)关系



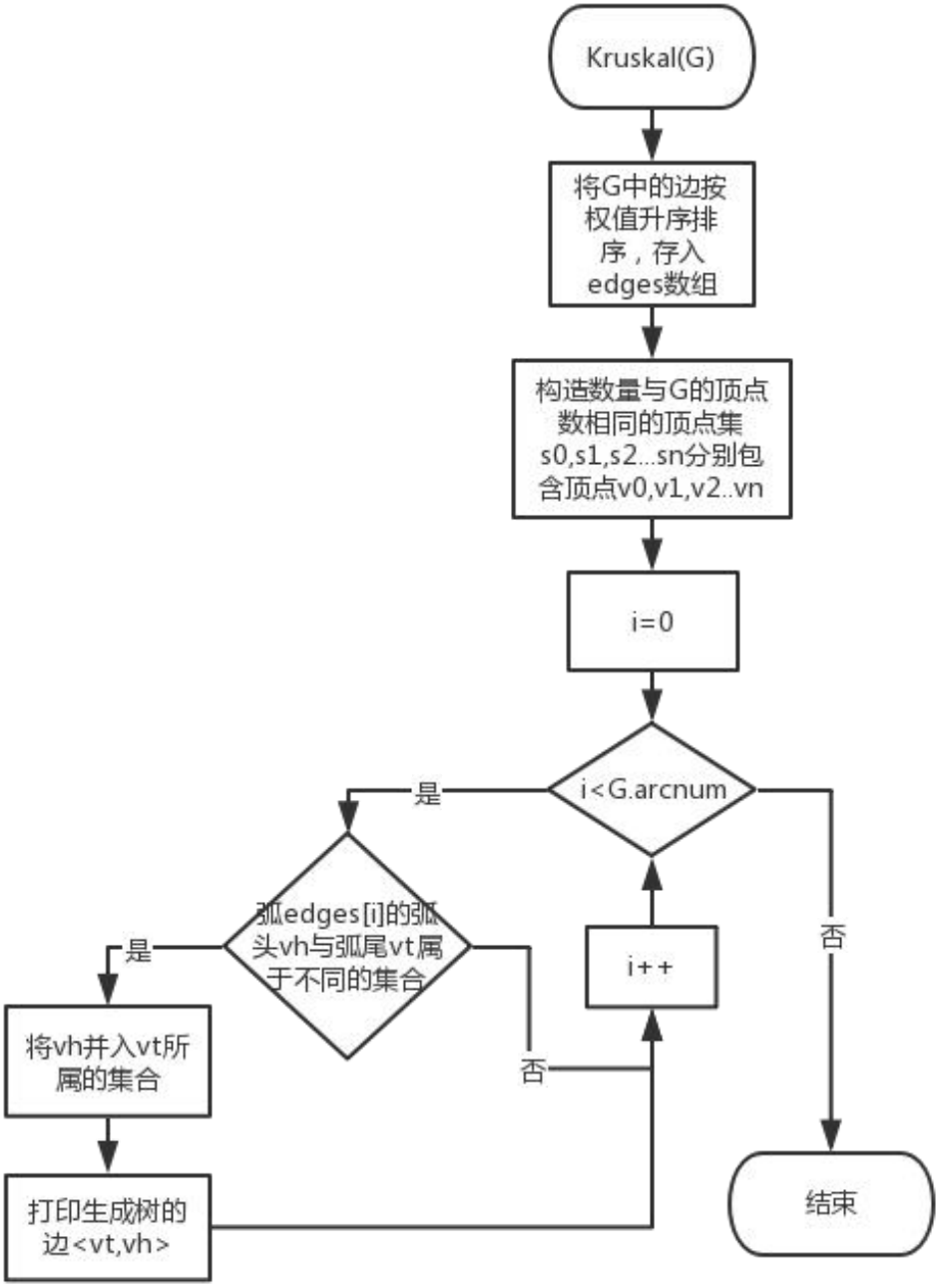
2. 主程序的流程


int main()
是否为有向图
否
是
void CreateGraph(ifstream &,UDGraph&)
void CreateGraph(ifstream &,DGraph &)
void PrintAdjMatrix()
void PrintAdjList()
void Prim(UDGraph, VertexType)
void Kruskal(DGraph)
void PrintAdjMatrix()
void PrintAdjList()
void Dijkstra(DGraph,int)
void TopologicalSort(UDGraph)

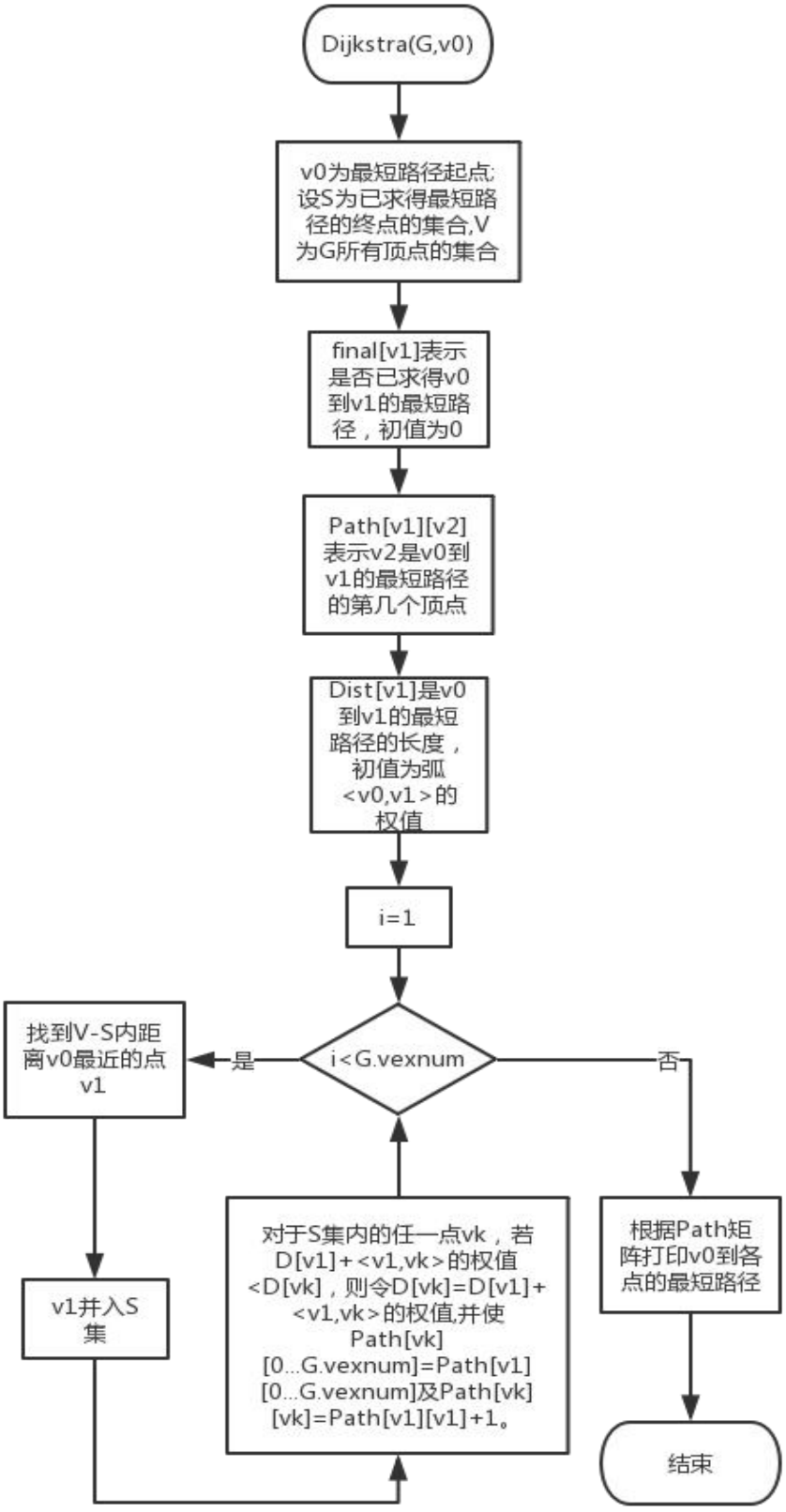
3. Prim 算法.cpp 的流程



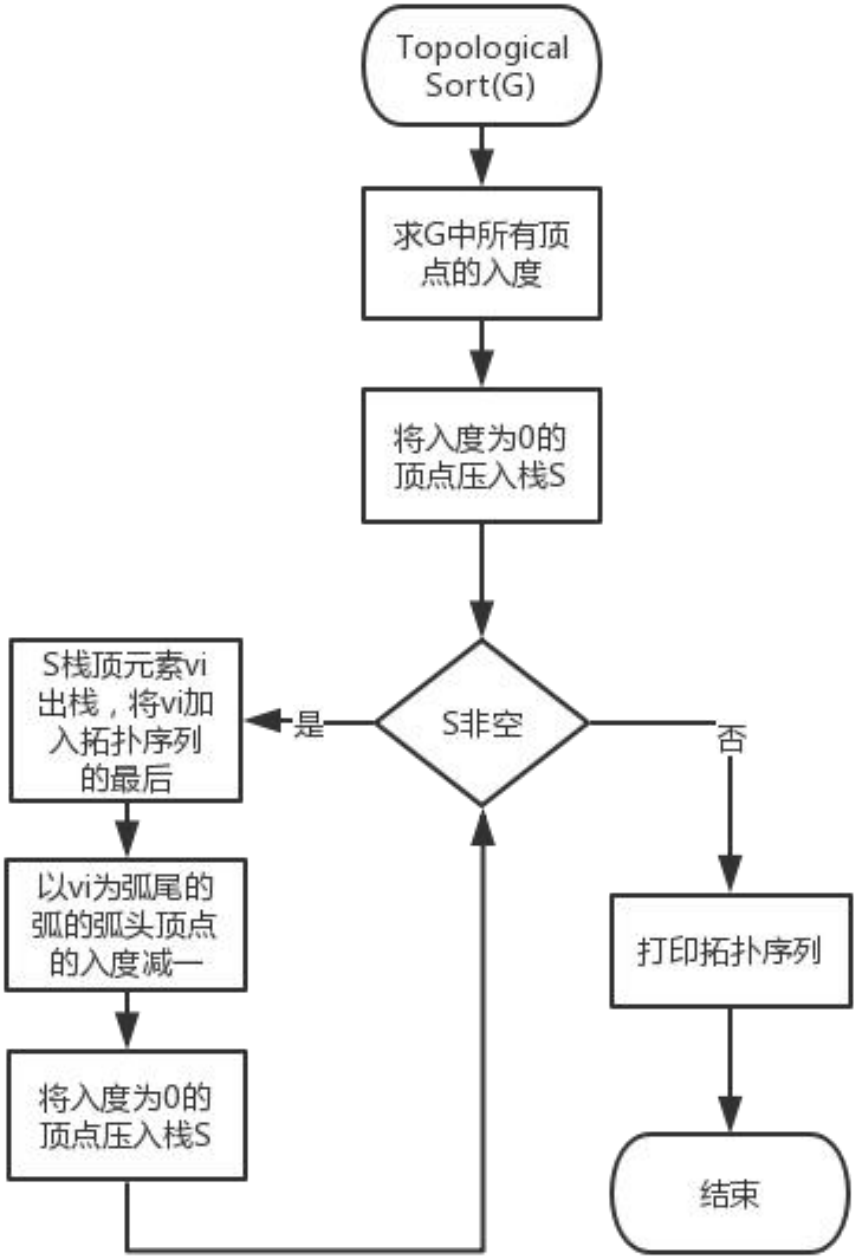
4.Kruskal 算法.cpp 的流程



5.Dijkstra.cpp 的流程



6. 拓扑排序算法. cpp 的流程



四． 实验结果与分析

1.用户使用说明

(1)创建一个名为”input”的 txt 文件，在其中按以下格式输入需要测试的图的信息：

①如果图为无向图，则输入字符 U；如果图为有向图，则输入字符 D。

②换行。

③输入图的顶点数和边/弧数，中间以空格隔开。(例如 5 10 表示图有 5 个顶点，10 条边)

④顺序输入图的每个顶点的名称，中间以空格隔开。(例如 a b c d e 表示图的 0 号到 4 号顶点的名称依此为 a,b,c,d,e)

⑤输入若干行信息，每一行包括一条边的起点、终点的名称和边的权值，中间以空格隔开。(例如 a b 1 表示从顶点 a 到顶点 b 的权值为 1 的边)

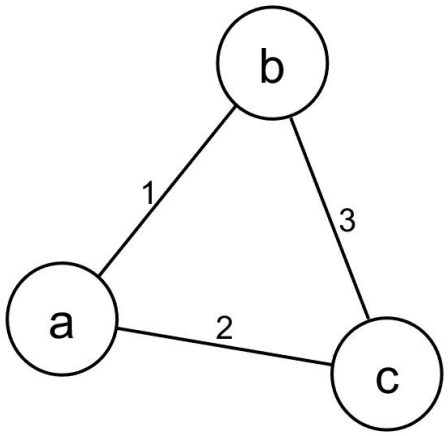
⑥换行。

⑦如果要输入下一个图的信息，回到①；如果所有图的信息已经输入完成，结束。

【示例】：

U
3 3
a b c
a b 1
a c 2
b c 3

表示无向图：



(2)保存 input.txt，将其放到 main.exe 所在的目录下。

(3)运行 main.exe。

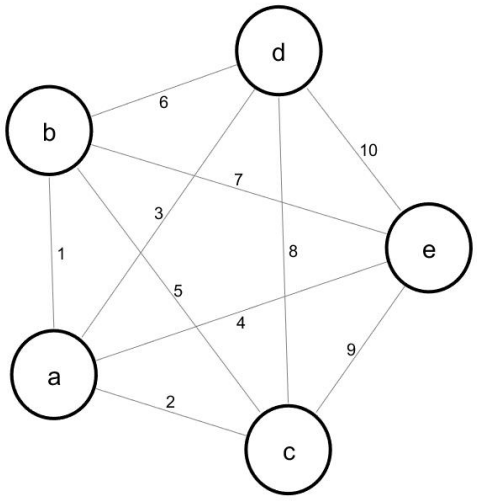
(4)在控制台窗口查看图的邻接矩阵、邻接表以及 Prim、Kruskal、Dijkstra 和拓扑排序算法的结果。

2. 测试结果

输入信息 1:

```
U
5 10
a b c d e
a b 1
a c 2
a d 3
a e 4
b c 5
b d 6
b e 7
c d 8
c e 9
d e 10
```

表示无向图 G1:



测试结果为:

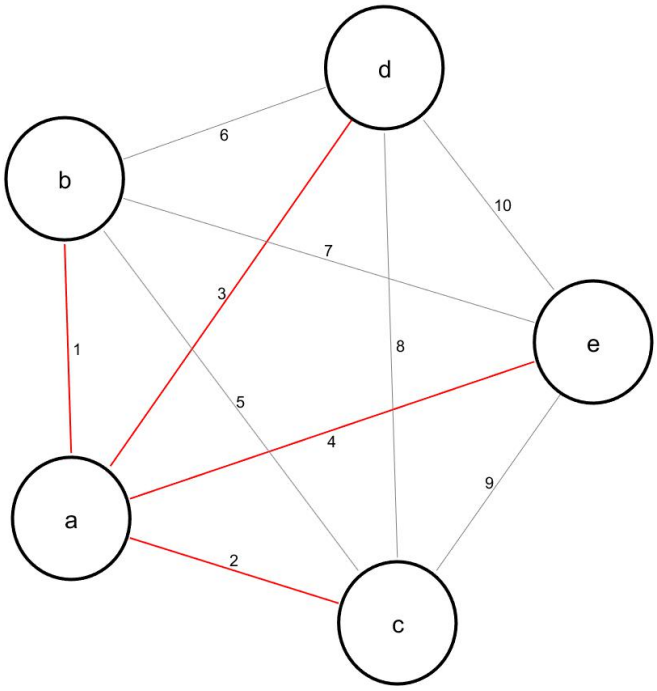
```
图的邻接矩阵为:
          a       b       c       d       e
     a    0       1       2       3       4
     b    1       0       5       6       7
     c    2       5       0       8       9
     d    3       6       8       0      10
     e    4       7       9      10       0
图的邻接表为:
     a    1 1     2 2     3 3     4 4
     b    1 1     2 5     3 6     4 7
     c    2 2     2 5     3 8     4 9
     d    3 3     3 6     3 8     4 10
     e    4 4     4 7     4 9     4 10

Prim算法:

从顶点a开始:
路径    权值
a->b    1
a->c    2
a->d    3
a->e    4
总权值=10

Kruskal算法:
路径    权值
a->b    1
a->c    2
a->d    3
a->e    4
总权值=10
```

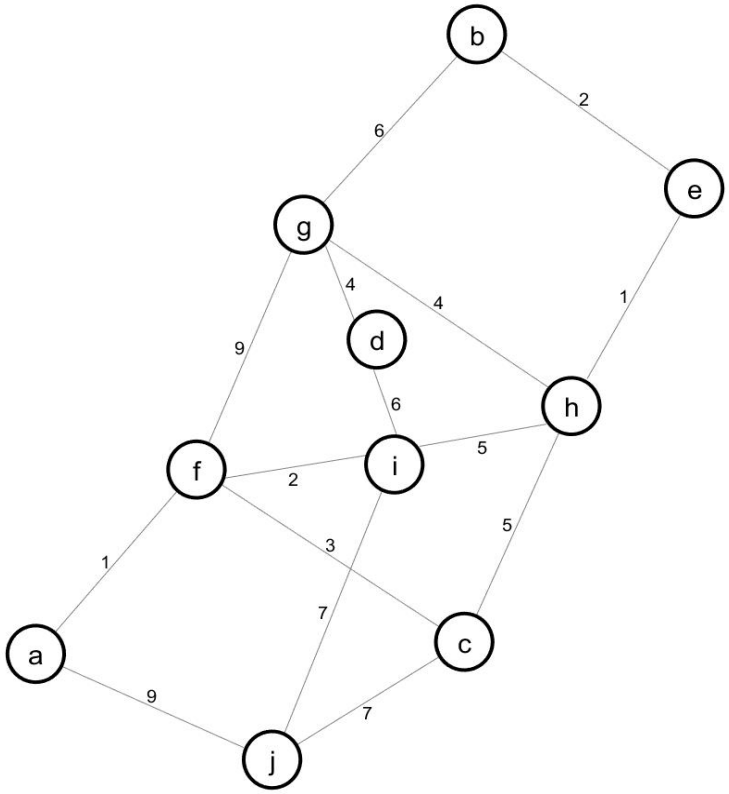
最小生成树为：



输入信息 2：

表示无向图 G2：

```
U
10 15
a b c d e f g h i j
a f 1
a j 9
b e 2
b g 6
c f 3
c h 5
c i 7
d g 4
d i 6
e h 1
f g 9
f i 2
g h 4
h i 5
i j 7
```



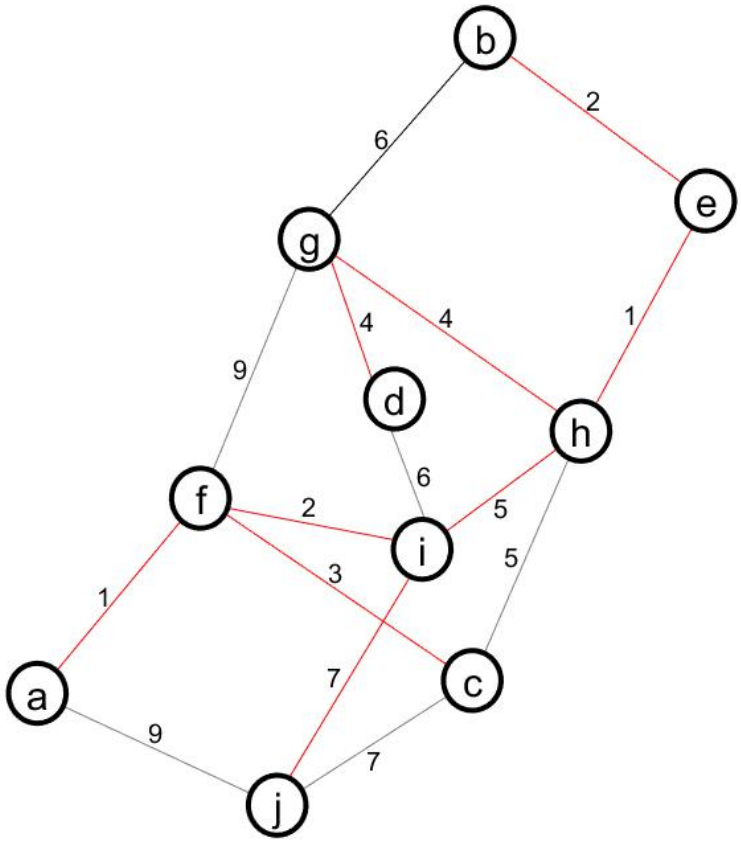
测试结果为:

```
图的邻接矩阵为:
          a      b      c      d      e      f      g      h      i      j
    a     0      0      0      0      0      1      0      0      0      9
    b     0      0      0      0      2      0      6      0      0      0
    c     0      0      0      0      0      3      0      5      7      0
    d     0      0      0      0      0      0      4      0      6      0
    e     0      2      0      0      0      0      0      1      0      0
    f     1      0      3      0      0      0      9      0      2      0
    g     0      6      0      4      0      9      0      4      0      0
    h     0      0      5      0      1      0      4      0      5      0
    i     0      0      7      6      0      2      0      5      0      7
    j     9      0      0      0      0      0      0      0      7      0
图的邻接表为:
    a     5 1      9 9
    b     4 2      6 6
    c     5 3      7 5      8 7
    d     6 4      8 6
    e     4 2      7 1
    f     5 1      5 3      6 9      8 2
    g     6 6      6 4      6 9      7 4
    h     7 5      7 1      7 4      8 5
    i     8 7      8 6      8 2      8 5      9 7
    j     9 9      9 7

Prim算法:

从顶点a开始:
路径    权值
a->f    1
f->i    2
f->c    3
i->h    5
h->e    1
e->b    2
h->g    4
g->d    4
i->j    7
总权值=29
Kruskal算法:
路径    权值
a->f    1
e->h    1
b->e    2
f->i    2
c->f    3
d->g    4
g->h    4
c->h    5
h->i    5
d->i    6
b->g    6
c->i    7
i->j    7
f->g    9
a->j    9
总权值=71
```

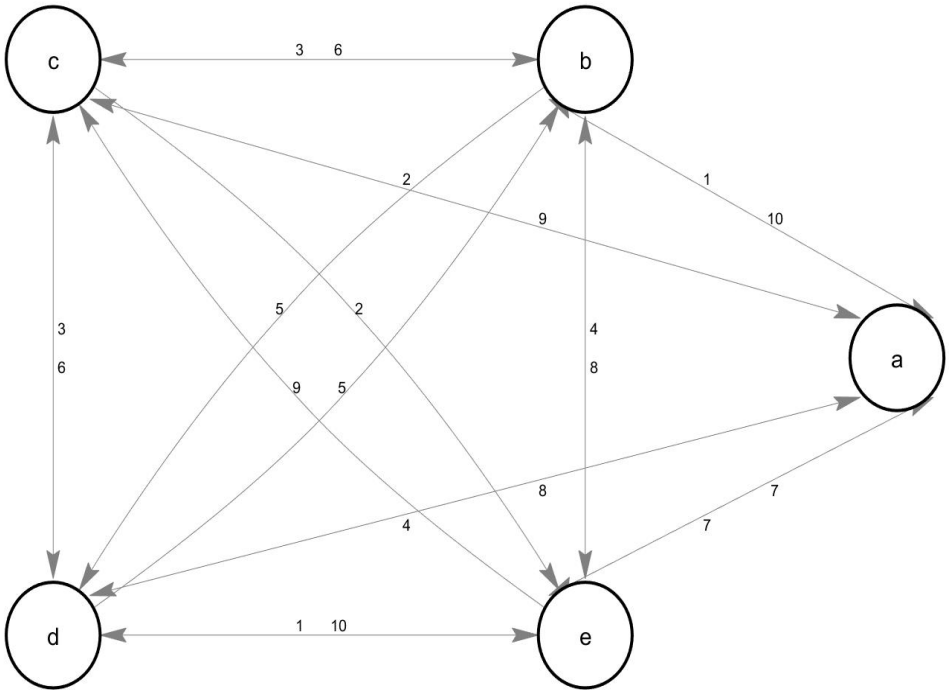
最小生成树为：



输入信息 3：

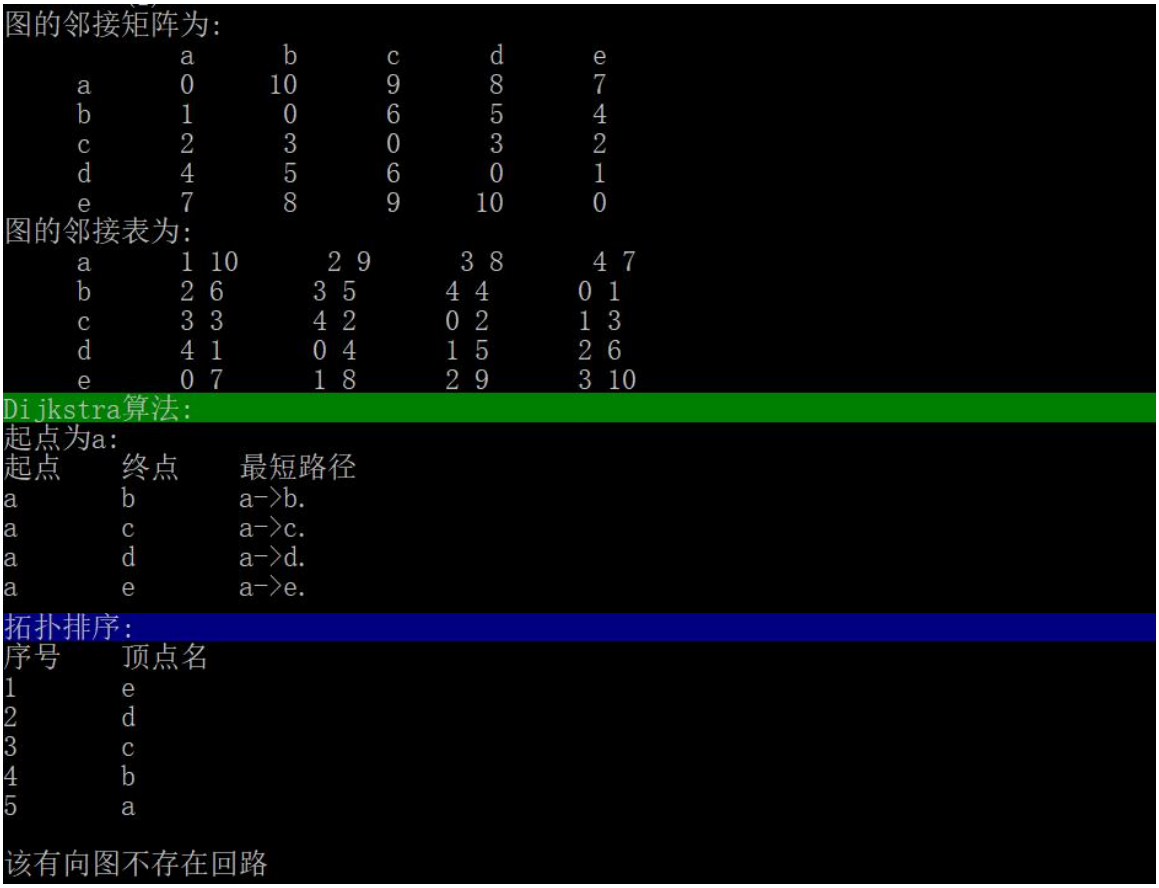
D
5 20
a b c d e
a b 10
a c 9
a d 8
a e 7
b c 6
b d 5
b e 4
c d 3
c e 2
d e 1
b a 1
c a 2
c b 3
d a 4
d b 5
d c 6

表示有向图 G3：



e a 7
e b 8
e c 9
e d 10

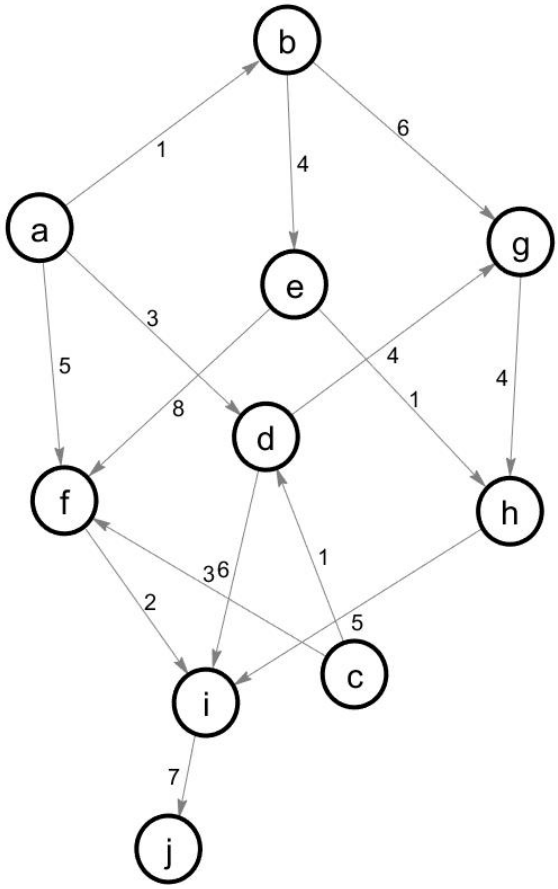
测试结果为:



输入信息 4:

```
D
10 15
a b c d e f g h i j
a b 1
a c 3
a f 5
b e 4
b g 6
c d 1
c f 3
d g 4
d i 6
e f 8
e h 1
f i 2
g h 4
h i 5
i j 7
```

表示有向图 G4:



测试结果为:

```
图的邻接矩阵为:
          a       b       c       d       e       f       g       h       i       j
    a     0       1       3       0       0       5       0       0       0       0
    b     0       0       0       0       4       0       6       0       0       0
    c     0       0       0       1       0       3       0       0       0       0
    d     0       0       0       0       0       0       4       0       6       0
    e     0       0       0       0       0       8       0       1       0       0
    f     0       0       0       0       0       0       0       0       2       0
    g     0       0       0       0       0       0       0       4       0       0
    h     0       0       0       0       0       0       0       0       5       0
    i     0       0       0       0       0       0       0       0       0       7
    j     0       0       0       0       0       0       0       0       0       0
图的邻接表为:
    a       1 1       2 3       5 5
    b       4 4       6 6
    c       3 1       5 3
    d       6 4       8 6
    e       5 8       7 1
    f       8 2
    g       7 4
    h       8 5
    i       9 7
    j
```

```
Dijkstra算法:
起点为a:
起点    终点    最短路径
a       b       a->b.
a       c       a->c.
a       d       a->c,c->d.
a       e       a->b,b->e.
a       f       a->f.
a       g       a->b,b->g.
a       h       a->b,b->e,e->h.
a       i       a->f,f->i.
a       j       a->f,f->i,i->j.
拓扑排序:
序号    顶点名
1       j
2       i
3       h
4       g
5       f
6       e
7       d
8       c
9       b
10      a

该有向图不存在回路
```

3.调试分析

(1)调试过程中遇到的问题：

问题①：

Kruskal 算法中，对有向图的弧按照权值从小到大排列后，导致 G.edges 中弧的顺序不再符合输入的先后次序。

解决方法：

创建一个顺序容器 temp_edges，把排序后新的弧序列放入 temp_edges 里，使得算法执行后不会影响图中的弧原来的顺序。在 Dijkstra 算法.cpp 中，具体作法如下:

```
SqList edge_no_list;
edge_no_list.length=0;
for(int i=0;i<G.arcnum;++i){
    edge_no_list.r[i+1].key=G.edges[i].weight;//把无向图 G 的边的权值装入 sqlist
    edge_no_list.r[i+1].otherinfo=G.edges[i].no; //边的序号装入 sqlist
    edge_no_list.length++;
}
UDGraph::edges_type temp_edges(G.edges);//将原来的 edges 保存起来
quick_sort(edge_no_list);//按照边的权值将 edge_no_list 升序排序
for(int i=0;i<G.arcnum;++i){
    temp_edges[i]=G.edges[edge_no_list.r[i+1].otherinfo];//temp_edges 替换为 edge_no_list 的顺序
}
```

问题②：

Dijkstra 算法中，在打印一点到其他各顶点的最短路径时，无法利用 Path 矩阵直接输出路径上依此经过的各点。

解决方法：

根据 Path 矩阵计算出 Path_ad 矩阵。在 Dijkstra 算法.cpp 中，具体作法如下：

```
for(int v=0;v<G.vexnum;++v){
    for(int w=0;w<G.vexnum;++w){
        if(Path[v][w]>0) Path_ad[v][Path[v][w]]=w;
    }
}
```

则输出 Path_ad[v][1,2,3...]就可以输出从起点到点 v 最短路径上的第 1、2、3...个经过的点。

问题③：

从 txt 文件读入图的信息时，误把有向图当作无向图计算后，每条弧都产生了多余的一条反向弧。

解决方法：

在读入图的其他信息前先读入图的类型，用’U’和‘D’代表无向图和有向图，分别调用不同的重载函数，避免将有向图和无向图混淆。在 main.cpp 中，具体作法如如下:

```
void CreateGraph(ifstream &infile,UDGraph &G){
    int arc_no=0;
        vector<ArcType> v_arc;
        for(int i=0;i<G.vexnum;++i){
            infile>>G.vexs[i];//顶点名称
        }
        for(int i=0;i<G.arcnum;++i){//读入边
            VertexType c1,c2;
            int v1,v2;
            ArcType weight;
            infile>>c1>>c2>>weight;
            v1=G.LocateVex(c1); v2=G.LocateVex(c2);
            G.add_edge(v1,v2,weight,arc_no++);

G.adj_matrix.arc_weight[v1][v2]=G.adj_matrix.arc_weight[v2][v1]=weight;//边是双向的
        }
}
void CreateGraph(ifstream &infile,DGraph &G){
    int arc_no=0;
        vector<ArcType> v_arc;
        for(int i=0;i<G.vexnum;++i){
            infile>>G.vexs[i];//顶点名称
        }
        for(int i=0;i<G.arcnum;++i){//读入边
            VertexType c1,c2;
```

```
            int v1,v2;
            ArcType weight;
            infile>>c1>>c2>>weight;
            v1=G.LocateVex(c1); v2=G.LocateVex(c2);
            G.add_arc(v1,v2,weight,arc_no++);
            G.adj_matrix.arc_weight[v1][v2]=weight;//弧是单向的
        }
    }
```

(2)算法的时间复杂度分析：

① Prim 算法

算法的主要步骤为：

1.将初始顶点 v0 加入到 U 中，对其余每一个顶点 v，将 closedge[v].lowcost 初始化为边<v0,v>的权值。

2.循环 n-1 次

(1)从各组最小边 closedge[v]中选出 closedge[v].lowcost 最小的边 closedge[vk](v,vk 属于 V-U);

(2) 将 closedge[vk].adjvex 加入到 U 中;

(3) 对于所有的最小边 closedge[v](v 属于 V-U)：如果边<vk,v>的权值比 closedge[v].lowcost 小，则将 closedge[v].lowcost 更新为<vk,v>的权值。

其中步骤 1 需要执行 n 次；步骤 2 外层循环执行 n 次，内层的 2 (1)执行 n 次，2 (2)执行 1 次，2 (3)执行 n 次。总执行次数为 T(n)=n+n*(n+1+n)=2n^2+2*n，因此时间复杂度为 O(n^2)。

② Kruskal 算法

算法的主要步骤为：

1.假设 G=（V，{E}）是连通网，最小生成树的初始状态为只有 n 个顶点而无边的非连通图 T=(V,{})，图中每个顶点自成一个连通分量。

2.直至T中所有顶点都落在同一连通分量上为止,执行以下操作：

在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边。

步骤1执行1次；步骤2至多对e条边各扫描一次，若以堆来存放网中的边，则每次选择最小代价的边需要O(loge)的时间。又生成树T的每个连通分量可看成是一个等价类，则构造T加入新的边的过程类似于求等价类的过程，需要O(eloge)的时间。因此时间复杂度为O(eloge)。

③ Dijkstra 算法

算法的主要步骤为：

(1)假设G=(V,{E})是有向图。用带权的邻接矩阵arcs来表示带权有向图，arcs[i][j]表示弧<vi,vj>上的权值。若<vi,vj>不存在，则置arcs[i][j]为∞。S为已找到从v出发的最短路径的终点的集合，它的初始状态为空集。那么，从v出发到图上其余各顶点(终点)vi可能到达的最短路径长度的初值为:D[i]=arcs[LocateVex(G,v)][i]，vi∈V。

(2)选择vj，使得D[j]=Min{D[i]|vi∈V-S}，vj就是当前求得的一条从v出发的最短路径的终点，令S=S∪{j}。

(3)修改从v出发到集合V-S上任一顶点vk可达的最短路径长度。如果D[j]+arcs[j][k]<D[k]，则D[k]=D[j]+arcs[j][k]。

(4)重复操作(2)(3)共n-1次(n为G的顶点数)。由此求得从v到图上其余各顶点的最短路径是依路径长度递增的序列。

步骤(1)需要执行n次；步骤(2)和（3）执行n-1次，每次执行的时间为O(n)。因此时间复杂度为O(n^2)。

④ 拓扑排序算法

(1)求出有向图G中所有顶点的入度，将入度为0的顶点压入栈S

(2)执行以下步骤，直至S为空：

①S的栈顶元素vi出栈，加入拓扑序列的尾端。

②以vi为弧尾的所有弧的弧头顶点入度减一，若其入度为0则压入栈S。

步骤(1)中，图中每条边需要累加一次入度，所以执行次数为边数 e;步骤(2)中，每个顶点都要进栈出栈一次，所以执行的次数为顶点数 2n。所以算法的时间复杂度为 O(n+e)。

(3)算法的改进设想

在 Dijkstra 算法中，需要对有向图的所有弧按权值大小递增排序，目前使用快速排序方法。在最差的情况下，即弧数组中的所有弧已经按权值大小正序或倒序排列，快速排序的时间复杂度为 O(n^2)(n 为弧数)。因此，可以使用堆排序来改进算法，使得在最差的情况下排序的时间复杂度仍然为 O(nlogn)。

具体作法如下:

(1)在 Kruskal.cpp 开头的排序算法中加入堆排序算法：

```
41  void HeapAdjust(SqList &H,int s,int m,int order){
42      DataType rc=H.r[s];
43      if(order==0){//小顶堆
44          for(int j=2*s;j<=m;j*=2){
45              if(j<m && H.r[j].key>H.r[j+1].key) ++j;
46              if(rc.key>H.r[j].key){//堆上元素比堆下元素小,则交换
47                  H.r[s]=H.r[j];
48                  s=j;
49              }
50              else break;
51          }
52      }
53      else if(order==1){//大顶堆
54          for(int j=2*s;j<=m;j*=2){
55              if(j<m && H.r[j].key<H.r[j+1].key) ++j;
56              if(rc.key<H.r[j].key){//堆上元素比堆下元素大,则交换
57                  H.r[s]=H.r[j];
58                  s=j;
59              }
60              else break;
61          }
62      }
63      H.r[s]=rc;
64  }
65  void HeapSort(SqList &H,SqList &L,int order){
66      for(int i=H.length/2;i>0;--i)
67          HeapAdjust(H,i,H.length,order);
68      for(int i=H.length;i>1;--i){
69          L.r[1+H.length-i]=H.r[1];
70          DataType t=H.r[1];
71          H.r[1]=H.r[i];
72          H.r[i]=t;
73          HeapAdjust(H,1,i-1,order);
74      }
75      L.r[H.length]=H.r[1];
76  }
```

⑵ 将 Kruskal 算法.cpp 的第 88 行语句:quick_sort(edge_no_list);
修改为:

```
SqList LT=edge_no_list;
HeapSort(LT,edge_no_list,0);
```

```
75      L.r[H.length]=H.r[1];
76  }
77  void Kruskal(UDGraph G){
78      SqList edge_no_list,temp_list;//为了调用排序函数而，创建存放边序号的临时顺序表
79      edge_no_list.length=0;
80      vector<int>set_no_list;//顶点所在的集合编号表
81      int total_weight=0; //最小生成树的总权值
82      for(int i=0;i<G.arcnum;++i){//注意，sqlist的r[0]是哨兵
83          edge_no_list.r[i+1].key=G.edges[i].weight;//把无向图G的边的权值装入sqlist
84          edge_no_list.r[i+1].otherinfo=G.edges[i].no; //序号装入sqlist
85          edge_no_list.length++;
86      }
87      UDGraph::edges_type temp_edges(G.edges);//将原来的edges保存起来
88  //  quick_sort(edge_no_list);//按照边的权值将edge_no_list升序排序
89      SqList LT=edge_no_list;
90      HeapSort(LT,edge_no_list,0);
91      for(int i=0;i<G.arcnum;++i){
92          temp_edges[i]=G.edges[edge_no_list.r[i+1].otherinfo];//temp_edges替换为edge
93      }
94      for(int i=0;i<G.vexnum;++i){
95          set_no_list.push_back(i);//初始时，每个顶点所在集合的编号都为顶点编号
96      }
97      cout<<"路径\t"<<"权值\n";
98      for(int i=0;i<G.arcnum;++i){
99          int no1,no2,v1=temp_edges[i].tail,v2=temp_edges[i].head;
100         no1=set_no_list[v1];
101         no2=set_no_list[v2];
102         if(no1!=no2){//一条边的两个顶点不属于同一集合
103             set_no_list[no2]=no1;//将其中一个顶点并入另一顶点所在集合
104             total_weight+=temp_edges[i].weight;
105             cout<<G.vexs[v1]<<"->"<<G.vexs[v2]<<"\t"<<temp_edges[i].weight<<endl;//
106         }
107     }
108     cout<<"总权值="<<total_weight<<endl;
```

五． 总结（收获与体会）

在设计算法的过程中，我重新复习了关于最小生成树、数组排序、栈等数据结构的基本算法，加深了对算法的理解。同时在一些 IT 技术网站上查阅并自学了一些算法，锻炼了自我学习能力。

在编写源程序的过程中，我巩固了自己对 C 语言和数据结构的掌握，发现了许多以前学习中忽视的细节。对于一些以前比较薄弱的知识点，比如有向图邻接表的创建、最小生成树的计算、最短路径的求解等有了更深刻的理解，能够不看书上的代码自己编写这些算法。同时，这次实验也促使我学习了一些 C++的思想，例如继承、封装、多态等等，学会用面向对象的方式来编程。发现了使用 C 语言处理某些问题时存在许多不便与局限，因此需要利用 C++的一些方法来增强程序的可移植性与简洁。例如创建了一个图的类，它拥有两个子类无向图与有向图，两者共用图类的一些算法与成员变量，也有各自的不同算法与成员变量。这样就不必在一个图的类中同时包含无向图与有向图各自的算法，增加了程序的简洁程度。

在调试程序的过程中，我独立解决了大量程序错误，发现了自己对有关图的算法有许多地方没有理解透彻。同时使得我学会在 VScode、DevC++等不同的 IDE 中利用单步调试、监视变量等方式找到了错误的来源，并设计修改方案。

总而言之，本次课程设计巩固了我对数据结构基础知识的掌握，也激发了我设计算法的创新能力，同时也提高了我的信息检索能力与自学能力。此外，这次实验加深了我对于本课程学习方法的理解：数据结构的学习重点在于实践，只有自己独立设计编写代码并调试程序，才能完全理解算法的意义。

六． 源程序

1.Graph.h

```
#pragma once
int const MAXINT=32767;//边(弧)的最大权值
int const MVNum=100;//图的最大顶点数量
typedef char VertexType;//顶点的信息类型
typedef int ArcType;//边(弧)的权值类型

class Graph{
//成员变量
    public:
    static int total_number;//创建的图的总数量
    int no=0;//图的编号
    int vexnum=0;//顶点数
    int arcnum=0;//边(弧)的数量
    int indegree[MVNum]={0};//记录每个顶点入度的数组
    VertexType vexs[MVNum];//记录顶点的数组
    struct{
        ArcType arc_weight[MVNum][MVNum];
    }adj_matrix;//邻接矩阵的结构定义
//成员函数
    public:
    Graph(int a,int b);//构造函数
    void add_vertex(VertexType v);//将顶点添加到顶点数组里的函数
    void PrintAdjMatrix();//打印邻接矩阵的函数
    int LocateVex(VertexType v);//由顶点信息找顶点序号的函数
};
int Graph::total_number=0;
class UDGraph:public Graph{
//类型定义
    public:
    typedef struct Edge{
        int no;//编号
        int tail;//顶点 1
```

```
        int head;//顶点 2
        int weight;//权值
        Edge* next_edge=NULL;//指向下一条边的指针
    }Edge;//边的类型定义
    typedef vector<Edge> edges_type;//包含所有边的数组的类型定义
//成员变量
    public:
    edges_type edges;//包含所有边的数组
    struct VNode{
        VertexType data;
        Edge* first_edge=NULL;
    }adj_list_UD[MVNum];//邻接表
    int indegree[MVNum];//记录每个顶点入度的数组
//成员函数
    public:
    UDGraph(int vexnum,int arcnum);//构造函数
    void adj_list_insert(Edge &e);//将边添加到邻接矩阵里的函数
    void add_edge(int tail,int head,int weight,int no);//将边添加到边数组里的函数
    void PrintAdjList();//打印邻接表
    void FindAllInDegree();//计算每个顶点的入度
};
class DGraph:public Graph{
//类型定义
    public:
    typedef struct Arc{
        int no;//编号
        int tail;//弧尾
        int head;  //弧头
        int weight;//弧的权值
        Arc* next_arc=NULL;//指向下一条弧的指针
    }Arc;//弧的类型定义
    typedef vector<Arc>arcs_type;//包含所有弧的数组的类型定义
//成员变量
    public:
    arcs_type arcs;//包含所有弧的数组
    struct VNode{
        VertexType data;
        Arc* first_arc=NULL;
    }adj_list_D[MVNum];//邻接表
//成员函数
    public:
    DGraph(int vexnum,int arcnum);//构造函数
    void adj_list_insert(Arc &e);//将弧添加到邻接矩阵里的函数
    void add_arc(int tail,int head,int weight,int no);//将弧添加到边数组里的函数
    void PrintAdjList();//打印邻接表
    void FindAllInDegree();//计算每个顶点的入度
};
```

2.Graph.cpp

```
#pragma once
#include<iostream>
#include"Graph.h"
using namespace std;
Graph::Graph(int vex_num,int arc_num){
    vexnum=vex_num;
    arcnum=arc_num;
    for(int i=0;i<vexnum;++i){
        for(int j=0;j<vexnum;++j){
            adj_matrix.arc_weight[i][j]=MAXINT;
        }
    }
    Graph::total_number++;
    no++;
}
void Graph::add_vertex(VertexType v){
    vexs[vexnum++]=v;
}
void Graph::PrintAdjMatrix(){
    cout<<setw(6)<<" "<<" ";
    for(int i=0;i<vexnum;++i)
        cout<<setw(6)<<vexs[i]<<" ";
    cout<<endl;
    for(int i=0;i<vexnum;++i){
        cout<<setw(6)<<vexs[i]<<" ";
        for(int j=0;j<vexnum;++j){
            if(adj_matrix.arc_weight[i][j]!=MAXINT)
                cout<<setw(6)<<adj_matrix.arc_weight[i][j]<<" ";
            else
                cout<<setw(6)<<"0"<<" ";
        }
        cout<<endl;
    }
}
int Graph::LocateVex(VertexType v){
    for(int i=0;i<vexnum;++i){
        if(vexs[i]==v)
            return i;
    }
    return -1;
}
UDGraph::UDGraph(int vexnum,int arcnum):Graph(vexnum,arcnum){
}
```

```
void UDGraph::adj_list_insert(Edge &e){
    Edge* p=adj_list_UD[e.tail].first_edge,*pt=p;
    if(!p){
        adj_list_UD[e.tail].first_edge=(Edge*)malloc(sizeof(Edge));
        *(adj_list_UD[e.tail].first_edge)=e;
    }
    else{
        while(p=p->next_edge){
            pt=p;
        }
        pt->next_edge=(Edge*)malloc(sizeof(Edge));
        *(pt->next_edge)=e;;
    }
    Edge* p2=adj_list_UD[e.head].first_edge,*pt2=p2;
    if(!p2){
        adj_list_UD[e.head].first_edge=(Edge*)malloc(sizeof(Edge));
        *(adj_list_UD[e.head].first_edge)=e;
    }
    else{
        while(p2=p2->next_edge){
            pt2=p2;
        }
        pt2->next_edge=(Edge*)malloc(sizeof(Edge));
        *(pt2->next_edge)=e;;
    }
}
void UDGraph::add_edge(int tail,int head,int weight,int no){
    Edge e;
    e.tail=tail;
    e.head=head;
    e.weight=weight;
    e.no=no;
    edges.push_back(e);
    adj_list_insert(e);
}
void UDGraph::PrintAdjList(){
    for(int i=0;i<vexnum;++i){
        cout<<setw(6)<<vexs[i]<<" ";
        Edge* p=adj_list_UD[i].first_edge;
        while(p){
            cout<<p->head<<" "<<left<<setw(4)<<p->weight<<" ";
            p=p->next_edge;
        }
        cout<<endl;
    }
}
void UDGraph::FindAllInDegree(){
```

```
    for(int i=0;i<vexnum;++i){
        for(Edge *p=adj_list_UD[i].first_edge; p; p=p->next_edge){
            ++indegree[p->tail];
            ++indegree[p->head];
        }
    }
}
DGraph::DGraph(int vexnum,int arcnum):Graph(vexnum,arcnum){//构造函数
}
void DGraph::adj_list_insert(Arc &e){
    Arc* p=adj_list_D[e.tail].first_arc,*pt=p;
    if(!p){
        adj_list_D[e.tail].first_arc=(Arc*)malloc(sizeof(Arc));
        *(adj_list_D[e.tail].first_arc)=e;
    }
    else{
        while(p=p->next_arc){
            pt=p;
        }
        pt->next_arc=(Arc*)malloc(sizeof(Arc));
        *(pt->next_arc)=e;;
    }
}
void DGraph::add_arc(int tail,int head,int weight,int no){
    Arc e;
    e.tail=tail;
    e.head=head;
    e.weight=weight;
    e.no=no;
    arcs.push_back(e);
    adj_list_insert(e);
}
void DGraph::PrintAdjList(){
    for(int i=0;i<vexnum;++i){
        cout<<setw(6)<<vexs[i]<<" ";
        Arc* p=adj_list_D[i].first_arc;
        while(p){
            cout<<p->head<<" "<<left<<setw(4)<<p->weight<<" ";
            p=p->next_arc;
        }
        cout<<endl;
    }
}
void DGraph::FindAllInDegree(){
    for(int i=0;i<vexnum;++i){
        for(Arc *p=adj_list_D[i].first_arc; p; p=p->next_arc){
            ++indegree[p->head];
```

```
        }
    }
}
```

3.main.cpp

```
using namespace std;
#include<iostream>
#include<string>
#include<vector>
#include<fstream>
#include<iomanip>
#include<windows.h>
#include"Graph.cpp"
#include"Prim 算法.cpp"
#include"Kruskal 算法.cpp"
#include"Dijkstra 算法.cpp"
#include"拓扑排序算法.cpp"

void CreateGraph(ifstream &infile,UDGraph &G){
    int arc_no=0;
        vector<ArcType> v_arc;
        for(int i=0;i<G.vexnum;++i){
            infile>>G.vexs[i];//顶点名称
        }
        for(int i=0;i<G.arcnum;++i){//读入边
            VertexType c1,c2;
            int v1,v2;
            ArcType weight;
            infile>>c1>>c2>>weight;
            v1=G.LocateVex(c1); v2=G.LocateVex(c2);
            G.add_edge(v1,v2,weight,arc_no++);

    G.adj_matrix.arc_weight[v1][v2]=G.adj_matrix.arc_weight[v2][v1]=weight;//边是
双向的
        }
}
void CreateGraph(ifstream &infile,DGraph &G){
    int arc_no=0;
        vector<ArcType> v_arc;
        for(int i=0;i<G.vexnum;++i){
            infile>>G.vexs[i];//顶点名称
        }
        for(int i=0;i<G.arcnum;++i){//读入边
            VertexType c1,c2;
            int v1,v2;
            ArcType weight;
```

```
            infile>>c1>>c2>>weight;
            v1=G.LocateVex(c1); v2=G.LocateVex(c2);
            G.add_arc(v1,v2,weight,arc_no++);
            G.adj_matrix.arc_weight[v1][v2]=weight;//弧是单向的
        }
}
enum Color { TRANSP=0, DARKBLUE, DARKGREEN, BLUE, RED, DARKPINK, DARKYELLOW, GRAY,
DARKGRAY, LIGHTBLUE, GREEN, TEAL, PINK, PURPLE, YELLOW, WHITE };
void SetColor(Color back_color,Color fore_color){
    HANDLE handle= GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(handle,fore_color|back_color*16);
}
int main(){
    SetConsoleTitle("\\-----图的算法-----\\");
    int n=1,n2=n;//读入矩阵的编号
    ifstream infile("input.txt");
    if(!infile){
      cout<<"error: fail to open \"input.txt\""<<endl;
      return -1;
    }
    while(!infile.eof()){
        char c;
        infile>>c;
        if(c=='U'){
            SetColor(RED,GRAY);
            cout<<">>无向图"<<n<<endl;
            SetColor(TRANSP,GRAY);
            int vexnum,arcnum;
            infile>>vexnum;//顶点数量
            infile>>arcnum;//边的数量
            UDGraph G(vexnum,arcnum);
            CreateGraph(infile,G);
            cout<<"--------("<<n<<")--------"<<endl;
            cout<<"图的邻接矩阵为:"<<endl;
            G.PrintAdjMatrix();
            cout<<"图的邻接表为:"<<endl;
            G.PrintAdjList();
            SetColor(DARKGREEN,GRAY);
            cout<<endl<<"Prim 算法:"<<endl;
            SetColor(TRANSP,GRAY);
            cout<<endl<<"从顶点"<<G.vexs[0]<<"开始:"<<endl;
            Prim(G,G.vexs[0]);
            SetColor(DARKBLUE,GRAY);
            cout<<endl<<"Kruskal 算法:"<<endl;
            SetColor(TRANSP,GRAY);
            Kruskal(G);
            cout<<endl;
```

```
            n++;
        }
        else if(c=='D'){
            SetColor(RED,GRAY);
            cout<<">>有向图"<<n2<<endl;
            SetColor(TRANSP,GRAY);
            int vexnum,arcnum;
            infile>>vexnum;//顶点数量
            infile>>arcnum;//边的数量
            DGraph G2(vexnum,arcnum);
            CreateGraph(infile,G2);
            cout<<"--------("<<n2<<")--------"<<endl;
            cout<<"图的邻接矩阵为:"<<endl;
            G2.PrintAdjMatrix();
            cout<<"图的邻接表为:"<<endl;
            G2.PrintAdjList();
            SetColor(DARKGREEN,GRAY);
            cout<<"Dijkstra 算法:"<<endl;
            SetColor(TRANSP,GRAY);
            cout<<"起点为"<<G2.vexs[0]<<":"<<endl;
            Dijkstra(G2,0);
            cout<<endl;
            SetColor(DARKBLUE,GRAY);
            cout<<"拓扑排序:"<<endl;
            SetColor(TRANSP,GRAY);
            TopologicalSort(G2);
            cout<<endl;
            n2++;
        }
    }
    infile.close();
    cout<<">>按任意键退出程序"<<endl;
    getchar();
}
```

4.Prim 算法.cpp

```
#include<iostream>
#include"Graph.h"
using namespace std;
typedef struct{
    VertexType adjvex;
    int lowcost;
}Closedge[MVNum];
static int minimum(UDGraph G,Closedge closedge){// 求集合 V-U 内到集合 U 内任意一点距离最短的点
    int x=0,x_min,t;//x_min 为到集合 U 内点距离最小的点
```

```
    while(x<G.vexnum && closedge[x].lowcost==0) ++x;//找到第一个不在集合U内的点
    if(x>G.vexnum-1){//若所有点已被包含在集合U里
        cout<<"ERROR"<<endl;
        return -1;
    }
    x_min=x++;//x跳到x_min的下一个顶点
    while(x<G.vexnum ){//在集合V-U中
        if((t=closedge[x].lowcost)!=0 && t<closedge[x_min].lowcost) x_min=x;//若t
小于最小的到集合U内点的距离，更新x_min
        x++;
    }
    return x_min;
}
void Prim(UDGraph G,VertexType u){//u是开始的顶点
    int total_weight=0;
    Closedge closedge;
    int k=G.LocateVex(u);
    for(int j=0;j<G.vexnum;++j){
        if(j!=k){
            closedge[j].adjvex=u;//点j到集合U中任意一点距离最短的是点k（一开始集
合U只包含点k）
            closedge[j].lowcost=G.adj_matrix.arc_weight[k][j];//点j到集合U中任意
一点的最短距离，初始化为点k到点j的弧的权值，若无弧则为MAXINT
        }
    }
    closedge[k].lowcost=0;//点k一开始就存在集合U中
    cout<<"路径\t"<<"权值\n";
    for(int i=1;i<G.vexnum;++i){//循环次数为剩下的顶点数
        k=minimum(G,closedge); //k为到集合外距离最短的点
        total_weight+=closedge[k].lowcost;

    cout<<closedge[k].adjvex<<"->"<<G.vexs[k]<<"\t"<<closedge[k].lowcost<<endl;//
打印生成的边
        closedge[k].lowcost=0;//k号顶点并入U集
        for(int j=0;j<G.vexnum;++j){//遍历所有（其他）点
            if(G.adj_matrix.arc_weight[k][j]<closedge[j].lowcost){//如果新加入的点
k到集合V-U内一点的距离小于此点到集合U内点的最小距离
                closedge[j].adjvex=G.vexs[k];//更新此点到集合U内距离最小的点为点k
                closedge[j].lowcost=G.adj_matrix.arc_weight[k][j];//更新此点到集合
U内点的最小距离为此点到点k的距离
            }
        }
    }
    cout<<"总权值="<<total_weight<<endl;
}
```

5.Kruskal 算法.cpp

```
#include<iostream>
#include"Graph.h"
using namespace std;
#define MAXSIZE 100 //顺序表的最大长度
//-----排序算法-----//
typedef int InfoType;
typedef struct{
    int key; //关键字项
    InfoType otherinfo; //其他数据项
}DataType;
typedef struct{
    DataType r[MAXSIZE+1]; //r[0]闲置或用作哨兵单元
    int length;
}SqList;
int Partition(SqList &L,int low,int high){
    int pivot_key=L.r[low].key;
    L.r[0]=L.r[low];
    while(low<high){
        while(low<high && L.r[high].key>=pivot_key)
            --high;
        L.r[low]=L.r[high];
        while(low<high && L.r[low].key<=pivot_key)
            ++low;
        L.r[high]=L.r[low];
    }
    L.r[low]=L.r[0];
    return low;
}
//快速排序
void q_sort(SqList &L,int low,int high){
    if(low<high){
        int pivot_loc=Partition(L,low,high);
        q_sort(L,low,pivot_loc-1);
        q_sort(L,pivot_loc+1,high);

    }
}
void quick_sort(SqList &L){
    q_sort(L,1,L.length);
}
void Kruskal(UDGraph G){
    SqList edge_no_list;//为了调用排序函数而，创建存放边序号的临时顺序表
    edge_no_list.length=0;
    vector<int>set_no_list;//顶点所在的集合编号表
    int total_weight=0; //最小生成树的总权值
    for(int i=0;i<G.arcnum;++i){//注意，sqlist 的 r[0]是哨兵
```

```
        edge_no_list.r[i+1].key=G.edges[i].weight;//把无向图 G 的边的权值装入 sqlist
        edge_no_list.r[i+1].otherinfo=G.edges[i].no; //序号装入 sqlist
        edge_no_list.length++;
    }
    UDGraph::edges_type temp_edges(G.edges);//将原来的 edges 保存起来
    quick_sort(edge_no_list);//按照边的权值将 edge_no_list 升序排序
    for(int i=0;i<G.arcnum;++i){
        temp_edges[i]=G.edges[edge_no_list.r[i+1].otherinfo];//temp_edges 替换为
edge_no_list 的顺序
    }
    for(int i=0;i<G.vexnum;++i){
        set_no_list.push_back(i);//初始时，每个顶点所在集合的编号都为顶点编号
    }
    cout<<"路径\t"<<"权值\n";
    for(int i=0;i<G.arcnum;++i){
        int no1,no2,v1=temp_edges[i].tail,v2=temp_edges[i].head;
        no1=set_no_list[v1];
        no2=set_no_list[v2];
        if(no1!=no2){//一条边的两个顶点不属于同一集合
            set_no_list[no2]=no1;//将其中一个顶点并入另一顶点所在集合
            total_weight+=temp_edges[i].weight;

    cout<<G.vexs[v1]<<"->"<<G.vexs[v2]<<"\t"<<temp_edges[i].weight<<endl;//打印生
成的路径
        }
    }
    cout<<"总权值="<<total_weight<<endl;
}
```

6.Dijkstra 算法.cpp

```
#include<iostream>
#include"Graph.h"
int const INFINITY=MAXINT+1;
bool final[MVNum];//final[v]为 1 当且仅当点 v 属于 S
int Path[MVNum][MVNum];//Path[v][w]表示点 w 在点 v0 到点 v 的最短路径上的顺序
int Path_ad[MVNum][MVNum];
int Dist[MVNum];//Dist[v]是 v0 到 V-S 集合内点 v 的最短路径的带权长度
void PrintPathMatrix(DGraph G,int v0){
    cout<<"Path["<<G.vexnum<<"]["<<G.vexnum<<"]:"<<endl;//打印 Path[][]
    cout<<setw(6)<<" "<<" ";
    for(int v=0;v<G.vexnum;++v)
        cout<<setw(6)<<G.vexs[v]<<" ";
    cout<<endl;
    for(int v=0;v<G.vexnum;++v){
        cout<<setw(6)<<G.vexs[v]<<" ";
        for(int w=0;w<G.vexnum;++w){
```

```
            cout<<setw(6)<<Path[v][w]<<" ";
        }
        cout<<endl;
    }
    cout<<"Path_ad["<<G.vexnum<<"]["<<G.vexnum<<"]:"<<endl;
    cout<<setw(6)<<" "<<" ";
    for(int v=0;v<G.vexnum;++v)
        cout<<setw(6)<<v<<" ";
    cout<<endl;
    for(int v=0;v<G.vexnum;++v){//打印 Path_ad[][]
        cout<<setw(6)<<G.vexs[v]<<" ";
        for(int w=0;w<G.vexnum;++w){
            cout<<setw(6);
            if(Path_ad[v][w]>=0)
                cout<<G.vexs[Path_ad[v][w]];
            else if(w==0)
                cout<<G.vexs[v0];
            else
                cout<<" ";
            cout<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
}
void PrintDistMatrix(DGraph G){
    cout<<"Dist["<<G.vexnum<<"]:"<<endl;
    for(int v=0;v<G.vexnum;++v)
        cout<<setw(6)<<G.vexs[v]<<" ";
    cout<<endl;
    for(int v=0;v<G.vexnum;++v)
        cout<<setw(6)<<Dist[v]<<" ";
}
void Dijkstra(DGraph G,int v0){
    int v;
    for(int i=0;i<G.vexnum;++i)
        for(int j=0;j<G.vexnum;++j)
            Path_ad[i][j]=-1;
    Path[G.vexnum][G.vexnum]=-1;
    for(v=0;v<G.vexnum;++v){ //初始化
        final[v]=0;
        Dist[v]=G.adj_matrix.arc_weight[v0][v];//点 v0 到点 v 的带权长度初始化为
<v0,v>的权值
        for(int w=0;w<G.vexnum;++w) Path[v][w]=-1;
        if(Dist[v]<INFINITY){
            Path[v][v0]=0;//第一个点为 v0 本身
            Path[v][v]=1;//第二个点为 v
```

```
        }
    }
    Dist[v0]=0;//点 v0 到点 v0 距离为 0
    final[v0]=1;
    //Path[v0][v0]=1;
    for(int i=1;i<G.vexnum;++i){
        int min=INFINITY;
        for(int w=0;w<G.vexnum;++w){
            if(!final[w])
                if(Dist[w]<min){
                    v=w;//找到距离最近的点 v
                    min=Dist[w];
                }
        }
        final[v]=1;
        for(int w=0;w<G.vexnum;++w){
            if(!final[w] && (min+G.adj_matrix.arc_weight[v][w]<Dist[w])){//点 v0 到
点 v 的最短距离+<v,w>小于点 v0 到点 w 的最短距离
                Dist[w]=min+G.adj_matrix.arc_weight[v][w];
                for(int t=0;t<G.vexnum;++t)//点 v0 到点 w 路径上存在的点变为点 v0 到点
v 路径上存在的点加上点 v 和点 w
                    Path[w][t]=Path[v][t];
                Path[w][w]=Path[v][v]+1;//w 是 v 后面的一个点
            }
        }
    }
    for(int v=0;v<G.vexnum;++v){//求 Path_ad[][]
        for(int w=0;w<G.vexnum;++w){
            if(Path[v][w]>0) Path_ad[v][Path[v][w]]=w;
        }
    }
    cout<<"起点\t"<<"终点\t"<<"最短路径\t"<<"权值\n";
    for(int v=0;v<G.vexnum;++v){
        if(v==v0) continue;
        cout<<G.vexs[v0]<<"\t"<<G.vexs[v]<<"\t";
        int vt=v0; bool flag=0;
        for(int w=1;w<G.vexnum;++w){
            if(Path_ad[v][w]>=0){
                flag=1;
                cout<<G.vexs[vt]<<"->"<<G.vexs[Path_ad[v][w]]<<',';
                vt=Path_ad[v][w];
            }

        }
        cout<<"\t"<<Dist[v]<<" ";
        if(flag) cout<<"\b.";
        cout<<endl;
```

```
    }
}
```

7.拓扑排序算法.cpp

```
#include"Graph.cpp"
#include <stack>
int TopologicalSort(DGraph G){
    int *indegree=(int*)malloc(G.vexnum*sizeof(int)),count=0;
    stack<int> S;
    for(int i=0;i<G.vexnum;++i)//初始化;
        indegree[i]=0;
    G.FindAllInDegree();//求所有顶点入度
    for(int i=0;i<G.vexnum;++i)
        if(!indegree[i])//如果一个顶点的入度为0，则压进栈
            S.push(i);
    if(!S.empty())
        cout<<"序号\t 顶点名"<<endl;
    while(!S.empty()){
        int i;
        i=S.top();
        S.pop();
        ++count;
        cout<<count<<"\t"<<G.vexs[i]<<endl;
        for(DGraph::Arc* p=G.adj_list_D[i].first_arc; p; p=p->next_arc){
            if(!(--indegree[p->head]))//以该顶点为弧尾的弧的弧头顶点入度减 1
                S.push(p->head);
        }
    }
    if(count<G.vexnum){//有顶点未被遍历到
        cout<<"\n 该有向图的顶点";
        for(int i=0;i<G.vexnum;++i){
            if(indegree[i]>0)
                cout<<G.vexs[i]<<" ";
        }
        cout<<"\b 间存在回路\n";
        return -1;
    }
    else{
        cout<<"\n 该有向图不存在回路\n";
        return 1;
    }
}
```