电网建设造价模拟系统

作 者 姓 名： 王凌

学 号： 1951504

指 导 教 师： 张颖 \_\_

学院、 专业： 软件工程 \_\_

同济大学

Tongji University

目录

[1 分析 2](#_Toc6393)

[1.1 项目简介 2](#_Toc11390)

[1.2 功能分析 2](#_Toc14590)

[2 设计 3](#_Toc20933)

[2.1 Edge类的设计 3](#_Toc21071)

[2.2 Graph类的设计 3](#_Toc7939)

[2.3 UF类的设计 4](#_Toc287)

[2.4 primTree函数的设计 4](#_Toc10390)

[2.5 主函数的设计 5](#_Toc5503)

[3实现 5](#_Toc32457)

[3.1 Edge类的实现 5](#_Toc25865)

[3.2 Graph类的实现 5](#_Toc30054)

[3.2.1 Graph(int num)自定义构造函数 6](#_Toc16038)

[3.2.2 GetFirstNeighbor()与GetNextNeighbor()查找结点函数 6](#_Toc26828)

[3.2.3 InsertEdge(string Node1,string Node2,int weight)插边函数 7](#_Toc14807)

[3.2.4对外接口函数 8](#_Toc29872)

[3.3 UF类的实现 9](#_Toc9446)

[3.3.1 UF(int N)自定义构造函数 9](#_Toc29173)

[3.3.2 find(int p)寻找父节点和路径压缩函数 9](#_Toc12768)

[3.3.3 connected(int p,int q)、unionPQ(int p,int q)检测与连接函数 10](#_Toc31812)

[3.3.4接口函数getCount() 10](#_Toc16475)

[3.4 primTree函数的实现 10](#_Toc1343)

[3.5主函数的实现 12](#_Toc21061)

[4 测试 15](#_Toc27279)

[4.1 功能测试 15](#_Toc27729)

[4.1.1 测试1 15](#_Toc27930)

[4.1.2 测试2 17](#_Toc26310)

# 1 分析

## 1.1 项目简介

假设一个城市有n个小区，要实现n个小区之间的电网都能够相互接通，构造这个城市n个小区之间的电网，使总工程造价最低。请设计一个能够满足要求的造价方案。 在每个小区之间都可以设置一条电网线路，都要付出相应的经济代价。n个小区之间最多可以有n（n-1）/2条线路，选择其中的n-1条使总的耗费最少。

## 1.2 功能分析

首先，n个小区可以简化为n个结点；其次，电网线路含有点的连通与经济代价两个信息，所以可以简化为一条有权值的边，这样，这个问题场景就可以简化为拓扑结构中的加权无向图模型。

因此选择其中(n-1)条使总的耗费最少的问题可以简化为求一个加权无向图的最小生成树的问题。

我用了Prim算法。在算法中我首先定义了Graph类来存储加权无向图，之后定义了Edge类来记录边的起点、终点和边的权值，还定义了UF（并查集）类来判断图是否连通。为了使结点的名字与序号相对应，我调用了<unordered\_map>库来记录int类结点序号和string类结点名字的对应关系。

# 2 设计

## 2.1 Edge类的设计

Edge类代表了图中的边，它有三个成员变量：边的两个顶点head，tail与权值weight。

## 2.2 Graph类的设计

私有成员变量包含一个int类的二维数组mapp来记录邻接矩阵，和两个int类nodeNum和edgeNum来记录结点和边的个数。

公有成员变量含有一个可联系string类名字和int类序号的 unordered\_map类变量node\_name。

公有函数及其作用如下：

Graph(int num):nodeNum(num),edgeNum(0)是自定义构造函数。构造函数将给定顶点个数num，并且让边数为0，而且为邻接矩阵的num\*num个空间进行初始化为INF，它的意义是每个结点都是一个简单图，都没有连通。

int GetFirstNeighbor(int v)函数用来返回给出顶点位置为v的第一个邻接顶点的位置 。

int GetNextNeighbor(int v1,int v2)用来返回顶点v1的某邻接顶点v2的下一个邻接顶点 。

bool InsertEdge(string Node1,string Node2,int weight)是插入边的函数，首先要判断输入结束字符“？ ？ 0”的情况，还要排除两个结点相同和变得权值为负的情况。排除完这两种的情况之后，通过map红黑树来寻找结点名字为Node1和Node2的序号。同时，如果没有找到其中一个结点或者两个结点之间已经有了路径，要报错退出。排除这两个错误之后，边数edgeNum加一，并且在邻接矩阵中记录序号为left\_Index，right\_Index的边的权值。由于是无向边，所以坐标为（left\_Index，right\_Index）和（right\_Index，left\_Index）的两个位置都需要记录权值。

由于面向对象的封装特性，定义下面三个个对外接口：int GetWeight(int left\_Index,int right\_Index)函数用来返回边的权值，int GetNodeNum()用来返回结点数目，int GetEdgeNum()用来返回边的个数。

私有成员函数有一个——bool isConnected()，用来判断是否连通。判断结点是否连通的做法很简单，遍历mapp数组，如果两个结点之间有边相连，则把两个结点放到一个集合之中。最终判断总的连通分量的数量如果为1，则连通；如果数量大于1，则不连通。

## 2.3 UF类的设计

这是并查集类。它有两个个公有成员变量，id存储每个顶点的父节点，count记录当前连通分量数量。

公有成员函数设计如下：

UF(int N)是自定义构造函数。将当前所以节点的父节点赋值为他们本身；并且让当前连通分量数量count等于顶点总数N，这代表了当前有N个连通分量，每两个顶点之间都互不连通。

int find(int p)用来寻找当前结点p的父节点。当p!=id[p]时进行while循环的意思是如果p元素的父亲指针指向的不是自己，说明p并不是集合中的根元素，还需要一直向上查找和路径压缩，直至循环结束。与此同时，我还在在find查询中嵌入一个路径压缩的操作，即id[p]=id[id[p]]，意思是p元素不再选择原来的父亲节点，而是直接选择父亲节点的父亲节点来做为自己新的一个父亲节点。这样的操作使得树的层数被压缩了，减小了运算规模。

bool connected(int p,int q)用来检测是不是在同一个集合里，如果p,q的父节点相等，则在同一个集合之中。

void unionPQ(int p,int q)用来连接p、q两个结点，将它们放进一个集合之中。

getCount()用来返回当前连通分量数量count。

## 2.4 primTree函数的设计

定义两个辅助数组——lowCost与nearVex数组。lowCost数组存放生成树顶点集合内顶点到生成树外各个顶点的各边上的当前最小权值；nearVex数组记录生成树顶点集合外各顶点距离集合内哪个顶点最近。

初始化时除了首个加入生成树的节点lowCost[0]=0和nearVex[0]=-1外，其余lowCost[i]定义为mapp[0][i]，nearVex[i]=0。

之后用for循环找出未连接结点中到当前生成树距离最小的结点，将之移至生成树中，如此重复，直到所有结点都进入生成树中。此时的生成树就是最小生成树。

## 2.5 主函数的设计

先调用PrintScreen()在输入窗口给出操作面板，之后创建一个map将用户输入的结点名字和节点序号相互对应。然后进入循环等待用户进行操作，直到isContinue为false的时候停止while循环。

如果用户选择操作A，就让用户输入num个顶点和他们的名字，然后程序会生成一个含有num个结点且不含有边的图。

如果选择B操作，将不断循环，将用户输入的正确的两个顶点与边的权值加入到加权无向图中。当用户输入“？ ？ 0”时，退出当前输入边的循环。

如果选择C操作，则调用primTree()函数，生成最小生成树

如果选择D操作，将遍历存放Edge类的不定长数组edgeVector，并且在map类型变量node\_name找到每一个Edge类边的head、tail结点序号所对应的名字，最终显示最小生成树。

# 3实现

## 3.1 Edge类的实现

Edge类代表了图中的边，它有三个成员变量：边的两个顶点head，tail与权值weight。

1. **class** Edge
2. {
3. **public**:
4. **int** head;
5. **int** tail;
6. **int** weight;
7. };

## 3.2 Graph类的实现

私有成员变量包含一个int类的二维数组mapp来记录邻接矩阵，和两个int类nodeNum和edgeNum来记录结点和边的个数；公有成员变量含有一个可联系string类名字和int类序号的 unordered\_map类变量node\_name。

私有成员函数包括一个判断图是否连通的函数isConnected()；公有成员函数包括自定义构造函数Graph(int num)，给出结点序号为v的下一个邻接结点位置的函数GetFirstNeighbor(int v)，给出结点序号为v1的某个邻接结点v2的下一个邻接结点的函数GetNextNeighbor(int v1,int v2)，插入边的函数InsertEdge(string Node1,string Node2,int weight)，还有三个封装用的对外接口函数，分别是返回边的权值的函数GetWeight，返回结点数的函数GetNodeNum，返回边数的函数GetEdgeNum。

### 3.2.1 Graph(int num)自定义构造函数

构造函数将给定顶点个数num，并且让边数为0，而且为邻接矩阵的num\*num个空间进行初始化为INF，它的意义是每个结点都是一个简单图，都没有连通。

1. Graph(**int** num):nodeNum(num),edgeNum(0)//自定义构造函数
2. //生成结点数为num，边数为0的图
3. {
4. **for**(**int** i=0;i<num;i++)
5. **for**(**int** j=0;j<num;j++) mapp[i][j]=INF;
6. }

### 3.2.2 GetFirstNeighbor()与GetNextNeighbor()查找结点函数

int GetFirstNeighbor(int v)函数用来返回给出顶点位置为v的第一个邻接顶点的位置；int GetNextNeighbor(int v1,int v2)用来返回顶点v1的某邻接顶点v2的下一个邻接顶点。

1. **int** GetFirstNeighbor(**int** v)
2. //给出顶点位置为v的第一个邻接顶点的位置
3. {
4. **for**(**int** i=0;i<nodeNum;i++)
5. **if**(mapp[v][i]!=INF) **return** i;
6. **return** -1;
7. }
8. **int** GetNextNeighbor(**int** v1,**int** v2)
9. //给出顶点v1的某邻接顶点v2的下一个邻接顶点
10. {
11. **bool** v2\_found=**false**;//记录结点有没有被找到
12. **for**(**int** i=0;i<nodeNum;i++)
13. {
14. **if**(v2\_found) **if**(mapp[v1][i]!=INF) **return** i;
15. //v2被找到之后的下一个结点
16. **if**(i==v2) v2\_found=**true**;
17. }
18. **return** -1;
19. }

### 3.2.3 InsertEdge(string Node1,string Node2,int weight)插边函数

bool InsertEdge(string Node1,string Node2,int weight)是插入边的函数，首先要判断输入结束字符“？ ？ 0”的情况，还要排除两个结点相同和变得权值为负的情况。排除完这两种的情况之后，通过map红黑树来寻找结点名字为Node1和Node2的序号。同时，如果没有找到其中一个结点或者两个结点之间已经有了路径，要报错退出。排除这两个错误之后，边数edgeNum加一，并且在邻接矩阵中记录序号为left\_Index，right\_Index的边的权值。由于是无向边，所以坐标为[left\_Index，right\_Index]和[right\_Index，left\_Index]的两个位置都需要记录权值。

1. **bool** InsertEdge(string Node1,string Node2,**int** weight)//插入边
2. {
3. **if**((Node1=="?"||Node1=="？") && (Node2=="?"||Node2=="？")&& weight==0)
4. //中英文输入下的?都能被识别
5. {
6. **if**(isConnected()) **return** **true**;
7. **else**
8. {
9. cout<<"您的各个电网之间并未全部连通，请确认您的节点是否正确\n";
10. **return** **false**;
11. }
12. }
13. **if**(Node1==Node2)//是同一个结点
14. {
15. cout<<"两个结点信息相同，请确认后在输入\n";
16. **return** **false**;
17. }
18. **if**(weight<=0)//边的权值必须为正
19. {
20. cout<<"两个城市线路的花费不能小于等于0，请重新确认后输入\n";
21. **return** **false**;
22. }
23. **int** left\_Index=0,right\_Index=0;
24. //分别记录名字为Node1,Node2的结点标号
25. //红黑树map库中的find函数
26. **if**(node\_name.find(Node1)!=node\_name.end()) left\_Index=node\_name.find(Node1)->second;
27. **if**(node\_name.find(Node2)!=node\_name.end()) right\_Index=node\_name.find(Node2)->second;
28. **if**(node\_name.find(Node1)==node\_name.end() || node\_name.find(Node2)==node\_name.end())
29. {
30. cout<<"未找到其中一个结点，请重新输入\n";
31. **return** **false**;
32. }
33. **if**(mapp[left\_Index][right\_Index]!=INF)
34. {
35. cerr<<"两个结点间已有路径，请检查您的节点是否输入正确\n";
36. **return** **false**;
37. }
38. //无向图，所以邻接矩阵是对称的
39. mapp[left\_Index][right\_Index]=weight;
40. mapp[right\_Index][left\_Index]=weight;
41. edgeNum++;
42. **return** **false**;
43. }

### 3.2.4对外接口函数

由于面向对象的封装特性，定义下面三个个对外接口：int GetWeight(int left\_Index,int right\_Index)函数用来返回边的权值，int GetNodeNum()用来返回结点数目，int GetEdgeNum()用来返回边的个数。

1. //由于面向对象的封装特性，定义下面几个对外接口
2. **int** GetWeight(**int** left\_Index,**int** right\_Index)
3. {
4. **return** mapp[left\_Index][right\_Index];
5. }
6. **int** GetNodeNum() {**return** nodeNum;}
7. **int** GetEdgeNum() {**return** edgeNum;}

## 3.3 UF类的实现

### 3.3.1 UF(int N)自定义构造函数

UF(int N)是自定义构造函数。将当前所以节点的父节点赋值为他们本身；并且让当前连通分量数量count等于顶点总数N，这代表了当前有N个连通分量，每两个顶点之间都互不连通。

1. UF(**int** N)//初始化
2. {
3. **for**(**int** i=0;i<N;i++) id.push\_back(i);
4. count=N;//此时每个点都是一个连通分量
5. }

### 3.3.2 find(int p)寻找父节点和路径压缩函数

int find(int p)用来寻找当前结点p的父节点。当p!=id[p]时进行while循环的意思是如果p元素的父亲指针指向的不是自己，说明p并不是集合中的根元素，还需要一直向上查找和路径压缩，直至循环结束。与此同时，我还在在find查询中嵌入一个路径压缩的操作，即id[p]=id[id[p]]，意思是p元素不再选择原来的父亲节点，而是直接选择父亲节点的父亲节点来做为自己新的一个父亲节点。这样的操作使得树的层数被压缩了，减小了运算规模。

1. **int** find(**int** p)
2. {
3. **while**(p!=id[p])
4. {
5. //如果p元素的父亲指针指向的不是自己，说明p并不是集合中的根元素，还需要一直向上查找和路径压缩
6. //在find查询中嵌入一个路径压缩操作
7. id[p]=id[id[p]];
8. //p元素不再选择原来的父亲节点，而是直接选择父亲节点的父亲节点来做为自己新的一个父亲节点
9. //这样的操作使得树的层数被压缩了
10. p=id[p];
11. //p压缩完毕后且p并不是根节点，p变成p新的父节点继续进行查找和压缩的同时操作
12. }
13. **return** p;
14. }

### 3.3.3 connected(int p,int q)、unionPQ(int p,int q)检测与连接函数

connected(int p,int q)用来检测是不是在同一个集合里，如果p,q的父节点相等，则在同一个集合之中；unionPQ(int p,int q)用来连接p、q两个结点，将它们放进一个集合之中。

1. **bool** connected(**int** p,**int** q)//检测是不是在同一个集合里
2. {
3. **return** find(p)==find(q);//若父节点相同，则在同一个集合之中
4. }
5. **void** unionPQ(**int** p,**int** q)//连接P、Q两个结点
6. {
7. **int** i=find(p),j=find(q);
8. **if**(i==j) **return**;//如果两个已经在一个集合了则退出
9. id[j]=i;
10. count--;
11. }

### 3.3.4接口函数getCount()

getCount()用来返回当前连通分量数量count。

1. //面向对象的封装特性
2. **int** getCount()
3. {
4. **return** count;
5. }

## 3.4 primTree函数的实现

定义两个辅助数组——lowCost与nearVex数组。lowCost数组存放生成树顶点集合内顶点到生成树外各个顶点的各边上的当前最小权值；nearVex数组记录生成树顶点集合外各顶点距离集合内哪个顶点最近。

初始化时除了首个加入生成树的节点lowCost[0]=0和nearVex[0]=-1外，其余lowCost[i]定义为mapp[0][i]，nearVex[i]=0。

之后用for循环找出未连接结点中到当前生成树距离最小的结点，将之移至生成树中，如此重复，直到所有结点都进入生成树中。此时的生成树就是最小生成树。

1. **void** primTree(Edge\* edgeVector,Graph\* graph)
2. //用PRIM建立最小生成树
3. {
4. **int** nodeNum=graph->GetNodeNum();
5. **int** edgeNum=graph->GetEdgeNum();
6. **int** count=0;//已经分离一个顶点(首顶点0)
7. //初始化lowCost与nearVex数组
8. **int** \*lowCost=**new** **int**[nodeNum];
9. //存放生成树顶点集合内顶点到生成树外各个顶点的各边上的当前最小权值
10. **int** \*nearVex=**new** **int**[nodeNum];
11. //记录生成树顶点集合外各顶点距离集合内哪个顶点最近
12. //初始化这两个数组
13. lowCost[0]=0;
14. nearVex[0]=-1;//-1表示已经加入生成树的顶点
15. **for**(**int** i=1;i<nodeNum;i++)
16. {
17. lowCost[i]=graph->GetWeight(0,i);//顶点0到各点的代价
18. nearVex[i]=0;
19. }
20. **for**(**int** j=1;j<nodeNum;j++)//循环n-1次，加入n-1条边
21. {
22. **int** vIndex=0;
23. **int** minn=INF;
24. //求生成树外顶点到生成树内顶点具有最小权值的边
25. //vIndex为当前具有最小权值边的位置
26. **for**(**int** i=0;i<nodeNum;i++)
27. {
28. **if**(nearVex[i]!=-1 && lowCost[i]<minn)
29. {
30. vIndex=i;
31. minn=lowCost[i];
32. }
33. }
34. //把这条边加入不定长数组内记录,并将该树外顶点加入树内
35. **if**(vIndex)//如果vIndex==0表示再也找不到要求顶点了
36. {
37. Edge tempEdge;
38. tempEdge.head=vIndex;
39. tempEdge.tail=nearVex[vIndex];
40. tempEdge.weight=lowCost[vIndex];
41. nearVex[vIndex]=-1;//对该边已经加入生成树标记
42. edgeVector[count++]=tempEdge;//选出的边加入生成树
43. **for**(**int** i=1;i<nodeNum;i++)
44. {
45. **if**(nearVex[i]!=-1 && graph->GetWeight(vIndex,i)<lowCost[i])
46. //i不在生成树之中
47. {
48. lowCost[i]=graph->GetWeight(vIndex,i);
49. nearVex[i]=vIndex;
50. }
51. }
52. }
53. }
54. cout<<"生成Prim最小生成树!\n\n";
55. **delete**[] lowCost;
56. **delete**[] nearVex;
57. }

## 3.5主函数的实现

1. **int** main()
2. {
3. PrintScreen();
4. Graph\* G=NULL;
5. //UF\* uf\_set=NULL;
6. **int** num;
7. **bool** AisDone=**false**,BisDone=**false**,CisDone=**false**;
8. **bool** isContinue=**true**;
9. Edge\* edgeVector=NULL;
10. string name0;
11. **while**(isContinue)
12. {
13. cout<<"请输入操作: ";
14. **char** ops;
15. cin>>ops;
16. **switch**(ops)
17. {
18. **case** 'A':
19. //如果A做过了，给出提醒
20. **if**(AisDone)
21. {
22. cout<<"您已做完A操作,请进行其他操作\n";
23. }
24. **else**
25. {
26. cout<<"请输入顶点的个数:";
27. cin>>num;
28. //uf\_set=new UF(num);
29. cout<<"请依次输入各顶点的名称:\n";
30. string name0;
31. G=**new** Graph(num);
32. edgeVector=**new** Edge[num+1];
33. **for**(**int** i=0;i<num;i++)
34. {
35. cin>>name0;
36. G->node\_name[name0]=i;//序号和名字存入map中
37. }
38. AisDone=**true**;
39. }
40. cout<<endl;
41. **break**;
42. **case** 'B':
43. {
44. **if**(!AisDone) cout<<"请先进行A操作\n";
45. **int** num=G->GetNodeNum();
46. string name1;
47. string name2;
48. **int** weight;
49. **do**{
50. cout<<"请输入两个顶点及边(中间用空格,以? ? 0结尾,中英文输入均可):";
51. cin>>name1>>name2>>weight;
52. } **while**(!G->InsertEdge(name1,name2,weight));
53. cout<<endl;
54. BisDone=**true**;
55. **break**;
56. }
57. **case** 'C':
58. **if**(!AisDone)
59. {
60. cout<<"请先进行A操作\n";
61. **break**;
62. }
63. **else** **if**(!BisDone)
64. {
65. cout<<"请先进行B操作\n";
66. **break**;
67. }
68. primTree(edgeVector,G);
69. CisDone=**true**;
70. **break**;
71. **case** 'D':
72. **if**(!AisDone)
73. {
74. cout<<"请先进行A操作\n";
75. **break**;
76. }
77. **else** **if**(!BisDone)
78. {
79. cout<<"请先进行B操作\n";
80. **break**;
81. }
82. **else** **if**(!CisDone)
83. {
84. cout<<"请先进行C操作\n";
85. **break**;
86. }
87. cout<<"最小生成树的顶点及边为: \n";
88. **for**(**int** i=0;i<G->GetNodeNum()-1;i++)
89. {
90. **for**(auto iter=G->node\_name.begin();iter!=G->node\_name.end();++iter)
91. **if**(iter->second==edgeVector[i].head)
92. //找到边上的头结点的序号对应的名字
93. {
94. cout<<iter->first<<"-<"<<edgeVector[i].weight<<">-";
95. **break**;
96. }
97. **for**(auto iter=G->node\_name.begin();iter!=G->node\_name.end();++iter)
98. **if**(iter->second==edgeVector[i].tail)//找到边上的尾结点的序号对应的名字
99. cout<<iter->first<<"\t\t";
100. **if**((i+1)%4==0) cout<<endl;//一行输入四个
101. }
102. cout<<endl<<endl;
103. **break**;
104. **case** 'E':
105. isContinue=**false**;
106. **break**;
107. **default**:
108. cout<<"错误的操作符,请确认后再输入\n";
109. **break**;
110. }
111. }
112. **delete** G;
113. **delete** edgeVector;
114. **return** 0;
115. }

# 4 测试

## 4.1 功能测试

### 4.1.1 测试1

测试数据：

A

4

a b c d

B

a b 8

b c 7

c d 5

d a 11

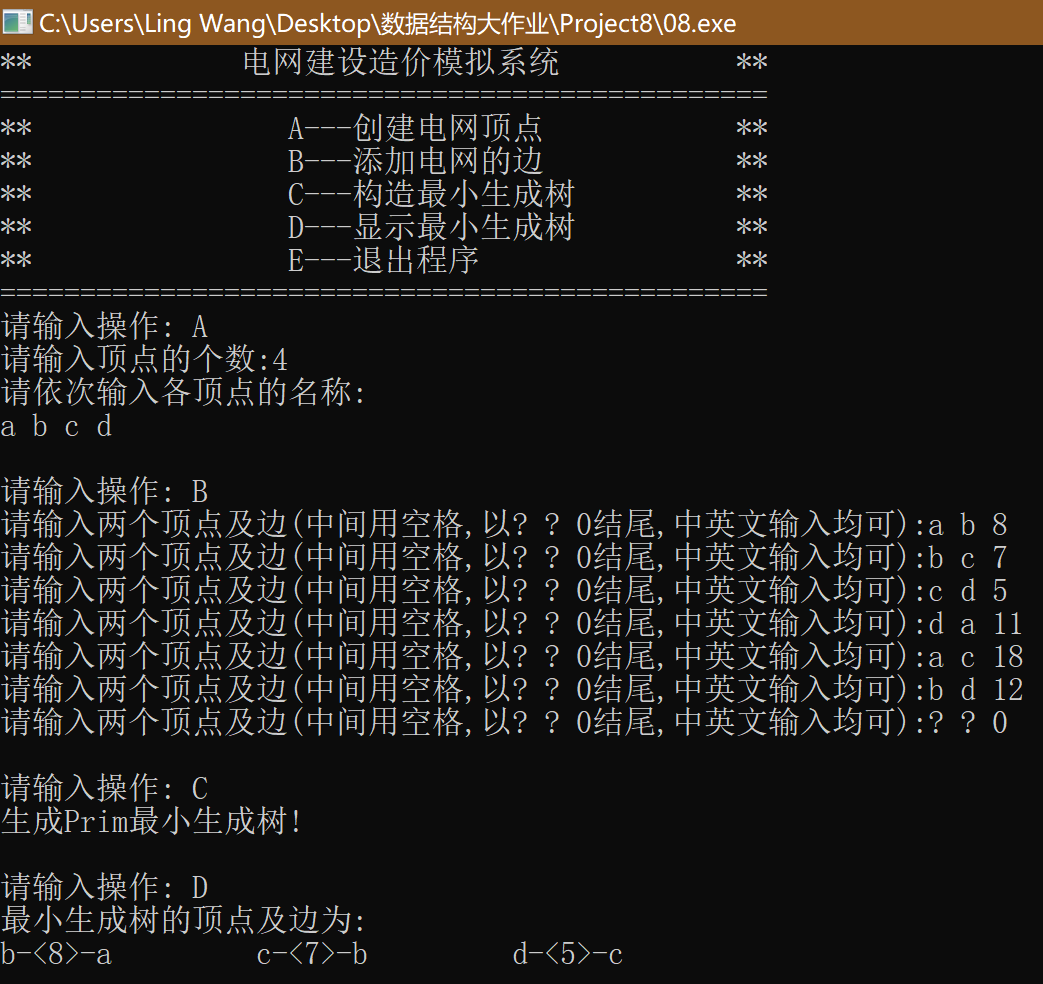
a c 18

b d 12

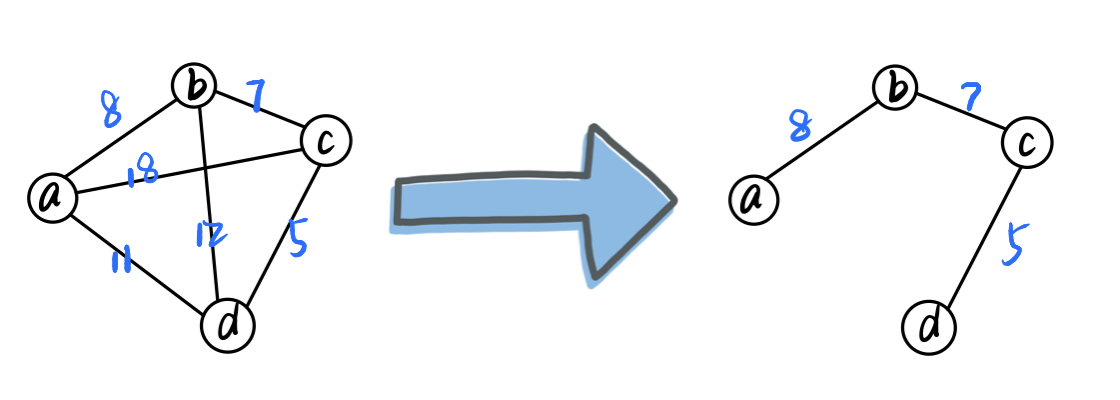
? ? 0

C

D



最小生成树如下图：



### 4.1.2 测试2

测试数据：

A

6

1 2 3 4 5 6

B

1 2 6

1 3 1

1 4 5

2 3 5

2 5 3

3 5 6

3 6 4

4 3 5

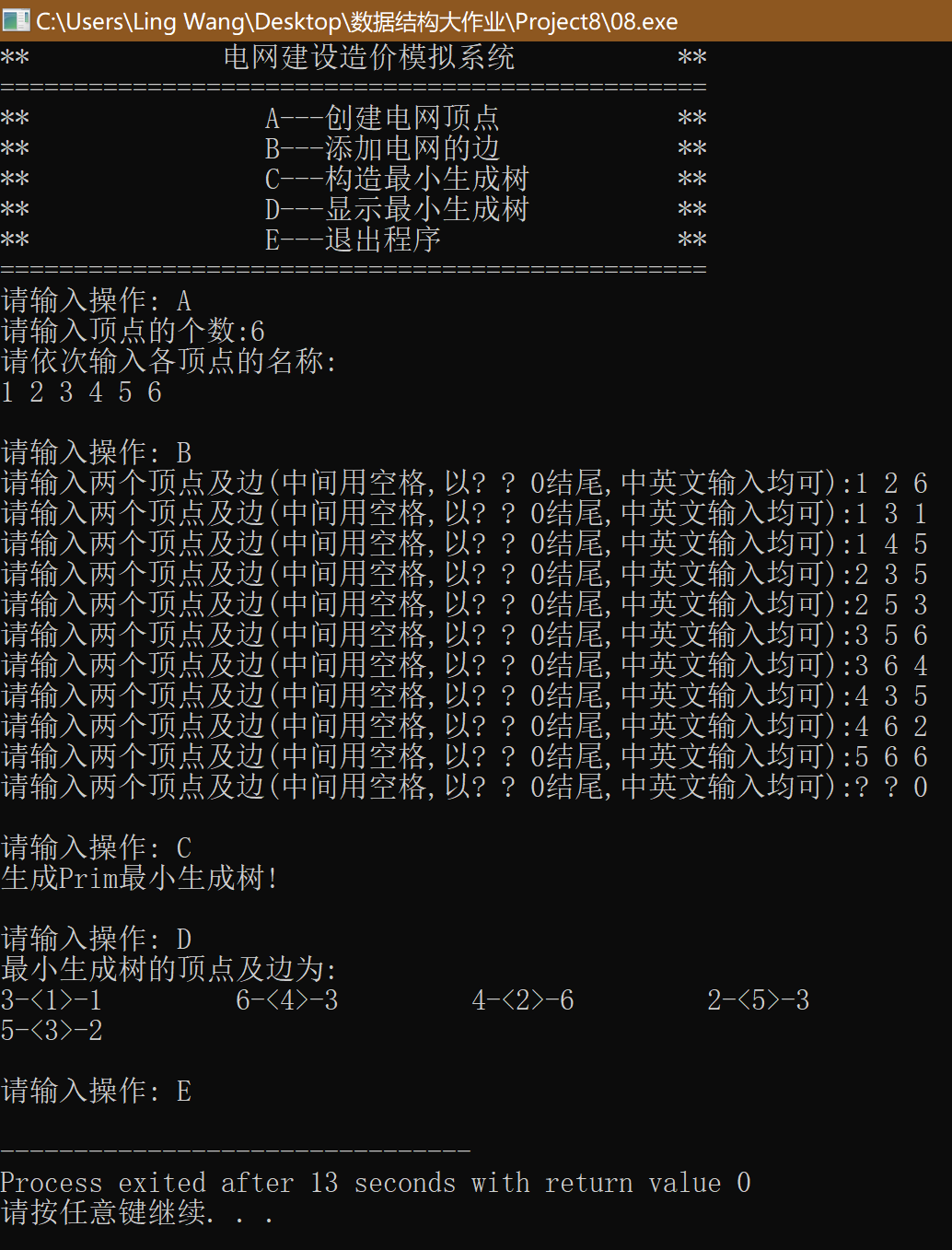
4 6 2

5 6 6

? ? 0

C

D



相应的最小生成树如下图：

