电网建设造价模拟系统

作 者 姓 名： 李翠琪

学 号： 1751022

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 1](#_Toc25759428)

[1.1 项目简介 1](#_Toc25759429)

[1.2 功能分析 1](#_Toc25759430)

[2 设计与实现 2](#_Toc25759431)

[2.1 Edge.h设计与实现 2](#_Toc25759432)

[2.2 EdgeWeightedGraph.h设计与实现 2](#_Toc25759433)

[2.3 Queue.h与Node.h设计与实现 4](#_Toc25759434)

[2.4 MinPQ.h设计与实现 5](#_Toc25759435)

[2.5 UF.h设计与实现 7](#_Toc25759437)

[2.6 KruskalMST.h设计与实现 9](#_Toc25759438)

[2.7 MainTest.cpp设计与实现 10](#_Toc25759439)

[3 测试 13](#_Toc25759440)

[3.1 功能测试 13](#_Toc25759441)

[3.1.1 测试1 13](#_Toc25759442)

[3.1.2 测试2 15](#_Toc25759443)

# 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,每个小区当作一个顶点,那么就有n个顶点。两个小区之间设置一条电网线路相互联通且有相应的经济代价。所以可以抽取为一个加权无向图的模型。

所以问题就成了求一个加权无向图的最小生成树。所以首先需要用户输入小区即顶点数，并且给每个小区进行命名以方便查看。然后用户将添加若干个电网线路以及他们之间的花费。而程序就负责找出这个加权无向图的最小生成树，并且显示出来。

我用了Kruskal算法,所以需要许多类来辅助。大致架构是，MainTest类与用户交互，负责接受用户输入，并且用EdgeWeightedGraph类构造加权无向图，并且通过KruskalMST类得到一个存储边（Edge类）的队列（Queue类）（存储了最小生成树的边），然后再将这些边显示出来。而KruskalMST类还需要最小堆（MinPQ类）找最小的边与UF类辨认连通分量。所以总的来说有八个类以及一个input.txt存储一些测试数据。

# 2 设计与实现

## 2.1 Edge.h设计与实现

Edge类代表了图中的边，它有三个成员变量：边的两个顶点v，w与double类型的权值weight。

构造函数就将这三个参数传入；还有三个函数分别得到v，w，weight的值，不过有些不同的是other函数是根据传入的边传出另外一条边，更加的灵活（如other（v）传回w）。最后重载了小于运算符<,根据weight进行比较。

class Edge  
{  
public:  
 Edge(){}  
 Edge(int v, int w, double weight) : v(v), w(w), weight(weight) {}  
  
 int either() const  
 {  
 return v;  
 }  
  
 int other(int vertex) const  
 {  
 if(vertex==v)  
 return w;  
 else if(vertex==w)  
 return v;  
 }  
  
 double getWeight() const  
 {  
 return weight;  
 }  
  
 bool operator < (const Edge & that) const  
 {  
 if (weight < that.weight)  
 return true;  
 else  
 return false;  
 }  
  
private:  
 int v;  
 int w;  
 double weight;  
};

## 2.2 EdgeWeightedGraph.h设计与实现

代表了图。三个成员变量，分别是顶点的数量V，边的数量E以及邻接表adj。其中邻接表存储了每个顶点所含有的边（类型为vector<vector<Edge>>）。

构造函数将给定顶点数量N，并且让边为0，而且为邻接表adj开辟N个空间，使得每个顶点i都能使用adj[i]。

函数有返回V和E数量的函数，以及添加边的函数addEdge（）：传入一条边，并且让边的两个顶点的邻接表分别加入这条边，最后让边数量+1。还有一个返回所以边的函数edges（）：很简单，就是遍历邻接表，对于每个顶点的每条边，将边放入队列中，需要注意的是，因为是无向图，所以如果所有顶点的边都加入就重复了，所以只加入v<w

的边，防止重复。

class EdgeWeightedGraph  
{  
public:  
 EdgeWeightedGraph(int v) : V(v),E(0)  
 {  
 vector<vector<Edge>> temp(V);  
 adj=temp;  
 }  
  
  
 void addEdge(Edge e)  
 {  
 int v = e.either();  
 int w = e.other(v);  
 adj[v].push\_back(e);//往邻接表中添加  
 adj[w].push\_back(e);  
 ++E;  
 }  
  
 Queue<Edge> edges()//返回所有边  
 {  
 Queue<Edge> queue;  
 for (int v = 0; v < V; ++v)  
 {  
 for(Edge e:adj[v])  
 {  
 if(e.other(v)>v)  
 {  
 queue.enqueue(e);  
 }  
 }  
 }  
 return queue;  
 }  
  
 int getV()  
 {  
 return V;  
 }  
  
 int getE()  
 {  
 return E;  
 }  
  
private:  
 int V;  
 int E;  
 vector<vector<Edge>> adj;//邻接表  
  
};

## 2.3 Queue.h与Node.h设计与实现

因为很多地方用到队列，所以又自己实现了一遍。队列需要用到Node.h之前说过很多此就不说了。

队列为模板类，存储T类型。成员变量是指向队首的first指针以及指向队尾的last指针，还有一个存储元素个数的N。

队列主要就是enqueue，dequeue以及empty和size四个函数。empty只要判断N==0即可，size也只要返回N即可。enqueue就是在队尾加入元素，具体操作为：如果队列为空，让头尾指针同时指向加入的元素，让计数N++。如果不为空，让当前的last->next指向新加入元素，再让last指向当前的最后元素，让计数N++。dequeue就将队首弹出，让first指向下一个元素，让计数N--，再稍微处理异常情况，比如弹出队首后让last也为空。

template <class T> class Node  
{  
public:  
  
 Node(const T & value, Node<T> \*next= nullptr) : value(value), next(next) {}  
 Node(Node<T> \*next= nullptr) : next(next) {}  
  
 T value;  
 Node<T> \*next;  
};

template <class T> class Queue  
{  
public:  
 void enqueue(T t)  
 {  
 if(empty())//为空指向同一个  
 {  
 first=new Node<T>(t, nullptr);  
 last=first;  
 }  
 else  
 {  
 Node<T> \* temp = new Node<T>(t, nullptr);  
 last->next=temp;  
 last=temp;  
 }  
 ++N;  
 }  
 T dequeue()  
 {  
 if (empty())  
 throw "队列为空";  
 T value=first->value;  
 first=first->next;  
 --N;  
 if(empty())  
 last= nullptr;  
 return value;  
 }  
  
 bool empty()  
 {  
 return N==0;  
 }  
  
 int size()  
 {  
 return N;  
 }  
  
private:  
 Node<T> \* first;  
 Node<T> \* last;  
 int N = 0;  
};

## 2.4 MinPQ.h设计与实现

### 当构造一个堆时，需要输入一个堆中应该存的元素数量最大值len，令堆中的成员变量数组的长度为len+1，（因为不用数组的vec[0]），堆的这个类还有一个N记录当前堆的元素数量。

公有函数：

1.isEmpty函数及size函数：因为N记录当前堆的元素数量,所以isEmpty函数返回N==0， size函数返回N即可。

2.push函数：令N+1,并且在N处添加元素，然后将这个元素swim，使得还是一个堆。

3.delMin函数：最小元素在堆顶，先用一个变量存储堆顶的值，将堆顶与堆中最后一个元素交换，令N-1，此时不是一个堆，需要将堆顶sink到合适的位置，最后返回之前存的堆顶元素的值。

私有函数：

1.sink函数：sink的思路是将这个元素与他的子节点做对比，并且将较小的放上面，一直对比到这个元素小于他的两个子节点为止。具体实现是，假如要下沉k，则先找它的子节点2\*k与2\*k+1中最小的一个（当然得判断是否存在），如果小于他的子节点的值，那么就交换他们的值，并且让k=2\*k或2\*k+1（＝最小的那个），这是为了继续追踪原本的k，继续和他此时的两个子节点做对比。不断的循环，直到他没有子节点或者他小于了子节点。

2.swim函数：直观看就是一个元素比较轻，然后它要不断上浮，直到它重与他的父节点。而代码的实现就是，查看他的父节点k/2的值是否小于它，如果不小于它，则交换它和它父节点的值，并且让k=k/2，即继续追踪这个元素，让它继续与它当前的父节点做对比。直到它本身为数组的第一个元素或者它不小于它的父节点。

template <class T> class MinPQ  
{  
public:  
 MinPQ(int N):pq(N+1)  
 {  
  
 }  
  
 bool isEmpty()  
 {  
 return size()==0;  
 }  
  
 int size()  
 {  
 return N;  
 }  
  
 T delMin()  
 {  
 T min = pq[1];  
 exch(1, N--);  
 sink(1);  
 return min;  
 }  
  
 void push(T t)  
 {  
 pq[++N] = t;  
 swim(N);  
 }  
  
private:  
 vector<T> pq;  
 int N=0;//元素个数,从1开始  
   
 void swim(int k)  
 {  
 while(k>1 && pq[k] < pq[k/2])  
 {  
 exch(k, k / 2);  
 k /= 2;  
 }  
 }  
   
 void sink(int k)  
 {  
 while(2 \* k <=N)  
 {  
 int j = 2 \* k;  
 if(j+1<=N && pq[j+1]<pq[j])  
 {  
 ++j;  
 }  
 if(pq[k]<pq[j])  
 {  
 break;  
 }  
 exch(k, j);  
 k = j;  
 }  
 }  
   
 void exch(int i,int j)  
 {  
 T temp = pq[i];  
 pq[i] = pq[j];  
 pq[j] = temp;  
 }  
};

## 2.5 UF.h设计与实现

该类负责构造连通分量。他有三个成员变量，id存储每个顶点的父节点，sz存储每个顶点所属于的那个连通分量的规模（想象如果将规模大的根节点链接到规模小的,那么就会使得节点平均访问到根节点的时间增加,所以需要将规模大的作为根节点），以及当前连通分量数量count。

构造函数将当前所以节点的父节点赋值为他们本身并且每个节点的size都是1，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。

主要有这几个函数：getCount()返回当前连通分量数量count；connected(int p, int q)检测两个顶点是否是连通的，这通过调用find(int p)来判断，如果find（p）==find(q),他们就是同一个连通分量；find(int p)则找到p的根节点，即顶点p所属于的连通分量。unionPQ(int p, int q)则将两个不连通的节点连通。

详细说明find（int p）：因为每次查找id[p],如果id[p]=p，代表p是p的根节点，代表这就是一个连通分量。而如果id[p]不等于p，则代表p是某个连通分量的子节点，而id[p]就是它的父节点，所以让p= id[p]使得p到了他的父节点，继续查找，直到找到这个连通分量的根节点，对应的就是最终的p= id[p]。

unionPQ(int p, int q)：先找到传入的p，q的根节点i，j，如果根节点相同，那么他们是同一个连通分量直接返回。接着，判断这两个根节点的规模sz[i],sz[j]。然后将规模小的链接到规模大的上面(也就是以规模大的这个根节点作为这两棵树合起来的根节点)，这样找某个节点的根节点更快。然后将该根节点的规模再加上规模小的那个节点的规模,这样才使得规模有意义。

class UF  
{  
public:  
 UF(int N)  
 {  
 for (int i = 0; i < N; ++i)  
 {  
 id.push\_back(i);  
 sz.push\_back(1);  
 }  
 count=N;  
 }  
  
 int getCount()  
 {  
 return count;  
 }  
  
 bool connected(int p,int q)  
 {  
 return find(p) == find(q);//检测是不是在同一颗树上  
 }  
  
 int find(int p)  
 {  
 while(p!=id[p])  
 {  
 p = id[p];  
 }  
 return p;  
 }  
  
 void unionPQ(int p,int q)  
 {  
 int i = find(p);  
 int j = find(q);  
 if(i==j)  
 {  
 return;  
 }  
 if(sz[i]<sz[j])  
 {  
 id[i] = j;  
 sz[j] += sz[i];  
 }  
 else  
 {  
 id[j] = i;  
 sz[i] += sz[j];  
 }  
 count--;  
 }  
   
private:  
 vector<int> id;//父链接数组  
 vector<int> sz;//没颗树高度  
 int count;//连通分量数量  
};

## 2.6 KruskalMST.h设计与实现

该类就负责计算最小生成树。需要传入构造好的加权无向图，它有个成员变量mst将负责返回存有最小生成树的所有边的队列。

主要就两个函数，getMST返回mst，即返回存有最小生成树的所有边的队列；构造函数传入加权无向图，调用MinPQ类，并且把树中所有的边都放入最小堆中。还调用了UF类，传入了图中的顶点数，代表了初试有V个连通分量。紧接着，当最小生成树中边的数量小于V-1并且最小堆中还有边的时候，就一直循环。在循环之中，不断的弹出最小堆中最小的边，而如果这条边的两个顶点是两个不同的连通分量，说明这条边是最小生成树中的边，于是将之加入到存有最小生成树的所有边的队列mst中，并且将这两个顶点连接起来，成为同一个连通分量。而如果弹出最小堆中最小的边的两个顶点是同一个连通分量，则这条边不是最小生成树中的边，则忽略它，继续循环。

class KruskalMST  
{  
public:  
 KruskalMST(EdgeWeightedGraph G)  
 {  
 MinPQ<Edge> pq(G.getE());  
 Queue<Edge> edges = G.edges();  
 while(!edges.empty())  
 {  
 pq.push(edges.dequeue());  
 }  
 UF uf(G.getV());  
  
 while(!pq.isEmpty() && mst.size()<G.getV()-1)  
 {  
 Edge e = pq.delMin();  
 int v = e.either();  
 int w = e.other(v);  
 if(uf.connected(v,w))  
 continue;  
 uf.unionPQ(v,w);  
 mst.enqueue(e);  
 }  
 }  
  
 Queue<Edge> getMST()  
 {  
 return mst;  
 }  
  
private:  
 Queue<Edge> mst;  
};

## 2.7 MainTest.cpp设计与实现

先给出相应的提示，之后创建一个map以将之后用户输入的节点代号和节点数字相互对应。然后进入循环等待用户进行操作。

如果用户选择操作A，就让用户输入N个顶点和他们的代号，然后程序会生成一个含有N个顶点的不含有边的图（这里图中存的只是顶点数字，而顶点代号与顶点数字的对应关系存在map中）。

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（当然首先得把用户输入的顶点代号通过map转换为对应的顶点数字）。当用户输入0 0 0时，退出当前输入边的循环。

如果选择C操作，则将加权无向图传入上述所实现的KruskalMST.h类的构造函数中，其实此时已经生成了最小生成树，就存在指向KruskalMST.h类的指针kruskal中。

如果选择D操作，将显示最小生成树，具体是，通过kruskal->getMST()得到存储有所有最小生成树边的队列。然后从队列中取出所有的边，并且将边的两个顶点数字通过map转换为顶点代号，并且打印出来。

int main()  
{  
  
 cout << "\*\* 电网建设造价模拟系统 \*\*" << endl;  
 cout << "================================================" << endl;  
 cout << "\*\* A---创建电网顶点 \*\*" << endl;  
 cout << "\*\* B---添加电网的边 \*\*" << endl;  
 cout << "\*\* C---构造最小生成树 \*\*" << endl;  
 cout << "\*\* D---显示最小生成树 \*\*" << endl;  
 cout << "\*\* E---退出程序 \*\*" << endl;  
 cout << "================================================" << endl;  
  
 unordered\_map<string, int> names;//节点名字与节点数字相互对应  
 EdgeWeightedGraph \* G = nullptr;  
 KruskalMST \* kruskal = nullptr;  
  
 while (true)  
 {  
 cout << "请选择操作:";  
 char operation;  
 cin >> operation;  
 if (operation == 'A')  
 {  
 cout << "请输入节点数量:";  
 int N;//节点数量  
 cin >> N;  
 cout << "请依次输入节点名称:";  
 for (int i = 0; i < N; ++i)  
 {  
 string node;  
 cin >> node;  
 names[node] = i;  
 }  
 G = new EdgeWeightedGraph(N);//生成一个没有边的图  
 }  
 else if (operation == 'B')  
 {  
 while (true)  
 {  
 cout << "请输入两个顶点及边：(输入0 0 0 结束)";  
 string node1, node2;  
 double nodeValue;  
 cin >> node1 >> node2 >> nodeValue;  
 if (node1 == "0" && node2 == "0" && nodeValue == 0)  
 {  
 break;  
 }  
 if (nodeValue <= 0)  
 {  
 cout << "输入错误,两个城市线路的花费不能小于等于0";  
 }  
 int v = 0, w = 0;  
 auto iter = names.find(node1);  
 if (iter != names.end())  
 {  
 v = iter->second;  
 }  
 auto iter2 = names.find(node2);  
 if (iter2 != names.end())  
 {  
 w = iter2->second;  
 }  
 if (v == -1 || w == -1)  
 {  
 cout << "输入错误,输入的顶点不在图中!";  
 }  
 else  
 {  
 Edge edge(v, w, nodeValue);  
 G->addEdge(edge);  
 }  
 }  
 }  
 else if (operation == 'C')  
 {  
 cout << "最小生成树构建成功!";  
 kruskal = new KruskalMST(\*G);  
 }  
 else if (operation == 'D')  
 {  
 cout << "最小生成树的顶点和边为:" << endl;  
 Queue<Edge> queue = kruskal->getMST();  
 while (!queue.empty())  
 {  
 Edge e = queue.dequeue();  
 int v = e.either();  
 int w = e.other(v);  
 string name1, name2;  
  
 int count = 0;  
 for (auto iter = names.begin(); iter != names.end(); ++iter)  
 {  
 if (iter->second == v)  
 {  
 name1 = iter->first;  
 ++count;  
 }  
 if (iter->second == w)  
 {  
 name2 = iter->first;  
 ++count;  
 }  
 if (count == 2)  
 {  
 break;  
 }//找到这两就退出  
 }  
 cout << "[" << name1 << "---" << name2 << " cost: " << e.getWeight() << "]" << endl;  
 }  
 // kruskal->printAll(); 需要字符显示  
 }  
 else if (operation == 'E')  
 {  
 cout << "期待您的下次使用!";  
 return 0;  
 }  
 }  
}

# 3 测试

## 3.1 功能测试

### 3.1.1 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

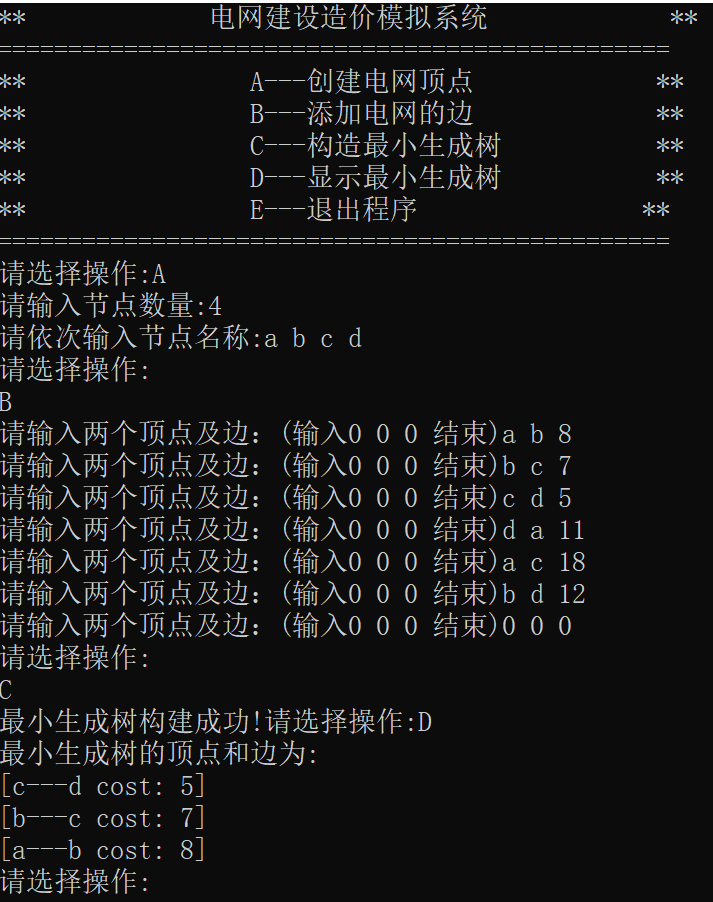
a c 18

b d 12

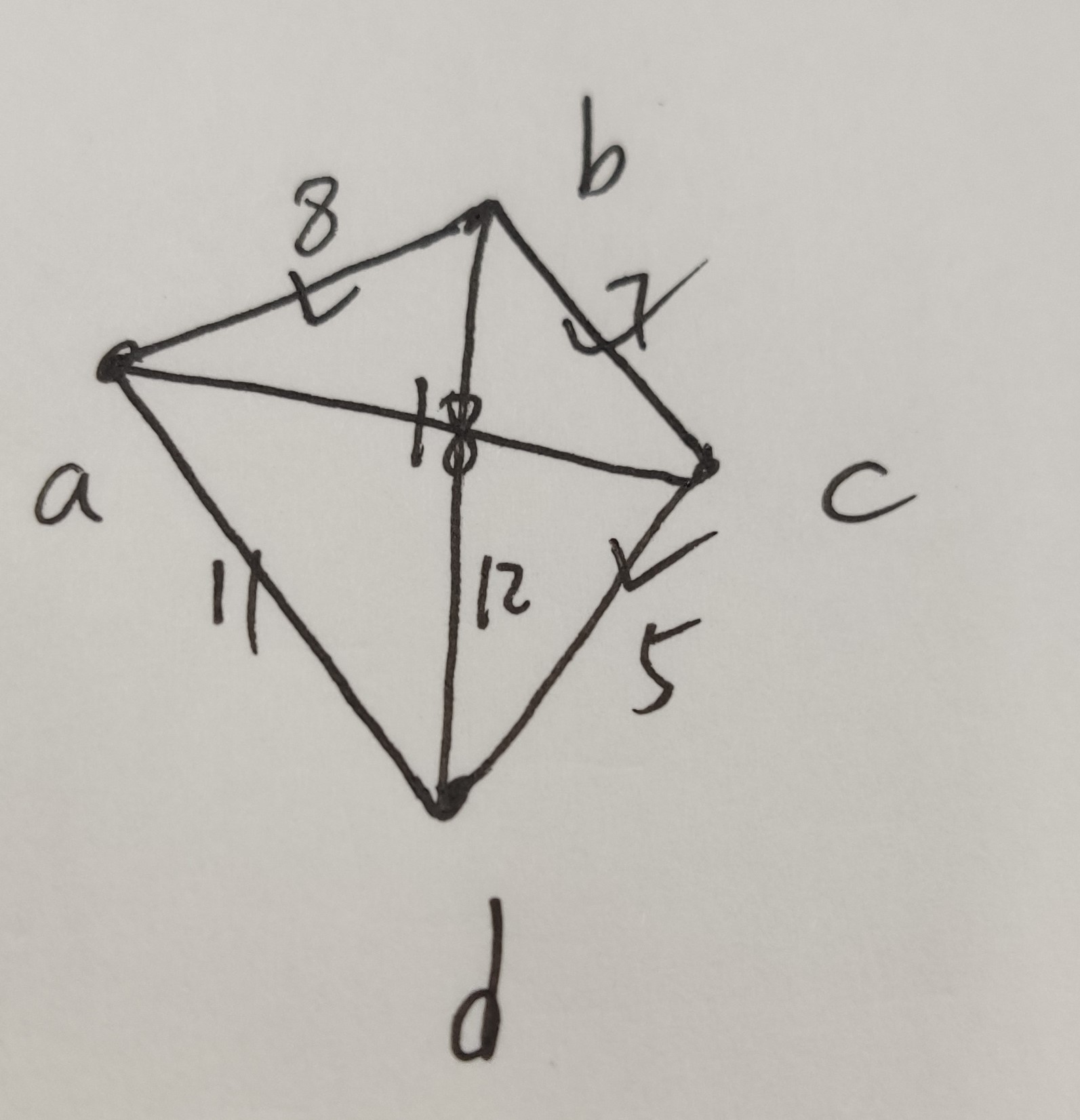
0 0 0

C

D



最小生成树如下图（打勾位置为最小生成树的边）：



### 3.1.2 测试2

测试数据：

A

8

0 1 2 3 4 5 6 7

B

4 5 0.35

4 7 0.37

5 7 0.28

0 7 0.16

1 5 0.32

0 4 0.38

2 3 0.17

1 7 0.19

0 2 0.26

1 2 0.36

1 3 0.29

2 7 0.34

6 2 0.40

3 6 0.52

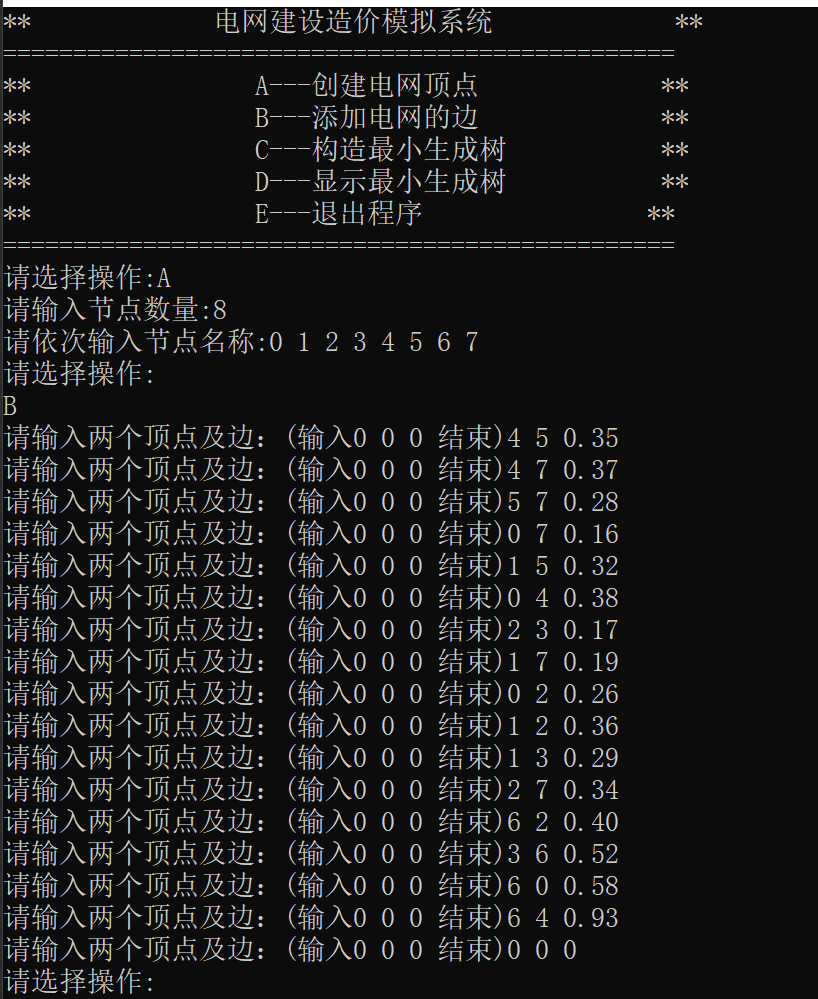
6 0 0.58

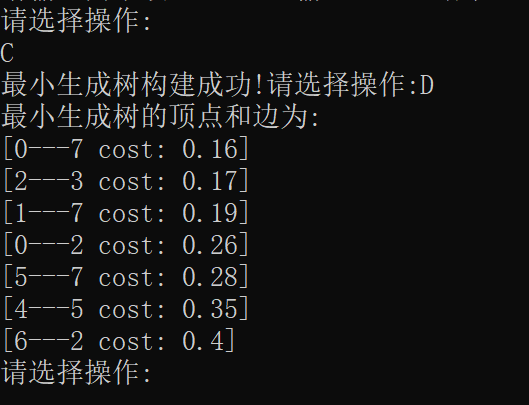
6 4 0.93

0 0 0

C

D





相应的最小生成树如下图：

