组号: 1

上海大学计算机工程与科学学院

实 验 报 告

（数据结构 2）

学　　期：2021-2022 年春季

组　　长：李昀哲

学　　号：20123101

指导教师：朱能军

成绩评定：

二〇二二年 4 月 5 日

小组信息				
登记序号	姓名	学号	贡献比	签名
72	李昀哲	20123101	25%	
21	唐铭锋	20121489	25%	
20	刘沛根	20121483	25%	
22	李正宇	20121517	25%	

实验列表		
实验一	（熟悉上机环境、进度安排、评分制度；分组）	
实验二	*有向网的邻接矩阵验证及拓展*	✓
实验三	无向网的邻接表验证和拓展	✓
实验四	(*实验题目*)	
实验五	(*实验题目*)	

实验三

一、实验题目

无向网的邻接表验证和拓展

二、实验内容

模仿有向图的邻接表类模板，完成（带权：非负）无向网 的邻接表类模板的设计与实现。要求实现图的基本运算（如增加删除顶点和边等），并增加如下成员函数：

1. CountDegree(v)，统计顶点 v 的度；

2. ConnectedComponent()，求图的连通分量数目；

3. 验证 Kruskal 和 Prim 两种最小生成树算法，并设计或实现除此以外的另一种最小生成树算法，如“破圈法”等（仅考虑连通图）；

4. hasUniqueMinTree()，判断无向网是否存在唯一的最小生成树（仅考虑连通图）。

三、解决方案

1、算法设计

在用邻接表作为存储表示的无向网的类模板中，可分为顶点结点类模板、弧结点类模板、邻接表类模板。在顶点结点类模板中定义了两个数据成员，其一是 data 域，它存储顶点信息；其二是 firstarc 域，它是一个指针，指向从该顶点出发的第一条弧结点。在弧结点类模板中定义了三个数据成员，其一是 adjVex 域，它存储弧头顶点的序号；其二是 weight 域，它存储这条弧上的权值；其三是 nextarc 域，它是一个指针，指向从该顶点出发的下一条弧结点。在无向网邻接表类模板的定义中，顶点表 vexTable 是一个顶点数组，每个顶点对应一个数组元素，用以存放顶点信息和指向从该顶点出发的第一个弧结点；还定义了 vexNum、vexMaxNum 和 arcNum 三个数据成员，分别记录图中当前顶点数目、允许的顶点最大数目和弧数；为了在图的遍历等算法中记录顶点是否访问，在此定义了一个标志数组 tag 需要时用以记录顶点的访问状况，另外数据成员 infinity 表示无穷大的值。对于插入边的操作，无向图和有向图略有不同，无向图需要考虑这条边关联的两个顶点，在两个顶点的邻接表中都需要加入它们的信息。

(1) 统计顶点 v 的度:创建一个指针，从顶点 v 出发，依次遍历其所连接的弧结点，直到指向 null 为止，此时到达边链表尾，其间每次遍历用计数器计数，最终返回的计数器值即为顶点 v 的度。

(2) 求图的连通分量数目:用 DFS 算法对图进行遍历，其中调用 DFS 函数的次数即为该图的连通分量数目。

(3) 验证 Kruskal 和 Prim 两种最小生成树算法，并设计或实现了“破圈法”求最小生成树（仅考虑连通图）：

首先我们所学到的 Kruskal 和 Prim 算法分别对同一个无向连通图求其最小生成树，并得到了相同的正确结果，从而验证了这两种算法的可行性。

接着，我们开始考虑除这两种算法之外的另一种算法，即“破圈法”，其大致思路是先找到图中所有存在的圈，接着在所有构成圈的边中选择一条权值最大的边并将其从图中删除，然后再次寻找图中是否还存在圈。如有则再次进行选权值最大的边删除的操作，如此重复，直到图中没有圈为止，此时该图即为最小生成树。

“破圈法”其实也是一种贪心算法，具体代码实现可简述为：

1. 用拓扑分类算法，依次找到图中度为 1 的顶点，可以保存在队列里，接着遍历队列，依次将各个顶点出队，然后与其邻接的顶点的入度减 1，并判断这些顶点在度数减 1 后是否度数为 1，若为则入队，这样往复操作，直到图中已不存在度数为 1 的顶点，即所有的顶点的度数都大于等于 2，那么剩下的边就都在环里了。如果没剩下边，说明没有环，算法结束。

2. 在步骤 1 结束后，剩下的边就都是环中的边了，找一个权最大的删除。

3. 再进行 1 操作，直到图中无圈，即所有的圈都已破掉，剩下的就是最小生成树了。

(4) 判断无向网是否存在唯一的最小生成树（仅考虑连通图）：

思路可简述为：使用 Prim 算法求最小生成树的过程中，若有两个顶点到某一顶点的边的权值相同且是最小权值，或者若有一个顶点到另外两个顶点的边权值相同且为最小值，则最小生成树不唯一，否则存在唯一最小生成树。

2、源程序代码

（1）统计顶点 v 的度

```
template<class ElemType,class WeightType>
int AdjListUnDirNetwork<ElemType, WeightType>::CountDegree(int v) const
{
    AdjListNetworkArc<WeightType> *p;
    int cnt = 0;
    p = vexTable[v].firstarc;
    while (p != NULL)
    {
        p = p->nextarc;
        ++cnt;
    }
    return cnt;
}
```

（2）求图的连通分量数目

```
template<class ElemType, class WeightType>
int AdjListUnDirNetwork<ElemType, WeightType>::ConnectedComponent() {
    if (IsEmpty())
        return 0;
    int num = 0;
    for(int i = 0;i<GetVexNum();i++){
        if(tag[i] != VISITED){
            DFS( v: i);
            num++;
        }
    }
    for(int i = 0;i<GetVexNum();i++)
        tag[i] = UNVISITED;
    return num;
}
```

（3）“破圈法”求最小生成树

1、带权边类的定义：

```
template <class ElemType, class WeightType>
class Edge     //带权边类
{
public:
    ElemType vertex1, vertex2;      // 边的顶点
    WeightType weight;              // 边的权值
    Edge(ElemType v1, ElemType v2, WeightType w);   // 构造函数
    Edge(){};               // 构造函数
    Edge<ElemType, WeightType> &operator =(const Edge<ElemType, WeightType> &Ed); // 赋值语句重载
    bool operator <=(const Edge<ElemType, WeightType> &Ed);     // 重载<=关系运算
    bool operator >(const Edge<ElemType, WeightType> &Ed);      // 重载>关系运算
};
```

```cpp
template <class ElemType, class WeightType>
bool Edge<ElemType, WeightType>::operator <= (const Edge<ElemType, WeightType> &Ed)
// 操作结果：重载<=关系运算
{
    return (weight <= Ed.weight);
}

template <class ElemType, class WeightType>
bool Edge<ElemType, WeightType>::operator > (const Edge<ElemType, WeightType> &Ed)
// 操作结果：重载>关系运算
{
    return (weight > Ed.weight);
}

template<class ElemType,  class WeightType>
Edge<ElemType, WeightType> &Edge<ElemType, WeightType>::operator = (const Edge<ElemType, WeightType> &Ed)
// 操作结果：将栈copy赋值给当前栈--赋值语句重载
{
    if (&Ed != this)    {
        vertex1 = Ed.vertex1;                   // 顶点vertex1
        vertex2 = Ed.vertex2;                   // 顶点vertex2
        weight = Ed.weight;                     // 权weight
    }
    return *this;
}
```

2、定义顶点度数结构体

```cpp
struct Degree      //顶点度数结构体
{
    int d;    //顶点度数
    bool tag;  //标记当前顶点是否被访问过
    int nearvex;  //最小生成树中当前顶点所在边的另一个顶点
};
```

3、破圈算法具体函数实现

```cpp
template <class ElemType, class WeightType>
void MiniSpanTreeBreakCycle(const AdjListUnDirNetwork<ElemType, WeightType> &g)
{
    ElemType v1, v2;
    int vexnum = g.GetVexNum();
    Degree *degree;
    int u, v,k;
    degree=new Degree[vexnum];
    for(v=0;v<vexnum;v++)   //初始化记录结点当前度数的数组
    {
        degree[v].d=g.CountDegree(v);
        degree[v].nearvex=0;
        degree[v].tag=0;
    }
    Edge<ElemType, WeightType> *edge;   //构建边权值从大到小的数组
    Edge<ElemType, WeightType> temp;
    edge=new Edge<ElemType, WeightType>[g.GetArcNum()];
    int edgenum=0;
    bool reach[vexnum][vexnum];  //创建一个可达矩阵
```

```
for (v = 0; v < g.GetVexNum(); v++)
    for (u = g.FirstAdjVex(v); u >= 0; u = g.NextAdjVex( v1: v,  v2: u))
        if (v < u)
        {   // 将v < u的边插入数组中
            reach[v][u]=reach[u][v]=1;   //可达矩阵初始化
            g.GetElem(v,  &: v1);
            g.GetElem( v: u,  &: v2);
            edge[edgenum].vertex1 = v1;
            edge[edgenum].vertex2 = v2;
            edge[edgenum].weight = g.GetWeight( v1: v, v2: u);
            edgenum++;
        }
for(int i=0;i<edgenum-1;i++)  //对边数组按权值从大到小排序
{
    for(int j=0;j<edgenum-i-1;j++)
    {
        if(edge[j]<=edge[j+1])
        {
            temp=edge[j];
            edge[j]=edge[j+1];
            edge[j+1]=temp;
        }
    }

}

SeqQueue<Edge<ElemType, WeightType>> e;
for(int i=0;i<edgenum;i++)  //将按权值从大到小排列的边依次加入队列中
{
    e.EnQueue( e: edge[i]);
}
int count=0;  //初始化计数器，记录已访问过的顶点数
SeqQueue<int> queue;  //创建一个存放度数为1的顶点的队列
while(count<vexnum)
{
    for(v=0;v<vexnum;v++)  //依次查找是否存在度数为1的顶点
    {
        if(degree[v].d==1&&degree[v].tag==0) {
            queue.EnQueue( e: v);
            degree[v].tag =1;
            k=g.FirstAdjVex(v);
            while(1)  //循环遍历找到与该顶点有边相邻且未被访问过的顶点
            {
                if((degree[k].tag==0)&&(reach[v][k]==1))
                    break;
                k=g.NextAdjVex( v1: v, v2: k);
            }
            degree[v].nearvex=k;
            count++;
        }
```

```
while(!queue.IsEmpty())   //对队列中的顶点元素依次出队并访问
{
    queue.DelQueue( &: v);
    u=degree[v].nearvex;
    degree[u].d-=1;
    g.GetElem(v, &: v1);
    g.GetElem( v: u, &: v2);
    cout << "边:( " << v1 << ", " << v2<< " ) 权:" << g.GetWeight( v1: v, v2: u) << endl ; // 输出边及权值
    if(degree[u].d==1)  //判断与出队顶点相邻的顶点在度数减1后度数是否等于1
    {
        queue.EnQueue( e: u);
        degree[u].tag =1;
        count++;
        if(count==vexnum)
            return;
        k=g.FirstAdjVex( v: u);
        while(1)  //循环遍历找到与该顶点有边相邻且未被访问过的顶点
        {
            if((degree[k].tag==0)&&(reach[u][k]==1))
                break;
            k=g.NextAdjVex( v1: u, v2: k);
        }
        degree[u].nearvex=k;
    }
}
    e.DelQueue( &: temp);  //将权值最大的边出队
    v1 = temp.vertex1;
    v2 = temp.vertex2;
    while((degree[g.GetOrder( &: v1)].tag==1)&&(degree[g.GetOrder( &: v2)].tag==1))
    {  //判断该出队的边是否与已被访问过的顶点相连，如果是则再出一条边
        e.DelQueue( &: temp);
        v1 = temp.vertex1;
        v2 = temp.vertex2;
    }
    degree[g.GetOrder( &: v1)].d-=1;  //将与该边所连的顶点度数减1
    degree[g.GetOrder( &: v2)].d-=1;
    reach[g.GetOrder( &: v1)][g.GetOrder( &: v2)]=0;  //将与该边所连的顶点在可达矩阵中对应的值置为0
    reach[g.GetOrder( &: v2)][g.GetOrder( &: v1)]=0;
}
```

（4）判断无向网是否存在唯一的最小生成树（仅考虑连通图）

```
bool AdjListUnDirNetwork<ElemType, WeightType>::hasUniqueMinTree() {
    WeightType min;
    ElemType v1, v2;
    CloseArcType<ElemType, WeightType> * closearc;
    int u0 = 0; // 0为第一个顶点

    int u, v, k;                          // 表示顶点的临时变量
    closearc = new CloseArcType<ElemType, WeightType>[vexNum];  // 分配存储空间
    for (v = 0; v < vexNum; v++)    {   // 初始化辅助数组adjVex，并对顶点作标志，此时U = {v0}
        closearc[v].nearvertex = u0;
        closearc[v].lowweight = GetWeight( v1: u0,  v2: v);
    }
    closearc[u0].nearvertex = -1;
    closearc[u0].lowweight = 0;
```

```
for (k = 1; k < vexNum; k++) {  // 选择生成树的其余g.GetVexNum() - 1个顶点
    min = GetInfinity();
    v = u0;  // 选择使得边<w, adjVex[w]>为连接V-U到U的具有最小权值的边
    for (u = 0; u < vexNum; u++)
        if (closearc[u].lowweight != 0 && closearc[u].lowweight < min) {
            v = u;
            min = closearc[u].lowweight;
        }
    for(int i=0;i<vexNum;i++)   // 若有两个顶点到同一树内顶点的距离相同且是目前的最短距离，则不唯一
        for(int j=i+1;j<vexNum;j++)
            if(closearc[i] == closearc[j] && (i == v || j == v))
                return false;
```

```
    if (v != u0) {
        GetElem( v: closearc[v].nearvertex, &: v1);
        GetElem(v, &: v2);
        closearc[v].lowweight = 0;      // 将w并入U
        for (u = FirstAdjVex(v); u != -1 ; u = NextAdjVex( v1: v, v2: u))     // 新顶点并入U后重新选择最小边
            if (closearc[u].lowweight != 0 && (GetWeight( v1: v, v2: u) < closearc[u].lowweight)) {   // <v, w>为新的最小边
                closearc[u].lowweight = GetWeight( v1: v, v2: u);
                closearc[u].nearvertex = v;
            }else if(closearc[u].lowweight != 0 && (GetWeight( v1: v, v2: u) == closearc[u].lowweight))
                return false;   // 若有一点到两个树内点距离相同，则不唯一
    }
}
delete []closearc;          // 释放存储空间
return true;
```

3、实验结果

显示用于测试的无向图：

1. 无向网清空.
2. 显示无向网.
3. 输出连通分支数.
4. 设置指定顶点的值.
5. 删除顶点.
6. 插入顶点.
7. 删除边.
8. 插入边.
9. 设置指定边的权.

a.求最小生成树

b.是否有唯一最小生成树

c.显示各个顶点的度数

0. 退出

选择功能(0~c):2

无向网共有6个顶点，9条边。

```
0:      A-->(5,19)-->(2,46)-->(1,34)
1:      B-->(4,12)-->(0,34)
2:      C-->(5,25)-->(3,17)-->(0,46)
3:      D-->(5,25)-->(4,38)-->(2,17)
4:      E-->(5,26)-->(3,38)-->(1,12)
5:      F-->(4,26)-->(3,25)-->(2,25)-->(0,19)
```

统计各个顶点的度数：

```
选择功能(0~c):c
顶点A的度数为：3
顶点B的度数为：2
顶点C的度数为：3
顶点D的度数为：3
顶点E的度数为：3
顶点F的度数为：4
```

求该图的连通分量数目

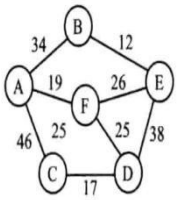
```
选择功能(0~c):3

连通分支数为1
```

Kruskal、Prim、“破圈”算法求最小生成树

```
Kruskal算法：最小生成树的边及权值为：
边:( B, E ) 权:12
边:( C, D ) 权:17
边:( A, F ) 权:19
边:( C, F ) 权:25
边:( A, B ) 权:34
Prim算法：最小生成树的边及权值为：
边:( A, F ) 权:19
边:( F, C ) 权:25
边:( C, D ) 权:17
边:( F, E ) 权:26
边:( E, B ) 权:12
破圈算法：最小生成树的边及权值为：
边:( A, F ) 权:19
边:( B, E ) 权:12
边:( E, F ) 权:26
边:( F, D ) 权:25
边:( D, C ) 权:17
```

判断该图是否有唯一最小生成树

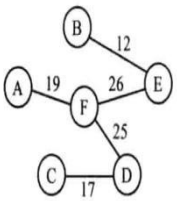
```
b.是否有唯一最小生成树
c.显示各个顶点的度数
0. 退出
选择功能(0~c):b
 不存在唯一最小生成树
```



(a) 网络

	A	B	C	D	E	F
A	0	34	46	∞	∞	19
B	34	0	∞	∞	12	∞
C	46	∞	0	17	∞	25
D	∞	∞	17	0	38	25
E	∞	12	∞	38	0	26
F	19	∞	25	25	26	0

(b) 邻接矩阵



(c) 最小生成树

实验现象与预想的实验现象一致。

4、算法分析

1. CountDegree 函数的时间复杂度为 O(n),空间复杂度为 O(1);
2. CountInDegree 函数的时间复杂度为 O(n+e),空间复杂度为 O(n);
3. MiniSpanTreeBreakCycle 函数时间复杂度为 O(n2),空间复杂度为 O(n);
4. hasUniqueMinTree 函数的时间复杂度为 O(n2)， 空间复杂度为 O(n);
5. n 为顶点个数，e 为边的数量

5、总结与心得

在本次实验的过程中我们小组通过讨论，实践，查阅资料等方法完成了实验任务。通过实验，我们对于邻接表的基本概念以及其基本功能的实现有了更深入的理解。此外，我们还复习了最小堆，并查集等并运用到了解决问题中，学习了求最小生成树的其他算法，如“破圈法”，并亲自编写代码将其实现。总之，通过此次实验，我们提高了算法能力，增加了数据结构知识。

四、分工说明

李昀哲：算法设计，代码编写，ppt 制作

唐铭锋：算法设计，代码编写，撰写报告

刘沛根：算法设计，代码编写，撰写报告

李正宇：算法设计，代码编写，ppt 制作