第 3 章　树与图的生成树

树是一种特殊的图，图的生成树是图的一种特殊子图。本章介绍树、森林、无向连通图的生成树、最小生成树等概念。本章主要讨论了求解无向连通图最小生成树的 3 种算法：克鲁斯卡尔(Kruskal)算法、Boruvka 算法和普里姆(Prim)算法，以及判断生成树是否唯一的方法。有向图的生成树不在本章的讨论范围。

3.1　树 与 森 林

3.1.1　树

树(Tree)：如果一个无向连通图中不存在回路，则这种图称为树，因此树是一种特殊的图。也可以从其他的角度来定义树，详见数据结构方面的书。

例如，图 3.1(a)所示的无向连通图存在回路，所以它不是一棵树。但可以从中去掉构成回路的边，如在图 3.1(b)中去掉了边(1, 4)和(6, 7)，这样图中就不存在回路了，因此该图就是一棵树。当然，去掉边(3, 4)和(5, 6)也可以构造一棵树。

为什么不存在回路的连通图被称为树呢？因为可以把这种图改画成一棵倒立的树。在图 3.1(c)和 3.1(d)中，分别将图 3.1(b)改画成根为顶点 4 的树和根为顶点 5 的树。

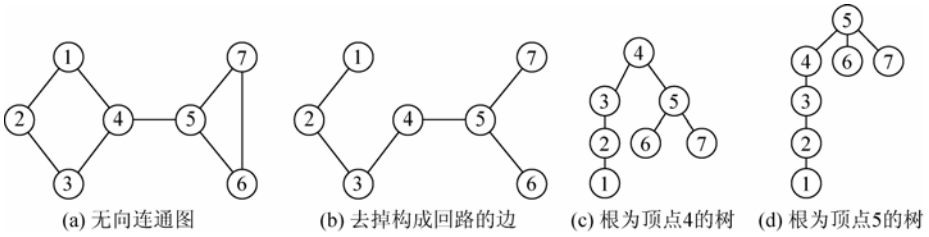


图 3.1　树

3.1.2　森林

森林(Forest)：如果一个无向图中包含了几棵树，那么该无向图可以称为森林。很明显森林是非连通图。例如，图 3.2 所示为一个包含 3 棵树的森林。

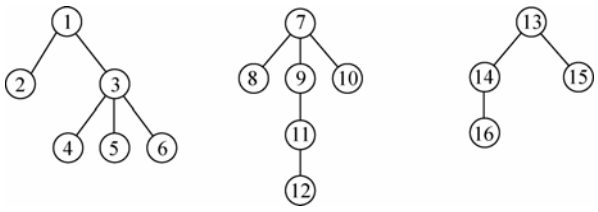


图 3.2　森林

3.2　生成树及最小生成树

3.2.1　生成树

本书 1.1.7 节已经介绍了生成树的概念，本节将进一步讨论生成树。

生成树(Spanning Tree)：无向连通图 G 的一个子图如果是一棵包含 G 的所有顶点的树，则该子图称为 G 的生成树。生成树是连通图的极小连通子图。这里所谓极小是指：若在树中任意增加一条边，则将出现一个回路；若去掉一条边，将会使之变成非连通图。

按照生成树的定义，包含 n 个顶点的连通图，其生成树有 n 个顶点、$n-1$ 条边。

根据第 2 章的知识可知，用不同的遍历方法遍历图，可以得到不同的生成树；从不同的顶点出发遍历图，也能得到不同的生成树。所以有时需要根据应用的需求选择合适的边构造一个生成树，如本章所要讨论的最小生成树。

3.2.2　最小生成树

对于一个带权的无向连通图(即无向网)来说，如何找出一棵生成树，使得各边上的权值总和达到最小，这是一个有着实际意义的问题。例如，在 n 个城市之间建立通信网络，至少要架设 $n-1$ 条线路，这时自然会考虑：如何选择这 $n-1$ 条线路，使得总造价最少？

在每两个城市之间都可以架设一条通信线路，并要花费一定的代价。若用图的顶点表示 n 个城市，用边表示两个城市之间架设的通信线路，用边上的权值表示架设该线路的造价，就可以建立一个通信网络。对于这样一个有 n 个顶点的网络，可以有不同的生成树，每棵生成树都可以构成通信网络。现在希望能根据各边上的权值，选择一棵总造价最小的生成树，这就是最小生成树的问题。

最小生成树(Minimum Spannirng Tree，MST)或者称为**最小代价生成树** Minimum-cost Spannirng Tree：对无向连通图的生成树，各边的权值总和称为生成树的权，权最小的生成树称为最小生成树。

构造最小生成树的准则有 3 条。

(1) 必须只使用该网络中的边来构造最小生成树。

(2) 必须使用且仅使用 $n-1$ 条边来连接网络中的 n 个顶点。

(3) 不能使用产生回路的边。

构造最小生成树的算法主要有：克鲁斯卡尔(Kruskal)算法、Boruvka 算法和普里姆(Prim)算法，它们都得遵守以上准则。它们都采用了一种逐步求解的策略。

如果一个连通无向网为 $G(V, E)$，顶点集合 V 中有 n 个顶点。最初先构造一个包括全部 n 个顶点和 0 条边的森林 Forest = { $T_0, T_1, \cdots, T_{n-1}$ }，以后每一步向 Forest 中加入一条边，它应当是一端在 Forest 中的某一棵树 T_i 上，而另一端不在 T_i 上的所有边中具有最小权值的边。由于边的加入，使 Forest 中的某两棵树合并为一棵。经过 $n-1$ 步，最终得到一棵有 $n-1$ 条边的、各边权值总和达到最小的生成树。

接下来分别讨论克鲁斯卡尔(Kruskal)算法、Boruvka 算法和普里姆(Prim)算法，以及判定最小生成树是否唯一的方法。

3.3 克鲁斯卡尔(Kruskal)算法

3.3.1 Kruskal 算法思想

Kruskal 算法的基本思想是以边为主导地位，始终都是选择当前可用的最小权值的边。具体如下。

(1) 设一个有 n 个顶点的连通网络为 $G(V, E)$，最初先构造一个只有 n 个顶点，没有边的非连通图 $T = \{ V, Ø \}$，图中每个顶点自成一个连通分量。

(2) 当在 E 中选择一条具有最小权值的边时，若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中；否则，即这条边的两个顶点落在同一个连通分量上，则将此边舍去(此后永不选用这条边)，重新选择一条权值最小的边。

(3) 如此重复下去，直到所有顶点在同一个连通分量上为止。

图 3.3(a)所示的无向网，其邻接矩阵如图 3.3(b)所示。利用克鲁斯卡尔算法构造最小生成树的过程如图 3.3(c)所示，首先构造的是只有 7 个顶点，没有边的非连通图。剩下的过程如下(图 3.3(c)中的每条边旁边的序号跟下面的序号是一致的)。

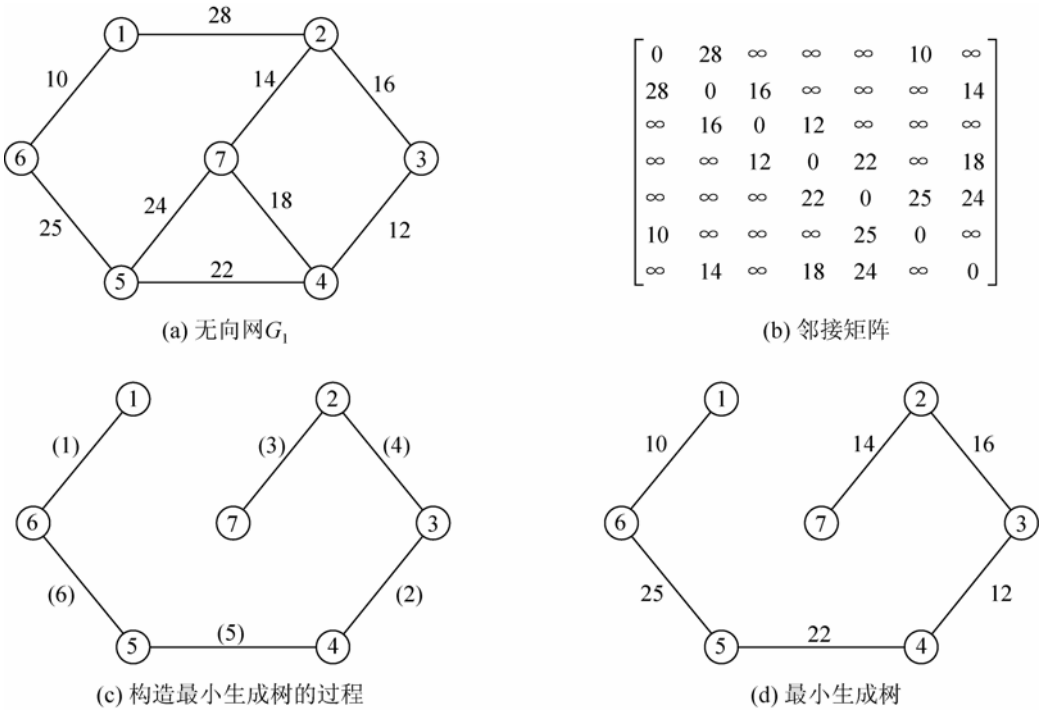


图 3.3 克鲁斯卡尔算法的基本思想

(1) 在边的集合 E 中选择权值最小的边，即(1, 6)，权值为 10。

(2) 在集合 E 剩下的边中选择权值最小的边，即(3, 4)，权值为 12。

(3) 在集合 E 剩下的边中选择权值最小的边，即(2, 7)，权值为 14。

(4) 在集合 E 剩下的边中选择权值最小的边，即(2, 3)，权值为 16。

(5) 在集合 *E* 剩下的边中选择权值最小的边，即(7, 4)，权值为 18，但这条边的两个顶点位于同一个连通分量上，所以要舍去；继续选择一条权值最小的边，即(4, 5)，权值为 22。

(6) 在集合 *E* 剩下的边中选择权值最小的边，即(7, 5)，权值为 24，但这条边的两个顶点位于同一个连通分量上，所以要舍去；继续选择一条权值最小的边，即(6, 5)，权值为 25。

至此，最小生成树构造完毕，最终构造的最小生成树如图 3.3(d)所示，生成树的权为 99。

克鲁斯卡尔算法的伪代码为：

```
T=(V, φ);
while( T 中所含边数 < n-1 )
{
    从 E 中选取当前权值最小的边(u, v);
    从 E 中删除边(u, v);
    if( 边(u, v)的两个顶点落在两个不同的连通分量上 )
        将边(u, v)并入 T 中;
}
```

Kruskal 算法在每选择一条边加入到生成树集合 *T* 时，有两个关键步骤如下。

(1) 从 *E* 中选择当前权值最小的边(*u*, *v*)，实现时可以用最小堆来存放 *E* 中所有的边；或者将所有边的信息(边的两个顶点、权值)存放到一个数组 edges 中，并将 edges 数组按边的权值从小到大进行排序，然后按先后顺序选用每条边。3.3.3 节例 3.1 中采用的是后一种方法。

(2) 选择权值最小的边后，要判断两个顶点是否属于同一个连通分量，如果是，则要舍去；如果不是，则选用，并将这两个顶点分别所在的连通分量合并成一个连通分量。在实现时可以使用并查集来判断两个顶点是否属于同一个连通分量以及将两个连通分量合并成一个连通分量。3.3.2 节将简单地介绍并查集的原理及使用方法。

3.3.2　等价类与并查集

并查集主要用来解决判断两个元素是否同属一个集合，以及把两个集合合并成一个集合的问题。

“同属一个集合”关系是一个等价关系，因为它满足**等价关系**(Equivalent Relation)的 3 个条件(或称为性质)。

(1) 自反性：如 $X \equiv X$，则 $X \equiv X$。(假设用“$X \equiv Y$”表示“*X* 与 *Y* 等价”。)

(2) 对称性：如 $X \equiv Y$，则 $Y \equiv X$。

(3) 传递性：如 $X \equiv Y$，且 $Y \equiv Z$，则 $X \equiv Z$。

如果 $X \equiv Y$，则称 *X* 与 *Y* 是一个**等价对**(Equivalence)。

等价类(Equivalent Class)：设 *R* 是集合 *A* 上的等价关系，对任何 $a \in A$，集合 $[a]_R = \{ x \mid x \in A$，且 $aRx \}$称为元素 *a* 形成的 *R* 等价类，其中，*aRx* 表示 *a* 与 *x* 等价。所谓元素 *a* 的等价类，通俗地讲，就是所有跟 *a* 等价的元素构成的集合。

等价类应用：设初始时有一集合 $S = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \}$；依次读若干事先定义的等价对 $1 \equiv 5$，$4 \equiv 2$，$7 \equiv 11$，$9 \equiv 10$，$8 \equiv 5$，$7 \equiv 9$，$4 \equiv 6$，$3 \equiv 12$，$12 \equiv 1$；现在需要根据这些等价对将集合 *S* 划分成若干个等价类。

在每次读入一个等价对后，把等价类合并起来。初始时，各个元素自成一个等价类(用{ }表示一个等价类)。在每读入一个等价对后，各等价类的变化依次如下。

初始：　{ 1 }, { 2 }, { 3 }, { 4 }, { 5 }, { 6 }, { 7 }, { 8 }, { 9 }, { 10 }, { 11 }, { 12 }。

1≡5：　{ 1, 5 }, { 2 }, { 3 }, { 4 }, { 6 }, { 7 }, { 8 }, { 9 }, { 10 }, { 11 }, { 12 }。

4≡2：　{ 1, 5 }, { 2, 4 }, { 3 }, { 6 }, { 7 }, { 8 }, { 9 }, { 10 }, { 11 }, { 12 }。

7≡11：{ 1, 5 }, { 2, 4 }, { 3 }, { 6 }, { 7, 11 }, { 8 }, { 9 }, { 10 }, { 12 }。

9≡10：{ 1, 5 }, { 2, 4 }, { 3 }, { 6 }, { 7, 11 }, { 8 }, { 9, 10 }, { 12 }。

8≡5：　{ 1, 5, 8 }, { 2, 4 }, { 3 }, { 6 }, { 7, 11 }, { 9, 10 }, { 12 }。

7≡9：　{ 1, 5, 8 }, { 2, 4 }, { 3 }, { 6 }, { 7, 9, 10, 11 }, { 12 }。

4≡6：　{ 1, 5, 8 }, { 2, 4, 6 }, { 3 }, { 7, 9, 10, 11 }, { 12 }。

3≡12：{ 1, 5, 8 }, { 2, 4, 6 }, { 3, 12 }, { 7, 9, 10, 11 }。

12≡1：{ 1, 3, 5, 8, 12 }, { 2, 4, 6 }, { 7, 9, 10, 11 }。

并查集(Union-Find Set)这个数据结构可以方便快速地实现这个问题。并查集对这个问题的处理思想是：初始时把每一个对象看作是一个单元素集合；然后依次按顺序读入等价对后，将等价对中的两个元素所在的集合合并。在此过程中将重复地使用一个**搜索**(Find)运算，确定一个元素在哪一个集合中。当读入一个等价对 $A≡B$ 时，先检测 A 和 B 是否同属一个集合，如果是，则不用合并；如果不是，则用一个**合并**(Union)运算把 A、B 所在的集合合并，使这两个集合中的任两个元素都是等价的(依据是等价的传递性)。因此，并查集在处理时主要有**搜索**和**合并**两个运算。

为了方便并查集的描述与实现，通常把先后加入到一个集合中的元素表示成一个树结构，并用根结点的序号来代表这个集合。因此定义一个 parent[n]的数组，parent[i]中存放的就是结点 i 所在的树中结点 i 父亲结点的序号。例如，如果 parent[4] = 5，就是说 4 号结点的父亲是 5 号结点。约定：如果结点 i 的父结点(即 parent[i])是负数，则表示结点 i 就是它所在集合的根结点，因为集合中没有结点的序号是负的；并且用负的绝对值作为这个集合中所含结点个数。例如，如果 parent[7] = −4，说明 7 号结点就是它所在集合的根结点，这个集合有 4 个元素。初始时，所有结点的 parent 值为−1，说明每个结点都是根结点(N 个独立结点集合)，只包含一个元素(就是自己)。

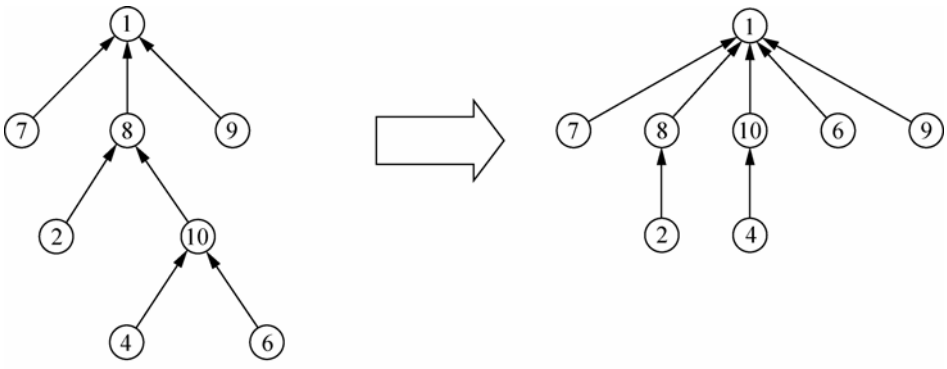
实现并查集数据结构主要有 3 个函数。代码如下。

```
void UFset( )   //初始化
{
    for( int i=0; i<N; i++ )
        parent[i]=-1;
}
int Find( int x )   //查找并返回结点 x 所属集合的根结点
{
    int s;  //查找位置
            //一直查找到 parent[s]为负数(此时的 s 即为根结点)为止
    for( s=x; parent[s]>=0; s=parent[s] );
    while( s!=x )   //优化方案——压缩路径，使后续的查找操作加速
    {
        int tmp=parent[x];
```

```
            parent[x]=s;
            x=tmp;
        }
        return s;
    }
    //R1 和 R2 是两个元素，属于两个不同的集合，现在合并这两个集合
    void Union( int R1, int R2 )
    {
        //r1 为 R1 的根结点，r2 为 R2 的根结点
        int r1=Find(R1),  r2=Find(R2);
        int tmp=parent[r1]+parent[r2]; //两个集合结点个数之和(负数)
        //如果 R2 所在树结点个数 > R1 所在树结点个数
        //注意 parent[r1]和 parent[r2]都是负数
        if( parent[r1]>parent[r2] ) //优化方案——加权法则
        {
            parent[r1]=r2;      //将根结点 r1 所在的树作为 r2 的子树(合并)
            parent[r2]=tmp;     //更新根结点 r2 的 parent[ ]值
        }
        else
        {
            parent[r2]=r1;      //将根结点 r2 所在的树作为 r1 的子树(合并)
            parent[r1]=tmp;     //更新根结点 r1 的 parent[ ]值
        }
    }
```

接下来对 Find 函数和 Union 函数的实现过程做详细解释。

Find 函数：在 Find 函数中如果仅仅靠一个循环来直接得到结点所属集合的根结点，那么通过多次的 Union 操作就会有很多结点在树的比较深层次中，再查找起来就会很费时。可以通过**压缩路径**来加快后续的查找速度：增加一个 While 循环，每次都把从结点 x 到集合根结点的路径上经过的结点直接设置为根结点的子女结点。虽然这增加了时间，但以后的查找会更快。如图 3.4 所示，假设从结点 x=6 开始压缩路径，则从结点 6 到根结点 1 的路径上有 3 个结点：6、10、8，压缩后，这 3 个结点都直接成为根结点的子女结点，如图 3.4(b)所示。



(a) 压缩前　　(b) 从x=6压缩路径

图 3.4　并查集：Find 函数中的路径压缩

Union 函数：两个集合并时，任一方可作为另一方的子孙。怎样来处理呢？现在一般采用加权合并，把两个集合中元素个数少的根结点作为元素个数多的根结点的子女结点。这样处理有什么优势呢？直观上看，可以减少树中的深层元素的个数，减少后续查找时间。

例如，假设从 1 开始到 n，不断合并第 i 个结点与第 i+1 个结点，采用加权合并思路的过程如图 3.5 所示(各子树根结点上方的数字为其 parent[ ]值)。这样查找任一结点所属集合的时间复杂度几乎都是 $O(1)$！

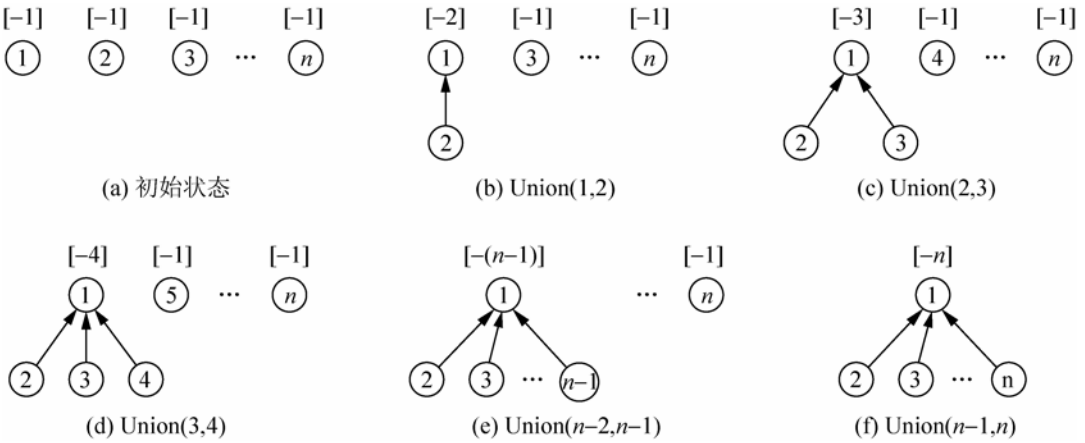


图 3.5　并查集：加权合并

不用加权规则可能会得到如图 3.6 所示的结果。这就是典型的退化树(只有一个叶结点，且每个非叶结点只有一个子结点)现象，再查找起来就会很费时，例如查找结点 n 的根结点时复杂度为 $O(n)$。

图 3.6　并查集：合并时不加权的结果

图 3.7 所示为用并查集实现前面的等价类应用例子时完整的查找和合并过程。

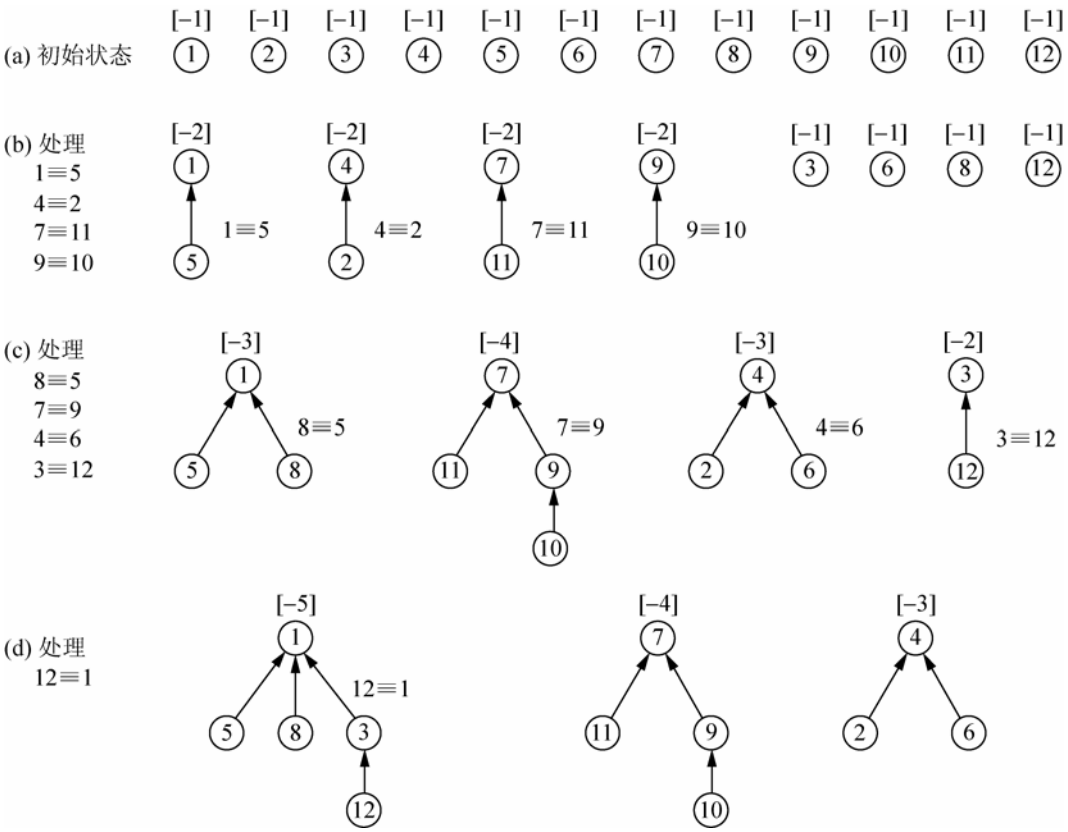


图 3.7　并查集：完整的查找合并过程

3.3.3　Kruskal 算法实现

本节首先以图 3.3(a)所示的无向网为例，解释 Kruskal 算法执行过程中并查集的初始化、路径压缩、合并等过程，如图 3.8 所示。

如图 3.8(a)所示，并查集的初始状态为各个顶点各自构成一个连通分量，每个顶点上方的数字表示其 parent[ ]元素值。

图 3.8(b)所示是 9 条边组成的数组，并且已经按照权值从小到大排好序了，在 Kruskal 算法执行过程当中，从这个数组中依次选用每条边，如果某条边的两个顶点位于同一个连通分量上，则要舍去这条边。

在图 3.8(c)中，依次选用(1, 6)、(3, 4)、(2, 7)这 3 条边后，顶点 1 和 6 组成一个连通分量，顶点 3 和 4 组成一个连通分量，顶点 2、7 组成一个连通分量，顶点 5 单独构成一个连通分量。

在图 3.8(d)中，选用边(2, 3)后，要合并顶点 2 和顶点 3 分别所在的连通分量，合并的结果是顶点 3 成为顶点 2 所在子树中根结点(即顶点 2)的子女。

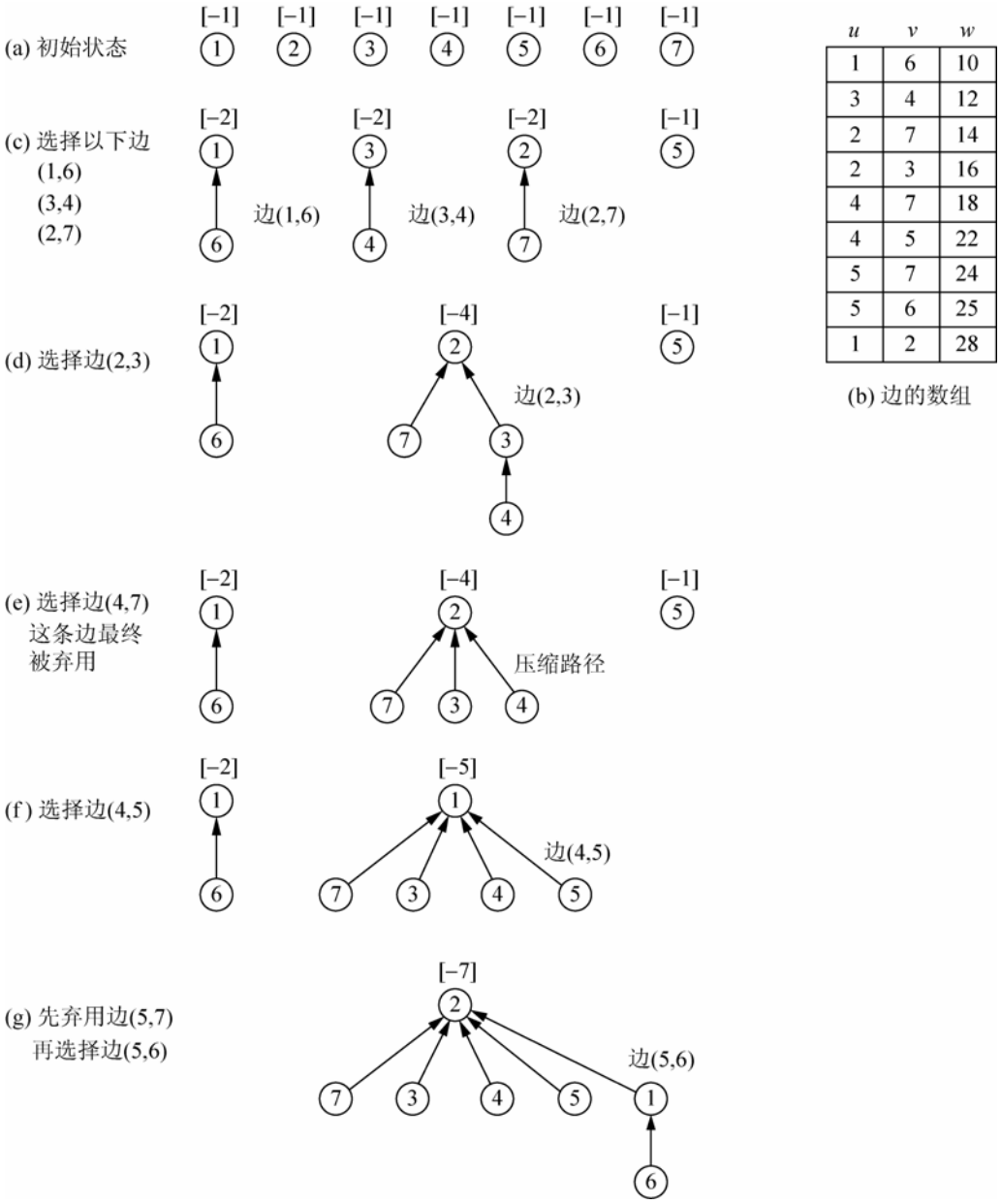
在图 3.8(e)中要特别注意，虽然选用边(4, 7)时，因为这两个顶点位于同一个连通分量上，这条边将会被弃用。但在查找顶点 4 的根结点时，会压缩路径，使得从顶点 4 到根结点的路径上的顶点都成为根结点的子女结点，这样有利于以后的查找。

在图 3.8(f)中，选用边(4, 5)后，要将顶点 5 合并到顶点 4 所在的连通分量上，合并的

结果是顶点 5 成为顶点 4 所在子树中根结点(即顶点 2)的子女。

在图 3.8(g)中，首先弃用边(5, 7)，再选用边(5, 6)，要将顶点 6 所在的连通分量合并到顶点 5 所在的连通分量上，因为前一个连通分量的顶点个数较少。

至此，Kruskal 算法执行完毕，选用了 $n-1$ 条边，连接 n 个顶点。



u	v	w
1	6	10
3	4	12
2	7	14
2	3	16
4	7	18
4	5	22
5	7	24
5	6	25
1	2	28

(b) 边的数组

图 3.8　Kruskal 算法的实现过程

例 3.1　利用 Kruskal 算法求图 3.3(a)所示的无向网的最小生成树，并输出依次选择的各条边及最终求得的最小生成树的权。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 n 和边数 m，然后输入

m 条边的数据。每条边的数据格式为：*u v w*，分别表示这条边的两个顶点及边上的权值。顶点序号从 1 开始计起。

分析：

在下面的代码中，首先读入边的信息，存放到数组 edges 中，并按权值从小到大进行排序。Kruskal 函数用于实现 Kruskal 算法：首先初始化并查集，然后从 edges 数组中依次选用每条边，如果这条边的两个顶点位于同一个连通分量，则要弃用这条边；否则合并这两个顶点所在的连通分量。

代码如下：

```
#define MAXN 11     //顶点个数的最大值
#define MAXM 20     //边的个数的最大值
struct edge         //边
{
    int u, v, w;    //边的顶点、权值
}edges[MAXM];       //边的数组
int parent[MAXN];   //parent[i]为顶点 i 所在集合对应的树中的根结点
int n, m;           //顶点个数、边的个数
int i, j;           //循环变量
void UFset( )       //初始化
{
    for( i=1; i<=n; i++ )  parent[i]=-1;
}
int Find( int x )   //查找并返回节点 x 所属集合的根结点
{
    int s;          //查找位置
    for( s=x; parent[s]>=0; s=parent[s] )
        ;
    while( s!=x )   //优化方案——压缩路径，使后续的查找操作加速
    {
        int tmp=parent[x];
        parent[x]=s;
        x=tmp;
    }
    return s;
}
//将两个不同集合的元素进行合并，使两个集合中任意两个元素都连通
void Union( int R1, int R2 )
{
    int r1=Find(R1),  r2=Find(R2);  //r1 为 R1 的根结点，r2 为 R2 的根结点
    int tmp=parent[r1]+parent[r2]; //两个集合结点个数之和(负数)
    //如果 R2 所在树结点个数 > R1 所在树结点个数(注意 parent[r1]是负数)
    if( parent[r1]>parent[r2] ) //优化方案——加权法则
    {
        parent[r1]=r2;  parent[r2]=tmp;
    }
    else
```

```
    {
        parent[r2]=r1;  parent[r1]=tmp;
    }
}
int cmp( const void *a, const void *b )    //实现从小到大排序的比较函数
{
    edge aa=*(const edge *)a;  edge bb=*(const edge *)b;
    return aa.w-bb.w;
}
void Kruskal( )
{
    int sumweight=0;    //生成树的权值
    int num=0;          //已选用的边的数目
    int u, v;           //选用边的两个顶点
    UFset( );           //初始化 parent 数组
    for( i=0; i<m; i++ )
    {
        u=edges[i].u; v=edges[i].v;
        if( Find(u) != Find(v) )
        {
            printf( "%d %d %d\n", u, v, edges[i].w );
            sumweight+=edges[i].w;  num++;
            Union( u, v );
        }
        if( num>=n-1 )  break;
    }
    printf( "weight of MST is %d\n", sumweight );
}
void main( )
{
    int u, v, w;                //边的起点和终点及权值
    scanf( "%d%d", &n, &m );    //读入顶点个数 n
    for( int i=0; i<m; i++ )
    {
        scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
        edges[i].u=u;  edges[i].v=v;  edges[i].w=w;
    }
    qsort( edges, m, sizeof(edges[0]), cmp );//对边按权值从小到大排序
    Kruskal( );
}
```

该程序的运行示例如下：

输入：

```
7 9
1 2 28
1 6 10
2 3 16
2 7 14
3 4 12
```

输出：

```
1 6 10
3 4 12
2 7 14
2 3 16
4 5 22
5 6 25
```

```
4 5 22                                    weight of MST is 99
4 7 18
5 6 25
5 7 24
```

Kruskal 算法的时间复杂度分析：在例 3.1 的代码中，执行 Kruskal 函数前进行了一次排序操作，时间代价为 $\log_2 m$；在 Kruskal 函数中，最多需要进行 m 次循环，共执行 $2m$ 次 Find( )操作，$n-1$ 次 Union 操作，其时间代价分别为 $O(2m\log_2 n)$和 $O(n)$。所以 Kruskal 算法的时间复杂度为：$O(\log_2 m + 2m\log_2 n + n)$。因此，Kruskal 算法的时间复杂度主要取决于边的数目，比较适合于稀疏图。

3.3.4　Boruvka 算法

Boruvka 算法是最古老的一个 MST 算法，其思想类似于 Kruskal 算法思想。Boruvka 算法可以分为两步：① 对图中各顶点，将与其关联、具有最小权值的边选入 MST，得到的是由 MST 子树构成的森林；② 在图中陆续选择可以连接两颗不同子树且具有最小权值的边，将子树合并，最终构造 MST。

例如，对图 3.3(a)所示的无向连通图，与顶点 1 关联的、权值最小的边为边(1, 6)，将其选入 MST，与顶点 6 关联的、权值最小的边也为边(1, 6)，这条边将顶点 1 和 6 连接成 MST 中的第 1 棵子树；按照类似的方法，得到另外两棵子树：顶点 2、顶点 7 组成的第二棵子树，顶点 3、4、5 组成第 3 棵子树，如图 3.9(a)所示。这是第 1 步。

第 2 步，选择边(2, 3)将第 2、3 棵子树合并，以及选择边(5, 6)再将第 1 棵子树合并进来，至此 MST 构造完毕，如图 3.9(b)所示。

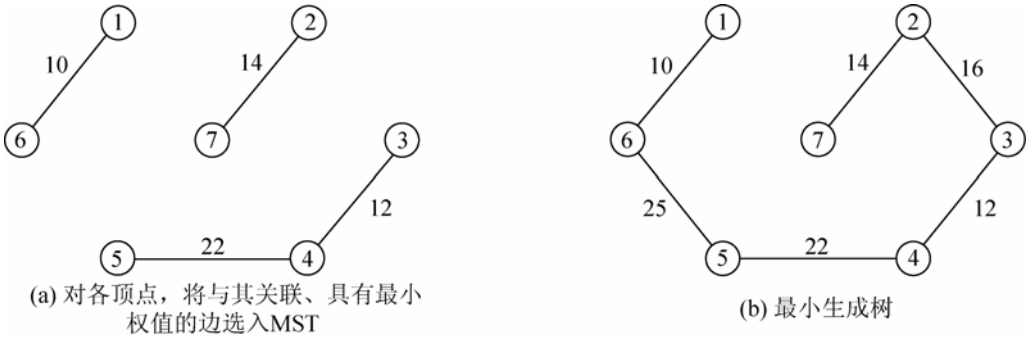


图 3.9　Boruvka 算法的实现过程

Boruvka 算法在实现时也需要使用并查集，读者可以在理解 Boruvka 算法思想的基础上编程实现例 3.1。

3.3.5　例题解析

以下通过两道例题的分析，再详细介绍克鲁斯卡尔算法的基本思想及其实现方法。

例 3.2　剑鱼行动(Swordfish)

题目来源：

Zhejiang University Local Contest 2002, Preliminary, ZOJ1203

题目描述：

给定平面上 N 个城市的位置，计算连接这 N 个城市所需线路长度总和的最小值。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为一个正整数 N，$0 \leqslant N \leqslant 100$，代表需要连接的城市数目；接下来有 N 行，每行为两个实数 X 和 Y，$-10\,000 \leqslant X, Y \leqslant 10\,000$，表示每个城市的 X 坐标和 Y 坐标。输入文件中最后一行为 $N=0$，代表输入结束。

输出描述：

对输入文件中每个测试数据，计算连接所有城市所需线路长度总和的最小值。每对城市之间的线路为连接这两个城市的直线。输出格式为：第 1 行为"Case #n:"，其中 n 为测试数据的序号，序号从 1 开始计起；第 2 行为"The minimal distance is: d"，其中 d 为求得的最小值，精确到小数点后两位有效数字。每两个测试数据的输出之间输出一个空行。

样例输入：

```
5
0 0
0 1
1 1
1 0
0.5 0.5
0
```

样例输出：

```
Case #1:
The minimal distance is: 2.83
```

分析：

在本题中，任意两个顶点之间都有边连通，权值为这两个顶点之间的距离。将所有边求出并存储到边的数组 edges 中后，按照 Kruskal 算法求解即可。

代码如下：

```
#define MAXN 100    //顶点个数的最大值
#define MAXM 5000   //边的个数的最大值
struct edge         //边
{
    int u, v;       //边的顶点
    double w;       //权值(两个点之间的距离)
}edges[MAXM];       //边的数组
int parent[MAXN];   //parent[i]为顶点 i 所在集合对应的树中的根结点
int N, m;           //顶点个数、边的个数
double X[MAXN], Y[MAXN];    //每个顶点的 X 坐标和 Y 坐标
int i, j;                   //循环变量
double sumweight;           //生成树的权值
void UFset( )               //初始化
{
    for( i=0; i<N; i++ )
        parent[i]=-1;
}
int Find( int x )   //查找并返回结点 x 所属集合的根结点
{
    int s;          //查找位置
```

```
    for( s=x; parent[s]>=0; s=parent[s] )
        ;
    while( s!=x )   //优化方案——压缩路径，使后续的查找操作加速
    {
        int tmp=parent[x];
        parent[x]=s;
        x=tmp;
    }
    return s;
}
//将两个不同集合的元素进行合并，使两个集合中任意两个元素都连通
void Union( int R1, int R2 )
{
    int r1=Find(R1),  r2=Find(R2);  //r1 为 R1 的根结点，r2 为 R2 的根结点
    int tmp=parent[r1]+parent[r2]; //两个集合结点个数之和(负数)
    //如果 R2 所在树结点个数>R1 所在树结点个数(注意 parent[r1]是负数)
    if( parent[r1]>parent[r2] )    //优化方案——加权法则
    {
        parent[r1]=r2;
        parent[r2]=tmp;
    }
    else
    {
        parent[r2]=r1;
        parent[r1]=tmp;
    }
}
int cmp( const void *a, const void *b )    //实现从小到大排序的比较函数
{
    edge aa=*(const edge *)a;
    edge bb=*(const edge *)b;
    if( aa.w>bb.w ) return 1;
    else return -1;
}
void Kruskal( )
{
    int num=0;  //已选用的边的数目
    int u, v;   //选用边的两个顶点
    UFset( );   //初始化 parent 数组
    for( i=0; i<m; i++ )
    {
        u=edges[i].u; v=edges[i].v;
        if( Find(u)!=Find(v) )
        {
            sumweight+=edges[i].w;  num++;
            Union( u, v );
        }
```

```
            if( num>=N-1 )  break;
        }
    }
    int main( )
    {
        double d;   //两个点之间的距离
        int kase=1;
        while( 1 )
        {
            scanf( "%d", &N );  //读入顶点个数 N
            if( N==0 )  break;
            for( i=0; i<N; i++ )
                scanf( "%lf%lf", &X[i], &Y[i] );
            int mi=0;   //边的序号
            for( i=0; i<N; i++ )
            {
                for( j=i+1; j<N; j++ )
                {
                    d=sqrt( (X[i]-X[j])*(X[i]-X[j])+(Y[i]-Y[j])*(Y[i]-Y[j]) );
                    edges[mi].u=i;  edges[mi].v=j;  edges[mi].w=d;
                    mi++;
                }
            }
            m=mi;
            //对边按权值从小到大的顺序进行排序
            qsort( edges, m, sizeof(edges[0]), cmp );
            sumweight=0.0;
            Kruskal( );
            if( kase>1 ) printf( "\n" );
            printf( "Case #%d:\n", kase );
            printf( "The minimal distance is: %0.2f\n", sumweight );
            kase++;
        }
        return 0;
    }
```

例 3.3　网络(Network)

题目来源：

Northeastern Europe 2001, Northern Subregion, ZOJ1542, POJ1861

题目描述：

Andrew 是某个公司的系统管理员，他计划为他的公司搭建一个新的网络。在新的网络中，有 *N* 个集线器，集线器之间可以通过网线连接。由于公司职员需要通过集线器访问整个网络，因此每个集线器必须能通过网线连接其他集线器(可以通过其他中间集线器来连接)。

由于有不同长度的网线可供选择，而且网线越短越便宜，因此 Andres 所设计的方案必须确保最长的单根网线的长度在所有方案中是最小的。并不是所有集线器之间都可以直接

连接，但 Andrew 会提供集线器之间所有可能的连接。

试帮助 Andrew 设计一个网络，连接所有的集线器并满足前面的条件。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数：N 和 M，N 表示网络中集线器的数目，$2 \leqslant N \leqslant 1\,000$，集线器的编号从 1～$N$；$M$ 表示集线器之间连接的数目，$1 \leqslant M \leqslant 15\,000$。接下来 M 行描述了 M 对连接的信息，每对连接的格式为：所连接的两个集线器的编号，连接这两个集线器所需网线的长度，长度为不超过 10^6 的正整数。两个集线器之间至多有一对连接；每个集线器都不能与自己连接。测试数据保证网络是连通的。

测试数据一直到文件尾。

输出描述：

对输入文件中的每个测试数据，首先输出连接方案中最长的单根网线的长度(必须使得这个值取到最小)；然后输出设计方案：先输出一个整数 P，代表所使用的网线数目；然后输出 P 对顶点，表示每根网线所连接的集线器编号，整数之间用空格或换行符隔开。

样例输入：

```
5 8
1 2 5
1 4 2
1 5 1
2 3 6
2 4 3
3 4 5
3 5 4
4 5 6
```

样例输出：

```
4
4
1 5
1 4
2 4
3 5
```

分析：

本题虽然没有直接要求求解生成树，但连接 N 个集线器的方案中如果有多于 $N-1$ 条边，那么必然存在回路，因此可以去掉某些边，使得剩下的边及所有顶点构成一个生成树，仍然可以连接这 N 个集线器。另外，可以证明对于一个图的最小生成树来说，它的最大边满足在所有生成树的最大边里最小。因此本题实际上就是求最小生成树，并要求输出长度最长的边等信息。

另外，本题中集线器的序号是从 1 开始计起的，下面的代码不使用 parent 数组第 0 个元素，使用第 1～N 个元素。

代码如下：

```
#define MAXN 1001   //顶点个数的最大值
#define MAXM 15001  //边的个数的最大值
struct edge //边
{
    int u, v, w;    //边的顶点、权值
}edges[MAXM];       //边的数组
int parent[MAXN];   //parent[i]为顶点 i 所在集合对应的树中的根结点
int ans[MAXN], ai;  //依次选择的边的序号,数组 ans 的下标
int N, M;           //集线器个数、边的个数
```

```
int num, maxedge;   //选用边的数目，及最长的单根网线的长度
int i, j;           //循环变量
void UFset( )       //初始化
{
    for( i=1; i<=N; i++ )  parent[i]=-1;
}

int Find( int x )   //查找并返回结点 x 所属集合的根结点
{
    int s;          //查找位置
    for( s=x; parent[s]>=0; s=parent[s] )  ;
    while( s!=x )   //优化方案——压缩路径，使后续的查找操作加速
    {
        int tmp=parent[x];
        parent[x]=s;
        x=tmp;
    }
    return s;
}
//将两个不同集合的元素进行合并，使两个集合中任意两个元素都连通
void Union( int R1, int R2 )
{
    int r1=Find(R1),  r2=Find(R2);  //r1 为 R1 的根结点，r2 为 R2 的根结点
    int tmp=parent[r1]+parent[r2]; //两个集合结点个数之和(负数)
    //如果 R2 所在树结点个数 > R1 所在树结点个数(注意 parent[r1]是负数)
    if( parent[r1]>parent[r2] )    //优化方案——加权法则
    {
        parent[r1]=r2;  parent[r2]=tmp;
    }
    else
    {
        parent[r2]=r1;  parent[r1]=tmp;
    }
}
int cmp( const void *a, const void *b )    //实现从小到大排序的比较函数
{
    edge aa=*(const edge *)a;
    edge bb=*(const edge *)b;
    return aa.w-bb.w;
}
void Kruskal( )
{
    ai=0;
    int u, v;   //选用边的两个顶点
    UFset( );   //初始化 parent 数组
    for( i=0; i<M; i++ )
    {
```

```
            u=edges[i].u; v=edges[i].v;
            if( Find(u) != Find(v) )
            {
                ans[ai]=i;  ai++;
                if( edges[i].w > maxedge )
                    maxedge=edges[i].w;
                num++;
                Union( u, v );
            }
            if( num>=N-1 )  break;
        }
    }
    int main( )
    {
        int i;
        while( scanf( "%d%d", &N, &M )!=EOF )
        {
            for( i=0; i<M; i++ )
                scanf( "%d%d%d", &edges[i].u, &edges[i].v, &edges[i].w );
            //对边按权值从小到大的顺序进行排序
            qsort( edges, M, sizeof(edges[0]), cmp );
            maxedge=0; num=0;
            Kruskal( );
            printf( "%d\n", maxedge );
            printf( "%d\n", num );
            for( i=0; i<num; i++ )
                printf( "%d %d\n", edges[ans[i]].u, edges[ans[i]].v );
        }
        return 0;
    }
```

练　　习

3.1　丛林中的道路(Jungle Roads), ZOJ1406, POJ1251

题目描述：

热带小岛 Lagrishan 的首领目前面临一个问题：几年前由外国资本投资在村子之间修建了道路，但是丛林无情地布满了道路，因此，整个道路维护的费用是很高的。委员会必须有选择地终止某些道路的维护。图 3.10(a)所示的地图表示目前在使用的道路，以及每条道路维护的费用。当然，必须保证村子之间都有道路相连。首领必须告诉委员会维护这些道路的最小费用。

在图 3.10 的地图中，村子用字母 *A* 到 *I* 标明，图 3.10(b)所示为需要维护的道路，这些道路可以连接所有村庄，并且总的费用是最少的，为 216。你的任务是编写一个程序，解决此问题。

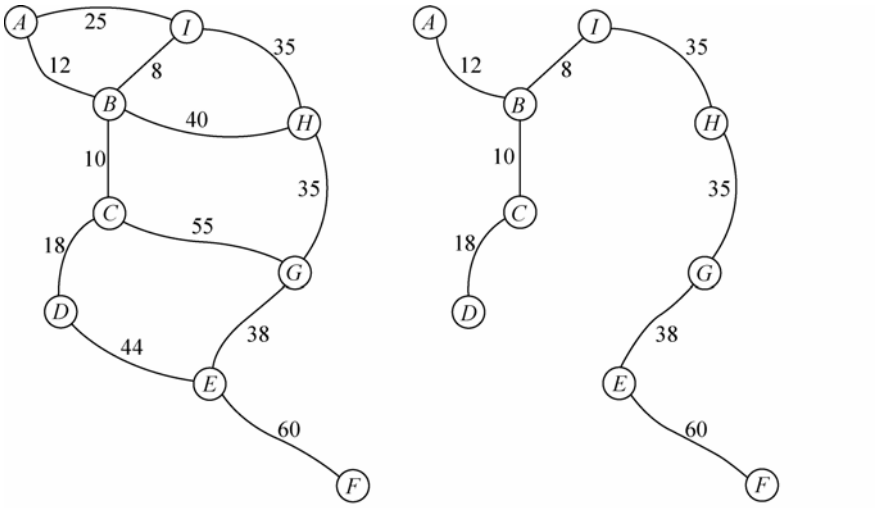


图 3.10　丛林中的道路

输入描述：

输入文件包含 1～100 个测试数据，最后一行是 0，表示输入结束。每个测试数据第 1 行为一个整数 n，表示村子的个数，$1<n<27$，这 n 个村子用字母表的前 n 个大写字母标明。每个测试数据中接下来是 $n-1$ 行数据，每行的第 1 个数据都是表示村子的大写字母，这 $n-1$ 行数据按字母顺序排列。最后一个村子没有数据。每行数据代表一个村子，首先是这个村子的字母，然后是整数 k，表示该村子与字母靠后的村子相连的道路有 k 条，如果 k 大于 0，接下来是 k 条道路的信息，每条道路的信息首先是道路另一端村子的标识字母，然后是这条道路的维护费用，该费用是一个小于 100 的整数。该行所有数据之间用空格隔开。道路网络保证所有村子都是相连的，道路网络中不会超过 75 条道路，与每个村子相连的道路不会超过 15 条。

输出描述：

对每个测试数据，输出道路网络的最小维护费用。

样例输入：

```
9
A 2 B 12 I 25
B 3 C 10 H 40 I 8
C 2 D 18 G 55
D 1 E 44
E 2 F 60 G 38
F 0
G 1 H 35
H 1 I 35
```

样例输出：

```
216
```

3.2　网络设计(Networking), ZOJ1372, POJ1287

题目描述：

试设计一个网络连接某个区域中的一些地点。给定这个区域的一些地点，以及这些地

点之间可以用网线连接的线路。对每条线路，给定了连接这两个地方所需网线的长度。注意，两个地点之间的线路可能有多条。假定，给定的线路可以(直接或间接地)连接该地区中的所有地点。试为这个地区设计一个网络系统，使得该地区所有地点都可以(直接或间接地)连接，并且使用的网线长度最短。

输入描述：

输入文件包含多个测试数据。每个测试数据描述了一个需要设计的网络。每个测试数据的第 1 行为两个整数：第 1 个整数 *P* 表示给定的地点数目，第 2 个整数 *R* 表示这些地点间的路线数目。接下来的 *R* 行描述了这些线路，每条线路由 3 个整数来描述：前两个整数标明了连线的地点，第 3 个整数表示线路的长度，这 3 个整数用空格隔开。如果 *P*=0，则代表输入结束。测试数据之间有一个空行。

测试数据中，点的数目的最大值为 50，线路长度最大为 100。线路的数目不限，地点用 1～*P* 之间的整数标明，地点 *i* 和地点 *j* 之间的线路在输入数据中表示成 *i j* 或 *j i*。

输出描述：

对输入文件中的每个测试数据，输出占一行，为搭建该网络所需网线长度的最小值。

样例输入：

```
5 7
1 2 5
2 3 7
2 4 8
4 5 11
3 5 10
1 5 6
4 2 12

0
```

样例输出：

```
26
```

3.3　修建空间站(Building a Space Station), ZOJ1718, POJ2031

题目描述：

如果您是空间站工程队的一员，被分配到建设空间站的任务中，希望编写一个程序完成这个任务。

空间站由许多单元组成，这些单元被称为单间。所有的单间都是球形的，且大小不必一致。在空间站成功地进入到轨道后每个单间被固定在预定的位置。很奇怪的是，两个单间可以接触，甚至可以重叠，在极端的情形，一个单间甚至可以完全包含另一个单间。

所有的单间都必须连接，因为宇航员可以从一个单间走到另一个单间。在以下情形，宇航员可以从单间 *A* 走到单间 *B*。

(1) *A* 和 *B* 接触，或相互重叠。

(2) *A* 和 *B* 用一个“走廊”连接。

(3) 存在单间 *C*，使得可以从 *A* 走到 *C*，也可以从 *B* 走到 *C*。

如果要求安排设计空间站的结构，也就是安排哪些单间需要用“走廊”连接。在设计走廊的结构时有一定的自由性。例如，如果有 3 个单间 *A*、*B* 和 *C*，相互之间没有接触也没有重叠，至少有 3 个可能的方案来连接这 3 个单间。第 1 个方案是在 *A*-*B* 和 *A*-*C* 之间修建

走廊；第 2 个方案是在 *B*-*C* 和 *B*-*A* 之间修建走廊；第 3 个方案是在 *C*-*A* 和 *C*-*B* 之间。修建走廊的费用正比于它的长度。因此，需要选择一个方案，使得走廊的总长度最短。

走廊的宽度可以忽略。走廊修建在两个单间的表面，可以修建成任意长，但当然需要选择最短的长度。即使两个走廊 *A*-*B* 和 *C*-*D* 在空间中交叉，它们也不会在 *A* 和 *C* 之间形成一个连接。换句话说，可以假设任意两条走廊都不交叉。

输入描述：

输入文件中包含多个测试数据。每个测试数据的格式如下。

n

*x*1 *y*1 *z*1 *r*1

*x*2 *y*2 *z*2 *r*2

...

xn yn zn rn

第 1 行中的整数 *n* 为单间的数目。*n* 为整数，且不超过 100。

接下来 *n* 行描述了这些单间。每行为 4 个数，前 3 个数为该单间中心的空间坐标，第 4 个数为单间的半径；每个数精确到小数点后 3 位有效数字；这 4 个数用空格隔开；这些数都为正数，且小于 100。

输入文件最后一行为 0，表示输入结束。

输出描述：

对输入文件中的每个测试数据，输出占一行，为走廊总长度的最小值，精确到小数点后 3 位有效数字。误差不超过 0.001。

注意，如果不需要修建走廊，也就是说，不需要走廊这些单间也是连接的，在这种情形下，走廊的总长度为 0.000。

样例输入：

```
5
5.729 15.143 3.996 25.837
6.013 14.372 4.818 10.671
80.115 63.292 84.477 15.120
64.095 80.924 70.029 14.881
39.472 85.116 71.369 5.553
0
```

样例输出：

```
73.834
```

3.4 修路(Constructing Roads), POJ2421

题目描述：

有 *N* 个村庄，编号从 1 到 *N*。现需要在这 *N* 个村庄之间修路，使得任何两个村庄之间都可以连通。称 *A*、*B* 两个村庄是连通的，当且仅当 *A* 与 *B* 之间有路直接连接，或者存在村庄 *C*，使得 *A* 和 *C* 两个村庄之间有路直接连接，且 *C* 和 *B* 两个村庄是连通的。

已知某些村庄之间已经有路直接连接了，试修建一些路使得所有村庄都是连通的、且修路总长度最短。

输入描述：

测试数据的第 1 行为正整数 *N*，3≤*N*≤100，表示村庄的个数。接下来是 *N* 行，第 *i* 行有 *N* 个整数，其中第 *j* 个整数表示村庄 *i* 和村庄 *j* 之间的距离(为[1, 1000]范围内的整数)。

接下来是一个整数 Q，$0 \leqslant Q \leqslant N * (N + 1) / 2$。然后是 Q 行，每行包含了两个整数 a 和 b，$1 \leqslant a \leqslant b \leqslant N$，表示 a、b 两个村庄已经有路相通了。

输出描述：

输出一个整数，为修路的总长度，所修的路使得所有村庄都连通且总长度最短。

样例输入：

```
3
0 990 692
990 0 179
692 179 0
1
1 2
```

样例输出：

```
179
```

3.4　普里姆(Prim)算法

3.4.1　Prim 算法思想

Prim 算法的基本思想是以顶点为主导地位：从起始顶点出发，通过选择当前可用的最小权值边依次把其他顶点加入到生成树当中来。

设连通无向网为 $G(V, E)$，在普里姆算法中，将顶点集合 V 分成两个子集合 T 和 T'。

(1) T：当前生成树顶点集合。

(2) T'：不属于当前生成树的顶点集合。

很显然有：$T \cup T' = V$。

普里姆算法的具体过程如下。

(1) 从连通无向网 G 中选择一个起始顶点 u_0，首先将它加入到集合 T 中；然后选择与 u_0 关联的、具有最小权值的边(u_0, v)，将顶点 v 加入到顶点集合 T 中。

(2) 以后每一步从一个顶点(设为 u)在 T 中，而另一个顶点(设为 v)在 T'中的各条边中选择权值最小的边(u, v)，把顶点 v 加入到集合 T 中。如此继续，直到网络中的所有顶点都加入到生成树顶点集合 T 中为止。

接下来以图 3.11(a)所示的无向网为例解释普里姆算法的执行过程。该无向网的邻接矩阵如图 3.11(b)所示。利用普里姆算法构造最小生成树的过程如图 3.11(c)所示。初始时集合 T 为空，首先把起始顶点 1 加入到集合 T 中，然后按如下步骤把每个顶点加入到集合 T 中(图 3.11(c)所示的每条边旁边的序号跟下面的序号是一致的)。

(1) 集合 T 中现在只有 1 个顶点，即顶点 1，一个顶点在 T，另一个顶点在 T'的边中，权值最小的边为(1, 6)，其权值为 10，通过这条边把顶点 6 加入到集合 T 中。

(2) 集合 T 中现在有 2 个顶点了，即顶点 1、6，一个顶点在 T，另一个顶点在 T'的边中，权值最小的边为(6, 5)，其权值为 25，通过这条边把顶点 5 加入到集合 T 中。

(3) 集合 T 中现在有 3 个顶点了，即顶点 1、6、5，一个顶点在 T，另一个顶点在 T'的边中，权值最小的边为(5, 4)，其权值为 22，通过这条边把顶点 4 加入到集合 T 中。

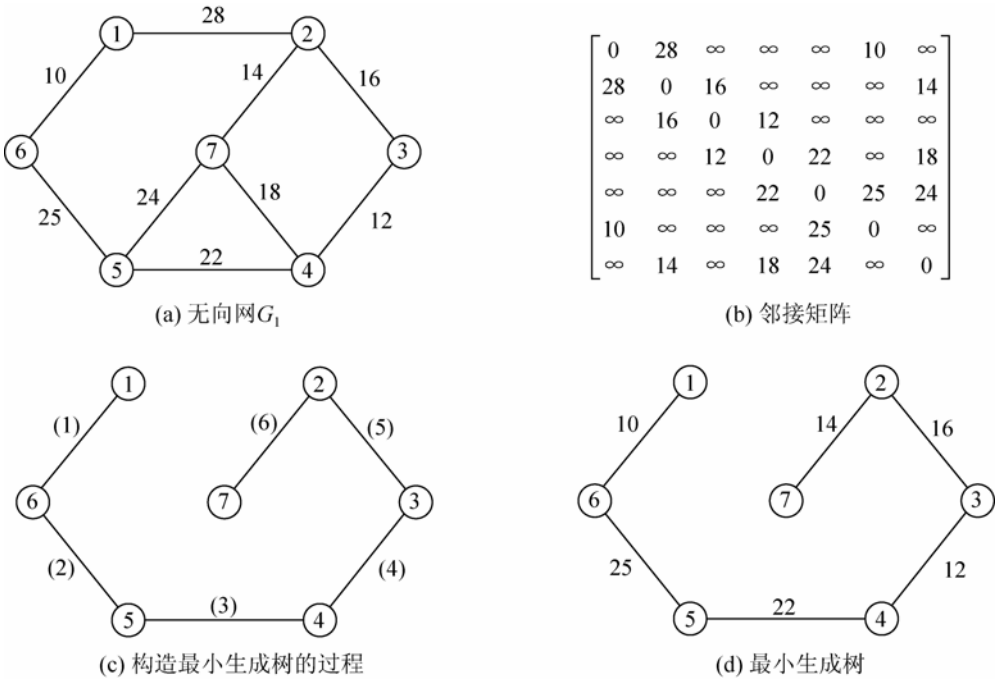
(4) 集合 T 中现在有 4 个顶点了，即顶点 1、6、5、4，一个顶点在 T，另一个顶点在

T'的边中，权值最小的边为(4, 3)，其权值为 12，通过这条边把顶点 3 加入到集合 *T* 中。

(5) 集合 *T* 中现在有 5 个顶点了，即顶点 1、6、5、4、3，一个顶点在 *T*，另一个顶点在 *T*'的边中，权值最小的边为(3, 2)，其权值为 16，通过这条边把顶点 2 加入到集合 *T* 中。

(6) 集合 *T* 中现在有 6 个顶点了，即顶点 1、6、5、4、3、2，一个顶点在 *T*，另一个顶点在 *T*'的边中，权值最小的边为(2, 7)，其权值为 14，通过这条边把顶点 7 加入到集合 *T* 中。

至此，所有顶点都已经加入到集合 *T* 中，最小生成树构造完毕，最终构造的最小生成树如图 3.11(d)所示，生成树的权值为 99。



$$\begin{bmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{bmatrix}$$

图 3.11　普里姆算法的基本思想

3.4.2　Prim 算法实现

假设采用邻接矩阵来存储图。在普里姆算法运算过程当中，需要知道以下两类信息：① 集合 *T*'内各顶点距离 *T* 内各顶点权值最小的边的权值；② 集合 *T*'内各顶点距离 *T* 内哪个顶点最近(即边的权值最小)。为了存储和表示这两类信息，必须定义两个辅助数组。

(1) lowcost[ ]: 存放顶点集合 *T*'内各顶点到顶点集合 *T* 内各顶点权值最小的边的权值。

(2) nearvex[ ]: 记录顶点集合 *T*'内各顶点距离顶点集合 *T* 内哪个顶点最近；当 nearvex[i] 为−1 时，表示顶点 i 属于集合 *T*。

注意，lowcost 数组和 nearvex 数组可以合二为一，详见例 3.4.3 节中的讨论。

以图 3.11(a)所示的无向网为例，如果选择的起始顶点为顶点 1，则这 2 个辅助数组的初始状态为：

	1	2	3	4	5	6	7
lowcost	0	28	∞	∞	∞	10	∞

	1	2	3	4	5	6	7
nearvex	−1	1	1	1	1	1	1

这是因为在生成树顶点集合 T 内最初只有一个顶点，即顶点 1，因此在 nearvex 数组中，只有表示顶点 1 的数组元素 nearvex[1]=−1，其他元素值都是 1，表示集合 T'内各顶点距离集合 T 内最近的顶点是顶点 1；而 lowcost 数组的初始值就是邻接矩阵中顶点 1 所在的行。

在 prim 算法里要重复做以下工作。

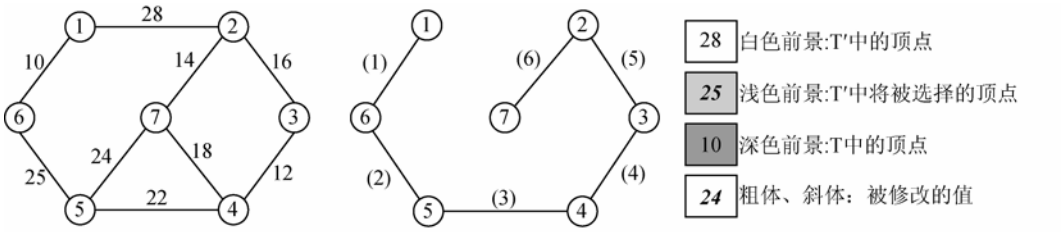
(1) 在 lowcost 数组中选择 nearvex[i] !=−1 且 lowcost[i]最小的顶点 i，用 v 标记它。则选中的权值最小的边为(nearvex[v], v)，相应的权值为 lowcost[v]。例如，在图 3.12(d)中，第一次选中的 v=6，则边(1, 6)是选中的权值最小的边，相应的权值为 lowcost[6]=10。

(2) 将 nearvex[v]改为−1，表示它已经加入到生成树顶点的集合 T，将边(nearvex[v], v, lowcost[v])输出来。

(3) 修改 lowcost[ ]：注意，原来顶点 v 不属于生成树顶点集合，现在 v 加入进来，则顶点集合 T'内的各顶点(设为顶点 i)到顶点集合 T 内的权值最小的边的权值要修改成：lowcost[i]=min{ lowcost[i], Edge[v][i] }，即把顶点 i 到新加入集合 T 的顶点 v 的距离(Edge[v][i])与原来它到集合 T 中顶点的最短距离(lowcost[i])做比较，取距离近的，作为顶点 i 到 T 内顶点的最短距离。

(4) 修改 nearvex[ ]：如果 T'内顶点 i 到顶点 v 的距离比原来它到顶点集合 T 中顶点的最短距离还要近，则修改 nearvex[i]：nearvex[i]=v；表示顶点 i 当前到 T 内顶点 v 的距离最近。

图 3.12(d)所示为在求图 3.12(a)所示的无向网的最小生成树过程中 lowcost 数组和 nearvex 数组各元素值的变化。



(a) 无向网G_1　(b) 构造最小生成树的过程　(c) 图(d)的图例

	1	2	3	4	5	6	7		1	2	3	4	5	6	7	
lowcost	0	28	∞	∞	∞	10	∞	nearvex	−1	1	1	1	1	1	1	v=6,选边(1,6)
(1)lowcost	0	28	∞	∞	***25***	10	∞	nearvex	−1	1	1	1	***6***	***−1***	1	v=5,选边(6,5)
(2)lowcost	0	28	∞	***22***	25	10	***24***	nearvex	−1	1	1	***5***	***−1***	−1	***5***	v=4,选边(5,4)
(3)lowcost	0	28	***12***	22	25	10	***18***	nearvex	−1	1	***4***	***−1***	−1	−1	***4***	v=3,选边(4,3)
(4)lowcost	0	***16***	***12***	22	25	10	18	nearvex	−1	***3***	***−1***	−1	−1	−1	4	v=2,选边(3,2)
(5)lowcost	0	16	12	22	25	10	***14***	nearvex	−1	−1	−1	−1	−1	−1	***2***	v=7,选边(2,7)
(6)lowcost	0	16	12	22	25	10	14	nearvex	−1	−1	−1	−1	−1	−1	***−1***	

(d) 普里姆算法执行过程中lowcost数组和nearvex数组各元素值的变化

图 3.12　普里姆算法的实现过程

例 3.4 利用 Prim 算法求图 3.12(a)所示的无向网的最小生成树，并输出依次选择的各条边及最终求得的最小生成树的权。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 *n* 和边数 *m*，然后输入 *m* 条边的数据。每条边的数据格式为：*u v w*，分别表示这条边的两个顶点及边上的权值。顶点序号从 1 开始计起。

分析：

下面的代码中，定义了一个函数 prim(int u0)，实现从顶点 u0 出发，执行普里姆算法求解最小生成树，而在主函数中调用函数 prim(1)，即从顶点 1 构造最小生成树。

在下面的代码中，对 prim 算法中每扩展一个顶点的所要进行的 4 个步骤分别用边框标明了，从中可以看出，prim 算法的思路是很清晰的。

代码如下：

```
#define INF 1000000          //无穷大
#define MAXN 21              //顶点个数最大值
int n, m;                    //顶点个数、边数
int Edge[MAXN][MAXN];        //邻接矩阵
int lowcost[MAXN];
int nearvex[MAXN];
void prim( int u0 )          //从顶点 u0 出发执行普里姆算法
{
    int i, j;                //循环变量
    int sumweight=0;         //生成树的权值
    for( i=1; i<=n; i++ )    //初始化 lowcost 数组和 nearvex 数组
    {
        lowcost[i]=Edge[u0][i];
        nearvex[i]=u0;
    }
    nearvex[u0]=-1;
    for( i=1; i<n; i++ )
    {
        int min=INF;                                    //(1)
        int v=-1;
        //在 lowcost 数组(的 nearvex[ ]值为-1 的元素中)中找最小值
        for( j=1; j<=n; j++ )
        {
            if( nearvex[j]!=-1 && lowcost[j]<min )
            { v=j; min=lowcost[j]; }
        }
        if( v!=-1 ) //v==-1，表示没找到权值最小的边
        {
            printf( "%d %d %d\n", nearvex[v], v, lowcost[v] );
            nearvex[v]=-1;                              //(2)
            sumweight+=lowcost[v];
            for( j=1; j<=n; j++ )                       //(3)(4)
            {
```

```
                if( nearvex[j]!=-1 && Edge[v][j]<lowcost[j] )
                {
                    lowcost[j]=Edge[v][j];
                    nearvex[j]=v;
                }
            }
        }//end of if
    }//end of for
    printf( "weight of MST is %d\n", sumweight );
}//end of prime
int main( )
{
    int i, j;                   //循环变量
    int u, v, w;                //边的起点和终点及权值
    scanf( "%d%d", &n, &m );    //读入顶点个数 n 和边数 m
    memset( Edge, 0, sizeof(Edge) );
    for( i=1; i<=m; i++ )
    {
        scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
        Edge[u][v]=Edge[v][u]=w;       //构造邻接矩阵
    }
    for( i=1; i<=n; i++ )       //对邻接矩阵中其他元素值进行赋值
    {
        for( j=1; j<=n; j++ )
        {
            if( i==j ) Edge[i][j]=0;
            else if( Edge[i][j]==0 )  Edge[i][j]=INF;
        }
    }
    prim( 1 );  //从顶点 1 出发构造最小生成树
    return 0;
}
```

该程序的运行示例如下。

输入：

```
7 9
1 2 28
1 6 10
2 3 16
2 7 14
3 4 12
4 5 22
4 7 18
5 6 25
5 7 24
```

输出：

```
1 6 10
6 5 25
5 4 22
4 3 12
3 2 16
2 7 14
weight of MST is 99
```

3.4.3 关于 Prim 算法的进一步讨论

1. 关于 lowcost 数组和 nearvex 数组的讨论

在 Prim 算法中，如果不需要输出构成生成树的边，只需要计算最小生成树的权，那么就不需要记录集合 T' 内各顶点距离 T 内哪个顶点最近，这样可以将 lowcost 数组和 nearvex 数组合二为一，从而省略了 nearvex 数组。而 lowcost 数组含义为：如果 lowcost[i]的值为−1，表示顶点 i 已经属于集合 T；否则 lowcost[i]的值表示集合 T' 内顶点 i 距离 T 内各顶点权值最小的边的权值。这种思想的应用详见例 3.5。

2. Prim 算法的时间复杂度分析

Prim 算法的时间复杂度为 $O(n^2)$，n 为图中顶点数目，证明略。Prim 算法的时间复杂度只与图中顶点的个数有关，与边的数目无关，因此该算法适合于稠密图。

3. Prim 算法、Kruskal 算法和 Boruvka 算法的对比分析

本章所介绍的 3 个 MST 算法都是从由单个顶点构成的子树(子树中没有边)所组成的森林作为 MST 的开始，并陆续选择一条连接森林中两棵子树的最小权值边将这两棵子树合并，并最终构造一棵完整的 MST 树。这 3 个算法的区别在于：Prim 算法选择的是连接当前 MST 和剩下的单个顶点之间的最小权值边，从而将该顶点合并到 MST 中；而 Kruskal 算法和 Boruvka 算法选择的是连接任意两棵子树的最小权值边，从而将这两棵子树合并。

表 3-1 对这 3 个 MST 算法的时间复杂度及特点作了对比分析。

表 3-1 MST 算法对比分析

算法	最坏情况下时间复杂度	特点
Prim 算法	$O(n^2)$	适合于稠密图
Kruskal 算法	$O(\log_2^m + 2m\log_2^n + n)$	时间复杂度主要取决于边的数目，适合于稀疏图
Boruvka 算法	$O(m\log_2^n)$	算法思想与 Kruskal 算法思想类似

3.4.4 例题解析

以下通过两道例题的分析，再详细介绍普里姆算法的基本思想及其实现方法。

例 3.5 QS 网络(QS Network)

题目来源：

Zhejiang University Local Contest 2003, Preliminary, ZOJ1586

题目描述：

在 cgb 星系的 w-503 星球上，有一种智能生物，名为 QS。QS 之间通过网络进行通信。如果两个 QS 需要通信，他们需要买 2 个网络适配器，每个 QS 一个，以及一段网线。注意每个网络适配器只用于单一的通信中，也就是说如果一个 QS 想建立 4 个通信，那么需要买 4 个适配器。在通信过程中，一个 QS 向所有和他连接的 QS 发送消息，接收到消息的 QS 又向所有和他连接的 QS 发送消息，通信一直到所有的 QS 接收到消息为止。

图 3.13 所示是 QS 网络的一个例子。

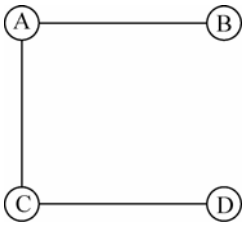


图 3.13　QS 网络的一个例子

每个 QS 有他自己喜欢的网络适配器品牌，在他建立的所有连接中总是使用他自己喜欢的适配器。QS 之间的距离是不同的。给定每个 QS 喜欢的适配器的价格，以及每对 QS 之间网线的价格，试编写一个程序计算建立 QS 网络的最小费用。

输入描述：

输入文件包含多个测试数据。输入文件的第一行为 1 个整数 *t*，表示输入文件中测试数据的个数。从第 2 行开始有 *t* 个测试数据。

每个测试数据的第 1 行为一个整数 *n*，表示 QS 的个数，第 2 行为 *n* 个整数，表示每个 QS 喜欢的适配器的价格，第 3 行到第 *n*+2 行为一个矩阵，表示每对 QS 之间网线的价格。

输入文件中的所有整数都是不超过 1 000 的非负整数。

输出描述：

对输入文件中的每个测试数据，输出占一行，建立 QS 网络的最小费用。

样例输入：

```
1
3
10 20 30
0 100 200
100 0 300
200 300 0
```

样例输出：

```
370
```

分析：

本题需要计算建立 QS 网络的最小费用，这是最小生成树的问题。在构造有向网时，每条边的权值为两个 QS 的适配器价格加上这两个 QS 之间网线的价格。并且本题只需计算最小生成树的权值，不需要记录构造最小生成树时选择的边，因此可以将 lowcost 数组和 nearvex 数组合二为一。

代码如下：

```
#define MAX 1000000
int Edge[1010][1010];   //邻接矩阵
int adapter[1010];      //每个 QS 喜欢的适配器价格
int lowcost[1010];  //充当 prim 算法中的两个数组(lowcost 数组和 nearvex 数组)的作用
int t;                  //测试数据的个数
int n;                  //每个测试数据中 QS 的个数
void init( )
{
    int i, k;                           //循环变量
    scanf( "%d", &n );                  //读入 QS 的个数
```

```
    for( i=0; i<n; i++ )
        scanf ( "%d", &adapter[i] );    //读入每个 QS 喜欢的适配器价格
    for( i=0; i<n; i++ )                //利用输入数据构造邻接矩阵
    {
        for( k=0; k<n; k++ )
        {
            scanf( "%d", &Edge[i][k] );
            if( i==k )  Edge[i][k]=MAX;
            else  Edge[i][k]+=adapter[i]+adapter[k];
        }
    }
    memset( lowcost, 0, sizeof ( lowcost ) );
}
void prim( )
{
    int i, k;           //循环变量
    int sum=0;          //sum 为最终求得的最小生成树的权值
    lowcost[0]=-1;      //从顶点 0 开始构造最小生成树
    for( i=1; i<n; i++ )
        lowcost[i]=Edge[0][i];
    for( i=1; i<n; i++ )        //把其他 n-1 个顶点扩展到生成树当中
    {
        int min=MAX, j;
        for( k=0; k<n; k++ )    //找到当前可用的权值最小的边
        {
            if( lowcost[k] !=-1  &&  lowcost[k]<min )
            {
                j=k;
                min=lowcost[k];
            }
        }
        sum+=min;
        lowcost[j]=-1;  //把顶点 j 扩展进来
        for( k=0; k<n; k++ )
        {
            if( Edge[j][k]<lowcost[k] )
                lowcost[k]=Edge[j][k];
        }
    }
    printf( "%d\n", sum );
}
int main( )
{
    scanf ( "%d", &t );     //测试数据的个数
    for ( int i=0; i<t; i++ )
    {
        init( );
```

```
            prim( );
        }
        return 0;
    }
```

例 3.6　卡车的历史(Truck History)

题目来源：

Czech Technical University Open 2003, ZOJ2158, POJ1789

题目描述：

高级货物运输公司 ACM 使用不同类型的卡车。有些卡车用来运蔬菜，有些用来运水果，还有一些用来运砖等。该公司对不同的卡车有自己的编码方法。卡车的编码为一个包含 7 个字符的字符串(每个位置上的字符都有特定的含义，但这一点对本题并不重要)。在 ACM 公司发展历史上的初期，只有一种卡车可供使用，只有一种卡车类型编码；后来又引进了新的一种卡车类型，新卡车的类型编码是从第 1 种卡车编码派生出来的；然后新的卡车类型编码又派生出其他卡车类型编码等。

今天，ACM 公司是如此的富有，可以让历史学家来研究公司的历史。历史学家试图弄明白的一件事情被称为派生方案，也就是，卡车的类型是如何派生的。他们将卡车类型编码的距离定义成卡车类型编码字符串中(7 个位置上)不同字符的位置数目。比如，一个卡车编码是 aaaaaaa，另一个卡车编码是 babaaaa，那么它们的距离值就是 2。他们假定每种卡车类型都是由其他一种卡车类型派生出来的，当然，第 1 种卡车类型除外，它不是由任何其他一种类型派生的。派生方案的优劣值定义成：

$$1/\sum_{(t_o,\ t_d)} d(t_o,\ t_d)$$

式中，求和部分为派生方案中所有类型对(t_o, t_d)的距离；t_o 为基类型；t_d 为派生出来的类型。

因为 ACM 公司卡车类型很多，历史学家很难判断这些类型编码之间派生关系。本题的目的是要求编写程序，实现：给定卡车类型的编码，求具有最高优劣值的派生方案。

输入描述：

输入文件中包含多个测试数据。每个测试数据第 1 行为一个整数 N，表示卡车类型的数目，$2 \leqslant N \leqslant 2\,000$。接下来有 N 行，每行为一种卡车的编码(由 7 个小写字母组成的字符串)。假定卡车的编码唯一地描述了这种卡车，也就是说，这 N 行字符串中任意两个都不相同。输入文件最后一行为 0，表示输入结束。

输出描述：

对输入文件中的每个测试数据，输出一行字符串："The highest possible quality is 1/Q."，其中 $1/Q$ 为最佳派生方案的优劣值。

样例输入：

```
4
aaaaaaa
baaaaaa
abaaaaa
aabaaaa
0
```

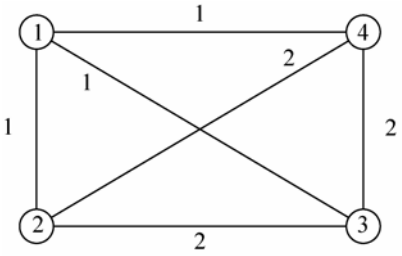
样例输出：

```
The highest possible quality is 1/3.
```

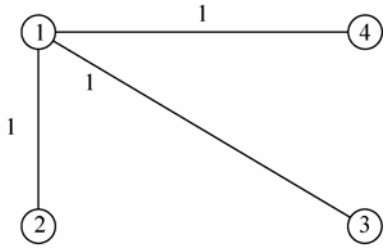
分析：

要使派生方案的优劣值：$1/\sum_{(t_o, t_d)} d(t_o, t_d)$ 最大，分母 $\sum_{(t_o, t_d)} d(t_o, t_d)$ 的值肯定取到最小。另外，要求考虑所有类型对(t_o, t_d)的距离，使得最终派生方案中每种卡车类型都是由其他一种卡车类型派生出来的(最初的卡车类型除外)。这样，如果将每种卡车类型理解成一个无向网中的顶点，所要求的最佳派生方案就是求最小生成树，而 $\sum_{(t_o, t_d)} d(t_o, t_d)$ 就是最小生成树的权值。

例如，样例输入中的测试数据所对应的无向网如图 3.14(a)所示，用 4 个顶点表示输入中 4 种卡车类型编码。每个顶点之间边的权值为对应两种卡车编码之间的距离，即卡车类型编码字符串中不同字符的位置数目。这样，求得的最小生成树如图 3.14(b)所示，权值为 3，因此最佳派生方案的优劣值为 1/3。



(a) 样例输入中测试数据对应的无向网



(b) 构造的最小生成树

图 3.14　卡车的历史

代码如下：

```
#define INF 1000000
#define MAXN 2000   //卡车类型数目的最大值
#define CODELEN 7   //编码长度
int N;                          //卡车类型数目
char codes[MAXN][CODELEN+3];    //存储每种卡车类型编码
int d[MAXN][MAXN];              //每对卡车类型之间的距离(邻接矩阵)
int lowcost[MAXN];  //充当prim算法中的两个数组(lowcost数组和nearvex数组)的作用
int min_tree( )
{
    int i, j, k;    //循环变量
    int dist;       //两个类型编码之间的距离
    memset( d, 0, sizeof(d) );
    for( i=0; i<N; i++ )    //求第i种类型与第j种类型编码之间的距离
    {
        for( j=i+1; j<N; j++ )
        {
            dist=0;
            for( k=0; k<7; k++ )
                dist+=codes[i][k] !=codes[j][k];
            d[i][j]=d[j][i]=dist;
        }
```

```
        }
        int sum=0;      //sum 为最终求得的最小生成树的权值
        lowcost[0]=-1;  //从顶点 0 开始构造最小生成树
        for( i=1; i<N; i++ )
            lowcost[i]=d[0][i];
        for( i=1; i<N; i++ )    //把其他 N-1 个顶点扩展到生成树当中
        {
            int min=INF;
            for( k=0; k<N; k++ )    //找到当前可用的权值最小的边
            {
                if( lowcost[k]!=-1  &&  lowcost[k]<min )
                {
                    j=k;
                    min=lowcost[k];
                }
            }
            sum+=min;
            lowcost[j]=-1;  //把顶点 j 扩展进来
            for( k=0; k<N; k++ )
            {
                if( d[j][k]<lowcost[k] )
                    lowcost[k]=d[j][k];
            }
        }
        return sum;
    }
    int main( )
    {
        int i;  //循环变量
        while( 1 )
        {
            scanf( "%d", &N );
            if( N==0 )  break;
            for( i=0; i<N; i++ )
                scanf( "%s", codes[i] );
            printf( "The highest possible quality is 1/%d.\n", min_tree( ) );
        }
        return 0;
    }
```

练　　习

3.5　北极的无线网络(Arctic Network), ZOJ1914, POJ2349

题目描述：

国防部想在北部的前哨之间建立一个无线网络连接这些前哨。在建立网络时使用了两种不同的通信技术：每个前哨有一个无线电收发器，有一些前哨还有一个卫星频道。

任何两个拥有卫星频道的前哨之间可以直接通过卫星进行通信，而且卫星通信跟距离和位置无关。否则，两个前哨之间通过无线电收发器进行通信，并且这两个前哨之间的距离不能超过 *D*，这个 *D* 值取决于无线电收发器的功率。功率越大，*D* 值也就越大，但是价格也就越高。出于对购买费用和维护费用的考虑，所有使用无线电接收器进行通信的前哨的收发器必须相同，即 *D* 值相同。

试计算无线电收发器 *D* 值的最小值，且保证每两个前哨之间至少有一条通信线路(直接或间接地连接这两个前哨)。

输入描述：

输入文件的第 1 行为一个整数 *N*，表示测试数据的数目。每个测试数据的第 1 行为两个整数：*S* 和 *P*，1≤*S*≤100，*S*<*P*≤500；*S* 表示卫星频道的个数，*P* 表示前哨的个数。

接下来有 *P* 行，给出了每个前哨的位置(*x*, *y*)，单位为 km，*x* 坐标和 *y* 坐标为 0～10 000 之间的整数。

输出描述：

对输入文件中的每个测试数据，输出占一行，为求得的 *D* 值最小值，精确到小数点后两位有效数字。

样例输入：

```
1
2 4
0 100
0 300
0 600
150 750
```

样例输出：

```
212.13
```

3.6　高速公路系统(Highways), ZOJ2048, POJ1751

题目描述：

岛国 Flatopia 的地面十分平坦。不幸的是，Flatopia 的高速公路系统十分差。Flatopian 政府已经意识到这个问题，已经在一些比较重要的城市之间修建起高速公路。然而，仍然有一些小城市无法通过高速公路到达。有必要再修建一些高速公路使得任意两个城市之间都可以通过高速公路连通。

Flatopian 国家的城镇编号为 1 至 *N*，第 *i* 个城镇的位置由笛卡儿直角坐标(x_i, y_i)给定。每一段高速公路只连接两个城镇。所有的高速公路(包括原来有的高速公路和需要再修建的高速公路)均为直线，因此它们的长度为所连接的两个城市之间的距离。所有的高速公路均为双向。高速公路可以自由交叉，但是司机只能在有公共城镇的两条高速公路之间进行切换。

Flatopian 政府希望修建新的高速公路的费用最小，并且要保证任何两个城镇之间都能通过高速路连通。由于 Flatopian 几国家的地面是如此的平坦，因此，修建高速公路的费用取决于高速路的长度。因此，费用最少的高速公路系统就是长度总和最短的高速公路系统。

输入描述：

输入文件包含多个测试数据。输入文件第 1 行为一个整数 *T*，然后是一个空行。接下来有 *T* 个输入块，每两个输入块之间有一个空行。每个输入块为一个测试数据，它包含两部分。第 1 部分描述了该城市所有的城镇，第 2 部分描述了已经修建好的高速公路。

每个测试数据的第 1 行为一个整数 N，$1 \leqslant N \leqslant 750$，表示城镇的数目，这 N 个城镇的编号为 1～N。接下来有 N 行，每行包含两个整数：x_i 和 y_i，用空格隔开，为第 i 个城镇的坐标。坐标的绝对值不超过 10 000。每个城镇的位置都是唯一的。

接下来一行为一个整数 M，$0 \leqslant M \leqslant 1\,000$，表示已经修建好的高速公路数目。接下来 M 行，每行包含两个整数，用空格隔开，表示每条公路所连接的两个城镇。每对城镇之间最多由一条高速公路连接。

输出描述：

对输入文件中的每个测试数据，输出为连接所有城镇且总长度最短的高速公路系统中新修建的每条高速公路，每条高速公路用它所连接的两个城镇序号表示，用一个空格隔开。

如果不需要再修建新的高速公路(所有的城镇已经连接了)，则只输出一个空行。

每两个测试数据的输出之间有一个空行。

样例输入：

```
1

9
1 5
0 0
3 2
4 5
5 1
0 4
5 2
1 2
5 3
3
1 3
9 7
1 2
```

样例输出：

```
1 6
3 7
4 9
5 7
8 3
```

3.7　农场网络(Agri-Net), POJ1258

题目描述：

农场主 John 被选为他所在城市的市长。他的竞选宣传之一是承诺给所有的农场连接到 Internet。当然了，他需要帮忙。

John 给他的农场申请了高速网络连接，他希望将他的网络连接共享给其他农场主。为了降低费用，他希望用最短的光纤将他的农场和其他农场连接。

给定连接每两个农场所需的光纤长度，试计算将所有农场连接起来所需光纤的最短长度。每个农场必须与其他农场连接起来，这样数据包可以从一个农场流向另一个农场。

任何两个农场之间的距离不超过 100 000。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为一个整数 N，$3 \leqslant N \leqslant 100$，表

示农场的数目。接下来的若干行描述了 $N \times N$ 的连接矩阵，其中每个元素表示两个农场之间的距离。在逻辑上，它们有 N 行，每行有 N 个整数，用空格隔开。在物理上，每行的长度不超过 80 个字符，因此有些行(如果显示不下)要延伸到其他行。当然，对角线元素为 0，因为第 i 个农场到它本身的距离对本题没有意义。

输出描述：

对输入文件中的每个测试数据，输出一个整数，代表连接所有农场所需光纤总长度的最小值。

样例输入：

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
```

样例输出：

```
28
```

3.8 Borg 迷宫(Borg Maze), POJ3026

题目描述：

Borg 是银河系三角洲中无比强大的类人生物。Borg 集体是用来描述的 Borg 文明群体意识的术语。每个 Borg 个体与集体通过复杂的子空间网络连接，以确保每个个体得到持续稳定的监督和指导。

试帮助 Borg 编写程序估计扫描整个迷宫并同化隐藏在迷宫中的相异个体的最小代价，扫描迷宫时可以向北、西、南、东移动。棘手的是搜索是由超过 100 个个体组成的群体进行的。当一个相异个体被同化时(或是在搜索的起点)，群体可以分裂成两个或多个子群体(但是这些子群体的意识仍然是聚积成一整体的)。搜索迷宫的代价被定义成参与搜索的所有子群体走过的步数总和。例如，如果原群体走了 5 步，然后分裂成两个子群体，每个子群体又分别走了 3 步，则总步数为：11=5+3+3。

输入描述：

输入文件的第 1 行为一个整数 N，$0<N\leqslant 50$，表示输入文件中测试数据的个数。每个测试数据的第 1 行为两个整数 x 和 y，$1\leqslant x, y\leqslant 50$；接下来是 y 行，每行为 x 个字符，其中：空格字符代表一个空的空间、'#'字符代表墙壁障碍物、'A'字符代表相异个体、'S'字符代表搜索的起点。迷宫的边界是封闭的，也就是说，从'S'处无法走出迷宫边界。迷宫中最多有 100 个相异个体，每个个体都是可到达的。

输出描述：

对输入文件中的每个测试数据，输出一行，为遍历完整个迷宫、同化所有相异个体所需的最小代价。

样例输入：

```
2
7 7
#####
#AAA###
#    A#
```

样例输出：

```
11
```

```
# S ###
#     #
#AAA###
#####
```

3.5　判定最小生成树是否唯一

3.5.1　最小生成树不唯一的原因分析

对于一个给定的连通无向网，它的最小生成树是唯一的吗？

图 3.15 所示的连通无向网，在构造最小生成树时，可以选择(1, 2)、(2, 3)、(3, 4)这 3 条边(如图 3.15(a)所示)，最小生成树的权为 6；也可以选择(2, 1)、(1, 4)、(4, 3)这 3 条边(如图 3.15(b)所示)，最小生成树的权也为 6 等。

(a) 构造最小生成树的方案1　　(b) 构造最小生成树的方案2

图 3.15　最小生成树不唯一的一个简单例子

通过上面的例子可以看出，在构造最小生成树时有可能可以选择不同的边，这样构造出来的最小生成树不相同，但最小生成树的权是唯一的！

毫无疑问，无向网中存在相同权值的边是 MST 不唯一的必要条件(但不是充分条件)。这是因为，如果无向网中各边的权值都不相同，则在用 Kruskal 算法构造 MST 时，选择边的方案是唯一的，因此构造的 MST 必定是唯一的；只有存在相同权值的边，才有可能存在不同的方案构造 MST。当然，如果相同权值的边不会被任何一个 MST 选入，则 MST 也必定唯一。

如果无向网中存在相同权值的边，还要分成以下情形考虑。以下讨论的情形都是针对 MST 中包含了相同权值的边。

1. 相同权值的边有公共顶点

如图 3.16 所示的两个无向网均存在相同权值的边，且这些边有公共顶点。图 3.16(a)所示的 MST 是唯一的，在图中用粗线标明了 MST 中的边。而图 3.16(b)所示的 MST 不是唯一的，粗线标明的边构成一个 MST，如果将边(4, 6)换成(1, 6)，同样构成了一个 MST，且 MST 的权值是相同的。

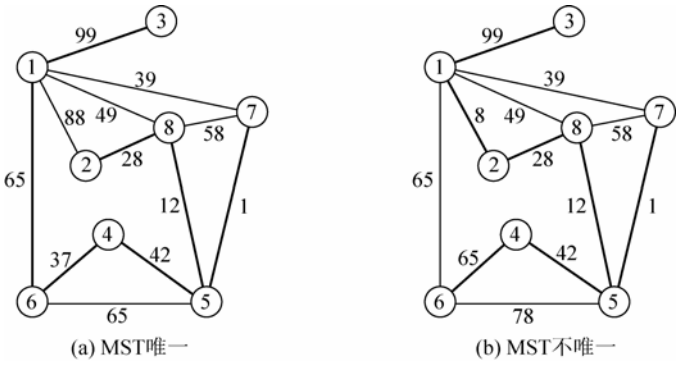


图 3.16　无向网中存在相同权值的边，且这些边有公共顶点

2. 相同权值的边没有公共顶点

图 3.17 所示的两个无向网均存在相同权值的边，且均没有公共顶点。图 3.17(a)所示的 MST 是唯一的，在图中用粗线标明了 MST 中的边。而图 3.17(b)所示的 MST 不是唯一的，粗线标明的边构成一个 MST，如果将边(4, 5)换成(1, 6)，同样构成了一个 MST，且 MST 的权值是相同的。

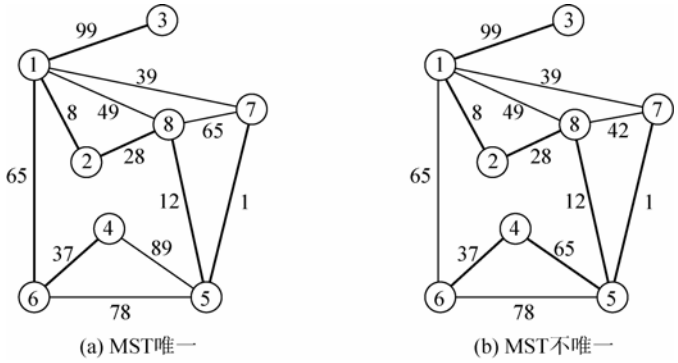


图 3.17　无向网中存在相同权值的边，且这些边没有公共顶点

3. 混合情形

即无向网中存在相同权值的边，且某些边有公共顶点，其他边没有公共顶点。对于这种情形，可以拆分成前面两种情形，在此不再赘述。

3.5.2　判定最小生成树是否唯一的方法

1. 用 Prim 算法判断最小生成树不唯一的思路

如果在用 Prim 算法求 MST 过程中，对某个顶点的扩展方式不唯一，则可判断 MST 不唯一。本节以图 3.18(a)所示的连通无向网为例介绍这种思路。该图所示的无向网是在图 3.10(a)所示的无向网中增加了一条边(2, 4)，权值为 16。

如图 3.18(b)和图 3.18(d)所示，应用普里姆算法求解最小生成树时，从顶点 1 出发，依次把顶点 6、5、4、3 扩展到生成树顶点集合 *T* 后，此时 lowcost 数组和 nearvex 数组各元素取值如图 3.18(d)中虚线框所示。

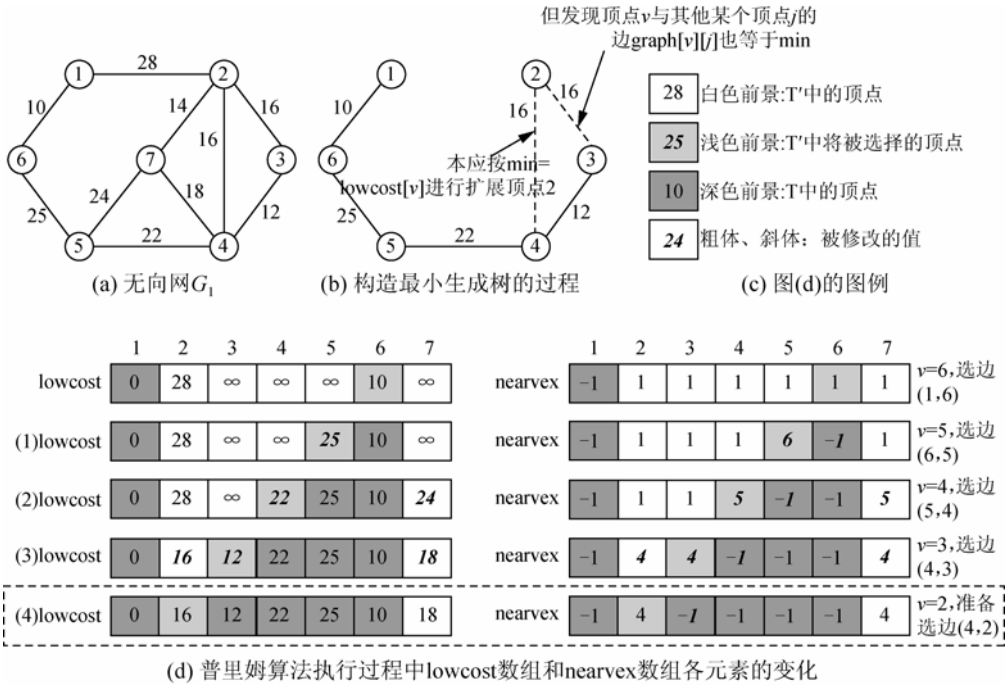


图 3.18　判定最小生成树不唯一的思路

此后，因为在 lowcost 数组中，顶点 2 的 lowcost[ ]值最小为 16，本来应该通过边(nearvex[2], 2)，即边(4, 2)将顶点 2 扩展到集合 *T* 中，但这时发现顶点 2 与顶点 3 之间的边的长度也为 16，也可以通过边(3, 2)将顶点 2 扩展进来，即顶点 2 的扩展方式不唯一，从而判定最小生成树不唯一。

因此利用这种思路判定最小生成树不唯一的方法如下。

首先当前已经找到 min=lowcost[*v*]，因此，待扩展的顶点是 *v*，待扩展的边是(nearvex[*v*], *v*)。

接下来要判断在集合 *T* 内是否还有其他顶点到 *v* 的距离也等于 lowcost[*v*]，根据以下 4 个条件来判定。

(1) nearvex[*j*]== −1：只考虑集合 *T* 内的顶点。

(2) graph[*v*][*j*] < MAX：要排除 min 刚好等于∞的情况，这个条件实际上可以去掉。

(3) graph[*v*][*j*]== min：顶点 *j* 到 *v* 的距离刚好也等于 min。

(4) nearvex[*v*] != j：而且 *j* 不等于 nearvex[*v*]。

如果以上 4 个条件同时成立，即可判定扩展顶点 *v* 的方式不唯一，从而判定最小生成树不唯一。

说明：以上思路在判断出当前顶点扩展方式不唯一时能判定 MST 不唯一，但即使每个顶点的扩展方式均唯一也不能判定 MST 唯一。例如，图 3.19 所示的无向网，在用 Prim 算法求 MST 时，每个顶点的扩展方式均唯一，但该图的 MST 不唯一，因为如果将图 3.19(a)所示的 MST 中边(5, 4)换成(5, 6)，将得到另一个 MST(如图 3.19(b)所示)。

2. 判定最小生成树是否唯一的思路

判定最小生成树是否唯一的一个正确思路如下。

(1) 对图中每条边，扫描其他边，如果存在相同权值的边，则对该边作标记。

(2) 然后用 Kruskal 算法(或 Prim 算法)求 MST。

(3) 求得 MST 后，如果该 MST 中未包含作了标记的边，即可判定 MST 唯一；如果包含作了标记的边，则依次去掉这些边再求 MST，如果求得的 MST 权值和原 MST 的权值相同，即可判定 MST 不唯一。

例如，对图 3.19 所示的无向网，先求出图 3.19(a)中粗线标明的 MST，然后去掉边(4,5)，可以求出图 3.19(b)中所示粗线标明的 MST，且权值总和与原 MST 一样，即可判定此无向网的 MST 不唯一。

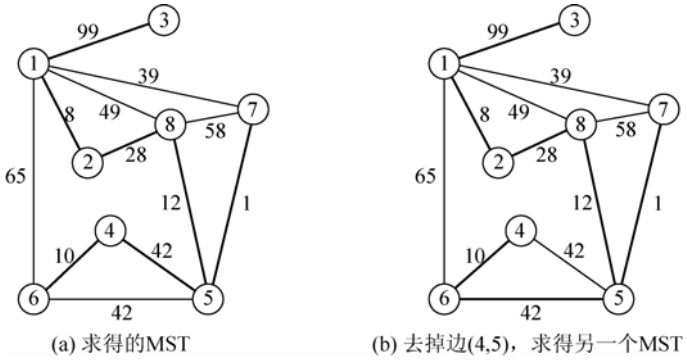


图 3.19　每个顶点的扩展方式唯一，但 MST 不唯一的情形

3.5.3　例题解析

例 3.7　判定最小生成树是否唯一(The Unique MST)

题目来源：

POJ Monthly-2004.06.27, POJ1679

题目描述：

给定一个连通无向网，判定它的最小生成树是否唯一。

输入描述：

输入文件的第 1 行为一个整数 t，$1 \leqslant t \leqslant 20$，表示测试数据的数目。每个测试数据描述了一个连通无向网。每个测试数据的第 1 行为两个整数：n 和 m，$1 \leqslant n \leqslant 100$，分别表示顶点的数目和边的数目。接下来有 m 行，每行为一个三元组(x_i, y_i, w_i)，表示一条边(x_i, y_i)，x_i 和 y_i 表示边的两个顶点，顶点序号从 1 开始计起，这条边的权值为 w_i。任何两个顶点间最多只有一条边。

输出描述：

对输入文件中的每个测试数据，如果最小生成树是唯一的，则输出最小生成树的权；如果最小生成树不唯一，则输出"Not Unique!"。

样例输入：

```
2
7 10
1 2 28
1 4 22
1 6 10
```

样例输出：

```
96
Not Unique!
```

```
2 3 16
2 7 14
3 4 12
4 5 22
4 7 18
5 6 25
5 7 24
7 12
1 2 6
1 6 6
1 7 6
2 3 2
2 7 3
3 4 3
3 7 2
4 5 1
4 7 4
5 6 8
5 7 3
6 7 7
```

分析：

在本题中，为了方便按照 3.5.2 节中介绍的第 2 种方法判定 MST 是否唯一，在表示边结构的结构体中增加了 3 个成员，其含义分别如下。

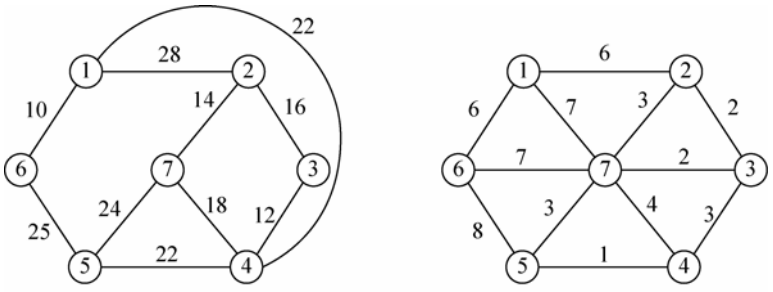
equal：标记是否存在其他边的权值与该边权值一样，1 表示存在，0 表示不存在。

used：第 1 次构造 MST 时选用的边。

del：在判定 MST 是否唯一时，依次“去掉”的边。

本题判定 MST 是否唯一的思路详见 3.5.2 节。本题采用 Kruskal 算法求 MST，在算法中判断如果是第 1 次构造 MST，则标记所采用的边的 used 成员为 1。在算法中还会跳过被去掉的边。

样例输入中两个测试数据所描述的无向网如图 3.20 所示，其中图 3.20(a)所示的 MST 是唯一的，其权值为 96，图 3.20(b)所示的 MST 不唯一。



(a) 无向网G_1的MST唯一　　(b) 无向网G_2的MST不唯一

图 3.20　判定最小生成树是否唯一

代码如下：

```
#define MAXN 101    //顶点个数的最大值
#define MAXM 10000  //边的个数的最大值
struct edge //边
{
    int u, v, w;    //边的顶点、权值
    int equal;      //标记，1 表示存在其他边权值跟该边一样，0 表示不存在
    int used;       //在第 1 次求得的 MST 中，是否包含该边，1 表示包含，0 表示不包含
    int del;        //边是否删除的标记，0 表示不删除，1 表示删除
}edges[MAXM];       //边的数组
int parent[MAXN];   //parent[i]为顶点 i 所在集合对应的树中的根结点
int n, m;           //顶点个数、边的个数
bool first;         //第 1 次求 MST 的标志变量
void UFset( )       //初始化
{
    for( int i=0; i<n; i++ )  parent[i]=-1;
}
int Find( int x )   //查找并返回节点 x 所属集合的根结点
{
    int s;          //查找位置
    for( s=x; parent[s]>=0; s=parent[s] )  ;
    while( s!=x )   //优化方案——压缩路径，使后续的查找操作加速
    {
        int tmp=parent[x];  parent[x]=s;  x=tmp;
    }
    return s;
}
//将两个不同集合的元素进行合并，使两个集合中任意两个元素都连通
void Union( int R1, int R2 )
{
    int r1=Find(R1),  r2=Find(R2);  //r1 为 R1 的根结点，r2 为 R2 的根结点
    int tmp=parent[r1]+parent[r2]; //两个集合结点个数之和(负数)
    //如果 R2 所在树结点个数>R1 所在树结点个数(注意 parent[r1]是负数)
    if( parent[r1]>parent[r2] )    //优化方案——加权法则
    {
        parent[r1]=r2;  parent[r2]=tmp;
    }
    else
    {
        parent[r2]=r1;  parent[r1]=tmp;
    }
}
int cmp( const void *a, const void *b )    //实现从小到大进行排序的比较函数
{
    edge aa=*(const edge *)a;
    edge bb=*(const edge *)b;
```

```
    return aa.w-bb.w;
}
int Kruskal( )
{
    int sumweight=0, num=0;     //生成树的权值、已选用边的数目
    int u, v;                   //选用边的两个顶点
    UFset( );                   //初始化 parent 数组
    for( int i=0; i<m; i++ )
    {
        if( edges[i].del==1 ) continue;    //忽略去掉的边
        u=edges[i].u;  v=edges[i].v;
        if( Find(u) != Find(v) )
        {
            sumweight+=edges[i].w;  num++;
            Union( u, v );
            if( first )  edges[i].used=1;
        }
        if( num>=n-1 )  break;
    }
    return sumweight;
}
int main( )
{
    int u, v, w;        //边的起点和终点及权值
    int t, i, j, k;     //测试数据个数、循环变量
    scanf( "%d", &t );  //读入测试数据数目
    for( i=1; i<=t; i++ )
    {
        scanf( "%d%d", &n, &m );    //读入顶点个数 n 和边的数目 m
        for( j=0; j<m; j++ )
        {
            scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
            edges[j].u=u-1;  edges[j].v=v-1;  edges[j].w=w;
            edges[j].equal=0;  edges[j].del=0;  edges[j].used=0;
        }
        for( j=0; j<m; j++ )    //标记相同权值的边
        {
            for( k=0; k<m; k++ )
            {
                if( k==j ) continue;
                if( edges[k].w==edges[j].w )  edges[j].equal=1;
            }
        }
//对边按权值从小到大的顺序进行排序
        qsort( edges, m, sizeof(edges[0]), cmp );
        first=true;
        int weight1=Kruskal( ), weight2;   //第 1 次求 MST
```

```
            first=false;
            for( j=0; j<m; j++ )
            {
    //依次去掉原 MST 中相同权值的边
                if( edges[j].used && edges[j].equal )
                {
                    edges[j].del=1;
                    weight2=Kruskal( );
                    if( weight2==weight1 )
                    {
                        printf( "Not Unique!\n" );  break;
                    }
                    edges[j].del=0;
                }
            }
            if( j>=m )  printf( "%d\n", weight1 );
        }
        return 0;
    }
```