第 8 章　图的连通性问题

连通性是图论中一个重要概念，第 1 章已经初步介绍了图的连通性，本章进一步讨论无向非连通图的连通分量，无向连通图的割顶集、顶点连通度、割点和点双连通分量，无向连通图的割边集、边连通度、割边和边双连通分量，以及有向图的强连通分量等概念和求解算法及应用。

8.1　基 本 概 念

本节将集中介绍一些基本概念，然后在 8.2～8.4 节分别介绍相应的求解方法及应用。

8.1.1　连通图与非连通图

如果无向图 G 中任意一对顶点都是连通的，则称此图是**连通图**(Connected Graph)；相反，如果一个无向图不是连通图，则称为**非连通图**(Disconnected Graph)。对非连通图 G，其极大连通子图称为**连通分量**(Connected Component，**连通分支**)，连通分支数记为 $w(G)$。

当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在的极大连通子图(即连通分量)中的所有顶点。若从无向图的每一个连通分量中的一个顶点出发进行遍历，就可以访问到所有顶点。

例如，图 8.1(a)所示的非连通无向图包含两个连通分量：顶点 A、B、C 和 E 组成的连通分量，顶点 D、F、G 和 H 组成的连通分量。对该图进行 DFS 遍历时，从顶点 A 出发可以遍历到第 1 个连通分量上的各个顶点，从顶点 D 出发可以遍历到第 2 个连通分量上的各个顶点，其 DFS 遍历过程如图 8.1(b)所示。

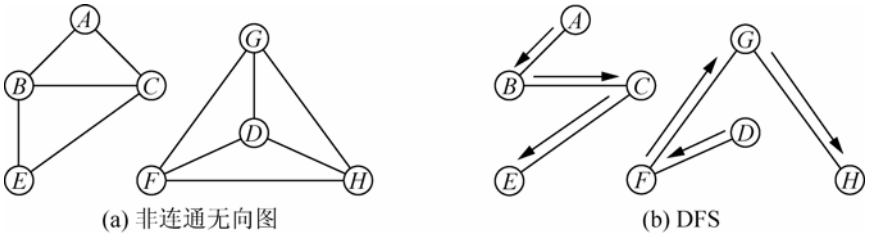


图 8.1　非连通无向图的 DFS 遍历

在用程序实现中，需要对无向图的每一个顶点进行检测：若已被访问过，则该顶点一定是落在图中已求得的连通分量上；若还未被访问，则从该顶点出发遍历，可求得图的另一个连通分量。

假设用邻接矩阵存储图(设顶点个数为 n)，下面的伪代码从每个未访问过的顶点出发进

行 DFS 搜索，可以遍历到所有顶点，并可以求得连通分量的个数。

```
subnets=0;          //表示连通分量个数的变量
for( k=0; k<n; k++ )
{
    if( !visited[k] )   //顶点 k 未访问过
    {
        DFS( k );       //从顶点 k 出发进行深度优先搜索
        subnets++;
    }
}
```

其中 DFS 函数的伪代码见 2.1.2 节。

观察图 8.2 所示的一些无向图。尽管这些无向图都是连通图，但其中某些图看起来比其他图“更为连通”，而某些图的连通性是如此“脆弱”，以至于移去某个顶点或某条边就导致图不连通。这意味着有必要引入一些能反映无向连通图连通程度的量，即顶点连通度与边连通度。

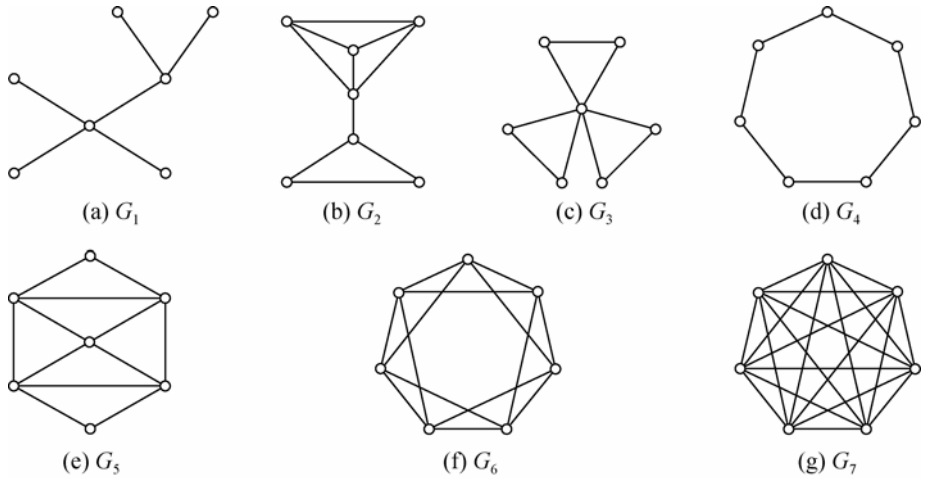


图 8.2　一些无向连通图

8.1.2　无向图的点连通性

所谓**点连通性**(Vertex Connectivity)，就是与顶点有关的连通性。研究无向图的点连通性，通常是通过删除无向图中的顶点(及与其所关联的每条边)后，观察和分析剩下的无向图连通与否。

1. 割顶集与顶点连通度

关于割顶集和顶点连通度有如下两种定义方式。

方式一：设 V'是连通图 G 的一个顶点子集，在 G 中删去 V'及与 V'关联的边后图不连通，则称 V'是 G 的**割顶集**(Vertex-cut Set)。如果割顶集 V'的任何真子集都不是割顶集，则称 V'为**极小割顶集**。顶点个数最小的极小割顶集称为**最小割顶集**。最小割顶集中顶点的个数，称作图 G 的**顶点连通度**(Vertex Connectivity Degree)，记做 $\kappa(G)$，且称图 G 是 ***κ*–连通**

图(κ–Connected Graph)。

方式二：设连通图 G 的阶数为 n，去掉 G 的任意 k−1 个顶点(及相关联的边)后(1≤k≤n)，所得到的子图仍然连通，而去掉某 k 个顶点(及所关联的边)后的子图不连通，则称 G 是 **κ–连通图**，k 称作图 G 的**顶点连通度**，记做 $\kappa(G)$。

如果割顶集中只有一个顶点，则该顶点可以称为**割点**(Cut-Vertex)或**关节点**。

规定，对 n 阶完全图 K_n，$\kappa(K_n)= n - 1$；对非连通图和平凡图，$\kappa(G) = 0$。

如图 8.3(a)所示的连通图，删除任意一个顶点(或两个顶点)都不会使剩下的子图不连通，而删除某 3 个顶点，就能使得剩下的子图不连通。例如，删除顶点子集{ v_2, v_6, v_8}及其所关联的边(在图 8.3(a)中用粗线标明)，剩下的子图如图 8.3(b)所示，为非连通图。因此原图的顶点连通度 $\kappa(G) = 3$，该图是 3 - 连通图。

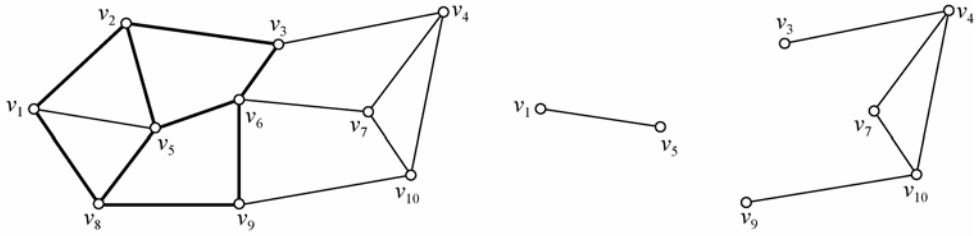


图 8.3　割顶集与顶点连通度

关节点的另外一种定义方式：在一个无向连通图 G 中，当删去 G 中的某个顶点 v 及其所关联的边后，可将图分割成 2 个或 2 个以上的连通分量，则称顶点 v 为**割点**，或者称为**关节点**。例如图 8.4(a)所示的无向连通图中，顶点 5, 4, 6, 8 都是关节点。

2. 点双连通图与点双连通分量

点双连通图：如果一个无向连通图 G 没有关节点，或者说点连通度 $\kappa(G)$>1，则称 G 为**点双连通图**，或者称为**重连通图**。

为什么称为点双连通图呢？因为在这种图中任何一对顶点之间至少存在 2 条无公共内部顶点(即除起点和终点外的顶点，n≥3)的路径，在删去某个顶点及其所关联的边时，也不会破坏图的连通性。例如，图 8.3(a)中，顶点 v_1 和 v_4 之间存在 3 条无公共内部顶点的路径：(v_1, v_2, v_3, v_4)、(v_1, v_5, v_6, v_7, v_4)和($v_1, v_8, v_9, v_{10}, v_4$)。

在一个表示通信网络的连通图中是不希望存在关节点的。在这种图中，用顶点表示通信站点，用边表示通信链路。如果一个通信站点是关节点，它一旦出现故障，将导致其他站点之间的通信中断。如果通信网络是点双连通图，那么某个站点一旦出现故障，也不会破坏图的连通性，整个系统还能正常运行。

点双连通分量：一个连通图 G 如果不是点双连通图，那么它可以包括几个点双连通分量，也称为**重连通分量**(或**块**)。一个连通图的重连通分量是该图的极大重连通子图，在重连通分量中不存在关节点。例如图 8.4(a)所示的连通无向图包含 6 个重连通分量，如图 8.4(b)所示。从图 8.4(b)可以看出，割点可以属于多个重连通分量，其余顶点属于且只属于一个重连通分量。

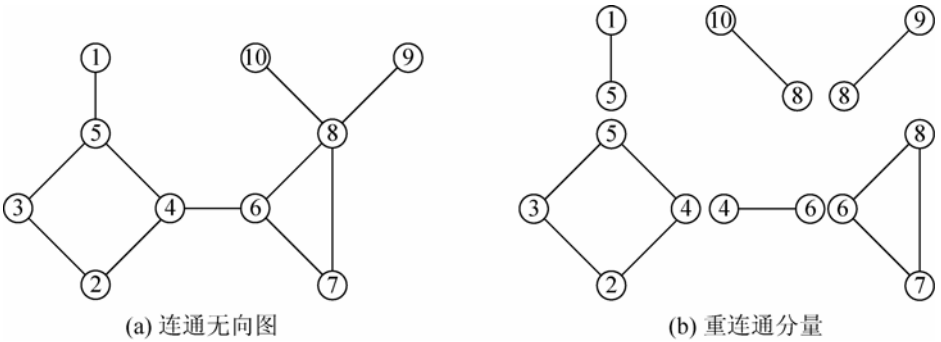


图 8.4　连通图和它的割点、重连通分量

8.1.3　无向图的边连通性

所谓**边连通性**(Edge Connectivity)，就是与边有关的连通性。研究无向图的边连通性，通常是通过删除无向图中的若干条边后，观察和分析剩下的无向图连通与否。

1. 割边集与边连通度

与割顶集和顶点连通度类似，割边集和边连通度也有如下两种定义方式。

方式一：设 E'是连通图 G 的边集的子集，在 G 中删去 E'后图不连通，则称 E'是 G 的**割边集**(Edge-Cut Set)。如果割边集 E'的任何真子集都不是割边集，则称 E'为**极小割边集**。边数最小的极小割边集称为**最小割边集**。最小割边集中边的个数，称作图 G 的**边连通度**(Edge Connectivity Degree)，记做 $\lambda(G)$，且称图 G 是 **λ-边连通图**(λ-Edge-Connected Graph)。

方式二：设连通图 G 的边数为 m，去掉 G 的任意 $\lambda-1$ 条边后($1 \leqslant \lambda \leqslant m$)，所得到的子图仍然连通，而去掉某 λ 条边后得到的子图不连通，则称 G 是 **λ－边连通图**，λ 称作图 G 的**边连通度**，记作 $\lambda(G)$。

如果割边集中只有一条边，则该边可以称为**割边**(Bridge)或**桥**。

如图 8.5(a)所示的连通图，删除任意一条边(或两条边)都不会使剩下的子图不连通，而删除某 3 条边，就能使得剩下的子图不连通。例如，删除边子集{ (v_2, v_3), (v_5, v_6), (v_8, v_9) }，剩下的子图如图 8.5(b)所示，为非连通图，因此原图的边连通度 $\lambda(G) = 3$，该图是 3－边连通图。

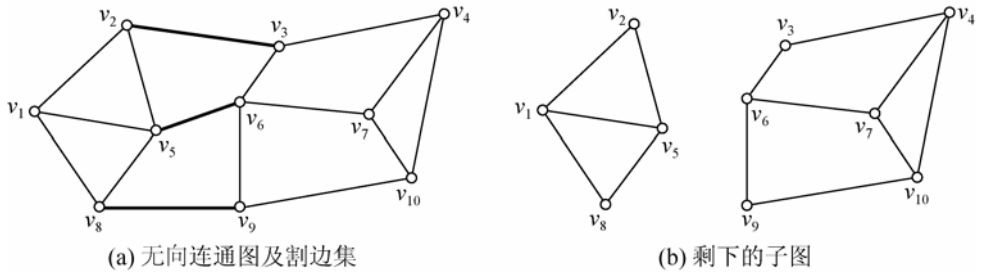


图 8.5　割边集与边连通度

割边同样也有另外一种定义方式：在一个无向连通图 G 中，当删去 G 中的某条边 e 后，可将图分割成两个或两个以上的连通分量，则称边 e 为**割边**，或者称为**桥**。例如图 8.4(a)

所示的无向连通图中，边(v_1, v_5)、(v_4, v_6)、(v_8, v_9)和(v_8, v_{10})都是割边。

2. 边双连通图与边双连通分量

边双连通图：如果一个无向连通图 G 没有割边，或者说边连通度 $\lambda(G)>1$，则称 G 为**边双连通图**。

为什么称为边双连通图呢？因为在这种图中任何一对顶点之间至少存在两条无公共边的路径(允许有公共内部顶点)，在删去某条边后，也不会破坏图的连通性。例如，图 8.5(a)中，顶点 v_8 和 v_4 之间存在两条无公共边的路径：(v_8, v_5, v_6, v_3, v_4)和(v_8, v_9, v_6, v_7, v_4)，这两条路径有一个公共内部顶点(当然这两个顶点之间还存在其他路径)。

边双连通分量：一个连通图 G 如果不是边双连通图，那么它可以包括几个边双连通分量。一个连通图的边双连通分量是该图的极大重连通子图，在边双连通分量中不存在割边。在连通图中，把割边删除，则连通图变成了多个连通分量，每个连通分量就是一个边双连通分量。例如，图 8.6(a)所示的连通无向图存在两条割边，(4,10)和(6,10)，把这两条割边删除后，得到 3 个边双连通分量，如图 8.6(b)所示。

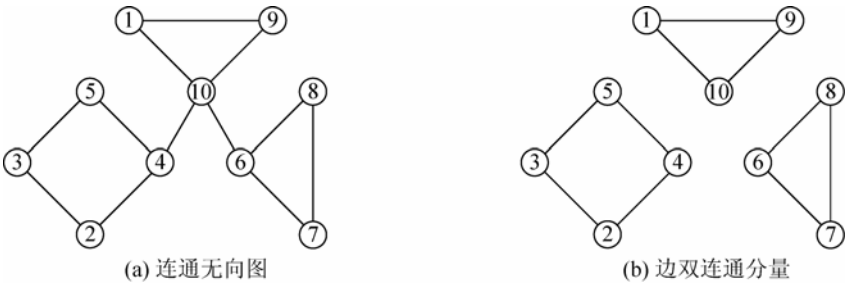


图 8.6　连通图和它的割边、边双连通分量

8.1.4　无向图顶点连通性和边连通性的联系

1. 顶点连通度和边连通度的联系

关于图的顶点连通度和边连通度，有如下定理。

定理 8.1(顶点连通度、边连通度与图的最小度的关系)　设 G 为无向连通图，则存在关系式：

$$\kappa(G)\leqslant\lambda(G)\leqslant\delta(G) \tag{8-1}$$

2. 割边和割点的联系

仔细观察图 8.4(a)和 8.6(a)中的割边和割点，两者之间有紧密的联系。具体可以用下面的定理来描述。

定理 8.2(割边和割点的联系)　设 v 是图 G 中与一条割边相关联的顶点，则 v 是 G 的割点当且仅当 $deg(v)\geqslant 2$。

8.1.5　有向图的连通性

由于有向图的边具有方向性，所以有向图的连通性比较复杂。根据有向图连通性的强弱可分为强连通、单连通和弱连通。

强连通(Strongly Connected)：若 G 是有向图，如果对图 G 中任意两个顶点 u 和 v，既存在从 u 到 v 的路径，也存在从 v 到 u 的路径，则称该有向图为**强连通有向图**。对于非强连通图，其极大强连通子图称为其**强连通分量**。

单连通(Simply Connected)：若 G 是有向图，如果对图 G 中任意两个顶点 u 和 v，存在从 u 到 v 的路径或从 v 到 u 的路径，则称该有向图为**单连通有向图**。

弱连通(Weak Connected)：若 G 是有向图，如果忽略图 G 中每条有向边的方向，得到的无向图(即有向图的基图)连通，则称该有向图为**弱连通有向图**。

强连通图一定也是单连通图和弱连通图，单连通图一定也是弱连通图。例如，图 8.7(a)所示的连通图 G_1 是强连通图，任何一对顶点间都存在双向的路径。图 8.7(b)所示的有向图 G_2 是单连通图，任何一对顶点间至少存在一个方向上的路径。图 8.7(c)所示的有向图 G_3 是弱连通图，基图连通，但顶点 6 和 5 之间不存在有向路径。

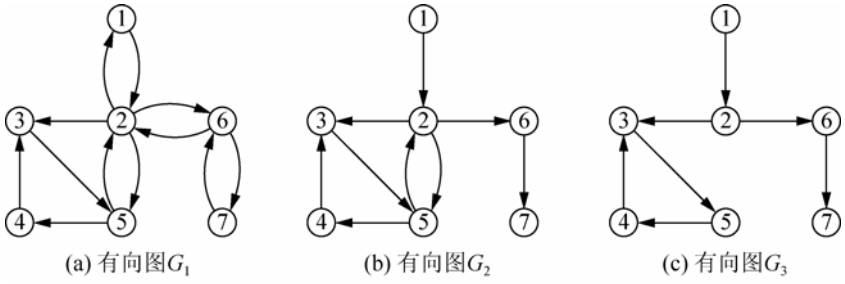


图 8.7　有向图的连通性

8.2　无向图点连通性的求解及应用

8.2.1　关节点的求解

1. 求关节点的朴素方法

判断关节点的一种朴素方法是从关节点的定义出发，依次去掉每个顶点(及其所关联的边)，然后用 DFS 去搜索整个图，可得到该图的连通分量个数，如果是大于 2，则该顶点是关节点。该方法的复杂度较高，为 $O(n^3)$，练习 8.1 可采取这种思路求解。当然，具体实现时并不真正需要去掉每个顶点(及其所关联的边)，只需要在搜索到该顶点时跳过该顶点就可以了。

2. 求关节点的算法——Tarjan 算法

前面介绍的求关节点的朴素方法，需要从每个顶点出发进行 DFS 遍历，用邻接矩阵存储时其复杂度为 $O(n^3)$。本节介绍的 Tarjan 算法只需从某个顶点出发进行一次遍历，就可以求得图中所有的关节点，因此其复杂度为 $O(n^2)$。接下来以图 8.8(a)所示的无向图为例介绍这种方法。

在图 8.8(a)中，对该图从顶点 4 出发进行深度优先搜索，实线表示搜索前进方向，虚线表示回退方向，顶点旁的数字标明了进行深度优先搜索时各顶点的访问次序，即深度优先数。在 DFS 搜索过程中，可以将各顶点的深度优先数记录在数组 dfn 中。

图 8.8(b)是进行 DFS 搜索后得到的根为顶点 4 的深度优先生成树。为了更加直观地描述树形结构，将此生成树改画成图 8.8(d)所示的树形形状。在图 8.8(d)中，还用虚线画出了两条虽然属于图 G，但不属于生成树的边，即(4, 5)和(6, 8)。

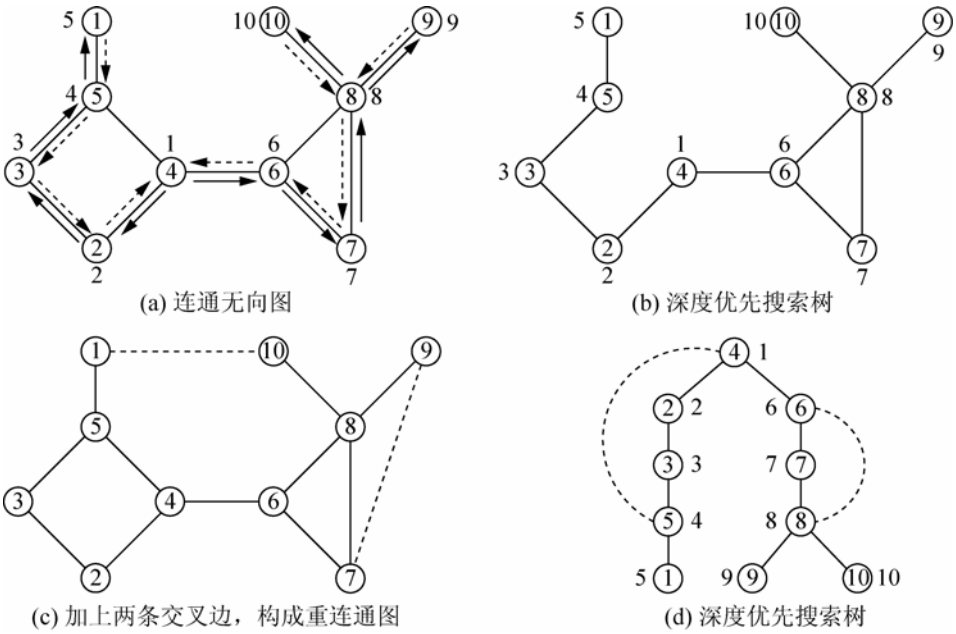


图 8.8　连通图和它的深度优先搜索树

请注意：在深度优先生成树中，如果 *u* 和 *v* 是两个顶点，且在生成树中 *u* 是 *v* 的祖先，则必有 dfn[u]<dfn[v]，表明 *u* 的深度优先数小于 *v*，*u* 先于 *v* 被访问。

图 G 中的边可以分为以下 3 种：

(1) 生成树的边，如(2, 4)、(6, 7)等。

(2) **回边**(Back Edge)：图 8.8(d)中虚线所表示的非生成树的边，称为回边。当且仅当 *u* 在生成树中是 *v* 的祖先，或者 *v* 是 *u* 的祖先时，非生成树的边(*u*, *v*)才成为一条回边。如图 8.8(a)及图 8.8(d)中的(4, 5)、(6, 8)都是回边。

(3) **交叉边**：除生成树的边、回边外，图 *G* 中的其他边称为交叉边。

请特别注意：一旦生成树确定以后，那么原图中的边只可能是回边和生成树的边，交叉边实际上是不存在的。为什么？(说明：对有向图进行 DFS 搜索后，非生成树的边可能是交叉边，详见 8.4 节。)

假设图 *G* 中存在边(1, 10)，如图 8.8(c)所示，这就是所谓的交叉边，那么顶点 10(甚至其他顶点都)只能位于顶点 4 的左边这棵子树中。另外，如果在图 *G* 中增加两条交叉边(1, 10)和(7, 9)，则图 *G* 就是一个重连通图，如图 8.8(c)所示。

顶点 *u* 是关节点的充要条件如下：

(1) **如果顶点 *u* 是深度优先搜索生成树的根，则 *u* 至少有两个子女**。为什么呢？因为删除 *u*，它的子女所在的子树就断开了，不用担心这些子树之间(在原图中)可能存在边，因为交叉边是不存在的。

(2) **如果 *u* 不是生成树的根，则它至少有一个子女 *w*，从 *w* 出发，不可能通过 *w*、*w***

的子孙，以及一条回边组成的路径到达 *u* 的祖先。为什么呢？这是因为如果删除顶点 *u* 及其所关联的边，则以顶点 *w* 为根的子树就从搜索树中脱离了。例如，顶点 6 为什么是关节点？这是因为它的一个子女顶点，如图 8.8(d)所示，即顶点 7，不存在如前所述的路径到达顶点 6 的祖先结点，这样，一旦顶点 6 删除了，则以顶点 7 为根结点的子树就断开了。又如，顶点 7 为什么不是关节点？这是因为它的所有子女顶点，当然在图 8.8(d)中只有顶点 8，存在如前所述的路径到达顶点 7 的祖先结点，即顶点 6，这样，一旦顶点 7 删除了，则以顶点 8 为根结点的子树仍然跟图 *G* 连通。

因此，可对图 *G* 的每个顶点 *u* 定义一个 low 值：low[*u*]是从 *u* 或 *u* 的子孙出发通过回边可以到达的最低深度优先数。low[*u*]的定义如下：

```
low[u]=Min
{
    dfn[u],
    Min{ low[w] | w 是 u 的一个子女},                                    (8-2)
    Min{ dfn[v] | v 与 u 邻接，且(u,v)是一条回边 }
}
```

即 low[*u*]是取以上 3 项的最小值，其中：第 1 项为它本身的深度优先数；第 2 项为它的(可能有多个)子女顶点 *w* 的 low[*w*]值的最小值，因为它的子女可以到达的最低深度优先数，则它也可以通过子女到达；第 3 项为它直接通过回边可以到达的最低优先数。

因此，顶点 *u* 是关节点的充要条件是：***u* 或者是具有两个以上子女的深度优先生成树的根，或者虽然不是一个根，但它有一个子女 *w*，使得 low[*w*]≥dfn[*u*]**。

其中，“low[*w*]≥dfn[*u*]”的含义是：顶点 *u* 的子女顶点 *w*，能够通过如前所述的路径到达顶点的最低深度优先数大于等于顶点 *u* 的深度优先数(注意在深度优先生成树中，顶点 *m* 是顶点 *n* 的祖先，则必有 dfn[*m*]<dfn[*n*])，即 *w* 及其子孙不存在指向顶点 *u* 的祖先的回边。这时删除顶点 *u* 及其所关联的边，则以顶点 *w* 为根的子树就从搜索树中脱离了。

每个顶点的深度优先数 dfn[*n*]值可以在搜索前进时进行统计，而 low[*n*]值是在回退的时候进行计算的。

接下来结合图 8.8 和表 8-1 解释在回退过程中计算每个顶点 *n* 的 low[*n*]值的方法(在表 8-1 中，当前计算出来的 low[*n*]值用粗体、斜体及下划线标明)。

表 8-1　计算各顶点的 dfn 与 low 值

顶点序号	1	2	3	4	5	6	7	8	9	10
dfn	5	2	3	1	4	6	7	8	9	10
low	***5***									***10***
low	5				***1***				***9***	10
low	5		***1***		1			***6***	9	10
low	5	***1***	1		1		***6***	6	9	10
low	5	1	1	***1***	1	***6***	6	6	9	10
	根的左子树，回退顺序为 1→5→3→2→4					根的右子树，回退顺序为 10→9→8→7→6				

(1) 在图 8.8(a)中，访问到顶点 1 后，要回退，因为顶点 1 没有子女顶点，所以 low[1]就等于它的深度优先数 dfn[1]，为 5。

(2) 从顶点 1 回退到顶点 5 后，要继续回退，此时计算顶点 5 的 low 值，因为顶点 5 可以直接通过回边(5, 4)到达根结点，而根结点的深度优先数为 1，所以顶点 5 的 low 值为 1。

(3) 从顶点 5 回退到顶点 3 后，要继续回退，此时计算顶点 3 的 low 值，因为它的子女顶点，即顶点 5 的 low 值为 1，则顶点 3 的 low 值也为 1。

(4) 从顶点 3 回退到顶点 2 后，要继续回退，此时计算顶点 2 的 low 值，因为它的子女顶点，即顶点 3 的 low 值为 1，则顶点 2 的 low 值也为 1。

(5) 从顶点 2 回退到顶点 4 后，要继续访问它的右子树中的顶点，此时计算顶点 4 的 low 值，因为它的子女顶点，即顶点 2 的 low 值为 1，则顶点 4 的 low 值也为 1。

根结点 4 的右子树在回退过程计算顶点的 low[*n*]，方法类似。

计算出各顶点的 low[*n*]值后，因为根结点，即顶点 4 有两个子女，所以顶点 4 是关节点；顶点 5 也是关节点，这是因为它的子女顶点，即顶点 1 的 low 值大于 dfn[5]；同样，顶点 6 和顶点 8 也是关节点。

求出关节点 *u* 后，还有一个问题需要解决：去掉该关节点 *u*，将原来的连通图分成了几个连通分量？答案如下。

(1) 如果关节点 *u* 是根结点，则有几个子女，就分成了几个连通分量。

(2) 如果关节点 *u* 不是根结点，则有 *d* 个子女 *w*，使得 low[*w*]≥dfn[*u*]，则去掉该结点，分成了 *d*+1 个连通分量。

以上方法的具体实现详见例 8.1。

3. 例题解析

例 8.1 SPF 结点(SPF)

题目来源：

Greater New York 2000, ZOJ1119, POJ1523

题目描述：

考虑图 8.9 中的两个网络，假定网络中的数据只在有线路直接连接的两个结点之间以点对点的方式传输。一个结点出现故障，比如图 8.3(a)所示的网络中结点 3 出现故障，将会阻止其他某些结点之间的通信。结点 1 和结点 2 仍然是连通的，结点 4 和结点 5 也是连通的，但这两对结点之间的通信无法进行了。因此结点 3 是这个网络的一个 SPF 结点。

严格的定义：对于一个连通的网络，如果一个结点出现故障，将会阻止至少一对结点之间的通信，则该结点是 SPF 结点。

注意：图 8.9(b)所示的网络不存在 SPF 结点。至少两个结点出现故障后，才会使得其他某对结点之间无法通信。

输入描述：

输入文件包含多个测试数据，每个测试数据描绘了一个网络。每个网络的数据包含多对整数，每对整数占一行，表示两个直接连接的结点。结点对中两个结点的顺序是无关的，1 2 和 2 1 表示同一对连接。结点序号范围为 1～1 000，每个网络的数据中最后一行为一个 0，表示该网络数据的结束。整个输入文件最后一行为一个 0，代表输入结束。读入时需要忽略输入文件中的空行。

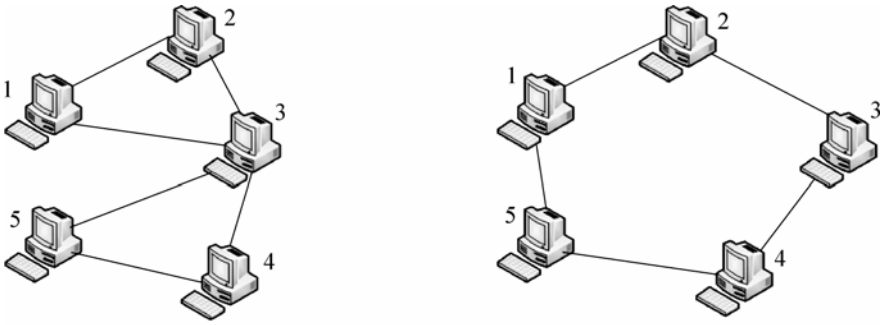


图 8.9　SPF 结点

输出描述：

对输入文件中的每个网络，首先输出该网络在输入文件中的序号，然后是该网络中的 SPF 结点。具体格式为：第 1 个网络的序号为"Network #1"；第 2 个网络的序号为"Network #2"，等等。对网络中的每个 SPF 结点，输出占一行，输出格式如样例输出所示，输出信息标明 SPF 结点的序号及该 SPF 结点出现故障后将整个网络分成几个连通的子网络。如果网络中不存在 SPF 结点，则只输出"No SPF nodes"。每两个网络的输出之间，输出一个空行。

样例输入：

```
1 2
5 4
3 1
3 2
3 4
3 5
0

1 2
2 3
3 4
4 5
5 1
0

0
```

样例输出：

```
Network #1
  SPF node 3 leaves 2 subnets

Network #2
  No SPF nodes
```

分析：

在用程序实现前面所述的求关节点的算法时，需要解决以下几个问题。

(1) 如何判断顶点 *v* 是顶点 *u* 的祖先结点。

(2) 如何判断边(*v*, *u*)是回边。

(3) 如何判断顶点 *v* 是顶点 *u* 的儿子结点。

这 3 个问题都是在深度优先搜索函数 dfs 中解决的。从顶点 *u* 出发进行 DFS 搜索时，要判断其他每个顶点 *v* 是否跟 *u* 是否邻接、是否未访问过。如果 *v* 跟 *u* 邻接，则在生成树中就是两种情况。

(1) 如果顶点 *v* 是顶点 *u* 的邻接顶点，且此时 *v* 还未访问过，则 *v* 是 *u* 的儿子结点。

(2) 如果顶点 *v* 是顶点 *u* 的邻接顶点，且此时 *v* 已经访问过了，则 *v* 是 *u* 的祖先结点，且(*v*, *u*)就是一条回边。

在下面的代码中，求每个顶点的 low[ ]值的代码特别地用边框标明了，从中可以看出，每次从顶点 *u* 的某一个邻接顶点回退到顶点 *u* 时，计算顶点 *u* 的 low[ ]值；当顶点 *u* 的所有邻接顶点都访问完毕，顶点 *u* 的 low[ ]值才计算完毕。

代码如下：

```
#define min(a,b) ((a)>(b)?(b):(a))
int Edge[1001][1001];   //邻接矩阵
int visited[1001];      //表示顶点访问状态
int nodes;              //顶点数目
int tmpdfn;             //在 dfs 过程中记录当前的深度优先搜索序数
int dfn[1001];          //每个顶点的 dfn 值
int low[1001];          //每个顶点的 low 值，根据该值来判断是否是关节点
int son;                //根节点的子女结点的个数(如果大于 2，则根节点是关节点)
int subnets[1001];      //记录每个结点(去掉该结点后)的连通分量个数
//深度优先搜索，记录每个结点的 low 值(根据 low 值来判断是否求关节点)
void dfs( int u )
{
    for( int v=1; v<=nodes; v++ )
    {
        //v 跟 u 邻接。在生成树中就是 2 种情况：
        //①v 是 u 的祖先结点，这样(v,u)就是一条回边；②v 是 u 的儿子结点
        if( Edge[u][v] )
        {
        if( !visited[v] )   //v 还未访问，v 是 u 的儿子结点，情形②
            {
                visited[v]=1;
                tmpdfn++;  dfn[v]=low[v]=tmpdfn;
                dfs( v );   //dfs(v)执行完毕后，low[v]值已求出
                //回退的时候，计算顶点 u 的 low 值
                low[u]=min( low[u], low[v] );
                if( low[v]>=dfn[u] )
                {
                    if( u!=1 )  subnets[u]++; //去掉该结点后的连通分量个数
                    //根结点的子女结点的个数(如果大于 2，则根结点是关节点)
                    if( u==1 )  son++;
                }
            }
            //此前 v 已经访问过了，v 是 u 的祖先结点((v,u)就是一条回边)：情形①
            else  low[u]=min(low[u], dfn[v]);
        }
    }
}
void init( )    //初始化函数
{
    low[1]=dfn[1]=1;
```

```
    tmpdfn=1;  son=0;
    memset( visited, 0, sizeof(visited) );
    visited[1]=1;
    memset( subnets, 0, sizeof(subnets) );
}
int main( )
{
    int i;              //循环变量
    int u, v;           //从输入文件中读入的顶点对
    int find;           //是否找到 SPF 节点的标志
    int number=1;       //测试数据数目
    while( 1 )
    {
        scanf( "%d", &u );
        if( u==0 )  break;  //整个输入结束
        memset( Edge, 0, sizeof(Edge) );
        nodes=0;
        scanf( "%d", &v );
        if( u>nodes )  nodes=u;
        if( v>nodes )  nodes=v;
        Edge[u][v]=Edge[v][u]=1;
        while( 1 )
        {
            scanf( "%d", &u );
            if( u==0 )  break;  //当前测试数据输入结束
            scanf( "%d", &v );
            if( u>nodes )  nodes=u;
            if( v>nodes )  nodes=v;
            Edge[u][v]=Edge[v][u]=1;
        }
        if( number>1 )  printf( "\n" ); //保证最后一个网络的输出之后没有空行
        printf( "Network #%d\n", number );
        number++;
        init( );    //初始化
        dfs( 1 );   //从顶点 1 开始搜索
        if( son>1 )  subnets[1]=son-1;
        find=0;
        for( i=1; i<=nodes; i++ )
        {
            if( subnets[i] )
            {
                find=1;
                printf(" SPF node %d leaves %d subnets\n", i, subnets[i]+1 );
            }
        }
        if( !find )  printf( "  No SPF nodes\n" );
    }
    return 0;
}
```

8.2.2 重连通分量的求解

在求关节点的过程中就能顺便把每个重连通分量求出。方法是：建立一个栈，存储当前重连通分量，在 DFS 过程中，每找到一条生成树的边或回边，就把这条边加入栈中。如果遇到某个顶点 u 的子女顶点 v 满足 dfn[u]≤low[v]，说明 u 是一个割点，同时把边从栈顶一条条取出，直到遇到了边(u, v)，取出的这些边与其关联的顶点，组成一个重连通分量。割点可以属于多个重连通分量，其余顶点和每条边属于且只属于一个重连通分量。

以上方法具体实现详见下面的例 8.2。

例 8.2 输出无向连通图各个连通分量

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为：第 1 行为两个整数 n 和 m，分别表示顶点个数和边数，然后有 m 行，每行表示一条边，为这条边的两个顶点的序号，顶点序号从 1 开始计起。假定无向图是连通的(可能存在割点，也可能没有割点)。$n = m = 0$ 时表示输入结束。

输出描述。对每个测试数据，以“u v”的形式依次输出各连通分量中的每条边，每个连通分量的数据占一行，用空格分隔每条边。各测试数据的输出之间用空行分隔开。

分析：

图 8.10 给出了两个测试数据。实线表示搜索的前进过程，虚线表示回退过程。搜索时从顶点 1 开始搜索。图 8.10(a)所示的无向图有两个割点、3 个重连通分量，图 8.10(b)所示的无向图没有割点、是重连通图。

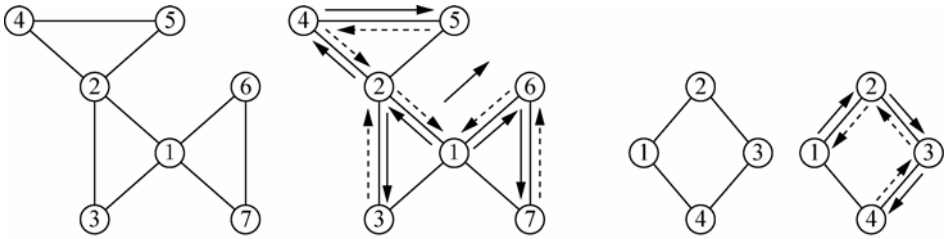


图 8.10 重连通分量的求解

下面的程序用一个数组来模拟栈。在搜索前进过程，依次将访问过的边入栈，在回退过程如果发现顶点 u 的子女顶点 v 满足 dfn[u]≤low[v]，说明 u 是一个割点，同时把边从栈顶一条条取出并输出，直到遇到边(u, v)为止(该边也要输出)，这些边组成了顶点 v 所在的重连通分量。

另外，为了防止访问过的边重复入栈，约定邻接矩阵 **Edge** 中，元素 **Edge**[u][v]取值为 1 表示顶点 u 和 v 之间有边连接、且该边没有遍历过，取值为 2 表示顶点 u 和 v 之间有边连接、且该边已经遍历过，取值为 0 表示顶点 u 和 v 之间没有边连接。

代码如下：

```
#define MAXN 20     //顶点数的最大值
#define MAXM 40     //边数的最大值
#define min(a,b) ((a)>(b)?(b):(a))
struct edge
```

```
{
    int u, v;       //边的两个顶点
    void output( ){ printf( "%d-%d", u, v ); }
    int comp( edge& t ){ return ( (u==t.u && v==t.v) || (u==t.v && v==t.u) );}
};
edge edges[MAXM];   //边的数组(模拟栈)
int se;             //栈顶
int Edge[MAXN][MAXN]; //邻接矩阵(1表示有连接,2表示有连接且已经走过,0表示没有连接)
int visited[MAXN];  //表示顶点访问状态
int n, m;           //顶点数、边数
int tmpdfn;         //在dfs过程中记录当前的深度优先搜索序数
int dfn[MAXN];      //每个顶点的dfn值
int low[MAXN];      //每个顶点的low值，根据该值来判断是否是关节点
//深度优先搜索，记录每个结点的low值(根据low值来判断是否求关节点)
void dfs( int u )
{
    for( int v=1; v<=n; v++ )
    {
        //v跟u邻接。在生成树中就是两种情况：
        //①v是u的祖先结点，这样(v,u)就是一条回边；②v是u的儿子结点
        if( Edge[u][v]==1 )
        {
            edge t;  t.u=u;  t.v=v;  edges[++se]=t; //边入栈
            Edge[u][v]=Edge[v][u]=2;    //设置边(u,v)已经访问过
            if( !visited[v] )   //v还未访问，v是u的儿子结点，情形②
            {
                visited[v]=1;
                tmpdfn++;  dfn[v]=low[v]=tmpdfn;
                dfs( v );   //dfs(v)执行完毕后，low[v]值已求出
                //回退的时候，计算顶点u的low值
                low[u]=min( low[u], low[v] );
                if( low[v]>=dfn[u] )    //删除顶点u，子女结点v所在的子树将脱离
                {
                    bool firstedge=true; //控制最后一条边后面没有空格的状态变量
                    while( 1 )
                    {
                        if( se<0 )  break;
                        if( firstedge )  firstedge=false;
                        else  printf( " " );
                        edge t1;
                        t1=edges[se];
                        t1.output( );       //输出栈顶结点t1
                        edges[se].u=0;  edges[se].v=0;  //"删除"栈顶t1
                        se--;
                        if( t1.comp(t) )  break;
                    }
                    printf( "\n" );
                }
            }
            //此前v已经访问过了，v是u的祖先结点((v,u)就是一条回边)：情形①
            else  low[u]=min(low[u], dfn[v]);
```

```
            }
        }
    }
    int main( )
    {
        int i;          //循环变量
        int u, v;       //从输入文件中读入的顶点对
        int number=1;   //测试数据数目
        while( 1 )
        {
            scanf( "%d%d", &n, &m );
            if( n==0 && m==0 )  break;  //整个输入结束
            memset( Edge, 0, sizeof(Edge) );
            for( i=1; i<=m; i++ )
            {
                scanf( "%d%d", &u, &v );
                Edge[u][v]=Edge[v][u]=1;
            }
            if( number>1 )  printf( "\n" ); //保证最后一个网络的输出之后没有空行
            number++;
            low[1]=dfn[1]=1;
            tmpdfn=1;
            memset( visited, 0, sizeof(visited) );
            visited[1]=1;
            memset( edges, 0, sizeof(edges) );
            se=-1;
            dfs( 1 );   //从顶点 1 出发进行 DFS 搜索
        }
        return 0;
    }
```

该程序的运行示例如下。

输入：

```
7 9
1 2
1 3
1 6
1 7
2 3
2 4
2 5
4 5
6 7
```

输出：

```
5-2 4-5 2-4
3-1 2-3 1-2
7-1 6-7 1-6
```

8.2.3 顶点连通度的求解

给定一个无向连通图，如何求其顶点连通度 $\kappa(G)$是本节要讨论的问题。$\kappa(G)$的求解需要转换成网络最大流问题。首先，介绍独立轨的概念。

独立轨：设 A、B 是无向图 G 的两个顶点，从 A 到 B 的两条没有公共内部顶点的路径，

互称为独立轨。A 到 B 独立轨的最大条数，记作 $P(A, B)$。例如，在图 8.11(a)所示的无向图中，v_1 和 v_4 之间有 3 条独立轨，用粗线标明。

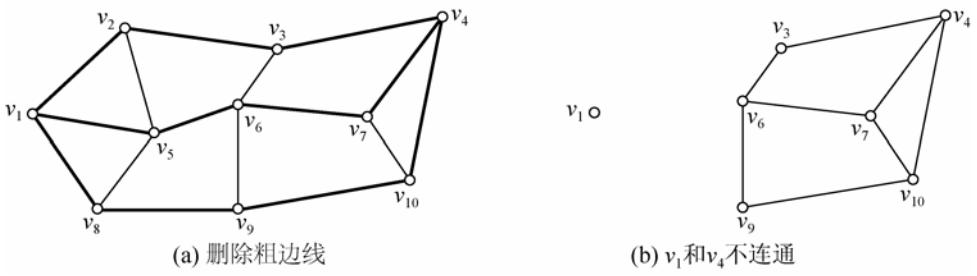


图 8.11　独立轨

设 A、B 是无向连通图 G 的两个不相邻的顶点，最少要删除多少个顶点才能使得 A 和 B 不再连通？答案是 $P(A, B)$个(证明略)。例如，在图 8.11(a)中，要使得 v_1 和 v_4 不再连通，可以在这两个顶点的 3 条独立轨上各选择一个顶点，如 v_2、v_5 和 v_8，删除这 3 个顶点后，v_1 和 v_4 不再连通了，如图 8.11(b)所示。注意，并不是在每条独立轨上任意删除一个顶点就可以达到目的。例如，在图 8.11(a)中，如果删除 v_2、v_5 和 v_{10}，则 v_1 和 v_4 仍然连通。

关于无向图 G 顶点连通度 $\kappa(G)$与顶点间独立轨数目之间的关系，有如下 Menger 定理。

定理 8.3(Menger 定理)　无向图 G 的顶点连通度 $\kappa(G)$和顶点间**最大独立轨数目**之间存在如下关系式：

$$k(G)=\begin{cases}|V(G)|-1 & \text{当 } G \text{ 是完全图}\\ \min\limits_{AB\notin E}\{P(A,B)\} & \text{当 } G \text{ 不是完全图}\end{cases} \tag{8-3}$$

在式 8-3 中，$AB \notin E$ 表示顶点 A 和 B 不相邻。为什么要强调不相邻？这是因为如果 A 和 B 相邻，则删除所有的其他顶点，A 和 B 都还是连通的。

那么如何求不相邻的两个顶点 A、B 间的最大独立轨数 $P(A, B)$呢，最少应删除图中哪些顶点(共 $P(A, B)$个顶点)才能使得 A、B 不连通呢？可以采用网络最大流方法来求解。

求 $P(A, B)$的方法如下。

(1) 为了求 $P(A, B)$，需要构造一个容量网络 N。

① 原图 G 中的每个顶点 v 变成网络 N 中的两个顶点 v'和 v''，顶点 v'到 v''有一条弧连接，即<v', v''>，其容量为 1。

② 原图 G 中的每条边 $e=(u, v)$，在网络 N 中有两条弧 $e'=\langle u'', v'\rangle$和 $e''=\langle u', v''\rangle$，$e'$ 和 e''的容量均为∞。

③另 A''为源点，B'为汇点。

(2) 求从 A″到 B′的最大流 F。

(3) 流出 A''的一切弧的流量和 $\sum\limits_{e\in(A'',v)} f(e)$，即为 $P(A, B)$，所有具有流量 1 的弧(v', v'')对应的顶点 v 构成了一个割顶集，在图 G 中去掉这些顶点后，则 A 和 B 不再连通了。

有了求 $P(A, B)$的算法基础，就可以得出 $\kappa(G)$的求解思路：首先设 $\kappa(G)$的初始值为∞；然后分析图 G 中的每一对顶点。如果顶点 A、B 不相邻，则用最大流的方法求出 $P(A, B)$和对应的割顶集；如果 $P(A, B)$小于当前的 $\kappa(G)$，则 $\kappa(G)=P(A, B)$，并保存其割顶集。如此直

至所有不相邻顶点对分析完为止，即可求出图的顶点连通度 $\kappa(G)$和最小割顶集了。具体实现时，可固定一个源点，枚举每个汇点，从而求出 $\kappa(G)$。

以上算法的程序实现，详见下面的例 8.3。

例 8.3 有线电视网络(Cable TV Network)

题目来源：

Southeastern Europe 2004, ZOJ2182, POJ1966

题目描述：

有线电视网络中，中继器的连接是双向的。如果网络中任何两个中继器之间至少有一条路，则中继器网络称为是连通的，否则中继器网络是不连通的。一个空的网络及只有一个中继器的网络被认为是连通的。具有 n 个中继器的网络的安全系数 f 被定义成：①f 为 n，如果不管删除多少个中继器，剩下的网络仍然是连通的；②f 为删除最少的顶点数，使得剩下的网络不连通。

例如，考虑图 8.12 所示的 3 个中继器网络，其中圆圈代表中继器，实线代表中继器之间的连接线缆。在图 8.12(a)中，删除任意多个顶点，剩下的网络仍然是连通的，因此，根据第①条规则，$f = n = 3$。在图 8.12(b)中，删除 0 个顶点，中继器网络就不连通，因此，根据第②条规则，$f = 0$。在图 8.12(c)中，至少需要删除中继器 1 和 2，或者 1 和 3，剩下的中继器网络不连通，因此，$f = 2$。

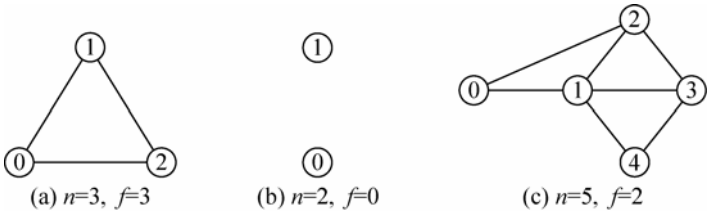


图 8.12 有线电视网络

输入描述：

编写程序以标准输入方式读入多个测试数据，计算每个测试数据所代表的中继器网络的安全系数。每个测试数据首先是两个整数 n 和 m，$0 \leqslant n \leqslant 50$，$n$ 表示网络中中继器数目，m 表示网络中线缆的数目；接下来是 m 个数据对(u, v), $u<v$，其中 u 和 v 为中继器的编号，中继器的编号为 0～n−1，(u, v)表示直接连接 u 和 v 的线缆。数据对可以以任何顺序出现。测试数据一直到文件尾。

输出描述：

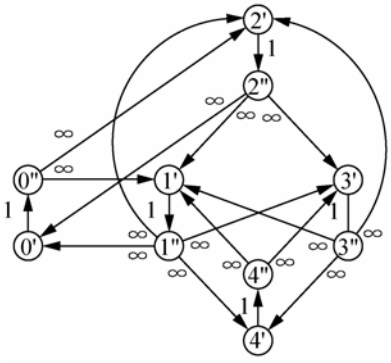
对每个测试数据，输出中继器网络的安全系数。

样例输入：	**样例输出：**
`3 3 (0,1) (0,2) (1,2)`	`3`
`2 0`	`0`
`5 7 (0,1) (0,2) (1,3) (1,2) (1,4) (2,3) (3,4)`	`2`

分析：

本题中的安全系数 f 实际上就是无向图的顶点连通度 $\kappa(G)$。以图 8.12(c)所示的测试数据为例描述 $\kappa(G)$的求解方法。根据前面介绍的方法，构造一个容量网络，如图 8.13(a)所示，

其邻接矩阵如图 8.13(b)所示。



(a) 构造的网络

	0'	1'	2'	3'	4'	0"	1"	2"	3"	4"
0'	0	0	0	0	0	1	0	0	0	0
1'	0	0	0	0	0	0	1	0	0	0
2'	0	0	0	0	0	0	0	1	0	0
3'	0	0	0	0	0	0	0	0	1	0
4'	0	0	0	0	0	0	0	0	0	1
0"	0	∞	∞	0	0	0	0	0	0	0
1"	∞	0	∞	∞	∞	0	0	0	0	0
2"	∞	∞	0	∞	0	0	0	0	0	0
3"	0	∞	∞	0	∞	0	0	0	0	0
4"	0	∞	0	∞	0	0	0	0	0	0

(b) 邻接矩阵

图 8.13　有线电视网络：顶点连通度的求解

在求每一对顶点的独立轨数目时，可固定源点为顶点 0"，枚举每个汇点，并记录最小的独立轨数目，该值就是所求的顶点连通度 $\kappa(G)$。

代码如下：

```
#define MAXN 105
#define INF 105
#define min(x,y) (x<y?x:y)
using namespace std;
int map[MAXN][MAXN];
int N, M;
//参数含义：结点数量 网络 源点 汇点
int max_flow( int num, int map[][MAXN], int source, int sink )
{
    int my_queue[MAXN], queue_first, queue_end; //数组做队列,实现 BFS 搜索路径
    int pre[MAXN], min_flow[MAXN]; //记录结点的父结点,当前路径中最小的一段的值,
                                     即限制值
    int flow[MAXN][MAXN];          //记录当前网络中的流
    int ans=0;                     //最终结果
    memset( flow, 0, sizeof(flow) );
    while( 1 )                     //一直循环，直到不存在增广路径
    {
        queue_first=0;             //初始化队列
        queue_end=0;
        my_queue[queue_end++]=source;
        memset( pre, -1, sizeof(pre) );
        min_flow[source]=INF;
        pre[source]=-2;                //源点的父结点需特殊标示
        while( queue_first<queue_end ) //BFS 寻找增广路径
        {
            int temp=my_queue[queue_first++];  //出队列
            for( int i=0; i<num; i++ )         //由结点 temp 往外扩展
            {
```

```
                //当结点 i 还未被探索到，并且还有可用流量
                if( pre[i]==-1 && flow[temp][i]<map[temp][i] )
                {
                    my_queue[queue_end++]=i;    //加入队列
                    pre[i]=temp;                //标示父结点
                    //求得 min_flow
                    min_flow[i]=min(min_flow[temp],(map[temp][i]-
                    flow[temp][i]));
                }
            }
//sink 的父结点不为初始值，说明 BFS 已经找到了一条路径
            if( pre[sink]!=-1 )
            {
                int k=sink;
                while( pre[k]>=0 )
                {
                    flow[pre[k]][k]+=min_flow[sink]; //将新的流量加入 flow
                    flow[k][pre[k]]=-flow[pre[k]][k];
                    k=pre[k];
                }
                break;
            }
        }
        if( pre[sink]==-1 )  return ans;    //不存在增广路径，返回
        else  ans+=min_flow[sink];
    }
}
int main( )
{
    while(scanf("%d%d",&N,&M) !=EOF )
    {
        int a, b, ans;
        int i;
        memset( map, 0, sizeof(map) );
        for( i=0; i<N; i++ )  map[i][i+N]=1;
        for( i=0; i<M; i++ )
        {
            scanf( " (%d,%d)", &a, &b );
            map[b+N][a]=map[a+N][b]=INF;
        }
        ans=INF;
        for( i=1; i<N; i++ )
        {
            ans=min( max_flow(N*2,map,0+N,i), ans );
        }
        if( ans==INF )  ans=N;
        printf( "%d\n", ans );
    }
    return 0;
}
```

练　习

8.1　电话线路网络(Network)

题目来源：

Central Eurpe 1996, ZOJ1311, POJ1144

题目描述：

TLC 电话线路公司正在新建一个电话线路网络。他们将一些地方(这些地方用 1～*N* 的整数标明，任何两个地方的标号都不相同)用电话线路连接起来。这些线路是双向的，每条线路连接两个地方，并且每个地方的电话线路都连接到一个电话交换机。每个地方都有一个电话交换机。从每个地方都可以达到其他一些地方(如果有线路连接的话)，然而这些线路不一定必须是直接连接的，也可以是通过几个电话交换机到达另外一个地方。但是有时会因为电力不足导致某个地方的交换机不能工作。TLC 的官员意识到一旦出现这种情况(在某个地方的交换机不工作，即这个结点与其他结点之间的线路都断开了)，除了这个出现故障的地方是不可达外，还可能导致其他一些(本来连通的)地方也不再连通。称这个地方为关节点。

现在 TLC 的官员努力想写一个程序来找到关节点的数目。请帮助他们。

输入描述：

输入文件包括多个测试数据，每个测试数据描述了一个网络。每个测试数据的第 1 行是一个整数 *N*，代表结点的数目(*N*<100)。接下来，该网络至多有 *N* 行信息，每行包括一个地方的号码，以及有直接线路通往其他地方的号码。这些行(至多 *N* 行)完整地描述了网络，也就是说，该网络中每条直接连接两个地方的线路将会至少出现在某一行中。每行中的整数都用空格隔开。每个网络的信息最后一行为数字 0，表示该网络数据的结束。

输入文件最后一行为 0，表示输入文件结束。

输出描述：

对输入文件中的每个网络，输出关节点的个数。

样例输入：	**样例输出：**
5 5 1 2 3 4 0 6 2 1 3 5 4 6 2 0 0	1 2

样例输入中两个测试数据所描绘的电话网络如图 8.14(a)和 8.14(b)所示，从中可以看出这两个网络中分别包含了 1 个和 2 个关节点。

8.2　仓库管理员

题目来源：

POI1999

(a) 测试数据1　　(b) 测试数据2

图 8.14　电话线路网络中的关节点

题目描述：

码头仓库是一块 $N\times M$ 个格子的矩形，有的格子是空闲的—没有任何东西，有的格子上已经堆放了沉重的货物—太重了而不可能被移动。现在，仓库管理员有一项任务，要将一个较小箱子推到指定的格子上去。管理员可以在仓库中移动，但不得跨过对方沉重的货物的格子。当管理员站在与箱子相邻的格子上时，他可以做一次推动，把箱子推到另一个相邻的格子。考虑到箱子比较重，仓库管理员为了节省体力，想尽量减少推箱子的次数。你能帮帮他么？

输入描述：

输入文件第一行有两个数 N、$M(1\leqslant N, M\leqslant 100)$，表示仓库是 $N\times M$ 的矩形。以下有 N 行，每行有 M 个字符，表示一个格子的状态。

S：表示该格子上放了不可移动的沉重货物。

w：表示该格子上没有任何东西。

M：表示仓库管理员初始的位置。

P：表示箱子的初始位置。

K：表示箱子的目标位置。

输出描述：

输出文件只有一行，为一个数，表示仓库管理员最少要推多少次箱子。如果仓库管理员不可能将箱子推到目标位置，那么请输出 NIE，表示无解。

样例输入：

```
10 12
SSSSSSSSSSSS
SwwwwwwwSSSS
SwSSSSwwSSSS
SwSSSSwwSKSS
SwSSSSwwSwSS
SwwwwwPwwwww
SSSSSSSwSwSw
SSSSSSMwSwww
SSSSSSSSSSSS
SSSSSSSSSSSS
```

样例输出：

```
7
```

8.3　备用交换机

题目描述：

n 个城市之间有通信网络，每个城市都有通信交换机，直接或间接与其他城市连接。

因电子设备容易损坏，需给通信点配备备用交换机。但备用交换机数量有限，不能全部配备，只能给部分重要城市配置。于是规定：如果某个城市由于交换机损坏，不仅本城市通信中断，还造成其他城市通信中断，则配备备用交换机。请根据城市线路情况，计算需配备备用交换机的城市个数，及需配备备用交换机城市的编号。

输入描述：

输入文件有一个测试数据，占有若干行：第 1 行为一个整数 n，表示共有 n 个城市($2 \leqslant n \leqslant 100$)，接下来有若干行(一直到文件尾)，每行两个正整数 a、b，a、b 是城市编号，表示 a 与 b 之间有直接通信线路。

输出描述：

输出的第一行为 1 个整数 m，表示需 m 个备用交换机，接下来有 m 行，每行有一个整数，表示需配备交换机的城市编号，输出顺序按编号由小到大。如果没有城市需配备备用交换机则输出 0。

样例输入：

```
7
1 2
2 3
2 4
3 4
4 5
4 6
4 7
5 6
6 7
```

样例输出：

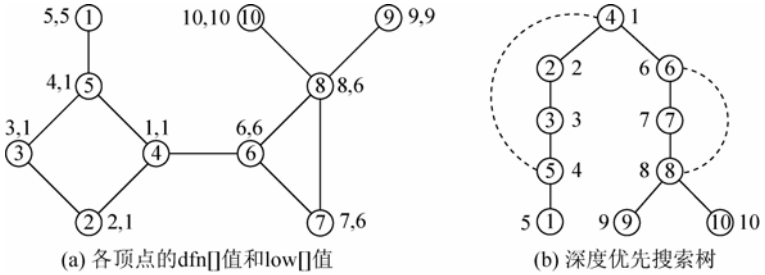
```
2
2
4
```

8.3　无向图边连通性的求解及应用

8.3.1　割边的求解

割边的求解过程与 8.2.1 节中求关节点的过程类似，判断方法是：无向图中的一条边(u, v)是桥，当且仅当(u, v)为生成树中的边，且满足 dfn[u]<low[v]。

例如，图 8.15(a)所示的无向图，如果从顶点 4 开始进行 DFS 搜索，各顶点的 dfn[ ]值和 low[ ]值如图 8.15(a)所示(每个顶点旁的两个数值分别表示 dfn[ ]值和 low[ ]值)，深度优先搜索树如图 8.15(b)所示。根据上述判断方法，可判断出边(1, 5)、(4, 6)、(8, 9)和(9, 10)为无向图中的割边。



(a) 各顶点的dfn[]值和low[]值　　(b) 深度优先搜索树

图 8.15　割边的求解

割边求解的程序实现详见下面的例 8.4。

例 8.4　烧毁的桥(Burning Bridges)

题目来源：

Andrew Stankevich's Contest #5, ZOJ2588

题目描述：

Ferry 王国是一个漂亮的岛国，一共有 N 个岛国、M 座桥，通过这些桥从每个小岛都能到达任何一个小岛。很不幸的是，最近 Ferry 王国被 Jordan 征服了。Jordan 决定烧毁所有的桥。这是个残酷的决定，但是 Jordan 的谋士建议他不要这样做，因为如果烧毁所有的桥梁，他自己的军队也不能从一个岛到达另一个岛。因此 Jordan 决定烧尽可能多的桥，只要能保证他的军队能从任何一个小岛都能到达每个小岛就可以了。

现在 Ferry 王国的人民很想知道哪些桥梁将被烧毁。当然，他们无法得知这些信息，因为哪些桥将被烧毁是 Jordan 的军事机密。然而，可以告知 Ferry 王国的人民哪些桥肯定不会被烧毁。

输入描述：

输入文件中包含多个测试数据。输入文件中第 1 行为一个整数 T，$1 \leqslant T \leqslant 20$，表示测试数据的数目。接下来有 T 个测试数据，测试数据之间用空行隔开。

每个测试数据的第 1 行为两个整数 N 和 M，分别表示岛的数目和桥的数目，$2 \leqslant N \leqslant 10\,000$，$1 \leqslant M \leqslant 100\,000$；接下来有 M 行，每行为两个不同的整数，为一座桥所连接的小岛的编号。注意，两个岛之间可能有多座桥。

输出描述：

对每个测试数据，首先在第 1 行输出一个整数 K，表示 K 座桥不会被烧毁；第 2 行输出 K 个整数，为这些桥的序号。桥的序号从 1 开始计起，按输入的顺序进行编号。

两个测试数据的输出之间有一个空行。

样例输入：

```
2

6 7
1 2
2 3
2 4
5 4
1 3
4 5
3 6

10 16
2 6
3 7
6 5
5 9
5 4
1 2
```

样例输出：

```
2
3 7

1
4
```

```
9 8
6 4
2 10
3 8
7 9
1 4
2 4
10 5
1 6
6 10
```

分析：

本题的意思是给定一个无向连通图，要求图中的割边，因为割边所表示的“桥”是不能被烧毁的(注意，此处加了双引号的“桥”是指题目中的桥梁，并不是指割边)。在本题中，由于有重边，所以无法用邻接矩阵来存储图，必须用邻接表来存储。对于重边的处理，只要顶点 u 和 v 之间有重边，那么这些重边任何一条都不可能是割边。

图 8.16 描述了题目中的两个测试数据，其中每条边旁边的数字为边的序号。在图 8.16(a)中，边(2, 4)和(3, 7)为割边，不能被删除。在图 8.16(b)中，边(5, 9)为割边，不能被删除；如果被删除，则顶点 3、9、8、7 构成一个连通分量，从原图中分离了。

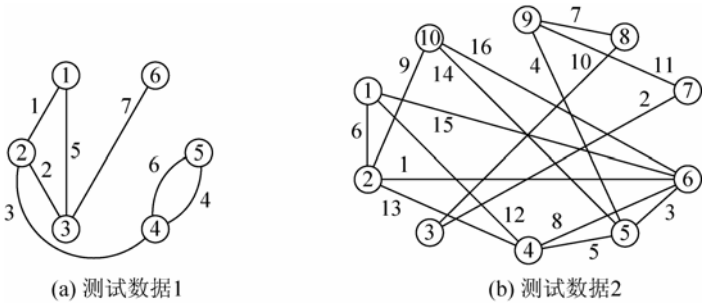


图 8.16　烧毁的桥：测试数据

在下面的代码中，从顶点 0 出发进行一次 DFS 遍历，并求得每个顶点的 dfn[ ]值和 low[ ]值。对除重边以外的每条生成树中的边(u, v)，如果满足 dfn[u]<low[v]，则边(u, v)是割边。

代码如下：

```
#define clr(a) memset(a,0,sizeof(a))
#define N 10005
#define M 100005
#define MIN(a,b) ((a)>(b)?(b):(a))
struct Node                 //边结点
{
    int j, tag, id;         //j 为另一个顶点的序号，tag 为重边的数量，id 为序号
    Node *next;             //下一个边结点
};
int n, m;           //顶点数、边数
int nid;            //nid 为输入时边的序号
Node mem[M*2];  int memp;   //mem 为存储边结点的数组，memp 为 mem 数组中的序号
Node *e[N];         //邻接表
```

```
int bridge[M];      //bridge[i]为 1，则第 i+1 条边为割边
int nbridge;        //求得的割边的数目
int low[N], dfn[N]; //low[i]为顶点 i 可达祖先的最小编号,dfn[i]为深度优先数
int visited[N]; //visited[i]为 0-未访问，为 1-已访问，为 2-已访问且已检查邻接顶点
//在邻接表中插入边(i,j)，如果有重边，则只是使得相应边结点的 tag 加 1
int addEdge( Node *e[], int i, int j )
{
    Node* p;
    for( p=e[i]; p!=NULL; p=p->next )
    {
        if( p->j==j )  break;
    }
    if( p!=NULL )   //(i,j)为重边
    { p->tag++; return 0; }
    p=&mem[memp++];
    p->j=j;  p->next=e[i];  e[i]=p;  p->id=nid;  p->tag=0;
    return 1;
}
//参数含义：i-当前搜索的顶点，father-i 的父亲顶点，dth-搜索深度
void DFS( int i, int father, int dth )
{
    visited[i]=1;  dfn[i]=low[i]=dth;
    Node *p;
    for( p=e[i]; p!=NULL; p=p->next )
    {
        int j=p->j;
        if( j!=father && visited[j]==1 )
            low[i]=MIN( low[i], dfn[j] );
        if( visited[j]==0 ) //顶点 j 未访问
        {
            DFS( j, i, dth+1 );
            low[i]=MIN( low[i], low[j] );
            if( low[j]>dfn[i]&&!p->tag )   //重边不可能是割边
                bridge[p->id]=++nbridge;
        }
    }
    visited[i]=2;
}
int main( )
{
    int i, j, k, T; //T 为测试数据数目
    scanf( "%d", &T );
    while( T-- )
    {
        scanf( "%d%d", &n, &m );
        memp=0;  nid=0;  clr(e);
        for( k=0; k<m; k++, nid++ )//读入边，存储到邻接表中
        {
            scanf( "%d%d", &i, &j );
            addEdge( e, i-1, j-1 );  addEdge( e, j-1, i-1 );
            bridge[nid]=0;
```

```
            }
            nbridge=0;  clr(visited);
            //从顶点 0 出发进行 DFS 搜索，顶点 0 是根结点，所以第 2 个参数为-1
            DFS( 0, -1, 1 );
            printf( "%d\n", nbridge ); //输出割边的信息
            for( i=0, k=nbridge; i<m; i++ )
            {
                if( bridge[i] )
                {
                    printf( "%d", i+1 );
                    if( --k )  printf( " " );
                }
            }
            if( nbridge )  puts("");
            if( T )  puts("");
        }
        return 0;
    }
```

8.3.2　边双连通分量的求解

与 8.2.2 节点双连通分量的求解相比，边双连通分量的求法更为简单。只需在求出所有的桥以后，把桥删除，原图变成了多个连通块，则每个连通块就是一个边双连通分量。桥不属于任何一个边双连通分量，其余的边和每个顶点都属于且只属于一个边双连通分量。

以下通过 2 道例题讲解边双连通分量的求解及应用。

例 8.5　圆桌武士(Knights of the Round Table)

题目来源：

Central Europe 2005, POJ2942

题目描述：

武士是一个十分吸引人的职业，因此近年来 Arthur 王国武士的数量得到了空前的增长。武士在讨论事情时很容易激动，特别是喝了酒以后。在发生一些不幸的打斗后，Arthur 国王要求智者 Merlin 确保将来不会发生武士的打斗。

Merlin 在仔细研究这个问题后，他意识到如果武士围着圆桌坐下，要阻止打斗则必须遵循以下两个规则。

(1) 任何两个互相仇视的武士不能挨着坐，Merlin 有一张清单，列出了互相仇视的武士，注意，武士是围着圆桌坐下的，每个武士有两个相邻的武士。

(2) 围着圆桌坐下的武士数量必须为奇数个。这将能保证当武士在争论一些事情时，能通过投票的方式解决争端。而如果武士数量为偶数个，则可能会出现赞同和反对的武士数量一样。

如果以上两个条件满足，Merlin 将让这些武士围着圆桌坐下，否则他将取消圆桌会议。如果只有一个武士到了，Merlin 也将取消会议，因为一个武士无法围着圆桌坐下。Merlin 意识到如果遵守以上两个规则，可能会使某些武士不可能被安排坐下，一种情况是一个武士仇视其他每个武士。如果一个武士不可能被安排坐下，他将被从武士名单中剔除掉。试帮助 Merlin 判断有多少个武士将会被剔除掉。

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 行为两个整数 n 和 m，$1 \leqslant n \leqslant 1\,000$，$1 \leqslant m \leqslant 1\,000\,000$，$n$ 表示武士数目，n 个武士编号为 1～n，m 表示武士相互仇视的对数。接下来有 m 行，描述了相互仇视的每对武士，每行为两个整数 k_1 和 k_2，表示武士 k_1 和 k_2 相互仇视。

$n = m = 0$ 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，为从名单中被剔除掉的武士数目。

样例输入：

```
5 5
1 4
1 5
2 5
3 4
4 5
0 0
```

样例输出：

```
2
```

分析：

题目的意思是：一群武士，某些武士之间互相仇视，如果在一起容易发生争斗事件。因此他们只有满足一定条件才能参加圆桌会议：① 圆桌边上任意相邻的两个武士不能互相仇视；② 同一个圆桌边上的武士数量必须是奇数。

为了使得不互相仇视的武士才能坐在一起，可以做如下处理：将各武士看成顶点，不互相仇视的武士之间存在边，建立无向图。例如，根据样例输入中的测试数据所构造得到的无向图如图 8.17 所示。

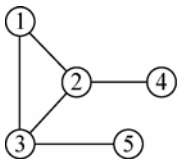


图 8.17　圆桌武士：测试数据

构造无向图以后，先按照要求①，将所有能坐在一起的武士分为一组，全部武士分为若干组，每一组在图中是一个双连通分量。然后根据双连通分量的性质，判断双连通分量中是否存在奇圈，如果存在，则这一组武士都能参与会议，反之这一组都不能参与会议。

具体的方法如下。

(1) 搜索双连通分量。深度优先搜索过程中，用一个栈保存所有经过的结点，判断割点，碰到割点就标记当前栈顶点的结点并退栈，直到当前结点停止并标记当前割点。标记过的结点处于同一个双连通分量。

(2) 交叉染色搜索奇圈。在一个结点大于 2 的双连通分量中，必定存在一个圈经过该连通分量的所有节点；如果这个圈是奇圈，则该连通分量内所有的点都满足条件；若这个圈是偶圈，如果包含奇圈，则必定还有一个奇圈经过所有剩下的点。因此一个双连通分量中只要存在一个奇圈，那么该双连通分量内所有的点都处于一个奇圈中。根据这个性质，

只需要在一个双连通分量内找奇圈即可判断该联通分量是否满足条件。交叉染色法就是在DFS的过程中反复交换着用两种不同的颜色对为染色过的点染色，若某次DFS中当前结点的子结点和当前节点同色，则找到奇圈。

(3) 需要注意的地方是：因为同一个点可能在多个双连通分量中，因此标记某个点是否满足条件必须专门用一数组标记。

代码如下：

```
    #define N 1001
    #define max(a,b) ((a)>(b)?(a):(b))
    #define min(a,b) ((a)<(b)?(a):(b))
    struct edge //邻接表结构
    {
        int belongto;
        edge *next;
        edge(int u,int v) : belongto(v), next(P[u]){ }
    }*P[N+1];
    int G[N+1][N+1], used[N+1], part[N+1], deep[N+1], anc[N+1], open[N+1],
color[N+1];
    int n, m, num, top;
    void ReadData( )    //读入数据
    {
        int i, j;
        memset(G,0,sizeof(G));
        while( m-- )
        {
            scanf( "%d%d", &i, &j );
            G[i][j]=G[j][i]=1;
        }
        for( i=1; i<=n; i++ )  G[i][i]=1;
        for( i=1; i<=n; i++ )
        {
            for( j=1; j<=n; j++ )
                G[0][j]=G[i][j];
            for( j=1; j<=n; j++ )
                if( !G[0][j] )
                    G[i][++G[i][0]]=j;
        }
    }
    void DFS( int s, int father, int d )   //DFS标记割点(搜索双连通分量)
    {
        int i, j, k;
        anc[s]=deep[s]=d;
        used[s]=1;
        open[top++]=s;
        for( i=1; i <= G[s][0]; i++ )
        {
```

```
            j=G[s][i];
            if( j != father&&used[j]==1 )
                anc[s]=min( anc[s], deep[j] );
            if( !used[j] )
            {
                DFS( j, s, d+1 );
                anc[s]=min( anc[s], anc[j] );
                if( anc[j]>=deep[s] )
                {
                    num++;
                    P[s]=new edge( s, num );
                    for( k=open[top]; k!=j; P[k]=new edge(k,num) )
                        k=open[--top];
                }
            }
        }
        used[s]=2;
    }
    void SearchConn( ) //搜索双连通分量
    {
        int i;
        memset(used,0,sizeof(used));
        memset(P,0,sizeof(P));
        num=0, top=0;
        for( i=1; i<=n; i++ )
            if( !used[i] ) DFS( i, -1, 1 );
    }
    bool OddCycle( int s, int col ) //DFS 交叉染色搜索判断奇圈
    {
        int i, j;
        color[s]=col;  //染色
        for( i=1; i<=G[s][0]; i++ )
        {
            j=G[s][i];
            if( part[j] )
            {
                if( color[j]==0&&OddCycle(j,-col) ) return 1;
                if( color[j]==col ) return 1;
            }
        }
        return 0;
    }
    int Calculate( )   //累计数目
    {
        int i, j, count=0;
        memset(used,0,sizeof(used));
        for( i=1; i<=num; i++ )
```

```
    {
        memset(part,0,sizeof(part));  memset(color,0,sizeof(color));
        for( j=1; j<=n;j++ )
        {
            for( edge *L=P[j]; L; L=L->next )
                if( L->belongto==i )
                {
                    part[j]=1;  break;
                }
        }
        for( j=1; j<=n; j++ )
        {
            if( part[j] )
            {
                if( OddCycle(j,1) )
                    for( j=1;j<=n;j++ ) used[j]+=part[j];
                break;
            }
        }
    }
    for( i=1; i<=n; i++ )
        if( !used[i] )  count++;
    return count;
}
int main( )
{
    while( scanf("%d%d",&n,&m)!=EOF&&n|m )
    {
        ReadData( );                    //读入数据
        SearchConn( );                  //搜索双连通分量
        printf( "%d\n", Calculate() ); //搜索奇圈并输出累计结果
    }
    return 0;
}
```

例 8.6　多余的路(Redundant Paths)

题目来源：

USACO 2006 January Gold, POJ3177

题目描述：

有 F 个牧场，$1 \leqslant F \leqslant 5\,000$，贝茜和她的牧群经常需要从一个牧场迁移到另一个牧场。奶牛们已经厌烦老是走同一条路，所以有必要再新修几条路，这样它们从一个牧场迁移到另一个牧场时总是可以选择至少两条独立的路。现在 F 个牧场的任何两个牧场之间已经至少有一条路了，奶牛们需要至少有两条。

给定现有的 R 条直接连接两个牧场的路，$F-1 \leqslant R \leqslant 10\,000$，计算至少需要新修多少条直接连接两个牧场的路，使得任何两个牧场之间至少有两条独立的路。两条独立的路是指

没有公共边的路，但可以经过同一个中间顶点。

输入描述：

测试数据的格式如下。

第 1 行为两个整数 F 和 R。

第 2～R+1 行，每行为两个整数，为一条道路连接的两个牧场。

输出描述：

输出一行，为一个整数，表示需要新修的道路数目。

样例输入：

```
7 7
1 2
2 3
3 4
2 5
4 5
5 6
5 7
```

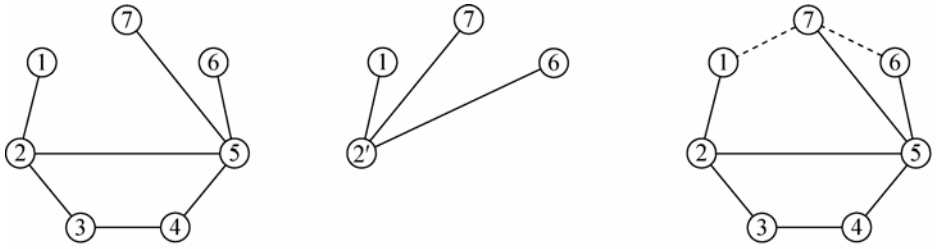
样例输出：

```
2
```

分析：

本题的意思是给定一个无向连通图，判断最少需要加多少条边，才能使得任意两点之间至少有两条相互“边独立”的道路。显然，在同一个边双连通分量里的所有点可以等价地看做一个点，收缩后，新图是一棵树，树的边是原无向图的桥。现在问题转化为“在树中至少添加多少条边能使图变为边双连通图”。结论是：添加边数=(树中度为 1 的结点数+1) /2。

例如，题目中给定的样例测试数据所描绘的牧场网络如图 8.18(a)所示，在图 8.18(a)中有一个边双连通分量，即顶点 2、3、4、5 组成的连通分量。将该边双连通分量收缩成一个顶点 2’，如图 8.18(b)所示，这样就得到一棵树，该树有 3 个叶子结点，因此添加的边数为(3+1)/2，为两条边。图 8.18(c)给出了一种添加方案，当然还存在其他的添加方案。



(a) 测试数据　(b) 收缩后得到的树　(c) 新修的道路(一种方案)

图 8.18　多余的路：测试数据

下面的代码采用并查集来收缩同一个边双连通分量上的顶点。将所有的边双连通分量分别收缩成一个顶点后，统计得到的树中叶子结点的个数，然后求应添加的边数。

代码如下：

```
#define clr(a) memset(a,0,sizeof(a))
#define MIN(a,b) ((a)>(b)?(b):(a))
#define N 1005
```

```
#define M 20005
struct Node                 //边结点
{
    int j;                  //j 为另一个顶点的序号
    Node *next;             //下一个边结点
};
int n, m;               //顶点数、边数
Node mem[M];  int memp; //mem 为存储边结点的数组，memp 为 mem 数组中的序号
Node *e[N];             //邻接表
int w;                  //原图中边双连通分量的个数
int belong[N];
int low[N], dfn[N]; //low[i]为顶点 i 可达祖先的最小编号,dfn[i]为深度优先数
int visited[N]; //visited[i]为 0-未访问，为 1-已访问，为 2-已访问且已检查邻接顶点
int bridge[M][2], nbridge;
void addEdge( Node *e[], int i, int j )    //在邻接表中插入边(i,j)
{
    Node *p=&mem[memp++];
    p->j=j;  p->next=e[i];  e[i]=p;
}
int FindSet( int f[], int i )  //并查集的查找函数
{
    int j=i, t;
    while( f[j]!=j )  j=f[j];
    while( f[i]!=i ){ t=f[i];  f[i]=j;  i=t; }
    return j;
}
void UniteSet( int f[], int i, int j ) //并查集的合并函数
{
    int p=FindSet(f,i),  q=FindSet(f,j);
    if( p!=q )  f[p]=q;
}
void DFS_2conn( int i, int father, int dth, int f[] )
{
    int j, tofather=0;
    Node *p;
    visited[i]=1;  low[i]=dfn[i]=dth;
    for( p=e[i]; p!=NULL; p=p->next )
    {
        j=p->j;
        if( visited[j]==1 && (j!=father||tofather) )
            low[i]=MIN(low[i],dfn[j]);
        if( visited[j]==0 )
        {
            DFS_2conn( j, i, dth+1, f );
            low[i]=MIN( low[i], low[j] );
            if(low[j]<=dfn[i]) UniteSet(f, i, j);  //i,j 在同一个双连通分量
            if( low[j]>dfn[i] )                    //边(i,j)是桥
                bridge[nbridge][0]=i,  bridge[nbridge++][1]=j;
        }
        if( j==father )  tofather=1;
    }
```

```
        visited[i]=2;
    }
    //求无向图极大边双连通分量的个数
    int DoubleConnection( )
    {
        int i, k, f[N],ncon=0;
        for( i=0; i<n; i++ )  f[i]=i, belong[i]=-1; //f[]并查集数组
        clr( visited );  nbridge=0;
        DFS_2conn( 0, -1, 1, f );
        for( i=0; i<n; i++ )
        {
            k=FindSet( f, i );
            if( belong[k]==-1 )  belong[k]=ncon++;
            belong[i]=belong[k];
        }
        return ncon;
    }
    int main( )
    {
        int i, j, k;
        while( scanf( "%d%d", &n, &m ) != EOF )
        {
            memp=0;  clr(e);
            for( k=0; k<m; k++ )    //读入边，并插入邻接表中
            {
                scanf( "%d%d", &i, &j );  i--;  j--;
                addEdge( e, i, j );  addEdge( e, j, i );
            }
            w=DoubleConnection( );  //求边双连通分量个数
            int d[N]={ 0 };         //收缩后各顶点的度数
            for( k=0; k<nbridge; k++ )
            {
                i=bridge[k][0];  j=bridge[k][1];
                d[belong[i]]++;  d[belong[j]]++;
            }
            int count=0;    //收缩后叶子结点的个数
            for( i=0; i<w; i++ )
                if( d[i]==1 )  count++;
            printf( "%d\n", (count+1)/2 );
        }
        return 0;
    }
```

8.3.3 边连通度的求解

给定一个无向连通图，如何求其边连通度$\lambda(G)$是本节要讨论的问题。$\lambda(G)$的求解也需要转换成网络最大流问题。首先，介绍弱独立轨的概念。

弱独立轨：设A、B是无向图G的两个顶点，从A到B的两条没有公共边的路径，互

称为弱独立轨。A 到 B 的弱独立轨的最大条数，记作 $P'(A, B)$。例如，在图 8.19(a)所示的无向图中，v_1 和 v_9 之间有 2 条弱独立轨，用粗线标明，这两条弱独立轨有公共顶点 v_5(当然，在图 8.19(a)中，可以选择其他边使得两条弱独立轨没有公共顶点)。

设 A、B 是无向连通图 G 的两个不相邻的顶点，最少要删除多少条边才能使得 A 和 B 不再连通？答案是 $P'(A, B)$个(证明略)。例如，在图 8.19(a)中，要使得 v_1 和 v_9 不再连通，可以在这两个顶点的两条弱独立轨上各选择一条边，如 e_1 和 e_2，删除这两条边后，v_1 和 v_9 不再连通了，如图 8.19(b)所示。注意，并不是在每条弱独立轨上任意删除一条边就可以达到目的。例如，在图 8.19(a)中，如果删除 e_1 和 e_{12}，则不会影响 v_1 和 v_9 的连通性。

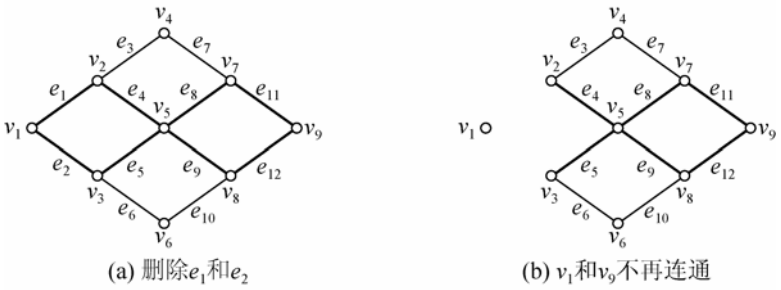


图 8.19　弱独立轨

关于无向图 G 边连通度 $\lambda(G)$与顶点间弱独立轨数目之间的关系，有如下 Menger 定理。

定理 8.4(Menger 定理) 无向图 G 的边连通度 $\lambda(G)$和顶点间**最大弱独立轨数目**之间存在如下关系式：

$$\lambda(G)=\begin{cases}|V(G)|-1 & \text{当 } G \text{ 是完全图}\\ \min\limits_{AB\notin E}\{P'(A,B)\} & \text{当 } G \text{ 不是完全图}\end{cases} \tag{8-4}$$

那么如何求不相邻的两个顶点 A、B 间的最大弱独立轨数 $P'(A, B)$呢，最少应删除图中哪些边(共 $P'(A, B)$条边)才能使得 A、B 不连通呢？可以采用网络最大流方法来求解。

求 $P'(A, B)$的方法如下。

(1) 为了求 $P'(A, B)$，需要构造一个容量网络 N。

① 原图 G 中的每条边 $e=(u, v)$变成重边，再将这两条边加上互为反向的方向，设 e' 为$<u, v>$，e''为$<v, u>$，e'和 e''的容量均为 1。

② 以 A 为源点，B 为汇点。

(2) 求从 A 到 B 的最大流 F。

(3) 流出 A 的一切弧的流量和 $\sum\limits_{e\in(A,v)} f(e)$，即为 $P'(A, B)$，流出 A 的流量为 1 的弧(A, v)组成一个割边集，在图 G 中删除这些边后，则 A 和 B 不再连通了。

有了求 $P'(A, B)$的算法基础，就可以得出 $\lambda(G)$的求解思路：首先设 $\lambda(G)$的初始值为∞；然后分析图 G 中的每一对顶点，如果顶点 A、B 不相邻，则用最大流的方法求出 $P'(A, B)$和对应的割边集；如果 $P'(A, B)$小于当前的 $\lambda(G)$，则 $\lambda(G)=P'(A, B)$，并保存其割边集。如此直至所有不相邻顶点对分析完为止，即可求出图的边连通度 $\lambda(G)$和最小割边集了。同样在具体实现时，可固定一个源点，枚举每个汇点，从而求出 $\lambda(G)$。

练　习

8.4　筑路(Road Construction)

题目来源：

CCC2007, POJ3352

题目描述：

现在，某个岛上的负责人想修复和升级岛上的道路，这些道路连接岛上不同的旅游景点。所有的道路都不会交叉，如果有交叉则通过桥梁和隧道来避免。当修路公司在修路时，这条路就不能使用了。这将导致从一个景点不能通往另一个景点，尽管修路公司只是在某个特定的时间占用了每条道路。

现在需要你来帮助他们解决这个问题。为了保证在最终的道路网络中，任何一条道路在维护当中，剩下的道路能保证任何两个景点之间都能连通，这样需要在某些景点之间新修一条路，试计算最少需要新修多少条路。

输入描述：

测试数据的第 1 行为两个正整数 n 和 r，$3 \leqslant n \leqslant 1\,000$，$2 \leqslant r \leqslant 1\,000$，$n$ 表示岛上旅游景点的数目，r 为道路的数目。景点编号为 1～n。接下来有 r 行，每行为两个整数 v 和 w，表示景点 v 和 w 之间有一条道路。道路是双向的，且任何两个景点之间最多只有一条道路。初始时，道路网络是连通的。

输出描述：

输出一个整数，表示需要新修道路的最少数目。

样例输入：

```
10 12
1 2
1 3
1 4
2 5
2 6
5 6
3 7
3 8
7 8
4 9
4 10
9 10
```

样例输出：

```
2
```

8.5　实习(Internship)

题目来源：

CYJJ’s Funny Contest #3, Having Fun in Summer, ZOJ2532

题目描述：

CIA 总部通过它的机密网络收集来自全国各地的数据。他们在光纤网络民用化前就广泛使用光纤网。但是最近由于数据量剧增，所以整个网络仍面临巨大的压力。因此，他们想采用新技术来升级网络，这种新技术可以提供数倍于现在的带宽。在实验测试阶段，他

们想升级其中一段网络，以便观察新技术能在多大程度上提升网络的性能。作为 CIA 的一名实习生，你的任务是调查哪段网络能提高 CIA 总部的带宽，假定每个城市有一定的数据量传送到 CIA 总部，而且路由算法已经优化了。

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 3 个正整数：N、M 和 L，分别表示城市的数目、中继站的数目以及网段的数目，城市的编号从 1～N，中继站的编号从 N+1～N+M，CIA 总部的编号为 0。$N + M \leqslant 100$, $L \leqslant 1\,000$。

接下来有 L 行，每行描述了一个网段，形式为：a b c，其中 a 表示网段的源点，b 为网段的目标结点，c 为带宽。a、b 和 c 均为整数，a 和 b 为有效的顶点编号(0～n+m)，c 为正整数。数据连接都有方向的。

输入文件最后一行为 $N = 0$，代表输入结束。

输出描述：

对输入文件中的每个测试数据，输出结果占一行，为符合标准的网段编号，用空格隔开，按升序列出。如果没有哪个网段符合要求，则输出空行。网段的编号从 1 开始计起。

样例输入：

```
2 1 3
1 3 2
3 0 1
2 0 1
2 1 3
1 3 1
2 3 1
3 0 2
0 0 0
```

样例输出：

```
2 3
(这里有一个空行)
```

8.6　网络(Network)

题目来源：

2008 Asia Hefei Regional Lontest Online by USTC, POJ3694

题目描述：

一个网络管理员管理一个很大的网络。网络包含了 N 台电脑和 M 对两台电脑之间的直接连接。网络中任何两台电脑要么直接连接，要么通过连续的连接间接地相连，因此数据可以在任何两台电脑之间进行传输。网络管理员发现某些直接连接对网络起着重要的作用，因为一旦这些连接中的任何一条断开了，就会导致某些电脑之间就无法传输数据。这种连接称为桥。他计划一条一条地增加一些新的连接，来消除所有的桥。

试向管理员报告添加每条新连接后网络中桥的数目。

输入描述：

输入文件包含多个测试数据。每个测试数据的第 1 行为两个整数 N 和 M，$1 \leqslant N \leqslant 100\,000$，$N-1 \leqslant M \leqslant 200\,000$；接下来有 M 行，每行为两个整数 A 和 B，$1 \leqslant A \neq B \leqslant N$，表示电脑 A 和 B 有网线直接相连，电脑的编号从 1 到 N，输入数据保证初始时，任何两台电脑都是连通的；接下来一行为一个整数 Q，$1 \leqslant Q \leqslant 1\,000$，表示管理员打算在网络中添加 Q

条新连接；接下来有 Q 行，第 i 行为两个整数 A 和 B，$1 \leqslant A \neq B \leqslant N$，表示第 i 条新连接连接的是电脑 A 和 B。

输入文件最后一行为两个 0，表示输入结束。

输出描述：

对输入文件中的每个测试数据，首先输出测试数据的序号(序号从 1 开始计起)，然后输出 Q 行，第 i 行为一个整数，表示前 i 条新连接添加进来后网络中桥的数目。每个测试数据的输出之后输出一个空行。

```
3 2                                    Case 1:
1 2                                    1
2 3                                    0
2
1 2
1 3
0 0
```

8.4 有向图强连通性的求解及应用

8.4.1 有向图强连通分量的求解算法

求解有向图强连通分量主要有 3 个算法：Tarjan 算法、Kosaraju 算法和 Gabow 算法，本节详细介绍前两个算法的思想和实现过程。

1. Tarjan 算法

Tarjan 算法是基于 DFS 算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的结点加入一个栈，回溯时可以判断栈顶到栈中的结点是否为一个强连通分量。当 dfn(u)=low(u)时，以 u 为根的搜索子树上所有结点是一个强连通分量，其中 dfn[ ]值和 low[ ]值的定义及含义详见 8.2.1 节。

Tarjan 算法的伪代码如下。

```
tarjan( u )
{
    dfn[u]=low[u]=++tmpdfn          //为结点u 设定dfn[]值和 low[]初值
    Stack.push( u )                 //将结点u 压入栈中
    for each ( u, v ) in E          //枚举每一条边
        if ( v is not visted )      //如果结点v 未被访问过
            tarjan( v )             //继续向下找
            low[u]=min( low[u], low[v] )
        else if ( v in Stack )      //如果结点v 还在栈内
            low[u]=min(low[u], dfn[v])
    if ( dfn[u]==low[u] )           //如果结点u 是强连通分量的根
        repeat
            v=Stack.pop             //将v 退栈，为该强连通分量中一个顶点
            print v
        until ( u== v )
}
```

接下来以图 8.20(a)所示的有向图为例解释 Tarjan 算法的思想和执行过程，在该有向图中，{ 1, 2, 5, 3 }为一个强连通分量，{ 4 }、{ 6 }也分别是强连通分量。

图 8.20(b)所示为从顶点 1 出发进行深度优先搜索后得到的深度优先搜索树。约定：如果某个顶点有多个未访问过的邻接顶点，按顶点序号从小到大的顺序进行选择。各顶点旁边的两个数值分别为顶点的深度优先数(dfn[ ])值和 low[ ]值。在图 8.20(b)中，虚线表示非生成树的边，其中边<5, 6>和<3, 5>为交叉边，边<5, 1>是回边(注意，8.2.1 节提到了无向图深度优先搜索生成树中非生成树的边都是回边，交叉边是不存在的)。

判断生成树的边、回边和交叉边的策略如下。

```
对顶点 u 的邻接顶点 v                    //对应的有向边为<u, v>
    如果 v 还没有访问                     //边<u, v>为生成树的边，v 为 u 的子女结点
        从顶点 v 出发进行 DFS
    否则                                  //即顶点 v 已经访问过，有向边<u, v>为非生成树的边
        如果顶点 v 还在栈中               //顶点 u 和 v 属于同一个强连通分量，v 是 u 的祖先
        <u, v>是回边
        否则                              //顶点 u 和 v 不属于同一个强连通分量
            <u, v>是交叉边
```

图 8.20(c)～8.20(f)演示了 Tarjan 算法的执行过程。在图 8.20(c)中，沿着实线箭头所指示的方向搜索到顶点 6，此时无法再前进下去了，并且因为此时 dfn[6] = low[6] = 4，所以找到了一个强连通分量。退栈到 u = = v 为止，{ 6 }为一个强连通分量。

在图 8.20(d)中，沿着虚线箭头所指示的方向回退到顶点 4，发现 dfn[4] = = low[4]，为 3，退栈后{4}为一个强连通分量。

在图 8.20(e)中，回退到顶点 2 并继续搜索到顶点 5，把顶点 5 加入栈。发现顶点 5 有到顶点 1 的有向边，顶点 1 还在栈中，所以 low[5] = 1，有向边<5, 1>为回边。顶点 6 已经出栈，所以<5,6>是交叉边，返回顶点 2，<2, 5>为生成树的边，所以 low[2] = low[5] = 1。

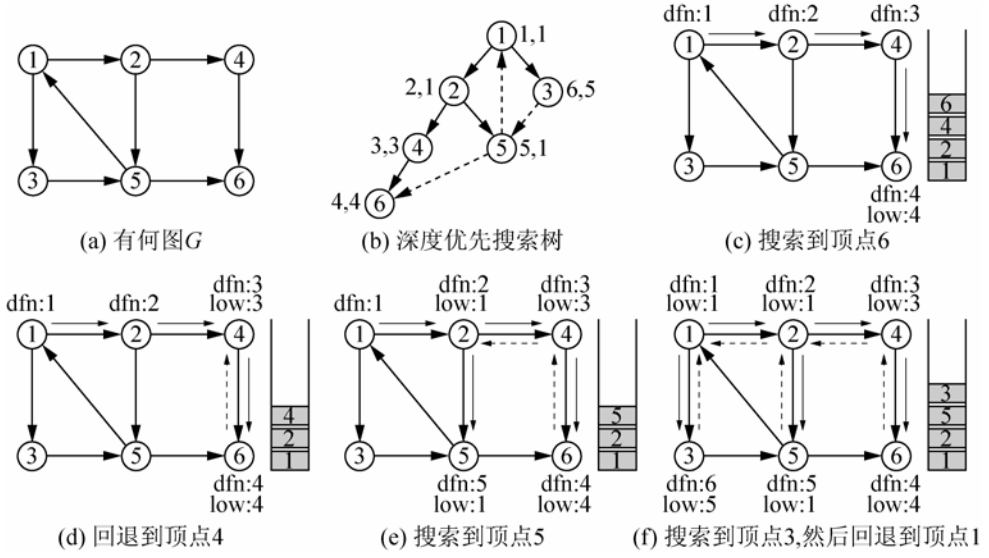


图 8.20　Tarjan 算法

在图 8.20(f)中，先回退到顶点 1，接着访问顶点 3。发现顶点 3 到顶点有一条有向边，顶点 5 已经访问过了，且 5 还在栈中，因此边<3, 5>为回边，所以 low[3] = dfn[5] = 5。返回顶点 1 后，发现 dfn[1] == low[1]，把栈中的顶点全部弹出，组成一个连通分量{ 3, 5, 2, 1 }。

至此，Tarjan 算法结束，求出了图中全部的 3 个强连通分量为{ 6 }、{ 4 }和{ 3, 5, 2, 1 }。

Tarjan 算法的时间复杂度分析：假设用邻接表存储图，在 Tarjan 算法的执行过程中，每个顶点都被访问了一次，且只进出了一次栈，每条边也只被访问了一次，所以该算法的时间复杂度为 $O(n+m)$。

2. Kosaraju 算法

Kosaraju 算法是基于对有向图 G 及其逆图 G^{T}(各边反向得到的有向图)进行两次 DFS 的方法，其时间复杂度也是 $O(n+m)$。与 Trajan 算法相比，Kosaraju 算法的思想更为直观。Kosaraju 算法的原理为：如果有向图 G 的一个子图 G'是强连通子图，那么各边反向后没有任何影响，G'内各顶点间仍然连通，G'仍然是强连通子图。但如果子图 G'是单向连通的，那么各边反向后可能某些顶点间就不连通了，因此，各边的反向处理是对非强连通块的过滤。

Kosaraju 算法的执行过程如下。

(1) 对原图 G 进行深度优先搜索，并记录每个顶点的 dfn 值。

(2) 将图 G 的各边进行反向，得到其逆图 G^{T}。

(3) 选择从当前 dfn 值最大的顶点出发，对逆图 G^{T} 进行 DFS 搜索，删除能够遍历到的顶点，这些顶点构成一个强连通分量。

(4) 如果还有顶点没有删除，继续执行第(3)步，否则算法结束。

接下来以图 8.21(a)所示的有向图 G 为例分析 Kosaraju 算法的执行过程。图 8.21(b)为正向搜索过程，搜索完毕后，得到各顶点的 dfn 值。图 8.21(c)所示为逆图 G^{T}。图 8.21(d)所示为从顶点 3 出发对逆图 G^{T} 进行 DFS 搜索，得到第 1 个强连通分量{1, 2, 5, 3}，图 8.21(e)和 8.21(f)所示为分别从顶点 4 和 6 出发进行 DFS 搜索得到另外两个强连通分量。

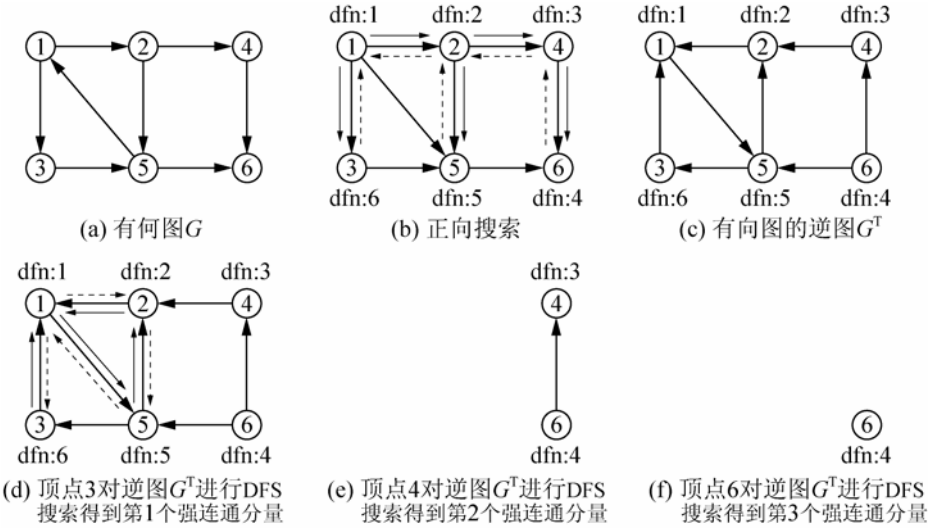


图 8.21　Kosaraju 算法

8.4.2　有向图强连通分量的应用

以下通过 4 道例题，详细讲述有向图强连通分量的求解及程序实现。

例 8.7　从 *u* 到 *v* 或从 *v* 到 *u* (Going from u to v or from v to u?)

题目来源：

POJ Monthly—2006.02.26, POJ2762

题目描述：

为了让他们的儿子变得更勇敢些，Jiajia 和 Wind 将他们带到一个大洞穴中。洞穴中有 *n* 个房间，有一些单向的通道连接某些房间。每次，Wind 选择两个房间 *x* 和 *y*，要求他们的一个儿子从一个房间走到另一个房间，这个儿子可以从 *x* 走到 *y*，也可以从 *y* 走到 *x*。Wind 保证她布置的任务是可以完成的，但她确实不知道如何判断一个任务是否可以完成。为了使 Wind 下达任务更容易些，Jiajia 决定找这样的一个洞穴，每对房间(设为 *x* 和 *y*)都是相通(可以从 *x* 走到 *y*，或者可以从 *y* 走到 *x*)的。给定一个洞穴，你能告诉 Jiajia，Wind 是否可以任意选择两个房间而不用担心这两个房间可能不相通吗？

输入描述：

输入文件的第 1 行为一个整数 *T*，表示测试数据的个数。接下来有 *T* 个测试数据。

每个测试数据的第 1 行为两个整数 *n* 和 *m*，0<*n*<1 001，*m*<6 000，分别表示洞穴中的房间数和通道数。接下来有 *m* 行，每行为两个整数 *u* 和 *v*，表示房间从房间 *u* 到 *v* 有一条单向通道。

输出描述：

对每个测试数据，如果洞穴具备题目中提到的属性，输出“Yes”，否则输出“No”。

样例输入：

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

样例输出：

```
Yes
```

分析：

本题虽然求解的是单连通性，但首先要转换成强连通分量的求解。这是因为，强连通分量中的顶点间存在双向的路径，因此可以将每个强连通分量**收缩**成一个新的顶点。在有向图的处理中经常需要将**强连通分量收缩成一个顶点**。

以样例输入中的测试数据为例，图 8.22(a)描述了该测试数据。在图 8.22(b)中，将两个强连通分量各收缩成 1 个新的顶点。

强连通分量收缩后，再求其拓扑排序。假设求得的拓扑序存储在 topo[*MAX*]中，topo[*i*]与 topo[*i*+1]存在边连通(*i* 到 *i*+1 或 *i*+1 到 *i*)，则一定有 *i* 到 *i*+1 的边。而如果每个 topo[*i*]与 topo[*i*+1]都存在边连通(即有 *i* 到 *i*+1 的边)时，topo[*i*]到任意 topo[*j*]便都有边连通。图 8.22(c)是进行拓扑排序的结果。

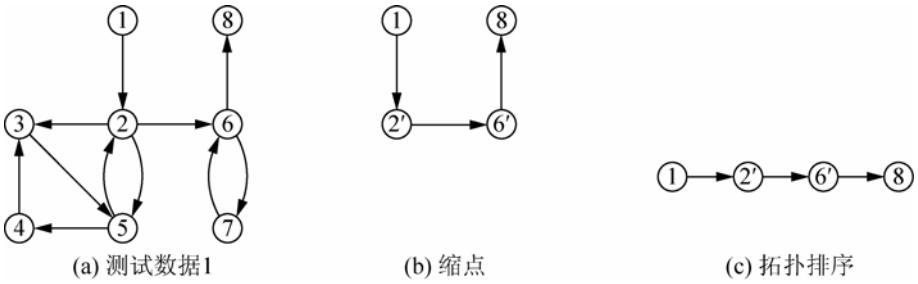


图 8.22　从 *u* 到 *v* 或从 *v* 到 *u*：测试数据

代码如下：

```
#define MAX 1002
using namespace std;
struct Node
{
    int e, val;
}item;
vector< vector<Node> > g(MAX);
stack <int> st;
int ord[MAX];       //ord[i]:结点 i 的访问次序
int low[MAX];       //low[i]:与 i 连接的结点最先访问的次序(最高的祖先)
int id[MAX];        //id[i]:记录 i 结点属于第几个连通分量
int cnt, scnt, n;   //cnt 记录访问次序，scnt 记录强连通数; n 记录结点数
int mat2[MAX][MAX];
int n2;
//Tarjan 算法：计算强连通，scnt 记录强连通数，id[i]记录 i 结点属于第几个连通分量
void Tarjan( int e )
{
    int t, i;
    int min=low[e]=ord[e]=cnt++;
    st.push( e );
    for( i=0; i<g[e].size(); i++ )
    {
        t=g[e][i].e;
        if( ord[t]==-1 )  Tarjan( t );
        if( low[t]<min )  min=low[t];
    }
    if( min<low[e] )    //有回边
    {
        low[e]=min;  return;
    }
    do  //在同一颗树(子树有回边)属于同一连通分量
```

```
        {
            id[t=st.top()]=scnt;  low[t]=n;
            st.pop( );
        }while( t!=e );
        scnt++;
    }
    void Search( int n )    //搜索强连通分量
    {
        int i;
        memset( ord, -1, sizeof(ord) );
        cnt= 0;  scnt= 0;
        for( i=0; i<n; i++ )
        if( ord[i]==-1 )  Tarjan(i);
    }
    //计算核心 DAG,得到 scnt 记录强连通数，id[i]记录 i 结点属于第几个连通分量。
    //返回核心 DAG 的结点数 n2,邻接矩阵 mat2[MAX][MAX]
    void base_vertex( )
    {
        int i, j, t;
        Search( n );    //调用求强连通分量
        n2=scnt;
        memset( mat2, 0, sizeof(mat2) );
        for( i=0; i<n; i++ )
        {
            for( j=0; j<g[i].size(); j++ )
            {
                t=g[i][j].e;  mat2[id[i]][id[t]]=1;
            }
        }
    }
    //拓扑排序：如果无法完成排序，返回 0，否则返回 1，ropo 返回有序点列
    //传入图的大小 n 和邻接阵 mat，不相邻点边权为 0
    int toposort( int n, int mat[][MAX], int *topo )
    {
        int d[MAX], i, j, k;
        for( i=0; i<n; i++ )    //初始化
        {
            d[i]=0;
            for( j=0; j<n; j++ )  d[i] += mat[j][i];    //入度数
        }
        for( k=0; k<n; k++ )
        {
            for( i=0; d[i] && i<n; i++ ) ;
            if( i==n )  return 0;   //无法完成拓扑排序,没有入度为 0 的顶点
            d[i]=-1;                //标记已经排序完
            for( j=0; j<n; j++ )    //删边(即减入度)
            d[j]-=mat[i][j];
            topo[k]=i;
        }
```

```
    return 1;   //完成
}
int main( )
{
    int m, i, s, e, cas;
    int topo[MAX];
    scanf( "%d", &cas );
    while( cas-- )
    {
        scanf( "%d%d", &n, &m );   //读入数据,初始化
        for( i=0; i<n; i++ )  g[i].clear( );
        for( i=0; i<m; i++ )
        {
            scanf( "%d%d", &s, &e );
            item.e=e - 1;  item.val=1;
            g[s-1].push_back(item);
        }
        base_vertex( );
        toposort( n2, mat2, topo );//拓扑排序
        int flag=1;
        for( i=0; i<n2-1; i++ )
        {
            if( !mat2[topo[i]][topo[i+1]] )
            {
                flag=0;  break;
            }
        }
        if( flag )  printf( "Yes\n" );
        else  printf( "No\n" );
    }
    return  0 ;
}
```

例 8.8　受牛仰慕的牛(Popular Cows)

题目来源：

USACO 2003 Fall, POJ2186

题目描述：

每头奶牛都梦想着成为牧群中最受奶牛仰慕的奶牛。在牧群中，有 N 头奶牛，$1 \leqslant N \leqslant 10\,000$，给定 M 对$(1 \leqslant M \leqslant 50\,000)$有序对$(A, B)$，表示 A 仰慕 B。由于仰慕关系具有传递性，也就是说，如果 A 仰慕 B，B 仰慕 C，则 A 也仰慕 C，即使在给定的 M 对关系中并没有(A, C)。试计算牧群中受每头奶牛仰慕的奶牛数量。

输入描述：

每个测试数据的格式如下。

(1) 第 1 行：两个用空格隔开的整数 N 和 M。

(2) 第 2～M+1 行：两个用空格隔开的整数 A 和 B，表示 A 仰慕 B。

输出描述：

输出一行，为一个整数，表示受每头奶牛仰慕的奶牛数目。

样例输入：

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

样例输出：

```
1
0
```

分析：

因为仰慕关系具有传递性，因此在同一个强连通分量中的顶点：如果强连通分量中一头牛 *A* 受强连通分量外另一头牛 *B* 的仰慕，则该强连通分量中的每头牛都受 *B* 的仰慕；如果强连通分量中一头牛 *A* 仰慕强连通分量外的另一头牛 *B*，则强连通分量中的每一头牛都仰慕 *B*。因此，本题可以将强连通分量缩为一个顶点，并构造新图。最后作一次扫描，统计出度为 0 的顶点个数，如果正好为 1，则说明该顶点(可能是一个新构造的顶点，即对应一个强连通分量)能被其他所有顶点走到，即该强连通分量为所求答案，输出它的顶点个数即可。

例如，题目中给出的两个测试数据如图 8.23 所示。在图 8.23(a)中，缩点后只有一个出度为 0 的顶点，因此只有 1 头牛受其他所有牛的仰慕。在图 8.23(b)中，缩点后有两个出度为 0 的顶点，因此没有哪头牛受其他所有牛的仰慕。

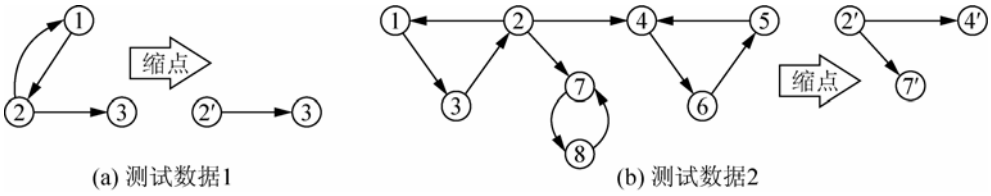


图 8.23　受牛仰慕的牛：测试数据

代码如下：

```
#define G_size 100000
#define V_size 11000
typedef struct Graph
{
    int id, next;
}Graph;
typedef struct Edge
```

```
{
    int s, e;
}Edge;
Edge E[G_size];
Graph GA[G_size], GT[G_size];
int N, M;
int G_end;
int order[V_size], id[V_size], vis[V_size], in[V_size];
int cnt, scnt, pos;
void Build( int s, int e ) //建立原图和逆图
{
    int p=s;
    while( GA[p].next )
        p=GA[p].next;
    GA[G_end].id=e; GA[p].next=G_end;
    p=e;
    while( GT[p].next )
        p=GT[p].next;
    GT[G_end].id=s; GT[p].next=G_end;
    G_end++;
}
void DFST( int x )          //DFS 搜索逆图
{
    int p, q;
    vis[x]=1;
    p=GT[x].next;
    while( p )
    {
        q=GT[p].id;
        if( !vis[q] )
            DFST( q );
        p=GT[p].next;
    }
    order[cnt++]=x;     //记录顺序
}
void DFSA( int x )      //DFS 搜索原图
{
    int p, q;
    vis[x]=1;
    id[x]=cnt;
    p=GA[x].next;
    while( p )
    {
        q=GA[p].id;
        if( !vis[q] )
            DFSA( q );
        p=GA[p].next;
    }
}
void Kosaraju( )        //Kosaraju 算法搜索强连通分量
{
```

```
    int s, e, i;
    memset( GA, 0, sizeof(GA) ); memset( GT, 0, sizeof(GT) ); //初始化
    memset( E, 0, sizeof(E) );
    G_end=N+1;
    for( i=0; i<M; i++ )
    {
        scanf( "%d %d", &s, &e );
        E[i].s=s-1; E[i].e=e-1;
        Build( s-1, e-1 );
    }
    //搜索原图
    memset( vis, 0, sizeof(vis) );
    for( i=0; i<N; i++ )
    {
        if( !vis[i] )
            DFST( i );
    }
    //搜索逆图
    memset( vis, 0, sizeof(vis) );
    cnt=0;
    for( i=N-1; i>=0; i-- )
    {
        if( !vis[order[i]] )
        {
            DFSA( order[i] );
            cnt++;
        }
    }
    //缩点
    for( i=0; i<M; i++ )
    {
        s=id[E[i].s]; e=id[E[i].e];
        if( s!=e )  in[s]++;
    }
    //统计出度为0的顶点个数
    scnt=cnt; cnt=0;
    for( i=0; i<scnt; i++ )
    {
        if( in[i]==0 )
        {
            pos=i; cnt++;
        }
    }
}
int main( )
{
    int i;
    while( scanf("%d %d", &N, &M)!=EOF )
        Kosaraju( );    // Kosaraju搜索强连通分量,并进行缩点
    if( cnt!=1 ) printf( "0\n" );
    else
```

```
        {
            //统计强连通分量的顶点个数
            cnt=0;
            for( i=0; i<N; i++ )
            {
                if( in[id[i]]==pos )  cnt++;
            }
            printf( "%d\n", cnt );
        }
        return 0;
    }
```

例 8.9　图的底部(The Bottom of a Graph)

题目来源：

University of Ulm Local Contest 2003, ZOJ1979, POJ2553

题目描述：

使用图论中以下标准定义。设 V 为一个非空有限集，它的元素称为顶点，E 为笛卡儿积 $V\times V$ 的子集，它的元素称为边(有方向)，这样 $G=(V, E)$称为一个有向图。

设 n 为一个正整数，$p=(e_1, \cdots, e_n)$是一个包含 n 条边的边序列，$e_i\in E$，其中 $e_i=(v_i, v_{i+1})$，这些顶点来自一个顶点序列$(v_1, \cdots, v_{n+1})$，则称 p 为一条从顶点 v_1 到 v_{n+1} 的路径，并称从 v_1 到 v_{n+1} 是可达的，记为$(v_1\rightarrow v_{n+1})$。

这里，做一些新的定义。设 v 是图 $G=(V, E)$的一个顶点，对图 G 中每个顶点 w，如果 v 可达 w，那么 w 也可达 v，则称 v 为汇点。图的底部为图的子集，子集中所有的顶点都是汇点，即 $\text{bottom}(G)=\{v\in V \mid \forall w\in V:(v\rightarrow w)\text{且}(w\rightarrow v)\}$。对于一个给定的图，求其底部。

输入描述：

输入文件中包含多个测试数据，每个测试数据描绘了一个有向图 G。每个测试数据的第 1 行为两个整数 v 和 e，v 表示图 $G=(V, E)$中顶点数目，顶点序号为 1～v，$1\leqslant v\leqslant 5\,000$，$e$ 表示图 G 中有 e 条边；第 2 行为 e 对顶点，其格式为 $v_1, w_1, \cdots v_e, w_e$，表示$(v_i, w_i)\in E$，除了这些整数对表示的边外，没有其他边。

输入文件最后一行为 0，代表输入结束。

输出描述：

对输入文件中的每个测试数据，输出一行，为求得的图的底部：输出图的底部中各个顶点的序号。如果图的底部为空，则输出一个空行。

样例输入：

```
7 10
2 1 2 3 2 5 2 6 3 5 4 3 5 2 5 4 6 7 7 6
3 3
1 3 2 3 3 1
0
```

样例输出：

```
1 6 7
1 3
```

分析：

本题要求解的是有向图中满足“自己可达的顶点都能到达自己”的顶点个数。与例 8.8 类似，强连通分量中如果有一个顶点是汇点，则所有顶点都是汇点。另外，如果强连通分量中某个顶点还能到达分量外的顶点，则该连通分量不满足要求。例如，图 8.24(a)所示测

试数据 1 中，顶点 2 所在的连通分量中，顶点 2 还能到达顶点 1 和 6，但顶点 1 和 6 都不能到达 2(实际上，如果顶点 1 或 6 也能到达顶点 2，那顶点 1 或 6 将属于顶点 2 所在的强连通分量)，因此不满足汇点的定义。

因此，本题要求的是将强连通分量缩点后所构造的新图中出度为 0 的顶点个数，如果是强连通分量收缩得到的新顶点，则连通分量里的所有顶点都满足定义。图 8.24(a)所示的测试数据 1 中，缩点后顶点 1 和顶点 6′满足要求，其中顶点 6′所强连通分量收缩所形成的新顶点，因此，在该测试数据中，顶点 1、6、7 是汇点。在图 8.24(b)所示的测试数据 2 中，顶点 1、3 是汇点。

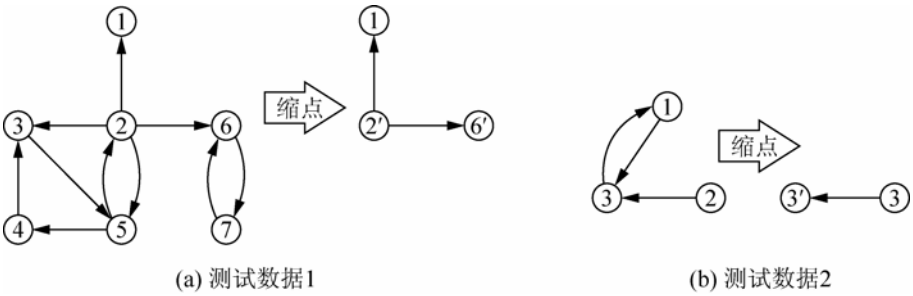


图 8.24　图的底部：测试数据

代码如下：

```
#define N 5005
#define M 50000
struct Edge         //邻接表
{
    int j;
    struct Edge* next;
};
Edge mem[M];
int memp;
bool flag;          //搜索方向标志,1 为正向,0 为反向
Edge* E[N];
//con[i]表示结点 i 所属强连通分量的编号, ncon 为极大强连通分支的个数
int con[N], ncon;
int n, m;
int out[N];         //出度
void addEdge( Edge* E[], int i, int j )    //插入邻接表
{
    Edge* p=&mem[memp++];
    p->j=j;
    p->next=E[i];
    E[i]=p;
}
void DFS( Edge* E[], int i, int mark[], int f[], int* nf )//DFS 搜索
{
    int j;
    Edge* p;
    if( mark[i] ) return;
```

```
        mark[i]=1;
        if( !flag ) f[i]=*nf;       //反向搜索，获取连通分量编号
        for( p=E[i]; p!=NULL; p=p->next )
        {
            j=p->j;
            DFS( E, j, mark, f, nf );
        }
        if( flag ) f[( *nf )++]=i; //正向搜索，获取时间戳
    }
    int Kosaraju( Edge* E[], int n, int con[] )
    {
        int i, mark[N], ncon;
        int time[N], ntime;             //time[i]表示时间戳为 i 的结点
        Edge *p, *RE[N];                //反向边
        //构造反向边邻接表
        memset( RE,0,sizeof( RE ) );
        for( i=0; i<n; i++ )
        {
            for( p=E[i]; p!=NULL; p=p->next )
                addEdge( RE, p->j, i );
        }
        //正向 DFS，获得时间戳
        flag=1;
        memset( mark,0,sizeof( mark ) );
        memset( time,0,sizeof( time ) );
        ntime=0;
        for( i=0; i<n; i++ )
        {
            if( !mark[i] )
                DFS( E, i, mark, time, &ntime );
        }
        //反向 DFS，获得强连通分量
        flag=0;
        memset( mark,0,sizeof( mark ) );
        memset( con,0,sizeof( con ) );
        ncon=0;
        for( i=n-1; i>=0; i-- )
        {
            if( !mark[time[i]] )
            {
                DFS( RE, time[i], mark, con, &ncon );
                ncon++;
            }
        }
        return ncon;
    }
    int main( )
    {
        int i, j, k, x, y;
        Edge *p;
        while( scanf( "%d",&n ) != EOF && n )
```

```
    {
        //初始化
        memp=0;
        memset( E,0,sizeof( E ) );
        scanf( "%d", &m );
        for( k=0; k<m; k++ )
        {
            scanf( "%d%d", &i, &j );
            addEdge( E,--i,--j );
        }
        ncon=Kosaraju( E, n, con );//计算强连通分量
        //计算出度
        memset( out,0,sizeof( out ) );
        for( i=0; i<n; i++ )
        {
            x=con[i];
            for( p=E[i]; p!=NULL; p=p->next )
            {
                y=con[p->j];
                if( x!=y ) out[x]++;
            }
        }
        //输出出度为0的顶点
        for( i=0, j=0; i<n; i++ )
        {
            k=con[i];
            if( out[k]==0 )
            {
                if( j )  printf( " " );
                j=1;
                printf( "%d", i+1 );
            }
        }
        printf( "\n" );
    }
    return 0;
}
```

例 8.10　学校的网络(Network of Schools)

题目来源：

IOI 1996, POJ1236

题目描述：

有一些学校连接到一个计算机网络。这些学校之间达成了一个协议：每个学校维护着一个学校列表，它向学校列表中的学校发布软件。注意，如果学校 *B* 在学校 *A* 的列表中，则 *A* 不一定在 *B* 的列表中。

任务 *A*：计算为使得每个学校都能通过网络收到软件，至少需要准备多少份软件拷贝。

任务 *B*：考虑一个更长远的任务，想确保给任意一个学校发放一个新的软件拷贝，该软件拷贝能发布到网络中的每个学校。为了达到这个目标，必须在列表中增加新成员。计

算需要添加新成员的最小数目。

输入描述：

测试数据的第 1 行为一个整数 N，表示网络中的学校数目，$2 \leqslant N \leqslant 100$，学校的编号为 1～$N$。接下来有 N 行，第 i+1 行描述了第 i 个学校的接收学校列表，每个列表最后为一个 0，如果列表为空，则只有最后的一个 0。

输出描述：

输出两行，第 1 行为一个整数，为任务 A 的解；第 2 行为任务 B 的解。

样例输入：

```
5
2 4 3 0
4 5 0
0
0
1 0
```

样例输出：

```
1
2
```

分析：

如果原网络 G 中存在强连通分量 G'，从 G' 中任意一个顶点出发，都能遍历到 G' 中的每个顶点。而从 G' 以外的顶点进入 G'，都可以通过 G' 中任意一个出度大于等于 1 的顶点离开 G'。这样，可以将网络中的强连通分量看成一个点，即缩点，这样图就得到了简化。寻找强连通分量的方法用 Kosaraju 算法。例如，样例输入中测试数据所描述的有向图如图 8.25(a)所示，进行缩点后，得到图 8.25(b)所示的有向图。

新生成的有向图是一个有向无环图。如果在该有向图中从某个顶点 u 出发有多条路径可以到达顶点 v，则还需要进一步处理：将这些路径都收缩成一条从 u 到 v 的边<u, v>，并去掉原来路径上的中间顶点。例如，如果在图 8.25(a)中增加一条边<3, 4>，得到如图 8.25(c)所示的有向图，缩点后得到图 8.25(d)。在图 8.25(d)中，从顶点 1′出发有两条路径可以到达顶点 4，因此，将这两条路径收缩成边<1′, 4>，并删除顶点 3，如图 8.25(e)所示。完成了这两步工作，最后得到的图将是一个森林。

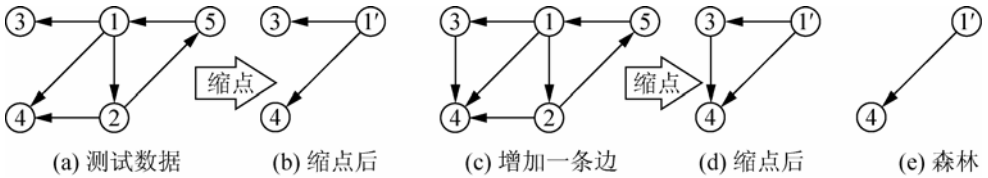


图 8.25　学校的网络：测试数据

从森林里任意一个点出发都可以遍历它的子树，因此，为了遍历所有的点，所有点的公共祖先就是任务 A 所求点，这些点的特征应该是入度为 0。

任务 B 的要求是添加最少边，使图完全连通，现在转化成将一个森林变化成完全连通，很显然，只需要将一棵树的叶子结点轮流着连接到“相邻的”树形结构的祖先结点，具体需要连接的数量视总的祖先结点和叶子结点的较大值，叶子节点的特征是出度为 0。必须注意的是有一种特殊情况，如果采取记录度的方式判断，需要注意当最终生成图中只有一个点(也就是说原图中只存在一个强连通分量)，不需要将出度为零的点与入度为零的点相

连(那样得到答案 2 为 1)，正确答案是 0。

代码如下：

```
#define N 101
struct Edge         //邻接表
{
    int dest;
    Edge *next;
}*GA[N], *GT[N], *G[N];
int used[N], path[N], part[N], mark[N][N], m, n;
int in[N], out[N];  //入度;出度
void addedge( Edge *T[],int i, int j ) //插入邻接表
{
    Edge *L;
    L=new Edge;
    L->dest=j; L->next=T[i];
    T[i]=L;
}
void DFSA( int s )  //DFS 搜索原图
{
    Edge *l;
    if( !used[s] )
    {
        used[s]=1;
        for( l=GA[s]; l!=NULL; l=l->next )
        DFSA( l->dest );
        path[0]++;
        path[path[0]]=s;
    }
}
void DFST( int s )  //DFS 搜索逆图
{
    Edge *l;
    if( !used[s] )
    {
        used[s]=1;
        for( l=GT[s]; l!=NULL; l=l->next )
            DFST( l->dest );
        part[s]=part[0];
    }
}
void Kosaraju( )    //Kosaraju 搜索强连通分量
{
    int i, j, k;
    Edge *L;
    //搜索原图
    memset( used, 0, sizeof( used ) );
    path[0]=part[0]=0;
    for( i=1; i<=n; i++ )
        DFSA( i );
```

```
        //搜索逆图
        memset( used, 0, sizeof( used ) );
        for( i=n; i>=1; i-- )
        {
            if( !used[path[i]] )
            {
                part[0]++; DFST(path[i]);
            }
        }
        //缩点
        memset( mark, 0, sizeof( mark ) );
        for( k=1; k<=n; k++ )
        {
            for( L=GA[k], i=part[k]; L != NULL; L=L->next )
            {
                j=part[L->dest];
                if( i!=j&&!mark[i][j] )
                {
                    mark[i][j]=1;
                    addedge( G, i, j );
                }
            }
        }
    }
    int main( )
    {
        int i, j, A, B;
        Edge *L;
        //读入数据
        scanf( "%d", &n );
        for( i=1; i<=n; i++ )  GA[i]=GT[i]=G[i]=NULL;
        for( i=1; i<=n; i++ )
        {
            while( scanf( "%d", &j ) && j )
            {
                addedge(GA,i,j);  addedge(GT,j,i);
            }
        }
        Kosaraju( );    //Kosaraju 搜索强连通分量
        //计算顶点的入度和出度
        memset( in, 0, sizeof( in ) );  memset( out, 0, sizeof( out ) );
        for( m=part[0], i=1; i<=m; i++ )
        {
            for( L=G[i]; L!=NULL; L=L->next )
            {
                out[i]++;  in[L->dest]++;
            }
        }
        //统计入度为 0 和出度为 0 的顶点个数,即任务 A 和 B 的解
        for( A=B=0, i=1; i <= m; i++ )
        {
```

```
            if( !in[i] )  A++;
            if( !out[i] )  B++;
        }
        B=A>B?A:B;
        if( m==1 ) B=0;
        printf( "%d\n%d\n", A, B );
        return 0;
    }
```

练　　习

8.7　圣诞老人(Father Christmas Flymouse)

题目来源：

POJ Monthly—2006.12.31, POJ3160

题目描述：

从武汉大学 ACM 集训队退役后，Flymouse 做起了志愿者，帮助集训队做一些琐碎的事情，比如打扫集训用的机房等。当圣诞节来临时，Flymouse 打扮成圣诞老人给集训队员发放礼物。集训队员住在校园宿舍的不同寝室里。为了节省体力，Flymouse 决定从某个寝室出发，沿着一些有向路一个接一个地访问寝室并顺便发放礼物，直到所有集训队员的寝室走遍为止。

以前 Flymouse 在集训队的日子里，他给其他队员留下了不同的印象。他们中的一些人，比如 Li Zhixu，对 Flymouse 的印象特别好，将会为他的好心唱赞歌；而其他一些人，比如 Snoopy，将不会宽恕 Flymouse 的懒惰。Flymouse 可以用一种安慰指数来量化他听了这些队员的话语后心情是好还是坏(正数表示是心情好，负数表示心情坏)。当到达一个寝室时，他可以选择进入寝室、发放礼物、倾听接收礼物的队员的话语，或者默默地绕开这个寝室。他可能会多次经过一个寝室，但决不会第 2 次进入该寝室。他想知道在他发放礼物的整个旅程，他收获安慰指数的最大数量。

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，$1<N\leqslant 30\,000$，$1<M\leqslant 150\,000$，分别表示有 N 名队员住在 N 个不同的寝室里，有 M 条有向路；接下来有 N 行，每行有 1 个整数，第 i 个整数表示第 i 个寝室队员话语的安慰指数。接下来有 M 行，每行为两个整数 i 和 j，表示第 i 个寝室到第 j 个寝室有一条有向路。测试数据一直到文件尾。

输出描述：

对每个测试数据，输出 Flymouse 收获安慰指数的最大数量。

样例输入：

```
2 2
14
21
0 1
1 0
```

样例输出：

```
35
```

8.8 国王的要求(King's Quest)

题目来源：

Northeastern Europe 2003, ZOJ2470, POJ1904

题目描述：

曾经有一个国王，他有 N 个儿子。同时在这个王国中有 N 个漂亮的女孩，国王知道他的每个儿子喜欢哪个女孩。国王的儿子都很年轻，可能会出现一个儿子喜欢多个女孩。

国王要求他的谋士为他的每个儿子挑一个他喜欢的女孩，让他的儿子娶这个女孩。谋士做到了，对国王的每个儿子，谋士为他选择了一个女孩，他喜欢这个女孩，并且将娶这个女孩。当然了，每个女孩只能嫁给国王的一个儿子。

然而，国王看完选择名单后，说道：我喜欢你安排的名单，但不是十分满意，我需要知道我的每个儿子可以和哪些女孩结婚，当然只要他和某个女孩结婚了，其他每个儿子仍然能选择到他喜欢的女孩结婚。

试帮助谋士解决这个问题。

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，表示国王的儿子数目，$1 \leqslant N \leqslant 2\,000$；接下来有 N 行，描述了每个儿子喜欢的女孩名单：首先是一个整数 K_i，表示第 i 个儿子喜欢的女孩数目，然后是 K_i 个不同的整数，表示女孩的序号。女孩的序号范围是 $1 \sim N$。K_i 的总和不超过 200 000；最后一行是谋士做出的原始安排名单——N 个不同的整数：国王每个儿子与名单中对应的女孩结婚。输入数据保证名单是正确的，也就是说，每个儿子的确是喜欢名单中他将娶的女孩。

输出描述：

对每个测试数据，输出 N 行。对国王的每个儿子，首先输出 L_i，表示第 i 个儿子喜欢并且可以结婚的女孩数目，当第 i 个儿子结婚后，其他每个儿子都可以选择到女孩结婚。然后是 L_i 个不同的整数，表示这些女孩的编号，按非减顺序排列。每个测试数据的输出之后输出一个空行。

样例输入：

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

样例输出：

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

8.9 瞬间转移(Instantaneous Transference)

题目描述：

South Central China 2008 hosted by NUDT, POJ3592

题目描述：

在很久以前玩的红警游戏中，可以对游戏中的物体执行一种魔法功能，称为瞬间转移。当一种物体使用这种功能时，它可以瞬间移动到指定位置，不管有多远。

现在有一个矿区，你驾驶一辆采矿的卡车。你的任务是采集到最大数量的矿。矿区是

一个长方形的区域，包含 $n \times m$ 个小方格，有些方格中藏有矿石，其他方格中没有。矿石采完后不能再生。

采矿车的起始位置为区域的西北角，它只能移动到东面或南面相邻方格，而不能移动到北面或西面的相邻方格。其中有些方格有魔法功能，能将矿车瞬间移动到指定方格。然而，作为矿车的驾驶员，你可以决定是否使用这种魔法功能。如果某个方格有魔法功能，则这个功能永远不会消失，你可以在到达任意一个此类方格时使用魔法功能。

输入描述：

输入文件的第 1 行为一个整数 T，表示测试数据数目。

每个测试数据的第 1 行为两个整数 N 和 M，$2 \leqslant N, M \leqslant 40$。接下来有 N 行描述了矿区的地图，每行为包含 M 个字符的字符串，每个字符可能为数字字符 $X(0 \leqslant X \leqslant 9)$、'*'或'#'字符。整数字符 X 表示该方格有 X 单位的矿石，你的采矿车可以全部采集，'*'字符表示该方格有魔法功能，'#'字符表示该方格布满了岩石，采矿车不能通过。假定起始方格不会是'#'。假设地图中有 K 个'*'字符，则接下来有 K 行，描述了每个'*'将采矿车移动到指定的方格，'*'的顺序为从北到南、从西到东。(起点在西北角，坐标方向为南-北，西-东，方格的坐标从 0 开始计起。)

输出描述：

对每个测试数据，输出你可以采到的最多矿石。

样例输入：

```
1
2 2
11
1*
0 0
```

样例输出：

```
3
```