无向图的连通性

ver2025.7.10

在图中搜索连通分量

给定一个无向图 G 有 n 个节点和 m 条边。我们需要在其中找到所有的连通分量,即若干个顶点组,使得在同一组内的每个顶点都可以从另一个顶点到达,而不同组之间没有路径。

解决问题的算法

我们可以使用深度优先搜索(Depth First Search)或宽度优先搜索(Breadth First Search)来解决这个问题。

事实上,单次调用 $DFS(u)$ (或 $BFS(u)$) 只会访问与 u 直接或间接相连的顶点，这一点可以被利用来在无向图中找到连通分量并对分量进行计数。

我们将若干次的DFS：第一轮从第一个节点开始,将第一个连通分量中的所有节点遍历(找到)。然后我们在剩余的节点中找到第一个未访问的节点,并在它上面运行深度优先搜索,从而找到第二个连通分量。如此继续,直到所有节点都被访问过。

这个算法的总运行时间为 $O(n+m)$:事实上,该算法不会在同一个顶点上运行两次,这意味着每条边将被看到两次(一次在一端,另一次在另一端)。

代码实现

```cpp
int n;
vector<vector<int>> adj;
vector<bool> vis;
vector<int> comp; //一个连通分量的顶点集

void dfs(int u) {
    vis[u] = true ;
    comp.push_back(u);
    for (int v : adj[u]) {
        if (!vis[v])
            dfs(v);
    }
}
void find_comps() {
    fill(vis.begin(), vis.end(), 0);
    for (int i = 0; i < n; i++) {
        if (!vis[i]) {
            comp.clear();
            dfs(i);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

$find\_comps()$,它用于查找和输出连通分量。

图是以邻接表表示存储的,即 $adj[v]$ 包含一个与顶点 v 相邻的顶点列表。

数组 $comp$ 包含当前连通分量中节点的列表。

代码的迭代实现

通常来说,深度递归函数是不好的。每个递归调用都需要在栈中占用一点内存,而程序默认情况下只有有限的栈空间。因此,当你在具有数百万个节点的连通图上执行递归DFS时,可能会遇到栈溢出。

通过手动维护一个栈数据结构,可以将递归程序翻译成迭代程序。由于这个数据结构是在堆上分配的,因此不会发生栈溢出。

```
int n;
vector<vector<int>> adj;
vector<bool> vis;
vector<int> comp;
void dfs_iter(int s) {
    stack<int> st;
    st.push(s);
    while (!st.empty()) {
        int u = st.top();  st.pop();
        if (!vis[u]) {
            vis[u] = true;
            comp.push_back(u);
            for (int i = adj[u].size() - 1; i >= 0; i--) { //逆序加入邻接表中的节点
                st.push(adj[u][i]);
            }
        }
    }
}
void find_comps() {
    fill(vis.begin(), vis.end(), 0);
    for (int v = 0; v < n ; v++) {
        if (!vis[v]) {
            comp.clear();
            dfs_iter(v);
            cout << "Component:" ;
            for (int u : comp)
                cout << ' ' << u;
            cout << endl ;
        }
    }
}
```

上面的代码实现了一个迭代的深度优先搜索 dfs_iter。它使用一个栈来模拟递归调用。主要步骤如下:

1. 将起始节点 s 推入栈中。
2. 循环执行以下操作,直到栈为空:
    - 从栈中弹出一个节点 u。
    - 如果 u 尚未访问过,将其添加到连通分量 $comp$ 中,并将 u 的所有邻居节点按相反顺序推入栈中。
3. 返回连通分量 $comp$。

这种迭代实现的时间复杂度与递归实现相同,都是 $O(n+m)$,其中 n 是节点数,而 m 是边数。但是,它不会遇到栈溢出的问题,因为所有数据结构都是在堆上分配的。

通过DFS生成树检查图边属性

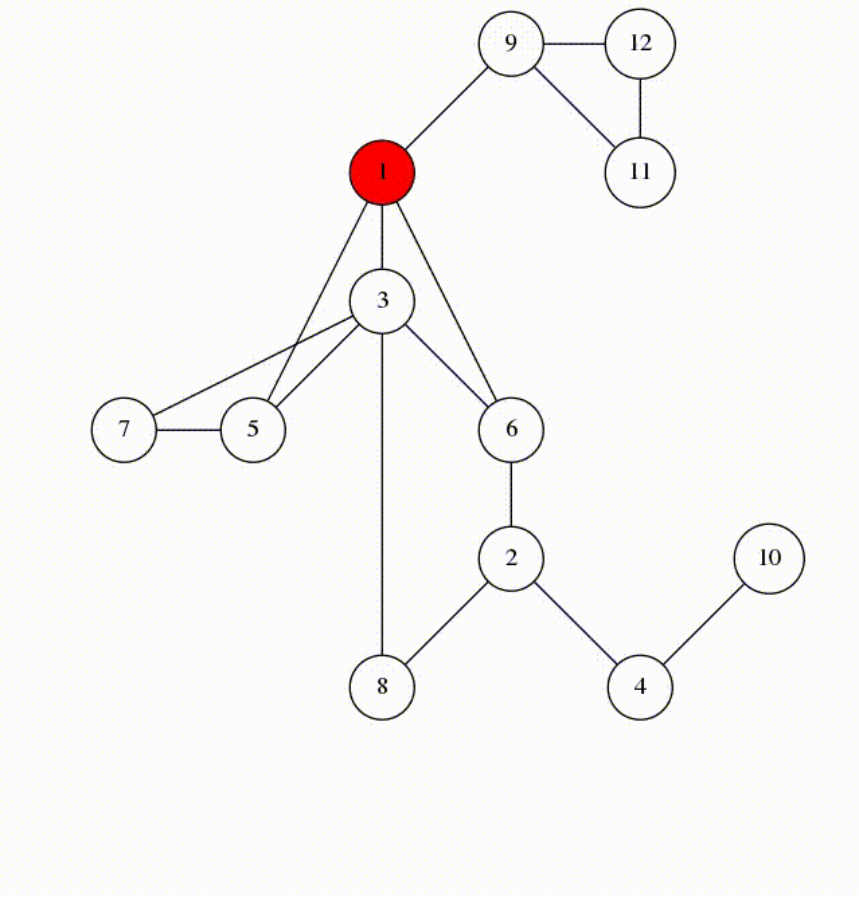
在图论问题中通过深度优先搜索遍历形成的生成树来检查边的属性，是一种非常重要且普遍的技术。

在图的一个连通分量上运行DFS可以形成DFS生成树(如果图有多个连通分量并且每个连通分量都运行DFS,则形成生成森林)。

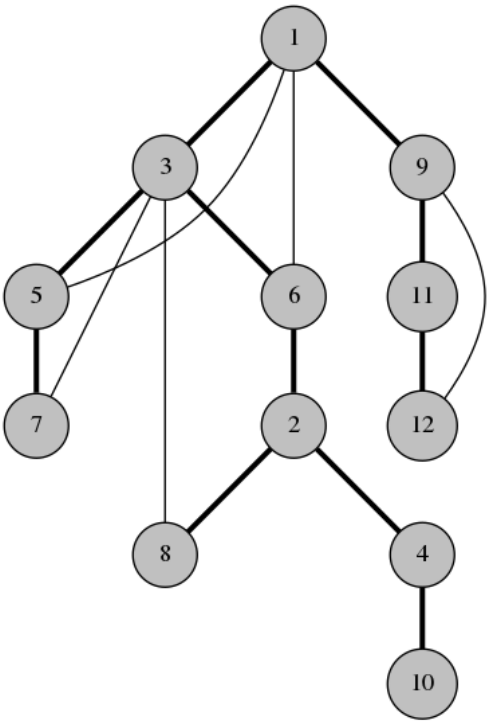
DFS的简要代码为：

```
void dfs(int u){
    vis[u] = true;
    for(auto v : adj[u]) {
        if(vis[v]) continue;
        标记边 (u, v)
        dfs(v);
    }
}
```

下面的动画演示了在图上执行 `dfs(1)` 的过程。在代码的第5行执行的标记边 (u, v)操作，将形成一棵生成树，这些边被称为 `树边 (tree edge)` 或者 `生成树边(span edge)`，而其它边被称为 `反向边 (back edge)`。这个标记边 (u, v)的操作，可以用 `fa[v] = u` 来简单地记录这棵生成树。



再一次画出形成的DFS生成树如下：



观察1. 图中的反向边(back-edges)都将一个顶点与生成树中的祖先相连。这就是为什么DFS树如此有用。

解释

假设存在一条边 (u, v),并且不失一般性,深度优先遍历先到达 u 而 v 尚未被探索。那么:

- 如果深度优先遍历通过边 (u, v) 从 u 到达 v,那么 (u, v) 就是一条生成树边(tree edge);
- 如果深度优先遍历没有通过边 (u, v) 从 u 到达 v,那么在第4步查看 v 时,它已经被访问过了。因此它是在探索 u 的其他邻居时被探索的,这意味着 v 是 u 在DFS树中的祖先。

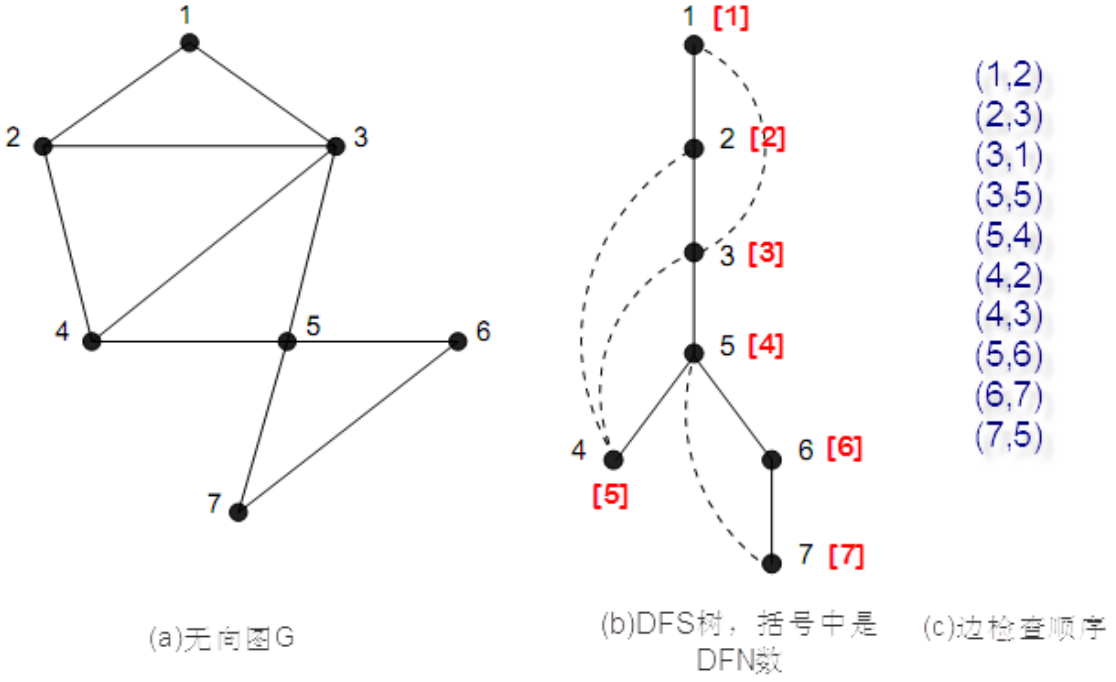
例如在上图中,顶点 4 和 8 之间不可能有反向边 (back edge) 相连,因为它们彼此都不是对方的祖先。如果 4 和 8 之间有边,遍历就会从 4 到达 8,而不会回到 2。

这是关于DFS树的最重要观察。DFS树之所以这么有用,是因为它简化了图的结构。我们不需要担心各种边,只需关注一棵树和一些额外的祖先-后代边。这种结构更容易思考和编写算法。

DFS序号/深度优先数 dfn

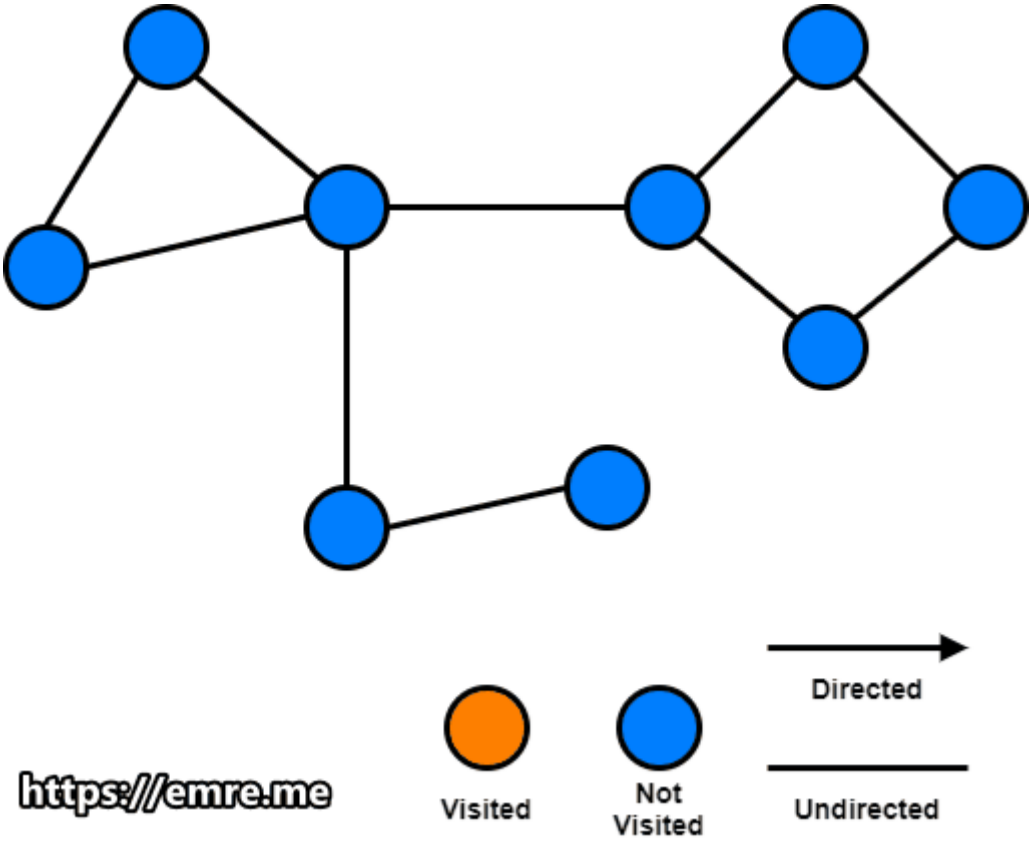
在图上执行深度优先搜索时，用 dfn 表示一个节点在深度优先遍历过程中第一次被访问的时间(dfn is the time when a node is visited first time while DFS traversal).

$dfn[u]$ 称为节点 u 的**深度优先数** 或者 **DFS序号**，在李煜东的书上，称为**时间戳**。显然，越先访问的节点的深度优先数(DFS 序号)越小。



上图展示无向图G的深度优先搜索。(b)中的实线边是树边，虚线边是反向边。顶点1是DFS树的根。(c)列出了边的检查顺序。DFS生成树不唯一，例如树边(5,6)可被反向边(5,7)替换。

下图的动画更清楚地演示了 dfn 形成的过程。对无向图进行DFS，蓝色点表示未访问的节点，橙色点表示已访问的节点，有向边表示边被检查的方向。



计算 dfn 的代码

```
int dfn[N];
int dcnt = 0;
void dfs(int u, int fa) {
  dfn[u] = ++ dcnt;
```

```
  for(auto v : adj[u]) {
    if(!dfn[v]) {
      printf("%d,%d is a tree edge.\n", u,v); //u->v是一条树边
      dfs(v, u);
    }
    else if(v != fa && dfn[v] < dfn[u])
      printf("%d,%d is a back edge.\n", u,v); //u->v是一条反祖边
  }
}

```

关于DFS序的应用，请参考《DFS序/欧拉序》专题。

低链接值(LOW-LINK VALUES)

一个节点 u 的低链接值 low-link(u) 是在与 u 同一连通分量中，通过遍历零个或多个树边 (tree edge) 后并经过**最多一条**反向边(back edge)可到达的最小节点DFS序号(dfn)，即:

$$\text{low-link}(u) = \min \begin{cases} \min\limits_{\substack{(u,v)\in E \\ u是v的父亲}} \text{low-link}(v) \\ \min\limits_{\substack{(u,v)\in \text{back-edges} \\ u\ 是\ v的后代}} \text{dfn}(v) \end{cases}$$

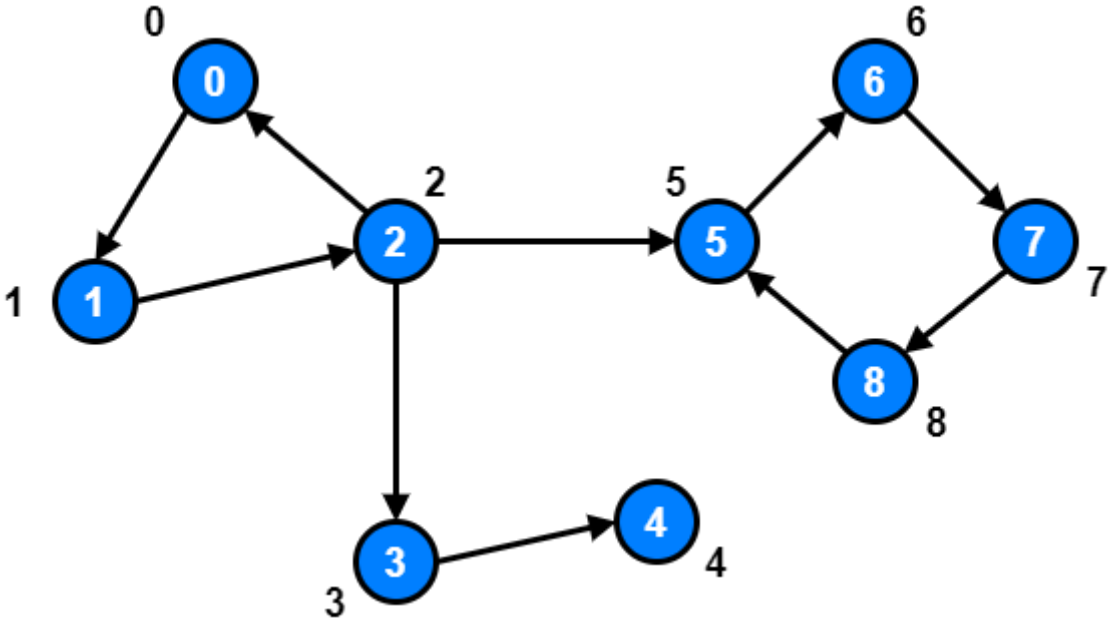
其中, E是树边的集合, back-edges是反向边的集合。

也就是说，一个节点 u 的低链接值是在DFS树中, 节点 u 或者以 u 为根的DFS子树中的任何节点经过仅一条反向边可到达的最小祖先节点DFS序号。

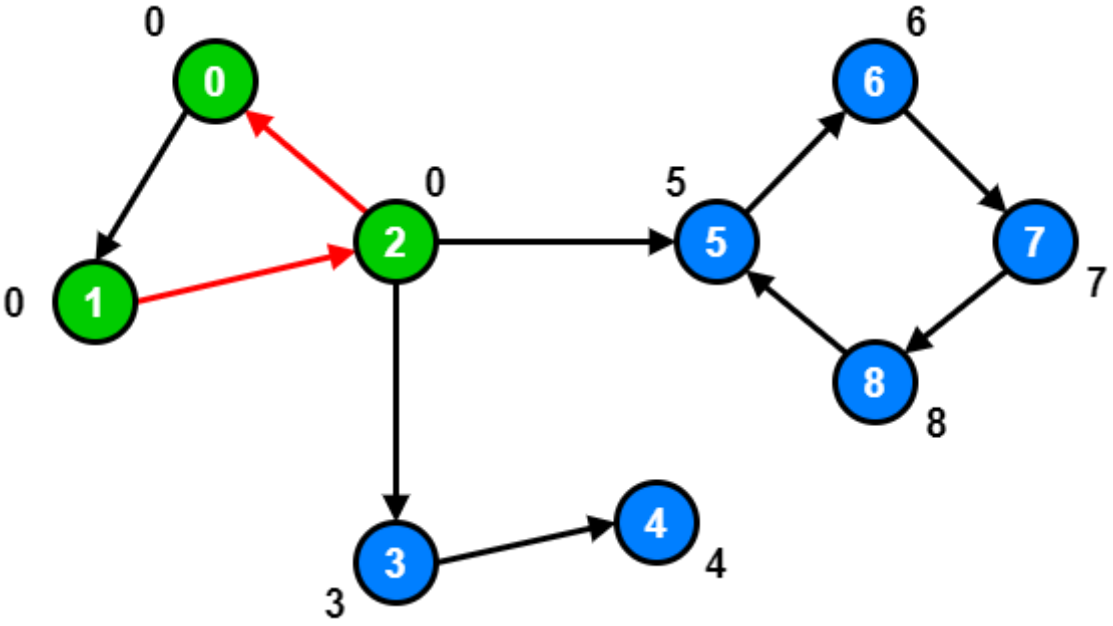
在李煜东书上，把 low-link 称为**追溯值** 。

低链接值的计算过程如下，low-link 简化为 low:

1. 首先，每个节点 u 的低链接值初值为它的 dfn，即: $low(u) = dfn(u)$



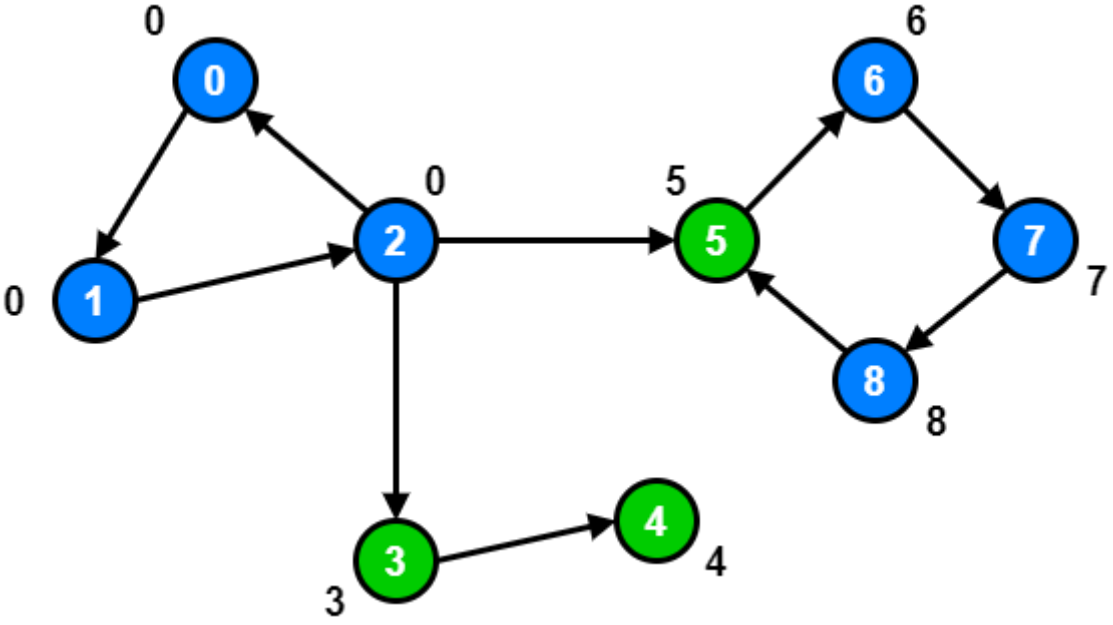
2. 然后，递归地检查每个节点的邻接点 v，如果 (u, v) 是树边，则与子节点 v 的 $low(v)$ 取较小者: $low(u) = min(low(u), low(v))$; 如果 (u, v) 是反向边，则与祖先节点 v 的 $dfn(v)$ 取较小者:
$low(u) = min(low(u), dfn(v))$



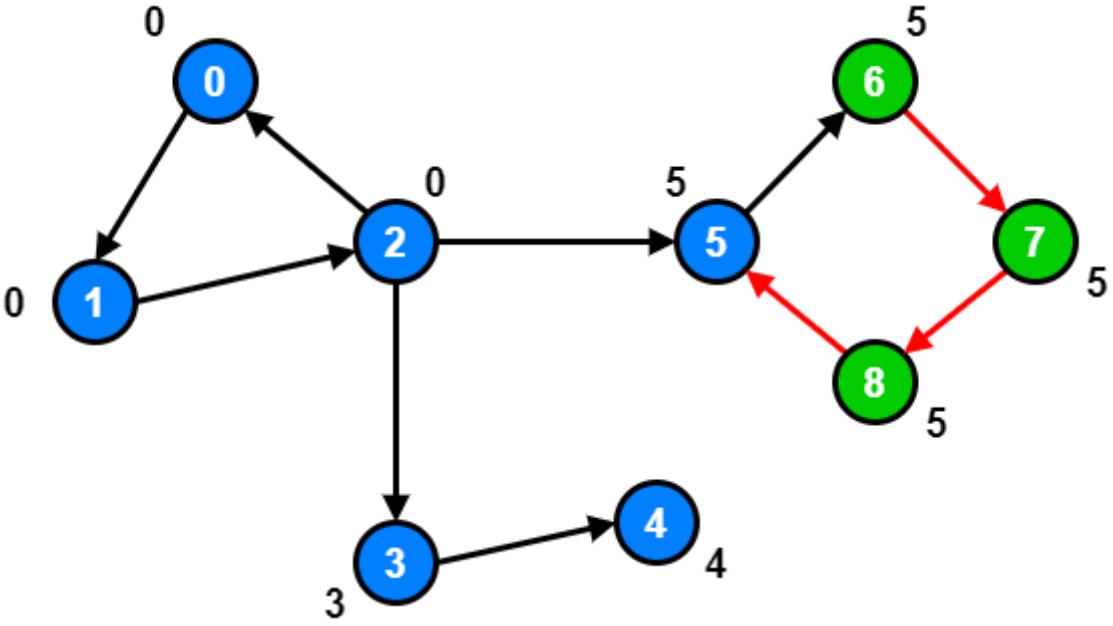
上图中的节点 2 的 $(2, 0)$ 是反向边，所以 $low(2) = min(low(2), dfn(0)) = min(2, 0) = 0$ 。而 2 的两个子节点 3 和 5 的 low 值分别是 3 和 5，均比 0 更大。
节点 1 的 $(1, 2)$ 是树边，所以 $low(1) = min(low(1), low(2)) = min(1, 0) = 0$

可以看出，计算 low 值与计算 dfn 值一样，是在DFS的递归中完成的。

下图展示了节点 $3, 4, 5$ 的 low 值确定过程。



下图展示了节点 $6, 7, 8$ 的 low 值确定过程。

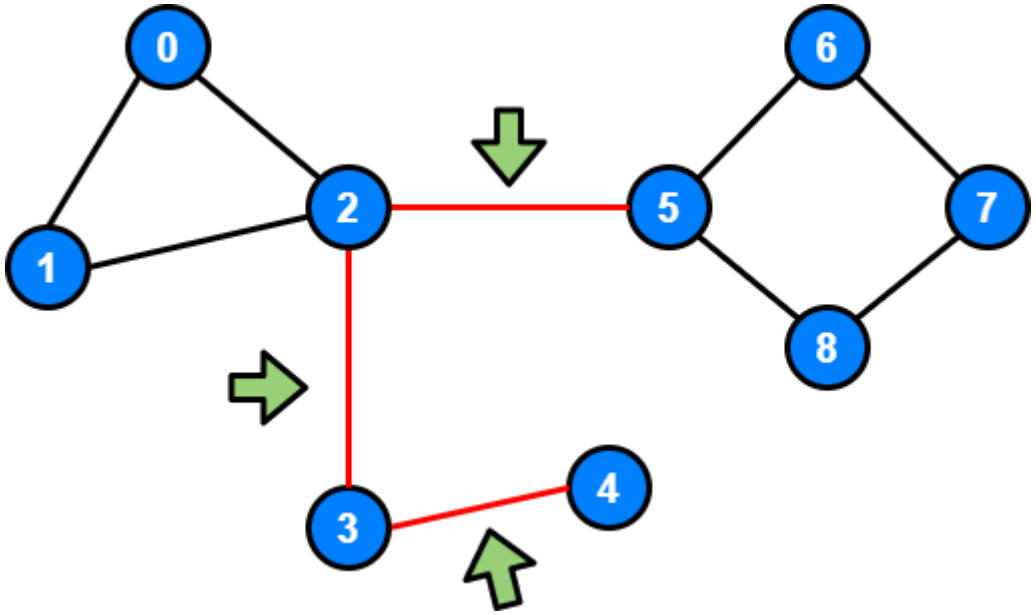


计算 low 值的代码如下：

```
void dfs(int u, int fa){
  low[u] = dfn[u] = ++dcnt;
  for(auto v : adj[u]) {
    if(!dfn[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
    }
    else if(v != fa)
      low[u] = min(low[u], dfn[v]);
  }
}
```

桥边/割边(bridge or cut-edge)

在图论中,桥(或割边)是指图中任何一条边,移除它会增加连通分量的数量。下图中的红色边均是割边。



DFS树和观察1是Tarjan查找桥边算法的核心。下面是如何在无向连通图G中找到桥的方法。考虑G的DFS树。

观察2.一条树边(u,v)是一条桥边当且仅当不存在一条反向边连接(u,v)的后代和祖先。换句话说,一条树边(u,v)是一条桥边当且仅当没有反向边"越过"(u,v)。

解释：移除边(u,v)将生成树分为两个不连通的部分:以(u,v)为根的子树和生成树的其余部分。如果这两个部分之间存在反向边,则图仍然是连通的,否则(u,v)就是一条桥边。反向边连接这两个部分的唯一方式就是连接(u,v)的后代和祖先。

例如,在上面的DFS树中,5和6之间的边不是桥,因为即使移除它,5和8之间的反向边也会保持图的连通性。另一方面,2和5之间的边就是桥边,因为如果移除(2,5),就没有反向边"越过"它来保持图的连通性。

观察3.反向边永远不是桥边。

解释：在无向图上执行DFS，对第一次检查到的无向边 (u,v) 定向成有向弧 $u \to v$，那么反向边一定处在一个以弧尾 v 为起点的有向环上。这个环由若干条树边和唯一的一条反向边组成。因此，反向边不可能是桥边。

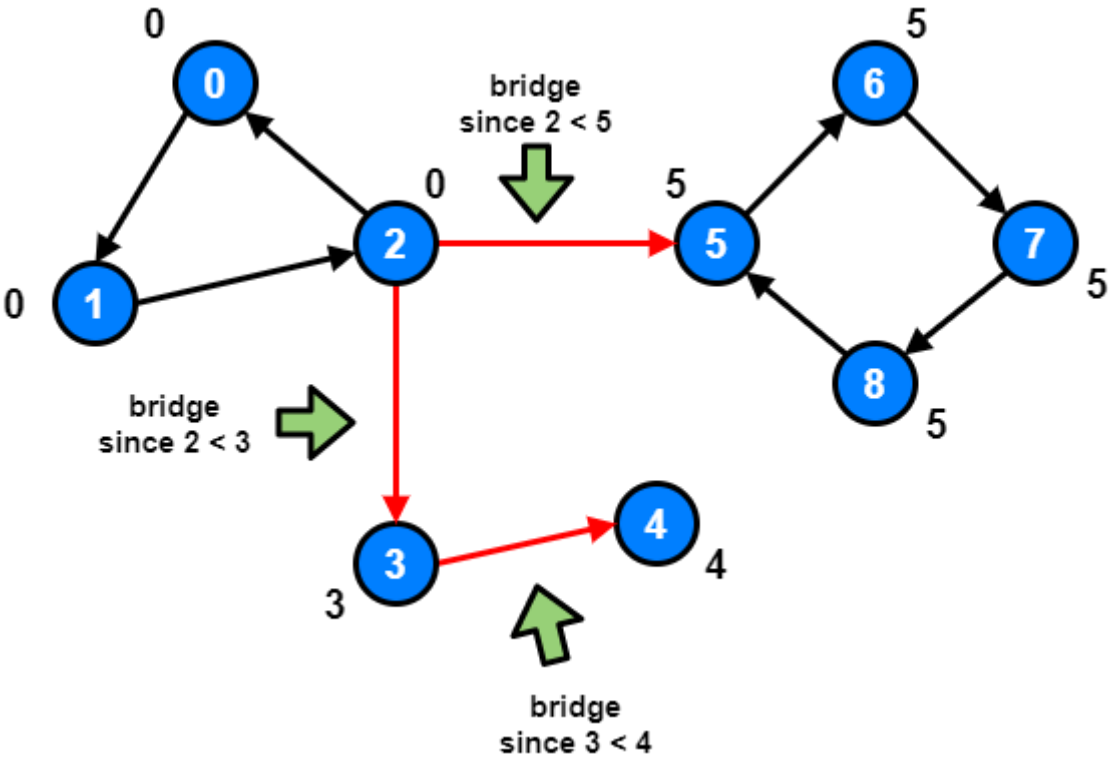
例如下图，有两个有向环。反向边 $(2,0)$ 位于一个由 3 条边组成的环上。反向边 $(8,5)$ 位于一个由 4 条边组成的环上。

这就导致了经典的寻找桥边的算法。给定一个图 G:

1. 找到图的DFS树;
2. 对于每条树边 (u,v), 判断是否存在一条反向边"越过"(u,v), 如果不存在, 那么 (u,v) 就是一条桥边。

在DFS过程中,如果一条正被检查的边的起点 u 的DFS序号小于该边的终点 v 的低链接值(low-link value),那么这条边是一条桥边。即桥边的判断条件是：$low(v) > dfn(u)$。

在下图中，圆圈内的白色数字表示节点的 dfn 值，圆圈外的黑色数字表示 low 值。可以发现， 三条桥边均满足 $low(v) > dfn(u)$。



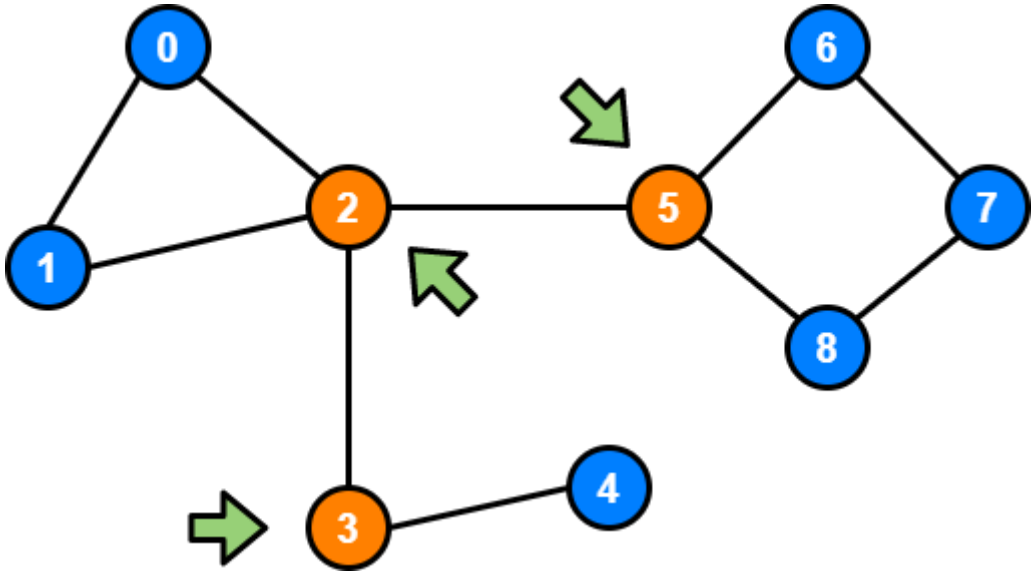
查找割边的代码实现：

```
vector<pair<int, int> > bri;  //割边存入一个pair类型的向量
void dfs(int u, int fa){
  low[u] = dfn[u] = ++dcnt;
```

```
  for(auto v : adj[u]) {
    if(!dfn[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if(low[v] > dfn[u]
          bri.push_back({u, v});
    }
    else if(v != fa)
        low[u] = min(low[u], dfn[v]);
  }
}
```

关节点(articulation point) / 割点 (cut-vertex)

一个关节点(或割点)是指在一个图中,移除这个节点及其关联的边会导致连通分量数量增加的任何节点。



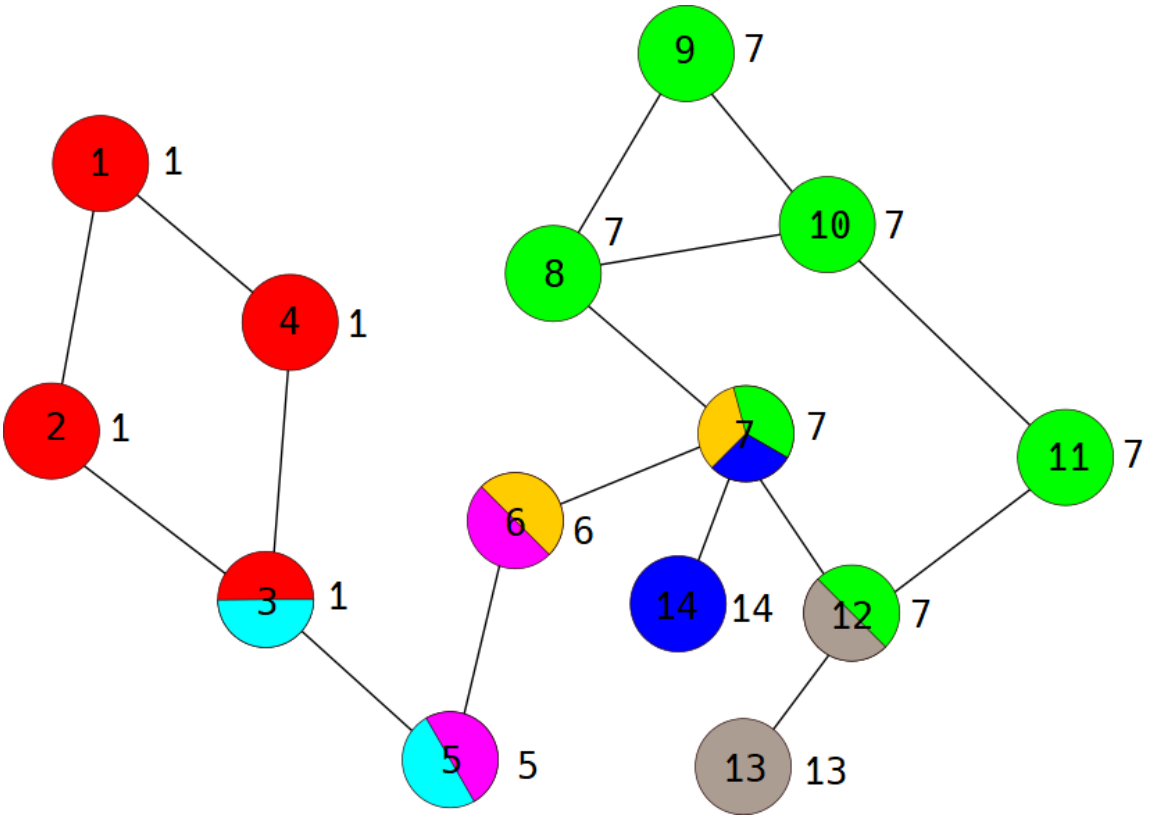
上图中的 3 个橙色节点均是割点。删除它们中任何一个节点及其关联的边后，都会分离出一个新的连通分量。

如果判断图中的割点呢?

观察4. 在DFS生成树上，如果一个非根节点 u 是一个割点,当且仅当存在 u 的一个子节点 v,使得 $low(v) \geq dfn(u)$。

解释：可以分成三种情况讨论。

- $low(v) > dfn(u)$，这时边 (u, v) 是割边，删除节点 u 及关联的边（当然也包含 (u, v)），显然节点 v 所属的连通分量被分离出来。下图中的 3 与 5, 5 与 6 等均是这种情况。
- $low(v) = dfn(u)$，这时DFS生成树以 v 为根的子树中，存在一条反向边指向 u，但不存在反向边指向DFS序更小的节点。这时，删除节点 u 及其关联的边后，节点 v 所属的连通分量被分离出来。下图中的 7 与 8 是这种情况。
- $low(v) < dfn(u)$，这时DFS生成树以 v 为根的子树中，存在一条反向边指向DFS序更小的节点，节点 u 和 v 在一个环路上，删除 u 及其关联的边后，节点 v 仍然可以通过环路上的其它边，与图的其它部分连通。因此不会分离 v 所在的连通分量。这时，u 不是 v 的割点。下图中的 2 与 3，8 与 9 是这种情况。



观察5. DFS生成树的根节点是割点当且仅当它至少有两个子节点。因此,只需要从根节点的每个子树(包括根节点本身)构建一个连通分量即可。

解释：以上图为例，当节点 1 出发执行DFS时，构建的生成树只有一个子节点 2。因此 1 不是割点。如果从节点 5 出发执行DFS，则构建的生成树有两个子节点 3 和 6，因此 5 是割点。

观察6. 割点具有"相对"性，即在DFS生成树上，一个节点 u 相对于 v 是割点，但相对于 v' 不是割点。

解释：例如如果 6 和 8 之间新增一条边，那么 7 相对 14 是割点，但相对 8 不是割点。因此，可以在深度优先搜索从 u 的每个子节点 v 返回时(即在 u 从深度优先搜索栈中弹出之前)测试条件 $low(v) \geq dfn(u)$。如果为真,则 u 将图分离为不同的连通分量。更准确地说，是把 v 所属的连通分量与 u 的父节点所属的连通分量分离开。这可以通过计算每个这样的 v 所形成的一个双连通分量来表示(包含 v 的分量将包含 v 的子树,加上 v),然后从树中删除 v 的子树。

判断割点的代码：

```
bool cut[N];     //标记一个节点是不是割点
int rson;        //根节点的子节点计数
void dfs(int u, int fa){
  low[u] = dfn[u] = ++dcnt;
  for(auto v : adj[u]) {
    if(!dfn[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if(low[v] > dfn[u]) {
         if(fa == -1) rson++;    //u是根，儿子数+1
         else  cut[u] = true;    //u 相对于 v 是割点
      }
    }
    else if(v != fa)
        low[u] = min(low[u], dfn[v]);
  }
```

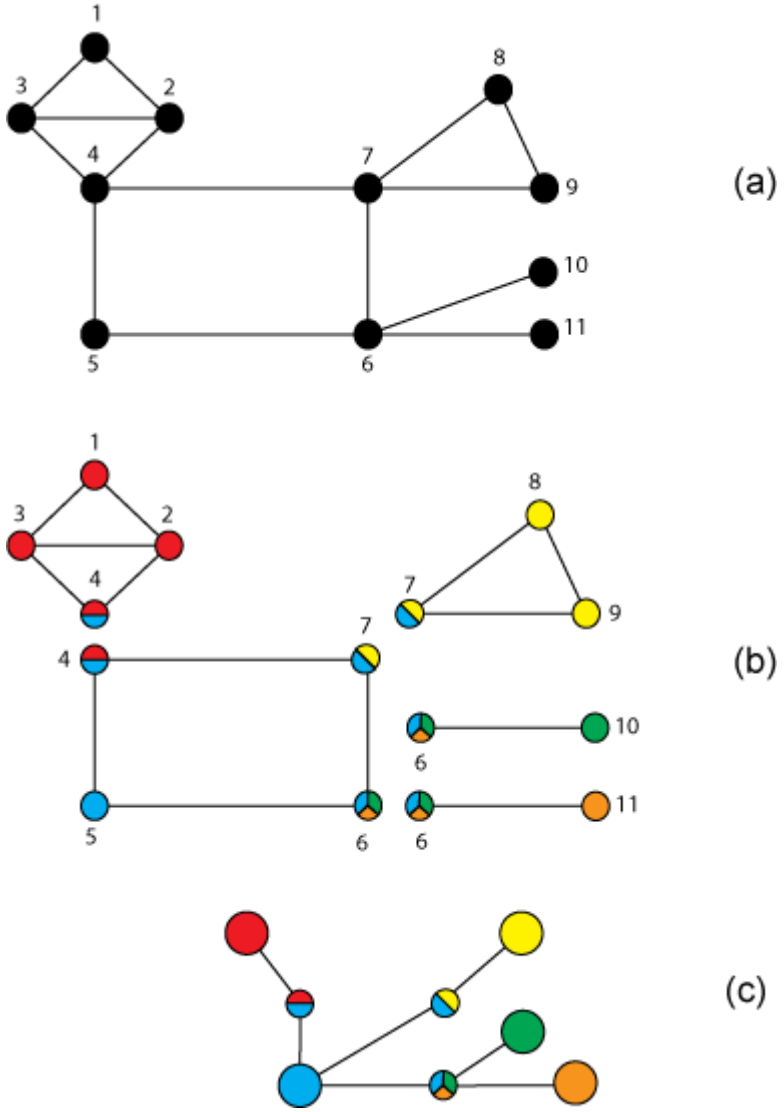
```
17 }
```

无向图的双连通分量

如果一个无向连通图移除任何一个顶点(及与该节点相关的所有边)都不会断开该图，则该连通图是**点双连通**的(vertex double connected)。

更一般地，图的**点双连通分量**(vertex double connected component)是顶点的**极大**子集,在从特定分量中移除一个顶点的情况下,该分量不会被断开。与连通分量不同,顶点可能属于多个双连通分量:那些属于多于一个双连通分量的顶点称为关节点或割点。因此,没有割点的图是点双连通的。

例如，上图中的 $\{1,2,3,4\},\{7,8,9,10,11,12\}$ 是两个点双连通分量。$\{8,9,10\}$ 也没有割点，但不是极大子图，因此不能算作点双连通分量。下图展示了另一个割点和双连通分量的例子:



点双连图图判定定理

一个无向图是点双连通的，当且仅当满足以下两个条件之一：

1. 图的顶点数不超过 2；
2. 图中任意两个顶点都同时包含在至少一个简单环中。

求点双连通分量的算法

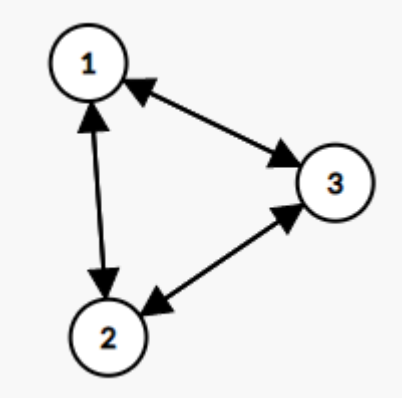
对于点双连通分量，实际上在求割点的过程中就能顺便把每个点双连通分量求出。

建立一个栈，存储当前双连通分量，在搜索图时，每找到一条树边或反向边，就把这条边加入栈中。如果当前检查的树边满足 $dfn(u) <= low(v)$，说明 u 是一个割点，同时把边从栈顶逐个取出，直到遇到了边 (u,v)，取出的这些边与其关联的点，组成一个点双连通分量。

前面的例子已经发现，割点可以属于多个点双连通分量，非割点只属于一个点双连通分量。而一条边只属于一个点连通分量。因此，在求点双连通分量时，栈中记录的是边，不是点。这一点一定要记清楚。

```
int dfn[N], low[N];
int dcnt=0, bcnt=0;
stack<pair<int,int>> stk;
vector<pair<int,int>> blo[MAXN]; //存点双连通分量
void dfs(int u, int fa) {
    low[u] = dfn[u] = ++dcnt;
    for(auto v : adj[u]) {
        if(!dfn[v])    {
            stk.push({u,v});
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(dfn[u] <= low[v]) {    //u是割点
                bcnt++;
                pair<int,int> e;
                do {
                    e = stk.top();    stk.pop();
                    blo[bcnt].push_back(e);
                }while( !(e.first == u && e.second ==v) );
            }
        }
        else if(v != fa) {
            low[u] = min(low[u], dfn[v]);
            if(dfn[u] > dfn[v])       //只存反向边，不存无向图的"前向边"
                stk.push(make_pair(u,v));
        }
    }
}
```

无向图存双向边，当从1->2->3访问，检查边$(3,1)$时，发现是反向边，入栈。但DFS回溯返回1时，会检查另一个方面的边 $1 \rightarrow 3$,即反向边的对称边，但这个方向就不能再入栈了。因此对于反向边入栈时，要特断一下。



例题 CF962F Simple Cycles Edges

题目大意

给定一个无向图,由 n 个顶点和 m 条边组成。图不一定是连通的。保证图中不包含重边(两个顶点之间存在多于一条边)或自环(从一个顶点到自身的边)。

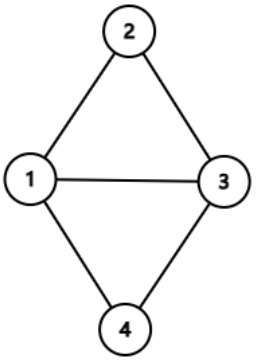
如果一个环在图中包含自身的每个顶点恰好一次,则称之为**简单环**。因此,简单环不允许在一个环中多次访问同一个顶点。

确定属于恰好一个简单环的边。

分析

在图论中,简单环(simple cycle)是指一条从某个顶点出发,经过一些其他顶点,最后回到起点的路径,且沿途没有重复经过任何一个顶点或边。换句话说,简单环是一个闭合的路径,其中不包含任何重复的顶点或边。简单环的重要性质：简单环至少要包含 3 个顶点。

例如,考虑下面这个无向图:

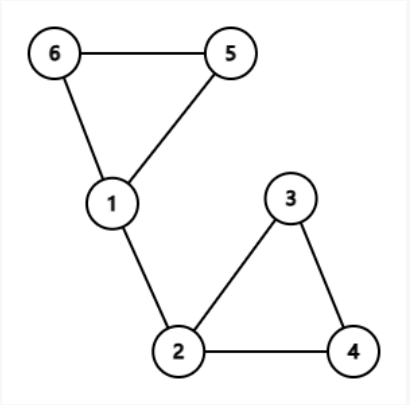
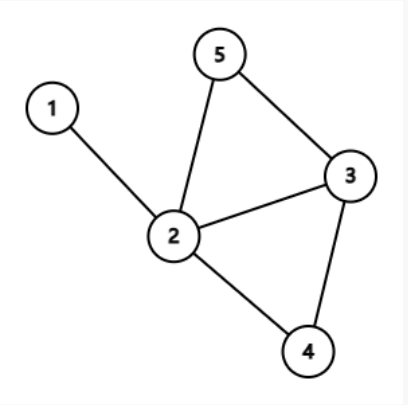


在这个图中:

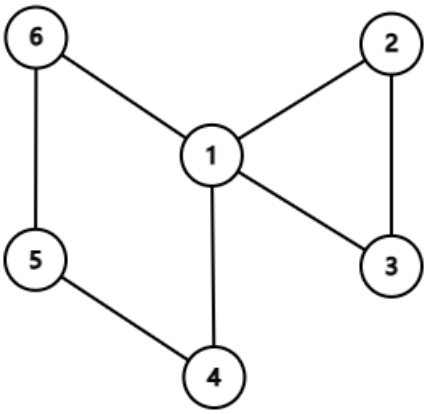
- $1 \to 2 \to 3 \to 1$ 是一个简单环。
- $1 \to 3 \to 4 \to 1$ 是一个简单环。
- $1 \to 2 \to 3 \to 4 \to 1$ 也是一个简单环。

因为每条边都出现在两个简单环中，因此都不是恰好属于一个简单环的边。观察上图后可以发现：**两个边相交的简单环，可以生成一个新的简单环。**

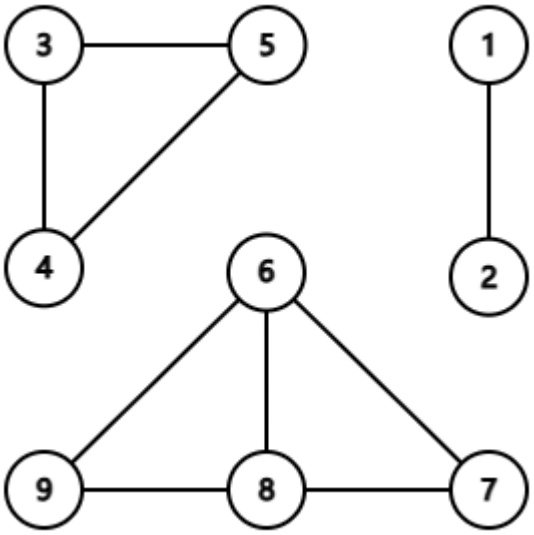
以下是两个样例的答案：

	
6	0

观察下图，有两个简单环相交于节点 1，但**不能**生成一个新的简单环。因为简单环不能重复经过同一个顶点。



基于以上观察，可以发现：**环是不能穿过割点的**。因此，点双连通分量之间互不影响，可以单独判断每一个点双连通分量中的边是否恰好属于一个简单环。



观察上图的 3 个双连通分量，只有 $3 \to 4 \to 5 \to 3$ 中的每条边恰好属于一个简单环。可以发现，在只有一个简单环的双连通分量中，**顶点数=边数**。不满足条件的双连通分量要么顶点数大于边数，要么顶点数小于边数。

如何求一个点双连通分量内部的顶点数和边数呢？我们在跑 Tarjan 算法的时候，定义两个栈，一个 sv，保存DFS沿途经过的节点。一个 se，保存DFS沿途经过的边。每当发现一个割点的时候，就把这两个栈属于这个点双连通分量的节点和边都退栈，判断顶点数与边数是否相等就行了。

```
vector<int> ans;
stack<int> sv; // tarjan经过的每个点
stack<int> se; // tarjan经过的边的编号
void dfs(int u) {
  sv.push(u);
  low[u] = dfn[u] = ++dcnt;
  for (auto e : adj[u]) {
    int v = e.v, id = e.id;
    if (!dfn[v]) {
      fa[v] = id;
      se.push(id);
      dfs(v);
```

```
      low[u] = min(low[u], low[v]);
      if (dfn[u] <= low[v]) {
        // 找到一个点双，统计这个点双里面点的个数和边的条数，如果点和边相等，说明只有一个环
        int cv = 1; // 点的个数，肯定包含割点自己
        int ce = 0; // 边的个数
        while (true) {
          int t = sv.top();
          sv.pop();
          cv++;
          if (t == v)
            break;
        }
        while (true) {
          int t = se.top();
          se.pop();
          vis[ce++] = t;
          if (t == fa[v])
            break;
        }
        if (cv == ce) { // 只有一个环，标记一下
          for (int j = 0; j < ce; j++) {
            ans.push_back(vis[j]);
          }
        }
      }
    } else if (id != fa[u]) {
      low[u] = min(low[u], dfn[v]);
      if (dfn[v] < dfn[u]) { // 反向边进栈
        se.push(id);
      }
    }
  }
}
```

本题在判断点双连通分量是否只包含一个简单环时，还有另一个方法。

根据返祖边的性质，两个简单环的相交处一定在DFS生成树的树边上，那么问题就转换成了边的覆盖。

那么对于每一个简单环，我们将它的返祖边覆盖在原图上，最后统计恰好被覆盖了一条返祖边的区域一定符合题意。

对于一条返祖边 (u, v) 进行差分，最后差分数组元素为1的即为答案。

这个方法对于理解DFS生成树的性质有帮助。建议大家挑战一下。

讨论：洛谷P3225 矿场搭建

无向图的边双连通分量

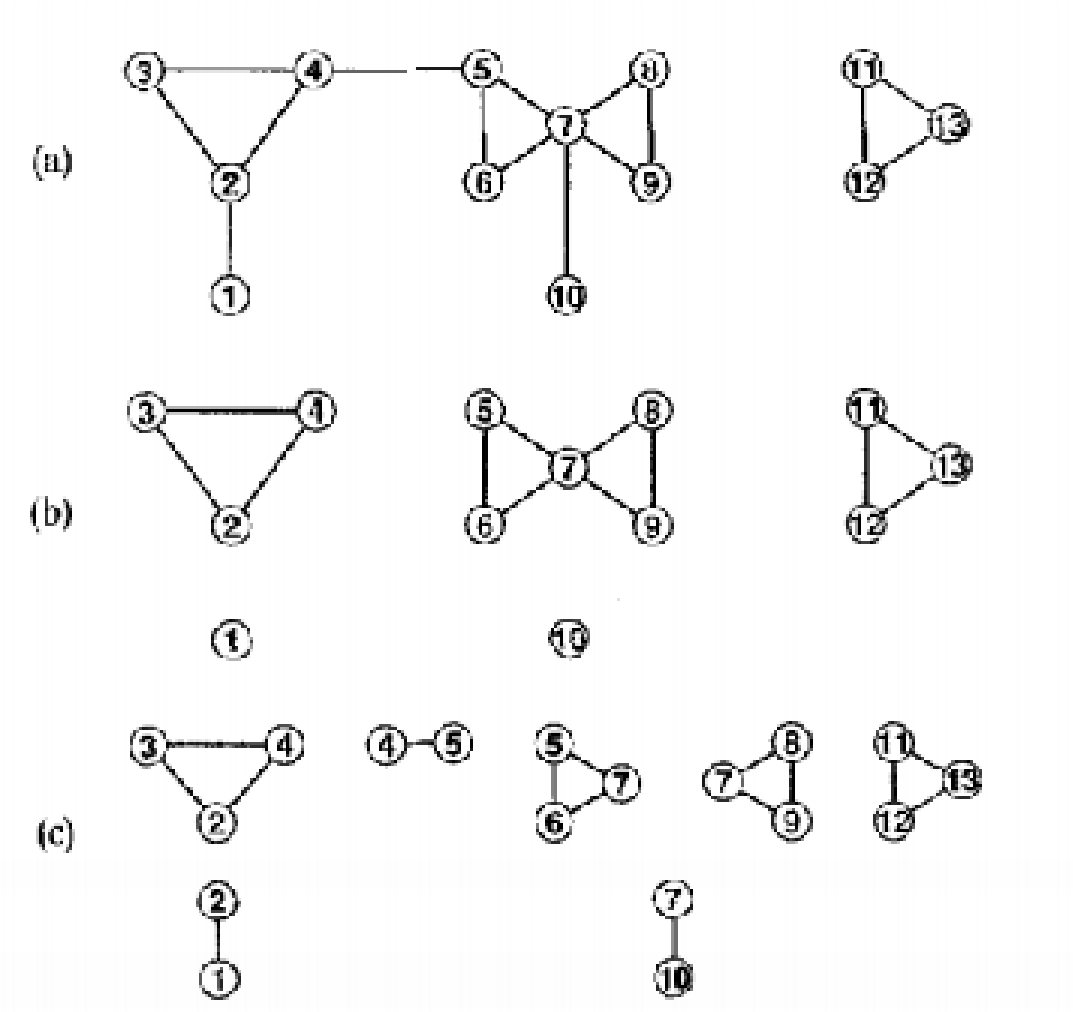
如果一个无向连通图不存在割边，则称为**边双连通图**。

边双连通分量是指不包含桥的极大连通子图。因此，边连通分量的任意两个点之间都有不共边的两条路（可以共点）。这也正是“边双连通分量”的称谓所指。既然边双连通分量不存在桥，那么，删除任意一条边后，分量仍能保持连通性。

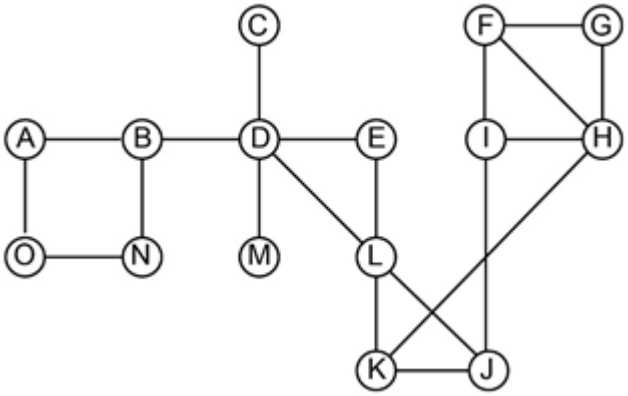
查找图中的边双连通分量，只需在求出所有的桥以后，把桥边删除，原图变成了多个连通块，则每个连通块就是一个边双连通分量。

桥不属于任何一个边双连通分量，其余的边和每个结点都属于且只属于一个边双连通分量。下图的 (a) 是原图，不连通，(b) 是边双连通分量，(c) 是点双连通分量。

需要注意的是，桥边分割出来的单个节点也认为是边双连通分量。例如下图中的节点 $1, 10$ 均是边双连通分量。



请找出下图的边双边双连通分量。



边双连通分量的判定定理

一个无向连通图是边双连通图，当且仅当任意一条边都包含在至少一个简单环中。

求边连通分量的算法

因为每个节点仅属于一个边双连通分量，这一点与有向图的强连通分量相似，故可以用求有向图强连通分量类似的方法，在一次DFS中，用一个栈维护当前遍历过的节点，每当发现当前检查的树边是割边时，就把栈中属于当前边双连通分量的节点退栈，并给每个节点打上一个边双连通分量编号的标记。

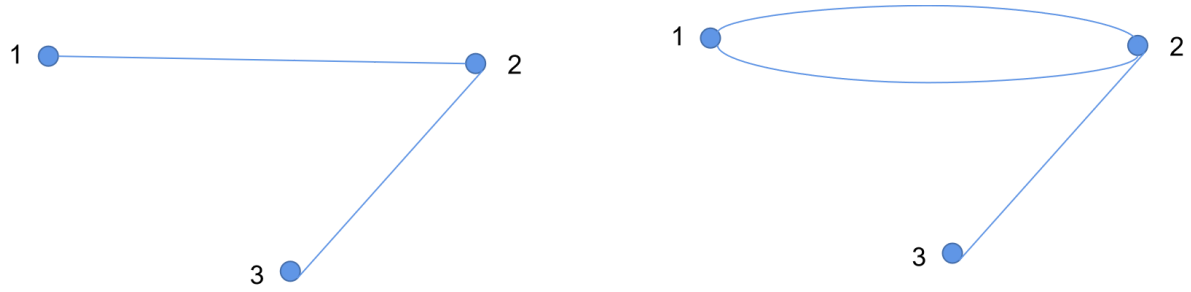
```
//求双连通分量
int dfn[N], low[N], blo[N];
int dcnt=0, blkcnt=0;
stack<int>stk;      //存边双中的结点
void dfs(int u, int fa) {
    low[u] = dfn[u] = ++dcnt;
    stk.push(u);
    for(auto v : adj[u]) {
        if(!dfn[v]){
            dfs(v, u);
            low[u] = min(low[u], low[v]);
        }
        else if(v != fa)
            low[u] = min(low[u], dfn[v]);
    }
    if(low[u]==dfn[u]){ //u的子树中,没有返祖边, (u, fa)是桥
        blkcnt++;
        int v;
        do{
            v = stk.top(); stk.pop(); //栈中的元素构成一个边双
            blo[v] = blkcnt;
        }while(v != u && !stk.empty());
    }
}
//利用blo数组，枚举原图中的边，可以建新图。
//一条边的两个端点的编号不同，则属于新图中的边

```

有重边的双连通分量的处理方法

根据双连通分量的定义可以知道，自环边对于点双连通图和边双连通量都没有影响。

考虑重边对双连通分量的影响，可以发现，**重边对点双连通没有影响，但是对于边双连通有影响。**

为什么重边对点双连通没有影响？因为在删除一个节点之后，与它相连的边都会被删除，那么既然会被删除，即使有再多重边对点双连通的判定也没有影响。但对于边双连通来说，重边会导致这组边自身形成环路，从而形成一个边双连通分量。如下图所示，左图有 3 个边双连通分量，而右图则在 $1, 2$ 之间有重边，合并为一个边双连通分量。



如何把图中有重边的部分识别为一个边连通分量?

既然重边彼此之间区别看待，那么重边就可看成反祖边。例如，上图中，从节点 1 出发执行DFS，$(1,2)$ 作为树边，到达节点 2 后，重边 $(2,1)$ 可看成返祖边，去更新节点 2 的 low 值。但请注意，无向图邻接表保存了两个方向的边：$u \to v$ 和 $v \to u$。它们本质上是同一条无向边。为了区别无向边两个方向的单向边与真正的重边，我们对每条无向图的两个单向边连续编号，一奇一偶。对入输入的第 i 条边，编号为 $2i$ 和 $2i+1$。要判别两条边是否是同一条边的两个单向边，只需要把其中一个编号与 1 异或后是否与另一个编号相等。

```
if(e1.id ^ 1 == d2.id) cout << "same edge.";
```

因此，在有重边的图的邻接表中，每个 $node$ 保存两个信息：边的端点，边的编号。

相应地，在Tarjan算法求桥边的代码中，也不能再传入父节点作为参数去判断是否是同一条边的另一个方向，此时应该传入边的编号。看以下代码实现。

例题 洛谷 P8436 边双连通分量

题目大意:

找出无向图的每个边双连通分量。图有重边。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5e5 + 5;
int n, m, dcnt;
int dfn[N], low[N];
struct node {
  int v, id;
};
vector<node> adj[N];
vector<vector<int>> ans;
stack<int> st;
void dfs(int u, int id) {

```

```cpp
  low[u] = dfn[u] = ++dcnt;
  st.push(u);
  for (auto e : adj[u]) {
    if (e.id == (id ^ 1))
      continue;
    if (!dfn[e.v]) {
      dfs(e.v, e.id);
      low[u] = min(low[u], low[e.v]);
    } else
      low[u] = min(low[u], dfn[e.v]);
  }
  if (dfn[u] == low[u]) {
    vector<int> vec;
    vec.push_back(u);
    while (st.top() != u) {
      vec.push_back(st.top());
      st.pop();
    }
    st.pop();
    ans.push_back(vec);
  }
}
int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);

  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back({v, i << 1}); // 每条边正反向编号分别是 2*i, 2*i+1
    adj[v].push_back({u, i << 1 | 1});
  }
  for (int i = 1; i <= n; i++)
    if (!dfn[i])
      dfs(i, 0);

  cout << ans.size() << '\n';
  for (auto vec : ans) {
    cout << vec.size() << ' ';
    for (auto v : vec)
      cout << v << ' ';
    cout << '\n';
  }
  return 0;
}
```

边双连通分量的缩点

对于给定的无向图 G，构建一个新图 G'，把 G 中每个边双连通分量看作 G' 的一个节点，把割边 (u,v) 看作连接编号为 $blo[u]$ 和 $blo[v]$ 的 双连通分量对应节点的无向边，这样得到的 G' 是一棵树。如果 G 不连通，则 G' 为森林。

这种把边双连通分量收缩为 G' 中一个节点的方法称为 “缩点”。新图 G' 需要存储在另一个邻接表中。

例题 CF1000E We Need More Bosses

题目大意：

给定一个 n 个点 m 条边的无向图，保证图连通。找到两个点 s,t，使得 s 到 t **必须**经过的边最多（一条边无论走哪条路线都要经过，这条边就是必须经过的边），
$2 <= n <= 3 * 10^5, 1 <= m <= 3 * 10^5$

分析

这是很明显的,我们只能将老板们放置在给定图的桥边上——如果一条边不是桥边,那么移除它不会使图变得不连通,因此任何一对顶点之间仍然存在路径。而且如果我们固定两个顶点 s 和 t,然后在它们之间找到一些简单路径,那么我们将在属于这条路径的所有桥边上放置老板(因为不管我们选择 s 和 t 之间的哪条简单路径,桥边集合都会保持不变)。

如果我们在给定的图中找到桥边,并将所有的边双连通分量(如果存在一条不包含桥边的路径连接两个顶点,那么这两个顶点属于同一个边双连通分量)压缩为单个顶点,我们将得到一棵特殊的树,称为桥边树。桥边树的每一条边对应于原始图中的一条桥边(反之亦然)。由于我们想要找到包含最多桥边的路径,我们只需要找到桥边树的直径,这就是问题的答案。

缩点操作的参考实现：

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 3e5 + 5;
int n, m, bcnt, dcnt;
vector<int> adj[N], adj2[N];
int low[N], dfn[N], bel[N], dis[N];
stack<int> st;
void tarjan(int u, int fa) {
  low[u] = dfn[u] = ++dcnt;
  st.push(u);
  for (auto v : adj[u]) {
    if (v == fa)
      continue;
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
    } else
      low[u] = min(low[u], dfn[v]);
  }
  if (low[u] == dfn[u]) {
    int v = -1;
    ++bcnt;
    while (v != u && !st.empty()) {
      v = st.top(), st.pop();
      bel[v] = bcnt;
    }
```

```
  }
}
void dfs(int u, int fa) {
  dis[u] = dis[fa] + 1;
  for (int v : adj2[u]) {
    if (v != fa)
      dfs(v, u);
  }
}
int dia() { // 两遍DFS求树的直径
  dfs(1, 0);
  int s = 1;
  for (int i = 1; i <= bcnt; i++) {
    if (dis[i] > dis[s])
      s = i;
  }
  dfs(s, 0);
  int ans = 0;
  for (int i = 1; i <= bcnt; i++) {
    ans = max(ans, dis[i]);
  }
  return ans - 1;
}
int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cin >> n >> m;
  for (int i = 1; i <= m; ++i) {
    int x, y;
    cin >> x >> y;
    adj[x].push_back(y);
    adj[y].push_back(x);
  }
  for (int i = 1; i <= n; ++i) {
    if (!dfn[i])
      tarjan(i, 0);
  }
  for (int u = 1; u <= n; u++) { // 缩点建新图
    for (int v : adj[u]) {
      if (bel[u] != bel[v]) {
        adj2[bel[u]].push_back(bel[v]);
      }
    }
  }
  int ans = dia();
  cout << ans << '\n';
  return 0;
}
```

官方实现中找桥边的方法是并查集+欧拉序。

```
#include <bits/stdc++.h>
using namespace std;
```

```
const int N = 500043;
vector<int> g[N];
vector<int> t[N];
int tin[N], tout[N], fup[N];
int p[N];
int T = 1;
int rnk[N];
vector<pair<int, int>> edge;
int st;
int d[N];
int n, m;
int find(int x) { return (p[x] == x ? x : p[x] = find(p[x])); }
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if (x == y)
    return;
  if (rnk[x] > rnk[y])
    swap(x, y);
  p[x] = y;
  rnk[y] += rnk[x];
}
int dfs(int x, int par = -1) {
  tin[x] = T++;
  fup[x] = tin[x];
  for (auto y : g[x]) {
    if (tin[y] > 0) {
      if (par != y)
        fup[x] = min(fup[x], tin[y]);
    } else {
      int f = dfs(y, x);
      fup[x] = min(fup[x], f);
      if (f > tin[x])
        edge.push_back({x, y});
      else
        unite(x, y);
    }
  }
  tout[x] = T++;
  return fup[x];
}
void build() {
  for (auto z : edge) {
    int x = find(z.first);
    int y = find(z.second);
    st = x;
    t[x].push_back(y);
    t[y].push_back(x);
  }
}
pair<int, int> bfs(int x) {
  for (int i = 0; i < n; i++)
    d[i] = n + 1;
  d[x] = 0;
  queue<int> q;
  q.push(x);
```

```
  int last = 0;
  while (!q.empty()) {
    last = q.front();
    q.pop();
    for (auto y : t[last])
      if (d[y] > d[last] + 1) {
        d[y] = d[last] + 1;
        q.push(y);
      }
  }
  return make_pair(last, d[last]);
}
int main() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < n; i++)
    rnk[i] = 1, p[i] = i;
  for (int i = 0; i < m; i++) {
    int x, y;
    scanf("%d %d", &x, &y);
    --x;
    --y;
    g[x].push_back(y);
    g[y].push_back(x);
  }
  dfs(0);
  build();
  pair<int, int> p1 = bfs(st);
  pair<int, int> p2 = bfs(p1.first);
  printf("%d\n", p2.second);
}
```

例题 POJ3694/百练3772 Network

题目大意

给出一个 N 个点 M 条边的无向连通图，执行 Q 次操作，每次向图中添加一条边， 然后询问当前图中割边的数量。

$N \leq 10^5, M \leq 2 * 10^5, Q \leq 10^3$

分析

算法1： 暴力找LCA

用Tarjan算法，求出原图中的割边。用数组 cut 标记找到的割边。当判断出树边 (u, v) 是割边后，在子节点 v 处打标记：$cut[v] = true$。再用 fa, dep 两个数组在跑Tarjan算法时分别记录每个节点的父节点和深度。跑完Tarjan算法可得到一棵DFS生成树。

每次添加新边 (u, v)的时候，会在DFS生成树上形成一个环，环上的割边将不再是割边，需要减少相应的数量。具体做法是：设 u 的深度更大，否则 u, v 互换。当 $dep[u] > dep[v]$ 时，u 不断爬升为父节点，并清除遇到的割点标记。然后当 $u! = v$ 时，u, v 不断爬升为父节点，并清除各自遇到的割点标记。当 $u == v$ 表示走到了 lca，本次操作结束，回答当前询问。

Tarjan算法时间复杂度为 $O(N + M)$，询问时间为 $O(QN)$，算法总时间为$O(M + QN)$。对于题目给定的数据规模，可以通过。本算法的优点是算法好想，代码好写。

```cpp
/*
ac 2024.5.12
POJ3694 C++98 3188MS ac
POJ3694 G++98 3094MS ac
Bailian G++17 371MS  ac
*/
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
const int N = 1e5 + 5;
int n, dcnt, ans;
int dfn[N], low[N], fa[N], dep[N];
bool cut[N];
vector<int> adj[N];
void dfs(int u, int pre) {
  dfn[u] = low[u] = ++dcnt;
  dep[u] = dep[pre] + 1; // 记录深度
  fa[u] = pre;           // 记录父节点
  for (int i = 0; i < adj[u].size(); i++) {
    int v = adj[u][i];
    if (v == pre)
      continue;
    if (!dfn[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (dfn[u] < low[v]) { // cut[v]表示在树中，以v为儿子结点的边是否为桥
        cut[v] = true;
        ans++;
      }
    } else
      low[u] = min(low[u], dfn[v]);
  }
}
void lca(int u, int v) {
  if (dep[u] < dep[v])
    swap(u, v);
  while (dep[u] > dep[v]) { // 深度大的先往上爬。遇到桥，就把它删去。
    if (cut[u])
      ans--, cut[u] = false;
    u = fa[u];
  }
  while (u != v) { // 当深度一样时，一起爬。遇到桥，就把它删去。
    if (cut[u])
      ans--, cut[u] = false;
    u = fa[u];
    if (cut[v])
      ans--, cut[v] = false;
    v = fa[v];
  }
}
void init() {
  dcnt = 0;
```

```
  ans = 0;
  memset(dfn, 0, sizeof dfn);
  memset(cut, 0, sizeof cut);
  for (int i = 1; i <= n; i++)
    adj[i].clear();
}
int main() {
  int m, q, cas = 0;
  while (scanf("%d%d", &n, &m) && (n || m)) {
    init();
    for (int i = 1; i <= m; i++) {
      int u, v;
      scanf("%d%d", &u, &v);
      adj[u].push_back(v);
      adj[v].push_back(u);
    }
    dep[1] = 0;
    dfs(1, 1);
    scanf("%d", &q);
    printf("Case %d:\n", ++cas);
    while (q--) {
      int a, b;
      scanf("%d%d", &a, &b);
      lca(a, b);
      printf("%d\n", ans);
    }
    printf("\n");
  }
  return 0;
}
```

算法2：并查集维护边双连通分量

用Tarjan算法，求出原图中每个边双联通分量，把属于同一个边双的节点用并查集合并。用 $p[u]$ 表示在并查集中节点 u 的代表元素，仍用 $fa[u]$ 分别表示在DFS生成树上 u 的父节点，但不需要 dep 数组，改用 dfn 来判断最近公共祖先。

怎样用并查集合并边双呢？一种方法是用栈记录DFS途经的节点，在判断出当前节点的 $dfn[u] == low[u]$时，说明 $(u, fa[u])$ 是割边，把同一个边双的节点退栈，加入并查集。

另一种更简单的方法是：在跑Tarjan算法时，对于树边 (u, v)，当 $dfn[u] < low[v]$ 时，割边数量加 1，否则就说明 u, v 属于同一个边双，合并 u, v。对于返祖边(u, v)，不用合并，因为它们已经沿着树边的路径合并在一起了。

```
void tarjan(int u, int pre) {
  dfn[u] = low[u] = ++dcnt;
  for (int i = 0; i < adj[u].size(); i++) { // for (int v : adj[u]) {
    int v = adj[u][i];
    if (v == pre)
      continue;
    if (!dfn[v]) {
      fa[v] = u;
      dep[v] = dep[u] + 1;
```

```
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (dfn[u] < low[v]) {
        ans++;
      } else
        unite(u, v);
    } else
      low[u] = min(low[u], dfn[v]);
  }
}
```

每次询问的时候，如果 u, v 在同一个边双分量中，那么$find(u) == find(v)$，则直接返回。否则，就合并$u \to LCA(u, v) \to v$ 成一个集合，同时统计路径上割边的数量。

这里用 dfn 来判断是否走到最近公共祖先的节点。具体方法是：当 $dfn[u] > dfn[v]$时，沿着 $u \to LCA(u, v)$ 的路径往上爬，每一步判断 u 和 $fa[u]$ 是否在同一个并查集中，如果不在，则 $(u, fa[u])$ 是割边，合并 u 和 $fa[u]$ 两个集合。当 $dfn[u] <= dfn[v]$时停止。

接来来反过来判断当 $dfn[v] > dfn[u]$ 时，沿着 $v \to LCA(u, v)$ 的路径往上爬，每一步判断 v 和 $fa[v]$ 是否在同一个并查集中，如果不在，则$(v, fa[v])$ 是割边，合并 v 和 $fa[v]$ 两个集合。当 $dfn[v] <= dfn[u]$时停止，这时一定走到了$LCA(u, v)$的节点。

```
void lca(int u, int v) {
  u = find(u), v = find(v);
  if (u == v)
    return;
  if (dfn[u] < dfn[v])
    swap(u, v);
  while (dfn[u] > dfn[v]) {
    int fx = find(u), fy = find(fa[u]);
    if (fx != fy) {
      ans--;
      p[fx] = fy;
    }
    u = fa[u];
  }
  while (dfn[v] > dfn[u]) {
    int fx = find(v), fy = find(fa[v]);
    if (fx != fy) {
      ans--;
      p[fx] = fy;
    }
    v = fa[v];
  }
}
```

本算法的查询时间复杂度为$O(QlogN)$，总时间为 $O(M + QlogN)$。当 Q 较大时，比暴力求LCA算法更高效。

本算法对并查集，Tarjan算法的理解深入，灵活，在不增加太多算法复杂度的基础上，让算法效率有较明显提升。要求大家一定要掌握。

```
/*
ac 2024.5.12
POJ3694     G++98  2719MS
```

```
POJ3694     C++98  2797MS
Bailian3772 C++17  138MS
*/
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
const int N = 1e5 + 5;
int n, m, ans;
vector<int> adj[N];
int dcnt;
int dfn[N], low[N], p[N], fa[N], dep[N];
void init() {
  dcnt = 0;
  ans = 0;
  memset(dfn, 0, sizeof(dfn));
  for (int i = 0; i <= n; i++)
    p[i] = i, adj[i].clear();
}
int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
void unite(int x, int y) {
  int fx = find(x), fy = find(y);
  if (fx != fy)
    p[fx] = fy;
}
void tarjan(int u, int pre) {
  dfn[u] = low[u] = ++dcnt;
  for (int i = 0; i < adj[u].size(); i++) { // for (int v : adj[u]) {
    int v = adj[u][i];
    if (v == pre)
      continue;
    if (!dfn[v]) {
      fa[v] = u;
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (dfn[u] < low[v]) {
        ans++;
      } else
        unite(u, v);
    } else
      low[u] = min(low[u], dfn[v]);
  }
}
void lca(int u, int v) {
  u = find(u), v = find(v);
  if (u == v)
    return;
  if (dfn[u] < dfn[v])
    swap(u, v);
  while (dfn[u] > dfn[v]) {
    int fx = find(u), fy = find(fa[u]);
    if (fx != fy) {
      ans--;
      p[fx] = fy;
    }
```

```
    u = fa[u];
  }
  while (dfn[v] > dfn[u]) {
    int fx = find(v), fy = find(fa[v]);
    if (fx != fy) {
      ans--;
      p[fx] = fy;
    }
    v = fa[v];
  }
}
int main() {
  int cas = 0;
  while (scanf("%d%d", &n, &m) != EOF) {
    if (n + m <= 0)
      break;
    cas++;
    init();
    for (int i = 1; i <= m; i++) {
      int u, v;
      scanf("%d%d", &u, &v);
      adj[u].push_back(v);
      adj[v].push_back(u);
    }
    tarjan(1, 1);
    int q;
    printf("Case %d:\n", cas);
    scanf("%d", &q);
    while (q--) {
      int u, v;
      scanf("%d%d", &u, &v);
      lca(u, v);
      printf("%d\n", ans);
    }
  }
  return 0;
}
```

算法3：缩点+重建图+并查集

这种算法与算法2的最大区别在于先把原图上的边双缩点后，重建图。在新图(实质是一棵树)上继续用并查集维护边双。

这种算法的编码复杂度较高，建议选择性掌握。

```
/*
ac
2024.5.12
POJ3694     G++98      RE
POJ3694     C++98      TLE
Bailian3772 C++17      78ms
*/
#include <algorithm>
#include <cmath>
```

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <vector>
using namespace std;
const int N = 1e5 + 10;
vector<int> adj[N], adj2[N];
int dcnt, dfn[N], low[N], bel[N], fa[N], dep[N];
int n, ans;
stack<int> st;
int find(int x) { return bel[x] == x ? x : bel[x] = find(bel[x]); }
void dfs(int u, int pre) {
  dfn[u] = low[u] = ++dcnt;
  st.push(u);
  for (int i = 0; i < adj[u].size(); i++) {
    int v = adj[u][i];
    if (v == pre)
      continue;
    if (!dfn[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u])
        ans++;
    } else
      low[u] = min(low[u], dfn[v]);
  }
  if (dfn[u] == low[u]) {
    int v;
    do {
      v = st.top();
      st.pop();
      bel[v] = u; // 把集合的编号设为联通分量的第一个点
    } while (v != u);
  }
}
void build(int u, int pre) {
  fa[u] = pre;           // 记录父亲节点
  dep[u] = dep[pre] + 1; // 记录深度
  for (int i = 0; i < adj2[u].size(); i++) {
    int v = adj2[u][i];
    if (v != pre) // 防止往回走
      build(v, u);
  }
}
int LCA(int u, int v) { // 左一步右一步地找LCA
  if (u == v)
    return u; // 因为两个结点一定有LCA， 所以一定有u==v的时候

  // 可能爬一步就爬了几个深度，因为中间的结点已经往上缩点了
  if (dep[u] < dep[v])
    swap(u, v); // 深度大的往上爬
```

```
  ans--;
  int lca = LCA(find(fa[u]), v);
  // 找到了LCA，在沿路返回的时候把当前节点的所属集合置为LCA的所属集合
  return bel[u] = lca;
}
void init() {
  dcnt = 0;
  memset(dfn, 0, sizeof(dfn));
  ans = 0;
  for (int i = 1; i <= n; i++)
    adj[i].clear(), adj2[i].clear();
}
int main() {
  int m, cas = 0;
  while (scanf("%d%d", &n, &m) && (n || m)) {
    init();
    for (int i = 1; i <= m; i++) {
      int u, v;
      scanf("%d%d", &u, &v);
      adj[u].push_back(v);
      adj[v].push_back(u);
    }
    dfs(1, 1);
    for (int u = 1; u <= n; u++)                  // 重建建图
      for (int i = 0; i < adj[u].size(); i++) { // for (int v : adj[u]) {
        int v = adj[u][i];
        int fu = find(u);
        int fv = find(v);
        if (fu != fv) {
          adj2[fu].push_back(fv); // 为什么这里只能建单向边？
          // adj2[fv].push_back(fu);   双向边 TLE
        }
      }
    dep[find(1)] = 0;
    build(find(1), find(1)); // 把无根树转为有根树

    int q, a, b;
    scanf("%d", &q);
    printf("Case %d:\n", ++cas);
    while (q--) {
      scanf("%d%d", &a, &b);
      LCA(find(a), find(b));
      printf("%d\n", ans);
    }
    printf("\n");
  }
  return 0;
}
```

例题 洛谷P2860 冗余路径 dundant Paths G

题面翻译

为了从 $F(1 \leq F \leq 5000)$ 个草场中的一个走到另一个，贝茜和她的同伴们有时不得不路过一些她们讨厌的可怕的树．奶牛们已经厌倦了被迫走某一条路，所以她们想建一些新路，使每一对草场之间都会至少有两条相互分离的路径，这样她们就有多一些选择。

每对草场之间已经有至少一条路径．给出所有 R $(F-1 \leq R \leq 10000)$ 条双向路的描述，每条路连接了两个不同的草场，请计算最少的新建道路的数量，路径由若干道路首尾相连而成．两条路径相互分离，是指两条路径没有一条重合的道路．但是，两条分离的路径上可以有一些相同的草场．对于同一对草场之间，可能已经有两条不同的道路，你也可以在它们之间再建一条道路，作为另一条不同的道路。

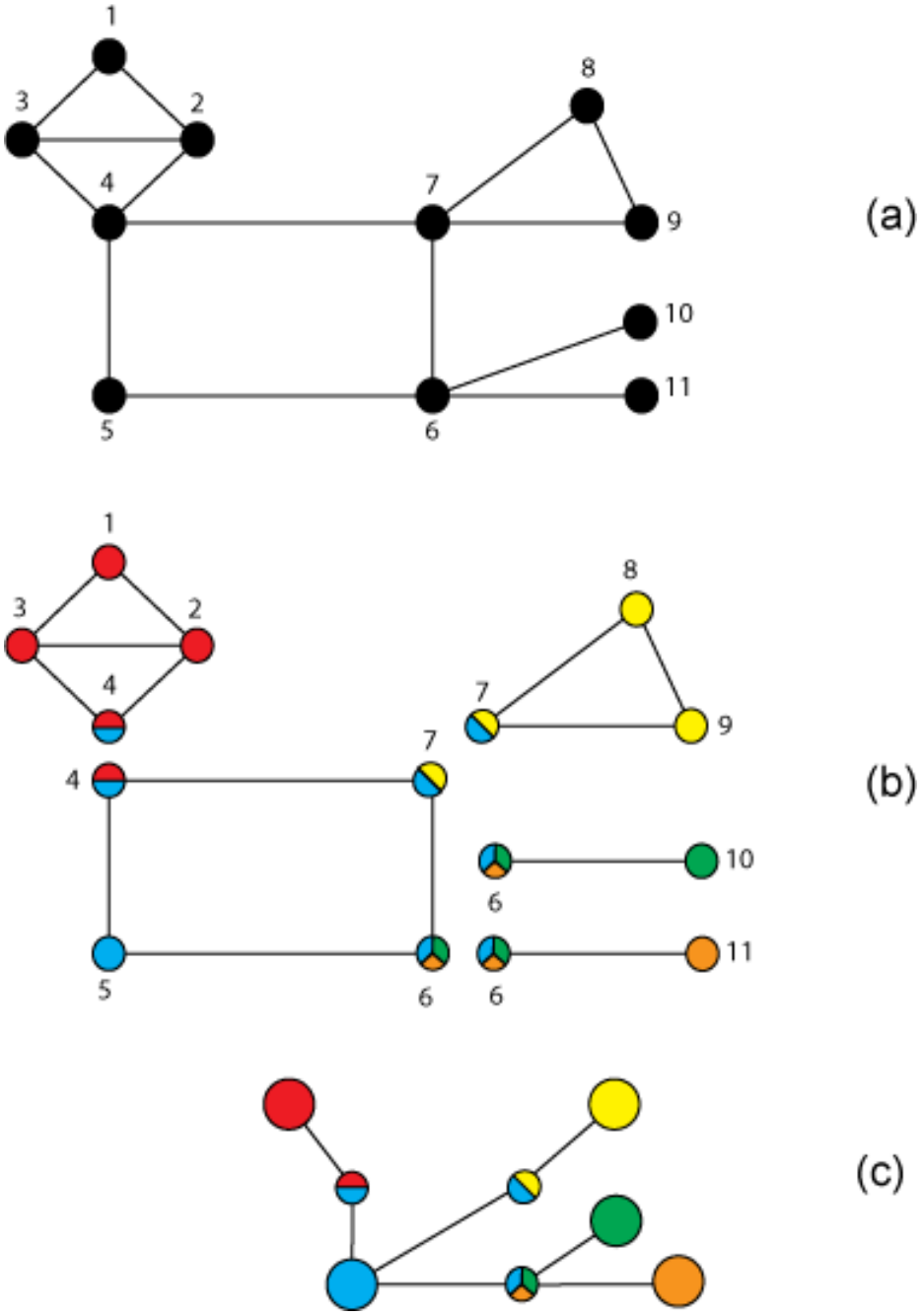
分析

- 找边双连通分量
- 缩点
- 建树,找叶结点 leaf
- $\text{ans} = (\text{leaf} + 1) \div 2$

点双连通分量的缩点

点双连通分量缩点比边双连通分量缩点要复杂一些，因为一个割点可能属于多个点双连通分量。设图中共有 P 个割点和 T 个点双连通分量，则缩点后的新图将有 $P+T$ 个节点，每个割点和点双连通分量都作为新图中的一个节点，在每个割点与包含它的所有点双连通分量之间连边。由于新图中不包含环路，因此是一棵树或者森林。

下图(a) 所示的无向连通图，(b) 是划分出来的 5 个点双连通分量，有 3 个割点。(c) 是缩点后生成的新图，有 8 个结点，其中大点表示原图中的点双连通分量，小点表示原图中的割点。



缩点建新图时，需要对每个点双连通分量中的每条边赋上双连通分量的编号(bcc)，对每个割点也赋上双连通分量的编号。

对于点双中的每条边赋编号，可以在递归的过程中进行，即一旦发现一个割点，就退栈输出点双的每条边时，对边赋上编号。

对于割点，则先标点那些点是割点，递归结束后，再枚举所有节点，对其中的割点赋上编号。

在建新图时，枚举每条边，仅需要对割点关联的边，取出割点的bcc和边的bcc，在新图上建新边。具体代码见例题。

例题 HDU3686 Traffic Real Time Query System

题目大意

给出一个n个点构成的图，求从一条边a到另一个条边b必须经过的点的数量。

分析

很明显必须经过的点就是a到b路径上的割点，把图中的每个点双连通通分量缩成一个点，那么全图就缩成了一棵树，树上有两类点，一种是割点，另一种是边双连通分量缩成的点，现在求的就是u，v这两个点所在的块经过最近公共祖先的路径上有多少个割点。

对于两条边X，Y，若他们在同一个双连通分量中，那他们之间至少有两条路径可以互相到达。

那么，对于两条不在同一个分量中的边X，Y，显然从X到Y必须要经过的的点的数目等于从X所在的边双连通分量到Y所在的双连通分量中的割点的数目。

于是，可以找出所有双连通分量，缩成一个点。对于 双连通分量A - 割点 - 双连通分量B，重构图成3个点 A - 割点 - B。

那么，所有的双连通分量缩完之后，新图G'成了一棵树（若存在环，那么环里的点都在同一个双连通分量中，矛盾）

那么题目变成了：从X所在的G'中的点，到Y所在的G'中的点的路径中，有多少点是由割点变成的。

由于图G'中的点都是 双连通分量 - 割点 - 双连通分量 - 割点 - 双连通分量.....的形式（两个双连通分量不会连在一起）

那么X到Y的割点的数目就等于两点距离除以2，暨$(dep[x] + dep[Y] - dep[LCA(X, Y)])/2$，其中$dep$是深度，$lca$是最近公共祖先。

```cpp
/*
https://www.cnblogs.com/oyking/p/3645984.html
AC 2025.7.10
*/
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
typedef long long LL;

typedef pair<int, int> PII;

const int MAXV = 10010;
const int MAXE = 200010;

int ans[MAXV];
vector<PII> query[MAXV << 1];

struct SccGraph {
  int head[MAXV << 1], fa[MAXV << 1], ecnt;
  bool vis[MAXV << 1];
  int to[MAXE << 1], nxt[MAXE << 1];
  int dep[MAXV << 1];

  void init(int n) {
    memset(head, -1, sizeof(int) * (n + 1));
    memset(vis, 0, sizeof(bool) * (n + 1));
```

```
    for (int i = 1; i <= n; ++i)
      fa[i] = i;
    ecnt = 0;
  }

  int find_set(int x) { return fa[x] == x ? x : fa[x] = find_set(fa[x]); }

  void add_edge(int u, int v) {
    to[ecnt] = v;
    nxt[ecnt] = head[u];
    head[u] = ecnt++;
    to[ecnt] = u;
    nxt[ecnt] = head[v];
    head[v] = ecnt++;
  }

  void lca(int u, int f, int deep) {
    dep[u] = deep;
    for (int p = head[u]; ~p; p = nxt[p]) {
      int &v = to[p];
      if (v == f || vis[v])
        continue;
      lca(v, u, deep + 1);
      fa[v] = u;
    }
    vis[u] = true;
    for (vector<PII>::iterator it = query[u].begin(); it != query[u].end();
         ++it) {
      if (vis[it->first]) {
        ans[it->second] =
            (dep[u] + dep[it->first] - 2 * dep[find_set(it->first)]) / 2;
      }
    }
  }
} G;

int head[MAXV], low[MAXV], dfn[MAXV], ecnt, dcnt;
int bcc[MAXV], bcnt;
int to[MAXE], nxt[MAXE], scc_edge[MAXE];
bool vis[MAXE], iscut[MAXV];
int stk[MAXE], top;
int n, m, q;

void init() {
  memset(head, -1, sizeof(int) * (n + 1));
  memset(dfn, 0, sizeof(int) * (n + 1));
  memset(iscut, 0, sizeof(bool) * (n + 1));
  memset(vis, 0, sizeof(bool) * (2 * m));
  ecnt = bcnt = dcnt = 0;
}

void add_edge(int u, int v) {
  to[ecnt] = v;
  nxt[ecnt] = head[u];
  head[u] = ecnt++;
  to[ecnt] = u;
```

```
  nxt[ecnt] = head[v];
  head[v] = ecnt++;
}

void tarjan(int u, int f) {
  dfn[u] = low[u] = ++dcnt;
  int child = 0;
  for (int p = head[u]; ~p; p = nxt[p]) {
    if (vis[p])
      continue;
    vis[p] = vis[p ^ 1] = true;
    stk[++top] = p;
    int &v = to[p];
    if (!dfn[v]) {
      ++child;
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (dfn[u] <= low[v]) {
        iscut[u] = true;
        ++bcnt;
        while (true) {
          int t = stk[top--];
          scc_edge[t] = scc_edge[t ^ 1] = bcnt;
          if (t == p)
            break;
        }
      }
    } else
      low[u] = min(low[u], dfn[v]);
  }
  if (f < 1 && child == 1)
    iscut[u] = false;
}

void build() {
  G.init(bcnt);
  for (int p = 0; p != ecnt; ++p) {
    int &v = to[p];
    if (iscut[v])
      G.add_edge(bcc[v], scc_edge[p]);
  }
}

void solve() {
  for (int i = 1; i <= n; ++i)
    if (!dfn[i])
      tarjan(i, 0);
  for (int u = 1; u <= n; ++u)
    if (iscut[u])
      bcc[u] = ++bcnt;
}

int main() {
  while (scanf("%d%d", &n, &m) != EOF) {
    if (n == 0 && m == 0)
      break;
```

```
    init();
    for (int i = 1; i <= m; ++i) {
      int u, v;
      scanf("%d%d", &u, &v);
      add_edge(u, v);
    }
    solve();
    build();
    for (int i = 0; i <= bcnt; ++i)
      query[i].clear();
    scanf("%d", &q);
    for (int i = 0; i < q; ++i) {
      int x, y;
      scanf("%d%d", &x, &y);
      x = scc_edge[x * 2 - 2];
      y = scc_edge[y * 2 - 2];
      query[x].push_back(make_pair(y, i));
      query[y].push_back(make_pair(x, i));
    }
    for (int i = 1; i <= bcnt; ++i)
      if (!G.vis[i])
        G.lca(i, 0, 0);
    for (int i = 0; i < q; ++i)
      printf("%d\n", ans[i]);
  }
}
```

例题 POJ2942/SPOJ KNIGHTS 圆桌骑士

题目大意

有 n 个骑士经常举行圆桌会议，商讨大事。每次圆桌会议至少有 3 个骑士参加，且相互憎恨的骑士不能坐在圆桌的相邻位置。如果发生意见分歧，则需要举手表决，因此参加会议的骑士数目必须是大于 1 的奇数，以防止赞同和反对票一样多。知道骑士之间相互憎恨的关系后，请你帮忙统计有多少骑士参加不了任意一个会议。

分析

以骑士为节点建立无向图 G。如果两个骑士不相互憎恨，则可以相邻而坐，在他们之间连一条无向边。

可以坐在一起组成一个会议的骑士数目必须是大于 1 的奇数，那么这些骑士将形成一个长度为奇数的简单环（简称为奇环）。有些骑士可能形成多个奇环，表示他们可以组成多个会议。

题目转化成求 图G 中不在任何一个奇环上的节点个数。如果 G 不连通，则对每一个连通分量分别求解。以下讨论，假定 G 是连通的。

简单环上的所有节点必然属于同一个双连通分量(点双？边双？)，因此需要先找出 G 中所有双连通分量。如何判断一个双连通分量中的节点是否在一个奇环上呢？

有一个关键结论：**如果双连通分量中包含一个奇环，则所有点都在至少一个简单奇环上。**

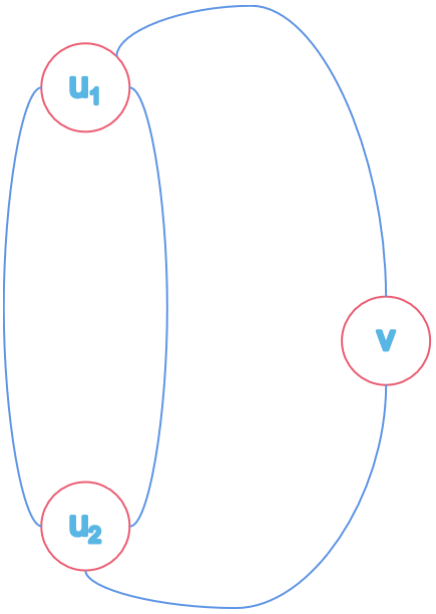
证明如下：

首先，找出双连通分量中的任意一个奇环，如何找见后文。

易证，一个简单奇环$\mathbb{C}$上任意两点 u_1, u_2 ，一定将奇环分解成一条长度为奇数的路径和一条长度为偶数的简单路径，这两条路径的节点与边均不相交。

显然，双连通分量上不在奇环 $\mathbb{C}$ 上的节点被分散成若干个连通分量，设其中一个分量为 X。对于任意节点 $v \in X$， 一定存在连接奇环上任意两点 u_1, u_2 的简单路径。若不是简单路径，则路径上一定存在割点，这与双连通分量的定义矛盾。

如果 $u_1 \to v \to u_2$ 这条路径长度为奇数，则与奇环 $\mathbb{C}$ 上 u_1, u_2 间长度为偶数的路径可以拼成奇环，否则和 u_1, u_2 间长度为奇数的路径可以拼成奇环。



这就证明了如果一个双连通分量上存在一个奇环，则所有点都在至少一个奇环上。

如果判断点双连通分量中是否存在奇环呢?

在双连通分量上做二分图染色，如果染色成功，则没有奇环。否则，存在奇环。

整理一下本题的算法。先根据输入的憎恨关系建补图，补图上跑Tarjan算法求点双，对于每一个点双二分图染色判断是否有奇环，统计不在任何一个有奇环的点双上的点，这些点都无法参加会议。

```cpp
/*
SPOJ KNIGHTS
ac
2024.5.12
370 ms
*/
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <stack>
#include <vector>
using namespace std;
const int N = 1005;
int n, m, dcnt, bcnt;
int dfn[N], low[N], bel[N], col[N], g[N][N];
bool cut[N], ok[N];
vector<int> adj[N];
struct node {
  int u, v;
```

```
};
stack<node> st;
bool dfs(int u, int id) { // 对一个点双做二分图染色检验，是二分图返回 true
  for (int v : adj[u]) {
    if (bel[v] != id)
      continue;
    if (col[v] == col[u])
      return false;
    if (!col[v]) {
      col[v] = 3 - col[u];
      if (!dfs(v, id))
        return false;
    }
  }
  return true;
}
void tarjan(int u, int fa) {
  dfn[u] = low[u] = ++dcnt;
  int rson = 0;
  for (int v : adj[u]) {
    if (!dfn[v]) {
      st.push({u, v});
      rson++;
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (dfn[u] <= low[v]) {
        cut[u] = true;
        bcnt++;
        while (true) {
          auto e = st.top();
          st.pop();
          if (bel[e.u] != bcnt)
            bel[e.u] = bcnt;
          if (bel[e.v] != bcnt)
            bel[e.v] = bcnt;
          if (e.u == u && e.v == v)
            break;
        }
        col[u] = 1;
        if (!dfs(u, bcnt)) // 如果存在奇环，则可行
          for (int i = 1; i <= n; i++)
            if (bel[i] == bcnt)
              ok[i] = true;
        col[u] = 0; // for cut vertex
      }
    } else if (v != fa) {
      if (dfn[v] < dfn[u])
        st.push({u, v});
      low[u] = min(low[u], dfn[v]);
    }
  }
  if (rson == 1 && fa == 0)
    cut[u] = false;
}
void init() {
  dcnt = 0;
```

```
  bcnt = 0;
  memset(g, 0, sizeof g);
  memset(dfn, 0, sizeof dfn);
  memset(bel, 0, sizeof bel);
  memset(col, 0, sizeof col);
  memset(cut, 0, sizeof cut);
  memset(ok, 0, sizeof ok);
  for (int i = 1; i <= n; i++)
    adj[i].clear();
}
void build() {
  for (int i = 1; i <= n; i++)
    for (int j = i + 1; j <= n; j++)
      if (!g[i][j])
        adj[i].push_back(j), adj[j].push_back(i);
}
int main() {
  while (scanf("%d%d", &n, &m) != EOF && (n || m)) {
    init();
    for (int i = 1; i <= m; i++) {
      int u, v;
      scanf("%d%d", &u, &v);
      g[u][v] = g[v][u] = 1;
    }
    build();
    for (int i = 1; i <= n; i++)
      if (!dfn[i])
        tarjan(i, 0);
    int ans = 0;
    for (int i = 1; i <= n; i++)
      if (!ok[i])
        ans++;
    printf("%d\n", ans);
  }
  return 0;
}
```

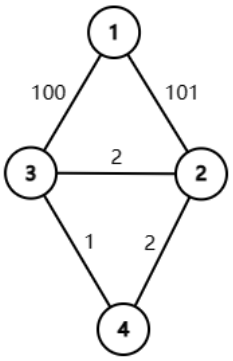
综合例题

例题 CF160D Edges in MST

题目大意

我们知道，最小生成树在有些图中是不唯一的。给出一个无向连通加权图，有 N 个点，M 条边，没有自环和重边。请判断这 M 条边，是以下三种类型中的哪一种：

1. 如果它在所有最小生成树方案中都存在，输出 `any`；
2. 如果它在所有最小生成树方案中都不存在，输出 `none`；
3. 如果它至少出现在一种最小生成树的方案中，输出 `at least one`.



如图所示，按照权值从大到小排列的 5 条边，所属类型分别是：

```
none
any
at least one
at least one
any
```

分析

有一个结论，设 T 是图 G 的一棵最小生成树，L 是 T 中一个边权值的有序列表，那么对于 G 的任何其他最小生成树 T'，L 也是 T' 中一个边权值的有序列表，参见《算法导论》P365 练习23.1-8。

因此，在图中权值相同的若干条边中，属于最小生成树的**边的数量**是确定的。这些权值相同的边有可能在MST中互相替换，也可能必须在MST中，也可能不存在于任何MST中。详细分析如下：

考虑Kruskal算法执行过程。首先按边权值的非降序对所有边进行排序，然后处理每一条边。如果某条边连接了两个不同的连通分量，那么将这条边加入 MST 并合并这两个连通分量。这里使用**并查集**来维护连通性。

根据以上结论，需要将相同权值的所有边一起处理。现在，在每一步中，我们面临一些权值相同的边和一个用小于当前权值的边构成的连通分量的森林。对于一条边来说，它连接的具体是哪两个节点并不重要，重要的是需要知道它的两个端点所属的连通分量。

对于当前这一步，如果被检查的一条边的两个端点已经属于同一个连通分量，那么这条边不可能添加到MST中，因此类型一定是 `none`。否则，这条边就可以被添加到MST中。但如何进一步区分是哪种类型呢?

现在构建一个新图 G'，其中 G' 中的每个节点代表森林中的一个连通分量，当前这一步的权值相同的边被加入到新图 G' 中，连接连通分量对应的节点。显然，这些边都添加后，新图 G' 不会形成环路。这时，对新形成的连通分量跑一遍 DFS 找出其中的割边，则这些割边一定存在于每一类MST中，类型就是 `any`。而不是割边的类型，则就只能是 `at least one`.

分析时间复杂度。考虑到那些与当前这一步新加边没有任何连接的连通分量不需要出现在 G' 中，所以 Tarjan 算法的 DFS 时间复杂度为 $O(|E|)$，其中 $|E|$ 是当前权值的边的数量。因为每条边在 G' 中只会被使用一次，除了排序之外的总时间复杂度是 $O(M)$。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 5;
struct node {
  int u, v, id, fu, fv;
};
int n, m, dcnt;
```

```cpp
int dfn[N], low[N], fa[N], ans[N];
bool brige[N];
vector<node> edge[N];
vector<pair<int, int>> adj[N];
int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }
void dfs(int u, int id) {
  dfn[u] = low[u] = ++dcnt;
  for (auto &e : adj[u]) {
    if (e.second == id)
      continue;
    if (!dfn[e.first]) {
      dfs(e.first, e.second);
      low[u] = min(low[u], low[e.first]);
      if (dfn[u] < low[e.first])
        brige[e.second] = true;
    } else
      low[u] = min(low[u], dfn[e.first]);
  }
}
int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++)
    fa[i] = i;
  for (int i = 1; i <= m; i++) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    edge[w].push_back({u, v, i, 0, 0});
  }
  for (int i = 1; i <= 1000000; i++) {
    if (!edge[i].size())
      continue;
    for (auto &e : edge[i]) {
      e.fu = find(e.u);
      e.fv = find(e.v);
      if (e.fu > e.fv)
        swap(e.fu, e.fv);
    }
    for (auto &e : edge[i]) {
      if (e.fu == e.fv) {
        ans[e.id] = 3;
      } else {
        adj[e.fu].clear();
        adj[e.fv].clear();
        dfn[e.fu] = dfn[e.fv] = 0;
      }
    }
    for (auto &e : edge[i]) {
      if (e.fu != e.fv) {
        adj[e.fu].push_back({e.fv, e.id});
        adj[e.fv].push_back({e.fu, e.id});
      }
    }
    for (auto &e : edge[i]) {
      if (e.fu != e.fv) {
        if (!dfn[e.fu])
          dfs(e.fu, 0);
```

```
        if (!dfn[e.fv])
          dfs(e.fv, 0);
      }
    }
    for (auto &e : edge[i]) {
      if (e.fu != e.fv) {
        if (brige[e.id])
          ans[e.id] = 1;
        else
          ans[e.id] = 2;
      }
    }
    for (auto &e : edge[i]) {
      int x = find(e.u);
      int y = find(e.v);
      if (x != y)
        fa[x] = y;
    }
  }
  for (int i = 1; i <= m; i++) {
    if (ans[i] == 1)
      puts("any");
    else if (ans[i] == 2)
      puts("at least one");
    else
      puts("none");
  }
  return 0;
}
```

参考链接

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.