连通图的割点、割边(桥)、块、缩点，有向图的强连通分量

一、基本概念

无向图

割点：删掉它之后(删掉所有跟它相连的边)，图必然会分裂成两个或两个以上的子图。

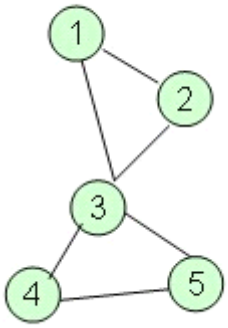
块：没有割点的连通子图

割边：删掉一条边后，图必然会分裂成两个或两个以上的子图，又称桥。

缩点：把没有割边的连通子图缩为一个点，此时满足任意两点间都有两条路径相互可达。

求块跟求缩点非常相似，很容易搞混，但本质上完全不同。割点可以存在多个块中(假如存在 k 个块中)，最终该点与其他点形成 k 个块，对无割边的连通子图进行缩点后(假设为 k 个)，新图便变为一棵 k 个点由 k-1 条割边连接成的树，倘若其中有一条边不是割边，则它必可与其他割边形成一个环，而能继续进行缩点。

有割点的图不一定有割边，如：



3 是割点，分别与(1,2)和(4,5)形成两个无割点的块

有割边的图也不定有割点,如：

w(1,2)为割边，

有向图

强连通分量：有向图中任意两点相互可达的连通子图，其实也相当于无向图中的缩点

二、算法

无向图

借助两个辅助数组 dfn[],low[]进行 DFS 便可找到无向图的割点和割边，用一个栈 st[]维护记录块和“缩点”后连通子图中所有的点。

dfn[i]表示 DFS 过程中到达点 i 的时间，low[i]表示能通过其他边回到其祖先的最早时间。low[i]=min(low[i],dfn[son[i]])

设 v,u 之间有边 w(v,u)，从 v->u：

如果 low[u]>=dfn[v],说明 v 的儿子 u 不能通过其他边到达 v 的祖先，此时如果拿掉 v,则必定把 v 的祖先和 v 的儿子 u，及它的子孙分开，于是 v 便是一个割点，v 和它的子孙形成一个块。

如果 low[u]>dfn[v]时，则说明 u 不仅不能到达 v 的祖先，连 v 也不能通过另外一条边直接到达，从而它们之间的边 w(v,u)便是割边，求割边的时候有一个重边的问题要视情况处理，如果 v,u 之间有两条无向边，需要仍视为割边的话，则在 DFS 的时候加一个变量记录它的父亲,下一步遇到父结点时不扩展回去，从而第二条无向重边不会被遍历而导致 low[u]==dfn[v]，而在另外一些问题中，比如电线连接两台设备 A,B 如果它们之间有两根电线，则应该视为是双连通的，因为任何一条电线出问题都不会破坏 A 和 B 之间的连通性，这个时候，我们可以用一个 used[]数组标记边的 id,DFS 时会把一条无向边拆成两条有向边进行遍历,但我们给它们俩同一个 id 号，在开始遍历 v->u 前检查它的 id 是否在上一次 u->v 时被标记，这样如果两点之间有多条边时，每次遍历都只标记其中一条，还可以通过其他边回去，形成第二条新的路

求割点的时候，维护一个栈 st 每遍历到一个顶点 v 则把它放进去，对它的子孙 u 如果 dfn[u]为 0 ,则表示还没有遍历到则先 DFS(u),之后再判断 low[u]和 dfn[v]，如果 low[u]>=dfn[v],则把栈中从栈顶到 v 这一系列元素弹出 ,这些点与 v 形成一个块，如果 u 的子孙 x 也是一个割点，这样做会不会错把它们

和 v，u 放在一起形成一个块呢，这种情况是不会发生的，如果发现 x 是一个割点，则 DFS 到 x 那一步后栈早就把属于 x 的子孙弹出来了，而只剩下 v,u 的子孙，它们之间不存在割点，否则在回溯到 v 之前也早就提前出栈了！画一个图照着代码模拟一下可以方便理解。

求割边也是一样的。

有向图

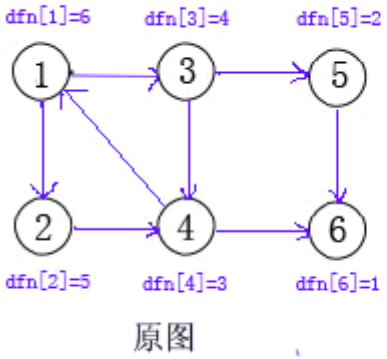
有向图强连通分量的算法有两个，一个是 Kosaraju,另一个是 Tarjan，前者需要两次 DFS ,代码量偏大但容易理解 ,后者只需要一次 DFS 和维护一个栈便可以，实现简单，详见这里>>

强连通分量 Kosaraju PK Tarjan

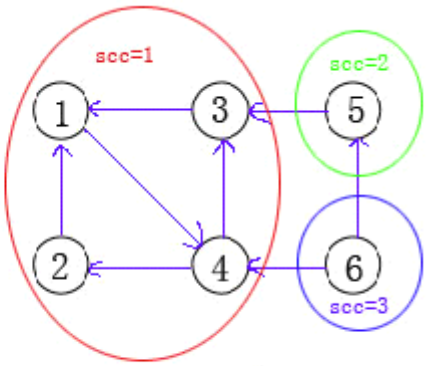
2010-04-22 16:23

Kosaraju 算法

对每个不在树中的点开始 DFS 一次，并记录离开各点的时间，这里是离开的时间，而不是到达时的，比如有图 1->2 2->3 则 1,2,3 分别对应的时间是 3 2 1，因为 3 没有出边，所以最先离开，其次是 2,最后是 1，



原图



逆图

DFS 后，在同一棵树中的点，如果 dfn[v]>dfn[u]则说明点从 v 有可能到达 u，而这棵树中的 dfn[]最大的点，肯定可以到达每个点，从而在原图的逆图中，每次都选没有访问过的最大的 dfn 值开始 DFS，如果可达点 x 则说明它们是强连通的

```
void DFS_T(int u)
{
    int i,v;
    if(used[u])return ;
    used[u]=1;id[u]=scc;
    for(i=q[u];i!=-1;i=Tedge[i].pre)
    {
        v=Tedge[i].d;
        if(!used[v])DFS_T(v);
    }
}
```

```
void DFS(int v){
    int i,u;
    if(used[v])return ;
    used[v]=1;
    for(i=p[v];i!=-1;i=edge[i].pre)
    {
        u=edge[i].d;
        if(!used[u])DFS(u);
    }
    order[++num]=v;
}
int Kosaraju()
{
    int i,j,k,v,u;
    memset(used,0,sizeof(used));num=0;
    for(i=1;i<=n;++i)if(!used[i])DFS(i);
    memset(used,0,sizeof(used));
    memset(id,0,sizeof(id));scc=0;
    for(i=num;i>=1;--i)if(!used[order[i]])scc++,DFS_T(order[i])
;

}
```

Tarjan 算法

dfn[v]记录到达点 v 的时间，跟上面的离开不同，low[v]表示通过它的子结点可以到达的所有点中时间最小值，即 low[i]=min(low[i],low[u]),u 为 v 的了孙，初始化时 low[v]=dfn[u]。如果 low[v]比 dfn[v]小，说明 v 可以通过它的子结点 u,u1,u2...到达它的祖先 v',则存在环，这个环上所有的点组成的子图便是一个强连通分量。换一个角度看，如果当 low[v]==dfn[v]时，则它的子树中所有 low[u]==dfn[v]的点都与 v 构成一个环，维护一个栈，DFS 过程中，每遍历一个点则把它放入栈中，当发现 low[v]==dfn[v]则依次把栈里的元素都弹出来，当栈顶元素为 v 时结束，这些点便构成一个以 v 为树根的强连通分量。

仍以上图为例，首先遍历点 1,并 dfn[1]=low[1]=++num，num 表示按先后访问时间编号 ,同时 1 入栈

a.从 3 深入 dfn[3]=low[3]=2; 3 入栈

b.从 3 到 5 dfn[5]=low[5]=3; 5 入栈

c.从 5 到 6 dfn[6]=low[6]=4; 6 入栈

d.发现 6 没有子结点可走，这时判断 dfn[6]==low[6],于是开始弹栈，当遇到 6 时则 break,即共弹出一个元素，于是 6 便是一个强连通分量

e.回溯至 5,同样判断和弹栈，发现 5 也是一个强连通分量

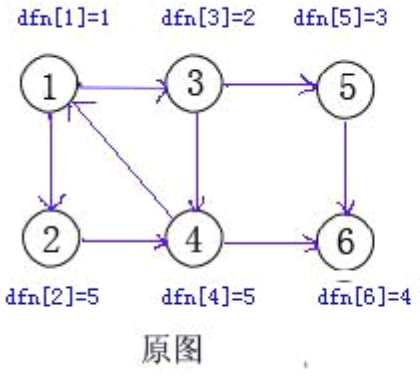
f.再回溯至 3,发现有边 3->4，dfn[4]=low[4]=5,4 入栈

g.4 有边到 1,由于 1 已经在栈里面，所以用 dfn[1]更新 low[4] 即 low[4]=min(low[4],dfn[1])=1

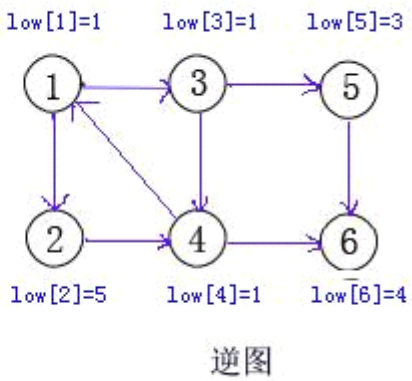
h.回溯更新 4 的父亲 3 的 low 值 low[3]=min(low[3],low[4])=1

i.再回溯至 1,发现有边 1->2 继续深度遍历，2 入栈,发现它的子结点 4 已经在栈中，直接更新 low[2]=min(low[2],dfn[4]);

j.回溯至 1,从而 1 所有出发的边都走了一遍，这时再比较 low[1]与 dfn[1]，发现相等，于是开始弹栈，找到 2,4,3,1 这四个元素，构成一个连通分量。



原图



逆图

```
void Tarjan(int v){
    dfn[v]=low[v]=++num;
    used[v]=1;
    st[++numSt]=v;
    for(int i=p[v];i!=-1;i=edge[i].pre){
        int u(edge[i].d);
```

```
        if(!dfn[u])//还没有标号的点
        {
            Tarjan(u);//先遍历它的子结点
            GetMin(low[v],low[u]);//用子结点更新当前点的 low 值
        }
        else if(used[u]&&GetMin(low[v],dfn[u]));
    }
    if(dfn[v]==low[v]){
        scc++;
        while(1){
            int u(st[numSt--]);
            id[u]=scc;
            used[u]=0;
            if(v==u)break;
        }
    }
}

int main(){

for(int i=1;i<=n;++i)if(!dfn[i])Tarjan(i);

}
```

这里有一个疑问，为什么当发现一个点 v 的子结点 u 已经在栈中时用 dfn[u]来更新 low[v],而不是用 low[u],感觉好象两个都可以用，因为只要保证 low[v]尽可能变小就行了，

三、代码实现

割点和块

求割点的时候由于不知道最开始选的树根是不是只有一个儿子，这样在 DFS 过来中不会满足 low[u]>=dfn[v]而判为割点，但有两个或两个以上儿子的根肯定也是一个割点，所以要特判！

```
void CutBlock(int v){
    dfn[v]=low[v]=++cnt;
    st[++top]=v;
    for(int i=p[v];i!=-1;i=edge[i].pre){
        int u(edge[i].d);
        if(dfn[u]==0){
            CutBlock(u);
            GetMin(low[v],low[u]);
            if(low[u]>=dfn[v]){ //V 是一个割点
                block[0]=0;
                while (true) {
```

```
                    block[++block[0]]=st[top];
                    if (st[top--] == u) //只能弹到 u 为止，v 还可以
在其他块中
                        break;
                }
                block[++block[0]]=v;//割点属于多个块，一定要补进去

                Count(block);
            }
        }
        else GetMin(low[v],dfn[u]);
    }
}
```

割边和缩点

```
void CutEdge(int v,int fa){
    dfn[v]=low[v]=++cnt;
    st[++top]=v;
    for(int i=p[v];i!=-1;i=edge[i].pre){
        int u(edge[i].d);
        if(u==fa)continue;
        if(!dfn[u]){
```

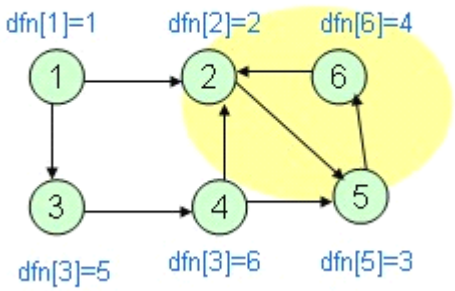
```
            CutEdge(u,v);
            GetMin(low[v],low[u]);
            if(low[u]>dfn[v]){//边 v->u 为一条割边
                cutE[++numE]=E(v,u);
                // 将 u 及与它形成的连通分量的所有点存起来
                ++numB;
                while(1){
                    id[st[top]]=numB;
                    if(st[top--]==u)break;
                }
            }
        }
        else GetMin(low[v],dfn[u]);
    }
}
```

有向图强连通分量

```
void Tarjan(int v){
    dfn[v]=low[v]=++num;
    used[v]=1;
    st[++numSt]=v;
    for(int i=p[v];i!=-1;i=edge[i].pre){
```

```
        int u(edge[i].d);
        if(!dfn[u])//还没有标号的点
        {
            Tarjan(u);//先遍历它的子结点
            GetMin(low[v],low[u]);//用子结点更新当前点的 low 值
        }
        else if(used[u]&&GetMin(low[v],dfn[u]));
    }
    if(dfn[v]==low[v]){
        scc++;
        while(1){
            int u(st[numSt--]);
            id[u]=scc;
            used[u]=0;
            if(v==u)break;
        }
    }
}
```

这里需要注意一个地方，上面标记为紫色的那行代码，相比上面两个代码，这里加了一个 used[]判断点是否在栈中，为什么前面的不要，而这里需要呢，举个例子



根据 dfn 可以看出搜索的顺序是 1->2->5->6 形成一个强连通分量(2,5,6) ,于是开始退栈,回溯到 1 从 3 出发到达 4,此时如果直接用 dfn[2]更新 low[4]的话，会得到 low[4]=2,变小后而与 dfn[4]不再相等，不能退栈，这与最后的 4 形成一个单独强连通分量是不符合的，所以，不在栈中的点，不能用来更新当前点的 low[]值，为什么无向图不用标记呢，那时因为，边是无向的，有边从 4->2 同时也必有边 2->4 由于 2 之前被标记过 ,而遍历到当前结点 4 又不是通过 w(2,4) 这条边过来的，则必还存在另一条路径可以使 2 和 4 是相通的，(即图中的 4-3-1-2),从而 2,4 是双连通的。

其实以上三个算法都源于 Tarjan，只是根据 dfn[]和 low[]判断条件不同而得到不同的结果，无限 Orz Trajan ！！！

于是，顺便总结一下 **LCA 的离线算法**

离线是指把所有的问题都存起来，类似邻接表的形式，能根据点 v 找到它关的点 u,处理完后一次性回答所有的答案。

DFS 到 v 时，用 used[]标记为已访问，下面分两部分完成

1、在 Q 个查询中对所有与 v 相关连的 ui,uj,uk,如果检查发现 used[]为真，则说明它们的最近公共祖先为 ui 当前能往上最大程度找到的祖先，这个可借助并查集实现，记录结果用以后面输出。

2、对 v 所有子结点 u(不同于上面的 u),进行 DFS() ，DFS 结束后，设置 u 的父亲为 v，即 fa[u]=v；

时间复杂度为 O(m+Q)，Q 为查询的总数，dist[]记录根到当前点的距离，如果最后要求任意两点 v 和 u 之间的距离，则为

dist[v]+dist[u]-2*dist[lca(v,u)]

```
void LcaTarjan(int v){
    used[v]=1;
    fa[v]=v;
    for(int i=q[v];i!=-1;i=e[i].pre){ //对跟 v 相关每个问题，尝试进行回答
        int u(e[i].d),id(e[i].id);
        if(used[u])ans[id]=Find(u);
    }
    for(int i=p[v];i!=-1;i=edge[i].pre){
        int u(edge[i].d),w(edge[i].w);
        if(!used[u]){
            dist[u]=dist[v]+w;
            LcaTarjan(u);
```

```
            fa[u]=v;
        }
    }
}
```

四、例题

pku1523>>

先求割点，第二问其实就是求块，一个割点存在 k 个块中，删掉后，便形成 k 个子图

pku2942>>

求块后，对每块有：如果存在奇圈，则可以分开开会，否则全 T 掉，判断奇圈可以用 DFS 二分染色的方法，当前点染为白色，它所有相邻点染为黑色，如果最后发现某条边两个端点同色，则存在奇圈。

pku3694>>

求割边后，并标记，这时新图形成一棵树，但并不需要缩点，否则反而不好处理，每加一条边 w(v,u)进去，必会形成一个圈，剩下的问题但是如何找圈，事先求出 v,u 的最小公共祖先，加入边 w(v,u)后，则这个圈的一部分便是从 v 到 lca(v,u)之间的树边，另一部分是 u 到 lca(v,u)之间的树边，由于一个图中割边的总条数不会超过 n，所以可用割边关联的两个顶点中的一个来记录它的位置，这样在沿 v 或 u 向 lca(v,u)往上找时，快速判断它与它父亲之间相连的边

是否为割边，是的话 ans-- 并标记为非，因为 w(v,u)的加入形成了环，环中原来所有的割边都会变成非割边。用 fa[v]表示 v 的父亲，set[v]表示 v 的祖先，虽然初始都表示 v 的父亲，但在 LCA 时要区分使用，一个只记录它的直接父亲，另一个并查集时压缩路径会改变。

pku 3352>> pku3177>>

求割边，缩点后，形成一棵树，统计度为 1 的点个数 t，需要连的边数则为 (t+1)/2 ,pku3177 只是多了重边处理，方法见上。

hdu3394>> 求块，如果一个块的顶点数等于边数，则这个块只有一个环，如果边数大于点数，则必有多个环，容易知道在一个 K 环的块中，每条边也必属于 K 个环,这样可以计算出在一个环和多个环里的边总数 ,剩下的便是不在环中的边。

hdu 3394 Railway 无向图求块

2010-06-02 22:07

hdu 3394 >>

求在 0 个环、1 个环、多个环里的边的条数

问题转化为无向图求块，即缩块。块是不存在割点的连通子图，如果一个块的顶点数等于边数，则这个块只有一个环，如果边数大于点数，则必有多个环，容易知道在一个 K 环的块中，每条边也必属于 K 个环,这样可以计算出在一个环和多个环里的边总数，剩下的便是不在环中的边。

```
#define arr         10010
```

```
#define brr         500010
struct Edge{
    int d,pre;
    Edge(){}
    Edge(int d1,int pre1):d(d1),pre(pre1){}
}edge[brr];
int p[arr];
int pn;
int dfn[arr];
int low[arr];
bool used[arr];
int cnt;
int block[arr];
int n,m;
int none,one,two;
int st[arr];
int top;
int fa[arr];
void Insert(int v,int u){
    edge[++pn]=Edge(u,p[v]);p[v]=pn;
    edge[++pn]=Edge(v,p[u]);p[u]=pn;
}
```

```
void Count(int* block){
    FF(i,block[0])used[block[i]]=1;
    int sum(0);
    FF(i,block[0]){
        int v(block[i]);
        for(int j=p[v];j!=-1;j=edge[j].pre){
            int u(edge[j].d);
            if(used[u])sum++;
        }
    }
    sum/=2;
    if(sum==block[0])//点和边总数一样多，刚好一个环
        one+=sum;
    else if(sum>block[0])//边比点多，存在多个环
        two+=sum;
    else none+=sum;
    FF(i,block[0])used[block[i]]=0;
}
void DFS(int v){
    dfn[v]=low[v]=++cnt;
    st[top++]=v;
    for(int i=p[v];i!=-1;i=edge[i].pre){
```

```
        int u(edge[i].d);
        if(dfn[u]==0){
            fa[u]=v;
            DFS(u);
            GetMin(low[v],low[u]);
            if(low[u]>=dfn[v]){
                block[0]=0;
                while (true) {
                    block[++block[0]]=st[top-1];
                    if (st[--top] == u)
                        break;
                }
                block[++block[0]]=v;
                Count(block);
            }
        }
        else if(u!=fa[v]) GetMin(low[v],dfn[u]);
    }
}
void Work(){
    clr(dfn,0);clr(low,0);
    clr(used,0);
```

```
    cnt=none=one=two=top=0;
    FF(i,n){
        if(dfn[i]==0)DFS(i);
    }
    printf("%d %d\n",none,two);
}
int main(){
    while(scanf("%d%d",&n,&m)!=EOF){
        if(n+m==0)break;
        clr(p,-1);pn=0;
        FF(i,m){
            int v,u;
            scanf("%d%d",&v,&u);
            v+=1;u+=1;
            Insert(v,u);
        }
        Work();
    }
    return 0;
}
```