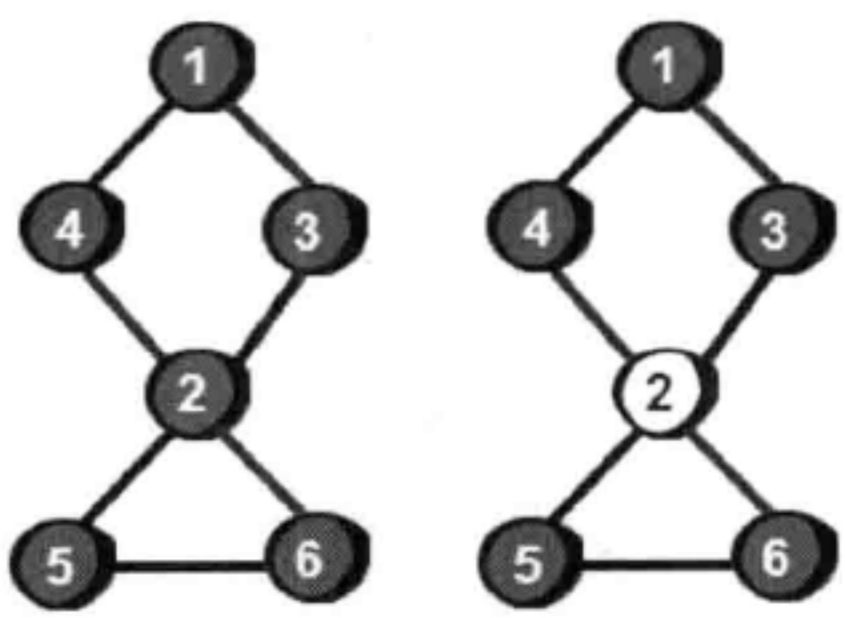
第 3 节　重要城市——图的割点

一场大战即将开始……

我们已经掌握了敌人的城市地图，为了在战争中先发制人，决定向敌人的某个城市上空投放炸弹，来切断敌人城市之间的通讯和补给，城市地图如下。



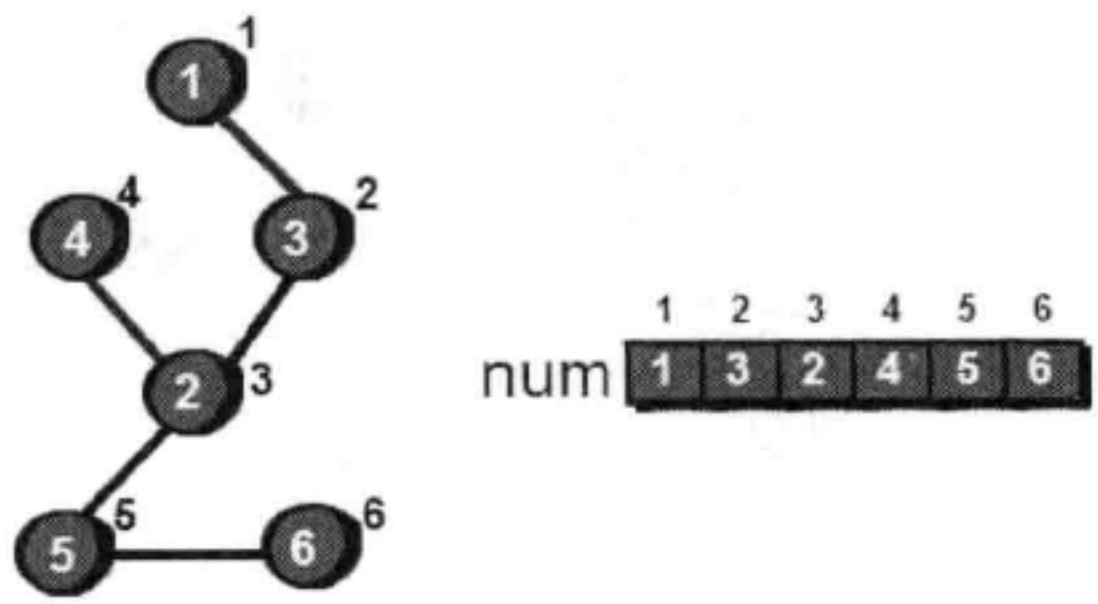
我们可以炸毁 2 号城市，这样剩下的城市之间就不能两两相互到达了。例如 4 号城市不能到 5 号城市，6 号城市也不能到达 1 号城市等等。

搞清楚问题后，下面将问题抽象化。在一个无向连通图中，如果删除某个顶点后，图不

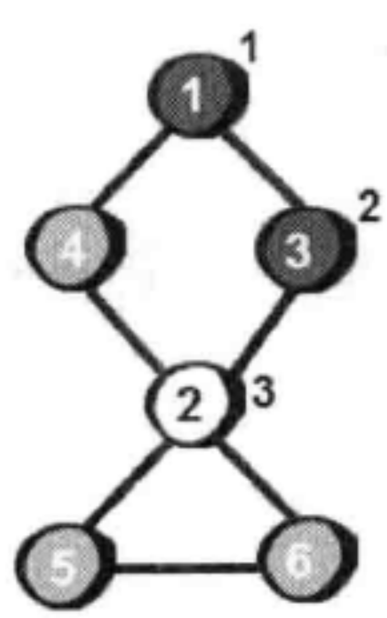
再连通（即任意两点之间不能相互到达），我们称这样的顶点为割点（或者称割顶）。那么割点如何求呢？

很容易想到的方法是：依次删除每一个顶点，然后用深度优先搜索或者广度优先搜索来检查图是否依然连通。如果删除某个顶点后，导致图不再连通，那么刚才删除的顶点就是割点。这种方法的时间复杂度是 $O(N(N+M))$。想一想有没有更好的方法呢？能找到线性的方法吗？

首先我们从图中的任意一个点（比如 1 号顶点）开始对图进行遍历（遍历的意思就是访问每个顶点），比如使用深度优先搜索进行遍历，下图就是一种遍历方案。从图中可以看出，对一个图进行深度优先遍历将会得到这个图的一个生成树（并不一定是最小生成树），如下图。有一点需要特别说明的是：下图中圆圈中的数字是顶点编号，圆圈右上角的数表示的是这个顶点在遍历时是第几个被访问到的，这还有个专有名词叫做“时间戳”。例如 1 号顶点的时间戳为 1，2 号顶点的时间戳为 3……我们可以用数组 num 来记录每一个顶点的时间戳。

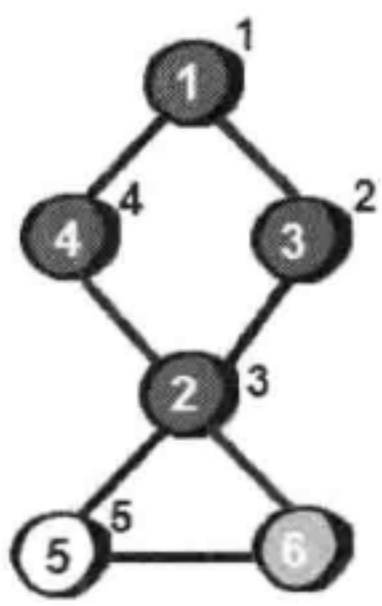


我们在遍历的时候一定会遇到割点（这不是废话吗），关键是如何认定一个顶点是割点呢。假如我们在深度优先遍历时访问到了 k 点，此时图就会被 k 点分割成为两部分。一部分是已经被访问过的点，另一部分是没有被访问过的点。如果 k 点是割点，那么剩下的没有被访问过的点中至少会有一个点在不经过 k 点的情况下，是无论如何再也回不到已访问过的点了。那么一个连通图就被 k 点分割成多个不连通的子图了，下面来举例说明。

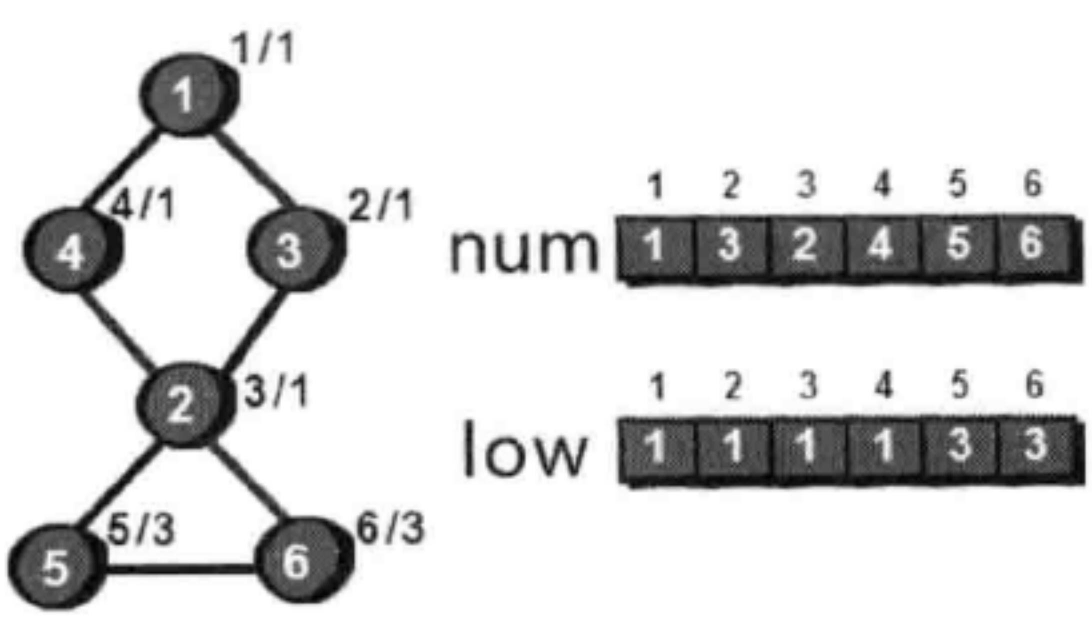


上图是深度优先遍历访问到 2 号顶点的时候。此时还没有被访问到的顶点有 4、5、6 号顶点。其中 5 和 6 号顶点都不可能在不经过 2 号顶点的情况下，再次回到已被访问过的顶点（1 和 3 号顶点），因此 2 号顶点是割点。

这个算法的关键在于：当深度优先遍历访问到顶点 u 时，假设图中还有顶点 v 是没有访问过的点，如何判断顶点 v 在不经过顶点 u 的情况下是否还能回到之前访问过的任意一个点？如果从生成树的角度来说，顶点 u 就是顶点 v 的父亲，顶点 v 是顶点 u 的儿子，而之前已经被访问过的顶点就是祖先。换句话说，如何检测顶点 v 在不经过父顶点 u 的情况下还能否回到祖先。我的方法是对顶点 v 再进行一次深度优先遍历，但是此次遍历时不允许经过顶点 u，看看还能否回到祖先。如果不能回到祖先则说明顶点 u 是割点。再举一个例子，请看下图。



上图是深度优先遍历访问到 5 号顶点的时候，图中只剩下 6 号顶点还没有被访问过。现在 6 号顶点在不经过 5 号顶点的情况下，可以回到之前被访问过的顶点有：1、3、2 和 4 号顶点。我们这里只需要记录它能够回到最早顶点的"时间戳"。"时间戳"这个概念我们在第 5 章第 1 节就已经遇到过。对于 6 号顶点来说就是记录 1。因为 6 号顶点能够回到的最早顶点是 1 号顶点，而 1 号顶点的时间戳为 1（圆圈右上方的数）。为了不重复计算，我们需要一个数组 low 来记录每个顶点在不经过父顶点时，能够回到的最小"时间戳"。如下图。



对于某个顶点 u，如果存在至少一个顶点 v（即顶点 u 的儿子），使得 low[v]>=num[u]，即不能回到祖先，那么 u 点为割点。对于本例，5 号顶点的儿子只有 6 号顶点，且 low[6]的值为 1，而 5 号顶点的“时间戳”num[5]为 5，low[6]<num[5]，可以回到祖先，因此 5 号顶点不是割点。2 号顶点的“时间戳”num[2]为 3，它的儿子 5 号顶点的 low[5]为 3，low[5]==num[3]，表示 5 号顶点不能绕过 2 号顶点从而访问到更早的祖先，因此 2 号顶点是割点。完整的代码实现如下。

```
#include <stdio.h>
int n,m,e[9][9],root;
int num[9],low[9],flag[9],index;//index用来进行时间戳的递增
//求两个数中较小一个数的函数
int min(int a,int b)
{
    return a < b ? a : b;
}
//割点算法的核心
void dfs(int cur,int father)//需要传入的两个参数，当前顶点编号和父顶点的编号
{
    int child=0,i,j; //child用来记录在生成树中当前顶点cur的儿子个数

    index++;//时间戳加1
    num[cur]=index;//当前顶点cur的时间戳
    low[cur]=index;//当前顶点cur能够访问到最早顶点的时间戳，刚开始当然是自己啦
    for(i=1;i<=n;i++)//枚举与当前顶点cur有边相连的顶点i
    {
        if(e[cur][i]==1)
        {
            if(num[i]==0)//如果顶点i的时间戳不为0，说明顶点i还没有被访问过
            {
                child++;
                dfs(i,cur);//继续往下深度优先遍历
                //更新当前顶点cur能否访问到最早顶点的时间戳
                low[cur]=min(low[cur],low[i]);
                //如果当前顶点不是根结点并且满足low[i] >= num[cur]，则当前顶点为割点
                if(cur!=root && low[i] >= num[cur])
                    flag[cur]=1;
                //如果当前顶点是根结点，在生成树中根结点必须要有两个儿子，那么这个根
结点才是割点
```

```
                if(cur==root && child==2)
                    flag[cur]=1;

            }
            else if(i!=father)
            //否则如果顶点i曾经被访问过，并且这个顶点不是当前顶点cur的父亲，则需要更新当前结点cur能否访问到最早顶点的时间戳
            {
                low[cur]=min(low[cur],num[i]);
            }
        }
    }
}
int main()
{
    int i,j,x,y;
    scanf("%d %d",&n,&m);
    for(i=1;i <= n;i++)
        for(j=1;j <= n;j++)
            e[i][j]=0;

    for(i=1;i <= m;i++)
    {
        scanf("%d %d",&x,&y);
        e[x][y]=1;
        e[y][x]=1;
    }
    root=1;
    dfs(1,root);//从1号顶点开始进行深度优先遍历

    for(i=1;i <= n;i++)
    {
        if(flag[i]==1)
            printf("%d ",i);
    }

    getchar();getchar();
    return 0;
}
```

可以输入以下数据进行验证。第一行有两个数 n 和 m，n 表示有 n 个顶点，m 表示有 m 条边。接下来 m 行，每行形如"a b"表示顶点 a 和顶点 b 之间有边。

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

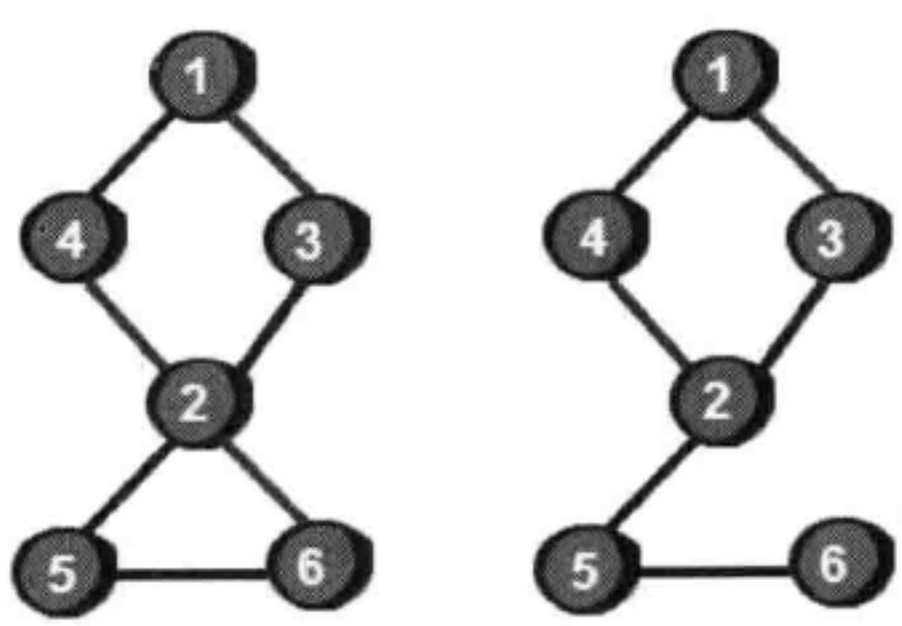
运行结果是：

```
2
```

细心的同学会发现，上面的代码是用的邻接矩阵来存储图，这显然是不对的，因为这样无论如何时间复杂度都会在 $O(N^2)$，因为边的处理就需要 N^2 的时间。这里这样写是为了突出割点算法部分，实际应用中需要改为使用邻接表来存储，这样整个算法的时间复杂度是 $O(N+M)$。

第 4 节　关键道路——图的割边

上一节我们解决了如何求割点，还有一种问题是如何求割边（也称为桥），即在一个无向连通图中，如果删除某条边后，图不再连通。下图中左图不存在割边，而右图有两条割边分别是 2-5 和 5-6。



很明显，将 2-5 这条边删除后图被分割成了两个子图，如下。