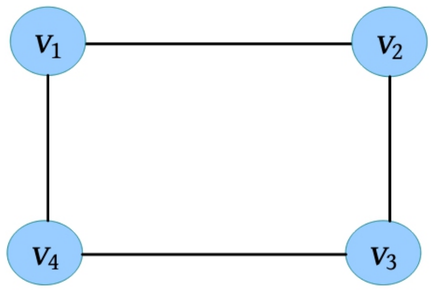
图的连通性

一、连通性的相关知识

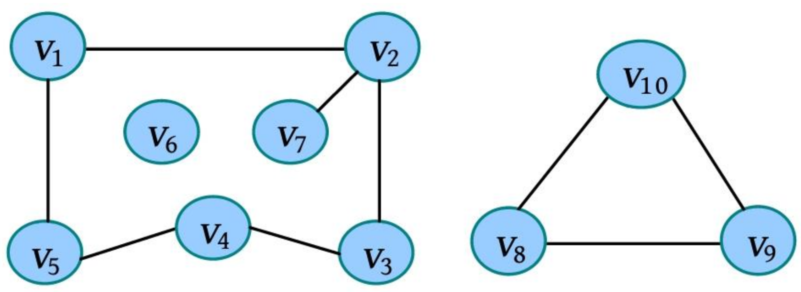
1. 无向图的连通分量

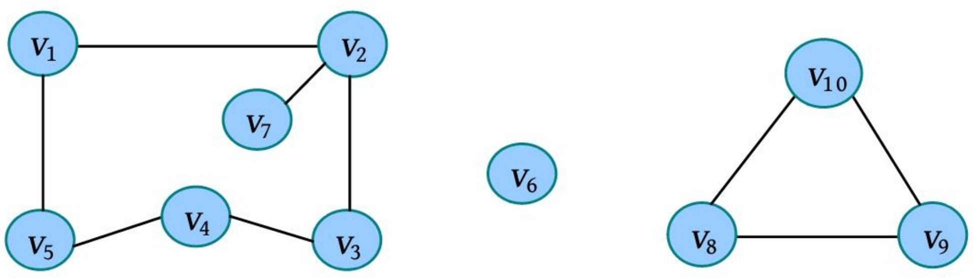
在无向图中，如果从节点 v_i 到节点 v_j 有路径，则称节点 v_i 和节点 v_j 是连通的。如果**图中任意两个节点都是连通的，则称图 G 为连通图**。如下图所示就是一个连通图。



无向图 G 的极大连通子图被称为图 G 的连通分量。极大连通子图是图 G 连通子图，如果再向其中加入一个节点，则该子图不连通。连通图的连通分量就是它本身；非连通图则有两个以上的连通分量。

例如在下图中有 3 个连通分量。





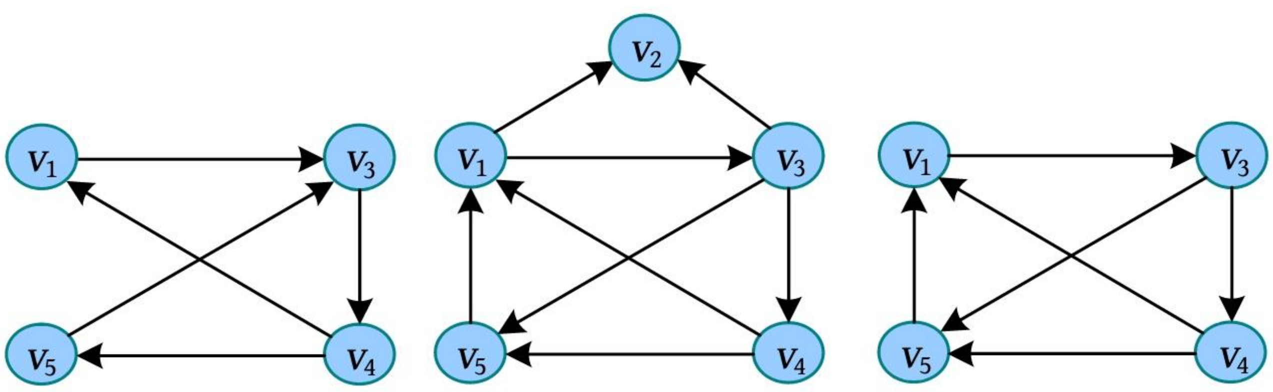
(a) 连通分量1　　(b) 连通分量2　　(c) 连通分量3

2. 有向图的强连通分量

在有向图中，如果**图中的任意两个节点从 v_i 到 v_j 都有路径，且从 v_j 到 v_i 也有路径，则称图 G 为强连通图**。

有向图 G 的极大强连通子图被称为图 G 的强连通分量。极大强连通子图是图 G 的强连通子图，如果再向其中加入一个节点，则该子图不再是强连通的。

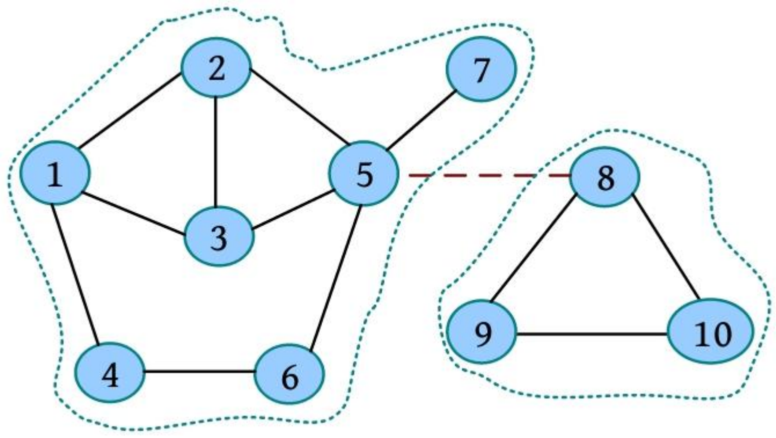
例如在下图中，(a)是强连通图，(b)不是强连通图，(c)是(b)的强连通分量。



(a)　　(b)　　(c)

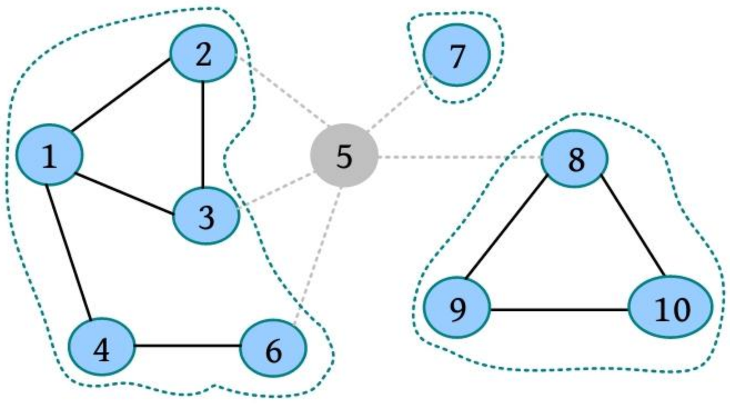
3. 无向图的桥与割点

在生活中，桥是连接河两岸的交通要道，桥断了，则河两岸不再连通。在图论中，桥有同样的含义，如下图所示，去掉边 5-8 后，图分裂成两个互不连通的子图，边 5-8 为图 G 的桥。同样，边 5-7 也为图 G 的桥。



如果在**去掉无向连通图 G 中的一条边 e 后，图 G 分裂为两个不相连的子图**，那么 **e 为图 G 的桥或割边**。

在日常网络中有很多路由器使网络连通，有的路由器坏掉也无伤大雅，网络仍然连通，但若非常关键节点的路由器坏了，则网络将不再连通。如下图所示，如果节点 5 的路由器坏了，图 G 将不再连通，会分裂成 3 个不相连的子图，则节点 5 为图 G 的割点。



如果在去掉无向连通图 G 中的一个点 v 及与 v 关联的所有边后，图 G 分裂为两个或两个以上不相连的子图，那么 v 为图 G 的割点。注意：删除边时，只把该边删除即可，不要删除与边关联的点；而删除点时，要删除该点及其关联的所有边。

割点与桥的关系：①有割点不一定有桥，有桥一定有割点；②桥一定是割点依附的边。

4. 无向图的双连通分量

如果在无向图中不存在桥，则称它为边双连通图。在边双连通图中，在任意两个点之间都存在两条及以上路径，且路径上的边互不重复。

如果在无向图中不存在割点，则称它为点双连通图。在点双连通图中，如果节点数大于 2，则在任意两个点间都存在两条或以上路径，且路径上的点互不重复。

无向图的极大边双连通子图被称为**边双连通分量**，记为 **e-DCC**。

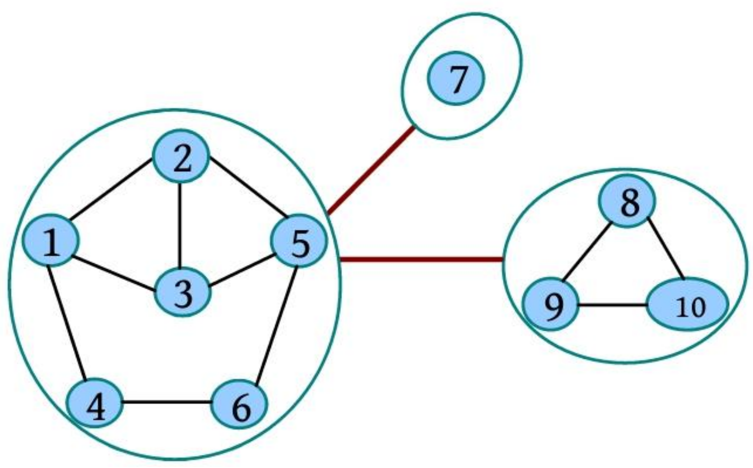
无向图的极大点双连通子图被称为**点双连通分量**，记为 **v-DCC**。

二者被统称为双连通分量 DCC。

5. 双连通分量的缩点

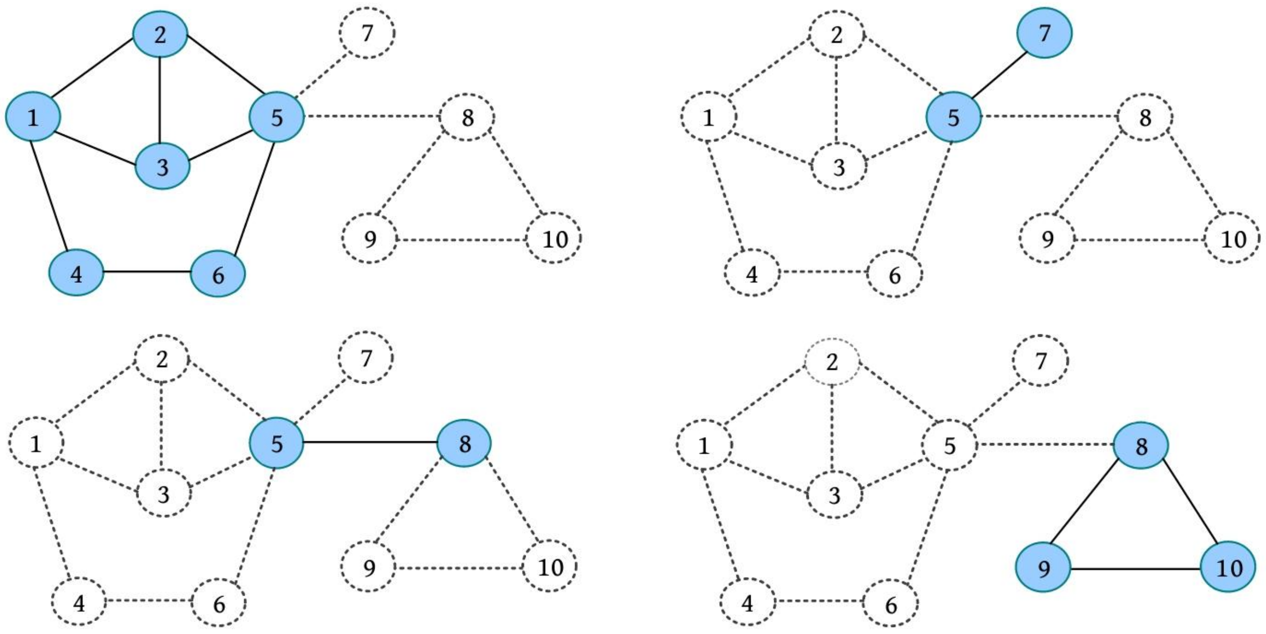
把**每一个边双连通分量 e-DCC 都看作一个点，把桥看作连接两个缩点的无向边，可得到一棵树**，这种方法被称为 **e-DCC 缩点。**

例如，在下图中有两个桥：5-7 和 5-8，将每个桥的边都保留，将桥两端的边双连通分量缩为一个点，生成一棵树。



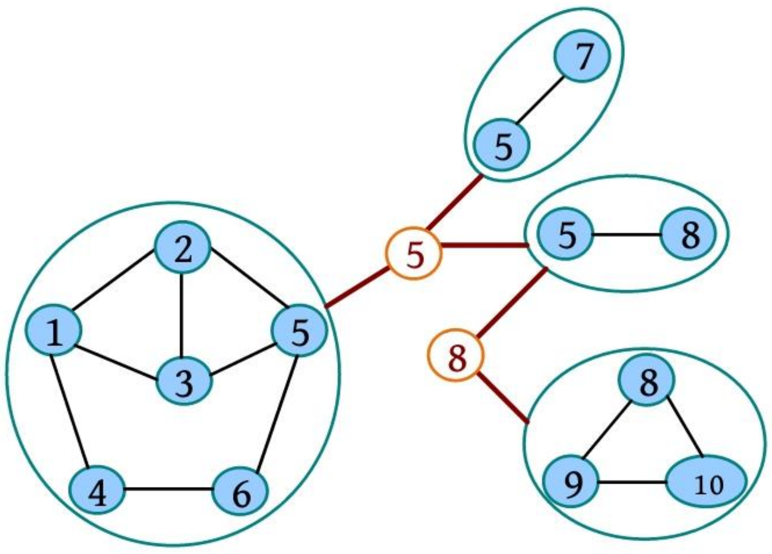
注意：边双连通分量就是删除桥之后留下的连通块，但**点双连通分量并不是删除割点后留下的连通块。**

在图 G 中有两个割点（5 和 8）及 4 个点双连通分量，如下图所示。



把每一个点双连通分量 v-DCC 都看作一个点，把割点看作一个点，每个割点都向包含它的 v-DCC 连接一条边，得到一棵树，这种方法被称为 **v-DCC 缩点。**

例如，在图 G 中有两个割点 5、8，前 3 个点双连通分量都包含 5，因此从 5 向它们引一条边，后两个点双连通分量都包含 8，因此从 8 向它们引一条边，如下图所示。



二、 Tarjan 算法

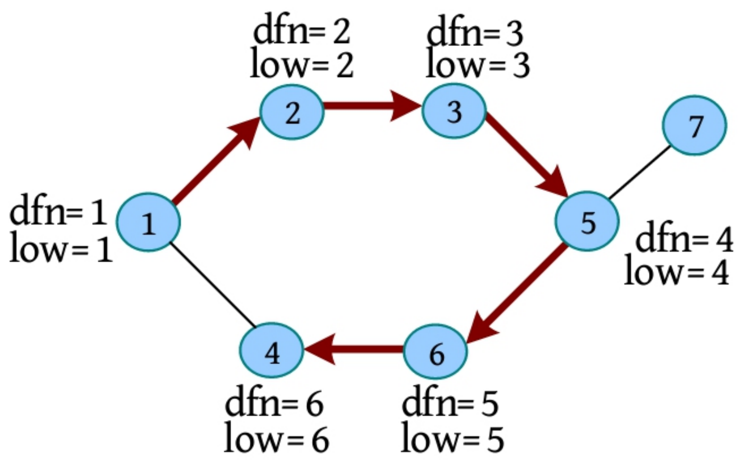
Robert Tarjan 以在数据结构和图论上的开创性工作而闻名，他的一些**著名算法包括 Tarjan 最近公共祖先离线算法、Tarjan 强连通分量算法**及 **Link-Cut-Trees 算法**等。其中，**Hopcroft-Tarjan 平面嵌入算法**是第 1 个线性时间平面算法。Robert Tarjan 也开创了重要的数据结构，例如**斐波纳契堆和 Splay 树**，另一项重大贡献是分析了**并查集。**

在介绍算法之前，首先引入时间戳和追溯点的概念。

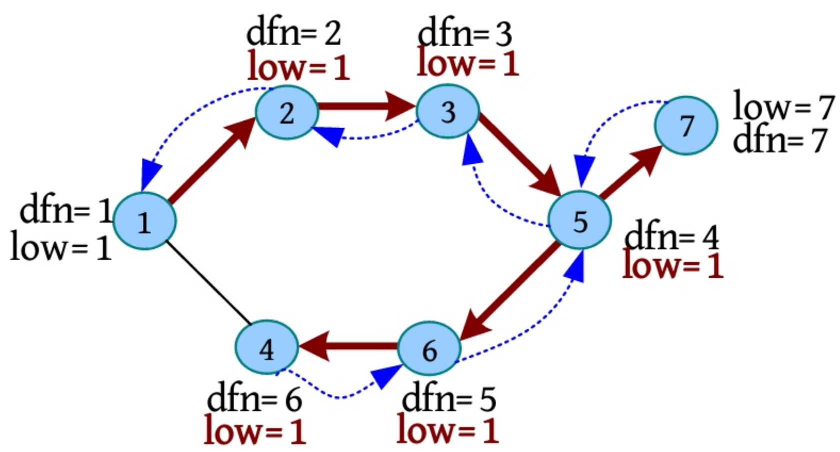
- **时间戳：dfn[u]表示节点 u 深度优先遍历的序号。**
- **追溯点：low[u]表示节点 u 或 u 的子孙能通过非父子边追溯到的 dfn 最小的节点序号，即回到最早的过去。**

例如，在深度优先搜索中，每个点的时间戳和追溯点的求解过程如下。

初始时，dfn[u]=low[u]，如果该节点的邻接点未被访问，则一直进行深度优先遍历，1-2-3-5-6-4，此时 4 的邻接点 1 已被访问，且 1 不是 4 的父节点，4 的父节点是 6（深度优先搜索树上的父节点）。



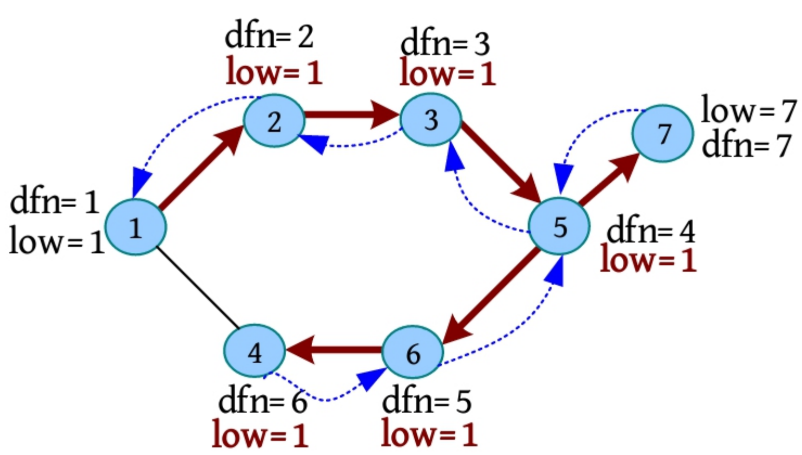
那么节点 4 能回到最早的节点是节点 1（dfn=1），因此 low[4]=min(low[4],dfn[1])=1。返回时，更新 low[6]=min(low[6],low[4])=1。更新路径上所有祖先节点的 low 值，因为子孙能回到的追溯点，其祖先也能回到。



1. 无向图的桥

桥判定法则：无向边 x-y 是桥，当且仅当在搜索树上存在 x 的一个子节点 y 时，满足 low[y]>dfn[x]。

也就是说，若孩子的 low 值比自己的 dfn 值大，则从该节点到这个孩子的边为桥。在下图中，边为 5-7，5 的子节点为 7，满足 low[7]>dfn[5]，因此边 5-7 为桥。



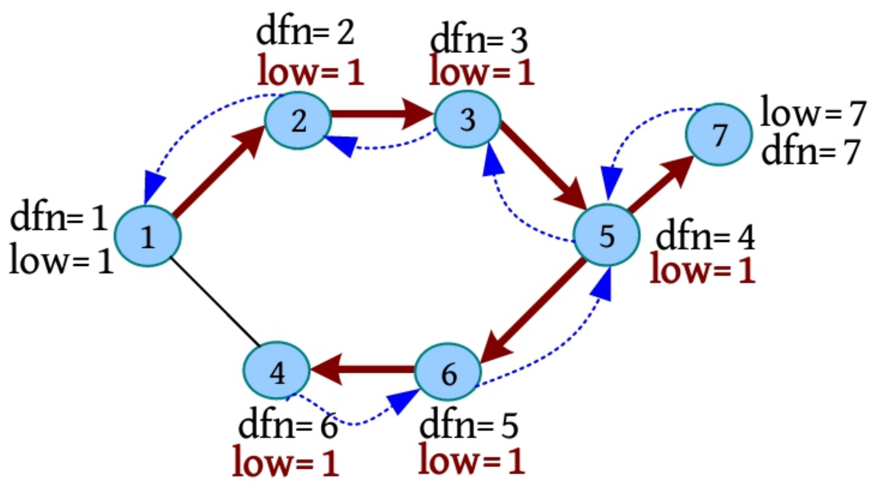
算法代码：

```
void tarjan(int u,int fa){
    dfn[u]=low[u]=++num;
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(v==fa)
            continue;
        if(!dfn[v]){
            tarjan(v,u);
            low[u]=min(low[u],low[v]);
            if(low[v]>dfn[u])
                cout<<u<<"—"<<v<<"是桥"<<endl;
        }
        else
            low[u]=min(low[u],dfn[v]);
    }
}
```

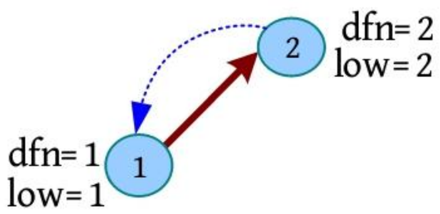
2. 无向图的割点

割点判定法则：**若 x 不是根节点，则 x 是割点，当且仅当在搜索树上存在 x 的一个子节点 y，满足 low[y]≥dfn[x]；若 x 是根节点，则 x 是割点，当且仅当在搜索树上至少存在两个子节点，满足该条件。**

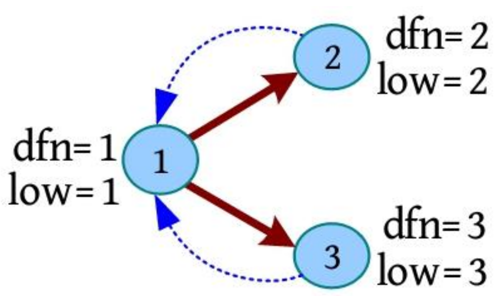
也就是说，如果不是根，且孩子的 low 值大于或等于自己的 dfn 值，则该节点就是割点；如果是根，则至少需要两个孩子满足条件。在下图中，5 的子节点是 7，满足 low[7]>dfn[5]，因此 5 是割点。



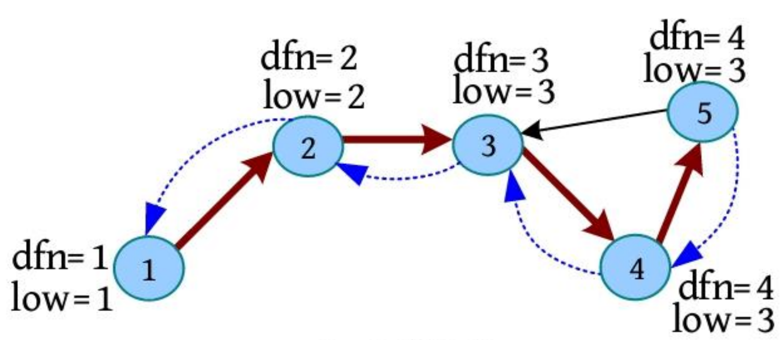
有几种割点判定情况，如下图所示：



1不是割点（1是根，只有一个孩子满足low[2]>dfn[1]）



1是割点（1是根，有两个孩子满足low[2]>dfn[1], low[3]>dfn[1]）



2、3是割点

（low[3]>dfn[2]，low[4]=dfn[3]）

算法代码：

```
void tarjan(int u,int fa){//求割点
    dfn[u]=low[u]=++num;
    int count=0;
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(v==fa)
            continue;
        if(!dfn[v]){
            tarjan(v,u);
            low[u]=min(low[u],low[v]);
            if(low[v]>=dfn[u]){
                count++;
                if(u!=root||count>1)
                    cout<<u<<"是割点"<<endl;
            }
        }
        else
            low[u]=min(low[u],dfn[v]);
    }
}
```

3. 有向图的强连通分量

算法步骤：

（1）深度优先遍历节点，在第 1 次访问节点 x 时，将 x 入栈，且 dfn[x]=low[x]=++num。

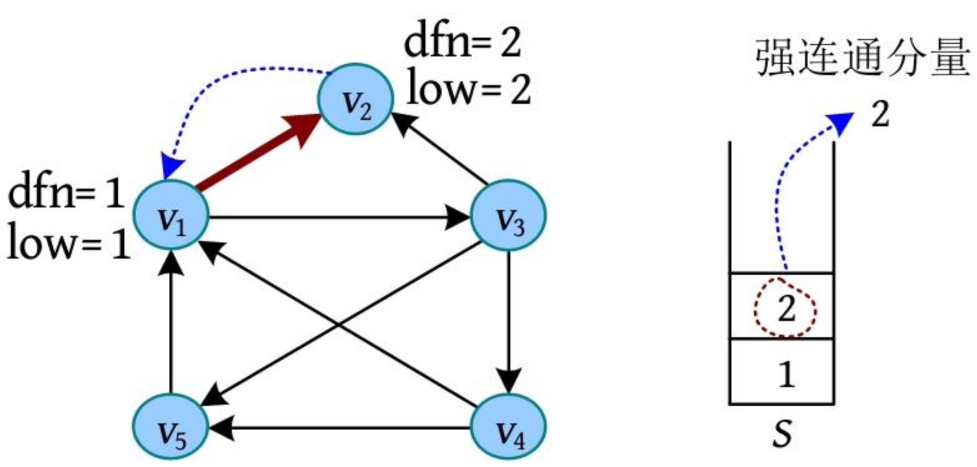
（2）遍历 x 的所有邻接点 y。

- 若 y 没被访问，则递归访问 y，返回时更新 low[x]=min(low[x],low[y])。
- 若 y 已被访问且在栈中，则令 low[x]=min(low[x],dfn[y])。

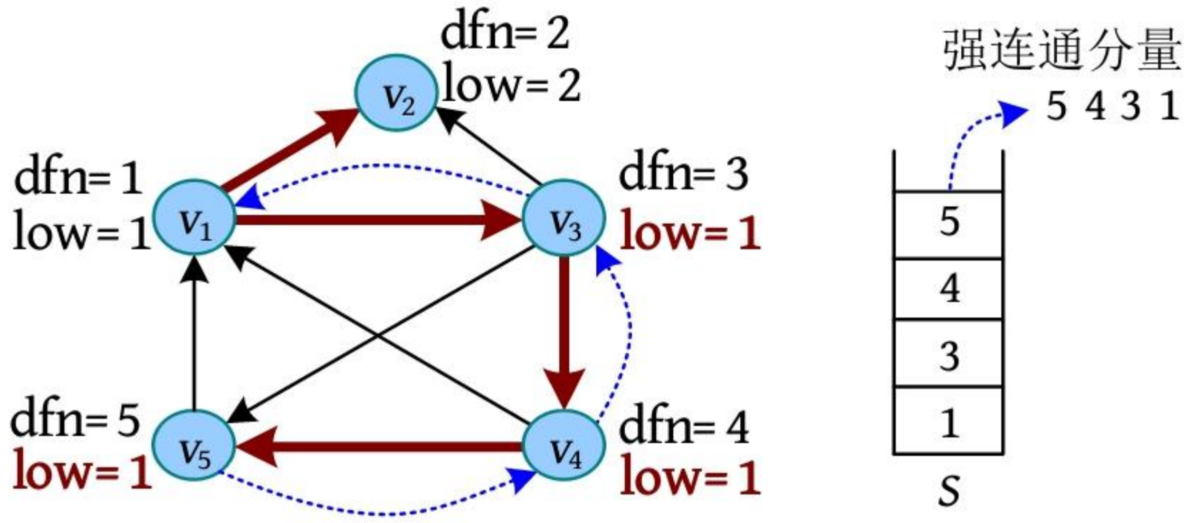
（3）在 x 回溯之前，如果判断 low[x]=dfn[x]，则从栈中不断弹出节点，直到 x 出栈时停止。**弹出的节点就是一个连通分量。**

例如，求解有向图的强连通分量，过程如下。

（1）从节点 1 出发进行深度优先搜索，dfn[1]=low[1]=1，1 入栈；dfn[2]=low[2]=2，2 入栈；此时无路可走，回溯。因为 dfn[2]=low[2]，所以 2 出栈，得到强连通分量 2。



（2）回溯到 1 后，继续访问节点 1 的下一个邻接点 3。接着访问 3-4-5，5 的邻接点 1 的 dfn 已经有解，且 1 在栈中，更新 low[5]=min(low[5],dfn[1])=1。回溯时更新 low[4]=min(low[4],low [5])=1，low[3]=min(low[3],low[4])=1，low[1]=min(low[1],low[3])=1。节点 1 的所有邻接点都已访问完毕，因为 dfn[1]=low[1]，所以开始出栈，直到遇到 1，得到强连通分量 5 4 3 1。




```
void tarjan(int u){//求强连通分量
    low[u]=dfn[u]=++num;
    ins[u]=true;
    s.push(u);
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(!dfn[v]){
            tarjan(v);
            low[u]=min(low[u],low[v]);
        }
        else if(ins[v])
            low[u]=min(low[u],dfn[v]);
    }
    if(low[u]==dfn[u]){
        int v;
        cout<<"强连通分量: ";
        do{
            v=s.top();
            s.pop();
            cout<<v<<" ";
            ins[v]=false;
        }while(v!=u);
        cout<<endl;
    }
}
```