双连通分量（DCC）——Tarjan

原理

和 SCC 一样都是在 dfs 生成树上记录 `dfn` 和 `low`，在环中编号最小的。

需要做一下概念上的区分：

1. SCC（强连通分量） 是针对有向图的概念。一个SCC内的任意两个节点都可以互相到达（即存在双向路径）。
   DCC（双连通分量） 是针对无向图的概念，但它主要分为两种：
   - **点双连通分量**：其内不存在割点（即删除任一节点后，分量仍然连通）。同一个点双内的任意两点间至少存在两条**点不相交**的路径。
   - **边双连通分量**：其内不存在桥（即删除任一边后，分量仍然连通）。同一个边双内的任意两点间至少存在两条**边不相交**的路径。
2. **缩点**（将每个分量收缩为一个节点）后：
   - SCC 缩点会将原图转化为一个**有向无环图**。这是因为如果缩点后的图存在环，那么环上的所有 SCC 本应属于同一个更大的 SCC。
   - DCC 缩点后则会形成一种树状结构。具体来说：
     - **点双连通分量** 缩点后形成的是**块割树**。
     - **边双连通分量** 缩点后形成的是**桥树**。
       这些树结构完整地保留了原图的连通性信息（如割点、桥），因此与以最小化边权总和为目的的**最小生成树**有本质区别，它更侧重于揭示图的连通骨架。

前置算法——求割边和割点

关键判断条件

- **割边判断**：`low[v] > dfn[u]` ⇒ 边(u,v)是桥
- **割点判断**：`low[v] >= dfn[u]` ⇒ u是割点（根节点需特殊处理）

算法原理

割边判断原理

```
if (low[v] > dfn[u]) {  // 说明v无法通过其他路径回到u的祖先
    bridges.push_back({u, v});  // (u,v)是桥
}
```

理解：如果v及其子树能追溯到的最早节点都比u晚，说明删除(u,v)后v将无法到达u

割点判断原理

```
// 非根节点
if (father != u && low[v] >= dfn[u]) {
    cut[u] = true;  // u是割点
}
```

理解：如果v无法绕过u到达更早的节点，说明u是关键连接点

根节点特殊处理

```
if (father == u && child >= 2) {
    cut[u] = true;  // 根节点有多个子树
}
```

理解：根节点如果有≥2个子树，删除后图会分裂

复杂度分析

- **时间复杂度**：O(n + m) - 每个节点和边只访问一次
- **空间复杂度**：O(n + m) - 存储图和DFS栈

注意事项

1. **无向图处理**：每条边存两次，数组开2倍
2. **重边处理**：当前代码可处理重边
3. **连通分量**：可能有多连通分量，需要循环调用
4. **根节点**：父节点设为自己，便于判断

Code

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
constexpr int maxn = 1e5+7;
vector<int> g[maxn];
set<int>cut;
vector<pair<int,int> >brg;
int n,m,u,v,dfn[maxn],low[maxn],timestamp;
void Tarjan(int u,int fa){
    dfn[u] = low[u] = ++timestamp;
    int son = 0;
    bool iscut = false;
    for(int v:g[u]){
        if(!dfn[v]){
            son++;
            Tarjan(v,u);
            low[u] = min(low[u],low[v]);
            if(low[v] > dfn[u])brg.push_back({u,v});//避免割边重复计入
            if(fa != u && low[v] >= dfn[u])cut.insert(u);//set自动去重
        }else if(v != fa)low[u] = min(low[u],dfn[v]);//ban situation of (fa - u - v)
    }
    if((fa == u && son >= 2) || iscut)cut.insert(u);//set自动去重
}
signed main(){
    ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    cin>>n>>m;
    while(m--){
        cin>>u>>v;
```

```
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for(int i=1;i<=n;i++)if(!dfn[i])Tarjan(i,i);
    cout<<cut.size()<<'\n';
    for(int i:cut)cout<<i<<' ';
    return 0;
}
```

然后贴上割边的 板子题

e-DCC

发现 **桥** 的两边各连接一个 e-DCC 所以说可以先找出所有的桥，再把所有桥 **锁定**，这样就可以再每一个 e-DCC 内部跑 DFS 来找到每一个 e-DCC内部的节点。

DFS 的代码大致如下：

```
void dfs(int u, int ndcc){
        dcc[u] = ndcc;//记录节点属于的edcc编号
        Ans[ndcc - 1].push_back(u);//把节点加入edcc
        for(auto [v,i]:g[u]){
                if (dcc[to] || b[i]) continue;//b[i] == truel 指 第i条边是桥
                dfs(to, ndcc);
        }
}
```

但是这样显然不够优化，我们尝试换一个 Tarjan 的方式。

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
constexpr int maxn = 5e5 + 7;
vector<pair<int,int> > g[maxn],br;
int n,m,u,v,dfn[maxn],low[maxn],vis[maxn],timer,cnt_col,col[maxn];
vector<int> edcc[maxn];//存储双连通分量
stack<int> stk;
void Tarjan(int u,int lst){
    dfn[u] = low[u] = ++timer;
    stk.push(u);
    for(auto [v,i]:g[u]){//i 是当前边的编号
        if(i == lst)continue;//不走来时路
        if(!dfn[v]){
            Tarjan(v,i);
            low[u] = min(low[u],low[v]);
            if(low[v] > dfn[u])br.push_back({min(u,v),max(v,u)});
        }else low[u] = min(low[u],dfn[v]);
    }
    if(low[u] == dfn[u]){//和scc一样
        ++cnt_col;
        int t;
        do{
```

```
            t = stk.top();
            stk.pop();
            col[t] = cnt_col;
            edcc[cnt_col].push_back(t);
        }while(t != u);
    }
}

signed main(){
    ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    cin>>n>>m;
    while(m--){
        cin>>u>>v;
        g[u].push_back({v,m});
        g[v].push_back({u,m});
    }
    for(int i=1;i<=n;i++){
        if(!dfn[i])Tarjan(i,0);
    }
    cout<<cnt_col;
    for(int i=1;i<=cnt_col;i++){
        cout<<'\n'<<edcc[i].size()<<' ';
        for(int i:edcc[i])cout<<i<<' ';
    }
    return 0;
}
```

对比 *SCC的代码* 发现有三处较大的改动：

1. `if(i == lst)continue;`：**由于 eDCC 处理的是无向图，每条边在邻接表中被存储了两次。**为了避免通过同一条无向边返回父节点，需要跳过该反向边。
2. `if(low[v] > dfn[u])br.push_back({min(u,v),max(v,u)});`：**这是桥（割边）的判定条件。**当满足此条件时，边 `(u,v)` 是一个桥，可将其记录到桥集合中。如果不需要显式记录所有桥，此代码可以省略。
3. `else low[u] = min(low[u],dfn[v]);`：**在无向图的 DFS 树中，不存在横叉边。**所有非树边都必然是连接当前节点与其祖先的后向边。因此，当访问到已访问过的节点时，该节点必定在当前节点的祖先链上，不会误入其他 eDCC，所以**不需要像 SCC 算法那样使用栈来判断节点是否在当前连通分量中。**

v-dcc

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
constexpr int maxn = 5e5+7;
vector<pair <int,int> > g[maxn];
vector<int>vdcc[maxn];
stack<int>s;
int n,m,u,v,dfn[maxn],low[maxn],timestamp,t,cnt_col;
void Tarjan(int u,int fa){
    dfn[u] = low[u] = ++timestamp;
    int son = 0;
    s.push(u);
    for(auto [v,i]:g[u]){
```

```
        if(i == fa)continue;
        son++;
        if(!dfn[v]){
            Tarjan(v,i);
            low[u] = min(low[u],low[v]);
            if(low[v] >= dfn[u]){
                ++cnt_col;
                do{
                    t = s.top();
                    s.pop();
                    vdcc[cnt_col].push_back(t);
                }while(t != v);
                vdcc[cnt_col].push_back(u);//u不出栈，但是要加入到vdcc中，一个点最多可以被两个vdcc包含
            }
        }else low[u] = min(low[u],dfn[v]);
    }
    if(!son && !fa)vdcc[++cnt_col].push_back(u);
}
signed main(){
    ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    cin>>n>>m;
    for(int i=1;i<=m;i++){
        cin>>u>>v;
        if(u == v)continue;
        g[u].push_back({v,i});
        g[v].push_back({u,i});
    }
    for(int i=1;i<=n;i++)if(!dfn[i])Tarjan(i,0);
    cout<<cnt_col;
    for(int i=1;i<=cnt_col;i++){
        cout<<'\n'<<vdcc[i].size()<<' ';
        sort(vdcc[i].begin(),vdcc[i].end());
        for(int node:vdcc[i])cout<<node<<' ';
    }
    return 0;
}
```

需要注意的地方有：

1. 若检测到 `low[v] >= dfn[u]` 则立即分割这个连通分量，而一个点可以包括在多个 v-dcc 内部，所以在最后要记得 `vdcc[cnt_col].push_back(u);`
2. 记得判断自环，因为程序 `if(i == fa)continue;` 这会导致一些孤立点没有被计算。
3. 记得判断孤立点——它自身就是一个 vdcc `if(!son && !fa)vdcc[++cnt_col].push_back(u);`