Tarjan算法 - 20230522

version 1.0 20230525 EDIT

一.Tarjan的区分

> Robert E. Tarjan（罗伯特·塔扬，1948~），生于美国加州波莫纳，计算机科学家。
>
> Tarjan 发明了很多算法和数据结构。不少他发明的算法都以他的名字命名，以至于有时会让人混淆几种不同的算法。

就咱个人而言，咱之前从来没有搞清楚过边双连通分量，点双连通分量，割点，割边，缩点之间的关系(甚至在边双的代码里出现SCC的变量名)，现在算是弄懂了吧(也许)

Tarjan算法可以分为两大类，求强连通分量(SCC)的Tarjan和求双连通分量(BCC)的Tarjan。前者可以用于处理缩点，后者可以用于处理割点，割边，点双和边双(个人理解，可以把割点看做基础算法，割边和割点及其相似，点双和边双则是割点割边的运用)。SCC_Tarjan是在有向图上的，通常缩点为一个DAG进行后续操作(或者求环)；而BCC_Tarjan则是在无向图上的，可以处理双连通分量相关问题。

二.求双连通分量(BCC)的Tarjan

1.割点及点双连通分量

(1)求割点的tarjan算法原理

首先定义：使用dfs访问的顺序称为 dfn (dfs序)，由于如果只取dfs过程中经过的点，那么原先的无向图会变成一棵树或者一个森林，这颗树上的边叫做树边，其他边则叫做非树边。最后我们定义一个节点的 low 值为其不经过父亲节点(或者理解为:不经过已出现的树边/不穿过(可以到达)所有祖先节点)能到的节点的 dfn 最小值，一个点的初始 dfn 为自己(显然)

那么我们不难得出，若节点 u 有一个儿子 v 使得 $low_v \geq dfn_u$，这说明子节点不经过父亲到达不了其他已访问过的节点，把节点 u 去除后 v 与上方的节点分隔开，故 u 是割点。

于是我们接下来就是考虑如何在dfs的过程中求出 low 数组。当我们从 u 访问到一个未被访问过的节点 v 时，对它使用tarjan进行递归(等于进行dfs)，此时 v 的 low 已经被求出，我们从 u 到 v 这个操作本身显然不会违反 low 的定义,那么 low_u 值就可以用 low_v 更新；当我们从 u 访问到**非父亲的**已被访问过的节点 v 时，因为"可以到达而不可以穿过"，于是用 dfn_v 来更新 low_u (注意不是 low_v)

最后是特殊判定，我们在判断割点时是判断子节点能否到达已访问过的节点，然而对于根节点来说上方根本没有已访问过的节点，于是特判，如果为搜索树根节点且有两个及以上儿子，那么他是割点(显然)。

(2)求割点的tarjan算法代码

```
vector<int> otp;//割点编号序列
//stack<int> st; //求BCC用的栈
void tarjan(int u,int fa){
    //st.push(u);  //求BCC入栈
    bool INS = 0;//防止在根节点处重复判定割点
    dfn[u] = ++dnt;
    low[u] = dfn[u];
    int son = 0;//计算儿子数量,特判父节点
    for(int i = head[u];i;i = e[i].nxt){
```

```
        const int v = e[i].to;
        if(!dfn[v]){//如果是未被访问过的节点
            ++son;
            tarjan(v,u);//递归求出v的low数组用于更新
            low[u] = min(low[u],low[v]);
            if(INS == 0 && fa != 0 && low[v] >= dfn[u]){
                otp.push_back(u);
                INS = 1;
                //AddBCC(u,v);    添加BCC
            }
        }else if(v != fa)low[u] = min(low[u],dfn[v]);
    }
    if(INS == 0 && fa == 0 && son >= 2){//特判根节点
        otp.push_back(u);
        //BCC[++bccs].push_back(u);    特判添加BCC，此节点自己是一个BCC
    }
}
```

(3)求BCC的修改原理及代码

只需要把上面四行注释掉的求BCC代码加上，使用如下的`AddBCC`函数即可求BCC。

原理很简单，如果一个点是割点，那么它子树中未属于一个BCC的点一定属于同一个BCC。

```
inline void AddBCC(int u,int v){
    ++bccs;
    while(!st.empty()){
        const int tp = st.top();
        st.pop();
        BCC[bccs].push_back(tp);
        if(tp == v)break;
    }
    BCC[bccs].push_back(u);
    return ;
}
```

2.桥及边双连通分量

求桥和求割点的代码基本相同，唯一的区别是`low[v] >= dfn[u]`改为`low[v] > dfn[u]`，可以理解一下如果 $low_v = dfn_u$，则说明 v 点不直接通过父亲节点刚好只能到达其父亲节点 u，此时 u 显然还是割点，因为去除了 u 那么 v 就无法连通到上面的部分；然而 $u \to v$ 的边即使割掉，通过其他路径也可以到达 u，那么它就不是桥。

除此以外求桥和边双连通分量没有区别

三.求强连通分量(SCC)的Tarjan

1.求SCC的Tarjan算法

求SCC的Tarjan和求BCC的Tarjan其实也大同小异，我们根据代码进行分析

```
stack<int> stk;
bool ins[MAXN];//记录是否在栈中
void tarjan(int u){
```

```
    dfn[u] = low[u] = ++dnt;
    stk.push(u);//  - \ 入
    ins[u] = 1;//   - / 栈
    for(int i = h1[u];i;i = e1[i].nxt){
        const int v = e1[i].to;
        if(dfn[v] == 0){//如果没有访问过，则搜索后用v的low更新
            tarjan(v);
            low[u] = min(low[u],low[v]);
        }else if(ins[v]){//否则如果在栈内则进行更新
            low[u] = min(low[u],dfn[v]);
        }//else：该节点已被访问但是不在栈中，唯一出栈的机会是已经在一个强连通分量里了
    }
    if(low[u] == dfn[u]){//一个节点的子树最早只能到这个节点，说明他是这个强连通分量的"根"
        while(!stk.empty()){
            const int tp = stk.top();
            stk.pop();
            ins[tp] = 0;
            bel[tp] = u;
            if(tp == u)break;
            p[u] += p[tp];
        }
        np.push_back(u);
    }
    return ;
}
```

我们可以考虑到，一个强连通分量只有它的"根"符合 $low = dfn$，并且这个点是第一个被访问到的点。原因很简单，在强连通分量里每一个点都可以相互到达(定义)，那么整个强连通分量的 low 值也一定都是整个强连通分量中最小的 dfn 值，也就是第一个被搜索到的节点的 dfn 值。

一个不那么好理解的地方是，为什么已经在一个强连通分量里的节点 v 不会更新 low_u？咱的一个解释是考虑一个两个完全独立的双连通分量，中间只有一个单向边连接，只在这种情况下才会在出现该节点已被访问且不在栈中。换而言之，如果你访问到了已经出栈的节点，那么这个节点必然不可能能够到达你，因为不然你也会在这个强连通分量里了。

另外记录一下自己的一个思考误区，强连通分量不一定是一个环，考虑两个环“扣起来”，扩展到多个环环环相扣。

2.SCC-Tarjan的常见用法：缩点

首先我们知道，在一个有向无环图(DAG)上我们可以进行很多操作(DP,拓扑等等)。一些题目里，我们获得的图并不是DAG，但是它们符合一个定律：重复经过的点不重复计算，这个时候我们就要想到Tarjan缩点了(不满足这个定律也可以缩点吗？有待思考)

因为在一个强连通分量中任意两点都可以互通，那么显然可以到了其中一个点就一定可以经过同一个SCC里的所有点，考虑到贡献不重复，那么就可以把一个SCC看做一个整体，也就是所谓的"缩点"。所有连向这个SCC中任意一个点都等效连向代表这个SCC整体的一个新点，同理从这个SCC中任意一个点连出去也等效于SCC的新点连出去。

3.2-SAT

> 鱼越大，鱼刺越大；鱼刺越大，鱼肉越少；鱼肉越少，鱼越小。所以鱼越大，鱼越小。——By 6657玩机器

2-SAT是一个用于处理布尔方程组求解的算法。根据个人的理解，咱认为它可以看做求解“依赖”关系的算法。“依赖”的意思就是有 A 就有 B，它等同于有 A 就有 B 并且有 $\neg B$ 有 $\neg A$，符号化为 $A \to B \ \wedge \neg B \to \neg A$。

我们注意到，"有 A 就没有 B" 和"有 A 就有 $\neg B$"是等价的。为了统一为“依赖关系”，我们只使用"有...就有..."而不使用"有...就没有...",显然后者可以翻译为"有...就有不..."。

2-SAT的核心在于建模，即使用题目中的背景翻译出对应的布尔方程组，在此不过多赘述。在建立起布尔方程组后，我们可以把这个方程组建为一个图。具体来说，一个“依赖关系”建立一条有向边，即 "$A \to B$"中的箭头具体化为了一条边。假设有 n 个布尔变量，那么就有 $2n$ 个点(因为有 A 就有 $\neg A$)。什么时候方程无解？当直接或间接地，出现 $A \to \neg A$ 或者 $A \to \cdots \to \neg A$时，就意味着出现了"鱼越大鱼越小"的情况，自相矛盾无解。对于这张有向图，我们就可以使用SCC-Tarjan求解强连通分量了。

当布尔方程组有解时，我们查询每一个变量 A 与其反变量 $\neg A$所在SCC的编号，哪个编号越小取哪个为这组解的值。这是一定正确但不一定唯一的，显然方程可能有多组解。

四.总结

1.Tarjan的核心思想

咱认为Tarjan的核心在于 low 数组，当 low_v 小于 dfn_u 时代表子树不经过该节点(或者边)也可以到达先前的节点，意味着 u 是"不必要的"，那么它就不是割点/桥，反之则是。

2.易错点

1. 如果使用Vector等序列存储割点割边，或者添加双连通分量的时候，记得判重，防止重复添加节点/连通分量。也可以通过使用bool数组打标记来解决这个问题。
2. 求BCC要特判搜索树根节点的情况