联通分量

本文探讨和联通分量有关的算法。

强连通分量分解

强连通分量分解常常用于缩点和求解 2-SAT 问题。

如果一个有向图中任意两个点相互可达，那么这个有向图**强连通**。而**强连通分量**（Strongly Connected Component，SCC）的定义是极大的强连通子图。所谓极大，也就是再包含任何其他一个点都会导致这个图不再强连通。

Tarjan 算法

求一个图的强连通分量比较常用的一个方法是 Tarjan 算法。

Tarjan 算法基于深度优先搜索，每一个搜索到的强连通分量都是搜索树的一棵子树。同时也用到了栈。

Tarjan 算法用到了两个十分重要的数组：dfn 和 low。两个数组均和节点相关。前者保存搜索时每个节点的时间戳（即被搜索到的次序），每个节点的时间戳各不相同，并且一旦确定就不再改变。后者保存每个节点本身及其子树，这些节点所能**回溯**到的在栈中的节点的 dfn 的最小值。当最终得到 $dfn[v]$ 和 $low[v]$ 相等时，就说明 v 及其子树构成一个强连通分量。

执行这个算法时分以下几个步骤：

0. 由于图不一定联通，对每一个点都应当遍历。如果某个点的时间戳尚未生成就进入搜索。
1. 设当前节点为 v。把点加入栈中，更新 $dfn[v]$，并令 $low[v] = dfn[v]$。
2. 检查每一个 v 可以到达的点，设为 u。
    - 若 u 未被搜索过，就令其进入搜索，并更新 $low[v] = \min\{low[v], low[u]\}$。
    - 若 u 已被搜索过且在栈中，就更新 $low[v] = \min\{low[v], dfn[u]\}$。
3. 如果 $dfn[v] = low[v]$，就将 v 及其子树所含节点从栈中弹出，弹出的这些节点构成一个强连通分量。对 v 的搜索结束。

容易看出，该算法的时间复杂度为 $O(|V| + |E|)$。

```
int dfn[100005], low[100005], D, in[100005];
int stack[100005], top;
void tarjan_scc(int id){
    dfn[id] = low[id] = ++D;
    in[id] = true;
    stack[top++] = id;
    int i = at[id], vv;
    while(i){
        vv = e[i].v;
        if(!dfn[vv]) tarjan_scc(vv), low[id] = min(low[id], low[vv]);
        else if(in[vv]) low[id] = min(low[id], dfn[vv]);
        i = e[i].nxt;
    }
    if(dfn[id] == low[id]){
        //do something
        do{
            top--;
```

```
                in[stack[top]] = false;
            }while(stack[top] != id);
        }
}
```

按照上面的算法过程得到的强连通分量的顺序是逆拓扑序。

Kosaraju 算法

另一个常用的强连通分解算法为 Kosaraju 算法。

该算法基于以下两个事实：

1. 一个 DAG 的拓扑序即为该 DAG 在 DFS 之后的顶点的逆后序排列。
2. 有向图的两个顶点可以互相访问，那么这两个顶点在同一个强连通分量之中。

该算法的运行也很简单，只分三步：

1. 对当前图 G 进行DFS，得到所有点的后序遍历。
2. 对当前图 G 构造反图 G^-。
3. 按照后序遍历编号从大到小对顶点进行检查，如果顶点未被 DFS 过就以该点为起点在 G^- 上进入搜索，每次搜索经过的所有点就构成一个强连通分量。

该算法的时间复杂度也为 $O(|V|+|E|)$。

```
/* 在e_和at_保存了反图 */
int at_[10005] = {0}, cnt_ = 0, rk_cmp[10005], R = 0;
bool in[10005] = {0};
int st[10005], top = 0;
void dfs(int id){
    in[id] = true;
    for(int i = at[id]; i; i = e[i].nxt)
        if(!in[e[i].v]) dfs(e[i].v);
    st[top++] = id;
}
void rdfs(int id){
    in[id] = false;
    rk_cmp[id] = R;
    for(int i = at_[id]; i; i = e_[i].nxt)
        if(in[e_[i].v]) rdfs(e_[i].v);
}
void kosaraju(){
    for(int i = 1; i <= V; ++i)
        if(!in[i]) dfs(i);
    for(int i = top - 1; i >= 0; --i)
        if(in[st[i]]) R++, rdfs(st[i]);
}
```

该算法相较 Tarjan 算法更麻烦，但能方便的得到强连通分量之间的拓扑序，因此可以较为方便地用于求解 2-SAT 问题。

应用：缩点

如果将所有的强连通分量都用一个点来重新表示的话，那么形成的新图就是一个 DAG。这样就可以套用一些 DAG 的方法来解题。

应用：2-SAT 问题

给定一个布尔方程，判断是否存在一组布尔变量的真值指派使整个方程为真的问题，被称为布尔方程的可满足性问题（SAT）。可以把布尔方程写成析取式的合取，即合取范式的形式。如果该布尔方程中，每个析取式至多只包含两个变量，就称该问题为 2-SAT 问题。

利用强连通分量分解可以在线性时间复杂度内解决 2-SAT 问题。把 $a \vee b$ 改写成 $(\neg a \to b) \wedge (\neg b \to a)$ 的形式，并且把每个变量拆成 a 和 $\neg a$ 两个点，根据蕴含关系建立边。随后，对整个图进行强连通分量分解，易知在同一个强连通分量中的变量真值必须相同。

分解完成后，如果一个变量 a 和 $\neg a$ 在同一个强连通分量中，那么显然这时无解。否则，可以根据强连通分量的拓扑序来给变量赋值，如果 a 所在分量的拓扑序在 $\neg a$ 之后就令 a 为真，否则为假。这是为了保证 $a \vee a$ 这样的条件可以得到满足。如此就完成了 2-SAT 问题的求解。

2-SAT 问题可以推广，只要能将命题的形式写成合取范式，然后变形即可。

双连通分量

双连通分量和桥、割点有关。

双连通分量可以看作是强连通分量在无向图上的一个版本，但是要稍微复杂一些。

在一张无向图中，对于两个点 u, v，如果删去任何一条边都不能使得两者无法相互可达，那么称这两个点**边双连通**；如果删去任何一个点（不能是 u 或者 v）都不能使得两者无法相互可达，那么称这两个点**点双连通**。

显然可以把这种双连通性用二元关系来表示。其中，边双连通具有传递性，而点双连通不具有传递性。

自然要问：如果两个点不边/点双连通，那么究竟是哪条边/哪个点导致了这一结果？

割点的寻找

对于一个联通的无向图而言，如果删除了一个点会导致该图不再联通，那么这个点就称为**割点**。

割点的寻找也依赖于 DFS 树。和强连通的情况类似，定义 dfn 和 low，分别表示一个点在 DFS 树中的时间戳和其能追溯到的、**除了父节点之外**节点的时间戳的最小值。那么对于一个点 u 而言，如果它在 DFS 树中的儿子 v 满足 $low[v] \geq dfn[u]$，那么 u 就**很有可能**是一个割点。

为什么要说是很有可能？对于 u 不为根的情况，如果有这样的 v，那么 u 就确实是一个割点，因为这表明 v 只有通过 u 才能和祖先或者此前其他子树相联系。但如果 u 是 DFS 树的树根的话，显然别的点所能追溯到的最早的点就是 u，这个判断原则就失去意义了。因此，对于 u 为树根的情况要单独处理：如果此时 u 在 DFS 树中有两个儿子，那么 u 是割点，因为这表明删掉 u 就会产生至少 2 个联通子图。

这样，按照类似于强连通分量的写法，就不难写出求割点的代码。算法的时间复杂度是 $O(|V| + |E|)$。

注意，在代码实现上可以有一个小技巧，即去掉 low 不能考虑父亲的限制，然后把割点判定条件从 $low[v] \geq dfn[u]$ 改成 $low[v] = dfn[u]$。这样可以少一个父节点的递归参数。

```
int rt;
void tarjan(int cur){
    dfn[cur] = low[cur] = ++D;
    int flag = 0;
    for (int i = at[cur]; i; i = nxt[i]){
        int v = to[i];
        if (!dfn[v]){
            tarjan(v);
            low[cur] = min(low[cur], low[v]);
            if (low[v] == dfn[cur]){
                ++flag;
```

```
                if (cur != rt || flag > 1) cutvex[cur] = true;
            }
        }else low[cur] = min(low[cur], dfn[v]);
    }
}
void solve(){
    for (int i = 1; i <= n; ++i)
        if (!dfn[i]) rt = i, tarjan(i);
}
```

由此算法的正确性也可以推知：如果一个节点在 DFS 树上是叶子节点，那么它必然不会是割点。

桥的寻找

对于一个联通的无向图而言，如果删除了一条边会导致该图不再联通，那么这条边就称为**桥**。

桥的情况和割点是类似的，我们研究儿子 v 到父亲 u 的边是不是桥。如果 v 无法绕过这条边到达另一子树或者祖先节点，那么这条边就是桥了。

于是，我们修改一下割点时的判定标准：如果 $low[v] > dfn[u]$，那么这条边是一个桥，因为这表明 v 只有通过 u 才能和祖先或者此前其他子树相联系。同时由于考虑的是边，就不需要特判 u 是根节点的情况了。

代码实现上和之前也是类似的。算法的时间复杂度是 $O(|V| + |E|)$。

```
void tarjan(int cur, int fa){
    dfn[cur] = low[cur] = ++D;
    for (int i = at[cur]; i; i = nxt[i]){
        int v = to[i];
        if (v == fa) continue;
        if (!dfn[v]){
            tarjan(v, cur);
            if (low[v] > dfn[cur]){
                id[++tot] = ((i & 1) ? (i + 1): i);
            }else {
                low[cur] = min(low[cur], low[v]);
                id[++tot] = ((i & 1) ? i: (i - 1));
            }
        }else low[cur] = min(low[cur], dfn[v]);
    }
}
void solve(){
    for (int i = 1; i <= n; ++i)
        if (!dfn[i]) tarjan(i);
}
```

应用：双连通分解

（等待补充）