《最小连通块》命题报告

浙江省杭州第二中学 潘骏跃

摘 要

本文介绍了作者在一次校内联测中命制的一道交互题。该题解法多样，需要选手对树的结构有较为深刻的认识并能灵活地运用树的性质，是一道考察选手对树这种数据结构的了解程度的好题。

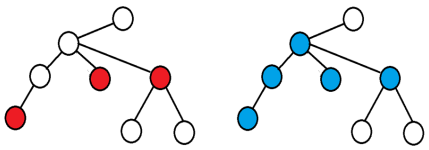
1 题目大意

1.1 题目描述

这是一道交互题。

对于一棵树 T，我们定义这棵树上的某个点集的最小连通块为包含这个点集中所有点的最小的树上连通块。

例如在下图中，蓝点所构成的点集就是红点所构成点集的最小连通块。



已知某一棵树的大小 n，你可以进行若干次询问，每次询问你可以给出一个点集 S 和这棵树上的一个点 x，交互库会返回一个布尔值表示 x 是否在点集 S 的最小连通块上。

你需要确定这棵树的形态。

本题保证所使用的树在交互开始之前已经完全确定，不会根据和你的程序的交互过程动态构造。

1.2 实现细节

你不需要，也不应该实现主函数，你只需要实现函数 $work(n)$，其中 n 表示所求树的点数。你可以调用如下四个函数来与交互库进行交互：

- *clear*()，表示清空当前的点集 S。
- *add*(x)，表示从点集 S 中加入 x 号点。
- *query*(x)，表示询问点 x 是否在点集 S 的最小连通块上。
- *report*(x, y)，表示确定所求树中存在一条边 (x, y)。

评测时，交互库会恰好调用 *work*(n) 一次。
我们只会对 *query* 操作的次数进行限制。

1.3 评分方式

本题共有一个测试包，内含若干个测试点。

对于每个测试点，若你的程序有不合法的询问或返回，或返回的树的形态不正确，则你在该测试点获得 0 分。否则令 *step* 表示你的程序的询问次数，则你在该测试点获得的分数将评定为 $min(\lfloor \frac{2.2\times10^6}{step} \rfloor, 100)$。

你所获得的这道题的分数即为所有数据点的分数的最小值。（可以发现，若要获得满分，则询问次数不能超过 22000 次）

1.4 限制与约定

对于所有数据，均满足 $n = 1000$。
时间限制：5s
空间限制: 1024MB

2 解法分析

2.1 算法一

在下文中，我们称一次“询问 x 是否在点集 S 的最小连通块上”的操作为 *query*(S, x)。

首先我们将给出的询问方式转化，询问 x 是否在点集 S 的最小连通块上等价于询问 S 中是否存在两个点 u, v，满足 x 在 u 到 v 的链上。

那么如果我们询问的点集 S 大小为 2，我们就可以直接询问出询问的点 x 是否在 S 中两个点所对应的那条链上。

我们可以直接得到一个多项式复杂度的做法，即对于任意三个点 u, v, w，通过 *query*($\{u, v\}, w$) 得到它们之间的关系，并由此确定整棵树的形态。

也就是说，我们得到了：

算法一：对于任意一对点 (u,v)，我们对剩下的任意一个点 w 进行一次 $query(\{u,v\},w)$，若这 $n-2$ 次询问的答案均为 false，则 u 与 v 直接相连。借此可以得出所有直接相连的点对，确定出整棵树的形态。

算法复杂度： $O(n^3)$

2.2 算法二

可以注意到我们并不需要对任意三个点 u,v,w 得到 $query(\{u,v\}.w)$ 的结果。实际上，将这棵树视为一棵有根树之后，如果我们能得出任意一对点之间的“祖先-后代”关系，那么也能确定出整棵树的形态。而这种做法也是解决这类问题的一种常用做法。

在下文中，我们都将该树视作一棵有根树，同时将该树的 1 号点视为根。

而要得出一对点 u,v 之间是否存在“祖先-后代”关系，只需要 $query(\{1,u\},v)$ 即可。

也就是说，我们得到了：

算法二：对于任意一对点 (u,v)，我们通过 $query(\{1,u\},v)$ 判断出 v 是否为 u 的祖先。我们可以借此得到任意一个点的祖先集合，设 i 号点的祖先集合为 Anc_i（特殊地，我们认为 i 号点本身也是 i 号点的祖先）。那么对于一个点 x 来说，它的父亲 f 就是满足 $Anc_f \subset Anc_x$ 的 $|Anc_f|$ 最大的点。确定每个点的父亲，我们确定了这棵树的形态。

算法复杂度： $O(n^2)$

2.3 算法三

注意到我们仍然只使用了点集大小为 2 时的询问，接下来我们将挖掘所给询问方式更多的用法。

既然可以得到两个点之间的“祖先-后代”关系，那么我们也可以用类似的方法只通过一次询问得出多个点之间是否存在“祖先-后代”关系。

对于一个点集 S 与一个点 u 我们进行一次 $query(S\cup\{1\},u)$，若结果为 true 则说明 S 中存在 u 的后代，否则不存在。这是因为如果 S 中同时存在两个点使得这两个点一个在 u 的子树内一个在 u 的子树外，那么它们所构成的那条链就会经过 u，而此时我们已经提供了一个子树外的点 1 号点；同样地，要想存在两个点使得它们对应的链经过 u，则这两个点必然有至少一个在 u 的子树内。故“ S 中存在 u 的后代”与“该询问返回值为 true”是充要的。

进一步挖掘，若确认 S 中存在 u 的后代后，我们可以利用“二分”的方式直接找到 S 中 u 的某一个后代，具体操作方式如下：

- 将点集 S 分为两个点集 S_L,S_R，使得 $S_L\cup S_R=S$ 且 $S_L\cap S_R=\emptyset$ 且 $-1\le|S_L|-|S_R|\le 1$。
- 进行一次询问 $query(S_L\cup\{1\},u)$，若返回值为 true 则将 S 修改为 S_L，否则将 S 修改为 S_R。

- 重复上述两个步骤直至 $|S| = 1$，此时 S 内的点即为所求的后代。

由于每次询问后 S 的大小至多变为 $\lceil \frac{|S|}{2} \rceil$，所以一次“找到某一个后代”的操作的复杂度为 $O(\log |S|)$。

更进一步挖掘，我们可以通过如下步骤找出 S 中 u 所有的后代：

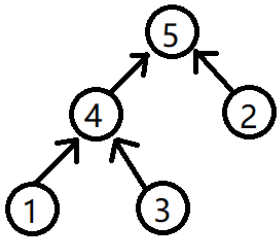
- 判断 S 中是否有 u 的后代，若无则直接退出。
- 找到 S 中 u 的一个后代。
- 从 S 中将找出的后代去掉，回到第一个步骤。

设 S 中 u 的后代个数为 m，则这样做的复杂度为 $O(m \log |S|)$。

得到上述工具后，我们发现可以将问题进行转化。

若我们将有根树的每一条边视为从儿子指向父亲的有向边，那么我们可以将有根树视作一张有向无环图，定义出一棵有根树的拓扑序。

例如下图中，对于这棵有根树的某种拓扑序，每个点上的数字展现了它在该拓扑序上的位置：



我们发现，如果我们能得出所求树的拓扑序，我们就能直接确定这棵树。具体的操作步骤如下：

- 设 $a_{1...n}$ 为该树拓扑序上第 i 个点的编号，集合 V 的初始值为该树的点集。
- 令 i 从 1 扫到 n，每一步在 V 中找到 a_i 所有的后代，然后将这些后代从 V 中去掉。

由于一个点非儿子的后代已经在这些后代对应的父亲处从 V 中被去掉，所以一个点找到的那些后代就是它在原树中所有的儿子。

可以证明这部分的算法复杂度为 $O(n \log n)$，因为每个点被作为后代找到只有一次，而每次找到一个后代的复杂度都是 $O(\log n)$。

从这里开始，我们把问题转化为了求出原树的一种拓扑序。接下来的所有做法都是以“找出拓扑序”为目的。

考虑拓扑排序的常用算法，我们每次找到一个入度为 0 的点，然后把它加入当前拓扑序的末尾并在原图中把它删去。转化到树上，就是每一次找到原树的一个叶子，再把它从树中去掉。

我们称呼这种做法为“剥叶子”。

而判断一个点 u 是否为叶子很简单，设该树的点集为 V，则 $query(V-\{u\},u)$ 即可。

也就是说，我们得到了：

算法三：每一轮我们扫一遍原树的每一个点并判断它是否为叶子，若是则将其加入拓扑序的末尾，然后在该轮结束时从原树的点集中把所有找到的叶子删掉。判断一个点是否为叶子的方法和找到拓扑序之后的做法在此以及之后不再赘述。

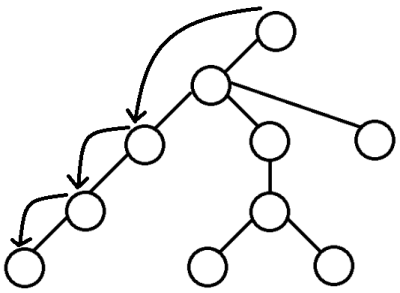
算法复杂度： $O(n^2)$

2.4 算法四

事实上我们每一次试图找到一个叶子的算法复杂度太高了，可以尝试在这里优化。

我们可以再一次利用之前的工具，每次不断从该树的点集中随机找到当前点的一个后代然后“跳”过去，直到找到一个叶子为止。

下图就是该算法“跳”的过程的一个例子：



令 siz_x 为 x 号点的子树大小，则对于每一个点 u 来说，它的后代里子树大小超过 $\lceil \frac{siz_u}{2} \rceil$ 的最多占到一半，所以最多期望 2 次就可以使当前所在点的子树大小变为原来的一半。由此可以证明，按照这种做法往下“跳”，期望“跳”的次数为 $O(\log n)$。而每次找到一个后代的复杂度也为 $O(\log n)$，故找到一个叶子的复杂度为 $O(\log^2 n)$。

也就是说，我们得到了：

算法四：这是一个随机算法。我们通过剥 n 次叶子来确定拓扑序。每次初始时我们令 $u=1$，再找到 u 号点的随机一个后代 v 号点，并将 v 赋值给 u，直到 u 成为一个叶子。

算法复杂度： $O(n\log^2 n)$

2.5 算法五

算法三还有另一种优化角度，那就是优化“剥叶子”的轮数。

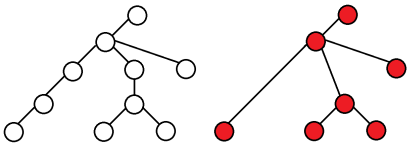
可以注意到若每次每一轮把原树的所有叶子剥掉，最劣情况下（也就是当原树为一条链时）需要剥 n 轮。

我们发现树有一个性质，那就是如果这棵树中没有二度点，则这棵树的叶子个数至少为总点数的一半。由此可以自然地想到如下的定义。

定义一棵树的虚树为对该树执行如下操作之后所形成的树：

- 找到当前树除根外的一个二度点 w，设它连接的两个点分别为 u, v。若找不到则退出。
- 将 w 以及以 w 为端点的边从树中删去，并加入一条边 (u, v)。
- 回到第一个步骤。

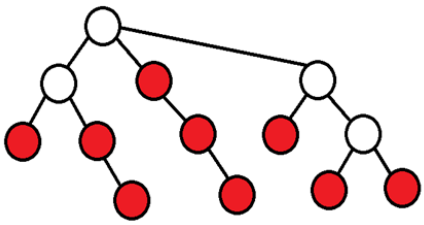
例如在下图中，右边的树即为左边的树的虚树。



可以注意到，如果我们能让每一轮所求树虚树的叶子都被剥掉，那么我们就可以用 $O(\log n)$ 轮剥掉所有的点。

而如果要使某一轮虚树的叶子都被剥掉，那么我们将要在原树中剥掉的点就是，原树中的叶子以及叶子之上那条由二度点构成的链，换句话说，后代中只有一个是叶子的那些点（特殊地，认为一个点也是它自己的后代）。

如下图，红点就是我们要在树中剥掉的点：



同时，找到这些点也是容易的，只需要先找出叶子集合 L，然后再对于每个点，判断它是否有大于等于 2 个后代在 L 中。判断方式是，先找出该点在 L 中的一个后代 v，然后看 $L - \{v\}$ 中是否存在该点的后代。

找出这些点之后按照它们对应的那个叶子后代分组，每组之内还要进行一次“祖先-后代”的排序。

这样我们就完成了一轮的工作，复杂度为 $O(n\log n)$。而我们一共要进行 $O(\log n)$ 轮，故总复杂度为 $O(n\log^2 n)$。需要特别注意的是，我们每一轮并没有让原树的点数变为原来的一半，所以不能认为复杂度是 $O(n\log n)$。

也就是说，我们得到了：

算法五：每一轮找出当前树中所有的叶子，再对于树上的其它点找出这些点的一个叶子后代并判断它们是否有大于等于 2 个叶子后代。将只有一个叶子后代的那些点按照叶子后代分组，每组内部进行排序，其中 u,v 之间通过 $query(\{1,u\},v)$ 来进行比较。最后将每一轮找到的那些点全部剥掉。

算法复杂度：$O(n\log^2 n)$

2.6 算法六

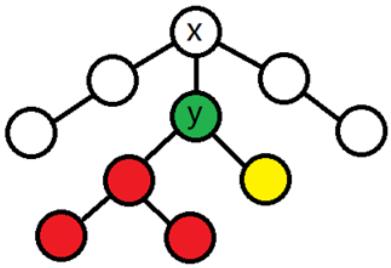
不再从“剥叶子”的角度入手，我们考虑这类问题的常用做法：分治。

我们令函数 $solve(x,S)$ 表示确定了 x 的子树的点集为 S，现在要将 x 子树内的点按照拓扑序排好。

随机找到 x 在 $S-\{x\}$ 中的一个后代 y，我们试图确定 y 的子树内的点所构成的点集为 S'，使得 x 的子树能被分为两个部分，以便进行分治。

实际上这与传统的点分治不太类似，但是我们依然能通过复杂度不错的确定性算法解决这个问题。

为了以较优的复杂度实现这个思路，我们需要先假设原树中这些点根据删掉点 y 之后所在的连通块被染成不同的颜色，如下图：



可以发现，我们的目的就是判断每一个点是否为白色。我们先把除了 y 以外的这些点按照结点标号排成一排，形如下图的样子：

称连续的极长的一串颜色相同的点为“一段”，现在我们尝试求出这排点段与段之间的分割线的位置。

我们知道当且仅当一个点集 A 中存在两个点颜色不同，$query(A,y)$ 会返回 true。那么我们就可以用一次询问判断连续的一串点是否全部颜色相同。当我们确定某一段的左端点时，就可以用这种操作二分出这一段的右端点，从左往右扫一遍就可以确定段与段之间的划分了。

最后可以使用 $query(\{1,u\},y)$ 判断点 u 是否为白色，借此判断出每一段是不是白色，就可以分治下去了。

考虑证明这样做的复杂度是对的。

设有 m 个点被染上了点数最多的那种颜色，试图确定段与段之间划分的复杂度是 $O((|S|-m)\log|S|)$ 的。这是因为每确定一段都需要付出 $O(\log|S|)$ 的复杂度，而段数是 $O(|S|-m)$ 的（出现次数最多的那种颜色最多被划分成 $|S|-m+1$ 段，而剩下的点也只有 $|S|-m$ 个）。

也就是说，我们可以视作每一次分治时，所染颜色不是点数最多的那种颜色的点会贡献 $O(\log|S|)$ 的复杂度。那么考虑每个点对复杂度的贡献：

- 若该点是一个白色点，由于白色不是出现次数最多的，说明被分开的另一个连通块比白色连通块大，白色连通块的大小小于等于原本连通块大小的一半，故只会发生不超过 $O(\log n)$ 次。
- 若该点是一个有色点，则若白色连通块是最大的就与第一种情况相同，否则该点所在连通块的根是 y 的一个轻儿子。又因为每个点被作为 y 选中至多只有一次，故这种情况也只会发生不超过 $O(\log n)$ 次。

由此，我们证明了总复杂度是 $O(n\log^2 n)$ 的。

也就是说，我们得到了：

算法六：处理 x 的子树时，设子树内的点分别为 $a_{1\ldots cnt}$（此处 $cnt=|S|$）。随机找到 x 的一个后代 y，假定树被因此染色。初始时令 $l=1$，每次二分找到最大的 r 满足 $l\le r\le cnt$ 且 $a_{l\ldots r}$ 同色，再令 $l=r+1$，直至 $l>cnt$。确定段与段之间的划分之后对于每个之前出现过的 l 询问 $query(\{1,a_l\},y)$ 判断该段颜色。最后根据颜色分治。

算法复杂度： $O(n\log^2 n)$

2.7 算法七

事实上采用类似分治的思想可以获得更优的复杂度。

令函数 $solve(x,S)$ 表示要求出 S 中有哪些点在 x 子树内并将它们按照拓扑序排好。也就是说，在 $solve(x,S)$ 之前我们并不确定 x 的子树内有哪些点。

算法流程如下：

- 找到 $S-\{x\}$ 中 x 的一个后代 y。若不存在这样的后代则退出。
- 执行函数 $solve(y,S)$，然后将 S 中 y 子树内的点删去，并将 y 子树内的点按照拓扑序加入当前拓扑序末尾。
- 回到第一个步骤。

初始时只要 $solve(1,\{1...n\})$ 即可。由于在该算法中每个点被作为 y 找到只有一次，所以总复杂度为 $O(n\log n)$。

这个算法的核心思想是，抛弃掉那些实际上无用的信息，将 x 的子树一块块剥掉，直到只剩 x。虽然流程十分简单，但是想到它的思维难度却并不低。

也就是说，我们得到了：

算法七：处理 x 的子树时，不断找到 x 的一个后代 y 并对 y 分治下去，最后把 x 子树内 y 的部分剥掉。

算法复杂度：$O(n\log n)$

2.8 算法八

上述的任意一种做法其实都是从树的结构出发，但实际上我们也可以直接在拓扑序上考虑。

我们知道对于这样一棵有根树的拓扑序有一个充要条件，那就是每个点的祖先都必须在这个点之后出现。如果我们能一个一个把点加入拓扑序并时刻维护这个性质，就可以解决整个问题。

我们已经有了判断点集 S 内是否存在 x 的后代的方法。那么只要二分出当前拓扑序的一个位置，使得这个位置之后不存在 x 的后代且这个位置尽量靠前，我们就可以保证加入这个点之后它所有的祖先仍然在它后面，它所有的后代仍然在它前面。

由于每加入一个点时我们只需要二分，所以总复杂度为 $O(n\log n)$。

也就是说，我们得到了：

算法八：枚举 i 从 1 到 n，设将 i 号点加入拓扑序之前，拓扑序为 $a_{1...i-1}$。二分一个位置 p 使得 $query(\{1\}\cup\{a_{p...i-1}\},i)$ 的返回值为 false 且 p 尽量小，将 i 插到第 p 个位置之前，也就是将拓扑序改为 $a_1,...,a_{p-1},i,a_p,...,a_{i-1}$。最后沿用得到拓扑序之后的做法即可。

算法复杂度：$O(n\log n)$

3 总结

本题题面简单精巧，所需要的知识仅是树的一些性质，却综合考察了选手制造工具、转化问题的能力，且解法中大量运用到了分治的技巧。该题对选手的代码能力要求不高，但对选手的思维有一定的要求。

解决本题需要将问题转化为“寻找原树的一种拓扑序”，在转化的过程中强调了“将给出的询问封装为一种种工具”的想法。将问题转化为“寻找拓扑序”之后，本题可以从“点分治”、“剥叶子”、“直接维护拓扑序”等多种角度切入，且都可以得到不错且相当有趣的算法。例如，我们巧妙地运用了“将点染色并划分为段”的思想得出算法六并大量使用树的性质证明了算法六的复杂度；再例如，我们利用“没有二度点的树至少有一半的点时叶子”的思想得出了算法五。

出题人认为这样一道不需要高深的知识但能深入考察选手对树的性质的了解的题称得上是好题，也希望通过本题起到抛砖引玉的作用，看到更多更有趣的题目。

4 感谢

感谢中国计算机学会提供学习和交流的平台。

感谢国家集训队教练高闻远的指导。

感谢杭州第二中学的李建老师对我的关心与指导。

感谢周欣同学、方尤乐同学、方汤骐同学等与我讨论该题算法，帮忙验题。

感谢父母对我的关心与支持。

参考文献

[1] Codeforces 1129E, Legendary Tree