**广州大学学生实验报告**

**开课实验室：**计算机科学与工程实验（电子楼418A）**2019年12月24日**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与网络工程学院 | | **年级、专业、班** | **计科185** | **姓名** | **覃浩** | **学号** | **1806100085** |
| **实验课程** | | **数据结构课程设计** | | | | | **成绩** |  |
| **实验项目** | | **数据结构课程设计** | | | | | **指导老师** |  |
| **一．课程设计题目及内容**  [问题描述]  某个太空神秘国度中有很多美丽的小村，从太空中可以想见，小村间有路相连，更精确一点说，任意两村之间有且仅有一条路径。小村 A 中有位年轻人爱上了自己村里的美丽姑娘。每天早晨，姑娘都会去小村 B 里的面包房工作，傍晚 6 点回到家。年轻人终于决定要向姑娘表白，他打算在小村 C 等着姑娘路过的时候把爱慕说出来。问题是，他不能确定小村 C 是否在小村 B 到小村 A 之间的路径上。你可以帮他解决这个问题吗？  [基本要求]  （1）输入由若干组测试数据组成。每组数据的第 1 行包含一正整数 N ( l 《 N 《 50000 ) , 代表神秘国度中小村的个数，每个小村即从0到 N - l 编号。接下来有 N -1 行输入，每行包含一条双向道路的两个端点小村的编号，中间用空格分开。之后一行包含一正整数 M ( l 《 M 《 500000 ) ，代表着该组测试问题的个数。接下来 M 行，每行给出 A 、 B 、 C 三个小村的编号，中间用空格分开。当 N 为 O 时，表示全部测试结束，不要对该数据做任何处理。  （2）对每一组测试给定的 A 、 B 、C，在一行里输出答案，即：如果 C 在 A 和 B 之间的路径上，输出 Yes ，否则输出 No.   1. **程序中使用的数据结构及主要符号说明**   本题的暴力做法的时间复杂度是O(nm)的，一个很显然的办法是求LCA（最近公共祖先），在得知A、B、C两两之间的LCA后，就可以在O(1)的时间解决这个问题。  设AB的LCA为LAB，AC的LCA为LAC，BC的LCA为LBC  容易看出，   1. 当LAC、LBC都等于C时，若LAB等于C，则C为LAB的最近公共祖先，显然AB的路径必经过C，否则，A、B、C同在一条链上，且C在链顶，显然C不在AB的路径上。 2. 若LAC和LBC中只有一个等于C时，显然A、B中有一个点位于以C的字子树中，而另一个点不在，则两点路径必然经过C 3. 其他情况C必然不在AB路径上   那么本题可以转为求解LCA  LCA的常见求法有：   1. 暴力 2. 倍增 3. 树链剖分 4. Tarjan(离线) 5. 转化为RMQ（区间最大/最小值查询）问题   注：下文的n指树的点数，m指查询个数，A-1指反阿克曼函数  显然暴力是最差的解法  而倍增的预处理是O(nlogn)的，查询是O(logn)的，空间复杂度是O(nlogn)的  树链剖分的预处理是O(n)的，查询是O(logn)的，空间复杂度是O(n)的，可以说是全方位优于倍增的  而Tarjan算法所需的预处理是O(nlogn+mA-1(n))的，查询是O(1)的，空间复杂度是O(m)的，时间和空间复杂度随着查询数增长的速度有点难以接受，且写起来常数较大  转化为RMQ问题的话可以实现预处理是O(nlogn)的，查询是O(l)的，空间复杂度是O(nlogn)的。  为了使得时空复杂度在各个方面都能让人接受，将LCA问题转化为RMQ问题是一个不错的选择。  那么如何转化为RMQ问题呢？  我们任意选定一个根节点对树进行dfs遍历，当第一次访问到某个节点或者回溯到某个节点时，都将其编号记录下来，会得到一个长度为2n-1的序列（这被称为树的欧拉序列）将节点u第一次出现在欧拉序列的位置记作pos(u)，将欧拉序列记作E[1...2n-1]  有了欧拉序列，我们可以再限线性时间内将LCA问题转化为RMQ问题  即pos(lca(u,v))=min{(pos(k)|k∈E[pos(u)..pos(v)]}  而对于RMQ问题的解决方案也是很多的，线段树，树状数组、ST表等数据结构都能解决。由于本问题不需要对序列进行修改操作，所以选择了查询速度快且非常容易实现的ST表。  ST表是一种基于倍增思想的数据结构，其预处理时空复杂度都为O(nlogn)，查询是O(1)的。   1. **程序流程图和带有注释的源程序**   **#include** <stdio.h>  **#define** **N** 100005  **#define** **M** 1000005  **#define** **LG** 20  **typedef** **struct** edge {**int** v;**struct** edge **\***next;} edge;//边的struct  edge **\***head[N], G[N **<<** 1];//存边的数组  **int** cnt, lg[N];//cnt为边的数量,lg为预处理的log值,由于浮点的log运算较慢,所以这可以使程序运行时间简短  **void** **addedge**(**int** u, **int** v) { G[cnt] **=** (edge){v, head[u]}, head[u] **=** **&**G[cnt**++**]; }//加边  **int** dfn[N **<<** 1], dep[N **<<** 1], dfncnt, pos[N], st[LG][N **<<** 1];  //dfn是存欧拉序列的数组,dep为欧拉序列元素对应的深度,pos为树上点在欧拉序列最先出现的地方,st则为st表预处理所需的空间  **void** **dfs**(**int** u, **int** d, **int** f) {//dfs遍历  dfn[**++**dfncnt] **=** u, pos[u] **=** dfncnt, dep[dfncnt] **=** d;  **for** (edge**\*** e **=** head[u]; e; e **=** e->next) **if** (e->v **!=** f) {  **dfs**(e->v, d **+** 1, u);  dfn[**++**dfncnt] **=** u, dep[dfncnt] **=** d;  }  }  **void** **pre**(**int** n) {//基于倍增的预处理  lg[0] **=** **-**1;  **for** (**int** i **=** 1; i **<** (n **<<** 1); i**++**) lg[i] **=** lg[i **>>** 1] **+** 1;  **for** (**int** i **=** 1; i **<** (n **<<** 1); i**++**) st[0][i] **=** i;  **for** (**int** i **=** 1; i **<=** lg[(n **<<** 1) **-** 1]; i**++**)  **for** (**int** j **=** 1; j **+** (1 **<<** i) **-** 1 **<** (n **<<** 1); j**++**)  st[i][j] **=** (dep[st[i **-** 1][j]] **<** dep[st[i **-** 1][j **+** (1 **<<** (i **-** 1))]]) **?** st[i **-** 1][j] **:** st[i **-** 1][j **+** (1 **<<** (i **-** 1))];  }  **inline** **int** **rmq**(**int** u, **int** v) {//查询  **int** l **=** pos[u] **<** pos[v] **?** pos[u] **:** pos[v], r **=** pos[u] **>** pos[v] **?** pos[u] **:** pos[v];  **int** s **=** lg[r **-** l **+** 1];  **return** dep[st[s][l]] **<** dep[st[s][r **-** (1 **<<** s) **+** 1]] **?** st[s][l] **:** st[s][r **-** (1 **<<** s) **+** 1];  }  **int** **main** () {  **int** n, m, A, B, C;  **scanf**("%d", **&**n);  **for** (**int** i **=** 1; i **<** n; i**++**) {//读入整棵树  **scanf**("%d%d", **&**A, **&**B);  **addedge**(A, B);  **addedge**(B, A);  }  **dfs**(0, 0, 0);  **pre**(n);  **scanf**("%d", **&**m);  **for** (**int** i **=** 1; i **<=** m; i**++**) {//查询  **scanf**("%d%d%d", **&**A, **&**B, **&**C);  **int** LAB **=** dfn[**rmq**(A, B)], LAC **=** dfn[**rmq**(A, C)], LBC **=** dfn[**rmq**(B, C)];  **if** (LAC **==** C **&&** LBC **==** C) **printf**("%s\n", (LAB **==** C) **?** "Yes" **:** "No");  **else** **if** (LAC **==** C **||** LBC **==** C) **printf**("Yes\n");  **else** **printf**("No\n");  }  }   1. **执行程序名，并打印程序运行时的初值和运算结果** 2. **实验结果分析，实验收获和体会**     编译RMQ.c并执行生成的可执行文件,从1.in文件中读入,输出结果至1.out文件  time命令用于显示程序运行的时间，1.in文件是使用编写的python程序生成的  生成了节点数为100000的一棵树以及1000000的查询。  可以看到，在不开启任何编译器优化且不使用读入优化的情况下，我们运行本程序需要大约1秒钟的时间  这里附上树链剖分以及Tarjan做法的运行时间      可以看出RMQ做法在时间上还是比较优秀的，不过由于在n和m增长的过程中，IO的速度也成了限制程序运行速度的一大问题，所以看起来不是那么明显。如果使用一些读入输出优化，可以使得程序在IO方面的影响降低，能更容易看出各个算法之间的差距、  参考资料：   1. OI wiki[最近公共祖先](https://oi-wiki.org/graph/lca/" \l "rmq) 2. 郭阳华 [《RMQ与LCA问题》](https://github.com/enkerewpo/OI-Public-Library/blob/master/IOI%E4%B8%AD%E5%9B%BD%E5%9B%BD%E5%AE%B6%E5%80%99%E9%80%89%E9%98%9F%E8%AE%BA%E6%96%871999-2019/2007/day2/3.%E9%83%AD%E5%8D%8E%E9%98%B3%E3%80%8ARMQ%E4%B8%8ELCA%E9%97%AE%E9%A2%98%E3%80%8B.ppt) | | | | | | | | |