数据关系的简化

长沙雅礼中学 何林

I. 摘要

数据之间的关系有着和数据本身同等的重要性。最常见的数据关系有线性关系、树关系和图关系。信息学竞赛的本质就是对数据的充分挖掘。然后有时候挖掘太过充分，反而会把问题复杂化。本来将讨论的就是如何通过适当的简化数据关系，实现数据的合理挖掘。

II. 关键字

树，图，序列，数据关系

III. 引言

我们经常面对大量的数据，但是他们之间不是杂乱无章的。一些用道路连接的城市表现出来的数据关系是图；一些行政部门的上司下属关系表现出来的是树关系；学校的成绩排名表现出来的数据关系是线性关系。

图、树和线性关系是我们在生活中、也是在信息学竞赛中遇到的三种最常见的关系。虽然信息学竞赛强调对输入信息的充分挖掘和应用，但有时候关系过于复杂反而让人眼花缭乱。本文就是要介绍一种重要的思想：简化数据结构。具体的说可以把图简化成树、把树简化成线性结构。这看起来不可思议，因为在简化的过程中必然会丢失一些信息，但是通过本文接下来的分析和举证，你又会发现这是切实可行的一种思想。

IV. 简化图关系

先来看一个题目：

坐船问题

雅礼中学有 n 个学生去公园划船。一条船最多可以坐两个人。如果某两个学生同姓或者同名就可以坐在一条船上。

学校希望每个同学都坐上船，但是小船的租用费用很高，学校想要租用最少的船。请问：学校至少要租多少船？

我们可以把每个学生看作一个顶点。如果两个学生同姓，就在两者之间连一条红边；如果他们同名，就在两者之间连一条蓝边。

这样我们就把问题的全部信息都**毫无遗漏**的包含在这个图里面了。剩下的问题就是求一个最小边覆盖，实际上也就是求最大匹配。

这个图并不一定是二分图，所以求最大匹配涉及到带花树，十分复杂。有没有更好的方法呢？

首先假设这个图是连通的。

下面我们用**一棵树来表示这个图**。这是一颗二叉树。每个节点的左儿子（如果存在）和它同姓，右儿子（如果存在）和它同名。

构树算法如下：

假设树中已经含有 k 个点了。如果 k=n，那么构造算法结束。否则因为整个图是连通的，剩下的点中至少有一个和当前的树连通。不妨设剩下的某个点 P 和树中的一个点 T 之间有边。

第一种情况是 P 和 T 同姓。沿着 T 的左儿子不断的“往左”走，直到到达某一个点 X 无法继续前进（也就是 X 没有左儿子）。因为 X 和 T 同姓，所以它也必然和 P 同姓；同时 X 没有左儿子，令 P 为 X 的左儿子即可。这样树的规模就增大了 1。

第二种情况是 P 和 T 同名。和第一种情况类似。

于是我们总是可以把树的规模增大 1，直到把所有的点都包含到树中。

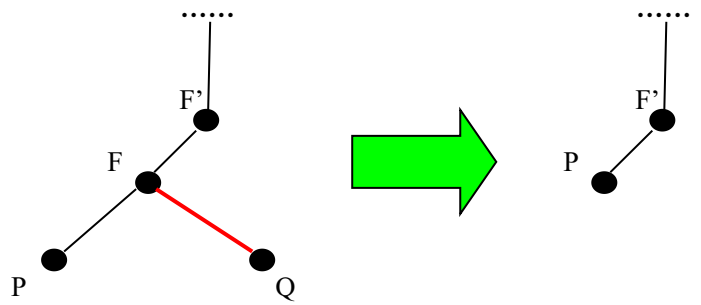
构造出这样一棵树有什么好处呢？下面考虑任意一个叶子节点 P，设它的父亲是 F。

如果 P 是 F 的唯一孩子，那么令 P 和 F 坐一条船。树的规模就减小了 2。否则 F 有两个孩子，设另一个孩子是 Q。为了方便讨论，不妨设 P 是 F 的左儿子，Q 是 F 的右儿子。

第一种情况：F 是根节点。

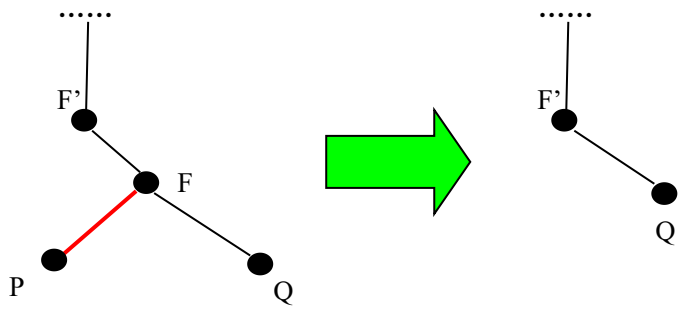
这时候总共有三个学生还没坐船。至少要两条船。令 F 和 P 坐一条船，Q 坐一条船即可。

第二种情况：F 是它父亲的左儿子。设 F 的父亲是 F’。



令 F 和 Q 坐一条船。然后把 F 和 Q 从树中删除。此时 P 的左儿子 P 落了单。因为 F 和 F’同姓，F 又和 P 同姓，所以 F’和 P 同姓。令 P 为 F’的左儿子即可。这就保证了树的连通性。

第三种情况：F 是它父亲的右儿子。



和第二种情况类似。见上图。

通过以上算法，可以不断的把树的规模减小。如果树有 n 个节点，那么最后的结果就是需要 $\left\lceil \frac{n}{2} \right\rceil$ 条船。毫无疑问这是最优解。

如果图不连通，那么把每个连通图按照上述算法分别处理，最后把船数相加即可。

整个算法的复杂度是 O(n)，不仅效率上大大优于匹配算法，而且两者的编程复杂度更是不能同日而语。

这个题目的数据是学生，数据关系是同姓或者同名。毫无疑问，用图来表示学生之间的关系是很自然的想法，而且图也的确可以毫无遗漏的把所有信息包含进去。但是正因为图的包罗万象，使得我们束手无策。虽然存在匹配算法，但是它不仅思维、编程复杂度高，而且效率也不尽人意。

在思维的路上转一个弯，我们用一棵二叉树来表现数据关系。尽管有些人之间的信息被忽略了，比如根节点和某个叶子节点可能同姓或者同名、而在树中他们两者之间也许没有直接关联的边；我们反而能够更加有效率的应用这些简化后的数据关系解题。

可见数据和数据关系的利用也不是越充分越好，有时候合理的运用更有效。

下面我们再来看一个例子。

仙人掌图的判定

如果一个有向图：

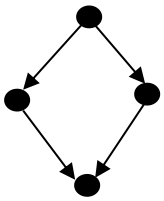
1. 它是一个强连通图。

2. 它的任意一条边都属于且仅属于一个环。

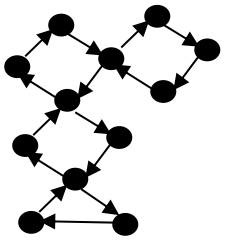
这个图就称为仙人掌图。

输入一个 n 个节点，m 条边的图（1<=n,m<=10^5），请判断它是不是仙人掌图。

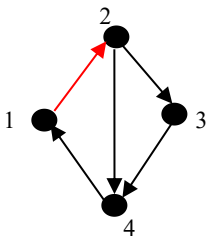
为了对仙人掌图有一个感性认识，我们先看下面三个图。



a)非强连通



b)仙人掌图



c)红色的弧同时属于两个圈

其中 a)和 c)都不是仙人掌图，因为 a)不是强连通的、c)中的红弧同时属于两个圈 1→2→3→4→1 和 1→2→4→1。

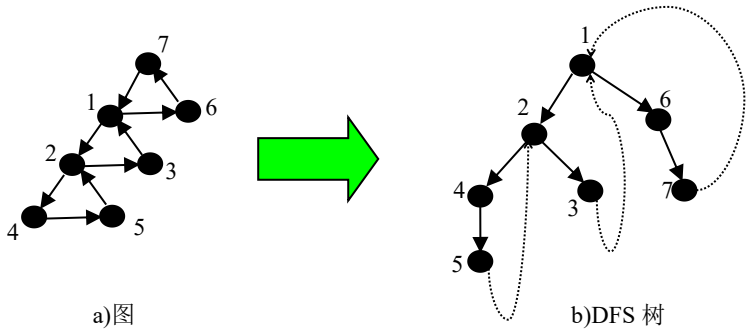
b)就是一个仙人掌。直观的说，仙人掌图就是一个一个的圈直接“粘”在一起的图，圈之间没有公共边。

于是我们很容易得到这样的一个算法：每次找一个圈，如果圈中不相邻的点之间有边，那么该图就不是仙人掌；否则把圈缩成点，然后把圈、点、边的关系进行适当的调整，继续缩圈。

权且不说具体该如何调整圈、点、边的关系，光是缩点这一项就要大费周章。该算法的思维和编程复杂度将非常高。

本文要介绍的另一个种算法是 DFS。对该图进行 DFS 遍历，建立 DFS 树。

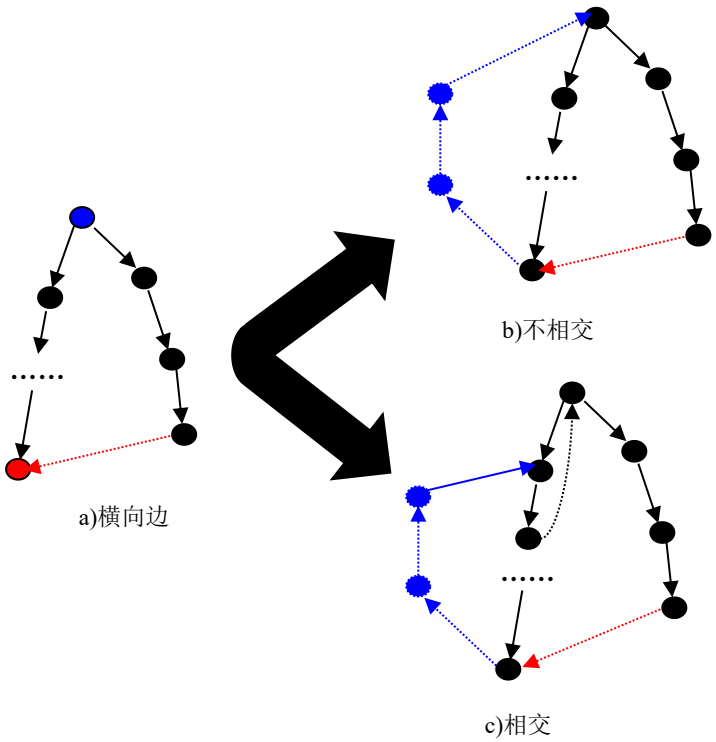
譬如下图：



a)图　　b)DFS 树

如果一个图是仙人掌的话，它的 DFS 生成树有什么特点呢？分析 DFS 树的一般方法是从逆向边和横向边入手。

首先考虑它的横向边。



a)横向边　　b)不相交

c)相交

上面 a)图中的红边是横向边。蓝点是红点的祖先，称从蓝点遍历到红点的这条路为红点的“祖先路径”，记作 P。因为仙人掌图强连通，所以必须存在一条从红点到蓝点的路。

如果这条路不和 P 相交，则如 b)图所示；如果这条路和 P 相交，则如 c)图所示。不论是 b)图还是 c)图，蓝色的点和边都同时隶属两个圈。也就是说，只要存在横向边，这棵 DFS 树的原图就肯定不是仙人掌图。

所以：

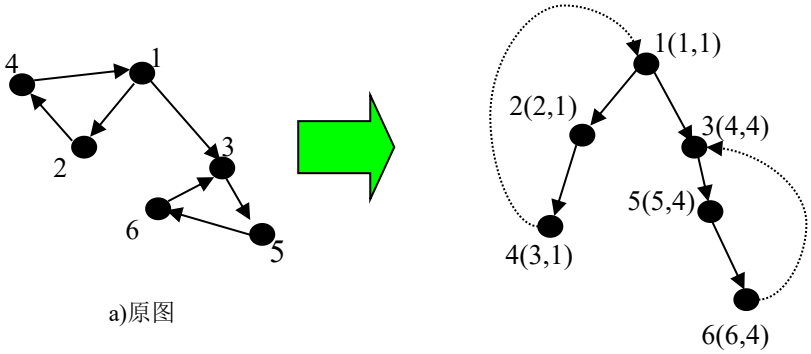
性质 1 仙人掌图的 DFS 树没有横向边。

下面我们进一步考虑逆向边。

对于每个节点 v，定义两个函数：

1. DFS(v)表示 v 在 DFS 树中是第几个被遍历到的点。
2. Low(v)表示通过从 v 以及 v 的所有后代直接指出去的边，可以访问到的 DFS 值最小的点的 DFS 值。

通过下图读者可以对 DFS 和 Low 的定义获得一个更感性的认识。



a)原图

b)DFS 树

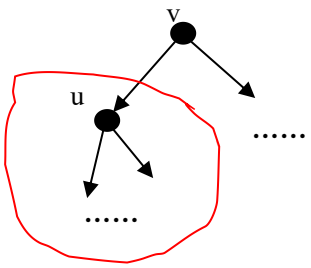
括号中第一个数是 DFS 值，第二个是 LOW 值

DFS 函数的求解可以通过设立一个计数器，在深度优先遍历的时候每碰到一个新的点就累加 1 来实现。

Low 函数的计算如下：

Low(v)=min{DFS(v), Low(u)} (其中 u 是 v 的儿子)

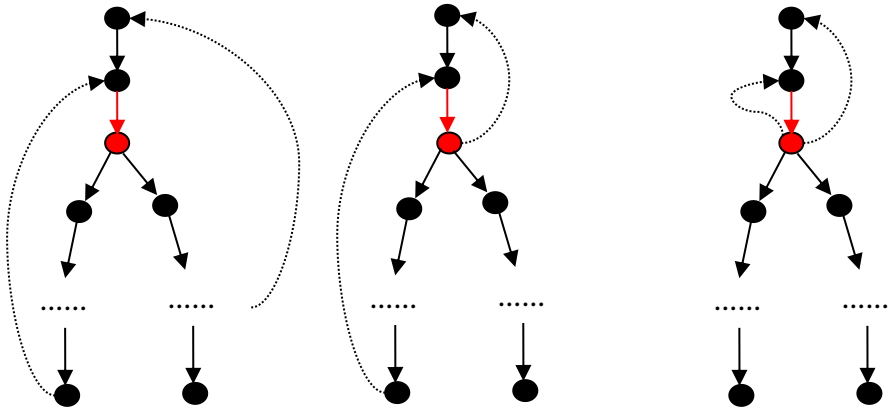
下面考虑某一个点 v。如果它存在一个儿子 u 满足 Low(u)>DFS(v)，那么这个 DFS 树的原图肯定不是仙人掌。



如上图所示，因为 Low(u)>DFS(v)，所以 u 以及 u 的后代都被限制在红色的线圈范围之内，没有指出去的逆向边。这样<v, u>就成了桥。我们知道在一个强连通图中是不可能有桥的，所以：

性质 2 Low(u)<=DFS(v) (u 是 v 的儿子)

然后看下面三个图。



a)图　　b)图　　c)图

上图中的红点都是 v。

在 a)图中，v 有两个儿子的 Low 值小于 DFS(v)，这时红边就同时属于两个圈了。

在 b)图中，v 有一个儿子的 Low 值小于 DFS(v)，同时 v 自己也有一条逆向边。这时红边也同时属于两个圈。

在 c)图中，v 有两条逆向边。这时红边同样属于两个圈。

以上三种情况下，原图都不是仙人掌。归纳起来就是：

性质 3 设某个点 v 有 a(v)个儿子的 Low 值小于 DFS(v)，同时 v 自己有 b(v)条逆向边。那么 a(v)+b(v)<2。

至此我们已经获得了仙人掌图 DFS 生成树的三条性质：

性质 1 仙人掌图的 DFS 树没有横向边。

性质 2 Low(u)<=DFS(v) (u 是 v 的儿子)

性质 3 设某个点 v 有 a(v)个儿子的 Low 值小于 DFS(v)，同时 v 自己有 b(v)条逆向边。那么 a(v)+b(v)<2。

与以上任意性质相悖的图肯定不是仙人掌；反之如果一个图的 DFS 生成树满足以上所有性质，那么它也肯定是仙人掌图（这个可以通过对生成树边和逆向边的分析证明，在此略）。

至此我们就得到了一个仙人掌图判定的 DFS 算法。时间复杂度是 O(n+e)。与“缩圈”算法比较起来，DFS 算法最吸引人的地方还在于其编程复杂度小。

我们把原图用 DFS 树的形式重新描述，这是解题过程中最本质的突破。利用仙人掌图的特殊性，我们可以把 DFS 树中的逆向边、横向边各个击破。同时因为树有祖先和后代之分，具有明显的层次感，为我们设计算法，比如 Low 函数，提供了灵感；反观原图，既没有明显的拓扑关系，仙人掌图的特殊性也无法有效的用图的性质来体现。

严格的说，仙人掌图的 DFS 判定算法并没有“简化”数据关系，它只是把数据关系用一种更有次序、“原始”的方式表现了出来。形式的简化将数据之间的关系更加清晰的浮上了水面。或许通过对图的分析也能最终解决问题，但是其过程无疑要复杂得多。

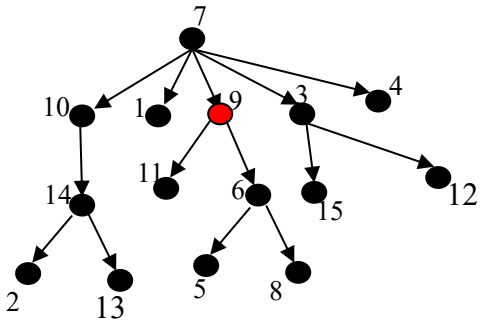
V. 简化树关系

先看一道题目：

树的统计

一棵含有 n 个节点的树，所有的节点依次编号为 1, 2, 3, ..., n。对于编号为 v 的节点，定义 t(v)为 v 的后代中所有编号小于 v 的节点个数。输入这棵树，请输出 t(1), t(2), t(3), ..., t(n)。

我们当然可以毫不费力的得出一个 $O(n^2)$的算法，但时间复杂度太高。通过对树本身的分析，我们也可以设计出一个 O(nlogn)级别的算法，但涉及到树的拆分等很复杂的操作(对此算法有兴趣的同学可以联系我)。下面我们来看一种别出心裁的算法（注：此题的命题者和该算法提供者是雅礼中学的雷涛同学）。



我们深度优先遍历该树，然后按照访问到的先后顺序把节点依次写下来：

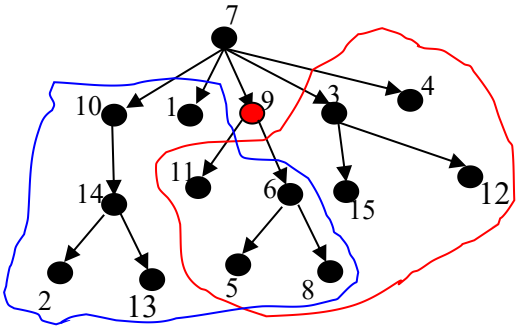
7 10 14 2 13 1 **9** 11 6 5 8 3 15 12 4（该序列称为“DFS 序列”）

我们发现数字 9 后面的部分正是下图中用红线框出来的区域。

然后我们把每个节点的儿子先后顺序倒过来，重新遍历，得到的序列如下：

7 4 3 12 15 **9** 6 8 5 11 1 10 14 13 2（该序列称为“逆 DFS 序列”）

这个序列中数字 9 后的部分正是下图中用蓝线框出来的区域。



很容易发现红蓝两个线圈有一个重叠区域，而这个区域正是“9 的所有后代”。这就提示我们用容斥原理。

图中除了红线和蓝线囊括的区域外，还有一个未被触及的盲区：那就是 9 本身和它的直系祖先。

定义 f(v, S)表示在 S 所描述的集合或者区域中，有多少个节点是小于 v 的。

我们可以得到如下公式：

f(9,红)+f(9,蓝)+f(9,9 的直系祖先)-f(9,整棵树)=f(9,9 的后代)

推广一下就是：

f(v,DFS 序列中 v 之后的部分)+f(v,逆 DFS 序列中 v 之后的部分)+f(v,v 的直系祖先)-f(v,整棵树)=f(v,v 的后代)

很容易观察到 f(v,整棵树)=v-1，而 f(v,v 的后代)就是 t(v)，所以：

t(v)=f(v,DFS 序列中 v 之后的部分)+f(v,逆 DFS 序列中 v 之后的部分)+f(v,v 的直系祖先)-(v-1)

通过一次 DFS 遍历联合线段树就能以 O(nlogn)的时间复杂度求出所有节点的直系祖先中有多少个比它本身小。对于 DFS 序列和逆 DFS 序列，可以采用线段树或者树状数组求出序列每个元素后面有多少个比它本身小；时间复杂度 O(nlogn)。

综上，O(nlogn)的时间复杂度即可解决此题。

这个算法最巧妙的地方是把树对应到了两个序列：DFS 序列和逆 DFS 序列。本来在树中，某个节点和它的后代之间存在具有层次性的拓扑关系；通过变换成序列后，这个关系简化成了一个线性序列中的先后关系。正是这个“简单”的先后关系使我们能够用线段树或者树状数组来解决问题。

下面我们来看一个更加经典的问题：

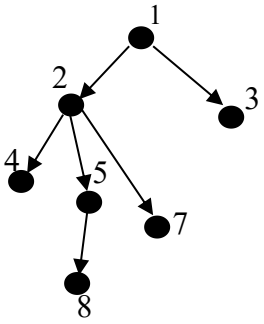
最近公共祖先

已知一棵 n 个节点的树，你会不断收到这类询问：“p 和 q 的最近公共祖先是什么？”。询问总数可能很大。请你设计算法回答所有询问。

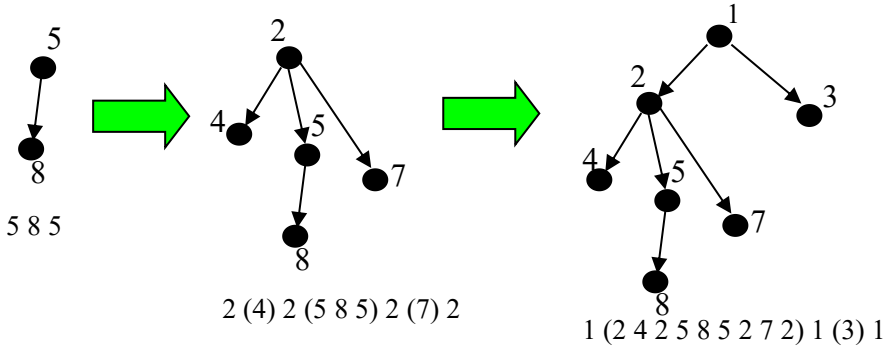
我们考虑把树用一个序列表示出来。

这是一个递归的过程。**如果整棵树就是一个节点，对应的序列就是(Root)；否则把根节点的所有子树分别表示成序列 $L_1, L_2, \ldots, L_t$，原树的序列就是(Root, L_1, Root, L_2, Root, L_3, …, L_{t-1}, Root, L_t, Root)。**

譬如下图：



我们可以这样把它变成一个序列：



我们将节点的深度写在旁边：

1(1) 2(2) 4(3) 2(2) 5(3) **8(4) 5(3) 2(2) 7(3)** 2(2) 1(1) 3(2) 1(1)

如果要求两个节点，比如 8 和 7 的最近公共祖先。考察序列中 8 和 7 之间的部分，如上红色部分标记的，其中深度最小的点是 2。所以 8 和 7 的最近公共祖先就是 2。

又比如求 5 和 3 的最近公共祖先：

1(1) 2(2) 4(3) 2(2) **5(3)** 8(4) **5(3)** 2(2) 7(3) 2(2) 1(1) 3(2) 1(1)

注意到序列中有两个 5，应该选哪一个呢？实际上任意选一个都没有影响，比如选如下蓝色部分：

1(1) 2(2) 4(3) 2(2) **5(3) 8(4) 5(3) 2(2) 7(3) 2(2) 1(1) 3(2)** 1(1)

其中深度最小的点是 1，所以 5 和 3 最近公共祖先就是 1。

一般的说：

在序列中任意找一个 p，任意找一个 q。在序列中 p 和 q 之间的部分，找一个深度最小的点，就是 p 和 q 的最近公共祖先了。

下面证明算法的正确性。回头看看我们的序列构造方法：

这是一个递归的过程。如果整棵树就是一个节点，对应的序列就是(Root)；否则把根节点的所有子树分别表示成序列 $L_1, L_2, \ldots, L_t$，原树的序列就是(Root, L_1, Root, L_2, Root, L_3, …, L_{t-1}, Root, L_t, Root)。

正确性的证明也是一个递归的过程。假设算法对于 $L_1, L_2, \ldots, L_t$ 这些序列都是正确的。如果两个点 p 和 q 属于同一个 L_i，那么该算法正确。否则 p 属于 L_i，q 属于 $L_j(i \neq j)$，那么 p 和 q 的最近公共祖先肯定是 Root；因为构造序列的时候在任意两个相邻的 L 序列之间都插入了一个 Root，而 Root 的深度又是最小的，因此在这种情况下算法也是正确的。

至此问题基本解决。具体如何求序列中某一连续段中的最小值，可以用线段树(回答一次询问的时间复杂度 O(logn))，也可以用经典的 01RMQ 算法(回答一次询问的时间复杂度 O(1))。

求最近公共祖先是一个很重要的问题，也是一个树转化到序列的经典例子。

VI. 总结

本文的主题是数据关系的简化。两种重要的简化是图→树以及树→序列。本文通过列举四个例子，希望让读者获得对“数据关系简化”这一思想的感性认识。

从严格的意义上来说，所谓简化只是一种形式的变化。用更简单、原始的数据结构表达本来用复杂数据结构表现的关系。这种“简单化”带来的往往是更加清晰的数据关系，为窥见问题根本矛盾创造条件。

具体的说，利用 DFS 生成树和 DFS 序列来解决与图、树有关的问题，是一种很实用有效的思想。

除了 DFS 当然还有其他的数据关系简化的应用。比如图论中的“弦图判定”定义了“完美消除序列”，就是一个很好的例子。

参考资料

1. 湖南省赛试题
2. 2000 年信息学国家集训队队员肖州的论文
3. <Introduction to Algorithms>
4. 湖南长沙雅礼中学雷涛同学的原创试题