arXiv:2302.01373v1 [cs.DS] 2 Feb 2023

最优最重诱导祖先

Panagiotis Charalampopoulos[1], Bartłomiej Dudek[2], Paweł Gawrychowski[2], 和 Karol Pokorski[2]

[1]伦敦大学伯贝克学院，英国，[2]计算机科学研究所，弗罗茨瓦夫大学，波兰

摘要

我们重新研究由 Gagie、Gawrychowski 和 Nekrich 在 [CCCG 2013] 上提出的最重诱导祖先（HIA）问题，该问题在字符串算法中有多种应用。设 T_1 和 T_2 为两棵有根树，其节点权重沿所有根到叶路径递增，叶节点带有标签，且每棵树中没有两个叶节点具有相同标签。当且仅当 u 和 v的叶后代共享一个标签时，节点对 $(u,v) \in T_1 \times T_2$ 才是诱导的。在 HIA 查询中，给定节点 $x \in T_1$ 和 $y \in T_2$，目标是找出总权重最大的诱导节点对 (u,v)，使得 u 是 x 的祖先，且 v 是 y的祖先。

设 n 为两棵树的大小上界。已知不存在大小为 $\mathcal{O}(n)$ 的数据˜结构，能够在 $o(\log n/\log\log n)$ 时间内回答 HIA 查询 [Charalampopoulos、Gawrychowski、Pokorski；ICALP 2020]。[1] 这一（无条件）下界与迄今已知用于 HIA 问题的最快、大小为 $\mathcal{O}(n)$ 的数据结构的查询时间相比，仍有一个 polyloglog。 n 因子的差距 [Abedin、Hooshmand、Ganguly、Thankachan；Algorithmica 2022]。在本工作中，我们通过提出一种可在 $\mathcal{O}(n)$ 时间内构建、并能在 $\mathcal{O}(\log n/\log\log n)$ 时间内回答 HIA 查询 ˜ 的数据结构，解决了近线性空间˜范围内 HIA 问题的查询时间复杂度。作为直接推论，我们得到一种大小为 $\mathcal{O}(n)$ 的数据结构，可维护一个静态字符串与一个动态字符串的最长公共子序列，两者长度均至多为 n，且在这一空间范围内达到最优时间。

我们的方法主要运用了分数级联和 $\mathcal{O}(\log n/\log\log n)$-深度树分解。后者使我们得以突破以往工作的 $\Omega(\log n)$ 下界；这一限制源于所考虑的重路径分解的深度。

1 引言

针对文本中算法问题的解决方案，往往需要构造文本索引。这类索引既能高效构建，又具有广泛的功能，同时不会显著增加空间开销。后缀树就是这类索引的典型代表，在字符串学中应用广泛。Weiner 的工作 [22] 首次提出了后缀树，并表明它可用于高效解决一系列基础开放问题，例如计算文本中模式串（以在线方式给定）的出现位置，或计算两个字符串的最长公共子串。然而，通常需要先为后缀树增添其他数据结构，它才能高效回答更复杂的查询，例如返回两个子串的最长公共前缀，或某个位置为中心的最长回文；在这些例子中，增添最近公共祖先数据结构便已足够[16, 17]。

关键的是，文本索引（例如后缀树）只需构建一次，之后便可进行任意多次查询。这一点正变得越来越重要：在许多现实场景中，大量

[1] $\tilde{\mathcal{O}}(\cdot)$ 记法隐藏了关于 n的多对数因子。

Size	查询时间	论文
$\tilde{\mathcal{O}}(n)$	$\Omega(\log\ n/\log\log\ n)$	[8]
$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n/\log\log\ n)$	[1]
$\mathcal{O}(n$对数 $n)$	$\mathcal{O}(\log\ n\log\log\ n)$	[1]
$\mathcal{O}(n$对数$^2\ n)$	$\mathcal{O}($对数 $n)$	概述于 [14], 参见附录A
$\tilde{\mathcal{O}}(n)$	$\mathcal{O}($对数 $n/$对数 对数 $n)$	本文
$\mathcal{O}(n^{1+\epsilon})$	$\mathcal{O}(1)$	[8]

表1：HIA 问题的大小与查询时间权衡；大小以机器字为单位。

信息存储在服务器上，并不断被大量远程客户端查询。从这个角度来看，花一些时间预处理服务器上存储的数据是合理的，以便之后能够快速响应远程用户。

由 Gagie 等人 [14] 提出并在下文定义的最重诱导祖先问题，已被证明有助于解决多个关于计算两个字符串最长公共子串的问题变体 [1, 4, 5, 8, 14]。

如果一棵树的每个节点 u 都关联一个权重，且根到叶路径上的权重递增，即对于任意非根节点 u，u 的权重大于 u的父节点的权重，则称这棵树为带权树。此外，如果一棵树的每个叶节点都被赋予一个来自 $[n]$ 的不同标签，其中 n 是叶节点的数量，则称这棵树为带标记树。以字符串 $S\$$ 的后缀树为例，其中 \$ 在 S 中没有出现；每个叶节点的标签为对应后缀的起始位置，每个节点的权重则为其所表示字符串的长度。

定义1. 对于两个以 n 个叶节点为根且带权的树 T_1 和 T_2，我们称两个节点 $u \in T_1$ 和 $v \in T_2$ 是由标签 ℓ 诱导的，当且仅当存在分别标记为 ℓ 的叶节点 x 和 y，使得 x 和 y 分别是 u 和 v 的弱后代。

最重诱导祖先（HIA）**输入：** 两个以 n 个叶节点为根、带权且带标记的树 T_1 和 T_2。
查询： 给定一对节点 $u \in T_1$ 和 $v \in T_2$，返回一对总权重最大的诱导节点（u'，v'），使得 u' 是 u 的祖先，且 v' 是 v 的祖先。

先前成果与我们的贡献。表1展示了我们开展这项工作之前针对 HIA 问题的最先进空间与查询时间权衡方案，以及我们的结果。Gagie 等人 [14] 提出了若干种权衡方案，此后这些方案均得到改进。我们强调，表1中列出的、大小为 $\mathcal{O}(n\log^2 n)$ 且查询时间为 $\mathcal{O}(\log n)$ 的数据结构仅在文献 [14] 中作了概述。我们在附录 A 中简要讨论这一概述，因为其中一些思路与我们~采用的思路相似。表1中其余已知的大小为 $\mathcal{O}(n)$ 的数据结构由 Abedin 等人 [1] 提出。Charalampopoulos 等人 [8] 给出了近线性规模数据结构的无条件下界，并提出了一种查询时间为 $\mathcal{O}(1)$、大小为 $\mathcal{O}(n^{1+\epsilon})$ 的数据结构，其中 $\epsilon > 0$ 为任意常数。现在我们正式陈述主要结果，该结果达到了 [8] 给出的下界。

~**定理 2**。 存在一种用于 *HIA* 问题的数据结构，其大小为 $\mathcal{O}(n)$，可在~$\mathcal{O}(n)$ 时间内构建，并可在 $\mathcal{O}(\log\ n/\log\log\ n)$ 时间内回答查询。

HIA 的应用。在讨论 HIA 问题在字符串算法中的一些具体应用，以及我们的结果对这些应用的影响之前，我们先从高层次说明一下，在计算最长公共子序列的各种变体中，HIA 问题是如何出现的。

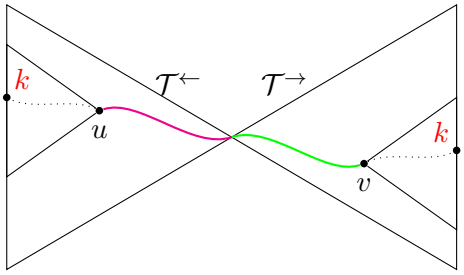

图 1：用于计算最长公共子序列的锚定技术示意图。构造出的字典树 $\mathcal{T}^{\leftarrow}$ 和 $\mathcal{T}^{\rightarrow}$ 绘制为根节点相连（位于中间）。任何锚定在 k处的子串，都可以通过从某个节点 $u \in \mathcal{T}^{\leftarrow}$ 到某个节点 $v \in \mathcal{T}^{\rightarrow}$，由左至右读取边标签来获得；这两个节点都具有一个标签为 k的叶节点后代。

考虑一个字符串 S，以及从其位置中选取的一个子集 A，我们称这些位置为锚点。再考虑以下两棵字典树：一棵字典树 $\mathcal{T}^{\leftarrow}$，用于存储 $\{S[1\,..\,k-1]^R$ 中的字符串：$k \in A\}$，其中 U^R 表示 U 的反转；另一棵字典树 $\mathcal{T}^{\rightarrow}$，用于存储 $\{S[k\,..\,|S|]$ 中的字符串：$k \in A\}$。换句话说，对于每个锚点 $k \in A$，第一棵字典树中都有一条路径表示 $S[1\,..\,k-1]^R$ 的每个前缀，第二棵字典树中都有一条路径表示 $S[k\,..\,|S|]$ 的每个前缀。我们用对应的锚点标记两棵字典树的每个叶节点。现在注意，跨越锚点 k 的子串 $S[i\,..\,j]$，即 $i < k \le j$，对应于字典树中的一对诱导节点。确实，第一棵字典树中有一条路径表示 $S[i\,..\,k-1]^R$，第二棵字典树中有一条路径表示 $S[k\,..\,j]$。图 1 展示了这一思路。具体应用中使用的锚点集合和执行的 HIA 查询，取决于该技术所应用的具体问题。对于某些用途，也可以考虑使用压缩字典树 [18]。

作为第一个应用，我们考虑维护静态字符串 T和动态字符串 S的最长公共子序列。将我们的 HIA 数据结构代入 [8], 的方法，可得到以下结果：将最先进水平提升了 $\mathrm{polyloglog}\, n$ 倍，并在可用近线性空间时达到更新时间下界 [8, 定理 1]。

推论 3. 我们可以维护动态字符串 S 和静态字符串 T的最长公共子序列，两者长度均至多为 n，每次替换操作的时间为 $\mathcal{O}(\log\, n/\log\log\, n)$，使用空间 $\mathcal{O}(n)$，预处理时间为~~an~~ $\mathcal{O}(n)$。

此外，[14] 的作者（隐式地）将问题归约为 HIA 问题：对以 LZ77 压缩形式给定的文本进行预处理，使得能够在线计算该文本与未压缩模式串的最长公共子序列。我们的 HIA 数据结构得出以下结果。

推论 4. 设 S 是一个长度为 N 的字符串，其 *LZ77* 分解由 n 个短语组成。我们可以用 $\mathcal{O}(n\log\, N + n\,\mathrm{polylog}\, n)$ 空间存储 S，使得给定一个长度为 m的模式串 P，我们可以在 $\mathcal{O}(m\log\, n/\log\log\, n)$ 时间内计算 S与 P的最长公共子序列。对于每个模式串 P，返回的结果可能以逆多项式概率（对所有查询保持一致地）出错，其中 n。[2]

HIA 问题在字符串算法中的其他应用可见于 [1]。

树分解。 在树上高效解决算法问题的一种显而易见的分治技术，是将树分解为更小的部分，并分别处理每一部分。树分解最重要的属性通常是其深度，即一条路径最多会与多少个部分相交，以及每个独立部分的结构（例如，部分若为路径，可能会带来优势）。接下来，我们介绍几种针对含有 n 个节点的树的树分解。对于节点 v，记 $s(v)$ 为 v 的子树中的节点数。

[2] 随机化仅用于构造；针对同一模式串的所有查询都会给出相同的结果 ts.

可以说，最著名的树分解是重路径分解 [17]。抽象来说，这种分解是将边划分为轻和重两类，使得：

- 删除轻边后得到的所有连通分量都是路径，称为重路径；
- 每条根到叶路径都由 $\mathcal{O}(\log n)$ 个重路径前缀和 $\mathcal{O}(\log n)$ 条轻边组成，即分解的深度为 $\mathcal{O}(\log n)$。

重路径分解有多种实现方式，以下是其中两种：

- HP1：树中每个非叶节点 u 都选择一个 v 最大的子节点 $s(v)$，并将从 u 到 v 的边指定为重边。从 u 出发的其余边均为轻边。
- HP2：当且仅当 $\lfloor \log s(u) \rfloor = \lfloor \log s(v) \rfloor$ 时，边 (u, v) 被指定为重边。[3]

直观地说，借助重路径分解，通常可以将仅适用于路径和/或平衡树的算法推广到任意树上——当然通常会带来一些额外开销。

所有先前关于 HIA 问题的研究都使用了重路径分解，正如前文所述，其深度为 $\Omega(\log n)$。这会对查询时间产生不利影响，因为查询时可能需要沿着根到叶路径遍历分解。因此，为了实现次对数查询时间，我们考虑了深度更小的树分解。有几种重路径分解的推广形式具有所需的深度，即 $\mathcal{O}(\log n/\log\log n)$。接下来我们讨论其中两种同样基于将边划分为轻边和重边的分解。需要注意的是，对于这两种分解，移除轻边后得到的连通分量都是树，我们称之为重树，而非路径，因此还需要做一些额外工作。[4]

重 α-树分解由 Bille 等人提出，[6], 其深度为 $\mathcal{O}(\log_\alpha n)$，其定义与 HP1 类似：每个非叶节点 u 选择其（至多）α 个按子树大小衡量最重的子节点；从 u 到这些子节点的边被指定为重边，而从 u 出发的其余边则被指定为轻边。令 $\alpha = \lfloor \log n \rfloor$ 即可得到所需的深度。

另一种选择是 Alstrup 等人提出的所谓 ART 分解 [2], ，对于输入整数参数 b，其深度为 $\mathcal{O}(\log_b n)$。为便于表述，我们令 b 等于 $\lfloor \log n \rfloor$，这样分解的深度为 $\mathcal{O}(\log n/\log\log n)$。边的划分是 ART 分解，当且仅当每棵重树中，拥有多个子节点（在重树中）的节点数为 $\mathcal{O}(\log n)$。Alstrup 等人 [2] 展示了如何通过计算一组最靠近叶子的轻边（沿用 HP2 的思路，但将对数的底从 2 改为 $\lfloor \log n \rfloor$），移除这些边及其后代，然后递归处理，从而计算 ART 分解。为方便起见，我们在此计算一种类似于重路径分解 HP2 实现的 ART 分解：当且仅当 $s(u)$ 和 $s(v)$ 都属于 $(n/\lfloor \log n \rfloor^{k+1}, n/\lfloor \log n \rfloor^{k}]$ 时，边 (u, v) 才是重边。对于每棵重树，我们将其最大向下树路径称为 分支；在这些路径中，除最深节点外，所有节点在重树中都恰有一个子节点；每棵重树有 $\mathcal{O}(\log n)$ 个分支。

我们的方法。为了回答节点 u 和 v 的 HIA 查询，我们考虑 $\mathcal{O}(\log n/\log\log n)$ 对重树，每一对由一棵位于根到 u 的路径上的重树和一棵位于根到 v 的路径上的重树组成。对于每一对，我们计算这两棵重树中分别为 x 和 y 的祖先且总权重最大的诱导对

[3]在一些研究中，这被称为质心分解 [10]。不要将其与通过递归删除质心节点而得到的树的层次分解混淆；质心节点是指删除后会将树分成三个大小大致相等的部分的节点 [7, 15]。[4]重树有时也称为微树，而通过收缩 T 中的每棵微树得到的树称为宏树。为避免与所谓的微宏分解 [3],混淆，我们不采用这种记法。对于正整数 $k \le n$，微宏分解是将 T 的顶点划分为 $\mathcal{O}(n/k)$个集合，使得每个集合 S 的大小为 $\mathcal{O}(k)$，它在 T 中诱导出一棵子树，并且其中至多有两个顶点的邻居不属于 S。

。与以往的工作类似，我们注意到，并非每一对重树都需要考虑。实际上，只需考虑数量与树分解深度呈线性关系的若干对重树；这一过程类似于判断有序列表中是否存在两个元素之和等于目标值 t 的自然算法：从两个指针开始，一个指向列表开头，另一个指向列表末尾，并让它们各自只朝一个方向移动（向右或向左）。对于每一对重树，我们构建数据结构，以便高效处理各种情况： u 和 v 的最低祖先在重树中的位置，与同标签叶节点的最低祖先（位于重树中）的位置之间的关系。每种数据结构考虑的情况与以往工作中的类似，但现在我们处理的是两棵树，而非两条路径，因此需要更加谨慎。通过这种方式，我们将一个 HIA 查询转化为 $\mathcal{O}(\log\ n/\log\log\ n)$ 次前驱查询。分别回答这些前驱查询后，我们得到一个能在 $\mathcal{O}(\log n)$ 时间内回答 HIA 查询的数据结构。实际上，在我们的情形下，每次前驱查询若独立回答，都需要 $\Omega(\log\log n)$ 时间。

然而，关键的是，我们展示了如何设计数据结构，使得所有前驱查询只需要两个值： u 的先序编号，或 v的先序编号。为此，我们重新排列树，使重边位于最后。这样，较大的先序编号就对应于分支上最低祖先更大的深度。这意味着，将我们的方法与分数级联结合起来，可以更快地回答所有前驱查询；对于每个查询值，第一次查询需要 $\mathcal{O}(\text{loglog } n)$ 时间，之后的每次查询只需 $\mathcal{O}(1)$ 时间。最后一个技术难题在于，分数级联要求所谓的底层目录图具有多对数度 [21]。如果 T_1 和 T_2 的度都是多对数度，那么构造这样的图很简单：大致而言，只需考虑两棵树的笛卡尔积，其中的节点分别表示 T_1 和 T_2各自的分支和重树。在一般情况下，我们通过在取笛卡尔积之前降低这些树的最大度，同时保留它们的所有优良性质，克服了这一困难。

2 预备知识

我们用 $[n]$ 表示集合 $\{1, 2, \ldots, n\}$。在整篇论文中，我们对 T_1 和 T_2 执行相同的操作，并在这些树中定义对象，因此我们将用 $T_\star$ 表示其中任意一棵树。同样，我们将用 $v_\star$ 表示 T_2 中的一个节点 $v_1 \in T_1$ 或节点 v_2 等，以此简写某个性质同时适用于 v_i 和 $i \in \{1, 2\}$的情况。

最近公共祖先。 经过$\mathcal{O}(n)$时间的预处理 [17]后，可以在常数时间内回答最近公共祖先查询。

最近公共祖先（LCA）**输入：** 一棵有根树 T。**查询：** u 和 v的公共祖先中，深度最大的节点是什么?

前驱查询。 对于一个静态集合 S，结合 x-快速字典树 [23] 和确定性字典 [20] ，可以得到一个大小为 $\mathcal{O}(n)$ 的数据结构，该结构可在 $\mathcal{O}(n)$ 时间内构建，并能以 $\mathcal{O}(\log\log\ U)$ 时间确定性地回答前驱查询（参见 [12, 命题 2]）；这是最优的 [19]。

前驱查询**输入：** 一个由 S 个来自 $[U]$ 的 n 整数构成的集合。**查询：** 给定一个整数 y，满足 $x \le y$ 的最大 $x \in S$ 是什么?

区间最小值查询。 经过 $\mathcal{O}(n)$ 时间的预处理 [11, 13] 后，区间最小值查询可在常数时间内完成。对于每个 i， $S[i] := |U| - S[i]$，我们可以得到一个用于对称区间最大值查询问题的数据结构。

区间最值查询（RMQ）
输入： 一个由 $[U]$中的 n 个整数构成的序列 S 。
查询： 给定位置 i和 j，且满足 $1 \leq i \leq j \leq n$，最小值是多少（或最小值所在的位置是）在 $S[i]$、 $S[i+1]$、.....、 $S[j]$中?

确定性静态字典。 字典是一种存储一组键（通常还关联有相应的值），并支持回答成员查询（或获取给定键对应的值）的结构。有多种随机化解法，但也存在一种确定性˜解法，只需 $\mathcal{O}(n)$空间、 $\mathcal{O}(n)$时间预处理，并支持常数时间查询 [20]。

分数级联。 考虑一个有向图 C，称为目录图，其每个节点中都存有一个有序列表（也称为目录）。设列表的总大小为 n。现在，假设我们要回答以下类型的查询：给定 C的一个连通子图 G 和一个查询值 v，在存储于 G各节点中的列表里，分别找出 v 的前驱。

解决这个问题的一种朴素方法是忽略任何预处理，并在 G 的每个节点的有序列表中分别进行二分搜索，总耗时为 $\mathcal{O}(|G|\log\ n)$。分数级联是一种通用的优化技术，可以加速在多个相关对象有序序列中针对同一值进行的多次二分搜索。

如果目录图中每个节点的度都有常数上界，那么 Chazelle 和 Guibas 的原始解法 [9] 能够在比较模型下经过线性时间预处理后，以 $\mathcal{O}(\log n + |G|)$ 时间回答查询。准确地说，目录图只需满足局部有界度 （根据 [9] 的定义1）。遗憾的是，这在我们的情形中并无用处。Shi 和 J´aJ´a [21] 后续的工作在字 RAM 计算模型中，针对最大度为多对数级的图实现了相同的复杂度（原始描述中的图是一棵树，不过，将其扩展到 Chazelle 和 Guibas [9] 所考虑的一般情形并无困难）。关键的是，这些数据结构也能处理在线方式逐个给出 G 中节点的情形；唯一的要求是，每个节点（第一个节点除外）都必须是此前某个节点的邻居。注意，复杂度中的 $\mathcal{O}(\log n)$ 项来自在所考虑列表中的第一个列表上进行二分搜索。随后，在每个列表中，查询值 v 的前驱可通过跟随常数个指针获得，而这些指针可在 $\mathcal{O}(1)$ 时间内取出。（在预处理阶段，会以适当的方式扩˜充目录，并构造这些指针。）在字 RAM 计算模型中，可以借助其他数据结构更快地解决第一次查询；例如，使用上述用于前驱查询问题的数据结构，耗时为 $\mathcal{O}(\log\log\ U)$。

3 ˜一个 $\mathcal{O}(n\ \mathcal{O})$-大小、具有最优查询时间的数据结构

令 $b > 1$ 为稍后选定的参数。考虑一棵有根、带权且带标记的树 T。节点 u 的权重记为 weight(u)。对于节点 v，我们用 $s(v)$ 表示 v的子树中节点的数量，包括 v。对于整数 k，当且仅当$n/b^{k+1} < s(v) \leq n/b^k$时，节点 v 位于第 k 层。连接同一层节点的边称为重边，其他边称为轻边。不包含任何轻边的极大子树称为重树。需要强调的是，重树可能是单节点树。

我们将每棵重树分解为若干分支，即由节点组成的极大向下树路径，其中除最深节点外，每个节点在重树中都只有一个子节点。最后一个节点要么是重树的叶节点，要么至少有两个子节点。注意，分支也可能

仅由一个节点组成。我们称一个节点为隐式节点，当且仅当它是重树中有一个子节点的内部（非叶）节点；否则，我们称它为显式节点。对于一棵重树，我们通过收缩其隐式节点的任一关联边，消除所有隐式节点，从而得到它的一个压缩版本，称为压缩重树。示例见图2。

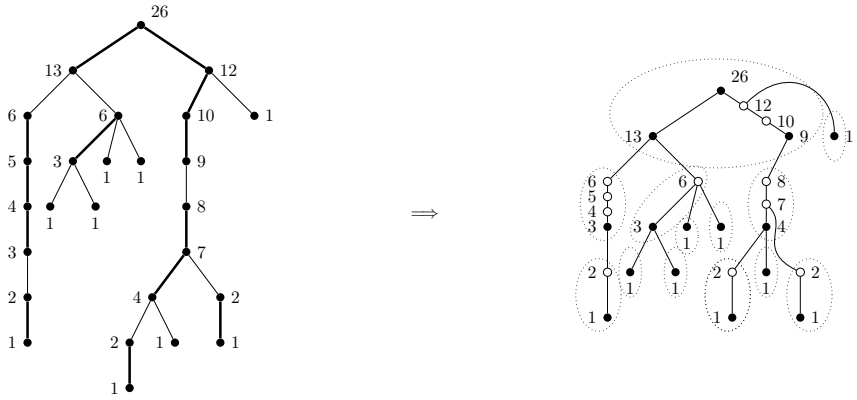

图2：展示了一个带有 $n = 26$ 的示例树。每个节点都标注了其子树大小，而 $b = 3$。左侧的重边较粗，轻边较细。右侧将树分解为重树（以椭圆标记）。重树的压缩示意如下：空心圆表示隐式节点，实心圆表示显式节点。

注意，在压缩树中，每个非叶节点至少有两个子节点。因此，内部节点的数量少于树的叶节点数量。由于重树中的每个叶节点下方都有一个足够大的子树，我们可以得到单个压缩重树中节点总数的一个界；这表明我们得到了 ART 分解 [2]。

引理 5。 树中至多有 $\mathcal{O}(\log\ n/\log\ b)$ 层，树 T 有 n 个叶节点，并且每个重树有 $\mathcal{O}(b)$ 个分支。

证明。 考虑树 H 及其压缩形式 H_C。设 r 为 H的根。对于 H的每个叶节点 ℓ，有 $s(\ell) > s(r)/b$，因为 ℓ 和 r 是同一棵重树中的节点，因此位于 T的同一层。由于以 H 的叶节点为根的 T 子树两两不相交，且其总大小至多为 $s(r)$，因此 H 至多包含 b 个叶节点。此外，由于 H_C中不存在只有一个子节点的内部节点， H_C中至多有 $b-1$ 个非叶节点，因此 H_C 有 $\mathcal{O}(b)$个节点。 H 中的所有分支两两不相交，因此共有 $\mathcal{O}(b)$个分支。

考虑树 T 的第 k 层，以及该层中的一个节点 u。由于 u 的子树至少包含一个节点，因此有 $1 \le s(u) \le n/b^k$，所以 $k \le \log_b n = \log\ n/\log\ b$。 □

对于 HIA 查询 (v_1, v_2)，从 v_1 和 v_2 到各自树根节点的路径称为查询路径。HIA 查询的结果是一对节点 (u_1, u_2)，它们分别位于从 v_1 和 v_2出发的查询路径上。为回答查询 (v_1, v_2)，我们确定包含从 v_1 和 v_2出发的查询路径上节点的重树序列 B_1 和 B_2，并对其中若干对重树执行受限 HIA 查询。更准确地说，在算法的每一步中，选定两棵重树 $B_1[i]$ 和 $B_2[j]$后，我们尝试找出 (v_1, v_2) 的诱导祖先对 $(u_1, u_2) \in B_1[i] \times B_2[j]$，使其组合权重最大；若不存在这样的祖先对，则予以判定。该过程的伪代码见算法 1。

为了使描述更具模块化，我们将 x_1、x_2 提供给受限 HIA 查询，其中x_i 是针对 v_i 在所考虑的重树中 $i \in \{1, 2\}$的最低祖先。注意， x_1 要么是 $B_1[i+1]$ 的根节点的父节点（如果 $i + 1 < |B_1|$），要么 x_1 等于 v_1 （否则）， x_2也同样如此。注意，在算法1中，我们针对同一对$(v_1$、$v_2)$提出受限 HIA 查询

算法 1： 用于回答 HIA 查询的算法。

```
1  函数hia(T_1, T_2, v_1, v_2)
2      (r_1, r_2) ← 空值
3      (B_1, B_2) ← 从 v_1 和 v_2 向下沿树的路径上的重树序列
4      i ← 0
5      j ← |B_2| − 1
6      重复
7          x_1 ← B_1[i]中 v_1 的最低祖先
8          x_2 ← v_2 在 B_2[j]中的最低祖先
9          (u_1, u_2) ← 受限 HIA(v_1, v_2, B_1[i], B_2[j], x_1, x_2)
10         如果 (r_1, r_2) = 为空或权重(u_1) + 权重(u_2) > 权重(r_1) + 权重(r_2)
            then
11             (r_1, r_2) ← (u_1, u_2)
12         如果 (i = |B_1| − 1) 且 (j = 0) 则
13             返回 (r_1, r_2)
14
15         如果 i + 1 < |B_1| 且 B_1[i + 1] 和 B_2[j] 的根节点是诱导的那么
16             i ← i + 1
17         else
18             j ← j − 1
19     直到为假
```

对于各种重树对 $(B_1[i], B_2[j])$，可能出现 $v_1 \notin B_1[i]$ 和/或 $v_2 \notin B_2[j]$；我们还提供节点 $x_\star$，其中 $v_\star$ 连接到 $B_\star$ 中相应的重子树

引理6。 算法*1* 执行 $O(\log\ n/\log\ b)$ 受限 *HIA* 查询，以查找节点 u和 v的最重诱导祖先。

证明。从 $v_\star$ 到根节点的路径经过层索引单调递减的重树，因此序列 B_1 和 B_2 至多包含 $\mathcal{O}$(log n/log b)个元素。每次执行受限 HIA 查询后，我们要么将 i 增加1，要么将 j 减少1，因此执行的受限 HIA 查询总数至多为 $|B_1| + |B_2| = \mathcal{O}$(log n/log b)。

正确性源于诱导性的单调性。对于一对诱导节点，它们的任意一对（弱）祖先也都是诱导的。反过来，如果一对节点不是诱导的，那么它们的任意一对（弱）后代也不是诱导的。

这意味着，如果 $B_1[i]$ 和 $B_2[j]$ 的根节点是诱导的，就无需查询任何满足 $(i' < i) \wedge (j' < j)$ 的 $(B_1[i'], B_2[j'])$ 对，因为这样的查询会返回一对（严格）祖先，而这对祖先是针对 $(B_1[i], B_2[j])$ 的受限 HIA 查询所返回节点对的祖先。我们称一对树 $(B_1[i'], B_2[j'])$ 为被支配，如果存在 $i > i'$ 和 $j > j'$，使得 $B_1[i]$和 $B_2[j]$ 的根节点是诱导的。我们证明，该算法恰好会在第 9 行对满足以下条件的重树执行受限 HIA 查询：(i) 未被支配；(ii) 受限 HIA 查询的结果非空。

该算法维护这样一个不变量：每次调用受限 HIA 查询时，所针对的树对的根节点都是诱导的，并且树对 $(B_1[i], B_2[j])$ 未被支配。对于第一次迭代，这一条件成立；此时考虑树对 $(B_1[0], B_2[|B_2| - 1])$，因为 $B_2[|B_2| - 1]$ 有一个叶节点，并且 $B_1[0]$ 的根节点就是 T_1 的根节点。现在我们证明该不变量在后续迭代中仍然成立。我们首先假设 restricted-hia$(v_1, v_2, B_1[i], B_2[j], x_1, x_2)$ 的结果非空，且该树对未被支配。现在，我们需要区分两种情况。

情况 1：接下来考虑数对 $(B_1[i + 1], B_2[j])$。这意味着第15行的检查确认这些树的根节点是诱导的，因此调用 restricted-hia$(v_1, v_2, B_1[i+$

1], $B_2[j], x_1, x_2$) 的结果不为 null。此外，这对重路径未被支配，因为 ($B_1[i], B_2[j]$) 未被支配。

情况 2：接下来考虑对 ($B_1[i], B_2[j-1]$)。只有在 $i = |B_1| - 1$ 或 $B_1[i+1]$ 和 $B_2[j]$ 的根节点未被诱导时，才会出现这种情况；无论是哪种情况，($B_1[i], B_2[j-1]$) 都不会被支配。此外，由于对 ($B_1[i], B_2[j]$) 的 HIA 查询结果不为 null，($B_1[i], B_2[j-1]$) 的查询结果也不可能为 null。

因此，在这两种情况下，不变量都得以保持。

显然，v_1, v_2 的最重诱导祖先对属于一对满足条件 (i) 和 (ii) 的重树。我们声称会处理所有这类树对。注意，对于固定的 j，存在两个下标 $0 \le i_1 < i_2 \le |B_1|$，使得对 $B_1[i], B_2[j]$ 对于 $0 \le i < i_1$ 被支配、对于 $i_1 \le i < i_2$ 不被支配，并且对应于 $i_2 \le i < |B_1|$ 的空值查询结果。根据不变量，在第 18 行每次减少 j 后，$i \ge i_1$ 都成立，因为 $B_1[i], B_2[j]$ 不被支配。实际上，$i = i_1$，因为对 $B_1[i-1], B_2[j]$ 的处理与前一步相同：我们考虑了对 $B_1[i], B_2[j+1]$，其查询结果不为空。由于接下来的步骤会处理所有 i，直到（但不包括）i_2，因此结论成立。□

3.1 受限 HIA 查询

在本小 节中，我们提出一种能够高效回答受限 HIA 查询的数据结构 .

定理 7. 对于任意两棵分别有 n 个叶节点的树 T_1、T_2，以及一个整数参数 $b \in [n]$，存在一种大小为 $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$ 的数据结构，可在 $\tilde{\mathcal{O}}(nb^2/\log^2 b)$ 时间内计算得出，并能：*(i)* 在常数时间内回答给定的两棵重树的根节点是否为诱导的查询；*(ii)* 在常数时间内回答任意受限 *HIA* 查询 *restricted-hia*($v_1, v_2, H_1, H_2, x_1, x_2$)，加上回答关于 *pre*($v_1$) 的一个前驱查询和关于 *pre*($v_2$) 的一个前驱查询所需的时间；这些前驱查询在为重树对 (H_1, H_2) 存储的 $\mathcal{O}(b^2)$ 个（经预处理的）列表中的两个列表上执行。

我们将证明分为三部分：首先介绍预处理阶段，然后讨论所构建数据结构的性质，最后给出查询过程。

预处理。首先，我们将每个 T_1, T_2 的边划分为重边和轻边，并据此确定重树。对于每个节点，我们记录其所属的重树以及所属的分支。此外，对于每个 $T_\star$，我们构建一个线性规模数据结构，以便在 $\mathcal{O}(1)$ 时间内回答最近公共祖先查询 [17]。对于 $T_\star$ 中的每个节点，我们固定其子节点的顺序，使属于同一重树的子节点排在最后。（与父节点之间通过同类型边（重边或轻边）相连的子节点，其顺序可以任意。）接下来，我们计算 $T_\star$ 的先序遍历；对于每个节点 u，记 pre(u) 为 u 的先序编号，并记 $T_\star$ 中先序编号为 p 的节点为 $T_\star[p]$；我们有 $T_\star[$pre(u)$] = u$。此外，对于每个标签，我们找出 T_1 和 T_2 中具有该标签的叶节点（回想一下，单棵树中的标签两两不同）。

接下来，对于每一对(ℓ_1, ℓ_2) 具有相同标签的叶节点，我们遍历其查询路径上的所有重树对($B_1^{\ell_1}[i_1]$, $B_2^{\ell_2}[i_2]$)，并为 $B_1^{\ell_1}[i_1] \times B_2^{\ell_2}[i_2]$中的每一对分支向数据结构中插入一个点。该过程形式化为算法2。

我们将由算法 2 处理的重树对（B_1, B_2）称为相关对。我们首先运行一次该算法，仅记录所有相关的重树对，而不向任何数据结构中插入点。然后，我们对相关对进行排序、去重，并据此构建一个确定性字典 [20]。这样，我们就可以在常数时间内检查两棵树的根节点是否是诱导的，因为这等价于检查这对树是否相关。对于每一对相关的重树，我们初始化一个数组 $D^{B_1,\ B_2}$ 其索引为分支对（e_1, e_2），其中 e_1 是 B_1 中的一个分支，而 e_2 是 B_2中的一个分支。在数组的每个条目中，我们创建

算法 2： 对具有相同标签的一对叶节点 (ℓ_1, ℓ_2) 进行预处理。

过程添加标签(T_1, T_2, ℓ_1, ℓ_2)
  ($B_1^{\ell_1}$,$B_2^{\ell_2}$) ← 从 ℓ_1 和 ℓ_2 出发的查询路径上的重树序列
  对于 $i_1 \leftarrow 0, 1, \ldots, |B_1^{\ell_1}| - 1$ **执行**
    对于 $i_2 \leftarrow 0, 1, \ldots, |B_2^{\ell_2}| - 1$ **执行**
      对于 $e_1 \leftarrow B_1^{\ell_1}[i_1]$ 的分支**执行**
        对于 $e_2 \leftarrow$分支 $B_2^{\ell_2}[i_2]$ **执行**
          $w_1 \leftarrow$ LCA(ℓ_1, e_1上的最低节点)
          $w_2 \leftarrow$ LCA(ℓ_2, e_2上的最低节点)
          将点 (pre(w_1), pre(w_2)) 插入到 $D^{B_1^{\ell_1}[i_1],\ B_2^{\ell_2}[i_2]}[e_1, e_2]$

（存储指向）相应分支对的数据结构的指针。随后，我们重新运行算法 2，并在需要时借助为相关重树对构建的确定性字典，将点插入这些数据结构中。由于每个分支 $e_\star$ 都属于唯一一棵重树，我们通常省略上标，用 $D[e_1, e_2]$ 代替 $D^{B_1、\ B_2}[e_1, e_2]$。当且仅当两个分支（e_1，e_2）各自所属的重树对相关时，我们称这对分支相关。根据引理 5，每条查询路径都被分解为位于不同层上的 $\mathcal{O}(\log\ n/\log\ b)$ 个部分，并且每棵重树有 $\mathcal{O}(b)$ 个分支。因此，对于每一对标签相同的叶节点（ℓ_1，ℓ_2），我们会向 $\mathcal{O}(b^2 \log^2 n/\log^2 b)$ 个数据结构中插入一个点。

最后，对于每一对相关分支 (e_1, e_2)，我们对结构 $D[e_1, e_2]$ 执行以下后处理：

- 删除所有满足以下条件的点 (x, y)：存在另一个点 (x', y')，使得 $x \le x', y \le y'$ 且 $(x, y) \neq (x', y')$。通过对这些点排序并按从左到右的顺序处理，可以在 $\mathcal{O}(|D[e_1, e_2]|)$ 时间内完成。

- 令 $D_x[e_1, e_2]$ 和 $D_y[e_1, e_2]$ 分别为剩余点的 x 坐标集和 y 坐标集。我们分别为 $D_x[e_1, e_2]$ 和 $D_y[e_1, e_2]$ 构建用于前驱查询问题的数据结构。

- 我们为按 x 坐标排序后留在 $D[e_1, e_2]$ 中的点构建用于区间最大值查询问题的数据结构，其中点 (x, y) 的权重为 weight($T_1[x]$)+weight($T_2[y]$)。

我们将上述阶段称为 $D[e_1, e_2]$ 的后处理。

总结树 $T_\star$ 的整个预处理阶段：对于每个 n 标签，我们向结构 $D[\cdot, \cdot]$ 中添加$\mathcal{O}(b^2 \log^2 n/\log^2 b)$ 个点，总计 $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$ 个点。对所有结构 $D[\cdot, \cdot]$ 进行后处理所需的时间与其大小近乎线性，因此总运行时间为 $\tilde{\mathcal{O}}(nb^2/\log^2 b)$。用于前驱查询问题和区间最大值查询问题的数据结构，其大小与其所处理的元素数量呈线性关系，因此总空间为 $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$。

结构的性质 D[e_1,e_2]。 在本段中，我们将展示结构 $D[e_1, e_2]$ 的一些性质，这些性质有助于高效回答受限 HIA 查询。

性质 8。 对于添加到 $D^{B_1^{\ell_1}[i_1],\ B_2^{\ell_2}[i_2]}[e_1, e_2]$ 的每一对 $(w_1,\ w_2)$，$w_\star$ 要么位于 $e_\star$ 上，要么位于从 $e_\star$ 的最高节点到 $B_\star^{\ell_\star}[i_\star]$ 的根节点的路径上。

证明。回想一下，$w_\star$ 是 $\ell_\star$ 的最近公共祖先，也是 $e_\star$上的最低节点 q。注意，由于 $B_\star^{\ell_\star}[i_\star]$ 位于从 $\ell_\star$出发的查询路径上，$B_\star^{\ell_\star}[i_\star]$ 的根节点 r 是 $\ell_\star$ 和q的共同祖先。因此，$w_\star$ 位于从 r到q 的路径上，由此可直接得出结论。 □

Not并且，在后处理的第一步之后， $D[e_1, e_2]$ 满足以下性质 :

性质 9. 后处理完成后，对于每一对分支(e_1, e_2) ，将 $D[e_1, e_2]$ 中的点按 x 坐标递增排序后，所得点序列也按 y 坐标递减排序。

通俗地说，现在我们可以考虑一个一维问题，其中的点构成一个序列，并且可以通过前驱查询，高效地按 x 坐标和 y 坐标遍历该序列。

接下来，我们将说明如何利用已计算出的数据结构 $D[e_1, e_2]$ 高效地回答受限 HIA 查询。

回答受限 HIA 查询。 现在，我们可以介绍如何针对节点对 (v_1, v_2)，以及查询路径中来自 v_1 和 v_2 的重树 B_1 和 B_2 ，回答受限 HIA 查询。设 (r_1, r_2) $=$ restricted-hia$(v_1, v_2, B_1, B_2, x_1, x_2)$ 是分别位于树 B_1 和 B_2 中、由诱导而成且总权重最大的 v_1 和 v_2 的一对祖先。回顾一下， $x_\star$ 是 $v_\star$ 在 $B_\star$ 中的最低弱祖先，并令 $e_\star$ 为包含 $x_\star$ 的分支。此外，令 ℓ 为诱导出 (r_1, r_2) 的标签，并令叶节点 $\ell_\star$ 共享该标签。

首先，我们证明，在后处理之前，可以利用结构 $D[e_1, e_2]$ 找到 v_1 和 v_2 的具有最大组合权重的诱导祖先对。然后，我们证明，在后处理阶段之后，通过对 pre(x_1) 和 pre(x_2) 执行前驱查询，仍然可以检索到正确答案，而且效率更高。最后，我们证明，可以对 pre(v_1) 和 pre(v_2) 执行前驱查询，而不是对 pre(x_1) 和 pre(x_2) 执行查询。对于单次受限 HIA 查询的正确性或效率而言，最后一步并不重要，但它改善了算法 1 的复杂度。事实上，由于所有前驱查询都针对 pre(v_1) 或 pre(v_2) 中的一个，我们可以使用分数级联。我们将在第 3.2 节中详细解释这一最终组成部分。

回顾一下，在算法 2 中，我们将点 $(\text{pre}(w_1)$,$\text{pre}(w_2))$ 插入 $D[e_1, e_2]$ 中，其中 $w_\star$ =最近公共祖先$(\ell_\star$, 上的最低节点 $e_\star)$。在性质 8 的证明中，我们提到 $w_\star$ 始终属于$B_\star$ ，因为 $B_\star$ 的根是 $\ell_\star$ 和 $e_\star$ 上最低节点的祖先。 和 $x_\star$ 在重树中的相对位置有两种可能:

- ℓ 位于下方 $x_\star$ ，当 $w_\star$ 是 $x_\star$的后代（不一定是真后代）时;
- ℓ 位于上方相连 $x_\star$ ，当 $w_\star$ 是 $x_\star$的真祖先时。

ℓ 相对于 x_1 和 x_2的相对位置有四种情况:

情况 1: ℓ 位于 x_1 和 x_2上方并与之相连,

情况 2: ℓ 位于 x_1 上方并与之相连，而 ℓ 位于 x_2下方,

情况 3: ℓ 位于 x_2 上方并与之相连，而 ℓ 位于 x_1下方,

情况 4: ℓ 位于 x_1 和 x_2的下方。

接下来，我们分别处理这些情况。对于每种情况，我们检索一对 x_1 和 x_2 的诱导祖先，使其在所有由标签 ℓ诱导的祖先对中具有最大的总权重，且该标签相对于 x_1 和 x_2的位置适当。每种情况都会为 x_1 和 x_2的受限最重诱导祖先提供一个候选对。最后，我们返回总权重最大的候选对。此前针对 HIA 问题的解决方案也采用了类似的情况分析，例如 [14]。

引理 10. 答案可以从后处理之前存储在 $D[e_1, e_2]$ 中的信息中检索得到，适用于受限 HIA $(v_1,\ v_2,$ $B_1,\ B_2,\ x_1,\ x_2)$ 。

证明。根据性质 8，对于添加到$D[e_1, e_2]$的任意两点 (pre(w_1), pre(w_2)) 和 (pre(w_1'), pre(w_2'))，我们有 w_1 要么是 w_1' 的弱祖先，要么是其后代；对于 w_2 和 w_2'，情况也类似。因此，节点的先序编号与其深度相对应，我们可以通过比较它们来检查祖先关系：对于路径上的节点 u、u'，u 是 u' 的弱祖先，当且仅当 pre(u) ≤pre(u')。

利用这一性质，我们说明如何将上文列出的四种情况分别归约为：在特定的点矩形子集中寻找一个特定的点。目前，我们暂不考虑查询效率（朴素实现需要线性时间），而是重点说明如何在后处理之前，从 $D[e_1, e_2]$ 中检索到受限 HIA 查询的正确答案。

情况 1：每个附着在节点 w_1 和 w_2 中 x_1 与 x_2 上方的叶节点 ℓ，都会使点对 (w_1, w_2) 成为受限 HIA 查询的候选结果。因此，我们需要在 $D[e_1, e_2]$ 中找到一个点 (x, y)，使得 $x <$ pre(x_1)、$y <$ pre(x_2)，且 weight($T_1[x]$) + weight($T_2[y]$) 最大。于是，$(T_1[x], T_2[y])$ 是 (v_1, v_2) 的受限 HIA 候选对。

情况 2：我们需要找到一个点 $(x, y) \in D[e_1, e_2]$，使得 $x <$ pre(x_1)、$y \geq$ pre(x_2)，并使 weight($T_1[x]$) 最大。于是，$(T_1[x], x_2)$ 是 (v_1, v_2) 的受限 HIA 候选对。

情况 3：此情况与情况 2 对称。我们需要找到一个点 $(x, y) \in D[e_1, e_2]$，使得 $x \geq$ pre(x_1)、$y <$ pre(x_2)，且 weight($T_2[y]$) 最大。然后，$(x_1, T_2[y])$ 是 (v_1, v_2) 的一个受限 HIA 候选对。

情况 4：我们需要检查是否存在一个点 (x, y)，使得 $x \geq$ pre(x_1) 且 $y \geq$ pre(x_2)。如果存在，则点对 (x_1, x_2) 是 (v_1, v_2) 的一个受限 HIA 候选对。

在每种情况下，如果存在，我们都会返回一对 (x_1, x_2) 的诱导祖先，因此它们也是 (v_1, v_2) 的诱导祖先。叶节点 ℓ_1 和 ℓ_2 的标签 ℓ 在 $B_1 \times B_2$ 中插入了点 $D[e_1, e_2]$，该点诱导了 (v_1, v_2) 的最重诱导祖先对 (r_1, r_2)，因为 B_1 和 B_2 分别位于从 ℓ_1 和 ℓ_2 出发的查询路径上。因此，在考虑四种情况之一时，会找到点对 (r_1, r_2)。 □

现在，我们说明，在后处理阶段之后，只需对 $D[e_1, e_2]$ 中的点运行上述算法。

引理 11. 受限 HIA $(v_1, v_2, B_1, B_2, x_1, x_2)$ 的答案可以从后处理之后存储在 $D[e_1, e_2]$ 中的信息中获取。

证明。正如引理 10 的证明中所讨论的，对于添加到 $D[e_1, e_2]$、w_i 和 w_i' 中的任意两点 (pre(w_1), pre(w_2)) 和 (pre(w_1'),pre(w_2'))，它们位于同一条根到叶路径上，因此它们的 pre 序号与它们在树中的深度顺序相同。回想一下，这些树的权重是单调的，也就是说，沿每条根到叶路径的权重递增，因此我们有 pre(w_i) ≤ pre(w_i') 蕴含 weight(w_i) ≤ weight(w_i')。

令 $w =$ (pre(w_1), pre(w_2)) 为引理 10 中所述算法找到的答案所对应的点。注意，返回的诱导祖先对不一定是 (w_1, w_2)，例如，它可以是 (w_1, x_2)。假设 w 在后处理阶段被移除。如果是这样，那是因为存在一个点 $w' =$ (pre(w_1'), pre(w_2'))，满足 pre($w_\star$) ≤ pre($w_\star'$) 且 $w \neq w'$。如果 w' 与 w在不同的情形下进行处理，那么与 w' 对应的祖先对的总权重大于返回的祖先对，从而产生矛盾。如果 w' 与 w在相同的情形下进行处理，那么与 w' 对应的祖先对所构成的祖先对，其总权重不小于返回的祖先对。因此，证明引理 10 中所述的归约对于后处理后的集合 $D[e_1, e_2]$ 仍然成立。 □

接下来，我们将说明如何利用点 10 中的点已在 $D[e_1, e_2]$ 完成后处理这一事实，高效地实现引理中的四种情况。

引理 12。受限 HIA $(v_1,\ v_2,\ B_1,\ B_2,\ x_1,\ x_2)$ 的答案可以通过两次前驱查询，从 $D[e_1, e_2]$ 中后处理后存储的信息中获取：一次查询 $\mathrm{pre}(x_1)$，另一次查询 $\mathrm{pre}(x_2)$。

证明。通过计算 $D_x[e_1, e_2]$ 中 $\mathsf{pre}(x_1)$ 的前驱，我们得到 $D_x[e_1, e_2]$ 的区间 $\mathcal{I}^{x<\mathrm{pre}(x_1)}{}_x$ and $\mathcal{I}^{x\geq\mathrm{pre}(x_1)}{}_x$。类似地，通过计算 $D_y[e_1, e_2]$ 中 $\mathsf{pre}(x_2)$ 的前驱，可得到 $D_y[e_1, e_2]$ 的区间 $\mathcal{I}^{y<\mathrm{pre}(x_2)}{}_y$ and $\mathcal{I}^{y\geq\mathrm{pre}(x_2)}{}_y$。注意，根据性质 9，$D_y[e_1, e_2]$ 中的点（分别来自 $\mathcal{I}^{y\geq\mathrm{pre}(x_2)}{}_y$ 和 $\mathcal{I}^{y<\mathrm{pre}(x_2)}{}_y$）对应于 $D_x[e_1, e_2]$ 区间中的点，我们将其记为 $\mathcal{I}^{y\geq\mathrm{pre}(x_2)}{}_x$ ($\mathcal{I}^{y<\mathrm{pre}(x_2)}{}_x$）。因此，我们可以将引理 10 各种情况中关于点的每个条件转换为 $D_x[e_1, e_2]$ 上两个区间的交集 $\mathcal{I}_x^{\mathrm{Case}\ i}$。这使得四种情况均可化简为：

情况 1：使用区间最值查询，在 $\mathcal{I}_x^{\mathrm{Case}\ 1}$ 中查找满足 $\mathsf{weight}(T_1[x])$ +$\mathsf{weight}(T_2[y])$ 的最大权重点。

情况 2：根据权重相对于 x 坐标的单调性，$\mathcal{I}_x^{\mathrm{Case}\ 2}$ 中最大权重 $\mathsf{weight}(T_1[x])$ 对应的点是 $\mathcal{I}_x^{\mathrm{Case}\ 2}$ 的最右元素。

情况 3：根据权重相对于 y-坐标的单调性以及性质 9，最大权重点 $\mathtt{weight}(T_2[y])$ 在 $\mathcal{I}_x^{\mathrm{Case}\ 3}$中是 $\mathcal{I}_x^{\mathrm{Case}\ 3}$ 的最左元素。

情况 4：只需检查 $\mathcal{I}_x^{\mathrm{Case}\ 4}$ 是否非空。 □

引理 13。受限 HIA $(v_1,\ v_2,\ B_1,\ B_2,\ x_1,\ x_2)$ 的答案可以在后处理后从存储于 $D[e_1, e_2]$ 中的信息中检索得到，只需进行两次前驱查询：一次针对$\mathrm{pre}(v_1)$，另一次针对$\mathrm{pre}(v_2)$。

证明。回顾引理12 中介绍的方法：我们在 $D_x[e_1, e_2]$ 中计算 $\mathsf{pre}(x_1)$ 的前驱，以便将 $D_x[e_1, e_2]$ 划分为 $\mathcal{I}^{x<\mathrm{pre}(x_1)}{}_x$ 和 $\mathcal{I}^{x\geq\mathrm{pre}(x_1)}{}_x$，并且 $D_x[e_1, e_2]$ 中的所有元素都具有 $\mathsf{pre}(w_1)$ 的形式，其中节点 w_1 位于从 e_1 上的最低节点到 B_1 的根节点的路径上。

我们只考虑 v_1 和 x_1，因为对 v_2 和 x_2 的分析是对称的。我们可以专注于 $v_1 \neq x_1$的情况，因为另一种情况显而易见。显然，$\mathsf{pre}(v_1)$ ≥$\mathsf{pre}(x_1)$，因为 v_1 是 x_1的后代。回顾一下，在预处理阶段，我们对每个节点的子节点重新排序，使得通过轻边连接的子节点排在通过重边连接的子节点之前。因此，如果 x_1' 是 x_1 在 e_1 上的一个子节点，我们有 $\mathsf{pre}(x_1)$ ≤$\mathsf{pre}(v_1)$ < $\mathsf{pre}(x_1')$，因为 v_1 是 x_1 的一个轻子节点的后代（根据 x_1的定义）。因此，$\mathsf{pre}(v_1)$的前驱与 $D_x[e_1, e_2]$中 $\mathsf{pre}(x_1)$的前驱相同。 □

定理 7的证明到此结束。

~最后，通过将 b 的值设为 $\lfloor \log n \rfloor$，我们得到一个 $\mathcal{O}(n)$-空间数据结构，它能够在常数时间内回答受限 HIA 查询，外加回答两个前驱查询所需的时间：一个针对 $\mathrm{pre}(v_1)$，另一个针对 $\mathrm{pre}(v_2)$。算法 1 执行 $\mathcal{O}(\log\ n/\log\ b)$次受限 HIA 查询来回答一个 HIA 查询，因此总时间为 $\mathcal{O}(\log n)$。然而，由于所有前驱查询都是在彼此相关的不同列表中查询两个值之一，我们可以利用分数级联。

3.2 分数级联

对于上一小节所述的大多数情况，我们的数据结构都在执行前驱查询。这也是采用我们的方法时，HIA 查询的时间复杂度尚未达到 $\mathcal{O}(\log\ n/\log\log\ n)$ 的唯一原因。我们将利用这样一个事实：所有这些查询查找的目标值都相同（为针对 T_1 构建的数据结构查找 $\mathsf{pre}(v_1)$，为针对 T_2 构建的数据结构查找 $\mathsf{pre}(v_2)$），但涉及不同的分支对，因此我们可以使用分数级联。

我们可以考虑从 $T_1 \times T_2$ 构建两个目录图，其中节点表示分支对（e_1, e_2）：一个目录图存储 $D_x[e_1, e_2]$ 的内容，另一个存储 $D_y[e_1, e_2]$的内容。随后，算法 1 的执行可以看作是在此类图中遍历一条路径：对于算法执行受限 HIA 查询所针对的每对重树（$B_1[i], B_2[j]$），我们查询代表分支对（e_1, e_2）的节点目录；这对分支分别包含（v_1, v_2）在 $B_1[i]$ 和 $B_2[j]$中的最低弱祖先。直接采用这种方法的问题在于，无法保证每个目录图中所有顶点的度都是多对数级的：我们可能需要从对应于两个分支（e_1, e_2）的节点，移动到对应于两个分支（e'_1, e'_2）的任意节点；其中，包含 e'_1 的重树连接到 e_1，而这样的分支甚至可能有 $\Omega(n)$ 个 e'_1。

我们需要构造目录图，使每个 HIA 查询所考虑的路径长度为 $\mathcal{O}(\log n/\log\log n)$，同时每个节点的度为 $\mathcal{O}(\text{polylog } n)$，以便能够应用 Shi 和 J´aJ´a [21]的结果。这样可以确保 $D_x[\cdot,\cdot]$ 和 $D_y[\cdot,\cdot]$ 中的所有前驱查询都能在常数时间内完成，首次查询除外；首次查询使用 $\mathcal{O}(\log \log n)$ 时间，并借助 x-快速字典。下面我们将介绍如何构造合适的目录图。由于 $D_x[\cdot,\cdot]$ 和 $D_y[\cdot,\cdot]$ 的图形结构相同，仅目录的内容不同，因此我们只介绍其中一个的构造方法。

我们分别对 T_1 和 T_2 进行预处理，首先计算树 $B(T_\star)$，然后构造目录图 $C(T_\star)$。基于此，我们构造目录图 C，它可以看作是$C(T_1)$ 和 $C(T_2)$的笛卡尔积。更准确地说，C 中的每个节点都是一对 (v, w)，其中 $v \in C(T_1)$，且 $w \in C(T_2)$。对于 $C(T_1)$中节点 v_1 和 v_2 之间的一条边，在最终的目录图中，我们会为每个 $w \in C(T_2)$，在节点 (v_1, w) 和 (v_2, w)之间创建边。类似地，对于 $C(T_2)$中节点w_1 和 w_2 之间的一条边，在最终的目录图中，我们会为每个 $v \in C(T_1)$，在节点 (v, w_1) 和 (v, w_2)之间创建边。这样，如果 $C(T_1)$ 和 $C(T_2)$的度都是多对数级的，那么 C的度也会是多对数级的。

现在说明如何由 $T_\star$ 构建树 $B(T_\star)$。$B(T_\star)$ 包含表示重树的节点（称为重树节点）以及表示分支的节点（称为分支节点）：

- 对于每棵重树 H，我们将表示 H 中各分支的所有分支节点连接为表示 H 的重树节点的子节点，
- 对于每棵重树 H（包含 $T_\star$ 根节点的树除外），我们将表示 H 的重树节点连接为表示包含 H 根节点之父节点的分支的分支节点的子节点。

命题 14. *$B(T_\star)$ 的深度为 $\mathcal{O}(\log n/\log\log n)$，且每个重树节点有 $\mathcal{O}(\log n)$ 个子节点。*

回忆一下，每棵重树有 $\mathcal{O}(b) = \mathcal{O}(\log n)$ 个分支，因此每个重树节点有 $\mathcal{O}(\log n)$ 个子节点。$B(T_\star)$ 中任意一条根到叶路径都会交替经过重树节点和分支节点。对于 $T_\star$ 中任意一条从根到 v 的路径 p，在 $B(T_\star)$ 中都有一条对应路径 p'，它会经过与和 p 相交的重树相对应的重树节点，以及满足以下条件的分支节点：算法 1（针对包含 v 的一对节点调用时）可能会对 $D_x[\cdot,\cdot]$ 和 $D_y[\cdot,\cdot]$ 执行前驱查询。对于任意 p，p' 的长度为 $\mathcal{O}(\log n/\log\log n)$，且可以通过沿着从包含 v 的重树到 $B(T_\star)$ 根节点的路径，在 $\mathcal{O}(|p'|)$ 时间内找到。

现在我们描述如何从 $C(T_\star)$ 构造 $B(T_\star)$目录图。该构造是递归的，源自引理 15的证明。其思路是用适当的构件替换子节点过多的分支节点的子节点结构，大致保留树的结构，不会渐近地增加树的深度，并将每个节点的度降至 $\mathcal{O}(\text{polylog } n)$。根据命题 14，我们无需改变重树节点的子节点结构。

引理 15. *对于任意具有 $\mathcal{O}(n)$ 个节点、深度为 $\mathcal{O}(\log n/\log\log n)$ 的树 $B(T_\star)$，存在一棵树$C(T_\star)$，满足以下所有条件：*

- $C(T_\star)$ 有 $\mathcal{O}(n)$ 个节点，
- $C(T_\star)$ 中的所有节点度为 $\mathcal{O}(\log n)$，
- $C(T_\star)$ 的深度为 $\mathcal{O}(\log n/\log\log n)$，
- 对于 $B(T_\star)$中的每条简单路径 p，都存在一条长度为 $\mathcal{O}(\log n/\log\log n)$ 的简单路径 p'，位于$C(T_\star)$中，可在 $\mathcal{O}(|p'|)$ 时间内计算得到，且 p 是 p'的子序列。

证明。考虑 $B(T_\star)$ 中的一个（分支）节点 e，其子节点从左到右依次为重树节点 h_1、h_2、.....、h_k，其中 $k > \log n$。我们将这个包含 $k+1$ 个节点的子图替换为一个构件图，其根节点为 e，节点集合是 $\{e\} \cup \{h_i$ 的超集：$i \in [k]\}$。

对于每个节点 u，令 $s(u)$ 表示所考虑的树中 u 的子树内节点的数量。令 $s_0 = 0$，并且对于 $i \geq 1$，令 s_i 为前缀和 $s(h_1) + s(h_2) + \ldots + s(h_i)$。若存在一个整数 ℓ，使得 $s_{i-1} < \ell \cdot s(e)/\log n$ 且 $s_i \geq \ell \cdot s(e)/\log n$，则标记 h_i。我们将每个由连续未标记节点组成的集合称为一个区间。由于 $\ell \leq \log n$，因此有 $\mathcal{O}(\log n)$ 个标记节点和$\mathcal{O}(\log n)$ 个区间。

我们按如下方式为 e 构造一个辅助结构（并递归地为构造过程中创建的其他一些节点构造，后文将对此进行说明）：

- 我们将所有已标记的节点或其区间中仅含一个元素的节点，都作为 e 的子节点。
- 对于包含多个节点的每个区间，我们创建一个新节点 i_j，称为区间节点，将其作为 e 的子节点，并将该区间内的所有节点作为 i_j 的子节点。

对于任何新创建且度大于 i_1、i_2、.....、$i_m$log n的区间节点，我们递归地应用相同的构造。图 3 展示了一个示例。

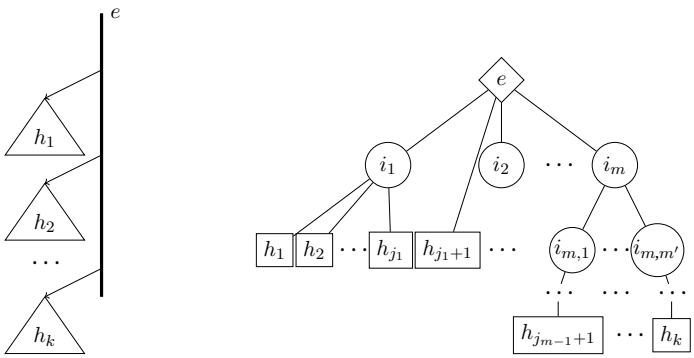

图3：左侧是 e中的一个分支，$T_\star$有指向 h_1、h_2、.....、h_k的出边，位于 $B(T_\star)$中。右侧是为 e构造的目录图构件，它属于 $C(T_\star)$的一部分。圆圈表示区间节点，矩形表示重树节点。部分区间会递归地替换为该构件，以降低其度。重树节点的度为 $\mathcal{O}(\log n)$。在$C(T_\star)$中，e的父节点是其所属的重树，而每个 h_i 的子节点都是 h_i中的分支。

从 t 通过构造可知，$C(T_\star)$中每个节点的度为 $\mathcal{O}(\log n)$，并且其大小为 $C(T_\star)$ 是 $\mathcal{O}(n)$，因为所有新节点的出度至少为 2。

设 u 是 $T_\star$ 中的一个节点。现在，我们通过考察 $C(T_\star)$ 中 e_u 上方的边，证明表示包含 u 的分支的节点 $e_u \in C(T_\star)$ 的深度为 $\mathcal{O}(\log n/\log\log n)$。这些边分为两种类型：

- 至少与一个非区间节点相连的边。根据$B(T_\star)$的深度，我们有 $\mathcal{O}(\log n/\log\log n)$ 条这样的边。

- 区间节点之间的边。对于每条这样的边 $(v,\ z)$，我们有 $s(v) \geq s(z) \cdot \log n$。因此，类似于引理 5的证明，从 $\mathcal{O}(\log\ n/\log\log\ n)$ 条这样的边位于从 $C(T_\star)$ 的根节点到 e_u的路径上。

至此， $C(T_\star)$深度上界的证明完成。

$B(T_\star)$中的每条简单路径 p 都自然对应于 $C(T_\star)$中的一条简单路径 p'。特别地，对于 $B(T_\star)$中的每条边（v, w），都可以显式存储一条路径 $C(T_\star)$，使得（v，w）与之对应。将 p 上各条边对应的所有此类路径连接起来，即可得到 $\mathcal{O}(|p'|)$中的一条简单路径 p'。由于 $C(T_\star)$的深度为 $\mathcal{O}(\log\ n/\log\log\ n)$，因此 $|p'|$ 的上界得证。

□

根据引理 15中构造的 $C(T_1)$和 $C(T_2)$，我们按照前文所述创建目录图 C。只有表示分支对的节点包含非空的原始目录。填充原始目录后，我们执行分数级联的预处理，并按照[9, 21]中的描述，为 C 中的所有节点创建适当的增强目录。算法 1中执行受限 HIA 查询所产生的$\mathcal{O}$(log n/log log n)个前驱查询，自然可以归约为对 x-fast 字典树的常数次查询，以及在 C中遍历长度为 $\mathcal{O}$(log n/log log n)的路径。这一过程总共需要 $\mathcal{O}$(log $n/\log\log\ n$)时间，至此完成对我们数据结构的描述以及定理 2的证明。

参考文献

[1] Paniz Abedin、Sahar Hooshmand、Arnab Ganguly 和 Sharma V. Thankachan。最重诱导祖先问题：更优的数据结构与应用。Algorithmica，84(7):2088–2105，2022。doi:10.1007/s00453-022-00955-7。[2]S. Alstrup、T. Husfeldt 和 T. Rauhe。标记祖先问题。载于《第39届年度计算机科学基础研讨会论文集》，第534–543页，1998。doi:10.1109/SFCS.1998.743504。[3] Stephen Alstrup、Jens P. Secher 和 Maz Spork。树中的最优在线递减连通性。Inf. Process. Lett.，64(4):161–164，1997。doi:10.1016/S0020-0190(97)00170-1。[4] Amihood Amir、Panagiotis Charalampopoulos、Costas S. Iliopoulos、Solon P. Pissis 和 Jakub Radoszewski。一次编辑操作后的最长公共因子。载于《字符串处理与信息检索：第24届国际研讨会 SPIRE 2017 论文集》，第14–26页，2017。doi:10.1007/978-3-319-67428-5_2。[5] Amihood Amir、Panagiotis Charalampopoulos、Solon P. Pissis 和 Jakub Radoszewski。动态与内部最长公共子串。Algorithmica，82(12):3707–3743，2020。doi:10.1007/s00453-020-00744-0。[6]Philip Bille、Inge Li Gørtz、Hjalte Wedel Vildhøj 和 Søren Vind。带通配符模式的字符串索引。Theory Comput. Syst.，55(1):41–60，2014。doi:10.1007/s00224-013-9498-4。.[7] Gerth Stølting Brodal、Rolf Fagerberg、Christian N. S. Pedersen 和 Anna Ostlin。通过实验构造进化树的复杂度。载于《自动机、语言与编程：第28届国际研讨会 ICALP 2001》，第140–151页，2001。doi:10.1007/3-540-48224-5_12。[8]Panagiotis Charalampopoulos、Paweł Gawrychowski 和 Karol Pokorski。多对数时间内的动态最长公共子串。载于《第47届自动机、语言与编程国际研讨会（ICALP 2020）》，第27:1–27:19页，2020。doi:10.4230/LIPIcs.ICALP.2020.27。

[9] Bernard Chazelle 和 Leonidas Guibas。分数级联：I. 一种数据结构技术。*Algorithmica*，1:133–162，1986年1月。doi:10.1007/BF01840440。[10] Richard Cole 和 Ramesh Hariharan。树上的动态最近公共祖先查询。*SIAM* 计算杂志，34(4):894–923，2005。doi:10.1137/S0097539700370539。[11] Erik D. Demaine、Gad M. Landau 和 Oren Weimann。关于笛卡尔树和区间最小值查询。*Algorithmica*，68(3):610–625，2014。doi:10.1007/s00453-012-9683-x。[12] Johannes Fischer 和 Pawel Gawrychowski。使用 wexponential 搜索树进行依赖字母表的字符串搜索。载于组合模式匹配 —— 第 *26* 届年度研讨会，*CPM 2015*，第160–171页，2015。doi:10.1007/978-3-319-19929-0_14。[13] Johannes Fischer 和 Volker Heun。静态数组区间最小值查询的空间高效预处理方案。*SIAM* 计算杂志，40(2):465–492，2011。doi:10.1137/090779759。

[14]Travis Gagie、Paweł Gawrychowski 和 Yakov Nekrich。最重诱导祖先与最长公共子串。载于第 *25* 届加拿大计算几何会议论文集，*CCCG 2013*，2013 年。网址：http://cccg.ca/proceedings/2013/papers/paper_29.pdf。

[15]Davide Della Giustina、Nicola Prezza 和 Rossano Venturini。《一种用于质心分解的新型线性时间算法》。载于《字符串处理与信息检索——第 26 届国际研讨会 SPIRE 2019》，第 274–282 页，2019 年。doi:10.1007/978-3-030-32686-9_20.[16] Dan Gusfield。《字符串、树和序列算法：计算机科学与计算生物学》。剑桥大学出版社，1997 年。doi:10.1017/CBO9780511574931.[17] Dov Harel 和 Robert Endre Tarjan。《寻找最近公共祖先的快速算法》。SIAM J. Comput.，13(2):338–355，1984 年。[18] Donald R. Morrison。《Patricia——用于检索字母数字编码信息的实用算法》。J. ACM，15(4):514–534，1968 年 10 月。doi:10.1145/321479.321481.[19] Mihai P˘atra¸scu。《统一单元探测下界的格局》。SIAM J. Comput.，40(3):827–847，2011 年。doi:10.1137/09075336X.[20] Milan Ru˘zić。《以接近排序的时间构建高效字典》。载于《自动机、语言与编程》第 35 届国际研讨会 ICALP 2008，第 84–95 页，2008 年。doi:10.1007/978-3-540-70575-8_8.[21] Qingmin Shi 和 Joseph J´aJ´a。《利用 q 堆的新型变换技术及其在计算几何中的应用》。SIAM J. Comput.，34:1474–1492，2005 年 1 月。doi:10.1137/S0097539703435728.[22] Peter Weiner。《线性模式匹配算法》。载于第 14 届 FOCS，第 1–11 页，1973 年。doi:10.1109/SWAT.1973.13.[23]Dan E. Willard。《在 $\Theta(n)$ 空间内可以实现对数对数级最坏情况区间查询》。信息处理快报，17(2):81–84，1983 年。doi:doi.org/10.1016/0020-0190(83)90075-3。

$\mathcal{O}$（对数 n）查询时间数据结构的描述[14]

正如引言中所提到的，Gagie 等人 [14] 在其论文第 2.1 小节最后一段中概述了一种大小为 $\mathcal{O}(n\log^2 n)$ 的数据结构，可在 $\mathcal{O}(\log n)$ 时间内回答 HIA 查询。他们为 T_1 和 T_2 的每一对重树构建一个数据结构，并将针对节点 u 和 v 的 HIA 查询转化为在每个由 $\mathcal{O}(\log n)$ 对重树组成的数据结构中进行前驱查询。要改进那个用于回答查询、经过完整描述的 $\mathcal{O}(\log\ n\log\log\ n)$ 时间过程，采用更高效的方法，其思路与我们类似，并涉及分数级联。他们需要构建一个目录图，其中的节点是重路径对，降低其度，并确保前驱查询的目标相同（通过查询 u 和 v的先序编号）。这与我们在第 3.2 节针对另一种树分解所描述的做法类似。