Optimal Near-Linear Space Heaviest Induced Ancestors

Panagiotis Charalampopoulos
Birkbeck, University of London, UK

Bartłomiej Dudek
Institute of Computer Science, University of Wrocław, Poland

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Karol Pokorski
Institute of Computer Science, University of Wrocław, Poland

Abstract

We revisit the Heaviest Induced Ancestors (HIA) problem that was introduced by Gagie, Gawrychowski, and Nekrich [CCCG 2013] and has a number of applications in string algorithms. Let T_1 and T_2 be two rooted trees whose nodes have weights that are increasing in all root-to-leaf paths, and labels on the leaves, such that no two leaves of a tree have the same label. A pair of nodes $(u, v) \in T_1 \times T_2$ is *induced* if and only if there is a label shared by leaf-descendants of u and v. In an HIA query, given nodes $x \in T_1$ and $y \in T_2$, the goal is to find an induced pair of nodes (u, v) of the maximum total weight such that u is an ancestor of x and v is an ancestor of y.

Let n be the upper bound on the sizes of the two trees. It is known that no data structure of size $\tilde{\mathcal{O}}(n)$ can answer HIA queries in $o(\log n/\log\log n)$ time [Charalampopoulos, Gawrychowski, Pokorski; ICALP 2020].[1] This (unconditional) lower bound is a $\operatorname{polyloglog} n$ factor away from the query time of the fastest $\tilde{\mathcal{O}}(n)$-size data structure known to date for the HIA problem [Abedin, Hooshmand, Ganguly, Thankachan; Algorithmica 2022]. In this work, we resolve the query-time complexity of the HIA problem for the near-linear space regime by presenting a data structure that can be built in $\tilde{\mathcal{O}}(n)$ time and answers HIA queries in $\mathcal{O}(\log n/\log\log n)$ time. As a direct corollary, we obtain an $\tilde{\mathcal{O}}(n)$-size data structure that maintains the LCS of a static string and a dynamic string, both of length at most n, in time optimal for this space regime.

The main ingredients of our approach are fractional cascading and the utilization of an $\mathcal{O}(\log n/\log\log n)$-depth tree decomposition. The latter allows us to break through the $\Omega(\log n)$ barrier faced by previous works, due to the depth of the considered heavy-path decompositions.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases data structures, string algorithms, fractional cascading

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.8

1 Introduction

The solutions to algorithmic problems on texts frequently involve the construction of text indexes that can be built efficiently and offer a broad functionality, without significantly increasing space usage. A prime example of such an index is the suffix tree, which is ubiquitous in stringology. The work of Weiner [22] that introduced it, showed that it can be used to efficiently solve a number of fundamental open problems such as the computation of occurrences of patterns (given in an online manner) in a text or the computation of the longest common substring of two strings. However, it is usually the case that a suffix tree needs to first be augmented with other data structures before it can efficiently answer

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation hides factors polylogarithmic in n.

more sophisticated queries, e.g., returning the longest common prefix of two substrings or the longest palindrome centered at some position; an augmentation with a lowest common ancestors data structure suffices for these examples [16, 17].

Crucially, a text index, such as the suffix tree, is built once and can then be queried an arbitrary number of times. This is increasingly relevant: in many real-world scenarios, large pieces of information are stored on servers and are constantly queried by a large number of remote clients. From this perspective, it makes sense to devote some time to preprocess the data stored on the server in order to be able to provide quick responses to remote users later.

The *Heaviest Induced Ancestors* problem, which was introduced by Gagie et al. [14] and is defined next, has been proved to be useful in solving several variants of the problem of computing a longest common substring of two strings [1, 4, 5, 8, 14].

We say that a tree is weighted if there is a weight associated with each node u of the tree, such that weights along root-to-leaf paths are increasing, i.e., for any node u other than the root the weight of u is larger than the weight of u's parent. Further, we say that a tree is labelled if each of its leaves is given a distinct label from $[n]$, where n is the number of leaves. As an example of a rooted, weighted, and labelled tree, consider the suffix tree of a string $S\$$, where \$ does not occur in S, with the label of each leaf being the starting position of the corresponding suffix and the weight of each node being the length of the string it represents.

▶ **Definition 1.** *For two rooted and weighted trees T_1 and T_2 on n leaves, we say that two nodes $u \in T_1$ and $v \in T_2$, are* induced *(by label ℓ) if and only if there are leaves x and y labelled with ℓ, such that x and y are weak descendants of u and v, respectively.*

HEAVIEST INDUCED ANCESTORS (HIA)
Input: Two rooted, weighted, and labelled trees T_1 and T_2 on n leaves.
Query: Given a pair of nodes $u \in T_1$ and $v \in T_2$, return a pair of induced nodes (u', v') with the largest total weight, such that u' is an ancestor of u and v' is an ancestor of v.

Previous results and our contribution. Table 1 shows the state-of-the-art size vs. query-time tradeoffs for the HIA problem prior to our work and our result. Gagie et al. [14] presented several tradeoffs which have been since improved. We stress that the $\mathcal{O}(n \log^2 n)$-size data structure with query-time $\mathcal{O}(\log n)$ included in Table 1 was only sketched in [14]. We briefly discuss this sketch in Appendix A, as some of the ideas involved are similar to the ones we use. The remaining $\tilde{\mathcal{O}}(n)$-size known data structures found in Table 1 are due to Abedin et al. [1]. Charalampopoulos et al. [8] showed an unconditional lower bound for near-linear size data structures and a data structure with query-time $\mathcal{O}(1)$ and size $\mathcal{O}(n^{1+\epsilon})$ for any constant $\epsilon > 0$. We now formally state our main result, which matches the lower bound of [8].

▶ **Theorem 2.** *For any $\epsilon > 0$, there is an $\mathcal{O}(n \log^{2+2\epsilon} n)$-size data structure for the HIA problem that can be constructed in $\tilde{\mathcal{O}}(n)$ time and answers queries in $\mathcal{O}(\log n / \log \log n)$ time.*

Applications of HIA. Before discussing some concrete applications of the HIA problem in string algorithms and the consequences of our results for them, we give a high-level description of how the HIA problem comes up in variants of computing an LCS.

Consider a string S and a chosen subset A of its positions, that we call *anchors.* Further, consider the following two tries: a trie $\mathcal{T}^{\leftarrow}$ for the strings in $\{S[1 \mathinner{.\,.} k-1]^R : k \in A\}$, where U^R denotes the reversal of U, and a trie $\mathcal{T}^{\rightarrow}$ for the strings in $\{S[k \mathinner{.\,.} |S|] : k \in A\}$. In other words, for every anchor $k \in A$, we have a path in the first trie for every prefix of $S[1 \mathinner{.\,.} k-1]^R$ and a path in the second trie for every prefix of $S[k \mathinner{.\,.} |S|]$. We label each leaf of the two tries

Table 1 Size vs. query-time tradeoffs for the HIA problem; the size is measured in machine words.

Size	Query time	Paper
$\tilde{\mathcal{O}}(n)$	$\Omega(\log n/\log\log n)$	[8]
$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n/\log\log n)$	[1]
$\mathcal{O}(n\log n)$	$\mathcal{O}(\log n\log\log n)$	[1]
$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(\log n)$	sketched in [14], see Appendix A
$\mathcal{O}(n\log^{2+\epsilon} n)$	$\mathcal{O}(\log n/\log\log n)$	*this work*
$\mathcal{O}(n^{1+\epsilon})$	$\mathcal{O}(1)$	[8]

with the anchor it corresponds to. Now, observe that a substring $S[i\mathinner{.\,.}j]$ that crosses an anchor k, i.e., $i < k \leq j$, corresponds to an induced pair of nodes in the tries. Indeed, there is a path representing $S[i\mathinner{.\,.}k-1]^R$ in the first trie and a path representing $S[k\mathinner{.\,.}j]$ in the second trie. An illustration of this idea is provided in Figure 1. The set of anchors and the HIA queries performed in an application of this technique depends on the specific problem it is used for. For some of the usages, one may consider using a compressed form of tries [18].

As a first application, consider the maintenance of an LCS of a static string T and a dynamic string S. By plugging our HIA data structure into the approach of [8], we obtain the following result, improving the state-of-the-art by polyloglog n factors, and matching the lower bound for the update-time when nearly-linear space is available [8, Theorem 1].

▶ **Corollary 3.** *We can maintain an LCS of a dynamic string S and a static string T, each of length at most n, in $\mathcal{O}(\log n/\log\log n)$ time per substitution operation using $\tilde{\mathcal{O}}(n)$ space, after an $\tilde{\mathcal{O}}(n)$-time preprocessing.*

Further, the authors of [14] (implicitly) reduced to the HIA problem, the problem of preprocessing a text given in LZ77 compressed form so that one can compute its LCS with uncompressed patterns given online. Our HIA data structure yields the following result.

▶ **Corollary 4.** *Let S be a string of length N whose LZ77 parse consists of n phrases. We can store S in $\mathcal{O}(n\log N + n\,\mathrm{polylog}\,n)$ space such that, given a pattern P of length m, we can compute the LCS of S and P in $\mathcal{O}(m\log n/\log\log n)$ time. For each pattern P, the returned result may be (consistently) incorrect with probability inverse polynomial in n.*[2]

[2] Randomization is only used in the construction; all queries for the same pattern give identical results.

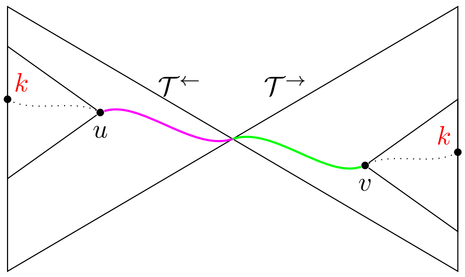

Figure 1 An illustration of the anchoring technique for LCS computation, with the constructed tries $\mathcal{T}^{\leftarrow}$ and $\mathcal{T}^{\rightarrow}$ drawn so that their roots are attached (in the middle). Any substring anchored at k, can be obtained by reading in a left-to-right manner the edge-labels from some node $u \in \mathcal{T}^{\leftarrow}$ to some node $v \in \mathcal{T}^{\rightarrow}$, that both have a leaf-descendant labelled with k.

Other applications of the HIA problem in string algorithms can be found in [1].

Tree Decompositions. One of the obvious divide-and-conquer techniques for efficiently solving algorithmic problems on trees is that of decomposing the tree(s) into smaller pieces and treating each of them separately. The most important attributes of a tree decomposition are usually its depth, i.e., the maximum number of pieces that one path can intersect, and the structure of each individual piece (e.g., pieces being paths may offer an advantage). We next describe some tree decompositions for a tree T with n nodes. For a node v, denote by $s(v)$ the number of nodes in v's subtree.

Arguably, the most well-known tree decomposition is the *heavy-path decomposition* [17]. Abstractly, this decomposition is a partition of the edges into *light* and *heavy*, such that:

- all connected components after deleting the light edges are paths, called *heavy paths*;
- each root-to-leaf path consists of $\mathcal{O}(\log n)$ prefixes of heavy paths and $\mathcal{O}(\log n)$ light edges, i.e., the depth of the decomposition is $\mathcal{O}(\log n)$.

A heavy-path decomposition can be realized in several ways; two of which are as follows:

- HP1: Each non-leaf node u of the tree chooses a child v with maximum $s(v)$ and the edge from u to v is designated as heavy. The remaining edges outgoing from u are light.
- HP2: An edge (u, v) is designated as heavy if and only if $\lfloor \log s(u) \rfloor = \lfloor \log s(v) \rfloor$.[3]

Intuitively, using a heavy-path decomposition, one may often lift an algorithm that only works for paths and/or balanced trees to work for arbitrary trees – usually with some overhead.

All previous works on the HIA problem used heavy-path decompositions, which, as discussed, are of depth $\Omega(\log n)$. This adversely affects their query times as one may have to traverse the decomposition along a root-to-leaf path at query time. Thus, in order to achieve sublogarithmic query time, we considered tree decompositions of smaller depths. There are a couple of generalizations of heavy-path decompositions that have the sought depth, i.e., $\mathcal{O}(\log n/\log\log n)$. We next discuss two such decompositions that are also based on partitioning the edges into light and heavy. The caveat is that, for each of them, the connected components after the removal of the light edges are trees, which we call *heavy trees*, instead of paths and hence some extra work is required.[4]

The heavy α-tree decomposition, introduced by Bille et al. [6], is of depth $\mathcal{O}(\log_\alpha n)$ and is defined analogously to HP1: each non-leaf node u chooses its (at most) α heaviest (with respect to subtree-sizes) children; the edge from u to each of these children is designated as heavy, while all remaining edges outgoing from u are designated as light. By setting $\alpha = \lfloor \log n \rfloor$ one gets the sought depth.

An alternative is the so-called ART decomposition due to Alstrup et al. [2], which, for an input integer parameter b, has depth $\mathcal{O}(\log_b n)$. For ease of presentation, we consider b to be equal to $\lfloor \log n \rfloor$ so that the depth of the decomposition is $\mathcal{O}(\log n/\log\log n)$. A partition of the edges yields such an ART decomposition if and only if each heavy tree contains $\mathcal{O}(\log n)$ nodes that have more than one child (in the heavy tree). Alstrup et al. [2] showed how to compute an ART decomposition by computing a set of leafmost light edges (in the spirit of

[3] In some works this has been called a centroid decomposition [10]. It should not be confused with the hierarchical decomposition of the tree obtained by recursively deleting a centroid node, that is, a node whose removal splits the tree into three roughly equal components [7, 15].

[4] Heavy trees are sometimes called *micro trees*, while the tree obtained from T by contracting each micro tree is called a *macro tree*. We avoid this notation to not confuse with the so-called *micro-macro decomposition* [3], which, for a positive integer $k \leq n$, is a partition of the vertices of T into $\mathcal{O}(n/k)$ sets, such that each set S is of size $\mathcal{O}(k)$ it induces a subtree of T and has at most two vertices that have neighbours that are not in S.

HP2 with the base of the logarithm changed from 2 to $\lfloor \log n \rfloor$), removing them along with their descendants from the tree, and recursing. Here, for convenience, we compute an ART decomposition similar to the HP2-realization of a heavy-path decomposition: an edge (u, v) is heavy if and only if both $s(u)$ and $s(v)$ are in $(n/\lfloor \log n \rfloor^{k+1}, n/\lfloor \log n \rfloor^{k}]$. For each heavy tree, we call *branches* its maximal down-the-tree paths in which all nodes except the deepest one have exactly one child (in the heavy tree); each heavy tree has $\mathcal{O}(\log n)$ branches.

Our techniques. In order to answer an HIA query for nodes u and v, we consider $\mathcal{O}(\log n / \log\log n)$ pairs of heavy trees that consist of a heavy tree in the root-to-u path and a heavy tree in the root-to-v path. For each such pair, we compute an induced pair (x, y) of nodes in these heavy trees that are ancestors of u and v, respectively, and have maximum total weight. Similarly to previous work, we observe that not every pair of heavy trees needs to be considered. Instead, it suffices to consider a number of pairs of heavy trees linear to the depth of the tree decompositions by a procedure analogous to the natural algorithm for checking whether there are two elements of a sorted list that sum to a target t: start with two pointers, one at the beginning of the list and one at the end and move each of them in only one direction (either to the right or left). For each pair of heavy trees, we construct data structures that can efficiently handle each of the cases of how the locations of the lowest ancestors of u and v in the heavy trees relate to the locations of the lowest ancestors (in the heavy trees) of same-label leaves. Each data structure considers similar cases as previous work, however now we are working with two trees instead of two paths, and hence need to be more careful. This way, we reduce an HIA query to $\mathcal{O}(\log n / \log\log n)$ predecessor queries. By answering each of these predecessor queries independently, we obtain a data structure that answers HIA queries in $\mathcal{O}(\log n)$ time. Indeed, in our case, each predecessor query requires $\Omega(\log\log n)$ time to be answered independently.

However, crucially, we show how to design the data structures so that all predecessor queries need only two values: the preorder number of u or the preorder number of v. This is achieved by reordering the trees so that heavy edges come last. Then, larger preorder numbers correspond to a larger depth of the lowest ancestor on a branch. This means that the combination of our techniques with fractional cascading would yield a faster algorithm for answering all the predecessor queries; the first one for each queried value would take $\mathcal{O}(\log\log n)$ time, while all subsequent ones would take $\mathcal{O}(1)$ time each. The final technical hurdle is that fractional cascading requires the so-called underlying catalog graph to have polylogarithmic degree [21]. The construction of such a graph is straightforward if T_1 and T_2 are of polylogarithmic degree: roughly speaking, it suffices to consider the Cartesian product of two trees whose nodes represent branches and heavy trees of each of T_1 and T_2. We overcome this difficulty in the general case by reducing the maximum degree of these trees prior to taking their Cartesian product while maintaining all of their desirable properties.

2 Preliminaries

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. Throughout the paper, we perform the same operations on T_1 and T_2 and define objects in these trees, so we are going to use $T_\star$ to denote any of the trees. Similarly, we are going to use $v_\star$ to denote a node $v_1 \in T_1$ or a node v_2 in T_2 etc. as an abbreviation of writing that some property holds for v_i for both $i \in \{1, 2\}$.

Lowest Common Ancestor. LCA queries can be answered in constant time after an $\mathcal{O}(n)$-time preprocessing [17].

LowestCommonAncestor (LCA)
Input: A rooted tree T.
Query: What is the node of largest depth that is an ancestor of both u and v?

Predecessor query. For a static set S, a combination of x-fast tries [23] and deterministic dictionaries [20] yields an $\mathcal{O}(n)$-size data structure that can be built in $\mathcal{O}(n)$ time and answers predecessor queries in $\mathcal{O}(\log\log U)$ time deterministically (cf., [12, Proposition 2]); this is optimal [19].

PredecessorQuery
Input: A set S of n integers from $[U]$.
Query: For a given integer y, what is the largest $x \in S$ such that $x \leq y$?

Range Minimum Query. RMQs can be answered in constant time after an $\mathcal{O}(n)$-time preprocessing [11, 13]. By setting, for each i, $S[i] := |U| - S[i]$, we get a structure for the symmetric RangeMaximumQuery problem.

RangeMinimumQuery (RMQ)
Input: A sequence S of n integers from $[U]$.
Query: For given positions i and j, with $1 \leq i \leq j \leq n$, what is (the position of) the minimum among $S[i], S[i+1], \ldots, S[j]$?

Deterministic Static Dictionary. A dictionary is a structure that stores a set of keys (often with associated values) and allows answering membership queries (or getting the value of a given key). There are multiple randomized solutions, but there is even a deterministic solution with $\mathcal{O}(n)$ space, $\tilde{\mathcal{O}}(n)$-time preprocessing, and constant-time queries [20].

Fractional cascading. Consider a directed graph C, called the *catalog graph*, which has a sorted list (also called a catalog) in each of its nodes. Let the total size of the lists be n. Now, suppose that we want to answer queries of the following type: for a connected subgraph G of C and a query value v, find the predecessor of v in each of the lists stored in the nodes of G.

A naive way of solving this problem would be to ignore any preprocessing and run a separate binary search in the sorted list of each of the nodes of G, for a total of $\mathcal{O}(|G|\log n)$ time. Fractional cascading is a general optimization technique that allows the speed-up of multiple binary searches for the same value over multiple related sorted sequences of objects.

If the degree of each node of the catalog graph is bounded by a constant, the original solution of Chazelle and Guibas [9] answers a query in $\mathcal{O}(\log n + |G|)$ time after a linear-time preprocessing in the comparison model. To be precise, it is sufficient for the catalog graph to have *locally bounded degree* (as per Definition 1 of [9]). Unfortunately, in our case, this is not useful. A subsequent work of Shi and JáJá [21] achieved the same complexities for graphs of polylogarithmic maximum degree in the word RAM model of computation (in the original description, the graph is a tree, however, there is no difficulty in extending this to the general setting considered by Chazelle and Guibas [9]). Crucially, these data structures can also handle the case where the nodes of G are given one by one in an online manner; the only requirement is that each node (other than the first) must be a neighbour of some previous one. Note that the $\mathcal{O}(\log n)$ term in the complexities comes from performing a binary search in the first of the considered lists. Then, the predecessor of the query value v in each of the subsequently considered lists is obtained by following a constant number of pointers, which can be retrieved in $\mathcal{O}(1)$ time. (During the preprocessing phase, the catalogs are augmented in an appropriate manner and said pointers are constructed.) In the word RAM model of computation, the first query can be solved faster using other data structures, e.g., in $\mathcal{O}(\log\log U)$ time with the structure discussed above for the PredecessorQuery problem.

3 An $\tilde{\mathcal{O}}(n)$-size Data Structure with Optimal Query Time

Let $b > 1$ be a parameter to be chosen later. Consider a rooted, weighted, and labelled tree T. The weight of a node u is denoted by $\text{weight}(u)$. For a node v, we denote by $s(v)$ the number of nodes in v's subtree, including v. For an integer k, a node v is on layer k if and only if $n/b^{k+1} < s(v) \leq n/b^k$. An edge that connects nodes of the same layer is called *heavy* and the other edges are *light*. Each maximal subtree that does not contain any light edges is called a *heavy tree*. We stress that a heavy tree might be a singleton.

We decompose each heavy tree into *branches*, that is, maximal down-the-tree paths of nodes, where every node apart from the deepest one has one child (in the heavy tree). The last node is either a leaf of the heavy tree or has at least two children. Note that there can be branches consisting of a single node. We call a node *implicit* if it is an internal (non-leaf) node of the heavy tree with one child; otherwise, we call it *explicit*. For a heavy tree, we obtain a *compacted* version of it, called *compacted heavy tree*, by eliminating all the implicit nodes through the contraction of either of their incident edges. See an example in Figure 2.

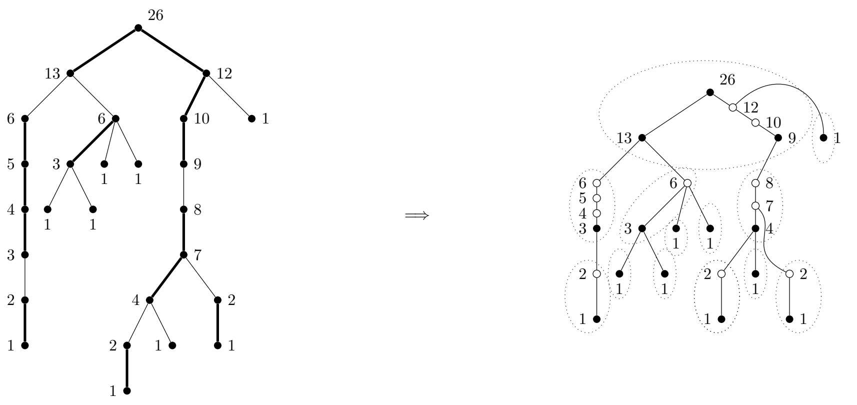

Figure 2 An example tree with $n = 26$ is shown. Each node is labelled with its subtree size, while $b = 3$. On the left, the heavy edges are thick, while the light edges are thin. On the right, the tree is decomposed into heavy trees (marked with ellipses). The compaction of heavy trees is illustrated as follows: empty circles denote implicit nodes, while full circles denote explicit nodes.

Observe that in a compacted tree, every non-leaf node has at least two children. Hence, there are fewer internal nodes than there are leaves of the tree. As every leaf in a heavy tree has a sufficiently big subtree underneath it, we obtain a bound on the total number of nodes inside a single compacted heavy tree; this shows that we obtain an ART decomposition [2].

▶ **Lemma 5.** *There are at most $\mathcal{O}(\log n/\log b)$ layers in a tree T on n leaves and each heavy tree has $\mathcal{O}(b)$ branches.*

Proof. Consider a heavy tree H of T and its compacted form H_C. Let r be the root of H. For every leaf ℓ of H, we have that $s(\ell) > s(r)/b$ as ℓ and r are nodes of the same heavy tree, and hence they are in the same layer of T. As the subtrees of T rooted at the leaves of H are disjoint and their total size is at most $s(r)$, H contains at most b leaves. Further, as there are no internal nodes with one child in H_C, there are at most $b-1$ non-leaf nodes in H_C, and hence H_C has $\mathcal{O}(b)$ nodes. All branches in H are disjoint, so there are $\mathcal{O}(b)$ of them.

Consider the k-th layer of the tree T and a node u in this layer. As the subtree of u has at least one node, we have $1 \le s(u) \le n/b^k$, so $k \le \log_b n = \log n / \log b$. ◀

For an HIA query (v_1, v_2), the paths from v_1 and v_2 to the roots of their respective trees are called *query paths*. The result of an HIA query is a pair of nodes (u_1, u_2), that are on the query paths from v_1 and v_2, respectively. To answer a query (v_1, v_2), we identify the sequences of heavy trees B_1 and B_2 that contain nodes on the query paths from v_1 and v_2, respectively, and perform *restricted HIA queries* for some pairs of those heavy trees. More precisely, in each step of the algorithm, having chosen two heavy trees $B_1[i]$ and $B_2[j]$, we try to find the pair of induced ancestors $(u_1, u_2) \in B_1[i] \times B_2[j]$ of (v_1, v_2) with the maximum combined weight or determine that there is no such pair. A pseudocode for this procedure is given as Algorithm 1.

Algorithm 1 Algorithm for answering HIA queries.

```
function hia($T_1$, $T_2$, $v_1$, $v_2$)
    $(r_1, r_2) \leftarrow$ null
    $(B_1, B_2) \leftarrow$ sequences of heavy trees on paths from $v_1$ and $v_2$ down-the-tree
    $i \leftarrow 0$
    $j \leftarrow |B_2| - 1$
    repeat
        $x_1 \leftarrow$ lowest ancestor of $v_1$ in $B_1[i]$
        $x_2 \leftarrow$ lowest ancestor of $v_2$ in $B_2[j]$
        $(u_1, u_2) \leftarrow$ restricted-hia($v_1$, $v_2$, $B_1[i]$, $B_2[j]$, $x_1$, $x_2$)
        if $(r_1, r_2) =$ null or weight($u_1$) + weight($u_2$) > weight($r_1$) + weight($r_2$)
         then
            $(r_1, r_2) \leftarrow (u_1, u_2)$
        if $(i = |B_1| - 1)$ and $(j = 0)$ then
            return $(r_1, r_2)$

        if $i + 1 < |B_1|$ and roots of $B_1[i+1]$ and $B_2[j]$ are induced then
            $i \leftarrow i + 1$
        else
            $j \leftarrow j - 1$
    until false
```

To make the description more modular, we provide x_1 and x_2 to the restricted HIA query, where $x_\star$ is the lowest ancestor of $v_\star$ in the considered heavy tree. Note that x_1 is either the parent of the root of $B_1[i+1]$ if $i + 1 < |B_1|$ or x_1 equals to v_1 otherwise, and similarly for x_2. We note that in Algorithm 1, we ask restricted HIA queries about the same pair (v_1, v_2) for various pairs $(B_1[i], B_2[j])$ of heavy trees and it may be that $v_1 \notin B_1[i]$ and/or $v_2 \notin B_2[j]$, whereas we also provide nodes $x_\star$, in which $v_\star$ connects to the respective heavy subtree in $B_\star$

▶ **Lemma 6.** *Algorithm 1 performs $O(\log n / \log b)$ restricted HIA queries to find the heaviest induced ancestors of nodes u and v.*

Proof. The paths from $v_\star$ to the roots pass heavy trees with monotonically decreasing indices of layers, so the sequences B_1 and B_2 contain at most $\mathcal{O}(\log n/\log b)$ elements. After every restricted HIA query, we either increase i or decrease j by 1, so the total number of restricted HIA queries that we perform is at most $|B_1| + |B_2| = \mathcal{O}(\log n/\log b)$.

The correctness follows from the monotonicity of being induced. For an induced pair of nodes, any pair of their (weak) ancestors is also induced. Conversely, if a pair is not induced, any pair of their (weak) descendants is also not induced.

This implies that if the roots of $B_1[i]$ and $B_2[j]$ are induced then there is no need to query for any pair $(B_1[i'], B_2[j'])$ with $(i' < i) \wedge (j' < j)$, as such a query would return a pair of (strict) ancestors of the pair returned by the restricted HIA query for $(B_1[i], B_2[j])$. We call a pair of trees $(B_1[i'], B_2[j'])$ *dominated*, if there exist $i > i'$ and $j > j'$ such that the roots of $B_1[i]$ and $B_2[j]$ are induced. We show that the algorithm performs restricted HIA queries at Line 9 exactly for those pairs of heavy trees that (i) are not dominated and (ii) for which the result of the restricted HIA query is not null.

The algorithm maintains the invariant that each of the restricted HIA queries is called for the pairs of trees for which their roots are induced and that the pair $(B_1[i], B_2[j])$ is not dominated. This is true for the first iteration, where the pair $(B_1[0], B_2[|B_2| - 1])$ is considered, because $B_2[|B_2| - 1]$ has a leaf and the root of $B_1[0]$ is the root of T_1. Now we show that the invariant is maintained later. We start with the assumption that the result of `restricted-hia`$(v_1, v_2, B_1[i], B_2[j], x_1, x_2)$ is not null and this pair is non-dominated. Now, we have to distinguish between two cases.

Case 1: We next consider pair $(B_1[i+1], B_2[j])$. It means that the check in Line 15 confirmed that the roots of the the trees are induced, so the the result of calling `restricted-hia`$(v_1, v_2, B_1[i+1], B_2[j], x_1, x_2)$ is not null. Further, this pair of heavy paths is not dominated since $(B_1[i], B_2[j])$ is not dominated.

Case 2: We next consider pair $(B_1[i], B_2[j-1])$. This can only happen if $i = |B_1| - 1$ or the roots of $B_1[i+1]$ and $B_2[j]$ are not induced; in either of these cases, $(B_1[i], B_2[j-1])$ is not dominated. Further, as the answer to the HIA query for pair $(B_1[i], B_2[j])$ is not null, the answer for $(B_1[i], B_2[j-1])$ cannot be null either.

Thus, the invariant is maintained in both cases.

Clearly, the heaviest induced pair of ancestors of v_1, v_2 belongs to a pair of heavy trees that satisfy conditions (i) and (ii). We claim that we process all such pairs. Observe that for a fixed j there are two indices $0 \le i_1 < i_2 \le |B_1|$ such that the pair $B_1[i], B_2[j]$ is dominated for $0 \le i < i_1$, non-dominated for $i_1 \le i < i_2$, and corresponds to a null answer for $i_2 \le i < |B_1|$. By the invariant, just after any decrease of j in Line 18 it holds that $i \ge i_1$, as $B_1[i], B_2[j]$ is non-dominated. Actually $i = i_1$, because the pair $B_1[i-1], B_2[j]$ is dominated as in the previous step we considered the pair $B_1[i], B_2[j+1]$ for which the answer was not null. As in the next steps we process all i up to (but excluding) i_2, the claim follows. ◀

3.1 Restricted HIA Queries

In this subsection, we present a data structure that efficiently answers restricted HIA queries.

▶ **Theorem 7.** *For every two trees T_1, T_2 on n leaves and an integer parameter $b \in [n]$, there exists an $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$-size data structure that can be computed in $\tilde{\mathcal{O}}(nb^2/\log^2 b)$ time and answers (i) queries about whether the roots of two given heavy trees are induced in constant time, (ii) any restricted HIA query* `restricted-hia`$(v_1, v_2, H_1, H_2, x_1, x_2)$ *in* *constant time plus the time required to answer a predecessor query about* $\mathrm{pre}(v_1)$ *and one about* $\mathrm{pre}(v_2)$*; these predecessor queries are performed on two out of* $\mathcal{O}(b^2)$ *(preprocessed) lists stored for the pair of heavy trees* (H_1, H_2)*.*

We divide the proof into three parts: we first describe the preprocessing phase, then discuss the properties of the created data structure, and, finally, present the query procedure.

Preprocessing. First, we compute the partition of the edges of each of T_1, T_2 into heavy and light, and the implied heavy trees. For each node, we store its assignment to the heavy tree and to the branch to which it belongs. Further, for each $T_\star$, we build a linear-size data structure for answering LCA queries in $\mathcal{O}(1)$ time [17]. For each node in $T_\star$, we fix the order of its children such that the children that are in the same heavy tree are last. (The order of children that are connected to the parent with the same type of edge, heavy or light, is arbitrary.) Next, we compute preorder traversals of $T_\star$, for each node u, we denote by $\mathrm{pre}(u)$ the preorder number of u and by $T_\star[p]$ the node of $T_\star$ whose preorder number is p; we have $T_\star[\mathrm{pre}(u)] = u$. Additionally, for each label, we identify the leaves of T_1 and T_2 with that label (recall the labels in a single tree are pairwise distinct).

Next, for each pair (ℓ_1, ℓ_2) of leaves with the same label, we iterate over all pairs $(B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2])$ of heavy trees on their query paths and insert a point to the data structure for each pair of branches in $B_1^{\ell_1}[i_1] \times B_2^{\ell_2}[i_2]$. This procedure is formalized as Algorithm 2.

Algorithm 2 Preprocessing for a pair of leaves (ℓ_1, ℓ_2) with the same label.

```
1 procedure add_label($T_1$, $T_2$, $\ell_1$, $\ell_2$)
2     $(B_1^{\ell_1}, B_2^{\ell_2}) \leftarrow$ sequence of heavy trees on query paths from $\ell_1$ and $\ell_2$
3     for $i_1 \leftarrow 0, 1, \ldots, |B_1^{\ell_1}| - 1$ do
4         for $i_2 \leftarrow 0, 1, \ldots, |B_2^{\ell_2}| - 1$ do
5             for $e_1 \leftarrow$ branch of $B_1^{\ell_1}[i_1]$ do
6                 for $e_2 \leftarrow$ branch of $B_2^{\ell_2}[i_2]$ do
7                     $w_1 \leftarrow \mathsf{LCA}(\ell_1, \text{lowest node on } e_1)$
8                     $w_2 \leftarrow \mathsf{LCA}(\ell_2, \text{lowest node on } e_2)$
9                     insert point $(\mathrm{pre}(w_1), \mathrm{pre}(w_2))$ to $D^{B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2]}[e_1, e_2]$
```

We call pairs (B_1, B_2) of heavy trees that are processed by Algorithm 2 *relevant*. We first run this algorithm once just to record all relevant pairs of heavy trees, without inserting any points to any structures. We then sort the relevant pairs, remove duplicates, and construct a deterministic dictionary over them [20]. This allows us to check in constant time if the roots of two trees are induced because this is equivalent to checking if the pair of trees is relevant. For each relevant pair of heavy trees, we initialize an array D^{B_1,B_2} indexed by pairs of branches (e_1, e_2), where e_1 is a branch in B_1 and e_2 is a branch in B_2. In each entry of the array, we create (store a pointer to) a data structure for the corresponding pair of branches. We then re-run Algorithm 2, inserting the points to the structures as needed, with the help of the deterministic dictionary built for relevant pairs of heavy trees. As each branch $e_\star$ belongs to a unique heavy tree, we often drop the superscript and write $D[e_1, e_2]$ instead of $D^{B_1,B_2}[e_1, e_2]$. We call a pair (e_1, e_2) of branches *relevant* if and only if pair of their assigned heavy trees is relevant. By Lemma 5, every query path is decomposed into $\mathcal{O}(\log n / \log b)$ parts on different layers and each heavy tree has $\mathcal{O}(b)$ branches. Hence, for every pair (ℓ_1, ℓ_2) of leaves with the same label, we insert a point to $\mathcal{O}(b^2 \log^2 n / \log^2 b)$ structures.

Finally, for each relevant pair of branches (e_1, e_2), we perform the following postprocessing of structure $D[e_1, e_2]$:

- Remove all points (x, y) for which there exists another point (x', y') such that $x \le x', y \le y'$ and $(x, y) \ne (x', y')$. This can be done in $\tilde{\mathcal{O}}(|D[e_1, e_2]|)$ time by sorting the points and processing them in the left-to-right order.
- Let $D_x[e_1, e_2]$ and $D_y[e_1, e_2]$ be the sets of x- and y-coordinates of the remaining points, respectively. We build a data structure for the PredecessorQuery problem for each of $D_x[e_1, e_2]$ and $D_y[e_1, e_2]$ separately.
- We build a data structure for the RangeMaximumQuery problem for the points remaining in $D[e_1, e_2]$ sorted by x-coordinate, where the weight of a point (x, y) is $\texttt{weight}(T_1[x]) + \texttt{weight}(T_2[y])$.

We call the above stage the postprocessing of $D[e_1, e_2]$.

To summarize the whole preprocessing stage for trees $T_\star$, for each of the n labels we add $\mathcal{O}(b^2 \log^2 n/\log^2 b)$ points to structures $D[\cdot,\cdot]$, for a total number of $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$ points. The postprocessing of all the structures $D[\cdot,\cdot]$ takes nearly linear time in their size and hence the total running time is $\tilde{\mathcal{O}}(nb^2/\log^2 b)$. The structures for the PredecessorQuery and RangeMaximumQuery problems have size linear in the number of elements they are built over and hence the total space is $\mathcal{O}(nb^2 \log^2 n/\log^2 b)$.

Properties of structures $\mathbf{D[e_1, e_2]}$. In this paragraph, we show some properties of the structures $D[e_1, e_2]$ that are useful for answering restricted HIA queries efficiently.

▶ **Property 8.** *For every pair (w_1, w_2) added to $D^{B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2]}[e_1, e_2]$, $w_\star$ is either on $e_\star$ or on the path from the highest node of $e_\star$ to the root of $B_\star^{\ell_\star}[i_\star]$.*

Proof. Recall that $w_\star$ is the lowest common ancestor of $\ell_\star$ and the lowest node q on $e_\star$. Observe that as $B_\star^{\ell_\star}[i_\star]$ is on the query path from $\ell_\star$, the root r of $B_\star^{\ell_\star}[i_\star]$ is an ancestor of both $\ell_\star$ and q. Hence, $w_\star$ lies on the r-to-q path, which directly yields the statement. ◀

Note that after the first step of postprocessing, $D[e_1, e_2]$ satisfies the following property:

▶ **Property 9.** *After the postprocessing, for every pair (e_1, e_2) of branches, after sorting the points of $D[e_1, e_2]$ increasingly by x-coordinate, the sequence of points is also sorted decreasingly by y-coordinate.*

Informally, we can now consider a one-dimensional problem, with points forming a sequence that can be efficiently navigated both in x- and y-coordinates via predecessor queries.

We next show how the computed data structures $D[e_1, e_2]$ enable us to answer restricted HIA queries efficiently.

Answering a restricted HIA query. We are now ready to present how to answer a restricted HIA query for a pair (v_1, v_2) of nodes and heavy trees B_1 and B_2 on the query paths from v_1 and v_2. Let $(r_1, r_2) = \texttt{restricted-hia}(v_1, v_2, B_1, B_2, x_1, x_2)$ be a pair of ancestors of v_1 and v_2 within the trees B_1 and B_2 that are induced and have the maximum total weight. Recall that $x_\star$ is the lowest weak ancestor of $v_\star$ that is in $B_\star$ and let $e_\star$ be the branch containing $x_\star$. Further, let ℓ be the label inducing (r_1, r_2) and let leaves $\ell_\star$ share this label.

First, we show that we can find an induced pair of ancestors of v_1 and v_2 with the maximum combined weight using the structure $D[e_1, e_2]$ before postprocessing. Then, we show that after the postprocessing stage, we can still retrieve the correct answer but more efficiently, by performing predecessor queries for $\text{pre}(x_1)$ and $\text{pre}(x_2)$. Finally, we show that we can call predecessor queries for $\text{pre}(v_1)$ and $\text{pre}(v_2)$ instead of $\text{pre}(x_1)$ and $\text{pre}(x_2)$. The last step is not important for the correctness or efficiency of a single restricted HIA query but improves the complexity of Algorithm 1. Indeed, as all predecessor queries are for one of $\text{pre}(v_1)$ or $\text{pre}(v_2)$, we can use fractional cascading. We explain this final component in detail in Section 3.2.

Recall that in Algorithm 2, we insert point $(\text{pre}(w_1), \text{pre}(w_2))$ to $D[e_1, e_2]$, where $w_\star = \mathsf{LCA}(\ell_\star, \text{lowest node on } e_\star)$. In the proof of Property 8, we mention that $w_\star$ always belongs to $B_\star$ as the root of $B_\star$ is an ancestor of both $\ell_\star$ and the lowest node on $e_\star$. There are two possible relative locations of $w_\star$ and $x_\star$ within a heavy tree:

- ℓ is *below* $x_\star$ when $w_\star$ is a (not necessarily proper) descendant of $x_\star$;
- ℓ is *attached above* $x_\star$ when $w_\star$ is a proper ancestor of $x_\star$.

There are four cases for the relative locations of ℓ with respect to x_1 and x_2:

Case 1: ℓ is attached above x_1 and x_2,

Case 2: ℓ is attached above x_1 and ℓ is below x_2,

Case 3: ℓ is attached above x_2 and ℓ is below x_1,

Case 4: ℓ is below x_1 and x_2.

We next treat each of these cases. For each of them, we retrieve the pair of induced ancestors of x_1 and x_2 with the largest total weight among all pairs of ancestors induced by a label ℓ appropriately located with respect to x_1 and x_2. Each of these variants gives us a candidate pair for the restricted heaviest induced ancestors of x_1 and x_2. In the end, we return the candidate with the largest total weight. Similar case analysis was performed in previous solutions for the HIA problem, e.g., in [14].

▶ **Lemma 10.** *The answer to* `restricted-hia` $(v_1, v_2, B_1, B_2, x_1, x_2)$ *can be retrieved from the information stored in* $D[e_1, e_2]$ *before the postprocessing.*

Proof. By Property 8, for every two points $(\text{pre}(w_1), \text{pre}(w_2))$ and $(\text{pre}(w'_1), \text{pre}(w'_2))$ added to $D[e_1, e_2]$, we have that w_1 is either a weak ancestor or a descendant of w'_1, and similarly for w_2 and w'_2. Hence, the preorder numbers of the nodes correspond to their depths and we can check the ancestry relation by comparing them: for nodes u, u' on a path, u is a weak ancestor of u' if and only if $\text{pre}(u) \le \text{pre}(u')$.

Using this property, we show how to reduce each of the four cases listed above to finding a specific point in a particular rectangular subset of points. For now, we ignore the efficiency of the queries (a trivial implementation takes linear time) and focus on showing that the correct answer to the restricted HIA query can be retrieved from $D[e_1, e_2]$ before the postprocessing.

Case 1: Every leaf ℓ that is attached above x_1 and x_2 in nodes w_1 and w_2 makes the pair (w_1, w_2) a candidate result of the restricted HIA query. Hence we need to find a point (x, y) in $D[e_1, e_2]$ such that $x < \text{pre}(x_1)$, $y < \text{pre}(x_2)$, and $\texttt{weight}(T_1[x]) + \texttt{weight}(T_2[y])$ is maximum. Then, $(T_1[x], T_2[y])$ is a restricted HIA candidate pair for (v_1, v_2).

Case 2: We need to find a point $(x, y) \in D[e_1, e_2]$ such that $x < \text{pre}(x_1)$, $y \ge \text{pre}(x_2)$ and $\texttt{weight}(T_1[x])$ is maximized. Then, $(T_1[x], x_2)$ is a restricted HIA candidate pair for (v_1, v_2).

Case 3: This case is symmetric to Case 2. We need to find a point $(x, y) \in D[e_1, e_2]$ such that $x \ge \text{pre}(x_1)$, $y < \text{pre}(x_2)$ and $\texttt{weight}(T_2[y])$ is maximized. Then, $(x_1, T_2[y])$ is a restricted HIA candidate pair for (v_1, v_2).

Case 4: We need to check if there exists a point (x, y) such that $x \ge \text{pre}(x_1)$ and $y \ge \text{pre}(x_2)$. If so, the pair (x_1, x_2) is a restricted HIA candidate pair for (v_1, v_2).

In each of the cases, we return a pair, if one exists, of induced ancestors of (x_1, x_2) and hence also of (v_1, v_2). The label ℓ of leaves ℓ_1 and ℓ_2 that induces the pair (r_1, r_2) of heaviest induced ancestors of (v_1, v_2) in $B_1 \times B_2$ inserted the point to $D[e_1, e_2]$, since B_1 and B_2 are on the query paths from ℓ_1 and ℓ_2, respectively. Hence, the pair (r_1, r_2) is found while considering one of the four cases. ◀

Now, we show that it suffices to run the above algorithm only for the points in $D[e_1, e_2]$ after the postprocessing stage.

▶ **Lemma 11.** *The answer to* `restricted-hia` $(v_1, v_2, B_1, B_2, x_1, x_2)$ *can be retrieved from the information stored in* $D[e_1, e_2]$ *after the postprocessing.*

Proof. As discussed in the proof of Lemma 10, for any two points $(\mathrm{pre}(w_1), \mathrm{pre}(w_2))$ and $(\mathrm{pre}(w'_1), \mathrm{pre}(w'_2))$ added to $D[e_1, e_2]$, w_i and w'_i lie on a single root-to-leaf path, so their preorder numbers are in the same order as their depths in the tree. Recall that the trees are monotonically weighted, that is, the weights along each root-to-leaf path are increasing, so we have that $\mathrm{pre}(w_i) \le \mathrm{pre}(w'_i)$ implies $\texttt{weight}(w_i) \le \texttt{weight}(w'_i)$.

Let $w = (\mathrm{pre}(w_1), \mathrm{pre}(w_2))$ be the point corresponding to the answer found by the algorithm presented in Lemma 10. Note that the returned induced pair of ancestors is not necessarily (w_1, w_2), e.g., it can be (w_1, x_2). Suppose that w was removed during the postprocessing phase. If so, it happened because there exists a point $w' = (\mathrm{pre}(w'_1), \mathrm{pre}(w'_2))$ where $\mathrm{pre}(w_\star) \le \mathrm{pre}(w'_\star)$ and $w \ne w'$. If w' is processed in a different case than w, then the pair of ancestors corresponding to w' has a larger total weight than the one returned, yielding a contradiction. If w' is processed in the same case as w, then the pair of ancestors corresponding to w' gives a pair of ancestors whose total weight is not smaller than that of the returned pair. Hence, the reduction presented in the proof of Lemma 10 still holds for the set $D[e_1, e_2]$ after the postprocessing. ◀

Next, we present how to implement each of the four cases in Lemma 10 efficiently using the fact that the points in $D[e_1, e_2]$ have been postprocessed.

▶ **Lemma 12.** *The answer to* `restricted-hia` $(v_1, v_2, B_1, B_2, x_1, x_2)$ *can be retrieved from the information stored in* $D[e_1, e_2]$ *after the postprocessing, with two predecessor queries: one for* $\mathrm{pre}(x_1)$ *and one for* $\mathrm{pre}(x_2)$.

Proof. By computing the predecessor of $\mathrm{pre}(x_1)$ in $D_x[e_1, e_2]$, we obtain intervals $\mathcal{I}_x^{x<\mathrm{pre}(x_1)}$ and $\mathcal{I}_x^{x\ge\mathrm{pre}(x_1)}$ of $D_x[e_1, e_2]$. Similarly, intervals $\mathcal{I}_y^{y<\mathrm{pre}(x_2)}$ and $\mathcal{I}_y^{y\ge\mathrm{pre}(x_2)}$ of $D_y[e_1, e_2]$ are obtained by computing the predecessor of $\mathrm{pre}(x_2)$ in $D_y[e_1, e_2]$. Note that by Property 9, points from $\mathcal{I}_y^{y\ge\mathrm{pre}(x_2)}$ (resp. $\mathcal{I}_y^{y<\mathrm{pre}(x_2)}$) of $D_y[e_1, e_2]$ correspond to points from the interval of $D_x[e_1, e_2]$ that we denote $\mathcal{I}_x^{y\ge\mathrm{pre}(x_2)}$ ($\mathcal{I}_x^{y<\mathrm{pre}(x_2)}$). Hence, we can translate each of the conditions on points in the cases of Lemma 10 to an intersection $\mathcal{I}_x^{\text{Case } i}$ of two intervals on $D_x[e_1, e_2]$. This reduces each of the four cases to:

Case 1: Find the point with maximum weight $\texttt{weight}(T_1[x]) + \texttt{weight}(T_2[y])$ in $\mathcal{I}_x^{\text{Case 1}}$ using an RMQ.

Case 2: By the monotonicity of weights with respect to x-coordinates, the point with maximum weight $\texttt{weight}(T_1[x])$ in $\mathcal{I}_x^{\text{Case 2}}$ is the rightmost element of $\mathcal{I}_x^{\text{Case 2}}$.

Case 3: By the monotonicity of weights with respect to y-coordinates and Property 9, the point with maximum weight $\texttt{weight}(T_2[y])$ in $\mathcal{I}_x^{\text{Case 3}}$ is the leftmost element of $\mathcal{I}_x^{\text{Case 3}}$.

Case 4: It suffices to check if $\mathcal{I}_x^{\text{Case 4}}$ is non-empty. ◀

▶ **Lemma 13.** *The answer to* `restricted-hia` $(v_1, v_2, B_1, B_2, x_1, x_2)$ *can be retrieved from the information stored in* $D[e_1, e_2]$ *after the postprocessing, with two predecessor queries: one for* $\mathrm{pre}(v_1)$ *and one for* $\mathrm{pre}(v_2)$.

Proof. Recall that in the approach presented in Lemma 12 we compute the predecessor of $\mathrm{pre}(x_1)$ in $D_x[e_1, e_2]$ in order to divide $D_x[e_1, e_2]$ into $\mathcal{I}_x^{x<\mathrm{pre}(x_1)}$ and $\mathcal{I}_x^{x\geq\mathrm{pre}(x_1)}$ and that all elements in $D_x[e_1, e_2]$ are of the form $\mathrm{pre}(w_1)$ for a node w_1 on the path from the lowest node on e_1 to the root of B_1.

We consider only v_1 and x_1 as the analysis for v_2 and x_2 is symmetric. We can focus on the case where $v_1 \neq x_1$, as the other case is immediate. We clearly have that $\mathrm{pre}(v_1) \geq \mathrm{pre}(x_1)$ as v_1 is a descendant of x_1. Recall that in the preprocessing stage, we reordered children of every node in such a way that children connected by a light edge are before those connected by a heavy edge, so if there is a child x_1' of x_1 on e_1 we have $\mathrm{pre}(x_1) \leq \mathrm{pre}(v_1) < \mathrm{pre}(x_1')$, as v_1 is a descendant of a light child of x_1 (by the definition of x_1). Hence, the predecessor of $\mathrm{pre}(v_1)$ is the same as the predecessor of $\mathrm{pre}(x_1)$ in $D_x[e_1, e_2]$. ◀

This concludes the proof of Theorem 7.

Finally, by setting the value of b to $\lfloor \log^\epsilon n \rfloor$ for any constant $\epsilon > 0$, we obtain a data structure using $\mathcal{O}(n \log^{2+2\epsilon} n/(\log\log^\epsilon n)^2) = \mathcal{O}(1/\epsilon^2 \cdot n \log^{2+2\epsilon} n/(\log\log n)^2)$ space capable of answering restricted HIA queries in constant time plus the time required for answering two predecessor queries: one for $\mathrm{pre}(v_1)$ and one for $\mathrm{pre}(v_2)$. Algorithm 1 performs $\mathcal{O}(\log n/\log b)$ restricted HIA queries in order to answer an HIA query, so the total time required is $\mathcal{O}(1/\epsilon \cdot \log n)$. However, as all the predecessor queries ask about one of two values in different lists that are related to each other, we can make use of fractional cascading.

We omit the $1/\epsilon$ factor in further sections and use the $\mathcal{O}_\epsilon(\cdot)$ notation instead to indicate a dependency on ϵ.

3.2 Fractional Cascading

For most of the cases described in the previous subsection, our structures are issuing predecessor queries. This is the only reason why the time complexity of an HIA query with our approach is not yet $\mathcal{O}_\epsilon(\log n/\log\log n)$. We will exploit the fact that all these queries look for the same target value ($\mathrm{pre}(v_1)$ for structures built for T_1 and $\mathrm{pre}(v_2)$ for structures for T_2) but for different pairs of branches, which enables us to use fractional cascading.

We can think of creating two catalog graphs from $T_1 \times T_2$ with nodes representing pairs (e_1, e_2) of branches, storing the contents of $D_x[e_1, e_2]$ in one catalog graph and those of $D_y[e_1, e_2]$ in the other one. The execution of Algorithm 1 can be then seen as the traversal of a path in such a graph where, for a pair $(B_1[i], B_2[j])$ of heavy trees for which a restricted HIA query is performed by the algorithm, we query the catalogs of the nodes representing the pair of branches (e_1, e_2) that contain the lowest weak ancestors of (v_1, v_2) that are in $B_1[i]$ and $B_2[j]$, respectively. The problem with this direct approach is that it is not guaranteed that the degree of all vertices in each catalog graph is polylogarithmic: we might need to move from the node corresponding to two branches (e_1, e_2) to any node corresponding to two branches (e_1', e_2'), where the heavy tree containing e_1' is attached to e_1, and there could be even $\Omega(n)$ such branches e_1'.

We need to create catalog graphs in which the length of the considered path for each HIA query is $\mathcal{O}_\epsilon(\log n/\log\log n)$, while the degree of each node is $\mathcal{O}(\mathrm{polylog}\, n)$ in order to be able to apply the result of Shi and JáJá [21]. This would ensure that all predecessor queries in $D_x[\cdot,\cdot]$ and $D_y[\cdot,\cdot]$ take constant time, apart from the first ones, which take $\mathcal{O}(\log\log n)$

time using an x-fast trie. We describe how to build the appropriate catalog graphs below. As the shape of the graph for $D_x[\cdot,\cdot]$ and $D_y[\cdot,\cdot]$ is the same and only the contents of catalogs differ, we will only describe how to build one of them.

We preprocess T_1 and T_2 separately, first to compute trees $B(T_\star)$ and then to build catalog graphs $C(T_\star)$. From this, we build the catalog graph C that can be seen as a Cartesian product of $C(T_1)$ and $C(T_2)$. More precisely, each node in C is a pair (v,w) for $v \in C(T_1)$ and $w \in C(T_2)$. For an edge between nodes v_1 and v_2 in $C(T_1)$, in the final catalog graph, we create edges between nodes (v_1,w) and (v_2,w) for each $w \in C(T_2)$. Analogically, for an edge between nodes w_1 and w_2 in $C(T_2)$, in the final catalog graph we create edges between nodes (v,w_1) and (v,w_2) for each $v \in C(T_1)$. This way, if the degrees of $C(T_1)$ and $C(T_2)$ are polylogarithmic, so is the degree of C.

We now explain how to build tree $B(T_\star)$ from $T_\star$. $B(T_\star)$ contains nodes representing heavy trees (called *heavy tree nodes*) and nodes representing branches (called *branch nodes*):

- for each heavy tree H, we connect all branch nodes representing branches in H as children of the heavy tree node representing H,
- for each heavy tree H, except the tree containing the root of $T_\star$, we connect the heavy tree node representing H as child of the branch node representing the branch containing the parent of the root of H.

▶ **Proposition 14.** *The depth of $B(T_\star)$ is $\mathcal{O}_\epsilon(\log n/\log\log n)$ and each heavy tree node has $\mathcal{O}(\log^\epsilon n)$ children.*

Recall that every heavy tree has $\mathcal{O}(b) = \mathcal{O}(\log^\epsilon n)$ branches, so every heavy tree node has $\mathcal{O}(\log^\epsilon n)$ children. Any root-to-leaf path in $B(T_\star)$ alternates between heavy tree nodes and branch nodes. For any root-to-v path p in $T_\star$, there is a corresponding path p' in $B(T_\star)$ that visits the heavy tree nodes that correspond to the heavy trees that intersect p and the branch nodes for which Algorithm 1 (when called for a pair of nodes containing v) could call predecessor queries for $D_x[\cdot,\cdot]$ and $D_y[\cdot,\cdot]$. For any p, p' is of length $\mathcal{O}_\epsilon(\log n/\log\log n)$ and can be found in $\mathcal{O}(|p'|)$ time by following the path from the heavy tree containing v to the root of $B(T_\star)$.

We now describe how to create a catalog graph $C(T_\star)$ from $B(T_\star)$. The construction is recursive and follows from the proof of Lemma 15 applied with $d = \mathcal{O}_\epsilon(\log n/\log\log n)$ and $b = \lfloor \log^\epsilon n \rfloor$. The idea is to replace the structure of children of branch nodes having too many children with appropriate gadgets that roughly preserve the structure of the tree, do not increase the depth of the tree asymptotically and reduce the degree of each node to $\mathcal{O}(\operatorname{polylog} n)$. Due to Proposition 14, we do not need to alter the structure of children for heavy tree nodes.

▶ **Lemma 15.** *For any depth-d tree $B(T_\star)$ with $\mathcal{O}(n)$ nodes and parameter b, there is a tree $C(T_\star)$ satisfying all the following conditions:*

- *$C(T_\star)$ has $\mathcal{O}(n)$ nodes,*
- *all nodes of $C(T_\star)$ have degree $\mathcal{O}(b)$,*
- *$C(T_\star)$ has depth $d + \mathcal{O}(\log n/\log b)$,*
- *for each simple path p in $B(T_\star)$, we can compute in $\mathcal{O}(|p'|)$ time a simple path p' in $C(T_\star)$, such that p is a subsequence of p' and $|p'| \le d + \mathcal{O}(\log n/\log b)$.*

Proof. Consider a (branch) node e of $B(T_\star)$ whose children, read left-to-right, are heavy tree nodes $h_1, h_2, \ldots, h_k$ for $k > b$. We replace this subgraph that contains $k+1$ nodes with a gadget graph whose root is e and whose set of nodes is a superset of $\{e\} \cup \{h_i : i \in [k]\}$.

For each node u, let $s(u)$ be the number of nodes in the subtree of u in the considered tree. Let $s_0 = 0$ and, for $i \geq 1$, let s_i be the prefix sum $s(h_1) + s(h_2) + \ldots + s(h_i)$. If there is an integer ℓ such that $s_{i-1} < \ell \cdot s(e)/b$ and $s_i \geq \ell \cdot s(e)/b$, we mark h_i. We call each set of consecutive unmarked nodes an *interval*. As $\ell \leq b$, there are $\mathcal{O}(b)$ marked nodes and $\mathcal{O}(b)$ intervals.

We create a gadget for e (and, recursively, for some other nodes created in the construction, as described later) as follows:

- We attach as a child of e every node that is either marked or is the only element of its interval.
- For each interval of more than one node, we create a new node i_j, called an *interval node*, attach it as a child of e, and attach all the nodes of the interval as children of i_j.

We recursively apply the same construction for any of the newly created interval nodes $i_1, i_2, \ldots, i_m$ whose degree is larger than b. See Figure 3 in Appendix B for an illustration.

From the construction, it follows that the degree of each node of $C(T_\star)$ is $\mathcal{O}(b)$ and that the size of $C(T_\star)$ is $\mathcal{O}(n)$, as all new nodes are of out-degree at least 2.

Let u be a node in $T_\star$. We now show that the depth for a node $e_u \in C(T_\star)$ representing a branch containing u is at most $d + \mathcal{O}(\log n/\log b)$ by considering the edges above e_u in $C(T_\star)$. The edges can be of two types:

- Edges that are incident to at least one node that is not an interval node. By the depth of $B(T_\star)$, we have at most d such edges.
- Edges between interval nodes. For each such edge (v, z), we have $s(v) \geq s(z) \cdot b$. Thus, similarly to the proof of Lemma 5, there are $\mathcal{O}(\log n/\log b)$ such edges on the path from the root of $C(T_\star)$ to e_u.

This concludes the proof of the bound on the depth of $C(T_\star)$.

Each simple path p in $B(T_\star)$ naturally corresponds to a simple path p' in $C(T_\star)$. In particular, for each edge (v, w) in $B(T_\star)$, one can explicitly store a path $C(T_\star)$ to which (v, w) corresponds. The concatenation of all such paths for edges on p yields a simple path p' in $\mathcal{O}(|p'|)$ time. As the depth of $C(T_\star)$ is $d + \mathcal{O}(\log n/\log b)$, the bound on $|p'|$ follows. ◀

From $C(T_1)$ and $C(T_2)$ constructed as in Lemma 15, we create the catalog graph C as described earlier. Only nodes that represent pairs of branches contain non-empty original catalogs. After the original catalogs are filled, we run the preprocessing of fractional cascading and appropriate augmented catalogs are created for all nodes in C as described in [9, 21]. The $\mathcal{O}_\epsilon(\log n/\log\log n)$ predecessor queries coming from restricted HIA queries performed in Algorithm 1 are naturally reduced to a constant number of queries to the x-fast tries and the traversal of a path of length $\mathcal{O}_\epsilon(\log n/\log\log n)$ in C. This takes $\mathcal{O}_\epsilon(\log n/\log\log n)$ time in total and concludes the description of our data structure and the proof of Theorem 2.

References

1 Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, 2022. doi:10.1007/s00453-022-00955-7.

2 S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings 39th Annual Symposium on Foundations of Computer Science*, pages 534–543, 1998. doi:10.1109/SFCS.1998.743504.

3 Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *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, and Jakub Radoszewski. Longest common factor after one edit operation. In *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Proceedings*, pages 14–26, 2017. doi:10.1007/978-3-319-67428-5_2.

5 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

6 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *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, and Anna Östlin. The complexity of constructing evolutionary trees using experiments. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001*, pages 140–151, 2001. doi:10.1007/3-540-48224-5_12.

8 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic Longest Common Substring in Polylogarithmic Time. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, pages 27:1–27:19, 2020. doi:10.4230/LIPIcs.ICALP.2020.27.

9 Bernard Chazelle and Leonidas Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, January 1986. doi:10.1007/BF01840440.

10 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

11 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/s00453-012-9683-x.

12 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

13 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

14 Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013*, 2013. URL: http://cccg.ca/proceedings/2013/papers/paper_29.pdf.

15 Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for centroid decomposition. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, pages 274–282, 2019. doi:10.1007/978-3-030-32686-9_20.

16 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

17 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

18 Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968. doi:10.1145/321479.321481.

19 Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.

20 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, pages 84–95, 2008. doi:10.1007/978-3-540-70575-8_8.

21 Qingmin Shi and Joseph JáJá. Novel transformation techniques using q-heaps with applications to computational geometry. *SIAM J. Comput.*, 34:1474–1492, January 2005. doi:10.1137/S0097539703435728.

22 Peter Weiner. Linear pattern matching algorithms. In *14th FOCS*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

23 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:doi.org/10.1016/0020-0190(83)90075-3.

A Description of the $\mathcal{O}(\log n)$-Query-Time Data Structure of [14]

As mentioned in the introduction, Gagie et al. [14] sketched an $\mathcal{O}(n \log^2 n)$-size data structure that answers HIA queries in $\mathcal{O}(\log n)$ time, in the last paragraph of Subsection 2.1 of their work. They construct a data structure for each pair of heavy trees of T_1 and T_2 and reduce an HIA query for nodes u and v to a predecessor query in the data structure of each of $\mathcal{O}(\log n)$ pairs of heavy trees. The idea for improving the (fully-described) $\mathcal{O}(\log n \log\log n)$-time procedure for answering queries with a more efficient one is similar to ours and involves fractional cascading. They would need to build a catalog graph with nodes being pairs of heavy paths, reduce its degree, and make sure that the predecessor queries have the same target (by asking for the preorder numbers of u and v). This is similar to what we describe in Section 3.2 for a different tree decomposition.

B Omitted Figure from Section 3.2

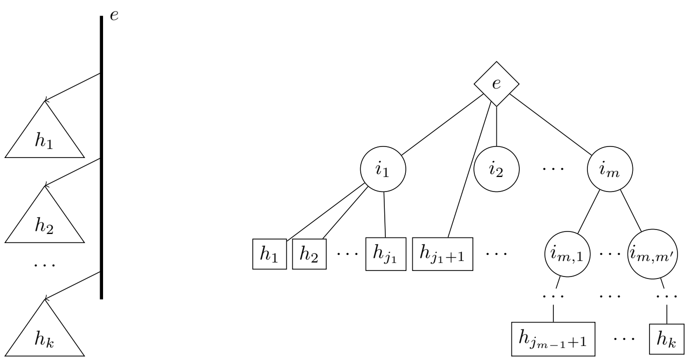

Figure 3 On the left there is a branch e of $T_\star$, with outgoing edges to $h_1, h_2, \ldots, h_k$ in $B(T_\star)$. On the right, there is a catalog graph gadget created for e, which is part of $C(T_\star)$. Circles denote interval nodes and rectangles heavy tree nodes. Some of the intervals are recursively replaced with the gadget to decrease their degree. Heavy tree nodes have degree $\mathcal{O}(b)$. In $C(T_\star)$, e's parent is the heavy tree to which it belongs, while the children of each h_i are the branches in h_i.