Algorithmica (2022) 84:2088–2105
https://doi.org/10.1007/s00453-022-00955-7

The Heaviest Induced Ancestors Problem: Better Data Structures and Applications

Paniz Abedin[1] · Sahar Hooshmand[2] · Arnab Ganguly[3] · Sharma V. Thankachan[4]

Received: 22 May 2020 / Accepted: 4 March 2022 / Published online: 29 March 2022

Abstract
Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two rooted trees with an equal number of leaves. The leaves are labeled, and the labeling of the leaves in $\mathcal{T}_2$ is a permutation of those in $\mathcal{T}_1$. Nodes are associated with weight, such that the weight of a node u, denoted by $W(u)$, is more than the weight of its parent. A node $x \in \mathcal{T}_1$ and a node $y \in \mathcal{T}_2$ are induced, iff their subtrees have at least one common leaf label. A *heaviest induced ancestor* query $\mathsf{HIA}(u_1, u_2)$ with input nodes $u_1 \in \mathcal{T}_1$ and $u_2 \in \mathcal{T}_2$ asks to output the pair (u_1^*, u_2^*) of induced nodes with the highest combined weight $\mathsf{W}(u_1^*) + \mathsf{W}(u_2^*)$, such that u_1^* is an ancestor of u_1 and u_2^* is an ancestor of u_2. This is a useful primitive in several text processing applications. Gagie et al. (Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013, Waterloo, Ontario, Canada, 2013) introduced this problem and proposed three data structures with the following space-time trade-offs: (i) $O(n \log^2 n)$ space and $O(\log n \log\log n)$ query time, (ii) $O(n \log n)$ space and $O(\log^2 n)$ query time, and (iii) $O(n)$ space and $O(\log^{3+\epsilon} n)$ query time. Here n is the number of nodes in both trees combined and $\epsilon > 0$ is an arbitrarily small constant.

An early version of this work appeared in CPM 2018 [1].

✉ Sharma V. Thankachan
sharma.thankachan@gmail.com

Paniz Abedin
pabedin@floridapoly.edu

Sahar Hooshmand
hooshmand@csudh.edu

Arnab Ganguly
gangulya@uww.edu

1 Department of Computer Science, Florida Polytechnic University, Lakeland, FL, USA

2 Department of Computer Science, California State University Dominguez Hills, Carson, CA, USA

3 Department of Computer Science, University of Wisconsin - Whitewater, Whitewater, WI, USA

4 Department of Computer Science, University of Central Florida, Orlando, FL, USA

We present two new data structures with better space-time trade-offs: (i) $O(n \log n)$ space and $O(\log n \log\log n)$ query time, and (ii) $O(n)$ space and $O(\log^2 n/\log\log n)$ query time. Additionally, we present new applications of these results.

Keywords Data structure · String algorithms · Orthogonal range queries

1 Introduction

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two rooted trees, having n_1 and n_2 nodes respectively, and let $n = n_1 + n_2$. Each node u in either of the trees is associated with a weight, denoted by $W(u)$. Moreover, $\mathsf{W}(u) > \mathsf{W}(\mathsf{parent}(u))$, where $\mathsf{parent}(u)$ is the parent of node u. For convenience, the pre-order rank of a node u is also denoted by u. Each tree has exactly $m \leq \min\{n_1, n_2\}$ leaves. Leaves in both trees are labeled, and the labeling of the leaves in $\mathcal{T}_2$ is a permutation of the labeling of the leaves in $\mathcal{T}_1$. A pair of nodes, each from $\mathcal{T}_1$ and $\mathcal{T}_2$, are *induced* if the leaves in the respective subtrees have at least one common label. Any node on the path from a node u to the tree's root is called an ancestor of u. Moreover, an ancestor v of u is a proper ancestor iff $u \neq v$. We revisit the following problem, introduced by Gagie et al. [22]. See Fig. 1 for an illustration.

Problem 1 (Heaviest Induced Ancestors (HIA) Problem [22]) *Given a node $u_1 \in \mathcal{T}_1$ and a node $u_2 \in \mathcal{T}_2$, find* $\mathsf{HIA}(u_1, u_2)$*, which is defined as the pair of* induced *nodes (u_1^*, u_2^*) with the highest combined weight* $\mathsf{W}(u_1^*) + \mathsf{W}(u_2^*)$*, such that u_1^* (resp., u_2^*) is an ancestor of u_1 (resp., u_2).*

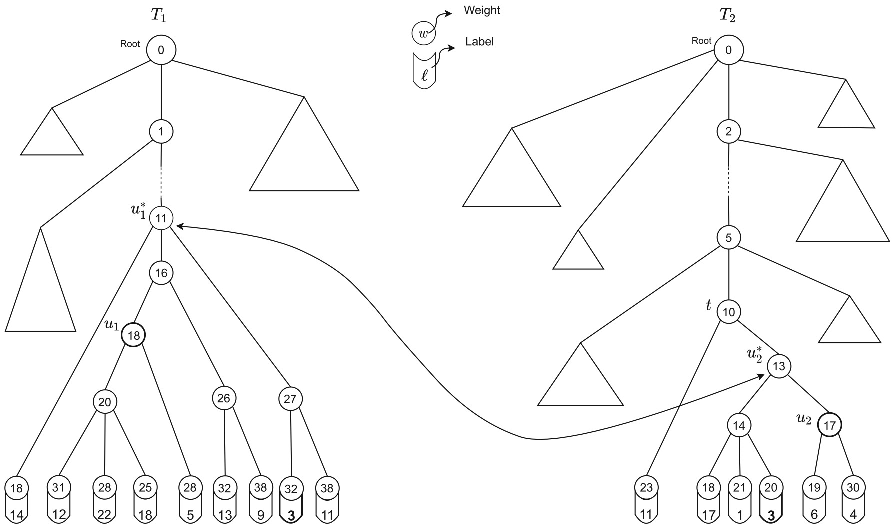

Fig. 1 Nodes u_1^* and u_2^* are ancestors of u_1 and u_2 respectively. They are induced since their subtrees have leaf label 3 in common. Note that u_1^* and t are also induced but we don't report them as the answer to the HIA problem since they have a lower combined weight (11+10 = 21) compared to $W(u_1^*) + W(u_2^*)$ which is 24

Gagie et al. [22] achieved the following space-time trade-offs in the standard word RAM model of computation with word size $\Omega(\log n)$ bits. Here and hereafter, ϵ is an arbitrarily small positive constant.

- an $O(n \log^2 n)$ space and $O(\log n \log\log n)$ query time
- an $O(n \log n)$ space and $O(\log^2 n)$ query time
- an $O(n)$ space and $O(\log^{3+\epsilon} n)$ query time.

Throughout this paper, the space is measured in words (unless specified otherwise). We present two new data structures, with improved bounds.

Theorem 1 *A heaviest induced ancestors query over two trees of n nodes in total can be answered*

- *in $O(\log n \log\log n)$ time using an $O(n \log n)$ space data structure, or*
- *in $O\left(\dfrac{\log^2 n}{\log\log n}\right)$ time using an $O(n)$ space data structure.*

The heaviest induced ancestors query is a useful primitive in several text/string processing applications. We now proceed to present some examples.

1.1 Applications to String Matching

Let $\mathsf{LCS}(X, Y)$ denote the longest common substring (LCS) of two strings X and Y.

1.1.1 Longest Common Substring of LZ77 Compressed Strings

Problem 2 *Build a data structure for a string S of length N, whose LZ77 parsing contains n phrases, that supports the following query: given a pattern P, report* $\mathsf{LCS}(S, P)$.

If one were to forego the compression requirement, the problem could be easily solved by maintaining a suffix tree [33] of S in $O(N)$ space yielding $O(|P|)$ query time. On the other hand, we can also answer $\mathsf{LCS}(S, P)$ queries using compressed/succinct data structures, such as the FM Index or Compressed Suffix Array [16, 23, 28], with a slight slow down in the query time. However, for strings having a repetitive structure, LZ77-based compression techniques [36] offer better space efficiency than those obtained using FM-Index or Compressed Suffix Array.

Gagie et al. [22] showed that Problem 2 can be solved using an $O(n \log N + n \log^2 n)$ space index with a very high probability (i.e., greater than $1 - |P|^{-c}$ for some constant c) in $O(|P| \log n \log\log n)$ query time. Alternatively, they presented an $O(n \log N)$ space index with query time $O(|P| \log^2 n)$. Using Theorem 1 and the techniques in [22], we present an improved result for Problem 2 (see Theorem 2). We omit the details as they are immediate from the discussions in [22].

Theorem 2 *Given a string S of length N, we can build an $O(n \log N)$ space structure that reports* $\mathsf{LCS}(S, P)$ *in $O(|P| \log n \log\log n)$ time with a very high probability, where n is the number of phrases in an LZ77 parsing of S.*

1.2 All-Pairs Longest Common Substring Problem

Here we are given a collection $T_1, T_2, \ldots, T_d$ of d strings, each of length roughly n, and the task is to compute $\mathsf{LCS}(T_i, T_j)$ for all (i, j) pairs. This is a useful primitive in several bioinformatics applications [2, 13]. However, a conditional lower bound based on the boolean matrix multiplication suggests that significant improvements over the naive $O(d^2 n)$ time algorithm is unlikely [31] in the general case. However, we present an improved solution for the cases where many strings are highly similar (a.k.a. highly repetitive). Specifically, we present a data structure of space and pre-processing time $\tilde{O}(nd)$, that computes $\mathsf{LCS}(T_i, T_j)$ for any i, j in time $\tilde{O}\left(\frac{\min\{|T_i|,|T_j|\}}{|\mathsf{LCS}(T_i,T_j)|}\right)$. We defer details to Sect. 5.2.

1.2.1 Dynamic Longest Common Substring Problem

In [5], Amir et al. introduced the following problem: build a data structure over two strings T_1 and T_2 of total n characters over an alphabet set Σ, such that given a query (p, α), where $p \in [1, |T_1|]$ and $\alpha \in \Sigma$, report $\mathsf{LCS}(T_1^*, T_2)$, where T_1^* is a new string obtained by replacing the pth character of T_1 by α. They presented an $O(n \log^3 n)$ space data structure with $O(\log^3 n)$ query time. We not only improve their bound, but also propose a solution to solve a more general case, where the query consists of a set S of $(position, character)$ pairs, and the task is to compute $\mathsf{LCS}(T_1^*, T_2)$, where T_1^* is obtained from T_1 by making the changes (substitutions) as specified by S. We achieve the same space-time trade-offs as that of Theorem 1, where time is the time per substitution. Details are deferred to Sect. 5.3.

The problem is even more complicated when the changes are allowed in both strings. Amir et al. [6, 7] proposed an $\tilde{O}(n)$ space solution with query time $\tilde{O}(n^{2/3})$. A new result improving this query to $\tilde{O}(1)$ amortized time has been announced recently [12]. They also showed that the query time is $\Omega(\log n/ \log\log n)$ for any polynomial-size data structure. See [3, 4, 8, 18–21, 32] for other related work.

1.3 Map

In Sect. 2, we revisit some of the well-known data structures that have been used to arrive at our results. Sect. 3 presents an overview of our techniques as an intermediate step into the final data structures. The final data structures for Theorem 1 are presented in Sect. 4. Section 5 is dedicated to applications. We conclude in Sect. 6 with some future directions.

2 Preliminaries and Terminologies

2.1 Predecessor/Successor Queries

Let $\mathcal{S}$ be a subset of $\mathcal{U} = \{0, 1, 2, 3, \ldots, U-1\}$ of size n. A predecessor search query p on $\mathcal{S}$ asks to return p if $p \in \mathcal{S}$, else return $\max\{q < p \mid q \in \mathcal{S}\}$. Similary, a successor query p on $\mathcal{S}$ asks to return p if $p \in \mathcal{S}$, else return $\min\{q > p \mid q \in \mathcal{S}\}$. By preprocessing S into a y-fast trie of size $O(n)$, we can answer such queries in $O(\log\log U)$ time [34].

2.2 Fully-Functional Succinct Tree

Let $\mathcal{T}$ be a tree having n nodes, such that nodes are numbered from 1 to n in the ascending order of their pre-order rank. Also, let ℓ_i denote the ith leftmost leaf. Then by maintaining an index of size $2n + o(n)$ bits, we can answer the following queries on $\mathcal{T}$ in constant time [29]:

– $\mathsf{parent}_{\mathcal{T}}(u)$ = parent of node u.
– $\mathsf{size}_{\mathcal{T}}(u)$ = number of leaves in the subtree of u.
– $\mathsf{nodeDepth}_{\mathcal{T}}(u)$ = number of nodes on the path from u to the root of $\mathcal{T}$.
– $\mathsf{levelAncestor}_{\mathcal{T}}(u, D)$ = ancestor w of u such that $\mathsf{nodeDepth}(w) = D$.
– $\mathsf{lMost}_{\mathcal{T}}(u) = i$, where ℓ_i is the leftmost leaf in the subtree of u.
– $\mathsf{rMost}_{\mathcal{T}}(u) = j$, where ℓ_j is the rightmost leaf in the subtree of u.
– $\mathsf{lca}_{\mathcal{T}}(u, v)$ = lowest common ancestor (LCA) of two nodes u and v.

We omit the subscript “$\mathcal{T}$” if the context is clear.

2.3 Range Maximum Query (RMQ) and Path Maximum Query (PMQ)

Let $A[1, n]$ be an array of n elements. A range maximum query $\mathsf{RMQ}_A(a, b)$ asks to return $k \in [a, b]$, such that $A[k] = \max\{A[i] \mid i \in [a, b]\}$. Path maximum query (PMQ) (or bottleneck edge query [14]) is a generalization of RMQ from arrays to trees. Let $\mathcal{T}$ be a tree having n nodes, such that each node u is associated with a score. A path maximum query $\mathsf{PMQ}_{\mathcal{T}}(a, b)$ returns the node k in $\mathcal{T}$, where k is a node with the highest score among all nodes on the path from node a to node b. Cartesian tree-based solutions exist for both problems. The space and query time are $2n + o(n)$ bits and $O(1)$, respectively [14, 17].

2.4 Orthogonal Range Queries in 2-Dimension

Let $\mathcal{P}$ be a set of n points in an $[1, n] \times [1, n]$ grid. Then,

– An orthogonal range counting query (a, b, c, d) on $\mathcal{P}$ returns the cardinality of $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \in [c, d]\}$
– An orthogonal range emptiness query (a, b, c, d) on $\mathcal{P}$ returns “EMPTY” if the cardinality of the set $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \in [c, d]\}$ is zero. Otherwise, it returns “NOT-EMPTY”.
– An orthogonal range predecessor query (a, b, c) on $\mathcal{P}$ returns the point in $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \leq c\}$ with the highest y-coordinate value, if one exists.
– An orthogonal range successor query (a, b, c) on $\mathcal{P}$ returns the point in $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \geq c\}$ with the lowest y-coordinate value, if one exists.
– An orthogonal range selection query (a, b, k) on $\mathcal{P}$ returns the point in $\{(x, y) \in \mathcal{P} \mid x \in [a, b]\}$ with the kth lowest y-coordinate value.

By maintaining an $O(n)$ space structure, we can answer orthogonal range counting queries in $O(\log / \log\log n)$ time [26], orthogonal range emptiness queries in $O(\log^{\epsilon} n)$ time [11], orthogonal range predecessor/successor queries in $O(\log^{\epsilon} n)$ time [27] and orthogonal range selection queries in $O(\log n/\log\log n)$ time [10]. Alternatively, by maintaining an $O(n \log\log n)$ space structure, we can answer orthogonal range emptiness and orthogonal range predecessor/successor queries in $O(\log\log n)$ time [11, 35].

2.5 Heavy Path and Heavy Path Decomposition

We now define the heavy path decomposition [25, 30] of a rooted tree $\mathcal{T}$ having n nodes. First, the nodes in $\mathcal{T}$ are categorized into light and heavy. The root node is *light*, and exactly one child of every internal node is heavy. Specifically, the child having the largest number of nodes in its subtree with ties broken arbitrarily. The first heavy path of $\mathcal{T}$ is the path starting at $\mathcal{T}$'s root and traversing through every heavy node to a leaf. Each off-path subtree of the first heavy path is further decomposed recursively. Clearly, a tree with m leaves has m heavy paths. Let u be a node on a heavy path H, then $\mathsf{hp\_root}(u)$ is the highest node on H and $\mathsf{hp\_leaf}(u)$ is the lowest node on H. Note that $\mathsf{hp\_root}(\cdot)$ is always light.

Fact 1 For a tree having n nodes, the path from the root to any leaf traverses at most $\lceil \log n \rceil$ light nodes. Consequently, the sum of the subtree sizes of all light nodes (i.e., the starting node of a heavy path) put together is at most $n\lceil \log n \rceil$.

3 Our Framework

We assume that both trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are compacted, i.e., any internal node has at least two children. This ensures that the number of internal nodes is strictly less than the number m of leaves. Thus, $n \leq 4m - 2$. We remark that this assumption can be easily removed without affecting the query time. We maintain the tree topology of $\mathcal{T}_1$ and $\mathcal{T}_2$ succinctly in $O(n)$ bits with constant time navigational support (refer to Sect. 2.2). Define two arrays, $\mathsf{Label}_k[1, m]$ for $k = 1$ and 2, such that $\mathsf{Label}_k[j]$ is the label associated with the jth leaf node in $\mathcal{T}_k$. The following is a set of m two-dimensional points based on tree labels.

$$\mathcal{P} = \{(i, j) \mid i, j \in [1, m] \text{ and } \mathsf{Label}_1[i] = \mathsf{Label}_2[j]\}$$

We pre-process $\mathcal{P}$ into a data structure to support various range queries described in Sect. 2.4. For range counting and selection, we maintain data structures with $O(n)$ space and $O(\log n/\log\log n)$ time. For range successor/predecessor and emptiness queries, we have two options: an $O(n \log\log n)$ space structure with $O(\log\log n)$ time, and an $O(n)$ space structure with $O(\log^{\epsilon} n)$ time. We employ the first result in our $O(n \log n)$ space solution and the second result in our $O(n)$ space solution.

3.1 Basic Queries

Lemma 1 (Induced-Check) *Given two nodes x, y, where $x \in \mathcal{T}_1$ and $y \in \mathcal{T}_2$, we can check if they are induced or not*

- *in $O(\log\log n)$ time using an $O(n \log\log n)$ space structure, or*
- *in $O(\log^{\epsilon} n)$ time using an $O(n)$ space structure, where ϵ is an arbitrarily small positive constant.*

Proof The task can be reduced to a range emptiness query, because x and y are induced iff the set $\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [\mathsf{lMost}(y), \mathsf{rMost}(y)]\}$ is not empty. □

Definition 1 (*Partner*) The partner of a node $x \in \mathcal{T}_1$ w.r.t a node $y \in \mathcal{T}_2$, denoted by $\mathsf{partner}(x/y)$ is the lowest ancestor y' of y, such that x and y' are induced. Likewise, $\mathsf{partner}(y/x)$ is the lowest ancestor x' of x, such that x' and y are induced.

Lemma 2 (Find Partner) *Given two nodes x, y, where $x \in \mathcal{T}_1$ and $y \in \mathcal{T}_2$, we can find* $\mathsf{partner}(x/y)$ *as well as* $\mathsf{partner}(y/x)$

- *in $O(\log\log n)$ time using an $O(n \log\log n)$ space structure, or*
- *in $O(\log^{\epsilon} n)$ time using an $O(n)$ space structure, where ϵ is an arbitrarily small positive constant.*

Proof To find $\mathsf{partner}(x/y)$, first check if x and y are induced. If yes, then $\mathsf{partner}(x/y) = y$. Otherwise, find the last leaf node $\ell_a \in \mathcal{T}_2$ before y in pre-order, such that x and ℓ_a are induced (ℓ_a denotes a-th leftmost leaf). Also, find the first leaf node $\ell_b \in \mathcal{T}_2$ after y in pre-order, such that x and ℓ_b are induced. Both tasks can be reduced to orthogonal range predecessor/successor queries:

$$(\cdot, a) = \arg\max_{j}\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [1, \mathsf{lMost}(y)]\}$$
$$(\cdot, b) = \arg\min_{j}\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [\mathsf{rMost}(y), m]\}$$

Specifically, a is the y-coordinate of the rightmost point in the rectangular region $[\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [1, \mathsf{lMost}(y)]$ and b is the y-coordinate of the leftmost point in the rectangular region $[\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [\mathsf{rMost}(y), m]$. Clearly, an ancestor of y and x are induced iff either ℓ_a or ℓ_b is in its subtree. Therefore, we report the lowest node among $u_a = \mathsf{lca}(\ell_a, y)$ and $u_b = \mathsf{lca}(\ell_a, y)$ as $\mathsf{partner}(x/y)$. The computation of $\mathsf{partner}(y/x)$ is analogous. □

3.2 Overview

For any two nodes u and v in the same tree $\mathcal{T}$, define $\mathsf{Path}(u, v, \mathcal{T})$ as the set of nodes on the path from u to v. Let root_1 be the root of $\mathcal{T}_1$ and root_2 be the root of $\mathcal{T}_2$. Throughout this paper, (u_1, u_2) denotes the input and $\mathsf{HIA}(u_1, u_2) = (u_1^*, u_2^*)$ denotes the output. Clearly, $u_2^* = \mathsf{partner}(u_1^*/u_2)$ and $u_1^* = \mathsf{partner}(u_2^*/u_1)$. Therefore,

$$(u_1^*, u_2^*) = \arg\max_{(x,y)}\{W(x) + W(y) \mid y \in \mathsf{Path}(\mathsf{root}_2, u_2, \mathcal{T}_2) and x = \mathsf{partner}(y/u_1)\}$$

To evaluate the above equation efficiently, we note that the path from root_2 to u_2 intersects at most $\log n$ heavy paths. Therefore, we first focus on solving a restricted version of HIA problem as follows: given $u_1 \in \mathcal{T}_1$ and $u_2, w \in \mathcal{T}_2$, where w is light and an ancestor of u_2, report the pair (u_1', u_2') of induced ancestors of (u_1, u_2) with maximum total weight (if it exists), such that u_2' is on the heavy path rooted at w. Clearly, the final output (u_1^*, u_2^*) is immediate via $O(\log n)$ such queries (one for each light ancestor of u_2). We will show that for any fixed w, an auxiliary structure of space $O(\mathsf{size}(w))$ can answer such queries efficiently. The key intuition is that we need to consider only the subtree of u_2 rooted at w and a corresponding induced subtree of $\mathcal{T}_1$, whose size is also bounded by $O(\mathsf{size}(w))$. Finally, the total size of all such structures is proportional to the sum of subtree sizes of all light nodes in $\mathcal{T}_2$, which is $O(n \log n)$. We then obtain our linear space solution via a compact space encoding of some of the critical components in the first solution.

We now proceed to present the details.

Definition 2 (*Special Nodes*) For each light node $w \in \mathcal{T}_2$, we identify a set $\mathsf{Special}(w)$ of nodes in $\mathcal{T}_1$ (which we call special nodes) as follows: a leaf node $\ell_i \in \mathcal{T}_1$ is *special* iff ℓ_i and w are induced. An internal node in $\mathcal{T}_1$ is special iff it is the lowest common ancestor of two special leaves. Additionally, for each node $x \in \mathsf{Special}(w)$, define its score w.r.t. w as the sum of weights of x and the node $\mathsf{partner}(x/\mathsf{hp\_leaf}(w)) \in \mathcal{T}_2$. Formally,

$$\mathsf{score}_w(x) = W(x) + W(\mathsf{partner}(x/\mathsf{hp\_leaf}(w))$$

Moreover, $|\mathsf{Special}(w)| \leq 2\mathsf{size}(w) - 1$ and $\sum_{w \text{ is a light node}} |\mathsf{Special}(w)| = O(n \log n)$.

To answer an HIA query (u_1, u_2), we first identify some nodes in $\mathcal{T}_1$ and $\mathcal{T}_2$ as follows. Nodes $w_1 = \mathsf{root}_2, w_2, \ldots, w_k$ are the *light* nodes in $\mathsf{Path}(\mathsf{root}_2, u_2, \mathcal{T}_2)$ (in the ascending order of their pre-order ranks). Nodes $t_1, t_2, \ldots, t_k$ are also in $\mathsf{Path}(\mathsf{root}_2, u_2, \mathcal{T}_2)$, such that $t_k = u_2$ and $t_h = \mathsf{parent}(w_{h+1})$ for $h < k$. Therefore, $\mathsf{Path}(\mathsf{root}_2, u_2, \mathcal{T}_2) = \cup_{h=1}^{k} \mathsf{Path}(w_h, t_h, \mathcal{T}_2)$. Next, $\alpha_1, \alpha_2, \ldots, \alpha_k$ and $\beta_1, \beta_2, \ldots, \beta_k$ are nodes in $\mathsf{Path}(\mathsf{root}_1, u_1, \mathcal{T}_1)$, such that for $h = 1, 2, \ldots, k$, $\alpha_h = \mathsf{partner}(t_h/u_1)$ and $\beta_h = \mathsf{partner}(w_h/u_1)$. Clearly, there exists an $f \in [1, k]$ such that $u_2^* \in \mathsf{Path}(w_f, t_f, \mathcal{T}_2)$. See Fig. 2 for an illustration. We now present several lemmas, which form the basis of our solution.

Lemma 3 *The node* $u_1^* \in \mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)$.

Proof We prove this by contradiction arguments.

– Suppose u_1^* is a proper ancestor of α_f. Then, α_f and t_f are induced and $W(\alpha_f) + W(t_f) > W(u_1^*) + W(u_2^*)$, a contradiction. Therefore, u_1^* is in the subtree of α_f.
– Suppose u_1^* is in the proper subtree of β_f. Then, u_1^* and w_f are also induced. Therefore, $\mathsf{partner}(w_f/u_1)$ is u_1^* or a node in the subtree of u_1^*. This implies, $\beta_f = \mathsf{partner}(w_f/u_1)$ is in the proper subtree of β_f, a contradiction. Therefore, u_1^* is an ancestor of β_f.

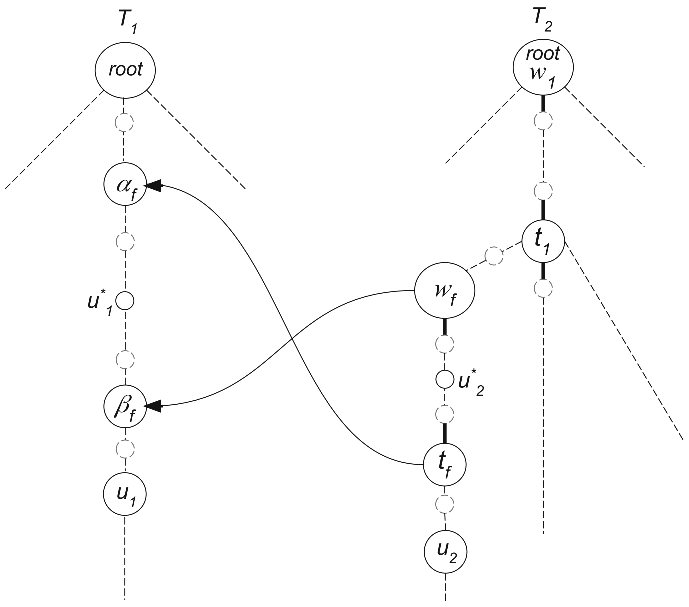

Fig. 2 We refer to Sect. 3.2 for the description of this figure

This completes the proof. □

Lemma 4 *The node* $u_1^* \in \mathsf{Special}(w_f) \cup \{\beta_f\}$.

Proof Let z (if exists) be the first node in $\mathsf{Special}(w_f)$ on the path from u_1^* to β_f. Then,

- if z exists, then $u_1^* \notin \mathsf{Special}(w_f)$ gives a contradiction as follows. The intersection of the following two sets is empty: (i) set of labels of the leaves in the subtree of u_1^*, but not in the subtree of z and (ii) set of labels associated with the leaves in the subtree of w_f. This implies, z and u_2^* are induced (because u_1^* and u_2^* are induced) and $W(z) + W(u_2^*) > W(u_1^*) + W(u_2^*)$, a contradiction.
- otherwise, if z does not exist, then it is possible that $u_1^* \notin \mathsf{Special}(w)$. However, in this case, $u_1^* = \beta_f$ (proof follows from similar arguments as above).

In summary, $u_1^* \in \mathsf{Special}(w_f) \cup \{\beta_f\}$. □

Lemma 5 *For any* $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)\backslash\{\alpha_f\}$, $\mathsf{partner}(x/u_2) = \mathsf{partner}(x/\mathsf{hp\_leaf}(w_f))$.

Proof We claim that for any $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)\backslash\{\alpha_f\}$, $\mathsf{partner}(x/u_2)$ is a proper ancestor of t_f. The proof follows from contradiction as follows. Suppose, there exists an $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)\backslash\{\alpha_f\}$, such that $\mathsf{partner}(x/u_2)$ is in the subtree of t_f. Then, x and t_f are induced. This means, $\alpha_f = \mathsf{partner}(t_f/u_1)$ is a node in the subtree of x, a contradiction.

Since, $\mathsf{partner}(x/u_2)$ is a proper ancestor of t_f, $\mathsf{partner}(x/u_2) = \mathsf{partner}(x/r)$ for any node r in the subtree of t_f. Therefore, by choosing $r = \mathsf{hp\_leaf}(w_f)$, we obtain Lemma 5. □

Corollary 1 *For any* $x \in \Big(\mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)\backslash\{\alpha_f\}\Big)$,

$$W(x) + W(\mathsf{partner}(x/u_2)) = W(x) + W(\mathsf{partner}(x/\mathsf{hp\_leaf}(w_f))) = \mathsf{score}_{w_f}(x).$$

Lemma 6 *The node* $u_1^* \in \{\alpha_f, \beta_f, \gamma_f\}$*, where*

$$\gamma_f = \arg\max_x\{\mathsf{score}_{w_f}(x) \mid x \in \mathsf{Special}(w_f) \cap \Big(\mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)\backslash\{\alpha_f, \beta_f\}\Big).$$

Proof Follows from Lemmas 3, 4, 5 and Corollary 1. □

Lemma 7 *Let* $\mathcal{C} = \cup_{h=1}^k\{\alpha_h, \beta_h, \gamma_h\}$*, where*

$$\gamma_h = \arg\max_x\{\mathsf{score}_{w_h}(x) \mid x \in \mathsf{Special}(w_h) \cap \Big(\mathsf{Path}(\alpha_h, \beta_h, \mathcal{T}_1)\backslash\{\alpha_h, \beta_h\}\Big).$$

Then,

$$(u_1^*, u_2^*) = \arg\max_{(x,y)}\{W(x) + W(y) \mid x \in \mathcal{C}and\, y = \mathsf{partner}(x/u_2)\}.$$

Proof Since f is unknown, we invoke Lemma 5 for $f = 1, 2, 3, \ldots, k \le \log n$. Recall that n is the total number of nodes in both trees. □

Next, we show how to transform the result in Lemma 7 into a data structure.

4 Our Data Structures

We start by defining a crucial component of our solution.

Definition 3 (*Induced Subtree*) The induced subtree $\mathcal{T}_1(w)$ of $\mathcal{T}_1$ w.r.t. a light node $w \in \mathcal{T}_2$ is a tree having exactly $|\mathsf{Special}(w)|$ number of nodes, such that

- for each node $x \in \mathcal{T}_1(w)$, there exists a node $\mathsf{Map}_w(x) \in \mathsf{Special}(w)$ and
- for each $x' \in \mathsf{Special}(w)$, there exists a node $\mathsf{invMap}_w(x') \in \mathcal{T}_1(w)$, such that

$$\mathsf{lca}_{\mathcal{T}_1}(\mathsf{Map}_w(x), \mathsf{Map}_w(y)) = \mathsf{Map}_w(\mathsf{lca}_{\mathcal{T}_1(w)}(x, y))$$

Note that a node x is a leaf in $\mathcal{T}_1(w)$ iff $\mathsf{Map}_w(x)$ is a leaf in $\mathcal{T}_1(w)$. In the following lemmas, we present two space-time trade-offs on induced subtrees.

Lemma 8 *By maintaining an* $O(n \log n)$ *space structure, we can compute* $\mathsf{Map}_w(\cdot)$ *and* $\mathsf{invMap}_w(\cdot)$ *for any light node* $w \in \mathcal{T}_2$ *in time* $O(1)$ *and* $O(\log\log n)$*, respectively.*

Proof Let $L_w[1, |\mathsf{Special}(w)|]$ be an array, such that $L_w[x] = \mathsf{Map}_w(x)$. For each w, maintain L_w and a y-fast trie [34] over it. The total space is $O(n \log n)$. Now, any $\mathsf{Map}_w(\cdot)$ query can be answered in constant time. Also, for any $x' \in \mathsf{Special}(w)$, $\mathsf{invMap}_w(x')$ is the number of elements in L_w that are $\le x'$. Therefore, an $\mathsf{invMap}_w(\cdot)$ can be reduced to a predecessor search and answered in $O(\log\log n)$ time. □

Lemma 9 *By maintaining an $O(n)$ space structure, we can compute* $\mathsf{Map}_w(\cdot)$ *and* $\mathsf{invMap}_w(\cdot)$ *for any light node $w \in \mathcal{T}_2$ in time $O(\log n / \log\log n)$.*

Proof Let node p be the rth leaf in $\mathcal{T}_1(w)$ and $q = \mathsf{Map}_w(p)$ be the sth leaf in $\mathcal{T}_1$. Then, s is the x-coordinate of the rth point in $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, m] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$ in the ascending order of x-coordinates. Also, r is the number of points in $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, s] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$. Therefore, given p, we can compute r, then s and q in $O(\log n / \log\log n)$ time via a range selection query on $\mathcal{P}$. Similarly, given q, we can compute s and then r and p in $O(\log n / \log\log n)$ time via a range counting query on $\mathcal{P}$.

Now, if p is an internal node in $\mathcal{T}_1(w)$, then $\mathsf{Map}_w(p)$ is the same as $\mathsf{lca}_{\mathcal{T}_1}(\mathsf{Map}_w(\ell_L), \mathsf{Map}_w(\ell_R))$, where ℓ_L and ℓ_R are the first and last leaves in the subtree of p. Similarly, if q is an internal node in $\mathcal{T}_1$, then $\mathsf{invMap}_w(q) = \mathsf{lca}_{\mathcal{T}_1(w)}(\mathsf{invMap}_w(\ell_A), \mathsf{invMap}_w(\ell_B))$ as follows:

$$(A, \cdot) = \arg\min_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(q), \mathsf{rMost}(q)] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$$
$$(B, \cdot) = \arg\max_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(q), \mathsf{rMost}(q)] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$$

Here, A and B can be computed via range successor/predecessor queries in $O(\log^\epsilon n)$ time. Therefore, the total time is $\log^\epsilon n + \log n / \log\log n = O(\log n / \log\log n)$ time. ☐

Lemma 10 *Given an input (a, b, w), where w is a light node in $\mathcal{T}_2$ and, a and b are nodes in $\mathcal{T}_1(w)$, we can report the node with the highest* $\mathsf{score}_w(\mathsf{Map}_w(\cdot))$ *over all nodes on the path from a to b in $\mathcal{T}_1(w)$ in $O(1)$ time using an $O(n)$ space structure.*

Proof For each $\mathcal{T}_1(w)$, maintain the Cartesian tree for path maximum query (refer to Sect. 2.3). Space for a particular w is $|\mathsf{Special}(w)|(2 + o(1))$ bits and space over all light nodes w in $\mathcal{T}_2$ is $O(n \log n)$ bits (from Fact 1), equivalently $O(n)$ space (in words). For an input (a, b, w), the answer is $\mathsf{PMQ}_{\mathcal{T}_1(w)}(a, b)$. ☐

4.1 Our $O(n \log n)$ Space Data Structure

We maintain $\mathcal{T}_1$ and $\mathcal{T}_2$ explicitly so that the weight of any node in either of the trees can be accessed in constant time. Moreover, we maintain a fully-functional succinct representation of their topologies (refer to Sect. 2.2) for supporting various operations in $O(1)$ time. Additionally, we maintain the structures for answering Induced-Check and Find-Partner queries in $O(\log\log n)$ time, data structures for range predecessor/successor queries on $\mathcal{P}$ in $O(\log\log n)$ time (refer to Sect. 2.4) and the structures described in Lemmas 8 and 10. Thus, the total space in words is $O(n \log n)$.

We now present the algorithm for computing the output (u_1^*, u_2^*) for a given input (u_1, u_2). The followings are the key steps:

1. Find w_h and t_h for $h = 1, 2, \ldots, k \le \log n$.
2. Find α_h and β_h for $h = 1, 2, \ldots, k \le \log n$.

3. Let α'_h be the first and β'_h be the last special node (w.r.t. w_h) on the path from α_h (excluding α_h) to β_h (excluding β_h). Also, let
$$\gamma_h = \mathsf{Map}_{w_h}\Big(\mathsf{PMQ}_{\mathcal{T}_1(w_h)}\Big(\mathsf{invMap}_{w_h}(\alpha'_h), \mathsf{invMap}_{w_h}(\beta'_h)\Big)\Big).$$
Compute γ_h for $h = 1, 2, \ldots, k \le \log n$.
4. Obtain $\mathcal{C} = \cup_{h=1}^{k}\{\alpha_h, \beta_h, \gamma_h\}$ and report
$$(u_1^*, u_2^*) = \arg\max_{(x,y)}\{W(x) + W(y) \mid x \in \mathcal{C} \textbf{ and } y = \mathsf{partner}(x/u_2)\}.$$

The correctness follows immediately from Lemma 7. We now bound the time complexity. Step 1 takes $O(k)$ time and step 2 takes $O(k)$ number of Find-Partner queries with $O(\log\log n)$ time per query. The procedure for computing α'_h and β'_h is the following.

Find the child α''_h of α_h on the path from α_h to β_h. Then $\alpha'_h = \mathsf{lca}_{\mathcal{T}_1}(\ell_{a_h}, \ell_{b_h})$, where ℓ_{a_h} (resp. ℓ_{b_h}) is the first (resp. last) special leaf in the subtree of α''_h (w.r.t w_h). To compute a_h and b_h, we rely on range predecessor/successor queries on $\mathcal{P}$:

$$(a_h, \cdot) = \arg\min_i\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(\alpha''_h), \mathsf{rMost}(\alpha''_h)] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$
$$(b_h, \cdot) = \arg\max_i\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(\alpha''_h), \mathsf{rMost}(\alpha''_h)] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

Find the rightmost special (w.r.t. w_h) leaf ℓ_{d_h} before β_h and the leftmost special (w.r.t. w_h) leaf ℓ_{g_h} after the last leaf in the subtree of β_h. For this, we rely on range predecessor/successor queries on $\mathcal{P}$:

$$(d_h, \cdot) = \arg\max_i\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, \mathsf{lMost}(\alpha''_h) - 1] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$
$$(g_h, \cdot) = \arg\min_i\{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{rMost}(\alpha''_h) + 1, m] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

Then, $\beta'_h = \mathsf{lca}_{\mathcal{T}_1}(\ell_{d_h}, \ell_{g_h})$ if β_h and w_h are not induced (i.e., there does not exist a special node (w.r.t. w_h) under β_h). Otherwise, β'_h is the lowest node among $\mathsf{lca}_{\mathcal{T}_1}(\ell_{d_h}, \beta_h)$ and $\mathsf{lca}_{\mathcal{T}_1}(\beta_h, \ell_{g_h})$.

The time for a range predecessor/successor query on $\mathcal{P}$ is $O(\log\log n)$. Therefore, the computation of α'_h and β'_h takes $O(\log\log n)$ time, and an additional $O(\log\log n)$ for evaluating γ_h. Therefore, the total time for step 3 is $O(k\log\log n)$. Finally, step 4 also takes $O(k\log\log n)$ time. By putting everything together, the total time complexity is $k\log\log n = O(\log n\log\log n)$.

4.2 Our Linear Space Data Structure

We obtain our linear space data structure by replacing all super-linear space components in the previous solution by their space-efficient counterparts. Specifically, we use linear space structures for Induced-Check, Find-Partner, and range predecessor/successor with query time $O(\log^{\epsilon} n)$. Also, we use the structure in Lemma 9 instead of the structure in Lemma 8. Thus, the total space is $O(n)$.

The query algorithm remains the same. The time complexity is: $O(k)$ for step 1, $O(k \log^{\epsilon} n)$ for step 2, $O(k \log n / \log\log n)$ for step 3 and $O(k \log^{\epsilon} n)$ for step 4. Thus, total time is $k \log n / \log\log n = O(\log^2 n / \log\log n)$.

5 Applications to String Processing

5.1 Suffix Trees and Suffix Arrays

Let $T[1,n]$ be a text over an alphabet set Σ and let $\$ \notin \Sigma$ be a special symbol. The suffix tree ST_T (resp., prefix tree PT_T) of T is a compact trie over all strings in the set $S = \{T[i,n] \circ \$ \mid i \in [1,n]\}$ (resp., $S^R = \{\overleftarrow{T[1,i]} \circ \$ \mid i \in [1,n]\}$). Here $\circ$ denotes concatenation and $\overleftarrow{T[1,i]}$ denotes the reverse of $T[1,i]$. The suffix array SA_T and the inverse suffix array ISA_T (resp., the prefix array PA_T and the inverse prefix array IPA_T) of T are arrays of length n, such that $\mathsf{SA}_T[i] = j$ and $\mathsf{ISA}_T[j] = i$ (resp., $\mathsf{PA}_T[i] = j$ and $\mathsf{IPA}_T[j] = i$) iff $T[j,n] \circ \$$ (resp., $\overleftarrow{T[1,j]} \circ \$$) is the ith smallest string in S (resp., S^R) in the lexicographic order.

Both ST_T and PT_T have exactly n leaves, and the edges are labeled. For a node u in either of the trees, we use $\mathsf{path}(u)$ to denote the concatenation of edge labels on the path from the tree's root to u. The $\mathsf{path}(\cdot)$ of ith leftmost leaf in ST_T (resp., PT_T) is the same as the ith smallest string in S (resp., S^R) in the lexicographic order. Therefore, for any two nodes u and v within the same tree with w being their lowest common ancestor (LCA), the longest common prefix (LCP) of $\mathsf{path}(u)$ and $\mathsf{path}(v)$ is the same as $\mathsf{path}(\mathsf{LCA}(u,v))$. Note that the prefix tree (resp., prefix array) of a text is equivalent to the suffix tree (resp., suffix array) of the reverse of the text.

The suffix tree over a collection of strings $T_1, T_2, \ldots, T_d$ over an alphabet set Σ is called a generalized suffix tree (GST), which is a compact trie over all strings in $\cup_{j=1}^{d} \{T_j[i, |T_j|] \circ \$_j \mid i \in [1, |T_j|]\}$, where $\$_1, \$_2, \ldots, \$_d$ are distinct symbols that do not appear in Σ. The corresponding (generalized) suffix array, prefix tree, and prefix array can be defined similarly.

All the above data structures can be constructed in linear space and time (assuming integer alphabet whose size is bounded by $n^{O(1)}$) [15, 33]. After that, we can find $\mathsf{LCA}(\cdot,\cdot)$ in $O(1)$ time [9]. Therefore, the length of the longest common prefix (or suffix) of any two strings (or their substrings) in the collection can be computed in constant time. We refer to [24] for further reading.

5.2 All-Pairs Longest Common Substring Problem

The formal problem definition along with our result are as below:

Definition 4 Given a collection $T_1, T_2, \ldots, T_d$ of d strings, compute $\mathsf{LCS}(T_i, T_j)$ for all (i, j) pairs.

Theorem 3 *A collection $T_1, T_2, \ldots, T_d$ of d strings of total length N can be preprocessed into a data structure in $\tilde{O}(N)$ space and time, that supports an $\mathsf{LCS}(T_i, T_j)$ query for any i, j pair in $\tilde{O}\left(\frac{\min\{|T_i|,|T_j|\}}{|\mathsf{LCS}(T_i,T_j)|}\right)$ time.*

5.2.1 Data Structure

We maintain a generalized suffix tree GST and a generalized prefix tree GPT over the collection. Additionally, for each $i \in [1, d]$, we maintain our HIA data structure over the pair of trees ST_{T_i} and PT_{T_i}, such that the weight of nodes being their $\mathsf{stringDepth}$. We say that a node u_1 in ST_{T_i} and a node u_2 in PT_{T_i} are induced iff there exists a k, such that the leaf (say ℓ_a) corresponding to the suffix $T_i[k, |T_i|]$ is under u_1 and the leaf (say ℓ_b) corresponding to the (reverse of the) prefix $T_i[1, k-1]$ is under u_2. To align this with the original definition of HIA problem, we assign label k to both ℓ_a and ℓ_b. The total space, as well as the preprocessing time, is bounded by $\tilde{O}(N)$.

5.2.2 Query Algorithm

Without loss of generality, we assume $|T_i| \leq |T_j|$. First, we present an efficient procedure for accomplishing the following task: compute $\lambda(T_i, k, T_j)$, the length of the longest substring of T_i, which covers a position in $\{k-1, k\}$ and also has an occurrence in T_j. We say that a substring $T_j[x, y]$ covers k iff $k \in [x, y]$.

- Find the largest f and the largest r, such that $T_i[k, k+f-1]$ and $T_i[k-r, k-1]$ occurs in T_j (say at positions a and b, respectively). This step takes $O(\log n)$ number of LCP queries on the GST. Then, find the lowest node u_1 in ST_{T_j} and the lowest node u_2 in PT_{T_j}, such that $T_i[k, k+f-1]$ is a prefix of $\mathsf{path}(u_1)$ and $\overleftarrow{T_i[k-r, k-1]}$ is a prefix of $\mathsf{path}(u_2)$. Note that u_1 is on the path from root ending at the leaf corresponding to the suffix of T_j starting at a. Similarly, u_2 is on the path from root ending at the leaf corresponding to the prefix of T_j ending at b. Therefore, u_1 and u_2 can be computed in $O(\log n)$ time via binary search.
- If $W(u_1) = f$ and $W(u_2) = r$, simply find $(u_1^*, u_2^*) = \mathsf{HIA}(u_1, u_2)$ and report $W(u_1^*) + W(u_2^*)$. Otherwise, find $(u_1', u_2') = \mathsf{HIA}(\mathsf{parent}(u_1), \mathsf{parent}(u_2))$, the lowest ancestor u_2'' of u_2 (resp., u_1'' of u_1), such that u_1 and u_2'' (resp., u_1'' and u_2) are induced. Then, report $\max\{f + W(u_2''), W(u_1'') + r, W(u_1') + W(u_2')\}$.

The time complexity is $\tilde{O}(1)$ and the correctness can be easily verified. We can use the above procedure to check whether $|\mathsf{LCS}(T_i, T_j)| \geq \tau$ for any given τ as follows: repeat the above procedure for all values of $k \in \{\tau, 2\tau, 3\tau, \ldots\}$. Then report YES if at least one among the answers is greater than or equal to τ, and NO otherwise. Therefore, via a simple binary search on τ, we can compute $|\mathsf{LCS}(T_i, T_j)|$. Note that the values of τ chosen are always greater than $|\mathsf{LCS}(T_i, T_j)|/2$. This yields the desired query time.

5.3 Dynamic Longest Common Substring Problem

Problem 3 (LCS after k changes) *Given two strings T_1 and T_2 of total length n over an alphabet set Σ, build a data structure that supports the following query: given a set S of k $(position, character)$ pairs, where*

$$S = \{(p_i, \alpha_i) \mid p_i \in [1, |T_1|], i \in [1, k] \textbf{\textit{ and }} \alpha_i \in \Sigma\}$$

report (the length of) $\mathsf{LCS}(T_1^*, T_2)$*, where T_1^* is the string obtained from T_1 by replacing its p_i-th character by α_i for each $i \in [1, k]$.*

We achieve the following result.

Theorem 4 *There exist data structures with the following space-time trade-offs for the* LCS after k changes *problem on two strings of total length n:*

1. *$O(n \log n)$ space and $O(k \cdot \log n \log \log n)$ query time, and*
2. *$O(n)$ space and $O\left(k \cdot \dfrac{\log^2 n}{\log \log n}\right)$ query time.*

Let $p_1 < p_2 < \cdots < p_k$ and $\phi = \cup_{i=1}^{k} \{p_i - 1, p_i\}$. Our approach is to find the longest common substring of T_1^* and T_2 (i) that does not cover any position in ϕ and (ii) covers a position in ϕ, and then report the longest among them as $\mathsf{LCS}(T_1^*, T_2)$. We handle these cases separately, and the following convention will be used: any subarray or substring over a range $[a, b]$ is empty if $a > b$.

5.3.1 Handling Case 1

Let $\psi = \{[1, p_1 - 2], [p_1 + 1, p_2 - 2], [p_2 + 1, p_3 - 2], \ldots, [p_k + 1, n]\}$. Clearly, the answer we are looking for is $\max\{|\mathsf{LCS}(T_1^*[i, j], T_2)| \mid [i, j] \in \psi\}$. Since $\mathsf{LCS}(T_1^*[i, j], T_2) = \mathsf{LCS}(T_1[i, j], T_2)$ for all $[i, j] \in \psi$, we can solve this case in $O(k \log n)$ time using the structure below.

Lemma 11 *The strings T_1 and T_2 can be preprocessed into an $O(n)$ structure, where $|T_1| + |T_2| = n$, that supports the following query in $O(\log n)$ time: given a range $[x, y]$, report (the length of)* $\mathsf{LCS}(T_1[x, y], T_2)$.

Proof Maintain an array L, where $L[i] = |\mathsf{LCP}(T_1[i, |T_1|], T_2)|$ and range maximum query (RMQ) data structure over it. To answer a query $[x, y]$, first find the rightmost position z, such that $z + L[z] - 1 < y$. This is possible in $O(\log n)$ time via a binary search since $i + L[i] - 1$ is monotonic. Then, compute $|\mathsf{LCS}(T_1[x, z]), T_2)| = \mathsf{RMQ}_L(x, z)$ and $|\mathsf{LCS}(T_1[z + 1, y]), T_2)| = y - (z + 1) + 1$, and report the largest among them. □

5.3.2 Handling Case 2

We maintain a generalized suffix tree and a generalized prefix tree of T_1 and T_2. Also, maintain our HIA data structure over the pair of trees ST_{T_2} and PT_{T_2} as in Sect. 5.2.1.

Moreover, we can compute (u_1^i, u_2^i, f_i, r_i) for $i = 1, 2, \ldots, k$ by processing the input set (of k changes) in $O(k \log n)$ time using standard techniques, where

1. f_i is the length of the longest substring of T_1^* that starts at position p_i
2. r_i is the length of the longest substring of T_1^* that ends at $p_i - 1$.
3. u_1^i is the lowest node in ST_{T_2}, s.t. $T_1[p_i, p_i + f_i - 1]$ is a prefix of $\mathsf{path}(u_1^i)$.
4. u_2^i is the lowest node in PT_{T_2}, s.t. $\overleftarrow{T_1[p_i - r_i, p_i - 1]}$ is a prefix of $\mathsf{path}(u_2^i)$.

Finally, for each p_i, we compute the (length of the) longest common substring of T_1^* and T_2 covering a position in $\{p_i - 1, p_i\}$ using an HIA query as in Sect. 5.2.2, and report the largest among.

In summary, the total space is dominated by the space of our HIA structure and the complexity of query time per change is $O(\log n)$ plus the time for an HIA query. This completes the proof of Theorem 4.

6 Summary and Open Problems

We have presented two new space-time trade-offs for the heaviest induced ancestors problem, which improves the previous result by Gagie et al. [22]. Open problems include even better space-time trade-offs and any non-trivial lower bounds. It is also interesting to investigate what other string problems can be solved efficiently using HIA framework.

Acknowledgements This research is supported in part by the U.S. National Science Foundation under the grants CCF-1703489 and CCF-2112643.

References

1. Abedin, P., Hooshmand, S., Ganguly, A., Thankachan, S.V.: The heaviest induced ancestors problem revisited. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, *LIPIcs*, vol. 105, pp. 20:1–20:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.20
2. Aluru, S.: Handbook of Computational Molecular Biology. Chapman and Hall/CRC (2005)
3. Amir, A., Boneh, I.: Locally maximal common factors as a tool for efficient dynamic string algorithms. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, *LIPIcs*, vol. 105, pp. 11:1–11:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.11
4. Amir, A., Boneh, I., Charalampopoulos, P., Kondratovsky, E.: Repetition detection in a dynamic string. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th Annual European Symposium on Algorithms, ESA 2019, September 9–11, 2019, Munich/Garching, Germany, *LIPIcs*, vol. 144, pp. 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.5
5. Amir, A., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P., Radoszewski, J.: Longest common factor after one edit operation. In: Fici, G., Sciortino, M., Venturini, R. (eds.) String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10508, pp. 14–26. Springer (2017). https://doi.org/10.1007/978-3-319-67428-5_2
6. Amir, A., Charalampopoulos, P., Pissis, S.P., Radoszewski, J.: Longest common substring made fully dynamic. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th Annual European Symposium on Algorithms, ESA 2019, September 9–11, 2019, Munich/Garching, Germany, *LIPIcs*, vol. 144, pp.

6:1–6:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.6

7. Amir, A., Charalampopoulos, P., Pissis, S.P., Radoszewski, J.: Dynamic and internal longest common substring. Algorithmica **82**(12), 3707–3743 (2020). https://doi.org/10.1007/s00453-020-00744-0
8. Amir, A., Kondratovsky, E.: Searching for a modified pattern in a changing text. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9–11, 2018, Proceedings, *Lecture Notes in Computer Science*, vol. 11147, pp. 241–253. Springer (2018). https://doi.org/10.1007/978-3-030-00479-8_20
9. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Panario, D., Viola, A. (eds.) LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings, Lecture Notes in Computer Science, vol. 1776, pp. 88–94. Springer (2000). https://doi.org/10.1007/10719839_9
10. Brodal, G.S., Jørgensen, A.G.: Data structures for range median queries. In: Dong, Y., Du, D., Ibarra, O.H. (eds.) Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16–18, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5878, pp. 822–831. Springer (2009). https://doi.org/10.1007/978-3-642-10631-6_83
11. Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13–15, 2011, pp. 1–10 (2011). https://doi.org/10.1145/1998196.1998198.
12. Charalampopoulos, P., Gawrychowski, P., Pokorski, K.: Dynamic longest common substring in polylogarithmic time. In: Czumaj, A., Dawar, A., Merelli, E. (eds.) 47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8–11, 2020, Saarbrücken, Germany (Virtual Conference), LIPIcs, vol. 168, pp. 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ICALP.2020.27
13. Chockalingam, S.P., Thankachan, S.V., Aluru, S.: A parallel algorithm for finding all pairs k-mismatch maximal common substrings. In: West, J., Pancake, C.M. (eds.) Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016, pp. 784–794. IEEE Computer Society (2016). https://doi.org/10.1109/SC.2016.66
14. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica **68**(3), 610–625 (2014). https://doi.org/10.1007/s00453-012-9683-x
15. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS ’97, Miami Beach, Florida, USA, October 19–22, 1997, pp. 137–143. IEEE Computer Society (1997). https://doi.org/10.1109/SFCS.1997.646102
16. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005). https://doi.org/10.1145/1082036.1082039
17. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011). https://doi.org/10.1137/090779759
18. Funakoshi, M., Mieno, T.: Minimal unique palindromic substrings after single-character substitution. In: Lecroq, T., Touzet, H. (eds.) String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings, Lecture Notes in Computer Science, vol. 12944, pp. 33–46. Springer (2021). https://doi.org/10.1007/978-3-030-86692-1_4
19. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest substring palindrome after edit. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, LIPIcs, vol. 105, pp. 12:1–12:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.12
20. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Faster queries for longest substring palindrome after block edit. In: Pisanti, N., Pissis, S.P. (eds.) 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18–20, 2019, Pisa, Italy, LIPIcs, vol. 128, pp. 27:1–27:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CPM.2019.27
21. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Computing longest palindromic substring after single-character or block-wise edits. Theor. Comput. Sci. **859**, 116–133 (2021). https://doi.org/10.1016/j.tcs.2021.01.014
22. Gagie, T., Gawrychowski, P., Nekrich, Y.: Heaviest induced ancestors and longest common substrings. In: Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013, Waterloo,

Ontario, Canada, August 8–10, 2013. Carleton University, Ottawa, Canada (2013). http://cccg.ca/proceedings/2013/papers/paper_29.pdf
23. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005). https://doi.org/10.1137/S0097539702402354
24. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
25. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984). https://doi.org/10.1137/0213024
26. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20–22, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3341, pp. 558–568. Springer (2004). https://doi.org/10.1007/978-3-540-30551-4_49
27. Nekrich, Y., Navarro, G.: Sorted range reporting. In: Fomin, F.V., Kaski, P. (eds.) Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4–6, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7357, pp. 271–282. Springer (2012). https://doi.org/10.1007/978-3-642-31155-0_24
28. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6–8, 2002, San Francisco, CA, USA., pp. 225–232 (2002). http://dl.acm.org/citation.cfm?id=545381.545410
29. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17–19, 2010, pp. 134–149 (2010). https://doi.org/10.1137/1.9781611973075.13
30. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. In: Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11–13, 1981, Milwaukee, Wisconsin, USA, pp. 114–122 (1981). https://doi.org/10.1145/800076.802464
31. Thankachan, S.V., Chockalingam, S.P., Aluru, S.: An efficient algorithm for finding all pairs k-mismatch maximal common substrings. In: Bourgeois, A.G., Skums, P., Wan, X., Zelikovsky, A. (eds.) Bioinformatics Research and Applications - 12th International Symposium, ISBRA 2016, Minsk, Belarus, June 5–8, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9683, pp. 3–14. Springer (2016). https://doi.org/10.1007/978-3-319-38782-6_1
32. Urabe, Y., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest lyndon substring after edit. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, LIPIcs, vol. 105, pp. 19:1–19:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.19
33. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15–17, 1973, pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13
34. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space theta(n). Inf. Process. Lett. **17**(2), 81–84 (1983). https://doi.org/10.1016/0020-0190(83)90075-3
35. Zhou, G.: Two-dimensional range successor in optimal time and almost linear space. Inf. Process. Lett. **116**(2), 171–174 (2016). https://doi.org/10.1016/j.ipl.2015.09.002
36. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714