The Heaviest Induced Ancestors Problem Revisited

Paniz Abedin
Dept. of Computer Science, University of Central Florida - Orlando, USA
paniz@cs.ucf.edu

Sahar Hooshmand
Dept. of Computer Science, University of Central Florida - Orlando, USA
sahar@cs.ucf.edu

Arnab Ganguly
Dept. of Computer Science, University of Wisconsin - Whitewater, USA
gangulya@uww.edu

Sharma V. Thankachan
Dept. of Computer Science, University of Central Florida - Orlando, USA
sharma.thankachan@ucf.edu

Abstract

We revisit the *heaviest induced ancestors* problem, which has several interesting applications in string matching. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two weighted trees, where the weight $\mathsf{W}(u)$ of a node u in either of the two trees is more than the weight of u's parent. Additionally, the leaves in both trees are labeled and the labeling of the leaves in $\mathcal{T}_2$ is a permutation of those in $\mathcal{T}_1$. A node $x \in \mathcal{T}_1$ and a node $y \in \mathcal{T}_2$ are induced, iff their subtree have at least one common leaf label. A heaviest induced ancestor query $\mathsf{HIA}(u_1, u_2)$ is: given a node $u_1 \in \mathcal{T}_1$ and a node $u_2 \in \mathcal{T}_2$, 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. Let n be the number of nodes in both trees combined and $\epsilon > 0$ be an arbitrarily small constant. Gagie et al. [CCCG' 13] introduced this problem and proposed three solutions with the following space-time trade-offs:

- an $O(n \log^2 n)$-word data structure with $O(\log n \log\log n)$ query time
- an $O(n \log n)$-word data structure with $O(\log^2 n)$ query time
- an $O(n)$-word data structure with $O(\log^{3+\epsilon} n)$ query time.

In this paper, we revisit this problem and present new data structures, with improved bounds. Our results are as follows.

- an $O(n \log n)$-word data structure with $O(\log n \log\log n)$ query time
- an $O(n)$-word data structure with $O\left(\frac{\log^2 n}{\log\log n}\right)$ query time.

As a corollary, we also improve the LZ compressed index of Gagie et al. [CCCG' 13] for answering longest common substring (LCS) queries. Additionally, we show that the LCS *after one edit* problem of size n [Amir et al., SPIRE' 17] can also be reduced to the heaviest induced ancestors problem over two trees of n nodes in total. This yields a straightforward improvement over its current solution of $O(n \log^3 n)$ space and $O(\log^3 n)$ query time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Data Structure, String Algorithms, Orthogonal Range Queries

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.20

Funding This research is supported in part by the U.S. National Science Foundation under the grant CCF-1703489.

© Paniz Abedin, Sahar Hooshmand, Arnab Ganguly and Sharma V. Thankachan;
licensed under Creative Commons License CC-BY
29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 20; pp. 20:1–20:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two weighted trees, having n_1 and n_2 nodes respectively. The weight of a node u in either of the two trees is given by $\mathsf{W}(u)$ and $\mathsf{W}(u) > \mathsf{W}(\mathsf{parent}(u))$, where $\mathsf{parent}(u)$ is the parent node of u. For simplicity, node u means the node with pre-order rank 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 $\mathcal{T}_1$. Two nodes, one each from $\mathcal{T}_1$ and $\mathcal{T}_2$, are *induced* if the leaves in the respective subtrees have at least one common label. For any two nodes u and v in a tree, the node v is an ancestor of u iff v is on the path from u to the root of the tree. Moreover, v is a proper ancestor u iff $u \neq v$. We revisit the following problem, which has several interesting applications in string matching.

▶ **Problem 1** (Heaviest Induced Ancestor Problem). 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).

Here and henceforth, ϵ is an arbitrarily small positive constant and $n = n_1 + n_2$ is the total number of nodes in the two trees. The model of computation is the standard Word RAM with word size $\Omega(\log n)$ bits. Gagie et al. [8] presented the following several results for the Heaviest Induced Ancestor problem.

- an $O(n \log^2 n)$-word index with $O(\log n \log\log n)$ query time
- an $O(n \log n)$-word index with $O(\log^2 n)$ query time
- an $O(n)$-word index with $O(\log^{3+\epsilon} n)$ query time.

Our contribution is summarized in the following Theorem.

▶ **Theorem 2.** *A heaviest induced ancestor query over two trees of n nodes in total can be answered*

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

1.1 Applications to String Matching

One motivation to study the heaviest induced ancestor problem is to design an LZ77 [20] compressed text index that can answer longest common substring $\mathsf{LCS}(S, P)$ queries efficiently. Formal definition is below.

▶ **Problem 3** (Longest Common Substring in LZ77 Compressed Strings [8]). *Given a string S of length N, whose LZ77 parsing contains n phrases, build a data structure that can efficiently report $\mathsf{LCS}(S, P)$, i.e., the longest common substring of S and P, where P is a query string of length $|P|$.*

If one were to forego the compression requirement, the problem can be easily solved by maintaining a suffix tree [17] of S in $O(N)$ words 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 [6, 9, 13], with a slight penalty in query time. However, for strings having a repetitive structure, LZ77-based compression techniques offer better space-efficiency than that obtained using FM-Index or Compressed Suffix Array.

Gagie et al. [8] showed that Problem 3 can be solved using an $O(n \log N + n \log^2 n)$-word index with very high probability in $O(|P| \log n \log\log n)$ query time. Alternatively, they also

presented an $O(n \log N)$-word index with $O(|P| \log^2 n)$ query time. Using Theorem 2 and the techniques in [8], we present an improved result for Problem 3 (see Theorem 4). We omit the details as they are immediate from the discussions in [8].

▶ **Theorem 4.** *Given a string S of length N, we can build an $O(n \log N)$-word index that reports $\mathsf{LCS}(S, P)$ in $O(|P| \log n \log \log n)$ time with very high probability, where n is the number of phrases in an LZ77 parsing of S and $|P|$ is the length of the input query string P.*

Another problem that we study is the recently introduced *longest common substring after one substitution* problem [1], defined as follows.

▶ **Problem 5** (Longest Common Substring after One Substitution [1]). *Given two strings X and Y of total length n over an alphabet set Σ, build a data structure that can efficiently report $\mathsf{LCS}_{(i,\alpha)}(X, Y)$, the length of the longest common substring of X_{new} and Y, where X_{new} is X after replacing its ith character by $\alpha \in \Sigma$.*

An $O(n|\Sigma|)$ space and $O(1)$ time solution is straightforward, but not efficient when $|\Sigma|$ is large. The solution by Amir et al. [1] takes $O(n \log^3 n)$ space and $O(\log^3 n)$ query time. Theorem 2 combined with other techniques implies an improved result to this problem, as summarized in the following theorem.

▶ **Theorem 6.** *Given two strings X and Y of total length n, we can build indexes with the following space-time trade-offs for reporting $\mathsf{LCS}_{(i,\alpha)}(X, Y)$*

1. *an $O(n \log n)$ space data structure with $O(\log n \log \log n)$ query time*
2. *an $O(n)$ space data structure with $O\left(\frac{\log^2 n}{\log \log n}\right)$ query time.*

Straightforward modifications to our approach leads to an index that can also support the case of single letter insertions or deletions in X, i.e., insert the character α after position i and delete the character at position i.

1.2 Map

In Section 2, we revisit some of the well-known data structures that have been used to arrive at out results. Section 3 presents an overview of our techniques, as an intermediate step into the final data structures. The final data structures for Theorem 2 are presented in Section 4. Section 5.1 is left to sketch our solution to Problem 5.

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 $\max\{q < p \mid q \in \mathcal{S}\}$. Similarly, 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 $O(n)$ words, we can answer such queries in $O(\log \log U)$ time [18].

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 denotes 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 [14]:

- $\mathsf{parent}_{\mathcal{T}}(u)$ = 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 [5]) 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 highest score among all nodes on the path from node a to node b. Cartesian tree based solutions exists for both problems. The space and query time are $2n + o(n)$ bits and $O(1)$, respectively [5, 7].

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 $O(n)$ word structure, we can answer orthogonal range counting queries in $O(\log / \log\log n)$ time [11], orthogonal range emptiness queries in $O(\log^{\epsilon} n)$ time [3], orthogonal range predecessor/successor queries in $O(\log^{\epsilon} n)$ time [12] and orthogonal range selection queries in $O(\log n / \log\log n)$ time [2, 4]. 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 [3, 19].

2.5 Heavy Path and Heavy Path Decomposition

We now define the heavy path decomposition [10, 15] of a 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 (ties are 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 7.** 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 of leaves (m). Thus, $n \le 4m - 2$. 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 Section 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, so as to support various range queries described in Section 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: and $O(n \log\log n)$ 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 8** (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.*

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 9** (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 10** (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.*

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.

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

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

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,

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

To evaluate the above equation efficiently, we explore the heavy path decomposition of $\mathcal{T}_2$.

▶ **Definition 11** (Special Nodes)**.** For each light node in $w \in \mathcal{T}_2$, 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 $w_1 = \mathsf{root}_2, w_2, \dots, w_k$ 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 $t_1, t_2, \dots, t_k$ 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, $\alpha_1, \alpha_2, \dots, \alpha_k$ and $\beta_1, \beta_2, \dots, \beta_k$ are nodes in $\mathsf{Path}(\mathsf{root}_1, u_1, \mathcal{T}_1)$, such that for $h = 1, 2, \dots, k$, $\alpha_h = \mathsf{partner}(t_h/u_1)$ 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 Figure 1 for an illustration. We now present several lemmas, which forms the basis of our solution.

▶ **Lemma 12.** *The node* $u_1^* \in \mathsf{Path}(\alpha_f, \beta_f, \mathcal{T}_1)$.

Proof. We prove this via proof by contradiction arguments.

- Suppose u_1^* is a proper ancestor of α_f. Then, α_f are t_f 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.

This completes the proof. ◀

▶ **Lemma 13.** *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.

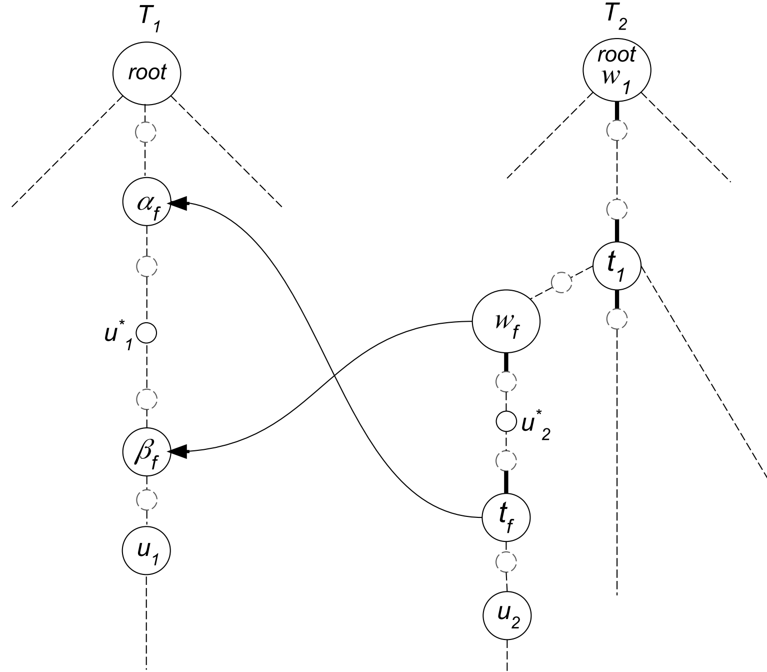

Figure 1 We refer to Section 3.2 for the description of this figure.

- 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 14.** *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 14. ◀

▶ **Corollary 15.** *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 16.** *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 Lemma 12, Lemma 13, Lemma 14 and Corollary 15. ◀

▶ **Lemma 17.** *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,

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

Proof. Since f is unknown, we invoke Lemma 14 for $f = 1, 2, 3, \ldots, k \leq \log n$. ◀

Next, we show how to transform the result in Lemma 17 into an efficient data structure.

4 Our Data Structures

We start by defining a crucial component of our solution.

▶ **Definition 18** (Induced Subtree)**.** The induced subtree of $\mathcal{T}_1(w)$ 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 19.** *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 [18] 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 $\leq x'$. Therefore, an $\mathsf{invMap}_w(\cdot)$ can be reduced to a predecessor search and answered in $O(\log\log n)$ time. ◀

▶ **Lemma 20.** *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 $\mathsf{lca}_{\mathcal{T}_1}(\mathsf{Map}_w(\ell_L), \mathsf{Map}_w(\ell_R))$, 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 21.** *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 $\mathcal{T}_1(w)$, we maintain the Cartesian tree for answering path maximum query (refer to Section 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 7), equivalently $O(n)$ 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 fully-functional succinct representation of their topologies (refer to Section 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 Section 2.4) and the structures described in Lemma 19 and Lemma 21. Thus, the total space is $O(n \log n)$ words.

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

1. Find w_h and t_h for $h = 1, 2, \ldots, k \leq \log n$.
2. Find α_h and β_h for $h = 1, 2, \ldots, k \leq \log n$.
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 $h = 1, 2, \ldots, k \leq \log n$.
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 17. 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, 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 every thing 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 counter parts. 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 20 instead of the structure in Lemma 19. Thus, the total space is $O(n)$ words.

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 $O(\log^2 n/\log\log n)$.

5 Applications

5.1 Longest Common Substring after One Substitution

Let X and Y be two strings of total length n over an alphabet set Σ. Define, $\mathsf{LCS}(X,Y)$ as the length of the longest common substring of X and Y and $\mathsf{LCS}_{(i,\alpha)}(X,Y)$ as the length of the longest common substring of X_{new} and Y, where X_{new} is X after replacing its ith character by $\alpha \in \Sigma$. Our task is to build a data structure for X and Y, so that $\mathsf{LCS}_{(i,\alpha)}(X,Y)$ for any input (i,α) can be reported efficiently.

5.1.1 The Data Structure

Let $\mathsf{LCS}_{(i,\alpha)}$ be $X[l,r]$. As observed by Amir et al. [1], two possible scenarios are: $i \notin [l,r]$ and $i \in [l,r]$. We handle each of these scenarios separately, i.e., we find the new longest common substring (with the character at position i replaced by α) with position i (a) not covered and (b) covered, and choose the longest. To obtain (a), simply store an array $A[1,|X|]$, where

$$A[i] = \max\{\mathsf{LCS}(Y, X[1\dots(i-1)]), \mathsf{LCS}(Y, X[(i+1)\dots|X|]\}$$

For case (b), we maintain the following structures.

1. A generalized suffix tree [17] of X and Y (GST), which is a compact trie over all suffixes of X and Y, after appending each string from X (resp., Y) with a unique symbol $\$_1$ (resp., $\$_2$).
2. A compact trie of reverse of all prefixes of X and Y (GPT), after appending each string from X (resp., Y) with a unique symbol $\$_1$ (resp., $\$_2$).
3. For each character $\alpha \in \Sigma$,
   - a compact trie $\mathcal{T}_\alpha$ of all strings in $\{Y[(i+1)\dots] \mid Y[i] = \alpha\}$ after appending each suffix with $\$_2$. We label $Y[(i+1)\dots]$ with i.
   - Another compact trie $\mathcal{T}'_\alpha$ of all strings in $\{\overleftarrow{Y[1\dots(i-1)]} \mid Y[i] = \alpha\}$. Here $\overleftarrow{Y[1\dots(i-1)]}$ is the reverse of $Y[1\dots(i-1)]$ and we label it with i.

- The data structure for HIA queries on $(\mathcal{T}_\alpha, \mathcal{T}'_\alpha)$. Here the weight of a node is its string-depth. Therefore, we can easily generalize our solution to the HIA problem to the case where the input (u_1, u_2) is such that u_1 and u_2 are not necessarily nodes, but locations on edges.

The total space is proportional to the size of an HIA structure over an input of size n.

5.1.2 Processing a query (i, α)

Get the LCS not covering i in constant time from the array A. For LCS covering i, do the following steps.

- Let ℓ_p be the leaf in GST corresponding to the suffix $X[(i+1)\dots]$. Find the lowest ancestor u of ℓ_p with at least one leaf corresponding to a suffix of Y (say $Y[a\dots]$) in its subtree.
- Let ℓ_q be the leaf in GPT corresponding to the reverse of the prefix $X[1\dots(i-1)]$. Find the lowest ancestor v of ℓ_q with at least one leaf corresponding to a reverse of a prefix of Y (say $Y[\dots b]$) in its subtree.
- Issue an HIA query $\mathsf{HIA}(x, y)$ on $(\mathcal{T}_\alpha, \mathcal{T}'_\alpha)$, where
  1. x is the location in $\mathcal{T}_\alpha$ on the path of the leaf corresponding to $Y[a\dots]$ at a distance of string-depth of u from the root.
  2. y is the location in $\mathcal{T}'_\alpha$ on the path of the leaf corresponding to $\overleftarrow{Y[\dots b]}$ at a distance of string-depth of v from the root.

  Let (x^*, y^*) be the output. Then, $\mathsf{LCS}_{(i,\alpha)}(X, Y)$ covering position i is $W(x^*) + W(y^*)$.

Therefore, final $\mathsf{LCS}_{(i,\alpha)}(X, Y)$ is $\max\{A[i], W(x^*)+W(y^*)\}$. Steps 1 and 2 can be performed in $O(\log n)$ time binary searches. Therefore, total time is dominated by the time for an HIA query. The correctness can be easily verified.

5.2 All-Pairs Longest Common Substring Problem

Let $\mathcal{S} = \{S_1, S_2, S_3, \dots S_n\}$ be a collection of n strings and let L be the length of the longest string in $\mathcal{S}$. We consider the problem of finding $\mathsf{LCS}(S_i, S_j)$ for all (i, j) pairs. This problem can be easily solved in $O(n^2 L)$ time. However, it is also possible to obtain a conditional lower bound of $\tilde{\Omega}(n^2 L)$ via a reduction from the boolean matrix multiplication [16]. To this end, we remark that the following run-time is possible with the aid of HIA framework.

$$\tilde{O}\left(nL + \sum_i \sum_{j<i} \frac{L}{\mathsf{LCS}(S_i, S_j)}\right)$$

We defer details to the full version of this paper.

References

1 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2017. doi:10.1007/978-3-319-67428-5_2.

2 Gerth Stølting Brodal and Allan Grønlund Jørgensen. Data structures for range median queries. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 822–831. Springer, 2009. doi:10.1007/978-3-642-10631-6_83.

3 Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

4 Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 241–251. SIAM, 2013. doi:10.1137/1.9781611973105.18.

5 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.

6 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

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

8 Travis Gagie, Pawel Gawrychowski, and Yakov Nekrich. 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. URL: http://cccg.ca/proceedings/2013/papers/paper_29.pdf.

9 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

11 Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004. doi:10.1007/978-3-540-30551-4_49.

12 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In Fedor V. Fomin and Petteri Kaski, editors, *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings*, volume 7357 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2012. doi:10.1007/978-3-642-31155-0_24.

13 Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 225–232. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545410.

14 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149. SIAM, 2010. doi:10.1137/1.9781611973075.13.

15 Daniel Dominic Sleator and Robert Endre Tarjan. 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*, pages 114–122. ACM, 1981. doi:10.1145/800076.802464.

16 Sharma V. Thankachan, Sriram P. Chockalingam, and Srinivas Aluru. An efficient algorithm for finding all pairs k-mismatch maximal common substrings. In *Bioinformatics Research and Applications - 12th International Symposium, ISBRA 2016, Minsk, Belarus, June 5-8, 2016, Proceedings*, pages 3–14, 2016.

17 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

18 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

19 Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016. doi:10.1016/j.ipl.2015.09.002.

20 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.