

Approximate Suffix-Prefix Dictionary Queries

Wiktor Zuba  

CWI, Amsterdam, The Netherlands



Grigorios Loukides  

King's College London, UK

Solon P. Pissis  

CWI, Amsterdam, The Netherlands

Vrije Universiteit, Amsterdam, The Netherlands

Sharma V. Thankachan  

North Carolina State University, Raleigh, NC, USA

Abstract

In the *all-pairs suffix-prefix* (APSP) problem [Gusfield et al., *Inf. Process. Lett.* 1992], we are given a dictionary R of r strings, S_1, \dots, S_r , of total length n , and we are asked to find the length $\text{SPL}_{i,j}$ of the *longest* string that is both a suffix of S_i and a prefix of S_j , for all $i, j \in [1..r]$. APSP is a classic problem in string algorithms with applications in bioinformatics, especially in sequence assembly. Since $r = |R|$ is typically very large in real-world applications, considering all r^2 pairs of strings explicitly is prohibitive. This is when the data structure variant of APSP makes sense; in the same spirit as distance oracles computing shortest paths between any two vertices given online.

We show how to quickly locate *k-approximate matches* (under the Hamming or the edit distance) in R using a version of the *k-errata tree* [Cole et al., STOC 2004] that we introduce. Let $\text{SPL}_{i,j}^k$ be the length of the longest suffix of S_i that is at distance at most k from a prefix of S_j . In particular, for any $k = \mathcal{O}(1)$, we show an $\mathcal{O}(n \log^k n)$ -sized data structure to support the following queries:

- **One-to-One^k(i, j)**: output $\text{SPL}_{i,j}^k$ in $\mathcal{O}(\log^k n \log \log n)$ time.
- **Report^k(i, d)**: output all $j \in [1..r]$, such that $\text{SPL}_{i,j}^k \geq d$, in $\mathcal{O}(\log^k n (\log n / \log \log n + \text{output}))$ time, where **output** denotes the size of the output.

In fact, our algorithms work for any value of k not just for $k = \mathcal{O}(1)$, but the formulas bounding the complexities get much more complicated for larger values of k .

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases all-pairs suffix-prefix, suffix-prefix queries, suffix tree, *k-errata tree*

Digital Object Identifier 10.4230/LIPIcs.MFCS.2024.85

Funding *Wiktor Zuba*: Received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement Grant Agreement No 101034253.

Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

Sharma V. Thankachan: Supported by the U.S. National Science Foundation (NSF) grant CCF-2316691.

1 Introduction

Given a dictionary R of r strings, S_1, \dots, S_r , of total length n , the *all-pairs suffix-prefix* (APSP) problem asks us to find, for each string S_i , $i \in [1..r]$, its longest suffix that is a prefix of string S_j , for all $j \neq i$, $j \in [1..r]$. APSP is a classic problem in string algorithms [18] with numerous applications in sequence assembly [18, 26, 31, 7, 10]. Gusfield et al. [19] presented



© Wiktor Zuba, Grigorios Loukides, Solon P. Pissis, and Sharma V. Thankachan; licensed under Creative Commons License CC-BY 4.0

49th International Symposium on Mathematical Foundations of Computer Science (MFCS 2024).

Editors: Rastislav Královic and Antonín Kučera; Article No. 85; pp. 85:1–85:18



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an algorithm running in the optimal $\mathcal{O}(n + r^2)$ time for solving APSP, assuming all strings in R are over an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$. Many other optimal algorithms exist for APSP aiming at a smaller memory footprint [27, 36, 24].

DNA sequences (fragments) read by sequencing machines are typically assembled by first computing the maximal overlap between every pair of sequences. The size r of dictionary R in this case is very large. In particular, r^2 usually dominates n , and thus the r^2 factor is the bottleneck both in the time and the space complexity of assembly applications. For instance, in typical benchmark datasets for genome assembly using short DNA fragments, r is in the order of 10^6 to 10^8 and n is in the order of 10^8 to 10^{10} . Hence r^2 dominates n significantly.

In [25], we initiated a study on several data structure variants of APSP. Let $\text{SPL}_{i,j}$, for any $i, j \in [1..r]$, denote the length of the *longest* suffix of S_i that is a prefix of S_j . We proposed $\mathcal{O}(n)$ -sized data structures to support the following queries:

- **One-to-One**(i, j): output $\text{SPL}_{i,j}$ in $\mathcal{O}(\log \log r)$ time.
- **Report**(i, d): output all distinct $j \in [1..r]$ such that $\text{SPL}_{i,j} \geq d$, for some $d \in \mathbb{N}$, in $\mathcal{O}(\log n / \log \log n + \text{output})$ time, where **output** denotes the size of the output.

We also proposed other $\mathcal{O}(n)$ -sized data structures for answering **One-to-All**(i), **Count**(i, ℓ), and **Top**(i, K) queries efficiently (see [25] for the precise definitions of these queries).

This framework is interesting both from a practical and from a theoretical perspective. By being able to answer different types of such queries efficiently, practitioners may be able to design alternative algorithms that *avoid the r^2 factor* in their time or space complexity; see the recent work of Talera et al. [33], which investigates practical aspects of our framework. Furthermore, the underlying data structure problems are also appealing from a theoretical perspective: (i) they are analogous to *distance oracles* for networks [35, 28, 14, 13, 11, 17]; and (ii) they are special types of *internal pattern matching* data structures [21, 20, 2, 1, 12, 4].

In this work, we make the next natural yet *challenging* step, that is, extend the framework we introduced in [25] to support *k-approximate suffix-prefix* queries. Indeed, these are the most useful queries in real-world applications. Approximate variants of APSP, under the Hamming or edit distance, have been considered in the literature. Barton et al. [5] showed how to solve the problem for two strings ($r = 2$), under Hamming or edit distance k , in $\mathcal{O}(nk)$ time. Thankachan et al. [34] showed how to solve the problem in $\mathcal{O}((n+r^2) \log^k n)$ time under Hamming or edit distance k , for any r and $k = \mathcal{O}(1)$. Many other proposed methods aimed at practical efficiency and are thus based on filtering strategies [29, 37, 22]. They typically use a two-step approach: candidate regions are identified that potentially correspond to sought matches; and those candidates are checked to actually verify the desired matching condition. These approaches do not yield strong theoretical time bounds but they are very efficient in practice. In any case, none of the proposed methods addresses the data structure variant.

Problems Statements. Let us start with the following basic definition.

► **Definition 1.** For any given set $R = \{S_1, \dots, S_r\}$ of r strings, we define the longest suffix-prefix of S_i and S_j with k -errors as the longest suffix U_i of S_i that is at Hamming/edit distance at most k from some prefix V_j of S_j . We denote the length of U_i by $\text{SPL}_{i,j}^k$.

By fixing k *a priori*, we want to efficiently compute the values $\text{SPL}_{i,j}^k$ for certain (S_i, S_j) pairs given in an online fashion. In particular, we consider the following queries:

- **One-to-One** ^{k} (i, j): output $\text{SPL}_{i,j}^k$.
- **Report** ^{k} (i, d): output all distinct $j \in [1..r]$ such that $\text{SPL}_{i,j}^k \geq d$, for some $d \in \mathbb{N}$.

The offline version (that is, the k -approximate variant of APSP), which we denote by APSP^k , is defined as follows: output $\text{SPL}_{i,j}^k$, for all $1 \leq i, j \leq r$. Recall that APSP^k has already been solved by Thankachan et al. [34] in $\mathcal{O}((n+r^2)\log^k n)$ time for any $k = \mathcal{O}(1)$.

► **Observation 2.** *For Hamming distance we always have that $|U_i| = \text{SPL}_{i,j}^k = |V_i|$. However, for edit distance it does not need to be the case: we cannot always maximize the lengths of both suffix and prefix simultaneously. For $S_i = \mathbf{xx} \cdots \mathbf{xabcd}$, $S_j = \mathbf{bcd} \mathbf{e} \mathbf{y} \cdots \mathbf{y}$, we have $\text{SPL}_{i,j}^1 = 4$ as \mathbf{abcd} is at edit distance 1 from \mathbf{bcd} , but \mathbf{bcd} is at edit distance 1 from \mathbf{bcde} , and \mathbf{abcd} is at edit distance 2 from \mathbf{bcde} . In this sense, edit distance is a bit more complicated.*

Results and Paper Organization. We assume the standard word-RAM model of computations with word size $w = \Omega(\log n)$, where n is the input size. We present our results below for any $k = \mathcal{O}(1)$; our algorithms work for any arbitrary value of k but the formulas bounding the complexities get much more complicated for larger values of k . For the exact time complexities we defer the reader to the actual theorem statements (Theorems 13 and 14).

Query	Space (words)	Query time	Note
One-to-One ^k (i, j)	$\mathcal{O}(n \log^k n)$	$\mathcal{O}(\log^k n \log \log n)$	Theorem 13
Report ^k (i, d)	$\mathcal{O}(n \log^k n)$	$\mathcal{O}(\log^k n (\log n / \log \log n + \text{output}))$	Theorem 14

Let us introduce the central notion of *extension-prefix pair* for any two strings X and Y .

► **Definition 3.** *A pair of strings (Y, X) is an extension-prefix pair, if X is a prefix of Y ; and hence Y is called an extension of X . A pair of strings (Y, X) is a k -approximate extension-prefix pair, if X is at (Hamming or edit) distance at most k from some prefix of Y .*

Section 2 introduces the necessary definitions and notation as well as a few observations. In Section 3, we introduce a version of the k -errata tree [15] and show its efficient construction as well as some of its properties that are crucial to arrive at our main results. Specifically in Section 3.1, we show the construction for edit distance, and then also state the changes for the (easier) case of the Hamming distance. In Section 3.2, we show the key combinatorial property of the extension-prefix pairs (Lemma 5) in our k -errata tree. Section 3.3 shows an upper bound on the size of the k -errata tree introduced here for the interesting case where $k = \mathcal{O}(\frac{\log n}{\log \log n})$, which is then used to bound the running time and space of the algorithms – we are not interested in $k = \omega(\frac{\log n}{\log \log n})$ as in this case we have $\text{poly}(n) = \mathcal{O}(\log^k n)$, and hence trivial algorithms become (at least) as efficient as our approach. Section 3.4 refines the combinatorial Lemma 5 to its algorithmic applications (Corollaries 10–12). For the sake of an application, consider that we have two dictionaries D_1 and D_2 . We concentrate not only on pairs of strings which are at distance at most k , but globally, for any string $Y \in D_2$, we show how to compute its longest prefix which is at distance at most k from any other string $X \in D_1$, and further show how to compute the set of lengths of all such prefixes.

Finally, in Sections 4.1 and 4.2, we apply our techniques to construct the data structures for answering One-to-One^k and Report^k queries, respectively. Full detailed proofs omitted from the main text are included in Appendix A.

2 Preliminaries

We consider strings over an integer alphabet $\Sigma = [1.. \sigma]$ of size σ . The elements of Σ are called *letters*. A *string* $X = X[1.. n]$ is a sequence of letters from Σ ; we denote by $|X| = n$ the *length* of X . The fragment $X[i.. j]$ of X is an *occurrence* of the underlying *substring* $S = X[i] \cdots X[j]$ occurring at *position* i in X . A *prefix* of X is a substring of X of the form

$X[1..j]$ and a *suffix* of X is a substring of X of the form $X[i..n]$. The *reverse* of X is the string $X^R = X[n] \cdots X[1]$. By $\text{LCP}(X, Y)$, we denote the length of the longest common prefix of strings X and Y . The *Hamming distance* of two equal-length strings is the number of positions where the strings differ. The *edit distance* (or *Levenshtein distance*) of two strings X and Y is the minimum number of edit operations required to transform X into Y . Here, by *edit operation* we mean an insertion, a substitution, or a deletion of a single letter. A set D of strings is called a *dictionary*.

The following edit-distance property (Lemma 4) underlies most string algorithms on edit distance; e.g., [23]. We provide a proof of this property in Appendix A for completeness.

► **Lemma 4.** *For any two strings X and Y , there exists a smallest sequence of edit operations changing X to Y satisfying recursively that the first operation occurs at position $\text{LCP}(X, Y) + 1$. By recursively, we mean that after applying the first operation on X to obtain X' , the leftmost operation to get from X' to Y occurs at position $\text{LCP}(X', Y) + 1 \geq \text{LCP}(X, Y) + 1$.*

Let D be a dictionary of $n = |D|$ strings. A node in the trie of D is called *branching* if it has at least two children and *terminal* if it represents a string in D . The *compacted trie* T of D is obtained from the underlying trie by removing all nodes except the root node, the branching and the terminal nodes. The removed nodes are called *implicit* while the remaining ones are called *explicit*. Each terminal node corresponding to a string X from D is labeled with the identifier of X : a pointer to X in D . Edge labels are stored as pointers to fragments of strings in D and so the compacted trie takes $|T| = \mathcal{O}(n)$ extra space.

Our algorithms use a version of the famous k -errata tree of Cole et al. [15]. The tree nodes store information about strings and a representation of edit operations applied on them. In order to efficiently operate on those edit operations we define their representation here as an abstract structure and describe the operations that are performed on such structures later.

For convenience we represent multisets of edit operations by lists sorted non-decreasingly. A Hamming distance list consists of up to k elements, where every element is denoted by l_S , for $l \in [1..n]$. An edit distance list consists of up to k elements, where every element is denoted by l_E for $l \in [1..n]$, $E \in \{S, I, D\}$; elements are first sorted by l , then by $D < S < I$, and l_D elements may repeat (denoting that several letter deletions occur at the same position l). A single list does not contain both l_S and l_I , or multiple such elements, for the same l . By $|\ell|$ we denote the number of elements in list ℓ (counting the multiplicities). For any two lists ℓ_1, ℓ_2 , by $\ell_1 \subseteq \ell_2$, we denote that every element of ℓ_1 appears in ℓ_2 and the multiplicity of any element in ℓ_1 does not exceed its multiplicity in ℓ_2 . By $\ell_1 \cup \ell_2$ we denote the union of the two lists; for each element of ℓ_1 or ℓ_2 its multiplicity is equal to its maximum multiplicity in ℓ_1 and ℓ_2 . By $\max^+(\ell)$ we denote the smallest element that can be inserted at the end of list ℓ without breaking the specified condition of the list being non-decreasing; that is, the maximum element of the list if this element may appear multiple times and the smallest larger element if the currently maximal element cannot appear multiple times ($\max^+(\ell) = l_D$ when the last element of ℓ is equal to $l_D, (l-1)_S$ or $(l-1)_I$, $\max^+(\emptyset) = l_D$). By ℓ^d we denote the prefix of list ℓ containing only the elements l_E such that $l \leq d$, and by $\text{surpl}(\ell)$ we denote the surplus of l_D operations over l_I operations in ℓ , that is $|\{l_D \in \ell\}| - |\{l_I \in \ell\}|$; i.e., by how many letters the length of the string decreases after applying the operations from ℓ .

3 Finding Approximate Extension-Prefix Pairs using the k -Errata Tree

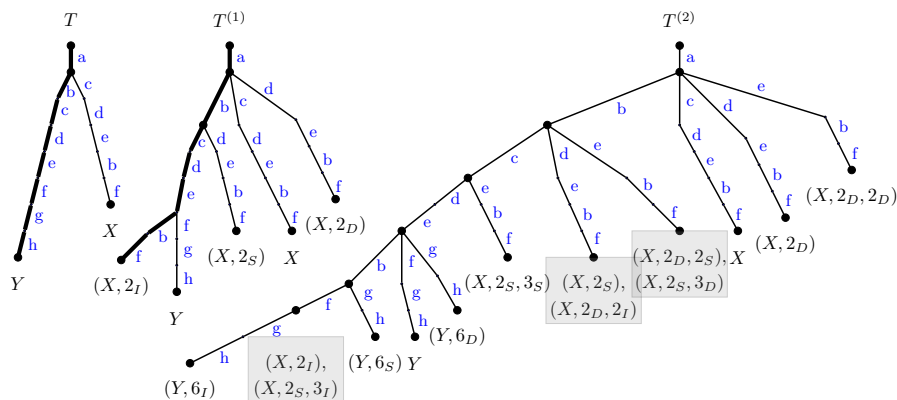
For any dictionary D , extension-prefix pairs can be found by using a trie of D . Every such pair corresponds to a descendant-ancestor node pair in the trie. By using this property, all such pairs can be listed in the optimal time. In this section, we show that for k -approximate

extension-prefix pairs, a similar property holds for a version of the k -errata tree of the trie of D . We start by showing the tree construction, then we show the counterpart of this property, and we end by showing how to use the tree for finding k -approximate extension-prefix pairs.

The Hamming distance can be treated as a special case of edit distance, where only substitution operations are allowed; we thus focus on the edit distance construction, and then comment on the differences that should be made to obtain the tree for Hamming distance. Let D be a dictionary of $n = |D|$ strings and T be the compacted trie of D . In the complexity analysis of our construction, we assume that $k \leq \frac{\log n}{2 \log \log n}$ [15]. (The construction itself and the combinatorial lemmas *do not assume any bound on k* .)

We compute the *heavy-light decomposition* of T in $\mathcal{O}(n)$ time [32]. Let $s(w)$ denote the number of node labels in the subtree of T rooted at w (these labels identify the full strings from D). We call an edge (u, v) of T *heavy* if $s(v)$ is maximal among every edge originating from u (breaking ties arbitrarily). All other edges are called *light*. We call a node that is reached from its parent through a heavy edge *heavy*; otherwise, it is called *light*. The heavy path of T is the path that starts at the root of T and at each node on the path descends to the heavy child as defined above. The heavy path decomposition is then defined recursively: it is a union of the heavy path of T and the heavy path decompositions of the off-path subtrees of the heavy path. The crucial well-known property of this decomposition is that every root to node path in T passes through $\mathcal{O}(\log n)$ light nodes. Our k -errata tree construction actually refers to the heavy-light decomposition of the uncompactd trie of D – in particular to the direct children of the branching nodes even if those are implicit. The decompositions of both the uncompactd and the compactd versions of T are in a bijection: a child of a branching node of the uncompactd version is light if and only if its closest explicit descendant is light in the compactd version. Therefore we refer to the nodes of the uncompactd version of T , and access all children of the branching nodes in time proportional to the size of its compactd version, obtaining the compactd version of the errata tree in the process.

3.1 The k -Errata Tree Construction



■ **Figure 1** Example of an edit distance 1- and 2-errata tree for trie T of strings $\{Y = abcdefgh, X = acdebf\}$. The rectangles mark the nodes with multiple distinct labels. Notice that a node with label $(Y, 6_I)$ has an ancestor with two labels (X, ℓ) , since $|\{2_I\} \cup \{6_I\}| \leq 2$, X is at edit distance at most 2 from some prefix of Y . Label $(X, 2_S, 3_I)$ does not provide such result as $|\{2_S, 3_I\} \cup \{6_I\}| = 3 > 2$.

We denote the k -errata tree of T by $T^{(k)}$ and construct it as a 1-errata of $T^{(k-1)}$, where $T^{(0)} = T$. Before constructing the 1-errata tree of T , we compute the heavy-light decomposition of T . For every *light node* v at string depth $l = d(v)$ in T that is reached from its parent through an edge labeled with $a \in \Sigma$, we create three copies of its subtree (inspect Figure 1):

- In one copy, corresponding to a substitution, we change the labels of the subtree nodes from X to (X, l_S) and merge it with the subtree of its heavy sibling.
- In one copy, corresponding to a deletion, we change the labels of the subtree nodes from X to (X, l_D) and merge it with the subtree of its parent.
- In one copy, corresponding to an insertion, we change the labels of the subtree nodes from X to (X, l_I) and merge it with the child of its heavy sibling reached through the edge labeled with letter a (or create such a child if it does not exist).

A node of 1-errata with label: (X, l_S) can be reached by replacing the l -th position of X by the heavy letter (the letter used to reach the heavy child from the node reached by spelling $X[1..l-1]$ in T); (X, l_D) can be reached by deleting the letter at the l -th position in X ; and (X, l_I) can be reached by inserting that heavy letter before the l -th position of X . It should be clear that in the Hamming distance case only the first copy is required.

For convenience, we represent all the added parts of labels collectively with lists of edit operations (see Section 2). To keep the data structure simpler, we avoid introducing redundant labels that would later make proofs and implementations more complicated (in particular an insertion or deletion would change the positions of the edits to the right). Specifically, we produce a copy of a label (X, ℓ) only if the added element l_E is greater or equal to $\max^+(\ell)$. This means that the lists are sorted non-decreasingly. We do not miss any significant sequence of edit operations due to Lemma 4 as shown in the proof of Lemma 5.

Since we may possibly delete a few next letters from the same position (after removing a single letter the next one takes its position) we have to allow elements l_D to appear in a list multiple times. In case of insertions or substitutions this is not needed (and we can prioritize deletions over other operations as there is no reason to first modify a letter and then delete it), hence we do not allow two elements l_I or l_S (or l_I and l_S) for the same $l \in [1..n]$. As such, the order of elements $l_E, l \in [1..n], E \in \{S, D, I\}$, is defined by first comparing l , and then $D < S < I$, thus the definition of $\max^+(\ell)$ (see Section 2).

In our construction of trie T , unlike the one used in the classic construction [15] where D is made prefix-free, two terminal nodes can lie on the same heavy path (for an extension-prefix pair (Y, X) for $X, Y \in D$ the node representing X may in fact be an internal labeled node that lies on a heavy path). This does not allow for checking if Y is a k -approximate prefix of X in the same way since it is already an extension of X (in case of edit distance one can always check if the difference of lengths is at most k , and in case of Hamming distance simply respond negatively if $|X| < |Y|$); for example, when $D = \{X = abc, Y = abcde\}$, $T^{(2)} = T$, there is a single heavy path since T has no light nodes. To mitigate this problem and still have the nice properties of our construction (ancestor-descendant relations of the labeled nodes), for a labeled node u with label (X, ℓ) that has (non-trivial) descendants, we also create a copy with label $(X, \ell \cup (d(u) + 1)_I)$ and add it to its heavy child (insertion after the last position of X). Substitution and deletion after the end of a string clearly do not make sense, thus an insertion operation is the only one allowed. This change does not influence our bounds on the k -errata tree size as the total size of all such subtrees is the number of the labeled nodes; we change this from $\mathcal{O}(|T| \log^k |T|)$ to $\mathcal{O}(|T|(\log |T| + 1)^k)$.

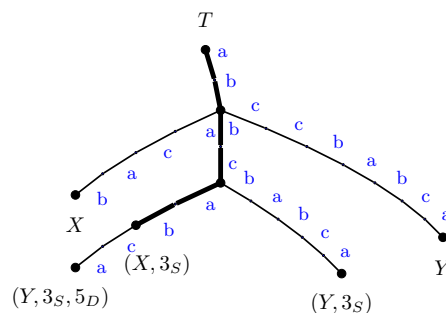
We remark, that our structure differs substantially from the original k -errata from [15]. In particular, we do not use extra letters outside of the alphabet to mark the edges transitioning from the original tree to a copied part, and instead actually merge the nodes corresponding to the same string. Thanks to this change we do not need to use the extra copies called group trees, which makes the search algorithm simpler in the case of our structure (the first place where the search can diverge from the heavy path is the place where the pattern actually diverges from the heavy path) and running in asymptotically the same time (up to an $f(k)$

factor for some f and $k = \mathcal{O}(\frac{\log n}{\log \log n})$. We do not describe the algorithm in detail however, since in this paper we are only interested in internal queries (the data structure is constructed for the fixed dictionary D and no other string is ever queried). All those simplifications pose the cost of dealing with the lists of edit operations, which are not present in the data structure from [15], but we show how to do that efficiently in the remainder of this section.

3.2 Extension-Prefix Pairs in a k -Errata Tree

We now prove the key combinatorial property of the described k -errata tree of dictionary D .

► **Lemma 5.** *For any two elements $X, Y \in D$, X is at edit (resp. Hamming) distance at most k from some prefix of Y if and only if there exist two nodes, u and its descendant v , in $T^{(k)}$ with labels (X, ℓ_1) and (Y, ℓ_2) respectively, such that $|\ell_1 \cup \ell_2| \leq k$, where $T^{(k)}$ is the edit (resp. Hamming) distance k -errata tree of D .*



■ **Figure 2** Illustration of a part of the edit distance 2-errata tree T for dictionary D containing $X = \text{abacab}$ and $Y = \text{abccbabc}$ with the choice of heavy paths depicted with thicker lines. X is at edit distance 2 from $Y' = \text{abccbabc}$, which is a prefix of Y ; the operations used to transform X into Y' are *substitution* at position 3 and *insertion* at position 5. The lowest common ancestor of nodes with labels X and Y is at string depth 2 (ab); as shown in the proof of Lemma 5 there exists a child of this node that is a common ancestor of nodes with labels (X, ℓ_1) and (Y, ℓ_2) such that $\ell_1 \cup \ell_2 = \{3_S\}$, and a descendant of this child with label (X, ℓ_1) that is an ancestor of a node with label (Y, ℓ_2) , such that $|\ell_1 \cup \ell_2| = 2$. Conversely, since there exists such an ancestor-descendant pair of nodes one can compute the list of at most 2 edit operations that transform X into Y' .

Sketch of Proof. The proof of the forward implication follows a path down the k -errata tree – one starts with empty lists ℓ_1 and ℓ_2 . Each time an edit operation is applied to X one adds an element to ℓ_1 or ℓ_2 or both (depending on the edit operation and the location of the heavy child) and finds a descendant of the current node that is a common ancestor of nodes with labels (X, ℓ_1) and (Y, ℓ_2) . The backward implication can be obtained by reversing that approach and producing a list of at most k edit operations that transform X into a prefix of Y (see Appendix A for the full proof and Figure 2 for an example of the construction). ◀

The full proof shows that, if a prefix Y' of Y is at distance at most k from X , then there exists a pair of nodes u and v with labels (X, ℓ_1) and (Y, ℓ_2) , respectively, in an ancestor-descendant relationship, such that $f(d(u)) - k + |\ell_1 \cup \ell_2| \leq |Y'| \leq f(d(u))$, where $f(l) = l + \text{surpl}(\ell_2^l)$ is the actual length of the prefix of Y represented by the ancestor of v at depth l . In particular, the longest prefix is represented by either such a pair for $u = v$ (the whole Y is at edit distance at most k from X), or a pair where $|\ell_1 \cup \ell_2| = k$, and hence $\ell_2 = \ell_2^{d(u)}$.

This property and the monotonicity of $f(l)$ for a single label (Y, ℓ_2) of node v imply that to find the longest prefix of Y at distance at most k from X it is enough to focus on the lowest ancestor u of v with label (X, ℓ_1) such that $|\ell_1 \cup \ell_2| \leq k$. In fact, if $\ell_2 \neq \ell_2^{d(u)}$, then both $u \neq v$ and $|\ell_1 \cup \ell_2^{d(u)}| < |\ell_1 \cup \ell_2| \leq k$, hence there must exist a pair u', v' with labels (X, ℓ'_1) and (Y, ℓ'_2) respectively, for which $d(u') + \text{surpl}(\ell_2^{d(u')}) > d(u) + \text{surpl}(\ell_2^{d(u)})$ (we can extend the prefix Y' by inserting the next letter of Y after its last position), hence we get the following corollary. (The algorithm can easily use $\ell_2^{d(u)}$ for the comparison between different ℓ_2 , but it is neater this way.)

► **Corollary 6.** *The length of the longest prefix of Y that is at edit distance at most k from X is equal to the maximum over all node pairs (u, v) of value $d(u) + \text{surpl}(\ell_2)$, where u has label (X, ℓ_1) , v has label (Y, ℓ_2) , u is a (potentially trivial) ancestor of v , and $|\ell_1 \cup \ell_2| \leq k$.*

This is also the case for the Hamming distance but then $\text{surpl}(\ell_2) = 0$.

When we are interested not only in the longest prefix of Y at edit distance at most k from X , but in the set of all the prefixes with this property, it is enough to compute the union of the intervals $d(u) + \text{surpl}(\ell_2) + [-k + |\ell_1 \cup \ell_2| .. 0]$ over all such pairs u and v .

► **Corollary 7.** *The set of lengths of prefixes of Y that are at edit distance at most k from X is equal to the union of intervals $d(u) + \text{surpl}(\ell_2) + [-k + |\ell_1 \cup \ell_2| .. 0]$ over all node pairs (u, v) , where u has label (X, ℓ_1) , v has label (Y, ℓ_2) , u is a (potentially trivial) ancestor of v , and $|\ell_1 \cup \ell_2| \leq k$.*

For Hamming distance the same property is simpler as the intervals are singletons $\{d(u)\}$.

3.3 Size of the k -Errata Tree

For completeness of the k -errata tree construction, we show a bound on the size of the data structure for a compacted trie T with at most n explicit nodes and n distinct node labels. We assume $k \leq \frac{\log n}{2 \log \log n}$ (for all further complexity considerations), but the properties and the correctness of the algorithms shown further do not require this assumption on k . For such a value of k , the size of our data structure is asymptotically the same as of the k -errata tree from [15] up to an $f(k)$ factor (the bound is not necessarily tight – optimizing the $f(k)$ value is not a focus of this paper). The proof of Lemma 8 can be found in Appendix A.

► **Lemma 8.** *For any compacted trie T with at most n explicit nodes and n distinct node labels, and for any $k \leq \frac{\log n}{2 \log \log n}$, the k -errata tree $T^{(k)}$ has $\mathcal{O}(nk!(c_\delta \log n)^k)$ explicit nodes and labels, where $c_\delta = 1$ for Hamming distance and $c_\delta = 3$ for edit distance.*

In particular, a single node label of T has $\mathcal{O}(k!(c_\delta \log n)^k)$ copies in $T^{(k)}$.

► **Corollary 9.** *For any $k \leq \frac{\log n}{2 \log \log n}$, we have $\log |T^{(k)}| = \mathcal{O}(k \log n)$.*

In what follows, we give the complexity of our algorithms including the proven bound on the $f(k)$ factor, and additionally the formulas in the most interesting case $k = \mathcal{O}(1)$.

3.4 Finding Approximate Prefixes

In Section 3.2, we were interested in a single extension-prefix pair (Y, X) , such that $X, Y \in D$. Here, say that we are given two subdictionaries D_1 and D_2 such that $D_1, D_2 \subseteq D$, and want to find for every element of D_2 its longest prefix that is at distance at most k from some element of D_1 using Corollary 6, or the lengths of all such prefixes using Corollary 7. Let $n = |D|, n_1 = |D_1|, n_2 = |D_2|$, and let m be the length of the longest string in D .

By running a DFS on $T^{(k)}$ and storing the labels from all the ancestors of a given node labeled (Y, ℓ_2) , $Y \in D_2$, we can find all the pairs where this node is the descendant. We need, however, to take care of the lists of edit operations: for the label (Y, ℓ_2) , we have to look for the elements (X, ℓ_1) , $X \in D_1$, stored in the set of ancestor labels, such that $|\ell_1 \cup \ell_2| \leq k$. We do not want to check all the elements from this set separately however (potentially there are $\Omega(n)$ such elements). We can group the elements based on their lists ℓ_1 ; still for a single ℓ_2 there can be $\binom{m}{k}$ such fitting lists ℓ_1 for Hamming distance (and even more for edit distance).

Corollary 6 tells us that as long as $|\ell_1 \cup \ell_2| \leq k$ it does not matter what is the content of ℓ_1 , nor what is the X in the label as long as $X \in D_1$. In order to avoid looking for each list ℓ_1 separately, we can store the elements (string depths of nodes with labels (X, ℓ_1) for any $X \in D_1$ and ℓ_1) grouped by the size of ℓ_1 and its intersection with every possible ℓ_2 .

More formally we can store the information about the ancestors of the current node of the DFS traversal in stacks indexed with pairs (ℓ, x) , where ℓ is a valid list of elements and $x \in [0 \dots k - |\ell|]$. For a single label (X, ℓ_1) of node u we store $d(u)$ in stacks (ℓ, x) , such that $\ell \subseteq \ell_1$, $|\ell_1| - |\ell| \leq x \leq k - |\ell|$ (ℓ_1 contains all elements of ℓ plus at most x other elements) upon reaching u (and remove it when returning to its parent). Since a list can have up to k elements, every single label (X, ℓ_1) is responsible for $2^{|\ell_1|} \cdot (k - |\ell_1| + 1) \leq 2^k(k + 1)$ elements in stacks, hence in total the DFS traversal will perform $\mathcal{O}(2^k(k + 1) \cdot |T^{(k)}|) = \mathcal{O}(n(k + 1)!(2 \cdot c_\delta \log n)^k)$ operations on stacks. Notice that we use a sparse representation (hash table) to store only the non-empty stacks explicitly – the universe of all possible stacks is of size at least m^k . We can access a stack in $\mathcal{O}(1)$ worst-case time by using perfect hashing [8].

Now, for a single label (Y, ℓ_2) of a node v , $Y \in D_2$, to find the deepest node u with label (X, ℓ_1) over all $X \in D_1$, such that $|\ell_1 \cup \ell_2| \leq k$, one only needs to take the maximum of the top elements over at most $2^{|\ell_2|} = \mathcal{O}(2^k)$ stacks $(\ell, k - |\ell_2|)$ for $\ell \subseteq \ell_2$. Through the whole DFS, we need to perform a total of $\mathcal{O}(n_2 \cdot 2^k k! (c_\delta \log n)^k)$ such operations for all $Y \in D_2$ as there are $\mathcal{O}(k! (c_\delta \log n)^k)$ copies of label Y in $T^{(k)}$ by Lemma 8.

► **Corollary 10.** *For any two dictionaries D_1 and D_2 such that $D_1, D_2 \subseteq D$, we can find the longest prefix of Y , for every $Y \in D_2$ at (Hamming or edit) distance at most k from some string $X \in D_1$, in $\mathcal{O}(n(k + 1)!(2 \cdot c_\delta \log n)^k)$ total time; for any $k = \mathcal{O}(1)$, this is $\mathcal{O}(n \log^k n)$.*

If we rather want to find the list of *all* the prefixes of Y which are at distance at most k from any $X \in D_1$, we need to look through the whole stacks, not only at the top elements, and also pay attention to the size of $\ell_1 \cup \ell_2$, for labels (Y, ℓ_2) .

For a single label (Y, ℓ_2) of a node v , if a node u with label (X, ℓ_1) , such that $|\ell_1 \cup \ell_2| = x \leq k$, is an ancestor of v , then upon visiting v , $d(u)$ will be stored on stack (ℓ, y) , where $\ell = \ell_1 \cap \ell_2$, and $y = |\ell_1| - |\ell| = |\ell_1 \cup \ell_2| - |\ell_2|$ (as $|\ell_1 \cup \ell_2| = |\ell_1| + |\ell_2| - |\ell|$). For such an element on the stack, we produce an interval $d(u) + \text{surpl}(\ell_2) + [-k + |\ell_1 \cup \ell_2| \dots 0] = d(u) + \text{surpl}(\ell_2) + [-k + y + |\ell_2| \dots 0]$ of lengths of prefixes of Y that are at edit distance at most k from X (by Corollary 7), or simply a length $d(u)$ of a prefix of Y at Hamming distance at most k from X .

The element (X, ℓ_1) can be also represented on other stacks (ℓ', y') for $\ell' \subseteq \ell_1 \cap \ell_2$ and $y' \geq |\ell_1| - |\ell'|$, but then $y' + |\ell_2| \geq |\ell_1| + |\ell_2| - |\ell'| \geq |\ell_1| + |\ell_2| - |\ell| = |\ell_1 \cup \ell_2| = y + |\ell_2|$; hence the interval generated this way is a subset of the interval represented by (ℓ, y) and thus does not change the union of such intervals (still we do not know what is the right ℓ or y , so we need to check all valid pairs).

This way we can compute the union of those at most $m \cdot 2^k(k + 1)$ elements of stacks for Hamming distance or of at most $(m + k) \cdot 2^k(k + 1)$ intervals for edit distance for a single label (Y, ℓ_2) of node v in time proportional to those values (in case of edit distance using the sweep line technique).

► **Corollary 11.** *For any two dictionaries D_1 and D_2 such that $D_1, D_2 \subseteq D$, we can find the lengths of all the prefixes of Y at (Hamming or edit) distance at most k from some string $X \in D_1$ for every $Y \in D_2$ in $\mathcal{O}((n + n_2(m + k))(k + 1)!(2 \cdot c_\delta \log n)^k)$ total time; for any $k = \mathcal{O}(1)$, this is $\mathcal{O}((n + n_2m) \log^k n)$, where m is the length of the longest string in D .*

If the focus of the output is oriented towards D_1 instead (we want to know what strings from D_1 are at distance at most k from some prefix of a string in D_2), then we can store in the stacks for each label (X, ℓ_1) the identifier of X instead. In this case if we run a DFS, and for each label (Y, ℓ_2) we collect the union of sets represented by the stacks (ℓ, y) , then the union of those results over all $Y \in D_2$ and all ℓ_2 would be equal to exactly the elements of D_1 at distance at most k from some prefix of some $Y \in D_2$.

To avoid handling unnecessary duplicates of strings X for a single (ℓ, y) (with different lists $\ell_1 \supseteq \ell$ and $\ell'_1 \supseteq \ell$), or for many different Y , instead of stacks, we store the information in a multiset data structure: a hash table storing as keys the elements of D_1 and as satellite data the multiplicity of every element. When an element $X \in D_1$ is read (and hence also returned), we delete it from every hash table and store it separately, so that it is never added again. The removal of those copies from every multiset is easy since we additionally store pointers to all of them partitioned by the elements of D_1 . For the label (Y, ℓ_2) , when reading this multiset, we iterate over the elements in the hash table in time proportional to their number (the deleted elements are stored separately). We implement multisets as dynamic perfect hash tables with $\mathcal{O}(1)$ -time worst-case operations [8].

► **Corollary 12.** *For any two dictionaries D_1 and D_2 such that $D_1, D_2 \subseteq D$, we can find all the elements of D_1 at (Hamming or edit) distance at most k from a prefix of some $Y \in D_2$ in $\mathcal{O}(n(k + 1)!(2 \cdot c_\delta \log n)^k)$ total time; for any $k = \mathcal{O}(1)$, this is $\mathcal{O}(n \log^k n)$.*

4 Application to Approximate Suffix-Prefix Dictionary Queries

Recall that we are given a dictionary $R = \{S_1, \dots, S_r\}$ of r strings whose total length is n , and we want to find, for a given set of pairs i, j , the value $\text{SPL}_{i,j}^k$ equal to the length of the longest suffix of S_i that is at (Hamming or edit) distance at most k from some prefix of S_j .

We focus on finding the longest prefix of S_j that is at distance at most k from some suffix of S_i , as our k -errata structures are more focused on the prefix lengths than suffix lengths; by Observation 2, those values are not necessarily maximized simultaneously in the edit distance case. Still the two problems reduce to one another by reversing all the strings in R . Henceforth, $\text{SPL}_{i,j}^k$ denotes the length of the longest prefix of S_j that is at (Hamming or edit) distance at most k from some suffix of S_i .

In preprocessing, we construct the generalized suffix tree ST_R of R (without the commonly used $\$ \notin \Sigma$ separators) [16]: ST_R is the compacted trie of the suffixes of all strings in R . A node u in ST_R is labeled by i if and only if u represents a suffix of string S_i from R . Provided that we are given an integer $k > 0$, we also construct the k -errata tree $ST_R^{(k)}$ of ST_R . We also distinguish the labels corresponding to the full strings S_i (and not their non-trivial suffixes) – we mark all their copies and store links to them for $\mathcal{O}(1)$ -time access.

4.1 Approximate One-to-One Queries

We first describe the full data structure for answering **One-to-One** ^{k} queries; then we give the querying algorithm; and, finally, we analyze the data structure size and the query time. Recall that **One-to-One** ^{k} (i, j) returns $\text{SPL}_{i,j}^k$, for two $i, j \in [1..r]$.

Data Structure. After preprocessing $ST_R^{(k)}$, we construct the data structure for answering One-to-One^k queries. It consists of $\mathcal{O}((k+1)2^k)$ trees, in total, of three types:

- the main tree $ST_R^{(k)}$;
- trees $ST_{i,\ell}$, which contain nodes of $ST_R^{(k)}$ with labels (i, ℓ_1) , such that $\ell \subseteq \ell_1$, for $i \in [1..r]$;
- trees $ST_{i,\ell,x}$, for $x \in [0..k - |\ell|]$, which contain nodes with labels (i, ℓ_1) , such that $\ell \subseteq \ell_1$ and $x \geq |\ell_1| - |\ell|$.

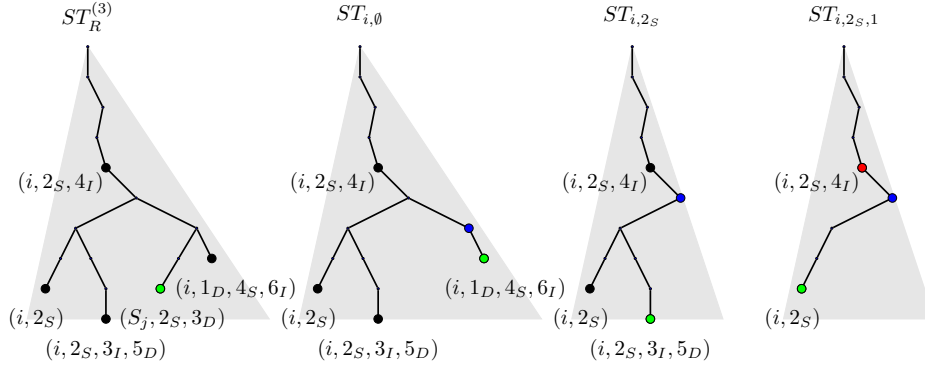
Note that $|\ell| \leq k$ and that we do not actually construct the empty trees (i.e., trees that contain no labels in the instance of the problem). Even though $ST_{i,\emptyset}$ is basically the counterpart of the tree ST_i from the One-to-One without errors solution [25], it is not necessarily equal to the k -errata tree of the suffix tree of S_i , since the heavy-light decomposition of ST_i and of the generalized suffix tree (and hence the errata trees) may differ. That is, to obtain a smaller tree, we make a copy of a larger tree, remove the labels that do not fit, remove the nodes that do not have any labels in their subtree, and finally compactify the tree.

The trees of the first two types are enhanced with rank-select (RS) [6] and lowest common ancestor (LCA) data structures [9]; the trees of the third type are enhanced with a weighted ancestor (WA) data structure [3] plus a pointer from each node to its closest ancestor with a label. (The latter information can be trivially computed by using a DFS tree traversal.)

While the LCA and WA data structures are by default defined on (rooted) trees, we need to specify how the RS data structures are defined on trees (as they are usually defined on arrays). In our case, the RS data structures span over all the node labels (in one of the constructions a single label will actually be given a few next positions). In particular, the positions are first ordered by the position of the node (left to right DFS first visit order), and next its labels (i, ℓ) are ordered lexicographically. In case of $ST_R^{(k)}$, the RS data structure is over the alphabet $[1..r]$, where the letter representing the node label (i, ℓ) is i . Each label of $ST_R^{(k)}$ is linked with its counterpart from $ST_{i,\emptyset}$. The tree $ST_{i,\ell}$ is connected to trees $ST_{i,\ell \cup \{l_E\}}$, where $l_E \geq \max^+(\ell)$ and trees $ST_{i,\ell,x}$, where $x \leq k - |\ell|$, and hence it contains two separate RS data structures. The first one is over the alphabet of valid elements: $[\max^+(\ell)..n_S]$ for Hamming distance; $[\max^+(\ell)..(n+k)_I]$ for edit distance. Each label (i, ℓ_1) is given a position for every element $l_E \in \ell_1$ greater or equal to $\max^+(\ell)$ and this element l_E is its representative. This position in the RS array is linked with the corresponding terminal node of tree $ST_{i,\ell \cup \{l_E\}}$. The second one is over alphabet $[0..k - |\ell|]$. Each node label (i, ℓ_1) is given $k - |\ell_1| + 1$ positions that contain values $x \in [|\ell_1| - |\ell|..k - |\ell|]$. Each such position is connected to the corresponding node label of the tree $ST_{i,\ell,x}$.

Querying. We want to find $\text{SPL}_{i,j}^k$, i.e., the longest prefix of S_j that is at (Hamming or edit) distance at most k from some suffix of S_i (if we store some additional information with the labels we can also find the length of this suffix). By Lemma 8, S_j corresponds to $\mathcal{O}(k!(c_\delta \log n)^k)$ different node labels (j, ℓ_2) in $ST_R^{(k)}$; we will consider each such label independently, and then choose the optimal result.

For a single node v with label (j, ℓ_2) , we want to find its lowest ancestor with label (i, ℓ_1) such that $|\ell_1 \cup \ell_2| \leq k$. We know that the ancestor is contained in one of the trees $ST_{i,\ell,k-|\ell_2|}$ for some $\ell \subseteq \ell_2$. If we know a node v' with a label that is stored in that tree, and is the closest to v in the DFS order of $ST_R^{(k)}$ (they have the deepest LCA among all such nodes v'), then we know that any ancestor of v that belongs to $ST_{i,\ell,k-|\ell_2|}$ is also an ancestor of v' , and that every ancestor of v' at depth at most $\text{LCA}(v, v')$ is also an ancestor of v . Thus for knowing the location of v' in $ST_{i,\ell,k-|\ell_2|}$ and the value of $\text{LCA}(v, v')$, it suffices to ask a WA query in $ST_{i,\ell,k-|\ell_2|}$ for node v' and depth $\text{LCA}(v, v')$ and then ask for the lowest ancestor of the node with a label (this information is stored in the node as a pointer) - the answer is



■ **Figure 3** An example traversal through the trees to reach the lowest ancestor (i, ℓ_1) of label $(S_j, 2_S, 3_D)$, such that $2_S \in \ell_1$ and $|\ell_1 \cup \{2_S, 3_D\}| \leq 3$. The green node depicts the label that is closest to $(S_j, 2_S, 3_D)$ in $ST_R^{(3)}$, out of the ones existing in the tree, while the blue one depicts the lowest common ancestor of the two nodes (equal to green in $ST_R^{(3)}$). The next green node is obtained via a rank-select query, while the blue nodes depth is obtained via an LCA query. After reaching a terminating tree, we ask for the depth of the lowest ancestor of the blue node that contains a label.

the sought ancestor of v . To obtain the location of v' and the value of $\text{LCA}(v, v')$, we jump through the trees in between, always storing the two pieces of information: the labeled node v'' closest to v that appears in the tree; and the value $\text{LCA}(v, v'')$.

In $ST_R^{(k)}$ we want to find the closest label (i, ℓ_1) for any ℓ_1 ; we simply ask an RS query to find the closest such node to the left and to the right of v . We compare those two nodes by the string depth of their LCA with v , choose the one for which this depth is larger (breaking ties arbitrarily), and then we proceed to the node with this label in $ST_{i,0}$.

Now, in tree $ST_{i,\ell}$, for $\ell \subseteq \ell_2$, we start with the labeled node v' closest to v in $ST_R^{(r)}$, and the string depth of their LCA. For each element $l_E \in \ell_2$, such that $l_E \geq \max^+(\ell)$, we find the closest node with label (i, ℓ_3) such that $\ell \cup \{l_E\} \subseteq \ell_3$ in $ST_{i,\ell}$ by asking the RS data structure (the one over $[\max^+(\ell) \dots (n+k)_I]$) for l_E – both left and right. We compare those two nodes based on the LCA string depth with v' to get the node v'' , then compute $\text{LCA}(v, v'') = \min\{\text{LCA}(v, v'), \text{LCA}(v', v'')\}$, and jump to the corresponding terminal node of $ST_{i,\ell \cup \{l_E\}}$ with the new LCA value. Additionally we ask the other RS data structure for the closest label represented by $k - |\ell_2|$, and jump to the labeled node of $ST_{i,\ell, k - |\ell_2|}$. There we find its lowest ancestor at depth at most the value of stored LCA string depth, and ask this ancestor for the string depth of its lowest ancestor storing a label. If this value is larger than the value of the currently stored candidate, we replace the candidate with the new one.

After the whole recursive procedure ends for each labeled node v corresponding to the full string S_j , the stored candidate is the final answer. Inspect Figure 3, for an example.

Data Structure Size. Each node of $ST_R^{(k)}$ belongs to $\mathcal{O}((k+1) \cdot 2^k)$ different trees. In total, the trees have size $N = \mathcal{O}(n(k+1)!(2c_\delta \log n)^k)$. The RS [6], LCA [9], and WA [3] data structures occupy $\mathcal{O}(N)$ space and can be constructed in $\mathcal{O}(N \log \log N)$ time.

Query Time. We have $\mathcal{O}(k!(c_\delta \log n)^k)$ node labels (S_j, ℓ_2) for a full string S_j . For each of those we reach at most 2^k further trees. Each operation of reaching the next tree (from the previous one) costs a constant number of RS, LCA and WA queries – that is, the cost of reaching each tree is $\mathcal{O}(\log \log n)$; hence the total cost is $\mathcal{O}((2c_\delta \log n)^k \log \log n)$. In particular, the RS [6] and WA [3] queries take $\mathcal{O}(\log \log n)$ time when $\mathcal{O}(n)$ space is used.

► **Theorem 13.** For any dictionary of strings of total length n and any $k \leq \frac{\log n}{2 \log \log n}$, we can construct a data structure of $\mathcal{O}(n(k+1)!(2c_\delta \log n)^k)$ words of space answering *One-to-One* ^{k} queries in $\mathcal{O}((k+1)!(2c_\delta \log n)^k \log \log n)$ time. The data structure can be constructed in $\mathcal{O}(n(k+1)!(2c_\delta \log n)^k \log \log n)$ time.

For any $k = \mathcal{O}(1)$, the data structure size is $\mathcal{O}(n \log^k n)$, the query time is $\mathcal{O}(\log^k n \log \log n)$ and the construction time is $\mathcal{O}(n \log^k n \log \log n)$.

4.2 Approximate Report Queries

Once again we use an inverted version of the problem that is equivalent to the original one; i.e., we want to answer *Report* ^{k} (j, d) queries that return all the values of $i \in [1..r]$ such that there exists a prefix of S_j at distance at most k from some suffix of S_i of length at least d . Recall that by Observation 2, for edit distance, this is not necessarily equivalent to finding all j such there exists a prefix of S_j of length at least d at distance at most k from some suffix of S_i – this can be done using the idea of Corollary 12 within the same time complexity.

By Corollary 6, to find whether a prefix of length at least d of S_j is at distance at most k from a suffix of S_i , it suffices to check, for every label (j, ℓ_2) of a node v representing the full string S_j , if it has an ancestor u with label (i, ℓ_1) , such that $|\ell_1 \cup \ell_2| \leq k$ and $d(u) \geq d - \text{surpl}(\ell_2)$. We first describe the data structure for answering *Report* ^{k} queries and analyze its size; then we give the querying algorithm and analyze the query time.

Data Structure. We start by constructing $ST_R^{(k)}$. For each pair (ℓ, x) , such that $x \leq k - |\ell|$ and there exists a label with list ℓ in $ST_R^{(k)}$, we create a linear-space 2D rectangle stabbing data structure [30]. For every node u with label (i, ℓ_1) , we create a rectangle $[L(u) .. R(u)] \times [0 .. d(u)]$ with label i , where $L(u)$ and $R(u)$ are the numbers of the leftmost and the rightmost label in the subtree of u in $ST_R^{(k)}$ when ordered in the left-to-right DFS traversal order, and insert it to data structures (ℓ, x) for each $\ell \subseteq \ell_1$ and $|\ell_1| - |\ell| \leq x \leq k - |\ell|$. Next in every structure we make the rectangles of the same type disjoint.

We have in total $|ST_R^{(k)}| = \mathcal{O}(nk!(c_\delta \log n)^k)$ labels, hence the total number of rectangles in all the data structures is $|ST_R^{(k)}| \cdot (k+1)2^k = \mathcal{O}(n(k+1)!(2c_\delta \log n)^k)$, and the dimensions of the rectangles are in $[0 .. |ST_R^{(k)}|] \times [0 .. n]$. Hence the total size is $\mathcal{O}(n(k+1)!(2c_\delta \log n)^k)$.

Querying. For each label (j, ℓ_2) of a node v representing the full string S_j , and for each $\ell \subseteq \ell_2$, we ask a query $(L(v), d - \text{surpl}(\ell_2))$ to the 2D rectangle stabbing data structure $(\ell, k - |\ell_2|)$. Over all labels (j, ℓ_2) , we have in total $\mathcal{O}(k!(2c_\delta \log n)^k)$ lists of total length $\mathcal{O}(|Q| \cdot k!(2c_\delta \log n)^k)$, where $|Q|$ is the size of the output, and we want to output their union. By using the 2D rectangle stabbing data structure from [30] we obtain the following result.

► **Theorem 14.** For any dictionary of strings of total length n and any $k \leq \frac{\log n}{2 \log \log n}$, we can construct a data structure of $\mathcal{O}(nk!(2c_\delta \log n)^k)$ words of space answering *Report* ^{k} queries in $\mathcal{O}((k+1)!(2c_\delta \log n)^k (\log n / \log \log n + |Q|))$ time. For any $k = \mathcal{O}(1)$, the data structure size is $\mathcal{O}(n \log^k n)$ and the query time is $\mathcal{O}(\log^k n (\log n / \log \log n + |Q|))$.

References

- 1 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11):276, 2020. doi:10.3390/A13110276.
- 2 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

- 3 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.
- 4 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theor. Comput. Sci.*, 922:271–282, 2022. doi:10.1016/j.tcs.2022.04.029.
- 5 Carl Barton, Costas S. Iliopoulos, Solon P. Pissis, and William F. Smyth. Fast and simple computations using prefix tables under hamming and edit distance. In Jan Kratochvíl, Mirka Miller, and Dalibor Fronček, editors, *Combinatorial Algorithms - 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014, Revised Selected Papers*, volume 8986 of *Lecture Notes in Computer Science*, pages 49–61. Springer, 2014. doi:10.1007/978-3-319-19315-1_5.
- 6 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.
- 7 Ilan Ben-Bassat and Benny Chor. String graph construction using incremental hashing. *Bioinform.*, 30(24):3515–3523, 2014. doi:10.1093/bioinformatics/btu578.
- 8 Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. *J. ACM*, 70(6):40:1–40:51, 2023. doi:10.1145/3625817.
- 9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- 10 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. FSG: fast string graph construction for de novo assembly. *J. Comput. Biol.*, 24(10):953–968, 2017. doi:10.1089/cmb.2017.0089.
- 11 Panagiotis Charalampopoulos, Pawel Gawrychowski, Yaowei Long, Shay Mozes, Seth Pettie, Oren Weimann, and Christian Wulff-Nilsen. Almost optimal exact distance oracles for planar graphs. *J. ACM*, 70(2):12:1–12:50, 2023. doi:10.1145/3580474.
- 12 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/s00453-021-00821-y.
- 13 Shiri Chechik. Approximate distance oracles with constant query time. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 654–663. ACM, 2014. doi:10.1145/2591796.2591801.
- 14 Shiri Chechik. Approximate distance oracles with improved bounds. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 1–10. ACM, 2015. doi:10.1145/2746539.2746562.
- 15 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don’t cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.
- 16 Martin Farach. 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*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
- 17 Sebastian Forster, Gramoz Goranci, Yasamin Nazari, and Antonis Skarlatos. Bootstrapping dynamic distance oracles. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPICs*, pages 50:1–50:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ESA.2023.50.

- 18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 19 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992. doi:10.1016/0020-0190(92)90176-V.
- 20 Tomasz Kociumaka. Efficient data structures for internal queries in texts. *PhD thesis, University of Warsaw, October 2018.*, 2018. URL: <https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 21 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.
- 22 Gregory Kucherov and Dekel Tsur. Improved filters for the approximate suffix-prefix overlap problem. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2014. doi:10.1007/978-3-319-11918-2_14.
- 23 Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989. doi:10.1016/0196-6774(89)90010-2.
- 24 Grigorios Loukides and Solon P. Pissis. All-pairs suffix/prefix in optimal time using Aho-Corasick space. *Inf. Process. Lett.*, 178:106275, 2022. doi:10.1016/j.ipl.2022.106275.
- 25 Grigorios Loukides, Solon P. Pissis, Sharma V. Thankachan, and Wiktor Zuba. Suffix-prefix queries on a dictionary. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPICs*, pages 21:1–21:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.CPM.2023.21.
- 26 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, September 2005. doi:10.1093/bioinformatics/bti1114.
- 27 Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Inf. Process. Lett.*, 110(3):123–128, 2010. doi:10.1016/j.ipl.2009.10.015.
- 28 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.
- 29 Kim R. Rasmussen, Jens Stoye, and Eugene W. Myers. Efficient q -gram filters for finding all ϵ -matches over a given length. *J. Comput. Biol.*, 13(2):296–308, 2006. doi:10.1089/cmb.2006.13.296.
- 30 Qingmin Shi and Joseph F. JáJá. Novel transformation techniques using q -heaps with applications to computational geometry. *SIAM J. Comput.*, 34(6):1474–1492, 2005. doi:10.1137/S0097539703435728.
- 31 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinform.*, 26(12):367–373, 2010. doi:10.1093/bioinformatics/btq217.
- 32 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.
- 33 Saumya Talera, Parth Bansal, Shabnam Khan, and Shahbaz Khan. Practical algorithms for hierarchical overlap graphs. *CoRR*, abs/2402.13920, 2024. doi:10.48550/arXiv.2402.13920.
- 34 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018. doi:10.1007/978-3-319-89929-9_14.

- 35 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.
- 36 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix-prefix problem. *J. Discrete Algorithms*, 37:34–43, 2016. doi:10.1016/j.jda.2016.04.002.
- 37 Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Inf. Comput.*, 213:49–58, 2012. doi:10.1016/j.ic.2012.02.002.

A Omitted Proofs

Proof of Lemma 4. Say, that the leftmost operation in an optimal sequence of edit operations between strings X and Y happens at position $l_1 \leq \text{LCP}(X, Y)$, that is, in particular $X[l_1] = Y[l_1]$.

The positions $1, \dots, l_1 - 1$ of strings X and Y are matched respectively, and we need to match the letter $Y[l_1]$ with something in X . Therefore, the leftmost operation (subset of subsequent operations) is either an insertion of letter $Y[l_1]$ or a possibly empty sequence of deletions followed by an identity or a substitution on the first not deleted position (if the first operation after deletions was an insertion, then we can pull this insertion left).

- If the first operation is insertion, then we can instead match letter $Y[l_1]$ with the letter $X[l_1]$, and insert the letter one position further - since the inserted letter and $X[l_1]$ are equal this does not change the number of operations nor the result.
- If a sequence of deletions is followed by an identity, then we can instead match letter $Y[l_1]$ with $X[l_1]$ and delete the matched letter $X[l_2]$ for $l_2 > l_1$ (if the sequence was empty then the position of the first operation was not l_1) shifting the left-most first operation to the right.
- If a sequence of deletions was followed by a substitution of letter $X[l_2]$, then we can instead match letter $Y[l_1]$ with $X[l_1]$ and delete letter $X[l_2]$ obtaining a set of operations of smaller size - a contradiction with the minimality of the set.

For the recursive part the same proof follows the same way after applying the first operation. ◀

Full proof of Lemma 5. (\Rightarrow) Let Y' be a prefix of Y at edit distance at most k from X . In particular there exists a sequence of at most k operations that transform X into Y' . Using this sequence of edit operations, we find two nodes u and v in $T^{(k)}$ with labels (X, ℓ_1) and (Y, ℓ_2) respectively, satisfying the conditions of the statement.

Recall that T denotes the compacted trie of D . We start our search at the root of T with two empty lists ℓ_1 and ℓ_2 . By Lemma 4 we can safely assume that the position of the first edit operation used to change X into Y' is $l = \text{LCP}(X, Y') + 1 = \text{LCP}(X, Y) + 1$. The letter at position l in X is changed: it is deleted; substituted with the l -th letter of Y' ; or the l -th letter of Y' is inserted right before this letter.

Let us first assume that neither X is a prefix of Y nor Y' is a prefix of X (or that we do not have any more edit operations to apply).

We know that in T the nodes with labels X and Y have their lowest common ancestor at depth $l - 1$. Now, assume, that the heavy child of this common ancestor is reached from its parent through an edge labeled with letter a . Now depending on the operation performed:

- If the operation is a substitution, then if $X[l] \neq a$, add l_S to ℓ_1 , and if $Y[l] \neq a$, add l_S to ℓ_2 (possibly both lists gain the same element).
- If the operation is a deletion, then if $X[l] \neq a$, add l_D to ℓ_1 , otherwise ($Y[l] \neq a$), add l_I to ℓ_2 .
- If the operation is an insertion, then, if $Y[l] = a$ ($X[l] \neq a$), add l_I to ℓ_1 , otherwise ($Y[l] \neq a$), add l_D to ℓ_2 .

Now the nodes with labels (X, ℓ_1) and (Y, ℓ_2) belong to the subtree of a node at depth $l - 1$ in $T^{(1)}$ (and hence also in $T^{(k)}$), and actually if the added label was not l_D also of the node at depth l (the heavy child of the one at depth $l - 1$), this ensures that the next element will be no smaller than $\max^+(\ell_1 \cup \ell_2)$. Furthermore, we know that the edit distance between X and Y' , with the operations in lists ℓ_1 and ℓ_2 applied, is smaller by 1 than the distance between X and Y' .

We iterate this technique at most k times walking down the tree. At step $x \leq k$ we start our work at a node of $T^{(x-1)}$ (a node of $T^{(k)}$ that was there before the x -th step of the k -errata construction) that is an ancestor of nodes with labels (X, ℓ_1) and (Y, ℓ_2) . After at most k steps we get to a node in $T^{(k)}$, such that its descendant with label (X, ℓ_1) is an ancestor of the node with label (Y, ℓ_2) as no further edit operations remain. Y' turns out to be the prefix of Y with the last $d(v) - d(u)$ letters removed.

Now consider the case where, after applying $x - 1$ operations, the nodes with labels (X, ℓ_1) and (Y, ℓ_2) , respectively, are already in an ancestor descendant relationship, and there are still operations changing X into Y' not processed yet.

If a node with label (X, ℓ_1) is a (non-trivial) descendant of node with label (Y, ℓ_2) the only possible next operation is deletion (we need to remove the suffix of X). If the first edge used to reach the node with label (X, ℓ_1) from the node with label (Y, ℓ_2) is heavy in $T^{(x-1)}$ we can still add l_I to ℓ_2 (due to the extra copy of labeled nodes), otherwise a node with label $(X, \ell_1 \cup \{l_D\})$ is a descendant of the node with label (Y, ℓ_2) in $T^{(x)}$ of smaller depth. We apply this until we reach a single node with both labels (X, ℓ_1) and (Y, ℓ_2) .

If a node with label (X, ℓ_1) is a (non-trivial) ancestor of a node with label (Y, ℓ_2) , we already know that the claim is true (for some prefix Y'' of Y , that is at smaller edit distance from X than Y'). We may still want to pair X with a longer (or a shorter) prefix of Y , in particular the longest possible. If the next applied operation is an insertion (we want to make the prefix longer) of letter $Y[l]$, then this is symmetric to the case considered in the previous paragraph. We do not provide any modification that allows applying deletion in this case since, in our main application, we care for the longest prefixes only. Still, we know, that the shortest prefix of Y we could obtain this way would be $k - x + 1$ letters shorter than Y'' , and hence every such prefix is actually represented in $T^{(k)}$.

In case of Hamming distance the set of operations consists solely of substitutions, and hence the proof reduces naturally (only elements l_S are inserted to the lists, hence only copies of labels with such lists are considered).

(\Leftarrow) This is the reversal of the forward (\Rightarrow) proof construction, that is, we change ℓ_1 and ℓ_2 into a single list of edit operations.

In the Hamming distance case, the set of mismatches between X and $Y[1..|X|]$ is contained in the set $\{l : l_S \in \ell_1 \cup \ell_2\}$, hence X and a prefix of Y are at Hamming distance at most $|\ell_1 \cup \ell_2| \leq k$ ($|\ell_1 \cup \ell_2|$ may be greater than the actual Hamming distance if a suboptimal pair of labeled nodes is chosen, but this is not a problem). In the edit distance case, the construction is more complicated due to the shifts made by insertions and deletions.

We have two nodes u and v with labels (X, ℓ_1) and (Y, ℓ_2) , respectively, such that u is an ancestor of v . Let Y' be the prefix of Y of length $|Y| - d(v) + d(u) = d(u) + \text{surpl}(\ell_2)$ assuming that $\ell_2 = \ell_2^{d(u)}$ ($l \leq d(u)$ for $l_E \in \ell_2$) – otherwise we can remove those larger elements as those only affect the part of Y that does not play a role in the transformation of X into Y' (there exists a descendant of u with label $(Y, \ell_2^{d(u)})$ which we can choose as our v). We know that after applying operations from ℓ_1 to X and operations from ℓ_2 to Y' we obtain the same string: the one obtained by reading the path from root to u in $T^{(k)}$.

Let $\ell = \ell_1 \cup \ell_2$, and for each element of ℓ we mark whether it comes from ℓ_1 , from ℓ_2 , or from both. For ℓ it may actually be the case that both l_S and l_I appear (coming from two different lists ℓ_1, ℓ_2); this will not be that much of a problem however since this is counted as two edit operations in $|\ell|$. We continue by showing that in fact ℓ encodes a transformation of X into Y' with the use of at most $|\ell|$ edit operations.

We read ℓ and modify X step by step to reach Y' keeping an invariant that when element l_E is about to be processed the first $l - 1 + i$ positions of the modified string X are equal to the first $l - 1 + i$ positions of Y , where the value i represents the imbalance between deletion and insertion operations from ℓ_2 already applied, and hence it starts from 0. We have that $i = \text{surpl}(\ell_2)$, where ℓ_2 is the prefix of ℓ_2 storing operations already applied - $\ell_2' = \ell_2$ when we move to the node at depth l for the first time.

- If the element is l_S , we substitute the letter at position $l + i$ of the current version of X with $Y[l + i]$.
- If the element l_D comes from both lists ℓ_1 and ℓ_2 (multiple l_D in both lists are matched into pairs), we replace the letter at position $l + i$ by $Y[l + i]$ and increase i by 1 (deletion of the corresponding letters of both strings is equivalent to a substitution, but the depth of the node decreases by one).
- If the element l_D comes from list ℓ_1 (a surplus in pairing), we remove the letter at position $l + i$ in the current version of X .
- If the element l_D comes from list ℓ_2 (a surplus in pairing), we add letter $Y[l + i]$ between letters at positions $l + i - 1$ and $l + i$ in X and increase i by 1.
- If the element l_I comes from both lists we do not modify X , and instead decrease i by 1 (inserting the same letter at corresponding positions in X and Y does not change the edit distance).
- If the element l_I comes from list ℓ_1 , we insert letter $Y[l + i]$ between letters at positions $l + i - 1$ and $l + i$ in X .
- If the element l_I comes from list ℓ_2 , we delete the letter at position $l + i$ in X and decrease i by 1.

Notice that when i decreases, the value l of the next l_E must be strictly greater, hence the sum $l + i$ never decreases between operations; and when $l + i$ increases, $X[l + i]$ and $Y[l + i]$ (for X after the changes applied and the old value of $l + i$) must actually be equal.

Further notice that, as claimed, the prefix Y' obtained from X by applying those edit operations (obtained from ℓ_1 and $\ell_2^{d(u)}$) has length equal to $d(u) + \text{surpl}(\ell_2^{d(u)})$. ◀

Proof of Lemma 8. Let $|T^{(k)}|$ denote the size of the k -errata of T . The proof proceeds by induction. $|T^{(0)}| = |T| \leq 3n$.

Assume, that $|T^{(k-1)}| \leq c_{n,k-1} \cdot n(k-1)!(c_\delta \log n)^{k-1}$, for a constant $3 \leq c_{n,k-1} \leq 16$, depending only on n and k , to be specified later.

A single node or label can be copied at most $c_\delta \log |T^{(k-1)}| + 1$ times, since the heavy-light decomposition of the $(k-1)$ -errata tree is weighted by the number of explicit nodes and labels. Note that c_δ comes from the number of copies made, and $+1$ comes from the extra copy of a labeled node that is not a leaf.

Now $c_\delta \log |T^{(k-1)}| + 1 \leq c_\delta \log [c_{n,k-1} \cdot n(k-1)!(c_\delta \log n)^{k-1}] + 1 \leq c_\delta [\log c_{n,k-1} + \log n + (k-1) \log k + (k-1) \log c_\delta + \log n] + 1 \leq c_\delta [k(\log n + \log k + 7)] = c_\delta k \log n [1 + \frac{\log k + 7}{\log n}]$.

Hence, $|T^{(k)}| \leq |T^{(k-1)}| \cdot (c_\delta \log |T^{(k-1)}| + 1) \leq c_{n,k-1} \cdot n(k-1)!(c_\delta \log n)^{k-1} \cdot c_\delta k \log n [1 + \frac{\log k + 7}{\log n}] = c_{n,k-1} \cdot [1 + \frac{\log k + 7}{\log n}] \cdot [nk!(c_\delta \log n)^k]$.

Hence for $c_{n,k} = c_{n,k-1} \cdot [1 + \frac{\log k + 7}{\log n}]$ the property $|T^{(k)}| \leq c_{n,k} \cdot nk!(c_\delta \log n)^k$ is satisfied, and it remains to show, that $c_{n,k} \leq 16$ if $k \leq \frac{\log n}{2 \log \log n}$. $c_{n,0} = 3, c_{n,k} \leq 3 \cdot [1 + \frac{\log k + 7}{\log n}]^k$.

Recall, that $(1 + \frac{1}{x})^x \rightarrow e$, and that for $k \leq \frac{\log n}{2 \log \log n}$ we have $\frac{\log k + 7}{\log n} \leq \frac{2 \log \log n}{\log n}$ (for large enough n), hence $c_{n,k} \leq 3e < 16$.

The bound on the number of copies of a single label follows analogously. ◀