

Deterministic Sparse Suffix Sorting in the Restore Model*

Johannes Fischer¹, Tomohiro I², and Dominik Köppl¹

¹Department of Computer Science, TU Dortmund, Germany

²Kyushu Institute of Technology, Japan

Abstract

Given a text T of length n , we propose a deterministic online algorithm computing the sparse suffix array and the sparse longest common prefix array of T in $\mathcal{O}(c\sqrt{\lg n} + m \lg m \lg n \lg^* n)$ time with $\mathcal{O}(m)$ words of space under the premise that the space of T is rewritable, where $m \leq n$ is the number of suffixes to be sorted (provided online and arbitrarily), and c is the number of characters with $m \leq c \leq n$ that must be compared for distinguishing the designated suffixes.

1 Introduction

Sorting suffixes of a long text lexicographically is an important first step for many text processing algorithms [36]. The complexity of the problem is quite well understood, as for integer alphabets suffix sorting can be done in optimal linear time and in-place [29, 18]. In this article, we consider a variant of the problem: instead of computing the order of *every* suffix, we address the **sparse suffix sorting problem**. Given a text $T[1..n]$ of length n and a set $\mathcal{P} \subseteq [1..n]$ of m arbitrary positions in T , the problem asks for the (lexicographic) order of the suffixes starting at the positions in \mathcal{P} . The answer is encoded by a permutation of \mathcal{P} , which is called the **sparse suffix array (SSA)** of T (with respect to \mathcal{P}) and denoted by $\text{SSA}(T, \mathcal{P})$.

Applications are found in external memory LCP-array construction algorithms [23], and in the search of maximal exact matches [26, 44], i.e., substrings found in two given strings that can be extended neither to their left nor to their right without getting a mismatch.

Like the “full” suffix arrays, we can enhance $\text{SSA}(T, \mathcal{P})$ with the lengths of the longest common prefixes (LCPs) between adjacent suffixes in $\text{SSA}(T, \mathcal{P})$. These lengths are stored in the **sparse longest common prefix array (SLCP)**, which we denote by $\text{SLCP}(T, \mathcal{P})$. In combination, $\text{SSA}(T, \mathcal{P})$ and $\text{SLCP}(T, \mathcal{P})$ store the same information as the **sparse suffix tree**, i.e., they implicitly represent a compacted trie over all suffixes starting at the positions in \mathcal{P} . The sparse suffix tree is an efficient index for pattern matching [28].

Based on classic suffix array construction algorithms [25, 33], sparse suffix sorting is easily conducted in $\mathcal{O}(n)$ time if $\mathcal{O}(n)$ words of additional working space is available. For $m = o(n)$, however, the working space may be too large, compared to the final space requirement of $\text{SSA}(T, \mathcal{P})$. Although some special choices of \mathcal{P} admit space-optimal $\mathcal{O}(m)$ -words construction algorithms (e.g. [24], see also the related work listed in [7]), the problem of sorting arbitrary suffixes in small space seems to be much harder. We are aware of the following results: As a deterministic algorithm, Kärkkäinen et al. [25] gave a trade-off using $\mathcal{O}(\tau m + n\sqrt{\tau})$ time and $\mathcal{O}(m + n/\sqrt{\tau})$ words of working space with a parameter $\tau \in [1..\sqrt{n}]$. If randomization is allowed, there is a technique based on Karp-Rabin fingerprints, first proposed by Bille et al. [7] and later improved by I et al. [20]. Gawrychowski and Kociumaka [16] presented an algorithm running with $\mathcal{O}(m)$ words of additional space in either $\mathcal{O}(n\sqrt{\lg m})$ expected time, or in $\mathcal{O}(n)$ time as a Monte Carlo algorithm (i.e., the output is correct only with high probability). Most recently, Prezza [35] presented a Monte Carlo algorithm in the restore model [8] that runs with $\mathcal{O}(m)$ words of space in $\mathcal{O}(n + m \lg^2 n)$ expected time.

*Parts of this work have already been presented at the 12th Latin American Symposium [12].

1.1 Computational Model

Let \lg and \log_x denote the logarithm to the base two and to the base x for a real number x , respectively. Our computational model is the word RAM model with word size $\Omega(\lg n)$. Here, characters use $\lceil \lg \sigma \rceil$ bits, where σ is the alphabet size; hence, $\lfloor \log_\sigma n \rfloor$ characters can be packed into one word. Comparing two strings X and Y therefore takes $\mathcal{O}(\text{lcp}(X, Y) / \log_\sigma n)$ time, where $\text{lcp}(X, Y)$ denotes the length of the LCP of X and Y .

We assume that the text T of length n is loaded into RAM. We work with the restore model [8], where algorithms are allowed to overwrite parts of T , as long as they can restore T to its original form at termination. Apart from this space, we are only allowed to use $\mathcal{O}(m)$ words. The positions in \mathcal{P} are assumed to arrive on-line, implying in particular that they need not be sorted. We aim at worst-case efficient *deterministic* algorithms.

1.2 Algorithm Outline and Our Results

Our main algorithmic idea is to insert the suffixes starting at the positions of \mathcal{P} into a self-balancing binary search tree [21]; since each insertion invokes $\mathcal{O}(\lg m)$ suffix-to-suffix comparisons, the time complexity is $\mathcal{O}(t_s m \lg m)$, where t_s is the cost for a suffix-to-suffix comparison. If all suffix-to-suffix comparisons are conducted naively by comparing the characters ($t_s = \mathcal{O}(n / \log_\sigma n)$), the resulting worst case time complexity is $\mathcal{O}(nm \lg m / \log_\sigma n)$. In order to speed this up, our algorithm identifies large identical substrings at different positions during different suffix-to-suffix comparisons. Instead of performing naive comparisons on identical parts over and over again, we build a data structure (stored in redundant text space) to accelerate subsequent suffix-to-suffix comparisons. Informally, when two (possibly overlapping) substrings in the text are detected to be the same, one of them can be overwritten.

To accelerate suffix-to-suffix comparisons, we devise a new data structure called *hierarchical stable parsing (HSP) tree* that is based on *edit sensitive parsing (ESP)* [11]. The HSP tree supports *longest common extension (LCE)* queries. An LCE query $\text{lce}(i, j)$ on an HSP tree asks for the length $\text{lcp}(T[i..], T[j..])$ of the LCP of two suffixes starting at the respective positions i and j of the text T on which the tree is built. Besides answering LCE queries, HSP trees are *mergeable*, allowing us to build a dynamically growing LCE index on substrings read in the process of the sparse suffix sorting. Consequently, comparing two already indexed substrings is done by a single LCE query.

In their plain form, HSP trees need more space than the text itself; to overcome this space problem, we devise a *truncated* version of the HSP tree, yielding a trade-off parameter between space consumption and LCE query time. By choosing this parameter appropriately, the truncated HSP tree fits into the text space. With a text space management specialized on the properties of the HSP, we achieve the result of Thm. 1 below.

We make the following definition that allows us to analyze the running time more accurately. Define $\mathcal{C} := \bigcup_{p, p' \in \mathcal{P}, p \neq p'} [p..p + \text{lcp}(T[p..], T[p'..])]$ as the set of positions that must be compared for distinguishing the suffixes starting at the positions of \mathcal{P} . Then sparse suffix sorting is trivially lower bounded by $\Omega(|\mathcal{C}| / \log_\sigma n)$ time. With the definition of \mathcal{C} , we now can state the main result of this article as follows:

Theorem 1. Given a text T of length n that is loaded into RAM, the SSA and SLCP of T for a set of m arbitrary positions can be computed deterministically in $\mathcal{O}(|\mathcal{C}| (\sqrt{\lg \sigma} + \lg \lg n) + m \lg m \lg n \lg^* n) = \mathcal{O}(|\mathcal{C}| \sqrt{\lg n} + m \lg m \lg n \lg^* n)$ time, using $\mathcal{O}(m)$ words of additional working space.

Excluding the loading cost for the text, the running time can be sublinear (when $|\mathcal{C}| = o(n / \sqrt{\lg n})$ and $m \lg m = o(n / \lg n \lg^* n)$). To the best of our knowledge, this is the first algorithm that refines the worst-case performance guarantee. All previously mentioned (deterministic and randomized) algorithms take $\Omega(n)$ time even if we exclude the loading cost for the text. Also, general string sorters (e.g., forward radix sort [2] or multikey quicksort [4]), which do not take advantage of the overlapping of suffixes, suffer from the lower bound of $\Omega(\ell / \log_\sigma n)$ time, where ℓ is the sum of all LCP values in the SLCP, which is always at least $|\mathcal{C}|$, but can in fact be $\Theta(nm)$.

Construction		Data Structure		Ref
Time	Working Space	Space	Query Time	
$\mathcal{O}(n\tau)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(\tau \lg \min(\tau, \frac{n}{\tau}))$	[42]
$\mathcal{O}(n^{2+\epsilon})$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(\tau)$	[6]
$\mathcal{O}\left(n \left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_{\sigma} n} \right)\right)$	$\mathcal{O}(\max(\frac{n}{\lg n}, \tau^{\lg 3} \lg^* n))$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\lg^* n \left(\lg\left(\frac{\ell}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_{\sigma} n} \right)\right)$	Thm. 3
$\mathcal{O}\left(n \left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_{\sigma} n} \right)\right)$	$\mathcal{O}(\tau^{\lg 3} \lg^* n)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\lg^* n \left(\lg\left(\frac{n}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_{\sigma} n} \right)\right)$	Cor. 29

Figure 1: Deterministic LCE data structures with trade-off parameters. The length returned by an LCE query is denoted by ℓ . ϵ and τ with $\epsilon > 0$ and $1 \leq \tau \leq n$ are constants. Space is measured in *words*. The column *Working Space* lists the working space needed to construct a data structure, whereas the column *Space* lists the final space needed by a data structure.

1.3 Relationship Between Suffix Sorting and LCE Queries

The LCE-problem is to preprocess a text T such that subsequent LCE queries can be answered efficiently. Data structures for LCE and sparse suffix sorting are closely related, as shown in the following observation:

Observation 2. Given a data structure that answers LCE queries in $\mathcal{O}(\tau)$ time for $\tau > 0$, we can compute sparse suffix sorting for m positions in $\mathcal{O}(\tau m \lg m)$ time by inserting suffixes into a balanced binary search tree [21]. Conversely, given an algorithm computing the SSA and the SLCP of a text T of length n for m positions in $\mathcal{O}(f(n, m))$ time with $\mathcal{O}(m)$ words of space for a function f , we can construct a data structure in $\mathcal{O}(\max(f(n, m), n/m))$ time with $\mathcal{O}(m)$ words of space, answering LCE queries on T in $\mathcal{O}(n^2/m^2)$ time.

Proof. The first claim is trivial. For the second claim, we use the data structure of [5, Theorem 1a] that answers LCE queries in $\mathcal{O}(\tau)$ time. The data structure uses the SSA and SLCP values of those suffixes whose starting positions are in a difference cover sampling modulo τ . This difference cover consists of $\mathcal{O}(n/\sqrt{\tau})$ text positions, and can be computed in $\mathcal{O}(\sqrt{\tau})$ time [9]. We obtain the claimed bounds on time and space by setting $\tau := n^2/m^2$. \square

There has been a great interest in devising deterministic LCE data structures with trade-off parameters (see Fig. 1), or in compressed space [43, 32, 19]. One of the currently best data structures with a trade-off parameter is due to Tanimura et al. [42], using $\mathcal{O}(n/\tau)$ words of space and answering LCE queries in $\mathcal{O}(\tau \lg \min(\tau, n/\tau))$ time, for a trade-off parameter τ with $1 \leq \tau \leq n$. However, this data structure has a preprocessing time of $\mathcal{O}(n\tau)$, and is thus not helpful for sparse suffix sorting. We develop a new data structure for LCE with the following properties.

Theorem 3. There is a *deterministic* data structure using $\mathcal{O}(n/\tau)$ words of space that answers an LCE query $\ell := \text{lce}(i, j)$ for two text positions i and j with $1 \leq i, j \leq n$ on a text of length n in $\mathcal{O}(\lg^* n (\lg(\ell/\tau) + \tau^{\lg 2} / \log_{\sigma} n))$ time, where $1 \leq \tau \leq n$. We can build the data structure in $\mathcal{O}(n(\lg^* n + (\lg n)/\tau + (\lg \tau)/\log_{\sigma} n))$ time with additional $\mathcal{O}(\max(n/\lg n, \tau^{\lg 3} \lg^* n))$ words during construction.

The construction time of our data structure is upper bounded by $\mathcal{O}(n \lg n)$, and hence it can be constructed faster than the deterministic data structures in [42] when $\tau = \Omega(\lg n)$.

1.4 Outline of this Article

We start with Sect. 2 introducing the edit sensitive parsing, and giving a motivation for our hierarchical stable parsing whose description follows in Sect. 3. Section 3.3 shows the general techniques for answering LCE queries with the HSP tree. Subsequently, Sect. 4 introduces our algorithm for the sparse suffix sorting problem with an abstract data type *dynLCE* that supports LCE queries and a merging operation. The remainder of that section shows that the HSP tree from Sect. 3 fulfills all properties of a *dynLCE*; in particular, HSP trees support the merging operation. The last part of this article is dedicated to the study

on how the text space can be exploited with the HSP technique to improve the memory footprint. This leads us to truncated HSP trees with a merging operation that is tailored to working in text space (Sect. 5). With the truncated HSP trees we finally solve the sparse suffix sorting problem in the time and space as claimed in Thm. 1.

1.5 Preliminaries

Let Σ be an ordered alphabet of size σ whose characters are represented by integers. For a string $X \in \Sigma^*$, let $|X|$ denote the length of X . For a position $1 \leq i \leq |X|$ in X , let $X[i]$ denote the i -th character of X . For positions i and j with $1 \leq i, j \leq |X|$, let $X[i..j] = X[i]X[i+1] \cdots X[j]$. Given $T = XYZ$ with $X, Y, Z \in \Sigma^*$, X , Y and Z are called a **prefix**, **substring**, **suffix** of T , respectively. In particular, the suffix beginning at position i is denoted by $T[i..]$. A **period** of a string Y is a positive integer $p < |Y|$ such that $Y[i] = Y[i+p]$ for all integers i with $1 \leq i \leq |Y| - p$.

For a binary string $T \in \{0, 1\}^*$ we are interested in the operation $T.\text{rank}_1(j)$ that counts the number of '1's in $T[1..j]$. This operation can be performed in constant time by a data structure [22] that takes $o(|T|)$ extra bits of space, and can be constructed in time linear in $|T|$.

An **interval** $\mathcal{I} = [b..e]$ is the set of consecutive integers from b to e , for $b \leq e$. For an interval \mathcal{I} , we use the notations $\text{b}(\mathcal{I})$ and $\text{e}(\mathcal{I})$ to denote the beginning and the end of \mathcal{I} ; i.e., $\mathcal{I} = [\text{b}(\mathcal{I})..\text{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of \mathcal{I} ; i.e., $|\mathcal{I}| = \text{e}(\mathcal{I}) - \text{b}(\mathcal{I}) + 1$.

2 Edit Sensitive Parsing

The crucial technique used in this article is the so-called alphabet reduction. The alphabet reduction is used to partition a string deterministically into blocks. The first work introducing the alphabet reduction technique to the string context was done by Mehlhorn et al. [31]. They presented the so-called *signature encoding*. The signature encoding is derived from a tree coloring approach [17]. It supports string equality checks in the scenario where strings can be dynamically concatenated or split. In the same context, Sahinalp and Vishkin [38] studied the maximal number of characters to the left and to the right of a substring Z of Y such that changing one of these characters to the left or to the right of Z can affect how Z is parsed by the signature encoding of Y . In a later work, Alstrup et al. [1] enhanced signature encoding with additional queries like LCE. Recently, an LCE data structure using signature encoding in compressed space was shown by Nishimoto et al. [32]. A slightly modified version of signature encoding is proposed by Sakamoto et al. [39], showing that alphabet reduction can be used to build a grammar compressor whose approximation ratio to the size of the smallest grammar is $\mathcal{O}(\lg^* n \lg n)$.

A modified parsing was introduced by Cormode and Muthukrishnan [11]. They modified the parsing by restricting the block size from two up to three characters, and named their technique *edit sensitive parsing* (ESP). Initially used for approximating the edit distance with moves, the ESP technique has been found to be applicable to building self-indexes [41]. We stick to the ESP technique, because the size of the subtree of a node in the ESP tree is bounded. In this section, we first introduce the ESP technique, and then give a motivation for a modification of the ESP technique, which we call hierarchical stable parsing (HSP). Before that, we recall the alphabet reduction and the ESP trees.

2.1 Alphabet Reduction

Given a string Y in which no two adjacent characters are the same, i.e., $Y[i-1] \neq Y[i]$ for every integer i with $2 \leq i \leq |Y|$, we can partition Y (except at most the first $\lg^* \sigma$ positions) into **blocks** of size two or three with a technique called **alphabet reduction** [11, Section 2.1.1]. It consists of three steps (see also Fig. 2): First, it reduces the alphabet size to at most eight, in which every character has a rank from zero to seven. Subsequently, it substitutes characters with ranks four to seven with characters having a rank between zero and two. By doing so, it shrink the alphabet size to three. Finally, it identifies certain text positions as landmarks that determine the block boundaries.

For reducing the alphabet size, we assume that $\sigma \geq 9$, otherwise we skip this step. The task is to generate a surrogate string Z on the alphabet $\{0, 1, 2\}$ such that the entry $Z[i]$ depends only on the substring $Y[i..i + \lg^* \sigma]$, for $1 \leq i \leq |Y| - \lg^* \sigma$. To this end, we regard Y as an array of binary numbers, i.e., $Y[i][\ell] \in \{0, 1\}$ for an integer ℓ with $1 \leq \ell \leq \lceil \lg \sigma \rceil$. We create an array Z of length $|Y| - 1$ storing integers of the domain $[0..2^{\lceil \lg \sigma \rceil} - 1]$. For each text position i with $2 \leq i \leq |Y|$, we compare $Y[i]$ with $Y[i - 1]$: We compute $\ell := \text{lcp}(Y[i - 1], Y[i])$, and write $2\ell + Y[i][\ell + 1]$ to $Z[i]$ (remember that we treat $Y[i]$ as a binary string¹). By doing so, no two adjacent integers are the same in Z [11, Lemma 1]. Having computed Z , we recurse on Z until Z stores integers of the domain $\{0, \dots, 5\}$. Note that the alphabet cannot be reduced further with this technique, since $2^{\lceil \lg x \rceil} \geq x$ for every integer x with $2 \leq x \leq 6$. To obtain the final Z , we recurse at most $\lg^* \sigma$ times. Let r be the number of recursions. Then we have $|Y| = |Z| + r$.

If we skipped this step because of a small alphabet size ($\sigma \leq 8$), then we set $Z[i]$ to the rank of $Y[i]$ induced by the linear order of Σ (e.g., $Z[i] = 0$ if $Y[i]$ is the smallest character). Since $|Y| = |Z|$, we set r to zero.

To reduce the domain further, we iterate over the values $j = 3, \dots, 8$ in ascending order, substituting each $Z[i] = j$ with the lowest value of $\{0, 1, 2\}$ that does not occur in its neighboring entries ($Z[i - 1]$ and $Z[i + 1]$, if they exist). Finally, Z contains only numbers between zero and two.

In the final step we create the landmarks that determine the block boundaries. The landmarks obey the property that the distance between two subsequent landmarks is greater than one, but at most three. They are determined by local maxima and minima: First, each number $Z[i]$ that is a local maximum is made into a landmark. Second, each local minimum that is not yet neighbored by a landmark is made into a landmark.

Finally, we create blocks by associating each position in Z with its closest landmark. Positions associated with the same landmark are put into the same block. As a tie breaking rule we favor the right landmark in case that there are two closest landmarks. The last thing to do is to map each block covering $Z[i..j]$ to $Y[i + r..j + r]$.

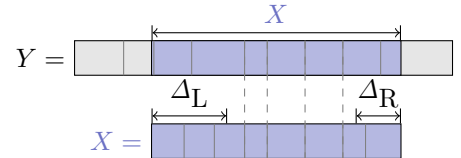
The tie breaking rule can cause a problem when $Z[1]$ and $Z[3]$ are landmarks, i.e., the leftmost block contains only one character. We circumvent this problem by fusing the blocks of the first and second landmark to a single block. If this block covers four characters, we split it evenly.

Altogether, the alphabet reduction needs $\mathcal{O}(|Y| \lg^* \sigma)$ time, since we perform $r \leq \lg^* \sigma$ reduction steps, while determining the landmarks and computing the blocks take $\mathcal{O}(|Y|)$ time. The steps are summarized in the following lemma:

Lemma 4. Given a string Y in which no two adjacent characters are the same, the alphabet reduction applied on Y partitions Y into blocks, except at most $\lceil \lg^* \sigma \rceil$ positions at the left. It runs in $\mathcal{O}(|Y| \lg^* \sigma)$ time.

The main motivation of introducing the alphabet reduction is the following lemma that shows that applying the alphabet reduction on a text Y and on a pattern X generates the same blocks in X as in all occurrences of X in Y , except at the left and right borders of a specific length:

Lemma 5 ([11, Lemma 4]). Given a substring X of a string Y in which no two adjacent characters are the same, the alphabet reduction applied to X alone creates the same blocks as the blocks representing the substring X in Y , except for at most $\Delta_L := \lceil \lg^* \sigma \rceil + 5$ characters at the left border, and $\Delta_R := 5$ characters at the right border.



Given a block β , we call the substring $Y[b(\beta) - \Delta_L..e(\beta) + \Delta_R]$ the **local surrounding** of β , if it exists (i.e., $b(\beta) - \Delta_L \geq 1$ and $e(\beta) + \Delta_R \leq |Y|$). Blocks whose local surroundings exist are also called **surrounded**. A consequence of Lemma 5 is the following: Given that X is the local surrounding of a surrounded block β , then the blocking of every occurrence of X in Y is the same, except at most Δ_L and Δ_R characters at the left and right borders, respectively. We conclude that the blocking of every occurrence of X has a block $X[1 + \Delta_L.. \Delta_L + |\beta|]$ that is equal to $Y[b(\beta)..e(\beta)]$ (see Fig. 3).

¹Fix an arbitrary rule whether $Y[i][1]$ is the least significant or most significant bit.

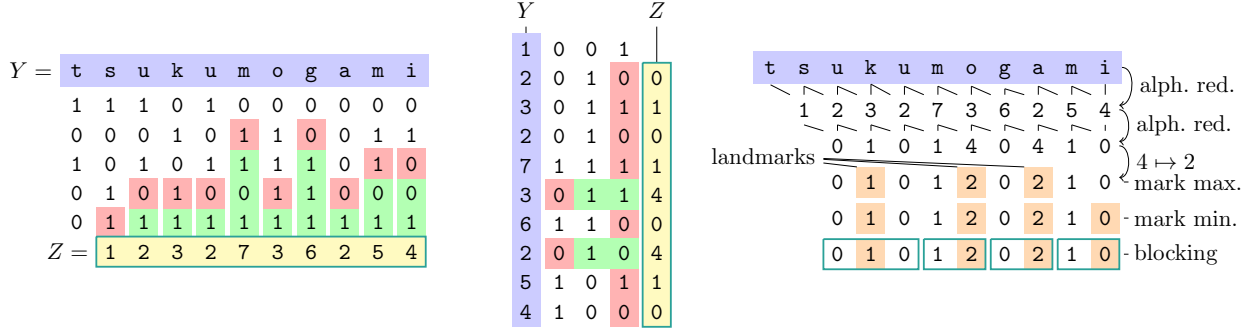


Figure 3: *Left:* Surrounded block β with local surrounding X contained in a string Y . *Right:* Occurrences of the local surrounding X of a surrounded block β in the string Y , which is partitioned into blocks (gray rectangles) by the edit sensitive parsing. Although the occurrences of X can be differently blocked at their borders, they all have a block equal to β in common.

2.2 Edit Sensitive Parsing

Whenever a string Y contains a repetition of a character at two adjacent positions, we cannot parse Y with the alphabet reduction. A solution is to additionally use an auxiliary parsing specialized on repetitions of the same character. With this auxiliary parsing, we can partition Y into substrings, where each substring is either parsed with the alphabet reduction, or with the auxiliary parsing. It is this auxiliary parsing where the aforementioned signature encoding and the ESP technique differ. The main difference is that the ESP technique restricts the lengths of the blocks: It first identifies so-called **meta-blocks** in Y , and then further refines these meta-blocks into blocks of length 2 or 3. The meta-blocks are created in the following 3-stage process (see also Fig. 4 for an example):

- (1) Identify maximal regions of repeated characters (i.e., maximal substrings of the form c^ℓ for $c \in \Sigma$ and $\ell \geq 2$). Such substrings form the **type 1** meta-blocks.
- (2) Identify remaining substrings of length at least 2 (which must lie between two **type 1** meta-blocks). Such substrings form the **type 2** meta-blocks.
- (3) Every substring not yet covered by a meta-block consists of a single character and cannot have **type 2**

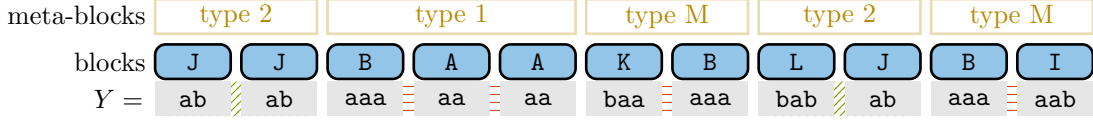


Figure 4: ESP of the string $Y = \text{ababaaaaaaabaaaaabababaaaaaab}$. The string is divided into blocks represented by the rectangular boxes at the bottom. Each block gets assigned a new character represented by the capital letters in the rounded boxes. The white rectangular boxes on the top level represent the meta-blocks that group the blocks. The blocks are connected with horizontal lines if they belong to a repeating meta-block, or by diagonal lines if they belong to a **type 2** meta-block.

meta-blocks as its neighbors. Such characters are fused with a neighboring meta-block. The meta-blocks emerging from this fusing are called **type M** (mixed).

Meta-blocks of **type 1** and **type M** are collectively called *repeating meta-blocks*. For (3), we are free to choose whether a remaining character should be fused with its preceding or succeeding meta-block (both meta-blocks are repeating). We stick to the following tie breaking rule²:

Rule M: Fuse a remaining character $Y[i]$ with its succeeding meta-block, or, if $i = |Y|$, with its preceding meta-block.

Meta-blocks are further partitioned into **blocks**, each containing two or three characters from Σ . Blocks inherit the type of the meta-block they are contained in. How the blocks are partitioned depends on the type of the meta-block:

Repeating meta-blocks. A repeating meta-block is partitioned greedily: create blocks of length three until there are at most four, but at least two characters left. If possible, create a single block of length two or three; otherwise (there are four characters remaining) create two blocks, each containing two characters.

Type-2 meta-blocks. A **type 2** meta-block μ is partitioned into blocks in $\mathcal{O}(|\mu| \lg^* \sigma)$ time by the alphabet reduction (Lemma 4). A block β generated by the alphabet reduction is determined by the characters $Y[\max(\mathbf{b}(\beta) - \Delta_L, \mathbf{b}(\mu)).. \min(\mathbf{e}(\beta) + \Delta_R, \mathbf{e}(\mu))]$ due to Lemma 5. Given the number of reduction steps r in Sect. 2.1, the alphabet reduction does not create blocks for the first r characters of each meta-block. The ESP technique blocks the first r characters in the same way as a repeating meta-block. The border case $r = 1$ (one character remaining) is treated by fusing the remaining character with the first block created by the alphabet reduction, possibly splitting this block in the case that its size is four.

A block is called *repetitive* if it contains the same characters. All blocks of a **type 1** meta-block and all blocks except at most the left- or rightmost block (these blocks can contain a fused character) in a **type M** meta-block are repetitive.

Let $\text{esp}: \Sigma^* \rightarrow (\Sigma^2 \cup \Sigma^3)^*$ denote the function that parses a string by the ESP technique. We regard the output of esp as a string of blocks.

2.3 Edit Sensitive Parsing Trees

Applying esp recursively on its output generates a context free grammar (CFG) as follows. Let $\langle Y \rangle_0 := Y$ be a string on an alphabet $\Sigma_0 := \Sigma$. The output of $\langle Y \rangle_h := \text{esp}^{(h)}(Y) = \text{esp}(\text{esp}^{(h-1)}(Y))$ is a sequence of blocks, which belong to a new alphabet Σ_h with $h \geq 1$. We call the elements of Σ_h with $h \geq 1$ *names*, and use the term *symbol* for an element that is a name or a character. A block $\beta \in \Sigma_h$ contains a string of symbols with length two or three ($\in \Sigma_{h-1}^2 \cup \Sigma_{h-1}^3$). We maintain an injective dictionary $\mathfrak{D}: \Sigma_h \rightarrow \Sigma_{h-1}^2 \cup \Sigma_{h-1}^3$

²The original version [11] prefers the left meta-block.

ESP Dictionary			
		Rule	$string(\cdot)$
Common Dictionary		A \rightarrow aa	a^2
		B \rightarrow aaa	a^3
		C \rightarrow AA	a^4
		D \rightarrow BB	a^6
		E \rightarrow BBB	a^9
		F \rightarrow DD	a^{12}
		G \rightarrow NN	$(ba)^2$
		H \rightarrow NNN	$(ba)^3$
		M \rightarrow CG	$a^4(ba)^2$
		O \rightarrow ANN	$a^2(ba)^2$
		R \rightarrow JJJ	$(ab)^3$
		HSP Dictionary	
		Rule	$string(\cdot)$
		$a_2 \rightarrow$ aa	a^2
		$a_3 \rightarrow$ aaa	a^3
		P \rightarrow a_3 J	a^4b
		Q \rightarrow a_3 I	a^5b

Figure 5: Names of the ESP (Sect. 2.2) and HSP (Sect. 3) nodes stored in the global dictionary of our examples. The common dictionary contains all names that are used by both ESP and HSP. Each name occurs on the left side only once across all dictionaries.

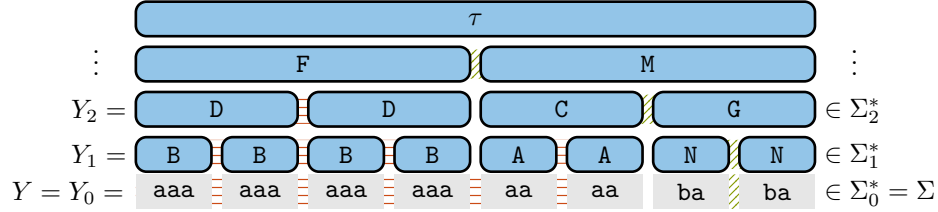


Figure 6: The ESP tree of the string $Y = \text{aaaaaaaaaaaaaaaaababa}$. Like in Fig. 4, nodes belonging to the same meta-block are connected by horizontal (repeating meta-block) or diagonal (**type 2** meta-block) lines.

to map a block to its symbols. The dictionary entries are of the form $\beta \rightarrow xy$ or $\beta \rightarrow xyz$, where $\beta \in \Sigma_h$ and $x, y, z \in \Sigma_{h-1}$. We write $\mathfrak{D}(X) := \mathfrak{D}(X[1]) \cdots \mathfrak{D}(X[|X|]) \in \Sigma_{h-1}^*$ for $X \in \Sigma_h^*$. Each block on height h is contained in a meta-block μ on height $h-1$, which is equal to a substring $\langle Y \rangle_{h-1}[i..j] \in \Sigma_{h-1}^*$. We call $\langle Y \rangle_{h-1}[i..j] \in \Sigma_{h-1}^*$ the **symbols** of μ . Since each application of esp reduces the string length by at least one half, there is an integer k with $k \leq \lg |Y|$ such that $\langle Y \rangle_k = esp(\langle Y \rangle_{k-1})$ is a single block $\tau \in \Sigma_k$. We write $\mathcal{V} := \bigcup_{1 \leq h \leq k} \Sigma_h$ for the set of names in $\langle Y \rangle_1, \langle Y \rangle_2, \dots, \langle Y \rangle_k$. The CFG for Y is represented by the non-terminals (i.e., the names) \mathcal{V} , the terminals Σ_0 , the dictionary \mathfrak{D} , and the start symbol τ . This grammar exactly derives Y .

Throughout this article, we comply with the convention to write symbols, i.e., characters ($\in \Sigma_0$) and names ($\in \Sigma_h$, $h \geq 1$), in typewriter font; characters and names are written in lower and upper cases, respectively. All examples use the same dictionary such that reappearing names are identical (see Fig. 5 for the used dictionary). Names restricted to a particular figure can be written with Greek letters (a necessity due to the limitation of having only 26 letters in the English alphabet).

The **ESP tree** $ET(Y)$ of a string Y is the derivation tree of the CFG defined above. Its root node is the start symbol τ . The nodes on height h are $\langle Y \rangle_h$ for each height $h \geq 1$. In particular, the leaves are $\langle Y \rangle_1$. Each leaf refers to a substring in Σ_0^2 or Σ_0^3 . The **generated substring** of a node $\langle Y \rangle_h[i]$ is the substring of Y generated by the symbol $\langle Y \rangle_h[i]$ (applying the h -th iterate of \mathfrak{D} to $\langle Y \rangle_h[i]$, yields a substring of Y , i.e., $\mathfrak{D}^{(h)}(\langle Y \rangle_h[i]) \in \Sigma^*$). We denote the generated substring of $\langle Y \rangle_h[i]$ by $string(\langle Y \rangle_h[i])$. For instance, in Fig. 6, $string(M) = \text{aaaababa}$. A node v on height h is said to be **built** on $\langle Y \rangle_{h-1}[b..e]$ iff $\langle Y \rangle_{h-1}[b..e]$ contains the children of v . Like with blocks, nodes inherit the type of the meta-block on which they are

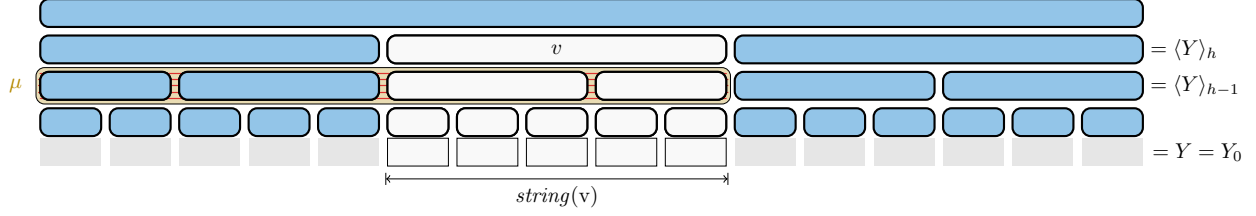


Figure 7: $\langle Y \rangle_h$ with a highlighted node v . The subtree rooted at v is depicted by the white, rounded boxes. The generated substring $string(v)$ of v is the concatenation of the white rectangular blocks on the lowest level in the picture. The meta-block μ , on which v is built, is the rounded rectangle covering the children of v and all nodes connected by a horizontal hatching on height $h - 1$.

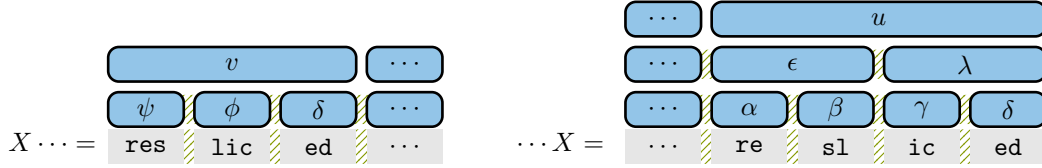


Figure 8: Excerpts of $ET(X \dots)$ (left) and $ET(\dots X)$ (right) with $X := \text{resliced}$. Under the assumption that $\lg^* \sigma = 8$, the common substring X can be blocked differently in both trees (depending on the characters preceding X in the right figure).

built. An overview of the definitions is given in Fig. 7.

In what follows, we present two shortcomings of the ESP trees. The first is that nodes with different names can have the same generated substring, i.e., $\mathfrak{D}^{(h)} : \Sigma_h \rightarrow \Sigma_0^*$ is not injective for $h \geq 2$ in general. The second is that it is not straight-forward to see which nodes of $ET(Y)$ and $ET(Z)$ are equal when Y is a substring of Z . Both cause problems when comparing subtrees of two nodes, which we later do for answering LCE queries.

Given two nodes u and v , it holds that $string(u) = string(v)$ if their names are equal. However, the other way around is not true in general. With $string(u) = string(v)$, it is not even assured that u and v are nodes on the same height. Suppose that Σ is a large alphabet with $\lg^* \sigma = 8$, and that $X := \text{resliced}$ occurs in the text that we parse with ESP (see Fig. 8). We parse an occurrence of X either (a) with the alphabet reduction if it is within a **type 2** meta-block, or (b) greedily if it is at the beginning of a **type 2** meta-block. In the former case (a), we apply the alphabet reduction and end at a reduced alphabet with the characters $\{0, 1, 2\}$. Suppose that this occurrence of X is reduced to the string in superscript of resliced . Then ESP creates the four blocks re|sl|ic|ed , whose boundaries are determined by the alphabet reduction. Further suppose that an application of esp creates two nodes of these blocks, which are put into a node u by an additional parse such that $string(u) = X$. In the latter case (b), ESP creates the three blocks res|lic|ed greedy. Suppose that an additional parse puts these blocks in a node v such that $string(v) = X$. Although $string(v) = string(u)$, the children of both nodes have different names, and therefore, both nodes cannot have the same name.

The second shortcoming is that it is not clear how to transfer the property of the alphabet reduction described in Lemma 5 from blocks to nodes. Given a substring Y of a string Z , the task is to analyze whether a node $\langle Y \rangle_h[i]$ is also present in the tree $ET(Z)$, i.e., we analyze changes of a node $\langle Y \rangle_h[i]$ when prepending or appending (pre-/appending) characters to Y . For the sake of analysis, we distinguish the two terminologies *block* and *node*, although a node is represented by a block: When we analyze a block in $esp(X) \in \Sigma_h^*$ for a string $X \in \Sigma_{h-1}^*$, we let X to be subject to pre-/appending characters of Σ_{h-1} , whereas when we analyze a node $\langle Y \rangle_h[i]$ on a height h of $ET(Y)$ of a string $Y \in \Sigma^*$, we let Y to be subject to pre-/appending

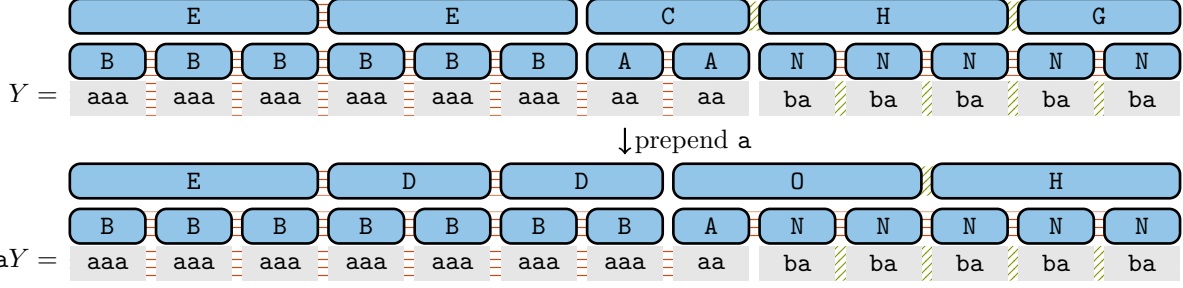


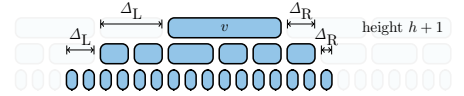
Figure 9: Excerpt of $ET(Y)$ and $ET(aY)$ (higher nodes omitted), where $Y = a^{9k+4}(ba)^{3k-1} = a^{22}(ba)^5$ for $k = 2$. For all $k \geq 2$, there is a unique node in $\langle Y \rangle_2$ with the name C . This name does not appear in $ET(aY)$.

characters of Σ . In this terminology, a block in $esp(X)$ is only determined by X , whereas $\langle Y \rangle_h[i]$ is not only determined by $esp^{(h-1)}(Y) \in \Sigma_{h-1}^*$, but also by Y itself. The difference is that a surrounded **type 2** block of $esp(X)$ cannot be changed by pre-/appending characters to X due to Lemma 5, whereas we fail to find integers $\Delta_{L,h}$ and $\Delta_{R,h}$ such that a **type 2** node on height h built on $\langle Y \rangle_{h-1}[\Delta_{L,h}.. \Delta_{R,h}]$ cannot be changed by pre-/appending characters to Y . That is because the names inside $\langle Y \rangle_{h-1}$ and $\langle aY \rangle_{h-1}$ for $h \geq 2$ can differ at arbitrary positions. This can be seen in the following example: When parsing the string $Y := a^{9k+4}(ba)^{3k-1}$ with the names defined in Fig. 5, we obtain $esp(esp(Y)) = esp(B^{3k}AAN^{3k-1}) = E^kCH^{k-1}G$. Let us focus on the unique occurrence of the name C , which is depicted in Fig. 9 for $k = 2$. On the one hand, there is a block representing the name C on height two. This block is surrounded for a sufficiently large k . Even for $k \geq 1$, it is easy to see that there is no way to change the name of this block by pre-/appending characters to the string $B^{3k}AAN^{3k-1}$. On the other hand, there is a unique node in $ET(Y)$ with name C on height two. Regardless of the value of k , prepending a to Y changes the name of v : $esp(esp(aY)) = esp(B^{3k+1}AN^{3k-1}) = E^{k-1}DDOH^{k-1}$. Nevertheless, we introduce the notion of surrounded nodes, since they are helpful to find rules that determine those nodes that cannot be changed by pre-/appending characters.

Surrounded Nodes. Analogously to blocks we classify nodes as surrounded when they are neighbored by sufficiently many nodes: A leaf is called surrounded iff its generated substring is surrounded.

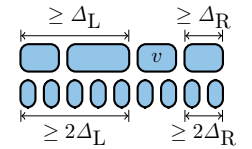
The local surrounding of a leaf is the local surrounding of the block represented by the leaf. Given an internal node v on height $h+1$

($h \geq 1$) whose children are $\langle Y \rangle_h[\beta]$, the **local surrounding** of v is the union of the nodes $\langle Y \rangle_h[b(\beta) - \Delta_L..e(\beta) + \Delta_R]$ and the local surrounding of each node in $\langle Y \rangle_h[b(\beta) - \Delta_L..e(\beta) + \Delta_R]$. If all nodes in the local surrounding of v are surrounded, we say that v is **surrounded**. Otherwise, we say that v is **non-surrounded**.



Lemma 6. There are at most $\Delta_L + \Delta_R$ many non-surrounded nodes on each height, summing up to $\mathcal{O}(\lg^* n \lg n)$ non-surrounded nodes in total.

Proof. We show that a node v on height h is surrounded if it has Δ_L preceding and Δ_R succeeding nodes. This is clear on height one by definition. Under the assumption that the claim holds for height $h-1$, v 's preceding (resp. succeeding) nodes have at least $2\Delta_L$ (resp. $2\Delta_R$) children in total, where at least the Δ_L rightmost nodes (resp. Δ_R leftmost nodes) are surrounded by the assumption. Hence, v is surrounded. \square



The above example contrasting blocks and nodes reveals that the property for surrounded blocks as shown on the right side of Fig. 3 cannot be transferred to surrounded nodes directly, since a surrounded node depends not only on its local surrounding, but also on the nodes on which it is built. Despite this discovery, we show that surrounded nodes can help us to create rules that are similar to Lemma 5.

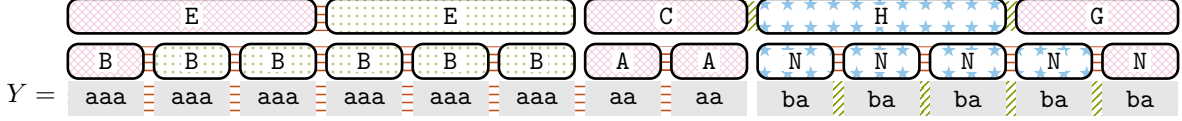


Figure 10: $\text{ET}(Y)$ of Fig. 9 with fragile, semi-stable and stable nodes highlighted. The fragile nodes are cross-hatched, the semi-stable nodes are dotted, and the stable nodes have stars attached. The leftmost nodes of the tree change their names when prepending a **b**. When prepending **a**'s, we observe that the children of the node with name **C** change. Assuming that $\Sigma = \{a, b\}$ (and hence $|\Sigma| = 2$), only the rightmost node of the meta-block containing nodes with name **N** is fragile.

2.4 Fragile and Stable Nodes in ESP Trees

We now analyze which nodes of $\text{ET}(Y)$ are still present in $\text{ET}(XYZ)$ for all strings X and Z . A node $\langle Y \rangle_h[j]$ in $\text{ET}(Y)$ at a height h is said to be **stable** iff, for *all* strings X and Z , there exists a node $\langle XYZ \rangle_h[k]$ in $\text{ET}(XYZ)$ with the same name as $\langle Y \rangle_h[j]$ and $|X| + \sum_{i=1}^{j-1} |\text{string}(\langle Y \rangle_h[i])| = \sum_{i=1}^{k-1} |\text{string}(\langle XYZ \rangle_h[i])|$. We also consider repeating nodes that are present with slight shifts; a non-stable repeating node $\langle Y \rangle_h[j]$ in $\text{ET}(Y)$ is said to be **semi-stable** iff, for *all* strings X and Z , there exists a node $\langle XYZ \rangle_h[k]$ in $\text{ET}(XYZ)$ with the same name as $\langle Y \rangle_h[j]$ and $\sum_{i=1}^{k-1} |\text{string}(\langle XYZ \rangle_h[i])| - |S| < |X| + \sum_{i=1}^{j-1} |\text{string}(\langle Y \rangle_h[i])| < \sum_{i=1}^{k-1} |\text{string}(\langle XYZ \rangle_h[i])| + |S|$, where $S = \text{string}(\langle Y \rangle_h[j]) = \text{string}(\langle XYZ \rangle_h[k])$.

Nodes that are neither stable nor semi-stable are called **fragile**. By definition, the children of the (semi-)stable nodes (resp. fragile nodes) are also (semi-)stable (resp. fragile). Figure 10 shows an example, where all three types of nodes are highlighted. The rest of this section studies how many fragile nodes exist in $\text{ET}(Y)$.

As a warm-up, we first restrict the ESP tree construction on strings that are square-free. A string Y is **square-free** iff there is no substring of Y occurring consecutively twice. Since a name of the ESP tree is determined by its generating substring, $\text{ET}(Y)$ cannot contain two consecutive occurrences of the same name on any height. We conclude that $\text{ET}(Y)$ has no repeating nodes, i.e., it consists only of **type 2** nodes. When studying the stability of **type 2** nodes, the following lemma is especially useful:

Lemma 7 ([11, Lemma 8]). A **type 2** node is stable if (a) it is surrounded and (b) its local surrounding does not contain a fragile node.

With Lemma 7 we immediately obtain:

Lemma 8. Given a square-free string Y , a fragile node of $\text{ET}(Y)$ is a non-surrounded node.

Proof. According to Lemma 7, we can bound the number of fragile nodes by the number of those nodes that do not satisfy the conditions in Lemma 7. Since $\text{ET}(Y)$ only contains **type 2** nodes, we can show that a fragile node is non-surrounded inductively for all heights of the ESP tree: Since leaves do not contain any nodes in their subtrees, surrounded leaves are stable due to Lemma 5. Therefore, the claim holds for $h = 1$. By definition, a node v on height h is surrounded if its local surrounding S on height $h - 1$ is surrounded. Given that the claim holds for $h - 1$, a node in S can only be fragile if it is not surrounded. This concludes that v can be fragile only if it is not surrounded. \square

Combining Lemma 8 with Lemma 6 yields:

Corollary 9. The number of fragile nodes of an ESP tree built on a square-free string of length n is $\mathcal{O}(\lg^* n \lg n)$. On each height, it contains $\mathcal{O}(\lg^* n)$ fragile nodes.

In Appendix A, we show that Cor. 9 cannot be generalized for arbitrary strings. There we show that the ESP technique changes $\Omega(\lg^2 n)$ nodes when changing a single character of a specific example string.

A new upper bound. With the examples in the appendix, we conclude that the $\mathcal{O}(\lg^* n \lg n)$ -bound on the number of fragile nodes for square-free strings (Lemma 8) does not hold for general strings. To obtain a

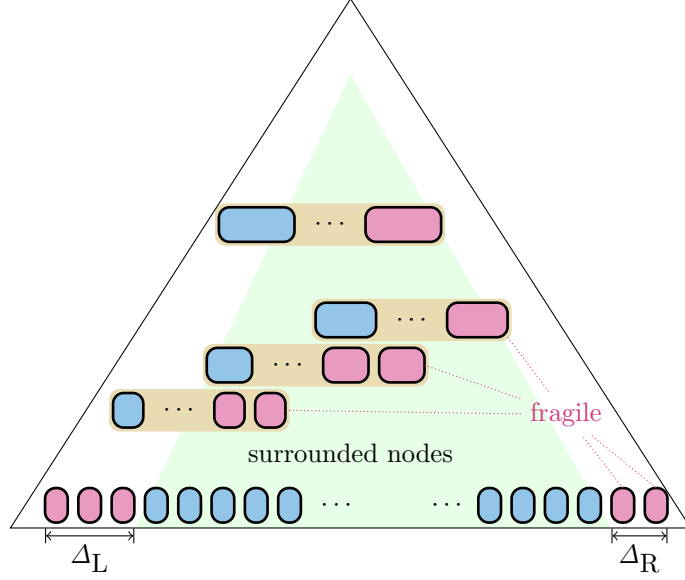


Figure 11: Division of an ESP tree in surrounded and fragile nodes. The surrounded nodes form an inner cone. Neighboring fragile blocks can appear in the non-surrounded areas. On each height, the ESP tree can have a constant number of fragile surrounded nodes that do not have fragile nodes in their subtrees.

general upper bound (we stick again to Rule M), we include the repeating meta-blocks in our study of fragile nodes. Fragile nodes can now be surrounded (trees of square-free strings do not have fragile surrounded nodes according to Lemma 8). Remembering that a node is fragile if it has a fragile child, a consequence is that a fragile **type 2** node is not necessarily non-surrounded (e.g., one of its children can be a fragile surrounded repeating node). Figure 11 sketches the possible occurrences of fragile surrounded nodes. A first result on a special case is given in the following lemma:

Lemma 10. A surrounded node v is contained in the local surroundings of $\mathcal{O}(\lg^* n \lg n)$ nodes. Given that all those nodes are of **type 2**, a change of v causes $\mathcal{O}(\lg^* n \lg n)$ name changes.

Proof. We follow [11, Proof of Lemma 9]: We count the number of nodes that contain v in its local surrounding. Given that v is a node on height i and u is v 's parent, then there are at most $\Delta_R/2 \leq \Delta_R$ nodes preceding u and $\Delta_L/2 \leq \Delta_L$ nodes succeeding u that have v in its local surrounding. We count one on height i , and $(\Delta_L + \Delta_R + 1)/2$ on height $i + 1$. Since the counted nodes on height $i + 1$ are consecutive, there are at most $(\Delta_L + \Delta_R + 1)/2$ nodes that are all parents of the counted nodes on height $i + 1$. Consequently, there are at most $(\Delta_L + \Delta_R + 1)/2 + \Delta_L + \Delta_R$ nodes on height $i + 2$ that have v in their local surroundings. Iterating over all heights gives an upper bound of $(\Delta_L + \Delta_R + 1) \sum_{h=0}^{\lg n-i} 1/2^h \leq 2(\Delta_L + \Delta_R + 1)$ nodes on each height. \square

Second, we narrow down the fragile blocks in repeating meta-blocks. The first block (cf. Fig. 12) and the two rightmost blocks (cf. Fig. 13) of a repeating meta-block can be fragile. Due to the greedy parsing, all other blocks of a repeating meta-block are (semi-)stable. A repeating meta-block containing fragile *surrounded* blocks needs to start very early or end within the last symbol, as can be seen by the following lemma:

Lemma 11. A repeating meta-block μ of $\text{esp}(Y)$ with $\mathbf{b}(\mu) \geq 4$ and $\mathbf{e}(\mu) \leq |Y| - 2$ cannot contain a fragile block.

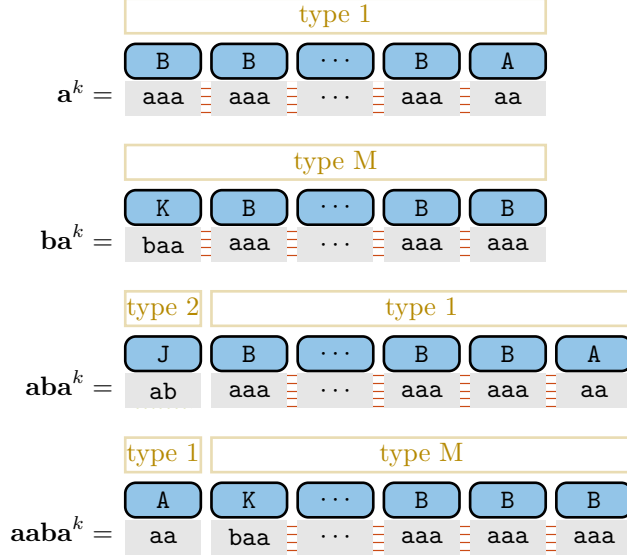


Figure 12: Prepending the string aab to the text a^k character by character. Each step is given as a row, in which we additionally computed the ESP of the current text. The last row shows an example, where a former **type 1** meta-block changes to **type M**, although it is right of a **type 2** meta-block. Here, $k \bmod 3 = 2$.

Proof. Since $b(\mu) \geq 4$, there are at least three symbols before μ that are assigned to one or more other meta-blocks. When prepending symbols, those meta-blocks can change, absorbing the new symbols or giving the leftmost symbol away to form a **type 2** meta-block. In neither case, they can affect the parsing of μ , since μ is parsed greedily. Similarly, the succeeding meta-blocks of μ keep μ 's blocks from changing when appending symbols. See Fig. 14 for a sketch. \square

Corollary 12. The edit sensitive parsing introduces at most two fragile surrounded blocks. These blocks are the two rightmost blocks of a repeating meta-block whose leftmost block is not surrounded.

Lemma 13. Changing the symbol in a substring of $\langle Y \rangle_{h-1}$ on which a repeating node on height h is built changes $\mathcal{O}(1)$ names on height h .

Proof. Let u be a repeating node on height h . Since it is repeating, it is built on a substring $X := \langle Y \rangle_{h-1}[b(X)..e(X)]$ of a repeating meta-block $\mu = \langle Y \rangle_{h-1}[b(\mu)..e(\mu)]$ with $\mathfrak{D}(u) = X$. Now change a symbol in X , say $\langle Y \rangle_{h-1}[i_u]$ with $b(X) \leq i_u \leq e(X)$. This causes the name of u to change. Additionally, it causes the meta-block μ to split into a repeating meta-block $\langle Y \rangle_{h-1}[b(\mu)..i_u - 1]$ and a **type M** meta-block $\langle Y \rangle_{h-1}[i_u..e(\mu)]$, causing the names of the two rightmost nodes built on the new meta-blocks to change. Altogether, there are $\mathcal{O}(1)$ name changes on height h . \square

An easy generalization of Lemma 13 is that changing k consecutive nodes on height $h - 1$ that are children of repeating nodes on height h changes $\mathcal{O}(k)$ names on height h . With Lemma 13, the following lemma translates the result of Cor. 12 for blocks to nodes:

Lemma 14. The ESP tree $ET(Y)$ of a string Y of length n has $\mathcal{O}(\lg^2 n \lg^* n)$ fragile nodes, and $\mathcal{O}(h \lg^* n)$ fragile nodes on height h .

Proof. While computing $\langle Y \rangle_{h+1}$ from $\langle Y \rangle_h$, the ESP technique introduces $\mathcal{O}(1)$ fragile surrounded blocks according to Cor. 12. Each fragile surrounded block corresponds to a fragile surrounded node.

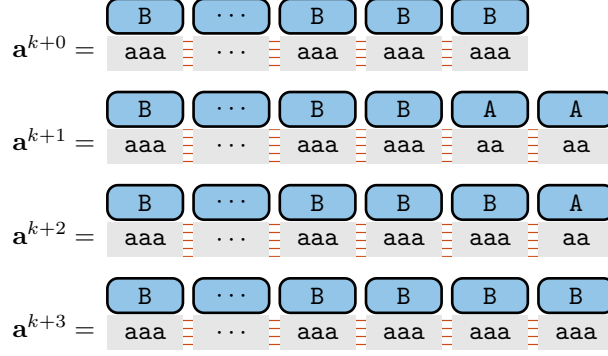


Figure 13: Greedy blocking of a **type 1** meta-block. The greedy blocking is related to the Euclidean division by three. The remainder $k \bmod 3$ is determined by the number of symbols in the last two blocks (here, $k \bmod 3 = 0$). In this example, the ESP technique creates a single, repeating meta-block on each input.

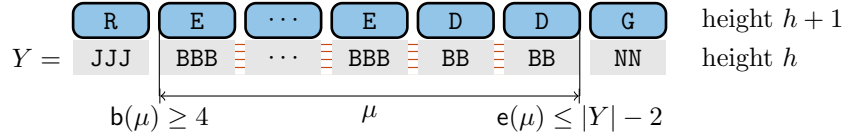


Figure 14: Setting of Lemma 11. According to Lemma 11, a meta-block μ in $\text{esp}(Y)$ of a string Y cannot contain a surrounded fragile block if $b(\mu) \geq 4$ and $e(\mu) \leq |Y| - 2$.

Similar to the proof of Lemma 8, we count all surrounded nodes as fragile whose local surrounding contains a fragile node. Lemma 10 shows that each introduced fragile surrounded block makes $\mathcal{O}(\lg^* n \lg n)$ nodes fragile. Although we considered only **type 2** nodes in Lemma 10, we can generalize this result for all fragile nodes with Lemma 13.

To sum up, there are $\mathcal{O}(h \lg^* n)$ fragile nodes on height h . Because $\text{ET}(X)$ has a height of at most $\lg n$, there are $\mathcal{O}(\lg^* n \sum_{h=1}^{\lg n} h) = \mathcal{O}(\lg^* n \lg^2 n)$ fragile nodes in total. \square

Showing that the number of fragile nodes is indeed larger than assumed makes ESP trees a more unfavorable data structure, since fragile nodes are cumbersome when comparing strings with ESP trees as done in [11]. Fortunately, we can restore the claimed number of $\mathcal{O}(\lg n \lg^* n)$ fragile nodes for a string of length n with a slight modification of the parsing, as shown in the following section.

3 Hierarchical Stable Parsing Trees

Our modification, which we call **hierarchical stable parsing** or **HSP**, augments each name with a **sur-name** and a **surname-length**, whose definitions follow: Given a name $Z \in \Sigma_h$, let h' with $0 \leq h' \leq h$ be the largest integer such that $\mathfrak{D}^{(h')}(Z)$ consists of the same symbol, say $\mathfrak{D}^{(h')}(Z) = Y^\ell \in \Sigma_{h-h'}^*$ for a symbol $Y \in \Sigma_{h-h'}$ and an integer $\ell \geq 1$. Then the surname and surname-length of Z are the symbol Y and the integer ℓ , respectively.³ For convenience, we define the surname of a character to be the character itself. Then all symbols in $\mathfrak{D}^{(j)}(Z)$ for every j with $1 \leq j \leq h'$ share the same surname with Z .

Having the surnames of the nodes at hand, we present the hierarchical stable parsing. It differs to ESP in how a string of names is partitioned into meta-blocks, whose boundaries now depend on the surnames: When factorizing a string of names into meta-blocks, we relax the check whether two names are equal; instead of

³By definition, the surname of Z is Z itself if $\ell = 1$.

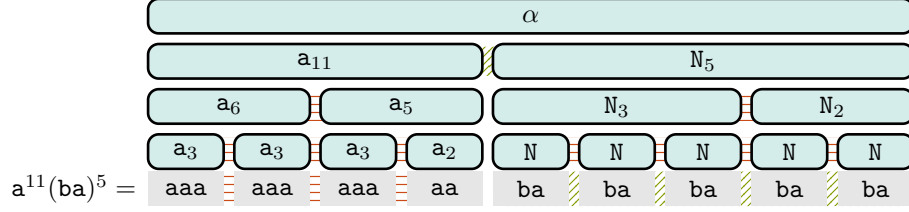


Figure 15: Hierarchical stable parsing. The repeating meta-blocks are determined by the surnames.

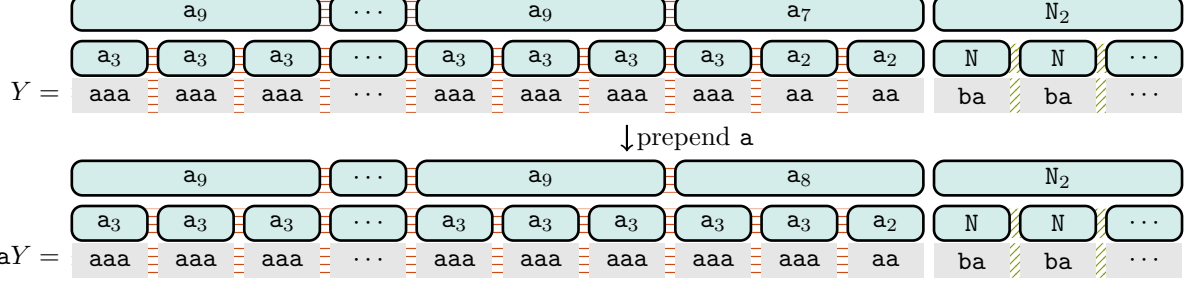


Figure 16: Excerpt of $\text{HT}(Y)$ and $\text{HT}(aY)$ (higher nodes omitted), where $Y = \mathbf{a}^k(\mathbf{ba})^{k'}$ with $k = 18 + 9^i + 7$ for an integer $i \geq 0$ and $k' \geq 2$ (cf. Fig. 9). The parsing of Y creates a repeating meta-block consisting of \mathbf{a}^k , and a **type 2** meta-block consisting of $(\mathbf{ba})^2$. For $k \geq 2$ it is impossible to modify the latter meta-block by prepending characters (bottom figure), since the parsing always groups adjacent nodes with the same surname into one repeating meta-block.

comparing names we compare by surnames.⁴ This means that we allow meta-blocks of **type 1** to contain different symbols as long as all symbols share the same surname. The other parts of the edit sensitive parsing defined in Sect. 2.2 are left untouched; in particular, the alphabet reduction uses the symbols as before. We write $\text{HT}(Y)$ for the resulting parse tree, called **HSP tree**, when the HSP technique is applied to a string Y . Figure 15 shows $\text{HT}(\mathbf{a}^{11}(\mathbf{ba})^5)$. In the rest of this article (and as shown in Fig. 15), we give a repetitive node with surname \mathbf{Z} and surname-length ℓ the name \mathbf{Z}_ℓ . We omit the surname-length if it is one (and thus, the label of a non-repetitive node is equal to its name). For the other nodes, we use the names of Fig. 5. We can do that because the name of a node can be identified by its surname and surname-length, as can be seen by the following lemma:

Lemma 15. The name of a node is uniquely determined by its surname and surname-length.

Proof. A node with surname-length one is not repetitive, and therefore, its name is equal to its surname. Given a repetitive node v with surname \mathbf{Z} and surname-length ℓ , there is a height h such that $\mathfrak{D}^{(h)}(v) = \mathbf{Z}^\ell$. For every height h' with $1 \leq h' \leq h$, $\mathfrak{D}^{(h')}(v)$ consists of the same symbol, and hence $\mathfrak{D}^{(h')}(v)$ is parsed greedily by HSP. This means that the iterated greedy parsing of the string \mathbf{Z}^ℓ determines the name of v . \square

3.1 Upper Bound on the Number of Fragile Nodes

The motivation of introducing the HSP technique becomes apparent with the three following facts:

Fact 1: Given that the surnames of the repetitive nodes in a repeating meta-block μ are w , the generated substring of each such repetitive node is a repetition of the form X^k with the same $X = \text{string}(w) \in$

⁴The check is relaxed since names with different surnames cannot have the same name.

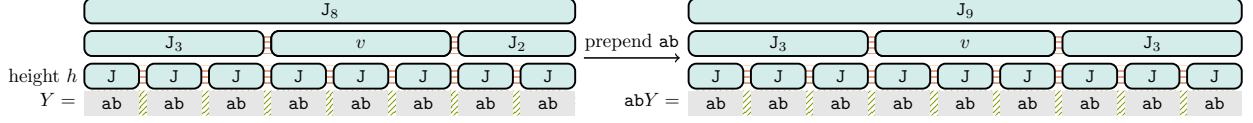


Figure 17: Comparison of $\text{HT}(Y)$ and $\text{HT}(\text{ab}Y) = \text{HT}(Y\text{ab})$, where $Y = (\text{ab})^8$. The node v with name J_3 is semi-stable. Its generated substring shifts with a length of $|\text{string}(J)| = 2$.

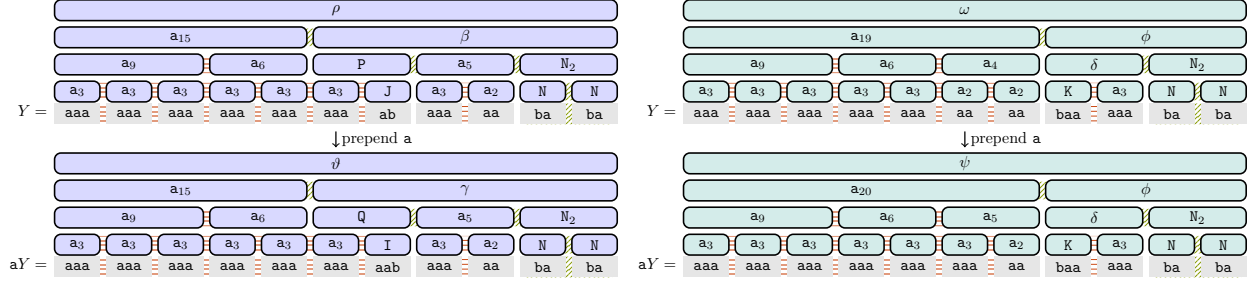


Figure 18: Impact of the tie breaking rule (Rule M) on emerging **typeM** nodes. A **typeM** node is created by fusing a single symbol with its sibling meta-block. Remember that Rule M prescribes to fuse the symbol with its *right* meta-block. To see why this rule is advantageous, the HSP trees on the *left* (resp. *right*) use the tie breaking rule choosing the *left* (resp. *right*) meta-block. While on the *right* side only the fragile nodes of the leftmost meta-blocks on each height differ after prepending **a** (e.g., the unique occurrence of a_4 changes to a_5), the change is more dramatical on the *left* side. In the top left tree, which is built on $Y = a^{19}ba^5(ba)^2$, the two rightmost nodes a_3 and J of the **typeM** meta-block on the bottom level are children of the leftmost node P of the right meta-block on the next level. Prepending the character **a** to Y (*bottom left*) changes the names of the nodes with names J and P to I and Q , respectively.

Σ^* (or $X = w$ in case $w \in \Sigma$), but with possibly different surname-lengths k (e.g., $\text{string}(N_3) = (\text{ba})^3$ and $\text{string}(N_2) = (\text{ba})^2$ in Fig. 15). Due to the greedy parsing of the repeating meta-blocks, the surname-lengths of the last two nodes in μ cannot be larger than the surname-lengths of the generated substrings of the other nodes (with the same surname) contained in μ . See Fig. 16 for an example when prepending a character to the input.

Fact 2: The shift of a semi-stable node is always a multiple of the length of its surname (recall that semi-stable nodes are defined like stable nodes, but with slight shifts, cf. Sect. 2.4): Let J be the surname of a semi-stable node $v \in \langle Y \rangle_h$ on height h . Given $J \in \Sigma_{h'}$ for a height h' with $h' \geq 0$, $\mathfrak{D}^{(h-h')}(v)$ is a repetition of the symbol J on height h' . A shift of v can only be caused by adding one or more J s to $\langle Y \rangle_{h'}$. In other words, the shift is always a multiple of $\mathfrak{D}^{(h')}(J)$. Figure 17 shows an example of a semi-stable node v .

Fact 3: A non-repetitive **typeM** block can be fragile only if it is non-surrounded. By definition, a repeating meta-block μ contains a non-repetitive block β iff μ is **typeM**. The block β can only be located at the beginning or ending of μ . Remembering Rule M, β 's none-repetitiveness is caused by

- fusing a symbol with its *succeeding* meta-block, or
- fusing the *last* symbol with its *preceding* meta-block.

In both cases, it is impossible that β is a surrounded block if $\mathbf{b}(\mu) \leq \Delta_L$. If β is surrounded, it is (semi-)stable due to Lemma 11. Note that with sticking to the choice made in Rule M, we also experience a more stable behavior like in Fig. 18.

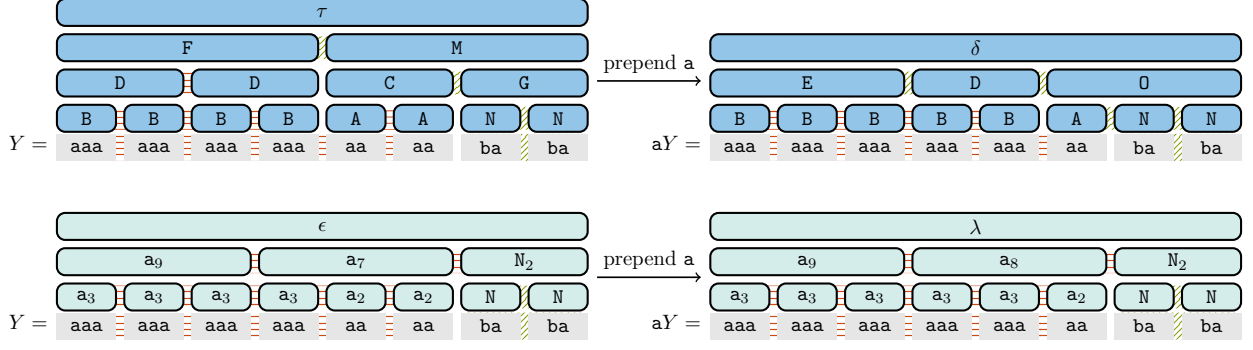


Figure 19: *Left*: $ET(Y)$ (*top*) and $HT(Y)$ (*bottom*) of the string Y defined in Fig. 6. *Right*: $ET(aY)$ (*top*) and $HT(aY)$. Unlike the two ESP trees on the top, the two HSP trees below share the same tree topology.

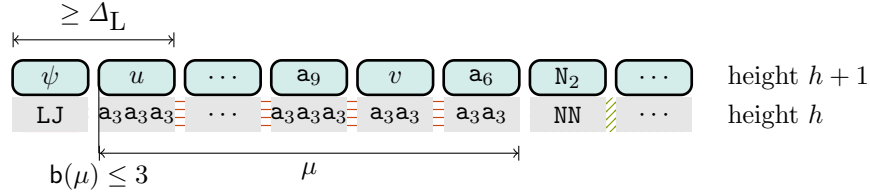


Figure 20: Setting of Cor. 16. According to Lemma 11, a meta-block μ can contain a surrounded fragile block if $b(\mu) \leq 3$ (cf. Fig. 14). In the figure, the node v is fragile, since prepending L changes its name. According to Cor. 16, there is a non-surrounded node u whose generated substring has the generated substring of v as a prefix.

These facts make the HSP technique more stable than the ESP technique, as can be seen in Fig. 19, for instance. In the following, we study the number of fragile surrounded nodes (like in Sect. 2.4 for the ESP trees), and show the invariant (Claim 3 in Lemma 18) that the generated substring of a fragile surrounded node is always the prefix of the generated substring of a name that is already stored in \mathfrak{D} . On block level, this is an easy conclusion of Lemma 11 and Facts 1 and 3.

Corollary 16. Given $n > 4$ and a repeating meta-block μ having a fragile surrounded block β , μ has at least one block preceding β that contains three symbols with the same surname. In particular, the leftmost of these preceding blocks is non-surrounded.

Proof. Since β is surrounded, the condition $|\mu| \geq \Delta_L - 2$ holds. By the definition of Δ_L in Lemma 5, $\Delta_L - 2 \geq 5$ for $n > 4$. Assuming that the repetitive blocks in μ have the surname Z , this means that there is at least one repetitive block γ with surname Z preceding β that contains three symbols of μ . But the fragile surrounded block β is also a repetitive block according to Fact 3. This means that the surname-length of β is at most as long as the surname-length of γ due to Fact 1, i.e., the generated substring of the node corresponding to β is a prefix of the generated substring of the node corresponding to γ . Let γ be the leftmost such block. Remembering that μ can start with a non-repetitive node in case that μ is of **type M**, it is not obvious that γ is non-surrounded. However, according to Lemma 11, $b(\mu) \leq 2$ must hold. This means that $b(\gamma) \leq 5 \leq \Delta_L$, so γ is non-surrounded. See Fig. 20 for a sketch (with $Z = a$). \square

In general, the aforementioned invariant does not hold for ESP trees, but is essential for the sparse suffix sorting in text space. There, our idea is to create an HSP or ESP tree on a newly found re-occurring substring. We would like to store the ESP tree in the space of one of those substrings, which we can do by truncating the tree at a certain height (removing the lower nodes), and changing the pointer of each

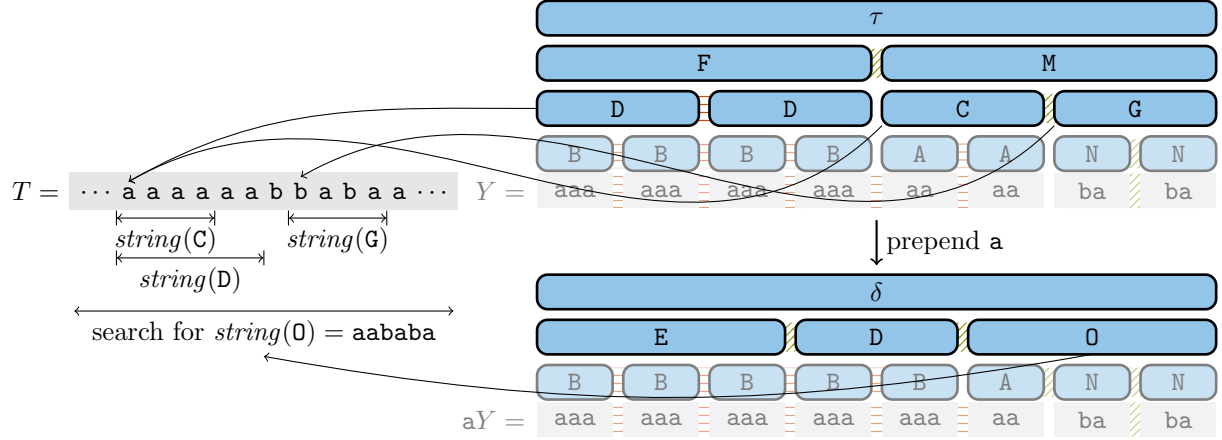


Figure 21: Problem with dynamic updates of ESP trees stored in text space. Truncated nodes are grayed out. Each leaf of the truncated trees is assigned a pointer to its generated substring, which are substrings of the text T (left). Suppose that we have built $ET(Y)$ (top right) on a substring Y of T (Y defined as in Fig. 19), and that the names D , C and G are already present in the dictionary (hence, they have different generated substrings). Further suppose that the space of Y in T has been overwritten. When prepending an a to $ET(Y)$ to form $ET(aY)$ (bottom right), the node G changes to 0 , for which we need to search its generated substring (assuming that 0 is not yet stored in the dictionary). The example can be elaborated such that G and 0 become surrounded nodes (prepend a^{9k} and append b^{9k} for a sufficiently large $k \geq 1$).

(new) leaf such that the name of a leaf refers to its generated substring that is found in the remaining text. Unfortunately, there is a problem when pre-/appending characters to enlarge the ESP tree, since a leaf could change its name such that its generated substring needs to be updated - which can be non-trivial if its generated substring refers to an already overwritten part of the text that is not present in the remaining text as a (complete) substring. Figure 21 demonstrates the problem when truncating ESP trees at height 2. Fortunately, the following lemmas restrict the problem of updating the generated substring when an HSP node is surrounded and fragile. We start with appending characters:

Lemma 17. There is no surrounded HSP node v whose name changes when appending characters.

Proof. Assume that v 's name changes on appending characters. Moreover, assume that v 's local surrounding does not contain a fragile node (otherwise swap v with this node). First, since there is no fragile node in v 's local surrounding, it has to be a repeating node according to Lemma 7. Second, according to Cor. 12, it has to be one of the last two nodes built on a repeating meta-block μ . But there is no way to change the names of the last two blocks of μ by appending characters unless these blocks are non-surrounded. So a surrounded node cannot have a node in its surrounding whose name changes when appending characters. \square

Lemma 18. Let v be a fragile surrounded node of an HSP tree. Then

Claim 1: v is a repetitive node,

Claim 2: pre-/appending characters cannot change v 's surname, and

Claim 3: the generated substring of v is always a prefix of the generated substring of an already existing node belonging to the same meta-block as v .

Proof. To show the lemma, let $n > \Delta_L + \Delta_R$, otherwise there are no surrounded nodes. There are two (non-exclusive) possibilities for a node to be fragile and surrounded:

- it belongs to the last two nodes built on a repeating meta-block (due to Cor. 12), or

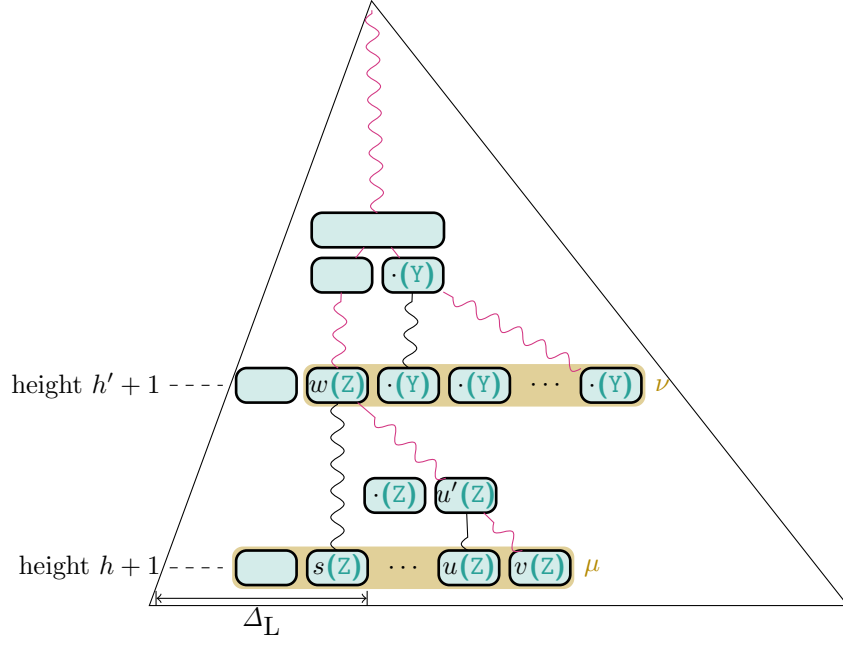


Figure 22: Sketch of the HSP tree used to show Lemma 18. In the sketch, we give the repetitive nodes of the meta-block ν the surname Y . Repetitive nodes are labeled with their surnames, which are put into parentheses.

- its subtree contains a fragile surrounded node, since by definition,
 - a node is fragile if it contains a fragile node in its subtree, and
 - all nodes in the subtree of a surrounded node are surrounded.

We iteratively show the claim for all heights, starting at the bottom: Let v be one of the *lowest* fragile surrounded nodes in $\text{HT}(Y)$ (*lowest* meaning that there is no fragile node in v 's subtree). Suppose that v is a node on height $h + 1$ with $h \geq 0$. Since there is no fragile surrounded node in v 's subtree, v is one of the last two nodes built on a repeating meta-block $\langle Y \rangle_h[\mu]$ (i.e., $Y[\mu]$ for $h = 0$). Due to Fact 3, Claim 1 holds for v ; let Z be its surname. Since v is fragile, $b(\mu) \leq 3$ must hold (otherwise we get a contradiction to Lemma 11). But since v is surrounded, there is a repetitive node u with surname Z preceding v that is built on three symbols ($\mathfrak{D}(u) \in \Sigma_h^3$) of μ due to Cor. 16. In particular, the leftmost repetitive node s of μ is not surrounded.

We only consider prepending a character (appending is already considered in Lemma 17). Assume that v 's name changes when prepending a specific character. By Fact 1, the HSP technique assigns a new name to v , but it does not change its surname (so Claim 2 holds for v). Additionally, $\text{string}(v)$ is a substring of $\text{string}(u)$, where u is one of v 's preceding nodes having the surname Z , and therefore Claim 3 holds for v . For example, let v be the node with name \mathbf{a}_7 in $\text{HT}(Y)$ of Fig. 16, then $\text{string}(v) = \mathbf{a}^7$, which is a prefix of $\text{string}(\mathbf{a}_9) = \mathbf{a}^9$. After prepending the character \mathbf{a} , v 's name becomes \mathbf{a}_8 with $\text{string}(v) = \mathbf{a}^8$. Still, $\text{string}(v)$ is a prefix of $\text{string}(\mathbf{a}_9)$.

Due to this behavior, the node v is always assigned to μ , regardless of what character is prepended. This means that it is only possible to extend or shorten μ on its left side, or equivalently, μ 's right end is *fixed*; the parsing of a meta-block succeeding μ cannot change. This means that the parsing assures that every surrounded node located to the right of $\langle Y \rangle_h[\mu]$ is (semi-)stable. We conclude that the claim holds for the heights $1, \dots, h + 1$.

Next, we show that the claim holds for all height $h + 2, \dots, h'$, where $h' + 1$ is the height of the lowest common ancestor w of s and v . Figure 22 gives a visual representation of the following observations: When following the nodes from v up to w , there is a path of ancestor nodes with surname Z. Except for w , each such ancestor node u' has a neighbor with surname Z. On changing the name of v , all nodes on the height of u' are unaffected, except u' . That is because the ancestor of s on the same height as u' is put with u' in the same repeating meta-block, which comprises all neighboring nodes with surname Z. By the analysis above, changing the name of u' cannot change the parsing of the other nodes on the same height. We conclude that the claim holds for the heights $h + 2, \dots, h'$.

Let us focus on the nodes on height $h' + 1$: The node w is not surrounded, because it contains the non-surrounded node s in its subtree. Having neighbors with different surnames, w is either blocked in a **type 2** or **type M** meta-block.

- In the former case (**type 2**), the analysis of Lemma 10 shows that w only affects the parsing of the non-surrounded nodes. There can be a non-surrounded meta-block on a height $h'' > h' + 1$ having a fragile surrounded node v' . But then v' cannot contain a fragile node (the descendants of w are the last fragile *surrounded* nodes, and w is non-surrounded). This means that we can apply the same analysis to v' as for v .
- In the latter case (**type M**), w is fused with a repeating meta-block to form a **type M** meta-block ν , changing the names of the leftmost and two rightmost nodes of ν , where the leftmost node is w . Assume that the two rightmost nodes of ν are fragile and surrounded (otherwise we conclude with the previous case that there are no fragile surrounded nodes on height $h' + 1$). Under this assumption, the rightmost nodes of ν are repeating nodes due to Fact 3. Hence, we can apply the same analysis as for v , and conclude the claim for all heights above h' . \square

A direct consequence is that there are $\mathcal{O}(1)$ fragile surrounded nodes on each height. With Lemma 14 we get the following theorem:

Theorem 19. The HSP tree $\text{HT}(Y)$ of a string Y of length n contains at most $\mathcal{O}(\lg^* n)$ fragile nodes on each height.

Having a bound on the number of fragile nodes, we start to study the algorithmic operations of an HSP tree. The first operation is how to actually build an HSP tree. For that, we have to think about its representation:

3.2 Tree Representation

Unlike Cormode and Muthukrishnan, who use hash tables to represent the dictionary \mathfrak{D} , we follow a deterministic approach. In our approach, we represent \mathfrak{D} by storing the HSP tree as a CFG. A name (i.e., a non-terminal of the CFG) is represented by a pointer to a data field (an allocated memory area), which is composed differently for leaves and internal nodes:

Leaves. A leaf stores a position i and a length $\ell \in \{2, 3\}$ such that $Y[i..i + \ell - 1]$ is the generated substring.

Internal nodes. An internal node stores the length of its generated substring, and the names of its children.

If it has only two children, we use a special, invalid name \perp for the non-existing third child such that all data fields are of the same length.

This information helps us to navigate from a node to its children or its generated substring in constant time, and to navigate top-down in the HSP tree by traversing the tree from the root in time linear in the height of the tree.

To accelerate substring comparisons, we want to give nodes with the same children (with respect to their order and names) the same name, such that the dictionary \mathfrak{D} is injective. To keep the dictionary injective, we do the following: Before creating a new name for the rule $b \rightarrow xyz$ (we set $z = \perp$ if the rule is $b \rightarrow xy$), we check whether there already exists a name for xyz . To perform this lookup efficiently, we need also the

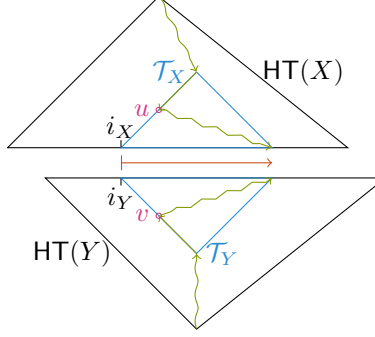


Figure 23: Conception of the proof of Lemma 21. To compute the longest common prefix of $X[i_X..]$ and $Y[i_Y..]$ (arrow in the center), we walk down the trees $\text{HT}(X)$ and $\text{HT}(Y)$ (depicted by the upper and the lower triangle, respectively) on the paths towards the leaves containing $X[i_X]$ and $Y[i_Y]$, respectively, by simultaneously visiting two nodes on the same height of both trees. The nodes u and v in the figure are on these paths. Suppose that they are on the same height and have the same surname. On visiting both nodes, we know that the longest common prefix is at least $\min(|\text{string}(u)|, |\text{string}(v)|)$ long. We update the destination of our traversal accordingly, such that we follow the paths from u and v to the leaves covering the not-yet checked parts of the longest common prefix that we want to compute.

reverse dictionary of \mathfrak{D} , with the right hand side of the rules as search keys. We want the reverse dictionary to be of size $\mathcal{O}(|Y|)$, supporting lookup and insert in $\mathcal{O}(t_{\text{look}})$ (deterministic) time for a $t_{\text{look}} = t_{\text{look}}(n)$ depending on n . For instance, a balanced binary search tree has $t_{\text{look}} = \mathcal{O}(\lg n)$.

With this tree representation, we can build HSP trees within the following time and space bounds:

Lemma 20. The HSP tree $\text{HT}(Y)$ of a string Y of length n can be built in $\mathcal{O}(n(\lg^* n + t_{\text{look}}))$ time. It takes $\mathcal{O}(n)$ words of space.

Proof. A name is inserted or looked-up in t_{look} time. Due to the alphabet reduction technique (see Lemma 4), applying *esp* on a substring of length ℓ takes $\mathcal{O}(\ell \lg^* n)$ time, returning a sequence of blocks of length at most $\ell/2$. \square

3.3 LCE Queries in HSP Trees

Like the trees [1, 32] based on signature encoding, we show that HSP trees are good at answering LCE queries. The idea is to compare the names of two nodes to test whether the generated substrings of both nodes are the same. Remembering that two nodes with the same generated substring can have different names (cf. the end of Sect. 2.3), we want to have a rule at hand saying when two nodes with different names must have different generated substrings. It is easy to provide such a rule when the input string is square-free: In this case, all fragile nodes are non-surrounded according to Lemma 8, and thus we know that the surrounded nodes are stable. Since each height consists of exactly one **type 2** meta-block, the equality of two substrings X and Y can be checked by comparing the names of two surrounded nodes whose generated substrings are X and Y , respectively. For general strings, we need additional information about the generated substring of each repeating node. That is because the names of two repeating nodes at *the same height* already differ when the generated substring of one node is a proper prefix of the generated substring of the other node. Fortunately, this additional information is given by the surnames and surname-lengths (see Fact 2 in Sect. 3.1):

Having a common dictionary \mathfrak{D} for all HSP trees that stores the length of the string $\mathfrak{D}^{(h)}(Z)$ for each name $Z \in \Sigma_h$, we explain how HSP trees can answer LCE queries efficiently.

Lemma 21. Given $\text{HT}(X)$ and $\text{HT}(Y)$ built on two strings X and Y with $|X| \leq |Y| \leq n$ and two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\text{lcp}(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg n \lg^* n)$ time.

Proof. We use the following property: If two nodes have the same surname Z , then the generated substrings of both nodes are Z^i and Z^j , respectively, with the respective surname-lengths i and j , where $Z = \text{string}(Z)$. This means that the generated substring of one node is a prefix of the generated substring of the other. In the particular case $i = j$, both nodes share the same subtree and consequently have the same name according to Lemma 15. In summary, this property allows us to omit the comparison of the subtrees of two nodes with the same surname, and thus speeds up the LCE computation, which is done in the following way (cf. Fig. 23):

- (1) We start with traversing the two paths from the roots of $\text{HT}(X)$ and $\text{HT}(Y)$ to the leaves λ_X and λ_Y whose generated substrings contain $\langle X \rangle_0[i_X]$ and $\langle Y \rangle_0[i_Y]$, respectively:
- (2) We traverse the two paths leading to the leaves λ_X and λ_Y , respectively, in a simultaneous manner such that we always visit a pair (u, v) of nodes on the same height belonging to $\text{HT}(X)$ and $\text{HT}(Y)$, respectively.
- (3) Given that u and v share the same surname $Z \in \Sigma_h$, we know the lengths of their generated substrings ($|\mathfrak{D}^{(h)}(Z)|^{\ell_u}$ and $|\mathfrak{D}^{(h)}(Z)|^{\ell_v}$) by having their surname-lengths ℓ_u and ℓ_v at hand. As a consequence, we know that $X[i_X..]$ and $Y[i_Y..]$ have a common prefix of at least $\min(|\mathfrak{D}^{(h)}(Z)|^{\ell_u}, |\mathfrak{D}^{(h)}(Z)|^{\ell_v})$. We update the variables λ_X and λ_Y to be the leaves whose generated substrings contain $\langle X \rangle_0[i_X + |\mathfrak{D}^{(h)}(Z)|^{\ell_u}]$ and $\langle Y \rangle_0[i_Y + |\mathfrak{D}^{(h)}(Z)|^{\ell_v}]$, respectively. Subsequently, we continue our tree traversals from u and v to the updated destinations ℓ_X and ℓ_Y , respectively. Since λ_X and λ_Y are not in the respective subtrees of u and v , we climb up the tree to the lowest common ancestor of u (resp. v) and λ_X (resp. λ_Y), and recurse on (2).
- (4) If we end up at a pair of leaves (i.e., $u = \lambda_X$ and $v = \lambda_Y$), we compare their generated substrings naively. If we find a mismatching character in both generated substrings, we can determine the value of ℓ and terminate. We also terminate if there is no mismatch, but λ_X or λ_Y is the rightmost leaf of $\text{HT}(X)$ or $\text{HT}(Y)$, respectively. In all other cases, we set λ_X and λ_Y to their respectively succeeding leaves, climb up to the parents of u and v , and recurse on (2).

During the traversals of both trees, we spend constant time for each navigation operation, i.e., (a) selecting a child, and (b) climbing up to the parent of a node: On the one hand, we select a child of a node v in constant time by following the pointer of the name of v (defined in Sect. 3.2). On the other hand, we maintain, for each tree, a stack storing all ancestors of the currently visited node during the traversal of the respective tree: Each stack uses $\mathcal{O}(\lg n)$ words, and can return the parent of the currently visited node in constant time.

To upper bound the running time of the traversals, we examine the nodes visited during the traversals. Starting at both root nodes, we follow the path from the root of $\text{HT}(X)$ (resp. $\text{HT}(Y)$) down to the roots of the minimal subtree \mathcal{T}_X of $\text{HT}(X)$ (resp. \mathcal{T}_Y of $\text{HT}(Y)$) covering $X[i_X..i_X + \ell]$ (resp. $Y[i_Y..i_Y + \ell]$).⁵ After entering the subtrees \mathcal{T}_X and \mathcal{T}_Y , we will never visit nodes outside of \mathcal{T}_X and \mathcal{T}_Y . The question is how many nodes of \mathcal{T}_X and \mathcal{T}_Y differ. This can be answered by studying the tree $\text{HT}(Z)$ built with the same dictionary \mathfrak{D} , where $Z := X[i_X..i_X + \ell - 1] = Y[i_Y..i_Y + \ell - 1]$: On the one hand, $\text{HT}(Z)$ has $\mathcal{O}(\lg^* n)$ fragile nodes on each height according to Thm. 19. On the other hand, each (semi-)stable node in $\text{HT}(Z)$ is found in both \mathcal{T}_X and \mathcal{T}_Y with the same name and surname. This means that when traversing $\text{HT}(X)$ and $\text{HT}(Y)$ within their respective subtrees \mathcal{T}_X and \mathcal{T}_Y , we only visit $\mathcal{O}(\lg^* n)$ pairs of nodes per height (remember that we follow the two paths to the leaves λ_X and λ_Y , respectively, up to the point where the surnames of the visited pair of nodes match).

To sum up, we (a) compute paths from the roots to $\langle X \rangle_0[i_X]$ and $\langle Y \rangle_0[i_Y]$, respectively, in $\mathcal{O}(\lg |Y|)$ time, and (b) compare the children of at most $\mathcal{O}(\lg^* n)$ nodes per height. Since both trees have a height of $\mathcal{O}(\lg |Y|)$, we obtain our claimed running time. \square

⁵We assume that $i_X + \ell \leq |X|$ and $i_Y + \ell \leq |Y|$. Otherwise, let \mathcal{T}_X and \mathcal{T}_Y cover $X[i_X..i_X + \ell - 1]$ and $Y[i_Y..i_Y + \ell - 1]$, respectively.

The following corollary is a small refinement of Lemma 21 that already shows the result of Thm. 3 for $\tau = 1$:

Corollary 22. Given $\text{HT}(X)$ and $\text{HT}(Y)$ built on two strings X and Y with $|X| \leq |Y| \leq n$ and two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\ell := \text{lcp}(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg \ell \lg^* n)$ time.

Proof. Our idea is to enhance an HSP tree with a data structure such that climbing up from a child to its parent can be performed in constant time. This can be achieved when we represent the tree topology of an HSP tree with a pointer based tree, in which each node stores its name and the pointer to its parent. The leaves are stored sequentially in a list. A bit vector with the same length as the input string is used to mark the borders of the generated substrings of the leaves. Given a text position i , we can access the leaf whose generated substring contains i in constant time with a rank-support on the bit vector. The bit vector with rank-support takes $n + o(n)$ bits. The pointer based tree can be built with the HSP tree without an additional time overhead, and takes $\mathcal{O}(n)$ words of space. \square

In the next section, we describe a preliminary version of our sparse suffix sorting algorithm that does not exploit the text space yet.

4 Sparse Suffix Sorting

The sparse suffix sorting problem asks for the order of suffixes starting at certain positions in a text T . In our case, these positions need only be given online, i.e., sequentially and in an arbitrary order. We collect them conceptually in a dynamic set \mathcal{P} with $m := |\mathcal{P}|$. The online sparse suffix sorting problem is to keep the suffixes starting at the positions stored in the incrementally growing set \mathcal{P} in sorted order. Due to the online setting, we represent the order of $\text{Suf}(\mathcal{P})$ by a dynamic, self-balancing binary search tree (e.g., an AVL tree). Each node of the tree is associated with a distinct suffix in $\text{Suf}(\mathcal{P})$; the lexicographic order is used as the sorting criterion.

The technique of Irving and Love [21] augments an AVL tree on a set of strings \mathcal{S} with LCP values so that $\ell_Y := \max\{\text{lcp}(X, Y) \mid X \in \mathcal{S}\}$ can be computed in $\mathcal{O}(\ell_Y / \log_\sigma n + \lg |\mathcal{S}|)$ time for a string Y . Inserting a new string Y into the tree is supported in the same time complexity (ℓ_Y is defined as before). Irving and Love called this data structure the *suffix AVL tree* on \mathcal{S} ; we denote it by $\text{SAVL}(\mathcal{S})$.

Remembering Sect. 1.2, our goal is to build $\text{SAVL}(\text{Suf}(\mathcal{P}))$ efficiently. However, inserting m suffixes naively takes $\Omega(|\mathcal{C}|m / \log_\sigma n + m \lg m)$ time. How to speed up the comparisons by exploiting a data structure for LCE queries is the topic of this section.

4.1 Abstract Algorithm

Starting with an empty set of positions $\mathcal{P} = \emptyset$, our algorithm updates $\text{SAVL}(\text{Suf}(\mathcal{P}))$ on the input of every new text position, involving LCE computations between the new suffix and suffixes already stored in $\text{SAVL}(\text{Suf}(\mathcal{P}))$. A crucial part of the algorithm is performed by these LCE computations, for which an LCE data structure is advantageous to have. In particular, we are interested in a *mergeable* LCE data structure that is mergeable in such a way that the merged instance answers queries faster than performing a query on both former instances separately. We call this a *dynamic LCE data type (dynLCE)*; it supports the following operations:

- $\text{dynLCE}(\mathcal{I})$ constructs a dynLCE data structure M on the substring $T[\mathcal{I}]$. Let $M.\text{ival}$ denote the interval \mathcal{I} .
- $\text{LCE}(M_1, M_2, p_1, p_2)$ computes $\text{lce}(p_1, p_2)$, where $p_i \in M_i.\text{ival}$ for $i = 1, 2$.
- $\text{merge}(M_1, M_2)$ merges two dynLCEs M_1 and M_2 such that the output is a dynLCE built on the string concatenation of $T[M_1.\text{ival}]$ and $T[M_2.\text{ival}]$.

We use the expression $t_C(|\mathcal{I}|)$ to denote the construction time of such a data structure on the substring $T[\mathcal{I}]$. We assume that the construction of $\text{dynLCE}(\mathcal{I})$ takes at least as long as scanning all characters on Y , i.e.,

Property 1: $t_C(|\mathcal{I}|) = \Omega(|\mathcal{I}| / \log_\sigma n)$.

We use the expressions $t_L(|X| + |Y|)$ and $t_M(|X| + |Y|)$ to denote the time for querying and the time for merging two such data structures built on two given strings X and Y , respectively. Querying two dynLCE s for a length ℓ is faster than the word-packed character comparison iff $\ell = \Omega(t_L(\ell) \lg n / \lg \sigma)$. Hence, we obtain the following property:

Property 2: A dynLCE on a text smaller than $g := \Theta(t_L(g) \lg n / \lg \sigma)$ is always slower than the word-packed character comparison.

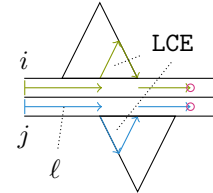
In the following, we build dynLCE s on substrings of the text. Each interval of the text that is covered by a dynLCE is called an **LCE interval**. The LCE intervals are maintained in a self-balancing binary search tree \mathcal{L} of size $\mathcal{O}(m)$. The tree \mathcal{L} stores the starting and the ending positions of each LCE interval, and uses the starting positions as keys to answer the queries

- whether a position is covered by a dynLCE , and
- where the next text position starts that is covered by a dynLCE ,

in $\mathcal{O}(\lg m)$ time. Additionally, each LCE interval is assigned to one dynLCE data structure (a dynLCE can be assigned to multiple LCE intervals) such that \mathcal{L} can not only retrieve the next position covered by a dynLCE , but actually return a dynLCE that covers that position. The dynLCE is retrieved by augmenting an LCE interval \mathcal{I} with a pointer to its dynLCE data structure M , and with an integer i such that $T[M.\text{ival} \cap [i..i + |\mathcal{I}| - 1]] = T[\mathcal{I}]$ (since M could be built on a text interval $M.\text{ival} \neq \mathcal{I}$ that contains an occurrence of $T[\mathcal{I}]$).

Given a new position $\hat{p} \notin \mathcal{P}$ with $1 \leq \hat{p} \leq |T|$, updating $\text{SAVL}(\text{Suf}(\mathcal{P}))$ to $\text{SAVL}(\text{Suf}(\mathcal{P} \cup \{\hat{p}\}))$ involves two parts: first *locating* the insertion node for \hat{p} in $\text{SAVL}(\text{Suf}(\mathcal{P}))$, and then *updating* the set of LCE intervals. **Locating.** The insertion operation performs an LCE computation for each node encountered in $\text{SAVL}(\text{Suf}(\mathcal{P}))$ while locating the insertion point of \hat{p} . Suppose that the task is to compare the suffixes $T[i..]$ and $T[j..]$ for two text positions i and j with $1 \leq i, j \leq |T|$. We perform the following steps to compute $\text{lce}(i, j)$:

- (1) Check whether the positions i and j are contained in an LCE interval, in $\mathcal{O}(\lg m)$ time with the search tree \mathcal{L} .
 - If both positions are covered by LCE intervals, then query the respective dynLCE s for the length ℓ of the LCE starting at i and j . Increment i and j by ℓ . Return the number of compared characters on finding a mismatch while computing the LCE.
 - Otherwise (if i or j are not contained in an LCE interval), find the smallest length ℓ such that $i + \ell$ and $j + \ell$ are covered by LCE intervals. Increment i and j by ℓ , and naively compare ℓ characters. Return the number of compared characters on a mismatch.



- (2) Return the total number of matched positions if a mismatch is found in (1). Otherwise, repeat the above check again (with the incremented values of i and j).

After locating the insertion point of \hat{p} in $\text{SAVL}(\text{Suf}(\mathcal{P}))$, we obtain $\bar{p} := \text{mlcparg}(\hat{p})$ and $\ell := \text{mlcp}(\hat{p})$ as a byproduct, where $\text{mlcparg}(p) := \arg\max_{p' \in \mathcal{P}, p \neq p'} \text{lcp}(T[p..], T[p'..])$ and $\text{mlcp}(p) := \text{lcp}(T[p..], T[\text{mlcparg}(p)..])$ for each text position p with $1 \leq p \leq |T|$. We insert \hat{p} into $\text{SAVL}(\text{Suf}(\mathcal{P}))$, and use the position \bar{p} and the length ℓ to update the LCE intervals.

Updating. The LCE intervals are updated dynamically, subject to the following properties (see Fig. 24):

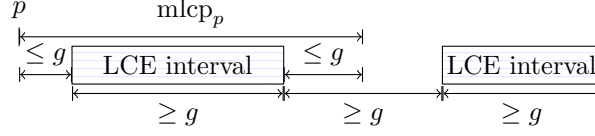


Figure 24: Sketch of two LCE intervals with Properties 3 to 5.

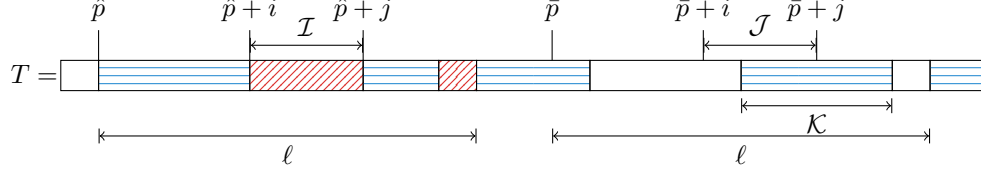


Figure 25: Application of Rules 1 to 4 for preserving the properties. The interval $\mathcal{I} := [\hat{p} + i.. \hat{p} + j]$ is not yet covered by an LCE interval, but is contained in $[\hat{p}.. \hat{p} + \ell - 1]$ — a conflict with Property 4. The conflict is resolved based on the LCE intervals covering the positions of $\mathcal{J} := [\bar{p} + i.. \bar{p} + j]$. The intervals with the horizontal lines are the LCE intervals, and the intervals with the diagonal lines are the intervals of $[\hat{p}.. \hat{p} + \ell - 1] \setminus U$. Here, \mathcal{J} intersects with an LCE interval \mathcal{K} . This case is treated in Rule 2.

Property 3: The length of each LCE interval is at least g (defined in Property 2).

Property 4: For every $p \in \mathcal{P}$, the interval $[p.. p + \text{mlcp}(p) - 1]$ is covered by an LCE interval, *except at most g positions at its left and right ends*.

Property 5: There is a gap of at least g positions between every pair of LCE intervals.

After adding \hat{p} to \mathcal{P} , we perform the following instructions to satisfy the properties. If $\ell \leq 2g$, we do nothing, because all properties are still valid (in particular, Property 4 still holds). Otherwise, we need to restore Property 4. There are at most two positions in \mathcal{P} that possibly invalidate Property 4 after adding \hat{p} , and these are \hat{p} and \bar{p} (otherwise, by transitivity, we would have created a longer LCE interval previously).

We introduce an algorithm that does not restore Property 4 directly, but first ensures that

Property 4': the intervals $[\hat{p}.. \hat{p} + \ell - 1]$ and $[\bar{p}.. \bar{p} + \ell - 1]$ are covered by one or multiple LCE intervals.

In a later step, we restore Property 4 by merging LCE intervals that are in conflict with Property 5, and thus restore all properties: Let $U \subset [1..n]$ be the set of all positions that belong to an LCE interval. The set $[\hat{p}.. \hat{p} + \ell - 1] \setminus U$ can be represented as a set of disjoint intervals of maximal length. For each interval $\mathcal{I} := [\hat{p} + i.. \hat{p} + j] \subset [\hat{p}.. \hat{p} + \ell - 1]$ of that set, apply the following rules with $\mathcal{J} := [\bar{p} + i.. \bar{p} + j]$ (for integers i, j with $0 \leq i \leq j \leq \ell - 1$, see Fig. 25) sequentially:

Rule 1: If \mathcal{J} is a sub-interval of an LCE interval \mathcal{K} , then declare \mathcal{I} as an LCE interval and let it refer to the dynLCE of \mathcal{K} .

Rule 2: If \mathcal{J} intersects with an LCE interval \mathcal{K} , enlarge the dynLCE on $T[\mathcal{K}]$ to cover $T[\mathcal{K} \cup \mathcal{J}]$ (create a dynLCE on $T[\mathcal{J} \setminus \mathcal{K}]$ and merge it with the dynLCE on $T[\mathcal{K}]$). Apply Rule 1.

Rule 3: Otherwise (there is no LCE interval \mathcal{K} with $\mathcal{J} \cap \mathcal{K} \neq \emptyset$), create $\text{dynLCE}(\mathcal{J})$, and make \mathcal{I} and \mathcal{J} to LCE intervals referring to $\text{dynLCE}(\mathcal{J})$.

We satisfy Property 4' on $[\bar{p}.. \bar{p} + \ell - 1]$ by updating U , computing the set of disjoint intervals $[\bar{p}.. \bar{p} + \ell - 1] \setminus U$, and applying the same rules on it. However, Rule 1 or Rule 3 can create LCE intervals shorter than g , violating Property 3. By construction, such a short LCE interval is adjacent to another LCE interval (the rules compute a cover of $[\hat{p}.. \hat{p} + \ell - 1]$ and $[\bar{p}.. \bar{p} + \ell - 1]$ with LCE intervals). This means that we can restore Property 3 by restoring Property 5. We do that by applying the following rule subsequently to Rule 3:

Rule 4: Merge a newly created or extended LCE interval violating Property 5 with its nearest LCE interval (ties can be broken arbitrarily). Merge those LCE intervals and their dynLCEs.

Rule 4 also restores Property 4 (since Property 4' and Property 5 hold). After applying all rules, we have introduced at most two⁶ new LCE intervals that cover the intervals $[\hat{p} + g.. \hat{p} + \ell - 1 - g]$ and $[\bar{p} + g.. \bar{p} + \ell - 1 - g]$, respectively, to satisfy Properties 3 to 5. The running time of this algorithm is analyzed in the following lemma:

Lemma 23. Given a text T of length n and a set of m arbitrary positions \mathcal{P} in T , the suffix AVL tree $\text{SAVL}(\text{Suf}(\mathcal{P}))$ with the suffixes of T starting at the positions \mathcal{P} can be computed deterministically in $\mathcal{O}(t_C(|\mathcal{C}|) + t_L(|\mathcal{C}|)m \lg m + t_M(|\mathcal{C}|)m)$ time.

Proof. The analysis is split into managing the dynLCEs, and the LCE queries:

- We build dynLCEs on substrings covering at most $|\mathcal{C}|$ characters of the text, taking at most $t_C(|\mathcal{C}|)$ time for constructing all dynLCEs. During the construction of the dynLCEs we spend $\mathcal{O}(|\mathcal{C}| / \log_\sigma n) = \mathcal{O}(t_C(|\mathcal{C}|))$ time on naive searches due to Property 1.
- The number of merge operations on the LCE intervals is upper bounded by $2m$ in total, since we create at most two new LCE intervals for every position in \mathcal{P} . In total, we spend at most $2 t_M(|\mathcal{C}|)m$ time for the merging in total.
- The algorithm performs $\mathcal{O}(m \lg m)$ LCE queries. LCE queries involve either (a) naive character comparisons or (b) querying a dynLCE. Given that we have $\delta < 2m$ LCE intervals, we switch between both techniques at most $4\delta + 1$ times for an LCE query.
 - (a) On the one hand, the overall time for the naive character comparisons is bounded by $\mathcal{O}(t_C(|\mathcal{C}|) + t_L(|\mathcal{C}|)m \lg m)$:
 - By Property 3, all substrings $T[p..p + \text{mlcp}(p) - 1]$ are covered by an LCE interval, except at most at $2g$ positions. This means that all substrings that are not covered by an LCE interval, but have been subject to a naive character comparison, are shorter than $2g$. For a naive character comparison with one of those substrings, we spend at most $\mathcal{O}(gm \lg m / \log_\sigma n) = \mathcal{O}(t_L(g)m \lg m) = \mathcal{O}(t_L(|\mathcal{C}|)m \lg m)$ time. In the case that $g > |\mathcal{C}|$, we do not create any LCE interval, and spend $\mathcal{O}(|\mathcal{C}| / \log_\sigma n + m \lg m) = \mathcal{O}(t_C(|\mathcal{C}|) + m \lg m)$ overall time due to Property 1.
 - If we compare more than g characters for an LCE query, we create at most two LCE intervals, possibly involving the construction of dynLCEs on the compared substrings. The construction of a dynLCE on an interval \mathcal{I} takes $t_C(|\mathcal{I}|) = \Omega(|\mathcal{I}| / \log_\sigma n)$ time due to Property 1.
 - (b) On the other hand, querying the dynLCEs take at most $\mathcal{O}(t_L(|\mathcal{C}|)m \lg m)$ overall time. Suppose that we look up $d < \delta$ LCE intervals for an LCE query. Since we look up an LCE interval in $\mathcal{O}(\lg m)$ time with \mathcal{L} , we spend $\mathcal{O}(d \lg m)$ time on the lookups during this LCE query. However, we subsequently merge all d looked-up LCE intervals, reducing the number of LCE intervals δ by $d - 1$. Consequently, we perform a look-up of an LCE interval at most $2m$ times in total. \square

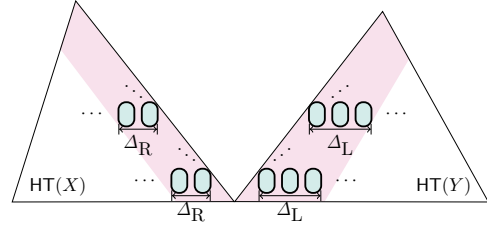
⁶The number of new LCE intervals could be indeed two: Although $\bar{p} \in \mathcal{P}$, we would not have created an LCE interval covering $[\bar{p} + g.. \bar{p} + \ell - 1 - g]$ if $\text{mlcp}(\bar{p})$ was smaller than g at the time when we inserted \bar{p} in \mathcal{P} with $\ell := \text{mlcp}(\bar{p})$.

The last step is to compute $\text{SSA} := \text{SSA}(T, \mathcal{P})$ and $\text{SLCP} := \text{SLCP}(T, \mathcal{P})$ from $\text{SAVL}(\text{Suf}(\mathcal{P}))$ by traversing $\text{SAVL}(\text{Suf}(\mathcal{P}))$ and performing LCE queries on the already computed dynLCEs: The $\text{SAVL}(\text{Suf}(\mathcal{P}))$ is a binary search tree storing all elements of $\text{Suf}(\mathcal{P})$ in lexicographically sorted order. This means that we can compute SSA with an in-order traversal of $\text{SAVL}(\text{Suf}(\mathcal{P}))$. Afterwards, we compute $\text{SLCP}[i] = \text{lce}(\text{SSA}[i], \text{SSA}[i-1])$. Given that the text positions $[\text{SSA}[i].. \text{SSA}[i] + \text{SLCP}[i] - 1]$ and $[\text{SSA}[i-1].. \text{SSA}[i-1] + \text{SLCP}[i-1] - 1]$ are not covered by an LCE interval, $\text{SLCP}[i] = \mathcal{O}(g)$ due to Property 3, and we spend at most $\mathcal{O}(g/\log_\sigma n)$ time on computing $\text{SLCP}[i]$ by naive character comparisons. Otherwise, we spend $\mathcal{O}(g/\log_\sigma n + t_L(\text{SLCP}[i])) = \mathcal{O}(t_L(\text{SLCP}[i]))$ time by querying a *single* dynLCE due to Property 4. Querying whether both text intervals are covered by an dynLCE costs $\mathcal{O}(\lg m)$ time with \mathcal{L} . In total, we can compute $\text{SLCP}[i]$ for each integer i with $2 \leq i \leq m$ in $\mathcal{O}(t_L(|\mathcal{C}|) + m \lg m)$ time, since $\mathcal{O}(g/\log_\sigma n) = \mathcal{O}(t_L(g))$ due to Property 2. The following corollary of Lemma 23 summarizes the achievements of this section:

Corollary 24. Given a text T of length n that is loaded into RAM, the SSA and SLCP of T for a set of m arbitrary positions can be computed deterministically in $\mathcal{O}(t_C(|\mathcal{C}|) + t_L(|\mathcal{C}|)m \lg m + t_M(|\mathcal{C}|)m)$ time. We need $\mathcal{O}(m)$ words of space, and space to store *dynLCE* on $|\mathcal{C}|$ positions.

4.2 Sparse Suffix Sorting with HSP Trees

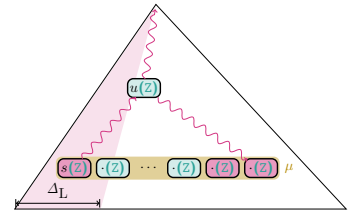
We show that the HSP tree is a dynLCE data structure. Remembering that the algorithm from Sect. 4.1 depends on the merging operation of dynLCE, we now introduce the merging of HSP trees. A naive way to merge two HSP trees $\text{HT}(X)$ and $\text{HT}(Y)$ is to build $\text{HT}(XY)$ completely from scratch. Since only the fragile nodes of $\text{HT}(X)$ and $\text{HT}(Y)$ can change when merging both trees, a more sophisticated approach would reparse only the fragile nodes of both trees. Remembering the properties studied in Sect. 2.4, we show such an approach in the following lemma:



Lemma 25. Merging $\text{HT}(X)$ and $\text{HT}(Y)$ of two strings $X, Y \in \Sigma^*$ into $\text{HT}(XY)$ takes $\mathcal{O}(t_{\text{look}}(\Delta_R \lg |X| + \Delta_L \lg |Y|))$ time.

Proof. First assume that $\text{HT}(X)$ and $\text{HT}(Y)$ only contain **type 2** nodes. In this case, we examine the rightmost nodes of $\text{HT}(X)$ and the leftmost nodes of $\text{HT}(Y)$ from the bottom up to the root: At each height h , we merge the nodes $\langle X \rangle_h$ and $\langle Y \rangle_h$ to $\langle XY \rangle_h$ by reparsing the Δ_R rightmost nodes of $\langle X \rangle_h$, and the Δ_L leftmost nodes of $\langle Y \rangle_h$. By doing so, we reparse all nodes of $\text{HT}(X)$ (resp. $\text{HT}(Y)$) whose local surrounding on the right (resp. left) side does not exist. Nodes of $\text{HT}(X)$ (resp. $\text{HT}(Y)$) that have a local surrounding on the right (resp. left) side are not changed by the parsing. In total, we spend $\mathcal{O}(t_{\text{look}}(\Delta_R \lg |X| + \Delta_L \lg |Y|))$ time on merging two trees consisting of **type 2** nodes.

Next, we allow repeating nodes. Lemma 17 shows that there are no fragile surrounded nodes in $\text{HT}(X)$ that need to be fixed. The remaining problem is to find and recompute the surrounded nodes in $\text{HT}(Y)$ whose names change on merging both trees. The lowest of these nodes belong to a repeating meta-block due to Lemma 7 and Cor. 12. To find this meta-block, we adapt the strategy of the first paragraph considering only **type 2** meta-blocks. On each height h , we reparse the Δ_L leftmost nodes of $\langle Y \rangle_h$. If the rightmost of these Δ_L nodes are contained in a repeating meta-block μ that does not end within those Δ_L leftmost nodes, chances are that the names of some nodes in μ change. Due to Cor. 12, it is sufficient to reparse the two rightmost nodes of μ . This is done as follows:



1. Take the leftmost repetitive node s of μ (which exists due to Cor. 16, and is one of the $\Delta_L + 1$ leftmost nodes on height h).

2. Given that s has the surname Z , climb up the tree to find the highest ancestor u with surname Z . The ancestor u is the lowest common ancestor of s and the rightmost repetitive node of μ .
3. Walk down from u to the rightmost nodes of μ .
4. Reparse μ 's two rightmost nodes.
5. Reparse all ancestors of these two nodes that are surrounded.
6. Check whether the reparsed ancestors invalidate the parsing of their meta-blocks; fix the parsing for those meta-blocks recursively.

Climbing up to find u and walking down to the rightmost nodes of μ takes $\mathcal{O}(t_{\text{look}} \lg |\mu|) = \mathcal{O}(t_{\text{look}} \lg(n/2^h))$ time, reparsing the surrounded ancestor nodes of the two rightmost nodes of μ takes $\mathcal{O}(t_{\text{look}} \lg(n/2^h))$ time. Given that the highest nodes of this reparsing are on a height $h' > h$, Lemma 18 states that up to the height $h' + 1$, there is no need to reparse a fragile surrounded node (we follow the paths of fragile nodes as depicted in Fig. 22). Given that there are μ_1, \dots, μ_k such meta-blocks (for which we apply Steps 1 to 6), we have $\mathcal{O}(t_{\text{look}} \sum_{i=1}^k \lg |\mu_i|) = \mathcal{O}(t_{\text{look}} \lg n)$ due to $\sum_{i=1}^k \lg \mu_i \leq \lg n$. Hence, we spend $\mathcal{O}((\Delta_L + \Delta_R)t_{\text{look}} \lg |Y|)$ time overall. \square

The following theorem combines the results of Cor. 24 and Lemma 25.

Theorem 26. Given a text T of length n and a set of m text positions \mathcal{P} , $\text{SSA}(T, \mathcal{P})$ and $\text{SLCP}(T, \mathcal{P})$ can be computed in $\mathcal{O}(|\mathcal{C}| (\lg^* n + t_{\text{look}}) + m \lg m \lg n \lg^* n)$ time. We need $\mathcal{O}(n + m)$ words of space.

Proof. We have

- $t_C(|\mathcal{C}|) = \mathcal{O}(|\mathcal{C}| (\lg^* n + t_{\text{look}}))$ due to Lemma 20,
- $t_L(|\mathcal{C}|) = \mathcal{O}(\lg^* n \lg n)$ due to Lemma 21, and
- $t_M(|\mathcal{C}|) = \mathcal{O}(t_{\text{look}} \lg n \lg^* n)$ due to Lemma 25.

Actually, the time cost for merging is already upper bounded by the cost for the tree creation. To see this, let $\delta \leq m$ be the number of LCE intervals. Since each HSP tree covers at least g characters, δg is at most $|\mathcal{C}|$, and we obtain $\delta t_M(|\mathcal{C}|) = \mathcal{O}(|\mathcal{C}| t_M(|\mathcal{C}|)/g) = \mathcal{O}(|\mathcal{C}| t_{\text{look}})$ overall time for merging, where $g = \Theta(t_L(|\mathcal{C}|) \lg n / \lg \sigma) = \Theta(\lg^* n \lg^2 n / \lg \sigma)$. Plugging the times $t_C(|\mathcal{C}|)$, $t_L(|\mathcal{C}|)$, and the refined analysis of the merging time cost in Cor. 24 yields the claimed time bounds. \square

5 Sparse Suffix Sorting in Text Space

Remembering the outline in the introduction, the key idea to solve the limited space problem is storing dynLCEs in text space. Taking two LCE intervals of the text containing the same substring, we free up the space of *one* part while marking the *other* part as a reference. The freed space could be used to store an HSP tree whose leaves refer to substrings of the other LCE interval. By doing so, we would use the text space for storing the HSP trees, while using only $\mathcal{O}(m)$ additional words for storing $\text{SAVL}(\text{Suf}(\mathcal{P}))$ and the search tree \mathcal{L} of the LCE intervals. However, an HSP tree built on a string of length n takes $\mathcal{O}(n \lg n)$ bits, while the string itself provides only $n \lg \sigma$ bits. Our solution is to truncate the HSP tree at a fixed height η , discarding the nodes in the lower part. The truncated version $\text{tHT}_\eta(Y)$ stores just the upper part, while its new leaves refer to (possibly long) substrings of Y . The resulting tree is called the η -*truncated HSP tree* (tHT_η), whose definition follows:

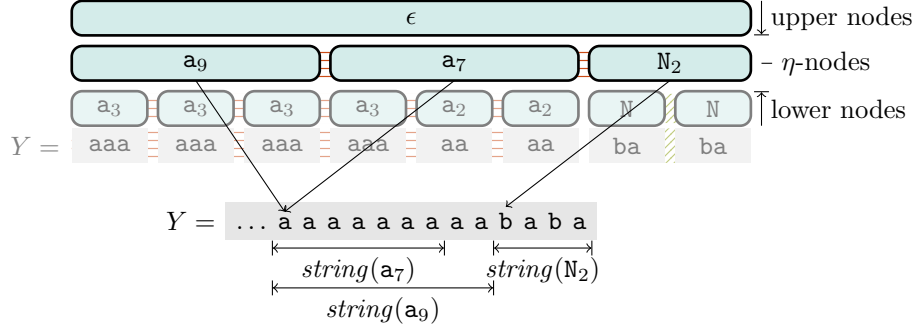


Figure 26: The η -truncated HSP tree $\text{tHT}_\eta(Y)$ of the substring Y defined in Fig. 6 with $\eta = 2$. Like in Fig. 21, the lower nodes are grayed out. An η -node is a leaf in $\text{tHT}_\eta(Y)$, and has a generated substring with a length between four and nine.

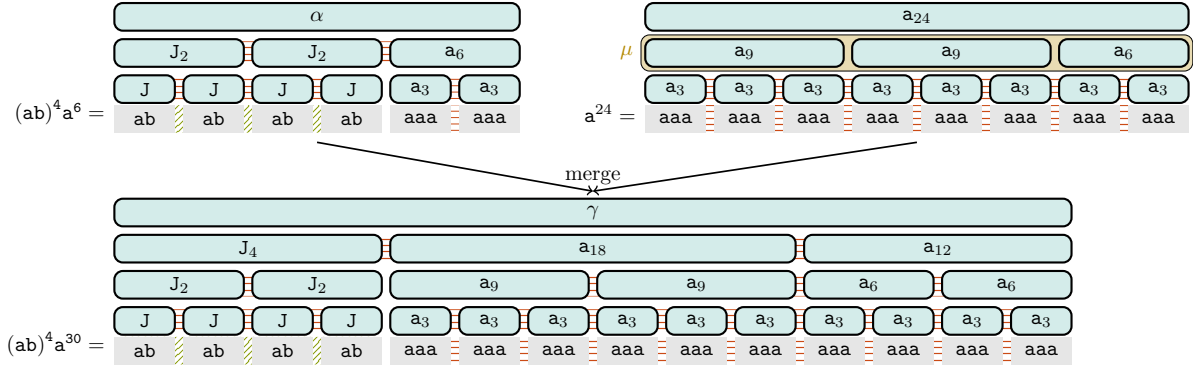


Figure 27: Merging $\text{HT}((ab)^4 a^6)$ with $\text{HT}(a^{24})$ (both at the top) to $\text{HT}((ab)^4 a^{30})$ (bottom tree). Reparsing the repeating meta-block μ on height one of the right tree is done by rearranging μ 's two rightmost nodes.

5.1 Truncated HSP Trees

We define a height η and delete all nodes at height less than η , which we call **lower nodes**. A node higher than η is called an **upper node**. The nodes at height η form the new leaves and are called **η -nodes**. Similar to the former leaves, their names are pointers to their generated substrings appearing in Y . Remembering that each internal node has two or three children, an η -node generates a string of length at least 2^η and at most 3^η . The maximum number of nodes in an η -truncated HSP tree of a string of length n is $n/2^\eta$. Figure 26 shows an example with $\eta = 2$.

Similar to leaves in untruncated HSP trees, we use the generated substring X of an η -node v for storing and looking up v : While the leaves of the HSP tree have a generated substring of constant size (two or three characters), the generated substring of an η -node can be as long as 3^η . Storing such long strings in a binary search tree representing the reverse dictionary of \mathfrak{D} is inefficient; it would need $\mathcal{O}(\ell \lg \sigma)$ time for a lookup or insertion of a key of length ℓ . Instead, we want a dictionary data structure storing $\mathcal{O}(|Y|)$ elements in $\mathcal{O}(|Y|)$ words of space⁷, supporting lookup and insert in $\mathcal{O}(t_{\text{look}} + \ell / \log_\sigma n)$ time for a key of length ℓ . For instance, Franceschini and Grossi's data structure [13] with word-packing supports the desired time and space bounds with $t_{\text{look}} = \mathcal{O}(\lg n)$.

Lemma 27. We can build an η -truncated HSP tree $\text{tHT}_\eta(Y)$ of a string Y of length n in $\mathcal{O}(n(\lg^* n + \eta / \log_\sigma n + t_{\text{look}}/2^\eta))$ time, using $\mathcal{O}(3^\eta \lg^* n)$ words of working space. The tree takes $\mathcal{O}(n/2^\eta)$ words of space.

⁷The data structure is not necessarily stored in consecutive space like an array.

Proof. Instead of building the HSP tree level by level, we compute the η -nodes one after another, from left to right. We can split the parsing of the whole string into several parts. Each part computes one η -node.

First assume that $\text{tHT}_\eta(Y)$ only contains **type 2** nodes. Then the name of an η -node v is determined by v 's local surrounding (as far as it exists) due to Lemma 7. This means that it is sufficient to keep v 's local surrounding at height $\eta - 1$, which we denote by X_v , in memory. X_v is a string of lower nodes. To parse a string of lower nodes by HSP, we have to give each lower node a name. Unfortunately, storing the names of all lower nodes in a dictionary would take too much space. Instead, we create the name of a lower node temporarily by setting the name of a lower node to its generated substring. This means that we cannot retrieve their names later. Luckily, we only need the names of the lower nodes for constructing X_v . We construct X_v as follows: Given that we parsed the local surrounding of v at height h ($0 \leq h \leq \eta - 3$) with HSP, we store the borders of the blocks on height $h + 1$ in an integer array such that we can access the name (i.e., the generated substring) of the i -th block on height $h + 1$. With this integer array, we can parse the blocks on height $h + 1$ to obtain the blocks on height $h + 2$, whose borders are again stored in an integer array. Having the borders of the blocks on height $h + 2$, we can remove the integer array on height $h + 1$. The blocks on height $\eta - 1$ are the nodes of X_v .

In the general case (when $\text{tHT}_\eta(Y)$ contains repeating nodes), it can happen that the name of a greedily parsed node (i.e., a repeating node or one of the Δ_L leftmost nodes of a **type 2** meta-block) depends not necessarily on its local surrounding, but on the length of its repeating meta-block, its surname and its children (in case of a **type M** node). This means that when computing X_v of an η -node v , we additionally have to consider the case when nodes in the local surrounding of v are contained in a meta-block μ on height $h < \eta$ that extends over the nodes in v 's surrounding at height h . It is sufficient to use a counting variable that tracks the position of the last block of μ belonging to the subtree of the preceding η -node of v (remember that the greedy parsing determines the blocks by an arithmetic progression). Another necessity is to maintain the surnames of the lower nodes. In our approach, each array storing the borders of the blocks on the heights below η is accompanied with two arrays. The first array stores the length of the prefix of the generated substring of each block β that is equal to β 's surname; the second array stores the surname-length of each block.

Working Space. We compute v after computing X_v . To compute X_v , we apply the HSP technique $(\eta - 1)$ -times on the generated substring of the nodes in X_v . Since the nodes of X_v cover at most $3^\eta(\Delta_L + \Delta_R)$ characters, we need $\mathcal{O}(3^\eta(\Delta_L + \Delta_R))$ words of working space to maintain the integer arrays storing the borders of the blocks at two consecutive heights. To cope with the meta-blocks extending over the border of the subtrees of two η -nodes, we store the last position of each such meta-block belonging to the local surrounding of the previous η -node. These positions take $\mathcal{O}(\eta)$ words, since such a meta-block can exist on every height below η .

Time. The time bound $\mathcal{O}(n \lg^* n)$ for the repeated application of the alphabet reduction is the same as in Lemma 20. The new part is the construction of an η -node by constructing X_v : To construct the lower nodes X_v , we apply the HSP technique $(\eta - 1)$ -times on $\text{string}(v)$. The HSP technique compares lower nodes by their generated substrings (instead of comparing by a name stored in \mathfrak{D}). It always compares two adjacent lower nodes during the construction of X_v . To bound the number of comparisons of the lower nodes, we focus on all lower nodes on a fixed height h with $1 \leq h \leq \eta - 1$: Since the sum of the lengths of the generated substrings of the lower nodes on height h is always n , the comparisons of the lower nodes on height h take $\mathcal{O}(n / \log_\sigma n)$ time, independent of the number of nodes on height h . Summing over all heights, these comparisons take $\mathcal{O}(n\eta / \log_\sigma n)$ time in total. By the same argument, maintaining the names of all η -nodes takes $\mathcal{O}(n / \log_\sigma n + t_{\text{look}} n / 2^\eta)$ time.

A name is looked-up in $\mathcal{O}(t_{\text{look}})$ time for an upper node. Since the number of upper nodes is at most $n / 2^\eta$, maintaining the names of the upper nodes takes $\mathcal{O}(t_{\text{look}} n / 2^\eta)$ time. This time is subsumed by the lookup time for the η -nodes.

Surnames. Augmenting the (remaining) nodes of the η -truncated HSP tree with surnames cannot be done as trivially as in the standard HSP tree construction, since a repetitive node can have a surname equal to the name of a lower node (remember that lower nodes are generated only temporarily, and hence are not maintained in the reverse dictionary). To maintain the surnames pointing to lower nodes, we need to save

the names of certain lower nodes in a supplementary reverse dictionary \mathfrak{D}' of \mathfrak{D} . This is only necessary when one of the remaining nodes (i.e., the upper nodes and the η -nodes) in the η -truncated HSP tree has a surname that is the name of a lower node. If such a remaining node v is an upper node having a surname equal to the name of a lower node, the η -nodes in the subtree rooted at v have also the same surname. Hence, the number of entries in \mathfrak{D}' is upper bounded by the number of η -nodes. The dictionary \mathfrak{D}' is filled with the surnames of the children of all η -nodes, whose number is at most $3n/2^\eta$. Filling or querying \mathfrak{D}' takes the same time as maintaining the η -nodes. \square

Similar to the standard HSP trees, we can conduct LCE queries on two η -truncated HSP trees in the following way:

Lemma 28. Let X and Y be two strings with $|X|, |Y| \leq n$. Given that $\text{tHT}_\eta(X)$ and $\text{tHT}_\eta(Y)$ are built with the *same* dictionary, and given two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\text{lcp}(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg^* n (\lg(n/2^\eta) + 3^\eta / \log_\sigma n))$ time using $\mathcal{O}(\lg(n/2^\eta))$ words of working space.

Proof. Lemma 21 gives the time bounds for computing the longest common prefix with two HSP trees. The lemma describes an LCE algorithm that uses the surnames to compare the generated substring of two nodes. By doing so, it accelerates the search for the first pair of mismatching characters in $X[i_X..]$ and $Y[i_Y..]$. To find this mismatching pair, it examines the subtrees of the two nodes if both nodes mismatch. Since we cannot access a child of an η -node in our η -truncated HSP trees without rebuilding its subtree (as we do not store the lower nodes in \mathfrak{D}), we treat the η -nodes as the leaves of the tree. This means that we compare two η -nodes (given their surnames are different) with a naive comparison of their generated substrings in $\mathcal{O}(3^\eta / \log_\sigma n)$ time, remembering that the length of the generated substring of an η -node is at most 3^η . For the upper nodes, the algorithm works identically to the original version such that it takes $\mathcal{O}(\lg^* n (\lg(\ell/2^\eta)))$ time for traversing those. \square

Applying the idea of Cor. 22 to Lemma 28 gives the following corollary:

Corollary 29. Let X and Y be two strings with $|X|, |Y| \leq n$. Given that $\text{tHT}_\eta(X)$ and $\text{tHT}_\eta(Y)$ are built with the *same* dictionary, we can augment both trees with a data structures such that given two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\ell := \text{lcp}(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg^* n (\lg(\ell/2^\eta) + 3^\eta / \log_\sigma n))$ time using $\mathcal{O}(\lg(n/2^\eta))$ words of working space. The additional data structures can be constructed in $\mathcal{O}(n)$ time with $\mathcal{O}(n/\lg n)$ words of space. Their space bounds are within the space bounds of the HSP trees.

Proof. To support accessing the parent of a node in constant time, we construct a pointer based tree structure of the truncated tree during its construction. Since $\text{tHT}_\eta(Y)$ contains at most $n/2^\eta$ nodes, the pointer based tree structure takes $\mathcal{O}(n/2^\eta)$ words.

Given that $\eta \leq \lg \lg n$, we augment the tree structure with a bit vector to jump from a text position to an η -node like in Cor. 22: We create a bit vector of length n marking the borders of the generated substrings of the η -nodes such that a rank-support data structure on this bit vector allows us to jump from a position $Y[i]$ to the η -node $\langle Y \rangle_\eta[j]$ with $1 + \sum_{k=1}^{j-1} \text{string}(\langle Y \rangle_\eta[k]) \leq i \leq \sum_{k=1}^j \text{string}(\langle Y \rangle_\eta[k])$ in constant time. The bit vector with rank-support takes $\mathcal{O}(n/\lg n)$ words, which is too much to obtain the space bounds of $\mathcal{O}(n/2^\eta)$ words when $\eta = \Omega(\lg \lg n)$.

Instead, we compute a sorted list of pairs if $\eta \geq \lg_3(\lg^2 n)$. During the construction of a truncated tree, we collect pairs of constructed η -nodes and their starting positions in a list. This list is automatically sorted by the starting positions as we construct the tree from left to right. The list takes $\mathcal{O}(n/2^\eta)$ words, and we can find the η -node whose generated substring covers a given position in $\mathcal{O}(\lg(n/2^\eta)) = \mathcal{O}(\lg n)$ time by binary searching the starting positions. This time is bounded by the time $\mathcal{O}(\lg^* n 3^\eta / \log_\sigma n)$ for scanning the generated substrings of all η -nodes during an LCE query, which is $\mathcal{O}(\lg^* n \lg n \lg \sigma)$ time when $\eta \geq \lg_3(\lg^2 n)$.

It is left to consider the case that $\lg \lg n < \eta < \lg_3 \lg^2 n$. Let k be the number of η -nodes such that $n/3^\eta \leq k \leq n/2^\eta$. We build the above bit vector in the representation of Pagh [34]. In this representation, the rank-support answers rank queries in constant time. The bit vector together with its rank-support takes $\mathcal{O}(k \lg(n/k) + k^2/n + k(\lg \lg k)^2 / \lg k) = \mathcal{O}(k\eta)$ bits (which are $\mathcal{O}(n/2^\eta)$ words) when $k = n/\lg^c n$ for a

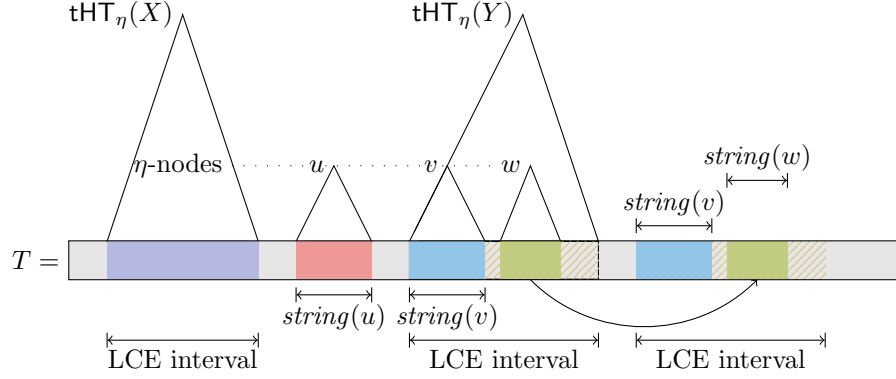


Figure 28: Problem with generated substrings when merging $\text{tHT}_\eta(X)$ and $\text{tHT}_\eta(Y)$. Assume that we want to merge $\text{tHT}_\eta(X)$ and $\text{tHT}_\eta(Y)$, and thus compute the bridging η -nodes (like u) between both trees. On the one hand, the generated substrings of the non-surrounded η -nodes (like v) and of the bridging nodes are marked protected, because we cannot find a surrogate substring in general. Although there is a second occurrence of $\text{string}(v)$ to the right, $\text{string}(v)$ can be extended or shortened when prepending characters (e.g., suppose that $\text{string}(v) = \mathbf{a}^k$, and that there is an \mathbf{a} to the left of the left occurrence of $\text{string}(v)$, but not to the left of the right occurrence). On the other hand, the space of the recyclable interval can be used for storing the η -truncated HSP trees, because here we find suitable surrogate substrings for the generated substrings of the η -nodes (like for w).

constant $c > 0$ [37, Theorem 4(b)]. The constant c exists, because $n/\lg^2 n < n/3^\eta \leq k \leq n/2^\eta < n/\lg n$. However, the construction needs $\mathcal{O}(n/\lg n)$ words of space. \square

With $\tau := 2^\eta$ we obtain the claim of Thm. 3.

Remark 30. In the following, we stick to the result obtained in Lemma 28 instead of Cor. 29. Although Lemma 28 has a slower running time for longest common prefixes that are short, the additional rank-support data structures of Cor. 29 makes it difficult to achieve our aimed running time for merging two trees (and therefore would restrain us from achieving our final goal stated in Thm. 1). To merge two trees, where each tree is augmented with the bit vector and its rank-support data structure, the task would be to build a rank-support data structure on the concatenation of the bit vectors (preferably in logarithmic time). Unfortunately, we are not aware of a rank-support data structure that is efficiently mergeable (a naive way would be to build the rank-support data structure of the large bit vector from scratch in linear time).

5.2 Sparse Suffix Sorting with tHTs

To use the η -truncated HSP trees as dynLCEs stored *in text space*, we have to think about how to merge them. Like with HSP trees, merging two η -truncated HSP trees involves a reparsing of the nodes at the facing borders (cf. Fig. 27). However, the reparsing of the η -nodes on that borders is especially problematic, as can be seen in Fig. 28: Suppose that we rename an η -node v from \mathbf{N}_2 to \mathbf{N}_3 with $|\text{string}(\mathbf{N}_2)| < |\text{string}(\mathbf{N}_3)|$. If the name \mathbf{N}_3 is not yet maintained in the dictionary, we have to create \mathbf{N}_3 , i.e., a pointer to a substring X of the text with $X = \text{string}(\mathbf{N}_3)$. The critical part is to find X in the not-yet overwritten parts of the text: Although we can create a suitably long string containing X by concatenating the generated substrings of v 's preceding and succeeding siblings, these η -nodes may point to text intervals that are not consecutive. Since the name of an η -node is the representation of a *single* substring, we would have to search X in the *entire* remaining text. In the case that v is surrounded, Lemma 18 shows that X is a prefix of the generated substring of a sibling η -node (unlike in Fig. 21, where the generated substring of the ESP node with name $\mathbf{0}$ cannot be easily determined). With this insight, we finally show an approach that proves Thm. 1. For that,

it remains to implement Rule 3 and Rule 4 from Sect. 4.1 in the context that we maintain η -truncated HSP trees *in text space*: We explain

Goal 1: how the parameter η has to be chosen such that $\text{tHT}_\eta(Y)$ fits into $|Y| \lg \sigma$ bits (needed for Rule 3), and

Goal 2: how to merge two η -truncated HSP trees without the need for extra working space (needed for Rule 4).

Our first goal is to store $\text{tHT}_\eta(T[\mathcal{I}])$ in a text interval \mathcal{I} . Since $\text{tHT}_\eta(T[\mathcal{I}])$ can contain nodes with $|\mathcal{I}|/2^\eta$ distinct names, it requires $\mathcal{O}(|\mathcal{I}|/2^\eta)$ words, i.e., $\mathcal{O}(|\mathcal{I}| \lg n / 2^\eta)$ bits of space that might not fit in the $|\mathcal{I}| \lg \sigma$ bits of $T[\mathcal{I}]$. Declaring a constant α (independent of n and σ , but dependent on the size of a single node), we can solve this space issue by setting $\eta := \log_3(\alpha \lg^2 n / \lg \sigma)$:

Lemma 31. The number of nodes of an η -truncated HSP tree on a substring of length ℓ is bounded by $\mathcal{O}(\ell(\lg \sigma)^{0.7}/(\lg n)^{1.2})$ with $\eta = \log_3(\alpha \lg^2 n / \lg \sigma)$.

Proof. To obtain the upper bound on the number of nodes, we first compute a lower bound on the number of bits taken by the generated substring of an η -node, which is already lower bounded by $2^\eta \lg \sigma$ bits. We begin with changing the base of the logarithm from 3 to $2/3$, and reformulate $\eta = \log_3(\alpha \lg^2 n / \lg \sigma) = (\log_3 2 - 1) \log_{2/3}(\alpha \lg^2 n / \lg \sigma) = \log_{2/3}(\alpha \lg^2 n / \lg \sigma)^{\log_3 2 - 1}$. This gives

$$2^\eta \lg \sigma = 3^\eta (2/3)^\eta \lg \sigma = \alpha (\alpha \lg^2 n / \lg \sigma)^{\log_3 2 - 1} \lg^2 n = (\alpha^{\log_3 2}) (\lg n)^{2 \log_3 2} (\lg \sigma)^{1 - \log_3 2}.$$

With the estimate $0.6 < \log_3 2 < 0.7$ we simplify this to

$$(\alpha^{\log_3 2}) (\lg n)^{2 \log_3 2} (\lg \sigma)^{1 - \log_3 2} > \alpha^{0.6} (\lg n)^{1.2} (\lg \sigma)^{0.3}.$$

Hence, the generated substring of an η -node takes at least $2^\eta \lg \sigma \geq \alpha^{0.6} (\lg n)^{1.2} (\lg \sigma)^{0.3}$ bits.

Finally, the number of nodes is bounded by

$$\ell/2^\eta \leq \ell \lg \sigma / (\alpha^{0.6} (\lg n)^{1.2} (\lg \sigma)^{0.3}) = \ell (\lg \sigma)^{0.7} / (\alpha^{0.6} (\lg n)^{1.2}). \quad \square$$

A consequence is that an η -node with $\eta = \log_3(\alpha \lg^2 n / \lg \sigma)$ generates a substring containing at most $3^\eta = \alpha (\lg n)^2 / (\lg \sigma)$ characters.

Plugging this value of η in Lemma 27 and Lemma 28 yields two corollaries for the η -truncated HSP trees:

Corollary 32. We can compute an η -truncated HSP tree on a substring of length ℓ in $\mathcal{O}(\ell \lg^* n + t_{\text{look}} \ell / 2^\eta + \ell \lg \lg n)$ time. The tree takes $\mathcal{O}(\ell / 2^\eta)$ words of space. We need a working space of $\mathcal{O}(\lg^2 n \lg^* n / \lg \sigma)$ characters.

Proof. The tree has at most $\ell/2^\eta$ nodes, and thus takes $\mathcal{O}(\ell/2^\eta)$ words of space. According to Lemma 27, constructing an η -node uses $\mathcal{O}(3^\eta \lg^* n) = \mathcal{O}(\lg^2 n \lg^* n / \lg \sigma)$ characters as working space. \square

Corollary 33. An LCE query on two η -truncated HSP trees can be answered in $\mathcal{O}(\lg^* n \lg n)$ time.

Proof. LCE queries are answered as in Lemma 28, where the time bound depends on η . Since an η -node generates a substring of at most $3^\eta = \alpha \lg^2 n / \lg \sigma$ characters, we can compare the generated substrings of two η -nodes in $\mathcal{O}(\alpha \lg n)$ time. Overall, we compare $\mathcal{O}(\lg^* n)$ many times two η -nodes, such that these additional costs are bounded by $\mathcal{O}(\lg^* n \lg n)$ time overall, and do not slow down the running time $\mathcal{O}(\lg^* n \lg(n/2^\eta) + \lg^* n \lg n) = \mathcal{O}(\lg^* n \lg n)$. \square

Our second and final goal is to adapt the merging used in the sparse suffix sorting algorithm (Sect. 4.1). Suppose that our algorithm finds two intervals $[i..i+\ell-1]$ and $[j..j+\ell-1]$ with $T[i..i+\ell-1] = T[j..j+\ell-1]$. Ideally, we want to construct $\text{tHT}_\eta(T[i..i+\ell-1])$ in the text space $[j..j+\ell-1]$, leaving $T[i..i+\ell-1]$ untouched so that parts of this substring can be referenced by the η -nodes. Unfortunately, Rules 1 to 4 cannot be applied directly due to our working space limitation. Since we additionally use the text space as working space, we have to be careful about what to overwrite. In particular, we focus on how to

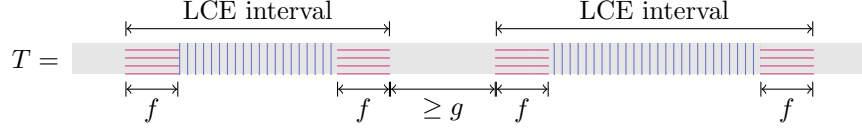


Figure 29: Division of LCE intervals in protected (horizontal lines) and recyclable (vertical lines) parts.

- (a) partition the LCE intervals such that the generated substrings of the fragile non-surrounded η -nodes are protected from becoming overwritten,
- (b) keep enough working space in text space available for merging two trees,
- (c) construct $\text{tHT}_\eta(T[i..i + \ell - 1])$ in the text space $[j..j + \ell - 1]$ when the intervals $[i..i + \ell - 1]$ and $[j..j + \ell - 1]$ overlap, and how to
- (d) bridge the gap $T[e(\mathcal{I}) + 1..b(\mathcal{J}) - 1]$ when merging $\text{tHT}_\eta(T[\mathcal{I}])$ and $\text{tHT}_\eta(T[\mathcal{J}])$ to $\text{tHT}_\eta(T[b(\mathcal{I})..e(\mathcal{J})])$ for two intervals \mathcal{I} and \mathcal{J} with $b(\mathcal{I}) < b(\mathcal{J})$ and $|[e(\mathcal{I}) + 1..b(\mathcal{J}) - 1]| < g$, as performed in Rule 4.

(a) Partitioning of LCE intervals. To merge two η -truncated HSP trees, we have to take special care of those η -nodes that are fragile, because their names can change due to a merge. If the parsing changes the name of an η -node v , we first check whether v 's new name is present in the dictionary. If it is not, we have to create v 's new name consisting of a text position i and a length ℓ such that $T[i..i + \ell - 1] = \text{string}(v)$. The new name of a fragile *surrounded* η -node v can be created easily: According to Lemma 18, the generated substring of v is always a prefix of the generated substring of an already existing η -node w , which is found in the reverse dictionary of the η -nodes. Hence, we can create a new name of v with $\text{string}(w)$.

Unfortunately, the same approach does not work with the non-surrounded η -nodes, because those nodes have generated substrings that are found at the borders of $T[j..j + \ell - 1]$ (remember Fig. 28). If the characters around the borders are left untouched (meaning that we prohibit overwriting these characters), they can be used for creating the names of the fragile non-surrounded η -nodes during a reparsing. To prevent overwriting these characters, we mark both borders of the interval $[j..j + \ell - 1]$ as protected. Conceptually, we partition an LCE interval into (1) **recyclable** and (2) **protected** intervals (see Fig. 29); we free the text of a recyclable interval for overwriting, while prohibiting write access on a protected interval. The recyclable intervals are managed in a dynamic, global list. We keep the property that

Property 6: $f := \lceil 2\alpha \lg^2 n \Delta_L / \lg \sigma \rceil = \Theta(g)$ text positions of the left and right ends of each LCE interval are *protected*.

This property solves the problem for the non-surrounded nodes, because a non-surrounded η -node has a generated substring that is found in $T[j..j + f - 1]$ or $T[j + \ell - 1 - f..j + \ell - 1]$.

(b) Reserving Text Space. We can store the upper part of the η -truncated HSP tree in a recyclable interval, because it needs $\ell/2^n \lg n \leq \ell \alpha^{0.6} (\lg \sigma)^{0.7} / (\lg n)^{0.2} = o(\ell \lg \sigma)$ bits. Since f depends on α and g , we can choose g (the minimum length of a substring on which an η -truncated HSP tree is built) and α (relative to the number of words taken by a single η -truncated HSP tree node) appropriately to always leave $f \lg \sigma / \lg n = \mathcal{O}(\lg^* n \lg n)$ words on a recyclable interval untouched, sufficiently large for the working space needed by Cor. 32. Therefore, we precompute α and g based on the input text T , and set both as *global* constants dependent on T . Since the same amount of free space is needed during a subsequent merging when reparsing an η -node, we add the following property:

Property 7: Each LCE interval has $f \lg \sigma / \lg n$ words of free space left on a recyclable interval.

In our algorithm for sparse suffix sorting, a special problem emerges when two computed LCE intervals overlap. For instance, this can happen when the LCE of a position $i \in \mathcal{P}$ with a position $j \in \mathcal{P}$ overlaps, i.e.,

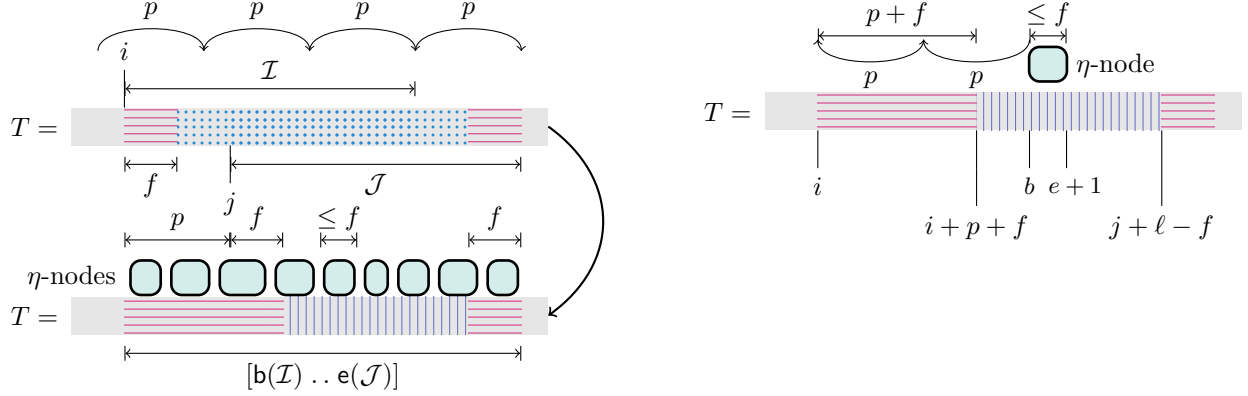


Figure 30: *Left*: Overlapping LCE intervals $\mathcal{I} = [i..i + \ell - 1]$ and $\mathcal{J} = [j..j + \ell - 1]$. *Right*: Finding the generated substring $T[b..e]$ of an η -node in a protected interval. Given that p is the smallest period of $T[\mathcal{I} \cup \mathcal{J}]$, it is sufficient to make $f + p$ characters on the left protected to find the generated substring of all η -nodes of $\text{tHT}_\eta(T[i..j + \ell - 1])$ in $T[i..i + p + f - 1]$.

$[i..i + \text{lce}(i, j) - 1] \cap [j..j + \text{lce}(i, j) - 1] \neq \emptyset$. The algorithm would proceed with merging both overlapping LCE intervals to satisfy Property 5. However, the merged LCE interval cannot respect Property 6 and 7 in general (consider that each interval has a length of $3g$, and both intervals overlap with $2g$ characters). In the case of overlapping, we exploit the periodicity caused by the overlap to make an η -truncated HSP tree fit into both intervals (while still assuring that Property 4 and Property 5 hold, and that we can restore the text).

(c) Interval Overlapping. In a more general setting, suppose that the intervals $\mathcal{I} := [i..i + \ell - 1]$ and $\mathcal{J} := [j..j + \ell - 1]$ with $T[\mathcal{I}] = T[\mathcal{J}]$ overlap, without loss of generality $i < j$. Given $\ell > 2g$, our task is to create $\text{tHT}_\eta(T[i..j + \ell - 1])$ (e.g., needed to comply with Property 4). Since $T[\mathcal{I}] = T[\mathcal{J}]$, the substring $T[i..j + \ell - 1]$ has a period p with $1 \leq p \leq j - i$, i.e., $T[i..j + \ell - 1] = X^k Y$, where $|X| = p$ and Y is a (proper) prefix of X , for an integer k with $k \geq 2$ ($k > 1$ since $j \leq i + \ell - 1$, otherwise $i > j$ or $\mathcal{I} \cap \mathcal{J} = \emptyset$). First, we compute the smallest period $p \leq j - i$ of $T[i..j + \ell - 1]$ in $\mathcal{O}(\ell)$ time [27]. By definition, each substring of $T[i + p..j + \ell - 1]$ appears also p characters earlier. We treat the substring $T[i..i + p + f - 1]$ as a reference and therefore mark it protected. Keeping the original characters in $T[i..i + p + f - 1]$, we can restore the generated substrings of every η -node by an arithmetic progression. This can be seen by two facts: First, the length of the generated substring of an η -node is at most $3^\eta = \alpha \lg^2 n / \lg \sigma \leq f/2$. Second, given an η -node with the generated substring $T[b..e]$ with $i + p + f \leq e \leq j + \ell - 1$, we find an integer k with $k \geq 0$ such that $T[b..e] = T[b - p^k..e - p^k]$ and $[b - p^k..e - p^k] \subseteq [i..i + p + f - 1]$ (since $e - b \leq f/2$). Hence, we can make the interval $[i + p + f + 1..j + \ell - 1 - f]$ *recyclable*, which is at least as large as f , since $|\mathcal{I} \cup \mathcal{J}| \geq j - i + 2g \geq p + 2g$ is at least $p + 3f$ for a sufficiently large g . The partitioning into protected and recyclable intervals is illustrated in Fig. 30.

For the actual merging operation, we elaborate an approach that respects Properties 6 and 7:

(d) Merging with a Gap. We introduce a merge operation that supports the merging of two η -truncated HSP trees whose LCE intervals have a gap of less than g characters. The difference to Lemma 25 is that we additionally build new η -nodes on the gap between both trees. The η -nodes whose generated substrings intersect with the gap are called **bridging** nodes.

Let $\text{tHT}_\eta(T[\mathcal{I}])$ and $\text{tHT}_\eta(T[\mathcal{J}])$ be built on two LCE intervals \mathcal{I} and \mathcal{J} with $1 \leq \text{b}(\mathcal{J}) - \text{e}(\mathcal{I}) \leq g$. Our task is to compute the merged tree $\text{tHT}_\eta(T[\text{b}(\mathcal{I}).. \text{e}(\mathcal{J})])$. We do that by (a) reprocessing $\mathcal{O}(\Delta_L + \Delta_R)$ nodes at every height of both trees (according to Lemma 25), and (b) building the bridging nodes connecting both trees. Like with the non-surrounded nodes, the generated substring of a bridging node can be a unique substring of the text. This means that overwriting $T[\text{e}(\mathcal{I}) - f.. \text{b}(\mathcal{J}) + f]$ would invalidate the generated substrings of the bridging nodes and of some (formerly) non-surrounded nodes. Therefore, we mark the

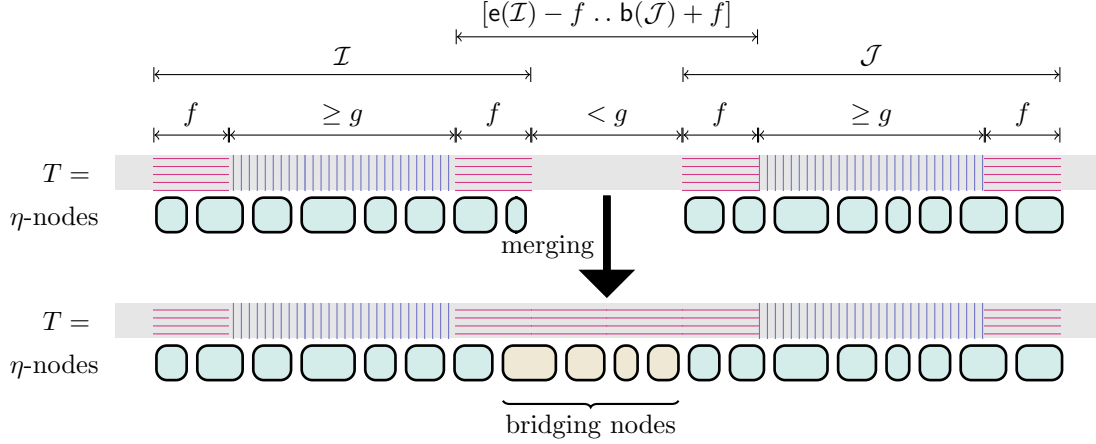


Figure 31: Merging $\text{tHT}_\eta(T[I])$ and $\text{tHT}_\eta(T[J])$ with $b(J) - g \leq e(I) \leq b(J) - 1$. The substring $T[e(I) - f .. b(J) + f]$ is marked protected for the sake of the bridging nodes.

interval $[e(I) - f .. b(J) + f]$ as protected. By doing so, we can use the characters of $T[e(I) - f .. b(J) + f]$ to

- create the bridging η -nodes, and to
- reparse the non-surrounded nodes of both trees (Fig. 31).

The bridging nodes and their ancestors take $o(\lg n \lg^* n)$ words of additional space since building $\text{tHT}_\eta(T[e(I) + 1 .. b(J) - 1])$ with $|b(J) - e(I)| = \mathcal{O}(g)$ takes $(g/2^\eta) \lg n = o(g \lg \sigma) = o(\lg^* n \lg^2 n)$ bits (or $o(\lg^* n \lg n)$ words) of space. By choosing g and α sufficiently large, we can store the bridging nodes in a recyclable interval while maintaining Property 7 for the merged LCE interval. Finally, the time bound for this merging strategy is given in the following corollary:

Corollary 34. Given two LCE intervals \mathcal{I} and \mathcal{J} with $b(\mathcal{I}) \leq b(\mathcal{J}) \leq e(\mathcal{I}) + g$, we can build $\text{tHT}_\eta(T[b(\mathcal{I}) .. e(\mathcal{J})])$ in $\mathcal{O}(g \lg^* n + t_{\text{look}} g / 2^\eta + g\eta / \log_\sigma n + t_{\text{look}} \lg^* n \lg n)$ time.

Proof. We adapt the merging of two HSP trees (Lemma 25) for the η -truncated HSP trees. The difference to Lemma 25 is that we reparse an η -node by rebuilding its local surrounding consisting of $\mathcal{O}((\Delta_L + \Delta_R)3^\eta)$ nodes that take $\alpha(\Delta_L + \Delta_R) \lg^2 n / \lg \sigma \leq f$ words for a sufficiently large α . According to Property 7, there are at least f words of space left in a recyclable interval to recompute an η -node, and to create the bridging nodes in the fashion of Cor. 32. Both creating and recomputing takes overall $\mathcal{O}(g \lg^* n + t_{\text{look}} g / 2^\eta + g\eta / \log_\sigma n)$ time. \square

There is one problem left before we can prove the main result of the paper: The sparse suffix sorting algorithm of Sect. 4.1 creates LCE intervals on substrings smaller than g between two LCE intervals temporarily when applying Rule 3. We cannot afford to build such tiny η -truncated HSP trees, since they cannot respect Property 6 and Property 7. Due to Rule 4, we eventually merge a temporarily created dynLCE with a dynLCE on a long LCE interval. Instead of temporarily creating an η -truncated HSP tree covering less than g characters, we apply the new merge operation of Cor. 34 directly, merging two trees that have a gap of less than g characters. With this and the other properties stated above, we come to the final proof:

Proof of Thm. 1. The analysis is split into suffix comparison, tree generation and tree merging:

- Suffix comparisons are done as in Cor. 24. LCE queries on η -truncated HSP trees and HSP trees are conducted in the same time bounds (compare Lemma 21 with Cor. 33).

- All positions considered for creating the η -truncated HSP trees belong to \mathcal{C} . Constructing the η -truncated HSP trees costs $\mathcal{O}(|\mathcal{C}| \lg^* n + t_{\text{look}} |\mathcal{C}| / 2^\eta + |\mathcal{C}| \lg \lg n)$ overall time, due to Cor. 32.
- Merging in the fashion of Cor. 34 does not affect the overall time: Since a merge of two trees introduces less than g new text positions to an LCE interval, we conclude with the same analysis as in Thm. 26 that the time for merging is upper bounded by the construction time.

Plugging the times for suffix comparisons, tree construction and merging in Cor. 24 yields the overall time

$$\mathcal{O}(|\mathcal{C}| \lg^* n + t_{\text{look}} |\mathcal{C}| / 2^\eta + |\mathcal{C}| \lg \lg n) = \mathcal{O}(|\mathcal{C}| (t_{\text{look}} (\lg \sigma)^{0.7} / (\lg n)^{1.2} + \lg \lg n)) = \mathcal{O}(|\mathcal{C}| (\sqrt{\lg \sigma} + \lg \lg n)),$$

since $t_{\text{look}} = \mathcal{O}(\lg n)$. The time for searching and sorting the suffixes is $\mathcal{O}(m \lg m \lg^* n \lg n)$. The auxiliary data structures used are $\text{SAVL}(\text{Suf}(\mathcal{P}))$, the search tree \mathcal{L} for the LCE intervals, and the list of recyclable intervals, each taking $\mathcal{O}(m)$ words of space. \square

6 Conclusion

In the first part, we introduced the HSP trees based on the ESP technique as a new data structure that (a) answers LCE queries, and (b) can merge with another HSP tree to form a larger HSP tree. With these properties, HSP trees are an eligible choice for the mergeable LCE data structure needed for the sparse suffix sorting algorithm presented here.

In the second part, we developed a truncated version with a trade-off parameter determining the height at which to cut off the lower nodes. Setting the trade-off parameter adequately, the truncated HSP tree fits into text-space. As a result of independent interest, we obtained an LCE data structure with a trade-off parameter, like other already known solutions. Although not shown here, an ESP tree can similarly (a) answer LCE queries, (b) be merged, and (c) be truncated. However, answering LCE queries or merging two ESP trees is by a factor of $\mathcal{O}(\lg n)$ slower than when the operations are performed with HSP trees.

In the appendix, we noted that the maximum number of fragile nodes in an ESP tree of a string of length n can be at least $\Omega(\lg^2 n)$, which invalidates the upper bound of $\mathcal{O}(\lg n \lg^* n)$ on the maximal number of fragile nodes postulated in [11]. This result also invalidates theoretical results that depend on the ESP technique (e.g., for approximating the edit distance with moves [11] or the LZ77 factorization [10], or for building indexes [41, 15, 30, 40]). We could quickly provide a new upper bound of $\mathcal{O}(\lg^2 n \lg^* n)$, but it remains an open problem to refine our bounds. Luckily, our proposed HSP technique can be used as a substitution for the ESP technique, since HSP trees and ESP trees share the same bounds for construction time and space usage. By switching to the HSP technique, we regain the promised $\mathcal{O}(\lg n \lg^* n)$ number of fragile nodes. It is easy to see that this result also recovers the postulated $\mathcal{O}(\lg n \lg^* n)$ approximation bound on the edit distance matching problem [11, 41]: Given $\text{ET}(T)$ of a string T of length n , it is assumed by Cormode and Muthukrishnan [11, Theorem 7] that changing/deleting a character of T , or inserting a character in T changes $\mathcal{O}(\lg^* n \lg n)$ nodes in $\text{ET}(T)$. Although we only provided proofs that pre-/appending characters to T changes $\mathcal{O}(\lg^* n \lg n)$ nodes of $\text{HT}(T)$, it is easy to generalize this result by applying a merge operation: Given that we insert a character $c \in \Sigma$ between $T[i]$ and $T[i+1]$, the trees $\text{HT}(T)$ and $\text{HT}(T[1..i]cT[i+1..])$ differ in at most $\mathcal{O}(\lg^* n \lg n)$ nodes, since appending c to $\text{HT}(T[1..i])$ and merging $\text{HT}(T[1..i]c)$ with $\text{HT}(T[i+1..])$ changes $\mathcal{O}(\lg^* n \lg n)$ nodes. The same can be observed when deleting or changing the i -th character.

In the light of the theoretical improvements of the HSP over the ESP, it is interesting to evaluate how HSP behaves practically. Especially, we are interested in how well HSP behaves in the context of grammar compression [3] like the ESP-index [30, 40] on highly repetitive texts, where a more stable behavior of the repetitive nodes could lead to an improved compression ratio.

From the theoretical point of view, it would be interesting to compute the sparse suffix sorting with a trade-off parameter adjusting space and query time such that this parameter can be chosen from a continuous domain like the result we presented for the LCE query data structure.

In the case that we can impose a restriction on the set of suffixes to sort, Kärkkäinen and Ukkonen [24] presented a sparse suffix sorting algorithm running in optimal $\mathcal{O}(n)$ time while using $\mathcal{O}(m)$ words of space,

given that \mathcal{P} is a set of equally spaced text positions. We wonder whether it is also possible to gain a benefit when only every i -th entry of SA is needed, i.e., the order of each i -th lexicographically smallest suffix for an arithmetic progression $i = c, 2c, 3c, \dots$ with a constant integer $c \geq 2$. Related to this problem is the suffix selection problem, i.e., to find the i -th lexicographically smallest suffix for a given integer i . Interestingly, Franceschini and Muthukrishnan [14] showed that the suffix selection problem can be solved in $\mathcal{O}(n)$ time in the comparison model, whereas suffix sorting is solved in $\Theta(n \lg n)$ time within the same model.

References

- [1] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proc. SODA*, pages 819–828. ACM/SIAM, 2000.
- [2] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. FOCS*, pages 714–721. IEEE Computer Society, 1994.
- [3] H. Bannai. Grammar compression. In *Encyclopedia of Algorithms*, pages 861–866. Springer, 2016.
- [4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. SODA*, pages 360–369. ACM/SIAM, 1997.
- [5] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014.
- [6] P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In *Proc. CPM*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015.
- [7] P. Bille, J. Fischer, I. L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse text indexing in small space. *ACM Trans. Algorithms*, 12(3):39:1–39:19, 2016.
- [8] T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the “restore” model. In *Proc. SODA*, pages 995–1004. SIAM, 2014.
- [9] C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000.
- [10] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. SODA*, pages 321–330. SIAM, 2005.
- [11] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1):2:1–2:19, 2007.
- [12] J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In *LATIN*, volume 9644 of *LNCS*, pages 483–496. Springer, 2016.
- [13] G. Franceschini and R. Grossi. No sorting? Better searching! *ACM Trans. Algorithms*, 4(1):2:1–2:13, 2008.
- [14] G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In *Proc. STOC*, pages 328–337, 2007.
- [15] S. Fukunaga, Y. Takabatake, T. I, and H. Sakamoto. Online grammar compression for frequent pattern discovery. In *Proc. ICGI*, volume 57 of *Workshop and Conference Proceedings*, pages 93–104. JMLR, 2016.
- [16] P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. SODA*, pages 425–439. SIAM, 2017.

- [17] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. STOC*, pages 315–324. ACM, 1987.
- [18] K. Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. *ArXiv CoRR*, abs/1703.01009, 2017.
- [19] T. I. Longest common extensions with recompression. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl, 2017.
- [20] T. I, J. Kärkkäinen, and D. Kempa. Faster sparse suffix sorting. In *Proc. STACS*, volume 25 of *LIPIcs*, pages 386–396. Schloss Dagstuhl, 2014.
- [21] R. W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. *J. Discrete Algorithms*, 1(5-6):387–408, 2003.
- [22] G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.
- [23] J. Kärkkäinen and D. Kempa. LCP array construction using $\mathcal{O}(\text{sort}(n))$ (or less) I/Os. In *Proc. SPIRE*, volume 9954 of *LNCS*, pages 204–217. Springer, 2016.
- [24] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON*, volume 1090 of *LNCS*, pages 219–230. Springer, 1996.
- [25] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
- [26] Z. Khan, J. S. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.
- [27] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS*, pages 596–604. IEEE Computer Society, 1999.
- [28] R. Kolpakov, G. Kucherov, and T. A. Starikovskaya. Pattern matching on sparse suffix trees. In *Proc. CCP*, pages 92–97, 2011.
- [29] Z. Li, J. Li, and H. Huo. Optimal in-place suffix sorting. *ArXiv CoRR*, abs/1610.08305, 2016.
- [30] S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.
- [31] K. Mehlhorn, R. Sundar, and C. Urig. Maintaining dynamic sequences under equality-tests in poly-logarithmic time. In *Proc. SODA*, pages 213–222. SIAM, 1994.
- [32] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. MFCS*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl, 2016.
- [33] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.
- [34] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.
- [35] N. Prezza. In-place sparse suffix sorting. In *Proc. SODA*, pages 1496–1508. SIAM, 2018.
- [36] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):1–31, 2007.

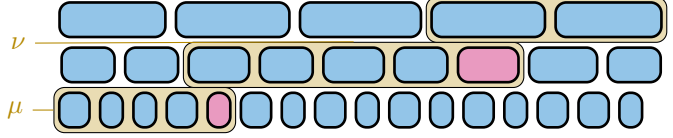
- [37] N. Rahman and R. Raman. Rank and select operations on binary strings. In *Encyclopedia of Algorithms*, pages 748–751. Springer, 2008.
- [38] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proc. STOC*, pages 300–309. ACM, 1994.
- [39] H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozone. A space-saving approximation algorithm for grammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009.
- [40] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proc. SEA*, volume 8504 of *LNCS*, pages 338–350. Springer, 2014.
- [41] Y. Takabatake, K. Nakashima, T. Kuboyama, Y. Tabei, and H. Sakamoto. siEDM: An efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26:1–26:18, 2016.
- [42] Y. Tanimura, T. I, H. Bannai, S. Inenaga, S. J. Puglisi, and M. Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Proc. CPM*, volume 54 of *LIPICs*, pages 1:1–1:10. Schloss Dagstuhl, 2016.
- [43] Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In *Proc. MFCS*, volume 83 of *LIPICs*, pages 10:1–10:15. Schloss Dagstuhl, 2017.
- [44] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

A A Lower Bound on the Number of Fragile ESP Tree Nodes

Here, we present two examples revealing that the ESP technique changes $\Omega(\lg^2 n)$ nodes when changing a single character. Each example contains a large number of **typeM** meta-blocks in a specific constellation. Remembering how the ESP technique parses its input, a remaining single symbol neighbored by two repeating meta-blocks is fused with one of them to form a **typeM** meta-block. Regardless of whether we favor to fuse a remaining symbol with either its preceding or succeeding (cf. Rule M) repeating meta-block to form a **typeM** meta-block, for each of the two tie breaking rules, we give an example string of length at most n whose ESP tree has $\Omega(\lg^2 n)$ fragile nodes. These examples contradict Lemma 9 in [11], where it is claimed that there are $\mathcal{O}(\lg^* n \lg n)$ fragile nodes in the ESP tree of a text of length n .

A.1 Fusing with the Preceding Repeating Meta-Block

Consider a **type 1** meta-block μ whose rightmost node is fragile. If the leftmost node of a repeating meta-block ν is built on μ 's rightmost node, then the rightmost node of ν can also be fragile.



Having this idea in mind, we build an example consisting of a chain of repeating meta-blocks, where the leftmost node of a repeating meta-block is built on the fragile rightmost node of a meta-block of one depth below (shaded in the right picture). The main idea is the following: Each meta-block of this chain can be of arbitrary (but sufficiently long) length. Keeping in mind that changing the name of a node means that the names of its ancestors also have to change, we can create an example string whose ESP tree contains fragile nodes appearing on each height at arbitrary positions:

Example 35. Let \mathbf{a}, \mathbf{b} and $\mathbf{c} \in \Sigma$ be three different characters. The text $Y := (X_0)^{3^k} (X_1)^{3^{k-1}} (X_2)^{3^{k-2}} \cdots (X_{k-1})^3$ with $k := \lfloor \log_3(n / \log_3 n) \rfloor$ has a length at most n , and its ESP tree has $\Omega(\lg^2 n)$ fragile nodes, where

$$X_0 := \mathbf{a}, \text{ and } X_i := \begin{cases} X_{i-1}^2 \mathbf{b}^{3^{i-1}} & \text{if } i \text{ is odd,} \\ X_{i-1}^2 \mathbf{c}^{3^{i-1}} & \text{if } i \text{ is even,} \end{cases} \quad \text{for } i = 1, \dots, k. \quad \begin{array}{l} X_0 = \mathbf{a}, \\ X_1 = \mathbf{aab}, \\ X_2 = \mathbf{aabaabc}^3, \\ X_3 = X_2^2 \mathbf{b}^9, \\ \vdots \end{array}$$

To show that the claim in the example is correct, we insert a lemma showing the associativity of esp on a special class of strings:

Lemma 36. Contrary to Rule M, assume that we favor fusing a remaining character with its preceding meta-block to form a **typeM** meta-block. Given a height h and two strings X, Y that are either empty or have a length of at least $2 \cdot 3^{h-1}$, $esp^{(h)}(X \mathbf{b}^{3^i} Y) = esp^{(h)}(X) esp^{(h)}(\mathbf{b}^{3^i}) esp^{(h)}(Y)$ if $i \geq h$, \mathbf{b} is neither a suffix of X nor a prefix of Y , and there is no prefix of $esp^{(j)}(Y)$ of the form \mathbf{cd}^k for some characters $\mathbf{c}, \mathbf{d} \in \Sigma_j$ with $\mathbf{c} \neq \mathbf{d}$, and integers k, j with $k \geq 2$ and $0 \leq j \leq h-1$.

Proof. The additional requirement for Y is to ensure that the leftmost block of $esp^{(j)}(Y)$ is not a non-repetitive **typeM** block that has been fused to its succeeding meta-block, only because it has no preceding meta-block. Regardless of which characters are prepended to $esp^{(j-1)}(Y)$, the first character of such a block would form with its preceding characters a new block.

For $h = 1$, esp divides the string $X \mathbf{b}^{3^i} Y$ into meta-blocks such that there is one **type 1** meta-block μ that exactly contains the substring \mathbf{b}^{3^i} . That is because of the following: If X (resp. Y) is not the empty string, then it contains at least two characters. Since we favor fusing with the preceding meta-block, there is no chance that characters of X can enter μ . Assume that Y is not the empty string. Since the first block of $esp(Y)$ is neither a non-repetitive **typeM** block nor a block starting with \mathbf{b} , it is not possible that characters of this block can enter μ .

Under the assumption that the claim holds for a given $h-1 \geq 0$, we have

$$esp^{(h)}(X \mathbf{b}^{3^{i+1}} Y) = esp(esp^{(h-1)}(X \mathbf{b}^{3^{i+1}} Y)) = esp(esp^{(h-1)}(X) esp^{(h-1)}(\mathbf{b}^{3^{i+1}}) esp^{(h-1)}(Y)).$$

The strings $esp^{(h)}(X)$ and $esp^{(h)}(Y)$ are either empty or contain at least two characters. Since $i \geq h$, $esp^{(h-1)}(\mathbf{b}^{3^i})$ is the repetition of the same character. This repetition has a length of at least three such that we can apply the shown associativity for $h = 1$ to show the claim. \square

Proof of Ex. 35. We start with determining the length of Y . Since $|X_0| = 3^0$, under the assumption that $|X_i| = 3^i$, we obtain that $|X_{i+1}| = 2|X_i| + 3^i = 3^{i+1}$. Therefore, $|X_i^{3^{k-i}}| = 3^k$ for all $i = 0, \dots, k-1$. We conclude that the length of Y is at most n , since $|Y| = k3^k \leq n \log_3(n/\log_3 n)/\log_3 n \leq n$.

We now show that each substring X_i of Y is the generated substring of a node x_i of $\mathbf{ET}(Y)$ on height i whose subtree is equal to the perfect ternary subtree $T_i := \mathbf{ET}(X_i)$, for $i = 1, \dots, k-1$. This is true for $i = 1, 2, 3$, as can be seen in Fig. 32. For the general case, we adapt the associativity shown for esp in Lemma 36 to the string X_i :

Sub-Claim. For every i with $0 \leq i \leq k-2$ it holds that

$$(I) \quad |esp^{(i+1)}(X_{i+1})| = 1,$$

$$(II) \quad esp^{(h)}(X_{i+1}) = esp^{(h)}(X_i X_i \mathbf{d}_i^{3^i}) = esp^{(h)}(X_i X_i) esp^{(h)}(\mathbf{d}_i^{3^i}) = esp^{(h)}(X_i) esp^{(h)}(X_i) esp^{(h)}(\mathbf{d}_i^{3^i}), \text{ and}$$

$$(III) \quad esp^{(h)}(X_{i+1}) \text{ starts with a repetition of a character,}$$

for every h with $0 \leq h \leq i$, where \mathbf{d}_i is a character with $\mathbf{d}_i = \mathbf{b}$ if i is even, otherwise $\mathbf{d}_i = \mathbf{c}$.

Sub-Proof. For $i = 0$ we have

$$(I) \quad |esp^{(1)}(X_1)| = |esp(\mathbf{aab})| = 1 \text{ (aab is put in a type M meta-block having exactly one block),}$$

$$(II) \quad esp^{(0)}(X_1) = X_1, \text{ and}$$

$$(III) \quad X_1 = \mathbf{aab} \text{ starts with a repetition of the character } \mathbf{a}.$$

Under the assumption that the claim holds for an integer i , we conclude that it holds for $i+1$ due to

$$\begin{aligned} esp^{(h)}(X_{i+2}) &= esp^{(h)}(X_{i+1} X_{i+1} \mathbf{d}_{i+1}^{3^{i+1}}) \\ &= esp^{(h)}(X_i X_i \mathbf{d}_i^{3^i} X_i X_i \mathbf{d}_i^{3^i} \mathbf{d}_{i+1}^{3^{i+1}}) \\ (\text{Lemma 36, } \mathbf{d}_i \neq \mathbf{d}_{i+1}) &= esp^{(h)}(X_i X_i \mathbf{d}_i^{3^i} X_i X_i \mathbf{d}_i^{3^i}) esp^{(h)}(\mathbf{d}_{i+1}^{3^{i+1}}) \\ (\text{Lemma 36, (I) or (III)}) &= esp^{(h)}(X_i X_i) esp^{(h)}(\mathbf{d}_i^{3^i}) esp^{(h)}(X_i X_i) esp^{(h)}(\mathbf{d}_i^{3^i}) esp^{(h)}(\mathbf{d}_{i+1}^{3^{i+1}}) \\ (\text{Lemma 36, (I) or (III)}) &= esp^{(h)}(X_i X_i \mathbf{d}_i^{3^i}) esp^{(h)}(X_i X_i \mathbf{d}_i^{3^i}) esp^{(h)}(\mathbf{d}_{i+1}^{3^{i+1}}) \\ &= esp^{(h)}(X_{i+1}) esp^{(h)}(X_{i+1}) esp^{(h)}(\mathbf{d}_{i+1}^{3^{i+1}}) \end{aligned}$$

for $1 \leq h \leq i$. The conditions of Lemma 36 hold because \mathbf{d}_i is neither a prefix nor a suffix of X_i , $\mathbf{d}_i \neq \mathbf{d}_{i+1}$, $|X_i X_i| = 2 \cdot 3^i$, and $esp^{(h)}(X_i X_i)$ starts with a repetition of a character due to

$$\begin{cases} (III) & \text{for } h < i, \text{ or due to} \\ esp^{(i)}(X_i X_i) =^{(II)} esp^{(i)}(X_i) esp^{(i)}(X_i) \text{ and (I)} & \text{for } h = i. \end{cases}$$

For $h = i+1$ we use that (I) holds for X_i , $|esp^{(i)}(\mathbf{d}_i^{3^i})| = 1$, and $esp^{(i)}(\mathbf{d}_{i+1}^{3^{i+1}})$ is a repetition of length 3 of the same character, to obtain

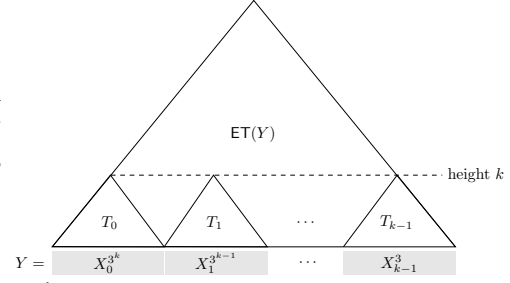
$$\begin{aligned} esp^{(i+1)}(X_{i+2}) &= esp(esp^{(i)}(X_{i+2})) \\ &= esp(esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i}) esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i}) esp^{(i)}(\mathbf{d}_{i+1}^{3^{i+1}})) \\ (\text{Lemma 36}) &= esp(esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i}) esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i})) esp(esp^{(i)}(\mathbf{d}_{i+1}^{3^{i+1}})) \\ (\text{evaluate and reformulate}) &= esp(esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i})) esp(esp^{(i)}(X_i X_i) esp^{(i)}(\mathbf{d}_i^{3^i})) esp(esp^{(i)}(\mathbf{d}_{i+1}^{3^{i+1}})), \end{aligned}$$

where we used that another application of esp puts $esp^{(i)}(X_i X_i) esp^{(i)}(d_i^{3^i})$ into a single **typeM** meta-block of length three, and that d_i is neither a prefix nor a suffix of X_i . This concludes (II). A consequence is (III): For $h \leq i$ we have $esp^{(h)}(X_{i+2}) = esp^{(h)}(X_{i+1}) esp^{(h)}(X_{i+1}) esp^{(h)}(d_{i+1}^{3^{i+1}})$, and $esp^{(h)}(X_{i+1})$ starts with a repetition of a character according to our assumption. For $h = i + 1$ we have $esp^{(i+1)}(X_{i+2}) = esp(esp^{(i)}(X_i) esp^{(i)}(X_i) esp^{(i)}(d_i^{3^i}) esp^{(i)}(X_i) esp^{(i)}(X_i) esp^{(i)}(d_i^{3^i}) esp^{(i)}(d_{i+1}^{3^{i+1}}))$. Due to (I), $|esp^{(i)}(X_i)| = |esp^{(i)}(d_i^{3^i})| = 1$; hence the last application of esp creates three blocks, where each of the first two represents the string $esp^{(i)}(X_i) esp^{(i)}(X_i) esp^{(i)}(d_i^{3^i})$ of length three. Another application of esp yields (I). ■

Let b_i and c_i denote the names of the roots of $ET(b^{3^i})$ and of $ET(c^{3^i})$, respectively. Set $d_i := b_i$ if i is even, otherwise $d_i := c_i$. Then $\langle X_{i+1} \rangle_{i+1} = x_{i+1}$ due to Sub-Claim (I), and $\langle X_{i+1} \rangle_i = x_i x_i d_i$ due to Sub-Claim (II). Consequently,

$$esp(((\langle X_{i+1} \rangle_i)^{3^{k-i-1}})) = esp((x_i x_i d_i)^{3^{k-i-1}}) = (esp(x_i x_i d_i))^{3^{k-i-1}} = x_{i+1}^{3^{k-i-1}}. \quad (1)$$

This means that $\langle X_i \rangle_h^{3^{k-i}} = \langle X_i^{3^{k-i}} \rangle_h$ is a repetition of length 3^{k-h} consisting of the same name, for every height $h = i, \dots, k$. We conclude that $T_i := ET((X_i)^{3^{k-i}})$ is a perfect ternary tree.



Finally, we show that $esp^{(h)}(Y) = esp^{(h)}(X_1^{3^k}) \dots esp^{(h)}(X_{k-1}^{3^1})$ for each height h with $1 \leq h \leq k$. On the one hand, we have

$$\begin{aligned} esp^{(h)}(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}) &= esp^{(h)}(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} d_{i-1}^{3^{i-1}} X_{i+1}^{3^{k-i-1}}) \\ \text{(III) with } 0 \leq i \leq h-2 &= esp^{(h)}(X_i^{3^{k-i-1}} X_{i-1} X_{i-1}) esp^{(h)}(d_{i-1}^{3^{i-1}}) esp^{(h)}(X_{i+1}^{3^{k-i-1}}) \\ &= esp^{(h)}(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} d_{i-1}^{3^{i-1}}) esp^{(h)}(X_{i+1}^{3^{k-i-1}}) \\ &= esp^{(h)}(X_i^{3^{k-i}}) esp^{(h)}(X_{i+1}^{3^{k-i-1}}) \end{aligned} \quad (2)$$

for $1 \leq h \leq i-1$ due to Lemma 36. On the other hand, we have

$$\begin{aligned} esp^{(h)}(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}) &= esp^{(h-i+1)}(esp^{(i-1)}(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}})) \\ \text{Eq. (2)} &= esp^{(h-i+1)}(esp^{(i-1)}(X_i^{3^{k-i}}) esp^{(i-1)}(X_{i+1}^{3^{k-i-1}})) \\ \text{Eq. (1)} &= esp^{(h-i)}(esp((x_{i-1} x_{i-1} d_{i-1})^{3^{k-i}} (x_{i-1} x_{i-1} d_{i-1} x_{i-1} x_{i-1} d_{i-1} \langle d_i^{3^i} \rangle_{i-1})^{3^{k-i-1}})) \\ \text{(apply } esp) &= esp^{(h-i)}(x_i^{3^{k-i}} (x_i x_i d_i)^{3^{k-i-1}}) \\ &= esp^{(h-i-1)}(esp(x_i^{3^{k-i}} x_i x_i d_i) esp((x_i x_i d_i)^{3^{k-i-2}})) \\ \text{(evaluate and reformulate)} &= esp^{(h-i-1)}(esp(x_i^{3^{k-i}}) esp((x_i x_i d_i)^{3^{k-i-1}})) \\ \text{Eq. (1)} &= esp^{(h-i-1)}(esp(x_i^{3^{k-i}}) esp(x_{i+1}^{3^{k-i-1}})) \\ \text{(Lemma 36)} &= esp^{(h-i)}(x_i^{3^{k-i}}) esp^{(h-i)}(x_{i+1}^{3^{k-i-1}}) \end{aligned} \quad (3)$$

for $i \leq h \leq k$. It is easy to extend the pairwise associativity $X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}$ for each i with $0 \leq i \leq k-2$ to $X_1^{3^k} \dots X_{k-1}^{3^1}$. This concludes that the root of T_i has the same name as the i -th leftmost node of $ET(Y)$ on height k . Figure 33(left) shows an excerpt of T_i and T_{i+1} . The crucial step in Eq. (3) is the re-formulation of the parsing

$$esp^{(h-i-1)}(\underbrace{esp(x_i^{3^{k-i}} x_i x_i d_i)}_{\text{belongs to } T_i} \underbrace{esp((x_i x_i d_i)^{3^{k-i-2}})}_{\text{belongs to } T_{i+1}}) = esp^{(h-i-1)}(esp(x_i^{3^{k-i}}) \underbrace{esp((x_i x_i d_i)^{3^{k-i-1}})}_{=:\mu_{i+1}}) \quad (4)$$

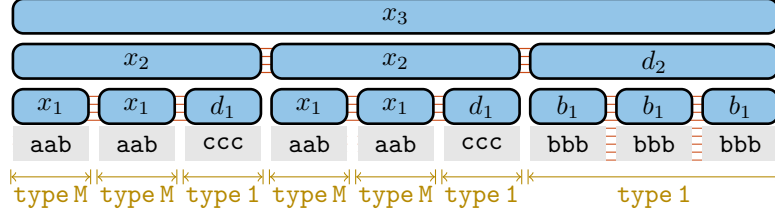


Figure 32: $\text{ET}(X_3)$ as defined in Ex. 35. The subtree of each node with name x_i is equal to $\text{ET}(X_i)$.

showing that there is a **type 1** meta-block μ_{i+1} covering all nodes of T_{i+1} and the rightmost node of T_i , on height $i + 1$. This meta-block is a repetition of the character $\text{esp}(x_i x_i d_i) = x_{i+1} \in \Sigma_{h+1}$.

Given that μ_0 is the first **type 1** meta-block of $\text{esp}(Y)$ (covering the prefix $X_0^{3^h+2}$), we now examine what happens with μ_i for each i with $0 \leq i \leq h - 1$ when removing the first **a** from Y . Let us call the shortened string Y' , i.e., $Y = \mathbf{a}Y'$. On removing the first **a** from Y , we claim that the meta-block μ_i contains one character x_i less, for every i with $0 \leq i \leq h - 1$ (cf. Fig. 33 showing the difference between $\langle Y \rangle_i$ and $\langle Y' \rangle_i$ on height i with $0 \leq i \leq k - 1$): For μ_0 , this is trivial. For an $i \geq 0$, focus on the substring $X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}$ of Y : We have

$$\begin{aligned} \text{esp}(\langle X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}} \rangle_i) &= \text{esp}(x_i^{3^{k-i}} (x_i x_i d_i)^{3^{k-i-1}}) = \text{esp}(\underbrace{x_i^{3^{k-i}}}_{\text{suffix of } \mu_i}) \underbrace{\text{esp}((x_i x_i d_i)^{3^{k-i-1}})}_{=\mu_{i+1}} \\ &= \text{esp}(x_i^{3^{k-i}}) x_{i+1}^{3^{k-i-1}} \end{aligned}$$

due to Eq. (3). Under the assumption that removing the first character **a** from Y causes μ_i to shrink by one character $x_i \in \Sigma_i$, we get

$$\begin{aligned} \text{esp}(x_i^{3^{k-i-1}-1} (x_i x_i d_i)^{3^{k-i-1}}) &= \text{esp}(x_i^{3^{k-i}} x_i d_i) \text{esp}((x_i x_i d_i)^{3^{k-i-1}-1}) \\ &= \text{esp}(x_i^{3^{k-i}} x_i d_i) x_{i+1}^{3^{k-i-1}-1} \\ &\neq \text{esp}(x_i^{3^{k-i-1}}) x_{i+1}^{3^{k-i-1}}. \end{aligned}$$

We observe that the length of μ_i is decremented by one, causing the name of its rightmost block to change, which is the leftmost node of T_{i+1} on height $i + 1$, and the first character of μ_{i+1} . Due to the tie breaking rule, this block gets fused with its preceding meta-block at height $i + 1$, decrementing the length of its succeeding meta-block μ_{i+1} by one (and hence, this process repeats for all $i = 0, \dots, k - 2$). This means that the leftmost node on height i of T_i changes, for $1 \leq i \leq k - 1$. Each of these nodes receives a new name such that it is fused with its preceding **type 1** meta-block to form a **type M** meta-block. Since changing a node on height i changes all its ancestors, at least $k - i$ nodes are changed in T_i . In total, at least $k + (k - 1) + (k - 2) + \dots + 2 = (k^2 + k)/2 - 1$ nodes are changed. Hence there is a lower bound of $\Omega(k^2) = \Omega(\log_3^2(n/\log_3 n)) = \Omega(\lg^2 n)$ fragile nodes. \square

Note that the later introduced HSP technique (see Sect. 3) with the same tie breaking rule also produces $\Omega(\lg^2 n)$ fragile nodes in this example.

A.2 Fusing with the Succeeding Repeating Meta-Block

The idea is similar to the previous example. In particular, we introduce a corollary of Lemma 36:

Corollary 37. Given a height h and a string Y that is either empty or has a length of at least $2 \cdot 3^{h-1}$, $\text{esp}^{(h)}(XY) = \text{esp}^{(h)}(X) \text{esp}^{(h)}(Y)$ if **a** is not a prefix of Y , where $X = \mathbf{b}^{3^i} \mathbf{a}^{3^j}$ with $i + j \geq h$, and $\mathbf{a}, \mathbf{b} \in \Sigma$ with $\mathbf{a} \neq \mathbf{b}$.

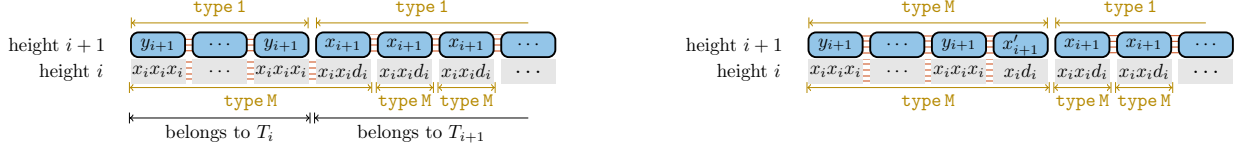


Figure 33: Differences between $\text{ET}(Y)$ (left) and $\text{ET}(Y')$ (right) on the heights i and $i+1$, where $Y = \mathbf{a}^{3^k}(\mathbf{a}^2\mathbf{b})^{3^{k-1}}((\mathbf{a}^2\mathbf{b})^2\mathbf{c}^3)^{3^{k-2}}\dots$ and $Y = \mathbf{a}Y'$ (defined in Ex. 35). The names y_{i+1} and x'_{i+1} are only used in this figure.

In the following example, we build a text whose ESP tree has a specific **typeM** meta-block on each height that we want to change. Given a **typeM** meta-block μ that emerged from prepending a character to a **type1** meta-block, we can create a new meta-block by prepending another character such that it precedes μ and absorbs μ 's first character (μ then returns to be a **type1** meta-block). We can arrange the **typeM** meta-blocks such that prepending a character to the text changes a **typeM** meta-block on each height:

Example 38. Let $k = \lfloor \log_3(n/\log_3 n) \rfloor$ be a natural number, and $\mathbf{a}, \mathbf{b} \in \Sigma$. Define $Y := X_0 X_1 \dots X_k$ with $X_i := \mathbf{b}^{3^i} \mathbf{a}^{3^k - 3^i}$, for $0 \leq i \leq k-1$. Then $|Y| \leq n$, and $\text{ET}(Y)$ has $\Omega(\lg^2 n)$ fragile nodes.

Proof. Given an integer i with $0 \leq i \leq k-1$, we have $|X_i| = 3^k$ and $|Y| = k3^k \leq n$. Corollary 37 yields $\text{esp}^{(h)}(X_i) = \text{esp}^{(h)}(\mathbf{b}^{3^i}) \text{esp}^{(h)}(\mathbf{a}^{3^k - 3^i})$ for all heights h with $0 \leq h \leq i$, since $3^k - 3^i \geq 3^k - 3^{k-1} = 2 \cdot 3^{k-1}$. Let $a_i := \langle \mathbf{a}^{3^k} \rangle_i[1]$ and $b_i := \langle \mathbf{b}^{3^k} \rangle_i[1]$ be the nodes on height i with $0 \leq i \leq k$ and $\text{string}(a_i) = \mathbf{a}^{3^i}$ or respectively $\text{string}(b_i) = \mathbf{b}^{3^i}$ ($a_0 := \mathbf{a}$, $b_0 := \mathbf{b}$), $\text{esp}(\text{esp}^{(h-1)}(X_i))$ partitions its input $\text{esp}^{(h-1)}(X_i)$ into two meta-blocks: a **type1** meta-block containing all b_i 's, and a subsequent **type1** meta-block containing all a_i 's. All blocks of these two meta-blocks contain three characters, since each meta-block has a length that is equal to a power of three. For the upper heights we get

$$\text{esp}^{(h+i)}(X_i) = \text{esp}^{(h)} \left(\overbrace{\underbrace{\text{esp}^{(i)}(\mathbf{b}^{3^i})}_{=b_i} \text{esp}^{(i)}(\mathbf{a}^{3^k - 3^i})}_{=a_i^{3^k - i - 1}} \right) \text{ for } 0 \leq h+i \leq k-1. \quad (5)$$

Hence, $\text{esp}^{(h+i)}(X_i)$ consists of exactly one **typeM** meta-block, which has the length 3^{k-h-i} , and each block contains three characters. We conclude that the tree $T_i := \text{ET}(X_i)$ is a perfect ternary tree, for $0 \leq i \leq k-1$. Since $|\text{esp}^{(h)}(X_i)| = 3^{k-h}$ for all i, h with $0 \leq i \leq k-1$ and $0 \leq h \leq k$, with Cor. 37 it is easy to see that $\text{esp}^{(h)}(Y) = \text{esp}^{(h)}(X_1 \dots X_{k-1}) = \text{esp}^{(h)}(X_1) \dots \text{esp}^{(h)}(X_{k-1})$ for all $0 \leq h \leq k$. A conclusion is that X_i is the generated substring of the i -th leftmost node $\text{ET}(Y)$ on height k whose name is equal to the name of the root of T_i , for $0 \leq i \leq k-1$.

For the proof, we prepend an \mathbf{a} to Y and call the new string Y' , i.e., $Y' = \mathbf{a}Y$. Our analysis of the difference between $\text{ET}(Y)$ and $\text{ET}(Y')$ focuses on the unique meta-block at height i of T_i : From Eq. (5) with $h = 0$, we observe that there is a single meta-block μ_i at height i of T_i , and this meta-block is a **typeM** meta-block (cf. Fig. 34(right)). Our claim is that prepending \mathbf{a} to Y changes the blocks of the borders of every μ_i ($0 \leq i \leq k-1$): The prepended \mathbf{a} forms a **type2** meta-block with the first character of X_0 by “stealing” the first character from μ_0 , and this character is a $b_0 = \mathbf{b}$. Assume that μ_i ($0 \leq i \leq k-1$) loses its first character (i.e., b_i). By relinquishing this character, μ_i becomes a **type1** meta-block, consisting only of a_i 's. The last two a_i 's contained in μ are grouped into a block a'_{i+1} of length *two*, where $a'_{i+1} := \langle a_i a_i \rangle_1[1]$ is the name of the root node of $\text{ET}(a_i a_i)$. Every newly appearing node a'_{i+1} gets combined with its right-adjacent node b_{i+1} to form a new **type2** meta-block. The used b_{i+1} is stolen from μ_{i+1} , and hence we observe an iterative process of stealing the first character b_{i+1} from μ_{i+1} for each height $i = 0, \dots, k-2$. Figure 35 visualizes this observation on the lowest two heights.

This can be inductively proven for each even integer i with $0 \leq i \leq h-2$. By Eq. (5), we know that

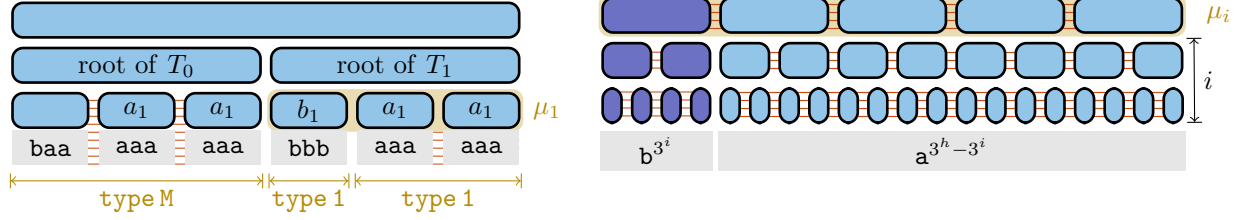


Figure 34: $ET(Y)$ of the example string Y defined in Ex. 38 with $k = 2$ (*left*) and as a schematic illustration (*right*) with the meta-block μ_i on height i (due to space issues the number of nodes/children is incorrect).

$\langle X_i \rangle_i = b_i a_i^{3^{k-i}-1}$ and $\langle X_{i+1} \rangle_i = b_i^3 a_i^{3^{k-i}-3}$. Then

$$\begin{aligned}
 esp(esp(\langle X_i \rangle_i \langle X_{i+1} \rangle_i)) &= esp(esp(b_i a_i^{3^{k-i}-1} b_i^3 a_i^{3^{k-i}-3})) \\
 &\stackrel{(\text{Cor. 37})}{=} esp(esp(b_i a_i a_i) esp(a_i^{3^{k-i}-3} b_{i+1} a_{i+1}^{3^{k-i-1}-1})) \\
 &= esp(esp(b_i a_i a_i) a_{i+1}^{3^{k-i-1}-1} b_{i+1} a_{i+1}^{3^{k-i-1}-1}) \\
 &\stackrel{(\text{Cor. 37})}{=} esp(esp(b_i a_i a_i) a_{i+1}^{3^{k-i-1}-1}) esp(b_{i+1} a_{i+1}^{3^{k-i-1}-1}) \\
 &= esp(esp(b_i a_i a_i) a_{i+1}^{3^{k-i-1}-1}) esp(b_{i+1} a_{i+1} a_{i+1}) esp(a_{i+1}^{3^{k-i-1}-3}),
 \end{aligned}$$

and $esp(a_{i+1}^{3^{k-i-1}-3}) = a_{i+2}^{3^{k-i-2}-1}$. Adding a'_i (set $a'_0 := \mathbf{a}$) to the string $\langle X_i \rangle_i \langle X_{i+1} \rangle_i$ yields

$$\begin{aligned}
 esp(esp(a'_i \langle X_i \rangle_i \langle X_{i+1} \rangle_i)) &= esp(esp(a'_i b_i a_i^{3^{k-i}-1} b_i^3 a_i^{3^{k-i}-3})) \\
 &\stackrel{(\text{Cor. 37})}{=} esp(esp(a'_i b_i) esp(a_i^{3^{k-i}-1} b_{i+1} a_{i+1}^{3^{k-i-1}-1})) \\
 &= esp(esp(a'_i b_i) esp(a_i^{3^{k-i}-3} a_i a_i) b_{i+1} a_{i+1}^{3^{k-i-1}-1}) \\
 &= esp(esp(a'_i b_i) a_{i+1}^{3^{k-i-1}-1} a'_{i+1} b_{i+1} a_{i+1}^{3^{k-i-1}-1}) \\
 &\stackrel{(\text{Cor. 37})}{=} esp(esp(a'_i b_i) a_{i+1}^{3^{k-i-1}-1}) esp(a'_{i+1} b_{i+1}) esp(a_{i+1}^{3^{k-i-1}-3} a_{i+1} a_{i+1}) \\
 &= esp(esp(a'_i b_i) a_{i+1}^{3^{k-i-1}-1}) esp(a'_{i+1} b_{i+1}) a_{i+2}^{3^{k-i-2}-1} a'_{i+2},
 \end{aligned}$$

and a'_{i+2} carries on to the nodes $\langle X_{i+2} \rangle_{i+2} \langle X_{i+3} \rangle_{i+2}$ on height $i+2$ due to Cor. 37.

Overall, the leftmost and rightmost node on height $i+1$ of T_i changes, for $i = 0, \dots, k-1$. In total, $\Omega(\lg^2 k)$ nodes are changed. \square

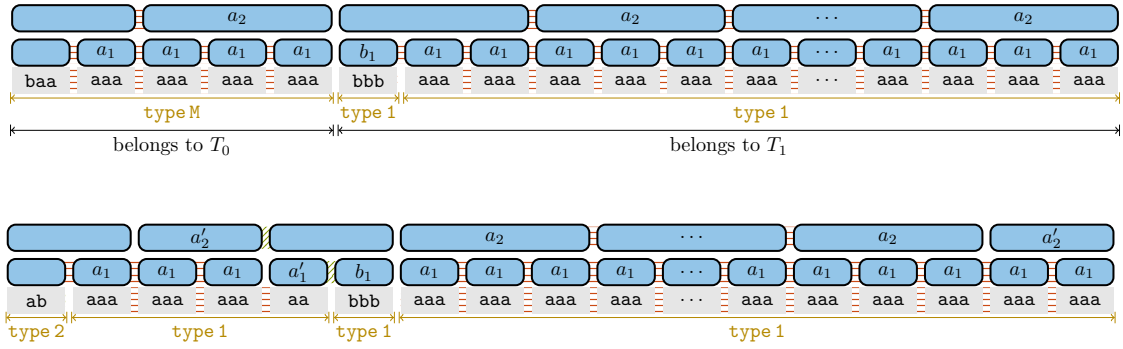


Figure 35: Excerpt of the ESP trees $ET(Y)$ (*top*) and $ET(Y')$ (*bottom*), where $Y = \mathbf{ba}^{3^k-1}\mathbf{b}^3\mathbf{a}^{3^k-3}\mathbf{b}^9\mathbf{a}^{3^k-9}\dots$ and $Y = \mathbf{a}Y'$ (defined in Ex. 38). Due to space issues we contracted T_0 to $ET(\mathbf{ba}^{14})$. Note that right of the rightmost a'_2 (bottom figure, top right node) is the node b_2 (not shown in the figure due to space issues), and both nodes are combined into a **type 2** meta-block.