Deterministic Sparse Suffix Sorting in the Restore Model

JOHANNES FISCHER, Department of Computer Science, TU Dortmund, Germany
TOMOHIRO I, Kyushu Institute of Technology, Japan
DOMINIK KÖPPL, Department of Computer Science, Kyushu University, Japan

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 $O(c\sqrt{\lg n} + m \lg m \lg n \lg^* n)$ time with $O(m)$ 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.

CCS Concepts: • **Information systems** → **Search engine indexing**; • **Theory of computation** → **Sorting and searching**; *Online algorithms*; • **Mathematics of computing** → *Combinatorics on words*;

Additional Key Words and Phrases: Sparse suffix sorting, online algorithms, deterministic algorithms, alphabet reduction, edit-sensitive parsing

ACM Reference format:
Johannes Fischer, Tomohiro I, and Dominik Köppl. 2020. Deterministic Sparse Suffix Sorting in the Restore Model. *ACM Trans. Algorithms* 16, 4, Article 50 (July 2020), 53 pages.
https://doi.org/10.1145/3398681

1 INTRODUCTION

Sorting suffixes of a long text lexicographically is an important first step for many text processing algorithms. The complexity of the problem is quite well understood (see Reference [36]), as for integer alphabets suffix sorting can be done in optimal linear time and in-place [19, 29]. In this article, we consider a variant of this problem: Instead of computing the order of *all* suffixes, we are content with sorting certain specified suffixes. This problem, called ***sparse suffix sorting problem***, is formally defined as follows: Given a text $T[1 \,..\, n]$ of length n and a set $\mathcal{P} \subseteq [1 \,..\, n]$ of m arbitrary positions in T, the sparse suffix sorting 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 $\mathrm{SSA}(T, \mathcal{P})$.

Applications are found in external memory longest common prefix (LCP) array construction algorithms [24] and in the search of maximal exact matches [27, 44], i.e., substrings found in

Parts of this work have already been presented at the 12th Latin American Symposium [12].
This work received funding by JSPS KAKENHI, grant numbers JP19K20213 (TI) and JP18F18120 (DK).
Authors' addresses: J. Fischer, Department of Computer Science, TU Dortmund, Otto-Hahn-Str. 14, Dortmund, 44221, Germany; email: johannes.fischer@cs.tu-dortmund.de; T. I, Kyushu Institute of Technology, Kawazu 680-4, Iizuka, 820-8502, Japan; email: tomohiro@ai.kyutech.ac.jp; D. Köppl, Department of Computer Science, Kyushu University, 744 Moto'oka, Fukuoka, 819-0395, Japan; email: dominik.koeppl@inf.kyushu-u.ac.jp.

1549-6325/2020/07-ART50 $15.00
https://doi.org/10.1145/3398681

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 $\mathrm{SSA}(T, \mathcal{P})$ with the lengths of the LCPs between adjacent suffixes in $\mathrm{SSA}(T, \mathcal{P})$. These lengths are stored in the ***sparse longest common prefix array (SLCP)***, which we denote by $\mathrm{SLCP}(T, \mathcal{P})$. In combination, $\mathrm{SSA}(T, \mathcal{P})$ and $\mathrm{SLCP}(T, \mathcal{P})$ 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 $O(n)$ time if $O(n)$ words of additional working space are available. For $m = o(n)$, however, the working space may be too large, compared to the final space requirement of $\mathrm{SSA}(T, \mathcal{P})$. Although some special choices of $\mathcal{P}$ admit space-optimal $O(m)$-words construction algorithms (e.g., Reference [26], see also the related work listed in Reference [5]), 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 $O(\tau m + n\sqrt{\tau})$ time and $O(m + n/\sqrt{\tau})$ words of working space, where τ is a trade-off parameter with $1 \leq \tau \leq \sqrt{n}$. If randomization is allowed, there is a technique based on Karp-Rabin fingerprints, first proposed by Bille et al. [5] and later improved by I et al. [21]. Gawrychowski and Kociumaka [17] presented an algorithm running with $O(m)$ words of additional space in either $O(n\sqrt{\lg m})$ expected time as a Las Vegas algorithm, or in $O(n)$ expected time as a Monte Carlo algorithm. Most recently, Prezza [35] presented a Monte Carlo algorithm in the restore model [8] that runs with $O(m)$ words of space in $O(n + m \lg^2 n)$ expected time.

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 $O(\mathrm{lcp}(X, Y)/\log_\sigma n)$ time, where $\mathrm{lcp}(X, Y)$ 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 $O(m)$ 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 Contribution

We devise our sparse suffix sorting algorithm in the ***restore model*** [8], where algorithms are allowed to overwrite parts of the input, as long as they can restore the input to its original form at termination. In the case of sparse suffix sorting, we assume that the text T is stored as a rewritable array of size $n \lg \sigma$ bits in RAM. Apart from this space, we are only allowed to use $O(m)$ 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:

Our main algorithmic idea is to insert the suffixes starting at the positions of $\mathcal{P}$ into a self-balancing binary search tree [22]; since each insertion invokes $O(\lg m)$ suffix-to-suffix comparisons, the time complexity is $O(t_S m \lg m)$, where t_S is the cost for a suffix-to-suffix comparison. If all suffix-to-suffix comparisons are conducted naïvely by comparing the characters ($t_S = O(n/\log_\sigma n)$ in the word random-access memory or machine (RAM) model), the resulting worst-case time complexity is $O(nm \lg m/\log_\sigma n)$. To speed this up, our algorithm identifies large identical substrings

at different positions during different suffix-to-suffix comparisons. Instead of performing naïve 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 the *edit sensitive parsing (ESP)* [11]. HSP trees support longest common extension (LCE) queries and 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 Theorem 1.1 below.

We make the following definition that allows us to analyze the running time more accurately: Define $C := \bigcup_{p,p' \in \mathcal{P}, p \neq p'} [p \,..\, p + \mathrm{lcp}(T[p\,..], T[p'\,..])]$ 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 $\Omega(|C|/\log_\sigma n)$ time. With the definition of C, we now can state the main result of this article as follows:

Theorem 1.1. *Given a text T of length n that is loaded into RAM, the sparse suffix array (SSA) and sparse longest common prefix array (SLCP) of T for a set of m arbitrary positions can be computed deterministically in $O(|C|\,(\sqrt{\lg \sigma} + \lg\lg n) + m \lg m \lg n \lg^* n)$ time, using $O(m)$ words of additional working space.*

Excluding the loading cost for the text, the running time can be sublinear (when $|C| = o(n/(\sqrt{\lg \sigma} + \lg\lg n))$ 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 $|C|$, but can in fact be $\Theta(nm)$.

As a result of independent interest, we uncover a flaw in the approximation bound of the algorithm of Cormode and Muthukrishnan [11] computing the *string edit distance with moves (SEDM)* approximatively. There, the authors postulated that they can approximate the SEDM of two strings of length n with a factor of $O(\lg n \lg^* n)$ with edit sensitive parsing (ESP) trees. However, there is a flaw in their analysis of the ESP trees. This flaw leads us to the discovery that the approximation factor is $\Omega(\lg^2 n)$ in worst case.

1.3 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 1.2. *Given a data structure that answers LCE queries in $O(t_{LCE})$ time for $t_{LCE} > 0$, we can compute sparse suffix sorting for m positions in $O(t_{LCE} m \lg m)$ time by inserting suffixes into a balanced binary search tree [22]. Conversely, given an algorithm computing the SSA and the SLCP of a text T of length n for m positions in $O(f(n,m))$ time with $O(m)$ words of space for a function f, we*

Construction		Data Structure		
Time	Working Space	Space	Query Time	Ref.
$O(n\tau)$	$O(\frac{n}{\tau})$	$O(\frac{n}{\tau})$	$O(\tau \lg \min(\tau, \frac{n}{\tau}))$	[42]
$O(n^{2+\epsilon})$	$O(\frac{n}{\tau})$	$O(\frac{n}{\tau})$	$O(\tau)$	[6]
$O\left(n\left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_\sigma n}\right)\right)$	$O\left(\max\left(\frac{n}{\lg n}, \tau^{\lg 3} \lg^* n\right)\right)$	$O(\frac{n}{\tau})$	$O\left(\lg^* n \left(\lg\left(\frac{\ell}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_\sigma n}\right)\right)$	Thm. 1.3
$O\left(n\left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_\sigma n}\right)\right)$	$O\left(\tau^{\lg 3} \lg^* n\right)$	$O(\frac{n}{\tau})$	$O\left(\lg^* n \left(\lg\left(\frac{n}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_\sigma n}\right)\right)$	Cor. 5.3

Fig. 1. Deterministic LCE data structures with trade-off parameters, where ϵ with $\epsilon > 0$ is a constant, and τ with $1 \leq \tau \leq n$ is a trade-off parameter. The length returned by an LCE query is denoted by ℓ. 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.

can construct a data structure in $O(\max(f(n,m), n/m))$ *time with* $O(m)$ *words of space, answering LCE queries on* T *in* $O(n^2/m^2)$ *time.*

Proof. The first claim follows from the time bounds of the binary search tree. For the second claim, we use the data structure of Reference [7, Theorem 1a] that answers LCE queries in $O(t_{\text{LCE}})$ time. The data structure uses the SSA and SLCP values of those suffixes whose starting positions are in a difference cover sampling modulo t_{LCE}. This difference cover consists of $O(n/\sqrt{t_{\text{LCE}}})$ text positions and can be computed in $O(\sqrt{t_{\text{LCE}}})$ time [9]. We obtain the claimed bounds on time and space by setting $t_{\text{LCE}} := n^2/m^2$. □

There has been a great interest in devising deterministic LCE data structures with trade-off parameters (see Figure 1) or in compressed space [20, 32, 43]. One of the currently best data structures with a trade-off parameter is due to Tanimura et al. [42], using $O(n/\tau)$ words of space and answering LCE queries in $O(\tau \lg \min(\tau, n/\tau))$ time, for a trade-off parameter τ with $1 \leq \tau \leq n$. However, this data structure has a preprocessing time of $O(n\tau)$ and is thus not helpful for sparse suffix sorting. We develop a new data structure for LCE with the following properties:

Theorem 1.3. *There is a* deterministic *data structure using* $O(n/\tau)$ *words of space that answers an LCE query* $\ell := lce(i,j)$ *for two text positions* i *and* j *with* $1 \leq i, j \leq n$ *on a text of length* n *in* $O(\lg^* n(\lg(\ell/\tau) + \tau^{\lg 3}/\log_\sigma n))$ *time, where* $1 \leq \tau \leq n$. *We can build the data structure in* $O(n(\lg^* n + (\lg n)/\tau + (\lg \tau)/\log_\sigma n))$ *time with additional* $O(\max(n/\lg n, \tau^{\lg 3} \lg^* n))$ *words during construction.*

The construction time of our data structure has an upper bound of $O(n \lg n)$, and hence it can be constructed faster than the deterministic data structures in Reference [42] when $\tau = \Omega(\lg n)$.

1.4 Outline of This Article

We start with Section 2.2 introducing the ESP, where we conduct a thorough analysis on its characteristics for comparing two substrings by their ESP trees. Within this analysis, we encounter some drawbacks of the ESP in Section 2.4, among others the aforementioned flaw for approximating the SEDM problem. These drawbacks are our motivation for presenting our novel HSP, whose description follows in Section 3. There, it is demonstrated that HSP is immune to the flaw of the ESP. Subsequently, Section 3.3 shows the general techniques for answering LCE queries with the HSP tree. This is followed by Section 4 introducing our algorithm for the sparse suffix sorting problem with an abstract data type *dynamic LCE data structure (dynLCE)* that supports LCE queries and a merging operation. The remainder of that section shows that the HSP tree from Section 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 (Section 5). With the truncated HSP trees, we finally solve the sparse suffix sorting problem in the time and space as claimed in Theorem 1.1.

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 ith 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 $T.\mathrm{rank}_1(j)$ that counts the number of '1's in $T[1..j]$. This operation can be performed in constant time by a data structure [23] 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 $\mathrm{b}(\mathcal{I})$ and $\mathrm{e}(\mathcal{I})$ to denote the beginning and the end of $\mathcal{I}$; i.e., $\mathcal{I} = [\mathrm{b}(\mathcal{I})..\mathrm{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathrm{e}(\mathcal{I}) - \mathrm{b}(\mathcal{I}) + 1$.

2 EDIT-SENSITIVE PARSING

The crucial technique used in this article is the *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], who called their approach *signature encoding*. The signature encoding is derived from a tree coloring approach [18]. 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 affects 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]. The most recent approach on signature encoding is by Gawrychowski et al. [16] presenting a mergeable LCE data structure. A slightly modified version of signature encoding is proposed by Sakamoto et al. [39]. They used the alphabet reduction to build a grammar compressor that is approximating the size of the smallest grammar by a factor of $O(\lg^* n \lg n)$.

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 ESP. Initially used for approximating the SEDM, the ESP technique has been found to be applicable to building self-indexes [40]. 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 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 Figure 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 shrinks the alphabet

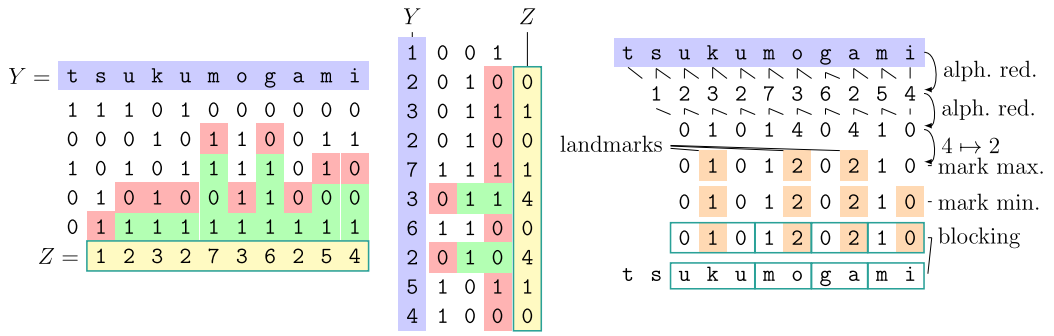


Fig. 2. Alphabet reduction applied on the string Y = tsukumogami. We represent the characters with the five lowest bits of the ASCII encoding. *Left:* A single step of the alphabet reduction. The bit representation of each character $Y[i]$ is shown vertically in the left figure (the most significant bit is on the top). The alphabet reduction matches the least significant bits (shaded green) of two adjacent entries and returns twice the number of matched bits plus the mismatched bit of the right character (shaded red). The resulting integer array Z is the last row. *Middle:* The second step of the alphabet reduction, where the result of the first alphabet reduction stored in Z is put into Y. *Right:* Computation of the blocks. Two steps of the alphabet reduction (seen in the left and in the middle image) yield a sequence consisting only of integers within the domain $\{0, \ldots, 4\}$. Subsequently, all '4's are replaced (in this case by '2', since the neighboring values are '0' and '1' in both cases), and the maxima and certain minima are made into landmarks (shaded orange). Finally, the boxes in the last two rows are the computed blocks.

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 interpret Y as an array of binary strings, i.e., we interpret the character $Y[i]$ with its binary representation $Y[i] \in \{0, 1\}^*$. By doing so, we have $Y[i][\ell] \in \{0, 1\}$ for all integers 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 $\ell := \mathrm{lcp}(Y[i-1], Y[i])$ and write $2\ell + Y[i][\ell + 1]$ to $Z[i]$ (remember that we treat $Y[i]$ as a binary string). 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 $\{0, \ldots, 5\}$. 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 $j = 3, \ldots, 8$ 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

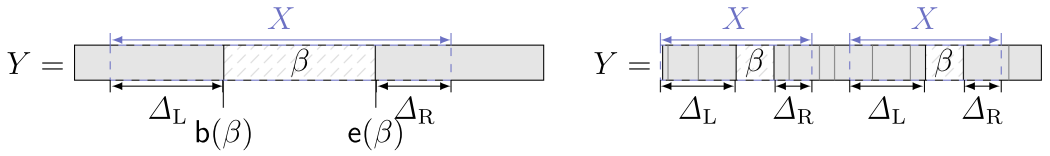


Fig. 3. *Left*: Surrounded block β with local surrounding X contained in a string Y. *Right*: Two occurrences of the local surrounding X of a surrounded block β in the string Y, which is partitioned into blocks (gray rectangles) by the edit-sensitive parsing. Although the occurrences of X can be differently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

the right landmark in case there are two closest landmarks. The last thing to do is to map each block covering $Z[i\,..\,j]$ to $Y[i+r\,..\,j+r]$.

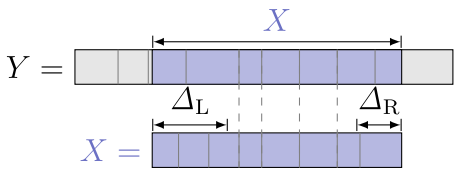
The tie-breaking rule can cause a problem when $Z[1]$ and $Z[3]$ are landmarks, i.e., the leftmost block contains only one character. We circumvent this problem by fusing the blocks of the first and second landmark to a single block. If this block covers four characters, we split it evenly.

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

Lemma 2.1. *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 $O(|Y|\lg^*\sigma)$ 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 2.2 ([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 $Y[\mathrm{b}(\beta)-\Delta_L\,..\,\mathrm{e}(\beta)+\Delta_R]$ the ***local surrounding*** of β if it exists (i.e., $\mathrm{b}(\beta)-\Delta_L \ge 1$ and $\mathrm{e}(\beta)+\Delta_R \le |Y|$). Blocks whose local surroundings exist are also called ***surrounded***. A consequence of Lemma 2.2 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 $Y[\mathrm{b}(\beta)\,..\,\mathrm{e}(\beta)]$ (see Figure 3).

2.2 Meta-blocks

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 ***edit sensitive parsing (ESP)*** technique differ. The main difference is that the ESP technique

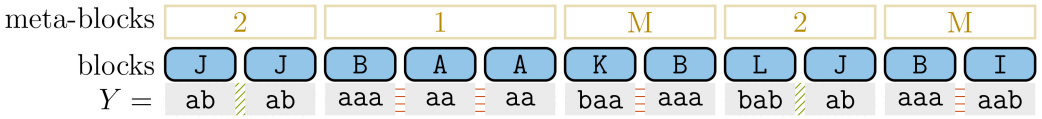


Fig. 4. ESP of the string Y = abababaaaaaaabaaaaababababaaaaab. The string is divided into blocks represented by the gray rectangular boxes at the bottom. Each block gets assigned a new character represented by the capital letters in the rounded boxes. The white/golden ( ) rectangular boxes on the top level represent the meta-blocks that group the blocks. Each such box is labeled with the type of its respective meta-block. The blocks are connected with red horizontal lines ( ) if they belong to a repeating meta-block or by green diagonal lines ( ) if they belong to a Type 2 meta-block.

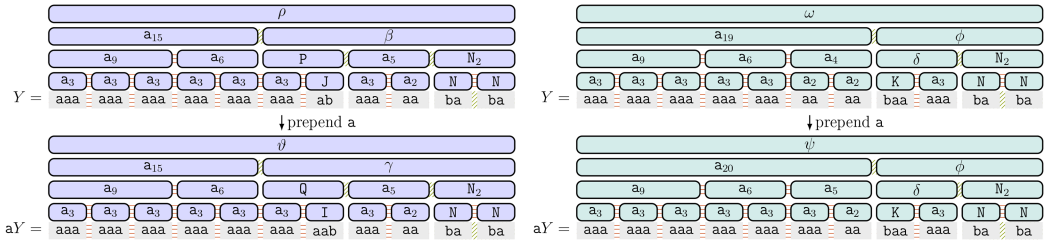


Fig. 5. Impact of the tie-breaking rule (Rule (M)) on emerging Type M nodes of the HSP trees built on $Y = \mathtt{a}^{19}\mathtt{ba}^{5}(\mathtt{ba})^{2}$. A Type M node (like I, J on the left or K on the right) is created by fusing a single symbol with its sibling meta-block. Remember that Rule (M) prescribes to fuse the symbol with its *succeeding* meta-block. To see why this rule is advantageous, the HSP trees on the *left* (respectively, *right*) use the tie-breaking rule Rule (M') (respectively, Rule (M)) favoring the *preceding* (respectively, *succeeding*) 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 $\mathtt{a}_4$ changes to $\mathtt{a}_5$), the change is more dramatic on the *left* side. Prepending the character a to Y (*bottom left*) changes the names of the nodes with names J and P to I and Q, respectively.

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 three-stage process (see also Figure 4 for an example):

(1) Identify runs with smallest period one (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 two (which must be bordered by 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 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 rule:

Rule (M): Fuse a remaining character $Y[i]$ with its succeeding[1] meta-block, or, if $i = |Y|$, with its preceding meta-block.

[1]The original version [11] prefers the preceding meta-block. We comply with Rule (M), as it behaves better. See Figure 5 for an example with the later-introduced HSP trees.

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 $O(|\mu| \lg^* \sigma)$ time by the alphabet reduction (Lemma 2.1). A block β generated by the alphabet reduction is determined by the characters $Y[\max(\mathrm{b}(\beta) - \Delta_{\mathrm{L}}, \mathrm{b}(\mu)) \,..\, \min(\mathrm{e}(\beta) + \Delta_{\mathrm{R}}, \mathrm{e}(\mu))]$ due to Lemma 2.2. Given the number of reduction steps r in Section 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 esp: $\Sigma^* \to (\Sigma^2 \cup \Sigma^3)^*$ 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 $\langle Y \rangle_h := \mathrm{esp}^{(h)}(Y) = \mathrm{esp}(\mathrm{esp}^{(h-1)}(Y))$ 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 (this string is in $\Sigma^2_{h-1} \cup \Sigma^3_{h-1}$). We maintain an injective dictionary $\mathfrak{D} : \Sigma_h \to \Sigma^2_{h-1} \cup \Sigma^3_{h-1}$ to map a block to its symbols. The dictionary entries are of the form $\beta \to xy$ or $\beta \to xyz$, 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 the elements of $\langle Y \rangle_{h-1}[i \,..\, j] \in \Sigma^*_{h-1}$ 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 $\langle Y \rangle_k = \mathrm{esp}(\langle Y \rangle_{k-1})$ is a single block $\rho \in \Sigma_k$. We write $\mathcal{V} := \bigcup_{1 \leq h \leq k} \Sigma_h$ for the set of names in $\langle Y \rangle_1, \langle Y \rangle_2, \ldots, \langle Y \rangle_k$. 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 ρ. This grammar exactly derives Y.

Throughout this article, we comply with the convention to write symbols in typewriter font; in particular, characters (elements of Σ_0) in lowercase and names (elements of Σ_h with $h \geq 1$) in uppercase letters. All examples use the same dictionary such that reappearing names are identical (see Figure 6 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*** $\mathsf{ET}(Y)$ of a string Y is the derivation tree of the CFG defined above. Its root node is the start symbol ρ. 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^2_0 or Σ^3_0. 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 hth iterate of $\mathfrak{D}$ to $\langle Y \rangle_h[i]$

Common Dictionary

Rule	string(·)
I → aab	a^2b
J → ab	ab
K → baa	ba^2
L → bab	bab
N → ba	ba

ESP Dictionary

Rule	string(·)
A → aa	a^2
B → aaa	a^3
C → AA	a^4
D → BB	a^6
E → BBB	a^9
F → DD	a^{12}
G → NN	$(ba)^2$
H → NNN	$(ba)^3$
M → CG	$a^4(ba)^2$
U → ANN	$a^2(ba)^2$
R → JJJ	$(ab)^3$

HSP Dictionary

Rule	string(·)
a_2 → aa	a^2
a_3 → aaa	a^3
P → a_3J	a^4b
Q → a_3I	a^5b

Fig. 6. Names of the ESP (Section 2.2) and HSP (Section 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.

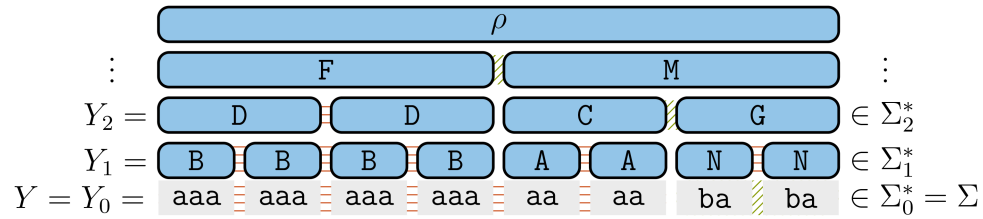


Fig. 7. The ESP tree of the string Y = aaaaaaaaaaaaaaaababa. Like in Figure 4, nodes belonging to the same meta-block are connected by red horizontal ( ) or green diagonal lines ( ) in case they belong to a repeating or a Type 2 meta-block, respectively.

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 Figure 7, string((M)) = aaaababa. A node v on height h is said to be ***built*** on $\langle Y\rangle_{h-1}[b\,..\,e]$ if $\langle Y\rangle_{h-1}[b\,..\,e]$ contains the children of v. Like with blocks, nodes inherit the type of the meta-block on which they are built. An overview of the definitions is given in Figure 8.

2.3.1 Shortcomings of ESP Trees. 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., $\mathfrak{D}^{(h)} : \Sigma_h \to \Sigma_0^*$ 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 sharing the same height. Suppose that Σ is a large alphabet with $\lg^* \sigma = 6$ and that $X :=$ `resliced` occurs in the text that we parse with ESP (see Figure 9). 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 this

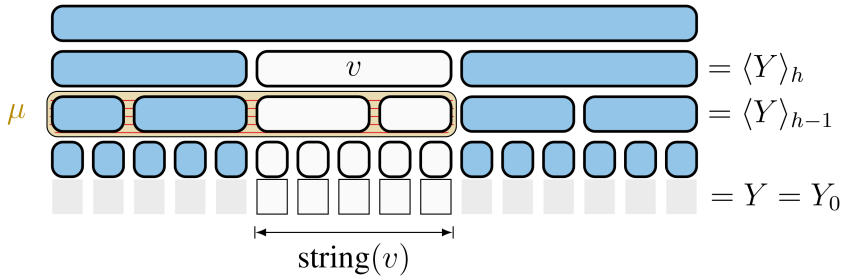


Fig. 8. $\langle Y \rangle_h$ with a highlighted node v. The subtree rooted at v is depicted by the white, rounded boxes. The generated substring string(v) of v is the concatenation of the white rectangular blocks on the lowest level in the picture. The meta-block μ, on which v is built, is the rounded golden ( ) rectangle covering the children of v and all nodes connected by a horizontal hatching ( ) on height $h - 1$.

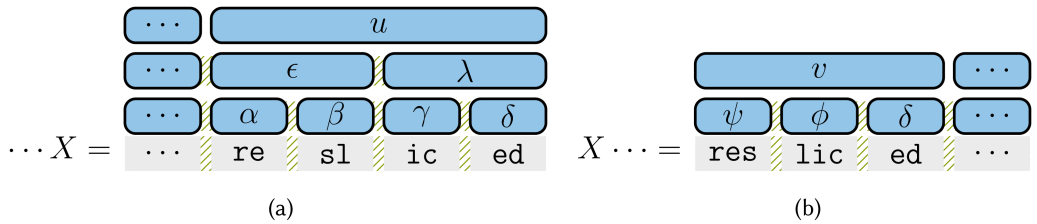


Fig. 9. Excerpts of (a) ET$(\cdots X)$ and (b) ET$(X \cdots)$ with $X :=$ resliced. Under the assumption that $\lg^* \sigma = 8$, the common substring X can be blocked differently in both trees (depending on the characters preceding X in the right figure).

occurrence of X is reduced to the string in superscript of $\cdots \overset{1021012101}{\texttt{resliced}} \cdots$. Then ESP creates the four blocks $\cdots$ |re|sl|ic|ed| $\cdots$, 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 first two blocks of res|lic|ed| $\cdots$ greedily. 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 2.2 from blocks to nodes. Given a substring Y of a string Z, the task is to analyze whether a node $\langle Y \rangle_h[i]$ in ET(Y) 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 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 2.2, whereas we fail to find integers $\Delta_{\mathrm{L},h}$ and $\Delta_{\mathrm{R},h}$ such that a Type 2 node on height h built on $\langle Y \rangle_{h-1}[\Delta_{\mathrm{L},h} \mathinner{..} \Delta_{\mathrm{R},h}]$ cannot be changed by pre-/appending characters to Y; that is because the names inside $\langle Y \rangle_{h-1}$ and $\langle \mathtt{a}Y \rangle_{h-1}$ for $h \geq 2$ can differ at arbitrary positions. This can be seen in the following example: When parsing the string $Y := \mathtt{a}^{9k+4}(\mathtt{ba})^{3k-1}$ with the names defined in

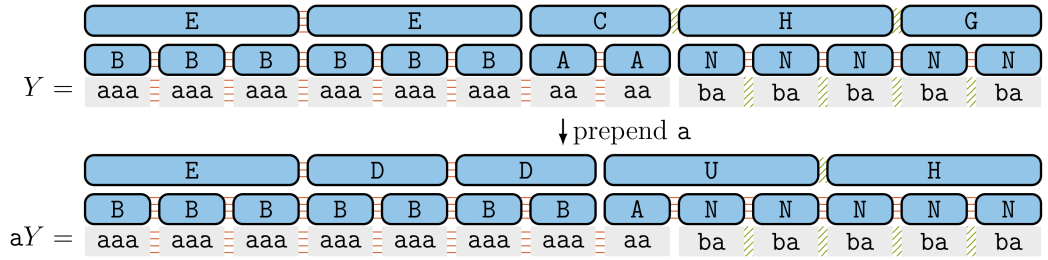


Fig. 10. Excerpt of $\mathsf{ET}(Y)$ and $\mathsf{ET}(\mathsf{a}Y)$ (higher nodes omitted), where $Y = \mathsf{a}^{9k+4}(\mathsf{ba})^{3k-1} = \mathsf{a}^{22}(\mathsf{ba})^5$ for $k = 2$. For all $k \geq 2$, there is a unique node in $\langle Y \rangle_2$ with the name C. This name does not appear in $\mathsf{ET}(\mathsf{a}Y)$.

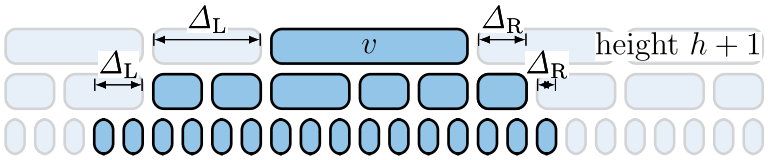


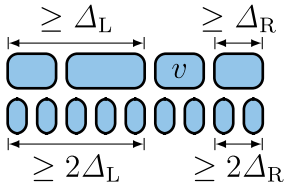
Fig. 11. Local surrounding of a node v at height $h + 1$.

Figure 6, we obtain $\mathsf{esp}(\mathsf{esp}(Y)) = \mathsf{esp}(\mathsf{B}^{3k}\mathsf{AAN}^{3k-1}) = \mathsf{E}^k\mathsf{CH}^{k-1}\mathsf{G}$. Let us focus on the unique occurrence of the name C, which is depicted in Figure 10 for $k = 2$. On the one hand, there is a block in $\langle Y \rangle_1$ with 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 $\mathsf{B}^{3k}\mathsf{AAN}^{3k-1}$. On the other hand, there is a unique node in $\mathsf{ET}(Y)$ with name C on height two. Regardless of the value of k, prepending a to Y changes the name of v: $\mathsf{esp}(\mathsf{esp}(\mathsf{a}Y)) = \mathsf{esp}(\mathsf{B}^{3k+1}\mathsf{AN}^{3k-1}) = \mathsf{E}^{k-1}\mathsf{DDUH}^{k-1}$.

In the following, we introduce the notion of surrounded nodes, since they are helpful to find rules that determine nodes that cannot be changed by pre-/appending characters.

2.3.2 Surrounded Nodes. Analogously to blocks, we classify nodes as surrounded when they are neighbored by sufficiently many nodes: A leaf is called *surrounded* if 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 $\langle Y \rangle_h[\mathsf{b}(\beta) - \Delta_L \,..\, \mathsf{e}(\beta) + \Delta_R]$ and the local surrounding of each node in $\langle Y \rangle_h[\mathsf{b}(\beta) - \Delta_L \,..\, \mathsf{e}(\beta) + \Delta_R]$. 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***. See Figure 11 for an illustration.

Lemma 2.3. *There are at most $\Delta_L + \Delta_R$ many non-surrounded nodes on each height, summing up to $O(\lg^* n \lg n)$ non-surrounded nodes in total.*



Proof. We show the following claim: 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 (respectively, succeeding) nodes have at least $2\Delta_L$ (respectively, $2\Delta_R$) children in total, where at least the Δ_L rightmost nodes (respectively, Δ_R leftmost nodes) are surrounded by the assumption. Hence, v is surrounded. □

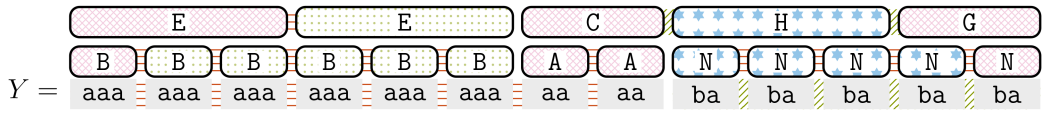


Fig. 12. ET(Y) of Figure 10 with fragile, semi-stable, and stable nodes highlighted. The fragile nodes are cross-hatched ( ), the semi-stable nodes are dotted ( ), and the stable nodes have stars attached ( ). The leftmost nodes of the tree change their names when prepending one b. When prepending a's, we observe that the children of the node with name C change. Assuming that $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ (and hence $|\Sigma| = 2$), only the rightmost node of the meta-block containing nodes with name N is fragile.

The examples of Section 2.3.1 shedding a light on the difference between blocks and nodes reveal that the property for surrounded blocks as shown on the right side of Figure 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 2.2.

2.4 Fragile and Stable Nodes in ESP Trees

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

Nodes that are neither stable nor semi-stable are called ***fragile***. By definition, the children of the (semi-)stable nodes (respectively, fragile nodes) are also (semi-)stable (respectively, fragile). Figure 12 shows an example, where all three types of nodes are highlighted. The rest of this section studies how many fragile nodes exist in $\mathsf{ET}(Y)$.

As a warm-up, we first restrict the ESP tree construction on strings that are square-free. Since a name of the ESP tree is determined by its generating substring, $\mathsf{ET}(Y)$ cannot contain two consecutive occurrences of the same name on any height. We conclude that $\mathsf{ET}(Y)$ has no repeating nodes, i.e., it consists only of Type 2 nodes.

Remembering Section 2.2, the ESP parsing differs from the signature encoding in the auxiliary parsing used for the repeating meta-blocks and the first $O(\lg^* n)$ symbols of a Type 2 meta-block. The signature encoding introduces an intermediate step where it replaces all runs with smallest period one with a new symbol such that no symbol occurs at two adjacent positions in the resulting string. This means that the signature encoding can apply the alphabet reduction on the *entire* string after applying this intermediate step. By doing so, the signature encoding introduces at most $O(\lg^* n)$ fragile nodes on each height [32, Lemma 9]. In the case of a square-free string, the auxiliary parsing is only required for the $O(\lg^* n)$ leftmost symbols on each height of both parsings (signature encoding and ESP): (a) the intermediate step of the signature encoding does not introduce any new symbols, and (b) the ESP creates only a single Type 2 meta-block. Hence, in this case, the maximal numbers of fragile nodes (a) in the signature encoding parse trees and (b) in the ESP tree have the same asymptotic upper bound. For completeness, we prove this statement explicitly, as the techniques will be used later to devise an upper bound in the general case. We start with the following lemma:

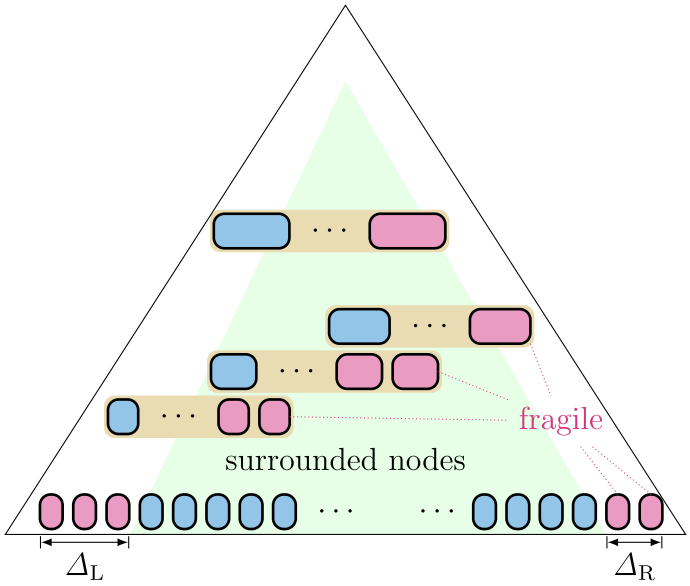


Fig. 13. Division of an ESP tree in surrounded and fragile nodes. The surrounded nodes form an inner cone. Neighboring fragile nodes can appear in the non-surrounded areas (e.g., the lowest leftmost nodes). On each height, the ESP tree can have a constant number of fragile surrounded nodes that do not have fragile nodes in their subtrees.

Lemma 2.4 ([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 2.4, we immediately obtain:

Lemma 2.5. *Given a square-free string Y, a fragile node of $\mathsf{ET}(Y)$ is a non-surrounded node.*

Proof. According to Lemma 2.4, we can bound the number of fragile nodes by the number of those nodes that do not satisfy the conditions in Lemma 2.4. Since $\mathsf{ET}(Y)$ only contains Type 2 nodes, we can inductively show that a fragile node is non-surrounded for all heights of the ESP tree: Surrounded leaves are stable due to Lemma 2.2. 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. □

Combining Lemma 2.5 with Lemma 2.3 yields the following corollary:

Corollary 2.6. *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 Corollary 2.6 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 2.5) does not hold for general strings. To obtain a 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 2.5). Remembering that a node is fragile if it has a fragile child, a fragile Type 2 node can also be surrounded (e.g., one of its children can be a fragile surrounded repeating node). Figure 13 sketches the possible occurrences of fragile surrounded nodes. A first result on a special case is given in the following lemma:

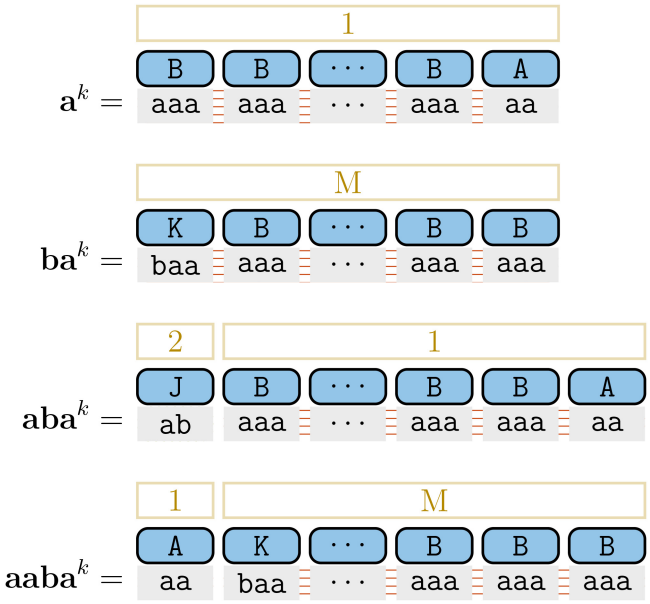
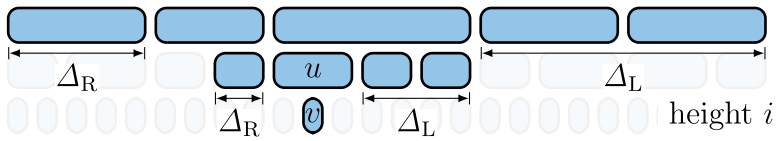


Fig. 14. Prepending the string aab to the text a^k character by character. Each step is given as a row, in which we additionally computed the ESP of the current text. The last row shows an example, where a former Type 1 meta-block changes to Type M, although it is right of a Type 2 meta-block. Here, $k \bmod 3 = 2$.

Lemma 2.7. *A surrounded node v is contained in the local surroundings of $O(\lg^* n \lg n)$ nodes. If all those nodes are of Type 2, then a change of v causes $O(\lg^* n \lg n)$ name changes.*

Proof. We follow Reference [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. □

Second, we narrow down the fragile blocks in repeating meta-blocks. The first block (cf. Figure 14) and the two rightmost blocks (cf. Figure 15) 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 cover one of the leftmost or rightmost symbols, as can be seen by the following lemma:

Lemma 2.8. *A repeating meta-block μ of $esp(Y)$ with $b(\mu) \geq 4$ and $e(\mu) \leq |Y| - 2$ cannot contain a fragile block.*

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

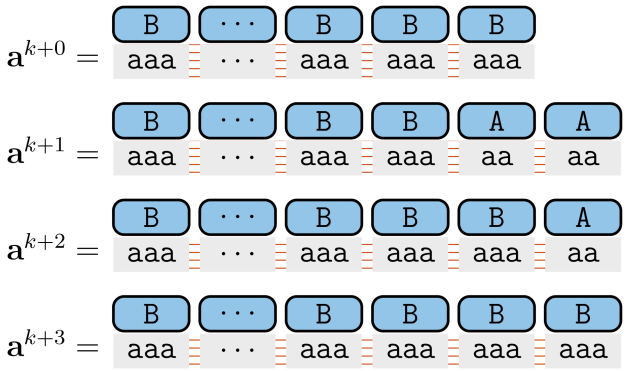


Fig. 15. Greedy blocking of a Type 1 meta-block. The greedy blocking is related to the Euclidean division by three. The remainder $k \mod 3$ is determined by the number of symbols in the last two blocks (here, $k \mod 3 = 0$). In this example, the ESP technique creates a single, repeating meta-block on each input.

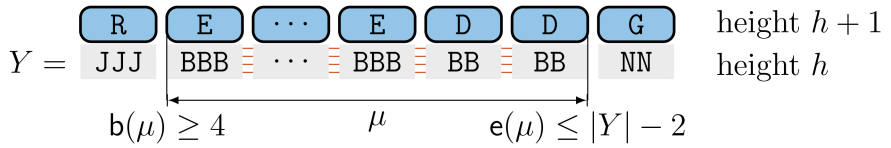


Fig. 16. Setting of Lemma 2.8. According to Lemma 2.8, a meta-block μ in $\mathrm{esp}(Y)$ of a string Y cannot contain a surrounded fragile block if $\mathsf{b}(\mu) \geq 4$ and $\mathsf{e}(\mu) \leq |Y| - 2$.

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 Figure 16 for a sketch. □

Corollary 2.9. *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 2.10. *Changing the symbol in a substring of $\langle Y \rangle_{h-1}$ on which a repeating node on height h is built changes $O(1)$ names on height h.*

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

An easy generalization of Lemma 2.10 is that changing k consecutive nodes on height $h-1$ that are children of repeating nodes on height h changes $O(k)$ names on height h. With Lemma 2.10, the following lemma translates the result of Corollary 2.9 for blocks to nodes:

Lemma 2.11. *The ESP tree $\mathsf{ET}(Y)$ of a string Y of length n has $O(\lg^2 n \lg^* n)$ fragile nodes and $O(h \lg^* n)$ fragile nodes on height h.*

Proof. While computing $\langle Y \rangle_{h+1}$ from $\langle Y \rangle_h$, the ESP technique introduces $O(1)$ fragile surrounded blocks according to Corollary 2.9. Each fragile surrounded block corresponds to a fragile surrounded node.

Similar to the proof of Lemma 2.5, we count all surrounded nodes as fragile whose local surrounding contains a fragile node. Lemma 2.7 shows that each introduced fragile surrounded block makes $O(\lg^* n \lg n)$ nodes fragile. Although we considered only Type 2 nodes in Lemma 2.7, we can generalize this result for all fragile nodes with Lemma 2.10.

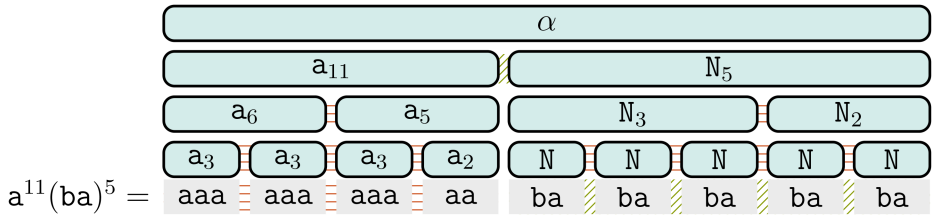


Fig. 17. Hierarchical stable parsing. The repeating meta-blocks are determined by the surnames.

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

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 Reference [11]. Fortunately, we can restore the claimed number of $O(\lg n \lg^* n)$ 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 (HSP)***, augments each name with a ***surname*** and a ***surname-length***, whose definitions follow: Given a name $\mathsf{Z} \in \Sigma_h$, let h' with $0 \le h' \le h$ be the largest integer such that $\mathfrak{D}^{(h')}(\mathsf{Z})$ consists of the same symbol, say, $\mathfrak{D}^{(h')}(\mathsf{Z}) = \mathsf{Y}^\ell$ for a symbol $\mathsf{Y} \in \Sigma_{h-h'}$ and an integer $\ell \ge 1$. Then the surname and surname-length of Z are the symbol Y and the integer ℓ, respectively.[2] For convenience, we define the surname of a character to be the character itself. Then all symbols in $\mathfrak{D}^{(j)}(\mathsf{Z})$ for every j with $1 \le j \le h'$ share the same surname with Z.

Having the surnames of the nodes at hand, we present the HSP. It differs from 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 comparing names, we compare by surnames.[3] As a consequence, 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 Section 2.2 are left untouched; in particular, the alphabet reduction uses the symbols as before. We write $\mathsf{HT}(Y)$ for the resulting parse tree, called ***HSP tree***, when the HSP technique is applied to a string Y. Figure 17 shows $\mathsf{HT}(\mathsf{a}^{11}(\mathsf{ba})^5)$. In the rest of this article (and as shown in Figure 17), we give a repetitive node with surname Z and surname-length ℓ the name 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 Figure 6. 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 3.1. *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 Z and surname-length ℓ, there is a height h such that $\mathfrak{D}^{(h)}(v) = \mathsf{Z}^\ell$. For every height h' with $1 \le h' \le h$, $\mathfrak{D}^{(h')}(v)$ consists of the same symbol, and hence $\mathfrak{D}^{(h')}(v)$ is parsed greedily by HSP. Consequently, the iterated greedy parsing of the string Z^ℓ determines the name of v. □

[2] By definition, the surname of Z is Z itself if $\ell = 1$.
[3] The check is relaxed, since names with different surnames cannot have the same name.

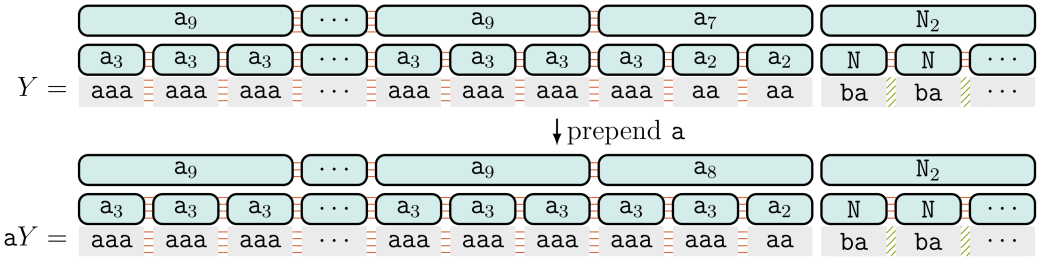


Fig. 18. Excerpt of HT(Y) (*upper part*) and HT(aY) (*lower part*), where $Y = \mathtt{a}^k(\mathtt{ba})^{k'}$ with $k = 18 + 9^i + 7$ for an integer $i \geq 0$ and $k' \geq 2$ (cf. Figure 10). The parsing of Y creates a repeating meta-block consisting of $\mathtt{a}^k$ and a Type 2 meta-block consisting of $(\mathtt{ba})^2$. For $k \geq 2$ it is impossible to modify the latter meta-block by prepending characters, since the parsing always groups adjacent nodes with the same surname into one repeating meta-block.

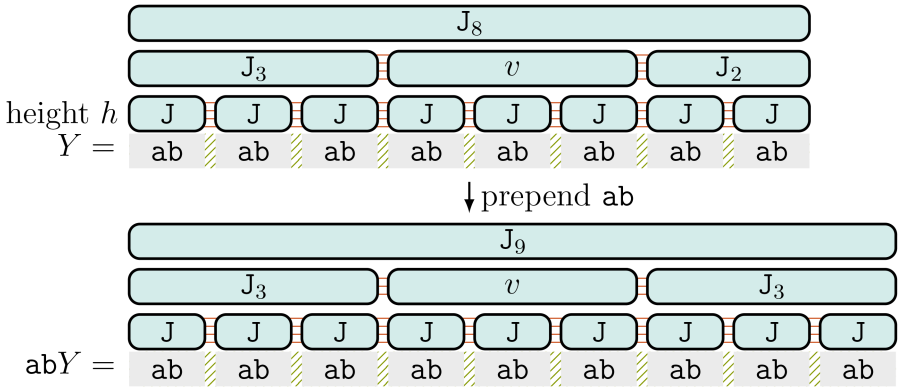


Fig. 19. Comparison of HT(Y) and HT(ab$Y$), where $Y = (\mathtt{ab})^8$. The node v with name $\mathtt{J}_3$ is semi-stable.

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 $X = \text{string}(w) \in \Sigma^*$ (or $X = w$ in case $w \in \Sigma$), but with possibly different surname-lengths k (e.g., $\text{string}(\mathtt{N}_3) = (\mathtt{ba})^3$ and $\text{string}(\mathtt{N}_2) = (\mathtt{ba})^2$ in Figure 17). 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 Figure 18 for an example when prepending a character to the input (observe that $\mathtt{a}_7$ changes to $\mathtt{a}_8$, whose generated substring is still a prefix of $\text{string}(\mathtt{a}_9)$).

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. Section 2.4): Let J be the surname of a semi-stable node $v \in \langle Y \rangle_h$ on height h. Given $\mathcal{J} \in \Sigma_{h'}$ 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 Js to $\langle Y \rangle_{h'}$. In other words, the shift is always a multiple of $\mathfrak{D}^{(h')}(\mathtt{J})$. Figure 19 shows an example of a semi-stable node v.

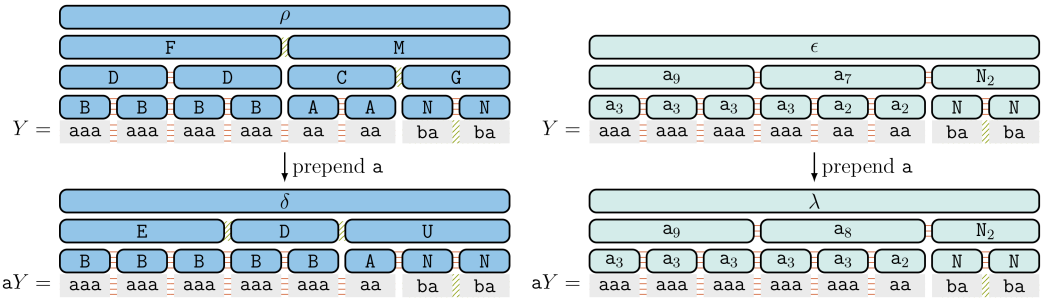


Fig. 20. *Top*: ET(Y) (*left*) and HT(Y) (*right*) of the string Y defined in Figure 7. *Bottom*: ET(aY) (*left*) and HT(aY) (*right*). Unlike the two ESP trees at the *left*, the two HSP trees at the *right* share the same tree topology.

Fact 3: A non-repetitive Type M block can be fragile only if it is non-surrounded. By definition, a repeating meta-block μ contains a non-repetitive block β if and only if μ is Type M. The block β can only be located at the beginning or ending of μ. Remembering Rule (M), β's non-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 $\mathsf{b}(\mu) \leq \Delta_{\mathrm{L}}$. If β is surrounded, it is (semi-)stable due to Lemma 2.8. With Rule (M), we also experience a more stable behavior like in Figure 5.

These facts make the HSP technique more stable than the ESP technique, as can be seen in Figure 20, for instance. In the following, we study the number of fragile surrounded nodes (like in Section 2.4 for the ESP trees) and show the invariant (Claim 3 in Lemma 3.4) 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 2.8 and Facts 1 and 3.

Corollary 3.2. *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 and fragile, $\mathsf{b}(\mu) \leq 2$ according to Lemma 2.8 and Corollary 2.9. Hence, $|\mu| \geq \Delta_{\mathrm{L}} - 2$ (otherwise, β would not be surrounded). By the definition of Δ_{L} in Lemma 2.2, $\Delta_{\mathrm{L}} - 2 \geq 5$ for $n > 4$. Assuming that the repetitive blocks in μ have the surname Z, 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. Due to Fact 1, the surname-length of β is at most as long as the surname-length of γ, 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, we know that $\mathsf{b}(\mu) \leq 2$. Hence, $\mathsf{b}(\gamma) \leq 5 \leq \Delta_{\mathrm{L}}$, yielding that γ is non-surrounded. See Figure 21 for a sketch (with Z = a). □

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,

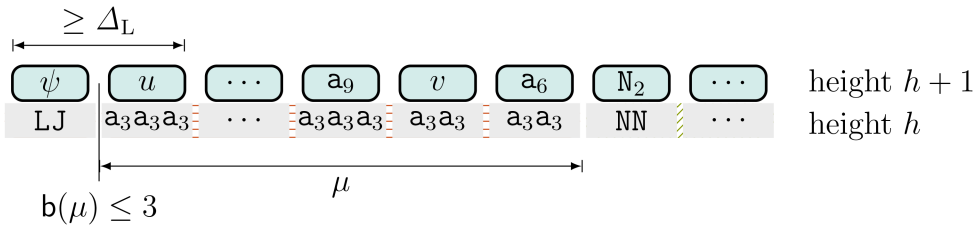


Fig. 21. Setting of Corollary 3.2. According to Lemma 2.8, a meta-block μ can contain a surrounded fragile block if $\mathsf{b}(\mu) \leq 3$ (cf. Figure 16). In the figure, the node v is fragile, since prepending L changes its name. According to Corollary 3.2, there is a non-surrounded node u whose generated substring has the generated substring of v as a prefix.

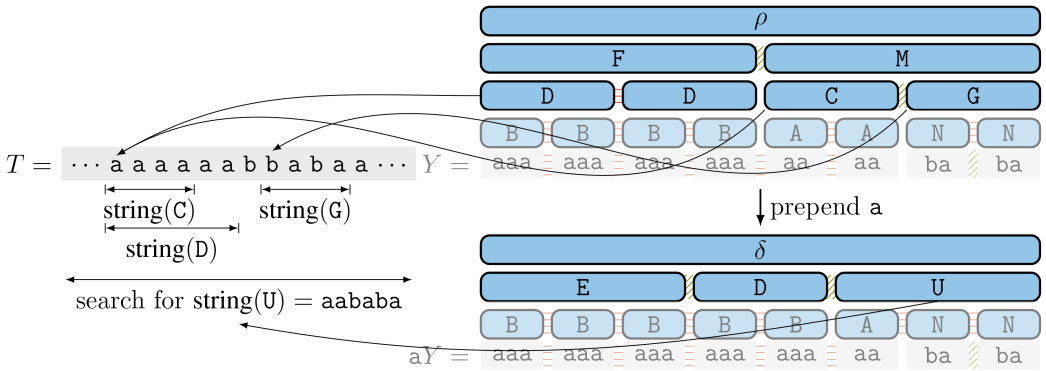


Fig. 22. Problem with dynamic updates of ESP trees stored in text space. Suppose that we truncate ESP trees at a certain height. Truncated nodes are grayed out. Each leaf of the truncated trees is assigned a pointer to its generated substring, which is a substring of the text T (*left*). Suppose that we have built $\mathsf{ET}(Y)$ (*top right*) on a substring Y of T (Y defined as in Figure 20) 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 $\mathsf{ET}(Y)$ to form $\mathsf{ET}(\mathsf{a}Y)$ (*bottom right*), the node G changes to U, for which we need to search its generated substring (assuming that U is not yet stored in the dictionary). The example can be elaborated such that G and U become surrounded nodes (prepend a^{9k} and append b^{9k} for a sufficiently large $k \geq 1$).

which we can do by truncating the tree at a certain height (removing the lower nodes) and changing the pointer of each (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 22 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 3.3. *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 2.4.

Second, according to Corollary 2.9, 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. □

Lemma 3.4. *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 Corollary 2.9), or
- 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 $\mathsf{HT}(Y)$ (*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, $\mathsf{b}(\mu) \leq 3$ must hold (otherwise, we get a contradiction to Lemma 2.8). 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 Corollary 3.2. In particular, the leftmost repetitive node s of μ is not surrounded.

We only consider prepending a character (appending is already considered in Lemma 3.3). 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, string(v) is a substring of 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 a_7 in $\mathsf{HT}(Y)$ of Figure 18, then string(v) = a^7, which is a prefix of string(a_9) = a^9. After prepending the character a, v's name becomes a_8 with string(v) = a^8. Still, string(v) is a prefix of string(a_9).

Due to this behavior, the node v is always assigned to μ, regardless of what character is prepended. 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. Put differently, 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 $1, \ldots, h+1$.

Next, we show that the claim holds for all height $h+2, \ldots, h'$, where $h'+1$ is the height of the lowest common ancestor (LCA) w of s and v. Figure 23 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 $h+2, \ldots, h'$.

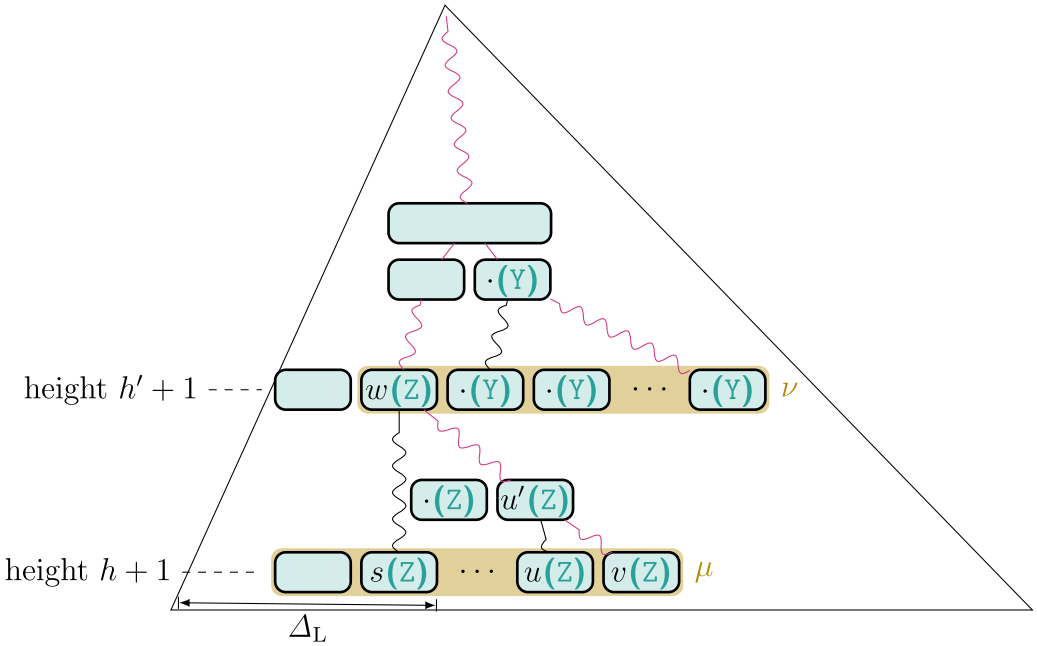


Fig. 23. Sketch of the HSP tree used to show Lemma 3.4. In the sketch, we give the repetitive nodes of the meta-block ν the surname Y. Repetitive nodes are labeled with their surnames, which are put into parentheses.

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 2.7 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). Hence, 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'. □

A direct consequence is that there are $O(1)$ fragile surrounded nodes on each height. We can adapt the result of Lemma 2.11 to HSP trees and obtain the following theorem:

Theorem 3.5. *The HSP tree* $\mathsf{HT}(Y)$ *of a string* Y *of length* n *contains at most* $O(\lg^* n)$ *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 $Y[i \mathinner{..} i + \ell - 1]$ 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 $b \to xyz$ (we set $z = \perp$ if the rule is $b \to xy$), we check whether there already exists a name for xyz. To perform this lookup efficiently, we need also the *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 $O(|Y|)$, supporting lookup and insert in $O(t_{\text{look}})$ (deterministic) time for a $t_{\text{look}} = t_{\text{look}}(n)$ depending on n. For instance, a balanced binary search tree has $t_{\text{look}} = O(\lg n)$.

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

Lemma 3.6. *The HSP tree* $\mathsf{HT}(Y)$ *of a string* Y *of length* n *can be built in* $O(n(\lg^* n + t_{\text{look}}))$ *time. It takes* $O(n)$ *words of space.*

Proof. A name is inserted or looked up in t_{look} time. Due to the alphabet reduction technique (see Lemma 2.1), applying esp on a substring of length ℓ takes $O(\ell \lg^* n)$ time, returning a sequence of blocks of length at most $\ell/2$. □

3.3 LCE Queries with HSP Trees

The idea of devising LCE data structures based on the alphabet reduction is not new. Alstrup et al. [1, Theorem 2] considered building signature encoding parse trees on a set of strings such that the LCP of two strings of this set can be computed efficiently. Nishimoto et al. [32, Lemma 10] enhanced these parse trees with an algorithm computing LCE queries. Similar to these two approaches, we show that HSP trees are also good at answering LCE queries. The common idea of all LCE algorithms 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. Section 2.3.1 and Figure 9), 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 2.5, 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 to enhance this rule for repeating nodes; 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. Our idea (and

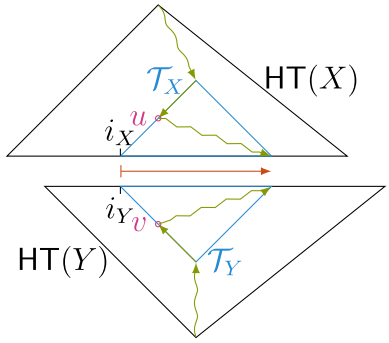


Fig. 24. Conception of the proof of Lemma 3.7. To compute the LCP of $X[i_X\,..]$ and $Y[i_Y\,..]$ (arrow in the center), we walk down the trees $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ (depicted by the upper and the lower triangle, respectively) on the paths towards the leaves containing $X[i_X]$ and $Y[i_Y]$, respectively, by simultaneously visiting two nodes on the same height of both trees. In this figure, each of these paths is depicted by a sequence of green arrows. The nodes u and v are on these paths. Suppose they are on the same height and have the same surname. On visiting both nodes, we know that the LCP is at least $\min(|\mathrm{string}(u)|, |\mathrm{string}(v)|)$ 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 LCP that we want to compute.

here our approach differs from References [1, 32]) is to compare two nodes not by their names but by their surnames and surname-lengths (we use the property described in Fact 2 of Section 3.1). With that idea, we explain how HSP trees can answer LCE queries efficiently. For that, we assume that all HSP trees have a *common* dictionary $\mathfrak{D}$ that additionally stores the length of the string $\mathfrak{D}^{(h)}(\mathsf{Z})$ for each name $\mathsf{Z} \in \Sigma_h$,

Lemma 3.7. *Given* $\mathsf{HT}(X)$ *and* $\mathsf{HT}(Y)$ *built on two strings* X *and* Y *with* $|X| \le |Y| \le n$ *and two text positions* $1 \le i_X \le |X|, 1 \le i_Y \le |Y|$, *we can compute* $lcp(X[i_X\,..], Y[i_Y\,..])$ *in* $O(\lg n \lg^* n)$ *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 $Z = \mathrm{string}(\mathsf{Z})$. In such a case, 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 3.1. 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. Figure 24):

(1) We start with traversing the two paths from the roots of $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ 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 $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$, respectively.
(3) Given that u and v share the same surname $\mathsf{Z} \in \Sigma_h$, we know the lengths of their generated substrings ($\ell_u|\mathfrak{D}^{(h)}(\mathsf{Z})|$ and $\ell_v|\mathfrak{D}^{(h)}(\mathsf{Z})|$) by having their surname-lengths ℓ_u and ℓ_v at hand. Given that i_u and i_v are the starting positions of $\mathrm{string}(u)$ and $\mathrm{string}(v)$, we know that $X[i_X\,..]$ and $Y[i_Y\,..]$ have a common prefix of at least

$$\min\left(\ell_u|\mathfrak{D}^{(h)}(\mathsf{Z})| - (i_u - i_X), \ell_v|\mathfrak{D}^{(h)}(\mathsf{Z})| - (i_v - i_Y)\right). \tag{1}$$

We update the variables λ_X and λ_Y to be the leaves whose generated substrings contain $\langle X\rangle_0[i_u + \ell_u|\mathfrak{D}^{(h)}(\mathsf{Z})|]$ and $\langle Y\rangle_0[i_v + \ell_v|\mathfrak{D}^{(h)}(\mathsf{Z})|]$, respectively.[4] Subsequently, we continue our tree traversals from u and v to the updated destinations λ_X and λ_Y, respectively. Since λ_X and λ_Y are no longer in the respective subtrees of u and v, we climb up the tree to the LCA of u (respectively, v) and λ_X (respectively, 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 naïvely. 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 $\mathsf{HT}(X)$ or $\mathsf{HT}(Y)$, 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 navigational 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 Section 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 $O(\lg n)$ 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 $\mathsf{HT}(X)$ (respectively, $\mathsf{HT}(Y)$) down to the roots of the minimal subtree $\mathcal{T}_X$ of $\mathsf{HT}(X)$ (respectively, $\mathcal{T}_Y$ of $\mathsf{HT}(Y)$) covering $X[i_X \mathinner{..} i_X + \ell]$ (respectively, $Y[i_Y \mathinner{..} i_Y + \ell]$).[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 $\mathsf{HT}(Z)$ built with the same dictionary $\mathfrak{D}$, where $Z := X[i_X \mathinner{..} i_X + \ell - 1] = Y[i_Y \mathinner{..} i_Y + \ell - 1]$. On the one hand, $\mathsf{HT}(Z)$ has $O(\lg^* n)$ fragile nodes on each height according to Theorem 3.5. On the other hand, each (semi-)stable node in $\mathsf{HT}(Z)$ is found in both $\mathcal{T}_X$ and $\mathcal{T}_Y$ with the same name and surname. Consequently, when traversing $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ within their respective subtrees $\mathcal{T}_X$ and $\mathcal{T}_Y$, we only visit $O(\lg^* n)$ 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 $O(\lg|Y|)$ time, and (b) we compare the children of at most $O(\lg^* n)$ nodes per height. Since both trees have a height of $O(\lg|Y|)$, we obtain our claimed running time. □

The following corollary is a small refinement of Lemma 3.7, which already shows the result of Theorem 1.3 for $\tau = 1$:

Corollary 3.8. *We can endow an HSP tree of a string of length n in $O(n)$ time with an $O(n)$ words data structure that has the following properties: Given two HSP trees $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ built on two strings X and Y with $|X| \le |Y| \le n$, we can compute $\ell := lcp(X[i_X \mathinner{..}], Y[i_Y \mathinner{..}])$ in $O(\lg \ell \lg^* n)$ time if both trees are endowed with this data structure, where i_X and i_Y are two text positions with $1 \le i_X \le |X|$ and $1 \le i_Y \le |Y|$.*

Proof. Our idea is to endow 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

[4] Instead of selecting the leaves whose generated substrings start at the end of the common prefix calculated in Equation (1), we bookkeep the difference between $\ell_u|\mathfrak{D}^{(h)}(\mathsf{Z})| - (i_u - i_X)$ and $\ell_v|\mathfrak{D}^{(h)}(\mathsf{Z})| - (i_v - i_Y)$.

[5] We assume that $i_X + \ell \le |X|$ and $i_Y + \ell \le |Y|$ such that $\mathcal{T}_X$ and $\mathcal{T}_Y$ cover the mismatching pair of characters $X[i_X + \ell] \ne Y[i_Y + \ell]$. Otherwise, ($i_X + \ell - 1 = |X|$ or $i_Y + \ell - 1 = |Y|$), let $\mathcal{T}_X$ and $\mathcal{T}_Y$ cover $X[i_X \mathinner{..} i_X + \ell - 1]$ and $Y[i_Y \mathinner{..} i_Y + \ell - 1]$, respectively.

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 in $O(n)$ time and takes $O(n)$ words of space. □

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 only need 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 $\mathcal{P}$ in sorted order. Due to the online setting, we represent the order of $Suf(\mathcal{P})$ 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 $Suf(\mathcal{P})$; the lexicographic order is used as the sorting criterion.

The technique of Irving and Love [22] augments an AVL tree on a set of strings $\mathcal{S}$ with the lengths of LCPs so $\ell_Y := \max\{\mathrm{lcp}(X, Y) \mid X \in \mathcal{S}\}$ can be computed in $O(\ell_Y / \log_\sigma n + \lg |\mathcal{S}|)$ time for a string Y, where the division by $\log_\sigma n$ is due to the word-packing technique. 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 $\mathsf{SAVL}(\mathcal{S})$.

Remembering Section 1.2, our goal is to build $\mathsf{SAVL}(Suf(\mathcal{P}))$ efficiently. However, inserting m suffixes naïvely takes $\Omega(|C|\, m/\log_\sigma n + m \lg m)$ 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 incrementally updates $\mathsf{SAVL}(Suf(\mathcal{P}))$ on the input of every new text position, involving LCE computations between the new suffix and suffixes already stored in $\mathsf{SAVL}(Suf(\mathcal{P}))$. 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 structure (dynLCE)***; it supports the following operations:

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

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 $\mathsf{dynLCE}(\mathcal{I})$ 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 $t_{\mathrm{Q}}(|X|+|Y|)$ and $t_{\mathrm{M}}(|X|+|Y|)$ 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 dynLCEs for a length ℓ is at least as fast as the word-packed character comparison if and only if $\ell = \Omega(t_{\mathrm{Q}}(\ell)\log_\sigma n)$.[6] Hence, we obtain the following property:

> Property 2: A dynLCE on a text smaller than $g := \Theta(t_{\mathrm{Q}}(g)\log_\sigma n)$ is always slower than the word-packed character comparison.

In the following, we build dynLCEs 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 $O(m)$. The tree $\mathcal{L}$ stores the starting and the ending positions of each LCE intervaland uses the starting positions as keys to answer the queries

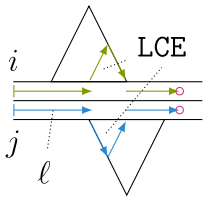
(1) whether a position is covered by a dynLCE, and
(2) where the next text position starts that is covered by a dynLCE,

in $O(\lg m)$ time. Additionally, each LCE interval is assigned to one dynLCE data structure (a dynLCE can be assigned to multiple LCE intervals) such that $\mathcal{L}$ also returns a dynLCE that covers the position returned by query (2) above. This is done by augmenting an LCE interval $\mathcal{I}$ with a pointer to its dynLCE data structure M, and with an integer i such that $T[M.\mathrm{ival} \cap [i\,..\,i + |\mathcal{I}| - 1]] = T[\mathcal{I}]$ (since M could be built on a text interval $M.\mathrm{ival} \neq \mathcal{I}$ that contains an occurrence of $T[\mathcal{I}]$).

Given a new position $\hat{p} \notin \mathcal{P}$ with $1 \le \hat{p} \le |T|$, updating $\mathsf{SAVL}(Suf(\mathcal{P}))$ to $\mathsf{SAVL}(Suf(\mathcal{P} \cup \{\hat{p}\}))$ involves two parts: first *locating* the insertion node for $\hat{p}$ in $\mathsf{SAVL}(Suf(\mathcal{P}))$ and then *updating* the set of LCE intervals.

Locating. Finding the insertion point of $\hat{p}$ involves an LCE computation for each node encountered in $\mathsf{SAVL}(Suf(\mathcal{P}))$. Suppose that the task is to compare the suffixes $T[i\,..]$ and $T[j\,..]$ for two text positions i and j with $1 \le i, j \le |T|$. We perform the following steps to compute $\mathrm{lce}(i, j)$:

(1) Check whether the positions i and j are contained in an LCE intervalin $O(\lg m)$ time with the search tree $\mathcal{L}$.
   - If both positions are covered by LCE intervals, then query the respective dynLCEs 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 naïvely 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 $\mathsf{SAVL}(Suf(\mathcal{P}))$, we obtain

$$\bar{p} := \mathrm{mlcparg}(\hat{p}) \text{ and } \ell := \mathrm{mlcp}(\hat{p})$$

[6] We assume that $t_{\mathrm{Q}}(\ell)$ is sub-linear in ℓ.

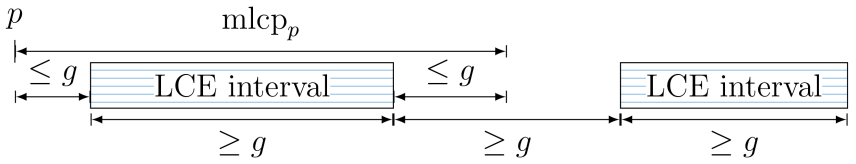


Fig. 25. Sketch of two LCE intervals with Properties 3 to 5.

as a byproduct, where

$$\text{mlcparg}(p) := \operatorname*{argmax}_{p' \in \mathcal{P}, p \neq p'} \text{lcp}(T[p\,..], T[p'\,..])$$

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 $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ 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 Figure 25):

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 the following, we first process the LCE intervals to satisfy Property 4^+ and then subsequently to satisfy Property 5. When Property 4^+ and Property 5 are satisfied, then Property 4 is also satisfied. We can satisfy Property 4^+ with the following steps: 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 Figure 26):

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 $T[\mathcal{K}]$). Then apply Rule 1.
Rule 3: Otherwise (there is no LCE interval $\mathcal{K}$ with $\mathcal{J} \cap \mathcal{K} \neq \emptyset$), create $\mathsf{dynLCE}(\mathcal{J})$, and make $\mathcal{I}$ and $\mathcal{J}$ to LCE intervals referring to $\mathsf{dynLCE}(\mathcal{J})$.

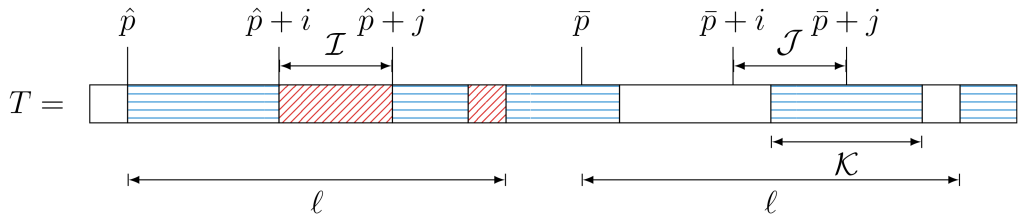


Fig. 26. Application of Rules 1 to 4 for preserving the properties. The interval $\mathcal{I} := [\hat{p}+i\,..\,\hat{p}+j]$ is not yet covered by an LCE interval, but is contained in $[\hat{p}\,..\,\hat{p}+\ell-1]$—a conflict with Property 4. The conflict is resolved based on the LCE intervals covering the positions of $\mathcal{J} := [\bar{p}+i\,..\,\bar{p}+j]$. The intervals with the blue horizontal lines (▤) are the LCE intervals, and the intervals with the diagonal red 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.

We repeat the application of the rules above until we finally satisfy Property 4^+. However, Rule 1 or Rule 3 can create LCE intervals shorter than g, violating Property 3. By construction, such short LCE intervals are adjacent to other LCE intervals (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 after the above process:

> Rule 4: Merge a newly created or merged LCE interval violating Property 3 with its nearest LCE interval (ties can be broken arbitrarily). Recurse until no merge occurs.

This finally restores Property 4 (since Property 4^+ and Property 5 hold). As a result, we have introduced at most two[7] 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 4.1. *Given a text T of length n and a set of m arbitrary positions $\mathcal{P}$ in T, the suffix AVL tree* $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ *with the suffixes of T starting at the positions $\mathcal{P}$ can be computed deterministically in $\mathcal{O}(t_{\mathrm{C}}(|C|) + t_{\mathrm{Q}}(|C|)m \lg m + t_{\mathrm{M}}(|C|)m)$ time.*

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

- We build dynLCEs on substrings covering at most $|C|$ characters of the text, taking at most $t_{\mathrm{C}}(|C|)$ time for constructing all dynLCEs. During the construction of the dynLCEs, we spend $\mathcal{O}(|C|/\log_\sigma n) = \mathcal{O}(t_{\mathrm{C}}(|C|))$ time on character comparisons 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 $2\,t_{\mathrm{M}}(|C|)m$ time for the merging.
- The algorithm performs $\mathcal{O}(m \lg m)$ LCE queries. LCE queries involve either (a) character comparisons or (b) querying a dynLCE.

(a) On the one hand, the overall time for the character comparisons is bounded by $\mathcal{O}(t_{\mathrm{C}}(|C|) + t_{\mathrm{Q}}(|C|)m \lg m)$:

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

— By Property 4, all substrings $T[p \,..\, p + \text{mlcp}(p) - 1]$ are covered by an LCE interval, except at most at $2g$ positions.[8] This means that all substrings that are not covered by an LCE interval, but have been subject to a character comparison, are shorter than $2g$. For a character comparison with one of those substrings, we spend at most $O(gm \lg m / \log_\sigma n) = O(t_Q(g) m \lg m) = O(t_Q(|C|) m \lg m)$ time. In the case that $g > |C|$, we do not create any LCE interval, and we spend $O(gm \lg m / \log_\sigma n) = O(t_Q(|C|) m \lg m)$ overall time due to Property 2.

— 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. Hence, the time needed for character comparisons is $O(|\mathcal{I}| / \log_\sigma n) = O(t_C(|\mathcal{I}|))$. This sums up to $O(t_C(|C|))$ total time spent on character comparisons of substrings longer than g characters.

(b) However, querying the dynLCEs takes at most $O(t_Q(|C|) m \lg m)$ overall time. Suppose that we look up $d < \delta$ LCE intervals for an LCE query, where $\delta < 2m$ is the total number of LCE intervals. Since we look up an LCE interval in $O(\lg m)$ time with $\mathcal{L}$, we spend $O(d \lg m)$ 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 lookup of an LCE interval at most $2m$ times in total. □

The last step is to compute $\text{SSA} := \text{SSA}(T, \mathcal{P})$ and $\text{SLCP} := \text{SLCP}(T, \mathcal{P})$ from $\text{SAVL}(Suf(\mathcal{P}))$ by traversing $\text{SAVL}(Suf(\mathcal{P}))$ and performing LCE queries on the already computed dynLCEs: The $\text{SAVL}(Suf(\mathcal{P}))$ is a binary search tree storing all elements of $Suf(\mathcal{P})$ in lexicographically sorted order. Consequently, we can compute SSA with an in-order traversal of $\text{SAVL}(Suf(\mathcal{P}))$. Afterwards, we compute $\text{SLCP}[i] = \text{lce}(\text{SSA}[i], \text{SSA}[i-1])$. If the text positions $[\text{SSA}[i] \,..\, \text{SSA}[i] + \text{SLCP}[i] - 1]]$ and $[\text{SSA}[i-1] \,..\, \text{SSA}[i-1] + \text{SLCP}[i] - 1]]$ are not covered by an LCE interval, then $\text{SLCP}[i] = O(g)$ due to Property 3, and we spend at most $O(g / \log_\sigma n)$ time on computing $\text{SLCP}[i]$ by character comparisons. Otherwise, we spend $O(g / \log_\sigma n + t_Q(\text{SLCP}[i])) = O(t_Q(\text{SLCP}[i]))$ time by querying a *single* dynLCE due to Property 4. Querying whether both text intervals are covered by a dynLCE costs $O(\lg m)$ time with $\mathcal{L}$. In total, we can compute $\text{SLCP}[i]$ for each integer i with $2 \le i \le m$ in $O(t_Q(|C|) m \lg m)$ time, since $O(g / \log_\sigma n) = O(t_Q(g))$ due to Property 2. The following corollary of Lemma 4.1 summarizes the achievements of this section:

Corollary 4.2. *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 $O(t_C(|C|) + t_Q(|C|) m \lg m + t_M(|C|) m)$ time. We need $O(m)$ words of space and the space to store instances of dynLCE on $|C|$ 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 Section 4.1 depends on the merging operation of dynLCE, we now introduce the merging of HSP trees. A naïve 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. Using the properties studied in Section 2.4, we show such an approach in the following lemma:

[8] If $\text{mlcp}(p) < 2g$, there is no need to cover $T[p \,..\, p + \text{mlcp}(p) - 1]$ by an LCE interval due to Property 4. Otherwise, we do not need to cover its first and last g positions.

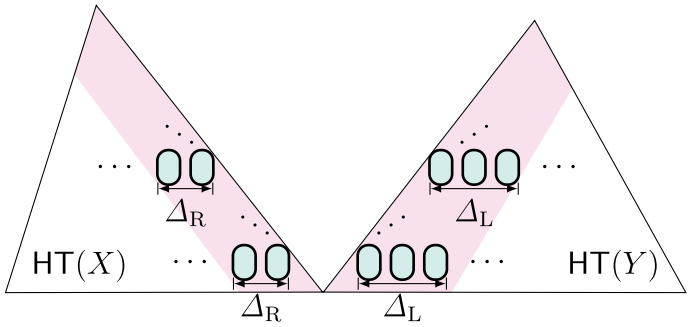


Fig. 27. Merging HT(X) and HT(Y). Given that both trees contain only Type 2 nodes, it suffices to apply the parsing on the Δ_R rightmost nodes and Δ_L leftmost nodes of HT(X) and HT(Y), respectively, to obtain HT(XY).

Lemma 4.3. *Merging* HT(X) *and* HT(Y) *of two strings* $X, Y \in \Sigma^*$ *into* HT(XY) *takes* $O(t_{\text{look}}(\Delta_R \lg |X| + \Delta_L \lg |Y|))$ *time, where* t_{look} *is the lookup and insertion time for the reverse dictionary defined in Section 3.2.*

Proof. First assume that HT(X) and HT(Y) only contain Type 2 nodes. In this case, we examine the rightmost nodes of HT(X) and the leftmost nodes of 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 $\langle Y\rangle_h$ (see Figure 27). By doing so, we reparse all nodes of HT(X) (respectively, HT(Y)) whose local surrounding on the right (respectively, left) side does not exist. Nodes of HT(X) (respectively, HT(Y)) that have a local surrounding on the right (respectively, left) side are not changed by the parsing. In total, we spend $O(t_{\text{look}}(\Delta_R \lg |X| + \Delta_L \lg |Y|))$ time on merging two trees consisting of Type 2 nodes.

Next, we allow repeating nodes. Lemma 3.3 shows that there are no fragile surrounded nodes in HT(X) that need to be fixed. The remaining problem is to find and recompute the surrounded nodes in HT(Y) whose names change on merging both trees. The lowest of these nodes belong to a repeating meta-block due to Lemma 2.4 and Corollary 2.9. 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 Corollary 2.9, it is sufficient to reparse the two rightmost nodes of μ. This is done as follows (cf. Figure 28):

(1) Take the leftmost repetitive node s of μ (which exists due to Corollary 3.2 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 LCA 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 $O(t_{\text{look}} \lg |\mu|) = O(t_{\text{look}} \lg(n/2^h))$ time, reparsing the surrounded ancestor nodes of the two rightmost nodes of μ takes $O(t_{\text{look}} \lg(n/2^h))$ time. Given that the highest nodes of this reparsing are on a height $h' > h$, Lemma 3.4 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 Figure 23). Given that there are $\mu_1, \ldots, \mu_k$ such

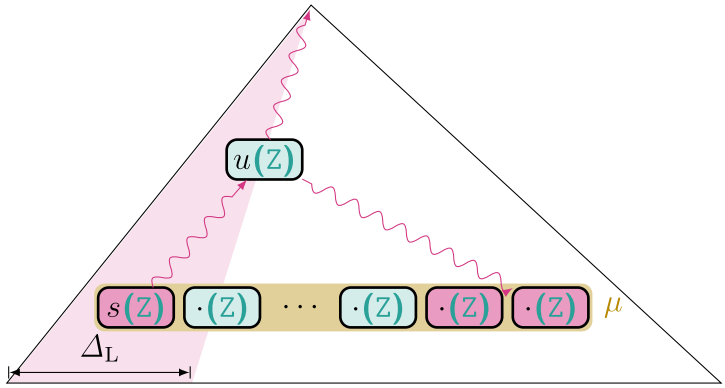


Fig. 28. Reparsing of fragile surrounded nodes during a merging operation. The lowest fragile surrounded nodes that need to be reparsed belong to a meta-block μ whose leftmost nodes are non-surrounded. Given that the repetitive nodes of μ have the surname Z, we can access all fragile nodes of μ by climbing up from the leftmost repetitive node s of μ to the highest node u with surname Z and subsequently descending down to the rightmost repetitive nodes of μ in $O(\lg |\mu|)$ time.

meta-blocks (for which we apply Steps 1 to 6), we have $O(t_{\text{look}} \sum_{i=1}^{k} \lg |\mu_i|) = O(t_{\text{look}} \lg n)$ due to $\sum_{i=1}^{k} \lg |\mu_i| \le \lg n$. Hence, we spend $O((\Delta_{\text{L}} + \Delta_{\text{R}}) t_{\text{look}} \lg |Y|)$ time overall. □

The following theorem combines the results of Corollary 4.2 and Lemma 4.3.

Theorem 4.4. *Given a text T of length n and a set of m text positions $\mathcal{P}$, $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$ can be computed in $O(|C|\,(\lg^* n + t_{\text{look}}) + m \lg m \lg n \lg^* n)$ time, using $O(n + m)$ words of space.*

Proof. We have

- $t_{\text{C}}(|C|) = O(|C|\,(\lg^* n + t_{\text{look}}))$ due to Lemma 3.6,
- $t_{\text{Q}}(|C|) = O(\lg^* n \lg n)$ due to Lemma 3.7, and
- $t_{\text{M}}(|C|) = O(t_{\text{look}} \lg n \lg^* n)$ due to Lemma 4.3.

Actually, the time cost for merging is already upper bounded by the cost for the tree creation. To see this, let $\delta \le 2m$ be the number of LCE intervals. Since each LCE interval covers at least g characters, δ is at most $|C|/g$, and we obtain $\delta\, t_{\text{M}}(|C|) = O(|C|\, t_{\text{M}}(|C|)/g) = O(|C|\, t_{\text{look}})$ overall time for merging, where $g = \Theta(t_{\text{Q}}(|C|) \lg n / \lg \sigma) = \Theta(\lg^* n \lg^2 n / \lg \sigma)$. Plugging the times $t_{\text{C}}(|C|)$, $t_{\text{Q}}(|C|)$, and the refined analysis of the merging time cost in Corollary 4.2 yields the claimed time bounds. □

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 could use the text space for storing the HSP trees while using only $O(m)$ additional words for storing $\mathsf{SAVL}(Suf(\mathcal{P}))$ and the search tree $\mathcal{L}$ of the LCE intervals. However, an HSP tree built on a string of length n takes $O(n \lg n)$ 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 $\mathsf{tHT}_{\eta}(Y)$ stores just the upper part, while its new leaves refer to (possibly long)

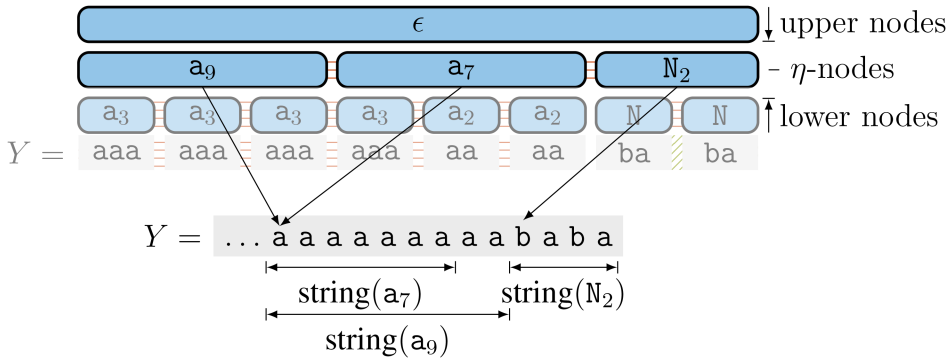


Fig. 29. The η-truncated HSP tree $\mathsf{tHT}_\eta(Y)$ of the substring Y defined in Figure 7 with $\eta = 2$. Like in Figure 22, the lower nodes are grayed out. An η-node is a leaf in $\mathsf{tHT}_\eta(Y)$ and has a generated substring with a length between four and nine.

substrings of Y. The resulting tree is called the ***η-truncated HSP tree*** (tHT_η), whose definition follows:

5.1 Truncated HSP Trees

We define a height η and delete all nodes at heights 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 29 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 $O(\ell \lg \sigma)$ time for a lookup or insertion of a key of length ℓ. Instead, we want a dictionary data structure storing $O(|Y|)$ elements in $O(|Y|)$ words of space,[9] supporting lookup and insert in $O(t_{\text{look}} + \ell/\log_\sigma n)$ 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 $t_{\text{look}} = O(\lg n)$.

Lemma 5.1. *We can build an η-truncated HSP tree $\mathsf{tHT}_\eta(Y)$ of a string Y of length n in $O(n(\lg^* n + \eta/\log_\sigma n + t_{\text{look}}/2^\eta))$ time using $O(3^\eta \lg^* n)$ words of working space. The tree takes $O(n/2^\eta)$ words of space.*

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 $\mathsf{tHT}_\eta(Y)$ 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 2.4. Thus, 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,

[9]The data structure is not necessarily stored in consecutive space like an array.

we create the name of a lower node temporarily by setting the name of a lower node to its generated substring. A drawback is 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 ith 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 $\mathsf{tHT}_\eta(Y)$ 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, cf. Figure 15). 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 construct v after constructing X_v. To construct 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 $O(3^\eta(\Delta_L + \Delta_R))$ 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 $O(\eta)$ words, since such a meta-block can exist on every height below η.

Time. The time bound $O(n \lg^* n)$ for the repeated application of the alphabet reduction is the same as in Lemma 3.6. 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 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 $O(n/\log_\sigma n)$ time, independent of the number of nodes on height h. Summing over all heights, these comparisons take $O(n\eta/\log_\sigma n)$ time in total. By the same argument, maintaining the names of all η-nodes takes $O(n/\log_\sigma n + t_{\text{look}} n/2^\eta)$ time.

A name is looked up in $O(t_{\text{look}})$ 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 $O(t_{\text{look}} n/2^\eta)$ 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 simplly 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. □

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

Lemma 5.2. *Let X and Y be two strings, each of length at most n. Given that $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$ are built with the* same *dictionary, and given two text positions i_X and i_Y with $1 \le i_X \le |X|$ and $1 \le i_Y \le |Y|$, we can compute $lcp(X[i_X\,..], Y[i_Y\,..])$ in $O(\lg^* n(\lg(n/2^\eta) + 3^\eta/\log_\sigma n))$ time using $O(\lg(n/2^\eta))$ words of working space.*

Proof. Lemma 3.7 gives the time bounds for computing the LCP 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 naïve comparison of their generated substrings in $O(3^\eta/\log_\sigma n)$ 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 $O(\lg^* n(\lg(\ell/2^\eta))$ time for traversing those. □

Applying the idea of Corollary 3.8 to Lemma 5.2 gives the following corollary:

Corollary 5.3. *Let X and Y be two strings with $|X|, |Y| \le n$. Given that $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$ are built with the* same *dictionary, we can augment both trees with data structures such that given two text positions $1 \le i_X \le |X|, 1 \le i_Y \le |Y|$, we can compute $\ell := lcp(X[i_X\,..], Y[i_Y\,..])$ in $O(\lg^* n(\lg(\ell/2^\eta) + 3^\eta/\log_\sigma n))$ time using $O(\lg(n/2^\eta))$ words of working space. The additional data structures can be constructed in $O(n)$ time with $O(n/\lg n)$ 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 $\mathsf{tHT}_\eta(Y)$ contains at most $n/2^\eta$ nodes, the pointer-based tree structure takes $O(n/2^\eta)$ words. In this sense, η has a direct impact on the size of the tree, and solutions using $O(n)$ bits become larger than the space bounds of the tree when $\eta = \omega(\lg\lg n)$. Here, we focus on three approaches for different values of η:

Given that $\eta \le \lg\lg n$, we augment the tree structure with a bit vector to jump from a text position to an η-node like in Corollary 3.8: We create a bit vector of length n marking the borders of the generated substrings of the η-nodes such that a rank-support on this bit vector allows us to jump from a position $Y[i]$ to the η-node $\langle Y\rangle_\eta[j]$ with $1 + \sum_{k=1}^{j-1} \mathrm{string}(\langle Y\rangle_\eta[k]) \le i \le \sum_{k=1}^{j} \mathrm{string}(\langle Y\rangle_\eta[k])$ in constant time. The bit vector with its rank-support takes $O(n/\lg n)$ words, which is too much to obtain the space bounds of $O(n/2^\eta)$ words when $\eta = \Omega(\lg\lg n)$.

Instead, we compute a sorted list of pairs if $\eta \ge \log_3(\lg^2 n)$. 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 $O(n/2^\eta)$ words, and we can find the η-node whose generated substring covers a given position in $O(\lg(n/2^\eta)) = O(\lg n)$ time by binary searching the starting positions. This time is bounded by the time $O(\lg^* n\, 3^\eta / \log_\sigma n)$ for scanning the generated substrings of all η-nodes during an LCE query, which is $O(\lg^* n \lg n \lg \sigma)$ time when $\eta \geq \log_3(\lg^2 n)$.

It is left to consider the case that $\lg \lg n < \eta < \log_3 \lg^2 n$. 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 $O(k \lg(n/k) + k^2/n + k(\lg \lg k)^2/\lg k) = O(k\eta)$ bits (which are $O(n/2^\eta)$ words) when $k = n/\lg^c n$ for a 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 $O(n/\lg n)$ words of space. □

With $\tau := 2^\eta$, we obtain the claim of Theorem 1.3.

Remark 5.4. In the following, we stick to the result obtained in Lemma 5.2 instead of Corollary 5.3. Although Lemma 5.2 has a slower running time for LCPs that are short, the additional rank-support of Corollary 5.3 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 Theorem 1.1). To merge two trees, where each tree is augmented with the bit vector and its rank-support, the task would be to build a rank-support for the concatenation of the bit vectors (preferably in logarithmic time). Unfortunately, we are not aware of a rank-support that is efficiently mergeable (a naïve solution is to build the rank-support of the large bit vector from scratch in linear time).

5.2 Sparse Suffix Sorting with Truncated HSP Trees

To use the η-truncated HSP trees as dynLCEs in the situation where they are stored *in text space*, we need an adapted merge operation. Like with HSP trees, merging two η-truncated HSP trees involves a reparsing of the nodes at the facing borders (cf. Figure 31). However, the reparsing of the η-nodes on those borders is especially problematic, as can be seen in Figure 30: Suppose that we rename an η-node v from N_2 to N_3 with $|\text{string}(\mathsf{N}_2)| < |\text{string}(\mathsf{N}_3)|$. If the name N_3 is not yet maintained in the dictionary, we have to create N_3, i.e., a pointer to a substring X of the text with $X = \text{string}((\mathsf{N}_3))$. 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 3.4 shows that X is a prefix of the generated substring of a sibling η-node (unlike in Figure 22, where the generated substring of the ESP node with name U cannot be easily determined). With this insight, we finally show an approach that proves Theorem 1.1. For that, it remains to implement Rule 3 and Rule 4 from Section 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 $\mathsf{tHT}_\eta(Y)$ fits into $|Y| \lg \sigma$ bits (needed when creating new trees in Rule 3), and
- Goal 2: how to merge two η-truncated HSP trees without the need of extra working space (needed in Rule 4).

5.2.1 Storing Truncated HSP Trees in Text Space. Our first goal is to store $\mathsf{tHT}_\eta(T[\mathcal{I}])$ in a text interval $\mathcal{I}$. Since $\mathsf{tHT}_\eta(T[\mathcal{I}])$ can contain nodes with $|\mathcal{I}|/2^\eta$ distinct names, it requires $O(|\mathcal{I}|/2^\eta)$ words, i.e., $O(|\mathcal{I}| \lg n/2^\eta)$ 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

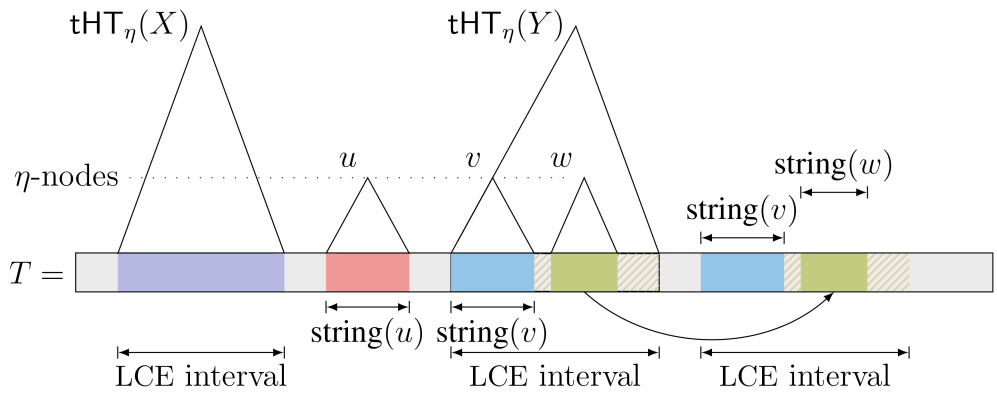


Fig. 30. Problem with generated substrings when merging tHT$_\eta(X)$ and tHT$_\eta(Y)$. Assume that we want to merge tHT$_\eta(X)$ and tHT$_\eta(Y)$, and thus compute the η-nodes (like u) between both trees. On the one hand, we cannot easily find a surrogate substring for the generated substring of a non-surrounded η-node like v or of a newly created η-node like u. Although there is a second occurrence of string(v) to the right, string(v) can be extended or shortened when prepending characters (e.g., suppose that string$(v) = \mathsf{a}^k$ and that there is an a to the left of the left occurrence of string(v), but not to the left of the right occurrence). Hence, it is a problem to overwrite string(u) or the left occurrence of string(v). On the other hand, we can find suitable surrogate substrings for the generated substrings of the η-nodes like for w that are not near the borders of an LCE interval.

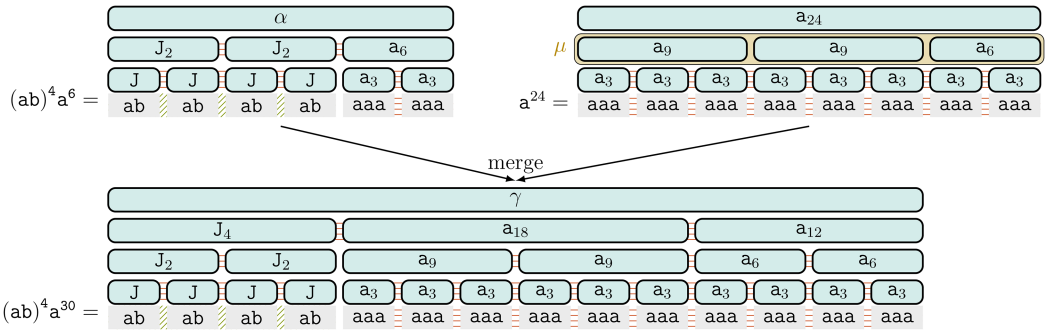


Fig. 31. Merging HT$((\mathsf{ab})^4\mathsf{a}^6)$ with HT(a^{24}) (both at the top) to HT$((\mathsf{ab})^4\mathsf{a}^{30})$ (bottom tree). Reparsing the repeating meta-block μ on height one of the right tree is done by recomputing μ's fragile nodes.

space issue by setting

$$\eta := \log_3(\alpha \lg^2 n / \lg \sigma).$$

Lemma 5.5. *The number of nodes of an η-truncated HSP tree on a substring of length ℓ is bounded by $O(\ell(\lg\sigma)^{\log_3 2}/(\lg n)^{2\lg_3 2}) = O(\ell(\lg\sigma)^{0.7}/(\lg n)^{1.2})$ 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

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

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

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

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

Plugging this value of η into Lemma 5.1 and Lemma 5.2 yields two corollaries for the η-truncated HSP trees:

Corollary 5.6. *We can compute an η-truncated HSP tree on a substring of length ℓ in $O(\ell \lg^* n + t_{\text{look}}\ell/2^\eta + \ell \lg\lg n)$ time. The tree takes $O(\ell/2^\eta)$ words of space. We need a working space of $O(\lg^2 n \lg^* n/\lg\sigma)$ characters.*

Proof. The tree has at most $\ell/2^\eta$ nodes, and thus takes $O(\ell/2^\eta)$ words of space. According to Lemma 5.1, constructing an η-node uses $O(3^\eta \lg^* n) = O(\lg^2 n \lg^* n/\lg\sigma)$ characters as working space. $\square$

Corollary 5.7. *An LCE query on two η-truncated HSP trees can be answered in $O(\lg^* n \lg n)$ time.*

Proof. LCE queries are answered as in Lemma 5.2, 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 $O(\alpha \lg n)$ time. Overall, we compare $O(\lg^* n)$ η-nodes, such that these additional costs are bounded by $O(\lg^* n \lg n)$ time overall, and do not slow down the running time $O(\lg^* n \lg(n/2^\eta) + \lg^* n \lg n) = O(\lg^* n \lg n)$. $\square$

5.2.2 Merging of Truncated HSP Trees. Our second and final goal is to adapt the merging used in the sparse suffix sorting algorithm (Section 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 $\mathsf{tHT}_\eta(T[i\,..\,i+\ell-1])$ in the text space $[j\,..\,j+\ell-1]$, leaving $T[i\,..\,i+\ell-1]$ untouched so 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

(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 $\mathsf{tHT}_\eta(T[i\,..\,i+\ell-1])$ 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 $T[\mathrm{e}(\mathcal{I})+1\,..\,\mathrm{b}(\mathcal{J})-1]$ when merging $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ to $\mathsf{tHT}_\eta(T[\mathrm{b}(\mathcal{I})\,..\,\mathrm{e}(\mathcal{J})])$ for two intervals $\mathcal{I}$ and $\mathcal{J}$ with $\mathrm{b}(\mathcal{I}) < \mathrm{b}(\mathcal{J})$ and $|[\mathrm{e}(\mathcal{I})+1\,..\,\mathrm{b}(\mathcal{J})-1]| < g$, as performed in Rule 4.

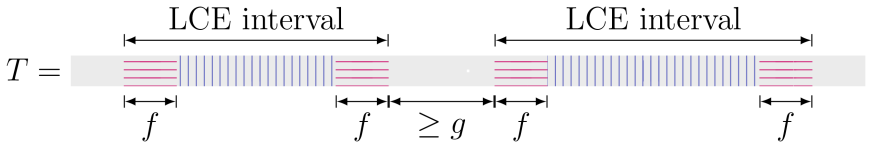


Fig. 32. Division of LCE intervals in protected and recyclable parts. The protected and the recyclable parts are depicted with horizontal magenta lines ( ) and vertical violet lines ( ), respectively.

(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 already 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 $T[i \mathinner{..} i + \ell - 1] = \text{string}(v)$. The new name of a fragile *surrounded* η-node v can be created easily: According to Lemma 3.4, 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 $T[j \mathinner{..} j + \ell - 1]$ (remember node v of Figure 30). 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 $[j \mathinner{..} j + \ell - 1]$ as protected. Conceptually, we partition an LCE interval into (1) ***recyclable*** and (2) ***protected*** intervals (see Figure 32); 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 comply with the following property:

> Property 6: $\lceil 2\alpha \lg^2 n \Delta_{\mathrm{L}} / \lg \sigma \rceil = \Theta(g)$ 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 $T[j \mathinner{..} j + f - 1]$ or $T[j + \ell - 1 - f \mathinner{..} j + \ell - 1]$.

(b) Reserving text space. We can store the upper part of the η-truncated HSP tree in a recyclable interval, because it needs $\ell/2^{\eta} \lg n \leq \ell \alpha^{0.6} (\lg \sigma)^{0.7} / (\lg n)^{0.2} = o(\ell \lg \sigma)$ 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 $f \lg \sigma / \lg n = O(\lg^* n \lg n)$ words on a recyclable interval untouched, sufficiently large for the working space needed by Corollary 5.6. Therefore, we precompute α and g based on the input text T and set both as *global* constants dependent on σ and n. 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.

(c) Interval overlapping. 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.,

$$[i \mathinner{..} i + \text{lce}(i, j) - 1] \cap [j \mathinner{..} j + \text{lce}(i, j) - 1] \neq \emptyset.$$

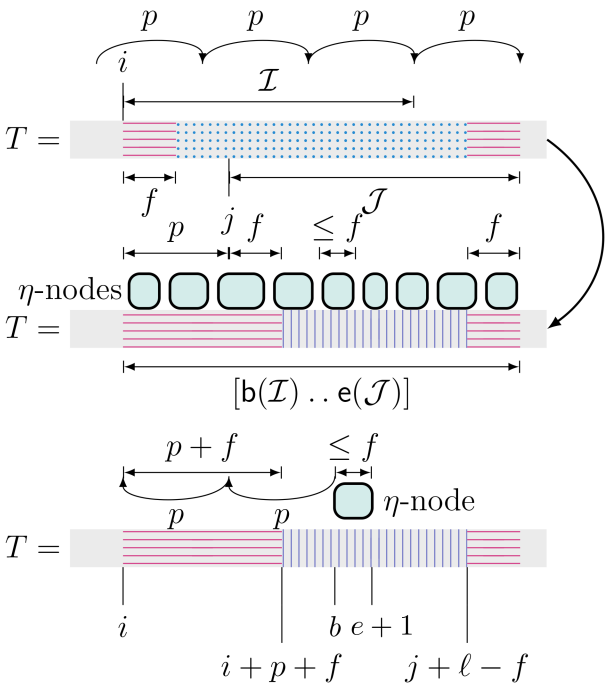


Fig. 33. *Top*: Overlapping LCE intervals $\mathcal{I} = [i \,..\, i+\ell-1]$ and $\mathcal{J} = [j \,..\, j+\ell-1]$. *Middle*: Partitioning $\mathcal{I}$ and $\mathcal{J}$ as described in (c). *Bottom*: Finding the generated substring $T[b \,..\, e]$ of an η-node in a protected interval. Given that p is a 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 $\mathsf{tHT}_\eta(T[i \,..\, j+\ell-1])$ in $T[i \,..\, i+p+f-1]$. The protected and the recyclable parts of the LCE intervals are depicted with horizontal magenta lines (▭) and vertical violet lines (▭), respectively. Parts that have not yet been declared as protected or recyclable are dotted (▭).

The algorithm would proceed with merging both overlapping LCE intervals to satisfy Property 5. However, the merged LCE interval cannot respect Properties 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).

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 $\mathsf{tHT}_\eta(T[i \,..\, j+\ell-1])$ (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 $1 \le p \le j-i$, 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 $k \ge 2$ ($k > 1$, since $j \le i+\ell-1$, otherwise, $i > j$ or $\mathcal{I} \cap \mathcal{J} = \emptyset$). 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 $3^\eta = \alpha \lg^2 n / \lg \sigma \le f/2$. Second, given an η-node with the generated substring $T[b \,..\, e]$ with $i+p+f \le e \le j+\ell-1$, we find an integer k with $k \ge 0$ 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 $e-b \le f/2$). Hence, we can make the interval $[i+p+f+1 \,..\, j+\ell-1-f]$ *recyclable*, which is at least as large as f, since $|\mathcal{I} \cup \mathcal{J}| \ge j-i+2g \ge p+2g$ is at least $p+3f$ for a sufficiently large g. This partitioning into protected and recyclable intervals is illustrated in Figure 33.

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. In contrast to

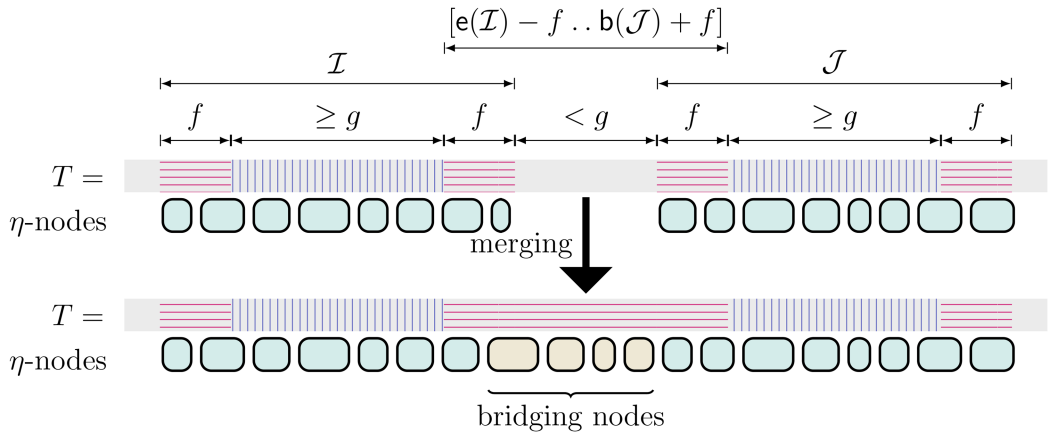


Fig. 34. Merging $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ with $\mathsf{b}(\mathcal{J}) - g \leq \mathsf{e}(\mathcal{I}) \leq \mathsf{b}(\mathcal{J}) - 1$. The substring $T[\mathsf{e}(\mathcal{I}) - f \,..\, \mathsf{b}(\mathcal{J}) + f]$ is marked protected for the sake of the bridging nodes.

Lemma 4.3, 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 $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ be built on two LCE intervals $\mathcal{I}$ and $\mathcal{J}$ with $1 \leq \mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I}) \leq g$. Our task is to compute the merged tree $\mathsf{tHT}_\eta(T[\mathsf{b}(\mathcal{I}) \,..\, \mathsf{e}(\mathcal{J})])$. We do that by (a) reprocessing $O(\Delta_\mathrm{L} + \Delta_\mathrm{R})$ nodes at every height of both trees (according to Lemma 4.3) 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 $T[\mathsf{e}(\mathcal{I}) - f \,..\, \mathsf{b}(\mathcal{J}) + f]$ would invalidate the generated substrings of the bridging nodes and of some (formerly) non-surrounded nodes. Therefore, we also mark the interval $[\mathsf{e}(\mathcal{I}) - f \,..\, \mathsf{b}(\mathcal{J}) + f]$ as protected. By doing so, we can use the characters of $T[\mathsf{e}(\mathcal{I}) - f \,..\, \mathsf{b}(\mathcal{J}) + f]$ to (a) create the bridging η-nodes and to (b) reparse the non-surrounded nodes of both trees (Figure 34). The bridging nodes and their ancestors take $o(\lg n \lg^* n)$ words of additional space, since building $\mathsf{tHT}_\eta(T[\mathsf{e}(\mathcal{I}) + 1 \,..\, \mathsf{b}(\mathcal{J}) - 1])$ with $|\mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I})| = O(g)$ 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 5.8. *Given two LCE intervals $\mathcal{I}$ and $\mathcal{J}$ with $\mathsf{b}(\mathcal{I}) \leq \mathsf{b}(\mathcal{J}) \leq \mathsf{e}(\mathcal{I}) + g$ and their respective η-truncated HSP trees, we can build $\mathsf{tHT}_\eta(T[\mathsf{b}(\mathcal{I}) \,..\, \mathsf{e}(\mathcal{J})])$ in $O(g \lg^* n + t_\mathrm{look} g/2^\eta + g\eta/\log_\sigma n + t_\mathrm{look} \lg^* n \lg n)$ time.*

Proof. We adapt the merging of two HSP trees (Lemma 4.3) for the η-truncated HSP trees. The difference to Lemma 4.3 is that we reparse an η-node by rebuilding its local surrounding consisting of $O((\Delta_\mathrm{L} + \Delta_\mathrm{R})3^\eta)$ nodes that take $\alpha(\Delta_\mathrm{L} + \Delta_\mathrm{R}) \lg^2 n / \lg \sigma \leq f$ 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 Corollary 5.6. Both creating and recomputing takes overall $O(g \lg^* n + t_\mathrm{look} g/2^\eta + g\eta/\log_\sigma n)$ time. □

There is one problem left before we can prove the main result of this article: The sparse suffix sorting algorithm of Section 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 Corollary 5.8 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 Theorem 1.1 The analysis is split into suffix comparison, tree generation, and tree merging:

- Suffix comparisons are done as in Corollary 4.2. LCE queries on η-truncated HSP trees and HSP trees are conducted in the same time bounds (compare Lemma 3.7 with Corollary 5.7).
- All positions considered for creating the η-truncated HSP trees belong to C. Constructing the η-truncated HSP trees costs $\mathcal{O}(|C| \lg^* n + t_{\text{look}} |C| / 2^\eta + |C| \lg \lg n)$ overall time, due to Corollary 5.6.
- Merging in the fashion of Corollary 5.8 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 Theorem 4.4 that the time for merging is upper bounded by the construction time.

Plugging the times for suffix comparisons, tree construction and merging in Corollary 4.2 yields the overall time

$$\begin{aligned}\mathcal{O}(t_C(|C|)) &= \mathcal{O}(|C| \lg^* n + t_{\text{look}} |C| / 2^\eta + |C| \lg \lg n) \\ &= \mathcal{O}\left(|C| \left(t_{\text{look}} (\lg \sigma)^{0.7} / (\lg n)^{1.2} + \lg \lg n\right)\right) \\ &= \mathcal{O}\left(|C| \left(\sqrt{\lg \sigma} + \lg \lg n\right)\right)\end{aligned}$$

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

6 CONCLUSIONS

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 be merged 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 of the HSP tree 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 a new deterministic LCE data structure with a trade-off parameter. Although not shown here, ESP trees can also (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 show that there are strings of length n whose ESP trees differs by $\Omega(\lg^2 n)$ nodes from the ESP trees of the original inputs modified by the first characters, which invalidates the upper bound of $\mathcal{O}(\lg n \lg^* n)$ on the maximal number of fragile nodes postulated in Reference [11]. This result also invalidates theoretical results that depend on the ESP technique (e.g., for approximating the SEDM [11] or the Lempel-Ziv-77 (LZ77) factorization [10], or for building indexes [14, 30, 40, 41]). 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 new HSP technique can be used as a

substitute for the ESP technique, since HSP trees and ESP trees share the same bounds for construction time and space usage. This way, there are only $\mathcal{O}(\lg n \lg^* n)$ fragile nodes, as promised in the original ESP paper. This also recovers the postulated $\mathcal{O}(\lg n \lg^* n)$ approximation bound on the edit distance matching problem [11, 40]: Given $\mathsf{ET}(T)$ 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 $\mathsf{ET}(T)$. Although we only provided proofs that pre-/appending characters to T changes $\mathcal{O}(\lg^* n \lg n)$ nodes of $\mathsf{HT}(T)$, it is easy to generalize this result by applying a merge operation: Given that we insert a character $\mathsf{c} \in \Sigma$ between $T[i]$ and $T[i+1]$, the trees $\mathsf{HT}(T)$ and $\mathsf{HT}(T[1\,..\,i]\mathsf{c}T[i+1\,..])$ differ in at most $\mathcal{O}(\lg^* n \lg n)$ nodes, since appending c to $\mathsf{HT}(T[1\,..\,i])$ and merging $\mathsf{HT}(T[1\,..\,i]\mathsf{c})$ with $\mathsf{HT}(T[i+1\,..])$ changes $\mathcal{O}(\lg^* n \lg n)$ nodes. The same can be observed when deleting or changing the ith character.

Our open problems are:

Practical evaluation. In light of the theoretical improvements of the HSP over the ESP, it would be interesting to evaluate how the HSP behaves practically. Especially, we are interested in how well the HSP behaves in the context of grammar compression [3] like the ESP-index [30, 41] on highly repetitive texts, where a more stable behavior of the repetitive nodes could lead to an improved compression ratio. Speaking about implementing the complete suffix sorting algorithm, a major problem is to choose the parameter α (defined in Section 5.2.1) and g (defined in Property 2) wisely, since they affect the practical computation time as well as the space needed for storing an η-truncated HSP tree. At least for highly repetitive texts, our approach can be practically faster than a naive approach, since the ESP grammar tends to become especially small on those kinds of texts [40, Tables 4 to 6], making our approach to store the trees in text space more feasible. However, we do not see a chance that our deterministic algorithm can compete with practical non-deterministic solutions resorting to hashing like the approach of Prezza [35], which also works in the restore model.

Suffix sorting with trade-off parameter. From the theoretical point of view, it would be interesting to compute the sparse suffix sorting with a trade-off parameter adjusting working space and construction time of SSA and SLCP.

Mergeable rank-support. Remembering Remark 5.4, we are unaware of whether rank-support data structures can be mergeable. Given two bit vectors B_1 and B_2, both with a rank-support data structure, the task is to compute a rank-support data structure on the concatenation of B_1 and B_2 in sub-linear time in the total lengths of both bit vectors.

Construction space aware compressed bit vectors. Although there are bit vectors with rank-support that can be stored in compressed space (e.g., Reference [34]), there is, to the best of our knowledge, no (compressed) bit vector representation that can be constructed within compressed space or on-line.

Improved Alphabet Reduction Technique. Finally, we thank an anonymous reviewer for pointing us to a new technique [15] that appeared after the initial submission of this article, which might improve the running time of our algorithm even further to $\mathcal{O}(|C|\,(\sqrt{\lg \sigma} + \lg \lg n) + m \lg m \lg n)$. We leave this as future work.

APPENDIX

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. The idea is to give an example that contains a large number of Type M

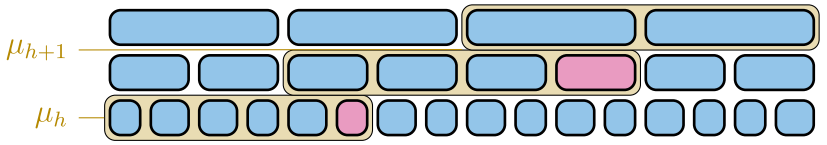


Fig. 35. Basic idea of our two counter-examples described in Theorems A.2 and A.4. We build a counter-example in such a way that the last node of a certain repeating meta-block μ_h on height h (a) is fragile ( ), and (b) is the child of the first node of a repeating meta-block μ_{h+1} having the same properties as μ_h. This property can cause a recursive chain reaction when prepending a suitable character such that the names of the last and the first block of a meta-block on height h and a meta-block on height $h+1$ are changed, on each height h.

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 Type M meta-block. We provide examples for Rule (M) and for the original tie-breaking rule for Type M meta-blocks:

> Rule (M'): Fuse a remaining character $Y[i]$ with its *preceding* meta-block, or, if $i = 1$, with its succeeding meta-block.

In each example, we present a string of length at most n whose ESP tree has $\Omega(\lg^2 n)$ fragile nodes. There, we modify the input string by one character and show that $\Omega(\lg^2 n)$ nodes differ from the original tree as a result of this modification. Since we can change $\Omega(\lg^2 n)$ nodes by changing one character in the input, we have a contradiction to Lemma 9 in Reference [11], where it is claimed that $O(\lg^* n \lg n)$ nodes change after such a modification.

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 (Figure 35). 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. Before giving such an example, we introduce a lemma showing the associativity of esp on a special class of strings, which helps us proving the lower bound:

Lemma A.1. *Suppose that we comply to Rule (M'). Given a height h and two strings X, Y that are either empty or have a length of at least $2 \cdot 3^{h-1}$, $esp^{(h)}(X\mathsf{b}^{3^i}Y) = esp^{(h)}(X)\, esp^{(h)}(\mathsf{b}^{3^i})\, esp^{(h)}(Y)$ if $i \geq h$, b is neither a suffix of X nor a prefix of Y, and there is no prefix of $esp^{(j)}(Y)$ of the form cd^k for some symbols $\mathsf{c}, \mathsf{d} \in \Sigma_j$ with $\mathsf{c} \neq \mathsf{d}$, 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 $\mathrm{esp}^{(j)}(Y)$ is not a non-repetitive Type M block that has been fused to its succeeding meta-block, only because it has no preceding meta-block. Regardless of which symbols are prepended to $\mathrm{esp}^{(j-1)}(Y)$, the first symbol of such a block would form with its preceding symbols a new block.

For $h = 1$, esp divides the string $X\mathsf{b}^{3^i}Y$ into meta-blocks such that there is one Type 1 meta-block μ that exactly contains the substring b^{3^i}. That is because of the following: If X (respectively,

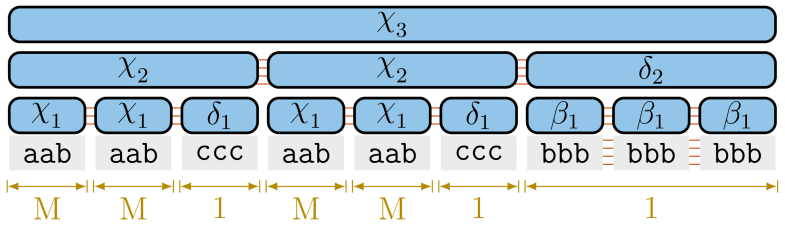


Fig. 36. $\mathsf{ET}(X_3)$ as defined in Theorem A.2. The subtree of each node with name χ_i is equal to $\mathsf{ET}(X_i)$. The meta-blocks of the lowest height are labeled with their types.

Y) is not the empty string, then X (respectively, Y) 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 $\mathrm{esp}(Y)$ is neither a non-repetitive Type M block nor a block starting with 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

$$\begin{aligned}\mathrm{esp}^{(h)}(X\mathsf{b}^{3^i}Y) &= \mathrm{esp}\Big(\mathrm{esp}^{(h-1)}(X\mathsf{b}^{3^i}Y)\Big)\\ &= \mathrm{esp}\Big(\mathrm{esp}^{(h-1)}(X)\,\mathrm{esp}^{(h-1)}(\mathsf{b}^{3^i})\,\mathrm{esp}^{(h-1)}(Y)\Big).\end{aligned}$$

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

Theorem A.2. *There is a text Y of length n whose ESP tree differs by $\Omega(\lg^2 n)$ nodes from the ESP tree of $Y[2]\cdots Y[n]$ when complying to Rule (M').*

Proof. Let a, b, and $\mathsf{c} \in \Sigma$ be three different characters. Further, let

$$Y := (X_0)^{3^k}(X_1)^{3^{k-1}}(X_2)^{3^{k-2}}\cdots(X_{k-1})^3 \text{ with } k := \left\lfloor \log_3(n/\log_3 n)\right\rfloor,$$

$$X_0 := \mathsf{a}, \text{ and } X_i := \begin{cases} X_{i-1}^2\mathsf{b}^{3^{i-1}} & \text{if } i \text{ is odd,}\\ X_{i-1}^2\mathsf{c}^{3^{i-1}} & \text{if } i \text{ is even,}\end{cases} \text{ for } i = 1,\ldots,k.$$

For instance, $X_0 = \mathsf{a}$, $X_1 = \mathsf{aab}$, $X_2 = \mathsf{aabaabc}^3$, and $X_3 = X_2^2\mathsf{b}^9$.

In the following, we show that the text Y has a length at most n and that its ESP tree has $\Omega(\lg^2 n)$ fragile nodes, which change when removing the first character of Y. 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 $i = 0,\ldots,k-1$. 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 χ_i of $\mathsf{ET}(Y)$ on height i whose subtree is equal to the perfect ternary subtree $T_i := \mathsf{ET}(X_i)$, for $i = 1,\ldots,k-1$. This is true for $i = 1, 2, 3$, as can be seen in Figure 36. For the general case, we adapt the associativity shown for esp in Lemma A.1 to the string X_i:

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

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

(II) $$\begin{aligned}\mathrm{esp}^{(h)}(X_{i+1}) &= \mathrm{esp}^{(h)}\left(X_iX_i\mathsf{d}_i^{3^i}\right)\\ &= \mathrm{esp}^{(h)}(X_iX_i)\,\mathrm{esp}^{(h)}\left(\mathsf{d}_i^{3^i}\right)\\ &= \mathrm{esp}^{(h)}(X_i)\,\mathrm{esp}^{(h)}(X_i)\,\mathrm{esp}^{(h)}\left(\mathsf{d}_i^{3^i}\right), \text{ and}\end{aligned}$$

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

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

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

(I) $|\mathrm{esp}^{(1)}(X_1)| = |\mathrm{esp}(\mathsf{aab})| = 1$ (aab is put in a Type M meta-block having exactly one block),
(II) $\mathrm{esp}^{(0)}(X_1) = X_1$, and
(III) $X_1 = \mathsf{aab}$ starts with a repetition of the character a.

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

$$\begin{aligned}
\mathrm{esp}^{(h)}(X_{i+2}) &= \mathrm{esp}^{(h)}\left(X_{i+1}X_{i+1}\mathsf{d}_{i+1}^{3^{i+1}}\right)\\
&= \mathrm{esp}^{(h)}\left(X_iX_i\mathsf{d}_i^{3^i}X_iX_i\mathsf{d}_i^{3^i}\mathsf{d}_{i+1}^{3^{i+1}}\right)\\
&\overset{(\text{Lemma A.1}, \mathsf{d}_i \neq \mathsf{d}_{i+1})}{=} \mathrm{esp}^{(h)}\left(X_iX_i\mathsf{d}_i^{3^i}X_iX_i\mathsf{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\\
&\overset{(\text{Lemma A.1,(I) or (III)})}{=} \mathrm{esp}^{(h)}(X_iX_i)\mathrm{esp}^{(h)}\left(\mathsf{d}_i^{3^i}\right)\mathrm{esp}^{(h)}(X_iX_i)\mathrm{esp}^{(h)}\left(\mathsf{d}_i^{3^i}\right)\\
&\qquad \mathrm{esp}^{(h)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\\
&\overset{(\text{Lemma A.1,(I) or (III)})}{=} \mathrm{esp}^{(h)}\left(X_iX_i\mathsf{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(X_iX_i\mathsf{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\\
&= \mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)
\end{aligned}$$

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

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

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

$$\begin{aligned}
\mathrm{esp}^{(i+1)}(X_{i+2}) &= \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_{i+2})\right)\\
&= \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathsf{d}_i^{3^i})\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}\left(\mathsf{d}_i^{3^i}\right)\right.\\
&\qquad \left.\mathrm{esp}^{(i)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\right)\\
&\overset{(\text{Lemma A.1})}{=} \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathsf{d}_i^{3^i})\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}\left(\mathsf{d}_i^{3^i}\right)\right)\\
&\qquad \mathrm{esp}\left(\mathrm{esp}^{(i)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\right)\\
&\overset{(\text{evaluate and reformulate})}{=} \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}\left(\mathsf{d}_i^{3^i}\right)\right)\\
&\qquad \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}\left(\mathsf{d}_i^{3^i}\right)\right)\mathrm{esp}\left(\mathrm{esp}^{(i)}\left(\mathsf{d}_{i+1}^{3^{i+1}}\right)\right),
\end{aligned}$$

where we used the two facts that

—another application of esp puts $\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathsf{d}_i^{3^i})$ into a single Type M 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 $h \le i$, we have $\mathrm{esp}^{(h)}(X_{i+2}) = \mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(\mathsf{d}_{i+1}^{3^{i+1}})$, and $\mathrm{esp}^{(h)}(X_{i+1})$ starts with a repetition of a symbol according to our assumption. For $h = i+1$, we have

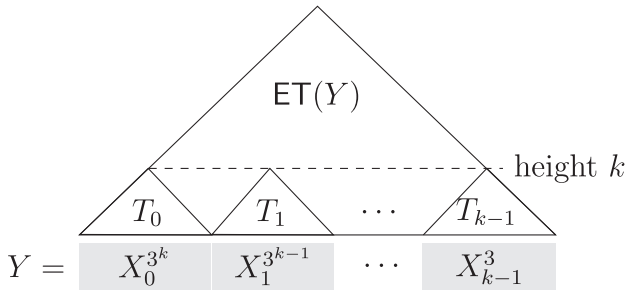
$$\mathrm{esp}^{(i+1)}(X_{i+2}) = \mathrm{esp}\Big(\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}\big(\mathsf{d}_i^{3^i}\big)\,\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}\big(\mathsf{d}_i^{3^i}\big)\,\mathrm{esp}^{(i)}\big(\mathsf{d}_{i+1}^{3^{i+1}}\big)\Big).$$

Due to (I), $|\,\mathrm{esp}^{(i)}(X_i)| = |\,\mathrm{esp}^{(i)}(\mathsf{d}_i^{3^i})| = 1$; hence, the last application of esp creates three blocks, where each of the first two represents the string $\mathrm{esp}^{(i)}(X_i)\ \mathrm{esp}^{(i)}(X_i)\ \mathrm{esp}^{(i)}(\mathsf{d}_i^{3^i})$ of length three. Another application of esp yields (I). This concludes the sub-proof.

Let β_i and γ_i denote the names of the roots of $\mathsf{ET}(\mathsf{b}^{3^i})$ and of $\mathsf{ET}(\mathsf{c}^{3^i})$, respectively. Set $\delta_i := \beta_i$ if i is even, otherwise, $\delta_i := \gamma_i$. Then $\langle X_{i+1}\rangle_{i+1} = \chi_{i+1}$ due to Sub-Claim (I), and $\langle X_{i+1}\rangle_i = \chi_i\chi_i\delta_i$ due to Sub-Claim (II). Consequently,

$$\mathrm{esp}\Big(\big(\langle X_{i+1}\rangle_i\big)^{3^{k-i-1}}\Big) = \mathrm{esp}\Big((\chi_i\chi_i\delta_i)^{3^{k-i-1}}\Big) = (\mathrm{esp}(\chi_i\chi_i\delta_i))^{3^{k-i-1}} = \chi_{i+1}^{3^{k-i-1}}. \tag{2}$$

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 $h = i, \ldots, k$. We conclude that $T_i := \mathsf{ET}((X_i)^{3^{k-i}})$ is a perfect ternary tree.



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

$$\begin{aligned}
\mathrm{esp}^{(h)}\Big(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\Big) &= \mathrm{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} \mathsf{d}_{i-1}^{3^{i-1}} X_{i+1}^{3^{k-i-1}}\Big) \\
&\overset{\text{(III) with } 0\le i\le h-2}{=} \mathrm{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1}\Big)\mathrm{esp}^{(h)}\big(\mathsf{d}_{i-1}^{3^{i-1}}\big)\,\mathrm{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big) \\
&= \mathrm{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} \mathsf{d}_{i-1}^{3^{i-1}}\Big)\mathrm{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big) \\
&= \mathrm{esp}^{(h)}\Big(X_i^{3^{k-i}}\Big)\mathrm{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big)
\end{aligned} \tag{3}$$

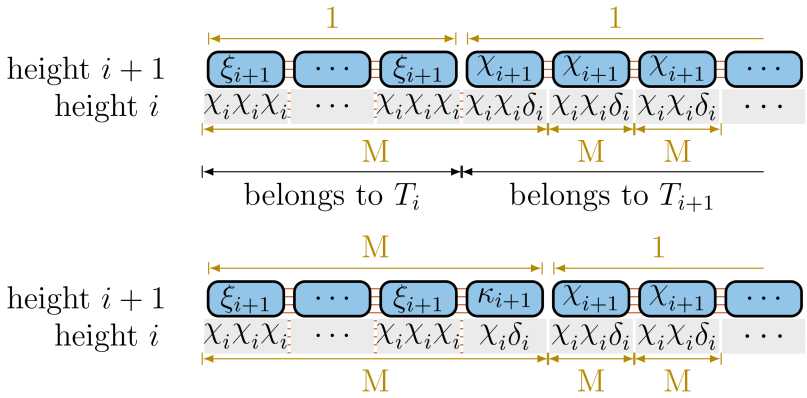


Fig. 37. Differences between ET(Y) (*top*) and ET(Y') (*bottom*) on the heights i and $i+1$, where $Y = \mathsf{a}^{3^k}(\mathsf{a}^2\mathsf{b})^{3^{k-1}}((\mathsf{a}^2\mathsf{b})^2\mathsf{c}^3)^{3^{k-2}}\cdots$ and $Y = \mathsf{a}Y'$ (defined in Theorem A.2). The names ξ_{i+1} and κ_{i+1} are only used in this figure. The meta-blocks on height i and $i+1$ are labeled with their types.

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

$$\begin{aligned}
\mathrm{esp}^{(h)}\left(X_i^{3^{k-i}}X_{i+1}^{3^{k-i-1}}\right) &= \mathrm{esp}^{(h-i+1)}\left(\mathrm{esp}^{(i-1)}\left(X_i^{3^{k-i}}X_{i+1}^{3^{k-i-1}}\right)\right)\\
&\overset{\text{Equation (3)}}{=} \mathrm{esp}^{(h-i+1)}\left(\mathrm{esp}^{(i-1)}\left(X_i^{3^{k-i}}\right)\mathrm{esp}^{(i-1)}\left(X_{i+1}^{3^{k-i-1}}\right)\right)\\
&\overset{\text{Equation (2)}}{=} \mathrm{esp}^{(h-i)}\Big(\mathrm{esp}\Big((\chi_{i-1}\chi_{i-1}\delta_{i-1})^{3^{k-i}}\\
&\qquad (\chi_{i-1}\chi_{i-1}\delta_{i-1}\chi_{i-1}\chi_{i-1}\delta_{i-1}\langle \mathsf{d}_i^{3^i}\rangle_{i-1})^{3^{k-i-1}}\Big)\Big)\\
&\overset{\text{(apply esp)}}{=} \mathrm{esp}^{(h-i)}\left(\chi_i^{3^{k-i}}(\chi_i\chi_i\delta_i)^{3^{k-i-1}}\right)\\
&= \mathrm{esp}^{(h-i-1)}\left(\mathrm{esp}\left(\chi_i^{3^{k-i}}\chi_i\chi_i\delta_i\right)\mathrm{esp}\left((\chi_i\chi_i\delta_i)^{3^{k-i-2}}\right)\right)\\
&\overset{\text{(evaluate and reformulate)}}{=} \mathrm{esp}^{(h-i-1)}\left(\mathrm{esp}\left(\chi_i^{3^{k-i}}\right)\mathrm{esp}\left((\chi_i\chi_i\delta_i)^{3^{k-i-1}}\right)\right)\\
&\overset{\text{Equation (2)}}{=} \mathrm{esp}^{(h-i-1)}\left(\mathrm{esp}\left(\chi_i^{3^{k-i}}\right)\mathrm{esp}\left(\chi_{i+1}^{3^{k-i-1}}\right)\right)\\
&\overset{\text{(Lemma A.1)}}{=} \mathrm{esp}^{(h-i)}\left(\chi_i^{3^{k-i}}\right)\mathrm{esp}^{(h-i)}\left(\chi_{i+1}^{3^{k-i-1}}\right)
\end{aligned} \tag{4}$$

for $i \le h \le k$. It is easy to extend the pairwise associativity $X_i^{3^{k-i}}X_{i+1}^{3^{k-i-1}}$ for each i with $0 \le i \le k-2$ to $X_1^{3^k}\cdots X_{k-1}^{3^1}$. This concludes that the root of T_i has the same name as the ith leftmost node of ET(Y) on height k. Figure 37(left) shows an excerpt of T_i and T_{i+1}. The crucial step in Equation (4) is the re-formulation of the parsing

$$\begin{aligned}
&\mathrm{esp}^{(h-i-1)}(\underbrace{\mathrm{esp}(\chi_i^{3^{k-i}}\chi_i\chi_i\delta_i)}_{\text{belongs to } T_i}\underbrace{\mathrm{esp}((\chi_i\chi_i\delta_i)^{3^{k-i-2}})}_{\text{belongs to } T_{i+1}})\\
&= \mathrm{esp}^{(h-i-1)}(\mathrm{esp}(\chi_i^{3^{k-i}})\underbrace{\mathrm{esp}((\chi_i\chi_i\delta_i)^{3^{k-i-1}})}_{=:\mu_{i+1}})
\end{aligned} \tag{5}$$

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 symbol $\mathrm{esp}(\chi_i\chi_i\delta_i) = \chi_{i+1} \in \Sigma_{h+1}$.

Given that μ_0 is the first Type 1 meta-block of $\mathrm{esp}(Y)$ (covering the prefix $X_0^{3^h+2}$), we now examine what happens with μ_i for each i with $0 \le i \le h-1$ when removing the first a from Y. Let us call the shortened string Y', i.e., $Y = \mathsf{a}Y'$. On removing the first a from Y, we claim that the meta-block μ_i contains one symbol χ_i less, for every i with $0 \le i \le h-1$ (cf. Figure 37 showing the difference between $\langle Y\rangle_i$ and $\langle Y'\rangle_i$ on height i with $0 \le i \le k-1$): For μ_0, this is trivial. For an $i \ge 0$, focus on the substring $X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}$ of Y: We have

$$\begin{aligned}\mathrm{esp}\left(\left\langle X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\right\rangle_i\right) &= \mathrm{esp}\left(\chi_i^{3^{k-i}} (\chi_i\chi_i\delta_i)^{3^{k-i-1}}\right)\\ &= \mathrm{esp}(\underbrace{\chi_i^{3^{k-i}}}_{\text{suffix of } \mu_i}) \underbrace{\mathrm{esp}\left((\chi_i\chi_i\delta_i)^{3^{k-i-1}}\right)}_{=\mu_{i+1}}\\ &= \mathrm{esp}\left(\chi_i^{3^{k-i}}\right)\chi_{i+1}^{3^{k-i-1}}\end{aligned}$$

due to Equation (4). Under the assumption that removing the first character a from Y causes μ_i to shrink by one symbol $\chi_i \in \Sigma_i$, we get

$$\begin{aligned}\mathrm{esp}\left(\chi_i^{3^{k-i}-1}(\chi_i\chi_i\delta_i)^{3^{k-i-1}}\right) &= \mathrm{esp}\left(\chi_i^{3^{k-i}}\chi_i\delta_i\right)\mathrm{esp}\left((\chi_i\chi_i\delta_i)^{3^{k-i-1}-1}\right)\\ &= \mathrm{esp}\left(\chi_i^{3^{k-i}}\chi_i\delta_i\right)\chi_{i+1}^{3^{k-i-1}-1}\\ &\neq \mathrm{esp}\left(\chi_i^{3^{k-i-1}}\right)\chi_{i+1}^{3^{k-i-1}}.\end{aligned}$$

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 symbol 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 $i = 0, \ldots, k-2$). This means that the leftmost node on height i of T_i changes, for $1 \le i \le k-1$. 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 (or removing the first character of Y for $i = 0$ changes all nodes built on this character), at least $k-i$ nodes are changed in T_i. In total, at least $k + (k-1) + (k-2) + \cdots + 2 = (k^2+k)/2 - 1$ nodes are changed. Hence, there is a lower bound of $\Omega(k^2) = \Omega(\log_3^2(n/\log_3 n)) = \Omega(\lg^2 n)$ changed nodes. □

Note that the later-introduced HSP technique (see Section 3) with the same tie-breaking rule also produces $\Omega(\lg^2 n)$ different nodes in this example. However, this is not the case when complying with Rule (M), as we will see later in Section 3.1.

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 A.1:

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

In the following example, we build a text whose ESP tree has a specific Type M meta-block on each height that we want to change. Given a Type M meta-block μ that emerged from prepending a symbol to a Type 1 meta-block, we can create a new meta-block by prepending another symbol such that it precedes μ and absorbs μ's first symbol (μ then returns to be a Type 1 meta-block). We

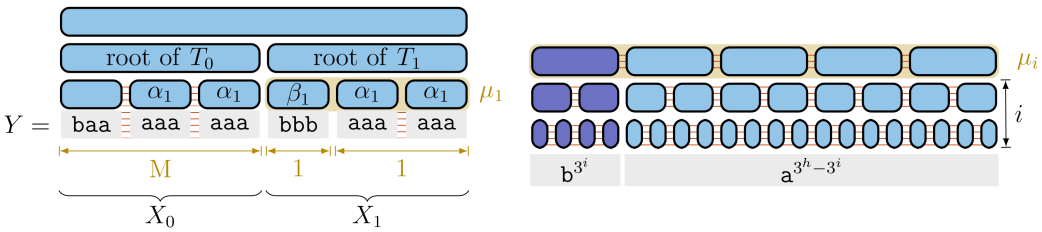


Fig. 38. ET(Y) of the example string Y defined in Theorem A.4 with $k = 1$ (*left*) and as a schematic illustration (*right*) with the meta-block μ_i on height i (due to space issues, the number of nodes/children is incorrect). The meta-blocks of the lowest height in the *left* figure are labeled with their types.

can arrange the Type M meta-blocks such that prepending a symbol to the text changes a Type M meta-block on each height:

Theorem A.4. *There is a text Y' of length n whose ESP tree differs by $\Omega(\lg^2 n)$ nodes from the ESP tree of $Y'[2] \cdots Y'[n]$ when complying with Rule (M).*

Proof. Let $k = \lfloor \log_3(n/\log_3 n) \rfloor$ be a natural number and $\mathsf{a}, \mathsf{b} \in \Sigma$. Define

$$Y := X_0 X_1 \cdots X_k \text{ with } X_i := \mathsf{b}^{3^i} \mathsf{a}^{3^k - 3^i},$$

for $0 \le i \le k-1$. Figure 38 gives an example on its left side. In the following, we show that $|Y| \le n$, and ET(Y) has $\Omega(\lg^2 n)$ fragile nodes, which change when prepending the character a to Y.

Given an integer i with $0 \le i \le k-1$, we have $|X_i| = 3^k$ and $|Y| = k3^k \le n$. Corollary A.3 yields $\mathrm{esp}^{(h)}(X_i) = \mathrm{esp}^{(h)}(\mathsf{b}^{3^i})\,\mathrm{esp}^{(h)}(\mathsf{a}^{3^k-3^i})$ for all heights h with $0 \le h \le i$, since $3^k - 3^i \ge 3^k - 3^{k-1} = 2 \cdot 3^{k-1}$. Let $\alpha_i := \langle \mathsf{a}^{3^k} \rangle_i[1]$ and $\beta_i := \langle \mathsf{b}^{3^k} \rangle_i[1]$ be the nodes on height i with $0 \le i \le k$ and, respectively, $\mathrm{string}(\alpha_i) = \mathsf{a}^{3^i}$ and $\mathrm{string}(\beta_i) = \mathsf{b}^{3^i}$ ($\alpha_0 := \mathsf{a}$, $\beta_0 := \mathsf{b}$).

The function esp applied on $\mathrm{esp}^{(h-1)}(X_i)$ partitions its input $\mathrm{esp}^{(h-1)}(X_i)$ into two meta-blocks: a Type 1 meta-block containing all β_i's and a subsequent Type 1 meta-block containing all α_i's. All blocks of these two meta-blocks contain three symbols, since each meta-block has a length that is equal to a power of three. For the upper heights, we get

$$\mathrm{esp}^{(h+i)}(X_i) = \mathrm{esp}^{(h)}\left(\overbrace{\underbrace{\mathrm{esp}^{(i)}(\mathsf{b}^{3^i})}_{=\beta_i}\underbrace{\mathrm{esp}^{(i)}(\mathsf{a}^{3^k-3^i})}_{=\alpha_i^{3^{k-i}-1}}}^{\text{having length } 3^{k-i}}\right) \text{ for } 0 \le h+i \le k-1. \tag{6}$$

Hence, $\mathrm{esp}^{(h+i)}(X_i)$ consists of exactly one Type M meta-block, which has length 3^{k-h-i}, and each block contains three symbols. We conclude that the tree $T_i := \mathrm{ET}(X_i)$ is a perfect ternary tree, for $0 \le i \le k-1$. Since $|\mathrm{esp}^{(h)}(X_i)| = 3^{k-h}$ for all i, h with $0 \le i \le k-1$ and $0 \le h \le k$, with Corollary A.3 it is easy to see that $\mathrm{esp}^{(h)}(Y) = \mathrm{esp}^{(h)}(X_1 \cdots X_{k-1}) = \mathrm{esp}^{(h)}(X_1) \cdots \mathrm{esp}^{(h)}(X_{k-1})$ for all $0 \le h \le k$. Consequently, X_i is the generated substring of the ith leftmost node v_i of ET(Y) on height k. The name of v_i is the name of the root of T_i, for $0 \le i \le k-1$.

For the proof, we prepend an a to Y and call the new string Y', i.e., $Y' = \mathsf{a}Y$. Our analysis of the difference between ET(Y) and ET(Y') focuses on the unique meta-block at height i of T_i: From Equation (6) 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 Type M meta-block (cf. the right side of Figure 38). Our claim is that prepending a to Y changes the first and the last block of every μ_i ($0 \le i \le k-1$): The prepended a forms a Type 2 meta-block with the first character of X_0 by "stealing" the first character from μ_0, and this character is a $\beta_0 = \mathsf{b}$. Assume that μ_i ($0 \le i \le k-1$) loses its first symbol (i.e., β_i).

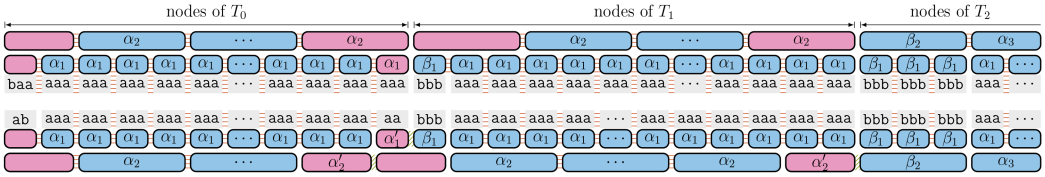


Fig. 39. Excerpt of the ESP trees ET(Y) (*top*) and ET(Y') (*bottom*, vertically flipped), where $Y = \mathsf{ba}^{3^k-1}\mathsf{b}^3\mathsf{a}^{3^k-3}\mathsf{b}^9\mathsf{a}^{3^k-9}\cdots$ and $Y' = \mathsf{a}Y$ (defined in Theorem A.4). The two trees differ in the nodes that are highlighted in magenta ( ). Note that right of the rightmost α'_2 (bottom tree, rightmost magenta node) is the node β_2, and both nodes form a Type 2 meta-block.

By relinquishing this symbol, μ_i becomes a Type 1 meta-block consisting only of α_i's. The last two α_i's contained in μ_i are grouped into a block α'_{i+1} of length *two*, where $\alpha'_{i+1} := \langle \alpha_i\alpha_i\rangle_1[1]$ is the name of the root node of ET($\alpha_i\alpha_i$). Every newly appearing node α'_{i+1} gets combined with its right-adjacent node β_{i+1} to form a new Type 2 meta-block. The used β_{i+1} is stolen from μ_{i+1}, and hence, we observe an iterative process of stealing the first symbol β_{i+1} from μ_{i+1} for each height $i = 0, \ldots, k-2$. Figure 39 visualizes this observation on the lowest two heights.

This observation can be inductively proven for each even integer i with $0 \le i \le h-2$. By Equation (6), we know that $\langle X_i\rangle_i = \beta_i\alpha_i^{3^{k-i}-1}$ and $\langle X_{i+1}\rangle_i = \beta_i^3\alpha_i^{3^{k-i}-3}$. Then

$$\begin{aligned}
\mathrm{esp}(\mathrm{esp}(\langle X_i\rangle_i\langle X_{i+1}\rangle_i)) &= \mathrm{esp}\Big(\mathrm{esp}\Big(\beta_i\alpha_i^{3^{k-i}-1}\beta_i^3\alpha_i^{3^{k-i}-3}\Big)\Big)\\
&\overset{\text{(Corollary A.3)}}{=} \mathrm{esp}\Big(\mathrm{esp}(\beta_i\alpha_i\alpha_i)\,\mathrm{esp}\Big(\alpha_i^{3^{k-i}-3}\Big)\,\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&= \mathrm{esp}\Big(\mathrm{esp}(\beta_i\alpha_i\alpha_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&\overset{\text{(Corollary A.3)}}{=} \mathrm{esp}\Big(\mathrm{esp}(\beta_i\alpha_i\alpha_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\Big)\,\mathrm{esp}\Big(\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&= \mathrm{esp}\Big(\mathrm{esp}(\beta_i\alpha_i\alpha_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\Big)\,\mathrm{esp}(\beta_{i+1}\alpha_{i+1}\alpha_{i+1})\\
&\quad\ \mathrm{esp}\Big(\alpha_{i+1}^{3^{k-i-1}-3}\Big),
\end{aligned}$$

and $\mathrm{esp}(\alpha_{i+1}^{3^{k-i-1}-3}) = \alpha_{i+2}^{3^{k-i-2}-1}$. Prepending α'_i (set $\alpha'_0 := \mathsf{a}$) to the string $\langle X_i\rangle_i\langle X_{i+1}\rangle_i$ yields

$$\begin{aligned}
\mathrm{esp}(\mathrm{esp}(\alpha'_i\langle X_i\rangle_i\langle X_{i+1}\rangle_i)) &= \mathrm{esp}\Big(\mathrm{esp}\Big(\alpha'_i\beta_i\alpha_i^{3^{k-i}-1}\beta_i^3\alpha_i^{3^{k-i}-3}\Big)\Big)\\
&\overset{\text{(Corollary A.3)}}{=} \mathrm{esp}\Big(\mathrm{esp}(\alpha'_i\beta_i)\,\mathrm{esp}\Big(\alpha_i^{3^{k-i}-1}\Big)\,\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&= \mathrm{esp}\Big(\mathrm{esp}(\alpha'_i\beta_i)\,\mathrm{esp}\Big(\alpha_i^{3^{k-i}-3}\alpha_i\alpha_i\Big)\,\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&= \mathrm{esp}\Big(\mathrm{esp}(\alpha'_i\beta_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\alpha'_{i+1}\beta_{i+1}\alpha_{i+1}^{3^{k-i-1}-1}\Big)\\
&\overset{\text{(Corollary A.3)}}{=} \mathrm{esp}\Big(\mathrm{esp}(\alpha'_i\beta_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\Big)\,\mathrm{esp}(\alpha'_{i+1}\beta_{i+1})\\
&\quad\ \mathrm{esp}\Big(\alpha_{i+1}^{3^{k-i-1}-3}\alpha_{i+1}\alpha_{i+1}\Big)\\
&= \mathrm{esp}\Big(\mathrm{esp}(\alpha'_i\beta_i)\,\alpha_{i+1}^{3^{k-i-1}-1}\Big)\,\mathrm{esp}(\alpha'_{i+1}\beta_{i+1})\\
&\quad\ \alpha_{i+2}^{3^{k-i-2}-1}\alpha'_{i+2},
\end{aligned}$$

and α'_{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 Corollary A.3.

Overall, the leftmost and rightmost node on height $i+1$ of T_i changes, for $i = 0, \ldots, k-1$. Such a changed node v of T_i on height $i+1$ has $k-i-1$ ancestors in T_i, which become changed by changing the name of v. Therefore, at least $k-1+(k-2)+(k-3)+\cdots+1 = (k^2-k)/2$ nodes are changed. Hence, there is a lower bound of $\Omega(k^2) = \Omega(\log_3^2(n/\log_3 n)) = \Omega(\lg^2 n)$ changed nodes. □

REFERENCES

[1] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2000. Pattern matching in dynamic texts. In *Proceedings of the SODA*. 819–828.

[2] Arne Andersson and Stefan Nilsson. 1994. A new efficient radix sort. In *Proceedings of the FOCS*. 714–721.

[3] Hideo Bannai. 2016. Grammar compression. In *Encyclopedia of Algorithms*. Springer, 861–866.

[4] Jon Louis Bentley and Robert Sedgewick. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the SODA*. 360–369.

[5] Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. 2016. Sparse text indexing in small space. *ACM Trans. Algor.* 12, 3 (2016), 39:1–39:19.

[6] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. 2015. Longest common extensions in sublinear space. In *Proceedings of the CPM (LNCS)*, Vol. 9133. 65–76.

[7] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. 2014. Time-space trade-offs for longest common extensions. *J. Disc. Algor.* 25 (2014), 42–50.

[8] Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. 2014. Selection and sorting in the "restore" model. In *Proceedings of the SODA*. 995–1004.

[9] Charles J. Colbourn and Alan C. H. Ling. 2000. Quorums from difference covers. *Inf. Proc. Lett.* 75, 1–2 (2000), 9–12.

[10] Graham Cormode and S. Muthukrishnan. 2005. Substring compression problems. In *Proceedings of the SODA*. 321–330.

[11] Graham Cormode and S. Muthukrishnan. 2007. The string edit distance matching problem with moves. *ACM Trans. Algor.* 3, 1 (2007), 2:1–2:19.

[12] Johannes Fischer, Tomohiro I, and Dominik Köppl. 2016. Deterministic sparse suffix sorting on rewritable texts. In *Proceedings of the LATIN (LNCS)*, Vol. 9644. 483–496.

[13] Gianni Franceschini and Roberto Grossi. 2008. No sorting? Better searching! *ACM Trans. Algor.* 4, 1 (2008), 2:1–2:13.

[14] Shouhei Fukunaga, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. 2016. Online grammar compression for frequent pattern discovery. In *Proceedings of the International Conference on Grammatical Inference (Workshop and Conference Proceedings)*, Vol. 57. 93–104.

[15] Michal Ganczorz, Pawel Gawrychowski, Artur Jez, and Tomasz Kociumaka. 2018. Edit distance with block operations. In *Proceedings of the ESA (LIPIcs)*, Vol. 112. 33:1–33:14.

[16] Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. 2018. Optimal dynamic strings. In *Proceedings of the SODA*. 1509–1528.

[17] Pawel Gawrychowski and Tomasz Kociumaka. 2017. Sparse suffix tree construction in optimal time and space. In *Proceedings of the SODA*. 425–439.

[18] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. 1987. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the STOC*. 315–324.

[19] Keisuke Goto. 2019. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proceedings of the PSC*. 111–125.

[20] Tomohiro I. 2017. Longest common extensions with recompression. In *Proceedings of the CPM (LIPIcs)*, Vol. 78. 18:1–18:15.

[21] Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. 2014. Faster sparse suffix sorting. In *Proceedings of the STACS (LIPIcs)*, Vol. 25. 386–396.

[22] Robert W. Irving and Lorna Love. 2003. The suffix binary search tree and suffix AVL tree. *J. Disc. Algor.* 1, 5–6 (2003), 387–408.

[23] Guy Joseph Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the FOCS*. IEEE Computer Society, 549–554.

[24] Juha Kärkkäinen and Dominik Kempa. 2016. LCP array construction using $O(sort(n))$ (or less) I/Os. In *Proceedings of the SPIRE (LNCS)*, Vol. 9954. 204–217.

[25] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936.

[26] Juha Kärkkäinen and Esko Ukkonen. 1996. Sparse suffix trees. In *Proceedings of the COCOON (LNCS)*, Vol. 1090. 219–230.

[27] Zia Khan, Joshua S. Bloom, Leonid Kruglyak, and Mona Singh. 2009. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics* 25, 13 (2009), 1609–1616.
[28] Roman Kolpakov, Gregory Kucherov, and Tatiana A. Starikovskaya. 2011. Pattern matching on sparse suffix trees. In *Proceedings of the Data Compression, Communications and Processing*. 92–97.
[29] Zhize Li, Jian Li, and Hongwei Huo. 2016. Optimal in-place suffix sorting. *ArXiv* abs/1610.08305 (2016).
[30] Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. 2013. ESP-index: A compressed index based on edit-sensitive parsing. *J. Disc. Algor.* 18 (2013), 100–112.
[31] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. 1994. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *Proceedings of the SODA*. 213–222.
[32] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2016. Fully dynamic data structure for LCE queries in compressed space. In *Proceedings of the MFCS (LIPIcs)*, Vol. 58. 72:1–72:15.
[33] Ge Nong, Sen Zhang, and Wai Hong Chan. 2011. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.* 60, 10 (2011), 1471–1484.
[34] Rasmus Pagh. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comp.* 31, 2 (2001), 353–363.
[35] Nicola Prezza. 2018. In-place sparse suffix sorting. In *Proceedings of the SODA*. 1496–1508.
[36] Simon John Puglisi, William F. Smyth, and Andrew Turpin. 2007. A taxonomy of suffix array construction algorithms. *Comput. Surv.* 39, 2 (2007), 1–31.
[37] Naila Rahman and Rajeev Raman. 2008. Rank and select operations on binary strings. In *Encyclopedia of Algorithms*. Springer, 748–751.
[38] Sühleyman Cenk Sahinalp and Uzi Vishkin. 1994. Symmetry breaking for suffix tree construction. In *Proceedings of the STOC*. 300–309.
[39] Hiroshi Sakamoto, Shirou Maruyama, Takuya Kida, and Shinichi Shimozono. 2009. A space-saving approximation algorithm for grammar-based compression. *IEICE Trans.* 92-D, 2 (2009), 158–165.
[40] Yoshimasa Takabatake, Kenta Nakashima, Tetsuji Kuboyama, Yasuo Tabei, and Hiroshi Sakamoto. 2016. siEDM: An efficient string index and search algorithm for edit distance with moves. *Algorithms* 9, 2 (2016), 26:1–26:18.
[41] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. 2014. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proceedings of the SEA (LNCS)*, Vol. 8504. 338–350.
[42] Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. 2016. Deterministic sub-linear space LCE data structures with efficient construction. In *Proceedings of the CPM (LIPIcs)*, Vol. 54. 1:1–1:10.
[43] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. 2017. Small-space LCE data structure with constant-time queries. In *Proceedings of the MFCS (LIPIcs)*, Vol. 83. 10:1–10:15.
[44] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. 2013. essaMEM: Finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics* 29, 6 (2013), 802–804.

Received March 2018; revised February 2020; accepted May 2020