Deterministic Sparse Suffix Sorting on Rewritable Texts

Johannes Fischer, Tomohiro I, and Dominik Köppl

Department of Computer Science, TU Dortmund, Germany
(johannes.fischer, tomohiro.i)@cs.tu-dortmund.de,
dominik.koeppl@tu-dortmund.de

Abstract. Given a rewritable text T of length n on an alphabet of size σ, we propose an online algorithm computing the sparse suffix array and the sparse longest common prefix array of T in $\mathcal{O}\big(c\sqrt{\lg n} + m \lg m \lg n \lg^* n)$ time by using the text space and $\mathcal{O}(m)$ additional working space, where $m \leq n$ is the number of suffixes to be sorted (provided online and arbitrarily), and $c \geq m$ is the number of characters that must be compared for distinguishing the designated suffixes.

1 Introduction

Sorting suffixes of a long text lexicographically is an important first step for many text processing algorithms [15]. The complexity of the problem is quite well understood, as for integer alphabets suffix sorting can be done in optimal linear time [10], and also almost in-place [14]. In this article, we consider a variant of the problem: instead of computing the order of *every* suffix, we address the ***sparse suffix sorting problem***. Given a text $T[1..n]$ of length n and a set $\mathcal{P} \subseteq [1..n]$ of m arbitrary positions in T, the problem asks for the (lexicographic) order of the suffixes starting at the positions in $\mathcal{P}$. The answer is encoded by a permutation of $\mathcal{P}$, which is called the ***sparse suffix array (SSA)*** of T (with respect to $\mathcal{P}$).

Like the "full" suffix arrays, we can enhance $\mathsf{SSA}(T, \mathcal{P})$ by the length of the longest common prefix (LCP) between adjacent suffixes in $\mathsf{SSA}(T, \mathcal{P})$, which we call the ***sparse longest common prefix array (SLCP)***. In combination, $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$ 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 $\mathcal{P}$. This allows us to use the SSA as an efficient index for pattern matching, for example.

Based on classic suffix array construction algorithms [10, 14], sparse suffix sorting is easily conducted in $\mathcal{O}(n)$ time if $\mathcal{O}(n)$ additional working space is available. For $m = \mathrm{o}(n)$, however, the needed working space may be too large, compared to the final space requirement of $\mathsf{SSA}(T)$. Although some special choices of $\mathcal{P}$ admit space-optimal $\mathcal{O}(m)$ construction algorithms (see [2]), the problem of sorting arbitrary choices of suffixes in small space seems to be much harder. We are aware of the following results: As a deterministic algorithm, Kärkkäinen

et al. [10] gave a trade-off using $\mathcal{O}(\tau m + n\sqrt{\tau})$ time and $\mathcal{O}(m + n/\sqrt{\tau})$ working space with a parameter $\tau \in [1..\sqrt{n}]$. If randomization is allowed, there is a technique based on Karp-Rabin fingerprints, first proposed by Bille et al. [2] and later improved by I et al. [8]. The latest one works in $\mathcal{O}(n \lg n)$ expected time and $\mathcal{O}(m)$ additional space.

1.1 Computational Model

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

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Here, characters use $\lceil \log \sigma \rceil$ bits, where σ is the alphabet size; hence, $\lg_\sigma n$ characters can be packed into one word. Comparing two strings X and Y therefore takes $\mathcal{O}(lcp(X,Y)/\lg_\sigma n)$ time, where $lcp(X,Y)$ denotes the length of the longest common prefix of X and Y.

1.2 Algorithm Outline and Our Results

Our main algorithmic idea is to insert the suffixes starting at positions of $\mathcal{P}$ into a self-balancing binary search tree [9]; since each insertion invokes $\mathcal{O}(\lg m)$ suffix-to-suffix comparisons, the time complexity is $\mathcal{O}(t_S m \lg m)$, where t_S is the cost for each suffix-to-suffix comparison. If all suffix-to-suffix comparisons are conducted by naively comparing the characters, the resulting worst case time complexity is $\mathcal{O}(nm \lg m)$. In order to speed this up, our algorithm identifies large identical substrings at different positions during different suffix-to-suffix comparisons. Instead of performing naive comparisons on identical parts over and over again, we build a data structure (stored in redundant text space) that will be used 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 focus on a data structure called ***edit sensitive parsing (ESP) tree*** [5]. The ESP tree supports ***longest common extension (LCE)*** queries. An LCE query on an ESP tree asks for the length of the longest common prefix of two suffixes of the string on which the tree is built. Besides answering LCE queries, ESP trees are *mergeable*, allowing us to build a dynamically growing LCE index on substrings read in the process of the sparse suffix sorting. Consequently, comparing two already indexed substrings is done by a single LCE query.

In their plain form, ESP trees need more space than the text itself; to overcome this space problem, we devise a *truncated* version of the ESP tree, yielding a trade-off parameter between space consumption and LCE query time. By choosing this parameter appropriately, the truncated ESP tree fits into the text space.

However, the need for merging still causes a problem due to the fact that leaves of an ESP tree point to substrings of the text. Although we can prohibit overwriting those referred substrings, a merging may create a new leaf whose substring is already overwritten by the in-text construction of a different ESP tree. To cope with this situation, we propose a new variant of ESP, called ***hierarchical stable parsing (HSP)***, allowing us to quickly find a surrogate substring. With a text space management specialized on the properties of the HSP, we achieve the result of Theorem 1 below.

We make the following definition that allows us to analyze the running time more accurately. Define $\mathcal{C} := \bigcup_{p,p' \in \mathcal{P}, p \neq p'} [p..p + lcp(T[p..], T[p'..])]$ as the set of positions that must be compared for distinguishing the suffixes from $\mathcal{P}$. Then sparse suffix sorting is trivially lower bounded by $\Omega(|\mathcal{C}| / \lg_\sigma n)$ time.

We now state the main result of this article as follows:

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

1.3 Relationship Between Suffix Sorting and LCE Queries

The LCE-problem is to preprocess a text T such that subsequent LCE-queries $lce(i,j) := lcp(T[i..], T[j..])$ giving the length of the longest common prefix of the suffixes starting at positions i and j can be answered efficiently. Data structures for LCE and sparse suffix sorting are closely related, as shown in the following observation:

Observation 1. *Given a data structure that computes LCE in $\mathcal{O}(\tau)$ time for $\tau > 0$, we can compute sparse suffix sorting for m positions in $\mathcal{O}(\tau m \lg m)$ time by inserting suffixes in a balanced binary search tree.*

Conversely, given an algorithm computing the SSA and the SLCP of a text T of length n for m positions in $\mathcal{O}(m)$ space and $\mathcal{O}(f(n,m))$ time for some f, we can construct a data structure in $\mathcal{O}(f(n,m))$ time and $\mathcal{O}(m)$ space, answering LCE queries on T in $\mathcal{O}\big(n^2/m^2\big)$ time [4], (using a difference cover sampling modulo n/m [10]).

The currently best deterministic data structure for LCE we are aware of is due to Bille et al. [3], using $\mathcal{O}(n/\tau)$ space and answering LCE queries in $\mathcal{O}(\tau)$ time, for any $1 \leq \tau \leq n$. However, this data structure has a preprocessing time of $\Omega\big(n^2\big)$, and is thus not helpful for sparse suffix sorting. We develop a new data structure for LCE with the following properties.

Theorem 2. *There is a data structure using $\mathcal{O}(n/\tau)$ space that answers LCE queries in $\mathcal{O}\big(\lg^* n \left(\lg (n/\tau) + \tau^{\lg 3} / \lg_\sigma n\right)\big)$ time, where $1 \leq \tau \leq n$. We can build the data structure in $\mathcal{O}(n (\lg^* n + (\lg n)/\tau + (\lg \tau)/\lg_\sigma n))$ time with additional $\mathcal{O}\big(\tau^{\lg 3} \lg^* n\big)$ words during construction.*

An advantage of our data structure against the deterministic data structures in [3] is its faster construction time, which is upper bounded by $\mathcal{O}(n \lg n)$.

1.4 Outline of this Article

The first part of the paper (Section 2) is dedicated to answering LCE queries (Theorem 2) with the (truncated) ESP tree. In Section 3 we describe our algorithm for the sparse suffix sorting problem with the abstract data type dynLCE that supports LCE queries and a merging operation. In Section 4 we study how the text space can be exploited to lower the memory footprint. To this end, we develop (truncated) HSP trees. By the properties of the HSP tree, we finally solve the sparse suffix sorting problem (Theorem 1) in the claimed time and space.

1.5 Preliminaries

Let Σ be an ordered alphabet of size σ. We assume that a character in Σ is represented by an integer. For a string $X \in \Sigma^*$, let $|X|$ denote the length of X. For a position i in X, let $X[i]$ denote the i-th character of X. For positions i and j, let $X[i..j] = X[i]X[i+1] \cdots X[j]$. For $W = XYZ$ with $X, Y, Z \in \Sigma^*$, we call X, Y and Z a prefix, substring, suffix of W, respectively. In particular, the suffix beginning at position i is denoted by $W[i..]$.

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 $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ to denote the beginning and end of $\mathcal{I}$; i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$.

2 Answering LCE queries with ESP Trees

Edit sensitive parsing (ESP) and ESP trees were proposed by Cormode and Muthukrishnan [5] to approximate the edit distance with moves efficiently. A similar technique is *signature encoding* [12]. Based on signature encoding, Alstrup et al. [1] and Nishimoto et al. [13] derive new data structures for supporting LCE queries. For several reasons, these data structures cannot be used in our context; we therefore show in this section that ESP trees can also be used to answer LCE queries.

2.1 Edit Sensitive Parsing

The aim of the ESP technique is to decompose a string $Y \in \Sigma^*$ into substrings of length 2 or 3 such that each substring of this decomposition is determined uniquely by its neighboring characters. To this end, it first identifies so-called ***meta-blocks*** in Y, and then further refines these meta-blocks into ***blocks*** of length 2 or 3.

The meta-blocks are created in the following 3-stage process:

(1) Identify maximal regions of repeated symbols (i.e., maximal substrings of the form c^ℓ for $c \in \Sigma$ and $\ell \geq 2$). Such substrings form the `type 1` meta-blocks.

(2) Identify remaining substrings of length at least 2 (which must lie between two `type 1` meta-blocks). Such substrings form the `type 2` meta-blocks.
(3) Any 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 $Y[i]$ are fused with the `type 1` meta-block to their right[1], or, if $Y[i]$ is the last character in Y, with the `type 1` meta-block to its left. The meta-blocks emerging from this are called `type M` (mixed).

Meta-blocks of `type 1` and `type M` are collectively called ***repeating meta-blocks***.

Although meta-blocks are defined by the comprising characters, we treat them as intervals on the text range.

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 2 or 3; otherwise create two blocks, each containing two characters.

Type-2 meta-blocks. A `type 2` meta-block μ is processed in $\mathcal{O}(|\mu| \lg^* \sigma)$ time by a technique called *alphabet reduction* [5]. The first $\lg^* \sigma$ characters are blocked in the same way as repeating meta-blocks. Any remaining block β is formed such that β's interval boundaries are determined by $Y[\max(\mathsf{b}(\beta) - \Delta_{\mathrm{L}}, \mathsf{b}(\mu)) .. \min(\mathsf{e}(\beta) + \Delta_{\mathrm{R}}, \mathsf{e}(\mu))]$, where $\Delta_{\mathrm{L}} := \lceil \lg^* \sigma \rceil + 5$ and $\Delta_{\mathrm{R}} := 5$ (see [5, Lemma 8]).

We call the substring $Y[\mathsf{b}(\beta) - \Delta_{\mathrm{L}} .. \mathsf{e}(\beta) + \Delta_{\mathrm{R}}]$ the ***local surrounding*** of β, if it exists. Blocks whose local surroundings exist are also called ***surrounded***.

Let $\tilde{\Sigma} \subseteq \Sigma^2 \cup \Sigma^3$ denote the set of blocks resulting from ESP (the "new alphabet"). We use $esp \colon \Sigma^* \to \tilde{\Sigma}^*$ to denote the function that parses a string by ESP and returns a string in $\tilde{\Sigma}^*$.

2.2 Edit Sensitive Parsing Trees

Applying *esp* recursively on its output generates a context free grammar (CFG) as follows. Let $Y_0 := Y$ be a string on an alphabet $\Sigma_0 := \Sigma$ with $\sigma_0 = |\Sigma_0|$. The output of $Y_h := esp^h(Y) = esp(esp^{h-1}(Y))$ is a sequence of blocks, which belong to a new alphabet Σ_h ($h > 0$). A block $b \in \Sigma_h$ contains a string $b \in \Sigma_{h-1}^*$ of length two or three. Since each application of *esp* reduces the string length by at least 1/2, there is a $k = \mathcal{O}(\lg |Y|)$ such that $esp(Y_k)$ returns a single block τ. We write $\mathcal{V} := \bigcup_{1 \le h \le k} \Sigma_h$ for the set of all blocks in $Y_1, Y_2, \ldots, Y_k$.

We use a (deterministic) dictionary $\mathfrak{D} : \Sigma_h \to \Sigma_{h-1}^2 \cup \Sigma_{h-1}^3$ to map a block to its characters, for each $1 \le h \le k$. The dictionary entries are of the form $b \to xy$ or $b \to xyz$, where $b \in \Sigma_h$ and $x, y, z \in \Sigma_{h-1}$. The CFG for Y is represented by

[1] The original version prefers the left meta-block, but we change it for a more stable behavior

the non-terminals $\mathcal{V}$, the terminals Σ_0, the dictionary $\mathfrak{D}$, and the start symbol τ. This grammar exactly derives Y.

Our representation differs from that of Cormode and Muthukrishnan [5] because it does not use hash tables.

Definition 1. *The* **ESP tree** $\mathsf{ET}(Y)$ *of a string* Y *is a slightly modified derivation tree of the CFG defined above.The internal nodes are elements of* $\mathcal{V} \setminus \Sigma_1$*, and the leaves are from* Σ_1*. Each leaf refers to a substring in* Σ_0^2 *or* Σ_0^3*. Its root node is the start symbol* τ*.*

For convenience, we count the height of nodes from 1, so that the sequence of nodes on height h, denoted by $\langle Y\rangle_h$, is corresponding to Y_h. The ***generated substring*** of a node $\langle Y\rangle_h[i]$ is the substring of Y generated by the symbol $Y_h[i]$ (applying $\mathfrak{D}$ recursively on $Y_h[i]$). Each node v represents a block that is contained in a meta-block μ, for which we say that μ ***builds*** v. More precisely, a node $v := \langle Y\rangle_h[i]$ is said to be built on a meta-block represented by $\langle Y\rangle_{h-1}[b..e]$ iff $\langle Y\rangle_{h-1}[b..e]$ contains the children of v. Like with blocks, nodes inherit the type of the meta-block on which they are built.

Surrounded Nodes. A leaf is called surrounded iff its representing block on text-level is surrounded. Given an internal node v on height $h+1$ ($h \geq 1$) whose children are $\langle Y\rangle_h[\beta]$, we say that v is ***surrounded*** iff the nodes $\langle Y\rangle_h[\mathsf{b}(\beta) - \Delta_{\mathrm{L}}..\mathsf{e}(\beta) + \Delta_{\mathrm{R}}]$ are surrounded.

2.3 Tree Representation

We store the ESP tree as a CFG. Every non-terminal is represented by a ***name***. The name is a pointer to a data-field, which is composed differently for leaves and internal nodes:

Leaves. A leaf stores a position i and a length $l \in \{2, 3\}$ such that $Y[i..i+l-1]$ 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 `0` for the non-existing third child such that all data fields are of the same length.

This representation allows us to navigate top-down in the ESP tree by traversing the tree from the root, in time linear in the height of the tree.

We keep the invariant that the roots of *isomorphic* subtrees have the *same* names. In other words, before creating a new name for the rule $b \to xyz$, we have to check whether there already exists a name for xyz. To perform this look-up efficiently, we need also the *reverse* dictionary of $\mathfrak{D}$, with the right hand side of the rules as search keys. We use a dictionary of size $\mathcal{O}(|Y|)$, supporting lookup and insert in $\mathcal{O}(t_\lambda)$ time.

More precisely, we assume there is a dictionary data structure, storing n elements in $\mathcal{O}(n)$ space, supporting lookup and insert in $\mathcal{O}(t_\lambda + l/\lg_\sigma n)$ time for a key of length l, where $t_\lambda = t_\lambda(n)$ depends on n. For instance, Franceschini and Grossi's data structure [7] with word-packing supports $t_\lambda = \mathcal{O}(\lg n)$.

Lemma 1. *An ESP tree of a string of length n can be built in $\mathcal{O}(n(\lg^* n + t_\lambda))$ time. It consumes $\mathcal{O}(n)$ space.*

2.4 LCE Queries in ESP Trees

ESP trees are fairly stable against edit operations: The number of nodes that are differently parsed after prepending or appending a string to the input is upper bounded by $\mathcal{O}(\lg n \lg^* n)$ [5, Lemma 11]. To use this property in our context of LCE queries, we consider nodes of $\mathsf{ET}(Y)$ that are still present in $\mathsf{ET}(XYZ)$; a node v in $\mathsf{ET}(Y)$ generating $Y[i_0..j_0]$ is said to be ***stable*** iff, for *all* strings X and Z, there exists a node in $\mathsf{ET}(XYZ)$ that has the same name as v and generates $(XYZ)[|X| + i_0..|X| + j_0]$. We also consider repeating nodes that are present with slight shifts; a non-stable repeating node v in $\mathsf{ET}(Y)$ generating $Y[i_0..j_0]$ is said to be ***semi-stable*** iff, for *all* strings X and Z, there exists a node in $\mathsf{ET}(XYZ)$ that has the same name as v and generates a substring intersecting with $(XYZ)[|X|+i_0..|X|+j_0]$. Then, the proof of Lemma 9 of [5] says that, for each height, $\mathsf{ET}(Y)$ contains $\mathcal{O}(\lg^* n)$ nodes that are not (semi-)stable, which we call ***fragile***. Since the children of the (semi-)stable nodes are also (semi-)stable, there is a border in $\mathsf{ET}(Y)$ separating the (semi-)stable nodes from the fragile ones.

In order to use semi-stable nodes to answer LCE queries efficiently, we let each node have an additional property, called ***surname***. A node $v := \langle Y \rangle_h[i]$ is said to be ***repetitive*** iff there exists $\langle Y \rangle_{h'}[\mathcal{I}]$ at some height $h' < h$ with $Y_{h'}[\mathcal{I}] = d^{|\mathcal{I}|}$, where $\langle Y \rangle_{h'}[\mathcal{I}]$ is the sequence of nodes on height h' in the subtree rooted at $\langle Y \rangle_h[i]$ and $d \in \Sigma_{h'}$. The surname of a repetitive node $v := \langle Y \rangle_h[i]$ is the name of a highest non-repetitive node in the subtree rooted at v. The surname of a non-repetitive node is the name of the node itself. It is easy to compute and store the surnames while constructing ESP trees.

The connection between semi-stable nodes and the surnames is based on the fact that a semi-stable node is repetitive: Let u be the node whose name is the surname of a semi-stable node v. If u is on height h, v's subtree consists of a repeat of u's on height h. A shift of v can only be caused by adding u's. So the shift is always a multiple of the length of the generated substring of u.

We now state a lemma that shows how ESP trees can be used for LCE queries; the proof (as well as all other missing proofs) can be found in the full version [6].

Lemma 2. *Let X and Y be strings with $|X| \leq |Y| \leq n$. Given $\mathsf{ET}(X)$ and $\mathsf{ET}(Y)$ built with the* same *dictionary and two text-positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $l := lcp(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg |Y| + \lg l \lg^* n)$ time.*

2.5 Truncated ESP Trees

Building an ESP tree over a string Y requires $\mathcal{O}(|Y|)$ words of space, which might be too much in some scenarios. Our idea is to truncate the ESP tree at some fixed height, discarding the nodes in the lower part. The truncated version stores

just the upper part, while its (new) leaves refer to (possibly long) substrings of Y. The resulting tree is called the ***truncated ET (tET)***. More precisely, we define a height η and delete all nodes at height less than η, which we call ***lower nodes***. A node higher than η is called an ***upper node***. The nodes at height η form the new leaves and are called ***η-nodes***. Similar to the former leaves, their names are pointers to their generated substrings appearing in Y. Remembering that each internal node has two or three children, an η-node generates a string of length at least 2^η and at most 3^η. So the maximum number of nodes in a truncated ESP tree of a string of length n is $n/2^\eta$.

Similar to leaves, we use the generated substring X of an η-node v for storing and looking up v: It can be looked up or inserted in $\mathcal{O}(|X| / \lg_\sigma n + t_\lambda)$ time.

Lemma 3. *We can build a truncated ESP tree of a string Y of length n in $\mathcal{O}(n(\lg^* n + \eta/\lg_\sigma n + t_\lambda/2^\eta)$ time, using $\mathcal{O}(3^\eta \lg^* n)$ words of working space. The tree consumes $\mathcal{O}(n/2^\eta)$ space.*

Lemma 4. *Let X and Y be strings with $|X|, |Y| \leq n$. Given $\mathsf{ET}(X)$ and $\mathsf{ET}(Y)$ built with the* same *dictionary and two text-positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $lcp(X[i_X..], Y[i_Y..])$ in $\mathcal{O}(\lg^* n(\lg(n/2^\eta) + 3^\eta/\lg_\sigma n))$ time.*

With $\tau := 2^\eta$ we get Theorem 2.

3 Sparse Suffix Sorting

Borrowing the technique of Irving and Love [9], an AVL tree on a set of strings $\mathcal{S}$ can be augmented with LCP values so that we can compute $l := \max\{lcp(X,Y) \mid X \in \mathcal{S}\}$ for a string Y in $\mathcal{O}(l/\lg_\sigma n + \lg|\mathcal{S}|)$ time. Inserting a new string into the tree is supported in the same time complexity. Irving and Love [9] called this data structure the ***suffix AVL tree*** on $\mathcal{S}$; we denote it by $\mathsf{SAVL}(\mathcal{S})$.

Given a text T of length n, we will use $\mathsf{SAVL}(Suf(\mathcal{P}))$ as a representation for $\mathsf{SSA}(T,\mathcal{P})$ and $\mathsf{SLCP}(T,\mathcal{P})$. Our goal is to build $\mathsf{SAVL}(Suf(\mathcal{P}))$ efficiently. However, inserting suffixes naively suffers from the lower bound $\Omega(n|\mathcal{P}| / \lg_\sigma n)$ on time. How to speed up the comparisons by exploiting a data structure for LCE queries is topic of this section.

3.1 Abstract Algorithm

Our idea is that creating a mergeable LCE data structure on the read substrings may be helpful for later queries. We call this abstract data type ***dynamic LCE (dynLCE)***; it supports the following operations:

- $\mathsf{dynLCE}(Y)$ constructs a dynLCE data structure M on a substring Y of T. Let $M.\texttt{text}$ denote the string Y on which M is constructed.
- $\mathsf{LCE}(M_1, M_2, p_1, p_2)$ computes $lcp(M_1.\texttt{text}[p_1..], M_2.\texttt{text}[p_2..])$, where $p_i \in [1..|M_i.\texttt{text}|]$ for $i = 1, 2$.
- $\texttt{merge}(M_1, M_2)$ merges two dynLCEs M_1 and M_2 such that the output is a dynLCE on the concatenation of $M_1.\texttt{text}$ and $M_2.\texttt{text}$.

We use the expression $t_{\mathrm{C}}(|Y|)$ to denote the construction time of such a data structure on a string Y. Further, $t_{\mathrm{L}}(|X|+|Y|)$ and $t_{\mathrm{M}}(|X|+|Y|)$ denote the LCE query time and the time for merging two such data structures on strings X and Y, respectively. Querying a dynLCE built on a string of length ℓ is faster than the word-packed character comparison iff $\ell = \Omega(t_{\mathrm{L}}(\ell) \lg n / \lg \sigma)$. Hence, there is no point in building a dynLCE on a text smaller than $g := \Theta(t_{\mathrm{L}}(g) \lg n / \lg \sigma)$.

We store the text intervals covered by the dynLCEs such that we know the text-positions where querying a dynLCE is possible. Such an interval is called an ***LCE interval***. An LCE interval $\mathcal{I}$ stores a pointer to its dynLCE data structure M, and an integer i such that $M.\mathtt{text}[i..i+|\mathcal{I}|-1] = T[\mathcal{I}]$. The LCE intervals themselves are maintained in a self-balancing binary search tree of size $\mathcal{O}(|\mathcal{P}|)$, storing their starting positions as keys.

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

Locating. The suffix AVL tree performs an LCE computation for each node encountered while locating the insertion point of $\hat{p}$. Assume that the task is to compare the suffixes $T[i..]$ and $T[j..]$ for some $1 \leq i, j \leq |T|$. First check whether the positions i and j are contained in an LCE interval, in $\mathcal{O}(\lg m)$ time. If both positions are covered by LCE intervals, then query the respective *dynLCE*s. Otherwise, look up the position where the next LCE interval starts. Up to that position, naively compare both substrings. Finally, repeat the above check again at the new positions, until finding a mismatch. After locating the insertion point of $\hat{p}$ in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$, we obtain $\bar{p} := \mathrm{mlcparg}_{\hat{p}}$ and $l := \mathrm{mlcp}_{\hat{p}}$ as a byproduct, where $\mathrm{mlcparg}_p := \mathrm{argmax}_{p' \in \mathcal{P}, p \neq p'}\, \mathit{lcp}(T[p..], T[p'..])$ and $\mathrm{mlcp}_p := \mathit{lcp}(T[p..], T[\mathrm{mlcparg}_p..])$ for $1 \leq p \leq |T|$.

Updating. The LCE intervals are updated dynamically, subject to the following constraints:

C1: The length of each LCE interval is at least g.
C2: For every $p \in \mathcal{P}$ the interval $[p..p+\mathrm{mlcp}_p-1]$ is covered by an LCE interval *except at most* g positions at its left and right ends.
C3: There is a gap of at least g positions between every pair of LCE intervals.

These constraints guarantee that there is at most one LCE interval that intersects with $[p..p+\mathrm{mlcp}_p-1]$ for a $p \in \mathcal{P}$.

The following instructions will satisfy the constraints: If $l < g$, we do nothing. Otherwise, we have to care about C2. Fortunately, there is at most one position in $\mathcal{P}$ that possibly invalidates C2 after adding $\hat{p}$, and this is $\bar{p}$; otherwise, by transitivity, we would have created some larger LCE interval previously. Let $U \subset [1..n]$ be the positions that belong to an LCE interval. The set $[\hat{p}..\hat{p}+l-1] \setminus U$ can be represented as a set of disjoint intervals of maximal length. For each interval $\mathcal{I} := [\hat{p}+i..\hat{p}+j] \subset [\hat{p}..\hat{p}+l-1]$ of that set (for some $0 \leq i \leq j < l$), apply the following rules with $\mathcal{J} := [\bar{p}+i..\bar{p}+j]$ sequentially:

R1: If $\mathcal{J}$ is a sub-interval of an LCE interval, then declare $\mathcal{I}$ as an LCE interval and let it refer to the dynLCE of the larger LCE interval.

- R2: If $\mathcal{J}$ intersects with an LCE interval $\mathcal{K}$, enlarge $\mathcal{K}$ to $\mathcal{K} \cup \mathcal{J}$, enlarging its corresponding dynLCE (We can enlarge an dynLCE by creating a new instance and merge both instances). Apply R1.
- R3: Otherwise, create a dynLCE on $\mathcal{I}$, and make $\mathcal{I}$ to an LCE interval.
- R4: If C3 is violated, then a newly created or enlarged LCE interval is adjacent to another LCE interval. Merge those LCE intervals and their dynLCEs.

We also need to satisfy C2 on $[\bar{p}..\bar{p}+l-1]$. To this end, update U, compute the set of disjoint intervals $[\bar{p}..\bar{p}+l-1] \setminus U$ and apply the same rules on it.

Although we might create some LCE intervals covering less than g characters, we will restore C1 by merging them with a larger LCE interval in R4. In fact, we introduce at most two new LCE intervals. C1 is easily maintained, since we will never shrink an LCE interval.

Lemma 5. *Given a text T of length n that is loaded into RAM, the SSA and SLCP of T for a set of m arbitrary positions can be computed deterministically in $\mathcal{O}(t_{\mathrm{C}}(|\mathcal{C}|) + t_{\mathrm{L}}(|\mathcal{C}|)m \lg m + m t_{\mathrm{M}}(|\mathcal{C}|))$ time.*

3.2 Sparse Suffix Sorting with ESP Trees

We will show that the ESP tree is a suitable data structure for dynLCE. In order to merge two ESP trees, we use a *common* dictionary $\mathfrak{D}$ that is stored *globally*. Fortunately, it is easy to combine two ESP trees by updating just a handful of nodes, which are fragile.

Lemma 6. *Assume that we have already created $\mathsf{ET}(X)$ and $\mathsf{ET}(Y)$ on two strings $X, Y \in \Sigma^*$. Merging both trees into $\mathsf{ET}(XY)$ takes $\mathcal{O}(t_\lambda(\Delta_{\mathrm{L}} \lg |Y| + \Delta_{\mathrm{R}} \lg |X|))$ time.*

The following theorem combines the results of Lemmas 5 and 6.

Theorem 3. *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 $\mathcal{O}(|\mathcal{C}| (\lg^* n + t_\lambda) + m \lg m \lg n \lg^* n)$ time.*

4 Hierarchical Stable Parsing

Remembering the outline in the introduction, the key idea is to solve the limited space problem by storing dynLCEs in text space. Taking two LCE intervals on the text containing the same substring, we overwrite *one* part while marking the *other* part as a reference. By choosing a suitably large η, we can overwrite the text of one LCE interval with a truncated ESP tree (tET) whose η-nodes refer to substrings of the other LCE interval. Merging two tETs involves a reparsing of some η-nodes. Assume that we want to reparse an η-node v, and that its generated substring gets enlarged due to the parsing. We have to locate a substring in the text that contains its new generated substring X. Although we can create a suitably large string containing X by concatenating the generated substrings of

its preceding and succeeding siblings, these η-nodes may point to text intervals that may not be consecutive. Since the name of an η-node is the representation of a *single* substring, we have to search for a substring equal to X in the text. Because this would be too inefficient, we will show a slight modification of the ESP technique that circumvents this problem.

4.1 Hierarchical Stable Parse Trees

Our modification, which we call *hierarchical stable parse trees* or ***HSP trees***, affects only the definition of meta-blocks. The factorization of meta-blocks is done by relaxing the check whether two characters are equal; instead of comparing names we compare by surname.[2] A more detailed study of HTs can be read in the full version of the paper [6].

Lemma 7. *An HSP tree on an interval of length l can be built in $\mathcal{O}(l\,(\lg^* n + t_\lambda))$ time. It consumes $\mathcal{O}(l)$ space.*

The modified parsing allows us to claim the following lemma:

Lemma 8. *If a surrounded node is neither stable nor semi-stable, it can only be changed to a node whose generated substring is a prefix of the generated substring of an already existing node.*

4.2 Sparse Suffix Sorting in Text Space

The ***truncated HSP tree (tHT)*** is the truncated version of the HSP tree. It is defined analogously as the truncated ESP tree (see Section 2.5), with the exception of the surnames: For each repetitive node, we mark whether its surname is the name of an upper node, of an η-node, or of a lower node. Therefore, we need to save the names of certain lower nodes in the reverse dictionary of $\mathfrak{D}$. This is only necessary when an upper node or an η-node v has a surname that is the name of a lower node. If 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 the same surname, too. So the number of lower node entries in the reverse dictionary is upper bounded by the number of η-nodes, and each lower node generates a substring of length less than 3^η. We conclude that the results of Lemma 3 and Lemma 4 apply to the tHT, too.

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

Lemma 9. *With η as defined above, the number of nodes in a truncated HSP tree is bounded by $\mathcal{O}\big(l(\lg\sigma)^{0.7}/(\lg n)^{1.2}\big)$. Further, an η-node generates a substring containing at most $\lceil\alpha(\lg n)^2/(\lg\sigma)\rceil$ characters.*

[2] The check is relaxed since nodes with different surnames cannot have the same name.

Applying Lemma 9 to the results elaborated in Section 2.5 for the tETs yields

Corollary 1. *We can compute a tHT on a substring of length l in $\mathcal{O}(l \lg^* n + t_\lambda l/2^\eta + l \lg\lg n)$ time. The tree takes $\mathcal{O}(l/2^\eta)$ space. We need a working space of $\mathcal{O}(\lg^2 n \lg^* n / \lg \sigma)$ characters.*

Corollary 2. *An LCE query on two tHTs can be answered in $\mathcal{O}(\lg^* n \lg n)$ time.*

We analyze the merging when applied by the sparse suffix sorting algorithm in Section 3.1. Assume that our algorithm found two intervals $[i..i+l-1]$ and $[j..j+l-1]$ with $T[i..i+l-1] = T[j..j+l-1]$. Ideally, we want to construct $\mathsf{tHT}(T[i..i+l-1])$ in the text space $[j..j+l-1]$, leaving $T[i..i+l-1]$ untouched so that parts of this substring can be referenced by the η-nodes. Unfortunately, there are two situations that make the life of a tHT complicated: the need for merging tHTs, and possible overlapping of the intervals $[i..i+l-1]$ and $[j..j+l-1]$.

Partitioning of LCE intervals. In order to merge trees, we have to take special care of those η-nodes that are fragile, because their names may have to be recomputed during a merge. In order to recompute the name of an η-node v, consisting of a pointer and a length, we have to find a substring that consists of v's generated substring and some adjacent characters with respect to the original substring in the text. That is because the parsing may assign a new pointer and a new length to an η-node, possibly enlarging the generated substring, or letting the pointer refer to a different substring.

The name for a surrounded fragile η-nodes v is easily recomputable thanks to Lemma 8: Since the new generated substring of v is a prefix of the generated substring of an already existing η-node w, which is found in the reverse dictionary for η-nodes, we can create a new name for v from the generated substring of w.

Unfortunately, the same approach does not work with the non-surrounded η-nodes. Those nodes have a generated substring that is found on the border area of $T[j..j+l-1]$. If we leave this area untouched, we can use it for creating names of a non-surrounded η-node during a reparsing. Therefore, we mark those parts of the interval $[j..j+l-1]$ as read-only. Conceptually, we partition an LCE interval into subintervals of green and red intervals; we free the text of a ***green interval*** for overwriting, while prohibiting write-access on a ***red interval***. The green intervals are managed in a dynamic, global list. We keep the invariant that

Invariant 1: $f := \lceil 2\alpha \lg^2 n \Delta_{\mathrm{L}} / \lg \sigma \rceil = \Theta(g)$ positions of the left and right ends of each LCE interval are *red*.

This invariant solves the problem for the non-surrounded nodes.

Allocating Space. We can store the upper part of the tHT in a green interval, since $l/2^\eta \lg n \leq l\alpha^{0.6}(\lg \sigma)^{0.7}/(\lg n)^{0.2} = \mathrm{o}(l \lg \sigma)$ holds. By choosing g and α properly, we can always leave $f \lg \sigma / \lg n = \mathcal{O}(\lg^* n \lg n)$ words on a green interval untouched, sufficiently large for the working space needed by Corollary 1. Therefore, we pre-compute α and g based on the input T, and set both as *global* constants. Since the same amount of free space is needed during a later merging when reparsing an η-node, we add the invariant that

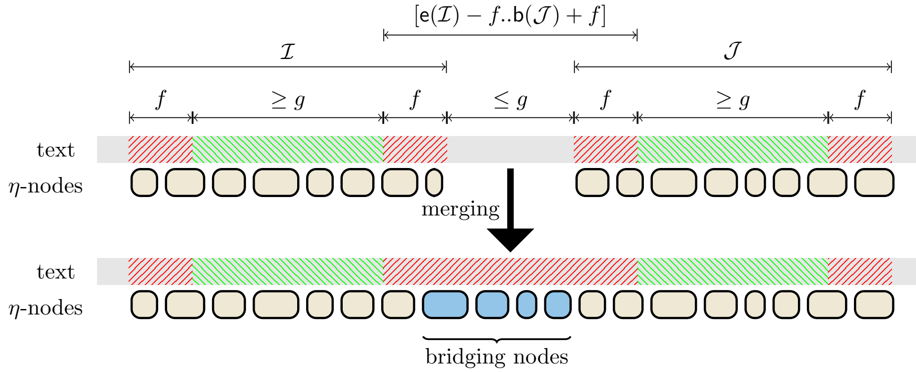

Fig. 1. The merging is performed only if the gap between both trees is less than g. The substring $T[\mathsf{e}(\mathcal{I}) - f..\mathsf{b}(\mathcal{J}) + f]$ is marked red for the sake of the bridging nodes.

Invariant 2: Each LCE interval has $f \lg \sigma / \lg n$ free space left on a green interval.

For the merging, we need a more sophisticated approach that respects both invariants:

Merging. We introduce a merge operation that allows the merge of two tHTs whose LCE intervals have a gap of less than g characters. The merge operation builds new η-nodes on the gap. The η-nodes whose generated substrings intersect with the gap are called ***bridging*** nodes. The bridging nodes have the same problem as the non-surrounded η-nodes, since the gap may be a unique substring of T.

Let $\mathcal{I}$ and $\mathcal{J}$ be two LCE intervals with $0 \leq \mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I}) \leq g$, where on each interval a tHT has been computed. We compute $\mathsf{tHT}(T[\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{J})])$ by merging both trees. By Lemma 6, at most $\mathcal{O}(\Delta_\mathrm{L} + \Delta_\mathrm{R})$ nodes at every height on each tree have to be reprocessed, and some bridging nodes connecting both trees have to be built. Unfortunately, the text may not contain another occurrence of $T[\mathsf{e}(\mathcal{I}) - f..\mathsf{b}(\mathcal{J}) + f]$ such that we could overwrite $T[\mathsf{e}(\mathcal{I}) - f..\mathsf{b}(\mathcal{J}) + f]$. Therefore, we mark this interval as red. So we can use the characters contained in $T[\mathsf{e}(\mathcal{I}) - f..\mathsf{b}(\mathcal{J}) + f]$ for creating the bridging η-nodes, and for modifying the non-surrounded nodes of both trees (Figure 1). Since the gap consists of less than g characters, the bridging nodes need at most $\mathcal{O}(\lg n \lg^* n)$ additional space. By choosing g and α sufficiently large, we can maintain Invariant 2 for the merged LCE interval.

Interval Overlapping. Assume that the LCE intervals $[i..i + l - 1]$ and $[j..j + l - 1]$ overlap, without loss of generality $j > i$. Our goal is to create $\mathsf{tHT}(T[i..i + l - 1])$. First, we compute the smallest period $d \leq j - i$ of $T[i..j + l - 1]$ in $\mathcal{O}(l)$ time [11]. The substring $T[i..i+d+f-1]$ is used as a reference and therefore marked red. Keeping the original characters in $T[i..i+d+f-1]$, we can restore the generated substrings of every η-node by an arithmetic progression. Hence, we can mark the interval $[i + d + f..j + l - 1 - f]$ *green*.

Finally, the time bound for the above merging strategy is given by

Corollary 3. *Given two LCE intervals* $\mathcal{I}$ *and* $\mathcal{J}$ *with* $0 \leq \mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I}) \leq g$*. We can build* $\mathsf{tHT}(T[\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{J})])$ *in* $\mathcal{O}(g \lg^* n + t_\lambda g/2^\eta + g\eta/\lg_\sigma n + t_\lambda \lg^* n \lg n)$ *time.*

It is now easy to modify our sparse suffix sorting algorithm of Section 3.1 for tHT on text space, yielding the result of Theorem 1.

References

[1] Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: SODA. pp. 819–828 (2000)

[2] Bille, P., Fischer, J., Gørtz, I.L., Kopelowitz, T., Sach, B., Vildhøj, H.W.: Sparse suffix tree construction in small space. In: Proc. ICALP. LNCS, vol. 7965, pp. 148–159 (2013)

[3] Bille, P., Gørtz, I., Knudsen, M., Lewenstein, M., Vildhøj, H.: Longest common extensions in sublinear space. In: Combinatorial Pattern Matching, LNCS, vol. 9133, pp. 65–76. Springer (2015)

[4] Bille, P., Gørtz, I., Sach, B., Vildhøj, H.: Time-space trade-offs for longest common extensions. In: Combinatorial Pattern Matching, vol. 7354, pp. 293–305. Springer (2012)

[5] Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. ACM Transactions on Algorithms 3(1) (2007)

[6] Fischer, J., I, T., Köppl, D.: Deterministic Sparse Suffix Sorting on Rewritable Texts. arXiv:1509.07417 (2015)

[7] Franceschini, G., Grossi, R.: No sorting? better searching! In: Foundations of Computer Science. pp. 491–498 (Oct 2004)

[8] I, T., Kärkkäinen, J., Kempa, D.: Faster sparse suffix sorting. In: STACS. pp. 386–396 (2014)

[9] Irving, R.W., Love, L.: The suffix binary search tree and suffix AVL tree. J. Discrete Algorithms 1(5-6), 387–408 (2003)

[10] Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)

[11] Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Foundations of Computer Science. pp. 596–. FOCS (1999)

[12] Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality-tests in polylogarithmic time. In: SODA. pp. 213–222. SIAM (1994)

[13] Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Dynamic index, LZ factorization, and LCE queries in compressed space. ArXiv 1504.06954 (2015)

[14] Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Computers 60(10), 1471–1484 (2011)

[15] Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. 39(2) (2007)