DYNAMIC PATTERN MATCHING[1]

STEPHEN ALSTRUP[2]

GERTH STØLTING BRODAL[3]

THEIS RAUHE[4]

ABSTRACT. Pattern matching is the problem of finding all occurrences of a pattern in a text. For a long period of time significant progress has been made in solving increasingly more generalized and dynamic versions of this problem. In this paper we introduce a fully dynamic generalization of the pattern matching problem. We show how to maintain a family of strings under split and concatenation operations. Given a string in the family, all occurrences of it in the family are reported within time $\mathcal{O}(\log n \log\log n + occ)$ time, where n is the total size of the strings and *occ* is the number of occurrences. Updates are performed in $\mathcal{O}(\log^2 n \log\log n \log^* n)$ time. These bounds are competitive or improve former results for less generalized versions of the problem. As an intermediate result of independent interest, we provide an almost quadratic improvement of the time bounds for the *dynamic string equality* problem due to Mehlhorn, Sundar and Uhrig.

[1]Partially supported by the ESPRIT Long Term Research Program of the EU under contract 20244 (project ALCOM-IT).

[2]Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark. E-mail: stephen@diku.dk.

[3]**BRICS**, Basic Research in Computer Science, Centre of the Danish National Research Foundation. Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: gerth@brics.dk.

[4]**BRICS**, Basic Research in Computer Science, Centre of the Danish National Research Foundation. Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: theis@brics.dk.

1. INTRODUCTION

Pattern matching on strings concerns the problem of determining all the occurrences of a pattern string P, of length p, as substring of a larger text string T, of length n. Optimal $\mathcal{O}(p+n)$ time solutions for this problem were given already in the 70s by Knuth, Morris, and Pratt [19], and Boyer and Moore [3]. Originally the problem was motivated (among other things) by problems in text editing. In subsequent work the problem has been generalized, extended and studied extensively for different applications. See *e.g.*, [1, 6, 25] for recent text books considering different pattern matching problems. An important direction of this progress has been the development of efficient algorithms to handle different dynamic versions of the problem, see *e.g.*, [11, 12, 13, 14, 18, 20, 28, 30]. In a text editor it is, *e.g.*, convenient to handle text updates and searches for different patterns without using time proportional to the full text for every text modification.

We generalize the problem to maintain a family of strings under two update operations, split and concatenate. Given an index i and a string $s = a_1 a_2 \ldots a_k$ in the family, the split operation splits s into the two substrings $a_1 \ldots a_{i-1}$ and $a_i \ldots a_k$, and inserts them into the family without preserving s. The concatenate operation takes two strings s_1 and s_2 from the family, and inserts their concatenation $s_1 s_2$ into the family again without preserving arguments s_1 and s_2. Finally for any string in the family, all occurrences of it within other strings in the family, can be be reported in $\mathcal{O}(\log n \log\log n + \mathit{occ})$ time, where n is the total size of the strings and *occ* the number of occurrences. Update operations are supported in $\mathcal{O}(\log^2 n \log\log n \log^* n)$ time. These bounds are competitive with or improve former results for less generalized versions of the problem (see below). To the best of our knowledge this is also the first result for *pattern matching* which supports split and concatenation of strings in polylogarithmic time per operation.

Pattern matching results for dynamic as well as static problems have shown to be essential tools for problems in computational biology [21, 29]. In DNA sequence analysis often involves operations as split, concatenation and reversals of strings, for which our set of operations can be helpful. For the classic text editor problem, we can handle operations such as moving large text blocks while supporting fast searches. This contributes to an efficient solution for an ideal editor, but for which a larger set of operations still would be preferable [2], *e.g.*, more efficient (persistent) methods to duplicate text blocks.

As an intermediate result we improve the bounds for the *dynamic string equality* problem due to Mehlhorn, Sundar and Uhrig [23]. This problem is to maintain a family of strings under *persistent* concatenate and split operations (the arguments remain in the family) such that the equality of two strings can be determined. Besides the improved time complexity, our solution for the string equality problem extends the set of supported queries. We support comparisons of the lexicographical order of two strings in constant time, and a longest common suffix and prefix operation in almost $\mathcal{O}(\log n)$ time. In [23], the problem is mainly motivated by problems in high-level programming languages like SETL. However later, this data structure has been applied in order to give efficient solutions for other problems, see *e.g.*, [4, 15].

Related work. Here, we sketch the history of pattern matching and refer to [14] for a more detailed account. Some of the early progress in making pattern matching dynamic is the *suffix tree*. In [20, 30] it is shown how to preprocess a text in linear time, such that queries can be answered on-line in $\mathcal{O}(p+\mathit{occ})$ time, where p is the length of the query pattern. Later in [28] the suffix tree is extended such that the text could be extended by a single character to the end. Gu *et al.* [18] were the first to consider the problem where the text could be manipulated fully dynamically, naming the problem *Dynamic text indexing*. The update operations supported were insertion and deletion of a *single character* to/from the text in $\mathcal{O}(\log n)$ time, where n is the current size of the text. The query operation is supported in $\mathcal{O}(p + \mathit{occ}\log i + i\log p)$ time, where i is the current number of updates performed. Next this problem was again extended by Ferragina [12] to efficiently handle insertions/deletions of a *string* into/from the text, called *incremental text editing*. Ferragina and Grossi [11, 13, 14] improve [12]. They provide the time bound $\mathcal{O}(n^{1/2}+s)$ for updates and $\mathcal{O}(p+\mathit{occ})$ for the search, or updates in $\mathcal{O}(s(\log s+\log\log n)+\log n)$ time with query time $\mathcal{O}(p+\mathit{occ}+i\log p+\log\log n)$, where s is the length of inserted/deleted string. In summary, none of the above results are in polylogarithmic time for all operations in terms of the size of the text. Finally in [24], Sahinalp and Vishkin claims the

following result for incremental text indexing. Searches in $\mathcal{O}(p + occ)$ time and updates in $\mathcal{O}(\log^3 n + s)$ time.

Outline of the paper. In Section 2 we review the signature encoding of strings from Mehlhorn *et al.* [23] and state our time bounds for the dynamic string equality problem. Then in Section 3 we describe our data structure for dynamic pattern matching. In Section 4 we provide the implementation for the string equality problem. Finally in the appendix we provide some additional details and figures.

Preliminaries. Given a string s over an alphabet Σ, we let $|s|$ denote the length of s, $s[i]$ the ith element of s ($1 \leq i \leq |s|$), and $s[i..j]$ the substring $s[i]s[i+1]\dots s[j]$ of s ($1 \leq i \leq j \leq |s|$). If $j < i$ then $s[i..j]$ denotes the empty string ϵ. For arbitrary i and j, $s[i..j] = s[\max(1,i)..\min(|s|,j)]$, $s[i..] = s[i..|s|]$ and $s[..j] = s[1..j]$. We let $\mathit{pref}_k(s) = s[..|s|-k]$, $\mathit{suf}_k(s) = s[k+1..]$, and $\mathit{inf}_k(s) = s[k+1..|s|-k]$. The reverse string $s[|s|]\dots s[2]s[1]$ is denoted s^R. For a mapping $f : \Sigma \to \mathcal{U}$, we extend $f : \Sigma^* \to \mathcal{U}^*$ by defining $f(a_1a_2\dots a_n) = f(a_1)f(a_2)\dots f(a_n)$. For two strings s_1 and s_2 we let $lcp(s_1,s_2)$ and $lcs(s_1,s_2)$ denote the longest common prefix and suffix respectively of s_1 and s_2. We assume w.l.o.g. throughout the paper that no string is equal to the empty string.

Let Σ be totally ordered. We define the lexicographical ordering on Σ^* by $s_1 \leq s_2$ if and only if $s_1 = lcp(s_1,s_2)$ or $s_1[|lcp(s_1,s_2)|+1] < s_2[|lcp(s_1,s_2)|+1]$. We let $u \leq_R v$ denote that the reverse of u is less than the reverse of v, *i.e.*, $u^R \leq v^R$.

We let $\log n = \ln n / \ln 2$, $\log^{(1)} n = \log n$, $\log^{(i+1)} n = \log\log^{(i)} n$, and $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$. When interpreting integers as bit-strings we let AND, OR, and XOR denote bitwise boolean operations, and $x \uparrow^i$ be the operation shifting x i bits to the left, *i.e.*, $x \uparrow^i = x \cdot 2^i$. For positive integers x and i we let $\mathrm{bit}(x,i)$ denote the ith bit in the binary representation of x, *i.e.*, $\mathrm{bit}(x,i) = (x \div 2^i) \bmod 2$.

2. Signature encoding of strings

In the following we describe the *signature encoding* of strings over some finite alphabet Σ. The signature encoding we use throughout this paper was originally described by Mehlhorn *et al.* in [23]. The basic idea is to associate a unique *signature* σ to each string s such that two strings are equal if and only if they have equal signatures. The signature encoding of a string $s \in \Sigma^*$ is defined relative to a signature alphabet $\mathcal{E} \subset \mathbb{N}$ and a partial injective mapping $\mathit{Sig} : \Sigma \cup \mathcal{E}^+ \cup (\mathcal{E} \times \mathbb{N}) \hookrightarrow \mathcal{E}$. The mapping Sig is extended during updates in order to keep it defined for all applied values.

The signature encoding of s consists of a sequence of signature strings from $\mathcal{E}^*$, $\mathit{shrink}_0(s), \mathit{pow}_0(s), \mathit{shrink}_1(s), \mathit{pow}_1(s), \dots, \mathit{shrink}_h(s)$. The strings are defined inductively by

$$\begin{aligned}
\mathit{shrink}_0(s) &= \mathit{Sig}(s)\\
\mathit{pow}_0(s) &= \mathit{Sig}(\mathit{encpow}(\mathit{shrink}_0(s)))\\
&\vdots\\
\mathit{shrink}_j(s) &= \mathit{Sig}(\mathit{encblock}(\mathit{pow}_{j-1}(s)))\\
\mathit{pow}_j(s) &= \mathit{Sig}(\mathit{encpow}(\mathit{shrink}_j(s)))\\
&\vdots\\
\mathit{shrink}_h(s) &= \mathit{Sig}(\mathit{encblock}(\mathit{pow}_{h-1}(s)))
\end{aligned}$$

where encpow and $\mathit{encblock}$ are functions defined below, and h the height of the encoding of s which is the smallest value for which $|\mathit{shrink}_h(s)| = 1$. We let $h(s)$ denote the height of the encoding of s.

The mapping encpow groups identical elements such that a substring σ^i is mapped into the pair (σ,i). Formally, for $s \in \mathcal{E}^*$ and $s = \sigma_1^{l_1}\dots\sigma_m^{l_m}$, $\sigma_i \in \mathcal{E}$ where $\sigma_i \neq \sigma_{i+1}$ for $1 \leq i < m$. Then $\mathit{encpow}(s) = (\sigma_1,l_1),(\sigma_2,l_2),\dots,(\sigma_m,l_m)$. The function $\mathit{encpow}(s)$ can be computed in time $\mathcal{O}(|s|)$.

The mapping $encblock$ decomposes a string into a sequence of small substrings of sizes between two and four, except for the first block which has size between one and four. Each substring is denoted a *block*. The strategy behind the decomposition is based on the *deterministic coin tossing* algorithm of Cole and Vishkin [5] which ensures the property that the boundaries of any block are determined by a small neighborhood of the block. This strategy is only applicable to strings where no two consecutive elements are identical and the role of the mapping $encpow$ is to ensure this property prior to employment of $encblock$.

Because the signature encoding is deterministic, two identical strings also have identical encodings. Figure 1 gives the signature encoding of a string of length 54.

The neighborhood dependence of a block decomposition is characterized by two parameters Δ_{L} and Δ_{R}, such that given a signature σ in a string it can be determined if σ is the first signature in a block by only examine Δ_{L} and Δ_{R} signatures respectively to the left and to right of σ. We assume in the following that N is a constant bounding the total number of signatures to be used, and we also assume that signatures and characters can be handled in constant time. Given a signature σ we let $\overline{\sigma}$ denote the string from Σ^* encoded by σ, and for a signature string $\sigma_1 \dots \sigma_k$ we let $\overline{\sigma_1 \dots \sigma_k} = \overline{\sigma_1} \dots \overline{\sigma_k}$.

The details of the block decomposition of Mehlhorn *et al.* [23] are included in Appendix A. From Lemma 6 in Appendix A it follows that $\Delta_{\mathrm{L}} = \log^* N + 6$ and $\Delta_{\mathrm{R}} = 4$.

2.1. **Persistent strings.** Mehlhorn *et al.* [23] considered how to maintain a family $\mathcal{F}$ of strings under the following operations.

- STRING(a) **:** A new single letter string containing the letter $a \in \Sigma$ is created. The resulting string is added to $\mathcal{F}$ and returned.
- CONCATENATE(s_1, s_2) **:** Concatenates the two strings $s_1, s_2 \in \mathcal{F}$. The resulting string is added to $\mathcal{F}$ and returned. The two strings s_1 and s_2 are *not* destroyed.
- SPLIT(s, i) **:** Splits s into two strings $s[..i-1]$ and $s[i..]$. The two resulting strings are added to $\mathcal{F}$ and returned. The string s is *not* destroyed.
- EQUAL(s_1, s_2) **:** Returns true if and only if $s_1 = s_2$.

Note that strings are never modified or destroyed, *i.e.*, the strings created are *persistent*. In the CONCATENATE operation s_1 and s_2 are allowed to refer to the same string, *i.e.*, it is possible to construct strings of exponential length in linear time. Mehlhorn *et al.* [23] proved the following theorem.

Theorem 1 (Mehlhorn *et al.* [23])**.** *There exists a persistent string implementation which supports* STRING *and* EQUAL *in* $\mathcal{O}(1)$ *time, and* CONCATENATE *and* SPLIT *in* $\mathcal{O}(\log n((\log^* N)^2 + \log n))$ *time, where* n *is the length of strings involved in the operations.*

In the above theorem we assumed that a lookup in the Sig function takes constant time. In [23] the Sig function is stored using a search tree, implying that it takes time $\log m$ to make a lookup, where m is the number of operations done so far. Constant time lookup for Sig can be achieved by using randomization or using more than linear space by either using dynamic perfect hashing [10] or using a digital search tree of degree N^c [22], $0 < c < 1$. The number of lookups to the Sig function for each CONCATENATE and SPLIT operation is $O(\log n \log^* N)$.

In Section 4 we show how to improve the bounds of [23] and to extend the set of supported persistent string operations with the following operations.

- COMPARE(s_1, s_2) **:** Returns the lexicographical order of s_1 relative to s_2, *i.e.*, if $s_1 = s_2$, $s_1 < s_2$, or $s_1 > s_2$.
- LCPREFIX(s_1, s_2) **:** Returns $|lcp(s_1, s_2)|$.
- LCSUFFIX(s_1, s_2) **:** Returns $|lcs(s_1, s_2)|$.

The following theorem summarizes our results in Section 4 for persistent strings.

Theorem 2. *There exists a persistent string implementation which supports* STRING *in* $\mathcal{O}(\log|\Sigma|)$ *time,* EQUAL *and* COMPARE *in* $\mathcal{O}(1)$ *time,* LCPREFIX *in* $\mathcal{O}(\log n)$ *time,* LCSUFFIX *in* $\mathcal{O}(\log n \log^* N)$ *time,*

and CONCATENATE *and* SPLIT *in* $\mathcal{O}(\log n \log^* N + \log|\Sigma|)$ *time, where* n *is the length of strings involved in the operations.*

3. DYNAMIC PATTERN MATCHING

In this section we will describe how to implement a data structure for the dynamic pattern matching problem, with the claimed update and query time bounds.

Let $\mathcal{G}$ denote a family of strings over a fixed alphabet Σ. An *occurrence* of a string s in family $\mathcal{G}$, is a pair (s', p) where $s' \in \mathcal{G}$ and p specifies the specific *location* of the occurrence within s'. Let $index(p)$ denote the index offset of this location in s', *i.e.*, it satisfies $s = s'[index(p)..index(p) + |s| - 1]$. We denote the set of all occurrences of s in $\mathcal{G}$ by $Occ(s, \mathcal{G})$.

The dynamic pattern matching problems is to maintain a data structure for a family of strings $\mathcal{G}$ which supports the updates STRING, SPLIT and CONCATENATE for strings in $\mathcal{G}$ defined as in last section, but *without* the persistence, *i.e.*, the arguments to SPLIT and CONCATENATE are removed from $\mathcal{G}$ by the call. In addition to these update operations the data structure supports the search query:

FIND(s) Return the set of all occurrences of $s \in \mathcal{G}$.

For the rest of this section we let n denote the total size of $\mathcal{G}$, *i.e.*, $n = \sum_{s\in\mathcal{G}} |s|$.

Theorem 3. *There exists an implementation for the dynamic pattern matching problem which supports* CONCATENATE, SPLIT *in* $\mathcal{O}(\log^2 n \log\log n \log^* n)$ *time,* STRING *in* $\mathcal{O}(\log n \log^* n)$ *time and* FIND(s) *in* $\mathcal{O}(occ + \log n \log\log n)$ *time where* occ *is the number of occurrences.*

The occurrences returned by the FIND operation, are represented by *pointers* into the specific occurrences in lists representing the strings. For such pointer we need additional $\mathcal{O}(\log n)$ time to compute the exact offset $index(p)$ of the occurrence. That is the time for FIND is $\mathcal{O}(occ \log n + \log n \log\log n)$ when output is required in this form.

3.1. **The data structure.**

The data structure consists of several ingredients, where the primary part consists of a combination of a range query data structure with the persistent string data structure given in 4.

3.1.1. *Dynamic lists and signature encodings for* $\mathcal{G}$. For each string in $s \in \mathcal{G}$ we maintain a list $l(s)$, where the ith character in s is the ith node in $l(s)$. These lists are maintained by balanced trees under join and split operations, such that given index i one can report the ith node $l(s)[i]$ and return the rank of a node, see *e.g.*, [6].

The set of all nodes for all lists for $\mathcal{G}$ is denoted L. For a list $l(s)$, we denote the ith node by $l(s)[i]$. Next every string in $\mathcal{G}$ is also represented in an auxiliary family of strings we call $\mathcal{F}$ using the persistent data structure from Section 2.1. Besides the strings in $\mathcal{G}$ family $\mathcal{F}$ also contains certain substrings of these described later. Furthermore we assume the reverse representation of every string $t \in \mathcal{F}$ to be in $\mathcal{F}$ as well, *i.e.*, $t^R \in \mathcal{F}$. This only increases the time requirement for the split and concatenation operation on $\mathcal{F}$ by a constant factor. For all the strings in $\mathcal{G} \subseteq \mathcal{F}$, we assume the terminology of Section 2.1 with respect to the signature encodings of these strings.

3.1.2. *Anchors.* Consider a string $s \in \mathcal{G}$ and a level $j \geq 0$ with the signature string $x = shrink_j(s)$ encoding all of s. We define the set of *breakpoints* for x by

$$\text{BP}(x) = \{\, i \mid x[i] \neq x[i+1] \,\}.$$

The *offset* $offset_s^j(i)$ of a breakpoint $i \in \text{BP}(x)$, is defined to be the index in s, where the signature $x[i]$ starts its encoding in s, *i.e.*, $offset_s^j(i) = |\overline{x[1..i-1]}| + 1$. Similarly we will talk about breakpoints in substrings of signature strings, *i.e.*, the breakpoints in substring $x[k..l]$ is simply $\text{BP}(x) \cap [k..l]$.

For a breakpoint $i \in \text{BP}(shrink_j(s))$ we associate an *anchor* that consists of two infixes of s which captures a certain context of that breakpoint. Furthermore the anchor also contains the node $l(s)[offset_s^j(i)]$.

Fix Δ to be an integer larger than $\Delta_{\mathrm{L}} + \Delta_{\mathrm{R}} + 4$ for the rest of this section. The *left boundary* of breakpoint i, denoted $\mathrm{lb}(i)$, is a breakpoint "16Δ breakpoints left" of i, formally:

$$\mathrm{lb}(i) = \max(\{\, j \in \mathrm{BP}(x) \mid |[j..i] \cap \mathrm{BP}(x)| > 16\Delta \,\} \cup \{\, 1 \,\}).$$

Similarly the *right boundary* of i is defined by

$$\mathrm{rb}(i) = \min(\{\, j \in \mathrm{BP}(x) \mid |[i..j] \cap \mathrm{BP}(x)| > 16\Delta \,\} \cup \{\, |x| \,\}).$$

Let l, p and r be the offsets s of $\mathrm{lb}(i)$, i and $\mathrm{rb}(i)$ respectively. The *anchor* associated breakpoint i, denoted $\mathrm{Anc}(i)$, is defined to be the triple

$$(s[l..p-1], s[p..r-1], l(s)[p]) \in \Sigma^* \times \Sigma^* \times L.$$

We refer to the strings $s[l..p-1]$ and $s[p..r]$ as the *context* strings anchor $\mathrm{Anc}(i)$.

For every breakpoint in some shrink signature string for strings in $\mathcal{G}$, there is an anchor, *i.e.*, the set of all anchors is: $\mathrm{Anchors}(\mathcal{G}) = \bigcup_{s\in\mathcal{G}, j\geq 0}\{\, \mathrm{Anc}(i) \mid i \in \mathrm{BP}(shrink_j(s)) \,\}$. For an anchor $(s_1, s_2, e) \in \mathrm{Anchors}(\mathcal{G})$, we maintain copies of the strings s_1 and s_2 in the family $\mathcal{F}$. That is we maintain the invariant $\mathrm{Anchors}(\mathcal{G}) \subseteq \mathcal{F} \times \mathcal{F} \times L$.

3.1.3. *Range query data structure.* For each level j, the anchors associated with breakpoint at this level, *i.e.*, breakpoint in strings $shrink_j(s)$ for $s \in \mathcal{G}$, is kept in a *range query data structure* $R_j \subseteq \mathrm{Anchors}(\mathcal{G})$. The data structure for R_j supports the operations

- $\textsc{Insert}(R_j, a)$**:** Inserts the anchor $a \in \mathrm{Anchors}(\mathcal{G})$ into R_j.
- $\textsc{Delete}(R_j, a)$**:** Deletes anchor a from R_j.
- $\textsc{Report}(R_j, t_1, t_2, t_3, t_4)$**:** For $t_1, t_2, t_3, t_4 \in \mathcal{F}$, reports all anchors $(s_1, s_2, e) \in R_j$ where $t_1 \leq_R s_1 \leq_R t_2$ and $t_3 \leq s_2 \leq t_4$.

The above operations are supported in a comparison based data structure, with access to the COMPARE operation on the strings in $\mathcal{F}$ (using the reverse representations of string in $\mathcal{F}$ for the $\leq_R$ comparisons), as described in Section 2.1. That is the above INSERT and DELETE operation, can be supported in worst-case time $\mathcal{O}(\log |R_j| \log\log |R_j|)$ and the REPORT operation in worst-case time $\mathcal{O}(\log |R_j| \log\log |R_j| + occ)$, see [8]. Note that $|R_j|$ is bounded by $\mathcal{O}(n)$.

3.2. **Searching for occurrences in $\mathcal{G}$.** In this section we will describe how to perform a search of all occurrences $Occ(s, \mathcal{G})$, provided the above representation for $\mathcal{G}$.

The search considers different cases depending on the signature encoding of the string s. Fix $\Delta > \Delta_{\mathrm{L}} + \Delta_{\mathrm{R}} + 4$ as in last section.

Case 1: $|pow_0(s)| \leq 12\Delta$.
Case 2: There exists $j > 0$ such that $|shrink_j(s)| > 3\Delta$ and $|\mathrm{BP}(shrink_j(s))| \leq 12\Delta$.

Lemma 1. *For any string $s \in \mathcal{G}$, either Case 1 or Case 2 (or both) are satisfied.*

Proof. Suppose Case 1 is *not* satisfied. Then let $j = \min\{\, i \mid |pow_i(s)| \leq 12\Delta \,\}$. Then $|pow_{j-1}(s)| > 12\Delta$ and since each block has size at most 4, we have $|shrink_j(s)| \geq \frac{1}{4}|pow_{j-1}(s)| > 3\Delta$. By minimality of j, $|\mathrm{BP}(shrink_j(s))| = |pow_j(s)| \leq 12\Delta$, so level j satisfies Case 2. □

Let $j = 0$ if Case 1 above is satisfied, or choose $j > 0$ as in the proof of above lemma such that Case 2 is satisfied and let $x = shrink_j(s)$. For Case 2 above we define the 'protected' set of breakpoints, denoted M, as the breakpoints in infix $inf_\Delta(x)$, *i.e.*, $M = \mathrm{BP}(x) \cap [\Delta + 1..|x| - \Delta]$. For Case 1 ($j = 0$), the "protected" breakpoints is simply all the breakpoints, *i.e.*, $M = \mathrm{BP}(shrink_0(s))$.

In this section we limit the exposition to the case where M is nonempty, *i.e.*, for Case 2, we assume the substring $inf_\Delta(x)$ of length at least Δ contains two different signatures. The special case where M is empty, *i.e.*, s contains a long substring of small periodicity, is omitted due to lack of space.

Hence let $i \in M$. The following lemma states that for each specific occurrence of s in $\mathcal{G}$, this occurrence is encoded by a shrink string that contains a breakpoint which is 'aligned' with breakpoint i.

Lemma 2. *For any occurrence $(t,p) \in Occ(s,\mathcal{G})$, there exists $i' \in BP(shrink_j(t))$ where $index(p) = offset_t^j(i') - offset_s^j(i) + 1$.*

We say that the breakpoint $i' \in \mathrm{BP}(shrink_j(t))$ from the above lemma *aligns* with breakpoint i, relative to occurrence $(t,p) \in Occ(s,\mathcal{G})$.

Proof. For $j = 0$, the lemma is immediate from definition of breakpoints and the facts $i' = offset_t^0(i')$ and $i = offset_s^0(i)$. Consider the case for $j > 0$, where the premises in Case 2 holds. Let $(t,p) \in Occ(s,\mathcal{G})$ and $i \in M$. Write t as $t = t_1\, s\, t_2$ where $|t_1| = index(p) - 1$. By repeated use of Lemma 9 it follows that

$$shrink_j(t) = u\, shrink_j(s)[\Delta + 1..|shrink_j(s)| - \Delta]\, v \tag{1}$$

for some $u, v \in \mathcal{E}^*$, where

$$\overline{u} = t_1\, \overline{shrink_j(s)[..\Delta]}. \tag{2}$$

Since i is a breakpoint in M, we can write (1) as

$$shrink_j(t) = u\, shrink_j(s)[\Delta + 1..i] shrink_j(s)[i + 1..|shrink_j(s)| - \Delta]\, v$$

and hence the index $i' = |u| + i - \Delta$ is a breakpoint in $\mathrm{BP}(shrink_j(t))$. Furthermore using (2)

$$\begin{aligned} offset_t^j(i') &= |\overline{u}| + |\overline{shrink_j(s)[\Delta + 1..i]}| \\ &= |t_1| + |\overline{shrink_j(s)[..\Delta]}| + |\overline{shrink_j(s)[\Delta + 1..i]}| \\ &= |t_1| + offset_s^j(i). \end{aligned}$$

Hence $index(p) = |t_1| + 1 = offset_t^j(i') - offset_s^j(i) + 1$ as desired. □

The next lemma states that for every breakpoint i' that aligns with i, the anchor associated i' has 'sufficiently large' context information with respect to s. Formally write $s = s_1 s_2$, where $s_1 = s[1..offset_s^j(i) - 1]$.

Lemma 3. *Let i' be any breakpoint which aligns with breakpoint i relative to an occurrence $(t,p) \in Occ(s,\mathcal{G})$. Let $(t_1,t_2,e) = Anc(i') \in R_j$. Then $|s_1| \le |t_1|$ and $|s_2| \le |t_2|$, i.e., $lcs(s_1,t_1) = s_1$ and $lcp(s_2,t_2) = s_2$.*

Proof. Let $t = t'\, s\, t''$ such that $|t'| = index(p) - 1$. Write $shrink_j(t) = u_1\, v_1\, v_2\, u_2$ where v_1 is the infix from the left boundary of i, *i.e.*, $shrink_j(t)[\mathrm{lb}(i')..i' - 1]$ and similar v_2 is until the right boundary, $v_2 = shrink_j(t)[i'..\mathrm{rb}(i')]$. We show $|t_1| = |\overline{v_1}| \ge |s_1|$ by showing $\overline{u_1}$ is a prefix of t'. By repeated use of Lemma 9 we can write:

$$shrink_j(t) = pref_\Delta(shrink_j(t'))\, w_1\, inf_\Delta(shrink_j(s))\, w_2\, suf_\Delta(shrink_j(t'')),$$

where $|w_1|, |w_2| \le \Delta$. Next we bound $|\mathrm{BP}(w_1\, inf_\Delta(shrink_j(s))\, w_2)| \le |\mathrm{BP}(shrink_j(s))| + 2\Delta \le 14\Delta$. Since the index i' has to be within this infix, and $|\mathrm{BP}(v_1)| = 16\Delta$ (u_1 is nonempty), $u_1 = shrink_j(t)[..i' - |v_1|]$ must be a prefix of $pref_\Delta(shrink_j(t'))$, and thus $\overline{u_1}$ is a prefix of t' as desired. By a similar argument $\overline{u_2}$ is a suffix of t'', implying $|t_2| \ge s_2$. □

Suppose the alphabet is extended with a special letter \$ in the alphabet larger than letters occurring in strings for $\mathcal{G}$. Lemma 2 and Lemma 3 tell us there is an anchor $(t_1,t_2,e) \in R_j$ with $s_1 \le_R t_1 \le_R \$s_1$ and $s_2 \le t_2 \le s_2\$$ if and only if node $e \in L$ points into $(t,p) \in occ(s,\mathcal{G})$ at offset $|s_1|$ right of $index(p)$. Thus the set of all occurrences can be computed by the following steps:

1. Find a level j such that one of the above cases are satisfied and a breakpoint $i \in M$.
2. Compute the offset $p = offset_s^j(i)$ of i in s. Construct and insert the strings $s_1 = s[1..p-1]$, $s_2 = s[s[p..|s|]$, $\$s_1$ and $s_2\$$ into $\mathcal{F}$ using the SPLIT and CONCATENATE operations for $\mathcal{F}$.
3. Report occurrences (represented as nodes in L) by the query $\textsc{Report}(R_j, s_1, \$s_1, s_2, s_2\$)$.

As remarked earlier, if occurrences have to be reported as indices into the corresponding strings, we need a rank query for each reported node.

3.2.1. *Time usage.* We summarize the time usage for the search can be summarized to. Step 1 takes time $\mathcal{O}(\log^* n)$, *i.e.*, if $|pow_0(s)| \geq 8\Delta$ we expand the root signature for s until we get a level j where $pow_j(s)$ has length at most 8Δ and $shrink_j(s)$ satisfy the premises in Case 2. Step 2 uses time $\mathcal{O}(\log n \log^* n)$ for the computation of the index in s of the chosen breakpoint, with subsequent calls of SPLIT and CONCATENATE to compute the arguments for the REPORT operation. Finally the single call of operation uses $\mathcal{O}(\log n \log\log n + occ)$ time, leading to the same total time cost for the search.

3.3. **Concatenate and split.** In this section we describe how to perform concatenation of two strings in $\mathcal{G}$. The split operation for a string in $\mathcal{G}$ is done in a similar manner and omitted.

CONCATENATE(s_1, s_2). Consider two strings $s_1, s_2 \in \mathcal{G}$ where we want to compute the concatenation $s = s_1 s_2$ and insert this string s into $\mathcal{G}$, destroying s_1 and s_2. First the signature encoding for s is computed and inserted into the auxiliary string family $\mathcal{F}$ through the CONCATENATE operation for this family. Next a new list $l(s)$ for s is created by joining $l(s_1)$ and $l(s_2)$. This means that the node information associated the anchors in the various range query structures R_j now is considered as nodes in $l(s)$ instead.

Now the main part of the computation consists of restructuring the various range query data structures R_j which now must contain anchors with respect to the signature encoding of s. In the following we will describe what is done at each level. Fix a level $j \geq 0$. Write $shrink_j(s_1) = u_1\, w'$ where w' is such that $|\text{BP}(w')| = 16\Delta$ and similarly for $shrink_j(s_2) = w''\, u_2$ with $|\text{BP}(w'')| = 12\Delta$. By repeated use of Lemma 9 we can write the new signature string $shrink_j(s) = u_1\, w\, u_2$ where $|\text{BP}(w)| \leq 34\Delta$. For i a breakpoint in u_1 or u_2, let $\text{Anc}(i)$ denote the anchor when it was associated $shrink_j(s_1)$ or $shrink_j(s_2)$, and let $\text{Anc}'(i)$ denote the anchor for $shrink_j(s)$.

Lemma 4. *For any breakpoint i within substrings u_1 or u_2, the context strings in $\text{Anc}(i)$ and $\text{Anc}'(i)$ are the same.*

Proof. For a breakpoint in prefix u_1, the left and right boundary are within $pref_\Delta(shrink_j(s_1))$ which is a prefix of $shrink_j(s)$ by Lemma 9. Hence the context strings for i defined Section 3.1.2 are the same. Similarly for breakpoints in u_2. □

In order to update R_j the following is done. First the anchors in R_j for breakpoints associated the suffix and prefix w' and w'' of the 'old' shrink signature strings are removed. Next for every breakpoint in the above infix w, a new associated anchor is created and inserted into R_j. After this, every anchor for $shrink_j(s)$ is correctly represented in R_j since for anchors not in a breakpoint in w', w'' and w, *i.e.*, a breakpoint in u_1 or u_2, the associated context strings are by Lemma 4 unchanged, and the associated node is also correct after the link of $l(s_1)$ and $l(s_2)$ forming $l(s)$. In order to create the new anchors for substring w, the breakpoints in w are determined, which is done through the information of $pow_j(s)$ and using techniques similar to the description given in Appendix C. For each such breakpoint i in w, we then need to create its new anchor, $\text{Anc}'(i)$. First the node $l(s)[offset_s^j(i)]$ is computed (again by the techniques from Appendix C and the query operations supported for list $l(s)$). Then in order to compute the new *context* strings for the anchor, we also need to compute the offsets in s of the left and right boundaries $\text{lb}(i)$ and $\text{rb}(i)$. Then finally by applying the persistent SPLIT operation on $s \in \mathcal{F}$ for the offsets of these boundaries, the two context strings for the anchor are generated in $\mathcal{F}$.

3.3.1. *Time usage.* The computation of breakpoints, offsets, context strings and the list operations are done within time $\mathcal{O}(\log n \log^* n)$ per anchor. The number of anchors manipulated is the number of breakpoints in w', w'' and w. For each of these anchors, a single call of INSERT or DELETE is done, using time $\mathcal{O}(\log n \log\log n)$, see Section 3.1.3. Hence in total the CONCATENATE(s_1, s_2) operation use worst-case time $\mathcal{O}((\log n \log^* n) \cdot (\log n \log^* n + \log n \log\log n)) = \mathcal{O}(\log^2 n \log\log n \log^* n)$.

4. PERSISTENT STRINGS

We represent a persistent string s by the root-signature σ of a signature encoding of s. We denote this the *implicit representation* of s. This implies that a string can be stored in a single word plus the space required to store the signature function Sig. The lower levels of the signature encoding of s can be extracted from σ by recursively applying Sig^{-1}, especially the neighborhoods which need to be considered by the different operations can be constructed when required. By storing Sig^{-1} as an array, Sig^{-1} can be computed in $\mathcal{O}(1)$ time.

We would like to note, that this is an essential difference compared to the representation used in Mehlhorn *et al.* [23]. They represent a string by a persistent data structure which consists of a linked list of rooted trees, each tree storing one level of the signature encoding of the string. Their representation implies an overhead of $\mathcal{O}(\log n)$ for accessing each level of the encoding of a string. Our simplified representation avoids this overhead.

By using the implicit representation of strings we get Lemma 5 below, improving and extending the result of Mehlhorn *et al.* [23].

Lemma 5. *The operations* STRING *and* EQUAL *can be supported in* $\mathcal{O}(1)$ *time,* CONCATENATE, SPLIT, *and* LCSUFFIX *in time* $\mathcal{O}(\log n \log^* N)$ *time, and* LCPREFIX *and* COMPARE *in* $\mathcal{O}(\log n)$ *time, where* n *is the length of strings involved in the operations.*

Proof. The operation $\textsc{String}(a)$ returns $Sig(a)$, and $\textsc{Equal}(s_1, s_2)$ returns true if and only if the root-signatures of the signature encodings of s_1 and s_2 are identical. The details of the remaining operations can be found in Lemma 10, Lemma 11 and Lemma 12 in the Appendix. □

Figure 2 contains an example of a LCPREFIX computation. The underlined characters are the leftmost characters which mismatch. The signatures in circles are the signatures from the signature encodings which actually are compared. Full circles denote successful comparisons. The sequence of signatures connected by dashed lines are the signatures which define the lcp of the two strings. Note that for the powers $9^6 = Sig^{-1}(12)$ and $9^4 = Sig^{-1}(27)$ we do not need to compare all signatures in 9^4.

4.1. **Maintaining strings sorted.** In the previous section we describe how to perform comparisons on persistent strings in $\mathcal{O}(\log n)$ time. In this section we show how reduce this to $\mathcal{O}(1)$ time while maintaining the asymptotic times for the update operations STRING, CONCATENATE and SPLIT except for an additive $\log|\Sigma|$ term. The ideas used are: *i*) keep all persistent strings lexicographical sorted, and *ii*) associate with each string s a *key* $\text{key}(s)$, such that two strings can be compared by comparing their associated keys in $\mathcal{O}(1)$ time.

Data structures for maintaining order in a list have been developed by Dietz [7], Dietz and Sleator [9] and Tsakalidis [27]. The data structure of Dietz and Sleator [9] supports $\textsc{Insert}(x, y)$, $\textsc{Delete}(x)$ and $\textsc{Order}(x, y)$ operations in worst-case $\mathcal{O}(1)$ time. The operation $\textsc{Insert}(x, y)$ inserts element y after x in the list, and $\textsc{Delete}(x)$ deletes x from the list. The query $\textsc{Order}(x, y)$ returns if x is before y in the list.

The key we associate with each persistent string is a "handle" given by the data structure of Dietz and Sleator [9]. A $\textsc{Compare}(s_1, s_2)$ query can now be answered in worst-case $\mathcal{O}(1)$ time by applying $\textsc{Equal}(s_1, s_2)$ and by applying $\textsc{Order}(\text{key}(s_1), \text{key}(s_2))$.

In the remaining of this section we describe how new strings created by STRING, CONCATENATE and SPLIT can be added to the lexicographical sorted list of strings, *i.e.*, how to locate where to insert new strings into the data structure of Dietz and Sleator. A straightforward implementation is to store the strings as elements in a balanced search tree and to use the COMPARE operation described in Lemma 12 when searching in the search tree. This implementation requires $\mathcal{O}(\log m \log n)$ time for each string created, where m is the number of strings stored. In the following we describe how to avoid the $\log m$ factor.

4.2. **Tries.** We now describe a data structure for maintaining a set of strings in sorted order. The basic data structure is a collection of *tries*. A trie is a rooted tree where all edges are label with symbols from some

alphabet, and where all child edges of a node are distinct labeled. Given a node v we let S_v denote the string consisting of the concatenation of the labels along the path from the root to v. Further information on tries can be found in, *e.g.*, [22]. We will use the terminology that v corresponds to the string S_v. First we give the outline of a static space inefficient data structure to illustrate the underlying search. After having sketched the basic idea we fill in the technical details to make the data structure dynamic.

Let $\mathcal{F}$ be the set of strings stored. For each level i of the signature encodings, we maintain a trie T^i storing all signature strings $shrink_i(s)$ for all $s \in \mathcal{F}$, and such that each leaf in T^i corresponds to $shrink_i(s)$ for an $s \in \mathcal{F}$. The child edges of a node is stored in a sparse array indexed by the edge labels. For each node v in T^i, $i \geq 1$, we have a pointer to a node v' in T^{i-1}. Let s be a string in $\mathcal{F}$ such that S_v is a prefix of $shrink_i(s)$. Then v' is the node satisfying that $S_{v'}$ is a prefix of $shrink_{i-1}(s)$ and $\overline{S_{v'}} = \overline{S_v}$.

We let the children of each node v of T^0 be sorted from left-to-right w.r.t. the ordering on the characters encoded by the edge signatures, and let the child list of each node in T^0 be stored in a balanced search tree. A left-to-right preorder traversal of T^0 will then report the nodes sorted w.r.t. the lexicographical order of the corresponding strings.

To each node v of T^0 we associate two keys k_v and k'_v in the Dietz and Sleator data structure such that k_v appears before all keys before the keys assigned to descendents of v and k'_v appears after all keys assigned to descendents of v.

Given the above described data structure we can find the ordering of a new string s among the strings in $\mathcal{F}$, *i.e.*, we can assign s a key from the Dietz and Sleator data structure.

The search for s is very similar to the computation of LCPREFIX of two strings in Appendix C. For each level i of the signature encoding of s we identify the node v_i in T^i such that S_{v_i} is the maximal prefix of $shrink_i(s)$ that matches a prefix of $shrink_i(t)$ for a string $t \in \mathcal{F}$. Note that the string t can vary from level to level as shown in Figure 3, where v_1 corresponds to $lcp(shrink_1(s), shrink_1(t_2))$ and v_0 corresponds to $lcp(shrink_0(s), shrink_0(t_1))$. When $v_i \in T^i$ has been found we follow the pointer to the node $v' \in T^{i-1}$ where $S_{v'}$ is a prefix of $shrink_{i-1}(s)$. From v' we can continue the search down the trie T^{i-1} as long as the prefix of $shrink_{i-1}(s)$ matches the labels along the path from the root to the current node, until we find v_{i-1}.

As in the implementation of LCPREFIX we incrementally expand the signature description of s along the search path by always keeping the $6 + \Delta_{\mathrm{R}}$ next signatures, *i.e.*, after v_i has been found we just have to keep $shrink_i(s)[|S_v| + 1..|S_v| + 6 + \Delta_{\mathrm{R}}]$. That it is sufficient to expand $6 + \Delta_{\mathrm{R}}$ signatures of s for each level follows similarly to the proof of Lemma 12, by showing a contradiction with that the matching prefix found for $shrink_{i+1}(s)$ is maximal. Note that $6 + \Delta_{\mathrm{R}}$ signatures in $shrink_i(s)$ can at most expand to $4(6 + \Delta_{\mathrm{R}})$ signature powers in $shrink_{i-1}(s)$. To be able to skip an arbitrary part of a signature power in $\mathcal{O}(1)$ time we associate an array A_e (indexed $1..|A|$) for each outgoing edge e of a node v with a signature different from the signature on the edge leading to v such that $A_e[j]$ is a pointer to the node in the subtree rooted at v which can be reached by a path labeled σ^j. We let $|A|$ be the length of the longest path beginning with e and where all edges have label σ. Each of the at most $5 + \Delta_{\mathrm{R}}$ matching signature powers can now be matched in constant time by using the A_e arrays to arrive at the correct nodes.

A new signature s can be inserted into the data structure, by first performing the above described search. The search identifies for each T^i the node v_i where a new path of nodes should be attached, corresponding to the unmatched tail of $shrink_i(s)$. Together with the nodes and keys we add in T^i we also have to add the appropriate pointers to nodes in T^{i-1} and update A_e arrays. The new child c of v_0 can be inserted into the sorted list of children of v_0 in time $\mathcal{O}(\log |\Sigma|)$. We can assign c keys as follows. If there exists a sibling v to the left of c then we can assign c keys such that k_c and k'_c appears immediately after k'_v. If c has no sibling to the left we create keys k_c and k'_c immediately after k_{v_0}. In both cases the keys can be assigned in $\mathcal{O}(1)$ time. The assignment of keys to the tail of the path is done by recursively applying the above key assignment. The total search time is $\mathcal{O}(\Delta_{\mathrm{R}} \log |s|)$ and the total update time is $\mathcal{O}(|s| + \log |\Sigma|)$ because the size of the levels of the encoding of s are geometrically decreasing.

4.3. **Tries with collapsed paths.** The update time in the previous section is $\mathcal{O}(|s|)$ because a linear number of new nodes need to be created. Except for the first child created in each trie all new nodes have at most

one child. The main idea to speedup updates is by collapsing long paths of nodes with only one child. This is vaguely related to the standard notion of compressed tries [22], except that we do not necessarily collapse all nodes with one child, and that we use a different approach to represent collapsed edges.

For each trie T_i we define a set of *down* nodes. Down nodes are: *i*) the root of T_i, *ii*) nodes with two or more children, and *iii*) nodes corresponding to strings in $\mathcal{F}$. Note that *iii*) implies that all leaves are down nodes. We will maintain the invariant that only for down nodes we have a pointer to the corresponding nodes in T_{i-1}. We call these nodes in T_{i-1} *up* nodes. The remaining nodes all have exactly one child and are denoted *collapsible*. Only non-collapsible nodes are stored explicitly. A collapsible path is a path where all nodes are collapsible except for the first and the last node. Each collapsible path is stored as a *super* edge. To each super edge e connecting v and a descended u in T_i we associate the start and the end indices of the substring spanned by e, *i.e.*, $|\overline{S_v}|$ and $|\overline{S_u}|$. Finally we store the first $4(6 + \Delta_{\mathrm{R}})$ signature powers along e. The stored signature powers guarantee that if the trie search in T_i as described in the previous section starts at v then we have sufficient signatures stored along e required by the search. Because collapsible nodes are not stored we do not have to assign keys to these nodes, *i.e.*, in each trie T_i at most $\mathcal{O}(|\mathcal{F}|)$ nodes are stored and have assigned keys.

To perform a search we proceed basically as described in the previous section, except that parts of the trie structures must be expanded in the search (like the expansion of the search string) and that going from T_i to T_{i-1} is more complicated because only down nodes have pointers to the tries one level below. The information we incrementally expand for the tries is the following. Assume that we have found the node v_i in T_i which corresponds to the longest matching prefix with $shrink_i(s)$. We maintain the invariant that if v_i is a collapsible node, then we have a pointer to the super edge containing v_i and we have expanded the $6 + \Delta_{\mathrm{R}}$ signatures after v_i in the collapsible path represented by e. To find the corresponding node in T_{i-1} there are two cases: If v_i has a pointer we are done, otherwise we find the nearest ancestor down node w_i of v_i and follow the down pointer stored at w_i. There are at most $\Delta_{\mathrm{R}} + 2$ up nodes which must be visited before the down node is reached.

¿From the reached node we can find the node in T_{i-1} corresponding to v_i by expanding the prefix of the signatures between w_i and v_i. If this node is implicitly represented by a super edge e' we can expand the signatures required by the search in T_{i-1} from the signatures we have expanded for e and from the explicit stored signature power strings stored in T_{i-1} and T_i. To handle signature powers we modify the arrays we introduced in the previous section to only store pointers to non-collapsed nodes, *i.e.*, the arrays are sparse. To match a signature power we then make a brute force linear search in $\mathcal{O}(\Delta_{\mathrm{R}})$ time to find the non-collapsed node closest to the the longest matching prefix. If the linear search does not succeed we do an exponential search to find the super edge containing the node corresponding to the longest matching prefix in T_{i-1}. The exponential search is only required once for every string. In total a new string can be inserted in to the tries in total $\mathcal{O}(\log |s| \Delta_{\mathrm{R}}) = \mathcal{O}(\log |s|)$ time.

REFERENCES

[1] A. Apostolico and Z. Galil. Pattern matching algorithms. *Oxford university press*, 1997.

[2] P. Atzeni and G. Mecca. Cut and paste. In *16th Ann. ACM Symp. on Principles of Database Systems (PODS)*, pages 144–153, 1997.

[3] R. Boyer and J. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[4] S. Cheng and M. Ng. Isomorphism testing and display of symmetries in dynamic trees. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 202–211, 1996. Note, to cover some of the problems in that paper, the result from this paper should be combined with technical report 98-6, Alstrup *et al.*, Comp. Sc. Dept., University of Copenhagen.

[5] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[6] T.H. Cormen, C.E. Leiserson and R.L. Rivest. Introduction to algorithms. *The MIT electrical engineering and computer science series*, Eight printing 1992, chapter 34.

[7] Paul F. Dietz. Maintaining order in a linked list. In *Proc. 14th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 122–127, 1982.

[8] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.
[9] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.
[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proc. 29th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 524–531, 1988.
[11] P. Ferragina. Dynamic data structures for string matching problems. *Ph.D. Thesis:TD–3/97.*, Department of informatica, University of Pisa.
[12] P. Ferragina. Dynamic text indexing under string updates. *Journal of algorithms.*, 22(2):296–328, 1997. See also (ESA'94).
[13] P. Ferragina and R. Grossi. Fast incremental text indexing In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 531–540, 1995.
[14] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM Journal on Comp.*, 27(3):713–736, 1998. See also FOCS'95.
[15] G.S. Frandsen, T. Husfeldt, P.B. Miltersen, T. Rauhe and S. Skyum. Dynamic Algorithms for the Dyck Languages. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, pages 98–108, 1995.
[16] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.
[17] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.
[18] M. Gu, M. Farach and R. Beigel. An efficient algorithm for dynamic text indexing. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 697–704, 1994.
[19] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Comp.*, pages 63–78, 1977.
[20] E. McCreight. A space–economical suffix tree contruction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
[21] J. Meidanis and J. Setubal. Introduction to computational molecular biology. *PWS Publishing Company, a division of international Thomson publishing Inc.*, first print 1997.
[22] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer Verlag, Berlin, 1984.
[23] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
[24] S.C. Sahinalp and U. Vishkin. Efficient approximate results and dynamic matching of patterns using a label paradigm. In *Proc. 37th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 320–328, 1996.
[25] G.A. Stephen. String searching algorithms. *World Scientific publishing company*, 1995.
[26] Mikkel Thorup. Undirected single source shortest paths in linear time. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 12-21, 1997.
[27] A. K. Tsakalidis. Maintaining order in a generalized list. *Acta Informatica*, 21(1):101–112, 1984.
[28] E. Ukkonen. On–line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
[29] M.S. Waterman Introduction to computational biology. *Chapman and Hall*, Second printing 1996.
[30] P. Weiner. Linear pattern matching algorithm. In *IEEE Symp. on Switching and Automata Theory (now FOCS)*, pages 1–11, 1973.

APPENDIX A. DETERMINISTIC BLOCK DECOMPOSITION

In this section we recall the deterministic block decomposition of Mehlhorn *et al.* [23], which is based on a three-coloring technique of Goldberg *et al.* [17] which in turn is a generalization of a deterministic coin tossing algorithm of Cole and Vishkin [5].

In this section we only consider strings from $\mathbb{N}^*$. A string s is said to be *k-colored* if $i)$ $0 \leq s[i] < k$ for all i with $1 \leq i \leq |s|$, and $ii)$ s is repetition-free, *i.e.*, $s[i] \neq s[i+1]$ for all i with $1 \leq i < |s|$. A *k-coloring* of an N-colored string s is a k-colored string t with $|t| = |s|$.

Lemma 6 (Mehlhorn *et al.* [23])**.** *For every integer N there is a function $f : [-1..N-1]^{\log^* N+11} \rightarrow \{0,1\}$ such that for every N-colored string s of length n, the string $d \in \{0,1\}^n$ defined by $d[i] := f(\tilde{s}_{i-\log^* N-6}, \ldots, \tilde{s}_{i+4})$, where $\tilde{s}_j = s[j]$ for all j with $1 \leq j \leq n$ and $\tilde{s}_j = -1$ otherwise, satisfies:*

1. $d[i] + d[i+1] \leq 1$ *for* $1 \leq i < n$,
2. $d[i] + d[i+1] + d[i+2] + d[i+3] \geq 1$ *for* $1 \leq i < n-3$,

i.e., among two adjacent $d[i]$'s there is at most one 1, and among four adjacent $d[i]$'s there is at least one 1.

Proof. Here we only give the constructive algorithm for generating d. For the proof of correctness we refer to [23].

The construction of d proceeds in three phases. First a 6-coloring is constructed. In the second phase the 6-coloring is converted to a 3-coloring, which then in the third phase is used to construct d.

In the first phase we construct a sequence of colorings $X^0, X^1, \ldots$ of s, where $X^0 = s$ and X^i for $i > 0$ is inductively defined by

$$X^i[j] = \begin{cases} \mathrm{bit}(X^{i-1}[j], 0) & \text{if } j = 1\ , \\ \mathrm{bit}(X^{i-1}[j], p) + 2p & \text{otherwise,} \end{cases}$$

where $p = \min\{q | \mathrm{bit}(X^{i-1}[j], q) \neq \mathrm{bit}(X^{i-1}[j-1], q)\}$. It is straightforward to prove that if X^i is k-colored then X^{i+1} is $(1 + 2\lfloor \log k \rfloor)$-colored, and $X^i[j]$ only is a function of $s[j - i..j]$. It follows that if X^0 is N-colored, then $X^{\log^* N+2}$ is 6-colored [23]. Let $Y^6 = X^{\log^* N+2}$ be the 6-coloring of s.

In the second phase 5-, 4-, and 3-colorings Y^5, Y^4 and Y^3 of s are constructed by

$$Y^i[j] = \begin{cases} \min\{0,1,2\} \setminus \{Y^{i+1}[j-1], Y^{i+1}[j+1]\} & \text{if } Y^{i+1}[j] = i\ , \\ Y^{i+1}[j] & \text{otherwise,} \end{cases}$$

assuming $Y^{i+1}[0] = Y^{i+1}[n+1] = \infty$. It is straightforward to prove that $Y^3[j]$ only is a function of $Y^6[j-3, j+3]$, *i.e.*, of $s[j - \log^* N - 5, j + 3]$.

Finally let $d[j] = 1$ if only if $Y^3[j]$ is a local maximum. Formally,

$$d[j] = \begin{cases} 1 & \text{if } 1 < j < n \text{ and } Y^3[j-1] < Y^3[j] > Y^3[j+1]\ , \\ 0 & \text{otherwise.} \end{cases}$$

Again, it is straightforward to prove that $d[j]$ only is a function of $Y^3[j-1, j+1]$, *i.e.*, of $s[j - \log^* N - 6, j + 4]$. □

Note that the algorithmic construction in the proof of Lemma 6 requires time $\Theta(n \log^* N)$ to compute the string d (assuming $X^i[j]$ can be computed in $\mathcal{O}(1)$ time from $X^{i-1}[j-1..j]$). In Lemma 7 we prove that the construction can be done in linear time, provided a small table has been precomputed.

Lemma 7. *The construction of d in Lemma 6 can be done in $\mathcal{O}(n)$ time, provided a table of size $o(\log N)$ has been precomputed.*

Proof. The construction proceeds exactly as in the proof of Lemma 6, except that we in linear time construct $X^{\log^* N+2}$ directly from X^2, *i.e.*, $X^3, \ldots, X^{\log^* N+1}$ are not constructed. This reduces the total construction time to $\mathcal{O}(n)$.

The computation of $X^i[j]$ for $i = 1, 2$ can be done in $\mathcal{O}(1)$ time as follows. Let $z = X^{i-1}[j-1]$ XOR $X^{i-1}[j]$. Note that p is the index of the first nonzero in z. Let $z' = ((z-1)$ XOR $z)$ AND $z) = 2^p$. Then $\mathrm{bit}(X^{i-1}[j], p) = 1$ if and only if $X^{i-1}[j]$ AND $z' \neq 0$. The extraction of p from $z' = 2^p$ can be done in $\mathcal{O}(1)$ time by either converting z' into a floating pointer number and extracting the exponent by shifting [26], by a constant number of multiplications [16], or by a table lookup. See [16, 26] for further discussion.

The elements of X^2 are only $w = 2 + \lceil \log\log\lceil \log N \rceil \rceil$ bit integers and $X^{\log^* N+2}[j]$ depends only on $X^2[j - \log^* N..j]$, *i.e.*, $X^{\log^* N+2}[j]$ is a function of $w(1 + \log^* N)$ bits. Let g denote this function. The function g will be stored as a precomputed table of size $2^{w(1+\log^* N)} = o(\log N)$. Because each entry of g can be constructed in time $\mathcal{O}((\log^* N)^2)$ as in the proof Lemma 6, we have that g can be constructed in time $\mathcal{O}((\log^* N)^2 2^{w(1+\log^* N)}) = o(\log N)$ time.

In linear time we construct a string $\hat{X}$ such that $\hat{X}[i]$ corresponds to the concatenation of the bits in the substring $X^2[i - \log^* N..i]$ by

$$\begin{aligned} \hat{X}[0] &= X^2[0] \uparrow^w \text{ OR } X^2[0]\ , \\ \hat{X}[i] &= (\hat{X}[i-1] \uparrow^w \text{ OR } X^2[i]) \text{ AND } 1^{w(1+\log^* N)}\ . \end{aligned}$$

Note that to mark the beginning of the string we prepend $X^2[0]$ by itself. Because X^2 is repetition-free this representation is unique.

Finally, $X^{\log^* N+2}$ is computed in time $\mathcal{O}(n)$ from $X^{\log^* N+2}[i] = g(\hat{X}[i])$. □

Lemma 6 can be used to construct the block decomposition $encblock(s)$ of an N-colored string s by letting each $d[j] = 1$ mark the beginning of a new block b_i, where the first block always contains $s[1]$.

Lemma 8. *Let s be an N-colored string and $encblock(s) = b_1 \dots b_k$ the block decomposition of s defined above, then*

1. $1 \leq b_1 \leq 4$, *and* $2 \leq |b_i| \leq 4$ *for all* $2 \leq i \leq k$,
2. $k \leq \lceil |s|/2 \rceil$.

Appendix B. Split and Concatenate

The implementation of Split and Concatenate described below is similar to the implementation given in Section 4.2 of [23], but the implicit representation of persistent strings implies that the implementation details become very different. Lemma 9 below is an extension of Lemma 11 in [23].

Lemma 9. *Let s_1, s_2 and $s_3 = s_1 s_2$ be strings in $\mathcal{F}$. Let integers D_i^1, D_i^2, I_i^1, and I_i^2 be given by*

$$shrink_i(s_3) = \mathit{pref}_{D_i^1}(shrink_i(s_1))\ w_i\ \mathit{suf}_{D_i^2}(shrink_i(s_2)) \ ,$$

where $w_i \in \mathcal{E}^*$, $|w_i| = I_i^1 + I_i^2$, s_1 is a prefix of $\overline{shrink_i(s_3)[..|shrink_i(s_1)| - D_i^1 + I_i^1]}$, and D_i^1, D_i^2, and I_i^1 are chosen as small as possible.

Then $D_i^1 \leq \Delta_R + 2$, $D_i^2 \leq \Delta_L + 2$, $I_i^1 \leq \Delta_R + 3$, and $I_i^2 \leq \Delta_L + 1$.

Proof. The proof is by induction. For $i = 0$ the lemma trivially holds because $D_0^1 = D_0^2 = I_i = 0$.

For $i > 0$ consider $shrink_i(s_1)$. By the induction hypothesis at most a suffix of length D_{i-1}^1 of $shrink_{i-1}(s_1)$ can change when concatenating s_1 and s_2. Then at most a suffix of length $1 + D_{i-1}^1$ of $pow_{i-1}(s_1)$ changes. The marking in the block decomposition of $pow_{i-1}(s_1)$ can then only be different in a suffix of length $\Delta_\mathrm{R} + 1 + D_{i-1}^1$. Finally, only signatures in $shrink_i(s_1)$ can change which span a signature from the last $2 + \Delta_\mathrm{R} + D_{i-1}^1$ signatures of $pow_{i-1}(s_1)$. We have the constraint that $2(D_i^1 - 1) \leq 1 + \Delta_\mathrm{R} + D_{i-1}^1$, implying $D_i^1 \leq \Delta_\mathrm{R} + 5/2$, *i.e.*, $D_i^1 \leq \Delta_\mathrm{R} + 2$.

Symmetrically it follows that at most a prefix of $pow_{i-1}(s_2)$ with length $D_{i-1}^2 + 1$ can change and the markings at most change for a prefix of length $D_{i-1}^2 + 1 + \Delta_\mathrm{L}$. The signatures in $shrink_i(s_2)$ which can change are the signatures spanning signatures from the prefix of $pow_{i-1}(s_1)$ of length $D_{i-1}^2 + 1 + \Delta_\mathrm{L}$. We then get the constraint $2(D_i^2 - 2) \leq D_{i-1}^2 + 1 + \Delta_\mathrm{L} - 2$, implying $D_i^2 \leq \Delta_\mathrm{L} + 5/2$, *i.e.*, $D_i^2 \leq \Delta_\mathrm{L} + 2$.

If $shrink_{i-1}(s_3)$ contains a new substring w_{i-1} of length $I_{i-1}^1 + I_{i-1}^2$ then $pow_{i-1}(s_3)$ contains a new substring of length at most $(1 + I_{i-1}^1) + (I_{i-1}^2 + 1)$ where the first term of each of the sums is the smallest prefix of the substrings which can span $s_1[|s_1|]$. Then w_i at most spans this substring plus $1 + \Delta_\mathrm{R}$ signatures to the left of this string of which Δ_R get new markings and the Δ_L signatures to the right with new markings. The following constraints follow $2(I_i^1 - 2) \leq \Delta_\mathrm{R} + 1 + I_{i-1}^1 - 1$ and $2(I_i^2 - 1) \leq I_{i-1}^2 + 1 + \Delta_\mathrm{L} - 1$, implying $I_i^1 \leq \Delta_\mathrm{R} + 7/2$ and $I_i^2 \leq \Delta_\mathrm{L} + 3/2$, *i.e.*, $I_i^1 \leq \Delta_\mathrm{R} + 3$ and $I_i^2 \leq \Delta_\mathrm{L} + 1$. □

Lemma 10. Concatenate(s_1, s_2) *can be supported in time $\mathcal{O}((\Delta_L + \Delta_R)\log(|s_1| + |s_2|)$.*

Proof. Using Lemma 9 we can compute $s_3 = \textsc{Concatenate}(s_1, s_2)$ by only recomputing the encoding of the concatenation of the the last $\Delta_\mathrm{R} + 2$ signatures of $shrink_i(s_1)$ with the first $\Delta_\mathrm{L} + 2$ signatures of $shrink_i(s_2)$. We do this by a recursive procedure that takes two arguments x_1 and x_2. For the call at level i of the encodings, x_1 is the suffix of $shrink_i(s_1)$ of length $\Delta_\mathrm{L} + \Delta_\mathrm{R} + 2$ (or the complete signature string if it is shorter) and x_2 is the prefix of $shrink_i(s_2)$ of length $\Delta_\mathrm{L} + \Delta_\mathrm{R} + 2$ (or the complete signature string if it is shorter), *i.e.*, $x_1 x_2$ contains the signature substring to be recomputed and the left and right contexts

which influence the block decompositions. Let x_3 be the new encoding of x_1x_2 which is a substring of $shrink_i(s_3)$. The procedure returns x_3.

If $i = 0$ then $x_3 = x_1x_2$. Otherwise we compute the strings of signature powers $y_1 = Sig^{-1}(encblock^{-1}(Sig^{-1}(x_1)))$ and $y_2 = Sig^{-1}(encblock^{-1}(Sig^{-1}(x_2)))$, where $|y_1|, |y_2| \leq 4(\Delta_L + \Delta_R + 2)$. From y_1 we remove the last $\Delta_L + \Delta_R + 2$ signatures and from y_2 we remove the first $\Delta_L + \Delta_R + 2$ signatures. Let x'_1 respectively x'_2 be the deleted signatures from y_1 and y_2. Note that x'_1 and x'_2 are the suffix respectively the prefix of $shrink_{i-1}(s_1)$ and $shrink_{i-1}(s_1)$ of length $\Delta_L + \Delta_R + 2$, and $\overline{x_1x_2} = \overline{y_1x'_1x'_2y_2}$. From x'_1 and x'_2 we recursively compute the new encoding x'_3. A string of signature powers is created by concatenating y_1, x'_3 (by identifying signature powers) and y_2. By applying Sig to the resulting string we get a substring y_3 of $pow_{i-1}(s_3)$. Let $z = Sig(encblock(y_3))$, then $\overline{z} = \overline{x_1x_2}$. Because of the neighborhood dependence of the block decomposition, the prefix and the suffix of z may differ with the encoding of $shrink_i(s_3)$. By recycling the block decomposition given by $pref_{\Delta_R-2}(x_1)$ and $suf_{\Delta_L+2}(x_2)$, which by Lemma 9 should remain invariant under the concatenation, we can construct x_3.

By Lemma 7 the total time required for each of the $\mathcal{O}(\log(|s_1|+|s_2|))$ levels of the recurrence is $\mathcal{O}(\Delta_L + \Delta_R)$. □

Lemma 11. SPLIT(s, i) *can be supported in time* $\mathcal{O}((\Delta_L + \Delta_R) \log |s|)$.

Proof. Let $s_1 = s[..i-1]$ and $s_2 = s[i..]$. For level j, let i_j denote the index such that $shrink_j(s)[i_j]$ spans $s[i-1]$. Note $i_0 = i - 1$.

The implementation of SPLIT(S, i) is a recursive procedure that for each level j computes a suffix of $shrink_j(s_1)$ and a prefix of $shrink_j(s_2)$. The procedure for level j takes three arguments x_1, x_2, and ℓ, such that $x_1 = shrink_j(s)[i_j - \Delta_R - 2 - \Delta_L..i_j]$, $x_2 = shrink_j(s)[i_j + 1..i_j + \Delta_L + 1 + \Delta_R]$, and $\ell = |\overline{shrink_j(s)[..i_j]}|$, *i.e.*, x_1x_2 by Lemma 9 spans the substring of signatures that need to be recomputed plus Δ_L signatures to the left and Δ_R signatures to the right of this substring. The procedure returns y_1 and y_2 such that $\overline{y_1y_2} = \overline{x_1x_2}$ and y_1 is the suffix of $shrink_j(s_1)$ and y_2 is the prefix of $shrink_j(s_2)$.

If $j = 0$, then $y_1 = x_1$ and $y_2 = x_2$. Otherwise compute $z_1 = Sig^{-1}(encblock^{-1}(Sig^{-1}(x_1)))$ and $z_2 = Sig^{-1}(encblock^{-1}(Sig^{-1}(x_2)))$, where $|z_1| \leq 4(\Delta_L + \Delta_R + 3)$ and $|z_2| \leq 4(\Delta_L + \Delta_R + 1)$. Move the rightmost signatures of z_1 which only span substrings of s_2 to z_2. At most 4 signature powers need to be moved, where part of the last signature power may need to remain part of z_1. Let z_3 denote the signatures moved from z_1 to z_2. Then let $l' = \ell - |\overline{z_3}|$. Delete from z_1 the rightmost $\Delta_R + 3 + \Delta_L$ signatures, and let x'_1 denote the deleted substring. Delete from z_2 the rightmost $\Delta_R + 1 + \Delta_L$ signatures, and let x'_2 denote the deleted substring. Note $\overline{x_1x_2} = \overline{z_1x'_1x'_2z_2}$ and $\ell' = |\overline{shrink_{j-1}(s)[..i_j]}|$.

We can now for level $i - 1$ call the procedure recursively with x'_1, x'_2 and ℓ'. Let y'_1 and y'_2 be the returned signatures strings. Then $\overline{x_1x_2} = \overline{z_1y'_1y'_2z_2}$. Let z'_1 and z'_2 be the strings of signature powers given by $z'_1 = z_1y'_1$ and $z'_2 = y'_2z_2$. Similar to CONCATENATE we can now apply $encblock$ to find the block decomposition of z'_1 and z'_2. By combining the result with the block decomposition given by $pref_{\Delta_R-3}(x_1)$ and $suf_{\Delta_L+1}(x_2)$ and applying Sig we find y_1 and y_2.

By Lemma 7 the total time required for each of the $\mathcal{O}(\log |s|)$ levels of the recurrence is $\mathcal{O}(\Delta_L + \Delta_R)$. □

APPENDIX C. LONGEST COMMON PREFIX OF TWO STRINGS

Lemma 12. *Given two strings* s_1 *and* s_2, LCPREFIX(s_1, s_2) *and* COMPARE(s_1, s_2) *can be supported in* $\mathcal{O}(\log n)$ *time, and* LCSUFFIX(s_1, s_2) *can be supported in time* $\mathcal{O}(\log n \log^* N)$, *where* $n = \max\{|s_1|, |s_2|\}$.

Proof. To compute LCPREFIX(s_1, s_2) we compute $lcp(s_1, s_2)$. The search for $lcp(s_1, s_2)$ is done top-down on the signature encodings of s_1 and s_2 such that $lcp(pow_i(s_1), pow_i(s_2))$ and $lcp(shrink_i(s_1), shrink_i(s_2))$ are identified for all $i = 0, 1, \ldots, \min\{h(s_1), h(s_2)\}$. If we find that $pow_i(s_j) = lcp(pow_i(s_1), pow_i(s_2))$ or $shrink_i(s_j) = lcp(shrink_i(s_1), shrink_i(s_2))$ for $j = 1$ or 2, we have $s_j = lcp(s_1, s_2)$ and the search terminates.

The basic abstract steps of the lcp computation are the following. Assume $\ell = |lcp(shrink_i(s_1), shrink_i(s_2))|$ has been found and let ℓ' be given by $\overline{shrink_i(s_1)[..\ell]} = \overline{pow_{i-1}(s_1)[..\ell']}$. In a linear search find the smallest $j \geq \ell'$ such that $pow_{i-1}(s_1)[j+1] \neq pow_{i-1}(s_2)[j+1]$. Then $|lcp(pow_{i-1}(s_1), pow_{i-1}(s_2))| = j$. Let $(\sigma_1, \ell_1) = Sig^{-1}(pow_{i-1}(s_1)[j+1])$ and $(\sigma_2, \ell_2) = Sig^{-1}(pow_{i-1}(s_2)[j+1])$. Then

$$|lcp(shrink_{i-1}(s_1), shrink_{i-1}(s_2))| = \begin{cases} j & \text{if } \sigma_1 \neq \sigma_2 \ , \\ j + \min\{\ell_1, \ell_2\} & \text{otherwise.} \end{cases}$$

We first prove that the linear search only compares $\mathcal{O}(\Delta_{\mathrm{R}})$ signatures. Assume that $j' = j - \ell' \geq 5 + \Delta_{\mathrm{R}}$, *i.e.*, $pow_{i-1}(s_1)[..l' + 5 + \Delta_{\mathrm{R}}] = pow_{i-1}(s_2)[..l' + 5 + \Delta_{\mathrm{R}}]$. But then the marking produced by Lemma 6 of $pow_{i-1}(s_1)[..l' + 5]$ and $pow_{i-1}(s_2)[..l' + 5]$ are identical. This implies that the block decomposition of $pow_{i-1}(s_1)[\ell'+1..l'+5]$ and $pow_{i-1}(s_2)[\ell'+1..l'+5]$ are identical, and $shrink_i(s_1)[\ell+1] = shrink_i(s_2)[\ell+1]$ contradicting the assumption that $\ell = |lcp(shrink_i(s_1), shrink_i(s_2))|$. We conclude $j' \leq 4 + \Delta_{\mathrm{R}}$.

We now give a recursive implementation of LCPREFIX. For each level i we will not compute ℓ explicitly but only the following three values

$$\begin{aligned} m_i &= |\overline{lcp(shrink_i(s_1), shrink_i(s_2))}| \\ x_i &= shrink_i(s_1)[\ell + 1..\ell + K] \\ y_i &= shrink_i(s_2)[\ell + 1..\ell + K] \end{aligned}$$

where $K = 6 + \Delta_{\mathrm{R}}$ and $m_0 = |lcp(s_1, s_2)|$. Note that by definition $x_i[1] \neq y_i[1]$, provided that the search has not terminated because s_1 or s_2 is equal to $lcp(s_1, s_2)$. Let $x' = Sig^{-1}(x_i)$ and $y' = Sig^{-1}(y_i)$. Note that $j' = |lcp(x', y')| \leq 4 + \Delta_{\mathrm{R}}$, $(\sigma_1, \ell_1) = Sig^{-1}(x'[j'+1])$, and $(\sigma_2, \ell_2) = Sig^{-1}(y'[j'+1])$. Let $d = \min\{\ell_1, \ell_2\}$ if $\sigma_1 = \sigma_2$, otherwise let $d = 0$. We can now compute m_{i-1}, x_{i-1} and y_{i-1} by

$$\begin{aligned} m_{i-1} &= m_i + |\overline{lcp(x', y')}| + d|\overline{\sigma_1}| \\ x_{i-1} &= \mathrm{take}_K((\sigma_1, \ell_1 - d)Sig^{-1}(x'[j' + 2..K])) \\ y_{i-1} &= \mathrm{take}_K((\sigma_2, \ell_2 - d)Sig^{-1}(y'[j' + 2..K])) \ , \end{aligned}$$

where take_K returns the first K signatures from a string of signature powers. If the signature powers represent less than K signatures, take_K just returns these signatures. To prove that the constructed x_{i-1} and y_{i-1} are the required strings there are two cases to consider. If $|x_i| < K$ then x_{i-1} is the required string. If $|x_i| = K$ then observe that $|x'| \geq 2K - 1$ and at most $(4 + \Delta_{\mathrm{R}}) + 1$ signatures from x' can be matched. We get the constraint

$$2K - 1 - 5 - \Delta_{\mathrm{R}} \geq K \ ,$$

which is satisfied for $K \geq 6 + \Delta_{\mathrm{R}}$. For y_i the argument is similar.

Assume w.l.o.g. that $s_1 \neq s_2$. If the signature encodings of s_1 and s_2 have equal height, then the initial values of the search is $m_h = 0$, and x_h and y_h are the two root signatures, otherwise we first expand and apply take_K to the top levels of the highest tree to reach equal height.

When the recursion reaches the leaf level, we have found the position where s_1 and s_2 differs, and $x_0[1]$ and $y_0[1]$ contain the two characters. By comparing the two characters we find lexicographical ordering of s_1 and s_2. We conclude that LCPREFIX and COMPARE can be done in $\mathcal{O}(\Delta_{\mathrm{R}} \log n)$ time.

The implementation of LCSUFFIX is identical to LCPREFIX except that the expansion and comparison of signature strings in done from the right. The difference is that $|lcs(x', y')| \leq 3 + \Delta_{\mathrm{L}}$ and we have the constraint $2K - 1 - 4 - \Delta_{\mathrm{L}} \geq K$, implying that it is sufficient to have $k \geq 5 + \Delta_{\mathrm{L}}$ when computing LCSUFFIX. It follows that LCSUFFIX can be done in $\mathcal{O}(\Delta_{\mathrm{L}} \log n)$ time. □

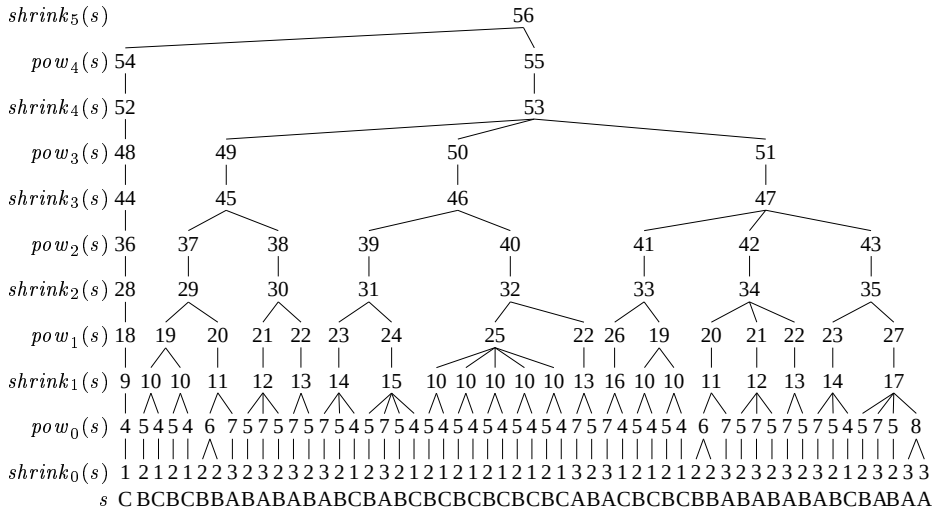

FIGURE 1. The signature encoding of a string of length 54 drawn as a tree, where the signatures $11 = Sig([6,7])$, $17 = Sig([5,7,5,8])$, $18 = Sig(9^1)$ and $25 = Sig(10^5)$.

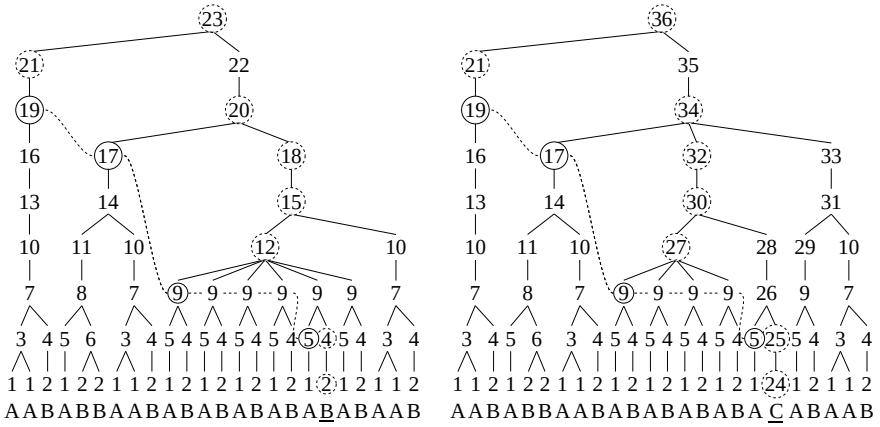

FIGURE 2. The computation of LCPREFIX of two strings.

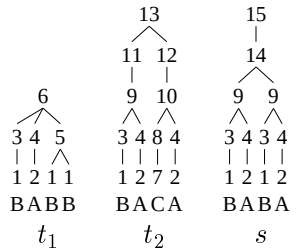

FIGURE 3. For $\mathcal{F} = \{t_1, t_2\}$ and s, the longest matching prefix of $shrink_i(s)$ with a string from F is with t_2 for $i = 0$ and with t_1 for $i = 0$.