Pattern Matching in Dynamic Texts*

Stephen Alstrup† Gerth Stølting Brodal‡ Theis Rauhe†

Abstract

Pattern matching is the problem of finding all occurrences of a pattern in a text. In a dynamic setting the problem is to support pattern matching in a text which can be manipulated on-line, *i.e.*, the usual situation in text editing.

We present a data structure that supports insertions and deletions of characters and movements of arbitrary large blocks within a text in $\mathcal{O}(\log^2 n \log\log n \log^* n)$ time per operation. Furthermore a search for a pattern P in the text is supported in time $\mathcal{O}(\log n \log\log n + occ + |P|)$, where occ is the number of occurrences to be reported. An ingredient in our solution to the above main result is a data structure for the *dynamic string equality* problem introduced by Mehlhorn, Sundar and Uhrig. As a secondary result we give almost quadratic better time bounds for this problem which in addition to keeping polylogarithmic factors low for our main result also improves the complexity for several other problems.

1 Introduction

Pattern matching on strings is the problem of determining all occurrences of a pattern string P as a substring of a larger text string T of length n. Optimal solutions achieving $\mathcal{O}(n)$ time for this problem were given in the 70s by Knuth, Morris, and Pratt [19], and Boyer and Moore [3]. Several text books, see *e.g.*, [1, 6, 25], address various pattern matching problems including the above classic problem. Originally the classic problem was motivated (among other things) in terms of text editing. In a text editing process it is desirable to effectively handle text updates and searches for different patterns, *i.e.*, avoid using time proportional to the full text for each text manipulation or search. Several papers including [11, 12, 13, 14, 18, 20, 28, 30, 24], describe data structures addressing various dynamic settings of efficient text manipulation and searching. In addition to text editing this work has also provided applications in other fields, *e.g.*, in computational biology [21, 29].

We present a data structure that supports insertions and deletions of characters and movements of arbitrary large blocks within a text in $\mathcal{O}(\log^2 n \log\log n \log^* n)$ time per operation. Furthermore a search for a pattern P in the text is supported in time $\mathcal{O}(\log n \log\log n + occ + |P|)$, where occ is the number of occurrences. Hence each text manipulation or search is supported in time polylogarithmic to the length of the text plus the necessary linear terms for input and output.

The data structure we present is based on the following closely related problem. A family of strings is maintained 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 argument s. The concatenate operation takes two strings s_1 and s_2 from the family, and inserts the concatenation $s_1 s_2$ into the family, again without preserving the arguments s_1 and s_2. Finally the search query supported for the family takes any string in the family and reports occurrences of this string within the other strings of the family. The query time is $\mathcal{O}(\log n \log\log n + occ)$, where n is the total size of the strings and occ the number of occurrences reported. The update operations are supported in $\mathcal{O}(\log^2 n \log\log n \log^* n)$ time. In the setting of text editing, *e.g.*, the problem of moving a block in a text, the family only consists of a single string representing the text. Movement of a block then consists of a constant number of split and concatenate operations. In order to search for a pattern P in the string (text) we in addition to the $\mathcal{O}(\log n \log\log n + occ)$ time for the search, also need additional $\mathcal{O}(|P|)$ time to construct a temporary version of P to be inserted in the string family such that it can be given as a parameter for the search operation.

Our main result, a fully dynamic pattern matching algorithm with polylogarithmic time per operations uses as a black box an algorithm for the *dynam-*

*Partially supported by the ESPRIT Long Term Research Program of the EU under contract 20244 (project ALCOM-IT). Part of this work was done while the last author was at BRICS.

†The IT University in Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, Denmark. E-mail: {stephen, theis}@itu.dk.

‡**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.

ic string equality problem [23]. As a secondary result we improve the time bounds of the results in [23]. Using the result of [23] our updates increases with a polylogarithmic factor. The *dynamic string equality* problem is a data structure to maintain a family of strings under *persistent* concatenate and split operations (the arguments are preserved in the family) such that the equality of two strings can be determined in constant time. We give an almost quadratic improvement of the time bounds for updates in [23]. In addition to this improvement we generalize the problem such that the equality query is strengthened to lexicographical order comparison between any pair of strings in the family within constant time. Furthermore we also support longest common prefix and suffix operations between a pair of strings in almost $\mathcal{O}(\log n)$ time. In [23], the problem is mainly motivated by problems in high-level programming languages like SETL. However subsequently this data structure has served as an important component for efficient solutions to other problems, which also benefit for our new bounds, see *e.g.*, [4, 15].

1.1 Related work In this section we sketch the history of pattern matching and refer to [14] for a more detailed account. Some of the early progress of making pattern matching dynamic is the *suffix tree*. In [20, 30] it is shown how to preprocess a text in linear time such that pattern matching queries can be answered on-line in $\mathcal{O}(|P| + occ)$ time. In [28] the suffix tree is extended such that the text can be extended by a single character at the end. Gu *et al.* [18] were the first to consider the problem where the text could be manipulated fully dynamically, and denoted this problem *dynamic text indexing*. The update operations they support are 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 $\mathcal{O}(|P| + occ \log i + i \log |P|)$ time, where i is the current number of updates performed. Ferragina [12] gave a more general solution that efficiently could handle insertions/deletions of a *string* into/from the text. The problem considered was denoted *incremental text editing*. Ferragina and Grossi [11, 13, 14] improved the result of Ferragina achieving time bounds $\mathcal{O}(n^{1/2} + s)$ for updates and $\mathcal{O}(|P| + occ)$ for the search, or updates in $\mathcal{O}(s(\log s + \log\log n) + \log n)$ time with query time $\mathcal{O}(|P| + occ + i \log p + \log\log n)$, where s is the length of the inserted/deleted string. Finally Sahinalp and Vishkin [24] gave the following result for incremental text indexing. Searches in $\mathcal{O}(p + occ)$ time and insert/delete of a string in $\mathcal{O}(\log^3 n + s)$ time.

1.2 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. We proceed in Section 3 with a description of our data structure for dynamic pattern matching. In Section 4 we provide the implementation for the generalised string equality problem.

1.3 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 $s[i]s[i+1]\ldots s[j]$ 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 $pref_k(s) = s[..|s|-k]$, $suf_k(s) = s[k+1..]$, and $inf_k(s) = s[k+1..|s|-k]$. The reverse string $s[|s|]\ldots s[2]s[1]$ is denoted s^R. For a mapping $f : \Sigma \to \mathcal{U}$, we extend $f : \Sigma^* \to \mathcal{U}^*$ by defining $f(a_1 a_2 \ldots a_n) = f(a_1)f(a_2)\ldots f(a_n)$. 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 without loss of generality 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 $Sig : \Sigma \cup (\mathcal{E}^1 \cup \mathcal{E}^2 \cup \mathcal{E}^3 \cup \mathcal{E}^4) \cup (\mathcal{E} \times \mathbb{N}) \hookrightarrow \mathcal{E}$. 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 $\mathcal{E}^*$, $shrink_0(s), pow_0(s), shrink_1(s), pow_1(s), \ldots, shrink_h(s)$ The strings are defined inductively by

$$\begin{aligned} shrink_0(s) &= Sig(s) \\ pow_0(s) &= Sig(encpow(shrink_0(s))) \\ &\vdots \\ shrink_j(s) &= Sig(encblock(pow_{j-1}(s))) \\ pow_j(s) &= Sig(encpow(shrink_j(s))) \\ &\vdots \\ shrink_h(s) &= Sig(encblock(pow_{h-1}(s))) \end{aligned}$$

where *encpow* and *encblock* are functions defined below, and h the height of the encoding of s which is the smallest value for which $|shrink_h(s)| = 1$. 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 $s = \sigma_1^{l_1} \ldots \sigma_m^{l_m}$, $\sigma_i \in \mathcal{E}$ where $\sigma_i \neq \sigma_{i+1}$ for $1 \leq i < m$. Then $encpow(s) = (\sigma_1, l_1), (\sigma_2, l_2), \ldots, (\sigma_m, l_m)$. The function $encpow(s)$ 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.

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 $\sigma_1 \ldots \sigma_k$ we let $\overline{\sigma_1 \ldots \sigma_k} = \overline{\sigma_1} \ldots \overline{\sigma_k}$.

The details of the block decomposition can be found in [23], from which 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 2.1. (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 $O(\log n \log^* N)$. Since the maximal block size is 4, Sig^{-1} can be computed in constant time if Sig^{-1} is stored as an array.

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 $|lcs(s_1, s_2)|$.

To be able to refer to the length of the string we in the following assume that each string length can be stored in a single word. We additionally assume that each signature σ has associated $|\overline{\sigma}|$. The following theorem summarizes our results in Section 4 for persistent strings.

THEOREM 2.2. *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 3.1. *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.*

Here, and in the following we have used the fact that $\mathcal{O}(\log^* N) = \mathcal{O}(\log^* n)$. The number of signatures N used to maintain $\mathcal{G}$ together with an auxilary internal family of strings, is always polynomially bounded in n, since each operation operation is polylarithmic in n and hence atmost introduce a polylogarithmic number of signatures.

The occurrences returned by the FIND operation are represented by *pointers* into the specific occurrences in lists representing the strings. For such a pointer we need (as usual) 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 search structure with the persistent string data structure.

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.

The strings in $\mathcal{G}$ and substrings of these (specified later) will be represented in a larger family of strings, denoted as $\mathcal{F}$. The family $\mathcal{F}$ will be maintained using the persistent string data structure, see Theorem 2.2. Hence we can efficiently concatenate, split, compare etc. the strings in $\mathcal{F}$. 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. To each string s in $\mathcal{G}$ we associate two values; its signature σ representing the string in $\mathcal{F}$, thus $\overline{\sigma} = s$, and a pointer to the tree structure associated to $l(s)$. These two values describe the interface between the pattern matching part in this section and the persistent data structure. Given σ and Sig^{-1} we can unpack the signature encoding $shrink_j(s)$ and $pow_j(s)$ for any level j. The tree structure can be used to access a node with index i in $l(s)[i]$ in $\mathcal{O}(\log n)$ time.

3.2 How to combine range search and persistent strings First we describe a simple method which combines the data structure for the persistent string data structure given in Section 2 with a dynamic two-dimensional orthogonal range search structure. The elements of this range search structure are pairs of strings from the persistent family of strings, with ordering provided through the lexicographic order of the strings (w.r.t. some arbitrary fixed ordering of the alphabet Σ). Our first simple approach for the dynamic pattern matching achieves the claimed search bound but without meeting the claimed time bounds

for split and concatenation. Next we extend this simple approach such that we obtain the claimed bounds for updates as well.

Consider a string s in our family of strings $\mathcal{G}$. For each index i in s assume that the two substrings $s[1..i-1]$ and $s[i..|s|]$, denoted the *context strings* for index i are in the string family $\mathcal{F}$. For a node $x \in l(s)$ with index i, we associate an *anchor*, denoted $\mathrm{Anc}(i)$, defined to be the triple $(s[1..i-1], s[i..|s|], x) \in \mathcal{F} \times \mathcal{F} \times L$. For all strings in $\mathcal{G}$ and indices in these, let R be the set of anchors kept in a dynamic range search structure. We claim that provided a string $w \in \mathcal{G}$, we can now efficiently report all occurrences of w within strings in $\mathcal{G}$. Choose any index i in w with anchor (a, b, x). Let \$ be a letter in the alphabet larger than letters occurring in strings for $\mathcal{G}$. The range search supported for R is now able to report anchors $(p, s, y) \in R$, where $a \leq_R p \leq_R \$a$ and $b \leq s \leq b\$$. Next we show that each reported anchor identify an occurrence, and each occurrence is reported once. For an anchor (p, s, y) reported we have that a is a suffix of p and b is as prefix of s. That is w occurs in the string $ps \in \mathcal{G}$ at index $i' - i + 1$, where $i' = \mathit{index}(y)$. We say the index i of w *aligns* with the index i' of ps. Finally, for each occurrence of w precisely one index in the occurrence aligns with an index i in w. The comparisons with respect to the lexicographic ordering among strings in $\mathcal{F}$ are done in worst-case constant time according to Theorem 2.2. Since the number of anchors equals the total length of strings in $\mathcal{G}$, this range search can be performed in time $\mathcal{O}(\log n \log\log n + occ)$ worst-case, see [8]. Furthermore constructing the strings $\$a$ and $b\$$ needed as the range bounds are done in time $O(\log n \log^* n)$ according to Theorem 2.2.

The problem with the above strategy is that concatenation and split operations on strings in $\mathcal{G}$ affects a number of anchors linear to the size of the updated strings. In order to avoid this we will limit the amount of indices we associate anchors to, together with a certain limitation on the lengths of the associated context strings. These limitations make extensive use of the properties with respect to the signature encodings of the strings.

3.3 Associating anchors to signatures Let $x = \mathit{shrink}_j(t)$ and $y = \mathit{shrink}_j(s)$ for a $j \geq 0$ in the signature encodings for two strings $s, t \in \mathcal{G}$. First we show how two indices $i \in x$ and $k \in y$ can align. The *offset* of an index i in x, denoted $\mathit{offset}_t^j(i) = |\overline{x[1..i-1]}| + 1$, is the index in t, where the signature $x[i]$ starts its encoding in t. Let s be a search string. We say index k aligns with index i if $s[1..\mathit{offset}_s^j(k)-1]$ is a suffix of $t[1..\mathit{offset}_t^j(i)-1]$, and $s[\mathit{offset}_s^j(k)..|s|]$ is a prefix of $t[\mathit{offset}_t^j(i)..|t|]$. For k aligned with i, we say this alignment is relative to the occurrence of s in t with offset $\mathit{offset}_t^j(k) - \mathit{offset}_s^j(i) + 1$. Note that for s a substring of t, it is not necessarily the case that $\mathit{shrink}_j(s)$ and $\mathit{shrink}_j(t)$ contains any aligned indices. However, choosing j sufficiently small this will be the case.

Let s be a substring of t. For level $j = 0$ every index in $\mathit{shrink}_j(s)$ aligns with an index in $\mathit{shrink}_j(t)$, corresponding to the approach given in Section 3.2. The context string which we will associate to an index at level j depend on the signature encoding at level j. Our goal is to maximize the level j thus minimizing the size of the signature encoding an anchor at that level depends on. However the level should still be small enough such that we can find an index which aligns. Fix $\Delta > \Delta_{\mathrm{L}} + \Delta_{\mathrm{R}} + 4 = \mathcal{O}(\log^* n)$. It is possible to show the following lemma.

LEMMA 3.1. *Let $t = t'st''$ and $s = s_1 s_2$ then*

i) $\mathit{shrink}_j(t) = \mathit{pref}_\Delta(\mathit{shrink}_j(t'))\, w_1\, \mathit{inf}_\Delta(\mathit{shrink}_j(s)) \cdot w_2\, \mathit{suf}_\Delta(\mathit{shrink}_j(t''))$, *where* $|w_1|, |w_2| \leq 2\Delta$,

ii) $\mathit{shrink}_i(s) = \mathit{pref}_\Delta(\mathit{shrink}_i(s_1))\, w_i \cdot \mathit{suf}_\Delta(\mathit{shrink}_i(s_2))$, *where* $|w_i| \leq 2\Delta$.

For every level j in the signature encoding it follows from lemma 3.1(i) that the (possible empty) infix $\mathit{protected}_j(s) = \mathit{shrink}_j(s)[\Delta + 1..|\mathit{shrink}_j(s)| - \Delta]$ must be a substring of $\mathit{shrink}_j(t)$.

Hence choosing j small enough such that $|\mathit{shrink}_j(s)| > 2\Delta$, we have $|\mathit{protected}_j(s)| > 0$. Thus for each occurrence of s in the string t, any index k in $\mathit{protected}_j(s)$ aligns with an index i in $\mathit{shrink}_j(t)$. We call the indices in substring $\mathit{protected}_j(s)$ within $\mathit{shrink}_j(s)$ for *protected* indices. The context strings associated to i should be large enough to cover the string s. Let the *left boundary* of an index i, denoted $\mathrm{lb}(i)$, be an index smaller than i. Similarly the *right boundary* of i, $\mathrm{rb}(i)$ is an index larger than i. Let l, p and r be the offsets of $\mathrm{lb}(i)$, i and $\mathrm{rb}(i)$ respectively. The *anchor* associated to i, $\mathrm{Anc}(i)$, is then the triple $(t[l..p-1], t[p..r-1], l(t)[p]) \in \mathcal{F} \times \mathcal{F} \times L$. Our goal is to minimize the distance of $\mathrm{lb}(i)$ and $\mathrm{rb}(i)$ from i, but still such that the context string associated i covers s. The larger we choose j, the smaller distance of the boundaries from i can be allowed. Hence j should be chosen as large as possible, but still small enough such that $|\mathit{protected}_j(s)| > 0$. However, we cannot ensure the length of $\mathit{shrink}_j(s)$ to be of bounded length for the maximal level with $|\mathit{protected}_j(s)| > 0$. That is $\mathrm{rb}(i) - \mathrm{lb}(i)$ need to be arbitrary large, implying

that we only can afford to have anchors to a subset of the indices at a given level. The idea is to exploit that there is a level where $|protected_j(s)| > 0$, and at the same time $shrink_j(s)$ only contains a few different signatures, *i.e.*, $pow_j(s)$ is of short length. Each index at level j for which we associate an anchor (with perhaps large context strings) is associated one of these different signatures, and hence the context strings only spans infixes with few anchors. In the following we formalize the above discussion.

Let $x = shrink_j(s)$. Define the set of *breakpoints* for x by $\mathrm{BP}(x) = \{\, i \mid x[i] \neq x[i+1] \,\}$. We consider two cases for a level j of the signature encoding of s.

Case 1 $|pow_j(s)| \leq 12\Delta$ for $j = 0$.

Case 2 $|shrink_j(s)| > 3\Delta$ and $|\mathrm{BP}(shrink_j(s))| \leq 12\Delta$ for some $j > 0$.

LEMMA 3.2. *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.

LEMMA 3.3. *Let $s, t \in \mathcal{G}$ and let j be such that Case 1 or Case 2 form Section 3 are satisfied. For any breakpoint i in $M(s)$ and any occurrence $(t, p) \in Occ(s, \mathcal{G})$, there exists $i' \in BP(shrink_j(t))$ such that i align with i' relative to occurrences of s in t with offset index$(p) = offset_t^j(i') - offset_s^j(i) + 1$.*

Proof. First if $j = 0$ with Case 1 satisfied the lemma is immediately true since all of $shrink_0(s)$ is a infix a position a offset p for each occurrence $(s, p) \in Occ(s, \mathcal{G})$.

Consider the case for $j > 0$ such that Case 2 satisfied. Let $(t, p) \in Occ(s, \mathcal{G})$ and $i \in M(s)$. Write t as $t = t_1\, s\, t_2$ where $|t_1| = index(p) - 1$. By Lemma 3.1(i) we have

(3.1)
$$shrink_j(t) = u\, shrink_j(s)[\Delta + 1..|shrink_j(s)| - \Delta]\, v$$

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

(3.2)
$$\overline{u} = t_1\, \overline{shrink_j(s)[..\Delta]}.$$

Since $i \geq \Delta + 1$ (it is in $M(s)$), we can write (3.1) 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 $i' = |u| + i - \Delta + 1$ is a breakpoint in $\mathrm{BP}(shrink_j(t))$. Furthermore using (3.2)

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

Hence $index(p) = |t_1| + 1 = offset_t^j(i') - offset_s^j(i) + 1$ and thus i' is the desired breakpoint aligned with i.

It is only the breakpoints we associate anchors. Anchors associated to the breakpoint in signature encodings at level j of all strings in $\mathcal{G}$ are kept in a range search structure denoted R_j. When a search for a string s is done we use the range search structure R_j for j chosen such that Case 1. or 2. are satisfied according to Lemma 3.2. Let $j = 0$ if Case 1 above is satisfied, or choose $j > 0$ as in the proof of the 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 $M(s)$, as the breakpoints in the infix $protected_j(s) = inf_\Delta(x)$, *i.e.*, $M(s) = \mathrm{BP}(x) \cap [\Delta + 1..|x| - \Delta]$. For Case 1 ($j = 0$), the protected breakpoints are simply all the breakpoints, *i.e.*, $M(s) = \mathrm{BP}(shrink_0(s))$. In this section we limit the exposition to the case where $M(s)$ 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 (tedious) case where $M(s)$ is empty, *i.e.*, s contains a long substring of small periodicity, is omitted.

Let $s \in \mathcal{G}$ be a infix of t. With the assumption that $|M(s)| > 0$, we have a breakpoint i in $protected_j(s)$. Hence in $shrink_j(t)$) there is an index i', with $\mathrm{Anc}(i')$, which aligns with i by Lemma 3.3). Then it suffices to show that the context string associated to $\mathrm{Anc}(i')$ covers all of s. In Lemma 3.4 we show this is satisfied by choosing $\mathrm{lb}(i) = \max(\{\, j \in \mathrm{BP}(x) \mid |[j..i] \cap \mathrm{BP}(x)| > 16\Delta \,\} \cup \{\, 1 \,\})$ and $\mathrm{rb}(i) = \min(\{\, j \in \mathrm{BP}(x) \mid |[i..j] \cap \mathrm{BP}(x)| > 16\Delta \,\} \cup \{\, |x| \,\})$. With sufficient large context strings we can find all occurrences using the range search structure R_j following the approach from Section 3.2. Write $s = s_1 s_2$, where $s_1 = s[1..offset_s^j(i) - 1]$. 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.

LEMMA 3.4. *Let $s, t \in \mathcal{G}$. Let i' be any breakpoint in $shrink_j(t)$ which aligns with index i in $shrink_j(s)$. Write $s = s_1 s_2$, where $s_1 = s[1..offset_s^j(i) - 1]$ and let $(t_1, t_2, e) = Anc(i') \in R_j$. Then $|s_1| \leq |t_1|$ and $|s_2| \leq |t_2|$, i.e., $lcs(s_1, t_1) = s_1$ and $lcp(s_2, t_2) = s_2$.*

Proof. Let p be the offset of the occurrence of s in t relative to the alignment of i' to i. Let $t = t'\,s\,t''$ such that $|t'| = p-1$. By Lemma 3.1(i) 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 $|w_1|, |w_2| \leq 2\Delta$. Let $v = shrink_j(t)[\mathrm{lb}(i')..i'-1]$. By the definition of an anchor $t_1 = \overline{v}$. Recall that i and i' are aligned and hence we only need to show that either $\mathrm{lb}(i') = 1$ or $\mathrm{lb}(i')$ is an index in $pref_\Delta(shrink_j(t'))$ in order to establish $|t_1| \geq |s_1|$. From definition of the left boundary we have $\mathrm{lb}(i') = 1$ or $|\mathrm{BP}(v)| = 16\Delta$. Since $|\mathrm{BP}(w_1 inf_\Delta(shrink_j(s)))| \leq |\mathrm{BP}(shrink_j(s))| + 2\Delta \leq 14\Delta$ according to Lemma 3.2, $\mathrm{lb}(i')$ must be an index in $pref_\Delta(shrink_j(t'))$. A similar argument shows $|s_2| \leq |t_2|$.

3.4 Searching A search operation is carried out in three steps:

1. Find level j according to Lemma 3.2, and a breakpoint $i \in M(s)$. Compute the offset $p = offset_s^j(i)$ of i in s.
2. Construct and insert the strings $s_1 = s[1..p-1]$, $s_2 = s[p..|s|]$, $\$s_1$ and $s_2\$$ into $\mathcal{F}$ using the SPLIT and CONCATENATE operations on $s \in \mathcal{F}$.
3. Report occurrences (represented as nodes in L) using the range search structure R_j.

In order to determine the quantities in step 1. above we show that it is sufficient to examine a portion of size $\mathcal{O}(\Delta)$ of the signature encoding of s.

First if $|pow_0(s)| \leq 12\Delta$ we let $j = 0$. Otherwise we expand the signature strings of s starting from the root signature until we reach a level j such that Case 2 is satisfied for j. Then by the Lemma 3.5 below, we can efficiently derive the quantities i and p needed in addition to j for step 1. Recall from Section 2 that we can expand a signature string to the next level using the inverse mapping Sig^{-1} in time linear to the length of the expanded string. Hence the total time to expand level by level until the string $pow_j(s)$ is expanded for a level j satisfying $|shrink_j(s)| > 3\Delta$ and $|\mathrm{BP}(shrink_j(s))| = |pow_j(s)| \leq 12\Delta$ (Case 2), is bounded by the total length of these expanded signature strings, *i.e.*, bounded by $\mathcal{O}(|pow_j(s)|) = \mathcal{O}(\Delta)$. Note that by Lemma 3.2 Case 2 will be satisfied at some stage in the absence of Case 1. In order to effectively test whether a level j satisfies Case 2, we need to test whether the length of $shrink_j(s)$ exceeds 3Δ without actually expanding it to its full length (unbounded in terms of Δ). By i) in the lemma below we can find the length of $shrink_j(s)$ in time $\mathcal{O}(\Delta)$ on basis of the expanded string $pow_j(s)$. Finally ii) and iii) of this lemma provide us with the remaining quantities for step 1.

LEMMA 3.5. *In time linear to the length of $pow_j(s)$ we can determine the following: i) the length of $shrink_j(s)$, ii) the first protected breakpoint $i \in M(s)$ (if it exists), iii) the offset $offset_s^j(i)$ of the breakpoint i.*

Proof. Let $m = |pow_j(s)|$. In time $\mathcal{O}(m)$ we can compute the list $Sig^{-1}(pow_j(s)) = (\sigma_1, l_1), (\sigma_1, l_2), \ldots, (\sigma_m, l_m)$ where each pair (σ_k, l_k) corresponds to substring $\sigma_k^{l_k}$ in $shrink_j(s)$ according to the definition of the signature encoding in Section 2. i) is simply determined by computing the sum $\sum_{k=1}^{l=m} l_k = |shrink_j(s)|$. ii) Let $u = \min_w \sum_{k=1}^{k=w} l_k > \Delta$. Then the first protected breakpoint is $i = \sum_{k=1}^{k=u} l_k$, and hence i can be determined by summing at most Δ terms l_k to obtain such u and i. Note that if $|shrink_j(s)| - i < \Delta$ there is no such breakpoint in $M(s)$ which we assumed not occurs. iii) To each signature σ_k $1 \leq k \leq m$, the signature encoding of s provide us with the lengths of expanded strings $\overline{\sigma_k}$. Hence we can determine the sum $offset_s^j(i) = (\sum_{k=1}^{k=u-1} l_k |\overline{\sigma_k}|) + (l_u - 1)|\overline{\sigma_u}| + 1$ in time bounded by the number of these terms, bounded by $\mathcal{O}(m)$.

We conclude that the time to find j satisfying Case 1. or 2., expand the signature strings until $pow_j(s)$, and the computation of the quantities for step 1. on basis of this string, by the discussion above and Lemma 3.5 takes time $\mathcal{O}(\Delta)$.

Let $s_1 = s[1..p-1]$, and $s = s_1 s_2$. According to Theorem 2.2 we can in time $\mathcal{O}(\log n \log^* n)$ construct and insert $s_1, s_2, \$s_1, s_2\$$ into $\mathcal{F}$. With these strings in $\mathcal{F}$ we can perform step 3, using the approach from Section 3.2, in time $\mathcal{O}(\log n \log\log n + occ)$ which dominates the total search time. If P is an external string, we use additional $\mathcal{O}(|P|)$ time for preprocessing P, the lexicographic order of P with any other string in $\mathcal{F}$ can checked in constant time.

3.5 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.

3.5.1 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 search structures R_j is considered as nodes in $l(s)$ instead.

The main part of the computation consists of restructuring the various range search structures R_j such that they contain anchors with respect to the new context information relevant for s. From Lemma 3.1(ii) we have that

$$\mathit{shrink}_i(s) = \mathit{pref}_\Delta(\mathit{shrink}_i(s_1))\, w_i\, \mathit{suf}_\Delta(\mathit{shrink}_i(s_2)),$$

where $|w_i| \leq 2\Delta$. We will only be concerned with associating anchors properly to the breakpoints (the constructing of $\mathit{shrink}_i(s)$ is a part of the persistent data structure). We have

- The anchors associated to the suffix and prefix of length Δ to respectively $\mathit{shrink}_i(s_1)$ and $\mathit{shrink}_i(s_2)$ should be deleted from R_i.
- The anchors in $\mathit{pref}_\Delta(\mathit{shrink}_i(s_1))$ and $\mathit{suf}_\Delta(\mathit{shrink}_i(s_2))$ which are depended on w_i should be updated.
- New anchors should be associated to the $\mathcal{O}(\Delta)$ breakpoints in w_i.

We will describe how to associate anchors to breakpoints in w_i and construct new anchors for those outside w_i with affected context. Removal of anchors are done in a similarly way and thus omitted. First we show that at most $\mathcal{O}(\Delta)$ new anchor have to be constructed. An anchor to a breakpoint k does by definition only depend of index l the number of breakpoints between k and l are $\mathcal{O}(\Delta)$. Hence, at most $\mathcal{O}(\Delta)$ breakpoints in $\mathit{shrink}_i(s_1)$ and $\mathit{shrink}_i(s_2)$ should have their associated anchors updated, thus bounding the total number of new anchors to be created to $\mathcal{O}(\Delta)$. By the same argument it follows that an infix $IP(i)$ of $\mathit{pow}_i(s)$ of size $\mathcal{O}(\Delta)$ covers all these breakpoints in $\mathit{shrink}_i(s)$, and including the indices these context strings depends on. Similar to the search routine above we will update anchors in $\mathit{shrink}_i(s)$ using $\mathit{pow}_i(s)$. However we only expand to the next level in order to get $IP(i)$. The technique to do this is (tedious and) similar to the search routine, and omitted here. Let l be the left most index in $IP(i)$. Given the offset $\mathit{offset}_s^i(l-1)$ and $IP(i)$ we proceed to show how to compute the new anchors. Each signature in $IP(i)$ represents a breakpoint for an index in $\mathit{shrink}_i(s)$. Denote the anchor for index k in $\mathit{shrink}_i(s)$ as $\mathrm{Anc}'(k)$. Scanning $IP(i)$ we detect $\mathrm{lb}(k)$ and $\mathrm{rb}(k)$ in $\mathcal{O}(\Delta)$ time. Using Lemma 3.5 and adding $\mathit{offset}_s^i(l-1)$ we get the offsets $\mathit{offset}_s^i(k)$, $\mathit{offset}_s^i(\mathrm{lb}(k))$, and $\mathit{offset}_s^i(\mathrm{rb}(k))$ in time $\mathcal{O}(\Delta)$. Using the tree structure associated to the list $l(s)$ we compute the node $l(s)[\mathit{offset}_s^i(k)]$ in time $\mathcal{O}(\log n)$. Finally by applying the persistent SPLIT operation on $s \in \mathcal{F}$ for the offsets, the two context strings for the anchor are generated in $\mathcal{F}$ in $\mathcal{O}(\log n \log^* n)$ time according to Theorem 2.2. The new anchor is inserted in the range search structure R_i in time $\mathcal{O}(\log n \log\log n)$, see [8].

In total, at each of the $\mathcal{O}(\log n)$ levels we update $\mathcal{O}(\Delta)$ anchors in time $\mathcal{O}(\log n \log^* n)$ and insert/delete these in an range search structure in time $O(\log n \log\log n)$, summing to $\mathcal{O}((\log n \log^* n) \cdot (\log n \log^* n + \log n \log\log n)) = \mathcal{O}(\log^2 n \log\log n \log^* n)$.

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 of a signature in a signature string which need to be considered by the different operations can be constructed when required.

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 4.1 below, improving and extending the result of Mehlhorn *et al.* [23].

LEMMA 4.1. *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 $\mathit{Sig}(a)$, 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 other operations will be given in the full version of the paper.

4.1 Maintaining strings sorted In this section we prove Theorem 2.2, *i.e.*, we describe how to reduce the time for performing comparisons on persistent strings 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* 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 INSERT(x, y), DELETE(x) and ORDER(x, y) operations in worst-case $\mathcal{O}(1)$ time. The operation INSERT(x, y) inserts element y after x in the list, and DELETE(x) deletes x from the list. The query 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 COMPARE(s_1, s_2) query can now be answered in worst-case $\mathcal{O}(1)$ time by applying EQUAL(s_1, s_2) and by applying 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 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. By maintaning a collection of tries we can avoid the $\log m$ factor. Details are left for the full version of the paper.

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.

[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.