Space-Efficient String Mining under Frequency Constraints

Johannes Fischer
Universiät Tübingen
Algorithms in Bioinformatics
Sand 14, D-72076 Tübingen
fischer@informatik.uni-tuebingen.de

Veli Mäkinen and Niko Välimäki
Department of Computer Science
University of Helsinki
Helsinki, Finland
{vmakinen|nvalimak}@cs.helsinki.fi

Abstract

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two databases (i.e. multisets) of d strings, over an alphabet Σ, with overall length n. We study the problem of mining discriminative patterns between $\mathcal{D}_1$ and $\mathcal{D}_2$ — e.g., patterns that are frequent in one database but not in the other, emerging patterns, or patterns satisfying other frequency-related constraints. Using the algorithmic framework by Hui (CPM 1992), one can solve several variants of this problem in the optimal linear time with the aid of suffix trees or suffix arrays. This stands in high contrast to other pattern domains such as itemsets or subgraphs, where super-linear lower bounds are known. However, the space requirement of existing solutions is $O(n \log n)$ bits, which is not optimal for $|\Sigma| \ll n$ (in particular for constant $|\Sigma|$), as the databases themselves occupy only $n \log |\Sigma|$ bits.

Because in many real-life applications space is a more critical resource than time, the aim of this article is to reduce the space, at the cost of an increased running time. In particular, we give a solution for the above problems that uses $O(n \log |\Sigma| + d \log n)$ bits, while the time requirement is increased from the optimal linear time to $O(n \log n)$. Our new method is tested extensively on a biologically relevant datasets and shown to be usable even on a genome-scale data.

1. Introduction

In many applications, e.g., in computational biology, the goal is to find interesting string patterns that discriminate well between two classes of data. Application areas are, among others, finding discriminative features for sequence classification or segmentation [4], discovering new binding motifs of transcription factors [6], or computation of the classical ranking scores in Information Retrieval [3].

In this paper, we focus on string mining under frequency constraints, i.e., predicates over patterns depending solely on the frequency of their occurrence in the data [13]. This category encompasses combined minimum/maximum support constraints, constraints concerning emerging substrings, and other constraints concerning statistically significant substrings. While most of these problems have their motivation in itemset mining [1], data miners also consider them for the domain of strings [8], as plenty of naturally occurring data can be modeled as strings (biological sequences, MIDI-data, etc.).

We concentrate on the fundamental instance of exact substring patterns, where optimal linear time algorithms can be obtained, which stands in high contrast to other pattern domains such as itemsets or sub-graphs, where super-linear lower bounds are known [33, 9]. Much of the related work in the domain of strings studies more complicated pattern classes, where the search space is typically of exponential size, and the objective is to optimize the time needed per each output element satisfying the frequency constraints, see e.g. [28, 2] for recent results on this line of research.

Our objective is to provide practical tools for the mining of very large data sets, such as the genome-scale sequences of molecular biology. For such data sets, one needs to pay special attention to the space usage. Even if the algorithm takes linear space proportional to the overall length n of sequences in the database, this may be too much: Measured in bits, a data structure having $O(1)$ integers per text character occupies asymptotically $O(n \log n)$ bits, whereas the database can be stored in $n \log |\Sigma|$ bits, where Σ is the underlying alphabet. Especially on DNA sequences (where $|\Sigma| = 4$) this is a significant difference.

1.1. Contributions of our work

In this paper, we show that frequency constrained mining tasks on exact substring patterns can be solved in much less space than previously known. We improve the known $O(n \log n)$ bits space usage into $O(n \log |\Sigma| + d \log n)$ bits with a logarithmic penalty in computation time against the optimal linear time algorithm [13]. Here, $d \ll n$ is the

number of strings in the databases. We emphasize that our algorithmic framework is general enough to handle *all* data mining tasks whose predicates are based on the frequency of strings alone (e.g., frequent substrings, emerging substrings, strings passing the χ^2-test, ...); this approach is much more general than designing individual solutions for each of these tasks seperately.

We have also tested our method empirically on realistically-sized data sets from computational biology and shown that in practice space is reduced by a factor of 6–7 compared to the optimal algorithm [13], while the running time is increased by a factor of about 80–90. Given that users are usually willing to wait longer if they can handle larger data sets in exchange, the increase in running time is compensated by the fact that due to the use of more input data (which is nowadays available in fast- evolving domains such as computational biology), the results of the mining tasks will be more significant in practice.

Sadakane [31] gives another succinct version for calculating frequencies. However, his problem setting is quite different from ours, as he designs a succinct *index* that allows to answer frequency queries for a given *pattern*. Our work, on the other hand, is situated in the field of data mining, where the goal is to extract interesting strings from statistical constraints alone. Because Sadakane's index needs $O(n \log n)$ bits at construction time [31], we cannot use it for our task, as it would result in no advantage at all over the non-succinct version. Moreover, our algorithm can be modified to give a space-efficient algorithm to build a part of Sadakane's succinct index.

1.2. Outline

In the following, we first give the formal definitions of the mining tasks we consider (§2). In §3, we explain the existing optimal algorithm. Then we show how the optimal algorithm can be carefully re-engineered to use less space (§4). The paper continues with other space/time-tradeoffs that can be obtained for the problem (§5), and with applications to Sadakane's succinct index for storing document frequencies. Experiments are reported in §6.

2. Preliminaries

For a finite ordered alphabet Σ, a *(text) string* $T \in \Sigma^\star$ is a chain $T_1 \dots T_n$ of letters $T_i \in \Sigma$. Here, $\Sigma^\star$ is the set of all strings over Σ. We write $T_{i..j}$ to denote the *substring* of T ranging from position i to j. We use $|T_{i..j}|$ to denote the length $j - i + 1$ of $T_{i..j}$. Substrings $T_{1..j}$, $1 \le j \le n$, are called *prefixes* and substrings $T_{i..n}$, $1 \le i \le n$, are called *suffixes* of T. For strings $\phi, \psi \in \Sigma^\star$ we write $\phi \preceq \psi$ if ϕ is a substring of ψ. Then $\mathrm{lcp}(\phi, \psi)$ gives the *length* of the *longest common prefix* (lcp) of ϕ and ψ. For example, $\mathrm{lcp}(\texttt{aab}, \texttt{abab}) = 1$. When clear from the context, we also use lcp for the longest common prefix itself (not its length). Given a multiset $\mathcal{D} \subseteq \Sigma^\star$ with strings over Σ (called *database*), we write $|\mathcal{D}|$ to denote the number of strings in $\mathcal{D}$, and $\|\mathcal{D}\|$ to denote their total length, i.e., $\|\mathcal{D}\| = \sum_{\phi \in \mathcal{D}} |\phi|$. We define the *frequency* of a *pattern* $\phi \in \Sigma^\star$ in $\mathcal{D}$ as follows:

$$\mathsf{freq}(\phi, \mathcal{D}) := |\{d \in \mathcal{D} \,:\, \phi \preceq d\}|$$

Note that this is not the same as counting all occurrences of a ϕ in $\mathcal{D}$, because one database entry could contain multiple occurrences of ϕ. In data mining applications it is important to use this definition of frequency, as one is usually looking for patterns that are highly relevant for the *whole* database, and not only for one or a few of its entries.

Now the *support* of a *pattern* $\phi \in \Sigma^\star$ in $\mathcal{D}$ can be defined as

$$\mathsf{supp}(\phi, \mathcal{D}) := \frac{\mathsf{freq}(\phi, \mathcal{D})}{|\mathcal{D}|} \ .$$

The first example problem that can be solved with our method is as follows (cf. [1]).

Problem 1 *Given m databases $\mathcal{D}_1, \dots, \mathcal{D}_m$ of strings over Σ, and m pairs of support thresholds $(\rho_i, \tau_i)_{i=1,\dots,m}$ satisfying $0 < \rho_i \le \tau_i \le 1$ for all i, the* Frequent Pattern Mining Problem *is to return all strings $\phi \in \Sigma^\star$ that satisfy $\rho_i \le \mathsf{supp}(\phi, \mathcal{D}_i) \le \tau_i$ for all $1 \le i \le m$.* ◇

As another example mining problem that can be solved with our algorithm, we consider a 2-class problem for a (positive) database $\mathcal{D}_1$ and a (negative) database $\mathcal{D}_2$. For this, we define the *growth-rate* from $\mathcal{D}_2$ to $\mathcal{D}_1$ of a string ϕ as

$$\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) := \frac{\mathsf{supp}(\phi, \mathcal{D}_1)}{\mathsf{supp}(\phi, \mathcal{D}_2)}, \text{ if } \mathsf{supp}(\phi, \mathcal{D}_2) \neq 0 \ ,$$

and $\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) = \infty$ otherwise. The following definition is motivated by the problem of mining Emerging Itemsets [10]:

Problem 2 *Given two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ of strings over Σ, a support threshold ρ_s ($1/|\mathcal{D}_1| \le \rho_s \le 1$), and a minimum growth rate $\rho_g > 1$, the* Emerging Substrings Mining Problem *is to find all strings $\phi \in \Sigma^\star$ such that $\mathsf{supp}(\phi, \mathcal{D}_1) \ge \rho_s$ and $\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \ge \rho_g$.* ◇

The patterns satisfying both the support- and the growth-rate condition are called *Emerging Substrings* (ESs). ESs with an infinite growth-rate are called *Jumping Emerging Substrings* (JESs), because they are highly discriminative

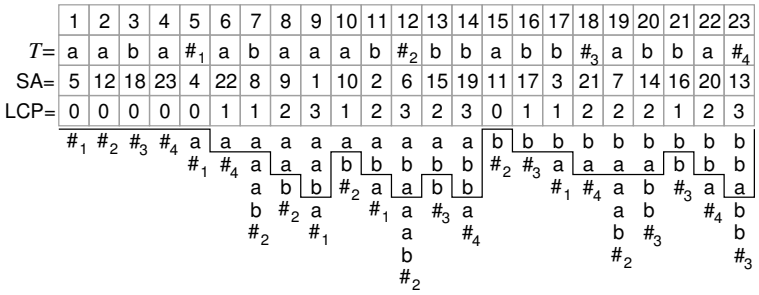

Figure 1. The suffix array for T and its lcp-table. Below position i we draw the string $T_{\mathsf{SA}[i]..n}$ until reaching the first end-of-string marker. The solid line going through these strings indicates the lcp-values.

for the two databases.[1]

Example 1 *Let* $\mathcal{D}_1 = \{\texttt{aaba}, \texttt{abaaab}\}$, $\mathcal{D}_2 = \{\texttt{bbabb}, \texttt{abba}\}$, $\rho_s = 1$, *and* $\rho_g = 2$. *Then the emerging substrings from* $\mathcal{D}_2$ *to* $\mathcal{D}_1$ *are* `aa`, `aab` *and* `aba`. *These are also the JESs.* ◇

2.1. Suffix- and lcp-arrays

This section introduces two fundamental data structures that we need for our algorithm.

The *suffix array* SA (see [26]) for T describes the lexicographic order of T's suffixes, in the sense that it "enumerates" the suffixes from the smallest to the largest. More formally, $\mathsf{SA}[1, n]$ is an array of integers s.t. its entries form a permutation of $[1 : n]$, and $T_{\mathsf{SA}[i]..n}$ is lexicographically less than $T_{\mathsf{SA}[i+1]..n}$ for all $1 \leq i < n$. The *generalized suffix array* of two databases $\mathcal{D}_1 = \{\phi_1, \ldots, \phi_{|\mathcal{D}_1|}\}$ and $\mathcal{D}_2 = \{\psi_1, \ldots, \psi_{|\mathcal{D}_2|}\}$ of strings is simply the suffix array built on top of the concatenated string

$$T := \phi_1 \#_1 \ldots \phi_{|\mathcal{D}_1|} \#_{|\mathcal{D}_1|} \psi_1 \#_{|\mathcal{D}_1|+1} \ldots \psi_{|\mathcal{D}_2|} \#_{|\mathcal{D}_1|+|\mathcal{D}_2|}.$$

The $\#_j$'s are (conceptual) new characters (lexicographically smaller than other characters) to separate the individual strings. See Fig. 1 for an example, which builds on the databases $\mathcal{D}_1$ and $\mathcal{D}_2$ from Ex. 1.

The suffix array for T can be computed in $O(n)$ time, either indirectly by constructing a suffix tree for T, or directly with some recent methods, e.g. [21].

Instead of suffix array SA we can use a compressed suffix array [27]. Different tradeoffs between space and access time to $\mathsf{SA}[i]$ are possible, e.g. one can access any $\mathsf{SA}[i]$ within time $t_{\mathsf{SA}} = O(\log^\epsilon n)$ for an arbitrary $0 < \epsilon \leq 1$, using an index of size $O(n(H_0 + \log\log|\Sigma|)) = O(n \log|\Sigma|)$ bits [15, 29]. Here $H_0 = H_0(T) \leq \log|\Sigma|$ is the 0*'th order empirical entropy* of the text T (lower bound for the average number of bits needed to code a symbol using a fixed code table).

The lcp-array $\mathsf{LCP}[1, n]$ for T is defined by $\mathsf{LCP}[i] = \mathrm{lcp}(T_{\mathsf{SA}[i]..n}, T_{\mathsf{SA}[i-1]..n})$ for all $1 < i \leq n$, and $\mathsf{LCP}[1] = 0$. That is, LCP contains the lengths of the longest common prefixes of T's suffixes that are consecutive in lexicographic order. Kasai et al. [20] gave an algorithm to compute LCP in $O(n)$ time. Sadakane [30] shows that representing the LCP-array can be done using $2n + o(n)$ bits (and constructed within that space), while accessing $\mathsf{LCP}[i]$ then takes $O(t_{\mathsf{SA}})$ time.

2.2. Range Minimum Queries

Another tool we need for our approach is a preprocessing of the lcp-array such that *range minimum queries* (RMQs) can be answered in constant time. The reason for using RMQs on LCP is that they generalize the lcp-array, in the sense that we can compute the lcp between arbitrary suffixes, and not only between those that are lexicographically adjacent. Formally, for two given indices i and j the query $\mathrm{RMQ}_{\mathsf{LCP}}(i, j)$ asks for the position of the minimum element in $\mathsf{LCP}[i, j]$, i.e., $\mathrm{RMQ}_{\mathsf{LCP}}(i, j) := \mathrm{argmin}_{k \in \{i,\ldots,j\}}\{\mathsf{LCP}[k]\}$. We return the smallest index if the minimum is not unique.

Lemma 1 *Let* $T \in \Sigma^\star$ *be a text and* LCP *be the lcp-array for* T. *Then for all* $1 \leq i < j \leq |T|$, $\mathrm{lcp}(T_{\mathsf{SA}[i]..|T|}, T_{\mathsf{SA}[j]..|T|})$ *is given by* $\mathsf{LCP}[\mathrm{RMQ}_{\mathsf{LCP}}(i + 1, j)]$.

This follows immediately from the definition of the lcp-array. Stated differently, Lemma 1 says that the i'th-

[1] Notice that to reduce the size of the output, it is customary to consider so-called left-maximal and right-maximal strings (those that cannot be extended to any direction to obtain another string satisfying the thresholds). For conciseness, we do not consider these enhancements. However, right-maximality is trivial to accommodate in our algorithms, and left-maximality requires more care, but can be solved within the same time/space bounds.

and the j'th-smallest suffix of t are equal in exactly their $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(i+1,j)]$ first characters.

It is well known that a linear preprocessing of any input array A is sufficient to find $\text{RMQ}_A(i,j)$ in time $O(1)$. This method has recently been refined to use only $2n + o(n)$ bits in addition to the input array, also at construction time [12]. With the succinct representation of the lcp-array, we thus need $4n + o(n)$ space to answer arbitrary lcp-queries in $O(t_{\mathsf{SA}})$ time.

3. Basic Mining Algorithm

This section reviews the basic algorithm for computing the string frequencies. It is a tight integration of Kasai et al.'s algorithm for visiting all *branching*[2] substrings of a text [20], and Hui's *color set size* technique [18]. Note that it is enough to visit all branching substrings, as by definition the frequencies of all other strings are equal to the frequency of their longest branching prefix. From now on, let T denote the string formed from the (positive and negative) databases $\mathcal{D}_1$ and $\mathcal{D}_2$ as explained in §2. Let d denote the total number of strings in the databases ($d = |\mathcal{D}_1| + |\mathcal{D}_2|$), and n denote T's length ($n = \|\mathcal{D}_1\| + \|\mathcal{D}_2\| + d$).

3.1. Visiting All Branching Substrings

First, we summarize the algorithm for visiting all branching substrings [20]. The idea is to simulate a depth-first traversal of the (virtual) suffix tree, solely by means of the suffix- and lcp-array. This works by visiting the leaves of the suffix tree in lexicographic order (i.e., in the order of the suffix array), keeping on a stack R just the *rightmost path* of the part of the suffix-tree that has been seen so far. Step i first deletes the elements from R that are removed from the rightmost path, and then inserts new elements to R. The details are as follows.

Consider step $i+1$ of the algorithm, so we are just about to visit suffix $\mathsf{SA}[i+1]$ (see also Fig. 2). The stack R contains the lengths of the prefixes of $T_{\mathsf{SA}[i]..n}$ that are branching (the so-called *string-depths* of nodes on the rightmost path). Then we pop all elements from R whose string-depth is greater than $\mathsf{LCP}[i+1]$, because $\mathsf{LCP}[i+1]$ is the string-depth of the lowest common ancestor (lca) v of the leaves represented by $\mathsf{SA}[i]$ and $\mathsf{SA}[i+1]$. If v is not already present in R, we push it on R (with string-depth $\mathsf{LCP}[i+1]$). Finally, we push $\mathsf{SA}[i+1]$ on R (with string-depth $n - \mathsf{SA}[i+1] + 1$). It is shown in [20] that this algorithm visits all branching substrings of T. (The basic insight is that every internal node *is* the lca of at least one pair of leaves.)

[2] A substring $\alpha \preceq T$ is called *branching* if there exist $a, b \in \Sigma$, $a \neq b$, s.th. both αa and αb occur in T. These are exactly the strings that correspond to an internal node in the (virtual) suffix tree of T.

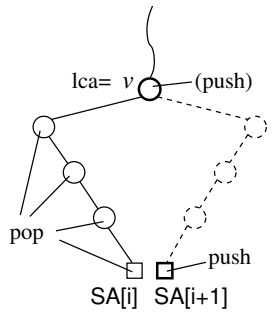

Figure 2. Going from suffix $T_{\mathsf{SA}[i]..n}$ to $T_{\mathsf{SA}[i+1]..n}$ when visiting all branching substrings. Solid nodes are on the stack, dashed nodes not yet. v is pushed if necessary, leaf $\mathsf{SA}[i+1]$ is always pushed.

3.2. Calculating Frequencies of Branching Substrings

Let us now describe Hui's approach [18] to calculate $\mathsf{freq}(\phi, \mathcal{D})$ for all branching substrings ϕ of $\mathcal{D}$. The idea is to calculate *two* counters $S(\phi, \mathcal{D})$ and $C(\phi, \mathcal{D})$ for each ϕ (simply S and C if clear from the context), such that $\mathsf{freq} = S - C$. S counts the number of *all* occurrences of ϕ in $\mathcal{D}$, and C counts the number of *duplicates* of ϕ in the same string in $\mathcal{D}$. More formally, $S(\phi, \mathcal{D}) = \sum_{\psi \in \mathcal{D}} |\{i \in [1 : |\psi|] : \phi = \psi_{i..i+|\phi|-1}\}|$, and $C(\phi, \mathcal{D}) = \sum_{\psi : S(\psi,\mathcal{D})>0} (S(\psi, \mathcal{D}) - 1)$.

For what follows, we need to define an additional array $D[1, n]$ on top of the generalized suffix array such that $D[i]$ gives the string number where suffix $T_{\mathsf{SA}[i]..n}$ points to, i.e. $D[i] = j$ if the first string separator in $T_{\mathsf{SA}[i]..n}$ is $\#_j$. By remembering the number h of the last string in $\mathcal{D}_1$, D also allows to infer whether a suffix points to $\mathcal{D}_1$ or $\mathcal{D}_2$.

The S-counters are easy to calculate during the simulated suffix-tree traversal: simply initialize them correctly for leaves, and when popping a node v from the stack, add v's S-value to its parent-node on the stack. More formally, when pushing a leaf $l = T_{\mathsf{SA}[i+1]..n}$ on R, we initialize $S(l, \mathcal{D}_1)$ with 1 and $S(l, \mathcal{D}_2)$ with 0 if $D[i+1] \leq h$, or vice versa if $D[i+1] > h$. When popping a node u from R, we add $S(u, \mathcal{D}_j)$ to $S(w, \mathcal{D}_j)$ to the direct ancestor w of u for $j = 1, 2$. Note that w is either the predecessor of u on R (if the string-depth of this predecessor is $\geq \mathsf{LCP}[i+1]$), or the newly pushed internal node v (otherwise).

Calculation of the C-counters is a little bit more intricate. We will just describe how to calculate $C(\phi, \mathcal{D}_1)$; the ideas for $\mathcal{D}_2$ are similar. Hui's key insight is that if a substring ϕ is repeated within a string $d \in \mathcal{D}_1$ from $\mathcal{D}_1$, then ϕ must be a prefix of the lcp of two different suffixes from d. Even more, if we enumerate d's suffixes in any order for all $d \in \mathcal{D}_1$ (say

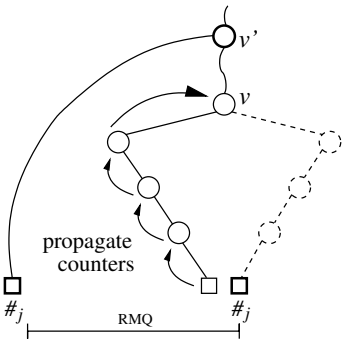

Figure 3. Determining the C-counters. Node v' represents the longest common prefix of two suffixes from the same string j, so $C[v']$ has to be increased by 1.

in the order they appear in SA), then the number of times that ϕ is a prefix of the lcp of consecutive suffixes (in that order) gives $C(\phi, \mathcal{D}_1)$.

With Lemma 1, this gives us all the tools we need to calculate the C-counters "on the fly" while visiting all branching substrings (see also Fig. 3): remember the position of the previous suffix of string j before position i for each $j \in [1 : d]$ in an array $P[1, d]$ (i.e., $P[j] = \max\{p \leq i : D[p] = j\}$). Then when at position $i + 1$, calculate the desired lcp as $l = \mathsf{LCP}[\mathrm{RMQ}_{\mathsf{LCP}}(P[D[i + 1]] + 1, i + 1)]$, and increment by 1 the C-counter of the node v' on R that has string-depth exactly l. (Note that such a node must be on R, as it is on the path from $\mathsf{SA}[i + 1]$ to the root.) The easiest way to find v' in R is to use another array of size n.

Like with the S-counters, when popping a node from R, we need to add the C-counters to its parent node. This step takes care of the fact that the RMQs from the above paragraph just locate the *longest* duplicates; but all *prefixes* of duplicates are duplicates as well.

3.3. Determining Interesting Patterns

In total, the integration of the above two techniques implies that when a node v is popped from the stack, the frequency of the respective substring is given by $\mathsf{freq} = S - C$. From this, we check if the string passes the frequency-based predicate (e.g., if it is an emerging substring). If so, we output $T_{\mathsf{SA}[i+1]+d-1}$ for all values d between the string-depth of v (inclusive) and that of its parent-node (exclusive).

4. Space-Efficient Version

The problem with the algorithm from the previous paragraph is that it still needs $O(n \log n)$ bits of space in the worst case, even if we take compressed representations of suffix- and lcp-arrays. This is because for degenerated suffix trees, the rightmost path could contain $O(n)$ nodes from the suffix tree; hence the space for stack R and all the counters would be $O(n \log n)$ bits. We show in this section how to achieve $O(n \log |\Sigma| + d \log n)$ bits of space.

4.1. New Data Structures

We now step through the data structures from §3 and show how to reduce the space for each of these.

4.1.1 Representing D and P

We use array P as is, but we will represent array D implicitly. Array P occupies $d \log n$ bits. During the algorithm one needs to inquery $P[D[i + 1]]$ when inserting the $(i+1)$-th suffix array element, after which one needs to update $P[D[i + 1]] = i + 1$. Value $D[i + 1]$ can be computed in time $O(t_{\mathsf{SA}})$ as follows: Store a bit-vector B that marks the boundaries of documents in the concatenation T by setting $B[j] = 1$ if position j starts a new document in T, otherwise $B[j] = 0$. Preprocess B for $rank$-queries, where $rank(B, j)$ gives the number of bits set in $B[1, j]$. That is, $rank(B, j)$ gives the document number in which the j-th position in T belongs to. It is possible attach to B an auxiliary structure of size $o(n)$ bits so that $rank(B, j)$ can be answered in constant time for any j [19]. Now we have $D[i + 1] = rank(B, SA[i + 1])$. Using compressed suffix array, the computation of $SA[i + 1]$ takes time $O(t_{\mathsf{SA}})$ and the $rank$-query on B takes constant time. The space used in addition to the already used compressed suffix array is $n + o(n)$ bits for the bit-vector B and its $rank$-structure.

4.1.2 Representing R

Stack R needs more functionality than being accessed only from top, as in order to increase the correct C-counter as described in §3.2, we need to quickly find the node on R with string-depth $l = \mathsf{LCP}[\mathrm{RMQ}_{\mathsf{LCP}}(P[D[i+1]]+1, i+1)]$. Thus, storing R as a difference-encoded list in $O(n)$ bits [16] would result in having to scan R in $O(n)$ time in the worst case after each RMQ.

Instead, we represent R via a dynamic succinct data structure for *searchable partial sums*. This is a data structure maintaining a sequence of symbols $A = a_1 \dots a_m$, supporting the following operations:

- $sum(A, i)$: returns $\sum_{j \leq i} a_j$
- $search(A, j)$: returns the smallest i such that $sum(A, i) \geq j$
- $update(A, i, \Delta)$: adds Δ to a_i ($\Delta = O(\mathrm{polylog}(n))$)
- $insert(A, i, x)$: inserts x between a_i and a_{i+1}

- $delete(A, i)$: deletes a_i

It has been shown in [25] that each of these operations can be supported in $O(\log n)$ time, by using an extension of the data structure from [5]. The space occupied by this data structure is only $n + o(n) + O(m) = O(n)$ bits, provided that the sum of the numbers in A is always $\leq n$.

In our case, if the elements in A represent the number of letters on the incoming *edges* of the nodes on R (and 0 for the root of the suffix tree), then the condition $\sum_{i=1}^{m} a_i \leq n$ is naturally satisfied (because the longest suffix has length n). A query $sum(A, i)$ gives the string-depth of the internal nodes (needed for popping all nodes with a string-depth $\geq$ $\mathsf{LCP}[i+1]$). The index (on R) of the node with string-depth l can be found by $r = search(A, l)$.

Note that we do not need the full functionality of the dynamic searchable partial sum structure, as the function $update()$ is not used at all, and we only have to insert and delete at the end of the sequence (corresponding to pushes and pops on R).

4.1.3 Representing C-counters

The C-counters (for counting the duplicates in a string) are also stored in a searchable partial sum data structure (see the previous section). This time, we only need the functions $insert$ (when a new node is pushed on the stack), $delete$ (when a node is popped), and $update(A, i, \Delta)$ (with $\Delta = 1$ when updating a_r after an RMQ, or with Δ being the C-value of the node that has just been popped from the stack).

This structure needs again $n+o(n)+O(m) = O(n)$ bits if we can assure that that the sum of all C-counters on the stack is always less than n. But this is true, as a C-counter of u is added to its parent node if and only if u's subtree has been traversed completely. So each suffix can contribute at most 1 to the set of all C-counters, hence the bound on their sum.

4.1.4 Representing S-counters

The S-counters (for counting the total number of occurrences of a string) are easier to handle, as they only need to be accessed from top of the stack. The only property we need to know is that the sum of the S-counters is never greater than n, as they "cover" disjoint sub-arrays in SA. Thus, we can encode them with variable-length prefix codes, e.g., Elias-δ-code [11]. This takes again $n + o(n) + O(m) = O(n)$ bits, while supporting deletions and insertions at the ends in $O(1)$ time.

4.2. Space and Time Analysis

The Compressed Suffix Array takes $O(n(H_0 + \log\log|\Sigma|))$ bits of space. Compressed LCP and RMQ values take overall $4n + o(n)$ bits of space. The database occupies $n \log|\Sigma|$ bits. Array P takes $d \log n$ bits. Bit-vector B and its $rank$-structure take $n + o(n)$ bits.

Interesting points are the peak space consumption of the data structures we use and their construction time. The Compressed Suffix Array can be constructed using $O(n \log|\Sigma|)$ bits of space in $O(n \log\log|\Sigma|)$ time [17], or even within space of the final structure [23] but using $O(n \log n)$ time. Once the Compressed Suffix Array is given, the lcp-representation can be constructed in $O(n \log^{\epsilon} n)$ time using no extra space in addition to the final structure [16]. Then, given the Compressed Suffix Array and the compressed lcp-representation, the linear time algorithm to construct the RMQ structure [12] takes $O(n \log^{\epsilon} n)$ time using no extra space.

During the main algorithm, we also allocate space for R, C and S. This space is bounded by $O(n)$ bits as analyzed earlier. The algorithm makes $O(n)$ queries and updates to the data structures. The most costly operations are the searches on R and updates on C which both take $O(\log n)$ time.

Theorem 1 *There is an algorithm for determining all F strings satisfying a frequency-based predicate in $O(n \log|\Sigma| + d \log n)$ bit of space and $O(n \log n)$ time. Writing the output takes additional $O(|F| \log^{\epsilon} n + \|F\|)$ time, where $0 < \epsilon \leq 1$ affects the constant of the space usage and $\|F\|$ is the total length of the output.*

5. Extensions and Applications

5.1. Less Space, More Time

Other tradeoffs are possible in Theorem 1 by using different variants of compressed suffix arrays. It is possible to obtain $nH_k + o(n \log|\Sigma|) + d \log n$ bits of space with the running time increasing to $O(n \log n(1 + \frac{\log|\Sigma|}{\log\log n}))$.Here $k = o(\log_{|\Sigma|} n)$ and H_k is the order-k entropy of $\mathcal{D}$. This is achieved by using the FM-index variant in [25, 14], and building on top of it the additional structures needed for the full functionality of compressed suffix arrays as in [32]. In this case, the text is not required to be stored at all (after the construction), as the structure is self-index and supports displaying any text substring of length ℓ in $O((\ell + \log^{1+\varepsilon} n) \log|\Sigma|)$ time. That is, outputting the result of the algorithm in Theorem 1 takes in this case $O(|F| \log^{1+\varepsilon} n \log|\Sigma| + \|F\| \log|\Sigma|)$ time, for any $\varepsilon > 0$ affecting the constants of the sub-linear structures.

5.2. Application to Storing Document Frequencies

Sadakane [31, Section 5.2] gives a succinct index structure that stores the document frequencies, i.e. values $S[i] -$

$C[i]$ that we compute on-the-fly in our algorithm. His structure consists basically of the compressed suffix array and a unary coding of the frequency values in the inorder of the virtual suffix tree. Sadakane shows that the final structure occupies $|CSA| + 2n + o(n)$ bits, where $|CSA|$ is the size of the compressed suffix array used. He does not give a space-efficient construction algorithm (a suffix *tree* is used as an intermediate structure, hence taking $O(n \log n)$ bits).

We can construct the required unary coding during our algorithm as follows: We maintain the balanced parentheses (BP) representation as in [32] using a dynamic bit-vector occupying $n + o(n)$ bits. This gives us the preorder of the virtual suffix tree nodes at each step. We use another dynamic bit-vector to store the C-counter values in the same unary coding as Sadakane uses. Using $rank$ and $select$ ($select(i)$ returns the position of the i'th 1 [19]) on both bit-vectors gives us a mapping between BP bit-vector and the C-counter bit-vector. Inserting a new node in BP means inserting 1 in the corresponding place of the C-counter bit-vector. Incrementing a C-counter works by finding the corresponding node in BP, mapping the position to C-counter bit-vector, and inserting 0 there. Finally, after all values are computed, the preorder of C-counter bit-vector can be turned into the inorder used in Sadakane's scheme, and the intermediate structures can be deleted. The algorithm uses the same space and time as reported in Theorem 1.

6. Experimental Results

We have implemented the algorithm from §4 in C++[3] and compared it to the C++-implementation of the optimal algorithm from §3. Our implementation deviates from our theoretical proposal as we use a compressed suffix array that is based on *sampling*. We use a standard sampling rate of $\log n$ that minimize the space usage, as this is the main objective of our approach. However, we will also see that a smaller (fixed) sampling rate drastically decreases the execution time, while leading only to a moderate increase in space usage. We present test results of time and maximum memory usage for different datasets of protein and genome data.

6.1. Protein Datasets

We used two datasets consisting of the primary structure of all protein data from human and mouse, which were obtained from Swissprot using the keywords HUMAN and MOUSE in the NEWT taxonomy browser. The human dataset ($\mathcal{D}_1$) contained 71,622 proteins of total length ≈27.3MB, and the mouse dataset ($\mathcal{D}_2$) contained 62,562 proteins of total length ≈26.3MB. It is interesting to compare these data sets because the genome of human and mouse is known to be largely the same, but (given the different phenotype of the two species!) must contain some significant differences.

All tests on protein data were run on a 3.0GHz Intel Pentium 4 CPU with 3GB of main memory. All output was redirected to the "null"-device in order to remove influences from secondary storage units.

The programs were tested both on mining frequent substrings (Probl. 1) and emerging substrings (Probl. 2). The results for the frequent substring mining are reported in Tbl. 1. Here, the maximum support threshold τ for $\mathcal{D}_2$ was fixed to .95. The more striking property of the space-efficient algorithm is that it uses only 183.1MB of memory, while the optimal algorithm needs 1,267.3MB. This means a space reduction of a factor of ≈ 6.9. As already mentioned in the introduction, space is often a more critical resource than time; e.g., users are often willing to wait 3–4 hours instead of 3 minutes, if this allows them to apply their methods to much bigger data sets.

Table 1. Running times (seconds) for mining *frequent* strings for the optimal (§3) and the space-efficient (§4) algorithm. ρ is the minimum support threshold for $\mathcal{D}_1$. The maximum support threshold τ was held fixed at .95. The last column denotes the size of the output.

ρ	optimal	space-efficient	|output|
.1	154.4	12,426	3,559
.2	151.5	12,403	1,211
.3	154.9	12,565	953
.4	156.5	12,442	694
.5	155.8	12,558	436
.6	152.6	13,420	196
.7	154.1	12,445	49
.8	155.2	12,524	7
.9	151.3	12,672	2
avg	154.0	12,606.1	—

The running times for the emerging substring tasks are reported in Tbl. 2 (for $\rho_g = 1.3333$) and Tbl. 3 (for $\rho_g = 2.0$). The results are largely along the lines of the frequent string mining tasks: space is reduced by a factor of 6–7, and the running time is increased by about two orders of magnitude.

To give an idea on how the dataset length affects computation time and space usage, we tested mining frequent substrings from different size prefixes of the protein data. Tests were run on dataset prefixes of total length 2–50MB and on the whole dataset of length ≈53.6MB. Here, the maxi-

[3]The implementation can be downloaded from `http://www.cs.helsinki.fi/group/suds/`

Table 2. Running times (seconds) for mining *emerging* strings for the optimal (§3) and the space-efficient (§4) algorithm. ρ_s is the minimum support threshold. The growth rate ρ_g was held fixed at 1.3333. The last column denotes the size of the output.

ρ_s	optimal	space-efficient	|output|
.1	154.2	12,597	12
.05	151.9	12,636	233
.01	155.1	12,734	35,839
.005	152.8	13,137	225,198
.001	156.0	12,679	60,449,586

Table 3. Running times (seconds) for mining *emerging* strings for the optimal (§3) and the space-efficient (§4) algorithm. ρ_s is the minimum support threshold. The growth rate ρ_g was held fixed at 2.0. The last column denotes the size of the output.

ρ_s	optimal	space-efficient	|output|
.1	154.2	12,380	0
.05	155.9	12,538	64
.01	154.4	12,637	34,343
.005	157.1	12,674	220,585
.001	159.1	13,228	60,410,579

mum support threshold τ was fixed to $.95$ and the minimum threshold ρ to $.40$. Maximum memory usage for the optimal (§3) and the space-efficient (§4) algorithm on different size datasets are reported in Fig. 4. Running times for the same datasets are shown in Fig. 5. Both figures show also a time-space tradeoff for the space-efficent algorithm by using a fixed samplerate ($= 3$) inside CSA — we see that a modest increase in memory consumption (Fig. 4) increases the running time by one order of magnitude (Fig. 5)!

Inspired by this, we tested various time-space tradeoffs for the space-efficient algorithm by using a denser sampling inside CSA. The test was run on the whole protein dataset with the same parameteres as above. The samplerate of CSA was given values between 3–25. The resulting time and space usage for each samplerate is reported in Fig. 6. Note that the default samplerate used by the algorithm is $\lfloor \log n \rfloor$ ($= 25$ for the whole dataset). We thus conclude that in practice we should choose a denser sampling.

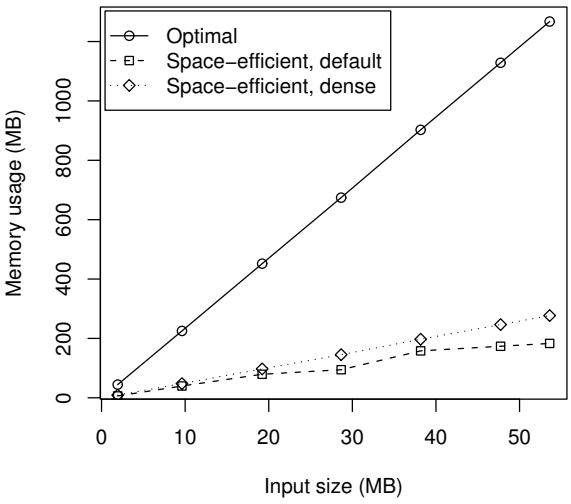

Figure 4. Maximum memory usage while mining frequent substrings from protein data for the optimal (§3) and the space-efficient (§4) algorithm. A time-space tradeoff with dense sampling is shown for the space-efficent algorithm.

6.2. Human Genome

To show that the space-efficient algorithm works also on genome-scale data, we used a DNA dataset of 22 human chromosomes (build NCBI34) of total length 2.9 billion base pairs. We measured time and maximum memory usage to solve the frequent substrings and emerging substrings problems. Time to output the result's substrings was excluded from these test results. Tests were run on a 3.0GHz Intel Xeon CPU with 128GB of main memory.

The space-efficient algorithm used 39.8 hours and required maximum of 9.3GB of memory for mining the whole genome. With a different time-space tradeoff (by denser sampling inside CSA) we achieved a running time of only 22.6 hours with 14.9GB of maximum memory usage.

The implementation of the optimal-time algorithm was tuned for 32 bit word length which is not enough for genome-scale data. However, we can estimate the time and space requirements from tests with 9 small chromosomes (a quarter of the whole genome). For this small portion of the genome, the optimal time algorithm used $\approx$17.3 minutes and required maximum of $\approx$13.1GB of heap. This suggests that genome-scale mining would require about an hour of time and about 50GB of memory. Note that the integer ar-

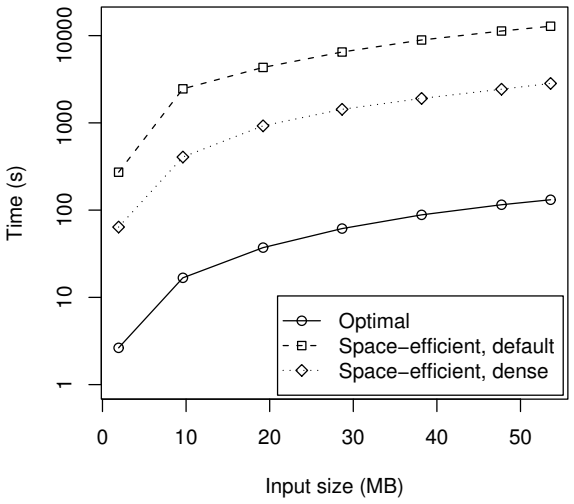

Figure 5. Running times for frequent substring mining with the optimal (§3) and the space-efficient (§4) algorithm on a logarithmic scale. A time-space tradeoff with dense sampling is shown for the space-efficent algorithm.

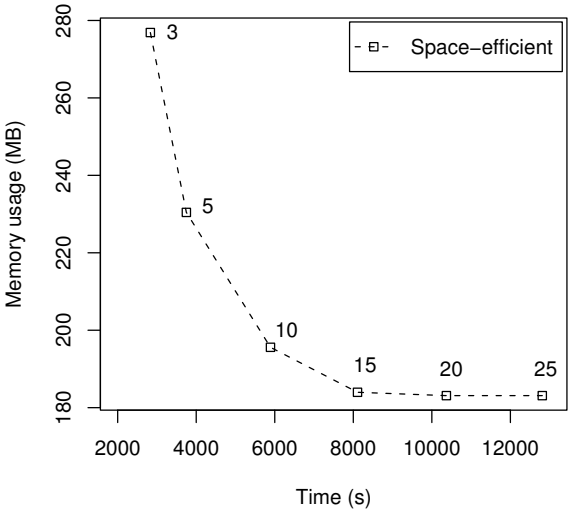

Figure 6. Time-space tradeoff for the space-effient algorithm with samplerates 3–25 on protein data. Decreasing the samplerate increases memory usage but makes our algorithm significantly faster.

rays for SA, LCP and C-counters already require $3n \log n$ bytes of memory (≈32.0GB for the whole genome).

6.3. Comparison to Other Algorithms

Let us briefly describe other algorithms for mining substrings. Algorithms VST [8] and FAVST [24] rely on a data structure called *Version Space Tree*. Because this tree is basically a suffix *trie* with $O(n^2)$ nodes in the worst case, these algorithms suffer from high memory requirement. In practice, we could not test these algorithms on any of the datasets above, as they are only applicable to input sizes up to several hundred kilobytes.

Chan et al. [7] presented an algorithm for the emerging substrings problem, but we could not find an implementation of their approach. Furthermore, no information about the practical memory requirement is reported. The algorithm itself is based on a *merged suffix tree* with $O(n)$ nodes that suggests a space requirement of $O(n \log n)$ bits. It has already been shown in a previous study [13] that suffix-array based methods are superior to those built on suffix trees or tries both in terms of time and space.

Two very recent improvements [22, 34] of the original linear-time algorithm [13] also addresses the problem of lowering the space consumption; however, they only achieve about half the space of the orignal algorithm. Furthermore, their theoretical space guarantee is no better than the $O(n \log n)$ bits of the original solution. Nevertheless, it must be said that due to the simplicity of these two algorithms [22, 34] they work faster in practice.

7. Conclusions

We hence conclude from the experiments that the practical performance of the two algorithms are in accordance with their theoretical guarantees. However, we must also say that the constants involved in the time-performance of the succinct data structures (§4) are still large.

Acknowledgments

Thanks to the discussions with Luis Russo after the Workshop on Compression, Text, and Algorithms, organized by Gonzalo Navarro at University of Chile, November, 2007, we were able to improve our result significantly.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

[2] H. Arimura and T. Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *Journal of Combinatorial Optimization*, 13:243–262, 2006. Special issue on bioinformatics.

[3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[4] F. Birzele and S. Kramer. A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics*, 22(24):2628–2634, 2006.

[5] D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In *Proc. SODA*, pages 11–19. ACM/SIAM, 2004.

[6] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Prediction of regulatory elements in silico on a genomic scale. *Genome Research*, 8:1202–1215, 1998.

[7] S. Chan, B. Kao, C. L. Yip, and M. Tang. Mining emerging substrings. In *Proc. of the Itl. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 119–126. IEEE Computer Society, 2003.

[8] L. De Raedt, M. Jäger, S. D. Lee, and H. Mannila. A theory of inductive query answering. In *Proc. Int. Conf. on Data Mining (ICDM)*, pages 123–130. IEEE Computer Society, 2002.

[9] L. De Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment mining. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 853–862. Morgan Kaufmann, 2001.

[10] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 43–52. ACM Press, 1999.

[11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[12] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. Int. Symp. on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.

[13] J. Fischer, V. Heun, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 4213 of *LNCS*, pages 139–150. Springer, 2006.

[14] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. Latin American Symp. on Theoretical Informatics (LATIN)*, volume 4957 of *LNCS*, pages 374–386. Springer, 2008.

[15] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[16] W.-K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002.

[17] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS*, pages 251–260. IEEE Computer Society, 2003.

[18] L. C. K. Hui. Color set size problem with application to string matching. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.

[19] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[21] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 200–210. Springer, 2003.

[22] A. Kügel and E. Ohlebusch. A space efficient solution to the frequent string mining problem for many databases. *Data Mining and Knowledge Discovery*, 17(1):24–38, 2008.

[23] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space- and time-efficient algorithm for constructing compressed suffix arrays. In *Proc. Annual Int. Computing and Combinatorics Conf. (COCOON)*, LNCS 2387, pages 401–410. Springer, 2002.

[24] S. D. Lee and L. De Raedt. An efficient algorithm for mining string databases under constraints. In *Proc. Itl. Workshop on Knowledge Discovery in Inductive Databases (KDID)*, volume 3377 of *LNCS*, pages 108–129. Springer, 2005.

[25] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, to appear 2008.

[26] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[27] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.

[28] L. Parida. *Pattern Discovery in Bioinformatics: Theory and Algorithms*. Chapman & Hall / CRC, 2007.

[29] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[30] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):569–607, 2007.

[31] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.

[32] N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen. Engineering a compressed suffix tree implementation. In *Proc. Int. Workshop on Experimental Algorithms (WEA)*, LNCS 4525, pages 217–228. Springer, 2007.

[33] L. Wang, H. Zhao, G. Dong, and J. Li. On the complexity of finding emerging patterns. *Theor. Comput. Sci.*, 335(1):15–27, 2005.

[34] D. Weese and M. H. Schulz. Efficient string mining under constraints via the deferred frequency index. In *Proc. 8th Industrial Conf. on Data Mining (ICDM)*, volume 5077 of *LNCS*, pages 374–388. Springer, 2008.