Optimal String Mining Under Frequency Constraints

Johannes Fischer[1], Volker Heun[1], and Stefan Kramer[2]

[1] Ludwig-Maximilians-Universität München, Institut für Informatik,
Amalienstr. 17, D-80333 München
{Johannes.Fischer, Volker.Heun}@bio.ifi.lmu.de
[2] Technische Universität München, Institut für Informatik/I12,
Boltzmannstr. 3, D-85748 Garching b. München
kramer@in.tum.de

Abstract. We propose a new algorithmic framework that solves frequency-related data mining queries on databases of strings in optimal time, i.e., in time linear in the input and the output size. The additional space is linear in the input size. Our framework can be used to mine frequent strings, emerging strings and strings that pass other statistical tests, e.g., the χ^2-test. In contrast to the presented result for strings, no optimal algorithms are known for other pattern domains such as itemsets. The key to our approach are several recent results on index structures for strings, among them suffix- and lcp-arrays, and a new preprocessing scheme for range minimum queries. The advantages of array-based data structures (compared with dynamic data structures such as trees) are good locality behavior and extensibility to secondary memory. We test our algorithm on real-world data from computational biology and demonstrate that the approach also works well in practice.

1 Introduction

In many applications, e.g., in computational biology, the goal is to find interesting string or sequence patterns in data. Application areas are, among others, finding discriminative features for sequence classification or segmentation [1], discovering new binding motifs of transcription factors, or probe design [2]. 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. This category encompasses combined minimum/maximum support constraints, constraints concerning emerging substrings, and constraints concerning statistically significant substrings. We present an algorithm that is able to answer such queries optimally, that is, in time linear in the size of the input database, plus the time to output the solution patterns.

In previous work [2], we investigated string mining approaches based on breakthrough results on index structures for strings, among them suffix arrays and longest common prefix (lcp) tables [3,4,5]. Suffix arrays are essentially a representation of the lexicographic order of all suffixes of a string. lcp tables contain the

J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.): PKDD 2006, LNAI 4213, pp. 139–150, 2006.

length of the longest common prefix of two consecutive suffixes in a suffix array. For both suffix arrays and lcp tables, fast construction algorithms are known. As in our previous approach, we assume that the suffix array and the lcp table are computed in a preprocessing step. The key to the approach presented here is another preprocessing scheme for so-called range minimum queries (RMQs). RMQs generalize the lcp table in the sense that the length of the longest common prefix can be answered for *arbitrary* suffixes. Taking advantage of recent results [6,7], it is possible to answer RMQs in constant time. Another technical novelty is the solution to computing the frequency counts. The solution first determines the number of all occurrences (counting several occurrences per example), and then subtracts so-called correction terms to obtain the final counts per example. It is shown that the presented approach is able to answer all frequency-related constraints in linear time, i.e., optimally. For instance, it is possible to compute all statistically significant substrings between two classes of strings in time linear in the total length of the strings in the database, plus the total length of all such significant strings (i.e., the output size). It is interesting to note that no optimality results are known for other pattern domains such as itemsets or graphs (see, e.g., [8]).

While the focus of this paper lies on the algorithm and the theoretical result, we also implemented and tested the approach to show that it works in practice. In our experiments, we compared protein sequences from humans and mice, in total more than 40MB of sequence data. The aim of the experiments was to mine all *frequent substrings* (Probl. 1, Sect. 2) and *emerging substrings*, respectively (Probl. 2). The only known algorithm for emerging substrings [9] runs in quadratic time, and is therefore not applicable. The experiments confirm that our approach works well in practice. In particular, most queries for emerging substrings can be answered in less than three minutes, and the mining of frequent substrings is 2–3 times faster than our previous method presented in [2].

2 Preliminaries

We consider patterns from the domain of strings. For a finite ordered alphabet Σ, a string ϕ is a chain $\phi_1 \dots \phi_n$ of letters $\phi_i \in \Sigma$. We often write $\phi_{n..m}$ to denote the substring of ϕ ranging from position n to m. $|\phi|$ denotes the number of letters in ϕ. $\Sigma^\star$ is the set of all strings over Σ. For $\phi, \psi \in \Sigma^\star$ we write $\phi \preceq \psi$ if ϕ is a substring of ψ. $\mathrm{lcp}(\phi, \psi)$ gives the *length* of the *longest common prefix* of ϕ and ψ. For example, $\mathrm{lcp}(\mathtt{aab}, \mathtt{abab}) = 1$. Given a database $\mathcal{D} \subseteq \Sigma^\star$ with strings over Σ, 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* and the *support* of a pattern $\phi \in \Sigma^\star$ in $\mathcal{D}$ as follows:

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

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. The main contribution

of this article is to show how one can compute the frequency (or support) of all strings occurring at least once in one of the databases in optimal time, i.e., in time linear in the size of the input databases. This allows us to solve frequency-related mining queries in optimal time, i.e., in time linear in the sum of the input- and the output-size. Naturally, the query must be computable from the frequency (or support) in constant time.

We now introduce three problems that can be solved optimally with our approach. The first one is as follows.

Problem 1. Given m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ of strings over Σ and m pairs of frequency thresholds $(min_1, max_1), \ldots, (min_m, max_m)$, the *Frequent Pattern Mining Problem* is to return all strings $\phi \in \Sigma^\star$ that satisfy $min_i \leq \mathsf{freq}(\phi, \mathcal{D}_i) \leq max_i$ for all $1 \leq i \leq m$.

This well-known problem has been addressed by many authors using different solution strategies and data-structures ([10,11,2]), but none of these is optimal.

Next, we consider a 2-class problem for a (usually positive) database $\mathcal{D}_1$ and a (usually negative) database $\mathcal{D}_2$. 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 Patterns [12]:

Problem 2. Given two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ of strings over Σ, a support threshold ρ_s $(0 < \rho_s \leq 1)$, and a minimum growth rate $\rho_g > 1$, the *Emerging Substrings Mining Problem* is to find all strings $\phi \in \Sigma^\star$ such that $\mathsf{supp}(\phi, \mathcal{D}_1) \geq \rho_s$ and $\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \geq \rho_g$.

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 for the two databases. The only known solution for finding ESs [9] is quadratic in the input size. The following example will be continued throughout this paper.

Example 1. Let $\mathcal{D}_1 = \{\mathtt{aaba}, \mathtt{abaaab}\}$, $\mathcal{D}_2 = \{\mathtt{bbabb}, \mathtt{abba}\}$, $\rho_s = 1$, and $\rho_g = 2$. Then the emerging substrings from $\mathcal{D}_2$ to $\mathcal{D}_1$ are `aa`, `aab` and `aba`. In this case, these are also the jumping ESs. ◇

As a last example problem that our method can solve optimally we mention the χ^2-test.

Problem 3. Given m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ of strings over Σ and a threshold ρ. Let $n = \sum_{j=1}^{m} |\mathcal{D}_j|$ be the total number of strings, $f = \sum_{i=1}^{m} \mathsf{freq}(\phi, \mathcal{D}_i)$ the total frequency of ϕ, and $\mathbf{E}_j = f \cdot |\mathcal{D}_j|/n$ be the expected value of ϕ's frequency. Then ϕ is significant if it passes the χ^2-test, i.e., if $\chi^2 = \sum_{j=1}^{m} \frac{(\mathsf{freq}(\phi, \mathcal{D}_j) - \mathbf{E}_j)^2}{\mathbf{E}_j} \geq \rho$.

2.1 Suffix- and lcp-Arrays

This section introduces two fundamental data structures that we need for our algorithm. We write $A[1,n]$ for an array A of length n, and $A[i]$ denotes the i'th entry of A. To make the following definitions as general as possible, let t denote an arbitrary string of length n. Later, t will be formed from the input databases (fully explained in Sect. 3.1). Recall that $t_{i..j}$ is the substring from i to j.

The *suffix array* SA (see [3,4]) for t is used to describe 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 contain all of the numbers from 1 to n (i.e., $\{\mathsf{SA}[1],\ldots,\mathsf{SA}[n]\} = \{1,\ldots,n\}$), and $t_{\mathsf{SA}[i]..n}$ is lexicographically less than $t_{\mathsf{SA}[i+1]..n}$ for all $1 \leq i < n$. See Fig. 1(a) 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. [13]. In practice, however, asymptotically slower algorithms [14,15] have been shown to perform faster. The method in [15] has the further advantage that it uses only ϵn additional bytes of space, which is close to optimal. Here, ϵ is a tunable parameter that determines the speed of the algorithm and can be made arbitrarily small.

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. [5] gave an algorithm to compute LCP in $O(n)$ time, and Manzini [16] adapted this algorithm to use only one integer array. It can be argued that most of the LCP-values are small compared with the size of the text and could thus be stored in less than n words, but we do not pursue this approach here.

2.2 Range Minimum Queries

The last tool we need for our linear-time 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., $\mathrm{RMQ}_{\mathsf{LCP}}(i,j) := \arg\min_{k\in\{i,\ldots,j\}}\{\mathsf{LCP}[k]\}$. 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- and the j'th-smallest suffix of t are equal in exactly their $\mathsf{LCP}[\mathrm{RMQ}_{\mathsf{LCP}}(i+1,j)]$ first characters.

It has been shown that a linear preprocessing of any input array A is sufficient to find $\mathrm{RMQ}_A(i,j)$ in time $O(1)$ [6]. This method has recently been refined

to use only $o(n)$ extra space [7]. The basic idea for both approaches is to divide the array into blocks of size $\Theta(\log n)$. Then each block is preprocessed such that a query that lies completely *inside* one block can be answered in constant time. This step is accomplished by applying the so-called Four-Russians-Trick [17] to the blocks (precomputation of all results for sufficiently small instances). A final step preprocesses the array such that queries that exactly span over several blocks can be answered efficiently. In total, each range minimum query can be decomposed into at most three sub-queries, where the first and the last of these are in-block-queries, and the second is an out-of-block-query. As each sub-query can be answered in constant time, the overall query time is $O(1)$ [6,7].

3 The New Algorithm

In this section we present our linear-time algorithm for answering frequency-related mining queries (e.g., emerging substrings). Logically, the algorithm can be divided into three main phases: (1) Preprocessing, (2) Labeling, and (3) Extraction. The preprocessing step constructs all necessary data structures: the suffix- and lcp-array, and the preprocessing for RMQ. The labeling step does the principal work for a fast calculation of the string-frequencies. Finally, the extraction step returns all strings passing the frequency-based criterion.

The main idea for computing the frequencies is as follows. Let $\mathcal{D}_j = \{s^{j,1}, \ldots, s^{j,|\mathcal{D}_j|}\}$ be the given databases ($1 \leq j \leq m$). For the strings ϕ occurring in any of the databases, we compute the *total* number of occurrences in $\mathcal{D}_j$ and store the respective numbers in $S_{\mathcal{D}_j}(\phi)$. We further compute so-called *correction terms* $C_{\mathcal{D}_j}(\phi)$ that count how often string ϕ has a repetition in the *same* string of $\mathcal{D}_j$:

$$S_{\mathcal{D}_j}(\phi) = |\{(i,k) : s^{j,k}_{i..i+|\phi|-1} = \phi\}|,\ C_{\mathcal{D}_j}(\phi) = \sum_{k=1}^{|\mathcal{D}_j|} (|\{i : s^{j,k}_{i..i+|\phi|-1} = \phi\}| - 1) \quad (1)$$

Then $\mathsf{freq}(\phi, \mathcal{D}_j)$ clearly equals $S_{\mathcal{D}_j}(\phi) - C_{\mathcal{D}_j}(\phi)$. In our example, $S_{\mathcal{D}_1}(\mathtt{ab}) = 3$ (there are 3 occurrences of `ab` in $\mathcal{D}_1$) and $C_{\mathcal{D}_1}(\mathtt{ab}) = 1$ (`ab` is repeated once in the second string $s^{1,2} = \mathtt{abaaab}$ of $\mathcal{D}_1$). We will see in Sect. 3.3 that it is not very hard to compute the S-numbers in linear time; the real difficulty lies in the computation of the C-numbers. The following lemma suggests how the lcp-array can be used to calculate these correction terms (cf. Lemma 1):

Lemma 2. *For any string ϕ occurring in $\mathcal{D}_j$, $C_{\mathcal{D}_j}(\phi)$ is given by the number of times that ϕ is a prefix of the longest common prefix of two lexicographically adjacent suffixes from the same string $s^{j,k}$ in $\mathcal{D}_j$:*

$$C_{\mathcal{D}_j}(\phi) = \sum_{k=1}^{|\mathcal{D}_j|} |\{(i,i') : \phi \text{ prefix of lex. adj. suffixes of } s^{j,k} \text{ starting at } i \neq i'\}|$$

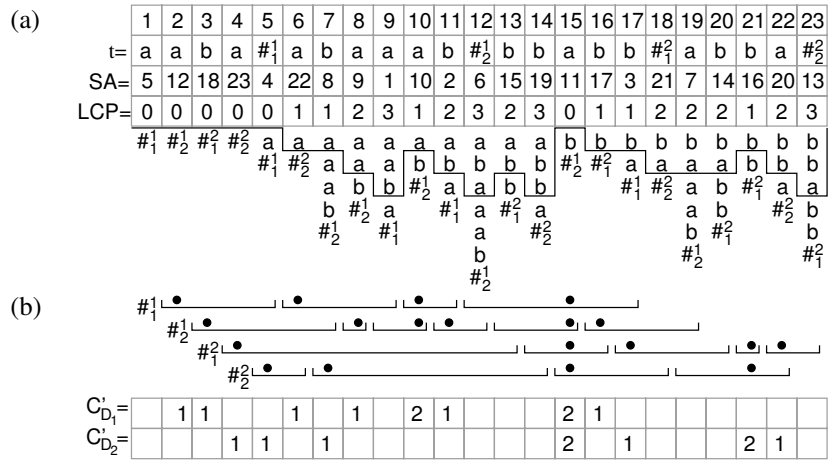

Fig. 1. (a) The suffix array for w and its lcp-table. Below position i we draw the string $t_{\mathsf{SA}[i]..n}$ until reaching the first end-of-string marker. The solid line going through these strings indicates the lcp-values. (b) Computation of the C'-numbers. The intervals are those for which range minimum queries on LCP are executed; the position of the minimum is depicted by a solid circle. Empty fields in the two arrays denote 0.

We omit the proof of this result due to space limitations. As an example, the suffixes of $s^{1,2} = \texttt{abaaab}$ are (in lexicographic order) `aaab`, `aab`, `ab`, `abaaab`, `b`, and `baaab`. The third and the fourth have `ab` as their longest common prefix. Because no suffixes from $s^{1,1} = \texttt{aaba}$ have `ab` as their longest common prefix, the value of $C_{\mathcal{D}_1}(\texttt{ab})$ is 1.

The calculation of the correction terms is done in phase (2) and (3) of our algorithm. In phase (2), we create auxiliary arrays that allow an easy computation of the actual correction terms. The computation of the C-terms is then done along with the computation of the S-numbers in phase (3). The following sections describe the three phases in greater detail.

3.1 Preprocessing

We form a (conceptual) string $s^{1,1}\#_1^1 \ldots s^{1,|\mathcal{D}_1|}\#_{|\mathcal{D}_1|}^1 \ldots s^{m,1}\#_1^m \ldots s^{m,|\mathcal{D}_m|}\#_{|\mathcal{D}_m|}^m$ which we denote by t. The $\#_i^k$'s are new symbols that do not occur in any of the databases and serve to mark the end of a string from the respective database. Note that the length of t is $n := \sum_{j=1}^{m}(\|\mathcal{D}_j\| + |\mathcal{D}_j|)$. The preprocessing then consists of constructing the following data structures for t (in this order): the suffix array SA, the lcp-array LCP, and the information to answer $\text{RMQ}_{\mathsf{LCP}}(i,j)$ in $O(1)$. All steps take time $O(n)$. See Fig. 1(a) for an example.

A short definition is necessary at this point: We say that entry $\mathsf{SA}[i]$ *points* to string $s^{j,k}$ in database $\mathcal{D}_j$ iff the first end-of-string marker in $t_{\mathsf{SA}[i]..n}$ is $\#_k^j$. For example, in Fig. 1(a), $\mathsf{SA}[8] = 9$ points to $s^{1,2}$, because $t_{9..23} = \texttt{aab}\#_2^1\texttt{bb}\ldots$.

Algorithm 1. Labeling of the lcp-array.

Input. suffix array SA and lcp-array LCP of size n for m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$
Output. m arrays $C'_{\mathcal{D}_1}, \ldots, C'_{\mathcal{D}_m}$ of size n

```
1  Let last_{D_i} be an array of size |D_i|, initialized with all 0 (i = 1, ..., m);
2  Let C'_{D_i} be an array of size n, initialized with all 0 (i = 1, ..., m);
3  for i = 1, ..., n do
4      Let j and k be defined such that SA[i] points to s^{j,k};
5      if last_{D_j}[k] ≠ 0 then
6          l ← RMQ_LCP(last_{D_j}[k] + 1, i);
7          increase C'_{D_j}[l] by 1;
8      end
9      last_{D_j}[k] ← i;
10 end
```

3.2 Labeling

Alg. 1 augments the lcp-array LCP with arrays $C'_{\mathcal{D}_1}, \ldots, C'_{\mathcal{D}_m}$ which facilitate the computation of the correction terms in phase 3. Although the $C'_{\mathcal{D}_j}$'s are represented by new arrays of size n, we call this step "labeling" because it is derived from the *tree labeling technique* by Hui [18]. We want $C'_{\mathcal{D}_j}[i]$ to be equal to the number of lexicographically adjacent suffixes from the *same* string in $\mathcal{D}_j$ that share a longest common prefix of length $\mathsf{LCP}[i]$. More formally, $C'_{\mathcal{D}_j}[i]$ equals the number of triples (a, b, k) that fulfill the following constraints:

1. $1 \leq a < i \leq b \leq n$, and $\mathsf{SA}[a]$ and $\mathsf{SA}[b]$ point to the same string $s^{j,k}$ in $\mathcal{D}_j$.
2. No entry strictly between a and b points to $s^{j,k}$.
3. $\mathrm{lcp}(t_{\mathsf{SA}[a]..n}, t_{\mathsf{SA}[b]..n|}) = \mathsf{LCP}[i]$.

Note that point 3 actually states that two lexicographically consecutive suffixes of $s^{j,k}$ have an lcp-value of $\mathsf{LCP}[i]$, because of 1 and 2. Note also that due to the definition of the lcp-array (lengths of longest common prefix of lexicographically adjacent suffixes) there *must* be an $i \in [a, b]$ with $\mathsf{LCP}[i] = \mathrm{lcp}(t_{\mathsf{SA}[a]..n}, t_{\mathsf{SA}[n]..n|})$, so $C'_{\mathcal{D}_j}[i'] \neq 0$ for at least one i' between a and b.

The C'-numbers are computed as follows: the for-loop (lines 3–10) scans the lcp-array from left to right. Array $last_{\mathcal{D}_j}[k]$ holds the rightmost position in SA to the left of i that points to $s^{j,k}$. Thus, if $\mathsf{SA}[i]$ points to $s^{j,k}$, setting $a = last_{\mathcal{D}_j}[k]$ and $b = i$ fulfills constraints 1 and 2 above. Because of Lemma 1, $\mathrm{lcp}(t_{\mathsf{SA}[a]..n}, t_{\mathsf{SA}[b]..n|})$ is given by $\mathrm{RMQ}_{\mathsf{LCP}}(last_{\mathcal{D}_j}[k] + 1, i)$ (line 6). See Fig. 1(b) for an example.

We now sketch how the C'-numbers help to compute the actual correction terms. We compute $C(\phi)$ for the strings ϕ that are *maximally repeated* (also called *branching* in [5]), which means that they occur more than once in t, say x times, but all extensions of ϕ (i.e., strings of which ϕ is a proper prefix) occur less than x times.[1] The number of such strings is clearly linear, and the frequency of all other strings can be derived from one of the maximally repeated strings.

[1] Note that the maximally repeated strings are exactly those strings that correspond to an internal node in the suffix tree [19] for t.

Lemma 3. *Let $\phi \preceq t$. The following is equivalent:*

1. *ϕ is maximally repeated.*
2. *There exist $1 \leq l \leq r \leq n$ s.t.*
   (a) $\mathsf{LCP}[l-1] < \mathsf{LCP}[l]$ and $\mathsf{LCP}[r] > \mathsf{LCP}[r+1]$,
   (b) $\mathsf{LCP}[i] \geq |\phi|$ for all $l \leq i \leq r$,
   (c) $\exists\ q \in \{l, \ldots, r\}$ with $\mathsf{LCP}[q] = |\phi|$ and $\phi = t_{\mathsf{SA}[q]..\mathsf{SA}[q]+|\phi|-1}$.

Parts (a) and (b) say that (l, r) is a maximal interval in SA where all suffixes have a common prefix (namely ϕ), and (c) says that at least two of the suffixes in this interval differ after position $|\phi|$. We refer the interested reader to [20] for a proof of this non-trivial result. From now on, we call (l, r) an *lcp-interval* representing string ϕ if it fulfills the conditions of Lemma 3. A *child-interval* of (l, r) is a maximal proper sub-interval of (l, r) that represents a different string. E.g., in Fig. 1(a), the lcp-interval representing `a` is $(6, 14)$, which has the child-intervals $(8, 9)$ (representing `aa`) and $(11, 14)$ (representing `ab`).

Now, if (l, r) is the lcp-interval that represents ϕ, with Lemma 2 we see that

$$C_{\mathcal{D}_j}(\phi) = \sum_{l \leq i \leq r} C'_{\mathcal{D}_j}[i] = \sum_{\substack{l \leq i \leq r \\ \mathsf{LCP}[i] = |\phi|}} C'_{\mathcal{D}_j}[i] \ + \sum_{\substack{(l',r') \text{ child-interval of } (l,r) \\ (l',r') \text{ represents } \psi \neq \phi}} C_{\mathcal{D}_j}(\psi) \ . \quad (2)$$

(The last step is to enable a recursive calculation of the C-terms.)

Example 2. In Fig. 1, the interval $(8, 9)$ (representing `aa`) gives $C_{\mathcal{D}_1}(\mathtt{aa}) = \sum_{8 \leq i \leq 9} C'_{\mathcal{D}_1}[i] = 1 + 0 = 1$, and the interval $(11, 14)$ (representing `ab`) gives $C_{\mathcal{D}_1}(\mathtt{ab}) = 1 + 0 + 0 + 0 + 0 = 1$. Having this, we can compute $C_{\mathcal{D}_1}(\mathtt{a})$ as $C'_{\mathcal{D}_1}[6] + C'_{\mathcal{D}_1}[7] + C_{\mathcal{D}_1}(\mathtt{aa}) + C'_{\mathcal{D}_1}[10] + C_{\mathcal{D}_1}(\mathtt{ab}) = 1 + 0 + 1 + 2 + 1 = 5$. ⋄

Kasai et al. [5] gave an algorithm that simulates a bottom-up-traversal of the suffix tree by scanning the lcp-array from left to right. We could thus calculate the C-numbers by a modification of their algorithm, applying (2) to all lcp-intervals in a bottom-up manner. However, this step can be incorporated into the extraction step (which we explain next), thereby avoiding the need to store the C-numbers in separate arrays.

3.3 Extraction

We now describe how to output all strings that pass the frequency-based criterion. As mentioned above, this step is accomplished by a simulated depth-first-traversal of the suffix tree [5], calculating for each lcp-interval representing string ϕ the values $S_{\mathcal{D}_j}(\phi)$ and $C_{\mathcal{D}_j}(\phi)$ for $j = 1, \ldots, m$, thereby yielding the frequency of ϕ in $\mathcal{D}_j$ as $S_{\mathcal{D}_j}(\phi) - C_{\mathcal{D}_j}(\phi)$. The formula for the C-numbers is given by (2), and for the S-numbers we have $S_{\mathcal{D}_j}(\phi) = \sum_{\substack{l-1 \leq i \leq r \\ \mathsf{SA}[i] \text{ points to } \mathcal{D}_j}} 1$ (again, (l, r) is ϕ's lcp-interval). As in (2), this can be rewritten to allow a recursive calculation.

Alg. 2 is used for the extraction phase. If one deletes lines 5, 17 and 19 from Alg. 2 and substitutes lines 8–13 by the single command "print $t_{\mathsf{SA}[i]..\mathsf{SA}[i]+v.h-1}$",

Algorithm 2. Extraction of all substrings satisfying p.

Input. suffix array SA, lcp-array LCP, $C'_{\mathcal{D}_j}$ as computed by Alg. 1 (all of size n), frequency-based predicate $p(\mathsf{supp}_{\mathcal{D}_1}, \ldots, \mathsf{supp}_{\mathcal{D}_m})$
Output. All substrings satisfying p

1 S is a stack holding tuples v of the form $(v.h, v.S_{\mathcal{D}_j}, \ldots, v.S_{\mathcal{D}_m}, v.S_{\mathcal{D}_2}, v.C_{\mathcal{D}_1}, \ldots, v.C_{\mathcal{D}_m})$
2 Let v be a stopper element with $v.h = -\infty$, push v on S
3 **for** $i = 1, \ldots, n+1$ **do**
4 $v \leftarrow \mathtt{top}(S)$ {v represents the string to be examined next}
5 $S_{\mathcal{D}_j} \leftarrow 0$ for all $j = 1, \ldots, m$
6 **while** $v.h > \mathsf{LCP}[i]$ **do**
7 $v \leftarrow \mathtt{pop}(S), w \leftarrow \mathtt{top}(S)$ {w points to top of stack throughout the loop}
8 **if** $w.h \geq \mathsf{LCP}[i]$ **then** $w.S_{\mathcal{D}_j}$ += $v.S_{\mathcal{D}_j}$, $w.C_{\mathcal{D}_j}$ += $v.C_{\mathcal{D}_j}$ for all j
9 $\mathsf{supp}_{\mathcal{D}_j} \leftarrow \frac{v.S_{\mathcal{D}_j} - v.C_{\mathcal{D}_j}}{|\mathcal{D}_j|}$ (for all $j = 1, \ldots, m$)
10 **if** $p(\mathsf{supp}_{\mathcal{D}_1}, \ldots, \mathsf{supp}_{\mathcal{D}_m})$ **then**
11 **for** $h = \max\{w.h, \mathsf{LCP}[i]\} + 1, \ldots, v.h$ **do** print $t_{\mathsf{SA}[i]..\mathsf{SA}[i]+h-1}$
12 **end**
13 $S_{\mathcal{D}_j} \leftarrow v.S_{\mathcal{D}_j}$ for all $j = 1, \ldots, m$
14 $v \leftarrow w$
15 **end**
16 **if** $v.h < \mathsf{LCP}[i]$ **then** push $(\mathsf{LCP}[i], S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, 0, \ldots, 0)$ on S
17 $\mathtt{top}(S).C_{\mathcal{D}_j}$ += $C'_{\mathcal{D}_j}[i]$ for all $j = 1, \ldots, m$
18 **if** $i \leq n$ **then**
19 Let $\mathsf{SA}[i]$ point to $\mathcal{D}_j$; set $S_{\mathcal{D}_j} \leftarrow 1$ and all other $S_{\mathcal{D}_{j'}}$'s to 0
20 push $(n - \mathsf{SA}[i] + 1, S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, 0, \ldots, 0)$ on S
21 **end**
22 **end**

this yields exactly the algorithm in Fig. 7 of [5] which solves the *substring traversal problem*, i.e., the enumeratation of all maximally repeated substrings. The idea behind this algorithm is to visit all suffixes of t in lexicographic order and to keep all maximally repeated prefixes of the current suffix on a stack S, ordered by their length with the longest being on top. A more formal desciption is as follows. Each element on S is represented by a tuple $(h, S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, C_{\mathcal{D}_1}, \ldots, C_{\mathcal{D}_m})$, where h is the length of the prefix (i.e., the corresponding prefix is $t_{\mathsf{SA}[i]..v.h-1}$), and the other variables are the counters as defined by (1) (page 143). At the beginning of step i of the for-loop (lines 3–22), we have that the $(i-1)$'th suffix and all maximally repeated prefixes of $t_{\mathsf{SA}[i-1]..n}$ are on S. Then the $(i-1)$'th suffix is visited (line 4) and the following steps are performed:

1. The while-loop (lines 6–15) removes from S all tuples representing strings with length at least $\mathrm{lcp}(t_{\mathsf{SA}[i-1]..n}, t_{\mathsf{SA}[i]..n}) = \mathsf{LCP}[i]$. These are exactly the prefixes of $t_{\mathsf{SA}[i-1]..n}$ which are not a prefix of $t_{\mathsf{SA}[i]..n}$. All strings passing the statistical criterion are returned (line 11).

2. The counter-values $S_{\mathcal{D}_j}(\phi)$ and $C_{\mathcal{D}_j}(\phi)$ of the current string v are added to the respective counters of the string on top of the stack (line 8). This step takes care of the last sum in (2), as v represents a child of the string on top.
3. When pushing the longest common prefix of two lexicographically adjacent suffixes on S (line 16), the counter-values are initialized correctly.
4. The C'-numbers are added to the correct string (line 17) which is again on top of the stack. This step takes care of the first sum in (2).
5. The suffix $t_{\mathsf{SA}[i]..n}$ is pushed on S with the correct counter-values (lines 18–21). Line 19 accounts for the initialisation of the $S_{\mathcal{D}_j}$-values.

It is shown in [5] that this algorithm visits all maximally repeated substrings of t, and its running time is $O(n)$ (apart from the for-loop that outputs the solutions, line 11). The discussion from Sect. 3.2 shows that the S- and C-values are calculated correctly, and thus in line 9 we have that the support of the string ϕ that is represented by v is calculated correctly. We thus have the following

Theorem 1. *For m databases of strings of total length n, all strings that satisfy a frequency-based criterion (e.g., emerging substrings) can be calculated in time $O(n + s)$, where s is the total size of the strings that satisfy the criterion.*

4 Practical Performance

The aim of this section is to show that our new method also works fast in practice, even on large datasets. We implemented the algorithm from Sect. 3 in C++ (available at `www.bio.ifi.lmu.de/~fischer/`) so that it finds emerging substrings (Problem 2) and frequent substrings (Problem 1), respectively. For the construction of SA we used the method presented in [15]. 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 [21]. The human dataset contained 57,020 proteins of total length ≈23MB, and the mouse dataset contained 50,680 proteins of total length ≈22MB. Because the implementation of the emerging-substring-miner from [9] is not publicly available we could not compare against their method. However, due the sheer size of the input (more than 45MB) it is very unlikely that their quadratic-time approach would work well. As an example, their fastest method takes 20–40 seconds to mine data of approximately 2MB total size, depending on the input parameters [9].

We ran several tests on an Athlon XP 3000 with 2GB of RAM under Linux. We redirected all output to the null-device in order to remove the influence of secondary storage devices. To remove the influences from the multi-user operating system (caching, network access,...) we repeated all experiments 5 times. Fig. 2 shows the average results for the ES-mining problem, for different values of ρ_g and ρ_s. The bottom line shows the time spent on preprocessing and labelling (phases 1 and 2). The top three lines are the total running times (phases 1–3) for $\rho_g = 6/3, 5/3$ and $4/3$, respectively. As expected, larger values for ρ_g result in shorter running times, because less strings have to be returned. This is also the case for larger values of ρ_s.

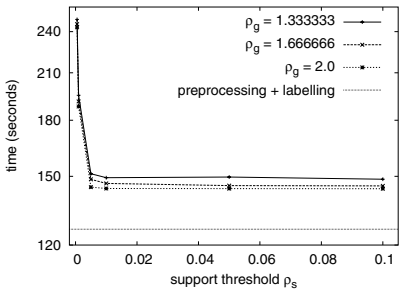

Fig. 2. Times for mining emerging substrings in two databases of appr. 23MB each. The bottom line shows the time for preprocessing. The top three lines represent different choices of ρ_g.

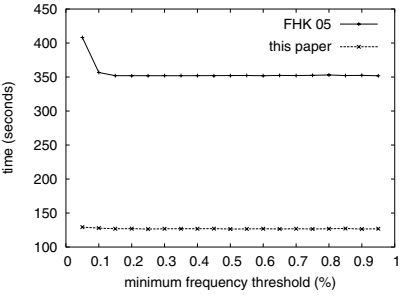

Fig. 3. Times for mining frequent patterns in the same two databases as in Fig. 2, compared with our older method FHK'05 [2]. The maximum frequency threshold was held fixed at 95%.

Fig. 3 shows results for mining frequent substrings. Again, we used the human dataset as the positive database, and the mouse dataset as the negative one. The maximum frequency threshold for MOUSE was held fixed at 95%, but similar graphs could be shown for other values. Because we have already shown in [2] that methods based on suffix tries [10,11] are not competitive with suffix-array based methods, we just compared with our previous approach [2]. As one can see in the figure, our new method is 2–3 times faster than our old method. This shows again that the method presented in this paper is also of practical value.

5 Conclusion

We presented a theoretically optimal solution to string mining under frequency constraints. As in previous work, we build upon results on index structures for strings. One of the building blocks is the fast computation of range minimum queries. Given this algorithmic framework, it is possible to compute solutions, e.g., for emerging substrings and patterns statistically associated with classes of sequences, very efficiently. In future work, we will focus on applications, such as finding new binding motifs, consider the integration of syntactic constraints and study the use of persistent index structures for strings.

References

1. Birzele, F., Kramer, S.: A new representation for protein secondary structure prediction based on frequent patterns. Submitted to Bioinformatics
2. Fischer, J., Kramer, S., Heun, V.: Fast frequent string mining using suffix arrays. In: Proc. ICDM, IEEE Computer Society (2005) 609–612
3. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In Frakes, W.B., Baeza-Yates, R.A., eds.: Information Retrieval: Data Structures and Algorithms. Prentice-Hall (1992) 66–82

4. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5) (1993) 935–948
5. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. CPM. Volume 2089 of LNCS. Springer (2001) 181–192
6. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2) (1993) 221–242
7. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Proc. CPM. Volume 4009 of LNCS. Springer (2006) 36–48
8. Wang, L., Zhao, H., Dong, G., Li, J.: On the complexity of finding emerging patterns. In: Proc. COMPSAC - Workshops and Fast Abstracts, IEEE Press (2004) 126–129
9. Chan, S., Kao, B., Yip, C.L., Tang, M.: Mining emerging substrings. In: Proc. DASFAA, IEEE Computer Society (2003) 119–126
10. De Raedt, L., Jäger, M., Lee, S.D., Mannila, H.: A theory of inductive query answering. In: Proc. ICDM, IEEE Computer Society (2002) 123–130
11. Lee, S.D., De Raedt, L.: An efficient algorithm for mining string databases under constraints. In: Proc. KDID 2004. Volume 3377 of LNCS. Springer (2005) 108–129
12. Dong, G., Li, J.: Efficient mining of emerging patterns: Discovering trends and differences. In: Proc. KDD, ACM Press (1999) 43–52
13. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. CPM. Volume 2676 of LNCS. Springer (2003) 200–210
14. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. In: Proceedings of ALENEX/ANALCO, SIAM Press (2005) 77–85
15. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1) (2004) 33–50
16. Manzini, G.: Two space saving tricks for linear time lcp array computation. In: Proc. SWAT. Volume 3111 of LNCS. Springer (2004) 372–383
17. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph. Dokl. Acad. Nauk. SSSR **194** (1970 (in Russian)) 487–488 English translation in *Soviet Math. Dokl.*, 11: 1209-1210, 1975.
18. Hui, L.C.K.: Color set size problem with application to string matching. In: Proc. CPM. Volume 644 of LNCS. Springer (1992) 230–243
19. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
20. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1) (2004) 53–86
21. Phan, I.Q.H., Pilbout, S.F., Fleischmann, W., Bairoch, A.: NEWT, a new taxonomy portal. Nucleic Acids Res **31**(13) (2003) 3822–3823