Fast Frequent String Mining Using Suffix Arrays

Johannes Fischer, Volker Heun and Stefan Kramer

TUM-I0512
November 05

TUM-INFO-11-I0512-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Fast Frequent String Mining Using Suffix Arrays[*]

Johannes Fischer Volker Heun
Ludwig-Maximilians-Universität München
Institut für Informatik
Amalienstr. 17, D-80333 München
Johannes.Fischer@bio.ifi.lmu.de
Volker.Heun@bio.ifi.lmu.de

Stefan Kramer
Technische Universität München
Institut für Informatik/I12
Boltzmannstr. 3, D-85748 Garching b. München
kramer@in.tum.de

Abstract

Mining frequent strings in databases has many interesting applications, e.g., in computational biology. We focus on a special kind of constraint-based frequent string mining, namely computing all strings that are frequent in one database and infrequent in another. We present a method to find such strings by using the suffix- and lcp-arrays, which can be computed extremely fast and space efficiently, and further exhibit a good locality behavior. We test our method on several biologically relevant data sets and show that it outperforms existing methods in terms of time and space.

1 Introduction

Mining in databases of graphs, trees and sequences has attracted a lot of interest in recent years. In many applications, e.g., in computational biology, the goal is to find interesting string or sequence patterns in data. In this paper, we focus on constraint-based string mining. This means that the user can impose certain conditions on patterns that must be fulfilled to make a pattern a part of the solution. In particular, we are interested in patterns that are frequent in one part of the database and infrequent in another [3].

Previous approaches to string mining are mainly based on suffix tries [3, 8]. However, the use of suffix tries implies that, in the worst case, the methods have to allocate space quadratic in the size of the database, which makes them impractical for realistically-sized problems. In contrast, we propose a new algorithm that builds upon recent progress in the construction of suffix arrays. Suffix arrays are essentially a representation of the lexicographic order of all suffixes of a string. They can be constructed very efficiently in terms of time and space. The use of suffix arrays makes string mining scalable to large databases. Taking advantage of theoretical results about suffix arrays, the algorithm efficiently extracts frequent (and infrequent) patterns from constructed suffix arrays. An interesting feature of the algorithm is that it operates in the lattice defined by the is-prefix-of and not by the is-substring-of relation. Moreover, it has a nice geometric interpretation, since we are looking for intervals in the suffix array covering a sufficient number of strings in the database. In the experiments, we show that the new method performs extremely well compared to existing approaches, and that it allows handling databases of strings that could not be handled before. We also test the potential of the method in the first step of microarray design, where probes (short stretches of sequences) differentiating between groups of sequences have to be found.

This paper is organized as follows: In Sec. 2, we introduce the notation and the prerequisites of the approach. Sec. 3 presents the algorithms for efficiently extracting solution substrings (i.e., those that satisfy the

[*]A short version of this report has been published as J. Fischer, V. Heun, S. Kramer: Fast Frequent String Mining Using Suffix Arrays, *Proceedings of the 5th IEEE International Conference on Data Mining*, IEEE Computer Society, 2005.

combined minimum and maximum frequency conditions) from suffix arrays. Next, we present experimental results comparing with existing approaches and showing the potential for real-world applications. Finally, Sec. 5 discusses the approach more generally and gives an outlook to further work.

2 Notation and tools

2.1 Strings and frequency queries

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, and ϵ is the empty string. For $\phi, \psi \in \Sigma^\star$ we write $\phi \sqsubseteq \psi$ to denote that ϕ is a (possibly empty) prefix of ψ. We further 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, $\mathrm{lcp}(\mathsf{aab}, \mathsf{abba}) = 1$.

Given a database $\mathcal{D} \subseteq \Sigma^\star$ with strings over Σ, we define the *frequency* of a pattern $\phi \in \Sigma^\star$ in $\mathcal{D}$ as

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

We can now define the predicates minfreq and maxfreq: $\mathsf{minfreq}(\phi, \mathcal{D}, min)$ is true iff $\mathsf{freq}(\phi, \mathcal{D}) \geq min$, and likewise $\mathsf{maxfreq}(\phi, \mathcal{D}, max)$ is true iff $\mathsf{freq}(\phi, \mathcal{D}) \leq max$. One usually wants to compute the set $Th(\mathsf{pred}) := \{\phi \in \Sigma^\star \;:\; \mathsf{pred}\text{ is true}\}$; i.e., all patterns that satisfy a given interestingness predicate. Because $\mathsf{minfreq}(\phi, \mathcal{D}, min)$ implies $\mathsf{minfreq}(\psi, \mathcal{D}, min)$ for all substrings ψ of ϕ, minfreq is *anti-monotonic* in the sense of [3]. Similarly, maxfreq is a *monotonic* predicate.

This article investigates a special kind of query of the form $\mathsf{a} \wedge \mathsf{m}$, where $\mathsf{a} := \mathsf{minfreq}(\phi, \mathcal{D}_1, min)$ and $\mathsf{m} := \mathsf{maxfreq}(\phi, \mathcal{D}_2, max)$. In Sec. 3 we give algorithms for computing $Th(\mathsf{a} \wedge \mathsf{m}) := Th(\mathsf{a}) \cap Th(\mathsf{m})$.

We now introduce an example that will be continued throughout this paper.

Example 1. Let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $\mathcal{D}_1 = \{\mathsf{bbabab}, \mathsf{abacac}, \mathsf{bbaaa}\}$, $\mathcal{D}_2 = \{\mathsf{aba}, \mathsf{babbc}, \mathsf{cba}\}$, $min = 2$ and $max = 2$. Then $Th(\mathsf{a} \wedge \mathsf{m}) = \{\mathsf{ab}, \mathsf{aba}, \mathsf{bb}, \mathsf{bba}\}$. Note in particular that ba belongs to $Th(\mathsf{a})$ (because it is a substring of all 3 strings in $\mathcal{D}_1$), but is not part of $Th(\mathsf{a} \wedge \mathsf{m})$ (because its frequency in $\mathcal{D}_2$ is 3). ◇

We will actually compute $Th(\mathsf{a} \wedge \mathsf{m})$ via the *border* sets

$$\mathsf{S} := \{\phi \in \Sigma^\star : \mathsf{minfreq}(\phi, \mathcal{D}_1, min) \;\wedge \forall\, x \in \Sigma : \neg\mathsf{minfreq}(\phi x, \mathcal{D}_1, min)\}$$

and

$$\begin{aligned}\mathsf{G} \;\; := \;\; & \{\psi \in \Sigma^\star : \exists\, \phi \in \mathsf{S} : \psi \sqsubseteq \phi \;\wedge \mathsf{maxfreq}(\psi, \mathcal{D}_2, max) \\ & \wedge\; \neg\mathsf{maxfreq}(\psi_{1..|\psi|-1}, \mathcal{D}_2, max)\} \;.\end{aligned}$$

The following theorem shows that these two sets characterize all strings that belong to $Th(\mathsf{a} \wedge \mathsf{m})$, and are thus a *representation* of $Th(\mathsf{a} \wedge \mathsf{m})$.

Theorem 1. *For all $\chi \in \Sigma^\star$: $\chi \in Th(\mathsf{a} \wedge \mathsf{m}) \iff \exists\, \phi \in \mathsf{S}, \psi \in \mathsf{G} \;:\; \psi \sqsubseteq \chi \sqsubseteq \phi$.*

Proof. This follows immediately from the fact that a and m are anti-monotonic and monotonic under the "$\sqsubseteq$"-relation, respectively. □

Note that Thm. 1 states that S and G are positive borders in the sense of version spaces [13] w.r.t. $\sqsubseteq$, because "$\sqsubseteq$" is a is-more-general-than relation [13]. S (or G) consists of all most specific (or all most general) strings (w.r.t. $\sqsubseteq$) satisfying both the minimum (or maximum, respectively) frequency query.

We note that the boundaries for the more familiar "is substring of"-relation could also be obtained from our S and G, namely by removing the longest proper suffixes of all $s \in \mathsf{S}$ from S, and removing the longest proper extensions of all $g \in \mathsf{G}$ from G. The reason why S and G are defined asymmetrically here will become apparent in Sec. 2.4.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
t =	b	b	a	b	a	b	$\#_1$	a	b	a	c	a	c	$\#_2$	b	b	a	a	a	$\#_3$	
SA=	7	14	20	19	18	17	5	3	8	12	10	6	16	4	2	9	15	1	13	11	
LCP=	−1	0	0	0	1	2	1	2	3	1	2	0	1	2	3	2	1	3	0	1	−1

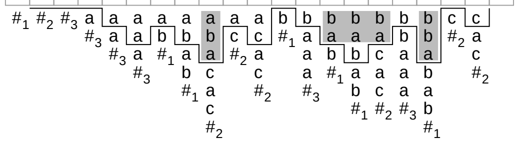


Figure 1: The suffix array for $\mathcal{D}_1$ and its LCP-table.

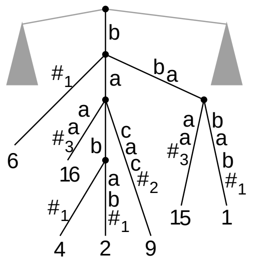


Figure 2: A part of the suffix tree for $\mathcal{D}_1$.

Example 2. Continuing from example 1, $\mathsf{S} = \{\mathtt{aba}, \mathtt{ba}, \mathtt{bba}\}$ and $\mathsf{G} = \{\mathtt{ab}, \mathtt{bb}\}$ are the borders of $Th(\mathtt{a} \wedge \mathtt{m})$ under "$\sqsubseteq$". (Note that ba would not be in S under the substring-relation.) ◇

2.2 Suffix- and LCP-arrays

This section introduces the fundamental tools we need for our algorithm. Given a text t of length n, the *suffix array* [10] for t is an array of integers $\mathsf{SA}[1,n]$ s.t. $\{\mathsf{SA}[1],\dots,\mathsf{SA}[n]\} = \{1,\dots,n\}$ and $t_{\mathsf{SA}[i]..n} \le t_{\mathsf{SA}[i+1]..n}$ for all $1 \le i < n$; i.e., SA describes the lexicographic order of t's suffixes. To avoid ambiguities one usually appends a special end-of-string symbol to the end of t which is lexicographically smaller than any symbol in Σ. Because the suffix array references n different text positions, it takes $4n$ bytes to store SA on current computer architectures (assuming n fits the computer's word size).

For expository purposes we will also need the concept of a *suffix tree*. The suffix tree [5] for t is a directed tree T with edge labels from $\Sigma^\star$, s.t. the concatenation of symbols from the root to a leaf represents a suffix of t. Further, each suffix is represented by a leaf with such a *path label*, and the labels of two outgoing edges from a node start with different characters.

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. [7]. In practice, however, the fastest algorithm for constructing SA is an $O(n^2 \log n)$-method due to Ferragina and Manzini [12], which has the further advantage that is uses only $(5+\varepsilon)n$ bytes (including the text), 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 $\mathsf{LCP}[1,n+1]$ for t is defined by $\mathsf{LCP}[i] = \mathrm{lcp}(t_{\mathsf{SA}[i]..n}, t_{\mathsf{SA}[i-1]..n})$ for all $1 < i \le n$, and $\mathsf{LCP}[1] = \mathsf{LCP}[n+1] = -1$. That is, LCP contains the lengths of the longest common prefixes of t's suffixes that are consecutive in lexicographic order. Kasai et al. [6] gave an algorithm to compute LCP in $O(n)$ time, and Manzini [11] adapted this algorithm to use only $4n$ bytes. It can be argued that most of the LCP-values are small compared to the size of the text and thus can be stored using less than $4n$ bytes, but we do not pursue this approach here.

Taken together, SA and LCP take $9n$ bytes of space, including the text t.

2.3 Generalized suffix arrays

Suppose a database $\mathcal{D}$ contains $m := |\mathcal{D}|$ strings $s_1,\dots,s_m$. To define a suffix array for $\mathcal{D}$, we form a (conceptual) string $t := s_1\#_1 s_2\#_2 \dots \#_{m-1} s_m \#_m$, where the end of string markers $\#_i$ are symbols that are not present in any of the strings s_j ($\Rightarrow n := |t| = \sum_{i=1}^{m} |s_i| + m$). We define an artificial lexicographic order on the end of string markers by $\#_1 < \dots < \#_m < a$ for all $a \in \Sigma$.[1] Similar to a generalized suffix *tree* [5] for $\mathcal{D}$, which is the (usual) suffix tree for t, we define the *generalized suffix array* for $\mathcal{D}$ as the (usual) suffix array

[1]Note that the algorithm can be implemented s.t. it uses only *one* end of string marker.

over the conceptual string t. The generalized LCP-array for $\mathcal{D}$ is also obtained by building the LCP-array for t. We will usually omit the term "generalized" when it is clear from the context.

Example 3. With the notation from example 1, the suffix array and the LCP-table for $\mathcal{D}_1$ can be seen in Fig. 1. To get a better insight into the meaning of these tables, we draw the string $t_{\mathsf{SA}[i]..n}$ below column i, until we reach one of the end of string markers. Further, the LCP-values are depicted by a line going through these strings. $\diamond$

The suffix *array* is closely related to the suffix *tree*, in the sense that a lexicographical depth-first search over T visits T's leaves in exactly the order of the suffix array. As an example, consider the part of the suffix tree for $\mathcal{D}_1$ in Fig. 2 (which shows all substrings starting with 'b'). Reading off the leaves from left to right, we get $6, 16, 4, 2, 9, 15$ and 1, which is $\mathsf{SA}[12..18]$.

The LCP-array can also be interpreted in terms of the suffix tree T, because it gives information on the depth of the internal nodes of T. This is stated more formally in the following

Theorem 2. *Let t be a string of length n, T be the corresponding suffix tree,* SA *be the corresponding suffix array, and* LCP *be the LCP-table. Then the following statements are equivalent:*

1. *There is an internal node v of T with path label ϕ.*
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) $\exists\, q \in \{l, \dots, r\}$ *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 [1] for a proof of this non-trivial result. As an example, consider the node in Fig. 2 with path label `ba`, which corresponds to the (l, r)-interval (14,16) in SA of Fig. 1. The essence of Thm. 2 is that suffix arrays, together with their LCP-array, can be used as a (very space efficient) representation of suffix trees.[2]

We define the *location* $\mathrm{loc}(s)$ of a string s in a suffix tree T as the node or edge where we end when we read s in T. We also say that $\mathsf{SA}[i]$ (or, equivalently, a leaf in T) *covers* string $s_j \in \mathcal{D}$ iff the first end of string marker in $t_{\mathsf{SA}[i]..n}$ is $\#_j$. For example, $\mathsf{SA}[16]$ covers string $s_2 = \mathtt{abacac}\#_2$ in Fig. 1, because $t_{\mathsf{SA}[16]..n} = \mathtt{bacac}\#_2\mathtt{bb}....$

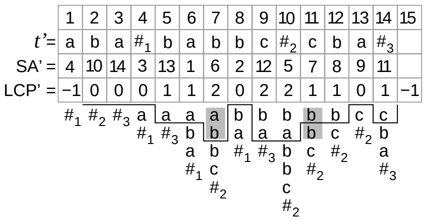
2.4 Relating SA and LCP with frequency queries

As Thm. 2 relates suffix trees to suffix arrays, we first state a well-known theorem [5] on suffix trees for minimum frequency queries.

Theorem 3. *Let $\mathcal{D}$ be a database of strings over Σ, T be the corresponding suffix tree and $\phi \in \Sigma^\star$. Then* $\mathsf{freq}(\phi, \mathcal{D}) = x$ *iff the leaves below* $\mathrm{loc}(\phi)$ *cover exactly x different strings in $\mathcal{D}$.*

This theorem implies that the maximal strings ϕ in $Th(\mathsf{minfreq}(\phi, \mathcal{D}, min))$ (w.r.t. $\sqsubseteq$) are exactly those nodes v with $v = \mathrm{loc}(\phi)$ in T for which the leaves below v cover at least min different strings, and the leaves below each of v's children cover less than min different strings in $\mathcal{D}$. For example, we have that `ba` is a maximal string in $\mathcal{D}_1$ with frequency ≥ 2, and in the tree in Fig. 2 the children below the node with path label `ba` cover s_1 (by $\mathtt{bab}\#_1$ and $\mathtt{babab}\#_1$), s_2 (by $\mathtt{bacac}\#_3$) and s_3 (by $\mathtt{baaa}\#_3$), but all of `ba`'s children cover only one string in $\mathcal{D}_1$.

[2]Though the concept of suffix links is not captured by SA and LCP.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t'=	a	b	a	$\#_1$	b	a	b	b	c	$\#_2$	c	b	a	$\#_3$	
SA' =	4	10	14	3	13	1	6	2	12	5	7	8	9	11	
LCP' =	−1	0	0	0	1	1	2	0	2	2	1	1	0	1	−1

Figure 3: The suffix array for $\mathcal{D}_2$ and its LCP-table.

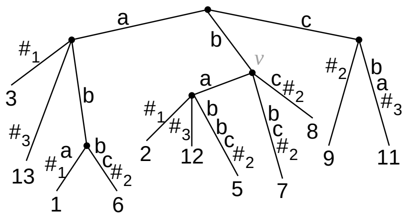


Figure 4: The suffix tree for $\mathcal{D}_2$.

Combining Thm. 2 with what we have just said, SA and LCP can be used to calculate S as follows: look for (l, r)-intervals s.t. $\mathsf{SA}[l-1], \ldots, \mathsf{SA}[r]$ cover at least min different strings in $\mathcal{D}$, and no sub-interval of (l, r) has this property. (The latter is because we want the *deepest* nodes in the suffix tree satisfying minfreq.) For example, the (l, r)-intervals of the strings in S are shown shaded in Fig. 1. These ideas will be used in Sec. 3.1.

We have seen above how Thm. 3 can be used to characterize the maximal strings in $Th(\mathsf{minfreq}(\phi, \mathcal{D}, min))$ in terms of suffix trees. Similarly, the minimal strings ψ in $Th(\mathsf{maxfreq}(\phi, \mathcal{D}, max))$ can be uniquely described as the nodes v in the suffix tree for $\mathcal{D}$ with $v = \mathrm{loc}(\psi_{1..|\psi|-1})$ s.t. the leaves below v cover more than max different strings, but the leaves below the child v' of v cover at most max different strings in $\mathcal{D}$, where the edge label between v and v' starts with $\psi_{|\psi|}$. As an example, bb is a minimal string in $\mathcal{D}_2$ with frequency at most 2, and in Fig. 4 (which shows the suffix tree for $\mathcal{D}_2$) the leaves below $v = \mathrm{loc}(\mathtt{b})$ cover more than 2 different strings in $\mathcal{D}_2$ (namely all 3 strings in $\mathcal{D}_2$), whereas the leaves below $\mathrm{loc}(\mathtt{bb})$ cover ≤ 2 different strings (only babbc).

Again, combining the ideas of the previous paragraph with Thm. 2 we get the following algorithmic idea for computing G: look for (l, r)-intervals s.t. $\mathsf{SA}[l-1], \ldots, \mathsf{SA}[r]$ cover more than max different strings in $\mathcal{D}$. Then all sub-intervals of (l, r) that cover at most max different entries in $\mathcal{D}$ are part of G. For instance, look at Fig. 3 which shows the suffix array and LCP-table for $\mathcal{D}_2$ of our running example. The interval $(5, 7)$ covers all three strings in $\mathcal{D}_2$ (implying that a is too frequent), but the sub-interval $(7, 7)$ covers only two different strings in $\mathcal{D}_2$, namely aba at $\mathsf{SA}'[6]$ and babbc at $\mathsf{SA}'[7]$. So ab is a minimal string in $Th(\mathsf{maxfreq}(\phi, \mathcal{D}_2, max))$. Further, if (l, r) is an interval s.t. $\mathsf{SA}[l-1], \ldots, \mathsf{SA}[r]$ cover more than max different strings in $\mathcal{D}$, then for $q \in \{l-1, \ldots, r\}$ the string $t_{\mathsf{SA}[q]..\mathsf{SA}[q]+l-1}$ is minimally infrequent if $\mathsf{LCP}[q] < l$ and $\mathsf{LCP}[q+1] < l$, and l is minimal. (These are the minimally infrequent strings that end on an edge leading to a leaf in the suffix tree.) An example is the interval $(9, 12)$ in Fig. 3 that covers all three strings in $\mathcal{D}_2$. But with $q = 11$ and $l = 2$ we have $\mathsf{LCP}'[11] = \mathsf{LCP}'[12] = 1 < l$, so bb is minimally infrequent. We will incorporate these ideas in Sec. 3.2.

3 Methods

The strings in $Th(\mathsf{a} \wedge \mathsf{m})$ are computed in two phases. The first, described in Sec. 3.1, finds all strings that satisfy the minimum frequency query by calculating S. The second, described in Sec. 3.2, finds for all $s \in \mathsf{S}$ the shortest prefixes that satisfy the maximum frequency query (i.e., G), and then returns all strings in $Th(\mathsf{a} \wedge \mathsf{m})$.

3.1 Answering minimum frequency queries

Algorithm 1 starts by computing the suffix array [12] and the LCP-table [11] for $\mathcal{D}_1$ (line 1). The outer while-loop (lines 4–21) scans LCP for all local maxima and tries to shorten the respective strings as little as possible s.t. they satisfy minfreq (i.e., they are part of S). We use an auxiliary bit-array *freq* in which we mark the frequent strings to avoid returning the same string more than once.

The local maxima are calculated in line 5 s.t. $\mathsf{LCP}[l-1] < \mathsf{LCP}[l] = \mathsf{LCP}[l+1] = \cdots = \mathsf{LCP}[r] > \mathsf{LCP}[r+1]$. Note that with $k := \mathsf{LCP}[r]$ we have that all length k prefixes of the $r-l+2$ strings $t_{\mathsf{SA}[l-1]..n}, t_{\mathsf{SA}[l]..n}, \ldots, t_{\mathsf{SA}[r]..n}$

are equal. Denote this prefix by p. In line 6 we count the number of patterns in $\mathcal{D}_1$ that contain p. (This number is not necessarily equal to $r - l + 2$ because one string in $\mathcal{D}_1$ could contain multiple instances of p.) Counting the frequency involves an auxiliary bit-array of length m in which we mark which strings of the database have p as a substring. These are exactly the strings which are covered by $\mathsf{SA}[l-1], \ldots, \mathsf{SA}[r]$. If the frequency of p is already greater than or equal to the minimum frequency threshold, we mark its position in SA as frequent and store p in a file S (lines 19–20). Otherwise, we try to find a shortest prefix p' of p that fulfills the minimum frequency constraint (lines 8–18). The shortest possible $p' \sqsubseteq p$ having a greater frequency than p itself has length $k := \max\{\mathsf{LCP}[l-1], \mathsf{LCP}[r+1]\}$ (line 9). If this length is ≤ 0 (i.e., no such prefix exists), we continue with the next iteration of the outer while-loop (line 10). Lines 11–16 adjust the pair (l, r) to (l', r') s.t. all LCP-values between l' and $l - 1$ (inclusive) are greater than or equal to k, and all LCP-values between $r + 1$ and r' are equal to k. If we encounter an LCP-value to the right of r greater than k we stop this iteration of the outer while-loop to find a new local maximum (line 12). If we encounter a frequent string s to the left of l we know that s must be longer than p (the string currently under consideration). So we stop this iteration of the outer while-loop (line 15), because $p' \sqsubseteq s$, and according to the definition of S we do not mark prefixes of other frequent strings as frequent.

Algorithm 1: Process minimum frequency query.

Input: database $\mathcal{D}_1$ represented by the text $t_{1..n}$, minimum frequency threshold min
Output: an alphabetically ordered file containing S

```
1  compute $\mathsf{SA}[1,n]$ and $\mathsf{LCP}[1,n+1]$
2  let *freq* be a bit-field of length $n$, initially all false
3  $r \leftarrow 1, \quad l \leftarrow 1$                                   // current $(l,r)$-interval
4  **while** $r \leq n$ **do**
5      $(l,r) \leftarrow$ next local maximum in LCP to the right of $r$ (see text)
6      $cov \leftarrow$ # strings in $\mathcal{D}_1$ covered by $\mathsf{SA}[l-1]..\mathsf{SA}[r]$
7      $k \leftarrow \mathsf{LCP}[r]$
8      **while** $cov <$ min **do**
9          $k \leftarrow \max\{\mathsf{LCP}[l-1], \mathsf{LCP}[r+1]\}$
10         **if** $k \leq 0$ **then continue** in line 4                      // not frequent
11         **while** $\mathsf{LCP}[r+1] = k$ **do** $r{+}{+}$                    // expand to the right
12         **if** $\mathsf{LCP}[r+1] > k$ **then continue** in line 4 to find a new local maximum
13         **while** $\mathsf{LCP}[l-1] \geq k$ **do**
14             $l{-}{-}$                                                     // expand to the left
15             **if** *freq*$[l-1]$ **then continue** in line 4 (this string is already frequent)
16         **end**
17         update $cov$ with newly covered strings in $\mathsf{SA}[l-1]..\mathsf{SA}[r]$
18     **end**
19     *freq*$[r] \leftarrow$ true                                          // this string is frequent
20     write $t_{\mathsf{SA}[r]..\mathsf{SA}[r]+k-1}$ to a file S
21 **end**
```

Example 4. Continuing from example 3, the algorithm first finds the local maximum $l = r = 6$ in LCP. As $\mathsf{SA}[5]$ and $\mathsf{SA}[6]$ cover only one string in $\mathcal{D}_1$, namely string s_3, the algorithm sets k to $\max\{\mathsf{LCP}[5], \mathsf{LCP}[7]\} = 1$ (line 9). Increasing r by 1 (line 11) it finds that $\mathsf{LCP}[8] = 2 > 1$, so it needs to find a new local maximum (line 12). This is at $l = r = 9$, and since $\mathsf{SA}[8]$ and $\mathsf{SA}[9]$ cover s_1 and s_2, we mark position 9 as frequent (line 19). This means that `aba` is in S (line 20). The next local maximum is at $l = r = 11$, but $\mathsf{SA}[10]$ and $\mathsf{SA}[11]$ just cover s_2, so k is set to $\max\{\mathsf{LCP}[10], \mathsf{LCP}[11]\} = 1$. Because $\mathsf{LCP}[12] = 0 < k$, r is not increased, and when decreasing l in lines 13–16, we find in line 15 that position 9 is already marked frequent and therefore go to the next iteration of the outer while-loop. The other patterns found are ba and bba. $\diamond$

3.2 Answering maximum frequency queries

We assume that the maximum frequency query follows the minimum frequency query and that it invokes a different database $\mathcal{D}_2$ (represented by the string t' of length n'). This has the consequence that the patterns $s \in \mathsf{S}$ found by the minimum frequency query first have to be located in the suffix array SA' for $\mathcal{D}_2$. Because we cannot ensure that the full string s occurs in $\mathcal{D}_2$, we search for its *longest prefix* $\mathrm{lp}_{\mathcal{D}_2}(s)$ occurring in SA'. To be precise, we define

$$\mathrm{lp}_{\mathcal{D}_2}(s) := \arg\max_{\phi \sqsubseteq s}\{|\phi| : \exists d \in \mathcal{D}_2 : \phi \preceq d\} \ .$$

As an example, for $\mathtt{ba} \in \mathsf{S}$ we have $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{ba}) = \mathtt{ba}$, because ba occurs completely in $\mathcal{D}_2$. But $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{bba}) = \mathtt{bb}$, because bb occurs in $\mathcal{D}_2$, but *not* bba.

Processing the maximum frequency query works roughly as follows: step through all patterns s in S, locate $\mathrm{lp}_{\mathcal{D}_2}(s)$ (Alg. 2) and find the shortest prefix of $\mathrm{lp}_{\mathcal{D}_2}(s)$ that satisfies maxfreq (Alg. 3). We first show how to find $\mathrm{lp}_{\mathcal{D}_2}(s)$.

A naive approach would use the suffix array search algorithm [10] for each $s \in \mathsf{S}$ (with running time $O(|s| \log n')$), resulting in $O(|\mathsf{S}| \log n')$ running time, where $|\mathsf{S}| := \sum_{s \in \mathsf{S}} |s|$. But because both S and SA' are sorted in alphabetic order, locating $\mathrm{lp}_{\mathcal{D}_2}(s)$ for all $s \in \mathsf{S}$ can be accomplished by a single pass over SA' in time $O(|\mathsf{S}| + n')$ by incorporating the LCP-information between consecutive suffixes (stored in LCP'), as described in the following paragraph.

Assume we wish to find the leftmost (i.e., smallest) position q of $\mathrm{lp}_{\mathcal{D}_2}(s)$ in SA', with respect to the constraint that if $|\mathrm{lp}_{\mathcal{D}_2}(s)| < |s|$, then $t_{\mathsf{SA}'[q+1]..\mathsf{SA}'[q+1]+\mathsf{LCP}'[q+1]} > s_{1..k+1}$. Let s_{old} be the string immediately preceding s (in lexicographic order) in S, and say that p is the position of $\mathrm{lp}_{\mathcal{D}_2}(s_{old})$ as found by the algorithm. Because $s > s_{old}$ we find that $\mathrm{lp}_{\mathcal{D}_2}(s) \geq \mathrm{lp}_{\mathcal{D}_2}(s_{old})$, so $q > p$. This implies that we only have to search from p onwards. Now assume a length-k prefix of s matches at position q in SA', i.e., $k = \mathrm{lcp}(s, t'_{\mathsf{SA}'[q]..n'})$. Now if $\mathsf{LCP}'[q+1] > k$, we know that $t'_{\mathsf{SA}'[q+1]..\mathsf{SA}'[q+1]+k-1}$ matches $s_{1..k}$, so these characters do not have to be re-matched. Further, character $t'_{\mathsf{SA}'[q+1]+k}$ cannot match s_{k+1}, so if there is a longer prefix of s than k in $\mathcal{D}_2$, it must occur at a position greater than $q+1$. If, on the other hand, $\mathsf{LCP}'[q+1] < k$, then q is the rightmost position of $\mathrm{lp}_{\mathcal{D}_2}(s)$, because all strings to the right of q in SA' are lexicographically greater than $s_{1..k}$. The only case where we have to match further characters of s and t' is when $\mathsf{LCP}'[q+1] = k$. These ideas are used in Alg. 2. The total running time of $O(|\mathsf{S}| + n')$ follows by noting that for $s \in \mathsf{S}$ each character of s is compared only once against some character from $\mathcal{D}_2$, and each position in SA' is visited at most once.

Example 5. In example 4 we have calculated $\mathsf{S} = \{\mathtt{aba}, \mathtt{ba}, \mathtt{bba}\}$, so assume for now that we have already found $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{aba})$ at position $q = 6$ in SA'. Next we search $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{ba})$. Because $k \leftarrow \mathrm{lcp}(\mathtt{aba}, \mathtt{ba}) = 0 < \mathsf{LCP}'[7]$, we simply increment q by one without comparing any letters (line 4). Now $k = 0 = \mathsf{LCP}'[8]$, and ba is matched completely at position $q = 8$ (lines 6–10). Finally, we search $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{bba})$. Because $k \leftarrow \mathrm{lcp}(\mathtt{ba}, \mathtt{bba}) = 1 < \mathsf{LCP}'[9]$, we first increase q to 9, and because $1 < \mathsf{LCP}'[10]$ we then also increase it to 10. Now $1 = \mathsf{LCP}'[11]$, so we try matching further characters in bba and $t_{\mathsf{SA}'[11]..n}$, which is just the second b. Because $\mathtt{c} > \mathtt{a}$, we exit the while-loop in line 9 with $q = 11$ being the position of $\mathrm{lp}_{\mathcal{D}_2}(\mathtt{bba}) = \mathtt{bb}$. $\diamond$

We are now ready to describe Algorithm 3 which processes the maximum frequency query. The outer for-loop (lines 3–15) steps through all $s \in \mathsf{S}$ and finds the shortest prefix of s that satisfies the maximum frequency query. This is done by finding the *longest* prefix of s with a frequency strictly greater than max, and then adding one more character to this one (cf. Sec. 2.4).

In line 4 we first locate the rightmost position p of $\mathrm{lp}_{\mathcal{D}_2}(s)$ as described before. We now want to check the frequency of the longest prefix of s having a possible frequency greater than s. If the length of $\mathrm{lp}_{\mathcal{D}_2}(s)$ (which

Algorithm 2: `find`(s, p): Find position and length of $\mathrm{lp}_{\mathcal{D}_2}(s)$ in SA'

```
   Input: string $s$ and a position $p$
   Output: pair $(q, k)$ s.t. $t'_{\mathsf{SA}'[q]..\mathsf{SA}'[q]+k-1} = \mathrm{lp}_{\mathcal{D}_2}(s)$
1  $q \leftarrow p, \quad k \leftarrow \mathrm{lcp}(s, t'_{\mathsf{SA}'[q]..n'})$
2  if $k = |s| \vee (\mathsf{LCP}'[q+1] > k \wedge t_{SA'[q+1]+k} > s_{k+1})$ then return $(q, k)$
3  while $q \leq n'$ do
4      if $\mathsf{LCP}'[q+1] > k$ then $q{+}{+}$
5      else if $\mathsf{LCP}'[q+1] < k$ then exit from while-loop
6      else // $\mathsf{LCP}'[q+1] = k$
7          while $k < |s| \wedge t'_{\mathsf{SA}'[q+1]+k} = s_{k+1}$ do $k{+}{+}$
8          $q{+}{+}$
9          if $k = |s| \vee t'_{\mathsf{SA}'[q]+k} > s_{k+1}$ then exit while
10     end
11 end
12 return $(q, k)$
```

is k) is greater than $\mathsf{LCP}'[q]$, we know that the frequency of $\mathrm{lp}_{\mathcal{D}_2}(s)$ in $\mathcal{D}_2$ (and therefore also that of s) is 1, so the first possible prefix of s having a frequency greater than s has length $\max\{\mathsf{LCP}'[p], \mathsf{LCP}'[p+1]\}$ (line 6). If, on the other hand, the length of $\mathrm{lp}_{\mathcal{D}_2}(s)$ is $\leq \mathsf{LCP}'[q]$, the first possible prefix of s with a greater frequency has length k, and k remains untouched. So k is now set to the longest prefix of s having a possible frequency greater than s.

Lines 8–9 find the leftmost and rightmost occurrences of $s_{1..k}$, and in line 10 the frequency of $s_{1..k}$ is counted. If it is greater than max, we know that all strings $s_{1..i}$, $k < i \leq |s|$, are in $Th(\mathsf{a} \wedge \mathsf{m})$ (line 14). If not, we set k to $\max\{\mathsf{LCP}'[l-1], \mathsf{LCP}'[l+1]\}$, because this is the length of the shortest prefix of $s_{1..k}$ having a possible frequency greater than that of $s_{1..k}$ (line 12). If $k = 0$, we exit from the loop, because a length-0 prefix (which is ϵ) needs not be checked (line 13).

Example 6. We continue from example 5. The first iteration of the for-loop finds $\mathrm{lp}_{\mathcal{D}_2}(\texttt{aba})$ at $q = 6$ with length $k = 3$ (the full pattern). Because $k > \mathsf{LCP}[6] = 1$, we set k to $\max\{\mathsf{LCP}'[6], \mathsf{LCP}'[7]\} = 2$ (line 6), set $r = l = 7$ (lines 8–9), and check the frequency of `ab`, which is $2 \leq max = 2$ (line 10). So k is set to $\max\{\mathsf{LCP}'[6], \mathsf{LCP}'[8]\} = 1$ (line 12), and the next iteration of the repeat-loop finds that the frequency of `a` is $3 > max$ (line 11). Therefore the loop is finished and the patterns `ab` and `aba` are returned (line 14). The next iteration of the for-loop finds $\mathrm{lp}_{\mathcal{D}_2}(\texttt{ba})$, whose frequency is already greater than max, so no patterns are returned. The last iteration returns `bb` and `bba`. ◇

We close this section by noting that minimum and maximum *length* constraints on the patterns of interest could easily be incorporated into the algorithms.

4 Experimental results

Constraint-based string mining as presented here (and further extensions) obviously has a great potential for applications working on biological sequences.[3] We applied the method to the nucleotide database of the ARB project [9]. The motivating application is microarray design, where the goal is also to find probes (short

[3]Currently, we focus on ungapped substrings without wildcards. However, it is conceivable to first find such patterns of consecutive substrings, and then find larger, gapped patterns (also known as sequential episodes in data mining) made of these as building blocks. Moreover, while there is a large body of literature on approximate string matching, index structures supporting this operation are still a topic of current research.

Algorithm 3: Process maximum frequency query.

```
Input: database $\mathcal{D}_2$ represented by the text $t'_{1..n'}$, file S (as computed by Alg. 1), maximum frequency
       threshold max
Output: all strings $s$ in $Th(\mathsf{a} \wedge \mathsf{m})$
1  compute $\mathsf{SA}'[1, n']$ and $\mathsf{LCP}'[1, n'+1]$ for $t'$
2  $p \leftarrow 1$                                        // position in SA′ where search starts
3  for $s \in \mathsf{S}$ in alphabetic order do
4      $(p, k) \leftarrow$ find $(s, p)$                                      // call Alg. 2
5      $r \leftarrow p, \quad l \leftarrow p$
6      if $k > \mathsf{LCP}'[p]$ then $k \leftarrow \max\{\mathsf{LCP}'[p], \mathsf{LCP}'[p+1]\}$      // next possible
7      repeat
8          while $\mathsf{LCP}'[r+1] \geq k$ do $r{+}{+}$                                // go right
9          while $\mathsf{LCP}'[l-1] \geq k$ do $l{-}{-}$                                 // go left
10         $cov \leftarrow$ # strings in $\mathcal{D}_2$ covered by $\mathsf{SA}'[l-1]..\mathsf{SA}'[r]$
11         if $cov >$ max then exit from repeat-loop
12         $k \leftarrow \max\{\mathsf{LCP}'[l-1], \mathsf{LCP}'[r+1]\}$
13     until $k \leq 0$
14     output $s_{1..i}$ for all $k < i \leq |s|$
15 end
```

stretches of sequence spotted on a microarray) differentiating well between groups of sequences. Additionally, the probes have to possess certain physico-chemical properties to qualify them for inclusion on the microarray.

4.1 Comparison to other methods

We implemented our method and compared it to VST [3] and FAVST [8]. All programs were written in C and compiled using the same compiler options. Further, all programs write their output in a file on a secondary storage unit. We ran several tests on an Athlon XP 3000 with 2GB of RAM. We used the Jan'03 release of the nucleotide database from the ARB project, containing rRNA of about 25,000 species. Because rRNA is highly preserved in evolution, the data bear a high sequence similarity. A phylogenetic tree partitions the species into different groups, of which we selected some for evaluation. The subsets used are shown in table 1.

For the tests in this section, we were forced to pick a very small subset of the ARB-database, because FAVST builds a suffix *trie* for the whole database $\mathcal{D}_1$ (of size $O(n^2)$ in the worst case). This is extremely space consuming, and we could only use FAVST for subsets of size less than 100KB.

The first test was a minimum frequency query on 50 random entries of the database. We adapted Alg. 1 to output *all* frequent strings instead of only S for a fair comparison. The results, for varying values of min, can be seen in Fig. 5. It is striking that our method is faster than both FAVST and VST, with factors of 10–100. Further, it is interesting to see that the running time of FAVST does not drop with increasing values of min, as it is the case for VST and our method. This is because constructing the suffix trie for the whole database is the most time consuming part of the algorithm and independent of min. Further tests with other random entries revealed similar results.

For a second test we selected two disjoint subsets with a higher biological relevance. The dataset chosen for minfreq was xanthom, and for minfreq we chose a subset of the β-dataset (β_{59}). Again, the space consumption of FAVST forced us to pick such small datasets. The results for the minimum frequency query can be seen in Fig. 6, and they resemble those from the previous experiments, except that for small values of min our method and FAVST perform about equally. Profiling showed that the sheer amount of frequent patterns to be written

subset	# species	size in KB	comment
all	25,734	43,515	complete ARB
prok	14,000	23,011	subset of all
bact	13,657	22,281	subset of prok
proteo	6062	9,867	subset of bact
β-γ	3878	6,343	subset of proteo
β	1151	1,860	subset of β-γ
β_{59}	59	88	subset of β
xanthogr	143	235	subset of proteo
xanthom	59	91	subset of xanthogr
random	50	71	subset of all
uniprot	184,304	81,615	amino acids

Table 1: Datasets used

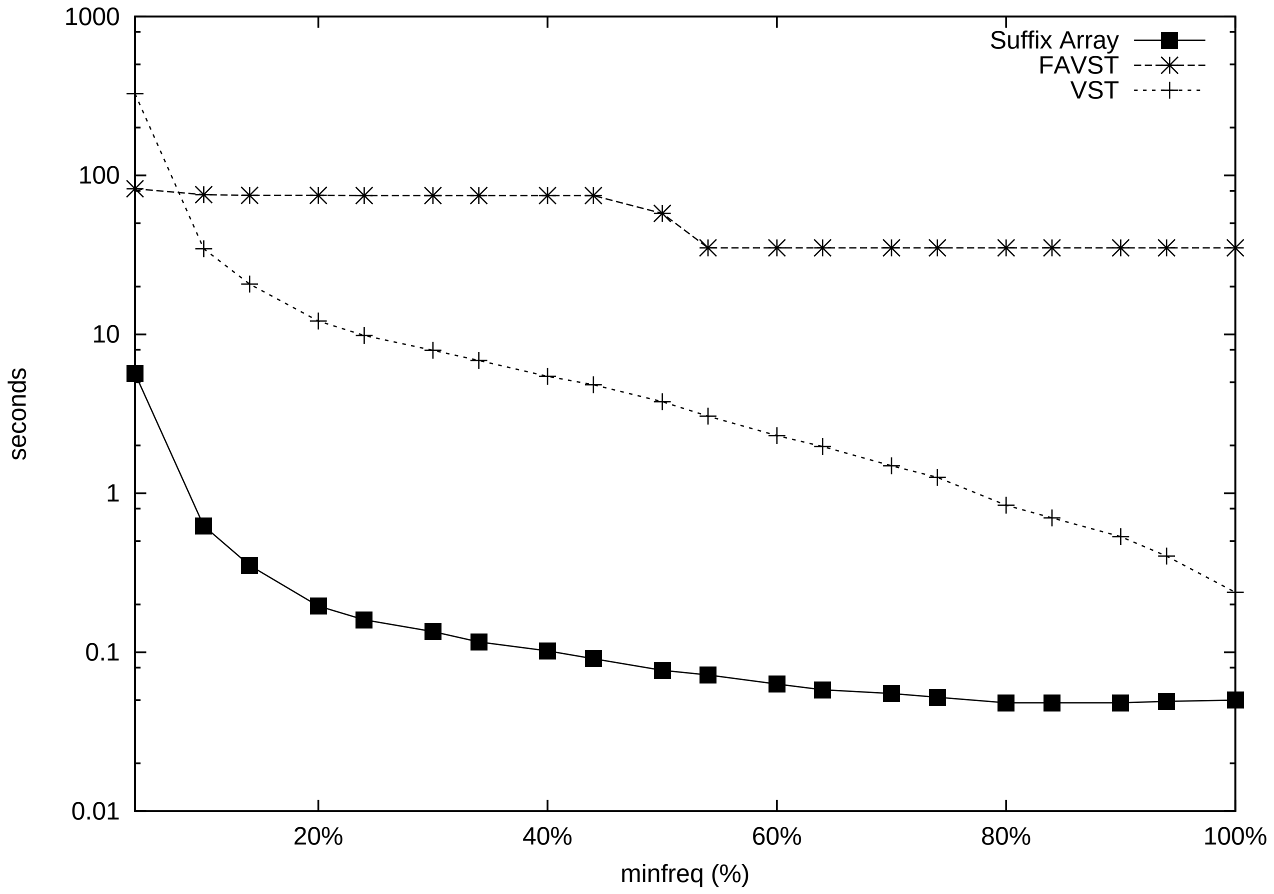


Figure 5: Results on random (log-scale, minfreq only).

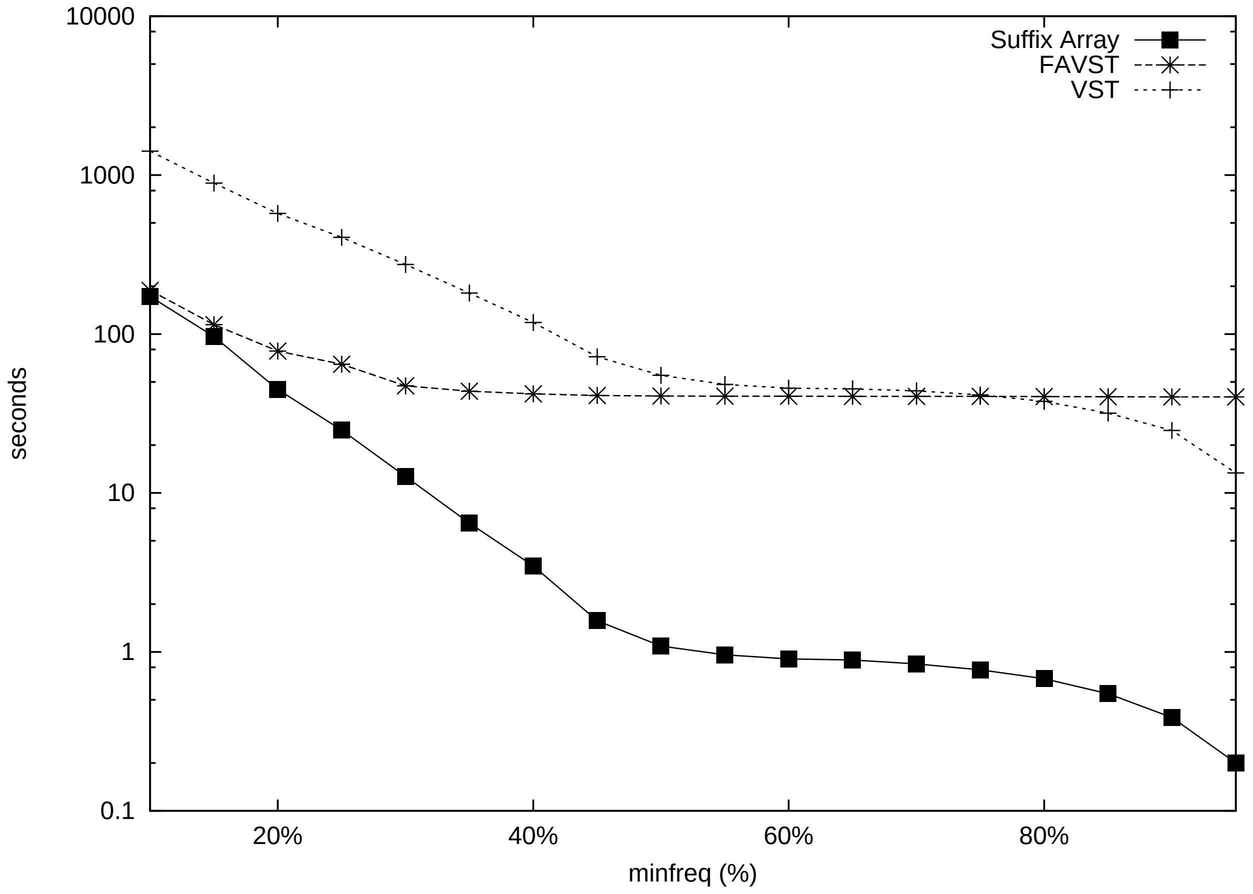


Figure 6: Results on xanthom (log-scale, minfreq only).

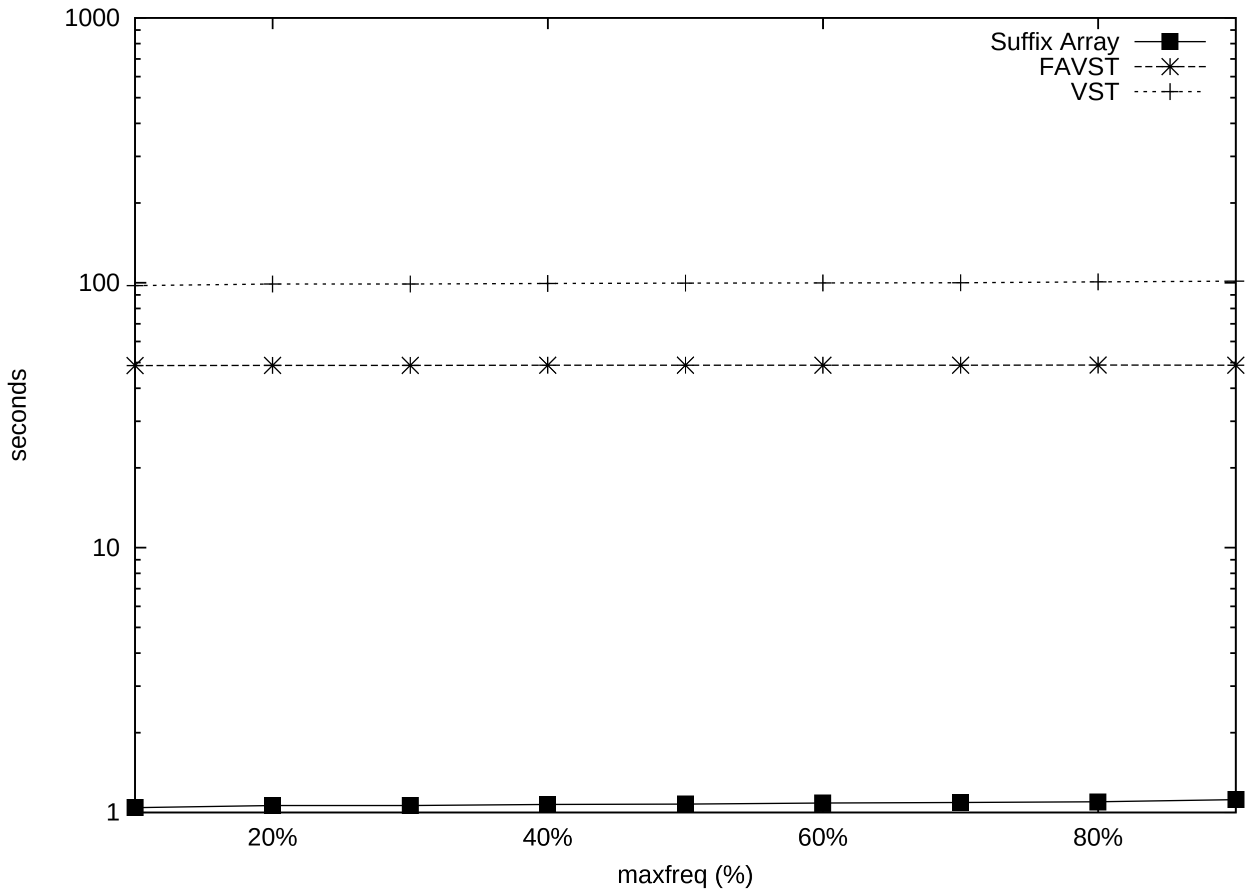


Figure 7: Results on xanthom and β_{59} (log-scale, minfreq/maxfreq combined)

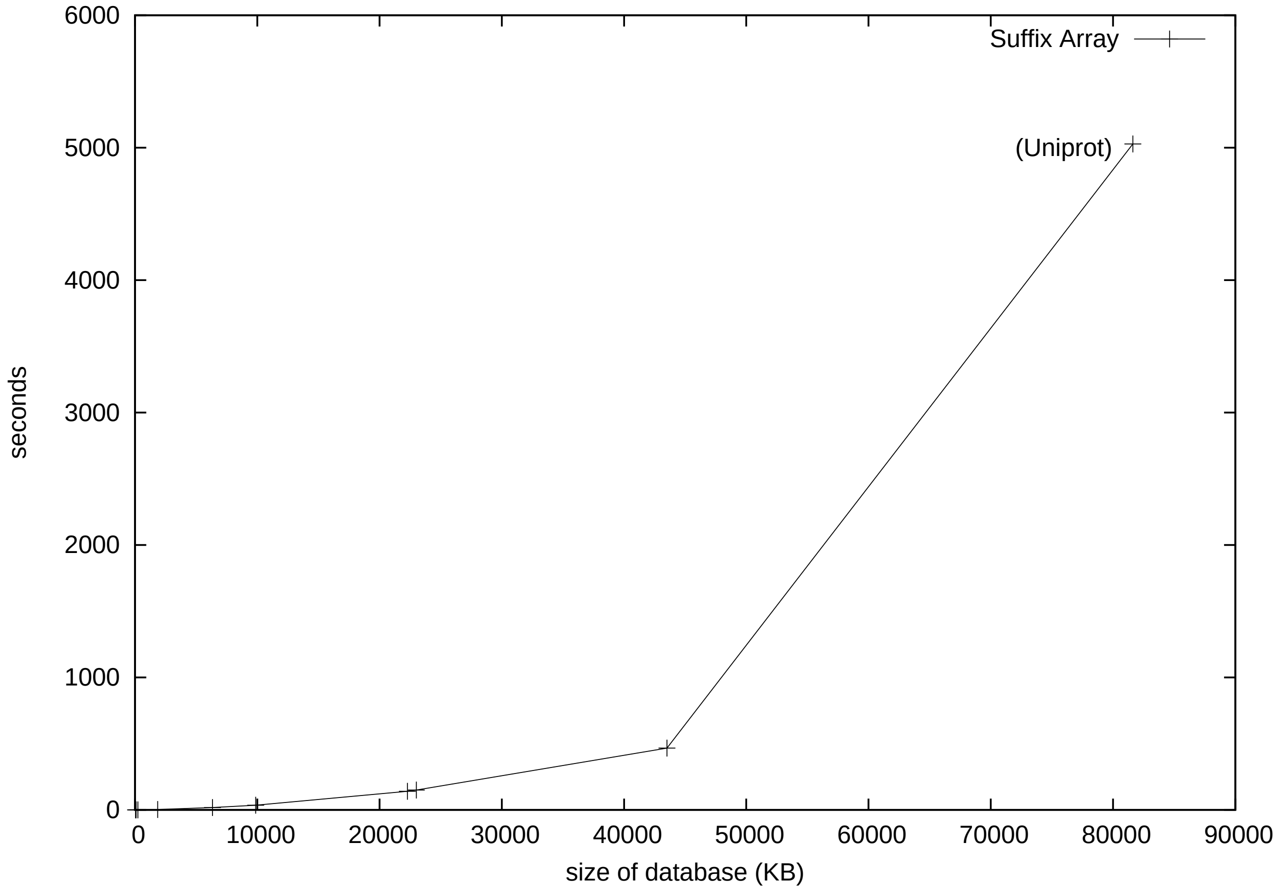


Figure 8: Running times for all databases in table 1 (minfreq = 50%).

on disk was the most time consuming part in these cases, which cannot be avoided by any method.

Figure 7 shows the results for a combined minimum and maximum frequency query, for different values of max. The value for the minfreq-query was held fixed at 30%, but similar graphs could be shown for other values of min. The running times do not depend as much on max for all three methods as it is the case for minfreq. The method based on suffix arrays proposed here outperforms existing approaches. An interesting point to note on Fig. 7 is that for small values of max, our algorithm is faster for a combined query than for a minimum query only (cf. Fig. 6 at $min = 30\%$). This is because the latter has more strings in the solution space, showing that for small datasets (such as xanthom and β_{59}) writing the patterns to disk constitutes the main part of the running time of our method.

4.2 Scalability

The aim of this section is to show that our method is capable of handling bigger databases than those in the previous tests. FAVST could not be used for comparison, because it is not able to cope with larger databases.[4] As the previous paragraph has shown that VST is always slower than the new method, we did not include it in this test. For the tests in this section we modified our program to suppress its output.

We checked our methods on all datasets in table 1. We also included the Uniprot [2] database for two reasons: First, it contains sequences of amino acids rather than nucleotides as in the other sets, which shows

[4]As an example, already the suffix trie xanthogr has 820,446,098 nodes, which is too large to fit into 2GB of main memory even with a very careful implementation of the trie.

that we can also deal with larger alphabets (size 20). Second, it is twice as large as the complete Jan'03 release of the ARB database, and is thus a real tryout for our method. The results for a minfreq-query can be seen in Fig. 8, where the frequency threshold was always 50% of the size of the database. Note that even the whole Uniprot was mined in less than 90 minutes.

5 Discussion and outlook

We presented a new approach to constraint-based string mining that effi ciently extracts information from suffi x arrays. The experiments show that the proposed algorithm outperforms existing approaches and allows the computation of interesting substrings for databases that could not be handled before. The motivating application for the experiments was microarray design. The paper also shows how recent progress from algorithms and data structures can be used to boost the performance in data mining by several orders of magnitude.

In contrast to suffi x trees, suffi x arrays exhibit nice behavior with respect to the locality of access. This enables extensions that construct suffi x arrays in secondary memory [4]. Extensions to secondary memory would allow us to scale up to very large string databases, such as all sequences available from EMBL. In future work, we will investigate how the algorithms presented in this paper can be adapted to work in a two-level memory model.

Acknowledgments

We wish to thank Lothar Richter for preparing the databases. Many thanks go to Sau Dan Lee for providing the source code of VST and FAVST.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J Discrete Algorithm*, 2:53–86, 2004.

[2] A. Bairoch et al. The universal protein resource (UniProt). *Nucleic Acids Res*, 33:D154–159, 2005.

[3] L. De Raedt, M. Jäger, S. D. Lee, and H. Mannila. A theory of inductive query answering. In *Proc. ICDM*, pages 123–130. IEEE Computer Society, 2002.

[4] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *Proc. Workshop on Algorithms and Experiments*, 2005 (in press).

[5] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[6] 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. CPM*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[7] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. CPM*, volume 2676 of *LNCS*, pages 200–210. Springer, 2003.

[8] S. D. Lee and L. De Raedt. An efficient algorithm for mining string databases under constraints. In *Proc. KDID 2004*, volume 3377 of *LNCS*, pages 108–129. Springer, 2005.

[9] W. Ludwig et al. Arb: a software environment for sequence data. *Nucleic Acids Res*, 32(4):1363–1371, 2004.

[10] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J Comput*, 22:935–948, 1993.

[11] G. Manzini. Two space saving tricks for linear time lcp array computation. In *Proc. SWAT*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

[12] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[13] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.