A Space Efficient Solution to the Frequent String Mining Problem for Many Databases

Adrian Kügel and Enno Ohlebusch

Faculty of Engineering and Computer Sciences, University of Ulm, D-89069 Ulm
`Adrian.Kuegel@uni-ulm.de, Enno.Ohlebusch@uni-ulm.de`

Abstract. The frequent string mining problem is to find all substrings of a collection of string databases which satisfy database specific minimum and maximum frequency constraints. Our contribution improves the existing linear-time algorithm for this problem in such a way that the peak memory consumption is a constant factor of the size of the largest database of strings. We show how the results for each database can be stored implicitly in space proportional to the size of the database, making it possible to traverse the results in lexicographical order. Furthermore, we present a linear-time algorithm which calculates the intersection of the results of different databases. This algorithm is based on an algorithm to merge two suffix arrays, and our modification allows us to also calculate the LCP table of the resulting suffix array during the merging.

1 Introduction

In string mining problems, one is given m databases $\mathcal{D}_1, ..., \mathcal{D}_m$ of strings and searches for the (unknown) strings that fulfill certain constraints, which are usually specified by the user. Here, we focus on the frequent string mining problem. In this problem, the constraints consist of m pairs of frequency thresholds $(\mathit{minf}_1, \mathit{maxf}_1), ..., (\mathit{minf}_m, \mathit{maxf}_m)$ and one wants to find all strings ϕ that satisfy $\mathit{minf}_i \leq \mathit{freq}(\phi, \mathcal{D}_i) \leq \mathit{maxf}_i$ for all i with $1 \leq i \leq m$, where $\mathit{freq}(\phi, \mathcal{D}_i) = |\{\psi \in \mathcal{D}_i : \phi \text{ is a substring of } \psi\}|$.

We would like to give a medical example as a motivation to study the problem. Suppose a genetic disease, e.g. *Huntington's disease*, is suspected of being caused by a defect on a certain locus of a certain chromosome, say on the short arm of chromosome 4. To find the cause of the disease, a possible approach would be to sequence that segment of the DNA molecules of many healthy individuals and ill persons. Then one database contains the DNA sequences of the healthy individuals, while the second database contains the DNA sequences of the ill individuals. Now, one searches for all strings (contiguous DNA subsequences) that occur frequently (or always) in one of the databases and not too often (or never) in the other database. If one finds, for example, that the string CAGCAGCAG...CAG, in which the codon CAG (coding for the amino acid glutamine) is tandemly repeated more than 36 times, occurs frequently (or always) in the database of ill persons but not too often (or never) in the database of healthy persons, then this gives a hypothesis for the cause of the disease.

Fischer et al. [1] presented an algorithm that solves the frequent string mining problem in optimal time, that is, in time linear in the size of the input (the databases) and the output (the strings that satisfy the constraints). Although their algorithm is a breakthrough in string mining, it still has certain disadvantages. First, their method is based on the construction of the suffix array of a very long string, namely the concatenation of all strings in all databases (the strings are separated by special separator symbols). Thus, the space consumption of their algorithm is proportional to the space occupied by all databases. It turned out that this space consumption is the bottleneck of their algorithm (Fischer 2007, personal communication). Second, the method is not very flexible if one is interested in different combinations of databases. For example, if just one database is replaced with another database, then the whole procedure has to be restarted again: One must construct the suffix array of the concatenation of all strings in all databases, etc. Third, their result is based on the assumption that the number of databases is treated as a constant.

In this paper, we presented an algorithm without these disadvantages. First, its peak memory consumption is merely proportional to the size of the largest database. Second, it is flexible in the sense that one of several databases can be replaced without having to recalculate everything, that is, intermediate data can be stored on file and be reused. Third, our algorithm has optimal worst case running time, regardless of the number of databases.

2 Preliminaries

We will consider strings $\phi = \phi_1 ... \phi_n$ consisting of symbols ϕ_i from a ordered alphabet Σ of constant size. The length of a string ϕ is the number of symbols it contains, and is denoted by $|\phi|$. A substring ranging from position n to m will be written as $\phi_{n..m}$. A substring $\phi_{n..|\phi|}$ is also called a suffix of ϕ, and a substring $\phi_{1..m}$ is also called a prefix of ϕ.

Let Σ^* be the set of all strings over Σ, and $\phi, \psi \in \Sigma^*$. If ϕ is a substring of ψ, we write $\phi \preceq \psi$. We define $\mathrm{lcp}(\phi, \psi)$ to be the length of the longest common prefix of ϕ and ψ.

We will call $\mathcal{D} \subseteq \Sigma^*$ a database of strings over Σ, and $|\mathcal{D}|$ denotes the number of strings in $\mathcal{D}$. The frequency of a string $\phi \in \Sigma^*$ is then defined as the number of strings of $\mathcal{D}$ which have ϕ as a substring. Formally, we can define it as follows: $\mathit{freq}(\phi, \mathcal{D}) := |\{\psi \in \mathcal{D} : \phi \preceq \psi\}|$. Note that we do not count duplicate occurrences of ϕ within one string of $\mathcal{D}$.

We will write $A[i]$ to refer to the value at position i in an array A. Arrays will be indexed starting from one (if not indicated otherwise). If we have an array A of tuples, where the tuples consist of variables (for example (a, b)), we will refer to the individual tuple entries as $A[i]$.<variable_name> (for example $A[i].a$).

We will use the notation $[i, j]$ to denote an interval of natural numbers.

The *suffix array SA* of a string ϕ is an array of integers in the range 1 to n, which describes the lexicographic order of the n suffixes of ϕ. More precisely,

$\phi_{SA[1]..n}, \phi_{SA[2]..n}, \ldots, \phi_{SA[n]..n}$ is the sequence of suffixes of ϕ in ascending lexicographic order; see Fig. 1.

We assume that we only calculate suffix arrays for strings of size $< 2^{32}$, i.e. we can store each number using 4 bytes of memory. The suffix array can be constructed in linear time [2–4].

In addition to the suffix array, we also need the *inverse suffix array* SA^{-1}, which is defined by $SA^{-1}[SA[i]] = i$ for all $1 \leq i \leq n$.

i	SA^{-1}	SA	LCP	$\phi_{SA[i]..16}$
1	9	16	0	$
2	8	15	0	#$
3	7	11	1	#aba#$
4	6	5	1	#baaab#aba#$
5	4	14	0	a#$
6	16	4	2	a#baaab#aba#$
7	10	3	1	aa#baaab#aba#$
8	11	2	2	aaa#baaab#aba#$
9	12	1	3	aaaa#baaab#aba#$
10	14	7	3	aaab#aba#$
11	3	8	2	aab#aba#$
12	13	9	1	ab#aba#$
13	15	12	2	aba#$
14	5	10	0	b#aba#$
15	2	13	1	ba#$
16	1	6	2	baaab#aba#$

Fig. 1. Suffix array, inverse suffix array and LCP table for the string ϕ = aaaa#baaab#aba#$

The *LCP table* is an array of integers which is defined relative to the suffix array of a string ϕ. It stores the length of the longest common prefix of two adjacent suffixes in the lexicographically ordered list of suffixes. Formally, $LCP[i] = lcp(\phi_{SA[i]..n}, \phi_{SA[i-1]..n})$ for $2 \leq i \leq n$, and $LCP[1] = 0$. The LCP table can be calculated in $\mathcal{O}(n)$ from the suffix array and the inverse suffix array (cf. [5]).

The LCP table can also be used to determine the length of the longest common prefix of several consecutive suffixes in the lexicographically ordered list of suffixes. Let $\phi_{SA[i]..n}, \ldots, \phi_{SA[j]..n}$ $(1 \leq i < j \leq n)$ be these consecutive suffixes. The length of the longest common prefix is $\min_{i<k\leq j}\{LCP[k]\}$.

We adopt the definition of lcp-intervals from [6]. Let $1 \leq i < j \leq n$. The interval $[i, j]$ is an *lcp-interval* of lcp-value l (also called *l-interval*), if the following conditions hold:
1. $LCP[i] < l$ and $LCP[j + 1] < l$.

2. $LCP[k] \geq l$ for all k with $i < k \leq j$.
3. $LCP[k] = l$ for at least one $i < k \leq j$.

Alternatively, for an l-interval $[i, j]$ we may also write l-$[i, j]$. Indices p with $i < p \leq j$ and $LCP[p] = l$ are called l-indices. Informally, an l-interval is a maximal set of lexicographically consecutive suffixes which have a longest common prefix of length l.

Let $[i, j]$ be an lcp-interval, and let ω be the longest common prefix of the suffixes $\phi_{SA[i]..n}, ..., \phi_{SA[j]..n}$. Then $[i, j]$ is also called the *ω-interval*.

We can define a parent-child relationship for lcp-intervals (cf. [6]): We say an l'-interval $[i', j']$ is embedded in an l-interval $[i, j]$ if $i \leq i' \leq j' \leq j$ and $l < l'$. Consequently we say the l-interval $[i, j]$ encloses the l'-interval $[i', j']$. If there is no other lcp-interval $[i'', j'']$ enclosing $[i', j']$ embedded in $[i, j]$, we say that the lcp-interval $[i', j']$ is a child interval of $[i, j]$, and $[i, j]$ is the parent interval of $[i', j']$. This parent-child relationship defines a tree of all lcp-intervals, which we will call the *lcp-interval tree*. We label each edge from an l-interval $[i, j]$ to a child interval l'-$[i', j']$ by the string $\phi_{SA[i]+l..SA[i]+l'-1}$. Note that this labelling is only for ease of presentation and is not used in an actual implementation. If we concatenate the edge labels from the root of the lcp-interval tree to some l-interval $[i, j]$, we get the string $\phi_{SA[i]..SA[i]+l-1}$. This is exactly the longest common prefix of the suffixes $\phi_{SA[i]..n}, ..., \phi_{SA[j]..n}$.

Let $[i, j]$ be an l-interval, and $p_1, p_2, ..., p_m$ be the l-indices. The child intervals of $[i, j]$ are $[i, p_1 - 1], [p_1, p_2 - 1], ..., [p_m, j]$. Some of these intervals can be singleton intervals, which are strictly speaking no lcp-intervals, but we can extend the definition of lcp-intervals to also include singleton intervals. We assign an lcp-value of $n - SA[i] + 1$ to the singleton interval $[i, i]$.

For every $a\omega$-interval ($a \in \Sigma$) of lcp-value l there is an ω-interval of lcp-value $l - 1$. We call the ω-interval the *suffix link interval* of the $a\omega$-interval. Since each l-index belongs to exactly one lcp-interval, we can store the left and right boundary of the suffix link interval of an lcp-interval at the first l-index. Suffix links can be constructed in linear time, cf. [6, 7].

In order to be able to evaluate so-called range minimum queries of the form $RMQ_{LCP}(i, j) := \arg\min_{i<k\leq j}\{LCP[k]\}$ in constant time we use the data structure presented in [8] which can be calculated in $\mathcal{O}(n)$ time using only $o(n)$ bits of extra memory. This data structure returns the smallest index k for a query if the answer is not unique. Therefore, it can also be used to traverse the lcp-interval tree. To determine the first l-index p_1 of an lcp-interval $[a, b]$, we evaluate $RMQ_{LCP}(a + 1, b)$. Given the position of some l-index p_i, the position of the next l-index can be found by evaluating $RMQ_{LCP}(p_i + 1, b)$. There is no next l-index if $p_i = b$ or if $LCP\left[RMQ_{LCP}(p_i + 1, b)\right] > LCP[p_i]$.

3 Algorithm for the Frequent String Mining Problem

The *Frequent String Mining Problem* is defined as follows (cf. [1]): Given m databases $\mathcal{D}_1, ..., \mathcal{D}_m$ of strings over Σ and m pairs of positive frequency thresholds $(\mathit{minf}_1, \mathit{maxf}_1), ..., (\mathit{minf}_m, \mathit{maxf}_m)$, find all strings $\phi \in \Sigma^*$ that satisfy $\mathit{minf}_i \leq \mathit{freq}(\phi, \mathcal{D}_i) \leq \mathit{maxf}_i$ for all $1 \leq i \leq m$. We will call the strings $\phi \in \Sigma^*$ which satisfy $\mathit{minf}_i \leq \mathit{freq}(\phi, \mathcal{D}_i) \leq \mathit{maxf}_i$ for at least one $1 \leq i \leq m$ *relevant substrings*. The solution to the Frequent String Mining Problem is the intersection of the relevant substrings of each database $\mathcal{D}_i$.

We now give an overview of our algorithm, which is explained in more detail in the following sections.

- For each database $\mathcal{D}$ from the set of databases $\{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$ do:
  - Preprocessing (as in [1], only for one database at a time):
    - $T^{\mathcal{D}} = s^1\#\ldots s^i\#\ldots\#s^{|\mathcal{D}|}\#\$$, where $\mathcal{D} = \{s^1, \ldots, s^i, \ldots, s^{|\mathcal{D}|}\}$ consists of the strings s^i.
    - Construct the suffix array SA and the LCP array of $T^{\mathcal{D}}$.
    - Preprocess the LCP array so that range minimum queries can be answered in constant time.
  - Extraction phase: [1]
    - Calculate array $C'_{\mathcal{D}}$ as in [1].
    - For each ω-interval $[l, r]$ compute $\mathit{freq}(\omega, \mathcal{D}) = S_{\mathcal{D}}(\omega) - C_{\mathcal{D}}(\omega)$, where $S_{\mathcal{D}}(\omega) = r - l + 1$ and $C_{\mathcal{D}}(\omega) = \sum_{i=l+1}^{r} C'_{\mathcal{D}}[i]$ is a correction term, see Sect. 3.2.
    - Store each relevant substring ϕ (i.e. $\mathit{minf} \leq \mathit{freq}(\phi, \mathcal{D}) \leq \mathit{maxf}$) at the lexicographically smallest suffix which has ϕ as a prefix. For details how to store these strings efficiently as results intervals, see Sect. 3.3.
    - Remove all relevant substrings that contain the separator symbol $\#$.
- Iteratively calculate the intersection of the relevant substrings of databases $\mathcal{D}_1$ and $\mathcal{D}_2$, then the intersection of the result with the relevant substrings of $\mathcal{D}_3$, and so on.
  - Intersection of relevant substrings of two databases $\mathcal{D}_1$ and $\mathcal{D}_2$:
    - Match the string $T^{\mathcal{D}_2}$ against the suffix array of $T^{\mathcal{D}_1}$, and calculate values which can be used to merge the suffix arrays of $T^{\mathcal{D}_1}$ and $T^{\mathcal{D}_2}$.
    - Process all suffixes of $T^{\mathcal{D}_1}$ and $T^{\mathcal{D}_2}$ in lexicographical order using the information calculated during the matching.
    - Reassign common relevant substrings to suffixes of $T^{\mathcal{D}_1}$.

3.1 Preprocessing step

We define $T := s^1\#s^2\#\ldots\#s^{|\mathcal{D}|}\#\$$, so T is a string consisting of the concatenation of the strings in $\mathcal{D}$, using $\#$ as a separation symbol and $\$$ as termination

[1] Although the algorithm of [1] has a similar extraction phase, we want to stress that our non-recursive calculation of the frequency $\mathit{freq}(\phi, \mathcal{D})$, as well as our implicit representation of the relevant substrings, are new.

symbol. Let n denote the length of T. # and \$ are selected in such a way that they do not occur in any string of $\mathcal{D}$. Note that it would be easier to use pairwise different separation symbols, but this would mean that the alphabet size is not constant any more. We require it to be constant, however, to obtain linear time complexity. We can work around this problem by filtering out substrings which have been wrongly recognized as relevant substrings at the end of the extraction phase.

In the preprocessing step we will set up data structures which are needed in later steps of the algorithm. In particular, we calculate the suffix array SA and the LCP table LCP for T. Furthermore, we calculate a data structure which supports range minimum queries on the LCP table in $\mathcal{O}(1)$ time. These data structures can be calculated in linear time.

3.2 Frequency calculation using correction terms

To solve the Frequent String Mining Problem, we need an efficient method to calculate $\mathit{freq}(\phi, \mathcal{D})$ for $\mathcal{D} \in \{\mathcal{D}_1, ..., \mathcal{D}_m\}$, where ϕ is a string which occurs as substring in a string of at least one of the databases. The idea used in [1] is to first calculate the number of times that a string ϕ occurs in $\mathcal{D}$ and then subtract so called *correction terms* which take care of multiple occurrences within the same string of $\mathcal{D}$. The method to calculate the correction terms is based on Hui's color set size technique (cf. [9]). As in [1] we will use the following definitions:

Let $\mathcal{D} = \{s^1, ..., s^{|\mathcal{D}|}\}$ be a given database of strings s^i.

$$S_{\mathcal{D}}(\phi) = |\{(j,k) : s^k_{j..j+|\phi|-1} = \phi\}| \tag{1}$$

$$C_{\mathcal{D}}(\phi) = \sum_{\substack{s^k \in \mathcal{D} \\ \phi \preceq s^k}} (|\{j : s^k_{j..j+|\phi|-1} = \phi\}| - 1) \tag{2}$$

Here, $S_{\mathcal{D}}(\phi)$ denotes the total number of occurrences of ϕ in $\mathcal{D}$, and $C_{\mathcal{D}}(\phi)$ is the correction term. Then, $\mathit{freq}(\phi, \mathcal{D}) = S_{\mathcal{D}}(\phi) - C_{\mathcal{D}}(\phi)$.

As in [1], we will use an array C' of length n to store intermediate values which can be used to calculate the correction terms. For each pair $T_{SA[i]..n}, T_{SA[j]..n}$ of lexicographically adjacent suffixes from the same string, we increase $C'[m]$ by one, where $m = \arg\min_{i<m\leq j} LCP[m]$. The details of the algorithm to calculate the array C' can be found in [1].

Lemma 1. *Let $\{T_{SA[i]..n} : l \leq i \leq r\}$ be all suffixes which have ϕ as prefix. Then, $C_{\mathcal{D}}(\phi) = \sum_{i=l+1}^{r} C'[i]$.*

Proof. Since ϕ is a prefix of $T_{SA[i]..n}$ for $l \leq i \leq r$, $LCP[i] \geq |\phi|$ for $l < i \leq r$. $C'[i]$ was increased only for pairs of lexicographically adjacent suffixes from the same string which have a longest common prefix of $T_{SA[i]..SA[i]+LCP[i]-1}$, so ϕ must be a prefix of their longest common prefix. Also, if two lexicographically adjacent suffixes from the same string have a longest common prefix which in turn has ϕ as a prefix, then both suffixes have ϕ as a prefix. Therefore some value $C'[i]$ has been increased, where $l < i \leq r$. Thus, it follows that $C_{\mathcal{D}}(\phi) = \sum_{i=l+1}^{r} C'[i]$. □

3.3 Extraction of the Relevant Substrings

The extraction of the relevant substrings is done by a post-order traversal of the lcp-interval tree. We can use the fact that for each ω-interval, the frequency of ω is the same as the frequency of $\omega_{1..i}$ ($l < i \leq |\omega|$), where l is the lcp-value of the parent lcp-interval. Therefore, we only need to calculate the frequency of ω for each ω-interval.

To determine the frequency $\mathit{freq}(\omega, \mathcal{D})$ of a longest common prefix ω of some ω-interval $[l..r]$, we need to calculate $S_{\mathcal{D}}(\omega)$ and $C_{\mathcal{D}}(\omega)$. Since we process only one database at a time, the value $S_{\mathcal{D}}(\omega)$ is just the size of the lcp-interval. Also, according to Lemma 1, $C_{\mathcal{D}}(\omega) = \sum_{i=l+1..r} C'[i]$. In an array C'' we will store the partial sums of the values C', formally, $C''[i] = \sum_{j=1..i} C'[j]$. We can evaluate $\sum_{i=l+1..r} C'[i]$ as $C''[r] - C''[l]$. In [1], a recursive calculation of $S_{\mathcal{D}}$ and $C_{\mathcal{D}}$ is used. Our simplification to calculate $C_{\mathcal{D}}$ could also be applied to their algorithm.

In [1] the results are not printed in lexicographic order. For the purpose of intersecting results of different string databases it would be better, however, to obtain the results in lexicographic order. We have found a way to store the results implicitly and process them later in lexicographic order.

All relevant substrings are prefix of at least one suffix, i.e. it is possible to assign each relevant substring to exactly one suffix which has this substring as a prefix. We will assign each relevant substring to the lexicographically smallest suffix which has this substring as a prefix. This enables us to print all relevant substrings in lexicographic order by processing the suffixes in lexicographic order, and print all relevant substrings which are assigned to the current suffix in order of increasing lengths.

Lemma 2. *Let $T_{p..n}$ be a suffix with at least one assigned relevant substring. Let a be the minimum length and b the maximum length of all relevant substrings assigned to $T_{p..n}$. Then for each $a \leq i \leq b$ there exists a relevant substring of length i which is assigned to $T_{p..n}$.*

Proof. Assume there is an $i \in [a, b]$ such that no relevant substring of length i was assigned to $T_{p..n}$. The string $T_{p..p+i-1}$ must be a relevant substring, because $\mathit{minf} \leq \mathit{freq}(T_{p..p+b-1}, \mathcal{D}) \leq \mathit{freq}(T_{p..p+i-1}, \mathcal{D}) \leq \mathit{freq}(T_{p..p+a-1}, \mathcal{D}) \leq \mathit{maxf}$. Obviously, $T_{p..p+i-1}$ is a prefix of $T_{p..n}$, so if it has not been assigned to $T_{p..n}$, it must have been assigned to a lexicographically smaller suffix. But then, $T_{p..p+a-1}$ can be assigned to the same suffix as $T_{p..p+i-1}$. This is a contradiction to the fact that $T_{p..p+a-1}$ has already been assigned to the lexicographically smallest suffix which has $T_{p..p+a-1}$ as a prefix. $\square$

In other words, for each suffix, the lengths of assigned relevant substrings form a (possibly empty) interval $[a, b]$. Let us call this interval the *results interval*.

The post-order traversal of the lcp-interval tree can be done as described in [5]. We go through the suffixes in lexicographic order. This means, in step i we process suffix $T_{SA[i]..n}$. We maintain a stack with values (l, h) where h is the length of the longest common prefix of $T_{SA[l]..n}$ and $T_{SA[i]..n}$, and $LCP[l] < h$. This corresponds to an lcp-interval $[l..r]$ with $r \geq i$. The tuples on the stack are sorted by l and h, i.e. there is no tuple (l, h) on top of a tuple (l_2, h_2) with $l \leq l_2$ or $h \leq h_2$.

At the beginning of step i, the stack consists of tuples corresponding to ω-intervals where ω is a prefix of $T_{SA[i-1]..n}$. Now, if there is an ω-interval on the stack with $|\omega| > LCP[i]$, then ω is not a prefix of $T_{SA[i]..n}$, because $T_{SA[i]+LCP[i]} \neq T_{SA[i-1]+LCP[i]}$. Therefore, we can remove all tuples with a value of $h > LCP[i]$. This means we have found the right boundary of an h-interval, and we can now calculate the frequency of the corresponding longest common prefix. We store the value l of the last tuple to be removed; this is the smallest index such that $\forall j : l < j \leq i, LCP[j] \geq LCP[i]$. If no tuple has been removed, we set l to $i - 1$. All remaining tuples correspond to lcp-intervals with longest common prefixes which are a prefix of $T_{SA[i]..n}$. We add another tuple $(l, LCP[i])$ to the stack if the tuple at the top of the stack has a value of $h < LCP[i]$. This corresponds to the lcp-interval starting at l which has a longest common prefix of length $LCP[i]$.

We can improve memory usage by storing only the values l on the stack. The h values can be calculated using the LCP table. This is very similar to the method used in [6] to calculate the child table. Let the stack contain the values $l_1, l_2, ..., l_k$ where l_k is the tuple on top of the stack. We know that $l_1 < l_2 < ... < l_k$, and each value l_i $(1 \leq i \leq k)$ is the left boundary of an ω_i-interval, where ω_i is a prefix of the currently processed suffix. Therefore, in the lcp-interval tree, the lcp-interval with left boundary l_i is a parent of the lcp-interval with left boundary l_{i+1} for each i with $1 \leq i < k$. Moreover, l_{i+1} must be an l-index of the lcp-interval with left boundary l_i, and it follows that $h_i = LCP[l_{i+1}]$. Since l_k corresponds to the singleton lcp-interval $[l_k, l_k]$, we know that $h_k = n - SA[l_k] + 1$, thus we can calculate all h values without having to store them on the stack.

To assign relevant substrings to suffixes, we also keep two arrays a and b of length n. When removing a value l from the stack in step i, we calculate the frequency of ω for the ω-interval $[l, i - 1]$ as described above. If we find that $\omega_{1..LCP[l]+1}, \ldots, \omega_{1..|\omega|}$ are relevant substrings, we can update $a[l]$ and $b[l]$. According to Lemma 2, we only need to keep track of the minimum and maximum length of all relevant substrings being assigned. $a[l]$ will hold the minimum length, $b[l]$ will hold the maximum length of the relevant substrings assigned to the suffix $\mathrm{T}_{l..n}$. Because we do a post-order traversal of the lcp-interval tree, we know that for each suffix the first relevant substring to be assigned is the one with maximum length, and the last relevant substring to be assigned is the one with minimum length.

To remove wrongly recognized relevant substrings, we calculate for each suffix the first occurrence of the separation symbol $\#$, and reduce the size of the results intervals such that they do not include any separation symbol. Let $next(i)$ $(1 \leq i < n)$ point to the first occurrence of the separation symbol $\#$ in suffix $T_{i..n}$, and $next(n) = 1$. Then, $next(i) = 1$ if $T_i = \#$, otherwise $next(i) = 1 + next(i+1)$. Therefore, the values $next$ can be calculated by processing the suffixes of T in order of increasing length. Using the $next$ values, we adjust the b values of the results intervals to $\min\{b[i], next(SA[i])-1\}$, thereby making sure that no results interval includes a separation symbol.

We store n tuples $(SA[i], LCP[i], a[i], b[i])$ which represent the information about the relevant substrings. Suffixes $T_{SA[i]..n}$ to which no relevant substring has been assigned will have $a[i] > b[i]$. The tuples with $a[i] > b[i]$ do not have to be stored explicitly, we just need to mark the corresponding suffixes that they do not have any relevant substring assigned.

3.4 Intersection of results of several string databases

To find those substrings which satisfy the frequency conditions for all databases, we need to intersect the relevant substrings of each database. The relevant substrings are represented as a table of tuples, as described in the previous section. Let us call such a table of tuples a *result table*. A result table is always linked to a certain database of strings, and the tuple values refer to the string T representing a database $\mathcal{D}$. This means each tuple represents a suffix of $T^{\mathcal{D}}$.

We use a kind of merging algorithm to build the intersection of two result tables and get a result table representing the intersection of the two input result tables. This algorithm is a modified version of the algorithm of [10] which merges two suffix arrays. The output result table will have the same format as the input result table, assigning the relevant substrings which occur in both input result tables to the lexicographically smallest suffix of the first input result table which has this substring as a prefix. It can be seen that actually no relevant substring of the first result table is reassigned, only the results intervals for some suffixes may be shortened. Therefore, we can use the merging algorithm iteratively to produce the intersection of more than two result tables.

Let L_1 and L_2 be result tables linked to databases $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Let $T^{\mathcal{D}_1}$ be the string representing $\mathcal{D}_1$ and $T^{\mathcal{D}_2}$ be the string representing $\mathcal{D}_2$. Furthermore, let $n_1 = |T^{\mathcal{D}_1}|$ and $n_2 = |T^{\mathcal{D}_2}|$. Let SA be the suffix array and LCP be the LCP table for the string $T^{\mathcal{D}_1}$.

In order to process the suffixes of $T^{\mathcal{D}_1}$ and $T^{\mathcal{D}_2}$ in lexicographical order, as in [10] we calculate values $c[i]$ for each suffix $T^{\mathcal{D}_1}_{SA[i]..n_1}$, which indicate how many suffixes of $T^{\mathcal{D}_2}$ have to be placed between $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$ and $T^{\mathcal{D}_1}_{SA[i]..n_1}$. The processing of the suffixes in lexicographical order can then be done easily: in step i we select the next $c[i]$ suffixes from the suffix array of $T^{\mathcal{D}_2}$, and then suffix $T^{\mathcal{D}_1}_{SA[i]..n_1}$. Note that in constrast to [10], we do not store the merged suffix array.

Calculating the c values can be done by matching the string $T^{\mathcal{D}_2}$ against the enhanced suffix array of $T^{\mathcal{D}_1}$. We calculate the index $p(i)$ during the matching such that $T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1} \leq T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[p(i)]..n_1}$, and then increase the counter $c[p(i)]$ by one.

We define the *matching statistics* $ms(i)$ $(1 \leq i \leq n_2)$ to be the length of the longest prefix of $T^{\mathcal{D}_2}_{i..n_2}$ which matches a substring of $T^{\mathcal{D}_1}$. Using suffix link intervals, matching statistics can be calculated in $\mathcal{O}(n+m)$ time; see [11, 12].

Let $[l, r]$ be the lcp-interval where the mismatch occurred when matching $T^{\mathcal{D}_2}_{i..n_2}$ against the lcp interval tree of $T^{\mathcal{D}_1}$, and let x be the currently processed symbol of $T^{\mathcal{D}_2}_{i..n_2}$ which did not match. Also, let $\alpha\beta$ be the prefix of $T^{\mathcal{D}_2}_{i..n_2}$ which has been matched, where α is the longest common prefix of $\{T^{\mathcal{D}_1}_{SA[l]..n_1}, ..., T^{\mathcal{D}_1}_{SA[r]..n_1}\}$. We already know that $T^{\mathcal{D}_1}_{SA[l-1]..n_1} < T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[r+1]..n_1}$, because otherwise we would be in a different branch of the lcp-interval tree.

If $|\beta| = 0$ (i.e. there is no edge label to a child interval starting with x), we determine the child interval $[l', r']$ with the smallest starting symbol y of its edge label with $y > x$. If there is no such child interval, it follows that $T^{\mathcal{D}_1}_{SA[r]..n_1} < T^{\mathcal{D}_2}_{i..n_2}$, and $T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[r+1]..n_1}$, i.e. $p(i) = r + 1$. Otherwise, we know that $T^{\mathcal{D}_1}_{SA[l'-1]..n_1} < T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[l']..n_1}$, i.e. $p(i) = l'$.

If $|\beta| \geq 1$, β is a prefix of the edge label of a child interval $[l', r']$ of $[l, r]$. If x is smaller than the next symbol of the edge label, $T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[l']..n_1}$, and $T^{\mathcal{D}_1}_{SA[l'-1]..n_1} < T^{\mathcal{D}_2}_{i..n_2}$, i.e. $p(i) = l'$. Otherwise, $T^{\mathcal{D}_1}_{SA[r']..n_1} < T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[r'+1]..n_1}$, i.e. $p(i) = r' + 1$.

In our algorithm we also need information about the length of the longest common prefix between suffixes of $T^{\mathcal{D}_1}$ and suffixes of $T^{\mathcal{D}_2}$. This is needed to determine the intersection of the results intervals of two suffixes. Therefore, for each pair of consecutively processed suffixes, we calculate the length of their longest common prefix (which is in fact the LCP table of the merged suffix arrays). Whenever two consecutively processed suffixes are from the same string, we can use the value of the corresponding LCP table of this string. But if these two suffixes belong to two different strings, we do not know the length of their longest common prefix. Since suffix $T^{\mathcal{D}_2}_{i..n_2}$ is placed between $T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1}$ and $T^{\mathcal{D}_1}_{SA[p(i)]..n_1}$, we also need to calculate $lcp(T^{\mathcal{D}_1}_{SA[p(i)]..n_1}, T^{\mathcal{D}_2}_{i..n_2})$ and $lcp(T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1}, T^{\mathcal{D}_2}_{i..n_2})$.

There are two cases:

(1) $lcp(T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1}, T^{\mathcal{D}_2}_{i..n_2}) = LCP[p(i)]$, $lcp(T^{\mathcal{D}_2}_{i..n_2}, T^{\mathcal{D}_1}_{SA[p(i)]..n_1}) = ms(i)$

(2) $lcp(T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1}, T^{\mathcal{D}_2}_{i..n_2}) = ms(i)$, $lcp(T^{\mathcal{D}_2}_{i..n_2}, T^{\mathcal{D}_1}_{SA[p(i)]..n_1}) = LCP[p(i)]$

Case (1) applies if $l \leq p(i) \leq r$, and case (2) occurs only if $p(i) = r + 1$. We can use the sign bit of $ms(i)$ to indicate which case applies; a positive sign indicates case (1), a negative sign indicates case (2).

We process the suffixes of $T^{\mathcal{D}_1}$ and $T^{\mathcal{D}_2}$ in reverse lexicographical order, i.e. we start with the lexicographically largest suffixes. For each relevant substring ϕ of $T^{\mathcal{D}_2}$ we need to find the lexicographically smallest suffix of $T^{\mathcal{D}_1}$ for which ϕ is a prefix (if there is such a suffix of $T^{\mathcal{D}_1}$). We maintain a set of relevant substrings of $T^{\mathcal{D}_2}$ which are a prefix of the currently processed suffix $T^{\mathcal{D}_1}_{SA[i]..n_1}$. This set will consist of all such relevant substrings assigned to suffixes of $T^{\mathcal{D}_2}$ which are lexicographically larger than $T^{\mathcal{D}_1}_{SA[i]..n_1}$, and the set can be represented as a results interval of $T^{\mathcal{D}_1}_{SA[i]..n_1}$, which can be proved similar to Lemma 2. We will denote this results interval by $[a_{\mathrm{cur}}, b_{\mathrm{cur}}]$.

Lemma 3. *Let $p(i)$ be defined such that $T^{\mathcal{D}_1}_{SA[p(i)-1]..n_1} \le T^{\mathcal{D}_2}_{i..n_2} < T^{\mathcal{D}_1}_{SA[p(i)]..n_1}$. The suffix $T^{\mathcal{D}_2}_{i..n_2}$ $(1 \le i \le n_2)$ can only have common relevant substrings with suffixes at positions $\le p(i)$ in the suffix array of $T^{\mathcal{D}_1}$.*

Proof. Suppose that there exists a suffix $T^{\mathcal{D}_1}_{SA[k]..n_1}$ $(k > p(j))$ to which a relevant substring ϕ was assigned, which was also assigned to $T^{\mathcal{D}_2}_{j..n_2}$. Since ϕ is a common relevant substring, $|\phi| \le lcp(T^{\mathcal{D}_2}_{i..n_2}, T^{\mathcal{D}_1}_{SA[k]..n_1})$. From the definition of $p(i)$ it follows that $lcp(T^{\mathcal{D}_2}_{i..n_2}, T^{\mathcal{D}_1}_{SA[k]..n_1}) \le lcp(T^{\mathcal{D}_2}_{i..n_2}, T^{\mathcal{D}_1}_{SA[p(i)]..n_1})$. This means ϕ can also be assigned to $T^{\mathcal{D}_1}_{SA[p(i)]..n_1}$, which is a contradiction to the condition that each relevant suffix of L_1 was assigned to the lexicographically smallest suffix of $T^{\mathcal{D}_1}$. □

It follows that in addition to the relevant substrings in the results interval $[a_{\text{cur}}, b_{\text{cur}}]$, the suffix $T^{\mathcal{D}_1}_{SA[i]..n_1}$ can only have common relevant substrings with suffixes from $T^{\mathcal{D}_2}$ which are processed between $T^{\mathcal{D}_1}_{SA[i]..n_1}$ and $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$. Therefore, when we process the suffixes from $T^{\mathcal{D}_2}$ which would be placed between $T^{\mathcal{D}_1}_{SA[i]..n_1}$ and $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$ in the merged suffix array, we update $[a_{\text{cur}}, b_{\text{cur}}]$. Before processing $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$, we can then calculate the intersection of $[a_{\text{cur}}, b_{\text{cur}}]$ and the results interval assigned to $T^{\mathcal{D}_1}_{SA[i]..n_1}$.

When processing the suffixes of $T^{\mathcal{D}_2}$ that would be placed between $T^{\mathcal{D}_1}_{SA[i]..n_1}$ and $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$ in the merged suffix array, we also need to calculate the results interval $[a_{\text{prev}}, b_{\text{prev}}]$ representing relevant substrings of $T^{\mathcal{D}_2}$ which are a prefix of $T^{\mathcal{D}_1}_{SA[i-1]..n_1}$. If $LCP[i] \ge a_{\text{cur}}$, $[a_{\text{prev}}, b_{\text{prev}}]$ can be initialized to $[a_{\text{cur}}, \min\{LCP[i], b_{cur}\}]$. Otherwise, we start with an empty results interval.

Now we will show how to handle the case in which one of the minimum frequency thresholds $minf_{\mathcal{D}_1}$ or $minf_{\mathcal{D}_2}$ is zero. This case is also supported by the algorithm of [1]. We can assume that not both $minf_{\mathcal{D}_1} = 0$ and $minf_{\mathcal{D}_2} = 0$, because there must be at least one database $\mathcal{D}_i$ with $minf_i > 0$ (otherwise, there would be infinitely many solutions), and we can select the order in which we merge result tables such that always the result table linked to $\mathcal{D}_i$ is involved.

Without loss of generality, assume $minf_{\mathcal{D}_2} = 0$. This means, we want to keep all relevant substrings from L_1 whose frequency in $\mathcal{D}_2$ do not exceed $maxf_{\mathcal{D}_2}$. Therefore, we only adjust the a values of the tuples of L_1, but leave the b values untouched. Note that since $minf_{\mathcal{D}_2} = 0$, the relevant substrings ϕ with $freq(\phi, \mathcal{D}_2) > maxf_{\mathcal{D}_2}$ are all substrings ϕ of $T^{\mathcal{D}_2}$ which do not belong to any results interval of L_2. This means that if for some tuple in L_1 there are no common relevant substrings with tuples in L_2, we have to remove those relevant substrings which are also substrings of $T^{\mathcal{D}_2}$. This can be done by setting $L_1[i].a$ to $\max\{L_1[i].a, maxlcp + 1\}$, where $maxlcp$ is the maximum length of a common prefix of $T^{\mathcal{D}_1}_{i..n_1}$ with a suffix of $T^{\mathcal{D}_2}$.

The time complexity to intersect the result tables of all databases $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_m$ can be determined as follows: We will successively intersect $\mathcal{D}_2$ with $\mathcal{D}_1$, then

$\mathcal{D}_3$ with $\mathcal{D}_1$, and so on until we have intersected $\mathcal{D}_m$ with $\mathcal{D}_1$. Since intersecting the result table of database $\mathcal{D}_i$ with the result table of the database $\mathcal{D}_1$ takes $\mathcal{O}(n_1 + n_i)$, the overall time complexity is $\mathcal{O}((m-1) \cdot n_1 + \sum_{i=2}^{m} n_i)$. If all values minf_i are positive, we may assume $\mathcal{D}_1$ is the smallest database (i.e. n_1 is smaller than or equal to n_i for all $1 < i \leq m$), and it follows that $(m-1) \cdot n_1 + \sum_{i=2}^{m} n_i \leq 2 \cdot \sum_{i=1}^{m} n_i$, so the overall time complexity to intersect all databases is linear in the total size of all databases.
Otherwise, if some value minf_i is zero, we reorder the databases such that $\mathit{minf}_1 > 0$. Now the database $\mathcal{D}_1$ may be the largest database. However, the time complexity in this case is not worse than the complexity of [1], since $(m-1) \cdot n_1 + \sum_{i=2}^{m} n_i \leq (m-1) \cdot \sum_{i=1}^{m} n_i + \sum_{i=1}^{m} n_i = \mathcal{O}(m \cdot \sum_{i=1}^{m} n_i)$.

4 Implementation

4.1 Program overview

We have compared performance results of the implementation of our algorithm described in Sect. 3 with performance results of the implementation of [1], which is available at [13]. We have made a small modification to the implementation of [13] to be able to evaluate the approximate peak memory consumption. We wrote a function which keeps track of the total amount of memory which is currently allocated. We will refer to the program of [13] as *frequent_linear*.

Our own program is written in C, and we use low-level I/O functions in order to obtain high performance. For example, we use the function mmap to be able to quickly read and write to a file. This functionality is necessary because we try to keep only those values in memory which are currently needed in order to reduce peak memory consumption as much as possible. In fact, peak memory consumption for our program will be at most 25 times the size of the largest database. We refer to our program as *slink_merge*.

In our program, we use the suffix array construction algorithm of [14] instead of one of the linear time algorithms, because it performs better for non-degenerate test cases. This suffix array construction algorithm is also used in [13]. Our implementation and the pseudocode of our algorithm are available at www.uni-ulm.de/in/theo/mitarbeiter/kuegel.html.

The implementation of [13] handles exactly two databases of strings, and uses a fixed value ∞ for maxf_1, and 0 for minf_2. Therefore, we have created a modified version of the code which handles a variable number m of databases, and supports arbitrary, valid minf_i and maxf_i values. In this version, we also included our idea to save memory by using the cumulative sums of the array C'. We will refer to this program as *frequent_linear2*.

4.2 Test data

As in [1], we use the proteins of human and mouse, obtained from Swissprot using the NEWT taxonomy browser ([15]) as one dataset. One database consists of the primary structure of 70747 proteins of humans, the other database consists of 61716 proteins of mice. Each protein represents one entry in the database. The total size of the databases is about 28 MB and 27 MB, respectively.

The second dataset consists of two databases, each containing 10000 random bitstrings of a length between 10000 and 20000. The total size of each database is about 150 MB.

For the first two datasets, we use fixed values $maxf_1 = \infty$ and $minf_2 = 0$ in order to be able to compare our results with the results of the program frequent_linear.

The third dataset consists of four databases, the first two are identical to the second dataset, the third and fourth also contain 10000 random bitstrings of a length between 10000 and 20000.

The fourth dataset consists of twelve databases, each containing 10000 random strings consisting of between 100 and 3000 lowercase letters, i.e. the alphabet size is 26. The total size of each database is about 15 MB.

4.3 Test results

All given time intervals are measured in seconds, and are calculated on a computer with a 2.8 GHz processor and 16 GB of RAM. We used fixed values of $maxf_1 = \infty$ and $minf_2 = 0$ to be able to compare our results to the results of frequent_linear. We picked different parameter combinations for $minf_1$ and $maxf_2$.

Table 1. Runtimes (in seconds) and memory consumption on the first dataset

Test parameters		slink_merge	frequent_linear	frequent_linear2
$\mathbf{minf_1}$	$\mathbf{maxf_2}$			
10	1000	216.88	264.19	296.67
500	1000	195.71	77.24	109.45
3000	57950	205.85	76.94	111.84
10000	57950	194.02	77.72	100.53
30000	57950	206.21	76.31	101.14
Max. memory		703 MB	1307 MB	1307 MB

We can see by looking at Table 1 and Table 2 that the program frequent_linear is the fastest. Our modified version frequent_linear2 is a little bit slower, because it does not have the number of databases hard-coded. Our own program slink_merge is slower by a factor of less than 2, which is caused by the string matching during the intersection of the relevant substrings of the two databases. The slower runtime of frequent_linear and frequent_linear2 in line 1 of Table 1 can be explained by the large amount of output, which is optimized in our program slink_merge. Note that even for two databases, the memory consumption of frequent_linear and frequent_linear2 is worse than the memory consumption of our program.

Table 2. Runtimes (in seconds) and memory consumption on the second dataset

Test parameters		slink_merge	slink_merge	frequent_linear	frequent_linear2
minf$_1$	**maxf**$_2$		reuse of results		
200	1000	1056.72	-	397.86	595.34
1000	9500	961.12	-	397.31	536.33
3000	9500	878.89	575.57	398.66	586.01
5000	9500	830.01	617.39	400.27	577.08
Max. memory		3745 MB	3745 MB	7193 MB	7193 MB

Also, our algorithm gives us the possibility to reuse the calculated relevant substrings for individual databases if another test contains these databases with the same *minf* and *maxf* parameters. With our second dataset, for example, we use three times $minf_2 = 0$, $maxf_2 = 9500$ as parameter for the second database, and $minf_1 = 1000$, 2000, and 3000 for the first database, respectively. The column "slink_merge reuse of results" gives the runtimes if we reuse the results of previous test runs, or "-" if the relevant substrings are calculated for the first time.

Table 3. Runtimes (in seconds) and memory consumption on the third dataset

Test parameters		slink_merge	frequent_linear2
minf$_i$	**maxf**$_i$		
50	1000	2050.73	-
200	1500	1988.32	-
500	1000	1888.98	-
Max. memory		3745 MB	$>$ 12 GB

Table 3 only shows the results of our program. We tried to run frequent_linear2 on this dataset, but it needed more memory than we had available.

Table 4. Runtimes (in seconds) and memory consumption on the fourth dataset

Test parameters		slink_merge	frequent_linear2
minf$_i$	**maxf**$_i$		
2	3	855.91	392.44
10	1000	855.00	390.52
Max. memory		386 MB	10614 MB

With a higher number of databases, the advantage of our algorithm becomes more apparent. Note that on this dataset, frequent_linear2 needs about 27 times more memory than our program.

Although the program frequent_linear performs faster on all our tests, the runtime of our program slink_merge is still competitive. Moreover, for large databases, memory consumption is the bottleneck, and it becomes more important to save memory than to save time. Thus, we conclude that our algorithm improves the possibility to solve large problem instances, especially when more than two databases are used.

References

1. Fischer, J., Heun, V., Kramer, S.: Optimal string mining under frequency constraints. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: PKDD. Volume 4213 of Lecture Notes in Computer Science., Springer (2006) 139–150
2. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J., eds.: ICALP. Volume 2719 of Lecture Notes in Computer Science., Springer (2003) 943–955
3. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In Baeza-Yates, R.A., Chávez, E., Crochemore, M., eds.: CPM. Volume 2676 of Lecture Notes in Computer Science., Springer (2003) 200–210
4. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In Baeza-Yates, R.A., Chávez, E., Crochemore, M., eds.: CPM. Volume 2676 of Lecture Notes in Computer Science., Springer (2003) 186–199
5. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amir, A., Landau, G.M., eds.: CPM. Volume 2089 of Lecture Notes in Computer Science., Springer (2001) 181–192
6. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1) (2004) 53–86
7. Maaß, M.G.: Computing suffix links for suffix trees and arrays. Inf. Process. Lett. **101**(6) (2007) 250–254
8. Fischer, J., Heun, V.: A new succinct representation of rmq-information and improvements in the enhanced suffix array. In Chen, B., Paterson, M., Zhang, G., eds.: ESCAPE. Volume 4614 of Lecture Notes in Computer Science., Springer (2007) 459–470
9. Hui, L.C.K.: Color set size problem with application to string matching. In Apostolico, A., Crochemore, M., Galil, Z., Manber, U., eds.: CPM. Volume 644 of Lecture Notes in Computer Science., Springer (1992) 230–243
10. Jeon, J.E., Park, H., Kim, D.K.: Efficient construction of generalized suffix arrays by merging suffix arrays. Journal of KISS : computer systems and theory **32**(6) (2005) 268–278
11. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica **12**(4/5) (1994) 327–344
12. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
13. Fischer, J.: Linear Frequent String Miner and Emerging Substring Miner (PKDD'06). http://www.bio.ifi.lmu.de/ fischer/frequentLinear.tgz (2007)
14. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1) (2004) 33–50
15. NEWT taxonomy browser. http://www.ebi.ac.uk/newt/ (2007)