Efficient String Mining under Constraints via the Deferred Frequency Index

David Weese[1] and Marcel H. Schulz[2]

[1] Department of Computer Science, Free University of Berlin, Takustr. 9, 14195 Berlin, Germany, weese@inf.fu-berlin.de
[2] Department of Computational Molecular Biology, Max Planck Institute for Molecular Genetics, Ihnestr. 73, 14195 Berlin, Germany, International Max Planck Research School for Computational Biology and Scientific Computing, marcel.schulz@molgen.mpg.de

Abstract. We propose a general approach for frequency based string mining, which has many applications, e.g. in contrast data mining. Our contribution is a novel algorithm based on a deferred data structure. Despite its simplicity, our approach is up to 4 times faster and uses about half the memory compared to the best-known algorithm of Fischer et al. Applications in various string domains, e.g. natural language, DNA or protein sequences, demonstrate the improvement of our algorithm.

1 Introduction

The storage of data in databases alone does not guarantee that all hidden information is readily available. A promising approach for knowledge discovery in databases is to mine frequent patterns, reviewed in [1]. This general paradigm can be applied in many application domains ranging from mining of customer data to optimize marketing strategies [2], and language identification [3], to finding protein fingerprints or binding motifs in biological sequences [4,5]. The latter is important in Computational Biology, where a gene is regulated by proteins, so-called transcription factors, that bind to its promoter sequence. A common approach taken, is to contrast promoter sequences of genes that are believed to be regulated by the same factor, with promoters of unrelated genes to detect the transcription factor's binding motif. The rationale behind it, is to find sequence motifs that are representative (frequent) for one set of sequences and absent (infrequent) in another, often called discriminatory or contrast data mining [6,7]. Here the Frequency of a motif is defined as the number of distinct sequences in a set that contain the motif at least once. In this paper we propose an approach that can efficiently solve any frequency based string mining problem including the problem introduced above.

1.1 Related Work

There have been several approaches in the context of mining substrings with frequency constraints. Raedt and co-workers introduced the first $\mathcal{O}(n^2)$ algorithm,

for databases of size n, in 2002 based on the level-wise Apriori algorithm [8]. This algorithm is not suitable for large databases due to repeated scanning of the whole database. Chan and others [9], as well as Lee et al. [10], suggested indexing the database with a suffix tree. Still, suffix trees can be nicely replaced by linear arrays [11], which was utilized by Fischer and colleagues [7] to devise a more efficient algorithm than that of Raedt et al. and Lee et.al [8,10]. One year later, an improvement to their previous algorithm, and the first optimal $\mathcal{O}(n)$ time algorithm was presented by Fischer and the same co-authors [12]. It was established as the fastest known algorithm for the problem, due to optimal time frequency calculation for substring indices via range minimum queries [13].

1.2 Motivation

Fischer and colleagues achieved the optimality [12] at the expense of complicating the algorithm and adding another $\Theta(n)$ space. In addition, both algorithms of Fischer et al. need to sort the whole suffix array and build additional arrays independent of the constraints of the problem. Hence, an interesting approach is to improve upon the frequency calculation of the algorithms [7,8,9,10], while retaining the problem-specific search space pruning. Indeed, we introduce an approach which combines both. We take advantage of partially constructed suffix trees, to design a problem-oriented algorithm like the one of Raedt et al. and Chan et al. [8,9]. Additionally, we utilize a clever solution for the frequency calculation, which comes as a by-product of the sorting procedure without any additional space overhead. On top of that, our approach is surprisingly simple and we show that it is always faster than the optimal algorithm of Fischer and colleagues over a broad range of pattern domains and for different types of frequency string mining problems.

2 Preliminaries

We consider strings over the finite ordered alphabet Σ and use the term *pattern* synonymously. Σ^* is the set of all possible strings over Σ. A string ϕ is a sequence of letters $\phi[1]\ldots\phi[n]$, where each $\phi[i] \in \Sigma$. $\phi\psi$ is the concatenation of two strings ϕ and ψ. $|\phi|$ denotes the length of the string ϕ and $\phi[i..j]$ is a substring of ϕ from position i to j. If $\psi \in \Sigma^*$ is a substring of ϕ, we write $\psi \preceq \phi$, and $\psi \prec \phi$ if $\psi \neq \phi$ holds in addition. For a non-empty set of strings $\Phi \subseteq \Sigma^*$, $\mathrm{lcp}(\Phi)$ gives the *longest common prefix* of all strings in Φ. If Φ contains exactly 1 string ϕ, $\mathrm{lcp}(\Phi)$ returns ϕ. A database $\mathcal{D} \subseteq \Sigma^*$ has $|\mathcal{D}|$ many strings over Σ.
The *frequency* and the *support* of a string $\phi \in \Sigma^*$ in $\mathcal{D}$ is defined as follows:

$$\mathrm{freq}(\phi, \mathcal{D}) := |\{ d \in \mathcal{D} \mid \phi \preceq d \}|, \ \mathrm{supp}(\phi, \mathcal{D}) := \frac{\mathrm{freq}(\phi, \mathcal{D})}{|\mathcal{D}|} \ . \tag{1}$$

For a set of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ we define the *frequency vector* of ϕ:

$$\mathrm{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m) := \Big(\mathrm{freq}(\phi, \mathcal{D}_1), \ldots, \mathrm{freq}(\phi, \mathcal{D}_m) \Big) \ . \tag{2}$$

For two vectors $u, v \in \mathbb{N}^m$ we define $u \leq v \Leftrightarrow \forall_{i=1,\ldots,m} u_i \leq v_i$.

Example 1. Suppose we are given two databases $\mathcal{D}_1 = \{\text{abab}, \text{babb}\}$ and $\mathcal{D}_2 = \{\text{baab}, \text{aaab}\}$, then $\text{freq}(\text{b}, \mathcal{D}_1, \mathcal{D}_2) = (2, 2)$ and $\text{freq}(\text{ba}, \mathcal{D}_1, \mathcal{D}_2) = (2, 1)$.

2.1 Predicates

A *frequency predicate* on a set of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ is defined as a function that for any frequency vector $v \in \mathbb{N}^m$ evaluates to either *true* or *false* and must be *false* for the null vector. In general, our approach is applicable to the task of finding patterns $\phi \in \Sigma^*$ whose frequencies satisfy a predicate *pred* on a given database set $\mathcal{D}_1, \ldots, \mathcal{D}_m$:

$$Th(pred) = \{\phi \in \Sigma^* \mid pred(\text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) \text{ is } true\} \ . \tag{3}$$

In the following, we will consider two specific examples of frequency string mining problems:

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

This problem has been considered in a series of research papers [7,8,10]. The next problem considers discriminatory strings for two databases $\mathcal{D}_1, \mathcal{D}_2 \in \Sigma^*$. $\mathcal{D}_1$ is usually called positive (foreground) set, where $\mathcal{D}_2$ is the negative (background) set. As a measure of difference the growth-rate from $\mathcal{D}_2$ to $\mathcal{D}_1$ for a string ϕ is defined as

$$\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) := \begin{cases} \frac{\text{supp}(\phi, \mathcal{D}_1)}{\text{supp}(\phi, \mathcal{D}_2)} & , \text{ if supp}(\phi, \mathcal{D}_2) \neq 0 \\ \infty & , \text{ otherwise} \end{cases} \ . \tag{4}$$

Problem 2. Given a support condition ρ_s ($\frac{1}{|\mathcal{D}_1|} \leq \rho_s \leq 1$), and a minimum growth rate $\rho_g > 1$, the *Emerging Substring Mining Problem* is to detect all strings $\phi \in \Sigma^*$ s.t. $\text{supp}(\phi, \mathcal{D}_1) \geq \rho_s$ and $\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \geq \rho_g$ [9].

The minimum support rate ρ_s limits the solution space to representative strings of database $\mathcal{D}_1$, where ρ_g is the discrimination threshold. Patterns which satisfy the conditions of Problem 2 are called *Emerging Substrings*. If the growth rate of the pattern is infinite it is called *Jumping Emerging Substring*, because it is a major discriminator between the databases under investigation.

Example 2. We now apply this problem to databases $\mathcal{D}_1$ and $\mathcal{D}_2$ from Example 1 with $\rho_s = 1$ and $\rho_g = 2$ and want to find all strings $\phi \in \Sigma^*$ with $\text{supp}(\phi, \mathcal{D}_1) \geq 1$ and $\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \geq 2$. The corresponding frequency predicate *pred* for the

Emerging Substring Mining Problem is a function that maps the frequency vector $(d_1, d_2) = \text{freq}(\phi, \mathcal{D}_1, \mathcal{D}_2)$ of a string $\phi \in \Sigma^*$ to a truth value as follows:

$$\begin{aligned} pred(d_1, d_2) &:= (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|) \\ &= (d_1 \geq 2) \wedge (d_1 \geq 2d_2) \ . \end{aligned} \quad (5)$$

The set of patterns whose frequencies satisfy *pred* is $Th(pred) = \{\text{bab}, \text{ba}\}$. *b* for example is not an *Emerging Substring*, because $\text{supp}(\text{b}, \mathcal{D}_1) = 1$ but $\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\text{b}) = 1 < \rho_g$.

2.2 Monotonicity

We will now introduce the monotonic property of frequency predicates that we use later to restrict the search space of our algorithm. Examples 3 and 4 show that the frequency predicates of Problem 1 and 2 contain a monotonic subpredicate.

Definition 1. *If for a frequency predicate pred* $: \mathbb{N}^m \to \{\text{true,false}\}$ *holds that:*

$$\forall_{u,v \in \mathbb{N}^m, u \leq v} \Big(pred(u) \Rightarrow pred(v) \Big) \ , \quad (6)$$

then pred is called monotonic*.*

Proposition 1. *For a monotonic*[3] *frequency predicate pred on databases* $\mathcal{D}_1, \ldots, \mathcal{D}_m \subseteq \Sigma^*$ *it holds that:*

$$\forall_{\phi,\psi \in \Sigma^*, \phi \preceq \psi} : \Big(pred(\text{freq}(\psi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) \Rightarrow pred(\text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) \Big) \ . \quad (7)$$

Proof. Each occurrence of ψ is also an occurrence of ϕ. Thus, $\text{freq}(\psi, \mathcal{D}_1, \ldots, \mathcal{D}_m) \leq \text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)$ holds. □

Example 3. As seen in Example 2 the frequency predicate for the *Emerging Substring Mining Problem* is:

$$pred(d_1, d_2) = (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|) \ . \quad (8)$$

Generally, *pred* is not monotonic as shown in Example 2. Recall that ba is *emerging* although b is not. However, if we consider only the left inequality:

$$pred_{\text{m}}(d_1, d_2) := (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \ , \quad (9)$$

$pred_{\text{m}}$ is monotonic, as for all $u, v \in \mathbb{N}^2, u \leq v$ holds $u_1 \geq \rho_s \cdot |\mathcal{D}_1| \Rightarrow v_1 \geq \rho_s \cdot |\mathcal{D}_1|$. Obviously it holds that $pred \Rightarrow pred_{\text{m}}$.

Example 4. For the *Frequent Pattern Mining Problem* with

$$pred(d_1, d_2) = (min_1 \leq d_1 \leq max_1) \wedge (min_2 \leq d_2 \leq max_2) \quad (10)$$

analogously

$$pred_{\text{m}}(d_1, d_2) := (min_1 \leq d_1) \wedge (min_2 \leq d_2) \quad (11)$$

is monotonic and $pred \Rightarrow pred_{\text{m}}$ holds.

[3] Note that what we call *monotonic* is called *anti-monotonic* in [8,7], as they consider *pattern* predicates instead of *frequency* predicates.

2.3 Suffix Trees and Suffix Arrays

In this section we will define the *generalized suffix tree* of a database $\mathcal{D} = \{\phi_1, \ldots, \phi_d\}$. To distinguish the suffixes of strings in $\mathcal{D}$, we will use string markers $\$_j$ at the end of each string ϕ_j. String markers are artificial symbols $\$_j$ that must not occur in any string of $\mathcal{D}$ and we implicitly assume $\$_j \in \Sigma$. We define the artificial order $\$_1 < \$_2 < \ldots < \$_d < c$ for any $c \in \Sigma \setminus \{\$_1, \ldots, \$_d\}$.
A generalized suffix tree for a database $\mathcal{D}$ over Σ is a rooted directed tree with edge labels from Σ^*, s.t. every concatenation of symbols from the *root* to a leaf node yields a suffix of $\phi_j\$_j$ for a string $\phi_j \in \mathcal{D}$. Each internal node has at least two children, and no two edges out of the same node are allowed to have edge-labels starting with the same character. By this definition, each node can be mapped one-to-one to the concatenation of symbols from the *root* to itself. The node of a concatenation string α will be denoted by $\bar{\alpha}$.
We will also need the concept of a *generalized suffix array* for a database $\mathcal{D}$ over Σ. Therefore, all strings $\phi_j \in \mathcal{D}$ are concatenated by their string markers $\$_j$ to form conceptually one string $\phi_1\$_1\phi_2\$_2 \ldots \phi_d\$_d$, the *union string* of $\mathcal{D}$. The generalized suffix array stores the starting positions of all lexicographically ordered suffixes of the union string [14].
Generalized suffix trees, generalized suffix arrays, and the union string of a set of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ are defined analogously using string markers $\$_1, \$_2, \ldots, \$_{\tilde{d}}$ with $\tilde{d} = \sum_{i=1}^{m} |\mathcal{D}_i|$. Figure 1 shows the generalized suffix tree of the Example 1 database $\mathcal{D}_1$.

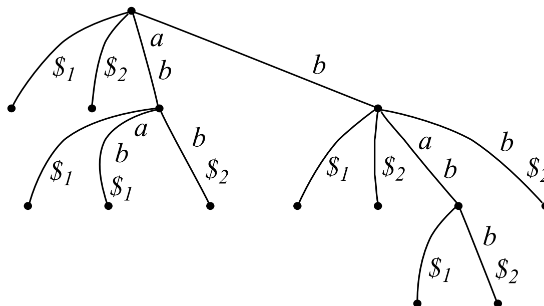

Fig. 1. The resulting generalized suffix tree for database $\mathcal{D}_1 = \{\text{abab}, \text{babb}\}$. As mentioned in the text, to the end of every string from $\mathcal{D}_1$ a unique string marker $\$_1$ and $\$_2$ for the first and second string is appended, respectively.

3 The Algorithm

This section introduces the *Deferred Frequency Index* (DFI) which is fundamentally based on a generalized lazy suffix tree [15]. The DFI algorithm constructs

only the upper part of a generalized suffix tree in a top-down manner. To understand the DFI algorithm we will at first explain the idea of the *write-only, top-down* construction algorithm, abbreviated as *wotd*-algorithm.

3.1 The *wotd*-algorithm

A lazy suffix tree is a suffix tree whose nodes are created on demand, i.e. when they are visited the first time. For instances where only upper parts of the suffix tree are required, using a lazy suffix tree can be more efficient than constructing the whole suffix tree. Giegerich et al. introduced the first lazy suffix tree data structure [16] that utilizes the *wotd*-algorithm [15,16] for the on-demand node expansion.
The *wotd*-algorithm is a suffix tree construction algorithm that expands a rooted directed tree starting with a tree consisting of only the root node step-by-step to at most the entire suffix tree. We describe a variant of the *wotd*-algorithm to create a generalized suffix tree. Suppose a given non-empty database $\mathcal{D} = \{\phi_1, \ldots, \phi_d\}$ over Σ and a rooted directed tree T. Each node in T is either in *expanded* or *unexpanded* state. In the beginning, T contains only the *unexpanded* root node. Let R be a function that returns for any string $\alpha \in \Sigma^*$ the set of suffixes with string markers of strings in $\mathcal{D}$ so that:

$$R(\alpha) := \{\alpha\beta\$_j \mid \alpha\beta \text{ is a suffix of } \phi_j\} \ . \tag{12}$$

$R(\alpha)$ comprises all suffixes of strings in $\mathcal{D}$ that begin with the string α. In relation to nodes $\bar{\alpha}$ of the generalized suffix tree of $\mathcal{D}$, $R(\alpha)$ contains the concatenated edge labels of paths between the root node and leaf nodes below $\bar{\alpha}$. When an *unexpanded* node $\bar{\alpha}$ of the lazy suffix tree has to be expanded, $R(\alpha)$ can be used to determine the subtree below $\bar{\alpha}$. The node expansion of $\bar{\alpha}$ works as follows: $R(\alpha)$ is divided into groups $R(\alpha c)$ of the same character c that follows α. Let $\alpha c\beta$ be the longest common prefix of $R(\alpha c)$. Out of $\bar{\alpha}$ an edge will be created, labeled with $c\beta$ leading to a node $\overline{\alpha c\beta}$. If $R(\alpha c)$ is a singleton group, the leaf node $\overline{\alpha c\beta}$ is marked as *expanded*. Otherwise, it is a branching node and marked as *unexpanded*. After all groups were processed, $\bar{\alpha}$ will be marked as *expanded*.
Algorithm 1 shows how the whole generalized suffix tree can be constructed recursively starting with $expandNode(root)$ on a tree T that contains only the *unexpanded* root node. It can easily be modified to create only an upper part of the suffix tree.

3.2 Monotonic Hull

We now show how to connect arbitrary frequency predicates with the *wotd*-algorithm. To do so, we give a theoretical description of the minimal set of nodes that need to expanded.

Definition 2. *Given frequency predicates pred and* $pred_{\text{hull}}$. $pred_{\text{hull}}$ *is called* a monotonic hull *of pred, if it is monotonic and pred* $\Rightarrow$ $pred_{\text{hull}}$ *holds.*

Algorithm 1: expandNode($\bar{\alpha}$)

```
   Input    : unexpanded node ᾱ
1  Divide R(α) into subsets R(αc) of suffixes starting with character c after α
2  foreach c ∈ Σ and R(αc) ≠ ∅ do
3  |  αcβ ← lcp(R(αc))
4  |  if |R(αc)| = 1 then                                        // leaf node
5  |  |  Create the expanded node αcβ (overlined) below ᾱ
6  |  else                                                  // branching node
7  |  |  Create the unexpanded node αcβ (overlined) below ᾱ
8  |  |  expandNode(αcβ (overlined))
9  Mark ᾱ as expanded
```

The most trivial monotonic hull of each frequency predicate *pred* is $pred_{\text{hull}} \equiv$ *true*. If we take a look at the generalized suffix tree T of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$, we make the following observations:

Proposition 2. *Let pred be an arbitrary frequency predicate and* $pred_{\text{m}}$ *an arbitrary* monotonic *frequency predicate on* $\mathcal{D}_1, \ldots, \mathcal{D}_m$*. For all pairs of fathers and sons* $\bar{\alpha}$ *and* $\overline{\alpha\beta}$ *in* T *it holds that:*

1. *If pred*$(\text{freq}(\alpha\beta, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is* true *then for each string* χ *with* $\alpha \prec \chi \preceq \alpha\beta$ *pred*$(\text{freq}(\chi, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is* true*.*
2. *If* $pred_{\text{m}}(\text{freq}(\alpha\beta, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is* true *then* $pred_{\text{m}}(\text{freq}(\alpha, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is* true*.*

Proof. *The frequency vectors of* $\alpha\beta$ *and* χ *with* $\alpha \prec \chi \preceq \alpha\beta$ *must be equal. If not, there would be a branching node between* $\bar{\alpha}$ *and* $\overline{\alpha\beta}$ *which contradicts the assumption* $\bar{\alpha}$ *would be the father of* $\overline{\alpha\beta}$*. Hence 1. holds. 2. is a direct consequence of Proposition 1 as* α *is a substring of* $\alpha\beta$*.* □

In consequence of Proposition 2, it satisfies to evaluate *pred* only on the nodes of T to compute the set *Th*(*pred*). For every monotonic hull $pred_{\text{hull}}$ of *pred* the set of nodes, whose frequencies satisfies $pred_{\text{hull}}$, is a directed connected subgraph of T, which if non-empty, contains the root node. Outside of this subgraph there is no node fulfilling *pred*. Our algorithm exclusively traverses this subgraph to compute the set *Th*(*pred*). Hence, we are interested in keeping the subgraph as small as possible, leading to the next definition:

Definition 3. $pred_{\text{hull}}$ *is called* the optimal monotonic hull *of pred, if it is a monotonic hull of pred, and for each monotonic hull* $pred'_{\text{hull}}$ *of pred, it holds that* $pred_{\text{hull}} \Rightarrow pred'_{\text{hull}}$*.*

In other words, if $pred_{\text{hull}}$ is optimal, the corresponding subgraph is minimal.

Algorithm 2: expandNodeWithConstraint($\bar{\alpha}$, *pred*, $pred_{\text{hull}}$)

```
   Input    : unexpanded node ᾱ
1  Freq = divideAndCountFreq(ᾱ)
2  foreach c ∈ Σ and R(αc) ≠ ∅ do
3      αcβ ← lcp(R(αc))
4      if pred(Freq[c]) then
5          Output strings χ with αc ⪯ χ ⪯ αcβ ⁴
6      if pred_hull(Freq[c]) then
7          if |R(αc)| = 1 then                          // leaf node
8              Create the expanded node \overline{αcβ} below ᾱ
9          else                                          // branching node
10             Create the unexpanded node \overline{αcβ} below ᾱ
11             expandNodeWithConstraint(\overline{αcβ}, pred, pred_hull)
12 Mark ᾱ as expanded
```

3.3 The Deferred Frequency Index

In the following we will show how the DFI can be built for any given frequency predicate *pred* and a monotonic hull $pred_{\text{hull}}$.

Algorithm 2 starts with *expandNodeWithConstraint*(*root*, *pred*, $pred_{\text{hull}}$) on a tree T with α as the *unexpanded* root node. First, *divideAndCountFreq* is called for the current node $\bar{\alpha}$ in line 1. Identically to algorithm 1, the set $R(\alpha)$ is divided into groups $R(\alpha c)$ of suffixes starting with the same character $c \in \Sigma$ after their prefix α. In addition, an array *Freq*, that stores in *Freq*[c] the frequency vector $\text{freq}(\alpha c, \mathcal{D}_1, \ldots, \mathcal{D}_m)$, is returned. In the next section we explain the implementation details of function *divideAndCountFreq*. The longest common prefix of every non-empty group $R(\alpha c)$ is determined and assigned to $\alpha c\beta$ in line 3. If the predicate *pred* evaluated with the frequency vector *Freq*[c] is *true*, by Proposition 2 all strings χ with $\alpha c \preceq \chi \preceq \alpha c\beta$ belong to $Th(pred)$ and are output[4]. In line 6 $pred_{\text{hull}}$ is evaluated on *Freq*[c]. Only if *true* is returned, the subtree below the node $\overline{\alpha c\beta}$ may contain a node $\bar{\gamma}$ with $\gamma \in Th(pred)$ and will be expanded recursively. If *false* is returned, the node $\overline{\alpha c\beta}$ is not created, as no further subtree expansion is necessary.

Algorithm 2 is correct and outputs the set $Th(pred)$ because of the following: For each database substring ϕ there is a path from the *root* ending in a node or on an edge to a node. This node has the same frequency vector as ϕ and will be visited if it satisfies $pred_{\text{hull}}$ and output iff it satisfies *pred*. As $pred_{\text{hull}}$ is a monotonic hull, no node that satisfies *pred* is left out by the algorithm.

For the *Emerging Substring Mining Problem* and the *Frequent Pattern Mining Problem* one only needs to replace *pred* and $pred_{\text{hull}}$ in Algorithm 2 with the predicates deduced in Examples 3 and 4, respectively. The monotonic hulls for these problems are also optimal as Proposition 3 and 4 prove (see Appendix).

[4] In fact, we omit to output strings χ with a trailing $\$_j$ of a string.

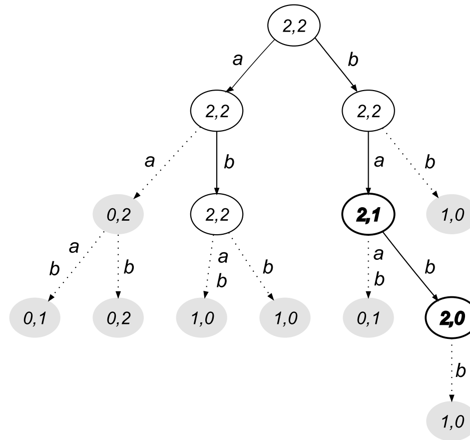

Fig. 2. The generalized suffix tree of our example databases $\mathcal{D}_1$ and $\mathcal{D}_2$. For clarity, the artificial string markers $\$_j$ are omitted. Considering the problem of Example 2, the DFI would construct only the white nodes. Grey nodes are not built. The bold nodes $\overline{\text{ba}}, \overline{\text{bab}}$ represent the *Emerging Substrings*. Each node holds the frequency vector of its corresponding substring (compare the frequencies of $\bar{\text{b}}$ and $\overline{\text{ba}}$ with Example 1).

Figure 2 shows the DFI for the *Emerging Substring Mining Problem* considered in Example 2.

3.4 Algorithm Details

In this section we explain the function *divideAndCountFreq* in detail (Algorithm 3). The sets $R(\alpha)$ are in fact not stored as sets of strings, but as intervals of a generalized suffix array SA. SA is initialized with numbers from 1 to $|S|$, where S is the union string of $\mathcal{D}_1, \ldots, \mathcal{D}_m$. We need a function *getSeqNo* that returns, for each character position i, the sequence number j if $\$_j$ is the next string marker at or to the right of position i in S. We also need a function *getDatabaseNo* that returns, for each sequence number j, the corresponding database number k if $\$_j$ is a string marker of a string in $\mathcal{D}_k$.
When *divideAndCountFreq*$(\bar{\alpha}, \mathit{pred}, \mathit{pred}_{\text{hull}})$ is called, $\text{SA}[l..r]$ contains the start positions of suffixes of S starting with α. Each start position corresponds to a suffix in $R(\alpha)$. Because $\bar{\alpha}$ is *unexpanded*, the suffixes in $\text{SA}[l..r]$ have been sorted with counting sort [17] up to the first $|\alpha|$ characters by previous function calls. Because counting sort is stable, the positions in $\text{SA}[l..r]$ are in increasing order. Therefore, the corresponding sequence numbers of the positions are stored in contiguous blocks. Counting sort divides $R(\alpha)$ into buckets $R(\alpha c)$ for each character $c \in \Sigma$ (lines 3–4). The frequency of each bucket can simply be counted by counting blocks of equal sequence numbers (line 5).

We keep track of three arrays in the size of the alphabet, i.e. $|\Sigma|$, namely *Bucket*, *Freq* and *Last*. *Bucket* is the original array from counting sort, and $Bucket[c]$ counts the occurrences of αc. *Freq* stores frequency vectors, and $Freq[c][k]$ determines how often αc occurred in distinct sequences of $\mathcal{D}_k$. *Last* is used to construct *Freq* (lines 5–8).

Algorithm 3: divideAndCountFreq($\bar{\alpha}$)

```
   Input    : unexpanded node ᾱ
   Output   : freq(αc, D_1, ..., D_m) for each c ∈ Σ
   Require  : SA[l..r] stores all suffixes starting with α, suffixes with equal
              sequence numbers are contiguous in this interval
   Ensure   : suffixes with equal sequence numbers are contiguous in output
              intervals SA[Bucket[c]..Bucket[c + 1] − 1]
 1 Init Bucket, Freq, Last with 0s
   // start to sort the first char after prefix α
 2 for i ← l to r do
 3     c ← S[SA[i] + |α|]
 4     Bucket[c] ← Bucket[c] + 1
 5     if Last[c] ≠ getSeqNo(SA[i]) then
 6         Last[c] ← getSeqNo(SA[i])
 7         k ← getDatabaseNo(getSeqNo(SA[i]))
 8         Freq[c][k] ← Freq[c][k] + 1
 9 Sort suffixes in SA[l..r] stable using Bucket (Counting sort [17] lines 6–11)
   // now Freq[c] contains the frequency vector freq(αc, D_1, ..., D_m)
10 return Freq
```

4 Experiments

To evaluate the performance of our algorithm, we conducted a number of experiments with databases of different characteristics. We used a previously compiled set of human and drosophila core promoters [18], the UniProt [19] proteome sets of human and mouse, release 12.6, verses of the King James Bible and the Bible in Basic English, and posts of 5 computer newsgroups from the UCI Machine Learning Repository divided into Windows and non-Windows groups. The alphabet size $|\Sigma|$ or the sizes of these databases are shown in Table 1.
An experiment consists of two databases $\mathcal{D}_1, \mathcal{D}_2$. These were searched for *Emerging Substrings* and for the solution of the *Frequent Pattern Mining Problem* with different values of ρ_s and varying min_1, respectively. As ρ_g and max_2 had no measurable influence on the tested algorithms only the results for $\rho_g = 5$ and $max_2 = \frac{|\mathcal{D}_2|}{2}$ are shown. The results for other values look similar [7]. We made no other restrictions, i.e. $max_1 = \infty$, $min_2 = 0$.
The theoretically optimal algorithm of Fischer et al. has turned out to be the

hitherto fastest algorithm in practice for the two introduced string mining problems [12]. Hence, we used the implementation of Fischer's algorithm as reference in our experiments. Both programs were written in C++ and compiled using the same compiler options. They run under Linux on an Intel Xeon 3.2 GHz with 2 GB of RAM. To reduce influences from the operating system and secondary storage units, the output was redirected to the null-device, and each experiment was repeated 5 times. We measured the running time and space consumption of both algorithms using the GNU tools `time` and `memusage`.

Table 1. Characteristics and short names for the different databases we used.

name	description	$\vert\Sigma\vert$	size (mb)	#seqs	source
HProm	human promoters	5	23	15011	Fitzgerald et al. [18]
DProm	drosophila promoters	5	16.7	10914	
HProt	human proteome	24	17.6	40827	Uniprot [19]
MProt	mouse proteome	24	16	35344	
KJB	king james bible	128	4.1	31102	Chinese and English
BBE	bible in basic english	128	4.2	31102	Bible Online[5]
WN	windows newsgroup	128	3.9	2000	Machine Learning
CN	computer newsgroup	128	3.4	3000	Repository [20]

Table 2. Experimental setups and space consumption in MB for various minimum support values.

experiment name	$\mathcal{D}_1, \mathcal{D}_2$	Fischer $\rho_s \in [0,1]$	DFI $\rho_s = .001$	DFI $\rho_s = .01$	DFI $\rho_s = .1$
DNA Promoters	HProm , DProm	919.57	531.19	475.82	468.15
Proteomes	HProt , MProt	779.01	365.49	330.95	327.74
English Bibles	KJB , BBE	193.83	109.46	96.96	95.38
Newsgroups	WN , CN	167	104.43	82.25	79.76

Figure 3 shows, that our approach is in all cases faster than the approach of Fischer et al. even for small values ρ_s or min_1 when the whole suffix tree needs to be constructed. As an example, for $\rho_s = 0.2$ the DFI is with 16 seconds on the Proteome datasets roughly 4 times faster than the algorithm of Fischer et al. Considering reasonable[6] values of $\rho_s < 0.2$ and $min_1 < 0.2 \cdot |\mathcal{D}_1|$ our algorithm

[5] `http://www.o-bible.com/`

[6] Dong and Li [21] report that a minimum support of 1%–20% for finding *Emerging Patterns* could contribute significantly to knowledge discovery.

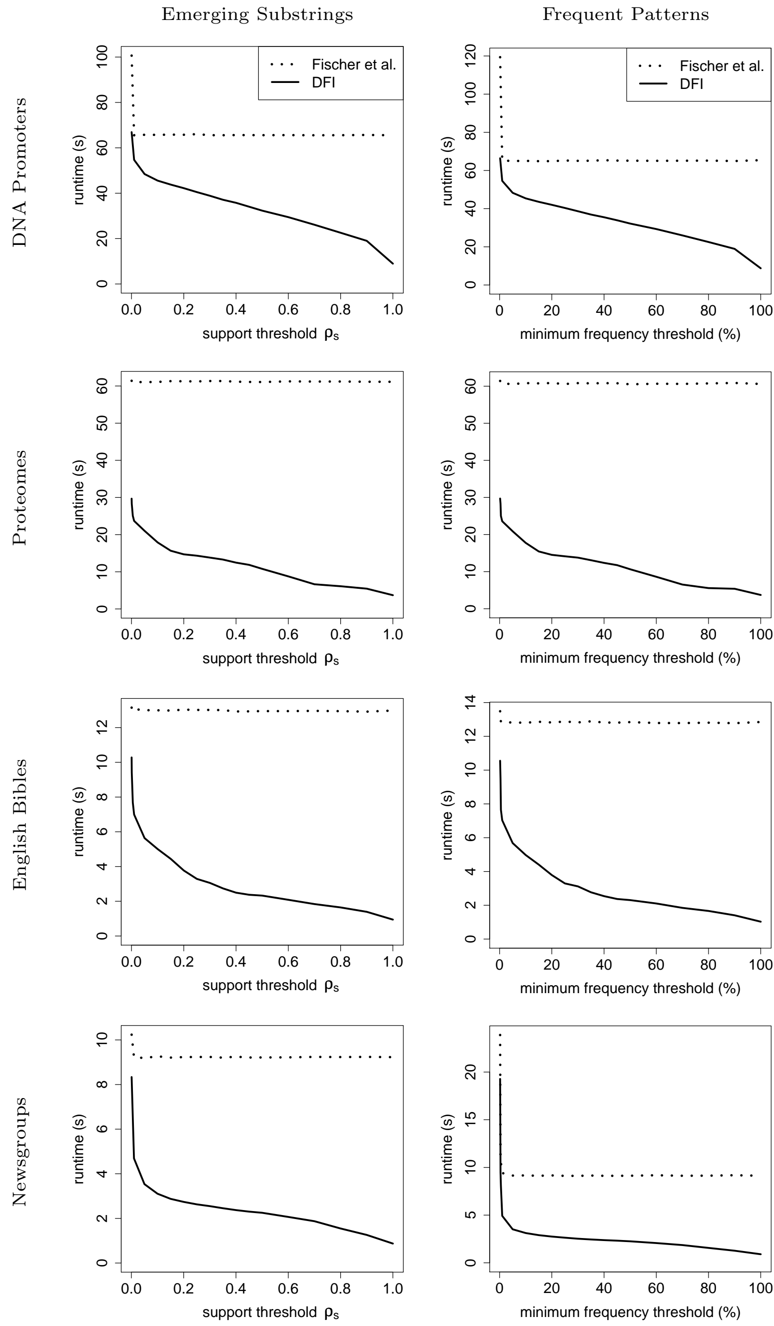

Fig. 3. Runtime comparison of the algorithm of [12] (dotted) and our DFI implementation (solid) for the *Emerging Substring Problem* (left) and the *Frequent Pattern Mining Problem* (right). Experiment details are listed in Table 2.

is 1.5–4 times faster in practice. This is surprising, because our algorithm has a worst case running time of $\mathcal{O}(n^2)$ [15], in contrast to the $\mathcal{O}(n)$ algorithm of Fischer and colleagues. Both algorithms have an $\mathcal{O}(n)$ memory consumption, but ours needs only about half of the memory, see Table 2. Fischer's algorithm has an almost constant running time and space consumption as it does not take advantage of the monotonic pruning of the suffix tree like our deferred approach does. The runtime peaks for small values of ρ_s or min_1 are due to the high amount of strings in the solution space that were output.

5 Discussion and Future Work

We presented a new approach to constraint-based string mining that outperforms the best-known algorithm by Fischer et al. [12] in runtime and space consumption as the experiments show. The better running time can be attributed to various factors. Most importantly, the optimal monotonic hull of a frequency predicate, is incorporated to prune the search space to a minimum, resulting in the deferred frequency index. Moreover, the frequency information is extracted as a constant time by-product during the suffix tree construction. Our algorithm inherits the good cache locality from the *wotd*-algorithm [15] and in addition uses less memory than Fischer's algorithm.

Depending on the problem at hand, the implementation of our algorithm could be improved. If the DFI should only be used to output the result of $Th(pred)$, the memory consumption of the algorithm could be further reduced. As each node is visited at most once, at any time only nodes of the suffix tree on the path from the *root* to the current node need to be stored. A small alphabet (e.g. DNA) leads to a dense suffix tree with many branching nodes at the top, as observed by Kurtz [22]. In that case, a runtime improvement could be expected by replacing the top of the suffix tree with a q-gram index.

We believe that our constraint oriented algorithm will be useful for the data mining community. Considering constraints during the mining process will play an important role in further algorithmic development, because reducing the solution space of any mining approach to a compact but representative set is one of the open challenges, as mentioned by Han et al. [1]. In the spirit of this observation, the simplicity of our approach opens various avenues of further research. One is to combine *Jumping Emerging Substrings* to build powerful classifiers as was done for *Jumping Emerging Patterns* [23]. This could be achieved by restricting to minimal and highly significant *Jumping Emerging Substrings*. In a recent work [24], a formulation of a similarity pattern predicate composed of an anti-monotonic part was introduced. Our idea can easily be applied, to improve on their approach. Another direction is to extend the algorithm presented here to deal with gap constraints like was done in the work of Ji and colleagues [25]. Our algorithm is freely available at `http://www.seqan.de/projects/dfi.html` and part of the C++ Sequence Analysis Library SeqAn [26].

Acknowledgements

We thank Knut Reinert who brought the topic to our attention, Clemens Gröpl for helpful discussions, Markus Bauer, Ole Schulz-Trieglaff, and Killian McCutcheon for proofreading.

References

1. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Min. Knowl. Discov. **15**(1) (2007) 55–86
2. Berry, M.J., Linoff, G.S. In: Data Mining Techniques: For Marketing, Sales, and Customer Support. 1 edn. John Wiley & Sons (1997) 51–62
3. Muthusamy, Y.K., Barnard, E., Cole, R.A.: Reviewing automatic language identification. IEEE Sig. Proc. Mag. **11**(4) (1994) 33–41
4. Zhang, M.Q.: Computational analyses of eukaryotic promoters. BMC Bioinformatics **8(Supp 6)** (2007) S3
5. Birzele, F., Kramer, S.: A new representation for protein secondary structure prediction based on frequent patterns. Bioinformatics **22**(21) (2006) 2628–2634
6. Redhead, E., Bailey, T.L.: Discriminative motif discovery in dna and protein sequences using the DEME algorithm. BMC Bioinformatics **8** (2007) 385
7. Fischer, J., Heun, V., Kramer, S.: Fast frequent string mining using suffix arrays. In: IEEE ICDM '05, IEEE Computer Society (2005) 609–612
8. Raedt, L.D., Jaeger, M., Lee, S.D., Mannila, H.: A theory of inductive query answering. In: IEEE ICDM '02, IEEE Computer Society (2002) 123–130
9. Chan, S., Kao, B., Yip, C.L., Tang, M.: Mining emerging substrings. In: DASFAA '03, IEEE Computer Society (2003) 119–126
10. Lee, S.D., Raedt, L.D.: An efficient algorithm for mining string databases under constraints. In: KDID '04, Springer (2004) 108–129
11. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2** (2004) 53–86
12. Fischer, J., Heun, V., Kramer, S.: Optimal string mining under frequency constraints. In: PKDD '06, Springer (2006) 139–150
13. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: CPM '06, Springer (2006) 36–48
14. Manber, U., Myers, E.: Suffix arrays: a new method for on-line string searches. In: SODA '90, SIAM (1990) 319–327
15. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Software Pract. Exper. **33**(11) (2003) 1035–1049
16. Giegerich, R., Kurtz, S.: A comparison of imperative and purely functional suffix tree constructions. Sci. Comput. Program. **25** (1995) 187–218
17. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: 8.2: Counting sort. In: Introduction to Algorithms. 2 edn. MIT Press and McGraw-Hill (2001) 168–170
18. Fitzgerald, P.C., Sturgill, D., Shyakhtenko, A., Oliver, B., Vinson, C.: Comparative genomics of drosophila and human core promoters. Genome Biol. **7** (2006) R53
19. The UniProt Consortium: The Universal Protein Resource (UniProt). Nucl. Acids Res. **36**(suppl_1) (2008) D190–195 `ftp://ftp.ebi.ac.uk/pub/databases/integr8/uniprot/proteomes`.
20. Asuncion, A., Newman, D.: UCI machine learning repository. `http://www.ics.uci.edu/~mlearn/MLRepository.html` (2007)

21. Dong, G., Li, J.: Efficient mining of emerging patterns: discovering trends and differences. In: KDD '99, ACM (1999) 43–52
22. Kurtz, S.: Reducing the space requirement of suffix trees. Software Pract. Exper. **29**(13) (1999) 1149–1171
23. Li, J., Dong, G., Ramamohanarao, K.: Making use of the most expressive jumping emerging patterns for classification. In: PADKK '00, Springer (2000) 220–232
24. Mitasiunaite, I., Boulicaut, J.F.: Looking for monotonicity properties of a similarity constraint on sequences. In: SAC '06, ACM (2006) 546–552
25. Ji, X., Bailey, J., Dong, G.: Mining minimal distinguishing subsequence patterns with gap constraints. Knowl. Inf. Syst. **11**(3) (2007) 259–286
26. Döring, A., Weese, D., Rausch, T., Reinert, K.: SeqAn an efficient, generic C++ library for sequence analysis. BMC Bioinformatics **9** (2008) 11

Appendix

Proposition 3. *Let $\mathcal{D}_1, \mathcal{D}_2$ be two databases, $\rho_s, \rho_g \in \mathbb{R}$, and $pred : \mathbb{N}^2 \rightarrow \{true, false\}$ be defined as:*

$$pred(d_1, d_2) = (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|) \ . \tag{13}$$

The monotonic hull $pred_{\text{hull}}$ of pred with:

$$pred_{\text{hull}}(d_1, d_2) := (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \tag{14}$$

is optimal.

Proof. We assume $pred_{\text{hull}}$ is a non-optimal monotonic hull of *pred*. Then there exists a monotonic hull $pred'_{\text{hull}}$ of *pred* with $pred_{\text{hull}} \not\Rightarrow pred'_{\text{hull}}$. Thus, $d \in \mathbb{N}^2$ exist so that $pred_{\text{hull}}(d_1, d_2)$ is *true* and $pred'_{\text{hull}}(d_1, d_2)$ is *false*. By the contraposition of the monotonicity criterion, $pred'_{\text{hull}}(d_1, 0)$ also is *false*. It holds that $pred(d_1, 0) = pred_{\text{hull}}(d_1, d_2) = true$ and $pred \not\Rightarrow pred'_{\text{hull}}$. This is a contradiction to $pred'_{\text{hull}}$ being a monotonic hull of *pred*. Hence the proposition holds. □

Proposition 4. *Let $min_1, max_1, min_2, max_2 \in \mathbb{N}$, $(min_1, min_2) \leq (max_1, max_2)$, and $pred : \mathbb{N}^2 \rightarrow \{true, false\}$ be defined as:*

$$pred(d_1, d_2) = (min_1 \leq d_1 \leq max_1) \wedge (min_2 \leq d_2 \leq max_2) \tag{15}$$

The monotonic hull $pred_{\text{hull}}$ of pred with:

$$pred_{\text{hull}}(d_1, d_2) := (min_1 \leq d_1) \wedge (min_2 \leq d_2) \tag{16}$$

is optimal.

Proof. Analogously holds for a $pred'_{\text{hull}}$ and $d \in \mathbb{N}^2$: $pred_{\text{hull}}(d)$ is *true* and $pred'_{\text{hull}}(d)$ is *false*. Thus it holds that $(min_1, min_2) \leq d$ and $pred'_{\text{hull}}(min_1, min_2)$ also is *false*. It holds that $pred(min_1, min_2) = true$ and $pred \not\Rightarrow pred'_{\text{hull}}$. This is a contradiction to $pred'_{\text{hull}}$ being a monotonic hull of *pred*. Hence the proposition holds. □