# An Efficient Algorithm for Mining String Databases under Constraints (Extended Abstract) *

Sau Dan Lee  Luc De Raedt

Institute for Computer Science, University of Freiburg, Germany
{danlee,deraedt}@informatik.uni-freiburg.de

**Abstract.** We study the problem of mining substring patterns from string databases. Patterns are selected using a conjunction of monotonic and anti-monotonic predicates. Based on the earlier introduced version space tree data structure, a novel algorithm for discovering substring patterns is introduced. It has the nice property of requiring only one database scan, which makes it highly scalable and applicable in distributed environments, where the data are not necessarily stored in local memory or disk. The algorithm is experimentally compared to a previously introduced algorithm in the same setting.

## 1 Introduction

In recent years, the number of string databases (particularly, in bioinformatics) has grown enormously [1]. One of the motivations for constructing and maintaining these databases is the desire to discover new knowledge from these databases using data mining techniques. While more traditional data mining techniques, such as frequent itemset mining [2] and frequent sequence mining [3], can be adapted to mine string databases, they do not take advantage of some properties specific to strings to accelerate the mining process. By specifically targeting string databases, it should be possible to devise more effective algorithms for discovering string patterns.

The most important contribution of this paper is the introduction of a novel algorithm, called FAVST, for mining string patterns from string databases. This algorithm combines ideas from data mining with string processing principles. More specifically, we employ ideas from suffix trees [4,5] to represent and compute the set of patterns of interest. The data structure used is that of Version Space Trees (VST, introduced by [6]) to organize the set of substring patterns being discovered. We have observed that a suffix trie can be treated as a deterministic automata so that we can visit all the substring patterns contained in a data string efficiently. We exploit this property of the suffix trie in VST and devised the FAVST algorithm (see Sect. 5.2). This algorithm performs frequency counting

* This work was supported by the EU IST FET project cInQ, contract number IST-2000-26469.

in only one database scan. It is thus is especially efficient when database access is slow (e.g. over the internet). We also compare FAVST with a more traditional level-wise data mining algorithm, called VST, that we developed earlier [6]. As it employs the same data structure VST and the same setting as FAVST, this provides an appropriate setting for comparison. Although FAVST consumes more memory than VST our experiments (Sect. 6) show that the memory requirements are relatively cheap by today's hardware standards. Furthermore, as we will show, it can be controlled by imposing an upper bound on the length of patterns to be discovered, making FAVST very attractive in practice.

The rest of this paper is organized as follows. Important definitions are introduced in Sect. 2. We take a closer look into the search space of the problem in Sect. 3 and describe a data structure to handle it in Sect. 4. Two algorithms are devised to construct this data structure. They're presented in Sect. 5. Our approach are verified by experiments presented in Sect. 6. Finally, we come up with conclusions in Sect. 7.

## 2 Definitions

### 2.1 Database and Substring Patterns

Since our goal is to mine substring patterns from a database, we have to define these two terms first. Further, not all substring patterns are interesting. We express the interestingness of the patterns using a predicate. Patterns not satisfying the predicate are considered to be uninteresting, and hence should not to be generated.

**Definition 1.** *A* database *$D$ over an alphabet $\Sigma$ is a bag (i.e. multi-set) of strings over $\Sigma$. A* substring pattern *(or "pattern" for short) $s$ over an alphabet $\Sigma$ is a string over $\Sigma$. The empty string, which has a length of zero, is denoted $\epsilon$. A* predicate *$\mathcal{P}$ for substring patterns over $\Sigma$ is a boolean function on a substring pattern $s \in \Sigma^*$ and (sometimes) a database $D$.*

**Definition 2.** *A string $s \in \Sigma^*$ is a substring of another string $t \in \Sigma^*$ if and only if $t = xsy$ where $x, y \in \Sigma^*$. (Both $x$ and $y$ may be empty.) We write $s \sqsubseteq t$ and equivalently $t \sqsupseteq s$. Define two predicates:*

$$\begin{aligned} \mathit{substring\_of}(s;t) &\equiv s \sqsubseteq t \\ \mathit{superstring\_of}(s;t) &\equiv s \sqsupseteq t \end{aligned}$$

*where $t \in \Sigma^*$ is a constant string.*

*Example 3.* Using $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, substring_of(ab; abc) and superstring_of(bcd; bc) evaluate to *true* whereas substring_of(cd; abc) and superstring_of(b; bc) evaluate to *false*.

Analogous to frequent itemset mining, we may express our interestingness in frequent substring patterns by imposing a minimum occurrence frequency.

**Definition 4.** *Given a database $D$ over an alphabet $\Sigma$ and a pattern string $s \in \Sigma^*$, we define the* frequency *$freq(s; D)$ to be the number of strings in $D$ that is a superstring of $s$. i.e.*

$$freq(s, D) = |\{d \in D | s \sqsubseteq d\}|$$

*We define two predicates related to frequency. Given a database $D$ and an integer $\theta$, define*

$$\mathit{minimum\_frequency}(s, D; \theta_{\min}) \iff freq(s, D) \geq \theta_{\min}$$

$$\mathit{maximum\_frequency}(s, D; \theta_{\max}) \iff freq(s, D) \leq \theta_{\max}$$

*Example 5.* Let $\Sigma_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $D = \{\mathtt{abc}, \mathtt{abd}, \mathtt{cd}, \mathtt{d}, \mathtt{cd}\}$. With this database, we have $freq(\mathtt{abc}) = 1$, $freq(\mathtt{cd}) = 2$, $freq(\mathtt{c}) = 3$, $freq(\mathtt{abcd}) = 0$. And trivially, $freq(\epsilon) = |D| = 5$. Thus, the following predicates evaluate to true: minimum_frequency(c, $D$; 2), minimum_frequency(cd, $D$; 2), maximum_frequency(abc, $D$; 2), maximum_frequency(cd, $D$; 2).

In some applications (e.g. MolFea [7]), it is useful to partition the database $D$ into different subsets $D_1, \ldots, D_n$ and define frequency predicates that counts only a subset of $D$. e.g. $\mathcal{A}_1 =$ minimum_frequency$(s, D_1; \theta_1)$ and $\mathcal{M}_2 =$maximum_frequency$(s, D_2; \theta_2)$. Then, we can construct a compound predicate $\mathcal{P} = \mathcal{A}_1 \wedge \mathcal{M}_2$ to mine the patterns that are frequent in the subset $D_1$ but not in $D_2$. Our experiments in Sect. 6 make use of such a setting.

### 2.2 The Substring Mining Problem

**Definition 6.** *Given an alphabet $\Sigma$, a database $D$, and a predicate $\mathcal{P}$, the problem of* Mining Substring Patterns *is to find the set of substring patterns over $\Sigma$ satisfying $\mathcal{P}$:*

$$Sol(\mathcal{P}, D, \Sigma^*) = \{s \in \Sigma^* \mid \mathcal{P}(s; D)\}$$

*Example 7.* Continuing from Example 5, let $P_1(s, D) \equiv$ minimum_frequency$(s, D_1; 2) \wedge$ superstring_of$(s; \mathtt{d})$. Then, we have $Sol(\mathcal{P}_1, D_1, \Sigma_1^*) = \{\mathtt{d}, \mathtt{cd}\}$.

## 3 The Search Space

To solve the problem of Mining Substring Patterns, one naïve approach is of course a brute-force search: check all the substrings over $\Sigma^*$ against $\mathcal{P}$ and print out the satisfying ones. However, since $\Sigma^*$ is countably infinite, one can never exhaust the whole pattern space, although one can enumerate them in a certain order.

A much better idea, as in itemset mining, is to exploit the structure of the search space, noting that $\sqsubseteq$ is a partial order relation. We will restrict the predicates to one of the following two types, or a conjunction of any number of them.

**Definition 8.** *An* anti-monotonic *predicate $\mathcal{A}$ is a predicate that satisfies:*

$$\forall s_1, s_2 \in \Sigma^* \text{ such that } s_1 \sqsubseteq s_2, \quad \mathcal{A}(s_2) \Rightarrow \mathcal{A}(s_1)$$

**Definition 9.** *A* monotonic *predicate $\mathcal{M}$ is a predicate that satisfies:*

$$\forall s_1, s_2 \in \Sigma^* \text{ such that } s_1 \sqsubseteq s_2, \quad \mathcal{M}(s_1) \Rightarrow \mathcal{M}(s_2)$$

*Example 10.* substring_of and minimum_frequency are anti-monotonic predicates whereas superstring_of and maximum_frequency are monotonic predicates.

With a compound query $\mathcal{P} = (\mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_m) \wedge (\mathcal{M}_1 \wedge \cdots \wedge \mathcal{M}_k)$, we can rewrite it as $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$, where $\mathcal{A} = \mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_m$ and $\mathcal{M} = \mathcal{M}_1 \wedge \cdots \wedge \mathcal{M}_k$. Note that $\mathcal{A}$ is anti-monotonic and $\mathcal{M}$ is monotonic. Therefore, we only need to consider predicates of the form $\mathcal{A} \wedge \mathcal{M}$.

While confining ourselves to predicates of this form may appear restrictive, we should note that in most formulations of data mining problems in the past years, an even more restrictive form of the predicate is used. For example, in most frequent-itemset, frequent-sequence mining problems, only a minimum-frequency predicate (anti-monotonic) is used. The consideration of using monotonic predicates has appeared only recently, and is still a rarity. [8,9] The general conjunction of an arbitrary number of monotonic and anti-monotonic predicate is seldom seen, either. Thus, our restricted form $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$ is already quite expressive. In other works [6,10], we suggested how to support queries that are arbitrary boolean functions of anti-monotonic and monotonic predicates.

### 3.1 Version Space

Restricting the predicate to the form $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$, the set of solutions to the Subsection Mining Problem $Sol(\mathcal{P}, D, \Sigma^*)$ turns out to be a version space [11] under the $\sqsubseteq$ relation. This means that there exists two sets $S, G \subseteq \Sigma^*$ with the following properties.

- $S = \{p \in Sol(\mathcal{P}, D, \Sigma^*) \mid \nexists q \in Sol(\mathcal{P}, D, \Sigma^*) \text{ such that } p \sqsubseteq q\}$
- $G = \{p \in Sol(\mathcal{P}, D, \Sigma^*) \mid \nexists q \in Sol(\mathcal{P}, D, \Sigma^*) \text{ such that } q \sqsubseteq p\}$
- $\forall p \in Sol(\mathcal{P}, D, \Sigma^*),\ \exists s \in S \wedge g \in G \text{ such that } g \sqsubseteq p \sqsubseteq s$
- $\forall p, q, r \in \Sigma^*$ such that $p \sqsubseteq q \sqsubseteq r$, we have: $p, r \in Sol(\mathcal{P}, D, \Sigma^*) \Rightarrow q \in Sol(\mathcal{P}, D, \Sigma^*)$

*Example 11.* The solution set from Example 7 has $S = \{\texttt{cd}\}$ and $G = \{\texttt{d}\}$.

The set $S$ is called the maximally *specific* set and $G$ is called the maximally *general* set. For more details on how this mining problem relates to version spaces, please refer to our previous works [6,10]. In this paper, we focus on the algorithms and optimizations.

## 4 Version Space Trees

To facilitate mining of string patterns, we have devised a data structure, which we called the version space tree (VST). [6,10]. We will give a brief overview of it here. The VST is based on suffix tries, similar to suffix trees [4,5].

A trie is a tree with each edge labelled with a symbol from the alphabet $\Sigma$ concerned. Moreover, the labels on every edge emerging from a node must be unique. Each node $n$ in a trie thus uniquely represents the string $s(n)$ containing the characters on the path from the root $r$ to the node $n$. The root node itself represents the empty string $\epsilon$.

A suffix trie is a trie with the following properties:

- For each node $n$ in the trie, and for each suffix $t$ of $s(n)$, there is also a node $n'$ in the trie representing $t$, i.e. $t = s(n')$.
- Each node $n$ has as a *suffix link* $\mathit{suffix}(n) = n'$ where $s(n')$ is the suffix obtained from $s(n)$ obtained by dropping the first symbol. Note that $|s(n')| = |s(n)| - 1$. The root node is special because it represents $\epsilon$, which has no suffixes. We define $\mathit{suffix}(\mathit{root}) = \perp$, where $\perp$ is a virtual node, acting as a null pointer.

*Example 12.* The VST for $\mathit{Sol}(\mathcal{P}_1, D_1, \Sigma_1^*)$ from Example 7 is depicted in Fig. 1. The numbers in the each node $n$ shows $\mathit{freq}(s(n), D_1)$. The label of each node is shown to the left of the node. The dashed arrows show the suffix links. The suffix links of the first level of nodes have been omitted for clarity. They all point to the root node. Note that this diagram is for illustrative purpose. In practice, we would prune away all branches containing only $\ominus$ nodes to save memory.

What makes VST unique is that we make two major deviations from the main stream approach in the suffix tree culture. The first one is that instead of building a suffix trie on all the suffixes of a *single* string, we are indexing all the suffixes of a *set of strings* in a database $D$. This means multiple strings are stored in the tree. As intermediate computation results, we even keep a count of occurrences of each such substring. In addition to a count, we also store a label on each node of the VST. The label $\oplus$ indicates that the represented string pattern is in our solution $\mathit{Sol}(\mathcal{P}, D, \Sigma^*)$. Otherwise, it is $\ominus$. Our algorithms in Sect. 5 exploit this label to store intermediate mining results. A second difference is that most studies on suffix trees usually consider a more compact form of suffix trie in which a chain of nodes with only one out-going edges are coalesced into one edge label with the string containing the symbols involved. We are not using this representation, as our algorithms need to keep flags and counts with each substring represented in the tree.

## 5 The Algorithms

In this section, we present two algorithms to build a version space tree, given as input an alphabet $\Sigma$, a database $D$, and a predicate $\mathcal{P}$ of the form $\mathcal{A} \wedge \mathcal{M}$.

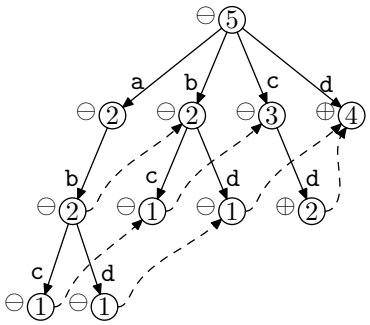


**Fig. 1.** A Version Space Tree

Algorithm VST is a level-wise algorithm based on the well-known Apriori [12] algorithm. It was first introduced in [6]. The other algorithm, FAVST is new and is based on techniques in the suffix-tree literature [4], which is much faster when the database size is large and the database access time is not negligible.

In the algorithms, the alphabet $\Sigma$ being used is restricted to the subset of "interesting symbols", i.e. those symbols that we want to appear in the discovered patterns. Uninteresting or irrelevant symbols are dropped.

### 5.1 Algorithm VST

The VST (Version Space Tree) algorithm [6] is based on Agrawal's Apriori [12] algorithm. It consists of two phases:

1. Top-down growing of the version space tree using the anti-monotonic predicate $\mathcal{A}$.
2. Bottom-up marking of the version space tree using the monotonic predicate $\mathcal{M}$.

Both phases are designed to minimise the number of database scans. As such, they both exhibit the cyclic pattern: candidate generation, candidate testing (database scan) and pruning. The cycle terminates when no more new candidates patterns are generated.

Since only the anti-monotonic predicate is handled in phase 1, the idea of Apriori is reused and it results in very efficient pruning of the search space. The second phase is implemented with algorithm Ascend. This phase handles the monotonic predicate $\mathcal{M}$.

After these two phases, both $\mathcal{A}$ and $\mathcal{M}$ have been handled. With a simply tree traversal, we can prune away branches which now contains only $\ominus$ children. We have a resulting tree $T$ that is a pruned suffix trie representing all the strings satisfying $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$.

**Theorem 13.** *The VST algorithm performs at most* $2m$ *database scans, where* $m$ *is the longest string satisfying* $\mathcal{A}$.

### 5.2 Algorithm FAVST

The drawback of the previous algorithm is that it still has to scan the database $2m$ times, where $m$ is the length of the longest string satisfying $\mathcal{A}$. Actually, strings exhibit some properties not exhibited by itemsets. Therefore, there is still room for improvements. Our next algorithm, FAVST[1] makes use of techniques from the suffix-tree literature to improve performance. It is well-known in that literature that the suffix-tree of a string can be built in linear time. Some of these ideas are employed in the FAVST algorithm to make it possible to build the version space tree with just a single database scan. We show here only how frequency-based predicates are handled. Database-independent predicates can be handled efficiently without database scanning. For other types of database-dependent predicates, predicate-specific adaptations would be needed.

**Algorithm 1** FAVST

**Input:** $D = D_1, \ldots, D_n$ the database (divided into subsets)
$\Sigma$ = the pattern alphabet
$\mathcal{A} = \bigwedge_{i=1}^{n}$ minimum_frequency$(s, D_i, \theta_{\min_i})$ the anti-monotonic predicate
$\mathcal{M} = \bigwedge_{i=1}^{n}$ maximum_frequency$(s, D_i, \theta_{\max_i})$ the monotonic predicate $\quad \text{len}_{\max}$ = maximum length of substring pattern
**Output:** $T$ = version space tree representing strings satisfying $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$.
**Body:**
$T \leftarrow$ InitTree$(D_1, \Sigma, \theta_{\min_1}, \theta_{\max_1}, \text{len}_{\max})$
Prune away branches in $T$ with only $\ominus$ nodes.
**for all** $i = 2, \ldots, n$ **do**
CountAndUnmark$(T, D_i, \Sigma, \theta_{\min_i}, \theta_{\max_i})$
Prune away branches in $T$ with only $\ominus$ nodes.
**end for**

The FAVST algorithm is shown in Algorithm 1. It first calls InitTree to process the first minimum_frequency predicate and scan the first database subset (see Sect. 2.1) to build an initial VST. Then, it invokes CountAndUnmark to process the remaining database subsets and the corresponding minimum_frequency predicates. Note that CountAndUnmark will not grow the VST. It will only count the frequency of the patterns in the corresponding database subset and mark those not satisfying the thresholds $\theta_{\min_i}$ and $\theta\max_i$ with $\ominus$. Branches with only $\ominus$ are pruned away immediately after the the scanning of each database subset to reduce the number of patterns that need to be checked against the subsequent subsets. The parameter $\text{len}_{\max}$ specifies an upper bound on the length of the substring patterns to be discovered. When set appropriately, this parameter makes FAVST to be very efficient both in terms of computation time and memory usage (see Sect. 6).

We will later see that each of the subalgorithms InitTree and CountAndUnmark scans the specified database subset only once. So, the whole FAVST algorithm

[1] FAVST stands for "Finite Automata-based VST construction".

scans each database subset only once. If the database subsets are disjoint, then we can in implementation scan only the subset of data being processed. In that case, FAVST completes in only one scan of the whole database. So, FAVST is a single-scan algorithm.

Algorithm InitTree(not shown here own to page limit) scans each string in the database subset symbol for symbol, going down the tree as it proceeds. If a node does not have a suitable child for it to go downward, one such child is created, so that we can go down the tree. Next, InitTree increments the count of the destination of this going down, as well as all its suffixes. Then, the next symbol is processed, continuing from this destination. When an uninteresting symbol (i.e. $\notin \Sigma$) or the end of a string is encountered, it starts the downward travel from the root node again.

This is basically a suffix tree building algorithm, with four modifications. The first is that we do frequency counting on the way as we go. The second is that we put an upper bound on the length of the substring patterns, or the depth of the trie. Thirdly, we jump back to the root when we encounter uninteresting symbols, saving the need to process any strings containing such symbols. Last but not least, we handle multiple strings, instead of a concatenation of these strings.

After scanning the database, InitTree performs a traversal of the trie and checks if the counts satisfy the specified thresholds. If not, it labels that node with "$\ominus$".

The CountAndUnmark algorithm is similar to InitTree except that it does not create new nodes. For any node $n$ already present in $T$, we need not recreate it or re-calculate the suffix link. We only need to increase the support count for that node. If the node $n$ is not in $T$ as we do the downward walk, we know that the string $s(n)$ that it would represent is not present in $T$, and hence it is not a pattern we are looking for (because it doesn't satisfy the predicate $a_0$ used to build the initial trie). So, there is no need to create that node. The support counts are thus counted in this manner. Again, as we visit a node $n$, we increment the counter on that node to count the occurrence of the corresponding substring pattern.

The major efficiency improvement of FAVST comes from the single database scan. Firstly, note that the algorithm does not work level-wise in the style of Apriori. Rather, it examine the predicates one by one and invokes InitTree and CountAndUnmark to scan the concerned database subsets.

On the space efficiency, since the suffix trie is $O(|D|^2)$ in size (where $|D|$ is the total number of symbols in $D$), FAVST is less space-efficient than VST. Nevertheless, in practice, we can specify a relatively small upper bound $d$ on the length of the longest substring pattern we are going to find. This can effectively limit the depth of the VST to $d$, reducing the amount of memory that FAVST would need.

## 6 Experiments

The algorithms VST and FAVST have been implemented in C. The experiments are performed on PC computer with a Pentium-4 2.8GHz processor, 2GB main memory, and running Linux operating system (kernel 2.4.19, glibc 2.2.5).

### 6.1 Unix Command History Database

The database $DB$ used in the experiments are command history collected from 168 Unix users over a period of time. [13] The users are divided into four groups: computer scientists, experienced programmers, novice programmers and non-programmers. The corresponding data subsets are denoted "sci", "exp", "nov" and "non", respectively. Each group has a number of users. When each users accesses the Unix system, he first logs in, then type in a sequence of commands, and finally logs out. Each command is taken as a symbol in the database, The sequence of commands from log in to log out constitutes a login session, which is mapped to a string in our experiment. Each user contributes to many login sessions in the database. Table 1 gives some summary data on the database. To study the effectiveness of the $\text{len}_{\max}$ parameter to FAVST, we repeated FAVST twice for each experiment: once with $\text{len}_{\max} = \infty$, essentially disabling the length limit; and once with $\text{len}_{\max} = 10$. These are denoted "FAVST" and "FAVST$^{\text{ml}}$", respectively, in all the subsequent tables and the figures.

**Table 1.** Summary statistics of the data

| Subset ($D_i$) | no. of users | no. of strings | $\theta_{\min}$ | frequent patterns |
|---|---|---|---|---|
| nov | 55 | 5164 | 24 | 294$^\star$ |
| exp | 36 | 3859 | 80 | 292 |
| non | 25 | 1906 | 80 | 293 |
| sci | 52 | 7751 | 48 | 295 |

$^\star$Of these 294 patterns, 36 have length $> 10$. They are thus dropped in FAVST$^{\text{ml}}$.

### 6.2 Performance

In the following experiment, we use the algorithms VST and FAVST to compute two version space trees $T_1$ and $T_2$, each representing a set of strings satisfying a predicate of the form $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$, where $\mathcal{A}$ is minimum_frequency and $\mathcal{M}$ is maximum_frequency. The details are tabulated in Table 2. The database used is the same as described above. The thresholds for the minimum_frequency predicate are copied from Table 1, where as those for the maximum_frequency are from Table 1 less 25%. After computing these trees, the union of them, $U$ is computed by a naïve tree-merging operation. Note that $U$ is no longer a version space tree, as it represents a subsets of $\Sigma^*$ which is not a version space anymore. We have a theoretical treatment of such pattern sets in a related work [10].

**Table 2.** An experiment on compound predicates

| Trie | $\mathcal{P} = \mathcal{A} \wedge \mathcal{M}$ |
|---|---|
| $T_1$ | minimum_frequency(non; 24) $\wedge$ maximum_frequency(sci; 60) |
| $T_2$ | minimum_frequency(nov; 80) $\wedge$ maximum_frequency(exp; 36) |
| $U$ | $T_1 \cup T_2$ |

The results of the experiments are shown in Table 3. Each row shows the time that either algorithm used to build that tree. The time taken to compute $U$ in either case is negligible, as it is done completely in memory. It takes so little time (less than 0.01 second) that we cannot reliably measure because of the granularity of the time-measurement program we are using.

**Table 3.** Results on finding the union of two version spaces

| Trie | Time (seconds) | | | number of nodes | | |
|---|---|---|---|---|---|---|
| | VST | FAVST | FAVST$^{\text{ml}}$ | labeled $\oplus$ | labeled $\ominus$ | total |
| $T_1$ | 0.56 | 0.39 | 0.19 | 166 | 40 | 206 |
| $T_2$ | 1.23 | 1.19 | 0.37 | 237$^\star$ | 18 | 255 |
| $U$ | negligible | | | 401$^\star$ | 47 | 448 |

$^\star$Of these $\oplus$ nodes, 36 are at depth $> 10$, representing patterns of length $> 10$. These are pruned away in the FAVST$^{\text{ml}}$ case.

It is encouraging that FAVST runs faster than VST. FAVST$^{\text{ml}}$ takes even less time to compute the result, although 36 patterns are not discovered due to the length limit.

The longest pattern found (represented by the deepest node in $U$ having a $\oplus$ label) was "`pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix`", which has a length of 19. The deepest $\ominus$ node in $U$ represented the string "`cd ls cd ls`", of length 4, which has an interesting (labelled $\oplus$) child representing the string "`cd ls cd ls e`". If we did not have any monotonic predicates (i.e. maximum_frequency in this case), "`cd ls cd ls`" would have been considered interesting because it is frequent enough. However, with the monotonic predicates, this string is now *too* frequent and hence it is considered uninteresting and marked with $\ominus$ in $U$. This illustrates the increased power and expressiveness of using both anti-monotonic and monotonic predicates together in data mining. The ability to compute $U$ efficiently by manipulating the results $T_1$ and $T_2$ shows the power of these algorithms in combination with the results in [6,10].

### 6.3 Memory Footprints

The speed up is a trade off with memory usage. We performed experiments to find out the memory usage. This time, we mine each data subset with the minimum frequency thresholds as given in Table 1 (and no maximum frequency

thresholds). Table 4 shows the maximum amount of memory consumed by the two algorithms for the data structures. VST has a memory footprint in the order of tens of kilobytes, whereas that of FAVST is in megabytes. With today's computing equipments, the memory consumption of FAVST is absolutely affordable. Imposing a limit on the length of patterns makes $\mathsf{FAVST}^{\mathrm{ml}}$ build a much smaller trie than FAVST, significantly reducing the memory consumption by a factor of 2–5.

**Table 4.** Memory usage of the two algorithms

| Data subset($D_i$) | Max. mem. usage (bytes) | | | Ratio | |
|---|---|---|---|---|---|
| | VST | FAVST | $\mathsf{FAVST}^{\mathrm{ml}}$ | FAVST/VST | $\mathsf{FAVST}^{\mathrm{ml}}$/VST |
| nov | 57158 | 17456767 | 3573695 | 305 | 63.5 |
| exp | 88870 | 14439604 | 5367732 | 162 | 60.4 |
| non | 59918 | 6797462 | 2081558 | 113 | 34.7 |
| sci | 94454 | 23107503 | 8997455 | 245 | 95.3 |

It should be noted that the memory consumption of the algorithms has no direct relation to the database sizes. From Table 1, we can see that the data set "nov" is larger than "exp" and "non". However, it turns out to cause the algorithms to consume less memory than the other two data sets. Our explanation is that the "nov" data set has more repeated string patterns. Since our algorithm uses the same trie node for the same pattern, the fewer the number of distinct string patterns, the fewer nodes are created, and hence the less memory consumed. In other words, the memory consumption is related to the number of distinct string patterns, but not the database size. Thus, our algorithms exhibits very nice properties for data mining applications. They scale well with database size, and use an amount of memory depending on the amount of interesting patterns that will be discovered.

## 7 Conclusions

In this paper, we have addressed the problem of Mining Substring Patterns under conjunctive constraints. This is a core component of a more general data mining framework in a couple of related works [6,10,14].

The main contribution of this paper was the FAVST algorithm, which employs the VST data structure of [6,10] and combines principles of constraint based mining with those of suffix-trees. This algorithm has the very nice property of requiring only one database scan, at the expense of very affordable memory overheads. Such overheads can be significantly reduced by imposing an upper-bound on the length of patterns. The data structure and algorithms have been empirically proved to be practical and useful for finding substring patterns in a unix user command database.

One direction for further research is concerned with data streams. It might be possible to combine the present framework with that proposed by Han et al.

## References


1. Creighton, C., Hanash, S.: Mining gene expression databases for association rules. Bioinformatics **19** (2003) 79–86
2. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In Buneman, P., Jajodia, S., eds.: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., U.S.A. (1993) 207–216
3. Agrawal, R., Srikant, R.: Mining sequential patterns. In Yu, P.S., Chen, A.S.P., eds.: Eleventh International Conference on Data Engineering, Taipei, Taiwan, IEEE, IEEE Computer Society Press (1995) 3–14
4. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
5. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14 IEEE Symposium on Switching and Automata Theory. (1973) 1–11
6. De Raedt, L., Jaeger, M., Lee, S.D., Mannila, H.: A theory of inductive query answering (extended abstract). In Kumar, V., Tsumoto, S., Zhong, N., Philip S. Yu, X.W., eds.: Proc. The 2002 IEEE International Conference on Data Mining (ICDM'02), Maebashi, Japan (2002) 123–130 ISBN 0-7695-1754-4.
7. Kramer, S., De Raedt, L., Helma, C.: Molecular feature mining in HIV data. In: KDD-2001: The Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery (2001) ISBN: 158113391X.
8. Boulicaut, J.F., Jeudy, B.: Using constraints during set mining: Should we prune or not? In: Actes des Seizième Journées Bases de Données Avancées (BDA'00), Blois, France (2000) 221–237
9. Bonchi, F., Giannotti, F., Mazzanti, A., Pedreschi, D.: ExAMiner: Optimized level-wise frequent pattern mining with monotone constraints. [15] 11–18
10. De Raedt, L., Jaeger, M., Lee, S.D., Mannila, H.: A theory of inductive query answering. (2003) (submitted to a journal).
11. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18** (1982) 203–226
12. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In Bocca, J.B., Jarke, M., Zaniolo, C., eds.: Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, Morgan Kaufmann (1994) 487–499
13. Greenberg, S.: Using unix: Collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Alberta, Canada. (1988)
14. Lee, S.D., De Raedt, L.: An algebra for inductive query evaluation. [15] 147–154
15. Wu, X., Tuzhilin, A., Shavlik, J., eds.: Proceedings of The Third IEEE International Conference on Data Mining (ICDM'03). In Wu, X., Tuzhilin, A., Shavlik, J., eds.: Proceedings of The Third IEEE International Conference on Data Mining (ICDM'03), Melbourne, Florida, USA, Sponsored by the IEEE Computer Society (2003)