Frequent Patterns that Compress

Ronnie Bathoorn

Arne Koopman

Arno Siebes

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2006-048

www.cs.uu.nl

ISSN: 0924-3275

Frequent Patterns that Compress

Ronnie Bathoorn
Department of Computer Science
Utrecht University
ronnie@cs.uu.nl

Arne Koopman
Department of Computer Science
Utrecht University
koopman@cs.uu.nl

Arno Siebes
Department of Computer Science
Utrecht University
arno@cs.uu.nl

1-10-2006

Abstract

One of the major problems in frequent pattern mining is the explosion of the number of results, making it difficult to identify the interesting frequent patterns. In a recent paper [14] we have shown that an MDL-based approach gives a dramatic reduction of the number of frequent item sets to consider. Here we show that MDL gives similarly good reductions for frequent patterns on other types of data, viz., on sequences and trees. Reductions of two to three orders of magnitude are easily attained on data sets from the web-mining field.

Keywords: MDL, frequent sequences, frequent trees.

1 Introduction

Since the first paper on association rules [2], the discovery of frequent item sets and, more general, frequent patterns has been a popular topic in data mining research. One of the main reasons for this popularity is the insight that frequent patterns potentially offer.

However, a major obstacle in the usage of frequent patterns in practice is the explosion of the number of results at low thresholds for minimal support. High thresholds result mainly in well-known results, hence low thresholds are necessary to achieve new insight.

Over the years, many solutions have been proposed for this setting, such as closed [18] and maximal [13] frequent patterns. Note that these solutions were often introduced for frequent item sets but can be generalised to frequent patterns. Most, if not all, of these solutions can be seen as reducing the volume of the resulting set of frequent patterns; either lossless (closed) or lossy (maximal).

In a recent paper [14] we have taken a radically different approach based on the Minimal Description Length (MDL) principle [7]:

> *A set of frequent patterns is interesting iff it gives a good compression of the database.*

In [14] we have shown that this approach results in sets of frequent item sets that are many orders of magnitude smaller than the set of all frequent item sets. That is, the result set is far smaller than, e.g., the set of closed frequent item sets.

The aim of this paper is to extend this approach to frequent patterns on structured data. That is, we show that in the general case MDL-based compression picks a, relatively, small set of frequent patterns that describe the structured data well. As examples of structured data we use sequences [12] and trees [4].

Although more MDL applications with structured data appear in literature, it is normally put to use to find occurring patterns within a structure via compression, not for the reduction of a large set of patterns. For example, SUBDUE [5] uses MDL to compress a given graph to find larger hierarchical structures in data. Similarly, the ED algorithm [8] uses MDL on sequence data, and captures periodical patterns in one long sequence as opposed to a collection of sequences.

2 Structured Data and Frequent Patterns

Before we describe our compression based approach to filter for a small set of frequent patterns that describe our database well, we first give a brief introduction to the specific structured data types and patterns we use in this paper.

As mentioned already in the introduction, we focus on sequences and trees. Even then, there is a variety of ways in which one can define the (frequent) patterns. Our specific choice is based on two criteria:

1. Since we want to have a lossless compression of the database, we have to cover each structured element in the database completely. Patterns with gaps make this process unduly complex, hence we restrict our attention to *gapless* patterns.

2. The other criterion is that we require our description to match easily to our test-data, which is web-mining data.

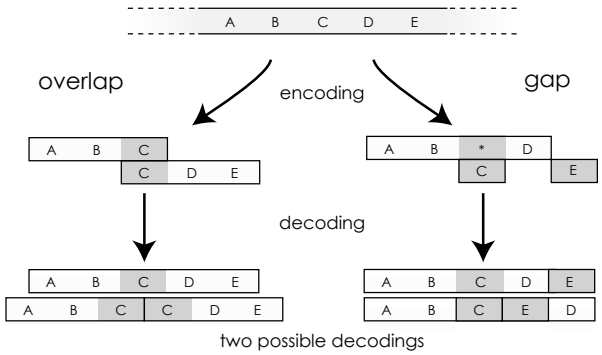

Figure 1: Ambiguity in the decoding caused by allowing overlap or gaps in the cover.

To ensure a lossless compression of the database we do not allow overlap in the database cover which results from the selected structured data elements. Without this restriction it would no longer be possible to perform a unique de-coding of the database. To illustrate this, we assume that overlap is allowed and observe the situation for two sequence codes ABC and CDE that occur both in our selected set of patterns. In this case the original sequence transaction could have been ABCDE (with overlap) or ABCCDE (without overlap); both are indistinguishable when looking at the coded string when overlap is allowed. This is illustrated in Figure 1. To solve this ambiguity we chose to not allow any overlap. Although one could argue to solve it by making the overlap mandatory with a fixed number of elements, we chose the former method.

Another consideration to take into account when preserving the lossless compression of a database is the use of gaps in patterns. Because gaps in general can have an arbitrary size, it is impossible to ensure an unique decoding. To illustrate this, the example shows the patterns AB*D,C and E that have been selected as our set of patterns. Now the original string could have been ABCDE if the gap had size 1 or ABCED if the gap size was 2. To prevent this ambiguity we disallowed gaps in our patterns. Naturally, these examples extend to other types of structured data.

2.1 Structured Data: Sequences and Trees

The basis of both the sequences and the trees we consider are *events*. A sequence is simply an ordered set of such events. We define a tree as an ordered rooted tree in which each node in the tree is labeled by one of the events [6]. For example, if the events are the web pages of a given website, a sequence would, e.g., represent

the path of pages a user has visited on that web site. The tree would represent the subtree of the website the user has visited. In our representation, we assume an order on the children of a node in the tree.

The formal definitions are as follows:

Definition Given a finite set of events Σ,

1. A sequence s over Σ is an ordered set

$$s = \{(s_i, i)\}_{i \in \{1,\ldots,n\}},$$

in which the $s_i \in \Sigma$. If $1 \leq i \leq j \leq n$,
then $(s_i, i) \preceq (s_j, j)$.

2. An ordered rooted tree t over Σ is given by a tuple:

$$t = \{V, E, v_0, L, \preceq\}$$

for which

- V is the set of nodes and E the set of edges of the tree with root v_0.
- $L : V \rightarrow \Sigma$ labels each node with its event.
- $\preceq$ is a partial order on V, which puts an order on the children of a node. For all $x, y \in V$, if $L(x) \leq L(y)$, then $x \preceq y$.

2.2 Patterns and Occurrences

As usual in structured data mining, the patterns we consider are themselves again sequences and trees over Σ. The crucial matter is the definition of an *occurrence.* As stated above, we do not allow gaps. That is, for a pattern x to occur in structured data y, we need an an injective mapping $\Phi : x \rightarrow y$ such that $\Phi(x)$ is a connected component of y. More precise, we have the following definition.

Definition Let x and y be both sequences or both trees over Σ, and $\Phi : x \rightarrow y$ a *label preserving, injective* mapping. x occurs in y, denoted by $x \subseteq y$, iff.

1. When both are sequences, if for $x_i, x_j \in x$:
   (a) $\Phi(x_i) \preceq \Phi(x_j) \Leftrightarrow x_i \preceq x_j$
   (b) $\exists y_k \in y : \Phi(x_i) \preceq y_k \preceq \Phi(x_j) \Leftrightarrow$
   $\exists x_k \in x : \Phi(x_k) = y_k \wedge x_i \preceq x_k \preceq x_j$.
2. When both are trees, x occurs in y if for $x_i, x_j \in x$:

(a) $(x_i, x_j) \in E_x \Leftrightarrow (\Phi(x_i), \Phi(x_j)) \in E_y$
(b) $\Phi(x_i) \preceq \Phi(x_j) \Leftrightarrow x_i \preceq x_j$

The mapping Φ is called the occurrence.

Note, that we do allow multiple overlapping occurrences of x in y. For db is a set of sequences or trees, the support of a pattern x is the sum of the occurrences x has in the elements $y \in db$. A pattern is called frequent if it occurs more often than a given minimal support threshold.

3 Compression using Frequent Patterns

3.1 MDL

MDL (minimum description length) [7], like its close cousin MML (minimum message length) [15], is a practical version of Kolmogorov Complexity [11]. All three embrace the slogan *Induction by Compression.* For MDL, this principle can be roughly described as follows.

Given a set of models $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimises $L(H) + L(D|H)$ in which:

- $L(H)$ is the length, in bits, of the description of H, and
- $L(D|H)$ is the length, in bits, of the description of the data when encoded with H.

In our case, $\mathcal{H}$ will consist of (ordered) sets of patterns for sequences or trees.

3.2 Code Tables and database covers

The key idea of our compression based approach [14] is that of a code table. The code table is a table with two columns, the first right-hand column contains (frequent) patterns, the second left-hand column contains the code for that pattern. We assume that each code table contains at least the singleton patterns (sequences with only one event, or subtrees with only one node). These singleton patterns are also known as the *alphabet* of our encoding. The second assumption on code tables is that we assume that its entries are ordered, i.e., there is a first entry, a second, et cetera.

With these two assumptions, we can encode each database with a code table as follows. Let $e \in db$ be a structured database element, i.e., a sequence or a tree. We search the code table for the first pattern p_i in the code table CT such that $p_i \subseteq e$. *All occurrences* of p_i in e are then replaced by the code c_i for p_i as given

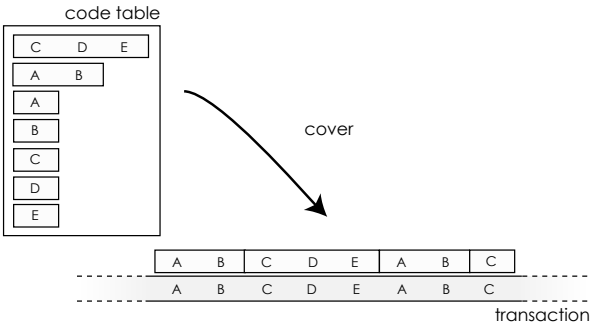

Figure 2: Sequence and tree covering.

by CT. As long as e is not completely covered by patterns in the code table, the algorithm continues. An example of a covered database is given in Figure 2. The total amount of occurrences for this pattern is given by $freq(p_i)$, and the length of replaced pattern is defined as $l_{(CT,db)}(p_i)$.

There are three things one should note about this algorithm. Firstly, each element of the database is covered by non-overlapping patterns. This is necessary to allow a decoding of the coded database; replace each code element by its pattern. Secondly there is an important distinction between entries in the code table and the database cover. As discussed earlier, we do not allow overlap in the database cover. In contrast, code table entries are allowed to contain overlap between each other. For example, both code table elements ABC and CDE can co-occur in the selected set of patterns. In addition, if both patterns occur together in sequential order frequently in the database, the larger pattern ABCDE will also appear in the code table. In fact, this pattern will be listed higher in the code table, and may make the smaller patterns redundant and may be removed by pruning.
Thirdly, since the code table contains the singleton patterns, this covering algorithm will terminate.

3.3 Formalizing the Problem

The actual code we use for a pattern is not important. What is important for MDL-based reduction is its size. Since we want optimal compression, the pattern that is used most often in covering the database should receive the shortest code.

Definition Given a database db and a code table CT, the *frequency* of a pattern $p \in CT$, denoted by $freq(p)$, is the number of times it is used to cover the elements of db.

The relative frequency of a pattern $p \in CT$ is the probability that it is used to cover a random element $e \in db$. From information theory [7] we know that an optimal code for CT will have length:

$$l_{(CT,db)}(c_i) = -\log(P(c_i)) = -log\left(\frac{freq(p_i)}{\sum_{p_j \in CT} freq(p_j)}\right)$$

Knowing the size of the codes, we also know the size of the encoded database. This is the size of the codes times the number of times they are used to cover the database:

$$L_{CT}(db) = -\sum_{p_i \in CT} freq(p_i) \times log\left(\frac{freq(p_i)}{\sum_{p_j \in CT} freq(p_j)}\right)$$

So, to apply the MDL principle, we only have to determine the size of the code table. We already know the size of the right-hand side column, it is the sum of all code sizes. In other words, we only have to determine the size of the left-hand side column.

To do this, note that the initial code table contains only the singleton patterns. If these patterns are ordered descending on their *support* in the database, this code table is called the *standard code table*. The encoding of patterns with the standard code table is called their *standard encoding (ST)*. This is the encoding we use for the left-hand side column.

Finally, note that it does not make sense to have elements in a code table that are never used in covering the database. Hence, to compute the size of the code table, we do not count entries with frequency zero. With this we have:

$$L_{db}(CT) = \sum_{p_i \in CT, freq(p_i) \neq 0} l_{(ST,db)}(p_i) + l_{(CT,db)}(c_i)$$

Now we can formally state our problem:

> *Given a database db and a set of patterns P, find code table CT whose patterns are all in P that minimizes:*
>
> $$L_{CT}(db) + L_{db}(CT)$$

Note, that a code table is assumed to contain all alphabet elements. If P does not contain these elements, we augment P so that it does.

3.4 Exact Solutions?

Ideally, we would like to have an exact algorithm that solves our problem in a feasible amount of time. However, such algorithms are unlikely to exist.

Firstly, we have to consider all subsets of P that contain all alphabet elements. Moreover, we have to consider all possible orders on these subsets. This means there are far too many possible code tables to make exhaustive search feasible.

Secondly, there is structure in the search space, but it does not help. For we know that if $P_1 \subseteq P_2 \subseteq P$, then the best possible code table we can derive from P_2 is at least as good as the best possible code table we can derive from P_1.

In other words, we will have to use heuristics to find a good code table. We will use the heuristic proven to work well in [14]

3.5 Ordering the sets

The heuristic is based on orders, one on the entries in the code table, the other on the set of frequent patterns that are used as input.

Clearly, we already assume that code tables are ordered. However, now we define a standard order on a code table. Given two patterns $p_1, p_2 \in CT$, the p_1 entry is before the p_2 entry iff:

1. p_1 is a bigger pattern than p_2, i.e., it is a longer sequence or it is a tree with more nodes.
2. Or, if they have equal size, because p_1 has larger support in the database than p_2.
3. If both measures are equal, the order is arbitrary.

Together with the covering algorithm we already described, this order means that preference is given to large patterns over small embedded ones.

Next to the order on the code table, we also enforce an order on the set of patterns that form the input of our algorithm. The patterns will be considered in this order for insertion in the (growing) code table. For two patterns p_1 and p_2 in the input ordered input set, p_1 occurs before p_2 iff:

1. p_1 has a larger support than p_2,
2. if they have the same support, p_1 is a larger pattern than p_2.
3. If both measures are equal, the order is arbitrary.

```
COMPRESS(Patternset)
1  codetable = allsingletonpatterns;
2  minDBsize = computeSize(codetable);
3  foreach pattern in frequentPatterns (in order)
4      codetable.add(pattern); (in place)
5      newDBsize = computeSize(codetable);
6      if (newDBsize < minDBsize)
7          minDBsize = newDBsize;
8      else
9          codetable.remove(pattern);
10 return codetable;
```

Figure 3: Compress algorithm. Algorithm to find the minimal describing coding set.

3.6 Algorithm description

A standard frequent tree or sequence mining algorithm mines for frequent patterns and are feeded into our compression algorithm in an ordered fashion (as described above). These frequent patterns are used to build a code table to compress the database. Initially filled with only the singleton patterns, the code table gradually grows with added patterns that are sequentially picked from the ordered candidate set.

For each new code table CT, the algorithm uses the algorithm $computeSize$ to compute the MDL size, $L_{CT}(db) + L_{db}(CT)$, for this code table (see Figure 3). It first computes the cover as discussed in paragraph 3.2. Then it computes the sizes of the code table and the compressed database. If the total size is smaller than the current minimal size the pattern is kept in the code table and the minimal size is updated, otherwise the sequence is dropped from the code table since it does not contribute to the compression of the database.

As patterns may be preserved that have lost their use for compressing the database, we apply a code table pruning method to further improve the frequent pattern reduction (see Figure 4). In this final processing step we apply a greedy pruning algorithm that starts at the bottom of the code table to remove the non-contributing smallest patterns. The effect of this lesion is derived by recomputing the cover. If this results in a better compression the pattern is left out, otherwise it is reinserted in the code table. In this fashion all items in the code table are evaluated to see if they can be removed. To ensure the complete database cover singleton patterns are never pruned.

```
PRUNE(codetable)
1  foreach code in codetable CT \ singletons α (in reverse order)
2      codetable.remove(code);
3      newDBsize = computeSize(codetable);
4      if (newDBsize ≤ minDBsize)
5          minDBsize = newDBsize;
6      else
7          codetable.add(code);
8  return codetable;
```

Figure 4: Prune algorithm. Removes obsolete patterns as post-processing from the code table.

3.7 Related work

In literature a similar MDL-based method to extract structured patterns from a database is known, called SUBDUE. However, there are some differences on which we like to focus. Firstly, SUBDUE's goal is different as it aims at finding hierarchical compression patterns within the single graph structured data, in contrast to the use of MDL to select the interesting patterns from an existing set of frequent pre-mined set of patterns. The prime focus on single graphs makes finding patterns over transactions not feasible without preprocessing the input data for SUBDUE. SUBDUE sequentially replaces smaller structures by a single node, and repeats the MDL concept compression, which leads to a structure hierarchy.
Secondly, a more important difference with SUBDUE is the fact that SUBDUE performs candidate generation based on MDL heuristics opposed to our exploration of the generated frequent pattern set. This can lead to some different results due to the fact that sub patterns with a larger MDL-size will not be expanded to a larger pattern still having a high frequency, while still being a potential interesting pattern. This could be partly compensated by the inexact graph matching, but would make lossless compression infeasible.
As the beam size, which is the selected set of patterns that is to be expanded iteratively with a single node, is fixed, and the amount of patterns that are available per size is variable, this is possible to happen. If for example the database is very skewed, as in our experiments later on, a lot of small patterns can occur and together with only a few larger ones. SUBDUE will not select the small fragments from the larger strings as they will not likely contribute much in the database compression in terms of MDL compression, instead it will expand the plethora of smaller subitems, which may not occur at all in the larger patterns.
The original authors noted the effect of possible lack of finding patterns in their

Table 1: Database characteristics. The databases vary widely in their record length and have a short average. The number of singletons (α) is high.

data type	data set	# records	avg.size	# α
sequences	KDDcup	234,954	3	835
trees	logml	8074	8	9060
	US 2430	7409	8	9284
	US 304	7628	7	8928

earlier work as an effect of their heuristic beam wise search. For our specific purpose however, this would be unsuitable to use, and the point of making this remark is solely to emphasize our different approach angle.

In the field of sequence data mining, there also is an MDL-based method for the extraction of patterns called Episode Discovery (ED). ED tries to discover specifically periodic features in a sequence using MDL. This is not posed as a restriction to our code table elements, as they only need to aid in the compression of the database. Furthermore, an important difference is that ED starts with a smaller collection of maximal frequent episodes from which it generates candidates. For each of these candidates a compression ratio is determined based on MDL that is used to tag potentially interesting single patterns. In contrast we search for a minimal set of frequent patterns that together give the best MDL-description of the database. This leads to the inclusion of less frequent patterns that are potentially interesting.

4 Experiments

We have applied the described MDL-based reduction to some publicly available data sets from the web-mining field. As we are interested in the volume reduction and not so much the actual compression factors, we focus on reduction first and subsequently present the quality of the selected patterns in terms of the size distribution and induced database cover per pattern. Throughout the following sections we have presented results for both the sequence and the tree data-type cases. The depicted 2 data sets are representative for all other conducted experiments. Our databases are skewed, making them harder for MDL compression as some transactions are around 100 times larger than the smaller records. Moreover, the larger alphabets indicate a wide variety of possible patterns which is also potentially difficult for MDL. In short, the chosen data sets pose an interesting challenge.

4.1 Sequences

For the sequence data experiments we selected the KDD Cup 2000 data set which consists of clickstream and customer data of an e-commerce retail web site (see table 1) [9]. It contains 777,480 clicks divided over 234,954 sequences. From the clickstream data we derived a collection of frequent sequences without gaps. In the 2 experiment sets we limited the windows size to 60 and 120 seconds, resulting in 2 collections of frequent sequences that fit within these window sizes. From this set we compose the code table that is used to compress the database. The results for these two experiments can be seen in Table 2.

The compression method was applied on the frequent sequences using 3 different minimal support levels to show the effect of this variation. Reduction increases for decreasing minimal support levels and attains ratios down to 37%. After compression we applied pruning to remove the items that lost their frequency to the longer encapsulating sequences that were added later during the compression. This resulted in another reduction in the number of interesting patterns; now levels of 10% are achieved. If we allow longer patterns through the use of a larger window size the reduction becomes a little bit less, although still comparable, for the same minimal support level. This is caused by the fact that longer patterns can already result in compression at a lower frequency as longer patterns have a higher change of being smaller than their code in the code table.

Figure 5 shows the pruning effect on the size distribution of the used sequences in the code table. We see a shift towards the use of longer sequences when comparing all frequent sequences to the sequences in the code table; long sequences have a large effect on the database compression. When we compare the content of the code tables before and after pruning, we see that the largest and smallest sequences are often pruned from the code table. As for the singleton elements, due to our

Table 2: Sequence reduction results. The amount of elements left in the code table after reduction $(\#CT)$ is heavily reduced compared to the original amount of candidate frequent sequences. $(\#sequences)$

window size	**60 sec**			**120 sec**		
minsup	0.06%	0.04%	0.02%	0.06%	0.04%	0.02%
#sequences	**1477**	**1955**	**3,076**	**1719**	**2365**	**3,876**
$\#\boldsymbol{CT}$	1,422	1,569	1,845	1,513	1,757	2,184
$\#\boldsymbol{CT^p}$	1,079	1,095	1,129	1,200	1,274	1,377
$\#\boldsymbol{CT}\backslash\boldsymbol{\alpha}$	587	734	1,010	675	922	1,349
$\#\boldsymbol{CT^p}\backslash\boldsymbol{\alpha}$	**244**	**260**	**294**	**365**	**439**	**542**
$\%\boldsymbol{CT^p}\backslash\boldsymbol{\alpha}$	**16.5%**	**13.3%**	**9.6%**	**21.2%**	**18.6%**	**14.0%**

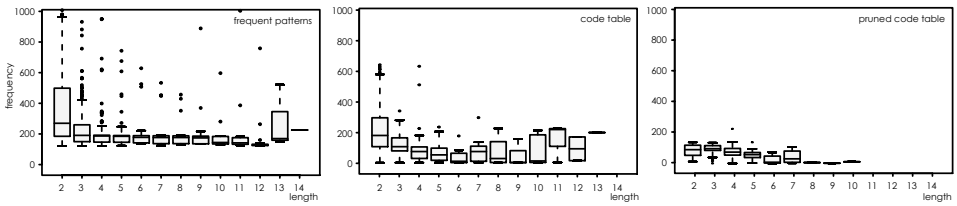

Figure 5: Distribution analysis of the frequent pattern size histograms for the KD-Dcup data with window size 60sec and minsup 150. In (a) we see that the input frequent pattern set has an expected monotone decay. However, in (b) the distribution of the resulting code table is more balanced, and most of the smaller patterns have been removed. After pruning (c), a further shift to the more averaged sized patterns is shown.

complete cover restriction only a small fraction is used frequently (see Figure 8).

The singleton patterns contained in the code table can be attributed partly to the data distribution which includes many single element transactions that require to be covered by these singletons.

4.2 Trees

In the experiments to reduce the set of frequent subtrees, we have chosen three different data sets; logml, US 304 and US 2430, which all contain weblog data (see table 1 for details) [16]. To generate all frequent induced subtrees, we used the FREQT algorithm implemented by Taku Kudo [1, 17, 10], and we applied a

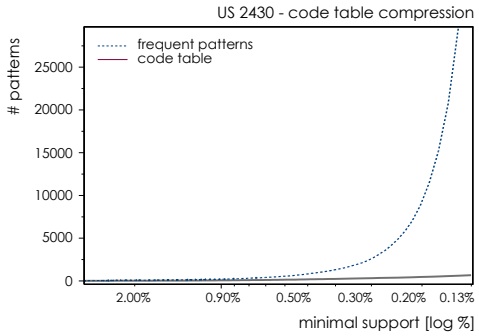

Figure 6: The dashed line indicates the frequent pattern set growth as a function of the minimal support for the US 2430 data set. The corresponding code table grows less steep.

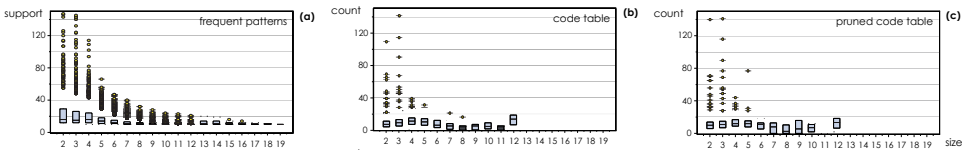

Figure 7: Distribution analysis of the pattern size histograms for the US2430 data set. The input frequent pattern set (a) has an expected monotone decay. However, the code table before (b) and after (c) pruning shows more balanced distributions.

subtree matching algorithm similar as described in [3]

While generating the frequent subtrees for all data sets, we counted the frequency per row for the minimal support level as described in section 2.2. Here again we applied compression on all data sets, followed by a prune on their respective code tables, removing in reverse support order the code table elements that do not contribute to the database compression anymore.

We obtain similar good reduction results that occur up to three orders of magnitude: 0.17% of the frequent pattern set that was originally generated from the logml data base. The specific obtained reduction ratios are a function of the minimal support levels and the data characteristics, which both potentially lead to large available patterns in the resulting set. In general we see that the pattern set growth for the lower minimal support levels is steeper than the code table growth, which stays tamed at a reasonable level (see Figure 6).

From our databases we derived the frequent subtrees up to a predefined minimal support level. As one would expect from the a-priori principle, frequent subtree mining results in a collection of subtrees diminishing monotone over pattern size (see Figure 7a). This is even accentuated due to the skewness of the database used

Table 3: Reduction results. The amount of elements left in the code table after reduction $(\#CT)$ and pruning $(\#CT^p)$ is heavily reduced compared to the original amount of candidate frequent subtrees $(\#trees)$.

dataset	**US 2430**	**US 304**	**logml**
minsup	0.13%	0.20%	0.15%
#trees	**46,232**	**196,392**	**275,377**
$\#\boldsymbol{CT}$	10,001	9,409	9,650
$\#\boldsymbol{CT^p}$	9,855	9,312	9,540
$\#\boldsymbol{CT}\backslash\boldsymbol{\alpha}$	717	481	590
$\#\boldsymbol{CT^p}\backslash\boldsymbol{\alpha}$	**571**	**384**	**480**
$\%\boldsymbol{CT^p}\backslash\boldsymbol{\alpha}$	**1.24%**	**0.20%**	**0.17%**

in our experiments which contains a large number of short transactions. When we look at the resulting code tables, we see a more stable result, in the sense that larger patterns occur relatively more often. Small patterns still dominate, as they most likely complete the cover of the database. Most importantly, a large amount of small non-informative patterns is removed, as the MDL principle selects those patterns that contribute to the database description. In the distribution of the code table elements, the number of outliers are reduced before and after pruning via MDL.

When we look at the usage of the alphabet elements (see Figure 9), we see that again only a minority of the alphabet is used often and the majority is used infrequently. In contrast to the sequence data set, the trees lack singleton tree transactions, which can be seen in the lower number of cover attained by the most frequent singleton code table element.

4.3 Coverage Analysis

In order to further assess the information capacity of the code table elements, we have run an analysis on the contribution of each code table element on the cover of the database. Here we view the cover contribution per pattern as an interestingness measure. In order to compute this, we first define the partial cover as:

$$partial\_cover(1, x) = \sum_{i=1}^{x} freq(c_i) \times l_{(CT,db)}(c_i)$$

Note that for $x = |CT|$ the partial cover is equal to the original database size.
Using this definition, we now define the derivative of this partial cover as the increase of database area cover caused by the current pattern:

$$\frac{\Delta partial\_cover(1, x)}{\Delta i} = freq(c_i) \times l_{(CT,db)}(c_i)$$

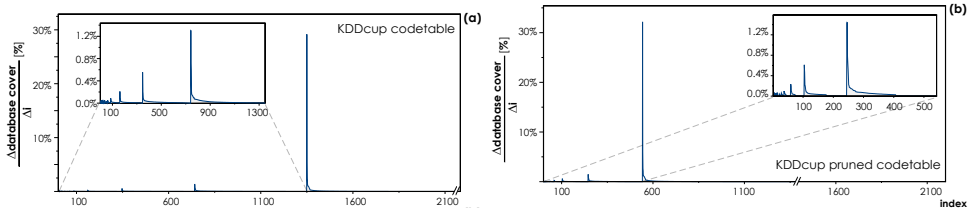

Figure 8: The plots shown here depict the pattern contribution of the database cover for the frequent patterns of the KDD cup data set (minsup 0.02%). The derivative of the code table before (a) and after (b) pruning show that the most contributing patterns lie in the top.

In both scenarios we see that the main contribution of the database cover lies in the non-singleton part of found code table, shown in the left of the graphs see Figs. 8 and 9). In the inlay of the graph the derivative of this part of the code table is enlarged. Here we also see that the median of the cover is reached before half of the contents of this code table portion (47.5% and 32.4% for sequences and trees respectively). This indicates that our presented order places the most covering patterns close to the top of our code table, indicating that when experts evaluate the contents of the code table, they will find these patterns early on in their evaluation. The consecutive peaks in the code table patterns cover contribution is due to the code table ordering. Patterns of specific window lengths are contained in the database, and upon reaching a specific size barrier in the code table, a large portion of the database is covered.

The final high peak is derived from the first alphabet items in our code table which are used in the cover: only a small fraction is used often. The remainder is mainly used to 'fill up' the database cover, and makes our earlier consideration to ensure that all alphabet items are contained in the code table apparently just.
Furthermore, the pruning of the code table does not affect the encoding capabilities. The smaller set of non-singleton code table elements show a similar, and even slightly better, behaviour in terms of the partial derivative of the database cover. Meaning this smaller set of patterns also encodes the complete database, and have a higher value in terms of coverage. Here again the 'point of gravity' lies equally early in the derivative cover distribution (47.5% and 32.6% for sequences and trees respectively).

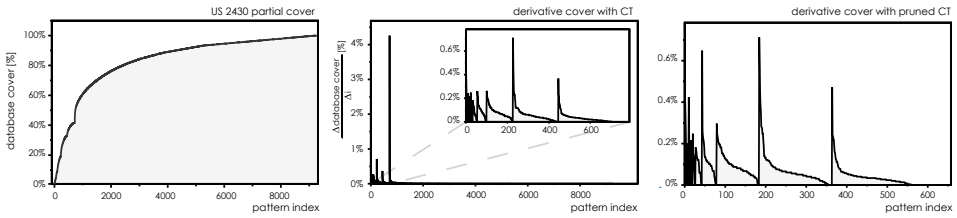

Figure 9: The partial cover derivatives before (b) and after (c) of the US 2430 data set for a minimal support level of 0.13% show the contribution per pattern of the database cover.

5 Discussion

We obtain good results for a wide variety of sparse web-mining databases with various minimal supports and frequent pattern set sizes. Compression clearly reduces the frequent pattern set significantly, and makes it applicable for domain expert evaluation. Around 10% of the original set is considered relevant for sequence data and remains preserved. We see an even higher reduction to about 1% for tree structured data. In general, we see the trend that the code table growth as a function of the minimal support follows the frequent pattern set explosion only moderately. We also confirm that low min-sup thresholds do reveal more of the structure in the database than higher thresholds. Note that the larger the average number of nodes the structured elements in the database have, the larger the reduction we attain is. This makes sense, as larger structured elements give more opportunity for compression.

Improvement continues when pruning removes obsolete patterns from the reduced pattern set that is located in the code table. The reduction is even more impressive if we disregard the alphabet elements. Since the novelty for the user is in general found in the longer patterns, this is the number of frequent patterns the user will have to pursue. In the logml database, this means that the user will only have to look at 480 out of the 275.000 frequent subtrees; a three orders of magnitude reduction.

In parallel to reduction in number, further improvement is gained as small and trivial patterns are removed, leading to a more balanced set of patterns in terms of length that are potentially more interesting see Figs. 5b and 7b. Pruning leads to additional data enhancement by stripping of the non-contributing outlier patterns, which could indicate that the selected patterns are used more generally for the database cover.

In terms of coverage, we also see that our code table plays a major role in describing the patterns. Even when focussing on the code table itself, we see that the major contribution lies in the top of the selected patterns (see Figs. 8 and 9). Post pruning, we still see correct results in terms of coverage: less patterns describe an equal part of the database, thus making the pruned code table even more informative.

6 Conclusion

Our MDL approach picks small informative sets of patterns from the potentially vast sets of frequent patterns from structured data. Reductions up to three orders of magnitude have been seen in the experiments. We have also shown that the

exponential explosion of the frequent pattern set for lower minimal support levels is followed much more moderately. This volume reduction, especially at low minimal support levels, allows the inspection of our selection of patterns: the code table. Moreover, this reduction does not simply favour small or large patterns. Rather the informative patterns are a balanced choice of frequent patterns of all possible sizes. Finally, we have seen that the code table is friendly in terms of user evaluation. In terms of coverage the most interesting patterns are listed at the top of our code table. This in practice will mean that the user will even have to evaluate a sub set of this already reduced set of patterns.

As noted in the discussion, the rate of reduction we attain depends on the average size of the structured elements in the database.Therefore we plan to extend our algorithms and the experiments to XML data. In such data, the structured elements tend to be far larger, thus possibly allowing for even higher reduction ratios.

References

[1] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Optimized substructure discovery for semi-structured data. In *PKDD '02: Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 1–14, Helsinki, Finland, 2002. Springer-Verlag.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[3] Cedric Chauve. Tree pattern matching for linear static terms. In Arlindo L. Oliveira Alberto H. F. Laender, editor, *SPIRE 2002: Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, pages 160–169, Lisabon, Portugal, 2002. Springer-Verlag.

[4] Yun Chi, Siegfried Nijssen, Richard Muntz, and Joost Kok. Frequent subtree mining - an overview. *Fundamenta Informaticae*, 2004.

[5] Jeffrey Coble, Diane J. Cook, Lawrence B. Holder, and Runu Rathi. Structure discovery from sequential data. In *FLAIRS 2004: Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*, Florida, USA, 2004. AAAI Press.

[6] Jeroen de Knijf and Ad Feelders. Monotone constraints in frequent tree mining. In *BENELEARN:Proceedings of the 14th Annual Machine Learning Conference of Belgium and the Netherlands*, pages 13–20, Enschede, The Netherlands, 2005.

[7] Peter Grünwald. *Advances in Minimum Description Length. A tutorial introduction to the minimum description length principle*. MIT Press, 2005.

[8] E. O. Heierman, G. M. Youngblood, and D. J. Cook. Mining temporal sequences to discover interesting patterns. In *In Proceedings of KDD 2004*, Seattle, WA, USA, 2004.

[9] Ron Kohavi, Carla Brodley, Brian Frasca, Llew Mason, and Zijian Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[10] Taku Kudo. http://chasen.org/ taku/software/freqt/.

[11] M. Li and P. Vitányi. *An introduction to kolmogorov complexity and its applications*. Springer-Verlag, 1993.

[12] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[13] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93, New York, NY, USA, 1998. ACM Press.

[14] Arno Siebes, Jilles Vreeken, and Matthijs van Leeuwen. Itemsets that compress. In *SIAM 2006: Proceedings of the SIAM Conference on Data Mining*, pages 393–404, Maryland, USA, 2006.

[15] C.S. Wallace. *Statistical and inductive inference by minimum message length*. Springer, 2005.

[16] Mohammed Zaki. http://www.cs.rpi.edu/ zaki/software/.

[17] Mohammed Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.

[18] Mohammed J. Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *In Proceedings of 3 rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, 1998.