Reducing the Frequent Pattern Set

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

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 [7] 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, the discovery of frequent patterns has been a popular topic in data mining research. One of the main reasons for this popularity is the insight that these 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 and maximal 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 [7] we have taken a radically different approach based on the Minimal Description Length (MDL) principle [3]: "*A set of frequent patterns is interesting iff it gives a good compression of the database.*"

In [7] 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. 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 and trees.

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

To ensure a lossless compression of the database, we do not allow overlap in the database cover which results from the selected patterns. To illustrate this, we assume that overlap is allowed and see that for two sequence codes ABC and CDE that occur both, the original sequence transaction could have been ABCDE (with overlap) or ABCCDE (without overlap). (Figure 1). To solve this ambiguity we chose to not allow any overlap.

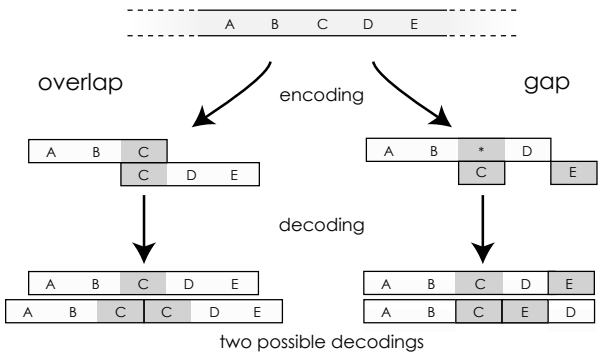

Figure 1. Ambiguity in the decoding caused by allowing overlap or gaps in the cover.

Another consideration to take into account when preserving the lossless compression of a database is the use of gaps in patterns. Figure 1b 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. Again, to prevent this ambiguity we disallowed gaps in our patterns.

2.1 Sequences and Trees

The basis of both the sequences and the trees we consider are *events*. A sequence is 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. 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 encoding

MDL (minimum description length) [3], 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.

The key idea of our compression based approach [7] is that of a code table. The code table is a table with two columns, the first column contains (frequent) patterns, the second column contains the code for that pattern. We assume that each code table contains at least the singleton patterns (or *alphabet* α): sequences with only one event, or subtrees with only one node. The second assumption on code tables is that we assume that its entries are ordered; larger patterns over smaller, and frequent over less frequent.

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 ($freq(p_i)$) are then replaced by the code c_i for p_i as given by CT. The total amount of occurrences for this pattern is given by $freq(p_i)$, and the length of replaced pattern is defined as $l_{(CT,db)}(p_i)$. The covering continues as long as e is not completely covered by patterns in the code table.

3.2 Algorithm description

A standard frequent tree or sequence mining algorithm mines for frequent patterns that are fed into our compression algorithm in an ordered fashion. 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 computes the MDL size [7]. If the total size is smaller than the previous minimal size the pattern is kept in the code table and the minimal size is updated, otherwise it is dropped from the code table. As patterns may be preserved that have lost their use for compressing the database, we prune those patterns that do not minimise the encoded MDL size. To ensure the complete database cover singleton patterns are never pruned.

3.3 Related work

In literature a similar MDL-based method to extract structured patterns from a database is known, called SUBDUE [2]. However, there are some differences on which we like to focus. Firstly, SUBDUE's goal is different as it primarily aims at finding hierarchical compression patterns within the single graphs, in contrast to collections. Secondly, SUBDUE performs candidate generation based on MDL heuristics opposed to our exploration of the generated frequent pattern set. If for example the database is skewed, as in our experiments, a lot of small patterns can occur 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.

4 Experiments

We have applied the described MDL-based reduction to some publicly available data sets from the web-mining field. 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 results 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 are also potentially difficult for MDL.

Table 1. Database characteristics.

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

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) [4]. 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 [6] that fit within these window sizes.

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) [8]. To generate all frequent induced subtrees, we used the FREQT algorithm implemented by Taku Kudo [5].

4.1 Reduction

From these frequent pattern sets we compose the code table that is used to compress the database. The results for these two experiments can be seen in Tables 2 and 3. Reduction increases for decreasing minimal support levels and attains ratios down to 37% for the sequence data sets ($CT\backslash\alpha$). After compression we applied pruning which resulted in an-

Table 2. Sequence reduction results.

window size	60 sec	120 sec
minsup	0.02%	0.02%
#sequences	**3,076**	**3,876**
$\#CT$	1,845	2,184
$\#CT^p$	1,129	1,377
$\#CT\backslash\alpha$	1,010	1,349
$\#CT^p\backslash\alpha$	**249**	**542**
$\%CT^p\backslash\alpha$	**9.6%**	**14.0%**

Table 3. Tree Reduction results.

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

other reduction in the number of interesting patterns; now levels of 10% are achieved ($CT^p \setminus \alpha$). If we use 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.

For trees, 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. 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 reasonable levels.

4.2 Pattern Size

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 2a). This is even accentuated due to the skewness of the database used 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. 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.

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

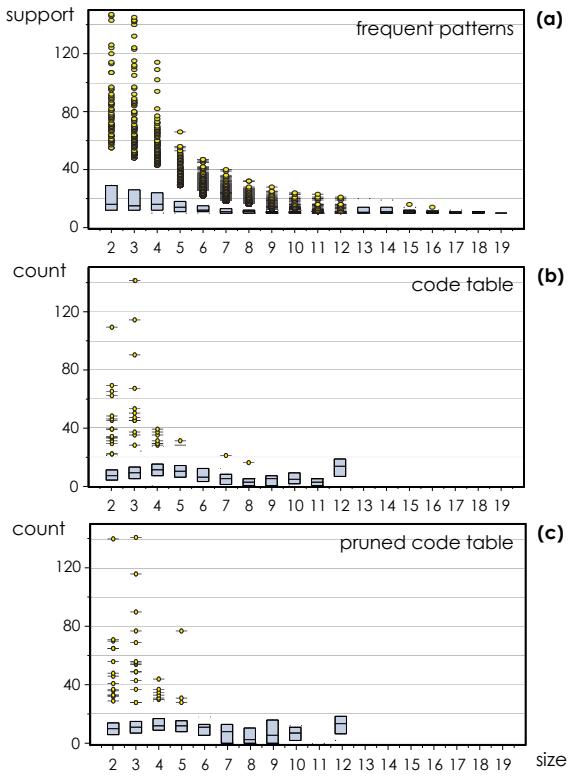

Figure 2. Distribution analysis for the US2430 data set. the decay (a) is more balanced after compression (b) and pruning (c).

4.3 Coverage

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 to the database cover. 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:

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

In both scenarios we see that the main contribution of the database cover lies in the non-singleton part of the found code table, shown in the left of the graph (see Fig. 3). In

the inlay of 3b 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. Again while pruning increases the volume reduction it maintains the quality in terms of this 'point of gravity'.

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.

As expected, 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.

5 Conclusion

Our MDL approach picks small informative sets of patterns from the potentially vast sets of frequent patterns from structured data.

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 we see an even higher reduction to about 1% for tree structured data. Furthermore, we confirm that low min-sup thresholds do reveal more of the structure in the database than higher thresholds. Note that larger structured elements in the database leads to a higher reduction. This makes sense, as they give more opportunity for compression.

Improvement continues when pruning removes obsolete patterns and when 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. We use compression to find small informative sets of patterns. Small and trivial patterns are removed, leading to a more balanced and ordered set of patterns that are potentially more interesting. More details related to the presented work can be found in the accompanying technical report [1].

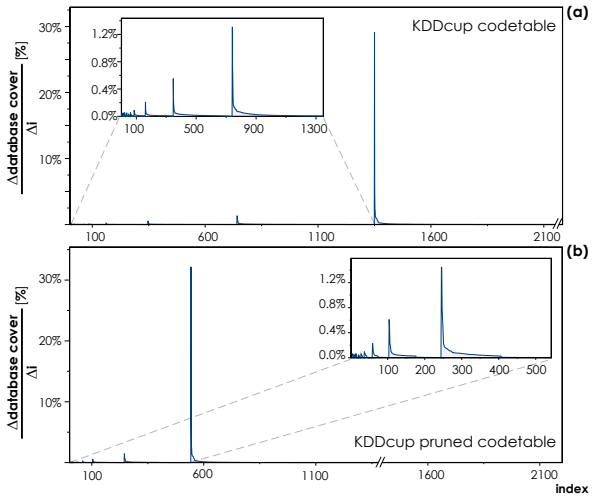

Figure 3. Derivatives of the partial cover for the KDD cup data set (a) and post prune (b).

References

[1] Ronnie Bathoorn, Arne Koopman, and Arno Siebes. Frequent patterns that compress. Technical Report UU-CS-2006-048, Department of Information and Computing Sciences, Universiteit Utrecht, 2006.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

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