SHrimp: Descriptive Patterns in a Tree

Sibylle Hess, Nico Piatkowski, and Katharina Morik

TU Dortmund University, 44227 Dortmund, Germany
sibylle.hess@tu-dortmund.de
http://www-ai.cs.tu-dortmund.de

Abstract. The appliance of the minimum description length (MDL) principle to the field of theory mining enables a precise description of main characteristics of a dataset in comparison to the numerous and hardly understandable output of the popular frequent pattern mining algorithms. The loss function that determines the quality of a pattern selection with respect to the MDL principle is however difficult to analyze and the selection is computed heuristically for all known algorithms. With SHrimp, the attempt to create a data structure that reflects the influences of the pattern selection to the database and that enables a faster computation of the quality of the selection is initiated.

Keywords: Pattern Mining, MDL, Pattern Selection, Itemsets.

1 Introduction

The identification of interesting subsets and pattern mining in general is a fundamental concept when it comes to compute characteristics of large databases. Patterns shall reflect the inherent structure of the dataset, particular interesting or at least reoccurring parts of it. As described by Mannila and Toivonen [7] the theory of the dataset, represented by the subsets that satisfy a given predicate of interest, is required. The most common practice is the frequent pattern mining [1], where the relevance of a pattern is identified with its frequency. The monotonic property of frequent sets, that all subsets of a frequent pattern are also frequent, results however in an output with highly redundant patterns. In addition, the threshold that denotes the minimal frequency to be fulfilled, is hard to set. A high threshold often results in a small set of patterns that reveals nothing but common knowledge and a lower one lets the number of issued patterns literally explode. These enormous amounts of patterns are hard to understand and the connections or correlations between them are not given explicitly.

Another approach has been introduced by Siebes et al. with Krimp [12] where the most interesting patterns are selected according to the *minimum description length* (MDL) principle [3]. The best set of patterns is identified as the one that compresses the database best. This strategy reduces the number of returned patterns drastically, e.g. from billions to less than 700 itemsets for the mushroom database, and has a variety of applications [10,11,5]. KRIMP has as input a preferable large set of frequent patterns, the code word candidates, a set that is likely to be very much larger than the input dataset. KRIMP tries to find the best selection of patterns from this candidate set with respect to the encoding of the database in a greedy procedure. Every candidate pattern is regarded once and the compression size with the considered pattern is computed. Since the consequences for the encoding of the database when a single candidate is added to the set of code words is difficult to derive, the whole database has to be traversed for each of the candidates to identify affected parts of the database such that their encoding can be recomputed.

The algorithm SLIM [9] encounters this problem by a candidate pattern generation that follows directly from the current selection of code words. In every iteration, candidates are generated and sorted by their estimated compression gain and the first pattern that enhances the compression size indeed is accepted. The estimation of the compression size requires however an identification of the affected parts of the database and for some candidates the actual compression size has to be computed as well. These operations are performed for most of the time and the combinatorial possibilities for the candidate generation are numerous, thus, an indexing structure that supports these operations and that gives insight into the consequences of integrating a pattern into the encoding, is desirable.

With SHRIMP we introduce a tree structure that facilitates a fast identification of the interesting parts of a dataset and enables a direct determination of the consequences that result from a change of the current pattern selection. In addition, the tree structure reflects the dependencies of mined patterns completely and can be used to get fundamental as well as more profound views of the characteristics of the given dataset.

We proceed as follows: In section 2 we give a theoretical introduction to the principles of KRIMP and the algorithmic procedure. We proceed with an explanation of the tree structure of SHRIMP and its application concerning the candidate selection in section 3. Runtime comparisons for some well studied datasets are shown and discussed in section 4 and we conclude in section 5.

2 Preliminaries

Let $\mathcal{P}(X)$ denote the power set of X. Given a set of items $\mathcal{I}$ we define a database $\mathcal{D} \subseteq \mathcal{P}(\mathcal{I})$ as a set of transactions $t \subseteq \mathcal{I}$. For a given minimum support $minsup \in [0,1]$ a set $X \in \mathcal{I}$ is called frequent if

$$sup(X) = \frac{|\{t \in \mathcal{D} | X \subseteq t\}|}{|\mathcal{D}|} \geq minsup.$$

The value $sup(X)$ is called the support of an itemset X. We define $\mathcal{F}$ as the set of all frequent patterns, the regarded candidates.

2.1 The MDL principle

MDL has been introduced by Rissanen et al. [8] as an applicable version of the Kolmogorov complexity [6]. Given a set of models $\mathcal{M}$, the best model is identified as the one that minimizes the compression size

$$L(\mathcal{D}, M) = L(\mathcal{D}|M) + L(M),$$

whereby $L(\mathcal{D}|M)$ denotes the compression size of the database in bits, assuming that model M is used for the encoding and $L(M)$ is the description size in bits of the model M itself.

2.2 Encoding the Database

We define a coding set $CS \subseteq \mathcal{P}(\mathcal{I})$ as a set of patterns that contains at least all singleton itemsets $\{\{x\}|x \in \mathcal{I}\}$. Let $\mathit{code} : CS \to \{0,1\}^*$ be a mapping from patterns in the coding set to a finite, unique and prefix-free code. A *code table* is denoted by a set of pairs

$$CT \subseteq \{(X, \mathit{code}(X))|X \in CS\}.$$

Code tables represent the compressing models in Krimp and can be interpreted as dictionaries for code words. Once the *code* function is determined, the coding set induces a code table. The problem can thus be formalized as the task of finding the best compressing coding set of a database. The explicit *code* function is introduced in section 2.3.

The function $\mathit{cover} : \mathcal{P}(\mathcal{I}) \to \mathcal{P}(CS)$ selects the patterns of a coding set, and with that the code words, that encode a specified transaction or itemset in general. With respect to a given transaction t, the set $\mathit{cover}(t)$ is called *cover set* and the items $x \in X \in \mathit{cover}(t)$ are called *covered by* X.

2.3 Computing the Compression Size

The codes are created such that the more frequently used patterns get the shorter codes. The existence of such a code is guaranteed by Theorem 5.4.1 in Cover & Thomas [2], that states for a given distribution P over a finite set $\mathcal{X}$, that there exists an optimal prefix-free code such that the length of the code $L(\mathit{code}(X))$ for $X \in \mathcal{X}$ is given by

$$L(\mathit{code}(X)) = -\log(P(X)).$$

Codes that satisfy this property are e.g the Shannon-Fano or Huffman code. The probability distribution over all itemsets in the code table is defined as follows. Let CT be a code table and $X \in CS$ an itemset of the respective coding set. The probability of X to be used for the encoding in a given database is defined as

$$P(X) = \frac{\mathit{usage}_{CT}(X)}{\sum_{Y \in CT} \mathit{usage}_{CT}(Y)},$$

where $usage_{CT}(X)$ denotes the number of transactions that use X for their encoding. So, the usage of an itemset $X \in CT$ is equal to the frequency of the code $code(X)$ in the encoded database. Now, the size of a database $\mathcal{D}$ compressed by a code table CT can be computed as

$$\begin{aligned} L(\mathcal{D}|CT) &= \sum_{t \in \mathcal{D}} \sum_{X \in cover(t)} L(code_{CT}(X)) \\ &= -\sum_{X \in CS} usage_{CT}(X) \cdot \log(P(X)). \end{aligned}$$

The latter formulation as a summation over the coding set allows a faster and more incremental way of computation. Since the code table contains assumably much less sets than transactions in the database exist, this sum is computed with low expenses if the usage function is computed capably.

A code table CT is represented by means of the *standard code table* that provides codes for singleton itemsets. The items in the coding set of CT are represented by their codes from the standard code table ST, such that the description length of a code table is calculated as

$$\begin{aligned} L(CT) &= \sum_{X \in CS} \left(L(code_{CT}(X)) + \sum_{x \in X} L(code_{ST}(x)) \right) \\ &= -\sum_{X \in CS} \left(\log\left(P(X)\right) + \sum_{x \in X} \log\left(sup(\{x\})\right) \right). \end{aligned}$$

Applying the MDL principle, the total compression size is calculated as the sum of the description sizes $L(\mathcal{D}|CT) + L(CT)$. We observe, that the compression size depends mainly on the usage function. Thus, an understanding of usage dependencies is likely to be a crucial point for all Krimp-related algorithms.

2.4 The Algorithm Krimp

Krimp (Alg. 1) has as input a database $\mathcal{D}$ and the frequent patterns of a preferably low minimum support $\mathcal{F}$. The frequent patterns are sorted in *standard candidate order* that is first decreasing on support, second decreasing on cardinality and at last lexicographically (line 2). The code table is initialized to the standard code table (line 3) and each candidate pattern is regarded in the specified order. A candidate is added to the code table if this reduces the compression size (lines 4-9). Since the compression size decreases monotonically in the number of regarded candidates, the best compression in this procedure is achieved if the minimum support is set to $\frac{1}{|\mathcal{D}|}$. This results however in an extremely large candidate set due to the pattern explosion. A speed-up of the usage calculation that is carried out for each of the frequent item sets would thus accelerate the whole process significantly.

Algorithm 1 Krimp [12].

```
 1: procedure KRIMP($\mathcal{D}, \mathcal{F}$)
 2:     $\mathcal{F} \leftarrow sort(\mathcal{F})$                ▷ in standard candidate order
 3:     $CT \leftarrow$ STANDARDCODETABLE($\mathcal{D}$)
 4:     for $fp \in \mathcal{F} \setminus \mathcal{I}$ do
 5:         $CT_c \leftarrow CT \cup fp$
 6:         if $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$ then
 7:             $CT \leftarrow CT_c$
 8:         end if
 9:     end for
10: end procedure
```

Computing the Usage. The usage calculation relies on the method StandardCover (Alg. 2). It is invoked for every transaction that contains the currently regarded candidate pattern. Assuming that the code table is sorted by a total order, the algorithm traverses the code table and selects the first pattern that is contained in the specified transaction (line 2). The used order for this procedure is called *standard cover order* that is first decreasing on cardinality, second decreasing on support and at last lexicographically. If the transaction is covered completely by elements of the code table, the algorithm stops (line 3), otherwise the procedure is called recursively for the uncovered part of the transaction (line 6). The identification of those transactions that contain a specified

Algorithm 2 Standard Cover [12].

```
1: procedure STANDARDCOVER($t, CT$)
2:     $X^* \leftarrow \min\{X | X \subseteq t \wedge X \in CT\}$
3:     if $t \setminus X^* \leftarrow \emptyset$ then
4:         return $\{X^*\}$
5:     else
6:         return $\{X^*\} \cup$ STANDARDCOVER($t \setminus X^*$)
7:     end if
8: end procedure
```

candidate pattern is not trivial, the implementation of transactions as bit vectors improves the process significantly, but the question arises if there exists some representation of the database that enables an identification of affected parts without a scan of the whole database.

3 SHrimp

With Shrimp we present a tree structure, called *SH-tree*, that reflects the encoded database and enables a faster computation of the usage function. The tree is similar to to the *FP-tree* introduced by Han et al. [4], except that nodes contain sets of items and not only single items. Each branch from the root to a leaf represents a transaction and provides the information about all possible and the current encoding for a given coding set. More precisely, the nodes fulfill the following properties.

Definition 1 (SH-tree). *Given a total order on itemsets $\preceq$, a SH-tree is a tree structure with the following properties*

1. *The root of the tree is labeled as null.*
2. *Each node $n \neq null$ is described by a pattern (n.pattern), a set of inactive items (n.inact), a counter (n.freq) and the pointers to its children (n.children) and the parent node (n.parent). $n.freq$ denotes the number of transactions represented by the branch from the root to that node.*
3. *For a node n and the parent node $n_p = n.parent \neq null$ it holds that*

$$n_p.pattern \preceq n.pattern$$

The meaning of the field *inact* requires a more detailed explanation.

Definition 2. *Let n be a node with $|n.pattern| \geq 2$ and let $anc(n)$ denote the ancestor nodes of n. For an item $x \in n.pattern$ it holds that*

$$x \in n.inact \Leftrightarrow \exists n_a \in anc(n) : n_a.inact = \emptyset \wedge x \in n_a.pattern.$$

Items of a node n occurring in the set n.inact are called inactive, otherwise active. Accordingly, nodes that have inactive items are called inactive, otherwise active.

Inactive nodes denote patterns that would be used for a transaction if some of their items were not already encoded by another pattern. The application of these nodes may reduce the complexity of the tree, because singletons that occur in inactive nodes must not be displayed explicitly, but it may also enlarge the complexity due to the reflection of redundant information. Inactive nodes are however integrated into the tree because they define the consequences of a pattern selection change and enable thereby a fast computation of usage impacts if a certain pattern is removed or added. To get now a fundamental understanding of the algorithm, we examine an example of a tree first.

Example 1 (SH-tree). Let $\mathcal{D}$ be the sample database displayed in Table 1, $\preceq$ the standard cover order and the coding set be given by the patterns $\{b, d, e\} \preceq \{a, c\} \preceq \{a, g\}$ besides of the singleton itemsets. The resulting cover sets of $\mathcal{D}$ computed by the standard cover function are displayed in the right column of Table 1. The corresponding tree representation of the database is shown in Fig. 1. We can see that each transaction is represented by a branch, every leaf is marked by the number of the equivalent transaction. Inactive nodes and items are greyed out.

TID	transaction	cover set
1	a, b, d, e, f, g	$\{b, d, e\}\{a, g\}\{f\}$
2	a, b, c, d, e, g	$\{b, d, e\}\{a, c\}\{g\}$
3	a, c, e, f	$\{a, c\}\{e\}\{f\}$
4	a, b, c, d, e, f	$\{b, d, e\}\{a, c\}\{f\}$
5	a, c, e, g, f	$\{a, c\}\{e\}\{g\}\{f\}$
6	a, b, d, e, g	$\{b, d, e\}\{a, g\}$

Table 1: A sample transactional database and the resp. cover sets.

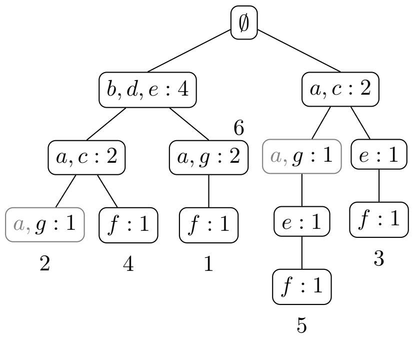

Fig. 1: The tree representation of the covered database.

The question arises how this structure can be utilized to compute the usage function after a pattern is integrated. For this occasion we further examine our running example.

Example 2 (Usage Computation). We imagine that the regarded candidate is the pattern $\{c, e, f\}$, and $\{b, d, e\} \preceq \{c, e, f\} \preceq \{a, c\} \preceq \{a, g\}$. The computation of emerging usage differences starts with an identification of branches that use the designated pattern for their encoding. The algorithm examines the smallest child of the root node, i.e. the node with the pattern $\{b, d, e\}$ first. Since b and e would be encoded by this child furthermore, the candidate pattern is not used in this sub tree. The algorithm proceeds with the right sub tree, finds that transactions 5 and 3 would use the candidate and stores the corresponding leaves with the singleton $\{e\}$. In these branches, the effects of the encoding by the candidate are calculated, i.e. $\{a, c\}$ is not used anymore, but the node with $\{a, g\}$ becomes active again. If the resulting usage function gives a better compression size, the pattern is integrated into the tree. The consequent tree is displayed in Fig. 2.

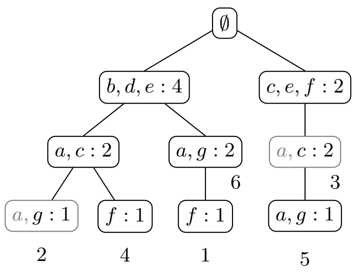

Fig. 2: The SH-tree of Example 1 when the pattern $\{c, e, f\}$ is inserted.

3.1 The Algorithm SHrimp

The procedure of SHrimp (Alg. 3) is similar to that of Krimp. The input set of frequent patterns is sorted (line 2) and the tree is initialized (line 3). Since the initial code table is the standard code table that contains only singleton patterns, the initial tree equals the FP-tree. For every candidate pattern, the transactions that would use the candidate are computed and stored by the corresponding leaves called *fpLeaves* (line 6) and the resulting usage function is computed (line 7). If the inclusion of the pattern improves the compression, the pattern is integrated into the tree (line 10).

Algorithm 3 SHrimp.

```
 1: procedure SHrimp($\mathcal{D}, \mathcal{F}$)
 2:     $sort(\mathcal{F})$                                    ▷ in Standard Candidate Order
 3:     $tree \leftarrow initTree(\mathcal{D})$
 4:     $usage \leftarrow \{(item, item.frequency) | item \in \mathcal{D}\}$
 5:     for $fp \in \mathcal{F} \setminus \mathcal{I}$ do
 6:         $fpLeaves \leftarrow$ UsingTransactions($fp, tree$)
 7:         $usage_c \leftarrow$ UsageIncluding($fp, fpLeaves, tree$)
 8:         if $L(usage_c) < L(usage)$ then
 9:             $usage \leftarrow usage_c$
10:             Insert($fp, fpLeaves, tree$)
11:         end if
12:     end for
13: end procedure
```

The method UsingTransactions($fp, tree$) (line 6) identifies the using transactions in a depth-first like search, exploiting the order of the tree as described in Example 2. For each of the returned leaves, the method DiffUsage (Alg. 4) is called. This procedure considers a branch as an ordered sequence of ancestor nodes of the specified leaf that are succeeding to the candidate pattern fp with regard to the standard cover order (line 4). The set *bounded* (line 5) collects all items that are covered now and the set $freed$ (line 6) those that are not covered anymore if fp is inserted. By means of these sets, it is checked for every node of the branch whether it would change its status of activity and the usage function is adapted accordingly (line 7-15).

The method Insert($fp, tree$) in Alg. 3 (line 10) creates the nodes of the candidate pattern and alters the tree accordingly. The integration of a node induces bounds and branches. Singletons that are covered by the inserted node are removed and a branch might appear more bound as in Example 2. Generally speaking, the concerning branch is divided into these transactions that contain the candidate pattern and the remaining ones. For this reason, the branch of transactions containing the specified pattern has to be split from the original one. The method Modifications($fp, fpLeaves$) (Alg. 5) computes thereby the arising structural changes that come with the insertion of fp. This algorithm

Algorithm 4 Computing the resulting usage differences concerning the inclusion of the pattern fp.

```
1: procedure UsageIncluding($fp, fpLeaves, tree$)
2:     $usage_c \leftarrow usage(tree)$
3:     for $fpLeaf \in fpLeaves$ do
4:         $branch \leftarrow \{n \in anc(leaf) | fp \prec n\}$        ▷ sorted in Standard Cover Order
5:         $bounded \leftarrow fp$
6:         $freed \leftarrow \emptyset$
7:         for $n \in branch$ do
8:             if $\emptyset = n.inact \wedge (bounded \cap n.pattern \neq \emptyset)$ then
9:                 $usage_c(n.pattern) \leftarrow usage_c(n.pattern) - u$
10:                $freed \leftarrow freed \cup n.pattern$
11:            else if $(\emptyset \neq n.inact \subseteq freed) \wedge (bounded \cap n.pattern = \emptyset)$ then
12:                $usage_c(n.pattern) \leftarrow usage_c(n.pattern) + u$
13:                $bounded \leftarrow bounded \cup n.pattern$
14:            end if
15:        end for
16:    end for
17:    return $usage_c$
18: end procedure
```

traverses the tree from the specified leaves up to the first node that is preceding to fp, the prospective parent node (lines 4-10). For each of the regarded nodes, the modifications of their fields in the branch with the integrated pattern are computed (line 6,7). Accordingly to the gained information, the nodes are created and the tree altered. Inactive nodes are integrated by the same procedure.

4 Experiments

The experiments are invoked as *RapidMiner*[1] processes, based on the JAVA programming language. The implemented operator Krimp is modelled after the original C++ implementation[2], i.e. transactions are represented as bit vectors, the code words are stored as a list of lists, that enables an immediate access to the insertion point of a candidate pattern, etc. The frequent patterns are mined by the available FP-Growth operator from RapidMiner. Due to storage limitations, the minimum support for the mined candidates differs in dependence of the dataset, which was mainly dependent on the performance of the FP-Growth algorithm. Table 2 shows the basic characteristics of the used datasets and the parameters for the candidate generation. The datasets were taken from the FIMI repository[3] and a collection of prepared UCI datasets[4].

[1] http://rapidminer.com
[2] http://www.patternsthatmatter.org/implementations/#krimp
[3] http://fimi.ua.ac.be/data
[4] https://dtai.cs.kuleuven.be/CP4IM/datasets

Algorithm 5 Computes the modifications of the tree resulting from the integration of the pattern fp.

```
1: procedure MODIFICATIONS($fp, fpLeaves$)
2:     $fpParents \leftarrow \emptyset$
3:     for $fpLeaf \in fpLeaves$ do
4:         $n \leftarrow fpLeaf$
5:         while $fp \prec n$ do
6:             $mod(n.freq) \leftarrow mod(n.freq) + fpLeaf.freq$
7:             $mod(n.parent.children) \leftarrow mod(n.parent.children) \cup n$
8:             $n \leftarrow n.parent$
9:         end while
10:        $fpParents \leftarrow fpParents \cup \{n\}$
11:    end for
12: end procedure
13: return mod
```

The runtime results, comparing Krimp and SHrimp, are displayed in Figure 3. We can see that SHrimp has an exceptionally better performance on the FIMI datasets (Mushroom, Chess and Connect). For these datasets, the minimum support was very hard to set, because a small decrease in the threshold resulted in an enormous growth of frequent patterns that caused the pattern generation process to crash due to an available memory capacity of about 100GiB. Regarding the Soybean dataset, the results are assimiable well. For about 20% of the first regarded candidates, SHrimp is slightly faster, but the trend reverses with time. For the rather small Tic-tac-toe and the Tumor dataset, Krimp is however eminently faster than SHrimp. It might be noted, that the amount of time spent on the insertion of patterns into the tree is negligible small in comparison to the time needed for the usage calculation of all candidates. Further insights into node dependencies and the usage calculation in general might thus improve the algorithm thoroughly.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{I}\|$	density	$minsup$	$\|\mathcal{F}\|$
Tic-tac-toe	958	27	33%	0	250985
Soybean	630	50	32%	0.01	2613499
Primary-tumor	336	31	48%	0.01	1290968
Mushroom	8124	119	18%	0.1	574431
Chess(k-k)	3196	75	50%	0.6	254944
Connect	67557	129	33%	0.9	27127

Table 2: Basic characteristics of examined datasets.

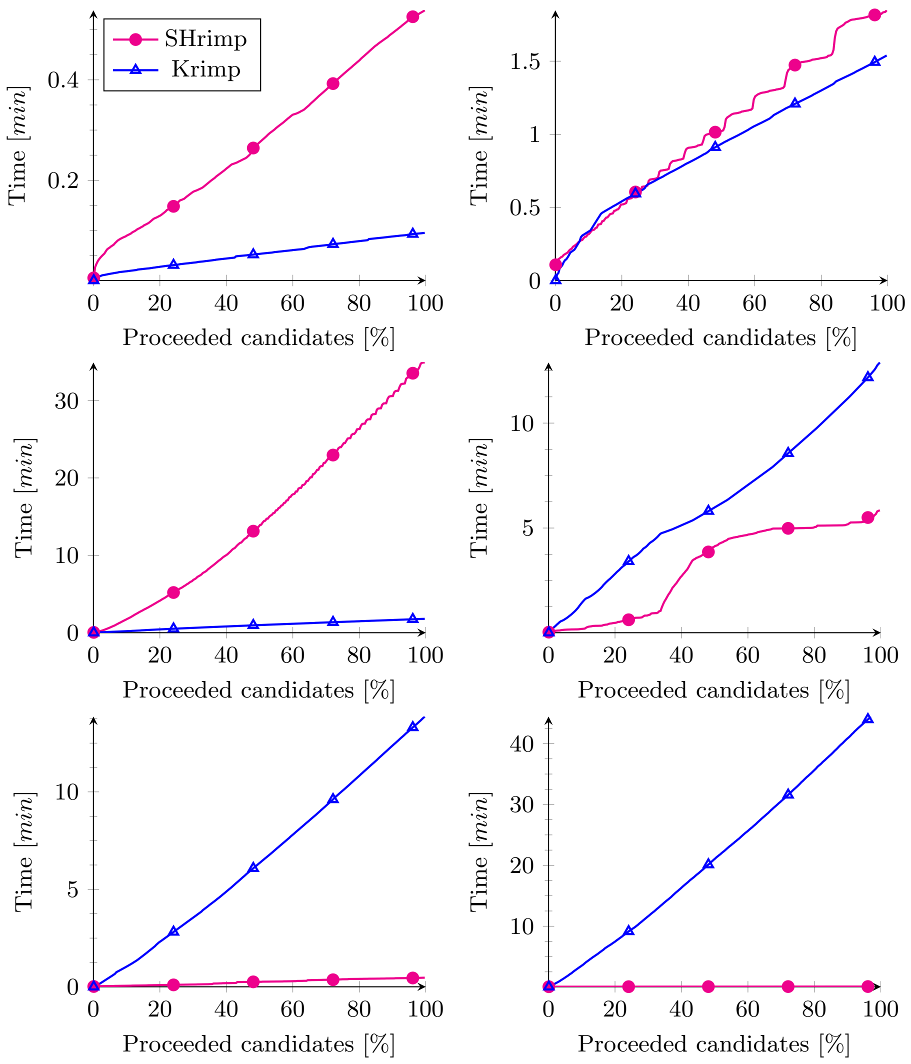

Fig. 3: Runtime for Krimp (blue triangle marks) and SHrimp (red square marks) in relation to the percentage of examined patterns for the Tic-tac-toe (left above), Soybean (right above), Primary-tumor (left middle), Mushroom (right middle), Chess (left below) and Connect (right below) dataset.

5 Conclusion

A first attempt to create a data structure that reflects the possibilities of summarization due to the inherent structure of the dataset has been substantiated with the development of SHrimp. A prototype implementation facilitates a much faster computation of influences on the compression size when the coding set changes, for at least some well known datasets. This data structure might be applied to other MDL approaches, e.g. SLIM, as well. The structure offers the possibility of a human readable understanding of the main characteristics of the dataset. We can think of an output that reveals only the treetop, e.g. all nodes up to a certain level, where the composition of the most priorized codes in relation to the standard cover order is displayed.

Further exploitations of the order of the code table, respectively the nodes, and a less redundant representation of codes that are subsets of other codes are likely to improve the results and are worth to be explored.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. SIGMOD Rec. 22(2), 207–216 (1993)
2. Cover, T., Thomas, J.: Elements of information theory. Wiley-Interscience (2006)
3. Grünwald, P.: Minimum Description Length Principle. MIT press, Cambridge, MA (2007)
4. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. 29(2), 1–12 (2000)
5. van Leeuwen, M., Bonchi, F., Sigurbjörnsson, B., Siebes, A.: Compressing tags to find interesting media groups. In: CIKM. pp. 1147–1156. ACM (2009)
6. Li, P.V.M.: An Introduction to Kolmogorov Complexity and Its Applications. Springer (1997)
7. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery (1997)
8. Rissanen, J.: Modeling by shortest data description. Automatica 14, 465–471 (1978)
9. Smets, K., Vreeken, J.: Slim: Directly mining descriptive patterns. In: SDM. pp. 236–247. SIAM / Omnipress (2012)
10. Vreeken, J., van Leeuwen, M., Siebes, A.: Characterising the difference. In: KDD. pp. 765–774. ACM (2007)
11. Vreeken, J., van Leeuwen, M., Siebes, A.: Preserving privacy through data generation. In: ICDM. pp. 685–690. IEEE Computer Society (2007)
12. Vreeken, J., van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Min. Knowl. Discov. 23, 169–214 (2011)