Item Sets That Compress

Arno Siebes Jilles Vreeken Matthijs van Leeuwen
Department of Computer Science
Universiteit Utrecht
{arno, jillesv, mleeuwen}@cs.uu.nl

Abstract

One of the major problems in frequent item set mining is the explosion of the number of results: it is difficult to find the most interesting frequent item sets. The cause of this explosion is that large sets of frequent item sets describe essentially the same set of transactions. In this paper we approach this problem using the MDL principle: *the best set of frequent item sets is that set that compresses the database best.* We introduce four heuristic algorithms for this task, and the experiments show that these algorithms give a dramatic reduction in the number of frequent item sets. Moreover, we show how our approach can be used to determine the best value for the *min-sup* threshold.

Keywords: *Frequent Item Sets, MDL*

1 Introduction

Frequent item set mining is one of the major innovations Data Mining has contributed to the broad area of Inductive Inference. It started as a phase in the discovery of association rules [2], but has been generalised independent of these to many other patterns. For example, frequent sequences [3], episodes [11], and frequent subgraphs [9].

The problem of frequent item set mining [2] can be described as follows. The basis is a set of items $\mathcal{I}$, e.g., the items for sale in a store. A transaction $t \in \mathcal{P}(\mathcal{I})$ is a set of items, e.g., representing the set of items a client bought at that store. A database over $\mathcal{I}$ is simply a set of transactions, e.g., the different sale transactions in the store on a given day. An item set $I \subset \mathcal{I}$ occurs in a transaction $t \in db$ iff $I \subseteq t$. The *support* of I in db, denoted by $supp_{db}(I)$ is the number of transactions in the database in which I occurs. The problem of frequent item set mining is: given a threshold *min-sup*, determine all item sets I such that $supp_{db}(I) \geq$ *min-sup*. These *frequent item sets* represent, e.g., sets of items customers buy together often enough.

There are many efficient algorithms to mine frequent item sets, e.g., [2, 8]. However, there is one major obstacle that precludes frequent item sets to be used successfully more often. The problem is the *number* of frequent item sets produced. If the threshold *min-sup* is put high, the resulting sets are well-known. If *min-sup* is put low the the number of frequent item sets explodes.

Much research has been devoted to reduce the resulting set of frequent item sets. For example, by using *interestingness measures* such as *lift* [12]. Or, more general by *constraints*, such as requiring that the resulting item sets are *closed* [15]. An item set I is closed if there exists no item set J such that $I \subset J$ and $supp_{db}(I) = supp_{db}(J)$.

The set of closed frequent item sets is a *lossless compression of the set of all frequent item sets.* That is, the set of all frequent item sets and their support can be reconstructed from the set of all closed frequent item sets and their support. There also exist approaches that define, or can be seen as, *lossy compression.* For example, *maximal* frequent item sets [4] and *boundary cover sets* [1]. While these two still allow to reconstruct all frequent item sets, their support information is lost. There are also lossy compression schemes that put a bound on the support of the "missing" item sets. For example, *δ-free sets* [5] and the recently introduced *compressed frequent pattern sets* [14]. These latter two, require an extra user-defined parameter δ.

In other words, many of these compression approaches focus on the complete set of frequent patterns. By choosing this focus, they do not address the reason of the explosion of frequent item sets: *large sets of frequent item sets essentially describe the same set of transactions.* Clearly, closed item sets partially solve this problem: if J and $J \cup \{I\}$ have the same support, they describe exactly the same set of transactions. Both δ-free sets and compressed frequent pattern sets do not require that the transactions are *exactly* the same, but at the cost of an extra, arbitrary parameter δ.

In this paper we propose a completely different approach. We take the cause of the explosion head on: a set of item sets is interesting iff it gives a good description of the database. That is, a set of item sets is interesting iff *it yields a good (lossless) compression of the database* rather than a good compression of the

set of all frequent item sets.

To determine whether or not a subset of the set of frequent item sets yields a good compression of the database, we use the *Minimum Description Length Principle (MDL)* [7]. The MDL principle gives us a fair way to balance the size of the compressed database and the size of the code table; note, we need both for a lossless compression of the database! If we use too few frequent item sets the database will hardly be compressed. If we use too many, the code table (for coding/decoding the database) will become too large.

The MDL approach gives a good way to determine the most interesting frequent item sets; the reduction is by four orders of magnitude as we show in this paper. This reduction is reached without any user defined threshold. Moreover, we can also use this approach for a related problem, viz., what is the best value for the threshold *min-sup*? We will briefly discuss this problem and a solution at the end of the paper.

While MDL removes the need for user defined parameters, it comes with its own problems: only heuristics, no guaranteed algorithms. However, our experiments show that these heuristics give already dramatic reduction in the number of item sets.

The remainder of this paper is organised as follows. In the next section we formally define our problems. In Section 3 we introduce heuristic algorithms that for a set of frequent item sets J determine that subset $C \subseteq J$ that gives the best compression of the database. Subsequently, in Section 4, we report on our experiments and discuss the results. In Section 5, we give a heuristic algorithm for the more general problem: find the set of frequent item sets that compress the database best. Moreover, we present some experimental results. The paper ends with Section 6 in which we formulate our conclusions and give some directions for further research.

2 Problem Statement

In this section we state our problem formally. First we briefly discuss the MDL principle. Next we show how sets of item sets can code a database and what the size of the coded database is. Then we introduce code tables and their size. With these ingredients, we formally state the problems studied in this paper.

2.1 MDL MDL (minimum description length) [7], like its close cousin MML (minimum message length) [13], is a practical version of Kolmogorov Complexity [10]. All three embrace the slogan *Induction by Compression.* For MDL, this principle can be roughly described as follows.

Given a set of models[1] $\mathcal{H}$, 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 exist of sets of item sets and coding tables. For, as explained in the introduction, we are looking for those item sets that together best (in the MDL sense) describe the database. Before we can formally state our problem, we first have to define both $\mathcal{H}$ and how a $H \in \mathcal{H}$ is used to describe the data.

2.2 Coding with Item Sets The first step in defining the description length of a database by a set of item sets is to define how sets of item sets can be used to code the database. Clearly, the coding item sets should at least be able to "describe" each transaction in the database. That is, each transaction should be the union of some of the coding item sets. Moreover, the coding item sets that *cover* a transaction should be mutually disjoint. This intuition is formalised in the definition of an item set cover.

Definition 2.1. *A set of item sets C is an* item set cover *for a database of transaction db iff for each $t \in db$ there is a subset $C(t) \subseteq C$ such that:*

1. $t = \bigcup_{c_i \in C(t)} c_i$
2. $\forall c_i, c_j \in C(t) : c_i \neq c_j \rightarrow c_i \cap c_j = \emptyset$

We say that $C(t)$ covers *t.*

To use an item set cover to encode the database, we need to know which set of items is to be used to cover a transaction $t \in db$. This is formalised as a coding scheme.

Definition 2.2. *A* coding scheme *CS for a database db is a pair (C, S) in which C is an item set cover of db and S is a function $S : db \rightarrow \mathcal{P}(C)$ such that $S(t)$ covers t.*

Each coding scheme is part of a possible model. For a given coding scheme we know exactly how to cover each transaction in the database. To use our coding scheme to actually encode the database, we need to assign a

[1] MDL-theorists tend to talk about *hypothesis* in this context, hence the $\mathcal{H}$; see [7] for the details.

code to each element of the item set cover. Naturally, the coding set that is used most often should get the shortest code. The number of times an item set is used by a coding scheme is called its frequency[2].

Definition 2.3. *The* frequency *of an element $c \in C$ for a given coding scheme (C, S) of a database db is defined by:*

$$freq(c) = |\{t \in db | c \in S(t)\}|$$

Now we can assign codes to all elements of C for some coding scheme (C, S). However, we are not so much interested in the actual code as in the size of the compressed database. In other words, we are only interested in the *length* of the code for each $c \in C$. Fortunately, there is a nice correspondence between codes and probability distributions (see, e.g., [10])

Theorem 2.1. *Let P be a distribution on some finite set D, there exists a (unique) code CD on D such that the length the code for $d \in D$, denoted by $L(d)$ is given by*

$$L(d) = -\log(P(d))$$

Moreover, this code is optimal in the sense that it gives the smallest expected code size for data sets drawn according to P.

The optimality property means that we introduce no bias using this code length. The probability distribution induced by a coding scheme is, of course, simply given by the relative frequency of each of the item sets. This intuition is formalised in the next definition.

Definition 2.4. *A coding scheme (C, S) induces a probability distribution P on C, called the* Coding Distribution *by:*

$$\forall c \in C : P(c) = \frac{freq(c)}{\sum_{d \in C} freq(d)}$$

With this probability distribution, each $c \in C$ receives a code with length $-\log(P(c))$. Each $t \in db$ is covered by $S(t)$, hence, its coded version is simply the concatenation of the codes for the $c \in S(t)$. In other words, the code length of t is:

$$L(t) = \sum_{c \in S(t)} L(c)$$

Summing up over all tuples yields our first result.

Lemma 2.1. *For a given coding scheme (C, S), the size of the coded database is given by*

$$L_{(C,S)}(db) = -\sum_{c \in C} freq(c) \log\left(\frac{freq(c)}{\sum_{d \in C} freq(d)}\right)$$

[2]So, in this paper the frequency and the support of an item set are two completely different things!

2.3 Code Tables With the previous lemma, we can compute $L(D|H)$. To use the MDL principle, we still need to know what $L(H)$ is. To answer this question, we first need to know what $\mathcal{H}$ actually is. That is, what are our models? We have already noted that the coding schemes are part of our models. How do we describe coding schemes?

A coding scheme consists of a set of item sets together with a function that tells which item sets are used in the encoding of which transaction. Now all functions can be written as a set of (x, y) pairs (clearly, not necessarily the most efficient representation, we'll come back to this issue). In other words, a coding scheme can be seen as a table in which each row consists of an item set and a list of transactions in which it is used for encoding.

While such tables describe the coding abstractly, they are not sufficient to actually code and decode the database. For this, we have to add one more column containing the actual code used for a given item set. Of course, the length of the code used should correspond with the frequency of the item set in the coding scheme. Formally, such tables are defined as follows.

Definition 2.5. *For a database of transactions db over item set $\mathcal{I}$, a* coding/decoding table *is a table with three columns,* Item Set, Code *and* Tuple List *such that*

- *The columns* Item Set *and* Tuple List *form a coding scheme.*
- *Let N_i be the number of elements in the* Tuple List *of* Item Set *i, then the* length *of the code for i should equal $-\log\left(\frac{N_i}{\sum_{j \in \text{Item Set}} N_j}\right)$*

As already noted above, the *Tuple List* is not necessarily the most efficient way to represent the mapping from item sets to tuples. Moreover, the number of bits necessary for this encoding may simply swamp the size of the remaining columns. For, if that database has n transactions, we need $\log(n)$ bits to encode the pointer to each transaction. Which implies that $n \log(n)$ bits as a lower bound on the number of bits necessary to encode the mapping. In practice, where we expect that each transaction is covered by a few item sets and each item set is used to cover many transactions, the actual number of bits may be far higher.

An alternative representation is by an algorithm described in some programming language. Given that we assume that we have a very large database, the choice for a particular programming language is immaterial as usual in Kolmogorov Complexity [10].

Both alternatives share one big disadvantage. We need not only find the set of item sets to compress the

database best, we also need to find the mapping that gives this best compression. A wrong mapping may give a very bad compression rate. In the general case, finding the shortest program for a given task is an undecidable problem, more precise, it is upper semi-computable [10]. If for a given set of item sets we want to find the best mapping, we have a finite setting. Hence the problem is trivially decidable. However, there are no known efficient algorithms for the finite case.

Because of these reasons, we simplify our problem. We take a fixed algorithm to code the database. We are going to assume an *order* on a code table; a table containing pairs of item sets and codes. To code a transaction t, we simply take the first item set i in the code table that is a subset of t. We continue this procedure recursively on $t \setminus i$ until the remainder is empty. This simple procedure induces a coding/decoding table from any set of item sets that can cover all transactions in *db*. To show this, we first need the notion of a *coding set*:

Definition 2.6. *An ordered set of item sets that contains all the singleton item sets $\{I\}$ for $I \in \mathcal{I}$ is called a* coding set. *The order, called the* coding order *is denoted by* $<$.

Given a coding set, we can compute a cover for a transaction as sketched above. The **Cover** algorithm given in figure 1 gives a more formal specification. Its parameters are a Coding set C and a transaction t, the result is a set of elements of C that cover t. **Cover** is well-defined function on any coding set and

```
Cover(C, t)
1  S := smallest element c of C in coding order
   for which c ⊆ t
2  if t \ S = ∅
3      then Res := {S}
4      else Res:= {S} ∪ Cover(C, t \ S)
5  return Res
```

Figure 1: The Cover Algorithm

any transaction over $\mathcal{I}$. Moreover, it defines a coding scheme over any database of such transactions.

Lemma 2.2. *Let C be any coding set, db a database of transactions over $\mathcal{I}$ and let $t \in db$, then:*

1. **Cover**(C, t) *is a valid cover of* t.
2. *C and* **Cover** *induce a unique coding scheme on db.*

Proof. 1. A coding set contains all singleton item sets, and, hence, **Cover** can compute a cover for each possible transaction.

2. Construct a table of pairs (c, l), one for each $c \in C$. Such that l is a list of transactions $t \in db$ for which $c \in$ **Cover**(C, t).

With the right set of codes, the code scheme induced by C and **Cover** can even be extended to a coding/decoding table. To formalise this, we first need the notion of a code table.

Definition 2.7. *Let C be a coding set, K a set of codes and db a database. $CT \subseteq C \times K$ is a* code table *for db iff*

- $\forall (c_1, k_1), (c_2, k_2) \in CT : c_1 = c_2 \rightarrow k_1 = k_2$
- $\forall (c, k) \in CT : L(k) = -\log(P(c))$ *where $P(c)$ is the probability of c in the coding scheme for db induced by C and* **Cover**.

Now we can formalise our intuition:

Theorem 2.2. *Let C be a coding set and db a database, then*

1. *C and* **Cover** *induce a unique code table*
2. *This code table can be extended to a coding/decoding table.*

Proof. 1. Lemma 2.2 gives a unique coding scheme. From Theorem 2.1 get the existence of the necessary code H. Together this gives the required code table.

2. The code table can be extended to a coding/decoding table using the lists of transactions constructed in, again. Lemma 2.2.

Now we can state what our models are: they are the code tables induced by coding sets. We can use **Cover** and table look-up to code the transactions in a database and simply table look-up to decode a coded transaction.

2.4 The Size of a Code Table Now that we know what our collection of models $\mathcal{H}$ is, we can answer the question: what is $L(H)$ for an $H \in \mathcal{H}$? In part this is easy, as for the right hand side of a code table we already know the length. It is $-\log(P(c))$ where $P(c)$ is the probability of the item set in the left hand side. So, the remaining question is: how do we encode the item sets on the left hand side and what is the length of this encoding?

This may look like a pointless question: the right hand side is already a code for the left hand side, why

do we need another code? But it isn't pointless at all. The code for the item set on the right hand side is the code that is used to describe the database. The code on the left hand side describes the item set itself. This description can only be done in terms of $\mathcal{I}$. That is, in terms of the singleton item sets.

The set $\{I\}_{I \in \mathcal{I}}$ is a coding set. Hence, it induces an encoding. This is the encoding we will use. It allows us to reconstruct the database upto the names of the individual items. If necessary, one could add an ASCII table giving the names of the individual items. Since this table is the same for all code tables, it does not have any effect on the choice for the best possible encoding. Therefore, we do not consider the table with names in this paper.

Definition 2.8. *The* standard encoding *of an item set i for a given database db over $\mathcal{I}$ is the encoding induced by the coding set $\{I\}_{I \in \mathcal{I}}$.*

With this definition, we can compute the size of the code table $CT \subseteq C \times K$. The length of the first element of a pair $(c, code(c)) \in CT)$ is the length of the standard encoding of c, denoted by $L_{st}(c)$. While the length of $code(c)$, is the length of the encoding induced by the coding set C. This latter length is denoted by $L_C(c)$ Summing up over C, we have

Lemma 2.3. *The size of the coding table induced by a coding set C for a database db is given by*

$$L(CT_C) = \sum_{c \in C : freq(c) \neq 0} (L_{st}(c) + L_C(c))$$

Note that we do not take item sets with zero frequency into account. Such item sets are not used to code.

2.5 The Problem Let C be a coding set and db a database. Denote by (C, S_C) the coding scheme induced by C and by CT_C the coding table induced by C. The total size of the encoded version of db is then given by

$$L_C(db) = L_{(C,S_C)}(db) + L(CT_C)$$

Lemma's 2.1 and 2.3 tell how the two sizes on the right hand side can be computed.

With this remark, we can finally give our formal problem statement:

Minimal Coding Set Problem:

For a database db over a set of items $\mathcal{I}$, find a coding set C for which $L_C(db)$ is minimal.

A minimal coding set is a set of item sets that gives the best compression of the database db. Before we consider this general problem, we will consider a simpler problem, one in which we are given a set of item sets J. The problem is to find a subset of J which leads to a minimal encoding; where minimal pertains to all possible subsets of J. To make sure that this is possible, J should contain the singleton item sets. We will call such a set, a proto coding set

Minimal Coding Subset Problem:

For a database db over a set of items $\mathcal{I}$ and J a proto coding set, find a coding set $C(J) \subseteq J$, for which $L_{C(J)}(db)$ is minimal.

Note that $C \subseteq J$ in this context means that C is a subset of J if we forget the coding order associated with C.

A solution for the coding subset problem allows us to find the "best" (closed) frequent item sets for a given minimal support.

2.6 How Hard is the Problem? The number of coding sets does not depend on the actual database. Because of this, we can compute the size of our search space rather easily.

A coding set consists of a set of item sets that contains the singleton item sets plus an order. That is, a coding set is based on a set that contains the singleton item sets plus an almost arbitrary subset of $\mathcal{P}(\mathcal{I})$ Almost, since we are not allowed to choose the $|\mathcal{I}|$ singleton item sets. In other words, there are

$$\binom{2^{|\mathcal{I}|} - |\mathcal{I}| - 1}{j}$$

such sets with $j + |\mathcal{I}|$ elements. Since a set with n elements admits $n!$ orders, we have:

Lemma 2.4. *For a set of items $\mathcal{I}$, the number of coding sets is given by:*

$$NCS(\mathcal{I}) = \sum_{j=0}^{2^{|\mathcal{I}|}-|\mathcal{I}|-1} \binom{2^{|\mathcal{I}|} - |\mathcal{I}| - 1}{j} \times (j + |\mathcal{I}|)!$$

So, even for a rather small set $\mathcal{I}$ we already have a huge search space. Table 1, gives an approximation of NCS for the first few sizes. Clearly, the search space is far too large to consider exhaustive search. There is some structure in the search space, given in Lemma 2.5.

$\lvert\mathcal{I}\rvert$	1	2	3
$NCS(\mathcal{I})$	1	8	8742
$\lvert\mathcal{I}\rvert$	4	5	6
$NCS(\mathcal{I})$	2.70×10^{12}	1.90×10^{34}	4.90×10^{87}

Table 1: The number of coding sets

LEMMA 2.5. *Let J_1 and J_2 be two proto coding sets such that $J_1 \subset J_2$, then*

$$L_{C(J_1)}(db) \geq L_{C(J_2)}(db)$$

Proof. Any coding set in J_1 is also a coding set in J_2.

This lemma doesn't suggest a pruning strategy for the search space. Therefore, we use heuristic algorithms in this paper. However, it is instrumental in defining a strategy to answer the **Minimal Coding Set Problem** using the results for the **Minimal Coding Subset Problem**. To exploit this opportunity, we first introduce and test algorithms for the latter problem.

3 Discovering Item sets That Compress

In this section, we introduce four algorithms for the **Minimal Coding Subset Problem**. As noted before, all these algorithms are heuristic. In fact, they are all based on the same simple greedy strategy. This strategy is first introduced and then two strategies for improvement are discussed and exploited.

3.1 The Basic Heuristic The basic heuristic we employ is a simple greedy strategy:

- Start with the code consisting only of the singleton item sets.
- Add the other item sets one by one. If the resulting codes leads to a better compression, keep it. Otherwise discard that item set.

To turn this sketch into an algorithm, some questions have to be answered. Firstly, in which order do we add item sets? Secondly, where does a new item set fit in the order of the coding set? Phrased differently, what is its place in the code table? Finally, when do we say that the compression is better? Or, do we *prune* the newly constructed coding set before we check whether it compresses better or not?

Before we discuss each of this questions, we briefly describe the initial encoding. This is, of course, the standard encoding defined in definition 2.8. For this, we need to construct a coding set form the elements of $\mathcal{I}$. The algorithm called **Standard**, given in figure 2, returns such a coding set.

It takes a set of items and a database as parameters and returns a coding set. Note that the actual order in which we put the coding set is immaterial. As each coding set will give a code with exactly the same length to each of the items. This is formalised in lemma 3.1.

LEMMA 3.1. *Let $\mathcal{I}$ be a set of items and db a database of transactions over $\mathcal{I}$. Moreover, let J and K be two arbitrary code sets containing exactly the elements of $\mathcal{I}$, then*

$$L_J(db) = L_K(db)$$

Proof. The singleton item sets are necessarily disjoint. This means that in whatever order is defined on the set of singletons,

$$P(I) = \frac{supp_{db}(I)}{\sum_{L \in \mathcal{I}} supp_{db}(L)}$$

where $supp_{db}(I)$ denotes the support of I in db.

In other words, **Standard** is vacuously correct. The reason that we prefer this particular order is that it fits with order we define in the next subsection.

Standard$(\mathcal{I}, db)$
1 **foreach** $I \in \mathcal{I}$
2 $\quad freq(I)$:= support of I in db
3 $Res := \mathcal{I}$ ordered descendingly on $freq$
4 **return** Res

Figure 2: The Standard Algorithm

3.2 Order in the Set! We have two questions on the order of item sets. Firstly in which order do we check whether they ought to be in our code table or not? Secondly, where should they be placed in the code table?

To a large extend, the answer to the first question is immaterial. As long as we try all item sets and put them in the (unknown!) correct place in the code table, the order in which we do this doesn't matter at all. It is only the final code table that counts. However, there are two reasons to take the question seriously. Firstly, the order in which we consider the item sets may make it easier or more difficult to add them to the code table. Secondly, a more prosaic reason is that our algorithm will need a definite order; random choice doesn't seem the wisest of ideas.

Given a coding set CS, it is in principle easy to determine the best spot for each item set in CS. Simply try all possible permutations. However, the $|CS|!$ possibilities make this a not too practical approach. Fortunately, there is a simple lemma that suggests a (partial) solution.

LEMMA 3.2. *Let J be an item set in coding set CS. Moreover, let $\{J_1, \ldots, J_k\}$ also be item sets in CS, such that the J_i form a cover of J. If J occurs in CS after all the J_i, the frequency of J will be zero.*

Proof. **Cover** will find a cover for each transaction t that could be covered by J before it even considers J. Hence, no transaction will get J in its cover.

If we denote the number of items in an item set J by $size(J)$, lemma 3.2 suggests that we order the item sets in CS by size. In this order, item sets with a larger size will come before item sets with a smaller size. In other words, we have a stratified code set. But, what order should we choose if two item sets have the same size?

So, we have two distinct $J_1, J_2 \in CS$ with $size(J_1) = size(J_2)$. When does the order of J_1 and J_2 in the code table matter? Answer: if there is a transaction in the database for which both J_i could be part of the cover. That is, transactions that could be covered by $J_1 \cup J_2$.

In other words, if there is an item set $J \in CS$ such that $J_1 \cup J_2 \subseteq J$, the relative order of the J_i is unimportant. If there is no such J in CS (in particular, $J_1 \cup J_2 \notin CS$), this means that the number of transactions that could be covered by $J_1 \cup J_2$ is too small to be of importance. That is, either $J_1 \cup J_2$ (or its closure) had too small a support to meet the threshold. Or, it has been *pruned* (see section 3.4 for details).

In this case, a choice for either of the J_i as the smaller in the order will diminish the frequency of the other. Since high frequencies lead to small codes (and thus well compressed databases), the a priori best choice is to give preference to the J_i with the highest support in the database.

Concluding, we sort by size and then by support. This is only a heuristic. But, given the observations above, it seems a reasonable heuristic. Any better heuristic will require far more computational effort. This order is our *standard order*.

DEFINITION 3.1. *Let db be a set of transactions over a set of items $\mathcal{I}$. Moreover, let $C \in \mathcal{P}(\mathcal{I})$ be an ordered set of item sets. C is in* standard order *for db iff for any two $J_1, J_2 \in C$*

- $size(J_1) \leq size(J_2) \rightarrow J_2 \preceq_C J_1$;
- $size(J_1) = size(J_2) \wedge supp_{db}(J_1) \leq supp_{db}(J_2)) \rightarrow J_2 \preceq_C J_1$.

While we use the standard order for the code table, we are greedy in which item set to use next: the one with the highest cover in the database. In the *cover order*, the item set with the largest cover is maximal. For a set of item sets J and a database db, **Cover-Order**(J, db) returns the version of J with this order.

3.3 Naive Compression We now have the ingredients for a naive compression algorithm:

- Start with the code consisting only of the singleton item sets.
- Add the other item sets one by one. Each time, take the item set that is maximal with regard to the cover order. If the resulting codes leads to a better compression, keep it. Otherwise discard that item set.

This intuition is formalised as the **Naive-Compression** algorithm specified in figure 3

Naive-Compression$(\mathcal{I}, J, db)$

1 $CodeSet :=$ **Standard**$(\mathcal{I}, db)$
2 $J := J \setminus \mathcal{I}$
3 $CanItems :=$ **Cover-Order**(J, db)
4 **while** $CanItems \neq \emptyset$ **do**
5 $\quad cand :=$ maximal element of $CanItems$
6 $\quad CanItems := CanItems \setminus \{cand\}$
7 $\quad CanCodeSet := CodeSet \oplus \{cand\}$
8 $\quad$ **if** $L_{CanCodeSet}(db) < L_{CodeSet}(db)$
9 $\quad\quad$ **then** $CodeSet := CanCodeSet$
10 **return** $CodeSet$

Figure 3: Naive Compression

Naive-Compression takes as input the set of items $\mathcal{I}$, the database db, and the code set J out of which the subset should be found that leads to the shortest encoding. The result is the best code set the heuristic algorithm has seen. "Maximal" in line 5 means maximal with regard to the cover order according to which $CanItems$ is ordered. The symbol $\oplus$ in line 7 signifies that $cand$ is put into its proper position according to the standard order.

Now, it may seem that each iteration of **Naive Compression** can only lessen the frequency of an item set. For, if $J_1 \cap J_2 \neq \emptyset$ and J_2 is added before J_1 in a coding scheme, the frequency of J_1 will go down (provided the support of J_2 does not equal zero).

While this is true, it is not the whole story. Because, what happens if we now add an item set J_3 before J_2 such that:

- $J_1 \cap J_3 = \emptyset$ and
- $J_2 \cap J_3 \neq \emptyset$.

The frequency of J_2 will go down, while the frequency of J_1 will go up again; by the same amount, actually.

So, even item sets with frequency zero can not be removed from the code set. However, since they are not used in the actual coding, they are not taken into account while computing the total compressed size for the current solution.

Only in the end, item sets with frequency zero should be removed. After all, they do not code, so they are not part of the optimal answer.

A useful observation about **Naive-Compression** is that the average frequency of item sets is getting smaller all the time.

3.4 Pruning the Code Set What about item set J with a small frequency, say 1? They have a very small probability and thus a code with a very high length. Such short codes may make better code sets unreachable. Consider, e.g., the following three code sets:

- $CS_1 = \{\{I_1, I_2\}, \{I_1\}, \{I_2\}, \{I_3\}\}$
- $CS_2 = \{\{I_1, I_2, I_3\}, \{I_1, I_2\}, \{I_1\}, \{I_2\}, \{I_3\}\}$
- $CS_3 = \{\{I_1, I_2, I_3\}, \{I_1\}, \{I_2\}, \{I_3\}\}$

Assume that $supp(\{I_1, I_2, I_3\}) = supp(\{I_1, I_2\}) - 1$. It is very well possible that $L_{CS_2}(db) > L_{CS_1}(db)$ while at the same time $L_{CS_3}(db) < L_{CS_1}(db)$. However, given these facts, **Naive-Compression** will never consider CS_3.

To alleviate this problem, we can *prune* $CanCodeSet$. Pruning means that we remove an element from $CanCodeSet$ and check if the resulting compression is better than that of $CodeSet$ itself. Clearly, we only have to check those item sets whose cover has become smaller. Since there may be more such item sets, the question is: which one do we prune out first? In line with our standard order philosophy, we take the one with the smallest cover. This is formalised in Algorithm **Prune-on-the-Fly** specified in figure 5. Note that the $\ominus$ in line 7 simply signifies that we remove *cand* from the (ordered) candidate set. The details on how this procedure is used in the search for the best compression is discussed in Section 3.6

3.5 Noise in the Database? In Section 3.4 we discussed pruning item sets from the code set that cover just a few transactions. Form another perspective, if a transaction needs an “exotic” item set in its cover, perhaps this transaction itself is simply *noise.* That is, a transaction that doesn’t fit the regularity exhibited by the rest of the database.

If we simply *remove* such freak transactions from the database, the regularity patterns become far more visible. Because it also implies that the very infrequent item sets in a code set will be removed “automatically”. To implement this intuition, we need to answer four questions:

Prune-on-the-fly$(CanCodeSet, CodeSet, db)$

1 $PruneSet :=$
  $\{J \in CodeSet | cover_{CanCodeSet}(J) < cover_{CodeSet}(J)\}$
2 $PruneSet :=$ **Standard**$(PruneSet, db)$
3 **while** $PruneSet \neq \emptyset$ **do**
4   $cand :=$ element of $PruneSet$ with minimal cover
5   $PruneSet := PruneSet \setminus \{cand\}$
6   $PosCodeSet := CodeSet \ominus \{cand\}$
7   **if** $L_{PosCodeSet}(db) < L_{CanCodeSet}(db)$
8   **then** $CanCodeSet := PosCodeSet$
9 **return** $CanCodeSet$

Figure 4: Prune before acceptance testing

1. How do we decide a transaction is a freak transaction?
2. How do we encode freak transactions? For, after all, we should encode the complete database. Otherwise, the best compression is simply: remove all transactions.
3. When do we remove the freak transactions?
4. What is the size of the compressed database if we remove freak transactions?

Deciding which transactions are freak transactions, is not too complicated. Our goal is to compress the database. While compressing the database, it is very well possible that the code for some transactions becomes longer than its standard code. We would be far better off, if we would encode such transactions with their standard code:

DEFINITION 3.2. *Let CS be a code scheme for database db over the set of items $\mathcal{I}$. Moreover, let S be the standard code for db induced by $\mathcal{I}$. A transaction $t \in db$ is a* freak transaction *for CS iff:*

$$L_S(t) < L_{CS}(t)$$

The freak transactions are removed by three small algorithms. **Noise** determines the freak transactions in the database. **Denoise** removes the freak from the database. While **Sanitize** removes the item sets with a zero cover in the denoised database.

The third question was when can we remove the noise? Doing it while growing our code set is not a good

Noise$(\mathcal{I}, CodeSet, db)$

```
Noise := ∅
foreach t ∈ db do
    if L_S(t) < L_CS(t) then
        Noise := Noise ∪ {t}
return Noise
```

Denoise$(\mathcal{I}, CodeSet, db)$

```
Noise := Noise(I, CodeSet, db)
db := db \ Noise
return db
```

Sanitize$(\mathcal{I}, CodeSet, db)$

```
db := Denoise(I, CodeSet, db)
foreach J ∈ CodeSet do
    if cover_db(J) = ∅ then
        CodeSet := CodeSet ⊖ J
return CodeSet
```

Figure 5: Removing the Noise

idea. For, the code size of a transaction may shrink if a new item set is added to the code set. Hence, removing the noise should only be done as a post-processing step.

Finally, what is the size of the compressed database if we remove freak transactions? The only reasonable size is to compute the size of the sanitized code set, the size of the denoised database under this sanitized code set, the size of the *Noise* set and add these three. The encoding of the *Noise* set is straightforward. It is coded by the standard encoding:

$$L_{CS}(Noise) = \sum_{t \in Noise} L_{standard}(t)$$

With this, we can define our generalised compressed database size.

Definition 3.3. *Let CS be a coding set for a database of transactions db over a set of items $\mathcal{I}$. The generalised length of db coded with CS is given by:*

$$\begin{aligned} LN_{CS}(db) = &\ L_{CS}(\mathbf{Denoise}(\mathcal{I}, CS, db)) \\ &+ L_{CS}(\mathbf{Sanitize}(\mathcal{I}, CS, db)) \\ &+ L_{CS}(\mathbf{Noise}(\mathcal{I}, CS, db)) \end{aligned}$$

An important observation is that this defines a proper extension of our earlier definition of length.

Lemma 3.3. *Let CS be a coding set for a database of transactions db over a set of items $\mathcal{I}$. If* **Noise**$(\mathcal{I}, CodeSet, db) = \emptyset$*, then*

$$LN_{CS}(db) = L_{CS}(db)$$

Proof. If **Noise**$(\mathcal{I}, CS, db) = \emptyset$, then

- **Denoise**$(\mathcal{I}, CS, db) = db$
- **Sanitize**$(\mathcal{I}, CS, db) = CS$
- $L_{CS}(\mathbf{Noise}(\mathcal{I}, CS, db)) = 0$

Hence, it makes perfect sense to compare the two different lengths. In fact, if it is clear from the context, we'll blur the distinction and simply write $L_{CS}(db)$ to denote either of the two.

3.6 Alternatives for Better Compression.

With **Prune-on-the-fly** specified in Section 3.4 and the set of algorithms **Noise**, **Denoise**, and **Sanitize** specified in Section 3.5 we have a number of opportunities to try and improve the performance of our naive algorithm **Naive-Compression** from Section 3.3.

First of all, we can extend **Naive-Compression** by **Prune-on-the-fly**, this algorithm is called **Compress-and-Prune**. Secondly, we can extend **Naive-Compression** using **Sanitize** as a post-processing step, this algorithm is called **Compress-and-Sanitize**. Finally, we can extend **Naive-Compression** by both methods, this algorithm is called **All-out-Compression**. All three algorithms are given in figure 6. Note that **Compress-and-Prune**, and thus **All-out-Compression**, could perform worse than **Naive-Compression**, because the intermediate pruning steps could have detrimental effects later on in the search. Only **Compress-and-Sanitize** is guaranteed to do no worse. For, if **Sanitize** doesn't produce a better code, it simply does nothing. Only experiments can indicate which approach is the best.

4 Experiments

In this section we present and discuss our experimental results. First we give the set-up of the experiments. Then we give our results. Finally, we discuss these results.

4.1 The Set-Up

The main thing we want to test is, of course, do these algorithms work? That is, will they reduce the number of item sets? And, if so, by how much? Given that closed item sets are a well known compression of item sets and that there are often far fewer closed frequent item sets than frequent item sets, we also want to compare with the number of closed frequent item sets.

Compress-and-Prune$(\mathcal{I}, J, db)$

$CodeSet :=$ **Standard**$(\mathcal{I}, db)$
$J := J \setminus \mathcal{I}$
$CanItems :=$ **Cover-Order**(J, db)
while $CanItems \neq \emptyset$ **do**
  $cand :=$ maximal element of $CanItems$
  $CanItems := CanItems \setminus \{cand\}$
  $CanCodeSet := CodeSet \oplus \{cand\}$
  if $L_{CanCodeSet}(db) < L_{CodeSet}(db)$ **then**
    $CanCodeSet :=$
    Prune-on-the-fly$(CanCodeSet, CodeSet, db)$
    $CodeSet := CanCodeSet$
return $CodeSet$

Compress-and-Sanitize$(\mathcal{I}, J, db)$

$Result :=$ **Naive-Compression**$(\mathcal{I}, J, db)$
$Result :=$ **Sanitize**$(\mathcal{I}, Result, db)$
return $Result$

All-out-Compression$(\mathcal{I}, J, db)$

$Result :=$ **Compress-and-Prune**$(\mathcal{I}, J, db)$
$Result :=$ **Sanitize**$(\mathcal{I}, Result, db)$
return $Result$

Figure 6: Compression with Pruning and Sanitization

In fact, because the number of closed frequent item sets is far smaller it is also interesting to see whether our methods can compress the set of closed frequent item sets and, by how much. In the same vein, it is interesting to see which of the two inputs for the same *min-sup*, all frequent or all closed frequent, yields the better results.

Moreover, in both cases we want to see whether our optimisation schemes work. If they do, we also want to know which of the three optimised algorithms works best.

To do these tests, we have taken two data sets from the FIMI web site [6], viz., the chess and the mushroom data sets. We have taken these two data sets because they differ very much in their ration of frequent item sets versus closed frequent item sets. The chess database has, at *min-sup* = 1500, 2 million frequent item sets versus 550,000 closed frequent item sets. The mushroom database has, at *min-sup* = 724, 945,000 frequent item sets versus only 7800 closed frequent item sets.

4.2 The Results

In table 2 we give the results of all four algorithms on both the closed and all frequent item sets for *min-sup* = 724 on the mushroom database.

The results on the closed frequent item sets are impressive, we end up with less than 2% of the closed frequent item sets. Only 147 respectively 149 closed item sets are necessary to capture the structure in the mushroom database! This small subset allows us to compress the database by over 50%; hence, it certainly captures the structure. There is no a posteriori noise to remove, but pruning removes 2 closed frequent sets at the cost of 1 bit.

The results for all frequent item sets is even more impressive, The set of 945,000 has been reduced to 338 respectively 282 frequent sets; this is four orders of magnitude! This small subset gives a much better compression than the one constructed from closed frequent item sets. There seems to be structure in the database that is not captured by closed sets. That is, focusing on closed sets only may make you miss important frequent sets.

The chess database gives very much similar results. Uncompressed it is 605739 bits, **Compress-and-Prune** compresses it in total to 304034 bits using only 186 of the 2076366 available frequent item sets with *min-sup* = 1500; again 4 orders of magnitude reduction. If we only look at the closed item sets, **Compress-and-Prune** uses 174 of these out of 549920 available, for a total compressed size of 329870. So, again there is some structure not completely captured by the closed item sets.

The main effects we see in these results also bear out in more extensive testing. In figure 7 we plot the total compression size of the chess database for various support levels for both all and only closed frequent item sets: both input sets give comparable results.

Figure 8 illustrates the effect of pruning in a similar fashion on the mushroom data set. The two lines are simply on top of each other. Clearly, the (necessary) scale has some influence on this, witness the numbers mentioned above, but the algorithms are pretty comparable.

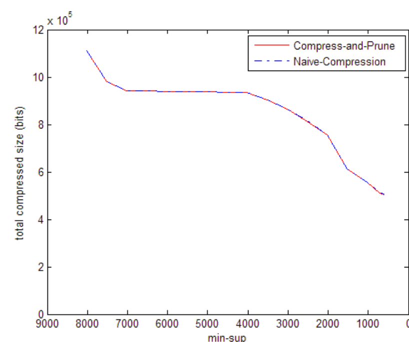

Figure 8: Naive vs Compress-and-Prune on Mushroom

Source	Algorithm	Coding Items	Code Table	Database	Noise	Total Compressed Size
All	**Naive**	338	9901	425977	0	435878
	Compress-and-Sanitize	338	9901	425977	0	435878
	Compress-and-Prune	282	8252	423487	0	431739
	All-out-Compression	282	8252	423487	0	431739
Closed	**Naive**	149	4754	507691	0	512445
	Compress-and-Sanitize	149	4754	507691	0	512445
	Compress-and-Prune	147	4631	507633	0	512246
	All-out-Compression	147	4631	507633	0	512246

Table 2: Algorithm results on Mushroom, *min-sup* = 724. The standard code size of the database is 1111287 bits.

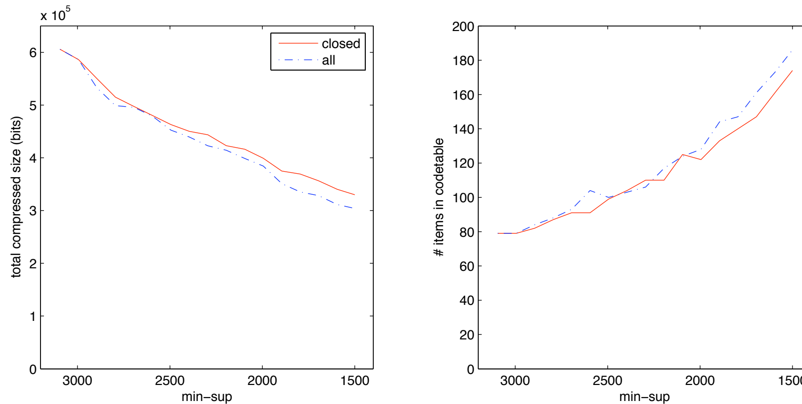

Figure 7: Closed or All on Chess

4.3 Discussion The experiments show that our MDL approach works. Just a few frequent item sets are necessary to give a succinct description of the database. The numbers mentioned are even more impressive if one realises that most of the singleton item sets are still present in the code table. In other words, there are even fewer "real" item sets to consider.

The pruning strategy works. This is not true for the noise removal. The compression doesn't find many freak transaction, none in the numbers presented here. We checked whether there were codes in the code table that had a code-length longer than standard. Again, we couldn't find any. This might still be an artifact of the databases we used. Further tests are necessary.

Based on the tests, it is clear that using all frequent item sets gives slightly better results in compression. On the other hand, using just closed ones gives far less item sets with comparable compression. Pruning can be used without much risk. Again, leaving less item sets in the result.

5 One Step Beyond

In this section we introduce an algorithm for the **Minimal Coding Set Problem**. In fact, it is an algorithm scheme rather than an algorithm.Using a particular instantiation, we give some experimental results.

5.1 What About Min-Sup The key difference between the **Minimal Coding Set Problem** and the **Minimal Coding Subset Problem** is that the latter only has to consider the item sets it gets as input, while the former has to consider all item sets. This may seem like a far more complex problem, however, lemma 2.5 gives an important clue. It shows that we can use a level-wise like search approach. The stratification is such that $Level_{i+1}$ is constructed using a lower *min-sup* than $Level_i$ Then. lemma 2.5 tells us that

$$L_{Level_{i+1}}(db) \leq L_{Level_i}(db)$$

That is, we use the following scheme:

- Choose a *min-sup*
  - Determine all frequent item sets for thus *min-sup*

– Determine the best coding set from these frequent item sets

- Choose a smaller *min-sup* and continue until the search converges.

This may seem like a vacuous scheme. Of course it will converge, at the very latest when all possible item sets (with non-zero support) have been considered. However, it is not as bad as it seems.

Firstly, for sparse data sets it is not impossible to compute all item sets with very low thresholds, especially since we can concentrate on the closed item sets. The extensive experiments in the FIMI workshops [6] show that these can be computed relatively fast.

Secondly, we do not have to maintain this enormous list of (closed) item sets. In each iteration, the set of newly generated item sets is compressed to a coding set that is orders of magnitude smaller as we have seen in the previous section.

Finally, and most importantly, we can re-use heuristics that exist for algorithms such as MCMC. That is, we plot the compressed database size against *min-sup*. If the plot becomes more or less horizontal, we decide that it has converged. We can then simply read off the optimal threshold *min-sup* from the plot.

Clearly, just as for MCMC, this is only a heuristic. The iteration after the last one we looked at could give a (huge) drop in the size of the compressed database. But, it seems a better heuristic than simply defining a threshold and hope that all interesting patterns are covered. Figure 8 illustrates the idea. It shows both the viability and the dangers of this approach.

6 Conclusions

In this paper we introduced a new way to fight the explosion of frequent item sets that occurs at low thresholds. Rather than trying to compress the set of frequent items, we compress the database. Using the MDL principle, we search for that subset of all frequent item sets that compresses the database best.

The greedy **Naive Compression** algorithm we introduced in this paper does impressively well. Pruning reduces the set of frequent item sets even more, while giving comparable compression of the database.

The compression algorithms reduce the number of frequent item sets by orders of magnitude, both on all frequent item sets and on closed frequent item sets. Moreover, the compression levels reached show that these small sets of item sets do indeed capture the structure in the database.

Finally, we have shown that this MDL-based approach can be used to compute the optimal threshold value for *min-sup*.

There are two directions we are currently exploring. Firstly, the implementation of the algorithms is rather naive. More sophisticated and, thus, faster implementations and perhaps some more heuristics should be possible. Secondly, with these faster implementations we want to do more extensive testing.

References

[1] F. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent item sets. In *Proc ACM SIGKDD conference*, pages 12–19, 2004.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD conference*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE conference*, pages 3–14, 1995.

[4] R. Bayardo. Efficiently mining long patterns from database. In *Proc. ACM SIGMOD conference*, pages 85–93, 1998.

[5] B. Crémilleux and J-F. Boulicaut. Simplest rules characterizing classes generated by delta-free sets. In *Proceedings of the 22nd SGAI International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, pages 33–46, 2002.

[6] B. Goethals et.al. FIMI website. In *fimi.cs.helsinki.fi*.

[7] P.D. Grünwald. Minimum description length tutorial. In P.D. Grünwald, I.J. Myung, and M.A. Pitt, editors, *Advances in Minimum Description Length*. MIT Press, 2005.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD conference*, pages 1–12, 2000.

[9] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. PKDD conference*, pages 13–23, 2000.

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

[11] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[12] A. Silberschatz and A. Tuzhilin. What makes patterns interesting in knowledge discovery. *IEEE Transactions on Knowledge and Data Engineering*, 8:970–974, 1996.

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

[14] D. Xin, J. Han, X. Yan, and H. Chend. Mining compressed frequent-pattern sets. In *Proc. VLDB conference*, pages 709–720, 2005.

[15] M.J. Zaki and M. Orihara. Theoretical foundations of association rules. In *Proc. ACM SIGMOD workshop on research issues in KDD*, 1998.