Characteristic Relational Patterns

Arne Koopman

The research reported in this thesis was supported by BSIK/BRICKS within the BRICKS IS2 project.

SIKS Dissertation Series No. 2010-28
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Ridderprint Offsetdrukkerij BV, The Netherlands

ISBN 978-90-393-5320-2

Characteristic Relational Patterns

Karakteristieke Relationele Patronen
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof.dr. J.C. Stoof., ingevolge het besluit
van het college voor promoties in het openbaar te verdedigen op
maandag 31 mei 2010 des middags te 4.15 uur

door

Adrianus Cornelis Matheus Koopman
geboren op 22 januari 1977 te Bergen op Zoom

Promotor: Prof.dr. A.P.J.M. Siebes

Contents

Chapter 1

Introduction

Once upon a time, there was a banker with a small bank. Since his bank was small, he knew all his clients very well. He knew the credit status and the number of loans they had by heart, and even knew some of them personally. Because of this, he did so well, that his small bank grew into a big multinational, with offices all over the world. Although business went well, he lacked the keen insight into the behaviour of his customers like he used to.

Before you, the reader, might think that the topic of this thesis is not what you expected it to be: don't panic. Our banker is simply the main illustrative example throughout this introduction and reappears in the conclusions.

In the last decades, many real world processes have been automated thanks to advancements in computer science. Many of these automated processes keep track of many details and store large quantities of data automatically. For example, for the banking process all financial transactions need to be recorded in order to keep track of the credit balance of clients. One can imagine that this puts a strong emphasis on a correct data storage and management. While there are many ways to store data, such as a simple text file or XML data, there are many reasons to choose for one specific approach, namely relational database systems.

First of all, nowadays a relational database system allows for the efficient storage and management of up to petabytes of data. That is, it can store up to 1.000.000.000.000.000 bytes, which is the equivalent to around a 2 km high stack of cd-rom discs. However, what makes a relational database system really stand out from other storage approaches, is that it facilitates in much more than storage alone. One is that it allows multiple users to access and manipulate the database at the same time without compromising the integrity of the data. In our bank example, this is a crucial feature if one wants to ensure the correct credit balances of its clients. Moreover, a relational database can model many

different entities. Not only does this provides a one-suits-many approach, it also provides a rich enough data formalism such that many different entities can be modelled. For example, one database can both harbour data about the financial transactions, as well as the organisation network of the personnel of this bank.

These, and many other reasons, have led to the fact that relational databases are the de facto standard for data storage. They can be found serving applications for a wide variety of users, from large multinational companies to small scouting groups.

In these databases, many details are stored that relate to the underlying process's behaviour. One can imagine that within these huge collections of stored data new useful insight can be found. But how does one get this insight? To manually examine petabytes of data is virtually impossible, let alone to obtain a useful insight this way. This fact has led to the rise of a new discipline within computer science: automatically deriving insight from collections of data, known as *data mining*.

Pattern Mining

One of the most important fields in data mining is the discovery of patterns in data, also known as *pattern mining*. A pattern is simply a regularity in the data. Note that many of such patterns can be found within one database. For our banker, one pattern could be that clients that have a high mortgage also have a credit card. This example illustrates one of the main benefits of using patterns: they are easily interpretable. If these patterns are used in decision making, a decision based on patterns not only shows *that* it performs well, but also *why* it performs well.

In order to successfully apply pattern mining to real world sized databases, much effort has been put into making fast and efficient pattern miners. Currently, it is fairly easy to derive the set of all interesting patterns from a database. Only extremely hard, such as petabyte scale, databases still pose a challenge when mining for the set of all patterns. While this might sound that everything is OK for the not so extreme databases, there still remains a completely different problem. This problem lies in the interpretation of *interesting*.

A common interpretation is that a pattern is interesting if it is *frequent*: it occurs at least a certain number of times in the database. Setting this frequency threshold too high leads to a set of patterns that provides little new insight. Namely, if a pattern occurs nearly every time, it is likely that one knows this already. The alternative of course is to lower this threshold. However, this often leads to an explosion in the number of patterns that are interesting: too many (similar) patterns become 'interesting'. The focus on returning all frequent

patterns has left our banker, and many others data miners, with a flood of patterns that is often larger than the original database size itself!

Research Problem

This brings us to the topic of this thesis. In practice, pattern mining leads to pattern sets that are often too big. So before trying to mine patterns from a petabyte-sized relational database, we first need to solve a fundamental problem: the pattern set explosion. In this thesis we therefore address this problem. Instead of considering the set of *all* frequent patterns, we only select a few interesting ones. The rationale behind this, is that the set of *all* frequent patterns is not so much of interest. There is much redundancy between the patterns, as many of them are variations of the same theme. Consider the earlier mentioned pattern: *clients with a high mortgage also have a credit card*. If this pattern is frequent, then both *some clients have a credit card*, and *some clients have a high mortgage* are also frequent patterns. As these patterns can both be derived from the first, they provide no new insight.

The research problem of this thesis can therefore be stated as follows:

Derive a compact set of high-quality, non-redundant, and characteristic relational patterns that summarise the complete database well.

In order to model, or summarise, the database we focus on sets of patterns instead of individual patterns. That is, we use a set of patterns to model the database. Furthermore, we are not interested in just *any* set of patterns, we are interested in finding a *characteristic* set of patterns. Similar to the definition of interesting, the question becomes what do we define as characteristic? To answer this question, we choose the model that describes the database best. In order to do this, we use the Minimum Description Length (MDL) principle that, similar to Occam's razor, prefers the simplest model of the data that results in the best description. That is, it selects the simplest model that compresses the database best.

The rationale behind this is that the better our model compresses the database, the better it captures its regularities. The strong point of MDL is that is takes the complexity of the model into account. This ensures that the model does not get overly complex when summarising the data. In effect, this leads to non-redundant models that do not model noise, as capturing spurious patterns would increase the complexity of the model, without contributing to the compression.

While this all sounds too good to be true, there is a slight catch: finding the optimal model of the database is far too hard. The huge search space of all

possible models contains no useful structure that can be exploited to find our optimal model in feasible time. We therefore settle for finding the best model of the database based on heuristics.

Fortunately, we show that although we do not guarantee to obtain the optimal model, we do obtain very good models. With our heuristics and encodings, we show that we find compact and high quality models in various scenarios. The fact that these models contain very characteristic patterns is further confirmed by the fact that these lead to well performing classifiers.

Outline of this thesis

The remainder of this thesis is divided in chapters, most of which relate to earlier published work as noted at the beginning of each chapter. The topics of these chapters can be summarised as follows:

- In Chapter 2, we provide a short introduction to pattern mining. Pattern mining is often linked to a specific application that dictates a specific data type. In a simple setting, a pattern can be a set of items that is frequently bought in a supermarket. However, for certain applications it is important to consider more complex data types. We discuss various data types, such as sequences, trees, and those related to relational databases, and their specific pattern mining flavours. We discuss how different pattern types affect their related mining techniques, and discuss the Achilles' heel of pattern mining: the pattern set explosion.

- In Chapter 3, we address the issue of the frequent pattern set explosion for a single table of itemset data. To this end, we introduce the algorithm Krimp that selects a small set of characteristic patterns from a typically huge set of frequent patterns. Krimp uses the Minimum Description Length principle to select a model that describes the database well. In order to stay efficient we use heuristics and demonstrate that these lead to compact pattern sets.

- In Chapter 4, we show how the Krimp heuristics can be applied to structured data such as sequences and trees. While less general than relational data, structured data allows to express more complex structures compared to itemset data, and is widely used (e.g. in XML data). We show that in order to apply Krimp to structured data, we need some additional restrictions in the encoding of the database. Using this, we show that we can find compact structured pattern sets that describe the database.

– In Chapter 5, we transfer KRIMP to relational databases for a specific type of pattern, namely relational itemsets. We show that out of a huge set of candidate patterns, KRIMP selects a set of patterns that describe the database well. Moreover, we show that we attain a similar result with a much smaller set of candidates by utilising KRIMP's ability to grasp the underlying distribution. This increases the efficiency of our algorithm, termed R-KRIMP, both in space and in time complexity.

– In Chapter 6, we build upon the fact that relational databases can harbour complex patterns. We introduce our algorithm RDB-KRIMP that utilises both a newly introduced complex pattern language and a new relational database encoding scheme. We show that we can find compact description of the complete relational database. Moreover, we show that using more complex patterns is beneficial: it allows for a more succinct description of the database.

– In Chapter 7, we confirm the quality of the models derived by RDB-KRIMP by means of classification. We show that the RDB-KRIMP models perform well in terms of classification. The models based on more complex pattern languages perform better. That is, MDL picks out characteristic patterns that can discriminate well between classes.

In Chapter 8, we draw some conclusions and summarise the contributions of this thesis.

Chapter 2

Relational Pattern Mining

Data mining is the process to extract knowledge and insight from databases. It is well-known that increasingly more information is stored within databases, which thereby potentially harbour interesting knowledge. However, with their size and complexity, the manual analysis of these databases is hard or even impossible. The data mining community tries to facilitate this analysis by creating algorithms and approaches that automate (part of) this process.

One successful area within data mining is that of finding interesting patterns in data: pattern mining. Patterns are insightful, as they are interpretable partial models of the database. Although patterns by themselves are already interesting, they can also serve as input for many further analysis tasks. These tasks include areas such as classification, clustering, and imputation.

In this chapter we shall provide a brief overview in the field of pattern mining. Since its introduction for the analysis of market basket data, it has been incorporated to many other application areas. We will introduce various forms of pattern mining, that are either well-suited for different data types, or based on different measures of interestingness. We will conclude with the introduction of pattern mining to the area of relational databases.

2.1 Introduction

Patterns are nearly everywhere! This notion is supported by one of the most successful data mining approaches that have been developed in the last decades: pattern mining. Pattern mining focusses on finding interesting patterns, and often this is interpreted as those patterns that occur frequently in the data.

To illustrate this approach, consider the prototype setting of a supermarket that wishes to analyse their sales transactions. Every time a customer buys some products, we can associate a new transaction with the set of bought items. Over time, the supermarket can store these transactions. Given this set of transactions, we can now search for frequently occurring patterns: *"what items are often bought together"*. The prototype, semi humorous, example answer for this is that *diapers* and *beer* are often sold together.

Patterns in themselves are very informative. However, they can also serve as a starting point for follow-up data mining tasks. The patterns that we find in the complete database can provide insight for a single transaction. For example, say that we know that customer X has bought *beer*. As *beer* and *diapers* are often sold together, we could infer that customer X is likely to buy *diapers* as well. This concept has been transferred to data mining tasks such as classification (*"is customer X a diaper buyer?"*), clustering (*"which groups of customers buy similar goods?"*), or recommendation (*"what will customer X buy next?"*).

As a prelude to the remainder of this thesis, this chapter will introduce some formal notions related to pattern mining. After introducing these notions for transactional itemset databases, we will briefly discuss how these have been transferred to more structured data types. This chapter will present an overview on the wide variety of pattern mining flavours, together with some problems that pattern mining still faces.

2.2 Pattern Mining

Frequent pattern mining was first introduced by Agrawal et al. in the context of mining frequent sets of items from a transactional database [2]. This is reflected in the standard definitions of frequent itemset mining. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. Each transaction t contains a set of items: $t \subseteq \mathcal{I}$. The database $\mathcal{D}$ is a bag of transactions, and we denote by $|\mathcal{D}|$ the number of transactions it contains. In general, we will denote by $|A|$ the number of elements in set or bag A.

In this frequent itemset mining setting, a pattern F is defined as a set of items: $F \subseteq \mathcal{I}$. We define the *cardinality* of F as the number of items it contains, and define the *singletons* as those patterns that have *cardinality* 1.

Given an itemset database $\mathcal{D}$, one could mine the set of *all* patterns from $\mathcal{D}$. However, this very easily leads to humongous pattern sets! Given $\mathcal{I}$, *every* subset can be considered a pattern ($F \in \mathcal{P}(\mathcal{I})$). Fortunately, from a practical viewpoint, we can focus on those patterns that actually *occur* in $\mathcal{D}$. To reduce the resulting pattern set even further, we can resort to those patterns that are *frequent*.

Given a transaction $t \in \mathcal{D}$, we say that a pattern F *occurs* in t iff $F \subseteq t$. The *support* of F , $sup_{\mathcal{D}}(F)$, is the number of $t \in \mathcal{D}$ it occurs in: $sup_{\mathcal{D}}(F) = |\{t \in \mathcal{D} \mid F \subseteq t\}|$. Note that when $\mathcal{D}$ is clear from the context, we write $sup(F)$ instead of $sup_{\mathcal{D}}(F)$. The fact whether F is *frequent* or not, depends of course on what is defined as frequent. For this, we define a support threshold above which all patterns are denoted *frequent*. This threshold is termed the minimum support value, named *minsup* or θ for short, for the absolute and relative support respectively. A pattern is *frequent* iff $sup_{\mathcal{D}}(F) \geq minsup$, where $0 \leq minsup \leq |\mathcal{D}|$, or $sup_{\mathcal{D}}(F) \geq \theta|\mathcal{D}|$, where $0 \leq \theta \leq 1$. We denote $\mathcal{F}$ as the set of all patterns that are frequent .

In order to derive $\mathcal{F}$ from a given database $\mathcal{D}$, one could first naïvely generate the patterns that occur and then select those patterns that are frequent. In practice, this approach is far from efficient as the set of occurring patterns is huge. To this end, Agrawal et al. have introduced the a-priori principle to mine the set of frequent patterns in a much more efficient manner [3, 67].

The fundamental observation is the following: given two patterns F_1 and F_2, if $F_1 \subseteq F_2$ then $sup(F_1) \geq sup(F_2)$. Agrawal et al. have used this a-priori principle in a breadth-first search style algorithm to find all frequent patterns [3]. The rough sketch for this algorithm is as follows. Starting with the *singleton* patterns, gradually higher cardinalities are evaluated. At every step, only the *frequent* patterns of cardinality k are extended as candidate patterns of cardinality $k + 1$. In this manner, large parts of the search space can be pruned using the a-priori principle.

The a-priori principle has proven to be very fruitful and has inspired many frequent itemset mining algorithms that address aspects such as memory and computational efficiency [37, 96]. Recently, the focus has shifted from generating all frequent patterns (from now on also referred to as *all* patterns) to selecting a smaller set of patterns [35]. The reason for this becomes clear when we look at the size of $\mathcal{F}$ (e.g. see *all* patterns in Figure 2.1). In general, we see that the size of $\mathcal{F}$ steeply increases for low values of *minsup* , as more and more patterns become frequent.

The size of $\mathcal{F}$ can be kept small if the *minsup* value is chosen sufficiently high. The drawback of this approach is that the resulting $\mathcal{F}$ contains only highly frequent patterns at high *minsup* values. This results in common knowledge and provides very little new insight. Therefore, in order for patterns

within $\mathcal{F}$ to be of any practical use, we often need to mine at low *minsup* levels. Regrettably, as mentioned, this often leads to very large $\mathcal{F}$ where one gets swamped with patterns.

Fortunately, there are ways to bring down the size of $\mathcal{F}$, as there is much redundancy in the set of *all* patterns. As an example, consider a frequent pattern set $\mathcal{F} = \{ABC,\ AB, AC, BC, A, B, C\}$. It is clear, that if ABC is frequent, all of its subsets $\{AB, AC, BC, A, B, C\}$ are also frequent. Given this, one could question whether is it interesting to list either all of these patterns, only the itemset ABC, or some of the itemsets within $\mathcal{F}$.

To reduce this redundancy in $\mathcal{F}$, other measures of interestingness have been proposed in addition to frequency. Two well-known measures are *closed* patterns [73] and *maximal* patterns [77]:

Definition 1 (Closed and Maximal Patterns)**.**

- *we call a pattern F a* closed *frequent pattern given a database $\mathcal{D}$ iff F is frequent and there exist no pattern F' such that $F \subset F'$ and* $\sup(F) = \sup(F')$,
- *we call a pattern F a* maximal *frequent pattern given a database $\mathcal{D}$ iff F is frequent and there exists no pattern F' such that $F \subset F'$ and F' is frequent in $\mathcal{D}$.*

The *closed* frequent pattern set is lossless, in the sense that it can be used to derive the frequency information of the set of *all* frequent patterns. In contrast, the maximal frequent pattern set is lossy: while more compact, it usually does not contain the complete frequency information for the set of all frequent patterns.

To illustrate the reductions in set size, we consider the resulting pattern sets depicted in Figure 2.1. For this example, we picked the mushroom database from the UCI Machine Learning repository, which contains 8124 transactions with features to predict whether they are edible or not. Shown here for the mushroom database are the pattern set sizes for various *minsup* values. Depicted are the sets of all frequent patterns, closed patterns, and maximal patterns. We see that both measures decrease the pattern set size to a great extend.

As both measures reduce the resulting set size, they have formed the basis for numerous implementations that derive closed or maximal pattern sets. For *closed* patterns, amongst the plethora of miners, AFOPT [64], CHARM [98], FPClose [31] are well-known examples. Similarly, for *maximal* patterns a number of miners have been developed [10,77]. A comprehensive overview of pattern set miners can be found in [30]. Although both closed and maximal allow for a reduction in the pattern set size, we see that for low *minsup* values both

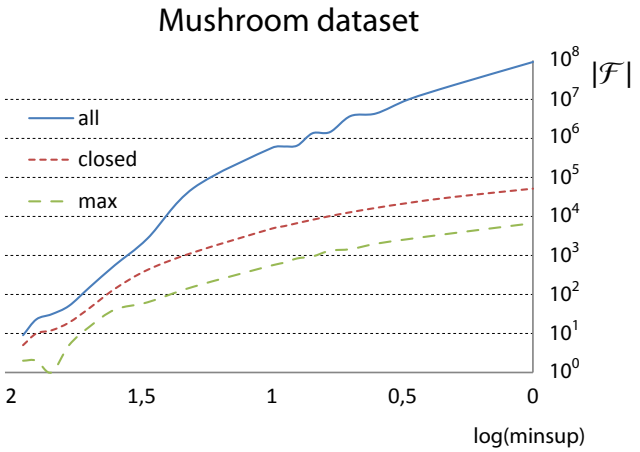


Figure 2.1: An illustrative example on the growth of the frequent pattern set $\mathcal{F}$ given an database $\mathcal{D}$, in this case the UCI Mushroom dataset

measures still lead to fairly large sets. This indicates the demand for more sophisticated ways to reduce the pattern set size.

One approach to have a more strict pattern selection, is to include background knowledge. Given background knowledge, large parts of the search space can be pruned that do not comply with the constraints of the user. For example, if one is only interested in male customers, we can discard all data relating to the female population. This data mining approach, *constraint based* data mining, has also been transferred to the area of pattern mining. For a detailed overview on these approaches see [35]. This approach of course only works when useful background knowledge is available.

Another approach is the selection of the *top-k* closed patterns [38]. In this approach, the resulting set is limited to k patterns. In order to determine these k patterns, the closed pattern set $\mathcal{F}$ needs to be ordered. For each $F_1, F_2 \in \mathcal{F}$, we define an order on $\mathcal{F}$ such that: $F_1 \prec F_2$ iif $sup(F_1) > sup(F_2)$. We will denote this support descending order as: $\downarrow \mathcal{F}$. Given $\downarrow \mathcal{F}$, the *top-k* closed patterns are the first k patterns in order. The parameter k allows for compact *top-k* pattern sets. However, the choice of k is arbitrary and introduces an additional parameter for the user to fine tune. Moreover, in general, the k most frequent closed patterns, is from the user's perspective an arbitrary set of patterns as it can have much overlap.

As a final similar approach, Calders and Goethals proposed a new measure

that allows for a condensed pattern set: *non-derivable* itemsets [11–13]. The authors introduce a set of deduction rules to derive upper and lower bounds on the support levels. If the pattern support can be derived, it can be removed from the resulting pattern set. The resulting pattern set therefore contains only those patterns that are *non-derivable*.

2.3 Mining Structured Data

In the previous sections life was straightforward. The patterns derived from the database $\mathcal{D}$ were unstructured sets of items. This led to simple patterns such as $\{A, B, C\}$. In the remainder of this thesis however, we will deal with patterns that are more complex. To be more specific, we are going to allow structure in our patterns. We do this for a good reason. As structured data is more complex, we need our patterns to match this increase of expressiveness in order to provide us with a more detailed insight into the data.

By using the same three items $\{A, B, C\}$ in a structured setting, we can express more patterns opposed to one unstructured itemset. Let's say we introduce structure in the sense that we order the items in the itemsets. This way, we now have 6 distinct patterns we can distinguish: $\{ABC, ACB, BAC, BCA, CAB, CBA\}$. The fact that BAC *does* occurs in $\mathcal{D}$ and the remainder of the patterns do not, informs us that this order is crucial for these three items to appear in $\mathcal{D}$.

There are various applications that can profit from treating the data in a structured manner. For example, consider a website that logs all visits to its webpages. In this, the obtained database contains ordered sets of visited webpages. In the derived patterns, it is relevant to preserve in which order the pages have been visited. Since this provides insight in how a user often navigates through the website. This in turn can be used to optimise the structure of the website, to allow users to find their information of interest more quickly.

In this section, we will briefly discuss how the concepts of pattern mining have been transferred to the domains of structured data. More specifically, we will focus on structured data types such as sequences, trees and graphs. We will discuss these data types in more detail together with their respective pattern mining flavours. The field of structured pattern mining is characterised by its many flavours such as: sequence mining, tree mining, graph mining, XML mining, and structure mining.

Which pattern type to use can depend on the specifics of the application of interest that dictates a specific data type. While graph patterns are the richest, and encapsulate the other discussed structured pattern types, they are also very hard to mine. The work of Bringmann et al. shows a nice example on how it serves to first focus on simpler patterns before considering more

complex patterns [9]. In this light, instead of a sole interest in graph mining, algorithms for various structured data types have been optimised that exploit their respective data type characteristics.

Sequences

One of the basic structured data types is that of a *sequence* (see Figure 2.2). A sequence, also referred to as an episode or as sequential data, is an ordered bag of items. To be more specific, we will first start with a formal definition of a sequence.

Definition 2 (Sequence)**.** *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items given a database $\mathcal{D}$. We define a sequence S of length n as an ordered bag of indexed items: $S = \{(s_i, i)\}$ in which $s_i \in \mathcal{I}$ and $i \in \{1, \ldots, n\}$. The set is ordered: $(s_i, i) \preceq (s_j, j)$ for $1 \leq i \leq j \leq n$.*

The indexes i can be omitted when clear within the context, so we can write s_i instead of (s_i, i). For example, $\{(A, 1), (B, 2), (C, 3)\}$ simply becomes ABC. In the context of structured data, s_i is also referred to as a *label*. Similar to the case of itemsets, our database $\mathcal{D}$ consists of a bag of sequences t. Each sequence t has exactly one sequence S of arbitrary finite length: $t = S$.

The patterns F within $\mathcal{D}$, are again sequences S over $\mathcal{I}$. In the case of itemset mining, the occurrence checking is quite straightforward: the itemset is either a subset of the transaction or not. In the case of structured data it is less straightforward. For example, given a sequence $ABCD$, do we say that a pattern AC occurs in this sequence or not? If one only want to consider 'connected' sub-sequences of the data, one would say no, otherwise yes. Therefore, for structured data, there is a need to redefine the occurrence function to support the desired occurrence notion. More formally, given a pattern F, we define an injective mapping function $\Phi : F \to S$, such that $\Phi(F)$ is a component of S. The exact definition of the occurrence function Φ depends on the application of interest. Without providing an exhaustive list of all possible occurrence functions Φ, we will outline two commonly used variants. The first allows gaps in the matching sequence from the database, the latter does not.

Definition 3 (Gap Sequence Occurrence)**.** *Let S and T be both sequences over $\mathcal{I}$, and $\Phi_{gaps} : S \to T$ a* label preserving, injective *mapping. S occurs in T, denoted by $S \subseteq T$, iff $\forall s_i, s_j \in S : s_i \preceq s_j \Leftrightarrow \Phi_{gaps}(s_i) \preceq \Phi_{gaps}(s_j)$.*

The mapping Φ_{gaps} is called the occurrence with *gaps*. Note that in practical use, the maximal gap between $\Phi_{gaps}(s_i)$ and $\Phi_{gaps}(s_{i+1})$ $s_i \in S$ is limited to a predefined number, e.g.: *maxgap*.

Definition 4 (Gapless Sequence Occurrence)**.** *Let S and T be both sequences over $\mathcal{I}$, and $\Phi_{no\ gaps} : S \to T$ a* label preserving, injective *mapping. S occurs in T, denoted by $S \subseteq T$, iff $\forall s_i, s_j \in S$:*

1. $s_i \preceq s_j \Leftrightarrow \Phi_{no\ gaps}(s_i) \preceq \Phi_{no\ gaps}(s_j)$
2. $\forall t \in T : \Phi_{no\ gaps}(s_i) \preceq t \preceq \Phi_{no\ gaps}(s_j) \Leftrightarrow$
   $\exists s_k \in S : \Phi_{no\ gaps}(s_k) = t \wedge s_i \preceq s_k \preceq s_j$.

The mapping $\Phi_{no\ gaps}$ is called the *gapless* occurrence.

Thus, given a sequence $S = ABCD$, and occurrence function Φ_{gaps}, a pattern $F = AC$ *would* be a matching pattern in S. However, given the same sequence S and $\Phi_{no\ gaps}$, the same pattern F would *not* match in S.

Now that we have defined our patterns and the occurrence functions, we can translate the earlier defined notions of *support* and *frequency* to the case of sequences. Unlike an itemset in an itemset database, one pattern F can occur multiple times within a sequence $t \in \mathcal{D}$. Therefore, the support of a pattern F is not by default limited to $|\mathcal{D}|$. As an alternative, we can count all occurrences that are found in $\mathcal{D}$, i.e. $sup_{\mathcal{D}}(F) = |\{\text{occurrences of } F \text{ in } \mathcal{D}\}|$. However, the sequence based support is similar to the case of itemsets and counts the number of sequences the pattern occurs in. More formally: $sup_{\mathcal{D}}(F) = |\{t \in \mathcal{D} \mid F \subseteq t\}|$.

Based on these notions, there has been a substantial effort in developing algorithms to mine sequences from sequential data. Of these prefixspan [74] and SPADE [97], are well-known. In order to process larger data sets similar to itemset mining, much effort is focussed on increasing the computational efficiency, e.g. by incorporating either the closed property in BIDE [88], or constraints in the form of regular expressions [25]. For a more detailed overview on the topic of sequence mining and its well-known algorithms, see the work of Zhao et al. [99].

Trees

A recently introduced and widely used data format is that of XML documents. These documents are often considered to be structured in a tree like manner: data is stored within node elements, and each node can have multiple branches to other nodes. The application of pattern mining techniques to tree structured data types is known as tree mining [18]. As with sequences, tree data types can be defined in various ways. In this section we will briefly discuss some common types encountered in the remainder of this thesis. The tree data types that we will briefly touch upon are: *unordered, ordered*, and *attributed* trees, and *induced* subtrees.

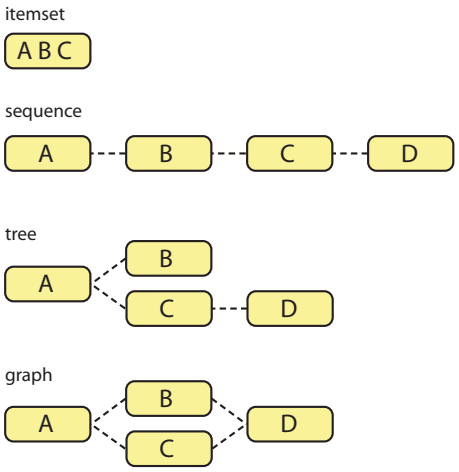


Figure 2.2: Given a set of items $\mathcal{I}$, we can define a number of pattern types. From top to bottom we have, an *itemset*, an ordered bag of items in a *sequence*, a *tree* pattern that allows branches, and a *graph* pattern that allows for the inclusion of cycles.

Definition 5 (Rooted Unordered Tree)**.** *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An* unordered rooted *tree T over $\mathcal{I}$ is given by a tuple: $T = \{V, E, v_0, \Sigma\}$. V is the set of nodes, v_0 denotes the root node, and E is the set of edges connecting the nodes: $E = \{(v_i, v_j) \in V \times V\}$. Each node is labelled with an item by the labelling function $\Sigma : V \to \mathcal{I}$.*

A path *between two nodes v_1 and v_n, denoted by $path(v_1, v_n)$ is a set of nodes $v_1, \ldots, v_n$, with $n > 1$ such that $\forall i : 1 \leq i < n : (v_i, v_{i+1}) \in E$, and $\forall i, j : 1 \leq i, j \leq n : i \neq j \rightarrow v_i \neq v_j$.*

For each non-root node $v \in T$ there exists a path to the root node: $\forall v \in V \setminus v_0 : \exists path(v_0, v)$. Furthermore, we do not allow cycles: $\forall v \in V \setminus v_0 : \exists! path(v, v)$.

We define the size of T as the number of nodes it contains: $size(T) = |V|$. Our database $\mathcal{D}$ consists of a bag of trees t. Each tree t contains a tree T of arbitrary finite size. In contrast to an unordered tree, we can order the nodes within the tree: an ordered tree.

Definition 6 (Ordered Tree)**.** *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An* ordered rooted *tree T over $\mathcal{I}$ is a special case of a unordered tree and is given by a tuple: $T = \{V, E, v_0, \Sigma, \preceq\}$. In addition to the restrictions inherited*

from an unordered tree (see Definition 5), an ordered tree has a partial order $\preceq$ on V: for all $X, Y \in V$, if $\Sigma(X) \leq \Sigma(Y)$, then $X \preceq Y$.

Shown in Figure 2.2 is an example of a tree. We see a tree T with a root node labelled A that has two children B and C, of which the latter node has a child node D. While trees are more complex than sequences, rooted ordered trees can be described in a string notation called SMILES [90]. In this notation, the example tree T is written as: $A(B)(C(D))$. Note that if T is considered an *unordered* tree, it matches with both $A(B)(C(D))$ and $A(C(D))(B)$.

As can be seen from this example, one single unordered tree can be transformed into multiple ordered trees. While this makes an unordered tree a more general representation, it makes it harder to check for occurrences. The occurrence checking of a pattern forms an important part of pattern mining algorithms. Therefore, we need to define the type of patterns that we wish to find in the database $\mathcal{D}$. Again we have some freedom in how to define this occurrence function. In this thesis we make use if *induced* trees.

Definition 7 (Induced Subtree)**.** *A ordered tree $T_1 = \{V_1, E_1, v_{0_1}, \Sigma_1, \preceq_1\}$ is an* induced subtree *of an ordered tree $T_2 = \{V_2, E_2, v_{0_2}, \Sigma_2, \preceq_2\}$, denoted as $T_1 \subseteq_I T_2$ iff:*

1. $V_1 \subseteq V_2$,
2. $E_1 \subseteq E_2$,
3. $\forall e = (v_i, v_j) \in E_1 : (\Sigma_1(v_i), \Sigma_1(v_j)) = (\Sigma_2(v_i), \Sigma_2(v_j))$,
4. $\forall v_i, v_j \in V_1 : v_i \preceq_2 v_j \rightarrow v_i \preceq_1 v_j$.

We say that if $T_1 \subseteq_I T_2$, that T_1 *occurs* in T_2. Note that when dealing with *unordered* trees, the last condition (4) does not need to hold. As an example, given $T_1 = A(B(DE))$, $T_2 = A(D)$, and $T_3 = A(B(D))$, we have: $T_2 \subsetneq_I T_1$, and $T_3 \subseteq_I T_1$.

A final tree data type that we would like to mention here is the *attributed* tree, as we will use it later on. While all previous data types all have one single label per node, attributed trees can have multiple attributes per node.

Definition 8 (Attributed Unordered Tree)**.** *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An* unordered rooted *tree T over $\mathcal{I}$ is given by a tuple: $T = \{V, E, v_0, \Sigma\}$. The restrictions are inherited from the unordered tree (see Definition 5). Σ is redefined, as each node can be labelled with multiple items by the labelling function $\Sigma : V \rightarrow \{\mathcal{I}\}$.*

Similar to the case of sequences, we can now define the notions of *support* for trees. Again, we can define a tree based support, that counts the number

of trees the tree pattern occurs in. More formally: $sup(F) = |\{t \in \mathcal{D} \mid F \subseteq t\}|$. Alternatively, we can count *all* occurrences that are found in $\mathcal{D}$, i.e. $sup_{\mathcal{D}}(F) = |\{\text{occurrences of } F \text{ in } \mathcal{D}\}|$.

Each different tree pattern type gives rise to a distinct manner of mining the patterns in an efficient manner. For *ordered induced* trees Asai et al. [4] have developed FREQT. For *unordered induced* trees two distinct miners have been developed simultaneously: Unot [5] and uFreqt [70]. The more complex attributed trees, can be efficiently mined by use of the FAT miner as proposed by De Knijf [47].

Graphs

Within the structured data domain there is one type of data structure that encapsulated the ones previously described: graphs. While graphs can be used to model all the previously addressed data types, they inherently have severe computational problems. For one, the isomorphism problem, which is part of the subgraph matching problem, is NP-Complete.

The existence of an efficient graph miner would make much of the previously addressed subjects perhaps obsolete. However, the complexity of graphs have restricted their use to those areas where their specific use is beneficial. These areas range from molecular science [7, 58] to social networks [36]. A graph can be defined as follows:

Definition 9 (Graph)**.** *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An graph G over $\mathcal{I}$ is given by a tuple: $G = \{V, E, \Sigma\}$. V is the set of all nodes, and E is the set of all edges connecting the nodes: $E = \{(v_i, v_j) \in V \times V\}$. Each node is labelled by an item by the labelling function $\Sigma : V \rightarrow \mathcal{I}$.*

Similar to the previous cases of sequences and trees, various specialised forms of graphs have been proposed. These include outerplanar graphs [43] and undirected acyclic graphs [78]. Their specific properties have been used to soften the hard computational complexities inherent to the above general graph definition.

Well-known graph mining algorithms are gSpan by Yan & Han [92], Gaston by Nijssen & Kok [72], and MoFa by Borgelt & Berthold [7]. While these approaches return all frequent patterns, SUBDUE [42] returns a set of graphs based on a heuristic MDL based approach. We will come back to SUBDUE in Chapter 4, as it is related to our approach.

A small example

Before we continue with enriching the pattern language, we provide a small example to link the previous section with the next. Consider the following

everyday example: parents with their children. Let's name the parent A and the children B, C, D. As the children are born in a certain order in time, we can use a sequence to model the birth order. For example, this would lead for parent A to a sequence in birth order $ABCD$ (see Figure 2.2).

However, all of these children can have children of their own. In order to model this ancestral relationship in a suitable fashion, we can use a tree. For example, parent A has two children B and C, and C has one child D.

The members of this small family are likely keep in touch with each other. We can model this social interaction, but we need to enrich our data type once more. For example, say that A often talks to both of his children B and C. C often talks to his child D, but in addition, B also talks often to D. This requires a graph as this relational structure is cyclic: e.g. A talks to B, B to D, D to C, and C to A (see Figure 2.2 bottom).

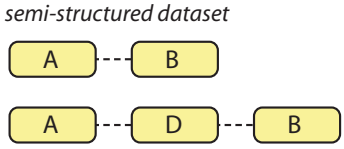
2.4 Mining Relational Databases

In the last decades, relational database management systems (RDBM) have become the standard for storing information. Inspired by a famous paper by Codd, RDBMS have found their way into many areas [19]. Aside from efficiently storing and searching data, the management system preserves the integrity of the data, and allows for back up facilities and concurrent user access. At first, RDBMS became very common for corporations to store their information, requiring large mainframes. Nowadays, a RDBM can run on desktop computers or small servers as well, which have led to an increase in its usage. Nowadays, the application of RDBMS range from storing petabytes of data for large companies, to supporting small webservers.

A RDBM stores the database in a relational format: a *relational database*. The basis for a relational database is a relational model that models the *entities* of the data and how they are related. In order to model the data, relational databases use *attributes*. In light of our previous family based example, one of the entities in our relational model would be Person. Each person would at least be characterised by the attributes *name* and *dob* (date of birth). In addition, to uniquely identify each Person, we introduce an unique identifier: *pid*.

In modelling the relational database, we specify for each attribute its domain, or *type*. This limits the range of possible values for this attribute. In the case of *dob*, an allowed value would be any valid date, in the past preferably. Common attribute types include integer values, text strings, date and time stamps, or even binary data such as digital photos.

To define an entity in our relational model we use a *schema*. A schema is a set of attributes. In our example, the schema of Person would be defined as



relational database

parent.pid	*child.name*	*child.dateofbirth*
1	A	1-1-2000
1	B	1-1-2009
2	A	1-1-2000
2	D	1-1-2002
2	B	1-1-2007

Figure 2.3: Relations data can either be expressed in structured data such as sequences, or as a relational database. Shown here is a set of sequences and a related table as a result of a SQL query.

the set of the following attributes: {*pid, name, dob*}. We denote this schema as: PERSON(*pid, name, dob*). The domains of *name* and *dob* would respectively be a text string, and a valid date. We postpone the domain of *pid* for now.

PERSON is called the *relation*, which can have multiple records or tuples that contain data according to the schema of PERSON. Each tuple t corresponds with an individual that is stored within the database (e.g. $t = (0, Mike, 11/11/1960)$). In this fashion, this fills the *extend* of a relation with many tuples that follow the schema. Throughout this thesis, we will use *table* to address the extend, the schema, or both interchangeably when this is clear from the context.

The current setup until now only allows for multiple individual tables. However, the strength of relational databases is that tables can relate to each other via *associations*. In order to illustrate this, we will introduce a second relation: CHILD(*pid, cid*). As every person can have any number of children, a tuple from PERSON can relate to zero or more tuples from CHILD. Common associations are: *binary*, where either a tuple has a related tuple or not, *one-to-many*, where one tuple in this table can relate to 1 or more, or *many-to-many*, where tuples in both tables can relate to multiple tuples in the opposing table. The association between PERSON and CHILD would be typically *many-to-many*. As 2 parents can together have multiple children.

In order for a set of tuples to be related, we need to be able to specify an unique tuple. For this, we define a set of attributes as the *key* of the relation that can uniquely identify the tuple. As its uniqueness forms a essential element of the relational database, it is common practice to introduce an integer type attribute specifically for this purpose. For the PERSON and CHILD, the keys respectively are: *pid* and (*pid, cid*).

Note that *pid* in Child refers to the key of Person. We call such a key a *foreign key*. In a relational database scheme, we define the notion of *referential integrity*. That is, we require that the values used for the foreign key *pid* in Child refer to actually existing values of *pid* in Person.

Relational Pattern Mining

Relational data mining algorithms make use of this more expressive relational data formalism and aim at working directly with the relational databases format. An important contribution to mine patterns directly from relational databases comes from the field of Inductive Logic Programming (ILP). In ILP approaches, patterns are denoted by first order logic clauses. An example of such a clause would be:

$$\text{Person}(pid = X) \leftarrow \text{Person}(pid = Y) \wedge \text{Child}(pid = X, cid = Y).$$

This example clause results in a list of all children for each parent X. The analogy with relational databases is clear, as each term can refer to an attribute (e.g. *pid*), and each predicate to a relation (e.g. Person). Note that the above clause is relational: none of the single tables contains enough information for this clause to be verified.

Although ILP has been used extensively in a wider context of relational systems, we focus on the discovery of frequent patterns within relational databases. The first usage of ILP for pattern mining, was introduced by Dehaspe et al. in the context of frequent datalog queries [21, 22]. A datalog query has the form $Q = H_1, H_2, \ldots, H_n$, where H_i are logical atoms. An example query that finds for each child its parent can be denoted by:

$$Q = \text{Person}(pid = Y), \text{Person}(pid = X), \text{Child}(pid = Y, cid = X).$$

A query Q can occur multiple times within a given database $\mathcal{D}$. In our example, we can have multiple *(child, parent)*, (X, Y), pairs. Thus, for each occurrence, we can use different values for attributes of Q.

That is, for $Q = H_1, H_2, \ldots, H_n$, with attributes $A_1, A_2, \ldots, A_m$ we can substitute $\theta = \{A_1/a_1, \ldots, A_m/a_m\}$, where $a_i \in Dom(A_i)$.

This notion of an occurrence plays an important role in frequent pattern mining, as it relates to the *support* of a pattern. In a single table database, the support is simply the number of transactions in which the pattern occurs. In a relational database setting, it is common practice to count the support based on the *key* of one table in particular, the so called *target* table. The support of Q then becomes the number of substitutions $Q\theta$ for the variables in the key of the *target* table. This *target* table is considered to be that table in

the relational database that is of special interest to the user. In our example, PERSON would be this *target* table, and PERSON.*pid* the key to count on. The support of this pattern is the number of distinct *pid* values that match with the query.

A well-known ILP-based algorithm to find frequent queries within a relational database is WARMR as developed by Dehaspe and Toivonen [21]. For a given database $\mathcal{D}$, a minimum support *minsup*, and a declarative language L that defines the *key* atom of a specific target table, WARMR finds all frequent queries. WARMR is inspired by the APRIORI algorithm and provides a level-wise algorithm. Similar to APRIORI, WARMR starts with patterns of cardinality l, and extends these to candidates of cardinality $l + 1$. To clarify, WARMR starts with the query $Q = key$, which in our example database is Q = PERSON.PID. From this query with cardinality 1 it extends this to queries with cardinality 2, and so on. Similar to APRIORI, WARMR uses the frequency to determine whether it will extend certain queries as candidate queries.

WARMR has inspired many follow up approaches including FARMER [69]. FARMER has a better computational efficiency and allows for a richer declarative language. Even in our simple example this richer language proves useful. In our above example, WARMR can only list one child per parent, while FARMER can return multiple children per parent. Alternatively, a simpler declarative language also has its merits. Goethals et al. have proposed the mining of a special class of queries: simple conjunctive queries. The majority of the everyday queries are of this type, namely select-project-join queries. Their miner called SMuRFIG can mine this type of common queries very efficiently [28, 29].

Propositionalisation

After this short outline of relational data mining techniques, one could wonder: why do we need mining algorithms to deal with relational data? Why do not simply join all tables together and apply single table mining techniques. While this might sound attractive, there are some issues with this naïve form of conversion. First of all, the structural information is lost through a join (e.g. it is not clear anymore which attributes belong to the PERSON or CHILD relation). Second, joining can introduce data skew [44]. To illustrate this, we assume we have one parent that has one child and another that has many. In this case, the attributes of latter parent will occur more often in the joined table, and will skew the support of its related patterns. Third, when no related information is available NULL values will need to fill up the empty space. Finally, joined tables, easily can become very large, which hampers the performance of single table methods considerably.

However as an alternative approach, one can derive an interesting *summary* of the relational database in a single table. The process of propositionalisation selects the relevant aspects of a relational database and transfers this to a single table database [50,53,59]. For example, instead of listing all children per parent, we can derive for each parent the number of children. This process is called aggregation; it summarises specific related attributes into a single new attribute. Common aggregation functions include operators such as: *count, max, min*. Given this summary table, we can now use single table techniques to mine for patterns. The resulting pattern set contains patterns that provide a partial summary of the database. This summary comes at some cost, as it loses some of the fine grained structure, and therefore might lose informative data. As an example, when summarising the number of children, we might oversee that two siblings are often born in either January or March respectively.

2.5 Conclusions

In this chapter we presented some important aspects of pattern mining. Since the introduction of the a-priori principle, we have seen many different algorithms that mine a set of patterns from data. A pattern, which is simply a regularity in the data, is easily interpretable and describes a part of the database in detail. Regrettably, pattern mining has one major shortcoming in the sense that one typically finds too many interesting patterns; an effect that is aptly named the pattern set explosion. Many different interestingness measures have been designed to minimise this effect, such as *closed* and *maximal*, but these still leave room for much improvement.

Nevertheless, the success of pattern mining has spread to many different types of data. Initially introduced for unstructured itemset data, its techniques can now can be found in various structured domains. We have shown how to patterns are defined within data such as sequences, trees and relational databases. With all these various domains in which pattern mining is applied, the question of solving its Achilles' heel is clearly an important one.

Chapter 3

Krimp: Mining Itemsets that Compress

In the previous chapter, we have discussed a successful approach within the field of data mining: pattern mining. In a plethora of cases, this concerns finding patterns that occur frequently in a database. The current frequent pattern mining techniques have become so efficient, that even large itemset databases pose little difficulty when mining pattern sets.

However, one major drawback of pattern mining is that even in the simple case of itemset databases, the user gets swamped by huge pattern sets. This is likely to be worse in structured cases as more complex data types are likely to result in even larger pattern sets. Regrettably, interesting patterns are likely to have a reasonable low support, and this is often where the exponential growth of the pattern set takes its toll. Therefore, much research effort is focussed on creating more compact and interesting pattern sets.

In this chapter, which is derived from earlier work of Vreeken et al. [86], we introduce a heuristic algorithm called KRIMP that finds such a compact and interesting pattern set. Using the Minimal Description Length (MDL) principle, the aim is to find that set of patterns that together describe the complete database best. We will show how KRIMP uses this pattern set to model the database, and that it leads to drastic reductions of the frequent pattern set.

[1]This is part of an edited version of a paper submitted as: [86]. *Krimp : Mining Itemsets that Compress.* J. Vreeken, M. van Leeuwen, and A.P.J.M. Siebes. To: Journal on Data Mining and Knowledge Discovery.

3.1 Introduction

In the previous chapter, we have discussed a successful approach within the field of data mining: pattern mining. The aim of pattern mining is to find those patterns from a database that are potentially interesting to the user. In a plethora of cases, this concerns finding patterns that occur frequently in a database. The current frequent pattern mining techniques have become so efficient, that even large itemset databases pose little difficulty to generate the pattern sets.

However, it is argued that in order for pattern mining to be used for 'serious' applications some considerable issues need to be addressed [35]. One of these is demonstrated in the previous chapter: even in the simple case of itemset databases, the user gets swamped by huge pattern sets. This is likely to be worse in structured cases as more complex data types are likely to result in even larger pattern sets. Regrettably, interesting patterns are likely to have a reasonable low support, and this is often where the exponential growth of the pattern set takes its toll. Therefore, much research effort is focussed on creating more compact and interesting pattern sets.

The initial efforts to reduce the pattern set sizes have led to alternative interestingness measures such as *closed* and *maximal* [73, 77]. Although these measures obtain reductions, they often still result in fairly large pattern sets. Recently, there is an increased interest to define measures based on sets of patterns instead of measuring the interestingness of patterns in isolation. The rationale behind this, is that by this the measure can steer towards reducing the redundancy between the patterns within the resulting pattern set. The approaches discussed in the remainder of this thesis are of this pattern set selection nature.

It is argued that the main focus of data mining should be on parameter free algorithms that treat data mining problems from the perspective of compression [23]. The rationale behind this, is that compression can automatically find the best model that suits the data. In this chapter, we also utilise compression for a data mining problem, namely to find a compact and interesting pattern set. Using the Minimal Description Length (MDL) principle, the aim is to find that set of patterns that together describe the complete database best. In order to remain efficient, we introduce a heuristic algorithm called Krimp . We will show how Krimp uses a pattern set to model the database, and provide the details of its heuristics. Finally, we will show that Krimp leads to drastic reductions of the frequent pattern set.

3.2 The Minimum Description Length Principle

MDL (Minimal Description Length) [32,76] like MML (Minimal Message Length) [87] is a practical implementation of Kolmogorov Complexity [63]. In this work we focus on the usage of MDL, which can be roughly described as follows:

Given a set of models $\mathcal{H}$ [1], we consider the best model, $H \in \mathcal{H}$, to be that one that minimises:

$$L(H) + L(\mathcal{D} \mid H)$$

in which:

- $L(H)$ is the length, in bits, of the description of H, and
- $L(\mathcal{D} \mid H)$ is the length, in bits, of the database when encoded by H.

This form of MDL is called crude MDL, or two-part MDL, as opposed to refined MDL [33] where the model and the data are encoded together. As we are interested in the contents of the model, a set of patterns, we use the crude MDL approach.

3.3 Encoding the Database

In order to use MDL, we need to provide practical implementation of our models $\mathcal{H}$ for our data $\mathcal{D}$. This includes the encoding of $\mathcal{D}$ given a specific model H.

The models in our approach are *code tables*. A *code table* is a two-part table, on the left side it contains patterns and on the right side it contains the respective codes.

Definition 10 (Code Table)**.** *Given a set of items $\mathcal{I}$, and a set of codes $\mathcal{C}$ a* code table *is a two-column table such that:*

- *The left column contains patterns, that is, subsets over $\mathcal{I}$. This column contains at least all singleton elements.*
- *The right column contains elements from $\mathcal{C}$, such that each element of $\mathcal{C}$ occurs only once.*

In future notation we introduce some simplicity. We say that an itemset X from the powerset of $\mathcal{I}$, $X \in \mathcal{P}(\mathcal{I})$, occurs in CT, denoted by $X \in CT$, iff X occurs in the left column of CT. Similarly, we say that a code $C \in \mathcal{C}$ occurs in CT, denoted by $C \in CT$, iff the code C occurs in the right column of CT.

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

The set of all itemsets within the code table, $\{X \in CT\}$ is called the *coding set: CS*. The number of elements in CT is denoted as $|CT|$: $|CT| = |\{X \in CT\}|$. Recall that the set of all patterns containing exactly one item, are called the *singletons*: $\{X \in \mathcal{I}\}$. Furthermore, $|CT \setminus \mathcal{I}|$ indicates the number of non-singletons in the code table.

Now that we have defined our model, how do we use it to encode our database $\mathcal{D}$? In this chapter, our database $\mathcal{D}$ is a set of individual transactions t over $\mathcal{I}$: $t \in \mathcal{P}(\mathcal{I})$. Each transaction t over $\mathcal{I}$ given a code table CT is covered using the cover function: $cover(t, CT)$. This function selects from CT a number of itemsets to cover t, and returns a disjoint set of elements of CT that cover t. More formally:

Definition 11 (cover)**.** *Let $\mathcal{D}$ be a database over a set of items $\mathcal{I}$, t a transaction drawn from $\mathcal{D}$, let $\mathcal{CT}$ be the set of all possible code tables over $\mathcal{I}$, and CT a code table with $CT \in \mathcal{CT}$. Then $cover : \mathcal{CT} \times \mathcal{P}(\mathcal{I}) \to \mathcal{P}(\mathcal{P}(\mathcal{I}))$ is a cover function iff it returns a set of itemsets such that:*

1. *$cover(CT, t)$ is a subset of CS, the coding set of CT, i.e. $X \in cover(CT, t) \to X \in CT$,*
2. *if $X, Y \in cover(CT, t)$, then either $X = Y$, or $X \cap Y = \varnothing$,*
3. *the union of all $X \in cover(CT, t)$ equals t, that is, $t = \bigcup_{X \in cover(CT,t)} X$.*

We say that $cover(CT, t)$ covers t. Note that there exists always a well-defined cover function on any code table CT over $\mathcal{I}$ for any transaction t, as CT always contains all singleton elements from $\mathcal{I}$.

As our database $\mathcal{D}$ contains individual transactions, we encode $\mathcal{D}$ using CT simply by replacing all transactions t in $\mathcal{D}$ by the codes of the itemsets in the cover of t,

$$t \to \{code_{CT}(X) \mid X \in cover(CT, t)\}.$$

Note that to ensure that we can decode an encoded database uniquely we assume that $\mathcal{C}$ is a prefix code, in which no code is the prefix of another code [20]. (Confusingly, such codes are also known as prefix-free codes [63].)

The aim for MDL is to find that model that compresses the data best. Therefore we need to assign shorter codes to those elements in CT that have a higher influence in shortening the database length; namely the more frequently used code table elements. That is, we should use an optimal prefix code. Note that we are not so much interested in the materialised codes, but only in the complexities of the model and the encoded data given that model. Therefore, we are only interested in the lengths of the codes. As there exists a nice correspondence between code lengths and probability distributions (see, e.g. [63]), we can calculate the optimal code lengths through the Shannon entropy.

Theorem 1. *Let P be a distribution on some finite set $\mathcal{D}$, there exists an optimal prefix code $\mathcal{C}$ on $\mathcal{D}$ such that the length of the code for $d \in \mathcal{D}$, 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 length for data sets drawn according to P. (For the proof, please refer to Theorem 5.4.1 in [20])

The optimality property means that we introduce no bias using this code length. The probability distribution induced by a cover function is, of course, simply given by the relative usage frequency of each of the itemsets in the code table. To determine this, we need to know how often a certain code is used. We define the usage count of an itemset $X \in CT$ as the number of transactions t from $\mathcal{D}$ where X is used to cover. Normalised, this frequency represents the probability that the code is used in the encoding of an arbitrary $t \in \mathcal{D}$. The optimal code length [63] then is $-\log$ of this probability, and a code table is optimal if all its codes have their optimal length. More formally, we have the following definition.

Definition 12. *Let $\mathcal{D}$ be a transaction database over a set of items $\mathcal{I}$, $\mathcal{C}$ a prefix code,* cover *a cover function, and CT a code table over $\mathcal{I}$ and $\mathcal{C}$. The usage count of an itemset $X \in CT$ is defined as*

$$usage_{\mathcal{D}}(X) = |\{t \in \mathcal{D} \mid X \in cover(CT, t)\}|.$$

The probability of $X \in CT$ being used in the cover of an arbitrary transaction $t \in \mathcal{D}$ is thus given by

$$P(X \mid \mathcal{D}) = \frac{usage_{\mathcal{D}}(X)}{\sum_{Y \in CT} usage_{\mathcal{D}}(Y)}$$

The $code_{CT}(X)$ for $X \in CT$ is optimal for $\mathcal{D}$ iff

$$L(code_{CT}(X)) = |code_{CT}(X)| = -\log(P(X \mid \mathcal{D})).$$

A code table CT is code-optimal for $\mathcal{D}$ iff all its codes,

$$\{code_{CT}(X) \mid X \in CT\},$$

are optimal for $\mathcal{D}$.

From now onward we assume that code tables are code-optimal for the database they are induced on, unless we state differently. Now, for any database $\mathcal{D}$ and a code table CT over the same set of items $\mathcal{I}$ we can compute $L(\mathcal{D} \mid CT)$. It is simply the summation of the encoded lengths of the transactions. The encoded length of a transaction is simply the sum of the lengths of the codes of the itemsets in its cover. In other words, we have the following trivial lemma.

Lemma 1. *Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$, CT be a code table over $\mathcal{I}$ and code-optimal for $\mathcal{D}$,* cover *a cover function, and* usage *the usage function for* cover*.*

1. *For any $t \in \mathcal{D}$ its encoded length, in bits, denoted by $L(t \mid CT)$, is*

$$L(t \mid CT) = \sum_{X \in cover(CT,t)} L(code_{CT}(X)).$$

2. *The encoded length of $\mathcal{D}$, in bits, when encoded by CT, denoted by $L(\mathcal{D} \mid CT)$, is*

$$L(\mathcal{D} \mid CT) = \sum_{t \in \mathcal{D}} L(t \mid CT).$$

With Lemma 1, we can compute $L(\mathcal{D} \mid H)$. To use the MDL principle, we still need to know what $L(H)$ is, i.e. the encoded length of a code table.

Recall that a code table is a two-column table consisting of itemsets and codes. As we know the length of each of the codes, the encoded length of the second column is easily determined: it is simply the sum of the lengths of the codes. For encoding the itemsets, the first column, we have to make a choice.

A naïve option would be to encode each item with a binary integer encoding, that is, using $\log(\mathcal{I})$ bits per item. Clearly, this is hardly optimal; there is no difference in encoded length between highly frequent and infrequent items.

A better choice is to encode the itemsets using the codes of the simplest code table, i.e. the code table that contains only the singleton itemsets $X \in \mathcal{I}$. This code table, with optimal code lengths for database $\mathcal{D}$, is called the standard code table for $\mathcal{D}$, denoted by ST. It is the optimal encoding of $\mathcal{D}$ when nothing more is known than just the frequencies of the individual items; it assumes the items to be fully independent. As such, it provides a practical bound: ST provides the simplest, independent, description of the data that compresses much better than a random code table. This encoding allows us to reconstruct the database up to the names of the individual items.

Definition 13. *Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$ and CT a code table that is code-optimal for $\mathcal{D}$. The length of CT in bits, denoted by $L(CT \mid \mathcal{D})$, is given by*

$$L(CT \mid \mathcal{D}) = \sum_{X \in CT : usage_{\mathcal{D}}(X) \neq 0} |code_{ST}(X)| + |code_{CT}(X)|.$$

Note that we do not take itemsets with zero usage into account. Such itemsets are not used to code. We use $L(CT)$ wherever $\mathcal{D}$ is clear from context.

With these results we know the total length of our encoded database. It is simply the length of the encoded database plus the length of the code table. That is, we have the following result.

Definition 14. *Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$ and let CT be a code table that is code-optimal for $\mathcal{D}$ and* cover *a cover function. The total compressed length of the encoded database and the code table, in bits, denoted by $L(\mathcal{D}, CT)$ is given by*

$$L(\mathcal{D}, CT) = L(\mathcal{D} \mid CT) + L(CT \mid \mathcal{D}).$$

Now that we know how to compute $L(\mathcal{D}, CT)$, we can formalise our problem using MDL. Before that, we discuss three design choices we did not mention so far, because they do not influence the total compressed length of a database.

3.4 Finding the Code Table

Our goal is to find the set of itemsets that best describe the database $\mathcal{D}$. Recall that the set of itemsets of a code table, i.e. $\{X \in CT\}$, is called the coding set CS. Given a coding set, a cover function and a database, a (code-optimal) code table CT follows automatically.

Given a set of itemsets $\mathcal{F}$, the problem is to find a subset of $\mathcal{F}$ which leads to a minimal encoding; where minimal pertains to all possible subsets of $\mathcal{F}$. To make sure this is possible, $\mathcal{F}$ should contain at least the singleton itemsets $X \in \mathcal{I}$. We will call such a set, a candidate set. By requiring the smallest coding set, we make sure the coding set contains no unused non-singleton elements, i.e. $usage_{CT}(X) > 0$ for any non-singleton itemset $X \in CT$. More formally, we define the problem as follows.

Minimal Coding Set Problem. *Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a dataset over $\mathcal{I}$,* cover *a cover function, and $\mathcal{F}$ a candidate set. Find the smallest coding set $CS \subseteq \mathcal{F}$ such that for the corresponding code table CT the total encoded length, $L(\mathcal{D}, CT)$, is minimal.*

A solution for the Minimal Coding Set Problem allows us to find the 'best' coding set from a given collection of itemsets, e.g. (closed) frequent itemsets for a given minimal support. For example, if $\mathcal{F} = \{X \in \mathcal{P}(\mathcal{I}) \mid sup_{\mathcal{D}}(X) > 0\}$, i.e. when $\mathcal{F}$ consists of all itemsets that occur in the data, there exists no candidate set $\mathcal{F}'$ that results in a smaller total encoded length. Hence, in this case the solution is truly the minimal coding set for $\mathcal{D}$ and *cover*.

In order to solve the Minimal Coding Set Problem, we have to find the optimal code table and cover function. To this end, we have to consider a humonguous search space.

The number of coding sets does not depend on the actual database, and nor does the number of possible cover functions. Because of this, we can compute the size of our search space rather easily.

A coding set contains the singleton itemsets plus an almost arbitrary subset of $\mathcal{P}(\mathcal{I})$. Almost, since we are not allowed to choose the $|\mathcal{I}|$ singleton itemsets.

In other words, there are

$$\sum_{k=0}^{2^{|\mathcal{I}|}-|\mathcal{I}|-1} \binom{2^{|\mathcal{I}|}-|\mathcal{I}|-1}{k}$$

possible coding sets. In order to determine which one of these minimises the total encoded length, we have to consider all corresponding (code-optimal) code tables using every possible cover function. Since every itemset $X \in CT$ can occur only once in the cover of a transaction and no overlap between the itemsets is allowed, this translates to traversing the code table once for every transaction. However, as each possible code table order may result in a different cover, we have to test every possible code table order per transaction to cover. Since a set of n elements admits $n!$ orders, the total size of the search space is as follows.

Lemma 2. *For one transaction over a set of items $\mathcal{I}$, the number of possible ways to cover it is given by NCP($\mathcal{I}$):*

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

So, even for a rather small set $\mathcal{I}$ and a database of only one transaction, the search space we are facing is already huge. Table 3.1 gives an approximation of NCP for the first few sizes of $\mathcal{I}$. Clearly, the search space is far too large to consider exhaustively.

To make matters worse, there is no useable structure that allows us to prune levelwise as the attained compression is not monotone w.r.t. the addition of itemsets. So, without calculating the usage of the itemsets in CT, it is generally impossible to call the effects (improvement or degrading) on the compression when an itemset is added to the code table. This can be seen as follows.

Suppose a database $\mathcal{D}$, itemsets X and Y such that $X \subset Y$, and a coding set CS, all over $\mathcal{I}$. The addition of X to CS, can lead to a degradation of the compression, first and foremost as X may add more complexity to the code table than is compensated for by using X in encoding $\mathcal{D}$. Second, X may get in 'the way' of itemsets already in CS, as such providing those itemsets with lower usage, longer codes and thus leading to massively worse compression.

Table 3.1: The number of cover possibilities for a database of one (1) transaction over $\mathcal{I}$.

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

Instead, let us consider adding Y. While more complex, exactly those items $Y \setminus X$ may replace the hindered itemsets. As such Y may circumvent getting 'in the way', and thus lead to an improved compression. However, this can just as well be the other way around, as exactly those items can also lead to low usage and/or overlap with other/more existing itemsets in CS.

3.5 Algorithms

In this section we present algorithms for solving the problem formulated in the previous section. As shown above, the search space one needs to consider for finding the optimal code table is far too large to be considered exhaustively. We therefore have to resort to heuristics.

To cut down a large part of the search space, we use the following simple greedy search strategy:

- Start with the standard code table ST, containing only the singleton itemsets $X \in \mathcal{I}$.
- Add the itemsets from $\mathcal{F}$ one by one. If the resulting codes lead to a better compression, keep it. Otherwise, discard the set.

To turn this sketch into an algorithm, some choices have to be made. First, in which order are we going to encode a transaction? So, what cover function are we going to employ? Second, in which order do we add the itemsets? Finally, do we prune the newly constructed code table before we continue with the next candidate itemset or not?

Before we discuss each of these questions, we briefly describe the initial encoding. This is, of course, the encoding with the standard code table. For this, we need to construct a code table from the elements of $\mathcal{I}$. The algorithm called Standard Code Table, given in pseudo-code as Algorithm 1, returns such a code table. It takes a set of items and a database as parameters and

Algorithm 1 The Standard Code Table Algorithm

Input: A transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$.
Output: The standard code table CT for $\mathcal{D}$.

1. $CT \leftarrow \varnothing$
2. **for all** $X \in \mathcal{I}$ **do**
3. insert X into CT
4. $usage_{\mathcal{D}}(X) \leftarrow sup_{\mathcal{D}}(X)$
5. $code_{CT}(X) \leftarrow$ optimal code for X
6. **end for**
7. **return** CT

returns a code table. Note that for this code table all cover functions reduce to the same, namely the cover function that replaces each item in a transaction with its singleton itemset. As the singleton itemsets are mutually exclusive, all elements $X \in \mathcal{I}$ will be used $sup_{\mathcal{D}}(X)$ times by this cover function.

Standard Cover Order

From the problem complexity analysis in the previous section it is quite clear that finding an optimal cover of the database is impossible, even if we are given the optimal set of itemsets as the code table: examining all $|CT|!$ possible permutations is already virtually impossible for one transaction, let alone expanding this to all possible combinations of permutations for all transactions.

We therefore employ a heuristic and introduce a standard cover function which considers the code table in a fixed order. The pseudo-code for this Standard Cover function is given as Algorithm 2. For a given transaction t, the code table is traversed in a fixed order. An itemset $X \in CT$ is included in the cover of t iff $X \subseteq t$. Then, X is removed from t and the process continues to cover the uncovered remainder, i.e. $t \setminus X$. Using the same order for every transaction drastically reduces the complexity of the problem, but leaves the choice of the order.

Again, considering all possible orders would be best, but is impractical at best. When choosing an order, we should take into account that the order in which we consider the itemsets may make it easier or more difficult to insert candidate itemsets into an already sorted code table.

We choose to sort the elements $X \in CT$ first decreasing on length, second decreasing on support in $\mathcal{D}$ and thirdly lexicographically increasing to make it a total order. To describe the order compactly, we introduce the following notation. We use $\downarrow$ to indicate an attribute is sorted descending, and $\uparrow$ to indicate it is sorted ascending:

Algorithm 2 The Standard Cover Algorithm

Input: Transaction $t \in \mathcal{D}$ and code table CT, with CT and $\mathcal{D}$ over a set of items $\mathcal{I}$.

Output: A cover of t using non-overlapping elements of CT.

1. $S \leftarrow$ smallest element X of CT in Standard Cover Order for which $X \subseteq t$.
2. **if** $t \setminus S = \varnothing$ **then**
3. $\quad Res \leftarrow \{S\}$
4. **else**
5. $\quad Res \leftarrow \{S\} \cup$ StandardCover$(t \setminus S, CT)$
6. **end if**
7. **return** Res

$$|X| \downarrow sup_{\mathcal{D}}(X) \downarrow lexicographically \uparrow$$

We call this the Standard Cover Order. The rationale is as follows. To reach a good compression we need to replace as many individual items as possible, by as few and short as possible codes. The above order gives priority to long itemsets, as these can replace as many as possible items by just one code. Further, we prefer those itemsets that occur frequently in the database to be used as often as possible, resulting in high usage values and thus short codes. We rely on MDL not to select overly specific itemsets, as such sets can only be infrequently used and would thus receive relatively long codes.

Standard Candidate Order

Next, we address the order in which candidate itemsets will be regarded. Preferably, the candidate order should be in concord with the cover strategy detailed above. We therefore choose to sort the candidate itemsets such that long, frequently occurring itemsets are given priority. Again, to make it a total order we thirdly sort lexicographically. So, we sort the elements of $\mathcal{F}$ as follows:

$$sup_{\mathcal{D}}(X) \downarrow |X| \downarrow lexicographically \uparrow$$

We refer to this as the Standard Candidate Order. The rationale for it is as follows. Itemsets with the highest support, those with potentially the shortest codes, end up at the top of the list. Of those, we prefer the longest sets first, as these will be able to replace as many items as possible. This provides the search strategy with the most general itemsets first, providing ever more specific itemsets along the way.

Algorithm 3 Krimp Algorithm

Input: A transaction database $\mathcal{D}$ and a candidate set $\mathcal{F}$, both over a set of items $\mathcal{I}$.

Output: A heuristic solution to the Minimal Coding Set Problem, code table CT.

1. $CT \leftarrow$ Standard Code Table($\mathcal{D}$)
2. $\mathcal{F}_0 \leftarrow \mathcal{F}$ in Standard Candidate Order
3. **for all** $F \in \mathcal{F}_0 \setminus \mathcal{I}$ **do**
4. $\quad CT_c \leftarrow (CT \cup F)$ in Standard Cover Order
5. $\quad$ **if** $L(\mathcal{D}, CT) < L(\mathcal{D}, CT_c)$ **then**
6. $\quad\quad CT \leftarrow CT_c$
7. $\quad$ **end if**
8. **end for**
9. **return** CT

A welcome advantage of the standard orders for both the cover function and the candidate order is that we can easily keep the code table sorted. First, the length of an itemset is readily available. Second, with this candidate order we know that any candidate itemset for a particular length will have to be inserted after any already present code table element with the same length. Together, this means that we can insert a candidate itemset at the right position in the code table in O(1) if we store the code table elements in an array (over itemset size) of lists.

The Krimp algorithm

We now have the ingredients for the basic version of our compression algorithm:

- Start with the standard code table ST;
- Add the candidate itemsets from $\mathcal{F}$ one by one. Each time, take the itemset that is maximal w.r.t. the standard candidate order. Cover the database using the standard cover algorithm. If the resulting encoding provides a smaller compressed length, keep it. Otherwise, discard it permanently.

This basic scheme is formalised as the Krimp algorithm given as Algorithm 3. For the choice of the name: 'krimp' is Dutch for 'to shrink'. The Krimp pattern selection process is illustrated in Figure 3.1. Krimp takes as input a database $\mathcal{D}$ and a candidate set $\mathcal{F}$. The result is the best code table the algorithm has seen, w.r.t. the Minimal Coding Set Problem.

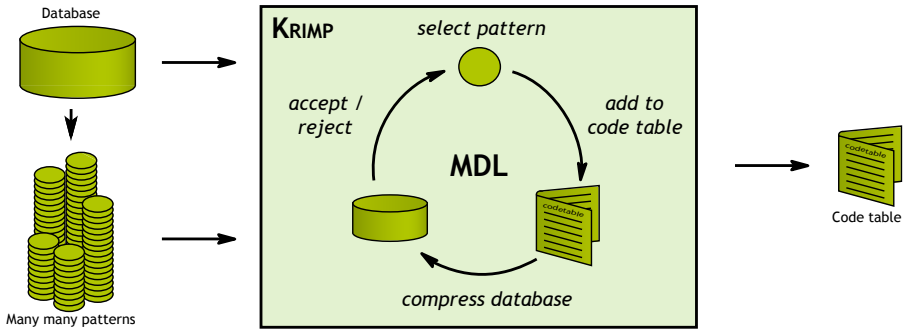


Figure 3.1: The Krimp algorithm takes the large set of frequent patterns derived from the database, and selects a small set of interesting patterns.

Now, it may seem that each iteration of KRIMP can only lessen the usage of an itemset in CT. For, if $F_1 \cap F_2 \neq \varnothing$ and F_2 is used before F_1 by the standard cover function, the usage of F_1 will go down (provided the support of F_2 does not equal zero). While this is true, it is not the whole story. Because, what happens if we now add an itemset F_3, which is used before F_2 such that:

$$F_1 \cap F_3 = \varnothing \text{ and } F_2 \cap F_3 \neq \varnothing$$

The usage of F_2 will go down, while the usage of F_1 will go up again; by the same amount, actually. So, taking this into consideration, even code table elements with zero usage cannot be removed without consequence. However, since they are not used in the actual encoding, they are not taken into account while calculating the total compressed length for the current solution.

In the end, itemsets with zero usage can be safely removed though. After all, they do not code, so they are not part of the optimal answer that should consist of the smallest coding set. Since the singletons are required in a code table by definition, these remain.

Pruning

That said, we can't be sure that leaving itemsets with a very low usage count in CT is the best way to go. As these have a very small probability, their respective codes will be very long. Such long codes may make better code tables unreachable for the greedy algorithm; it may get stuck in a local optimum. As

an example, consider the following three code tables:

$$\begin{aligned} CT_1 &= \{\{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\} \\ CT_2 &= \{\{X_1, X_2, X_3\}, \{X_1, X_2\}, \{X_1\}, \{X_2\}, \{X_3\}\} \\ CT_3 &= \{\{X_1, X_2, X_3\}, \{X_1\}, \{X_2\}, \{X_3\}\} \end{aligned}$$

Assume that $sup_{\mathcal{D}}(\{X_1, X_2, X_3\}) = sup_{\mathcal{D}}(\{X_1, X_2\}) - 1$. Given these facts, standard Krimp will never consider CT_3, but it is very well possible that $L(\mathcal{D}, CT_3) < L(\mathcal{D}, CT_2)$ and that CT_3 provides access to a branch of the search space that is otherwise left unvisited. To allow for searching in this direction, we can prune the code table that Krimp is considering.

There are many possibilities to this end. The most obvious strategy is to check the attained compression of all valid subsets of CT including the candidate itemset F, i.e. $\{CT_p \subseteq CT \mid F \in CT_p \wedge \mathcal{I} \subset CT_p\}$, and choose CT_p with minimal $L(\mathcal{D}, CT_p)$. In other words, prune when a candidate itemset is added to CT, but before the acceptance decision. Clearly, such a pre-acceptance pruning approach implies a huge amount of extra computation. Since we are after a fast and well-performing heuristic we do not consider this strategy.

A more efficient alternative is post-acceptance pruning. That is, we only prune when F is accepted: when candidate code table $CT_c = CT \cup F$ is better than CT, i.e. $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$, we consider its valid subsets. This effectively reduces the pruning search space, as only few candidate itemsets will be accepted.

To cut the pruning search space further, we do not consider all valid subsets of CT, but iteratively consider itemsets $X \in CT$ of which $usage_{\mathcal{D}}(X)$ has decreased for removal. The rationale is that for these itemsets we know that their code lengths have increased; therefore, it is possible that these sets now harm the compression.

In line with the standard order philosophy, we first consider the itemset with the smallest usage and thus the longest code. If by pruning an itemset the total encoded length decreases, we permanently remove it from the code table. Further, we then update the list of prune candidates with those itemsets whose usage consequently decreased. This post-acceptance pruning strategy is formalised in Algorithm 4. We refer to the version of Krimp that employs this pruning strategy (which would be on line 6 of Algorithm 4) as Krimp with pruning.

Algorithm 4 Code Table Post-Acceptance Pruning

Input: Code tables CT_c and CT, for a transaction database $\mathcal{D}$ over a set of items $\mathcal{I}$, where $\{X \in CT\} \subset \{Y \in CT_c\}$ and $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$.
Output: Pruned code table CT_p, such that $L(\mathcal{D}, CT_p) \leq L(\mathcal{D}, CT_c)$ and $CT_p \subseteq CT_c$.

1. $PruneSet \leftarrow \{X \in CT \mid usage_{CT_c}(X) < usage_{CT}(X)\}$
2. **while** $PruneSet \neq \varnothing$ **do**
3. $\quad PruneCand \leftarrow X \in PruneSet$ with lowest $usage_{CT_c}(X)$
4. $\quad PruneCand \leftarrow PruneSet \setminus PruneCand$
5. $\quad CT_p \leftarrow CT_c \setminus PruneCand$
6. $\quad$ **if** $L(\mathcal{D}, CT_p) \leq L(\mathcal{D}, CT_c)$ **then**
7. $\quad\quad PruneSet \leftarrow PruneSet \cup \{X \in CT_p \mid usage_{CT_p}(X) < usage_{CT_c}(X)\}$
8. $\quad\quad CT_c \leftarrow CT_p$
9. $\quad$ **end if**
10. **end while**
11. **return** CT_c

Complexity

Here we very briefly mention the complexity of the Krimp algorithm. For a more detailed analysis on the complexity of Krimp see [61, 84].

In the worstcase, all candidate patterns can be added to our code table. However due to MDL, the number of elements in the code table is usually very small, $|CT| \ll |\mathcal{D}| \ll |\mathcal{F}|$, in particular when pruning is enabled. In addition, by using techniques such as hash tables, bitmaps, and more sophisticated cover algorithms, we can restrain the estimate of the time complexity of Krimp with or without pruning to:

$$\mathrm{O}(|\mathcal{F}| \log |\mathcal{F}| + |\mathcal{F}|).$$

As for the memory requirements of the Krimp algorithm, its worstcase estimate is:

$$\mathrm{O}(|\mathcal{F}| + |\mathcal{D}| + |\mathcal{F}|).$$

Again, as the code table is dwarfed by the size of the database, it can be regarded a (small) constant. The major part is the storage of the candidate code table elements. Sorting these can be done in place. As it is iterated in order, it can be handled from the hard drive without much performance loss. Preferably, the database is kept resident, as it is covered many many times.

Table 3.2: For all datasets the candidate set $\mathcal{F}$ was mined with minsup = 1, and Krimp with post-acceptance pruning was used. For Krimp , the size of the resulting code table (minus the singletons), the compression ratio and the runtime is given. The compression ratio is the encoded length of the database with the obtained code table divided by the encoded length with the standard code table.

				Krimp	
Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert CT \setminus \mathcal{I}\rvert$	$\frac{L(\mathcal{D},CT)}{L(\mathcal{D},ST)}\%$
Adult	48842	58461763	468	1303	24.4
Chess (kr-k)	28056	373421	58	1684	61.6
Led7	3200	15250	24	152	28.6
Letter	20000	580968767	102	1780	35.7
Mushroom	8124	5574930437	119	442	20.6
Pen	10992	459191636	86	1247	42.3

3.6 Results

We now present some results on a small number of datasets to provide the reader with some measure and intuition on the performance of Krimp. To this end, we ran Krimp with post-acceptance pruning on six datasets, using all frequent itemsets mined at minsup = 1 as candidates. The results of these experiments are shown in Table 3.2. Per dataset, we show the number of transactions and the number of candidate itemsets. From these latter figures, the problem of the pattern explosion becomes clear: up to 5.5 billion itemsets can be mined from the Mushroom database, which consists of only 8124 transactions. It also shows that Krimp successfully battles this explosion, by selecting only hundreds of itemsets from millions up to billions. For example, from the 5.5 billion for Mushroom, only 442 itemsets are chosen; a reduction of 7 orders of magnitude.

For the other datasets, we observe the same trend. In each case, fewer than 2000 itemsets are selected, and reductions of many orders of magnitude are attained. The number of selected itemsets depends mainly on the characteristics of the data. These itemsets, or the code tables they form, compress the data to a fraction of its original size. This indicates that very characteristic itemsets are chosen, and that the selections are non-redundant. Given this small sample of results, we now know that indeed few and non-redundant itemsets are selected by Krimp , in number many orders smaller than the complete frequent itemset collections.

3.7 Related Work

MDL was introduced by Rissanen [76] as a noise-robust model selection technique. In the limit, refined MDL is asymptotically the same as the Bayes Information Criterion (BIC), but the two may differ (strongly) on finite data samples [33]. We are not the first to use MDL, nor are we the first to use MDL in data mining or machine learning. Many, if not all, data mining problems can be related to Kolmogorov Complexity, which means they can be practically solved through compression [23], e.g. clustering (unsupervised learning), classification (supervised learning), distance measurement. Other examples include defining a parameter-free distance measure on sequential data [45, 46], discovering communities in matrices [15], and evolving graphs [80].

Most, if not all pattern mining approaches suffer from the pattern explosion. As discussed before, its cause lies primarily in the large redundancy in the returned pattern sets. This has long since been recognised as a problem and has received ample attention.

Recently, the approach of finding small subsets of informative patterns that describe the database has attracted a significant amount of research [8, 51, 68]. First, there are the methods that provide a lossy description of the data. These strive to describe just part of the data, and as such may overlook important interactions. Summarization as proposed by Chandola and Kumar is a compression-based approach that identifies a group of itemsets such that each transaction is summarized by one itemset with as little loss of information as possible [16]. Wang and Karypis find summary sets, sets of itemsets that contain the largest frequent itemset covering each transaction [89].

Pattern Teams [52] are groups of most-informative length-k itemsets [51]. These are exhaustively selected through an external criterion, e.g. joint entropy or classification accuracy. As this approach is computationally intensive, the number of team members is typically < 10. Bringmann and Zimmermann proposed a similar selection method that can consider larger pattern sets [8]. However, it also requires the user to choose a quality measure to which the pattern set has to be optimized, unlike our parameter-free and lossless method.

Tiling is closely related to our approach [26]. A tiling is a cover of the database by a group of (overlapping) itemsets. Itemsets with maximal uncovered area are selected, i.e. as few as possible itemsets cover the data. Unlike our approach, model complexity is not explicitly taken into account. Another major difference in the outcome is that Krimp selects more specific (longer) itemsets. Xiang et al. proposed a slight reformulation of Tiling that allows tiles to also cover transactions in which not all its items are present [91].

Two approaches inspired by Krimp are Pack [82] and LESS [41]. Both approaches consider the data 0/1 symmetric, unlike here, where we only regard

items that are present (1s). LESS employs a generalised Krimp encoding to select only tens of low-entropy sets [40] as lossless data descriptions, but attains worse compression ratios than Krimp . Pack does provide a significant improvement in that regard. It employs decision trees to succinctly transmit individual attributes, and these models can be built from data or candidate sets. Typically, Pack selects many more itemsets than Krimp .

Our approach seems related to the set cover problem, as both try to cover the data with sets. Although NP-complete, fast approximation algorithms exist for set cover. These are not applicable for our setup though, as in set cover the complexity of the model is not taken into account. Another difference is that we do not allow overlap between itemsets. As optimal compression is the goal, it makes intuitive sense that overlapping elements may lead to shorter encodings, but it is not immediately clear how to achieve this in a fast heuristic.

Chapter 4

Reducing the Frequent Pattern Set

In the previous chapter we have introduced a radically different approach based on the Minimal Description Length (MDL) principle. For this we use the credo: *"A set of frequent patterns is interesting iff it gives a good compression of the database"*. We have shown that this approach results in itemset sets that are many orders of magnitude smaller than the set of all frequent itemsets [79].

The aim of this chapter is to extend this approach to frequent patterns on structured data. That is, to show that in the general case MDL-based compression picks a, relatively, small set of frequent patterns that describe the structured data well. We will show how KRIMP can be applied successfully on structured data, and utilise sequences and trees data types.

[1]This is an extended and edited version of an earlier paper published as: [6]. *Reducing the Frequent Pattern Set.* R.W. Bathoorn, A.C.M. Koopman, and A.P.J.M. Siebes. In: ICDM Workshops 2006, pages 55-59.

4.1 Introduction

In the previous chapter we have introduced a radically different approach based on the Minimal Description Length (MDL) principle [32]:

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

We have shown that this approach results in sets of frequent itemsets that are many orders of magnitude smaller than the set of all frequent itemsets [79]. That is, the resulting set is far smaller than, e.g., the set of closed frequent itemsets.

The aim of this chapter is to extend this approach to frequent patterns on structured data. That is, we show that in the general case, our 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 [66] and trees [18].

Although more MDL based methods for structured data appear in literature, the use of MDL here is different. While our focus in on finding a compact set of structured patterns that represent the database, other methods use MDL to find patterns within structured data. For example, SUBDUE [42] uses MDL to compress one graph to find hierarchical structures within the graph. Similarly, the ED algorithm [39] uses MDL on sequence data, and captures periodical patterns in one long sequence as opposed to a collection of sequences.

4.2 Structured Data and its Frequent Patterns

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

As mentioned already in the introduction, we focus on two types of structured data types: sequences and trees. There is a variety of ways in which one can define these structured data types. Our specific choice is based on two criteria:

1. As our goal is to have a lossless compression of the database, we have to cover each structured element in the database completely and exactly once. 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 with our test-data, which is web-mining data.

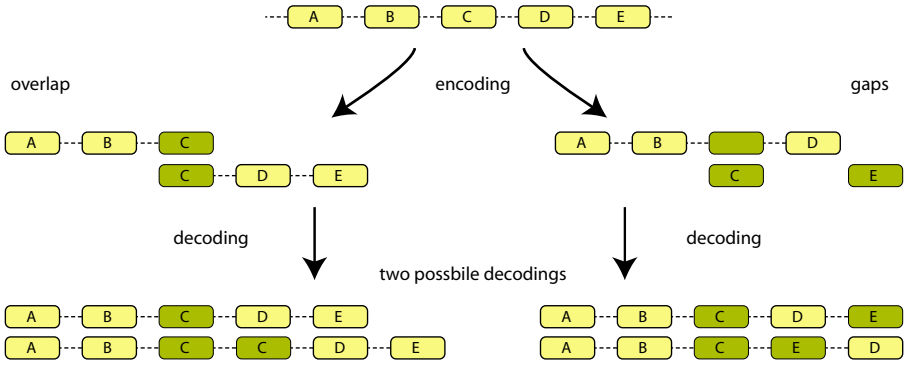


Figure 4.1: Ambiguity in the decoding caused by allowing overlap or gaps in the cover.

To ensure a lossless compression of the database we do not allow the selected structured patterns to overlap in the database cover. Without this restriction it would no longer be possible to perform a unique decoding of the database. To illustrate this restriction, we *do* allow overlap and observe the result when two sequences ABC and CDE are used to cover a sequence as shown in Figure 4.1. In this case after decoding, the original sequence transaction could have been $ABCDE$ (with overlap) or $ABCCDE$ (without overlap). Both are indistinguishable when looking at the coded string when overlap is allowed. Therefore, to solve this ambiguity we do not allow any overlap.

Another consideration to take into account for the lossless compression of a database is the use of gaps in patterns. Because gaps in general can have an arbitrary size, it is impossible to ensure an unique decoding. To illustrate this, Figure 4.1 shows an example where the patterns $AB * D, C$ and E that have been selected as our set of patterns. In this case, the original sequence could have been $ABCDE$ for a gap size of 1 or $ABCED$ for a gap size of 2. To prevent this ambiguity we do not allow gaps in our patterns. Naturally, these examples extend to other types of structured data, such as trees and graphs.

As a final note on structured data types, recall from Chapter 2 that we can write ordered rooted trees as strings using a SMILES notation [90]. In this notation, a ordered rooted tree T having a root A, and two children B and C, of which C has one child D is written as: $A(B)(C(D))$.

Structured Data: Sequences and Trees

In this chapter, our database is a bag of structured data types, that is, $\mathcal{D} = \{X\}$, where X is either a sequence or a tree. The basis of both sequences and trees are *events*. A sequence is an ordered set of such events, whereas we consider trees to be ordered rooted trees in which each node is labelled by one of the events [48]. For example, if the events are the web pages of a given website, a sequence would represent the path of pages a user has visited on that web site. A 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 of the data types are given in Chapter 2, see Definition 2 for the used sequence and Definition 6 for the used tree data type.

Patterns and Occurrences

As usual in structured data mining, the patterns we consider are themselves again structured data types, namely sequences and trees over Σ. As for our definition of an *occurrence*, it is crucial that we do not allow gaps. That is, we use the gapless sequence occurence as defined in Definition 4 for both the sequence and tree cases, as we will motivate shortly. As $\mathcal{D}$ is a bag of sequences or trees, $sup_{\mathcal{D}}(Y)$ is the sum of all occurrences of Y in all trees or sequences X from $\mathcal{D}$, i.e. $sup_{\mathcal{D}}(F) = |\{\text{occurrences of } F \text{ in } \mathcal{D}\}|$. A pattern is called frequent if it occurs more often than the given minimal support threshold: *minsup*.

4.3 Krimp for Structured Data

Lossless encoding

Having the patterns and their occurrence now defined, we will utilise these in an alternate version of the Krimp algorithm as defined in Chapter 3. The outline for the algorithm used in this chapter is roughly the same. The Minimum Description Length (MDL) principle is used to find the model, CT, that minimises $L(\mathcal{D}, CT)$. What makes things slightly more complex, is the fact that our database $\mathcal{D}$ and the set of candidate patterns $\mathcal{F}$ contain structured data and patterns.

A cornerstone element of Krimp , as introduced in Chapter 3, is the lossless encoding of $\mathcal{D}$ given a code table CT. In the case of itemsets, during the encoding phase, a code $code_{CT}(X)$ replaces an itemset X. When decoding, we can simply replace $code_{CT}(X)$ by the original itemset X and we are done. To apply a similar lossless MDL-encoding scheme to structured data, we need to make sure that the original order within the structured data of $\mathcal{D}$ is preserved

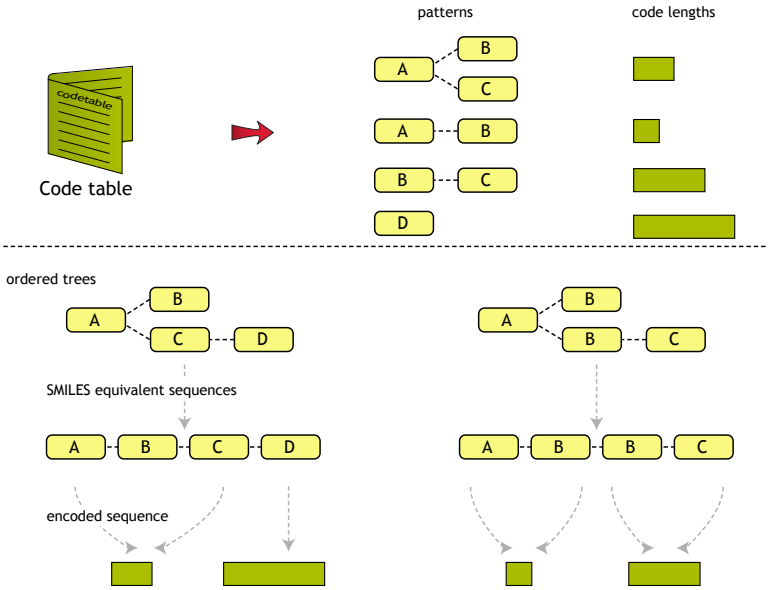


Figure 4.2: Lossless encoding of a database with ordered trees.

when decoding the encoded database. To this end we have to pre-process the data to make this possible.

Consider a tree transaction $T = A(B)(C(D))$ from a database $\mathcal{D}$ (see Figure 4.2 left). This tree could be partially covered by a pattern $F = A(B)(D)$ that has a code $code_{CT}(F)$. If we now partly encode T with F we obtain: $code_{CT}(F)(C)$. So far so good. However, when we decode the encoded transaction, we need to know to which leaf we would re-attach the remaining node C. We now have some alternatives: $A(B)(C(D))$, the correct one, and some faulty ones: $A(B)(C)(D)$, $A(B(C))(D)$, and $A(B)(D(C))$. This does not seem like a lossless encoding.

Therefore, in order to provide a lossless encoding of the database, we need to make a design choice. For the sequences, the restriction of gapless occurrences by itself already allows for a lossless encoding. For the trees however, we need to convert all ordered trees to their SMILES equivalent sequences. Then we do the same procedure as we do on the ordinary sequences. To continue our example with our lossless encoding, F would could *not* be used to cover. An example set of patterns that *could* be used to cover is $F' = A(B)(C)$ together with the singleton D.

Exact Solutions?

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

First, we have to consider all subsets of $\mathcal{F}$ that contain all alphabet elements. We have shown in Chapter 3 that this very quickly results in a search space of enormous propotions. In the case of sequences, we also need to consider all possible orders, and allow bags of items instead of just sets. The more complex a data type gets, the more possibilities we need to evaluate as candidates. In light of this, there are far too many possible code tables (and orders) to make exhaustive search feasible.

Second, there is structure in the search space, but it does not help. The only thing we know is that if $\mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \mathcal{F}$, then the best possible code table we can derive from $\mathcal{F}_2$ is at least as good as the best possible code table we can derive from $\mathcal{F}_1$.

All things considered, we will have to use heuristics to find a good code table. We will use the heuristics that are proven to work well in [79] and has been discussed in Chapter 3 and adapt them to this case.

That is, we use a set of structured patterns, the code table CT, as our model for the database $\mathcal{D}$. From $\mathcal{D}$ we derive an ordered set of structured frequent patterns $\downarrow \mathcal{F}$, and pick, in order, a candidate pattern F that is added to the code table CT. This code table is used to cover the database, where the occurrence checking is now based on gapless SMILES sequences.

4.4 Related Work

Another MDL-based method that is known in literature that extracts structured patterns from a database is called SUBDUE [42]. The rough sketch of the SUBDUE algorithm is as follows: it replaces structures within a single graph by one node, this replacement step is repeated until no significant new structures can be found. This results in an compressed hierarchical graph. Between SUBDUE and Krimp , there are some striking differences that we would like to address. First, the goal of SUBDUE is different as it aims at finding hierarchical patterns that compress a single graph, in contrast to selecting an interesting set of patterns from the frequent pattern set. The prime focus on single graphs makes finding patterns over sets of graphs not feasible without significant adjustments to SUBDUE.

Second, a more important difference with SUBDUE is the fact that SUBDUE performs candidate generation based on MDL heuristics opposed to our heuristic exploration of the complete frequent pattern set. Heuristic candidate generation can lead to different results, as candidates with a less optimal com-

pressed length will not be expanded to a new set of candidates. However, given the erratic nature of the search space, it could very well be that potentially interesting patterns will be pruned.

In SUBDUE, the set of patterns that is selected to be expanded by a single node is limited by the fixed *beam size*. Therefore, as the number of candidates that are allowed per pattern size is fixed, it is likely to happen that pruning will cut away interesting patterns. For very skewed databases, such as those used in our experiments, the large set of small sized patterns will only contain a few patterns that can be expanded to larger patterns. In order to include these interesting candidates in the beam, SUBDUE would need to be able to choose these few patterns based on their contribution to compression of the complete graph. However, as the contribution of these patterns and their extensions to the MDL compression is far from monotone, it is likely that they are pruned.

In their earlier work, the authors of SUBDUE noted this possible lack of finding patterns as an effect of their heuristics. Therefore, in our work we prefer to remove this heuristic effect, in order to evaluate more patterns that can lead to a better compression of the database.

In the field of sequence mining, MDL is also utilised by a mining algorithm called Episode Discovery (ED). The specific focus of ED is to discover periodic patterns within a sequence. Note that this poses a restriction on the code table elements opposed to our used sequence definition (see Definition 2). Moreover, the initial candidate set of the ED algorithm consists of maximal frequent episodes, opposed to all frequent episodes in the case of Krimp . For each maximal candidate episode, ED determines the MDL compression ratio and uses it to mark potentially interesting candidates.

4.5 Experiments

We have applied our algorithm to publicly available data sets from the web-mining field (see Table 4.1). As we are interested in the reduction of the pattern set and not so much the actual compression ratios of the database length, 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 present results for both the sequence and the tree data types. The depicted 2 sets of histograms are representative for all other conducted experiments (see Figures 4.3 and 4.5). Our databases are skewed: some transactions are around 100 times larger than others, which makes it hard for MDL to compress them. Moreover, the large number of singletons in combination with the small average transaction size indicates that it is more difficult to attain high compression ratios. In short, the chosen data sets pose an interesting challenge.

Table 4.1: Database characteristics. For each database $\mathcal{D}$, we have listed the number of records $|\mathcal{D}|$, the average number of items within the record $\overline{|X|}$, and the number of singletons ($|\mathcal{I}|$).

data type	$\mathcal{D}$	$\lvert\mathcal{D}\rvert$	$\overline{\lvert X\rvert}$	$\lvert\mathcal{I}\rvert$
sequences	KDDcup	234,954	3	835
trees	logml	8074	8	9060
	US 2430	7409	8	9284
	US 304	7628	7	8928

Sequences

For the experiments on sequence data we selected the KDD Cup 2000 data set that consists of clickstream and customer data of an e-commerce retail web site (see Table 4.1) [54]. It contains 777,480 clicks divided over 234,954 sequences. From the clickstream data we derived a collection of frequent sequences without gaps. We limited the window size, which relates to the maximum span in the database of a pattern, to 60 and 120 seconds, which resulted in two sets of frequent sequences that fit within these window sizes. Given each set we use the structured version of Krimp to derive the code table that is used to compress the database. The results for these two experiments can be seen in Table 4.2.

We applied our algorithm on these data sets using 3 different minimal support levels. The level of reduction increases when we decrease the minimal support level, and we see that we attain ratios of down to 37%. After compression we applied post-acceptance pruning to remove the patterns that do

Table 4.2: Reduction results for sequences databases. The size of the code table $|CT|$, and the pruned code table $|CT^p|$ are compared to the size of the frequent pattern set $|\mathcal{F}|$ for various window sizes.

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

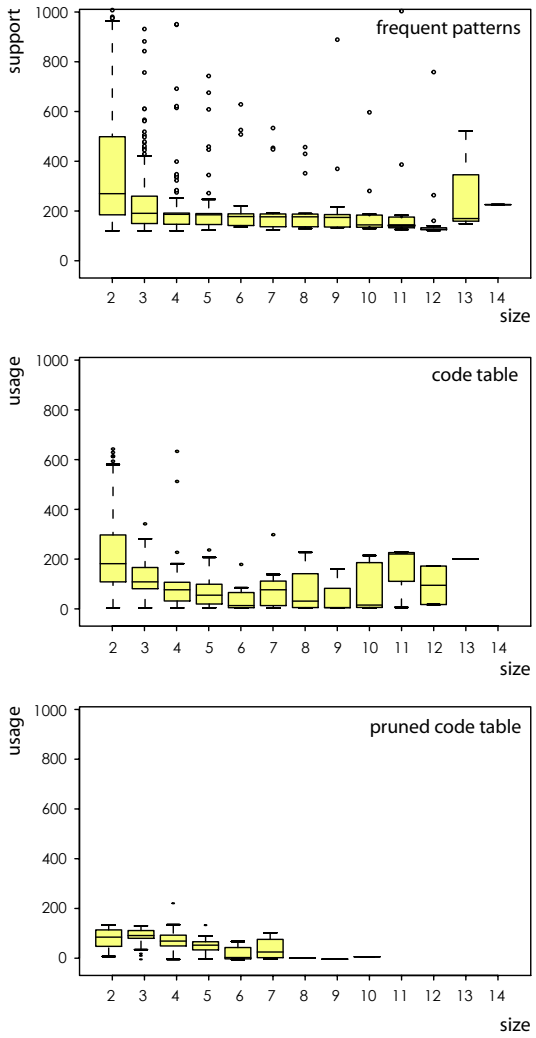


Figure 4.3: Pattern size histograms for the involved sequence pattern sets. For the KDDcup dataset with window size 60sec and *minsup* 150, we show: (top) the frequent pattern set, (middle) the pre-pruned code table, (bottom) and the post-pruned code table.

not contribute anymore to the compression of the complete database (see Algorithm 4). This resulted in an even higher reduction of the pattern set size; with pruning the pattern set is reduced to roughly 10% of its original size. Larger window sizes allow longer patterns to appear in our frequent pattern set, and result in a comparable reduction performances, albeit a bit less.

Figure 4.3 shows the pattern size distributions of the sequence patterns. From top to bottom, we see respectively the distribution for: the frequent pattern set, the code table, and the pruned code table. Here we clearly see the effect of the post-acceptance pruning. At first, we see that longer sequences *are* selected to be part of our un-pruned code table. However, upon pruning we see that most of these longer sequences disappear from the code table. One can conclude that these longer patterns are not required for a better description of the database. In these experiments many of these longer patterns are apparently not characteristic for the database, and thus do not become part of the resulting code table. In addition when looking at $\mathcal{F}$, doubling the window size only leads to a minor increase of the number of frequent patterns. This indicates that most of the (characteristic) sequence patterns have a small number of items.

Trees

In the experiments to reduce the set of frequent subtrees, we have chosen three different data sets: logml, US 304 and US 2430, which all contain weblog data (see Table 4.1 for details) [94]. To generate all frequent induced subtrees, we used the FREQT algorithm implemented by Taku Kudo [1, 60, 95].

We derived sets of frequent subtrees for all data sets for various different minimal support levels as described in Section 4.2. Then we applied the structured version of Krimp on all candidate sets, followed by a post-acceptance prune on the resulting code tables. The pruning phase removes the code table elements that do not contribute to the database compression.

Similar to the case of sequences we obtain good reductions: up to three orders of magnitude. For the logml database, only 0.17% of the originally generated frequent pattern set is preserved in the code table. In order to measure the influence of the minimum support threshold, we generate frequent pattern sets for a wide variety of minsups. In Figure 4.4, we have depicted the growth of the candidate set and the code table as a function of the minimum support for the US2430 database. For all databases we observe the same trend: increasingly lower minsup values lead to exponentially larger candidate sets. Also, we see that the growth of the code table is moderate.

From all databases we derived set of the frequent subtrees of up to a predefined minimal support level. As one would expect from the a-priori principle,

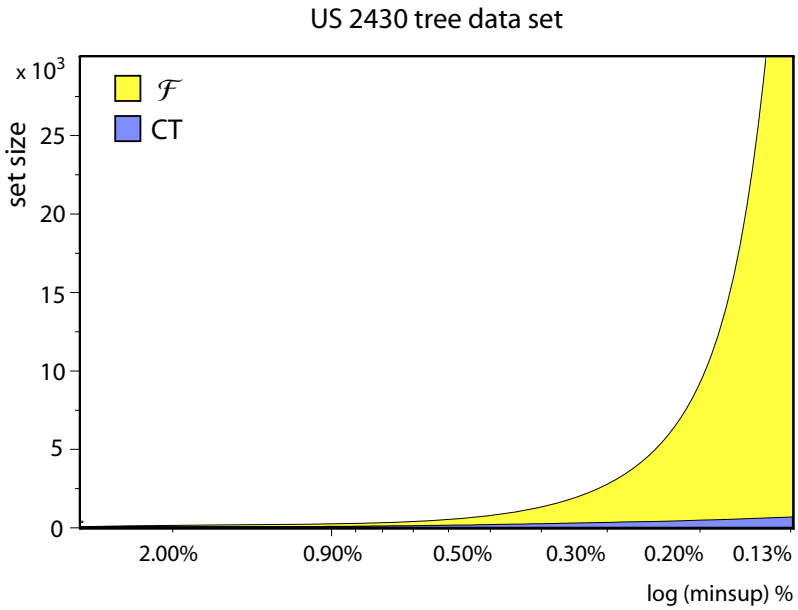


Figure 4.4: The selection based on MDL proves effective. While the number of frequent patterns in $\mathcal{F}$ grow exponentially for lower θ values, we see that the number of patterns in CT grow much less steeply. Shown are the results for the US 2430 tree data set.

Table 4.3: Reduction results for tree datasets. The size of the code table $|CT|$, and the pruned code table $|CT^p|$ are compared to the size of the frequent pattern set $|\mathcal{F}|$.

$\mathcal{D}$	**US 2430**	**US 304**	**logml**
θ	0.13%	0.20%	0.15%
$\|\mathcal{F}\|$	**46,232**	**196,392**	**275,377**
$\|\#CT\|$	10,001	9,409	9,650
$\|\#CT^p\|$	9,855	9,312	9,540
$\|\#CT \setminus \mathcal{I}\|$	717	481	590
$\|\#CT^p \setminus \mathcal{I}\|$	**571**	**384**	**480**
$\%CT^p \setminus \mathcal{I}$	**1.24%**	**0.20%**	**0.17%**

frequent subtree mining results in a collection of subtrees diminishing monotone over pattern size (see Figure 4.5 (top)). Longer patterns are likely to appear less frequent in a database than patterns of a smaller size. This is even accentuated due to the skewness of the databases as used in our experiments that contain large numbers of short trees.

When we look at the resulting (pruned) code tables we see a more stable result, in the sense that larger patterns occur relatively more often (see Figure 4.5 (middle) and (bottom)). Small patterns still have a high usage, as they are most likely completing much of the cover of the database. Still, a large number of small patterns is removed. This is because our algorithm only selects those patterns that contribute to the database description. From the fact that pruning does not have as much effect as in the case of sequences, we can conclude that there simply is more structure in the database that can be captured by our longer non-singleton patterns.

4.6 Discussion

We obtain good results for a variety of sparse and structured databases with various minimal supports and frequent pattern set sizes. Compression clearly reduces the frequent pattern set significantly, and makes it applicable for domain expert evaluation. Around 10% of the original frequent pattern set is considered relevant for sequence data and is consequently preserved. We see an even higher reduction to about 1% for tree structured data. In general, we see the trend that the code table growth as a function of the minimal support follows the frequent pattern set explosion only moderately. We also confirm that low *minsup* thresholds utilise more structure of the database than higher thresholds, as higher levels of compression are obtained. In our experiments, we have attained higher reductions when the database has a higher average number of items per transaction. This makes sense, as these databases can lead to longer patterns that can cover larger portions of the database and are subsequently replaced by a single code.

Improvement continues when pruning removes obsolete patterns from the code table. The reduction is even more impressive if we disregard the alphabet elements. Since the novelty for the user is in general found in the non-singleton 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.

Moreover, further improvement is gained as small and trivial patterns are removed, leading to a more balanced set of patterns in terms of size (see Figures 4.3b and 4.5b). Pruning leads to additional data enhancement by stripping the non-contributing patterns. This is indicated by the fact that the remaining patterns are used more often for the database cover.

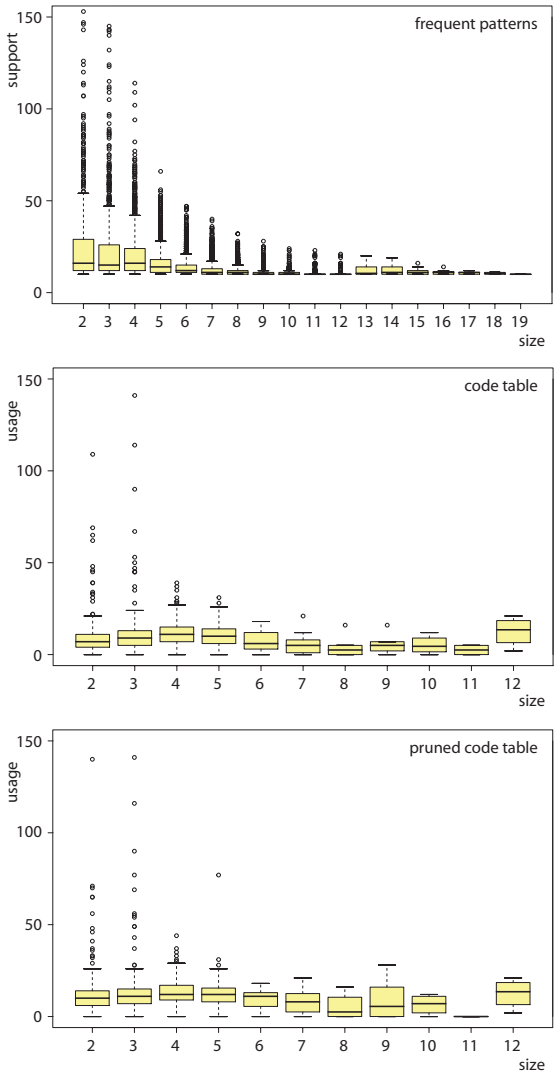


Figure 4.5: Distribution analysis of the pattern size histograms for the US2430 data set. (top) The input frequent pattern set has an expected monotone decay. (middle) However, the code table before pruning shows a more balanced distribution. (bottom) Similar to the distribution of the pruned code table.

4.7 Conclusion

Our MDL approach picks small informative sets of patterns from the potentially vast sets of frequent structured patterns. Reductions up to three orders of magnitude have been seen in our experiments. We have also shown that the exponential explosion of the frequent pattern set for lower minimal support levels is followed much more moderately. This volume reduction, especially at low minimal support levels, allows for a good inspection of our selected set of patterns: the code table. Moreover, this reduction does not simply favour small or large patterns. The interesting pattern set is a balanced set of frequent patterns of all possible sizes.

As noted in the discussion, the attained reductions depend on the average size of the structured elements in the database. Therefore a possible extension would to experiment on XML data, including its attribute data. In such data, the structured elements tend to be far larger in terms of the number of items and thus possibly allowing for even higher reduction ratios.

Chapter 5

Discovering Relational Itemsets Efficiently

Frequent itemset mining is a major data mining research area. Generalising from the standard single table case to a multi-relational setting is simple in principle, but hard in practice. That is, it is simple to define frequent itemsets in the multi-relational setting, as well as extending the A-Priori algorithm. It is hard, because the well-known frequent pattern explosion at low *minsup* settings is far worse than it is in the standard case.

In this chapter we introduce an effective algorithm for the discovery of frequent, multi-relational itemsets. These relational patterns show which itemsets occur together. Answering questions like: *'What type of Books are bought together with what Record types?'*. Hence, they provide a symmetric insight in the relation and reveal patterns that are relevant with respect to the relation. It extends our earlier work on using MDL to discover a small set of characteristic itemsets. The algorithm, R-Krimp, first discovers the small set of characteristic patterns in the single tables and then combines these to find a small set of characteristic multi-relational itemsets. This reduces the original search space dramatically and, hence, brings down the computational complexity by orders of magnitude. In the experiments we show that this approach yields a very good approximation of the naïve approach, joining all tables into one huge table, while being far more efficient.

[1]This is an extended and edited version of an earlier paper published as: [56]. *Discovering Relational Itemsets Efficiently.* A.C.M. Koopman, and A.P.J.M. Siebes. In: SDM 2008, pages 108-119.

5.1 Introduction

Frequent pattern mining is a major research area in data mining, with frequent itemset mining as premier example. Given the sales-records of a store, we want to discover all sets of items that customers buy together for at least *minsup* times. Clearly, this is an exponential problem, as all sets of items are potentially frequent and the number of such sets is exponential. The well-known A-Priori property [2], however, ensures that large parts of the search space can be pruned. Based on this, many algorithms exist that have good running times in practice. The only place where the exponential nature of the problem shows up is in the well-known frequent pattern set explosion. If the minimal support threshold is set low, the number of candidates and the number of results explodes.

In the generalisation of frequent itemset mining to the multi-relational case [49], the exponential nature of the problem returns with a vengeance. For example, assume we know what people bought both at a record store and at a bookstore. Patterns in this example are pairs (R, B), in which R is a set of records and B a set of books, both sets bought by the same customer. In this setting, we search for sets of records and sets of books that are frequently bought by customers. Clearly, the number of candidates is exponential both in the set of records and in the set of books.

Given these two tables with sales transactions of *Books* and *Records*, two transactions are related when a customer buys both sets of products. Finding collections of bought *Records* and *Books* provides insight in the related transactions. This helps to understand the relation between the two tables, as we can find frequently co-occurring patterns (e.g. are classical music and classic literature often bought together).

Our researched patterns are these symmetric frequently relating itemsets. To find these patterns we take a symmetric view of the database. This is because several *Book* sets can relate to one specific *Record* set, and vice versa. As we find the sales transactions from both tables equally important, we will need to count on both sides of the relation.

Our approach answers a different research question than standard Multi-Relational Data Mining (MRDM) approaches like WARMR [22]. A main concept in MRDM is the key table, which specifies the table that has a specific interest to the user [22, 49, 93]. As an example, consider the *Record* table as a target. A WARMR system would then count over the *Record* table to answer queries such as, which books are frequently bought together with a given record.

Our research question is to find a small characteristic set of these symmetric relational patterns. To obtain this small pattern set, we have to process

an exponential number of candidates and an exponential number of resulting frequent patterns. Fortunately, we are not interested in finding all frequent patterns, as this set grows exponentially, but only the interesting ones. For this we build on our earlier work in which we use the Minimum Description Length (MDL) principle to discover the interesting frequent patterns [79]. In particular, we use the KRIMP algorithm to select interesting patterns.

Given our aim to find a small set of patterns, we present three different methods to obtain them. When we approach the problem in a naïve manner we take a global view. This GLOBAL algorithm first computes the join of the tables and computes the frequent patterns from this materialised view. The GLOBAL algorithm quickly becomes impractical, because the materialised join table becomes huge.

We solve this first issue and achieve a better space complexity if we do not materialise the join. The idea of the LOCAL algorithm is to perform more computation locally at the single tables. Assume we have two tables, T_1 and T_2. First we compute the projections of both tables $T_1^\pi = \pi_{T_1}(T_1 \bowtie T_2)$ and $T_2^\pi = \pi_{T_2}(T_1 \bowtie T_2)$ which can be done without materialising the join. Then we find frequent itemsets for the semi-joins T_1^π and T_2^π individually. The pairs of frequent itemsets discovered from these tables are then fed into KRIMP to select the interesting ones. LOCAL is superior in space complexity to GLOBAL, and evaluates more candidates to lead to a better compression. LOCAL considers a candidate set which is locally frequent and is exponential in the number of candidates.

To reduce the number of candidates, our proposed algorithm, called R-KRIMP, first reduces the frequent itemset collections computed from T_1^π and T_2^π using KRIMP . Then it continues as LOCAL. Since KRIMP always gives a dramatic reduction in the number of frequent itemsets, R-KRIMP is computationally superior to LOCAL. The cost could be that the resulting set of frequent patterns is far less descriptive than those computed by LOCAL. However, with experiments we show that R-KRIMP approximates LOCAL very well; the results are almost the same. In other words, R-KRIMP is efficient in both space complexity and computational costs. It reduces candidate space efficiently by picking the content that is relevant and results in a small set of patterns that describe the data well. One could paraphrase this by the quasi formula:

$$\begin{aligned} \text{R-KRIMP}(T_1, \ldots, T_n) &= \text{KRIMP}(\prod_{i=1}^{n} \text{KRIMP}(T_i)) \\ &\approx \text{KRIMP}(\bowtie_{i=1}^{n} T_i) \end{aligned}$$

Id = { *Person Id* }
$\mathcal{I}_1$ = { abbey road, thriller, queen }

T_1 : Records	
Person Id	*Tr1*
1	{ abbey road }
1	{ queen , thriller }
2	{ abbey road , queen }

Id = { *Person Id* }
$\mathcal{I}_2$ = { colour of magic, good omens }

T_2 : Books	
Person Id	*Tr2*
1	{ colour of magic , good omens }
2	{ colour of magic }
2	{ good omens }

Join(T_1 ,T_2) - target: Records	
{ { abbey road , queen , thriller }	, { good omens } }
{ { abbey road , queen }	, { colour of magic } }
{ { abbey road , queen }	, { good omens } }

F_1 = { { abbey road, queen } , { ø } } supp(F_1) = 3

Join(T_1 ,T_2) - target: Books	
{ { abbey road }	, { colour of magic , good omens } }
{ { queen , thriller }	, { colour of magic , good omens } }
{ { abbey road , queen }	, { colour of magic , good omens } }

F_1 = { { ø } , { colour of magic, good omens } } supp(F_1) = 3

Join(T_1 ,T_2)	
{ { abbey road }	, { colour of magic , good omens } }
{ { queen , thriller }	, { colour of magic , good omens } }
{ { abbey road , queen }	, { colour of magic } }
{ { abbey road , queen }	, { good omens } }

F_1 = { { abbey road, queen } , { ø } } supp(F_1) = 2
F_2 = { { ø } , { colour of magic, good omens } } supp(F_2) = 2

Figure 5.1: Our example joined relational database with records and books. The join is performed via *Person Id*. Depending on the focus of interest, we either have frequent Record patterns, Book patterns, or the patterns that have *our* particular interest, those that describe the relation. In the shown result, only the largest patterns are depicted for clarity.

5.2 Relational Itemsets

To further illustrate the application of our approach, we will run along an example. Consider again the database that consists of a collection of tables, one contains transactions of book sales the other table contains record sales (see Figure 5.1). Each transaction consists of the titles of the work that is sold during that transaction. For example, *Person 1* has bought two books during one transaction: *"Colour of Magic"* and *"Good Omens"*. Two transactions are related when a customer has bought two product sets, one from each table. As an example, we see that *Person 1* has two related Record transactions: *"Abbey Road"*, and *"Queen, Thriller"*.

Given this database, we can pose a number of questions that relate to the relational nature of the database. First of all, we can be interested in finding record sales with books: *"What record titles are sold to a person together with a given frequent book title?"*. Alternatively, we can be interested in book sales related to the record store: *"What book titles are sold to a person together with a given frequent record title?"*. Patterns of both approaches answer a question which is relevant to a given table of interest. In order to derive the respective patterns we need to count over the respective target tables. The result is shown in Figure 5.1, together with a pattern found in each joined table.

Let us know now ask a different question: *"What record titles and book titles are frequently sold to a person?"*. This brings the focus from the tables to the relation itself. For clarity, consider the depicted example on the bottom of Figure 5.1, where the symmetrically joined table is shown together with some derived relational patterns. Note that counting on either the Book table, the Record table, or all tables leads to different joins, and relational patterns.

To answer our research question, we do not define a specific target table, as all tables are equally relevant. A clear area of interest is formed by applications where relational information is available for equally interesting entities. Some concrete examples that we present in our experiments include finding patterns such as *"What type of journal paper cites what type of papers?"* and *"What type of genes interact with what type of genes?"*.

5.3 Problem Statement

In this section we first define our data and patterns. In particular we introduce relational itemsets and their support. Moreover, we discuss the relation between relational itemsets and their well-known counterparts, classical itemsets. Next we introduce our problem informally. Then we show how MDL can be used for relational itemsets and, finally, we state our problem formally.

Data and Patterns

Different from traditional frequent pattern mining, we are interested in relational frequent itemsets. That means that the data resides in multiple related transaction tables. That is, each of these tables has one column that stores the transactions. Furthermore, these tables will have one or more columns containing *identifiers*, which are simply natural numbers. In contrast to primary keys, these identifiers are not assumed to be unique. Rather, they represent the relations between the tables. If two tables have common identifier columns, this means that the transactions in the two tables are related. Hence, our data is defined as follows.

Definition 15. *Let $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$ be a collection of itemsets for k tables. For each table j we have a transaction attribute Tr_j with domain $Dom(Tr_j) = \mathcal{P}(\mathcal{I}_j)$. The set of all transaction attributes is denoted by $Tr = \{Tr_1, \ldots, Tr_k\}$. Moreover, let $Id = \{Id_1, \ldots Id_l\}$ be a set of identifier attributes. These are attributes with domain $Dom(Id_j) = \mathbb{N}$. A relational transaction database $\mathcal{D}$ over (Id, Tr) is a set of tables $\{T_1, \ldots, T_k\}$, such that:*

- *The schema of $T_l = \{Id_{l_1}, \ldots, Id_{l_m}, Tr_l\}$ in which $Id_{l_j} \in Id$ and $Tr_l \in Tr$. Each transaction attribute is associated with one table of $\mathcal{D}$ only.*
- *A transaction tr in T_l, denoted by $tr \in T_l$ is a tuple $(i_{1_m}, \ldots, i_{l_m}, t)$ in which $i_j \in Dom(Id_j)$ and $t \subseteq \mathcal{I}_l$.*
- *Let tables T_i and T_j be two tables in $\mathcal{D}$ with schema $\{Id_{i_1}, \ldots, Id_{i_m}, Tr_i\}$ and $\{Id_{j_1}, \ldots, Id_{j_m}, Tr_j\}$ respectively. T_i and T_j are related iff:*

$$Join(T_i, T_j) = \{Id_{i_1}, \ldots, Id_{i_m}\} \cap \{Id_{j_1}, \ldots, Id_{j_m}\} \neq \varnothing.$$

- *The set of tables is connected. That is, for each pair of tables $T_i, T_j \in \mathcal{D}$, there is a list of tables such that:*
  - *All tables in the list are related to their successor.*
  - *T_i is related to the first element of the list and T_j is related to the last element in the list.*
- *Let T_i and T_j be two related tables, moreover, let $tr_1 \in T_i$ and $tr_2 \in T_j$. The two transactions are related iff:*

$$\pi_{Join(T_i,T_j)}(tr_1) \cap \pi_{Join(T_i,T_j)}(tr_2) \neq \varnothing.$$

First, note that the requirements that the set of tables is connected is not strictly necessary. However, it is without loss of generality. For a database with a disconnected set of tables, one can simply run our algorithms on the connected components.

Second, note that we use the term $Join(T_i, T_j)$, because whenever we join two related tables, denoted as usual by $T_i \bowtie T_j$, we implicitly use the equi-join with the equality restriction on $Join(T_i, T_j)$ [75].

Now that we have defined our data, we can define our patterns. The generalisation for a standard itemset to a relational itemset is straight forward. It is a tuple of itemsets, one for each table in the database. That is, we have the following definition.

Definition 16. *Let $\mathcal{D} = \{T_1, \ldots, T_k\}$ be a relational transaction database over the set of itemsets $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$. A relational itemset over $\mathcal{I}$ is a k-tuple :*

$$(I_1, \ldots, I_k) \text{ where } I_j \subseteq \mathcal{I}_j.$$

Note that an I_i in a relational itemset may be empty, i.e., $I_i = \varnothing$. That is, relational patterns may have "holes".

Generalising the notion of an occurrence of an itemset in a database to the relational setting, is also straight forward. It is simply a tuple of occurrences in the various tables, the support of a relational itemset is, again, the number of occurrences in the database. That is, we have the following definition.

Definition 17. *Let $\mathcal{D} = \{T_1, \ldots, T_k\}$ be a relational transaction database over the set of itemsets $\{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$. Moreover, let $\{I_1, \ldots, I_k\}$ be a relational itemset over $\mathcal{I}$. An occurrence of I in $\mathcal{D}$ is a tuple of transactions $(tr_1, \ldots, tr_k)$, with $tr_j \in T_j$, such that:*

1. *tr_i and tr_j are related whenever T_i and T_j are related.*
2. *$\forall j \in 1, \ldots, k : I_j \subseteq \pi_{Tr_j}(tr_j)$*

The support of I in $\mathcal{D}$, denoted by $sup_{\mathcal{D}}(I)$ is the number of occurrences of I in $\mathcal{D}$.

Note that an occurrence $(tr_1, \ldots, tr_k)$ contains a transaction from each of the underlying tables. Paraphrasing, relational itemsets may have holes, their occurrences may not. This may seem a restriction, but it is not. We return to this point after the formal statement of our problem.

There is an alternative way to define the occurrence and the support of a relational itemset. We can define them directly on the join of all the tables in $\mathcal{D}$. More precisely, let $J(\mathcal{D})$ be the join of all these tables, i.e,

$$J(\mathcal{D}) = T_1 \bowtie \cdots \bowtie T_k.$$

Now that we have joined all tables, all relationships that were represented by the identifier attributes are now explicit in $J(\mathcal{D})$. Hence, without loss of information, we can reduce $J(\mathcal{D})$ by projecting out all identifier attributes. The reduced database, $R(J(\mathcal{D}))$, contains k-tuples of the form:

$$(tr_1, \ldots, tr_k) \text{ with } tr_j \in \pi_{Tr_j}(T_j).$$

We can now define an occurrence of the relational itemset $I = (I_1, \ldots, I_k)$ in $R(J(\mathcal{D}))$ as a k-tuple $(tr_1, \ldots, tr_k) \in R(J(\mathcal{D}))$ such that:

$$\forall j \in \{1, \ldots, k\} : I_j \subseteq tr_j.$$

The support of a relational itemset is now again defined as the total number of occurrences in $R(J(\mathcal{D}))$. Given that our joins are equijoins, there is a one-one correspondence between the two approaches. More precisely, we have the following simple lemma.

Lemma 3. *Let $\mathcal{D} = \{T_1, \ldots, T_k\}$ be a relational transaction database over the set of itemsets $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$. Moreover, let $I = \{I_1, \ldots, I_k\}$ be a relational itemset over $\mathcal{I}$.*

- *There exists a bijective function Φ that maps each occurrence of I in $\mathcal{D}$ onto an occurrence of in $R(J(\mathcal{D}))$ and vice versa.*
- *The support of I in $\mathcal{D}$ equals the support of I in $R(J(\mathcal{D}))$.*

If we assume that all the elements of $\mathcal{I}$ are mutually disjoint, we can go even further. We can squash $R(J(\mathcal{D}))$ into a classical itemset database, denoted by $S(R(J(\mathcal{D})))$ over the set of items $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ by:

$$S(tr_1, \ldots, tr_k) = tr_1 \cup \cdots \cup tr_k.$$

Note that S is, again, a bijection, because all the $\mathcal{I}_j$ are mutually disjoint. Obviously we can apply S also to relational itemsets. S is then a bijection from the relational itemsets defined on $\mathcal{D}$ to the classical itemsets on $S(R(J(\mathcal{D})))$; again, because the $\mathcal{I}_j$ are mutually disjoint. Hence, we have the following result.

Theorem 2. *Let $\mathcal{D} = \{T_1, \ldots, T_k\}$ be a relational transaction database over the set of itemsets $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$. Moreover, let I be a relational itemset over $\mathcal{I}$. Finally, let $S(R(J(\mathcal{D})))$ be the squashed, reduced version of $\mathcal{D}$. If the $\mathcal{I}_J \in \mathcal{I}$ are mutually disjoint, then*

- *There exists a bijection Ψ_1 from the relational itemsets over $\mathcal{I}$ to the classical itemsets over $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$.*

– *There is a bijection* Ψ_2 *from the occurrences of* I *in* $\mathcal{D}$ *to the occurrences of* $\Psi_1(I)$ *in* $S(R(J(\mathcal{D})))$.

– *Hence, the support of* I *in* $\mathcal{D}$ *equals the support of* $\Psi_1(I)$ *in* $S(R(J(\mathcal{D})))$.

Given that we can always make the $\mathcal{I}_J \in \mathcal{I}$ mutually disjoint, e.g., by appending the table name T_j to the elements of $\mathcal{I}_j$, the reader may now think: why all the fuss? First we define relational itemsets, their occurrences and their support and then we show that this is completely equivalent to the classical, standard, case.

It is, however, always true for multi relational data mining that one could join all the tables in the database into one table and analyse that table. The point of multi relational data mining, however, is that the tables have semantics. Retaining those semantics makes the results far easier to interpret. Moreover, the multi relational setting allows for strictly more expressive models. Finally, a more practical aspect is that joining all tables into one often produces a truly gigantic table, which is too costly to analyse. So, the importance of the theorem is not that it yields a good computational approach. Its importance is that it shows that the way we generalised itemsets, their occurrences and support is sound.

The Problem: Informally

The relational patterns satisfy the A-Priori principle, just like their classical counterpart [37]. The order on relational patterns is simply derived from the classical order on itemsets

Definition 18. *Let* $I = (I_1, \ldots, I_k)$ *and* $J = (J_1, \ldots, J_k)$ *be two relational itemset over* $\mathcal{I}$. I *is more general than* J, *denoted by* $I \preceq J$, *if*

$$\forall i \in \{1, \ldots, k\} : I_i \subseteq J.$$

It is easy to see that $I \preceq J \rightarrow sup_{\mathcal{D}}(I) \geq sup_{\mathcal{D}}(J)$. Which means that the required A-Priori principle holds, and we can mine for frequent relational itemsets using, e.g., a level wise search algorithm.

While simple in theory, it is, unfortunately, not so easy in practice. We have already seen in the previous subsection that this problem is equivalent to computing frequent itemsets on the join of all tables in the database. This table is often huge. That is, the well-known frequent itemset explosion at low minsup settings returns with a vengeance. Practice shows that the larger the tables are, the more frequent itemsets are returned. In other words, while we could mine for all frequent relational itemsets, the result would not be that useful.

Hence, we do not want to return all frequent relational itemsets. Rather, our goal is to compute (and return) a small, characteristic, set of frequent relational itemsets. In Chapter 3, we discussed a similar problem in the context of classical frequent itemset mining. In order to derive this small characteristic set, the Minimum Description Length (MDL) principle [32] is used in the introduced Krimp algorithm, that approximates the optimal result. In subsequent research it is shown that the resulting small sets of itemsets characterise the underlying database very well.

In this chapter we extend this approach to relational itemset mining. In principle this is, of course, easy because of the equivalence between the relational and the classical case. However, as pointed out before, this equivalence might be very inefficient. Hence, the key point of this chapter is that we introduce a far more efficient algorithm, called R-Krimp, for the relational case in the next section. First, however, we discuss MDL in the relational setting after which we state our problem formally; this discussion is adapted from [79, 85].

MDL for Relational Itemsets

Similar to Krimp [79], we use MDL to find the model, or code table CT, that describes the data, $\mathcal{D}$, best. In this relational setting, we therefore need to redefine how CT can describe $\mathcal{D}$. Or more specifically, how the pattern occurrences relate to the code table.

Definition 19. *Let $\mathcal{I} = \mathcal{I}_1, \ldots, \mathcal{I}_k$ be a collection of sets of items and $\mathcal{C}$ a set of code words. A code table CT for $\mathcal{I}$ and $\mathcal{C}$ is a two column table such that:*

1. *The first column contains relational itemsets over $\mathcal{I}$, this column contains at least all singleton relational itemsets. These k-tuples are of the form:*

$$(\varnothing, \ldots, \varnothing, \{I\}, \varnothing, \ldots, \varnothing)$$

   That is, all entries but one are the empty set and the one non-empty set entry is a singleton set.

2. *The second column contains elements from $\mathcal{C}$, such that each element of C occurs at most once.*

A relational itemset I over $\mathcal{I}$ occurs in CT, denoted by $I \in CT$ iff I occurs in the first column of CT; similarly for a code $C \in \mathcal{C}$. For $I \in CT$, $code_{CT}(I)$ denotes its code, i.e, the corresponding element in the second column.

Before we can define how a database is encoded with a code table, we first have to decide what we are going to encode. Each table in the database contains one or more identifier attributes, as well as a transaction attribute. The

Join(Paper , Paper , Paper)		
{ Algebraic Geometry , 2001 }	{ Algebraic Geometry , 2000 }	{ Algebraic Geometry , 2000 }
{ Algebraic Geometry , 2002 }	{ Algebraic Geometry , 2001 }	{ Quantum Algebra , 2000 }
{ Algebraic Geometry , 2001 }	{ Quantum Algebra , 2000 }	{ Algebraic Geometry , 2000 }
{ Algebraic Geometry , 2002 }	{ Quantum Algebra , 2001 }	{ Algebraic Geometry , 2000 }

(a) original database

Join(Paper , Paper , Paper)				
code(1)	code(4)	{ , }	{ , }	{ Algebraic Geometry , }
code(1)	code(2)	{ , }	{ , }	{ Quantum Algebra , 2000 }
code(3)	code(4)	{ , }	{ , }	{ , }
code(2)	code (3)	{ , }	{ , }	{ , 2000 }

(b) partly covered database

CT	
code(C)	*C*
code(1)	{{ Algebraic Geometry },{ Algebraic Geometry }, { ø }}
code(2)	{{ 2002 },{ 2001 }, { ø }}
code(3)	{{ Algebraic Geometry },{ Quantum Algebra }, { Algebraic Geometry }}
code(4)	{{ 2001 },{ 2000 }, { 2000 }}

(c) code table

Figure 5.2: (a) The joined table *Join(Paper,Paper,Paper)* is covered by the code table CT. (b) During the COVER process, the original items from the relational itemsets are replaced by codes. (c) The length of these codes are derived for all code table elements in CT.

identifiers represent the relationships between transactions only. That is, if we, e.g., add 1 to each identifier, the identifiers still represent the same relationships. In other words, the actual value of identifier attributes is immaterial, our results should not depend on these values.

That is, encoding the transactions should be *independent of the identifiers*. That is, we encode $R(J(\mathcal{D}))$. Recall that the elements of $R(J(\mathcal{D}))$ are k-tuples $t = (t_1, \ldots, t_k)$ of related transactions in $T_1 \bowtie \cdots \bowtie T_k$. Moreover, let $I = (I_1, \ldots, I_k)$ be a relational itemset, t is an occurrence of I, denoted by $I \subseteq t$, if

$$\forall j \in \{1, \ldots, k\} I_j \subseteq t_j.$$

Finally, by $t \setminus I$ we denote the tuple:

$$t \setminus I = (t_1 \setminus I_1, \ldots, t_k \setminus I_k).$$

Slightly abusing notation, we will write $t \in \mathcal{D}$ rather than $t \in R(J(\mathcal{D}))$.

Algorithm 5 R-Krimp Compress

Local = Compress(R(J($\mathcal{D}$)), $\mathcal{F}_{\text{Local}}$)
Global = Compress(R(J($\mathcal{D}$)), $\mathcal{F}_{\text{Global}}$)
R-Krimp = Compress(R(J($\mathcal{D}$)), $\mathcal{F}_{\text{R-Krimp}}$)

Compress $(R(J(\mathcal{D})), \mathcal{F})$

1. $CT \leftarrow \mathcal{I}$
2. **for all** $F \in \mathcal{F}$
3. $\quad CT' \leftarrow CT_R$ + add F in order
4. $\quad$ **for all** $t \in R(J(\mathcal{D}))$
5. $\quad\quad$ Cover(CT, t)
6. $\quad\quad$ **if** $L(CT', R(J(\mathcal{D}))) < L(CT, R(J(\mathcal{D})))$
7. $\quad\quad\quad CT \leftarrow CT'$
8. $\quad$ **end for**
9. **end for**
10. **return** CT

Problem Statement: Formally

Given the definition of our class of models, code tables, and a way to compute the total length of the encoded database using Krimp (see Chapter 3), we can state our problem formally in terms of MDL.

Problem Statement. *Let $\mathcal{D} = \{T_1, \ldots, T_k\}$ be a relational transaction database over the set of itemsets $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$. Find a code table CT that minimises $L(\mathcal{D}, CT)$.*

Note that in view of the equivalence between relational itemsets on relational transaction databases and the classical setting of itemsets on transaction databases, we already know how to solve this problem. First transform $\mathcal{D}$ into $S(R(J(\mathcal{D})))$. Then apply the Krimp algorithm from [79]. Finally, transform the itemsets in the resulting code table back to relational itemsets, and you are done. However, as noted before, this is not likely to be an efficient approach. Therefore, the main goal of this chapter is to introduce an efficient algorithm that solves this problem.

A second remark is that one might object that we do not compress the complete database. For, if there is a transaction t in, say T_1, that is not related to any transaction in the other T_j's, then t is not encoded at all. In other words, there are transactions in our database that are not characterised by the resulting code table.

To characterise the complete database, we need a stratified approach. Let

Algorithm 6 CANDIDATE SET: GLOBAL

$\mathcal{F}_{\text{GLOBAL}}(T_1^\pi, T_2^\pi)$
1. $\mathcal{F}_1^\pi \leftarrow \text{FREQ}(T_1^\pi)$
2. $\mathcal{F}_2^\pi \leftarrow \text{FREQ}(T_2^\pi)$
3. $\mathcal{F} \leftarrow \varnothing$
4. **for all** $(F_1, F_2), F_1 \in \mathcal{F}_1^\pi, F_2 \in \mathcal{F}_2^\pi$ **do**
5. **if** $sup_{T_1^\pi, T_2^\pi}(F_1, F_2) > \theta$
6. $\mathcal{F} \leftarrow \mathcal{F} \cup \{(F_1, F_2)\}$
7. **end for**
8. **return** $\mathcal{F}$

$\mathcal{D} = \{T_1, \ldots, T_k\}$. First we compute the code table $CT_{\{1,\ldots,k\}}$ for $\{T_1, \ldots, T_k\}$. Then we consider the k databases with $k-1$ tables, $\mathcal{D}_i = \mathcal{D} \backslash T_i$. From each table in $\mathcal{D}_i$ we remove those transactions that are characterised by $CT_{\{1,\ldots,k\}}$. Then we compute a code table for each of these database, characterising relationships of $k-1$ transactions that are not part of a k transactions relations. Repeat this stratification until finally all transactions are characterised by a code table. Since the problem (and thus the algorithm) is the same in each step on the way, we concentrate on one step only.

5.4 Algorithms

In this section we introduce three algorithms that, approximately, solve our formal problem. As we base our algorithms on the KRIMP algorithm, that approximately solves the single table case, we refer to Chapter 3 for more details.

The Relational Case

The relational case is very similar to KRIMP as discussed above. If we define the size of a relational itemset as the sum of the sizes of its components, the order of the candidates and the code table are as for KRIMP. The question is: which candidates do we consider? There are three options:

1. $\mathcal{F}_{\text{GLOBAL}}$: First compute the join of all tables, then apply KRIMP to this huge table. This is the GLOBAL algorithm. Given the potentially huge size of join result, GLOBAL does have a rather bad space complexity. Moreover, the number of candidates that KRIMP will have to consider is also huge, hence, the computational complexity is not very good either.

Algorithm 7 Candidate Set: Local

$\mathcal{F}_{\text{Local}}(T_1^\pi, T_2^\pi)$
1. $\mathcal{F}_1^\pi \leftarrow \text{Freq}(T_1^\pi)$
2. $\mathcal{F}_2^\pi \leftarrow \text{Freq}(T_2^\pi)$
3. $\mathcal{F}_{T_1} \leftarrow \varnothing$
4. $\mathcal{F}_{T_2} \leftarrow \varnothing$
5. $\mathcal{F} \leftarrow \varnothing$
6. **for all** $F_1 \in \mathcal{F}_1^\pi$ **do**
7. **if** $sup_{T_1^\pi}(F_1) > \theta$
8. $\mathcal{F}_{T_1} \leftarrow \mathcal{F}_{T_1} \cup \{F_1\}$
9. **end for**
10. **for all** $F_2 \in \mathcal{F}_2^\pi$ **do**
11. **if** $sup_{T_2^\pi}(F_2) > \theta$
12. $\mathcal{F}_{T_2} \leftarrow \mathcal{F}_{T_2} \cup \{F_2\}$
13. **end for**
14. $\mathcal{F} \leftarrow \mathcal{F}_{T_1} \times \mathcal{F}_{T_2}$
15. **return** $\mathcal{F}$

2. $\mathcal{F}_{\text{Local}}$: The first step towards improvement is the realisation that we do not have to materialise the full join over all tables. Rather, we can first blow-up each individual table as follows:

$$T_i^\pi = \pi_{T_i}(T_1 \bowtie \cdots T_k)$$

which can be computed without storing the full join. Next, we compute the frequent itemsets on T_i^π. The candidates we consider are k-tuples of these 'local' candidates. This algorithm is called Local. Clearly, the Local algorithm has a far better space complexity than Global. However, the computational complexity might be worse. Some tuples of itemsets that are locally frequent will not be frequent globally. In other words, we may consider far more candidates. While this leads to a better code table, it takes more time.

3. $\mathcal{F}_{\text{R-Krimp}}$: The third option is based on the observation that Krimp produces a code table that characterises a table rather well. So, why do we consider all frequent sets on T_i^π? We can also compute a code table CT_i on T_i^π and consider only these as local candidates. This algorithm is called R-Krimp. Clearly, it enjoys the space complexity of Local, while having a far better computational complexity; far fewer candidates are considered. There is a risk involved, however. Because we consider far fewer candidates, our final code table might be far worse than the one

Algorithm 8 CANDIDATE SET: R-KRIMP

$\mathcal{F}_{\text{R-KRIMP}}(T_1^\pi, T_2^\pi)$
1. $CT_1^\pi \leftarrow \text{KRIMP}(T_1^\pi)$
2. $CT_2^\pi \leftarrow \text{KRIMP}(T_2^\pi)$
3. $\mathcal{F} \leftarrow \varnothing$
4. **for all** $\alpha_1 \in CT_1^\pi$ **do**
5. **if** $sup_{T_1^\pi}(\alpha_1) > \theta$
6. $CT_1^\pi \leftarrow CT_1^\pi \setminus \{\alpha_1\}$
7. **end for**
8. **for all** $\alpha_2 \in CT_2^\pi$ **do**
9. **if** $sup_{T_2^\pi}(\alpha_2) > \theta$
10. $CT_2^\pi \leftarrow CT_2^\pi \setminus \{\alpha_2\}$
11. **end for**
12. $\mathcal{F} \leftarrow CT_1^\pi \times CT_2^\pi$
13. **return** $\mathcal{F}$

computed by LOCAL. Only experiments can tell whether or not R-KRIMP is a viable approach.

These three algorithms are depicted in Algorithms 6, 7, and 8 respectively.

5.5 Experiments

In our experiments we have used a number of public relational databases.

Hepatitis Database

From the EMCL/PKDD 2005 dataset we have selected a number of databases that is comprised of data from hepatitis medical analysis . In each table, patient medical test information is stored that is linked to a central patient table containing their personal details. From this relational database we have selected a number of tables: *bio*, a table containing biopsy results, *ifn* contains information on interferon therapy, and *hemat*, which contains results on hematological analysis. In all cases, the original tables were converted to itemset databases. As the databases contained missing data, we have assigned these items with specific labels to mark them. From the generated frequent pattern set we have filtered the patterns containing missing values. As such, in our cover process these unknown values are forced to be covered by their singleton representatives. Therefore, none of the added code table elements will contain missing data.

Table 5.1: Database characteristics for all databases. We list the number of tuples, the number of items in each joined transaction ($|\mathcal{I}|$), and the number of transactions per table. The most right column shows the size of the joined table.

database	#tuples	$\lvert\mathcal{I}\rvert$	$\lvert T_1^\pi\rvert$	$\lvert T_2^\pi\rvert$	$\lvert T_3^\pi\rvert$	$\lvert \bowtie T_{1,2,3}^\pi\rvert$
2-table						
LOAN-TRANSACTION	54694	17	54694	682		929798
GENE-INTERACTION	150376	18	2437	5450		2706768
PAPER-PAPER	4585	12	1441	1366		55020
BIO-HEMAT	50766	15	689	32756		761490
BIO-IFN	334	12	334	197		4008
HEMAT-IFN	16956	18	16956	197		305208
3-table						
PAPER-PAPER-PAPER	11445	18	1065	913	1021	206010
BIO-IFN-HEMAT	31379	22	334	197	16956	690338

Gene-Interaction Database

Another database is derived from the KDDcup 2001 and contains information on interacting genes . The original database consisted of 2 tables: genes and interactions. The genes table consists of the following fields: *gene id, essential, class, complex, phenotype, motif, chromosome number, function,* and *localization*. In addition to the definition of relationships between genes, the interaction table contains the relation types that were detected between genes (e.g. physical- or genetic interaction) and the bond strength. As our method does not incorporate labelled links, we created a second gene table, *genes2*, which was augmented with this information; adding the type and strength fields to it. After extension the complete related transaction (t_1, t_2) therefore contains the following fields: t_1: (*essential, class, complex, phenotype, motif, chromosome number, function,* and *localization*). t_2: (*essential, class, complex, phenotype, motif, chromosome number, function, localization, interaction type* and *bond strength*).

High Energy Physics paper Database

For the KDDcup 2003, a large collection of High Energy Physics papers were made available. From this large database, a relational database is made available, which we have adapted for our use . It consists of HEP papers that are linked to journals, authors and other papers based on citations and authorship.

We focus here on which paper properties are often linked through citations to other papers. From the database we have selected available attributes of interest: *topic area, being published, year of publication, number of authors, number of times cited,* and *the number of citations to papers.* After this selection, we again projected the relation to create binned versions of the databases. An example pattern that could be found in the two tables would be of the form: (*topic area, number of authors & being published*).

Financial Database

The last database comes from the PKDD 1999 where financial information is linked via relations. In this database information is stored about customers, loans, and their financial transactions. We have selected features of provided loans and observed financial transaction features that are linked through the involved customer. The goal is to find related patterns of transaction features that result in provided loan features and vice versa. The selected attributes for the *loan* table are: *date, number, duration, payments, and status.* For the transactions table we have: *date, type, operation, number, balance, symbol, bank, and account.* Given two tables, an example pattern could be of the following type: (*number , balance & bank*).

5.6 Results

Relational Patterns

Before going into detail on how well our methods performs in terms of reducing complexity, we will first focus on our derived patterns. As defined in section 5.3, we are interested in finding a small set of characteristic related patterns. In our code table, a code table element consists of a tuple of itemsets (F_1, F_2). As an example, using R-Krimp, we find the following related itemset: ({unpublished, 3 authors, 2002}, {published, 3 authors, Mathematical Physics, 2001}) that is descriptive in the HEP paper database. Apparently, these citation characteristics occur often in the database.

In the MDL description there is a trade-off between the covered item sets and the number of relations covered by the relational patterns. R-Krimp automatically finds this trade off. Due to the inclusion of empty sets, we are not restricted to find patterns over all tables per se. Therefore we see that portions of the code table either cover one, two or three tables. An example pattern which is best described over a subset of tables is: $(1994, 1993, \varnothing)$. Apparently, paper citations between 1994 and 1993 are best described over 2 instead of 3 tables. R-Krimp chooses to describe the relation how it sees it best fit, as some occurring phenomena can be best described locally.

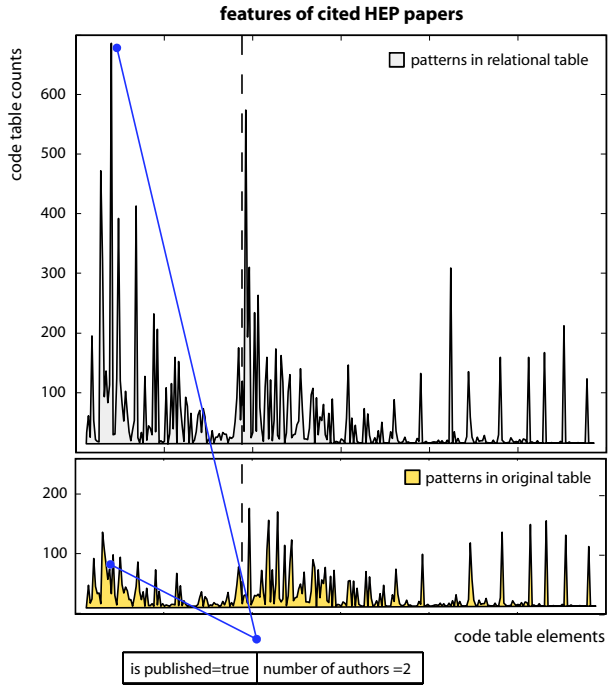


Figure 5.3: Incorporating relational information reveals patterns that would otherwise have been hidden. Not surprisingly from relational perspective, we see that papers are more often cited when they are written by more authors and when published.

Looking at Figure 5.2, we see how our code table maps to our joined table. Depicted on top is the joined table $T_1 \bowtie T_2 \bowtie T_3$. On the bottom is a visualisation of the relational code table. Note that single table itemsets can be used in several code table elements (e.g. *year:2001* in our shown example). When visualising the result we can merge these, and as such decompose the code table. When we decompose the relational code table, we obtain pattern graphs which indicate the pattern interaction. The depicted result in the middle in Figure 5.2 is partial. The most frequently used patterns are selected from our relational code table for visual clarity (shown patterns cover at least a third of the relation). The relational code table is a short description of the complete joined table, which maintains the structural information.

Capturing the Relational Influence

We see that we can find relational patterns over multiple relations, but does incorporating the relation provide us with additional insight? We are not merely interested in finding relational patterns; we are interested in finding patterns that have some relevancy with respect to the relation. In other words, can we now find patterns that we would not have found otherwise, which arise due to the relational influence?

To inspect this, we focus on the HEP paper database. Recall that the relation encodes the citation interactions, and the second table contains the papers that have been cited. To see what patterns make the difference, we measure the code table differences between the original table and its relational counterpart. To do this, we compare the code table usages, $usage_{CT}(C)$, and compare them for each code table. A high pattern usage relates to frequently used and relevant patterns. The results are shown in Figure 5.3. The most left part, left of the dashed line, depicts the non-alphabet elements; it is there where we see the most difference. When we look at the most prominent difference we observe an interesting result. The itemset (published, 2 authors) is very descriptive for the cited set.

Increasing the Efficiency

In all our previous work, we have seen that finding the smallest encoded length goes on par with finding the best description. Therefore, we would like to achieve the smallest encoded length for the database using only the smallest number of candidates possible. This translates to finding a small set of promising candidates that could compress the database well.

Reduce the Candidate Set

We have presented the candidate set reduction results in Table 5.2. From these results we see that the $\mathcal{F}_{\text{GLOBAL}}$ starts small, with respect to the additional candidates that $\mathcal{F}_{\text{LOCAL}}$ has to consider. The local *minsup* leads to a higher number of candidates. This candidate set explosion ranges from 4 to 260 times as much candidates.

In perspective, R-KRIMP reduction picks out a fraction of the candidates that are available in the LOCAL candidate set. We reduce the candidate set down to 1 percent of its original size compared to $\mathcal{F}_{\text{LOCAL}}$. In general, we know that these MDL reductions are increasingly better for lower θ (see Figure 5.4).

Table 5.2: **Candidate Set Reduction.** For various experiments, we see that when changing from $\mathcal{F}_{\text{Global}}$ to $\mathcal{F}_{\text{Local}}$ the local table *minsup* adds a large number of candidates. However, the $\mathcal{F}_{\text{Local}}$ is reduced significantly by our proposed $\mathcal{F}_{\text{R-Krimp}}$.

database	$\lvert\mathcal{F}_{\text{Local}}\rvert$	$\lvert\mathcal{F}_{\text{R-Krimp}}\rvert$	$\frac{\lvert\mathcal{F}_{\text{R-Krimp}}\rvert}{\lvert\mathcal{F}_{\text{Local}}\rvert}$ %	$\lvert\mathcal{F}_{\text{Global}}\rvert$	$\frac{\lvert\mathcal{F}_{\text{Global}}\rvert}{\lvert\mathcal{F}_{\text{Local}}\rvert}$ %
2-tables					
loan - transaction	262510	3431	1	12809	5
gene - interaction	37346	11032	27	1043	3
paper - paper	286065	30767	11	5043	2
bio - hemat	153049	14535	9	5887	4
bio - ifn	45017	477	1	10152	23
hemat - ifn	39737	4198	11	999	3
3-tables					
paper - paper - paper	341295	91798	27	1316	1
bio - hemat - ifn	9200794	83459	1	116649	1

Approximate the Quality

We have successfully reduced the candidate sets, the question now becomes, at what cost? First of all, we see that the Local approach does indeed manage to derive a smaller encoded length than Global. In all cases shown in Table 5.4, we see that the encoded length is up to 10 percent smaller.

In general, we see in all cases that R-Krimp approximates the Local approach very well. The encoded lengths in all cases resemble Local much more than Global. When we look at the efficiency of the encoding in terms of the number of code table elements ($CT_{\text{Local}}, CT_{\text{R-Krimp}}, CT_{\text{Global}}$), we also see a good approximation. In general, R-Krimp not only attains the same encoded length, but also uses roughly a similar sized set of patterns.

The main goal is to achieve a similar encoding length as Local with R-Krimp. However, it is interesting whether our candidate reduction keep up ahead of the worst performing approach: Global. In all cases, we see in Table 5.3 that we outperform Global. In a few cases, R-Krimp even has less code table elements in its code table than Global. With this smaller code table it attains a more efficient encoding. For example, in the *loan-transaction* database, we achieve a better compression with R-Krimp than Global - with less code table elements (see Tables 5.3 and 5.4).

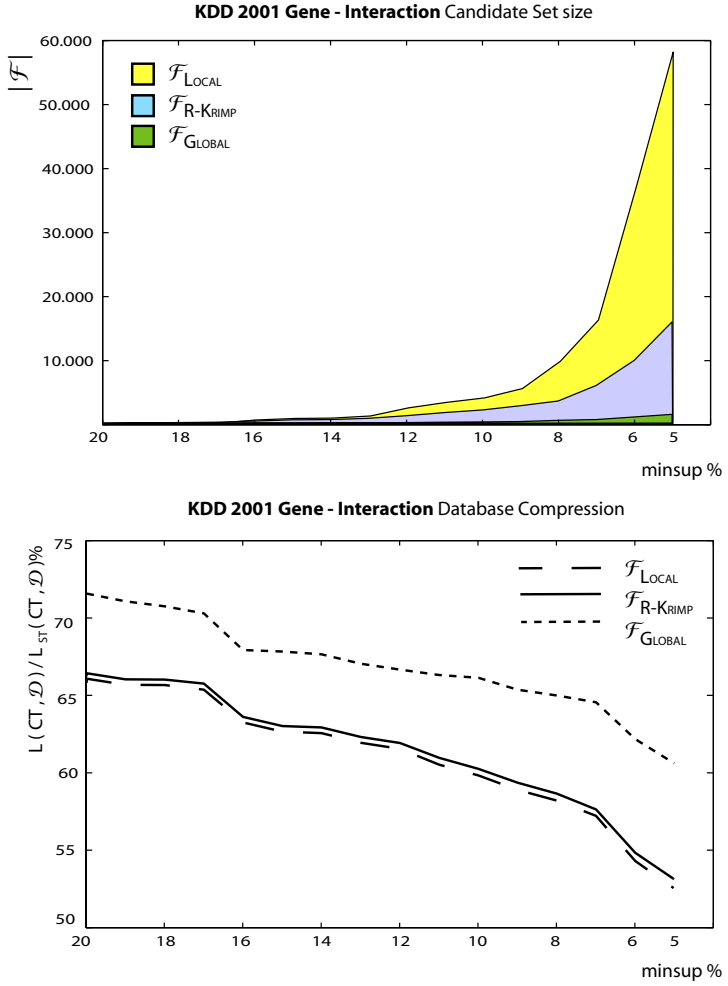


Figure 5.4: Here we show the results for the gene-interaction database. (top) The R-KRIMP candidate set is much smaller than the LOCAL case and reduced the exponential growth. (bottom) The $\mathcal{F}_{\text{GLOBAL}}$ reduction suffers severely in terms of compressed length $L(CT, \mathcal{D})$. R-KRIMP still approximates the LOCAL case very well.

Table 5.3: CT_{Local} is approximated very closely by $CT_{\text{R-Krimp}}$. We show for all three approaches the resulting number of patterns in CT. Note that we only show the non-singleton elements, that is $|CT \setminus \mathcal{I}|$. The pattern set reduction is up to several orders of magnitude.

database	$\lvert CT_{\text{Local}}\rvert$	$\lvert CT_{\text{R-Krimp}}\rvert$	$\lvert CT_{\text{Global}}\rvert$	$\frac{\lvert\mathcal{F}_{\text{Local}}\rvert}{\lvert CT_{\text{R-Krimp}}\rvert}$ %	$\lvert CT_{\text{R-Krimp}}\rvert$
2 relations					
loan - transaction	812	283	430	0.11	66×138
gene - interaction	1892	1915	181	5.13	71×99
paper - paper	744	448	468	0.16	158×184
bio - hemat	847	836	122	0.55	60×151
bio - ifn	18	18	33	0.04	18×4
hemat - ifn	84	84	149	0.21	83×2
3 relations					
paper - paper - paper	1863	1369	320	0.01	$79 \times 59 \times 65$
bio - ifn - hemat	512	811	342	0.40	$70 \times 224 \times 35$

Return what is Relevant

As a final remark on the results we like to focus on the capabilities to select a small characteristic set of relational patterns. We have shown in previous work that Krimp can select a small set of patterns. In the shown results, see Table 5.2, we see that the same still holds for relational data. To make an honest comparison, we need to compare the original candidate set $\mathcal{F}_{\text{Local}}$ with the resulting relational code table $\text{CT}_{\text{R-Krimp}}$. We do not compare $\mathcal{F}_{\text{R-Krimp}}$ with $\text{CT}_{\text{R-Krimp}}$, as both the candidate reduction as well as the resulting pattern set reduction both are fruits of our approach. On average we see a reduction of several orders of magnitude. For example, for the HEP paper citation database over 3 tables, down to 0.01% of the original candidates remain as characterising patterns. Similar to Krimp, R-Krimp obtains increasingly better reduction results for lower minsups.

5.7 Discussion

Finding relational patterns is associated with a high space complexity and a large associated candidate space. We solve the first issue with an approach we call Local which does not require a materialisation of the join. To solve the last issue, of an exponential candidate set, we extend this approach to our most successful approach R-Krimp. The growth of the candidate set, $\mathcal{F}_{\text{R-Krimp}}$, is much less for lower minimal supports, solving the exponential growth issue.

Table 5.4: CT_{LOCAL} is approximated very closely by $CT_{\text{R-KRIMP}}$. More importantly, it also achieves a similar relative encoded length L of the database (divided by the standard encoding L_{ST}). The pattern reduction is up to several orders of magnitude.

database	$\frac{L_{\text{LOCAL}}}{L_{ST}}$ %	$\frac{L_{\text{R-KRIMP}}}{L_{ST}}$ %	$\frac{L_{\text{GLOBAL}}}{L_{ST}}$ %
2 tables			
loan - transaction	35.89	36.20	39.37
gene - interaction	55.78	55.96	62.76
paper - paper	54.65	54.20	56.78
bio - hemat	68.54	68.98	69.54
bio - ifn	70.76	73.17	73.34
hemat - ifn	75.04	75.07	75.20
3 tables			
paper - paper - paper	73.26	76.17	82.60
bio - ifn - hemat	75.02	75.20	75.54

With respect to all approaches, the candidate set associated with LOCAL, $\mathcal{F}_{\text{LOCAL}}$, provides the candidates for the best description of the complete relation. R-KRIMP clearly grasps a similar set of relevant patterns well using a much smaller candidate set. With a much lower computational complexity we obtain almost the same relational encoding. As both space complexity and candidate set size are related to the time complexity, both reductions make R-KRIMP suitable to mine relational patterns on large databases.

Beforehand, it is hard to determine whether a pattern in the data is best described on a single table solely or over multiple tables in a relational form. R-KRIMP makes this selection via MDL automatically based on finding the best description, and determines in what dimension to capture patterns. With success; our relational code tables contain patterns that contain itemsets over all tables or a subset thereof.

5.8 Conclusions

Given a relational database, we find a small set of relational patterns in a symmetric fashion. Example questions that we would like to answer in a relational setting are: '*What record titles and book titles are frequently sold together*'. We have shown that we can find these patterns that describe the relation well. Our relational R-Krimp code tables consists of those patterns containing relevant relational information. Itemsets that otherwise would have not been shown relevant, now appear as interesting item sets. Using R-Krimp, we obtain a small description of the complete relation in the form of our relational code tables.

Multi-relational pattern mining suffers even more from the well-known frequent pattern explosion than the single table case. In this chapter, we show, however, that by using MDL one can still mine for the descriptive frequent relational itemsets in an effective way. In particular, the R-Krimp algorithm is both space and time efficient while its result is a good approximation of the far too expensive naïve approach.

One could paraphrase the results of this chapter by the quasi formula:

$$\begin{aligned} \text{R-Krimp}(T_1, \ldots, T_n) &= \text{Krimp}(\prod_{i=1}^{n} \text{Krimp}(T_i)) \\ &\approx \text{Krimp}(\bowtie_{i=1}^{n} T_i) \end{aligned}$$

In that sense it is a further confirmation that MDL in general and Krimp in particular picks the itemsets that matter. The advantage of R-Krimp over other methods is not only its efficiency in both space and time complexity, but also that it scales very well. For, the size of $\text{Krimp}(T)$ grows far less quickly than the size of T.

Finally, due to the excellent scaling properties of R-Krimp, mining in such generalised cases remains tractable without getting swamped in patterns.

Chapter 6

Characteristic Relational Patterns

Research in relational data mining has two major directions: finding global models of a relational database and the discovery of local relational patterns within a database. While relational patterns show how attribute values co-occur in detail, their huge numbers hamper their usage in data analysis. Global models, on the other hand, only provide a summary of how different tables and their attributes relate to each other, lacking detail of what is going on at the local level.

In this chapter we introduce a new approach that combines the positive properties of both directions: it provides a detailed description of the complete database using a small set of patterns. More in particular, we utilise a rich pattern language and show how a database can be encoded by such patterns. Then, based on the MDL-principle, the novel RDB-Krimp algorithm selects the set of patterns that allows for the most succinct encoding of the database. This set, the code table, is a compact description of the database in terms of local relational patterns. We show that this resulting set is very small, both in terms of database size and in number of its local relational patterns: a reduction of up to 4 orders of magnitude is attained.

[1]This is an extended and edited version of an earlier paper published as: [57]. *Characteristic Relational Patterns.* A.C.M. Koopman, and A.P.J.M. Siebes. In: KDD 2009, pages 437-446.

6.1 Introduction

Relational data mining seeks to generalise traditional, single-table data analysis to the analysis of multiple inter-related tables. As relational models are more expressive than their single table counterparts, they allow for a more succinct description of the database than when one has to summarise each and every table and all their connections separately. Current research in relational data mining follows one of two directions. Either one aims to find a *'global'* model of the complete database, or one seeks interesting *'local'* patterns in the database.

Both directions have their specific merits and shortcomings. A *'global'* database model, for example a Probabilistic Relational Model (PRM) [55], is often small and interpretable, but it lacks detail as it only shows how the attributes of the tables interact. On the other hand, one can use an approach like WARMR to find frequent patterns [2,21]. These patterns can be regarded as *'local'* models that describe a partial structure of the relational database. A well-known drawback of this method is the exponential pattern set growth when mining for less frequent, but often interesting, patterns.

In this chapter we present a new approach, RDB-Krimp, that combines the strengths of both approaches. It finds a global model that describes the complete database using only a small set of *characteristic* relational patterns. While collectively the patterns show the global structure, individually they reveal the local interactions in the database. A global model that describes the complete relational database must capture the behaviour of all tables and their interactions, and thus requires a focus on all tables. While frequent pattern mining techniques prune the pattern search space using a *single* 'target' table, *all* tables function as 'targets' in our global approach. Hence, our pattern language is essentially that of FARMER [71] except we do not restrict the patterns to have their 'roots' in a single target table.

Given this pattern set, we use the Minimum Description Length (MDL) principle [32] to select a small set of *characteristic* patterns for the database. RDB-Krimp uses the MDL-principle to find those patterns that together describe the database well. In contrast to Krimp [79], this enriched pattern language allows RDB-Krimp to find more complex patterns in the database. However, finding these richer models for *relational* databases requires a novel lossless encoding scheme that relies on its ordering. During the encoding, RDB-Krimp reorders the tuples based on the patterns of the model such that the encoded database can always be decoded in a lossless fashion. RDB-Krimp proves effective: with its lossless encoding, it can find models that stay compact and utilise our enhanced pattern language.

After presenting our theoretical framework and the RDB-Krimp algorithm, we validate our claims with experimental results obtained on publicly available

KDDcup databases. We show that the global models stay compact, in contrast to the original candidate pattern sets that grow large for lower minimum supports.

Furthermore, we show that we attain good results as a result of the specific characteristics of our enhanced pattern language. We show that simpler pattern sets lead to worse results and that target table based approaches are much less effective. We conclude by discussing the readability of our models, and our related work.

6.2 Data and Patterns

In this section we formally define the data type, relational patterns, how such patterns occur in the database, and how to calculate the support of a pattern. Although we already introduced a relational database and its patterns in Chapter 5, note that the definitions inhere are less restricted. In this chapter we do not restrict our database to be a path of connected tables, but instead allow it, and its patterns, to be more general.

Data

We assume that the data resides in a multi-relational database in which the relations between the tuples in the various tables is coded, as usual, via *foreign keys*. We assume that any pair of two tables have at most one foreign key relationship between them. This is without loss of generality as databases can always be losslessly recoded such that this assumption holds. Moreover, we assume that all attributes of all tables have a categorical domain. To introduce our notation, we give a brief formal description.

The database $\mathcal{D}$ consists of a set of tables, $\mathcal{D} = \{T^1, \ldots, T^n\}$, and we assume that all the table names (the T^i) are unique. Each table T has a schema $S(T)$. This schema consists of a *key*, 0 or more *foreign keys*, and 1 or more attributes, i.e,

$$S(T^i) = (K^i, \mathcal{FK}^i, \mathcal{A}^i)$$

in which:

- K^i is the key. Without loss of generality, we assume that the key name is unique. Its domain, $Dom(K^i)$, is a set of integers.

- $\mathcal{FK}^i$ is a set of 0 or more foreign keys. We assume that the database schema is consistent, that is

T^i			
K^i	FK^i_j	A^i_1	A^i_2
k	k_j	v_1	v_2

T^1 = ACCOUNT

$account_{ID}$	Frequency	Date
10	2	06/2007
11	3	03/2006
12	3	08/2006
13	2	03/2006
14	1	05/2008

T^2 = LOAN

$loan_{ID}$	$account_{ID}$	Date	Amount	Duration	Payment
30	10	06/2008	10245	12	A
31	10	09/2008	13722	24	B
32	11	08/2006	27313	36	B
33	12	09/2006	27147	12	B
34	12	05/2008	27194	36	D
35	13	09/2008	30289	12	B
36	13	06/2008	18203	12	C

T^3 = ORDER

$order_{ID}$	$account_{ID}$	Bank-To	Amount-To	Amount	Type
20	10	ST	141	1000	UVER
21	10	QR	359	2000	SIPO
22	11	YZ	850	1000	SIPO
23	13	ST	283	1000	NULL
24	13	OP	850	2000	SIPO

T^4 = DISPOSITION

$disp_{ID}$	$account_{ID}$	Type
40	10	OWNER
41	11	DISPONENT
42	11	OWNER
43	12	DISPONENT
44	12	OWNER

Figure 6.1: An illustrative relational database: an excerpt from the Financial database.

  - For each foreign key FK^i_j of T^i, there is a table, say, $T^l \in \mathcal{D}$ for which it is the key, i.e., $FK^i_j = K^l$. As noted above, we assume that there is at most one foreign key in T^i that refers to K^l
  - The domain of FK^i_j is the domain of that K^l.

– $\mathcal{A}^i$ consists of 1 or more attributes. Each attribute A^i_k has a categorical domain $Dom(A^i_k)$.

– Summing up, the domain of table T^i denoted by $Dom(T^i)$ is the cartesian product of all domains involved, i.e.,

$$Dom(T^i) = Dom(K^i) \times \prod_{FK^i_j \in \mathcal{FK}^i} Dom(FK^i_j) \times \prod_{A^i_k \in \mathcal{A}^i} Dom(A^i_k)$$

In our example database, shown bottom-right in Figure 6.1, the `DISPOSITION` table has as key: $disp_{ID}$ (depicted in bold) and as foreign key $account_{ID}$. Moreover, it has an attribute *Type* whose domain is: $\{Owner, Disponent\}$.

Next to a schema, each table has an *extend*, consisting of a set of tuples. As usual, we blur the distinction between the table and its extend and say that a tuple t is in table T^i; denoted by $t \in T^i$. The database as whole should satisfy

referential integrity. That is, foreign-key values in a tuple refer to existing tuples in the table for which this foreign key is the key. More formally we have:

A tuple for table T^i with $S(T^i) = (K^i, \mathcal{FK}^i, \mathcal{A}^i)$ is given by:

$$t = (K^i = k, \{FK^i_j = K_j\}, \{A^i_l = v_l\})$$

in which:

- $k \in Dom(K^i)$,
- the tuple has one entry for each $FK^i_j \in \mathcal{FK}^i$ and one entry for each $A^i_j \in \mathcal{A}^i$.
- let K_l be the key to which FK^i_j refers, then $k_j \in Dom(K^l)$,
- for referential integrity on the database $\mathcal{D}$, we have that for any tuple $t \in T^i$, there is a tuple $t' \in T^l$ such that $\pi_{FK^i_l}(t) = \pi_{K^l}(t')$.
- $v_l \in Dom(A^i_l)$.

Again as usual, we will suppress the labels in tuples whenever possible. That is, we simply write $(40, 10, Owner) \in$ `DISPOSITION` for a tuple in our example database in Figure 6.1.

Patterns

The prototypical example of patterns are itemsets. In the case of a (single) table of categorical data, an itemset generalises to a *selection*. Clearly such patterns should be included into our pattern language. However, "true" relational patterns should cross multiple tables. That is, they should describe related selections over multiple tables.

After formally introducing our pattern definition, we will illustrate it with an example.

Definition 20 (Pattern)**.** *Let $\mathcal{D} = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{FK}^i, \mathcal{A}^i)$.*

- *Let $\{A_1, \ldots, A_l\} \subseteq \mathcal{A}^i$ and let $v_j \in Dom(A_j)$, then the expression F defined as*

  $$F = T^i(\{A_1 = v_1, \ldots, A_l = v_l\})$$

  is a pattern *for T^i; this is denoted by $F \in \mathcal{F}^i$.*

– *Let T^i have key K^i, moreover, let K^i be a foreign key of T^j; that is, there is an $FK^j_l \in \mathcal{FK}^j$ such that $FK^j_l = K^i$. Let $F_0 \in \mathcal{F}^i$ and let $\{F_1, \ldots F_k\} \subseteq \mathcal{F}^j$. The expression F defined as*

$$F = F_0[F_1, \ldots, F_k]$$

is a pattern for T^i, i.e., $F \in \mathcal{F}^i$.

– *Let T^i have key K^i, moreover, let K^i be a foreign key of the q tables $T^{j_1}, \ldots, T^{j_q}$, such that if $r \neq s$, then $T^{j_r} \neq T^{j_s}$. Let $F_0 \in \mathcal{F}^i$ and for $l \in \{1, \ldots, q\}$, let $\{F^l_1, \ldots F^l_{k_l}\} \subseteq \mathcal{F}^{j_l}$. The expression F defined as*

$$F = F_0[[F^1_1, \ldots, F^1_{k_1}], \ldots, [F^q_1, \ldots F^q_{k_q}]]$$

is a pattern for T^i, i.e., $F \in \mathcal{F}^i$.

The third component is a generalisation of the second, which is included to simplify part of the following definitions.

In our pattern definition we define the *alphabet patterns* as those patterns that select one attribute and assign it to one value (i.e. $F = T^i(A_j = v_j)$).

Also, we define the *size* of a pattern as the number of attributes within the pattern:

$$\begin{aligned} &size(F = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})) = j \\ &size(F_0[[F^1_1, \ldots, F^1_{k_1}], \ldots, [F^m_1, \ldots F^m_{k_m}]]) = \\ &\quad size(F_0) + \textstyle\sum_{i=1}^{m} \sum_{j=1}^{k_i} size(F^i_j). \end{aligned}$$

Using our example database shown in Figure 6.1, we can now illustrate our pattern definition. An example of the first component, a single table selection on the `ACCOUNT` table, would be `ACCOUNT`({Frequency =2}). This pattern can be seen in the database in the set of tuples with account-ids 10 and 13.

The second component allows a pattern to have multiple selections from a table. As an example, in our database we see the pattern that *accounts*s of *Frequency 2* have *dispositions* of type *Owner* (seen at account-id 10). A more complex example is that there are *accounts* of *Frequency 3* that have both *dispositions* of type *Owner* and *dispositions* of type *Disponent*. For this last pattern, one tuple in the `ACCOUNT` table is related to two distinct tuples in the `DISPOSITION` table (account-ids 11 and 12). This last pattern is represented as follows: `ACCOUNT`({Frequency = 3}) [`DISPOSITION`({Type = Owner}), `DISPOSITION`({Type = Disponent})].

The third component extends this last example by allowing the patterns to span more than two tables.

Note that in our pattern language a pattern can 'start' at one of the tables $T \in \{T^1, \ldots, T^n\}$. We do not restrict the 'root' of these patterns to be at one specific target table, because interesting patterns within the database can start from all possible tables. Furthermore, as one single tuple can be joined with multiple other tuples, the structure of our patterns matches the complexity of the database.

Having a richer pattern language is no virtue in its own right. Alternatively, we could have used pattern languages such as those used in WARMR [21], FARMER [71], or in our earlier work [56]. The experiments however show that our pattern set, consisting of FARMER pattern sets without a fixed target table, allows us to capture more important structure within the data.

Pattern Occurrences

Our the patterns can become rather complicated structures, containing lists of lists of lists of ... of tuples. Hence, it is illustrative to consider what the domain of such patterns is. That is, what does an instance look like? The definition of these domains follows the inductive structure of the patterns.

Definition 21 (Domain)**.** *Let $\mathcal{D} = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{FK}^i, \mathcal{A}^i)$. Moreover, let $F \in \mathcal{F}^i$. The domain of F, denoted by $Dom(F)$, is given by*

- *If $F = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})$, then $Dom(F) = Dom(T^i)$.*
- *If $F = F_0[F_1, \ldots, F_k]$, then*

$$Dom(F) = Dom(F_0) \times [Dom(F_1), \ldots, Dom(F_k)].$$

- *If $F = F_0[[F_1^1, \ldots, F_{k_1}^1], \ldots, [F_1^q, \ldots F_{k_q}^q]]$, then $Dom(F) =$*

$$Dom(F_0) \times [[Dom(F_1^1), \ldots, Dom(F_{k_1}^1)], \ldots, [Dom(F_1^q), \ldots Dom(F_{k_q}^q)]].$$

Note that this domain definition is broad: it does not enforce that patterns describe related tuples. The reason for this liberal definition is that here we are only interested in the general structure of the domain. Referential integrity does play its role in the definition of an occurrence of a pattern.

While these domains may have a rather complicated structure, there is some simplicity. For a pattern $F \in \mathcal{F}^i$, the domain is either $Dom(T^i)$ or it is the cartesian product of $Dom(T^i)$ with a complicated list domain. That is, $Dom(F) = Dom(T^i) \times X$, in which X denotes a list domain. This observation has a useful consequence. It means that if F is a pattern for T^i (i.e., $F \in \mathcal{F}^i$), and t is an *instance* of F (i.e, $t \in Dom(F)$), we can project t on the keys,

Algorithm 9 Generate

Generate($\mathcal{D}, \theta$)

1. **for all** $T^i \in \mathcal{D}$ **do**
2. $\quad \mathcal{F}^i_\theta \leftarrow$ Farmer($\mathcal{D}, \theta, target = T^i$)
3. $\mathcal{F}_\theta \leftarrow \bigcup_i \mathcal{F}^i_\theta$
4. **return** $\mathcal{F}_\theta$

the foreign keys and the attributes of T^i. We will use these projections in the definition of an occurrence.

An occurrence of a pattern in the database is an instance of that pattern in the database. However, different from these instances, for occurrences we do require *referential integrity*. That is, the occurrences should consist of related tuples only. This makes the definition slightly more complex.

Definition 22 (Occurrence)**.** *Let $\mathcal{D} = \{T^1, \ldots, T^n\}$ be a database for which each table T^i has schema $S(T^i) = (K^i, \mathcal{FK}^i, \mathcal{A}^i)$. Moreover, let $F \in \mathcal{F}^i$.*

- *If $F = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})$, $occ(F)$ is the set of those tuples $t \in T^i$ for which*
$$\forall k \in \{1, \ldots, j\} : \pi_{A_k}(t) = v_k$$
- *If $F = F_0[F_1, \ldots, F_k]$, we know that $F_0 \in \mathcal{F}^i$ and that there is a table T^j such that firstly $\{F_1, \ldots, F_k\} \subseteq \mathcal{F}^j$ and secondly that the key K^i of T^i is a foreign key, say FK^j_l of T^j. An instance $t = (t_0, [t_1, \ldots, t_k])$ of F is an occurrence of F, denoted by $t \in occ(F)$, if:*
  - $t_0 \in occ(F_0)$ *and* $\forall m \in \{1, \ldots, k\} : t_m \in occ(F_m)$.
  - $\forall m, n \in \{1, \ldots, k\} : m \neq n \rightarrow t_m \neq t_n$
  - $\forall m \in \{1, \ldots, k\} : \pi_{FK^j_l}(t_m) = \pi_{K^i}(t_0)$
- *If $F = F_0[[F^1_1, \ldots, F^1_{k_1}], \ldots, [F^q_1, \ldots F^q_{k_q}]]$, then an instance $t \in Dom(F)$ given by*
$$t = (t_0, [[t^1_1, \ldots, t^1_{k_1}], \ldots, [t^q_1, \ldots, t^q_{k_q}]])$$
is an occurrence of F (i.e., $t \in occ(F)$), if
$$\forall l \in \{1, \ldots, q\} : (t_0, [t^l_1, \ldots, t^l_{k_l}]) \in occ(F_0[F^l_1, \ldots, F^l_{k_l}]).$$

As usual, the *support* of a pattern is the number of its occurrences. We say that a pattern is frequent if the support exceeds some user-defined threshold called the minimum support θ.

Note that we use lists in our occurrence definition. Each occurrence can span multiple tuples within one table T^i that are stored in a list $[t^i_1, \ldots, t^i_{k_i}]$. We will preserve this order of the tuples as it is essential to encode the database in a lossless manner, as we will show below.

Given this pattern definition, in order to derive the set of frequent patterns $\mathcal{F}_\theta$ one can resort to existing relational mining algorithms like FARMER [71] or attribute tree miners like FATminer [47]. Partial frequent pattern sets generated by either approach can be combined into $\mathcal{F}_\theta$ (see Algorithm 9).

Finally, we define a canonical order on our patterns. We assume for each table T^i, each attribute A_j, and each attribute value v_k a unique (string) label. We denote by $l(X)$ the unique label assigned to X ($X \in \{T^i, A_j, v_k\}$). Canonical forms are simply strings, hence we have the familiar lexicographic order, denote by $<_{lex}$, on them. Using this order, we define:

$$\begin{aligned} &\mathit{canonical}(F = T^i(\{A_1 = v_1, \ldots, A_j = v_j\})) = \\ &\quad l(T^i) : \{l(A_1) : l(v_1), \ldots, l(A_j) : l(v_j)\} \end{aligned}$$

$$\begin{aligned} &\mathit{canonical}(F_0[[F^1_1, \ldots, F^1_{k_1}], \ldots, [F^m_1, \ldots F^m_{k_m}]]) = \\ &\quad \mathit{canonical}(F_0) :^m_{i=1}:^{k_i}_{j=1} \mathit{canonical}(F^i_j) \end{aligned}$$

This allows us to define a canonical order on our patterns:
$F_0 <_{can} F_1$ iff $\mathit{canonical}(F_0) <_{lex} \mathit{canonical}(F_1)$.

6.3 Problem Statement

Now we have defined our patterns and database, we can present our problem formally.

The data that is encoded by our model resides within a relational database as defined in Section 6.2, or more specifically within its attribute data. Similar to earlier chapters our models are code tables CT.

We encode the relational database using the patterns from a code table CT. Figure 6.2 shows an example on how this encoding comes about through a database cover. Here, we show two patterns that (partially) cover the database. Using our pattern notation, we write the first code table pattern as:

$$\begin{aligned} &\texttt{ACCOUNT}(\{\text{Frequency} = 2\})[\\ &\quad [\texttt{ORDER}(\{\text{Bank-To} = \text{ST}, \text{Amount} = 1000\}), \\ &\quad \texttt{ORDER}(\{\text{Amount} = 2000, \text{Type} = \text{SIPO}\})] \\ &\quad [\texttt{LOAN}(\{\text{Date} = 06/2008, \text{Duration=12}\}), \\ &\quad \texttt{LOAN}(\{\text{Date} = 09/2008, \text{Payment=B}\})]] \end{aligned}$$

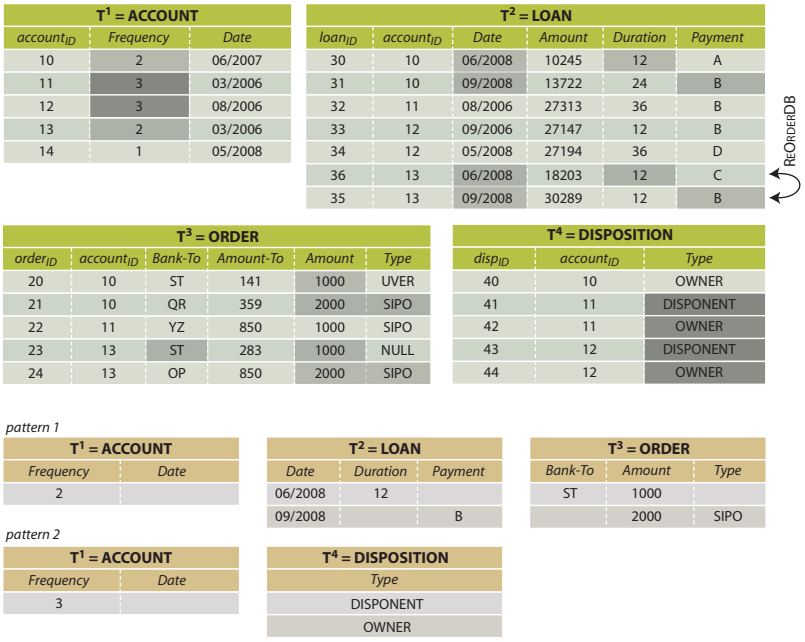


Figure 6.2: The database is partially covered with two code table patterns using RDB-Krimp. The uncoloured part of the database can be covered by alphabet patterns. Note that for a lossless decoding we incorporate the database order ReorderDB (seen at the swap of `LOAN`:loan-id=35 and 36).

We cover the database by replacing the related attribute values with the code of the pattern. As this pattern occurs at two yet uncovered locations in the database (id=10 and 13), it is used to describe this part of the database. Note that in this figure each code table element has its own distinct colour.

In order for the encoding to be lossless, we need to be able to decode every part of the occurrence. As an occurrence may span multiple tuples within the database, we need to write the code at each tuple covered by this occurrence. Furthermore, as each pattern has just one code the necessity of a database- and pattern order becomes clear: we need to know which tuple is covered with which pattern from the list.

We match the order of the tuples within the database with the order of the pattern. In the first pattern (F_1 in Figure 6.2), `LOAN` : $\{Date = 06/2008,$

Algorithm 10 ReOrderDB

ReOrderDB $(reorder, [t_1, \ldots, t_i])$
1. **if** $[t_1, \ldots, t_i] \cap reorder \subseteq^{ord} reorder$ **then**
2. $\quad reorder = reorder \cup [t_1, \ldots, t_i]$
3. $\quad$ **return true**
4. **else**
5. $\quad$ **return false**
6. **end if**

$A : [a_1, \ldots, a_n] \subseteq^{ord} B : [b_1, \ldots, b_m]$
7. **if** $A = \varnothing$ **then**
8. $\quad$ **return true**
9. **else if** $B = \varnothing$ **then**
10. $\quad$ **return false**
11. **else if** $a_1 = b_1$ **then**
12. $\quad$ **return** $[a_2, \ldots, a_n] \subseteq^{ord} [b_2, \ldots, b_m]$
13. **else if** $a_1 \neq b_1$ **then**
14. $\quad$ **return** $[a_1, \ldots, a_n] \subseteq^{ord} [b_2, \ldots, b_m]$
15. **end if**

$Duration = 12\}$ is ordered before `LOAN` : $\{Date = 09/2008, Payment = B\}$. Note the swap of tuples 35 and 36 to align the database order with the order of the pattern. Once covered, the complete database is encoded and looks like a mosaic, which can be decoded using the database order, and the code table (see Figure 6.2).

So, for unambiguous decoding, the order of the tuples in the database has to be aligned with the order in the code table patterns. Therefore, we allow a (partial) re-ordering of the tuples in the database. However, a pair of tuples is only re-ordered once, otherwise the unambiguous decoding property will be lost. Hence, we keep track of the order. Initially this ordered list, *reorder*, is empty. Whenever two (or more tuples) that are not yet in the list are re-ordered their identifiers are appended.

We cover the database with patterns F until the database is completely covered. An occurrence of a pattern can only cover the database if the database order can be aligned to match the pattern. This is the case when the tuples related to the occurrence are not yet ordered, or if they are already ordered in the correct order (e.g. the order of the tuples matches the order of the pattern). We update and check the partial database order via the ReOrderDB algorithm (see Algorithm 10). The algorithm takes as input the current (partial) database order *reorder* and a list of tuples of the current occurrence $[t_1, \ldots, t_i]$. Initially the database order *reorder* is an empty list as

Algorithm 11 OrderedCover and CoverDB

OrderedCover $(\mathcal{D}, F, reorder)$

1. $count \leftarrow 0$
2. **for all** $\{t\} \in occ(F)$ **do**
3. **if** $F \subseteq \{A^t\}$ **then**
4. **if** ReOrderDB$(reorder, \{t\})$ **then**
5. $\{A^t\} \leftarrow \{A^t\} \setminus F$
6. $count{+}{+}$
7. **end if**
8. **end if**
9. **end for**
10. **return** *count*

CoverDB $(\mathcal{D}, CT)$

11. $reorder \leftarrow \varnothing$
12. **for all** $F \in CT$ **do**
13. $count(F) \leftarrow$ OrderedCover$(\mathcal{D}, F, reorder)$
14. $usage(F) = count(F) \times |coverspots(F)|$
15. **end for**

the database is unordered. To check whether $[t_1, \ldots, t_i] \cap reorder$ is an ordered subset of *reorder* we use the $\subseteq^{ord}$ operator (line 1). If so, the order is updated by appending the current list of tuples that are not yet part of the database order (2). Otherwise, this particular occurrence cannot be used to cover the database.

Now that we can adjust *reorder* to align with a pattern occurrence, we can partially cover a database given a single code table pattern (see Algorithm 11). For each pattern, we have a set of occurrences $occ(F)$ (2). OrderedCover iterates over all occurrences and evaluates whether it can be covered (2). We only cover the current occurrence (the list of tuples) if all related attributes are still uncovered (3), and if the current occurrence is an ordered subset of the current database order (4). If so, the attributes of the occurrence are covered and the counts are updated (5,6). We define the *count* of a pattern as the number of times we cover the database with it.

Using the code table CT, we cover the complete relational database using the CoverDB algorithm (see Algorithm 11). Considering the patterns in the code table, it covers the database using the OrderedCover algorithm (13-14). During the cover process, we obtain for each code table pattern its *count* (line 13).

To be able to decode the encoded database, we have to write its code at

Algorithm 12 RDB-Krimp

RDB-Krimp $(\mathcal{D}, \mathcal{F}_\theta)$

1. $CT \leftarrow \mathcal{I}$
2. **for all** $F \in \mathcal{F}_\theta$ **do**
3. $\quad CT' \leftarrow CT \cup F$ in order
4. $\quad$ CoverDB$(\mathcal{D}, CT')$
5. $\quad$ **if** $L(\mathcal{D}, CT') < L(\mathcal{D}, CT)$ **then**
6. $\quad\quad CT \leftarrow CT'$
7. $\quad$ **end if**
8. **end for**

each involved tuple. We define *coverspots* as the set of tuples at which we need to write down the code for pattern F. More formally, we define *coverspots* as:

$$coverspots(F = T^i(\{A_i = v_i\})) = \{t\}$$
$$coverspots(F_0[[F_1^1, \ldots, F_{k_1}^1], \ldots, [F_1^m, \ldots F_{k_m}^m]]) =$$
$$coverspots(F_0) \cup_{i=1}^{m} \cup_{j=1}^{k_i} coverspots(F_j^i)$$

We denote the number of *coverspots* by: $|coverspots(F)|$.

Consider the first pattern, F_1 , from Figure 6.2. The number of *coverspots* is 5, as each occurrence has five distinct tuples in the database associated with it (one in `ACCOUNT`, two in `ORDER`, and two in `LOAN`). We denote the *usage* as the total number of times we have to write the code given a code table pattern F : $usage(F) = count(F) \times |coverspots(F)|$ (line 14).

Now we have defined $usage(F)$ for the elements within CT, we can derive $L(\mathcal{D}, CT)$ in a similar fashion as described in Chapter 3. Using this, we can now formally define the problem statement.

Problem Statement. *Let $\mathcal{D} = \{T^1, \ldots T^n\}$ be a relational database as defined in Section 6.2. Find the code table CT that minimises $L(\mathcal{D}, CT)$.*

6.4 RDB-Krimp Algorithm

As we have seen, finding the optimal code table, the one that compresses the database best, is a very hard problem. The algorithm we introduce to this end, RDB-Krimp, approximates the optimal code table for a given database. The RDB-Krimp algorithm is shown in pseudo code in Algorithm 12.

RDB-Krimp starts with a database $\mathcal{D}$ and a frequent pattern set $\mathcal{F}_\theta$ as input. To ensure that the complete database can be covered always, the code table is initialized with $\mathcal{I}$ (line 1), which contains all alphabet elements (i.e.

Table 6.1: Database Characteristics. Shown are for each table, the number of tuples ($|T|$), the number of attributes ($|\mathcal{A}|$), the number of keys ($|\mathcal{K}| \cup |\mathcal{FK}|$), and the average number of foreign key values per primary key value ($\overline{join}$).

	table	$\|T\|$	$\|\mathcal{A}\|$	$\|\mathcal{K}\| \cup \|\mathcal{FK}\|$	$\overline{join}$
FINANCIAL	ACCOUNT	682	2	3	5.43
	CLIENT	827	2	3	2
	LOAN	682	5	2	6.70
	CARD	36	2	3	1
	DISP	827	1	4	1.04
	ORDER	1513	3	3	1
GENES	GENES1	862	4	1	6.86
	GENES2	862	4	1	6.86
	INT	910	2	3	1
	META1	4151	4	2	1
	META2	4151	4	2	1
HEPATITIS	BIO	694	5	2	1
	IFN	198	4	2	1
	OLAB	31039	3	2	1
	PATIENT	771	3	1	42.4

$F = T^i(A_j = v_j)$). One by one, it takes a candidate pattern from F_θ and tests whether it contributes to improve compression (2-8). To do this, a new code table CT' is constructed by adding the candidate pattern to the previous code table CT (3). Using this code table we compute a cover of the database (4) and the compressed lengths of the old and new code table are compared (5). If the addition of the new candidate pattern improves compression, it is kept in the code table (6). Otherwise, it is permanently discarded.

6.5 Experiments

To experimentally validate our approach we run experiments on publicly available relational data sets from previous discovery challenges (see Table 6.1). Similar to Chapter 5 these databases are: the *financial* [1], *genes interaction* [2], and *hepatitis* [3] databases. We use FATminer, a frequent attributed tree

[1] http://lisp.vse.cz/challenge/
[2] http://pages.cs.wisc.edu/ dpage/kddcup2001/
[3] http://lisp.vse.cz/challenge/

miner [47] to generate the frequent pattern sets, as our relational patterns can be represented as attributed trees.

Describing the Database

In order to find the optimal model of the database, RDB-Krimp would ideally evaluate *all* patterns. However, in order to be efficient, we evaluate all *frequent* patterns. To measure the effect of the minimum support value, we generate a frequent candidate set $\mathcal{F}_\theta$ for various θ. Given a $\mathcal{F}_\theta$ we compress the database using RDB-Krimp, which results in a code table CT and an encoded database length $L(\mathcal{D}, CT)$.

The effect of sweeping the minimum support is shown in Figure 6.3a. For all used databases, we see that increasingly lower encoded database lengths are obtained for lower minimum support values.

The smaller encoded database lengths relate to the larger available sets of candidate patterns (see Figure 6.3b). In all cases we see that this candidate set growth is exponential. These larger candidate sets contain more patterns that can be inserted in the code table to contribute to the database description.

While we see that $\mathcal{F}_\theta$ grows exponentially for lower values of θ, we do not see this trend in the size of the code table (see Figures 6.3b and 6.3c respectively). Our code table grows for lower values of θ, but still only a small set is necessary to model the data. With respect to the original candidate set, our code tables achieve up to 4 orders of magnitude reduction in terms of number of patterns.

Initial Database Order

As described in Section 6.3, RDB-Krimp re-orders the database in order to ensure a lossless encoding for the database. As the initial database order is not determined by our algorithm, it can potentially influence the resulting tuple order. Therefore, we need to evaluate the extend of its influence on the resulting database order and the resulting encoded database length. In order to evaluate the effect of this initial order, we randomly shuffle the tuples within the tables.

The experiments indicate that the influence on the resulting compression is minimal. The number of patterns used to encode the database is very similar to the original versions and lead to a similar database compression. The deviation for the database encoding for the *financial*, *genes*, and *hepatitis* database is respectively 1.12%, 0.35%, and 0.01%. Hence, RDB-Krimp is robust with respect to the initial database order.

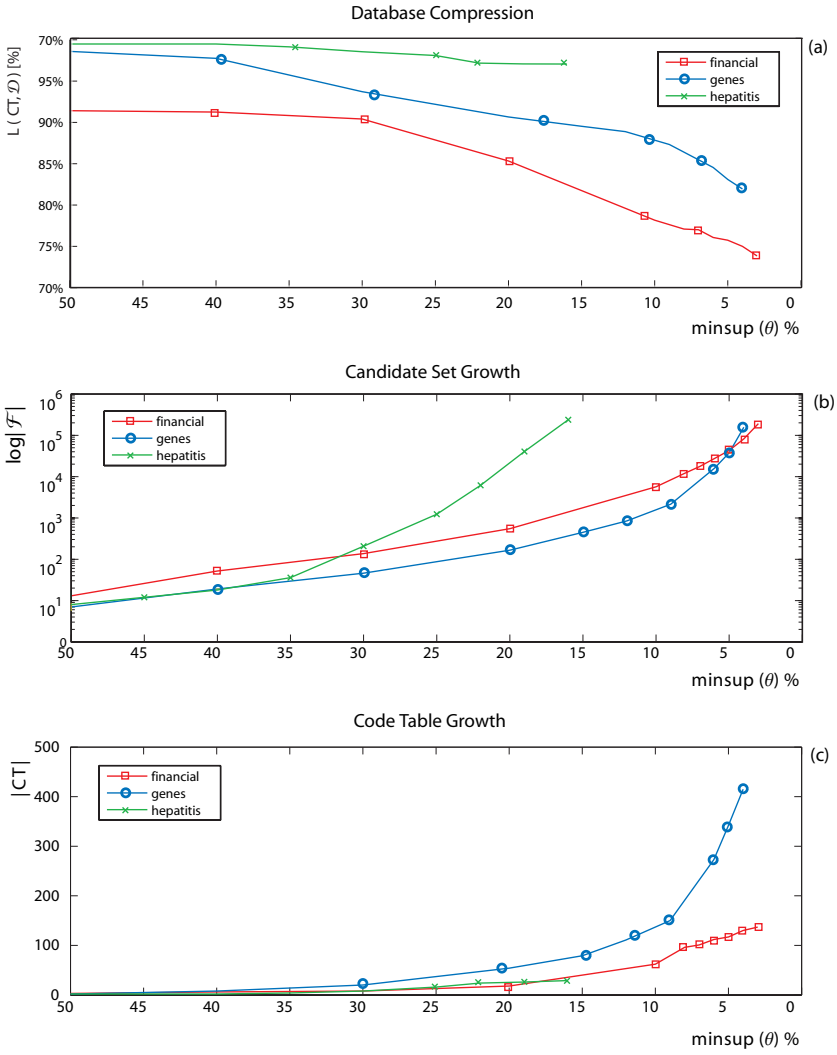


Figure 6.3: Results for different minimum support values θ: (a) The encoded length obtained for the database, (b) the number of frequent patterns, and (c) the number of code table patterns in CT.

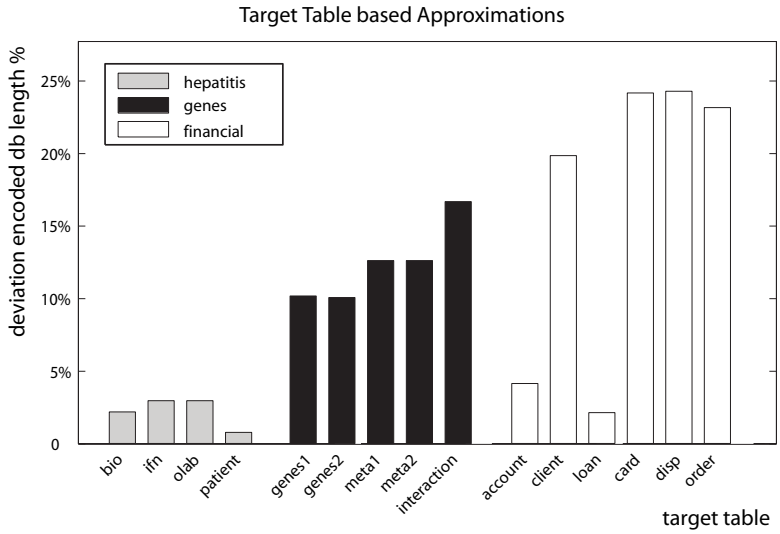


Figure 6.4: Choosing a particular table as a target decreases performance. For the lowest used minimum supports, the use of a target table leads to a worse encoding on all databases. Even for the star-shaped Hepatitis database, which seems well suited for a target table based approach.

Fixing a Target Table

In our approach, we generalise the FARMER pattern language such that we do not rely on a specific target table. We expect a better description of the database by considering patterns starting from all possible tables. In order to evaluate this, we compare the results from alternative trials using a fixed target table as is usual in relational data mining [21,71]. In these experiments, we encode the complete database using solely patterns that originate from a specific target table.

For all possible target tables, we determine how well we can approximate the original result that is obtained for the lowest used minimum support. We have depicted the deviation from the original result in Figure 6.4. We see that fixing the candidate set to a specific target table has a negative influence on the encoding of the database. The compression deteriorates up to 24% compared to the best obtained encoded length. This shows that allowing patterns to start at any table leads to a better description of the database.

The hepatitis database shows some additional interesting results. This database is a prime example of a target table based database, as all (medical test) data surrounds a main table: *patient*. Even in this case, we see that picking a single target table leads to an increase in encoding length (a worse code table). Apparently, we can describe some data better when we do *not* pick the patient table as the single target table.

Comparison to Other Pattern Types

In Section 6.2, we defined a rich pattern language to match the database complexity. To measure whether or not we can describe the database better using more intricate patterns, we here compress the database with an increasingly more general pattern definition. A better database description would lead to smaller encoded database lengths.

A single complex pattern can describe structure in the database that would otherwise require multiple simpler patterns, possibly leading to a better compression. We have used the following characteristic pattern definitions to evaluate:

Single Table Patterns. Here we only allow $F = T^i(\{A_1, \ldots, A_j\})$ patterns in our $\mathcal{F}_\theta$. In other words, all candidate patterns only cover one table, and no joins are allowed. With this candidate set, we compress the database solely with single table patterns.

WARMR-like Patterns. In this candidate set, each pattern covers one tuple per table at most. In our notation, we define these patterns as: $F = F_0[F_1, \ldots, F_k]$. These patterns are similar to those used in WARMR-like approaches, which use an existential quantifier to select strictly one tuple from a table. Unlike WARMR applications, we do allow these patterns to start at any table in the database, in stead of a single target table [21].

Our language. Here we consider patterns as defined in Section 6.2. In this set, the patterns are defined as $F = F_0[[F_1^1, \ldots, F_{k_1}^1], \ldots, [F_1^q, \ldots F_{k_q}^q]]$. Recall that this is similar to grouping all FARMER generated pattern sets for each table.

Note that we order all three candidate sets as defined in Section 6.4. For all above scenarios, we evaluate the effect on the candidate set size $|\mathcal{F}_\theta|$, the number of code table elements $|CT|$, and the compressed encoded length of the database $L(\mathcal{D}, CT)$.

The depicted result of the genes database in Figure 6.5a shows a typical result in terms of compressing the database. We see that the single table patterns lead to the worst encoded length for the database. A better compression can be derived when we allow WARMR patterns in our candidate set, which consequently can be improved by allowing all patterns to cover the database.

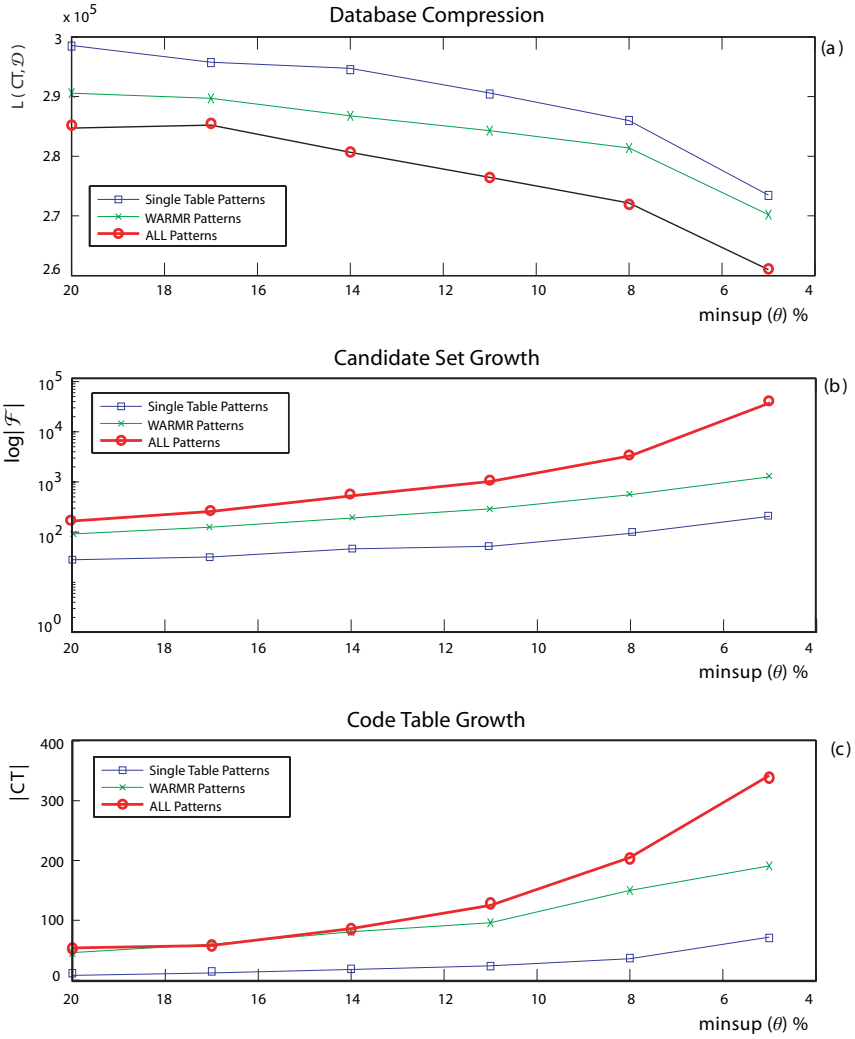


Figure 6.5: Our generalised relational patterns lead to better results. While the number of candidates grows large for low minimum supports (b), we obtain good database encodings (a) using compact code tables (c) (shown for the genes database).

Table 6.2: On all databases more general patterns lead to smaller encoded lengths for the database ($L(\mathcal{D}, CT)$). We also see that more patterns can be found that are relevant ($|CT \setminus \mathcal{I}|$).

	single table		WARMR		all	
	$L(\mathcal{D}, CT)\%$	$\lvert CT \setminus \mathcal{I}\rvert$	$L(\mathcal{D}, CT)\%$	$\lvert CT \setminus \mathcal{I}\rvert$	$L(\mathcal{D}, CT)\%$	$\lvert CT \setminus \mathcal{I}\rvert$
financial	91%	29	76%	130	76%	117
genes	87%	72	86%	191	83%	342
hepatitis	99%	5	98%	13	97%	26

We outline the obtained results on all databases in Table 6.2.

As before, we see that the number of candidate patterns grow very steeply for lower minimum support values. Also, we see that for more general pattern types we see a steeper candidate set growth (see Figure 6.5b). Note that we use a log scale to depict the number of patterns.

We have seen that achieving a smaller database encoding relies on the ability to draw patterns from an enriched pattern set. The inclusion of these enriched patterns indicates that essential characteristic structure within the database can be best described with these type of patterns: simpler patterns are apparently not sufficient. Although RDB-Krimp steers to small code tables, it allows these additional patterns in the code table only if they aid in the description of essential structure that would otherwise be described in a much less efficient manner (see Figure 6.5c).

Relational Code Tables

Compactness is not the only feat of interest. In order for our model to provide insight in the database it should be interpretable. As our code table contains a collection of characteristic patterns, we can pick single code table patterns to examine.

To show an example, we pick a code table pattern that is intuitive without expert knowledge. Shown in Figure 6.6 is a pattern that reads as follows: *'A gene localised in the nucleus having a transcription function that has two distinct physical interactions'*. As one would expect, a characteristic pattern for the GENES database is that transcription often involves physical interactions in order to copy information and that it occurs in the nucleus. Note that this pattern selects two distinct tuples from the `INTERACTION` table.

The small encoded database lengths are obtained largely due to the availability of the more complex patterns in our pattern language. Patterns that

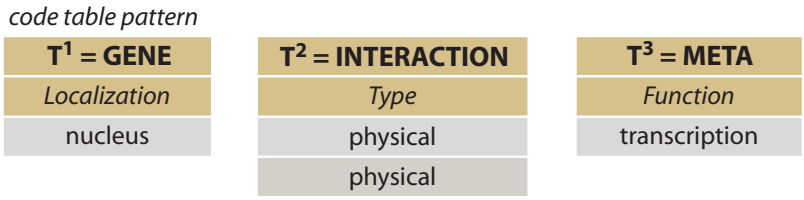


Figure 6.6: A partial description: a code table element from the description of the Genes database. Note that these patterns select multiple tuples from the same table.

select multiple tuples from one table are used more often in the database description, leaving simpler patterns to 'fill up' the database.

In the code tables obtained for the lowest used minsup, patterns that select multiple tuples from one table make up for 16%, 63%, and 71% of the content for the *financial*, *genes*, and *hepatitis* databases respectively.

6.6 Related Work

Our code tables are models for a complete database. Similarly, Probabilistic Relational Models (PRM) are graph based models that can be applied to model relational databases [27, 55]. A PRM is one graph, in which attribute values are linked with their co-occurrence probability. This contrasts to our code table, which is not a single graph, but a collection of local tree-like patterns. Moreover, an attribute value can occur multiple times in different local models, if this aids in describing the database better. This means that we can regard an attribute value in different contexts, instead of one.

In the field of Relational Data Mining (RDM), ILP based approaches, such as the WARMR algorithm, allow for the discovery of relational patterns [21,83]. As seen in our experiments, we obtain better database descriptions when we use our pattern language compared to WARMR-type patterns (see Figure 6.5).

The main application for a WARMR-like approach is when one is specifically interested in a target table, for example when trying to improve the classification scores on a target table given relational information [93]. This target table leads to an effective manner to prune the search space, and allows for aggregate functions to improve the efficiency [49]. An efficient implementation is FARMER [71], which in addition allows for a more general pattern language similar to the patterns used in this work.

In the work of both [24] and [65] the goal to find a different type of pattern: multi-valued dependencies (MVD) and functional dependencies (FD). These are 'global' patterns: a dependency is a 'higher order' pattern similar to a constraint on the relation, in contrast to code table patterns, which are local patterns.

R-Krimp

In Chapter 5, we introduced R-Krimp which finds patterns of the form: $[p_0, \ldots, p_n]$. These patterns are less expressive than the patterns used in RDB-Krimp, and thus have less descriptive potential. R-Krimp patterns are similar to the work of [28], who define these patterns as *simple conjunctive queries*.

In RDB-Krimp a single code table element can describe more structure in the database than with R-Krimp. As an example, consider a RDB-Krimp-style pattern from our illustrative database: *'An account with frequency 3 having both a disponent-type disposition and a owner-type disposition'* (see Figure 6.2). This single patterns translates requires two individual R-Krimp-style patterns: *'An account with frequency 3 having a disponent-type disposition'* and *'An account with frequency 3 having a owner-type disposition'*.

Note that R-Krimp in this case would cover `ACCOUNT` $\bowtie$ `ORDER` in an overlapping manner. As tuples can occur multiple times within a join, R-Krimp patterns allow for an overlapping cover. Thus, even in the initial case, when solely alphabet patterns are used, the covering of duplicate tuples will lead to a very different encoding.

6.7 Discussion

In our experiments, we see that compared to the candidate set the code table growth shows a much slower pace.Not only it stays small compared to the candidate set, it also stays compact compared to the original database. From the original set of frequent candidates only a few patterns contribute to the database description, leading up to 4 orders of magnitude reduction.

When we look at the influence of the initial database order, we see that this does not have much effect. We obtain only a slight deviation, up to around 1% from the original encoded length. Enforcing a single target table rather than treating all tables equal yields a much worse database description. The encoded database length shows an increase of up to 25% compared to the original result. In the *hepatitis* database all tables center around one single central table: the `PATIENT` table. While this database seems like a good case to pick as a target table, even here we obtain our best description when patterns are allowed to start at other tables.

Allowing a richer pattern language is a fruitful effort. We see that we generate more potentially interesting patterns in this manner, from which we can select those that describe the database best. Compared to WARMR-like patterns, we see that we can describe the database better: we achieve shorter encoded lengths. To obtain these good database descriptions, the code tables rely for the large part on these enriched patterns, which are of the type that selects multiple tuples from a table.

Our code tables stay compact and do not show exponential set growth as frequent pattern sets do for lower minimum supports. RDB-Krimp shows to be successfully to select compact models. Even under its MDL-selection pressure, larger enriched patterns are selected to describe the database, indicating that these patterns are very characteristic for the database.

6.8 Conclusion

Currently, in order to obtain insight from a relational database, either one aims to find a *'global'* model of the complete database, or one seeks a collection of *'local'* patterns in the database. While both approaches have their merits, they have their shortcomings. Global models tend to blur out interesting local structure for the sake of the global structure, and pattern collections tend to drown the global picture in a sea of patterns. In this chapter, we propose a method that combines the merits of both directions: it mines a compact but detailed description of the complete relational database.

For a database model to be descriptive it should be able to reflect the patterns that are present within it. Relational databases allow for tuples in one table to be connected to multiple tuples from other tables. Hence, if we want to find interesting patterns in a database, our pattern language should be rich enough to reflect such relations.

In this chapter, we introduce RDB-Krimp, an algorithm that describes the complete relational database using only a small collection of characteristic patterns: the code table. While the code table serves as a global model for the complete database, its patterns preserve the local details. Using the MDL principle, RDB-Krimp results in a compact set of patterns that together describe the database well. With respect to the original set of frequent candidates, we obtain up to 4 orders of magnitude reduction.

Our rich relational pattern language allows RDB-Krimp to draw from a larger pool of interesting frequent candidate patterns. The experiments reported on in this chapter verify our claims both on the usefulness of our pattern language and on the ability of RDB-Krimp to select just a few highly descriptive patterns. Firstly, each code table heavily relies on the introduced pattern language: up to 70% of the patterns select multiple tuples from one ta-

ble. Secondly, experiments show that our pattern language leads to a far better compression. In other words, these patterns highlight characteristic structure in the database. The fact that these patterns are not only characteristic but also easily interpretable is shown by an example from the Genes database (see Figure 6.6). As part of the code table we find a pattern that describes gene interactions: *'A gene localised in the nucleus having a transcription function that has two distinct physical interactions'*.

In contrast to current frequent pattern mining approaches, our approach does *not* rely on a target table. That is, our patterns can start from any table. In all cases this yields to improvements, of up to almost 25%. Even in a typical 'target table' style database, we see that we obtain a better description without the use of a single table as target.

Chapter 7

Relational Patterns are Better

Relational data mining is based on the premise that it is better to keep the relational structure of the database rather than joining all tables into one, humongous, single table. In Chapter 6 we have shown that relational patterns that model the structure of the data as faithfully as possible lead to a better compression than simpler classes of patterns. In the MDL philosophy, on which our work is based, this means that complex patterns capture the essential structure of the data better than simpler patterns.

In this chapter we give an independent verification of this claim. We show that our relational patterns yield better classifiers than simpler classes of patterns. Given our used set of pattern languages, we see that for every increase of the expressiveness of the language, we obtain an increase of the accuracy score. We also show that given more relational information, our MDL-based approach picks a few additional patterns to achieve this extra accuracy. Moreover, we show that joining all tables into a single table yields a classifier that is worse than our relational pattern based classifier. In other words, our class of relational patterns gives a better compression of the database and yields better classifiers. Thus verifying that relational patterns capture the data distribution better.

[1]This is an extended and edited version of an earlier paper submitted as: *Relational Patterns are Better.* A.C.M. Koopman, and A.P.J.M. Siebes. To: PKDD 2010

7.1 Introduction

Relational data mining seeks to generalise traditional, single table data analysis to the analysis of multiple inter-related tables. It is based on the premise that it is better to keep the relational structure of the database rather than, e.g., joining all tables into one, humongous, single table. The rationale for this premise is that in the construction of this single table, information is lost; irrespective of how the single table is constructed. Phrased differently: relational models are more expressive than their single table counterparts; a fact that is well-known from Logic [34].

In Chapter 6, we introduced a new pattern language for relational databases that aims to capture the structure in the relational database as faithfully as possible. This pattern language is strictly more expressive than earlier proposals, such as FARMER [71]. The most important difference is that no single "target table" is assumed. That is, patterns can fan out from any table in the database.

Moreover, in the same chapter we generalised our Krimp compression algorithm (see Chapter 3) to this new pattern language. The experiments showed that the resulting RDB-Krimp algorithm achieved better compression rates than versions that use simpler pattern languages; even if the database has a natural target table. Following the philosophy of MDL, on which our work is based, we concluded that our more complex pattern language captures the data distribution better than simpler languages.

MDL, however, warrants this conclusion only for the best model. Since RDB-Krimp is based on heuristics, we can not guarantee that it discovers the optimal model. Hence, there is a small possibility that the simpler languages can actually capture the data distribution better than our more complex language. To verify our claim that our more complex patterns are better, we present an independent test in this chapter.

Like Krimp, RDB-Krimp induces a natural classifier [62]. In the experiments presented in this chapter, the classifiers based on our more complex pattern language consistently perform best. Thus giving further evidence that our pattern language is better than simpler ones. This does not contradict Occam's razor: the pattern language should be as simple as possible, not simpler. While a more expressive pattern language may be more prone to overfitting, it is also able to capture the characteristics of the data better.

This may seem to contradict the conclusions of [9], which shows that a relatively simple pattern language performs on par with more complex patterns on graphs. However, there is no contradiction. That paper and this chapter illustrate the two sides of Occam's razor: a pattern language should neither be too complex nor too simple for the data at hand. One shouldn't be afraid of

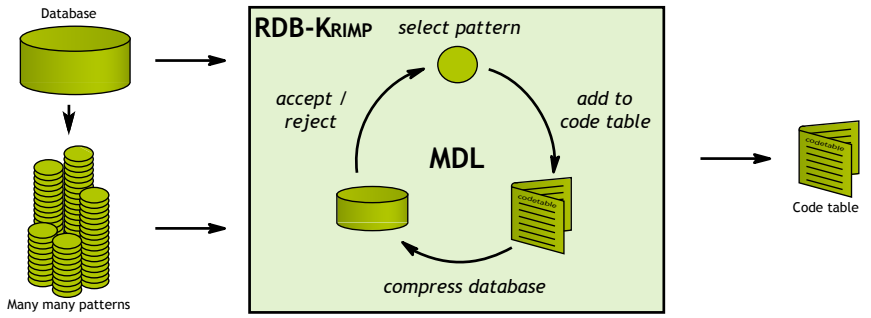


Figure 7.1: The relational database $\mathcal{D}$ is covered by patterns from the code table CT using RDB-Krimp.

simple patterns, but one shouldn't be afraid of complex patterns either.

Next to a comparison with other relational pattern languages, we also compare to our single-table approach. We compare the classification results of RDB-Krimp on the original database to the classification results of Krimp on the fully-joined single table. The fact that RDB-Krimp is also consistently better in these experiments gives further evidence for the premise of relational data mining: it is better to mine the original database. The added expressivity of both the relational model and our pattern language provide insight that would otherwise be lost.

7.2 RDB-Krimp

In order to find a good model for our relational database, we use the RDB-Krimp approach as presented in Chapter 6. We have illustrated the RDB-Krimp algorithm in Figure 7.1. Given a relational database $\mathcal{D}$ we derive an ordered set of frequent patterns. Each pattern is added to the code table CT and evaluated whether it contributes to the database description. If so, it gets accepted otherwise it is rejected.

Note that RDB-Krimp can use multiple pattern languages to model the database. These used patterns languages can be ordered on the basis of their complexity. That is, *single* is strictly less expressive than *WARMR*, which in turn is strictly less expressive than *all* (see Chapter 6 for details).

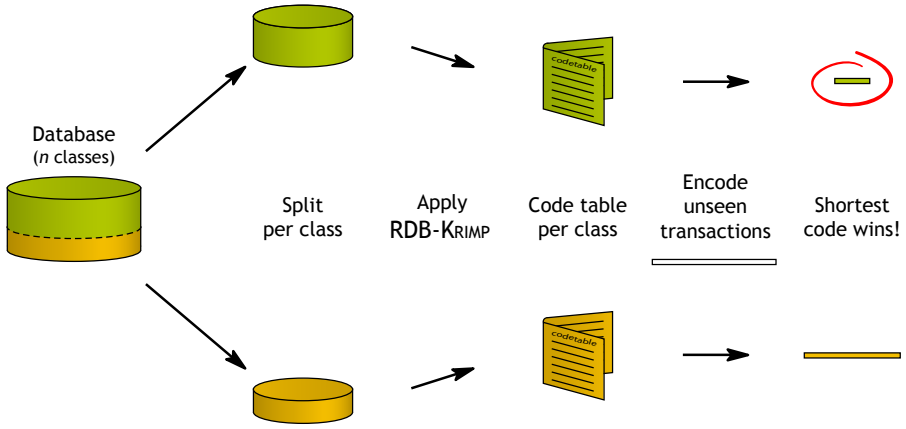


Figure 7.2: Given the CT_i derived by RDB-Krimp from the class databases $\mathcal{D}_i$, we classify unseen transactions to that class C_i that leads to the shortest encoded length.

7.3 Relational Classification

As noted in the Introduction of this chapter, the experiments in Chapter 6 show that our new class of relational patterns outperform simpler pattern languages. Moreover, RDB-Krimp yields small code tables. Relatively few non-alphabet patterns are needed to compress the database well. We can, however, not refer to the MDL-philosophy, on which our approach is based, and conclude that our pattern language captures the data distribution better than simpler pattern languages. For, RDB-Krimp is a heuristic algorithm with no guarantees that optimal models are discovered. Hence, it is necessary to test the quality of our pattern language and RDB-Krimp in an independent way.

So, what is a good independent test? In [62] we showed that in the single table case, Krimp induces a natural classifier. Simply split the table according to the class label, remove the class labels, and induce the code table for each class separately. Classification of a new tuple is simply the assignment of the class label associated with the code table that compresses this tuple best (see Figure 7.2).

The better a pattern language captures the data distribution, the better it will be for this type of classification algorithms. For, it is the difference between the distributions of the various classes that makes it work. The better you capture the distributions, the better you capture its differences.

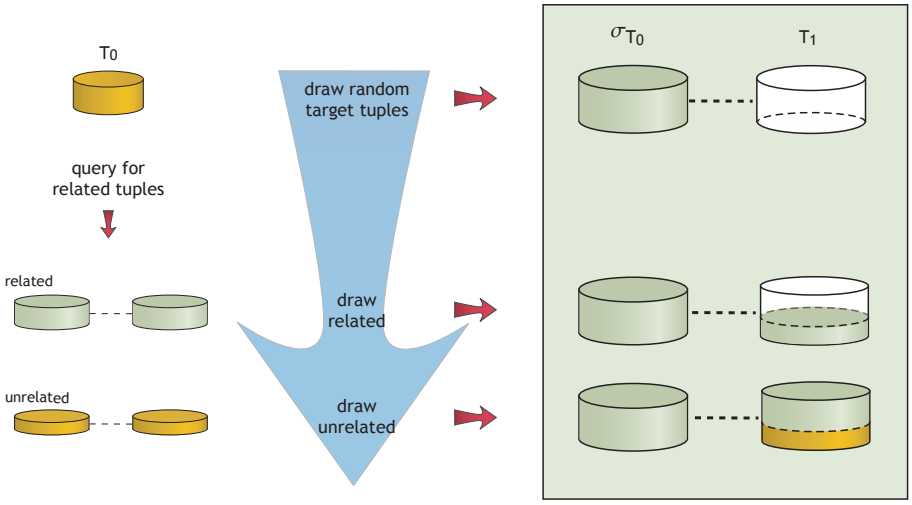


Figure 7.3: The class database $\mathcal{D}_i$ is derived by first drawing tuples from target T_0 given a class C_i. Then $\mathcal{D}_i$ is filled with drawn tuples for which there is a path to the already selected tuples in T^0. Then, if $\mathcal{D}_i$ is not yet filled, we complete it by drawing unrelated tuples from the original relational database.

In other words, classification is an independent test of the quality of our pattern language versus simpler pattern languages. Hence, our goal is to extend the classification scheme as outlined above to RDB-Krimp. This is less straight forward than one might assume. Splitting the database along class boundaries is more intricate in a relational setting.

Let $\mathcal{D}$ consist of the tables $\{T^0, T^1, \ldots, T^n\}$. Moreover, let T^0 be the table that has the class attribute C, that is T^0 is the target table. Splitting T^0 according to class labels is simple, we create a table T^0_i for each class C_i and remove the class labels. To split the other tables, we use the graphs induced by the tuples in T^0. A tuple $t_1 \in T^1$ that is connected to $t_0 \in T^0_i$ in the graph induced by t_0 is part of T^1_i.

To make this more precise, we introduce the notion of a path between two tuples as follows.

Definition 23 (Path)**.** *Let $\mathcal{D} = \{T^0, T^1, \ldots, T^n\}$, moreover, let $t_0 \in T^0$ and $t_i \in T^i$. There is a path from t_0 to t_i iff*

1. *The key K^0 of T^0 is a foreign key of T^i and $\pi_{K^0}(t_0) = \pi_{K^0}(t_i)$, or*

2. *The key* K^i *of* T^i *is a foreign key of* T^0 *and* $\pi_{K^i}(t_i) = \pi_{K^i}(t_0)$*, or*
3. *There exists a tuple* t_j *in table* T^j *such that there is a path from* t_0 *to* t_j *and a path from* t_j *to* t_i*.*

The predicate $path(t_0, t_i)$ *is true iff there exists a path from* t_0 *to* t_i*.*

Using these paths, it is easy to define the sub-tables of the T^j that "belong" to class C_i in T^0. Let T_i^0 be the sub-table of T^0 that corresponds to class C_i as above. Denote by T_i^j the sub-table of T^j that corresponds to class C_i:

$$t_j \in T^j \wedge \exists t_0 \in T_i^0 : path(t_0, t_j) \Rightarrow t_j \in T_i^j$$

Note that there may be multiple paths from T^0 to $t_j \in T^j$. In such a case, t_j occurs at most once in each sub-table T_i^j, but it can occur in multiple such sub-tables. That is, while $T_k^0 \cap T_i^0 = \emptyset$ for $i \neq k$, for $j \neq 0$, T_k^j may have a non-empty intersection with T_i^j.

Finally, there may be tuples $t \in T^j$ for which there is no path from any tuple $t_0 \in T^0$. One option is to assign such tuples to none of the sub-tables T_i^j of T^j; that is to discard them. However, that destroys information on the data distribution of T^j. From semi-supervised learning [17] we know that such additional density information is almost always useful. Hence, rather than discarding the non-connected tuples of T^j, we assign each of them to randomly to sub-tables T_i^j of T^j.

In this way, the database $\mathcal{D} = \{T^0, T^1, \ldots, T^n\}$ is divided in the class databases $\mathcal{D}_i = \{T_i^0, T_i^1, \ldots, T_i^n\}$, one for each class C_i in T^0. For the classifier we learn a code table CT_i for each $\mathcal{D}_i$, and again assign to the class whose code table gives the best compression.

There is one subtlety for this classification, we do not necessarily classify tuples from T^0 only. Rather we classify a graph of related tuples that have one node in T^0. We return to this issue in the experimental setup where we explain our train and test and cross-validation experiments.

7.4 Related Work

In the field of Relational Data Mining, ILP based approaches, such as WARMR, define a language that allows for the discovery of patterns within a relational database [21, 83]. These WARMR-type patterns fan out from a specific table, the target table, and can be typically applied for classification, where the classification of the tuples within the target table is improved by including relational information. A positive side-effect of this target table is that it induces an effective manner to prune the search space, and can be combined with aggregate functions to improve the efficiency [49].

A more complex pattern language is introduced by Nijssen et al. [71]. Their algorithm, FARMER, is computationally less expensive, and in addition can mine a more complex language of relational patterns. In this chapter, we utilise a pattern language that is strictly more expressive than both languages, as we do not require a target table, but allow patterns to fan out from every table in the database.

As the complexity of a pattern language directly relates to the computational complexity, it should not be needlessly complex. For this reason, Bringmann et al. compared several pattern languages in the structured pattern domain [9]. In their work, the most complex pattern language does not always render the best results. In some cases, simpler patterns such as multi-itemsets lead to the best accuracy scores. They conclude from this that a pattern language should not be more complex than the data actually requires.

In their work, the classifier is based on a set of k-best patterns that are chosen solely on pattern-based features. In our work, MDL takes the complexity of the model into account when selecting patterns. That is, a more complex pattern is only included in the code table iff it leads to a better description of the database. There is a trade-off between the complexity of the data, and the complexity of the model. The patterns in the model should not be too complex, but not too simple either. We include some, *not all*, of these more complex patterns in a description of the database.

The aim of this chapter is to demonstrate the usefulness of the pattern languages utilised by RDB-Krimp, and its resulting code tables. In order to demonstrate this feat we use a relational classifier. The rationale for this is that characteristic patterns that capture the underlying distribution well should be able to discriminate classes well, leading to a successful classification. Therefore, in the light of our used methodology, i.e. classification, it is relevant to discuss related relational classification algorithms.

A well-known classification method for relational databases is CrossMine [93]. The authors introduced a technique called tuple id propagation in order to remain efficient. CrossMine is shown to be scalable in the number of attributes and relations, which makes it suitable for complex relational schemas. In contrast to our approach, CrossMine is specifically designed for classification, and samples the database for tuples that can aid in a better classification score. This sampling technique is based on foil gain, and is shown to work well on databases consisting of data that is drawn from 2 classes. However, because of the sampling, their model does not necessarily describe the whole database well, which is our goal.

Another relational pattern based approach is that of Ceci et al. [14]. The authors based their classifiers on emerging patterns. Emerging patterns are patterns that have a significant difference (e.g. in support) between sets of

data. Based on the set of emerging patterns per class, they present two different classifiers: Mr-CAEP and Mr-PEPC. Using a relatively small set of patterns they obtain good classification scores. Again, in contrast to our approach, emerging patterns do not describe the complete data distribution.

A slightly different approach to build a classifier on relational data is to use a Probabilistic Relational Network (PRN) to model the underlying data. PRN's are shown to be used for classification [81]. Note that the use of a PRN provides less detailed insight than a pattern based approach, as it learns the probability that the attributes are related, instead of learning the actual co-occurrences of the attribute values.

7.5 Experiments

In this section, we discuss the relational folding in order to do the classification experiments. We performed the experiments on publicly available datasets mainly from previous conference discovery challenges. Using these datasets, we compare the results given several pattern languages in a relational setting, followed by the single table case. Finally, we show that our classifier reveals insight at the local level; we can see those patterns that make the difference between a wrong or right classification.

Experimental Set Up

To measure the accuracy of the RDB-Krimp classifier, we perform the usual 10-fold cross-validation. Creating a fold is the same procedure as creating the class database $\mathcal{D}_i$. The target table is split into a train and a test set as usual. These sub-tables can be seen as virtual classes and the rest of the database is divided as it is done for the real classes. This ensures, again, that related tuples in the various tables all end-up in the train database or in the test database. Whether one first computes the folds and then splits the database accoring to class or the other way around is immaterial. We first split on class and then compute the folds.

Note that this means that we test with a graph of tuples rather than with a single tuple in T^0. More precisely, for a t_0 in T^0 we compress the graph that consists of t_0 and all the maximal paths that fan out from t_0. This makes sense, since we want to measure the accuracy in the relational setting as opposed to the single table setting. The premise being that the non-target tables T^i add information to what is present in the target table T^0.

As mentioned in the Introduction of this chapter, we also compare to the Krimp classifier on the fully joined table. To make the fully joined tables for each fold for each class, we simply "join" the tuples along a maximal path that

fans out from a tuple t_0 in the target table T^0. If multiple maximal paths fan out from t_0 this yields multiple entries in the fully joined table. That is, t_0 "occurs" multiple times in the target table. To assign a class to t_0, we aggregate the KRIMP classifiers for each such occurrence of t_0 in the test database. In our setup, we have implemented this aggregate function as a majority vote. Note that since t_0 belongs to one class and gets assigned to either train or test in each fold, we do not test on data that has been seen in the training phase.

In the classification experiments we apply RDB-KRIMP without post acceptance pruning. This can result in code tables that contain elements with a *usage* of 0. Note that these elements with a *usage* of 0 form no problem in the database encoding. However, when classifying the unseen tuples in T_0 from the test database, all elements in CT can be used to encode the tuple. In order to acquire a valid code length, we require all elements to have a *usage* $\neq 0$. Therefore, we apply a Laplace correction to all elements in the trained code tables before classification. That is, we increase the *usage* of all elements in the code tables by a constant factor: 1.

Databases

We mostly used several relational databases that are publicly available. All of these databases consist of multiple tables, from which we have selected one specific attribute to be the target. The characteristics of these databases are shown in Table 7.1 and the schemas are shown in Figure 7.4. We have run experiments on a variety of databases with different characteristics. Note that although our method can be used on databases with many classes, the used databases all have 2 classes. In these databases, the numeric values have been categorised where applicable.

Some of these databases have been used in previous chapters: the Genes database [1], the Hepatitis database [2], and the Financial database [3]. The UCI Machine Learning repository [4] provided the Movies database that contains movie information together with the relating actors and awards. The Students database is derived from student results of the Computer Science department of the Universiteit Utrecht.

Relational Classification

Using this set of relational databases, we have used our classification algorithm to acquire accuracy scores based on the trained relational code tables. Table

[1] http://pages.cs.wisc.edu/ dpage/kddcup2001/
[2] http://lisp.vse.cz/challenge/
[3] http://lisp.vse.cz/challenge/
[4] http://archive.ics.uci.edu/ml/

Figure 7.4: The schemas of the used databases in the experiments. From top to bottom we have: Genes, Hepatitis, Financial, Movies and Students.

Table 7.1: The characteristics of the used relational databases. Each $\mathcal{D}$ has a number of items $|\mathcal{I}|$ over a number of tables $\#T$. $\#cl$ is the number of items that are class labels from the target T_0, A_0. The number of records of $\mathcal{D}$ is specified over the number of records in the tables $|T|$.

$\mathcal{D}$	genes	financial	hepatitis	students	movies
$\|\mathcal{I}\|$	847	182	1342	58	1949
$\#T$	5	6	5	2	4
T_0	gene	loan	biopsy	course	actor
A_0	localization	status	type	subject	gender
$\#cl$	2	2	2	2	2
$\|\mathcal{D}\|$	10628	4567	75616	32086	55941
$\|T\|$	4151	1513	771	840	89
	4151	682	31039	31246	48491
	554	827	198		48
	862	827	42914		6863
	910	36	694		
		682			

7.2 provides an overview of the obtained accuracy scores. We see that for all databases the accuracy scores improves with the use of our relational model. In most cases, far better, only for the Financial database the improvement is marginal. This is however, a hard and very skewed database to mine. For reference, CrossMine [93] attains an accuracy of 87.5% - 89.9% depending on the usage of tuple sampling or not. Note that CrossMine is designed and optimised specifically for classification in contrast to our approach. Given these accuracy scores we can conclude that we achieve similar scores as CrossMine for the Financial database. However note that, we do not do an in depth comparison with CrossMine, as our goal is not to build a classifier, but to determine how well the pattern languages capture the data distribution.

In order to test the quality of the pattern languages *single*, *WARMR*, and *all* patterns, we have run our classification algorithm for these languages. In all cases, the best reported accuracy score is attained when using all patterns to describe the database. Depicted in Figure 7.5 are the results for the Genes dataset. We see that the best scores for a given *minsup* are obtained for a more complex pattern language. In general we see that more complex pattern languages more quickly converge to higher accuracy levels.

Moreover, we achieve the highest accuracy scores when we incorporate more tables in our relational database. In Figure 7.6 we have depicted the results

Table 7.2: Here we show the best obtained accuracy scores (*acc.*) using RDB-Krimp on *all* patterns in comparison to the baseline score *base*. Also reported is the *minsup* level at which the best accuracy is obtained.

$\mathcal{D}$	genes	financial	hepatitis	students	movies
acc.	83.6%	89.0%	85.9%	80.8%	62.8%
base	66.0%	88.8%	70.4%	46.9%	61.6%
minsup	30	60	8	400	13

for different subsets of the Financial database. We see that the pattern selection RDB-Krimp picks patterns from all tables to improve the classification accuracy. We also see the trend that for higher *minsup* levels we attain better accuracy scores when more tables are incorporated.

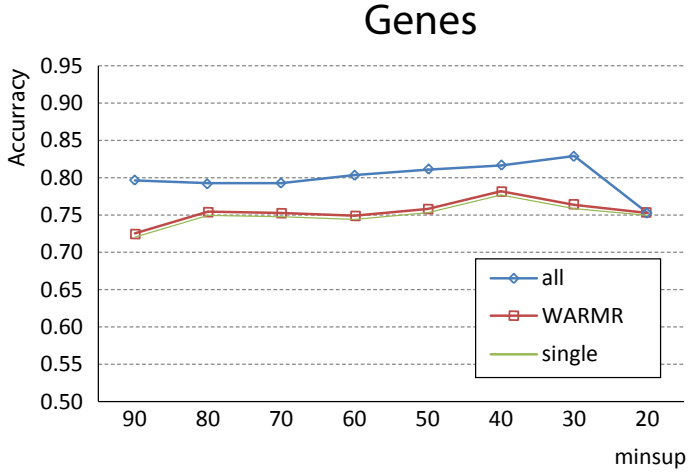


Figure 7.5: We consistently achieve better accuracy scores for more complex pattern languages. Shown here is the obtained accuracy scores for the Genes database. Note that the best accuracy scores occur at different *minsup* levels.

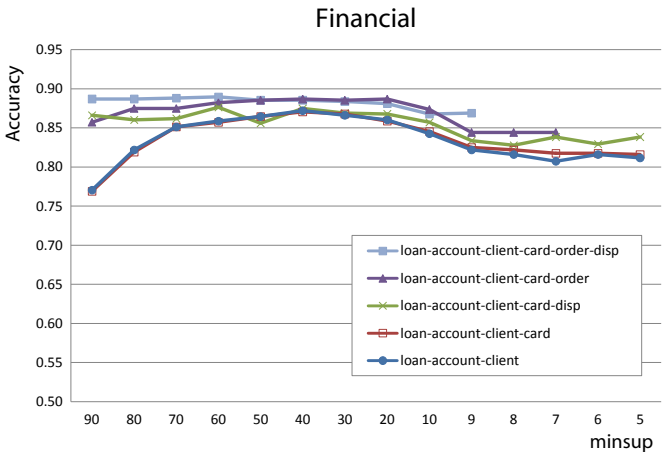


Figure 7.6: Shown here are the obtained accuracy scores for different subsets of the Financial database. We see that when we include more relational information into the database to be modelled, that we obtain better the accuracy scores.

Excluding Relational Information

In the previous section, we have shown that we can incorporate relational information to classify unseen tuples from the database. However, one can wonder: does all this relational information really help us? It could be very well be possible that the target table T^0 by itself already can provide enough information to lead to an equally well performance in terms of classification.

To asses the influence of relational information stored in the other tables, we run the single table Krimp classifier on the target table and compared the accuracy scores with RDB-Krimp classifier. The results are shown in Table 7.3. For both the Krimp and RDB-Krimp cases, we have listed the maximum obtained accuracy and the *minsup* at which it is attained. We see that for all databases, we can obtain a better classification score when relational information is incorporated. Moreover, in most cases, we observe that we obtain a better accuracy score at a higher *minsup* levels. This makes sense, as at higher *minsup* levels, the relational pattern set grows more quickly, and therefore can lead to code tables that capture the underlying distribution better.

Table 7.3: Excluding the relational information has its influence on the obtained accuracy scores. We compare two cases; accuracy results derived from the single target table T^0, and results derived from the complete $\mathcal{D}$. Shown are the best accuracy scores with their respective *minsup*. In all cases we see that including relational information leads to a better accuracy.

	T^0		$\mathcal{D}$	
$\mathcal{D}$	acc.	*minsup*	acc.	*minsup*
financial	81.1 %	20	88.9 %	60
genes	82.1 %	21	82.9 %	30
hepatitis	60.3 %	28	87.2 %	8
movies	62.0 %	53	62.2 %	13
students	67.9 %	400	80.8 %	400

Why Relational?

In order to determine how well our relational pattern language aids in characterising the data better, we evaluate the effect of pruning the structural information (as described in Section 7.5). That is, we compare the results obtained by our relational classifier, with the single table Krimp induced classifier on joined tuples. In earlier work, it is shown that this classifier performs on par with other single table classifiers [85].

We see that in all reported cases, the relational classifier outperforms the single table classifier (see Table 7.4). Note that in both cases, the classifier can base its classification on the same attribute data per tuple. We see that including structural information aids in classifying the target tuples better.

Also note that when mining directly on the relational database we do not need to compute a single joined table. In two cases, namely the Movies and Students database, the size joined versions of the relational databases precluded for the Krimp approach. This illustrates that joining a relational database can lead to severe problems when mining for patterns.

Characterising the Difference

In Subsection 7.5, we have seen that we obtain good accuracy scores with our RDB-Krimp classifier. Moreover, in all cases, we have seen that the best accuracy scores are obtained in combination with the most complex pattern language. One of the fruits of using a pattern-based classifier, is that it is fairly intuitive to analyse how the classification has come about. That is, for each tuple, we can see which patterns have been used to describe it. If for a set of

Table 7.4: Comparing the classifiers on the basis of their accuracy scores. We compare our RDB-Krimp-based relational classifier to the Krimp classifier that uses the joined table as described in Section 7.5.

$\mathcal{D}$	*minsup*	RDB-Krimp acc.	Krimp acc.
financial	8.7 %	88.9 %	86.8 %
genes	3.4 %	82.9 %	81.6 %
hepatitis	1.1 %	87.2 %	80.7 %

pattern languages the classification of one tuple is different, we can see which patterns are responsible.

As an example, we pick a tuple together with its tuple-induced graph from the target table of the Financial database. Shown in Figure 7.7 is the tuple-induced graph drawn from the tables LOAN and ORDER, together with a visualisation of its encoded length (see each subfigure). The width of each bar is proportional to the size of its code length. The encodings are visualised for each of the pattern languages: *single*, *WARMR*, and *all*. The more complex pattern languages lead to a shortened encoded length, and thus are represented by a shorter bar (see the middle of each subfigure). In the figure, we have coloured that part of the tuples that is covered by non-alphabet patterns, and have left the part of the tuple covered by alphabet elements white. The obtained classification label has been pointed out for each of the used pattern languages.

In Figure 7.7 we clearly see how the different languages lead to different covers of the same set of tuples. In the case where we used the *single* pattern language, the target tuple is covered solely by alphabet elements. When using WARMR-style patterns, the patterns are limited to cover one tuple per table. In this setting, both tuples from the ORDER table are completely covered by distinct relational patterns. In the *all patterns* case, we see that a more complex cover is favoured by compression. The tuples are covered by a code table pattern that groups both ORDER tuples.

The LOAN tuples within the Financial database fall into two classes; either good loans *"No Problems or OK so far"*, or bad loans *"Loan not paid or in debt"*. The tuple from the LOAN table shown in Figure 7.7 is a good loan. Both the single table and WARMR cases lead to a misclassification of the LOAN tuple. The encoded length of is either indistinguishable between classes, or is smaller for the incorrect class. However, when *all patterns* are used, the encoded length is the smallest for the correct class: *"No Problems or OK so far"*.

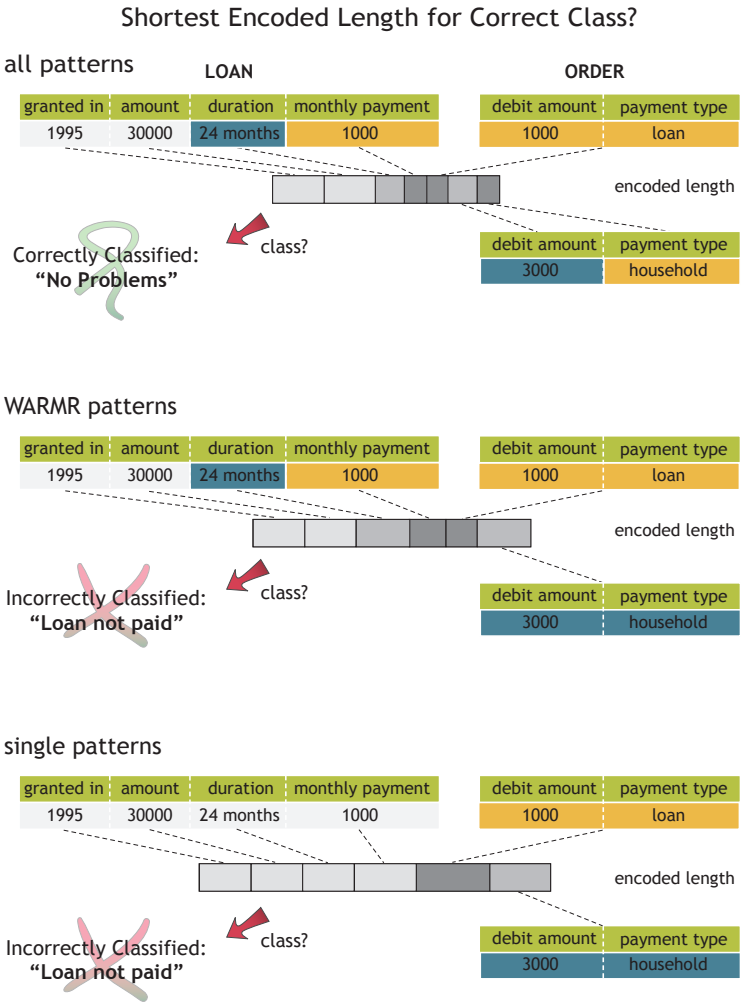


Figure 7.7: A target tuple t from LOAN together with its tuple-induced graph, from ORDER, are covered using either *all*, *WARMR*, or single table patterns. Shown are the shortest encoded versions of these transactions; the middle grey bar illustrates the encoded length. The target tuple is assigned to that class that leads to the shortest encoded length. This only goes well for the case of *all patterns*, as the other languages incorrectly classify it as *"Loan not paid"*.

7.6 Conclusions

In this chapter we investigated whether a rich pattern language can grasp the underlying data distribution of a relational database better than a simpler language. The results we presented confirm this hypothesis: not only does our relational pattern language allow for a better compression ratio, it also induces better classifiers. Hence, it is more capable of capturing the distribution of the data.

In Chapter 6, we introduced the RDB-Krimp algorithm to find compact models that describe complete relational databases. These models, called code tables, are characteristic sets of patterns. Using our defined pattern language, we obtained better compression ratios than with less expressive pattern languages, i.e. WARMR and FARMER. This indicates that our rich language allows for more accurate database descriptions. However, we needed an independent way to verify this.

The experiments presented in this chapter show that the richer a pattern language is, the better the induced models perform in terms of classification. Additionally, as our RDB-Krimp classifier is pattern based, it is possible to see which patterns make the difference. In our analysis we show that rich relational patterns are crucial for successful classification, and thus for capturing the distribution of the data more accurately.

The findings in this chapter may seem to contradict the conclusions of Bringmann et al. [9], which shows that a relatively simple pattern language performs on par with more complex patterns on graphs. However, there is no contradiction. We only use those complex patterns if they improve compression ratio of the database, while they select the complex patterns on an individual basis. That is, [9] and this chapter illustrate the two sides of Occam's razor: a pattern language should neither be too complex nor too simple for the data at hand. One shouldn't be afraid of simple patterns, but one shouldn't be afraid of complex patterns either.

Chapter 8

Conclusions

Patterns are nearly everywhere! That is how we started our introductory chapter on pattern mining. In our consecutive chapters, we have shown that finding patterns is indeed not hard. The aim of pattern mining is to find those patterns that provide new insight into the data. However, in order to derive these patterns, we need to loosen the constraints in the mining process. Regrettably, these loose constraints lead to a flood of 'interesting' patterns: most of them are slight variations of the same theme. This phenomenon, aptly named the *frequent pattern set explosion*, prevents pattern mining to provide the user with the insight he is looking for.

In the end, a user is typically not interested into finding *all* frequently occurring patterns in a relational database. On the contrary, a user preferably finds only a few high-quality patterns that require examination. This has therefore been set as the research goal of this thesis:

Derive a compact set of high-quality, non-redundant, and characteristic relational patterns that summarise the complete database well.

In order to find these high-quality patterns, we shift our focus from interesting individual patterns to interesting pattern sets. Our selection method focusses on finding that set of non-redundant patterns that forms a good summary of the database. Furthermore, as we focus on finding high-quality summaries of a relational database, we need our patterns themselves to be high-quality as well. Since a relational database may contain many complex structures, our database summary needs to be able to match this complexity. We therefore designed a pattern language to match this complexity as faithfully as possible.

In this thesis, we thus focussed on selecting a compact set of interesting

patterns that models the database well. In order to derive this set, we used the Minimum Description Length (MDL) principle that, similar to Occam's razor, prefers the simplest model of the data that results in the best description. That is, it selects the simplest model that compresses the database best.

This algorithm, called Krimp, selects the set of patterns that compress the database best. As this problem is extremely hard for the optimal set, we presented a practical algorithm that utilises heuristics. The outline of the algorithm can be sketched as follows:

The starting point of Krimp, its set of candidates, is formed by the set of frequent patterns. As part of its heuristics we order this set of candidates. Given this ordered candidate set, we pick out patterns one by one and add them to our model. Our model, the code table, is an ordered set of patterns that is used to encode the complete database. Typically, only a few candidates get chosen to be part of the code table. When an added pattern contributes to the description of the database, we keep it in the code table. Otherwise it is permanently discarded. If a pattern is not discarded, we can apply an additional pruning strategy on the code table to check if we can optimise it. We check if all the patterns in the code table still contribute to the description, or otherwise remove them permanently from the code table.

Using Krimp, we obtain a compact set of characteristic patterns that describe, and encode, the database well. However, in order to apply this approach to various relational scenarios, we need to apply different encoding strategies.

Simply put, relational structure makes things more complex. This especially becomes clear in the encoding of a database that contains structure. In order to assure the quality of our code table, we need to steer the search process towards one clear optimum. In our case, this is the best lossless compression of the relational database. Designing a suitable lossless encoding is more difficult in a relational setting, as can be seen in the various design choices made in this thesis. We introduced a toolbox of algorithms with suitable lossless encodings that can find these high-quality models in multiple scenarios.

- Our first contribution is in the related field of structured data mining. Although the used structured data types are typically less expressive than those used in a relational database, they can express relations between nodes. We showed that in order to encode the database in a lossless fashion with the use of MDL, we need to restrict the used patterns. With the use of this adjusted version of Krimp, we show that we can find compact models that describe the database well.

- As our goal is to find models for relational databases, we continued with finding pattern sets that summarise the database, or more in particular its join. In order to do so, we utilise a relational pattern type called

relational item sets. We show that we can derive compact models that describe the complete join very well. As KRIMP is inherently an intensive algorithm, we also address the improvement of its efficiency. We show that we can select a much smaller set of candidates by using KRIMP's ability to grasp the underlying distribution. It not only approximates the original description very well, it results in an increase of the efficiency, in both in space and time.

- As relational databases can harbour complex patterns, there is a need to model it as faithfully as possible. To this end, we present an algorithm RDB-KRIMP that uses a new lossless encoding scheme for a relational database. Using RDB-KRIMP, we can use a far more complex relational pattern language to describe the database. This allows the usage of a new pattern language that proves to be fruitful. The more complex our pattern language is, the more succint the description of the database becomes.
- The quality of our models are based on the Minimum Description Length principle. That is, we have selected sets of patterns that compress well. While this leads to well describing summaries of the database, one can argue whether it is a good quality measure. Therefore, we confirm the quality of the models derived by RDB-KRIMP by means of classification. We show that the RDB-KRIMP models perform well in terms of classification. That is, RDB-KRIMP picks out characteristic patterns that can discriminate well between classes. Moreover, the models based on more complex pattern languages perform better.

This is hardly the end point. From the single table case we know that KRIMP also performs well at other data mining tasks such as: imputation, finding interesting subgroups, and generating very similar but privacy-preserved databases. Since the approaches presented in this thesis utilise both MDL and similar heuristics, it is most likely that similar fruitful efforts can be attained in the relational case.

In this thesis, we addressed the Achilles' heel of pattern mining. That is, we showed that we attain high reductions of the number of patterns, up to several orders of magnitude. However, we still need to mine this set of all frequent patterns before we can pick the interesting ones. Therefore, we could greatly improve the efficiency of our approach if we could incorporate our selection criteria into the mining process itself.

While our model contains a rich relational pattern language, some natural extensions in this area still remain open problems. For one, as relational databases often contain numeric data, it would be interesting to see a pattern based lossless MDL encoding that utilises numeric data in a suitable manner.

As for our banker from the Introduction of this thesis. Did the results in this thesis help him? Since he is purely illustrative we cannot ask him. However, some of our results would probably provide him with a bit more understanding into the daily operation of his multinational bank.

We showed that we can find characteristic patterns that show what type of loans often relate to what type of financial transactions (Chapter 5). In order to provide him with some oversight, we showed that we can find a single but detailed model of the complete database of his bank (Chapter 6). Finally, we showed that these models are so good, that they even can be used to distinguish good from bad loans (Chapter 7).

In all, although mining for useful insight remains an art, we provided our banker with a slightly larger toolbox to use.

Bibliography

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized Substructure Discovery for Semi-structured Data. In *Proceedings of the PKDD '02*, pages 1–14. Springer-Verlag, 2002.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM SIGMOD '93*, pages 207–216. ACM Press, 1993.

[3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the VLDB'94*, pages 487–499. Morgan Kaufmann, 1994.

[4] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. In *Proceedings of the SDM '02*, pages 158–174. SIAM, 2002.

[5] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering Frequent Substructures in Large Unordered Trees. In *Proceedings of the DS '03*, pages 47–61. Springer-Verlag, 2003.

[6] R. Bathoorn, A. Koopman, and A. Siebes. Reducing the Frequent Pattern Set. In *Proceedings of the ICDM'06 Workshops*, pages 55–59. IEEE Computer Society, 2006.

[7] C. Borgelt and M. R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proceedings of the ICDM '02*, pages 51–58. IEEE Computer Society, 2002.

[8] B. Bringmann and A. Zimmermann. The Chosen Few: On Identifying Valuable Patterns. In *Proceedings of the ICDM '07*, pages 63–72. IEEE Computer Society, 2007.

[9] B. Bringmann, A. Zimmermann, L. De Raedt, and S. Nijssen. Don't Be Afraid of Simpler Patterns. In *Proceedings of the PKDD '06*, pages 55–66. Springer-Verlag, 2006.

[10] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a Maximal Frequent Itemset Algorithm for Transactional Databases. In *Proceedings of the ICDE '01*, pages 443–452. IEEE Computer Society, 2001.

[11] T. Calders and B. Goethals. Mining All Non-Derivable Frequent Itemsets. In *Proceedings of the PKDD '02*, pages 74–85. Springer-Verlag, 2002.

[12] T. Calders and B. Goethals. Depth-first Non-Derivable Itemset Mining. In *Proceedings of the SDM '05*, pages 250–261. SIAM, 2005.

[13] T. Calders and B. Goethals. Non-Derivable Itemset Mining. *J. of Data Mining and Knowledge Discovery*, 14(1):171–206, 2007.

[14] M. Ceci, A. Appice, and D. Malerba. Emerging Pattern Based Classification in Relational Data Mining. In *Proceedings of DEXA '08*, pages 283–296. Springer-Verlag, 2008.

[15] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully Automatic Cross-Associations. In *Proceedings of the KDD '04*, pages 79–88. ACM, 2004.

[16] V. Chandola and V. Kumar. Summarization - Compressing Data into an Informative Representation. *J. on Knowledge and Information Systems*, 12(3):355–378, 2007.

[17] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-supervised Learning*. MIT Press, Cambridge, MA, 2006.

[18] Y. Chi, S. Nijssen, R. Muntz, and J. Kok. Frequent Subtree Mining - an Overview. *Fundamenta Informaticae*, 66:161–198, 2004.

[19] E. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[20] T. Cover and J. Thomas. *Elements of Information Theory, 2nd ed.* John Wiley and Sons, 2006.

[21] L. Dehaspe and H. Toivonen. Discovery of Frequent DATALOG Patterns. *J. of Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[22] L. Dehaspe and H. Toivonen. Discovery of Relational Association Rules. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, 2001.

[23] C. Faloutsos and V. Megalooikonomou. On Data Mining, Compression, and Kolmogorov Complexity. *J. of Data Mining and Knowledge Discovery*, 15(1):3–20, 2007.

[24] P. A. Flach and I. Savnik. Database Dependency Discovery: a Machine Learning Approach. *AI Communications*, 12(3):139–160, 1999.

[25] M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. In *Proceedings of the VLDB '99*, pages 223–234. Morgan Kaufmann, 1999.

[26] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling Databases. In *Proceedings of DS '04*, pages 278–289. Springer, 2004.

[27] L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic Relational Models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

[28] B. Goethals, W. Le Page, and H. Mannila. Mining Association Rules of Simple Conjunctive Queries. In *In Proceedings of the SDM '08*, pages 96–107. SIAM, 2008.

[29] B. Goethals, W. L. Page, and M. Mampaey. Mining Interesting Sets and Rules in Relational Databases. In *Proceedings of the SAC '10*. ACM, 2010.

[30] B. Goethals and M. J. Zaki, editors. *FIMI '03: Frequent Itemset Mining Implementations*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[31] G. Grahne and J. Zhu. Fast Algorithms for Frequent Itemset Mining using FP-Trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(10):1347–1362, 2005.

[32] P. Grünwald, I. Myung, and M. Pitt. *A Tutorial Introduction to the Minimum Description Length Principle*. MIT Press, 2005.

[33] P. D. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[34] A. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1988.

[35] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent Pattern Mining: Current Status and Future Directions. *J. of Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[36] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2 edition, January 2006.

[37] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2000. ACM.

[38] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining Top-k Frequent Closed Patterns without Minimum Support. *Data Mining, IEEE International Conference on*, 0:211, 2002.

[39] E. O. Heierman, G. M. Youngblood, and D. J. Cook. Mining Temporal Sequences to Discover Interesting Patterns. In *In Proceedings of KDD Workshop '04*. ACM, 2004.

[40] H. Heikinheimo, J. K. Seppänen, E. Hinkkanen, H. Mannila, and T. Mielikäinen. Finding Low-Entropy Sets and Trees from Binary Data. In *Proceedings of the KDD '07*, pages 350–359. ACM, 2007.

[41] H. Heikinheimo, J. Vreeken, A. Siebes, and H. Mannila. Low-Entropy Set Selection. In *Proceedings of the SDM '09*, pages 569–580. SIAM, 2009.

[42] L. B. Holder, D. J. Cook, and S. Djoko. Substucture Discovery in the SUBDUE System. In *Proceedings of the KDD Workshop '94*, pages 169–180. AAAI Press, 1994.

[43] T. Horváth, J. Ramon, and S. Wrobel. Frequent Subgraph Mining in Outerplanar Graphs. In *Proceedings of the KDD '06*, pages 197–206. ACM, 2006.

[44] D. Jensen, J. Neville, and M. Hay. Avoiding Bias when Aggregating Relational Data with Degree Disparity. In *Proceedings of the ICML '03*, pages 274–281. AAAI Press., 2003.

[45] E. Keogh, S. Lonardi, C. Ratanamahatana, L. Wei, S.-H. Lee, and J. Handley. Compression-Based Data Mining of Sequential Data. *J. of Data Mining and Knowledge Discovery*, 14(1):99–129, 2007.

[46] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards Parameter-Free Data Mining. In *Proceedings of the KDD '04*, pages 206–215. ACM, 2004.

[47] J. De Knijf. FAT-miner: Mining Frequent Attribute Trees. In *Proceedings of the SAC '07*, pages 417–422. ACM, 2007.

[48] J. De Knijf and A. Feelders. Monotone Constraints in Frequent Tree Mining. In *Proceedings of the BENELEARN '05*, pages 13–20, 2005.

[49] A. Knobbe. *Multi-Relational Data Mining*. PhD thesis, Universiteit Utrecht, Utrecht, the Netherlands, 2004.

[50] A. J. Knobbe, M. de Haas, and A. Siebes. Propositionalisation and Aggregates. In *Proceedings of the PKDD '01*, pages 277–288. Springer, 2001.

[51] A. J. Knobbe and E. K. Y. Ho. Maximally Informative k-Itemsets and their Efficient Discovery. In *Proceedings of the KDD '06*, pages 237–244. ACM, 2006.

[52] A. J. Knobbe and E. K. Y. Ho. Pattern Teams. In *Proceedings of the PKDD '06*, pages 577–584. Springer, 2006.

[53] A. J. Knobbe, A. Siebes, and B. Marseille. Involving Aggregate Functions in Multi-Relational Search. In *Proceedings of the PKDD '02*, pages 287–298. Springer, 2002.

[54] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[55] D. Koller and A. Pfeffer. Probabilistic Frame-Based Systems. In *Proceedings of the IAAI '98*, pages 580–587. AAAI, 1998.

[56] A. Koopman and A. Siebes. Discovering Relational Items Sets Efficiently. In *Proceedings of the SDM '08*, pages 108–119. SIAM, 2008.

[57] A. Koopman and A. Siebes. Characteristic Relational Patterns. In *Proceedings of the KDD '09*, pages 437–446. ACM, 2009.

[58] S. Kramer, L. De Raedt, and C. Helma. Molecular Feature Mining in HIV Data. In *Proceedings of the KDD '01*, pages 136–143. ACM, 2001.

[59] M.-A. Krogel and S. Wrobel. Feature Selection for Propositionalization. In *Proceedings of the DS '02*, pages 430–434. Springer, 2002.

[60] T. Kudo. http://chasen.org/ taku/software/freqt/.

[61] M. van Leeuwen. *Patterns that Matter*. PhD thesis, Universiteit Utrecht, Utrecht, the Netherlands, 2010.

[62] M. van Leeuwen, J. Vreeken, and A. Siebes. Compression Picks Item Sets That Matter. In *Proceedings of the PKDD '06*, pages 585–592. Springer, 2006.

[63] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, 1993.

[64] G. Liu, H. Lu, J. X. Yu, W. Wang, and X. Xiao. AFOPT: An Efficient Implementation of Pattern Growth Approach. In *Proceedings of the FIMI '03*. CEUR-WS, 2003.

[65] H. Mannila and K.-J. Räihä. Algorithms for Inferring Functional Dependencies from Relations. *J. of Data and Knowledge Engineering*, 12(1):83–99, 1994.

[66] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *J. of Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[67] H. Mannila, H. Toivonen, and I. Verkamo. Efficient Algorithms for Discovering Association Rules. In *Proceedings of the KDD Workshop '94*, pages 181–192. AAAI Press, 1994.

[68] T. Mielikäinen and H. Mannila. The Pattern Ordering Problem. In *Proceedings of the KDD '03*, pages 327–338. Springer-Verlag, 2003.

[69] S. Nijssen and J. N. Kok. Faster Association Rules for Multiple Relations. In *Proceedings of the IJCAI '01*, pages 891–896. Morgan Kaufmann, 2001.

[70] S. Nijssen and J. N. Kok. Efficient Discovery of Frequent Unordered Trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[71] S. Nijssen and J. N. Kok. Efficient Frequent Query Discovery in FARMER. In *In Proceedings of the PKDD '03*, pages 350–362. Springer, 2003.

[72] S. Nijssen and J. N. Kok. The Gaston tool for Frequent Subgraph Mining. In *Proceedings of the Grabats '04*, pages 77–87. Elsevier, 2004.

[73] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. 1540:398–416, 1999.

[74] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In *Proceedings of the ICDE '01*, pages 215–224. IEEE Computer Society, 2001.

[75] R. Ramakrishnan and G. J. *Database Management Systems*. McGraw-Hill, 1998.

[76] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[77] J. Roberto J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proceedings of the SIGMOD '98*, pages 85–93. ACM Press, 1998.

[78] U. Rückert and S. Kramer. Frequent Free Tree Discovery in Graph Data. In *Proceedings of the SAC '04*, pages 564–570. ACM, 2004.

[79] A. Siebes, J. Vreeken, and M. van Leeuwen. Item Sets that Compress. In *Proceedings of the SDM '06*, pages 393–404. SIAM, 2006.

[80] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. GraphScope: Parameter-Free Mining of Large Time-Evolving Graphs. In *Proceedings of the KDD '07*, pages 687–696. ACM, 2007.

[81] B. Taskar, E. Segal, and D. Koller. Probabilistic Classification and Clustering in Relational Data. In *In Proceedings of the IJCAI '01*, pages 870–878. Morgan Kaufmann, 2001.

[82] N. Tatti and J. Vreeken. Finding Good Itemsets by Packing Data. In *Proceedings of the ICDM '08*, pages 588–597. IEEE Computer Society, 2008.

[83] M. S. Tsechansky, N. Pliskin, G. Rabinowitz, and A. Porath. Mining Relational Patterns from Multiple Relational Tables. *J. of Decision Support Systems*, 27(1-2):177–195, 1999.

[84] J. Vreeken. *Making Pattern Mining Useful.* PhD thesis, Universiteit Utrecht, Utrecht, the Netherlands, 2009.

[85] J. Vreeken, M. van Leeuwen, and A. Siebes. Characterising the Difference. In *Proceedings of the KDD '07*, pages 765–774. ACM, 2007.

[86] J. Vreeken, M. van Leeuwen, and A. Siebes. Krimp : Mining Itemsets that Compress. *Data Mining and Knowledge Discovery*, 2010.

[87] C. Wallace. *Statistical and Inductive Inference by Minimum Message Length*, volume XVI of *Information Science and Statistics*. Springer-Verlag New York, 2005.

[88] J. Wang and J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the ICDE '04*, pages 79–90. IEEE Computer Society, 2004.

[89] J. Wang and G. Karypis. On Efficiently Summarizing Categorical Databases. *J. Knowledge and Information Systems*, 9(1):19–37, 2006.

[90] D. Weininger. SMILES, a chemical language and information system. *J. of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[91] Y. Xiang, R. Jin, D. Fuhry, and F. F. Dragan. Succinct Summarization of Transactional Databases: an Overlapped Hyperrectangle Scheme. In *Proceeding of the KDD '08*, pages 758–766. ACM, 2008.

[92] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the ICDM '02*, pages 721–724. IEEE Computer Society, 2002.

[93] X. Yin, J. Han, J. Yang, and P. S. Yu. CrossMine: Efficient Classification Across Multiple Database Relations. In *Proceedings of the ICDE '04*, pages 399–411. IEEE Computer Society, 2004.

[94] M. Zaki. http://www.cs.rpi.edu/ zaki/software/.

[95] M. Zaki. Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.

[96] M. J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.

[97] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[98] M. J. Zaki and C.-J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *Proceedings of the SDM '02*, pages 457–473. SIAM, 2002.

[99] Q. Zhao and S. S. Bhowmick. Sequential Pattern Matching: A Survey. Technical Report 2003118, CAIS, Nanyang Technological University, 2003.

Index

Abstract

Nowadays, relational databases have become the de facto standard to store large quantities of data. As the manual analysis of these large quantities of data is practically impossible, the field of data mining provides methods that attempts to automatically acquire insight into the data. One cornerstone technique is that of pattern mining: finding interesting regularity in data.

Despite all good efforts, one can conclude that pattern mining still has a major Achilles' heel, that is, the ease at which patterns can be found. Many found patterns are slight variations on the same underlying theme, although many of them are still designated as interesting. In practice, a user gets swamped by too many similar patterns that do not contribute to a new insight into the database.

In this thesis we therefore propose a different approach. In contrast to selecting patterns on an individual basis, we propose the selection of pattern sets. In particular, we focus on a selection scheme based on a compression technique called the Minimum Description Length (MDL) principle. The selected pattern set, our model of the data, is used to compress the complete database. According to the MDL principle, the model that compresses the database best is also the one that describes it best.

As acquiring the optimal model of a database is simply too complex, we utilise a practical and heuristic approach, named Krimp. Based on this, we designed a toolbox of algorithms that derives models for different interpretations of the data. We discuss structured data types such as sequences and trees, the join of the database, and relational databases as a whole. These last models also show to result in good classifiers.

We back up the claims in this thesis by experimental evaluation. For many of the used databases, the number of patterns initially is huge. However, we show that from this huge collection of patterns, we select a compact and good set of characteristic relational patterns.

Samenvatting

Tegenwoordig worden er grote hoeveelheden informatie opgeslagen in zogenaamde relationele databases. Aangezien het handmatig analyseren van grote hoeveelheden data een ondoenlijke zaak is, ontwikkelt het wetenschappelijke gebied data mining methoden die automatisch dit inzicht trachten te verwerven. Een belangrijke taak is hier weggelegd voor patroon mining: het vinden van regelmatigheden in databases.

Ondanks de vele inspanningen heeft patroon mining tot op heden nog steeds een Achilleshiel; namelijk de hoeveelheid patronen die gevonden worden. Veel gevonden patronen zijn kleine variaties van hetzelfde onderliggende thema, en worden desondanks toch vrijwel allemaal als interessant aangemerkt. In de praktijk wordt een gebruiker overspoeld door te veel vergelijkbare patronen die in het geheel niet veel bijdragen aan nieuw verworven inzicht.

In dit proefschrift beargumenteren wij daarom een andere aanpak. In tegenstelling tot een selectie op basis van *individuele* patronen, stellen wij een selectie op basis van *verzamelingen* patronen voor. In het bijzonder richten wij ons op de selectie op basis van een compressietechniek genaamd het Minimum Description Length (MDL) principe. De geselecteerde patroonverzameling, ons model voor de data, wordt gebruikt om de gehele database te comprimeren. Volgens het MDL principe, is dat model welke de database het meest comprimeert, ook het best beschrijvende.

Daar het verkrijgen van het optimale model simpelweg te lastig is, presenteren wij een praktische, op vuistregels gebaseerde, aanpak genaamd Krimp. Hierop is een scala aan rekenrecepten gebaseerd om zo de verschillende interpretaties van relationele data te kunnen modelleren. Zo beschouwen wij gestructureerde datatypen zoals sequenties en bomen, de samenvoeging van een database, en de relationele database in zijn geheel. Deze laatste modellen blijken tevens goed bruikbaar om ongeziene data te classificeren.

Wij onderbouwen de claims in dit proefschrift via experimentele evaluatie. Voor veel van de gebruikte databases is het aantal patronen initieel aanzienlijk. Echter, we tonen aan dat we uit deze grote collectie, een compacte en kwalitatieve verzameling van karakteristieke relationele patronen vinden.

Dankwoord

Beste Arno, ik wil jou als eerste bedanken voor het mogelijk maken van mijn promotietraject. Je bent altijd een ontspannen begeleider geweest die tijd wist te maken als het er op aan kwam. Zo hebben we weekenden, en ja zelfs nachten, doorgebracht om aan een mooi resultaat te werken. Ik heb daarnaast veel aan je commentaar gehad, en aan de nodige vrijheid die je mij hebt gegeven. Bedankt!

Since each thesis undergoes a laborious review, I would like to thank the members of the reading committee, Hendrik Blockeel, Martin Ester, Linda van der Gaag, Martin Kersten, and Stefan Wrobel, for carefully reading my thesis.

Tijdens mijn promotie had ik een hechte club (oud)collega's, die ik graag wil bedanken voor hun bijdrage aan de gezelligheid: Hans, Lennart, Ad, Arno, Wouter, Ronnie, Jeroen, Edwin, Carsten, Rainer, Diyah, Mohammad, Nicola, en Tiddo. Bedankt voor de leuke lunch gesprekken met terugkerende thema's, het gezamenlijk introquizzen/whatthemovieen, whiskey tastings, Frank's Zoo, coast raften, en zo verder.

Hoewel sommige ze ooit omschreven als slightly hostile but friendly, sluit ik mij aan bij het laatste en wil ik natuurlijk mijn beide (oud) A113 kamergenoten Jilles en Matthijs heel erg bedanken. Al sinds mijn studie heb ik jullie leren kennen als goede vrienden, en sindsdien hebben we veel sfeerverhogende activiteiten meegemaakt. Deze activiteiten waren redelijk divers: culinair (whiskey, bier, en goed eten), cultuur (cabaret en smartlappenfestivals), wetenschappelijk (gelatine experimenten, springchipknipchecker, en de ADA dataset) en handvaardigheid (bebaarde naambordjes en dieren maskers), en nog veel meer. Jongens, heel erg bedankt, en dat we nog maar veel leuke zaken zullen ondernemen!

Verder zijn er de nodige mensen in mijn privé sfeer geweest die ik wil bedanken. Als eerste wederom een set kamergenoten waar ik buiten de studie een goede band mee opgebouwd heb: Bert-Jan en Jeroen. Jongens, bedankt dat jullie naast gezellige tijden ook tijdens mijn promotie meedachten en jullie kijk hierop gaven. Zelfs voor mijn huidige postdoc project wisten jullie al leuke tips te geven!

Natuurlijk wil ik ook mijn vrienden in Bergen op Zoom bedanken, die vele jaren voor veel afleiding in de vorm van gezellige en lange nachten gezorgd hebben. Wat begon als vrienden van school of van het stappen leidde tot een hechte vriendengroep die nu over het hele land verspreid is. Dank jullie wel voor de vele gesprekken en leuke activiteiten! Veel van jullie zie ik regelmatig (tijdens de vastenavend), en hoop ik nog vele jaren te blijven zien! Bedankt Jeroen en Judith, Ronald en Wilma, Paul, Marijn, Patricia, Saskia, Sylvia, Jeroen en Marieke, Chris en Claudia, Jeroen, Mariska, en André.

Een Bergenaar wil ik apart noemen. Han, al vele jaren geleden zijn we bevriend geraakt vanaf het moment dat we samen demo's begonnen te coden. Zonder dat begin had dit boekje er denk ik nooit geweest. Je bent een daarnaast gewoon een erg goede vriend, en samen met Selene zijn jullie vrienden die altijd voor mij klaar staan, betere vrienden kan ik niet wensen!

Met het einde van het dankwoord in zicht kom ik dichter bij huis. Zo wil ik Dhr. de Wild hartelijk bedanken die mij tijdens mijn studie en promotie praktisch als familie opgenomen heeft en zo een tweede thuis heeft geboden. Marie-Louise, wetenschap bracht ons destijds samen en ik ben heel blij dat we nu samen zijn zodat je mij kon steunen tijdens dit staaltje wetenschap. En met jou aan mijn zij was het promoveren eigenlijk best gemakkelijk. Als laatste wil ik natuurlijk mijn moeder bedanken. Ma, zonder jou had ik nooit de stappen kunnen nemen die ik heb willen zetten, en ik ben je erg dankbaar voor dat je mij met vertrouwen die steun gegeven hebt!

Hilversum, april 2010

Curriculum Vitae

Arne is born on the 22nd of January 1977 in Bergen op Zoom, the Netherlands. After completing the lower general secondary education at the RSG 't Rijks in Bergen op Zoom, he obtained his BSc. in Electronics at the Rens&Rens University in Hilversum. After this, he obtained his MSc. in Computer Science at the Universiteit Utrecht, where he specialised in Machine Learning algorithms.

During his studies, he performed many extracurricular research activities such as developing learning algorithms for robotics. This resulted in a successful thesis traineeship at the Ecole Polytechnique Fédérale de Lausanne, that was superbly supervised by Daniel Roggen.

After this, he took a small sidestep from research, and worked as an Information Management consultant at Shell Exploration and Production in Rijswijk. After roughly a year, in 2005, he started his PhD research at the Algorithmic Data Analysis group of Arno Siebes at the Universiteit Utrecht. He is currently employed as a post-doc at Leiden University.

SIKS Dissertation Series

1998	1	J. van den Akker (CWI), DEGAS - An Active, Temporal Database of Autonomous Objects
	2	F. Wiesman (UM), Information Retrieval by Graphically Browsing Meta-Information
	3	A.N.S. Steuten (TUD), A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
	4	D. Breuker (UM), Memory versus Search in Games
	5	E.W. Oskamp (RUL), Computerondersteuning bij Straftopmeting
1999	1	M. Sloof (VU), Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
	2	R. Potharst (EUR), Classification using decision trees and neural nets
	3	D. Beal (UM), The Nature of Minimax Search
	4	J. Penders (UM), The practical Art of Moving Physical Objects
	5	A. de Moor (KUB), Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
	6	N.J.E. Wijngaards (VU), Re-design of compositional systems
	7	D. Spelt (UT), Verification support for object database design
	8	J.H.J. Lenting (UM), Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation
2000	1	F. Niessink (VU), Perspectives on Improving Software Maintenance
	2	K. Holtman (TUE), Prototyping of CMS Storage Management
	3	C.M.T. Metselaar (UVA), Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief
	4	G. de Haan (VU), ETAG, A Formal Model of Competence Knowledge for User Interface Design
	5	R. van der Pol (UM), Knowledge-based Query Formulation in Information Retrieval
	6	R. van Eijk (UU), Programming Languages for Agent Communication
	7	N. Peek (UU), Decision-theoretic Planning of Clinical Patient Management
	8	V. Coupé (EUR), Sensitivity Analyis of Decision-Theoretic Networks
	9	F. Waas (CWI), Image database management system design considerations, algorithms and architecture
	11	J. Karlsson (CWI), Scalable Distributed Data Structures for Database Management
2001	1	S. Renooij (UU), Qualitative Approaches to Quantifying Probabilistic Networks
	2	K. Hindriks (UU), Agent Programming Languages: Programming with Mental Models
	3	M. van Someren (UVA), Learning as problem solving
	4	E. Smirnov (UM), Conjunctive and disjunctive version spaces with instance-based boundary sets
	5	J. van Ossenbruggen (VU), Processing Structured Hypermedia: A Matter of Style
	6	M. van Welie (VU), Task-based User Interface Design
	7	B. Schonhage (VU), Diva: Architectural Perspectives on Information Visualization
	8	P. van Eck (VU), A Compositional Semantic Structure for Multi-Agent Systems Dynamics
	9	P.J. 't Hoen (RUL), Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
	10	M. Sierhuis (UVA), Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design
	11	T.M. van Engers (VU), Knowledge management: The Role of Mental Models in Business Systems Design
2002	1	N. Lassing (VU), Architecture-Level Modifiability Analysis
	2	R. van Zwol (UT), Modelling and Searching Web-based Document Collections
	3	H.E. Blok (UT), Database Optimization Aspects for Information Retrieval
	4	J.R. Castelo Valdueza (UU), The Discrete Acyclic Digraph Markov Model in Data Mining
	5	R. Serban (VU), The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
	6	L. Mommers (UL), Applied legal epistemology
	7	P. Boncz (CWI), Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
	8	J. Gordijn (VU), Value-based requirements engineering: exploring innovative e-commerce ideas
	9	W-J. van den Heuvel (KUB), Integrating Modern Business Applications with Objectified Legacy Systems
	10	B. Sheppard (UM), Towards Perfect Play of Scrabble
	11	W.C.A. Wijngaards (VU), Agent based modelling of dynamics: biological and organisational applications
	12	A. Schmidt (UVA), Processing XML in Database Systems
	13	H. Wu (TUE), A Reference Architecture for Adaptive Hypermedia Applications
	14	W. de Vries (UU), Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
	15	R. Eshuis (UT), Semantics and Verification of UML Activity Diagrams for Workflow Modelling
	16	P. van Langen (VU), The Anatomy of Design: Foundations, Models and Applications
	17	S. Manegold (UVA), Understanding, Modeling, and Improving Main-Memory Database Performance
2003	1	H. Stuckenschmidt (VU), Ontology-based information sharing in weakly structured environments
	2	J. Broersen (VU), Modal Action Logics for Reasoning About Reactive Systems
	3	M. Schuemie (TUD), Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
	4	M. Petkovic (UT), Content-Based Video Retrieval Supported by Database Technology
	5	J. Lehmann (UVA), Causation in Artificial Intelligence and Law - A modelling approach
	6	B. van Schooten (UT), Development and Specification of Virtual Environments
	7	M. Jansen (UVA), Formal Explorations of Knowledge Intensive Tasks
	8	Y. Ran (UM), Repair Based Scheduling
	9	R. Kortmann (UM), The Resolution of Visually Guided Behaviour
	10	A. Lincke (UT), Some Experimental Studies on the Interaction between Medium, Innovation context and Culture

	11	S. Keizer (UT), Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
	12	R. Ordelman (UT), Dutch Speech Recognition in Multimedia Information Retrieval
	13	J. Donkers (UM), Nosce Hostem - Searching with Opponent Models
	14	S. Hoppenbrouwers (KUN), Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
	15	M. de Weerdt (TUD), Plan Merging in Multi-Agent Systems
	16	M. Windhouwer (CWI), Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
	17	D. Jansen (UT), Extensions of Statecharts with Probability, Time, and Stochastic Timing
	18	L. Kocsis (UM), Learning Search Decisions
2004	1	V. Dignum (UU), A Model for Organizational Interaction: Based on Agents, Founded in Logic
	2	L. Xu (UT), Monitoring Multi-party Contracts for E-business
	3	P. Groot (VU), A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
	4	C. van Aart (UVA), Organizational Principles for Multi-Agent Architectures
	5	V. Popova (EUR), Knowledge Discovery and Monotonicity
	6	B-J. Hommes (TUD), The Evaluation of Business Process Modeling Techniques
	7	E. Boltjes (UM), Voorbeeldig onderwijs
	8	J. Verbeek (UM), Politie en de Nieuwe Internationale Informatiemarkt
	9	M. Caminada (VU), For the Sake of the Argument; Explorations into Argument-based Reasoning
	10	S. Kabel (UVA), Knowledge-rich Indexing of Learning-objects
	11	M. Klein (VU), Change Management for Distributed Ontologies
	12	T. Duy Bui (UT), Creating Emotions and Facial Expressions for Embodied Agents
	13	W. Jamroga (UT), Using Multiple Models of Reality: On Agents who Know how to Play
	14	P. Harrenstein (UU), Logic in Conflict. Logical Explorations in Strategic Equilibrium
	15	A. Knobbe (UU), Multi-Relational Data Mining
	16	F. Divina (VU), Hybrid Genetic Relational Search for Inductive Learning
	17	M. Winands (UM), Informed Search in Complex Games
	18	V. Bessa Machado (UVA), Supporting the Construction of Qualitative Knowledge Models
	19	T. Westerveld (UT), Using Generative Probabilistic Models for Multimedia Retrieval
	20	M. Evers (Nyenrode), Learning from Design: Facilitating Multidisciplinary Design Teams
2005	1	F. Verdenius (UVA), Methodological Aspects of Designing Induction-Based Applications
	2	E. van der Werf (UM), AI techniques for the Game of Go
	3	F. Grootjen (RUN), A Pragmatic Approach to the Conceptualisation of Language
	4	N. Meratnia (UT), Towards Database Support for Moving Object data
	5	G. Infante-Lopez (UVA), Two-Level Probabilistic Grammars for Natural Language Parsing
	6	P. Spronck (UM), Adaptive Game AI
	7	F. Frasincar (TUE), Hypermedia presentation generation for semantic web information systems
	8	R. Vdovjak (TUE), A Model-driven Approach for Building Distributed Ontology-based Web Applications
	9	J. Broekstra (VU), Storage, Querying and Inferencing for Semantic Web Languages
	10	A. Bouwer (UVA), Explaining behaviour: using qualitative simulation in interactive learning
	11	E. Ogston (VU), Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
	12	C. Boer (EUR), Distributed Simulation in Industry
	13	F. Hamburg (UL), Een Computermodel voor het ondersteunen van euthanasiebeslissingen
	14	B. Omelayenko (VU), Web-Service configuration on the Semantic Web; Exploring how Semantics meets Pragmatics
	15	T. Bosse (VU), Analysis of the Dynamics of Cognitive Processes
	16	J. Graaumans (UU), Usability of XML Query Languages
	17	B. Shishkov (TUD), Software Specification Based on Re-usable Business Components
	18	D. Sent (UU), Test-selection Strategies for Probabilistic Networks
	19	M. van Dartel (UM), Situated Representation
	20	C. Coteanu (UL), Cyber Consumer Law, State of the Art and Perspectives
	21	W. Derks (UT), Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics
2006	1	S. Angelov (TUE), Foundations of B2B Electronic Contracting
	2	C. Chisalita (VU), Contextual issues in the Design and use of Information Technology in Organizations
	3	N. Christoph (UVA), The role of Metacognitive Skills in Learning to Solve Problems
	4	M. Sabou (VU), Building Web Service Ontologies
	5	C. Pierik (UU), Validation Techniques for Object-Oriented Proof Outlines
	6	Z. Baida (VU), Software-aided service bundling - intelligent methods tools for graphical service modeling
	7	M. Smiljanic (UT), XML schema matching - balancing efficiency and effectiveness by means of clustering
	8	E. Herder (UT), Forward, Back and Home Again - Analyzing User Behavior on the Web
	9	M. Wahdan (UM), Automatic Formulation of the Auditor's Opinion
	10	R. Siebes (VU), Semantic Routing in Peer-to-Peer Systems
	11	J. van Ruth (UT), Flattening Queries over Nested Data Types
	12	B. Bongers (VU), Interactivation - Towards an e-cology of people, our t-environment, and the arts
	13	H-J. Lebbink (UU), Dialogue and Decision Games for Information Exchanging Agents
	14	J. Hoorn (VU), Software requirements: update, upgrade, redesign
	15	R. Malik (UU), CONAN: Text Mining in the Biomedical Domain
	16	C. Riggelsen (UU), Approximation Methods for Efficient Learning of Bayesian Networks
	17	S. Nagata (UU), User Assistance for Multitasking with Interruptions on a Mobile Device
	18	V. Zhizhkun (UVA), Graph transformation for Natural Language Processing
	19	B. van Riemsdijk (UU), Cognitive Agent Programming: A Semantic Approach
	20	M. Velikova (UvT), Monotone models for prediction in data mining
	21	B. van Gils (RUN), Aptness on the Web
	22	P. de Vrieze (RUN), Fundaments of Adaptive Personalisation

	23	I. Juvina (UU), Development of Cognitive Model for Navigating on the Web
	24	L. Hollink (VU), Semantic Annotation for Retrieval of Visual Resources
	25	M. Drugan (UU), Conditional log-likelihood MDL and Evolutionary MCMC
	26	V. Mihajlovic (UT), Score Region Algebra: A Flexible Framework for Structured Information Retrieval
	27	S. Bocconi (CWI), Vox Populi: generating video documentaries from semantically annotated media repositories
	28	B. Sígurbjørnsson (UVA), Focused Information Access using XML Element Retrieval
2007	1	K. Leune (UvT), Access Control and Service-Oriented Architectures
	2	W. Teepe (RUG), Reconciling Information Exchange and Confidentiality: A Formal Approach
	3	P. Mika (VU), Social Networks and the Semantic Web
	4	J. van Diggelen (UU), Achieving Semantic Interoperability in Multi-agent Systems
	5	B. Schermer (UL), Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
	6	G. Mishne (UVA), Applied Text Analytics for Blogs
	7	N. Jovanovic' (UT), To whom it may concern - addressee identification in face-to-face meetings
	8	M. Hoogendoorn (VU), Modeling of Change in Multi-Agent Organizations
	9	D. Mobach (VU), Agent-Based Mediated Service Negotiation
	10	H. Aldewereld (UU), Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
	11	N. Stash (TUE), Incorporating cognitive learning styles in a gen-purpose adaptive hypermedia system
	12	M. van Gerven (RUN), Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
	13	R. Rienks (UT), Meetings in Smart Environments; Implications of Progressing Technology
	14	N. Bergboer (UM), Context-Based Image Analysis
	15	J. Lacroix (UM), NIM: a Situated Computational Memory Model
	16	D. Grossi (UU), Designing Invisible Handcuffs
	17	T. Charitos (UU), Reasoning with Dynamic Networks in Practice
	18	B. Orriens (UvT), On the development an management of adaptive business collaborations
	19	D. Levy (UM), Intimate relationships with artificial partners
	20	S. Jansen (UU), Customer Configuration Updating in a Software Supply Network
	21	K. Vermaas (UU), Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
	22	Z. Zlatev (UT), Goal-oriented design of value and process models from patterns
	23	P. Barna (TUE), Specification of Application Logic in Web Information Systems
	24	G. Ramírez Camps (CWI), Structural Features in XML Retrieval
	25	J. Schalken (VU), Empirical Investigations in Software Process Improvement
2008	1	K. Boer-Sorbán (EUR), Agent-Based Simulation of Financial Markets
	2	A. Sharpanskykh (VU), On computer-aided methods for modeling and analysis of organizations
	3	V. Hollink (UVA), Optimizing hierarchical menus: a usage-based approach
	4	A. de Keijzer (UT), Management of Uncertain Data - towards unattended integration
	5	B. Mutschler (UT), Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
	6	A. Hommersom (RUN), On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
	7	P. van Rosmalen (OU), Supporting the tutor in the design and support of adaptive e-learning
	8	J. Bolt (UU), Bayesian Networks: Aspects of Approximate Inference
	9	C. van Nimwegen (UU), The paradox of the guided user: assistance can be counter-effective
	10	W. Bosma (UT), Discourse oriented summarization
	11	V. Kartseva (VU), Designing Controls for Network Organizations: A Value-Based Approach
	12	J. Farkas (RUN), A Semiotically Oriented Cognitive Model of Knowledge Representation
	13	C. Carraciolo (UVA), Topic Driven Access to Scientific Handbooks
	14	A. van Bunningen (UT), Context-Aware Querying; Better Answers with Less Effort
	15	M. van Otterlo (UT), The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains
	16	H. van Vugt (VU), Embodied agents from a user's perspective
	17	M. Op 't Land (TUD), Applying Architecture and Ontology to the Splitting and Allying of Enterprises
	18	G. de Croon (UM), Adaptive Active Vision
	19	H. Rode (UT), From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
	20	R. Arendsen (UVA), Geen bericht, goed bericht
	21	K. Balog (UVA), People Search in the Enterprise
	22	H. Koning (UU), Communication of IT-Architecture
	23	S. Visscher (UU), Bayesian network models for the management of ventilator-associated pneumonia
	24	Z. Aleksovski (VU), Using background knowledge in ontology matching
	25	G. Jonker (UU), Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
	26	M. Huijbregts (UT), Segmentation, diarization and speech transcription: surprise data unraveled
	27	H. Vogten (OU), Design and Implementation Strategies for IMS Learning Design
	28	I. Flesch (RUN), On the Use of Independence Relations in Bayesian Networks
	29	D. Reidsma (UT), Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
	30	W. van Atteveldt (VU), Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
	31	L. Braun (UM), Pro-Active Medical Information Retrieval
	32	T.H. Bui (UT), Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
	33	F. Terpstra (UVA), Scientific Workflow Design; theoretical and practical issues
	34	J. De Knijf (UU), Studies in Frequent Tree Mining
	35	B.T. Nielsen (UvT), Dendritic morphologies: function shapes structure

2009 1 R. Jurgelenaite (RUN), Symmetric Causal Independence Models
2 W.R. van Hage (VU), Evaluating Ontology-Alignment Techniques
3 H. Stol (UvT), A Framework for Evidence-based Policy Making Using IT
4 J. Nabukenya (RUN), Improving the Quality of Organisational Policy Making using Collaboration Engineering
5 S. Overbeek (RUN), Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
6 M. Subianto (UU), Understanding Classification
7 R. Poppe (UT), Discriminative Vision-Based Recovery and Recognition of Human Motion
8 V. Nannen (VU), Evolutionary Agent-Based Policy Analysis in Dynamic Environments
9 B. Kanagwa (RUN), Design, Discovery and Construction of Service-oriented Systems
10 J.A.N. Wielemaker (UVA), Logic programming for knowledge-intensive interactive applications
11 A. Boer (UVA), Legal Theory, Sources of Law & the Semantic Web
12 P. Massuthe (TUE, Humboldt-Universitaet zu Berlin), Operating Guidelines for Services
13 S. de Jong (UM), Fairness in Multi-Agent Systems
14 M. Korotkiy (VU), From ontology-enabled services to service-enabled ontologies
15 R. Hoekstra (UVA), Ontology Representation - Design Patterns and Ontologies that Make Sense
16 F. Reul (UvT), New Architectures in Computer Chess
17 L. van der Maaten (UvT), Feature Extraction from Visual Data
18 F. Groffen (CWI), Armada, An Evolving Database System
19 V. Robu (CWI), Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
20 B. van der Vecht (UU), Adjustable Autonomy: Controling Influences on Decision Making
21 S. Vanderlooy (UM), Ranking and Reliable Classification
22 P. Serdyukov (UT), Search For Expertise: Going beyond direct evidence
23 P. Hofgesang (VU), Modelling Web Usage in a Changing Environment
24 A. Heuvelink (VUA), Cognitive Models for Training Simulations
25 A. van Ballegooij (CWI), RAM: Array Database Management through Relational Mapping
26 F. Koch (UU), An Agent-Based Model for the Development of Intelligent Mobile Services
27 C. Glahn (OU), Contextual Support of social Engagement and Reflection on the Web
28 S. Evers (UT), Sensor Data Management with Probabilistic Models
29 S. Pokraev (UT), Model-Driven Semantic Integration of Service-Oriented Applications
30 M. Zukowski (CWI), Balancing vectorized query execution with bandwidth-optimized storage
31 S. Katrenko (UVA), A Closer Look at Learning Relations from Text
32 R. Farenhorst and R. de Boer (VU), Architectural Knowledge Management: Supporting Architects and Auditors
33 K. Truong (UT), How Does Real Affect^3 Recognition In Speech?
34 I. van de Weerd (UU), Advancing in Software Product Management: An Incremental Method Engineering Approach
35 W. Koelewijn (UL), Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling
36 M. Kalz (OUN), Placement Support for Learners in Learning Networks
37 H. Drachsler (OUN), Navigation Support for Learners in Informal Learning Networks
38 R. Vuorikari (OU), Tags and self-organisation: a metadata ecology for learning resources in a multilingual context
39 C. Stahl (TUE), Service Substitution – A Behavioral Approach Based on Petri Nets
40 S. Raaijmakers (UvT), Multinomial Language Learning
41 I. Berezhnyy (UvT), Digital Analysis of Paintings
42 T. Bogers, Recommender Systems for Social Bookmarking
43 V. Franqueira (UT), Finding Multi-step Attacks in Computer Networks using Heuristic Search
44 R. Santana Tapia (UT), Assessing Business-IT Alignment in Networked Organizations
45 J. Vreeken (UU), Making Pattern Mining Useful
46 L. Afanasiev (UvA), Querying XML: Benchmarks and Recursion

2010 1 M. van Leeuwen (UU), Patterns that Matter
2 I. Wassink (UT), Work flows in Life Science
3 J. Geurts (CWI), A Document Engineering Model and Processing Framework for documents
4 O. Kulyk (UT), Do You Know What I Know? Situational Awareness of Co-located Teams
5 C. Hauff (UT), Predicting the Effectiveness of Queries and Retrieval Systems
6 S. Bakkes (UvT), Rapid Adaptation of Video Game AI
7 W. Fikkert (UT), A Gesture interaction at a Distance
8 K. Siewicz (UL), Towards an Improved Regulatory Framework of Free Software
9 H. Kielman (UL), A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging
10 R. Ong (UL), Mobile Communication and Protection of Children
11 A. Ter Mors (TUD), The world according to MARP: Multi-Agent Route Planning
12 S. van den Braak (UU), Sensemaking software for crime analysis
13 G. Folino (RUN), High Performance Data Mining using Bio-inspired techniques
14 S. van Splunter (VU), Automated Web Service Reconfiguration
15 L. Bodenstaff (UT), Managing Dependency Relations in Inter-Organizational Models
16 S. Verwer (TUD), Efficient Identification of Timed Automata, theory and practice
17 S. Kotoulas (VU), Scalable Discovery of Networked Resources
18 C. Gerritsen (VU), Caught in the Act: Investigating Crime by Agent-Based Simulation
19 H. Cramer (UvA), People's Responses to Autonomous and Adaptive Systems
20 I. Swartjes (UT), Whose Story Is It Anyway? How Improv Informs Agency
21 H. van Heerde (UT), Privacy-aware data management by means of data degradation
22 M. Hildebrand (CWI), End-user Support for Access to Heterogeneous Linked Data
23 B. Steunebrink (UU), The Logical Structure of Emotions
24 D. Tykhonov, Designing Generic and Efficient Negotiation Strategies
25 Z. Memon (VU), Modelling Human-Awareness for Ambient Agents
26 Y. Zhang (CWI), XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
27 M. Voulon (UL), Automatisch contracteren