Knowledge Discovery in Databases: The Search for Frequent Patterns

Heikki Mannila   Hannu Toivonen

September 24, 1998
(Minor modifications in 2002)

Contents

Preface

Knowledge discovery in databases (KDD), often called data mining, aims at the discovery of useful information from large collections of data. The discovered knowledge can be rules describing properties of the data, frequently occurring patterns, clusterings of the objects in the database, etc. Data mining has in the 1990's emerged as visible research and development area; both in industry and in science there is a need for methods for efficient analysis of large data sets.

This book is intended to give an overview of the basic issues in KDD. The technical emphasis is on the discovery of frequently occurring patterns, both in relational and in sequential data, and in the search for database integrity constraints from relational databases.

The book is divided into an introduction and 5 technical parts:

1. *Introduction* (Chapter 1): what is knowledge discovery?

2. *Discovery of association rules* (Chapter 2). Association rules are a simple but useful formalism for expressing regularities in databases. Association rules are a good example of how frequent patterns can be located efficiently in a vast search space.

3. *Case study* (Chapters 3 – 6): discovery of episodes in telecommunications alarm databases. In these chapters our aim is to give a general picture of the process of knowledge discovery, from the initial problem description to experiences with a fielded system. We also give detailed, domain independent algorithms for the analysis of sequences of events.

4. *Theoretical aspects of knowledge discovery* (Chapters 7 and 8). We give a universal formulation of the problem of discovering of all frequent patterns. We also provide a generic solution, and analyze properties of both the problem and the solution.

5. *Sampling* (Chapter 9). In the final chapter we discuss sampling and show how it can be used to boost the (exact) discovery of patterns.

Chapter 1

Introduction

Current technology makes it fairly easy to collect data, but data analysis tends to be slow and expensive. Knowledge discovery in databases (KDD), often called data mining, aims at the discovery of useful information from large collections of data. The motivation for KDD is a suspicion that there might be nuggets of useful information hiding in the masses of unanalyzed or underanalyzed data, and therefore semiautomatic methods for locating interesting information from data would be useful. The discovered knowledge can be rules describing properties of the data, frequently occurring patterns, clusterings of the objects in the database, etc. There are several successful applications of data mining. See [23] for a recent overview of the area.

This introductory chapter gives a short discussion of some of the issues in knowledge discovery. We start in Section 1.1 by looking at the definition of knowledge discovery and some small examples, and we briefly discuss the basic goals of knowledge discovery. The KDD process is considered in Section 1.2. Some applications of KDD are described in Section 1.3. Section 1.4 discusses the role of machine learning and statistics in KDD and data mining, Section 1.5 the role of databases.

1.1 What is knowledge discovery

Knowledge discovery in database can be loosely defined as the task of finding interesting and potentially useful knowledge from large masses of data. Examples of types of knowledge that could be discovered are

- association rules:
  "80 % of customers who buy beer and sausage buy also mustard"
- rules: "if Age < 40 then Income < 10"
- functional dependencies:
  "if $t[A] = u[A]$, then $t[B] = u[B]$"

- belief networks
- clusterings

To illustrate one of these concepts, we consider briefly association rules. An *association rule* [1] about a relation r over schema R is an expression of the form $X \Rightarrow B$, where $X \subseteq R$ and $B \in R \setminus X$. The intuitive meaning of the rule is that if a row of the matrix r has a 1 in each column of X, then the row tends to have a 1 also in column B.

Examples of data where association rules might be applicable include the following.

- A student database at a university: rows correspond to students, columns to courses, and a 1 in entry (s, c) indicates that the student s has taken course c.
- Data collected from bar-code readers in supermarkets: columns correspond to products, and each row corresponds to the set of items purchased at one time.
- A database of publications: the rows and columns both correspond to publications, and $(p, p') = 1$ means that publication p refers to publication p'.
- A set of measurements about the behavior of a number of systems, say exchanges in a telephone network. The columns correspond to the presence or absence of certain conditions, and each row corresponds to a measurement: if entry (m, c) is 1, then at measurement m condition c was present.

Given $X \subseteq R$, we denote by $fr(X, r)$ the *frequency* of X in r: the fraction of rows of r that have a 1 in each column of X. The *frequency* of the rule $X \Rightarrow B$ in r is defined to be $fr(X \cup \{B\}, r)$, and the *confidence* of the rule is $fr(X \cup \{B\}, r)/fr(X, r)$. The confidence is the observed probability with which a row containing X also contains B.

In the discovery of association rules, the task is to find all rules $X \Rightarrow B$ such that the frequency of the rule is at least a given threshold *min_fr* and the confidence of the rule is at least another threshold *min_conf*. In large retailing applications the number of rows might easily be 10^6 or 10^7, and the number of columns around 5000. The frequency threshold *min_fr* typically is around 10^{-2} — 10^{-5}. The confidence threshold *min_conf* can be anything from 0 to 1. From such a database one might obtain hundreds or thousands of association rules. (Of course, one has to be careful in assigning any statistical significance to findings obtained with such methods.)

Note that there is no predefined limit on the number of attributes of the left-hand side X of an association rule $X \Rightarrow B$; this is important so that unexpected associations are not ruled out before the processing starts. It also means that the search space of the rules has exponential size in the number of attributes of the input relation. Handling this requires some care for the algorithms, but there is a simple way of pruning the search space.

Example 1.1 Discovered in a student enrolment database, the association rule { Distributed Operating Systems, Introduction to Unix } $\Rightarrow$ Programming in C (frequency = 2 %, confidence = 96 %) states that 96 % of the students that have taken Distributed Operating Systems and Introduction to Unix, also have taken Programming in C, and that 2 % of all the students actually have taken all three courses. Such rules can be useful in obtaining a picture of the combinations of courses actually taken by the students. The acquired knowledge can be applied, e.g., in the design of individual courses and the whole curriculum.

In market basket analysis, the number of products being sold can be large and it is more difficult to have a hunch for all the associations in the data. On the other hand, discovered patterns can be very valuable for the business. A well-known story tells how a surprising association was discovered between diapers and beer in late afternoons. With some research a fairly natural cause was discovered: young fathers buying diapers on the way back home from the work. This simple but surprising observation could maybe be capitalized by placing beer closer to diapers, in order to make the impulse shopping behaviour stronger, and maybe by placing chips right next to diapers and beer. □

1.2 The KDD process

The goal of knowledge discovery is to obtain useful knowledge from large collections of data. Such a task is inherently interactive and iterative: one cannot expect to obtain useful knowledge simply by pushing a lot of data to a black box. The user of a KDD system has to have a solid understanding of the domain in order to select the right subsets of data, suitable classes of patterns, and good criteria for interestingness of the patterns. Thus KDD systems should be seen as interactive tools, not as fully automatic analysis systems.

Discovering knowledge from data should therefore be seen as a process containing several steps:

1. understanding the domain;
2. preparing the data set;
3. discovering patterns (data mining),

4. postprocessing and presenting the discovered patterns, and
5. putting the results into use.

(See [22] for a slightly different process model and excellent discussion.)

Understanding the domain of the data is naturally a prerequisite for extracting anything useful: the user of a KDD system has to have some sort of understanding about the application area before any valuable information can be obtained. As an example, consider the discovered association rule between diapers and beer. Beer probably is purchased in addition to diapers, and the rule indicates a link that can potentially be strengthened. On the other hand, if a strong association is discovered between bread and butter, the actions taken could be opposite: placing bread and butter far away from each other would force customers who really came to pick up both items to spend more time in the store. Critical interpretation of data mining results and the drawing of conclusions typically require good domain knowledge. Data mining does not work by pushing a button.

If very good human experts exist for a domain, it can be hard for semi-automatic tools to obtain any novel information. This can be the case in fairly stable domains, where the humans have had time to achieve expertise even in the details of the data. A possible example occurs in some areas of retailing, where the products and customer profiles can stay about the same for longer periods of time. The easiest application areas for KDD seem to be ones where general human experts can be found, but the actual micro-level properties of the data are changing. This seems to be the case in, e.g., telecommunications, where the operators of the networks have a fairly good overview of the systems characteristics, but changes and updates in equipment and software mean that actual expertise in the details of the data is more difficult to obtain.

Preparation of the data set involves selection of the data sources, integration of heterogeneous data, cleaning the data from errors, assessing noise, dealing with missing values, etc. This step can easily take up most of the time needed for the whole KDD process. This is not surprising: the difficulties in data integration are well known.

The pattern discovery phase in KDD is the step where the interesting and frequently occurring patterns are discovered from the data. In this paper we follow the terminology introduced in [22]: data mining refers to the pattern discovery part of knowledge discovery. Elsewhere, especially in industry, data mining is often used as a synonym for KDD.

The data mining step can use various techniques from statistics and machine learning, such as rule learning, decision tree induction, clustering, inductive logic programming, etc. The emphasis in data mining research is mostly on efficient discovery of fairly simple patterns.

The KDD process does not stop when patterns have been discovered. The user has to be able to understand what has been discovered, to view the data and patterns simultaneously, contrast the discovered patterns with background knowledge, etc. Postprocessing of discovered knowledge involves steps such as further selection or ordering of patterns, visualization, etc. Some approaches to KDD methodology put a heavy emphasis on postprocessing.

The KDD process is necessarily iterative: the results of a data mining step can show that some changes should be made to the data set formation step, postprocessing of patterns can cause the user to look for some slightly modified types of patterns, etc. Efficient support for such iteration is one important topic of development in KDD.

1.3 Applications of knowledge discovery

1.3.1 Scientific applications

Prominent applications of KDD include health care data, financial applications, and scientific data [65, 53, 20].

One of the more spectacular applications is the SKICAT system [21], which operates on 3 terabytes of image data. Image processing of the pixels produces approximately 2 billion objects, basically smudges of light, each with 40 attributes. To be usable, the objects have to be classified into a few classes: stars, galaxies, etc.

The task is obviously impossible to do manually. Using example classifications provided by experts, the system induced decision trees and extracted classification rules for the problem. The results are spectacular: verification shows that the resulting classification is accurate, and the classification has already been used to discover new high-redshift quasars.

Another astronomical application has been done on the radar data produced by the Magellan spacecraft that has surveyed the surface of Venus using radar. The basic problem was to find out which features observed on the surface are volcanos and which are not. The problem is complicated by the fact that finding the "ground truth" is by no means trivial: it is not possible to obtain more accurate information about the surface of Venus than that obtained by the Magellan spacecraft. The approach taken in [11] was to search automatically for volcanos by first training the system using examples provided by geologists.

In biological sciences the ability to analyze the 1-dimensional structure of genes and proteins has made significant advances possible. Finding the structures and functions of proteins is central in molecular biology. The protein and gene data sets yield several important data mining problems, such as the location of recurrent motifs from protein sequences, and similarity searches among large sets of sequences. Several research groups have recently used sophisticated hidden Markov model (HMM) methods to look for such structural patterns [54, 55].

1.3.2 Business applications

In business, the main area of application for KDD techniques has been marketing. A typical problem is targeting of mail advertising: how to determine which products should be offered to customers, on the basis of their past purchase behavior? A good example of this type of work is the system Opportunity Explorer [6].

In many business domains, publication of details of succesful systems is rare. Often the reason is simple: if data mining gives a competitive advantage, it is not wise to spread the word to competitors. Succesfull applications are reported in in portfolio management, fraud detection, manufacturing and production, and network management. See [10] for an overview of applications.

1.3.3 Data warehousing, OLAP, and knowledge discovery

In industry, the success of KDD is partly related to the rise of the concepts of *data warehousing* and *on-line analytical processing (OLAP)*. These strategies for the storage and processing of the accumulated data in an organization have become popular in recent years. KDD and data mining can be viewed as ways of realizing some of the goals of data warehousing and OLAP.

In principle, a data warehouse aims at the storage and processing of all relevant business data on an enterprise wide level for different analysis tasks. The motivation for data warehousing is that such integrated data storage can be immensely useful for making valuable inferences about the business across the enterprise.

Knowledge discovery is a good way of using the data in the data warehouse. Data warehousing, on the other hand, makes long-term knowledge discovery cheaper, as a data warehouse removes a lot of the effort needed in the early stages of the KDD process. On the other hand, data warehousing often is very expensive: the integration of data from different sources can be quite costly.

On-line analytical processing, OLAP, is a term introduced as a counterpart to *on-line transaction processing*, OLTP, the traditional mode of database usage. In OLAP the emphasis is on producing different types of multidimensional reports for the use of strategic decision making in the organization. A possible way of differentiating between KDD and OLAP is to say that OLAP is oriented towards *verification*, whereas KDD is typically aiming at *discovery*. In OLAP systems, the user typically has in his mind a question that he wants to be answered, whereas in KDD the user typically has a less clear view of what is to be found from the data.

The boundary of KDD and OLAP is vague, and typically a succesful KDD session leads to some more focused questions that could be labeled OLAP-type.

1.4 KDD versus machine learning and statistics

Data mining combines methods and tools from at least three areas: machine learning, statistics, and databases. Indeed, one can sometimes hear the following comments.

- Data mining is just machine learning!
- Data mining is just statistics!
- What does data mining have to do with databases?

In this section we discuss briefly the first two points; the next sections are devoted to a discussion on the third point.

The close links between machine learning, statistics, and data mining are fairly obvious. All three areas aim at locating interesting regularities, patterns, or concepts from empirical data. The exact relationships of these areas have been subject to some debate.

Machine learning methods form the core of data mining: decision tree learning or rule induction is one of the main components of several data mining algorithms. There are some differences, however.

The emphasis on the *process* of knowledge discovery is one; large parts of machine learning literature concentrate on just the learning or induction step, although exceptions of course exist.

The next difference concerns the relative roles of concepts and data. It seems that most of machine learning research assumes there is *something to be learned*, i.e., that there is an underlying interesting concept or mechanism that produces the data. The data can be corrupted by noise, errors, etc., but still the idea is that there is an interesting concept at the bottom. In knowledge discovery, on the other hand, the data is the primary thing, and one does not necessarily assume that there would be any sensible structure behind the data. For example, in analyzing retail sales data, the data is what it is, and the users are not interested in obtaining a full understanding of it; useful nuggets of information are sufficient. Of course, this difference is not absolute.

A third difference is related to the goals. KDD systems typically have fairly modest aims, in terms of the complexity of the obtained knowledge. Whereas parts of machine learning research aim at learning things that are difficult for humans to do, most of the work in KDD aims at finding knowledge that a competent data analyst would in principle be able to find, if he had the time. This distinction is particularly evident when one compares the area of machine discovery to knowledge discovery.

An often cited difference between KDD and machine learning is the amount of data. Traditionally, machine learning research has concentrated on looking at data sets containing hundreds or thousands of examples, while

KDD applications consider larger data sets. It is not clear how significant this distinction is, however: some machine learning work has been done on huge data sets, and KDD methods can be useful even on small data collections. Furthermore, the essential source of complexity in data mining is typically not the number of objects in the database, but rather the number of attributes: the number of possible patterns typically grows at least exponentially in the number of attributes. This growth is the real source of difficulty, not the number of objects in the database.

Summarizing, machine learning is at the core of KDD, but there are differences between the areas.

In *statistics* the term data mining has been used for a long time, often in slightly derogatory fashion, as referring to data analysis without clearly formulated hypotheses. A more fashionable term is *exploratory data analysis* (EDA) [86], which stresses the supremacy of data as guiding the analysis process. KDD and EDA have very similar aims and methods.

According to the interesting statistical perspective on KDD by Elder and Pregibon [19], the focus of statistics has gradually moved from model estimation to model selection. Instead of looking for the parameter values that make a model fit the data well, also the model structure is part of the search process. This trend fits the goals of KDD nicely: one does not want to fix the model structure in advance. The recent advances in, say Markov chain Monte Carlo (MCMC) methods, make it possible to consider far larger model spaces than previously. In addition to these techniques, the KDD community has lots to learn from statistics, e.g., in the handling of uncertainty.

The main difference between KDD and statistics is perhaps in the extensive use of machine learning methods in KDD, in the volume of data, and in the role of computational complexity issues in KDD. For example, even MCMC methods have difficulties in handling tens of thousands of parameter values; some sort of combinatorial preprocessing is needed to make the model selection task tractable. It seems that such combinations of methods can be useful: combinatorial techniques are used to prune the search space, and statistical methods are used to explore the remaining parts in great detail.

1.5 Databases and data mining

What is the role of database management systems in data mining? Normal use of databases can be seen as *deduction*, whereas knowledge discovery aims at *induction*. Furthermore, parts of database technology are not very relevant to knowledge discovery. For example, recovery and transaction management are issues that a KDD application typically does not have to care about.

Nevertheless, database management systems have been especially developed for the storage and flexible retrieval of large masses of structured data, so at least in principle they should be suited for KDD. What database systems have to offer is basically fast access to certain subgroups of a data set and efficient computation of some characteristics of such subgroups.

Usual database systems are not very well suited for such tasks. Implementation of classification algorithms (say, C4.5) or neural networks on top of a large database require tighter coupling with the database system and smart use of indexing techniques. For example, training a classifier on a large training set stored in a database requires possibly multiple passes through the data using different orderings between attributes. This can be implemented by utilizing DBMS support for aggregate operations, indexes and database sorting ('order by'). Clustering may require efficient implementations of nearest neighbor algorithms on the top of large databases. Finally, generation of association rules can be performed in many different ways, depending on the amount of main memory available. There have been a growing number of papers on this subject at recent VLDB and SIGMOD conferences.

Database mining should learn from the general experience of DBMS field and follow one of the key DBMS paradigms [43]: building optimizing compilers for ad hoc queries and embedding queries in application programming interfaces. Thus, the focus should be on increasing *programmer productivity* for KDD application development.

Queries, however have to be much more general than SQL; similarly, the queried objects have to be far more complex than records (tuples) in relational database.

Chapter 2

Discovery of association rules

In this chapter we study the discovery of association rules, a simple but important case of frequent patterns. Association rules we introduced in [1] for the analysis of market baskets. We specify the problem formally in Section 2.1. In Section 2.2 we review how association rules can be generated when all frequent sets of items are given as input. In Section 2.3 we give an efficient method for the discovery of all frequent sets. Experiments with the method are described in Section 2.4. Extensions and other related work are reviewed in Section 2.7. In Section 2.5 we consider the problem of selecting interesting rules from the set of generated rules. Section 2.6 gives some simple theoretical results about the association rule finding problem.

2.1 Association rules

Given a collection of sets of items, association rules describe how likely various combinations of items are to occur together in the same sets. A typical application for association rules is in the analysis of the so called supermarket basket data: the goal is to find regularities in the customer behavior in terms of combinations of products that are purchased often together. The simple data model we consider is the following.

Definition 2.1 Given a set R, a *0/1 relation r over R* is a collection (or multiset) of subsets of R. The elements of R are called *items*, and the elements of r are called *rows*. The number of rows in r is denoted by $|r|$, and the *size* of r is $||r|| = \sum_{t \in r} |t|$. □

Example 2.2 In the domain of supermarket basket analysis, items represent the products in the stock. There could be items such as *beer*, *chips*, *diapers*, *milk*, *bread*, and so on. A row in a basket database then corresponds to the contents of a shopping basket: for each product type in the basket, the corresponding item is in the row. If a customer purchased milk and diapers, then there is a corresponding row $\{$*milk*, *diapers*$\}$ in the database. The

Row ID	Row
t_1	$\{A, B, C, D, G\}$
t_2	$\{A, B, E, F\}$
t_3	$\{B, I, K\}$
t_4	$\{A, B, H\}$
t_5	$\{E, G, J\}$

Figure 2.1: An example 0/1 relation r over the set $R = \{A, \ldots, K\}$.

quantity or price of items is not considered in this model, only the binary information whether a product was purchased or not.

The number of different items can be in the order of thousands, whereas typical purchases only contain at most dozens of items. When practical storage structures for such sparse databases are used, the physical database size closely corresponds to $||r|| = \sum_{t \in r} |t|$. □

We use letters $A, B, \ldots$ from the beginning of the alphabet to denote items. The set of all items is denoted by R, and other sets of items are denoted by letters from the end of the alphabet, such as X and Y. Calligraphic symbols, such as $\mathcal{S}$, denote collections of sets. Databases are denoted by lowercase letters such as r, and rows by letters t and u.

An interesting property of a set $X \subseteq R$ of items is how many rows contain it. This brings us to the formal definition of the term "frequent".

Definition 2.3 Let R be a set and r a 0/1 relation over R, and let $X \subseteq R$ be a set of items. The item set X *matches* a row $t \in r$, if $X \subseteq t$. The (multi) set of rows in r matched by X is denoted by $\mathcal{M}(X, r)$, i.e., $\mathcal{M}(X, r) = \{t \in r \mid X \subseteq t\}$. The *frequency* of X in r, denoted by $fr(X, r)$, is $\frac{|\mathcal{M}(X,r)|}{|r|}$. We write simply $\mathcal{M}(X)$ and $fr(X)$ if the database is unambiguous in the context. Given a *frequency threshold* $min\_fr \in [0, 1]$, the set X is *frequent*[1] if $fr(X, r) \geq min\_fr$. □

Example 2.4 Consider the 0/1 relation r over the set $R = \{A, \ldots, K\}$ in Figure 2.1. We have, for instance, $\mathcal{M}(\{A, B\}, r) = \{t_1, t_2, t_4\}$ and $fr(\{A, B\}, r) = 3/5 = 0.6$. The database can be viewed as a relational database over the schema $\{A, \ldots, K\}$, where $A, \ldots, K$ are 0/1-valued attributes, hence the name "0/1 relation". Figure 2.2 presents the database in this form. □

A set X is frequent if it matches at least a fraction min_fr of the rows in the database r. The frequency threshold min_fr is a parameter given by the

[1] In the literature, also the terms *large* and *covering* have been used for "frequent", and the term *support* for "frequency".

Row ID	A	B	C	D	E	F	G	H	I	J	K
t_1	1	1	1	1	0	0	1	0	0	0	0
t_2	1	1	0	0	1	1	0	0	0	0	0
t_3	0	1	0	0	0	0	0	0	1	0	1
t_4	1	1	0	0	0	0	0	1	0	0	0
t_5	0	0	0	0	1	0	1	0	0	1	0

Figure 2.2: The example 0/1 relation r in relational form over 0/1-valued attributes $\{A, \ldots, K\}$.

user and depends on the application. For notational convenience, we next introduce notations for collections of frequent sets.

Definition 2.5 Let R be a set, r a 0/1 relation over R, and *min_fr* a frequency threshold. The collection of frequent sets in r with respect to *min_fr* is denoted by $\mathcal{F}(r, \mathit{min\_fr})$,

$$\mathcal{F}(r, \mathit{min\_fr}) = \{X \subseteq R \mid \mathit{fr}(X, r) \geq \mathit{min\_fr}\},$$

or simply by $\mathcal{F}(r)$ if the frequency threshold is clear in the context. The collection of frequent sets of size l is denoted by $\mathcal{F}_l(r) = \{X \in \mathcal{F}(r) \mid |X| = l\}$. □

Example 2.6 Assume that the frequency threshold is 0.3. The collection $\mathcal{F}(r, 0.3)$ of frequent sets in the database r of Figure 2.1 is then $\{\{A\}, \{B\}, \{E\}, \{G\}, \{A, B\}\}$, since no other non-empty set occurs in more than one row. The empty set $\emptyset$ is trivially frequent in every 0/1 relation; we often ignore the empty set as a non-interesting case. □

We now move on and define association rules. An association rule states that a set of items tends to occur in the same row with another set of items. Associated with each rule are two factors: its confidence and frequency.

Definition 2.7 Let R be a set, r a 0/1 relation over R, and $X, Y \subseteq R$ sets of items. Then the expression $X \Rightarrow Y$ is an *association rule* over r. The *confidence* of $X \Rightarrow Y$ in r, denoted by $\mathit{conf}(X \Rightarrow Y, r)$, is $\frac{|\mathcal{M}(X \cup Y, r)|}{|\mathcal{M}(X, r)|}$. The *frequency* $\mathit{fr}(X \Rightarrow Y, r)$ of $X \Rightarrow Y$ in r is $\mathit{fr}(X \cup Y, r)$. We write simply $\mathit{conf}(X \Rightarrow Y)$ and $\mathit{fr}(X \Rightarrow Y)$ if the database is unambiguous in the context.

Given a *frequency threshold min_fr* and a *confidence threshold min_conf*, $X \Rightarrow Y$ *holds* in r if and only if $\mathit{fr}(X \Rightarrow Y, r) \geq \mathit{min\_fr}$ and $\mathit{conf}(X \Rightarrow Y, r) \geq \mathit{min\_conf}$. □

In other words, the confidence $\mathit{conf}(X \Rightarrow Y, r)$ is the conditional probability that a randomly chosen row from r that matches X also matches Y.

The frequency of a rule is the amount of positive evidence for the rule. For a rule to be considered interesting, it must be strong enough and common enough. The association rule discovery task [1] is now the following: given R, r, *min_fr*, and *min_conf*, find all association rules $X \Rightarrow Y$ that hold in r with respect to *min_fr* and *min_conf*, and such that X and Y are disjoint and non-empty.

Example 2.8 Consider, again, the database in Figure 2.1. Suppose we have frequency threshold $min\_fr = 0.3$ and confidence threshold $min\_conf = 0.9$. The only association rule with disjoint and non-empty left and right-hand sides that holds in the database is $\{A\} \Rightarrow \{B\}$. The frequency of the rule is $0.6 \geq min\_fr$, and the confidence is $1 \geq min\_conf$. The rule $\{B\} \Rightarrow \{A\}$ does not hold in the database as its confidence 0.75 is below *min_conf*. □

Note that association rules do not have monotonicity properties with respect to expansion or contraction of the left-hand side. If $X \Rightarrow Y$ holds, then $X \cup \{A\} \Rightarrow Y$ does not necessarily hold, since $X \cup \{A\} \Rightarrow Y$ does not necessarily have sufficient frequency or confidence. Or, if $X \cup \{A\} \Rightarrow Y$ holds, then $X \Rightarrow Y$ does not necessarily hold with sufficient confidence. Association rules are not monotone with respect to expansion of the right-hand side neither: if $X \Rightarrow Y$ holds, then $X \Rightarrow Y \cup \{A\}$ does not necessarily hold with sufficient frequency or confidence. Association rules are only monotone with respect to contraction of the right-hand side: if $X \Rightarrow Y \cup \{A\}$ holds, then $X \Rightarrow Y$ holds.

2.2 Rule generation

Association rules that hold in a 0/1 relation can be discovered in two phases [1]. First, find all frequent item sets $X \subseteq R$ and their frequencies. Then test separately for all $Y \subset X$ with $Y \neq \emptyset$ whether the rule $X \setminus Y \Rightarrow Y$ holds with sufficient confidence. Algorithm 2.9 (from [1]) uses this approach to generate all association rules that hold in the input database. The harder part of the problem, the task of finding the frequent sets, is considered in the following section. Note that indentation is used in the algorithms to specify the extent of loops and conditional statements.

Theorem 2.10 Algorithm 2.9 works correctly.

Proof First note that $conf(X \Rightarrow Y, r) = \frac{|\mathcal{M}(X \cup Y, r)|}{|\mathcal{M}(X, r)|} = \frac{fr(X \cup Y, r)}{fr(X, r)}$.

Clearly, all association rules $X \Rightarrow Y$ that are output by the algorithm hold in the input database r: $fr(X \Rightarrow Y) \geq min\_fr$ since $fr(X \cup Y) \geq min\_fr$ (line 2), and $conf(X \Rightarrow Y) \geq min\_conf$ (line 6).

All association rules $X \Rightarrow Y$ that hold in the input database r are also output by the algorithm. Since $fr(X \Rightarrow Y) \geq min\_fr$, also $fr(X \cup Y) \geq min\_fr$,

Algorithm 2.9
Input: A set R, a 0/1 relation r over R, a frequency threshold *min_fr*, and a confidence threshold *min_conf*.
Output: The association rules that hold in r with respect to *min_fr* and *min_conf*, and their frequencies and confidences.
Method:

```
1.  // Find frequent sets (Algorithm 2.14):
2.  compute F(r, min_fr) := {X ⊆ R | fr(X, r) ≥ min_fr};
3.  // Generate rules:
4.  for all X ∈ F(r, min_fr) do
5.      for all Y ⊂ X with Y ≠ ∅ do
6.          if fr(X)/fr(X \ Y) ≥ min_conf then
7.              output the rule X \ Y ⇒ Y, fr(X), and fr(X)/fr(X \ Y);
```

---

and $X \cup Y$ must be in $\mathcal{F}(r, min\_fr)$ (line 2). Then the possible rule $X \Rightarrow Y$ will be checked (lines 4 and 5). Since $conf(X \Rightarrow Y) \geq min\_conf$, the rule will be output (line 6). □

2.3 Finding frequent sets

Exhaustive search of frequent sets is obviously infeasible for all but the smallest sets R: the search space of potential frequent sets consists of the $2^{|R|}$ subsets of R. A more efficient method for the discovery of frequent sets can be based on the following iterative approach. For each $l = 1, 2, \ldots$, first determine a collection $\mathcal{C}_l$ of candidate sets of size l such that $\mathcal{F}_l(r) \subseteq \mathcal{C}_l$, and then obtain the collection $\mathcal{F}_l(r)$ of frequent sets by computing the frequencies of the candidates from the database.

For large data collections, the computation of frequencies from the database is expensive. Therefore it is useful to minimize the number of candidates, even at the cost of the generation phase. To generate a small but sufficient collection of candidates, observe the following properties of item sets. Obviously a subset of items is at least as frequent as its superset, i.e., frequency is monotone increasing with respect to contraction of the set. This means that for any sets X and Y of items such that $Y \subseteq X$, we have $\mathcal{M}(Y) \supseteq \mathcal{M}(X)$ and $fr(Y) \geq fr(X)$, and we have that if X is frequent then Y is also frequent. Proposition 2.11 takes advantage of this observation and gives useful information for candidate generation: given a set X, if any of the subsets of X is not frequent then X can be safely discarded from the candidate collection $\mathcal{C}_{|X|}$. The proposition also states that it actually suffices to know if all subsets one smaller than X are frequent or not.

Proposition 2.11 Let $X \subseteq R$ be a set. If any of the proper subsets $Y \subset X$ is not frequent then (1) X is not frequent and (2) there is a non-frequent subset $Z \subset X$ of size $|X| - 1$.

Algorithm 2.14
Input: A set R, a 0/1 relation r over R, and a frequency threshold *min_fr*.
Output: The collection $\mathcal{F}(r, \mathit{min\_fr})$ of frequent sets and their frequencies.
Method:

```
C_1 := {{A} | A ∈ R};
l := 1;
while C_l ≠ ∅ do
    // Database pass (Algorithm 2.22):
    compute F_l(r) := {X ∈ C_l | fr(X, r) ≥ min_fr};
    l := l + 1;
    // Apriori candidate generation (Algorithm 2.18):
    compute C_l := C(F_{l-1}(r));
for all l and for all X ∈ F_l(r) do output X and fr(X, r);
```

Proof Claim (1) follows directly from the observation that if X is frequent then all subsets $Y \subset X$ are frequent. The same argument applies for claim (2): for any $Y \subset X$ there exists Z such that $Y \subseteq Z \subset X$ and $|Z| = |X| - 1$. If Y is not frequent, then Z is not frequent. □

Example 2.12 If we know that

$$\mathcal{F}_2(r) = \{\{A, B\}, \{A, C\}, \{A, E\}, \{A, F\}, \{B, C\}, \{B, E\}, \{C, G\}\},$$

then we can conclude that $\{A, B, C\}$ and $\{A, B, E\}$ are the only possible members of $\mathcal{F}_3(r)$, since they are the only sets of size 3 whose all subsets of size 2 are included in $\mathcal{F}_2(r)$. Further on, we know that $\mathcal{F}_4(r)$ must be empty. □

We now use Proposition 2.11 to define a candidate collection of sets of size $l + 1$ to consist of those sets that can possibly be frequent, given the frequent sets of size l. This definition is the heart of the Apriori algorithm.

Definition 2.13 Given a collection $\mathcal{F}_l(r)$ of frequent sets of size l, the *(Apriori) candidate collection* generated from $\mathcal{F}_l(r)$, denoted by $\mathcal{C}(\mathcal{F}_l(r))$, is the collection of sets of size $l + 1$ that can possibly be frequent:

$$\mathcal{C}(\mathcal{F}_l(r)) = \{X \subseteq R \mid |X| = l + 1 \text{ and } Y \in \mathcal{F}_l(r) \text{ for all } Y \subseteq X, |Y| = l\}.$$

□

We now finally give Algorithm 2.14 (Apriori) that finds all frequent sets. The subtasks of the algorithm, for which only specifications are given, are described in detail in following subsections.

Theorem 2.15 Algorithm 2.14 works correctly.

Proof We show by induction on l that $\mathcal{F}_l(r)$ is computed correctly for all l. For $l = 1$, the collection $\mathcal{C}_l$ contains all sets of size one (line 1), and collection $\mathcal{F}_l(r)$ contains then correctly exactly those that are frequent (line 5). For $l > 1$, assume $\mathcal{F}_{l-1}(r)$ has been correctly computed. Then we have $\mathcal{F}_l(r) \subseteq \mathcal{C}_l = \mathcal{C}(\mathcal{F}_{l-1}(r))$ by Proposition 2.11 (line 8). Collection $\mathcal{F}_l(r)$ is then correctly computed to contain the frequent sets (line 5).

Note also that the algorithm computes $\mathcal{F}_{|X|}(r)$ for each frequent set X: since X is frequent, there are frequent sets—at least the subsets of X—of sizes 1 to $|X|$, so the ending condition $\mathcal{C}_l = \emptyset$ is not true for $l \leq |X|$. □

From Proposition 2.11 it follows that Definition 2.13 gives a sufficiently large candidate collection. Theorem 2.16, below, shows that the definition gives the smallest possible candidate collection in general.

Theorem 2.16 For any collection $\mathcal{S}$ of subsets of X of size l, there exists a 0/1 relation r over R and a frequency threshold *min_fr* such that $\mathcal{F}_l(r) = \mathcal{S}$ and $\mathcal{F}_{l+1}(r) = \mathcal{C}(\mathcal{S})$.

Proof We use a simple trick: set $r = \mathcal{S} \cup \mathcal{C}(\mathcal{S})$ and $min\_fr = 1/|r|$. Now all sets in $\mathcal{S}$ and $\mathcal{C}(\mathcal{S})$ are frequent, i.e., $\mathcal{S} \subseteq \mathcal{F}_l(r)$ and $\mathcal{C}(\mathcal{S}) \subseteq \mathcal{F}_{l+1}(r)$. Further on, $\mathcal{F}_{l+1}(r) \subseteq \mathcal{C}(\mathcal{S})$ since there are no other sets of size $l+1$ in r. To complete the proof we show by contradiction that $\mathcal{F}_l(r) \subseteq \mathcal{S}$. Assume that $Y \in \mathcal{F}_l(r)$ is not in $\mathcal{S}$. Then there must be $X \in \mathcal{C}(\mathcal{S})$ such that $Y \subset X$. However, by Definition 2.13 all subsets of X of size l are in $\mathcal{S}$. □

In candidate generation, more information can be used than just whether all subsets are frequent or not, and this way the number of candidates can be further reduced. Sometimes even the exact frequency of a set can be inferred.

Example 2.17 Assume sets $\{A, B\}$, $\{A, C\}$, $\{A, D\}$, $\{B, C\}$, and $\{B, D\}$ are frequent. Definition 2.13 gives $\{A, B, C\}$ and $\{A, B, D\}$ as candidates for $l = 3$, and Theorem 2.16 shows that such a 0/1 relation exists where $\{A, B, C\}$ and $\{A, B, D\}$ are indeed frequent.

If, however, we know that $fr(\{A, B, C\}) = fr(\{A, B\})$, then we can infer that $fr(\{A, B, D\}) < min\_fr$. Intuitively, item C partitions the database: all of $\{A, B\}$ occurs with C, but less than *min_fr* of D occurs with C, since $fr(\{C, D\}) < min\_fr$, and therefore $\{A, B, D\}$ cannot be frequent. If the frequency of $\{A, B, C\}$ is computed first, it is not necessary to compute the frequency of $\{A, B, D\}$ from the database at all.

We have a slightly different situation if $fr(\{A, B\}) = fr(\{A\})$. Then we have $\mathcal{M}(\{A\}) \subseteq \mathcal{M}(\{B\})$, so $\mathcal{M}(\{A, C\}) = \mathcal{M}(\{A, B, C\})$ and $fr(\{A, C\}) = fr(\{A, B, C\})$. Thus the frequency $fr(\{A, B, C\})$ needs not to be computed from the database. □

An algorithm such as 2.14 could, in principle, take advantage of situations similar to the above examples. Such situations do not, however, occur

Algorithm 2.18
Input: A lexicographically sorted array $\mathcal{F}_l(r)$ of frequent sets of size l.
Output: $\mathcal{C}(\mathcal{F}_l(r))$ in lexicographical order.
Method:

1. **for** all $X \in \mathcal{F}_l(r)$ **do**
2.   **for** all $Y \in \mathcal{F}_l(r)$ such that $X < Y$ and X and Y share their $l-1$ lexicographically first items **do**
3.   **for** all $Z \subset (X \cup Y)$ such that $|Z| = l$ **do**
4.   **if** Z is not in $\mathcal{F}_l(r)$ **then** continue with the next Y at line 2;
5.   output $X \cup Y$;

---

frequently, and the effort saved can be less than the effort put into finding these cases. Furthermore, Algorithm 2.14 combines the computations of the frequencies of all candidate sets of size l to one pass; the number of database passes would seldom be reduced.

2.3.1 Apriori candidate generation

We now consider the computation of candidate collections as defined in Definition 2.13. The trivial method to compute the candidate collection $\mathcal{C}(\mathcal{F}_l(r))$ is to check for each possible set of size $l+1$ whether the definition holds, i.e., if all its $l+1$ subsets of size l are frequent. A more efficient way is to first compute potential candidates as unions $X \cup Y$ of size $l+1$ such that X and Y are frequent sets of size l, and then to check the rest of their subsets of size l. Algorithm 2.18 presents such a method. For efficiency reasons, it is assumed that both item sets and collections of item sets are stored as arrays, sorted in the lexicographical order. We write $X < Y$ to denote that X precedes Y in the lexicographical order.

Theorem 2.19 Algorithm 2.18 works correctly.

Proof First we show that the collection of potential candidates $X \cup Y$ considered by the algorithm is a superset of $\mathcal{C}(\mathcal{F}_l(r))$. Given a set W in $\mathcal{C}(\mathcal{F}_l(r))$, consider the subsets of W of size l, and denote by X' and Y' the first and the second subset in the lexicographical order, respectively. Then X' and Y' share the $l-1$ lexicographically first items of W. Since W is a valid candidate, X' and Y' are in $\mathcal{F}_l(r)$. In the algorithm, X iterates over all sets in $\mathcal{F}_l(r)$, and at some phase we have $X = X'$. Now note that every set between X' and Y' in the lexicographical ordering of $\mathcal{F}_l(r)$ must share the same $l-1$ lexicographically first items. Thus we have $Y = Y'$ in some iteration while $X = X'$. Hence we find a superset of the collection of all candidates. Finally, a potential candidate is correctly output if and only if all of its subsets of size l are frequent (line 4). $\square$

The time complexity of Algorithm 2.18 is polynomial in the size of the collection of frequent sets and it is independent of the database size.

Theorem 2.20 Algorithm 2.18 can be implemented to run in time $\mathcal{O}(l^2 \, |\mathcal{F}_l(r)|^2 \log |\mathcal{F}_l(r)|)$.

Proof The outer loop (line 1) and the inner loop (line 2) are both iterated $\mathcal{O}(|\mathcal{F}_l(r)|)$ times. Given X and Y, the conditions on line 2 can be tested in time $\mathcal{O}(l)$.[2] On line 4, the remaining $l-1$ subsets need to be checked. With binary search, a set of size l can be located from $\mathcal{F}_l(r)$ in time $\mathcal{O}(l \log |\mathcal{F}_l(r)|)$. The output on line 5 takes time $\mathcal{O}(l)$ for each potential candidate. The total time complexity is thus $\mathcal{O}(|\mathcal{F}_l(r)|^2(l + (l-1)l \log |\mathcal{F}_l(r)| + l)) = \mathcal{O}(l^2 \, |\mathcal{F}_l(r)|^2 \log |\mathcal{F}_l(r)|)$. □

The upper bound of Theorem 2.20 is met when $l = 1$: all pairs of frequent sets of size 1 are created. After that the number of iterations of the inner loop on line 2 is typically only a fraction of $|\mathcal{F}_l(r)|$.

Instead of only computing $\mathcal{C}(\mathcal{F}_l(r))$, several successive families $\mathcal{C}(\mathcal{F}_l(r))$, $\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r)))$, $\mathcal{C}(\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r))))$, ... can be computed and then checked in a single database pass. This trades off a reduction in the number of database passes against an increase in the number of candidates, i.e., database processing against main memory processing. Candidates of size $l+2$ are computed assuming that all candidates of size $l+1$ are in fact frequent, and therefore $\mathcal{C}(\mathcal{F}_{l+1}(r)) \subseteq \mathcal{C}(\mathcal{C}(\mathcal{F}_l(r)))$. Several candidate families can be computed by several calls to Algorithm 2.18.

Generating several candidate families is useful when the overhead of generating and testing the extra candidates $\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r))) \setminus \mathcal{C}(\mathcal{F}_{l+1}(r))$ is less than the effort of a database pass. Unfortunately, estimating the volume of extra candidates is in general difficult. The obviously useful situations are when $|\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r)))|$ is small.

Example 2.21 Assume

$$\begin{aligned}\mathcal{F}_2(r) &= \{\{A,B\},\{A,C\},\{A,D\},\{A,E\},\\ &\quad\ \{B,C\},\{B,D\},\{B,G\},\{C,D\},\{F,G\}\}.\end{aligned}$$

Then we have

$$\begin{aligned}\mathcal{C}(\mathcal{F}_2(r)) &= \{\{A,B,C\},\{A,B,D\},\{A,C,D\},\{B,C,D\}\},\\ \mathcal{C}(\mathcal{C}(\mathcal{F}_2(r))) &= \{\{A,B,C,D\}\}, \text{ and}\\ \mathcal{C}(\mathcal{C}(\mathcal{C}(\mathcal{F}_2(r)))) &= \emptyset.\end{aligned}$$

It would be practical to evaluate the frequency of all 5 candidates in a single pass. □

[2] Actually, the values of Y can be determined more efficiently with some extra book-keeping information stored every time candidates are generated. A more detailed method using this idea is presented in Section 4.2.2.

Algorithm 2.22
Input: A set R, a 0/1 relation r over R, a candidate collection $\mathcal{C}_l \supseteq \mathcal{F}_l(r, min\_fr)$, and a frequency threshold min_fr.
Output: The collection $\mathcal{F}_l(r, min\_fr)$ of frequent sets and their frequencies.
Method:

1. // Initialization:
2. **for** all $A \in R$ **do** $A.is\_contained\_in := \emptyset$;
3. **for** all $X \in \mathcal{C}_l$ and **for** all $A \in X$ **do**
4. $A.is\_contained\_in := A.is\_contained\_in \cup \{X\}$;
5. **for** all $X \in \mathcal{C}_l$ **do** $X.freq\_count := 0$;
6. // Database access:
7. **for** all $t \in r$ **do**
8.   **for** all $X \in \mathcal{C}_l$ **do** $X.item\_count := 0$;
9.   **for** all $A \in t$ **do**
10.   **for** all $X \in A.is\_contained\_in$ **do**
11. $X.item\_count := X.item\_count + 1$;
12.   **if** $X.item\_count = l$ **then** $X.freq\_count := X.freq\_count + 1$;
13. // Output:
14. **for** all $X \in \mathcal{C}_l$ **do**
15.   **if** $X.freq\_count/|r| \geq min\_fr$ **then** output X and $X.freq\_count/|r|$;

2.3.2 Database pass

We turn now to the database pass of Algorithm 2.14. Algorithm 2.22 presents a method for computing the frequencies of candidates from a database.

For each item $A \in R$ we maintain a list $A.is\_contained\_in$ of candidates that contain A. For each candidate X we maintain two counters. Variable $X.freq\_count$ is used to count the number of rows that X matches, whereas variable $X.item\_count$ records, for the current row, the number of items of X.

Theorem 2.23 Algorithm 2.22 works correctly.

Proof We need to show that the frequency of each candidate $X \in \mathcal{C}_l$ is computed correctly; obviously the correct sets are then output (line 15). The frequency counters are initialized to zero on line 5. The claim that remains to be shown is that for every X in $\mathcal{C}_l$, the frequency counter is increased on line 12 once for each row t such that $X \subseteq t$.

First consider the initialization phase. After lines 2 and 4, for each $A \in R$ we have $A.is\_contained\_in = \{X \mid X \in \mathcal{C}_l \text{ and } A \in X\}$. Consider now lines 8 to 11: given a row t, these lines compute for each set X in $\mathcal{C}_l$ the size of the intersection $t \cap X$ in the variable $X.item\_count$. The value of $X.item\_count$ reaches the size of X (lines 11 and 12) if and only if $X \subseteq t$, in which case the frequency counter is increased by one. $\square$

The time complexity of the algorithm is linear in the size of the database and in the product of the number of rows and the number and size of candidates.

Theorem 2.24 The time complexity of Algorithm 2.22 is $\mathcal{O}(||r|| + l\,|r|\,|\mathcal{C}_l| + |R|)$.

Proof The time complexity of initialization is $\mathcal{O}(|R| + l\,|\mathcal{C}_l| + |\mathcal{C}_l|)$ (lines 2–5). The time complexity of reading the database is $\mathcal{O}(||r||)$ (line 7). Initialization of candidates for all rows takes time $\mathcal{O}(|r|\,|\mathcal{C}_l|)$ (line 8). For each row, each candidate is updated at most l times; the worst-case time complexity for computing the frequencies is thus $\mathcal{O}(l\,|r|\,|\mathcal{C}_l|)$. Output takes time $\mathcal{O}(l\,|\mathcal{C}_l|)$. The time complexity for the whole database pass is thus $\mathcal{O}(|R| + l\,|\mathcal{C}_l| + |\mathcal{C}_l| + ||r|| + |r|\,|\mathcal{C}_l| + l\,|r|\,|\mathcal{C}_l| + l\,|\mathcal{C}_l|) = \mathcal{O}(||r|| + l\,|r|\,|\mathcal{C}_l| + |R|)$. □

There are a number of practical improvements to the algorithm. As an example, to determine that an infrequent set $X \subseteq R$ really is not frequent, one has to read at least a fraction $1 - \mathit{min\_fr}$ of the rows of the database. With a relatively large frequency threshold $\mathit{min\_fr}$ it could be practical to check and discard a candidate if there are less rows left than are needed for the candidate to be frequent. In the best case, all candidates could be discarded when less than a fraction $\mathit{min\_fr}$ of the database rows are left. Thus the best possible saving is less than a fraction $\mathit{min\_fr}$ of the original time.

2.4 Experiments

To illustrate the behavior of the methods in practice, we present experimental results with a course enrollment database of the Department of Computer Science at the University of Helsinki. The database consists of registration information of 4 734 students: there is a row per student, and the items are the courses offered by the department. The "shopping basket" of a student contains the courses the student has enrolled to during his or her stay at the university. The number of courses is 127, and a row contains on the average 4.3 courses. The experiments in this and the following chapters have been run on a PC with a 90 MHz Pentium processor and 32 MB of main memory, under the Linux operating system. The data collections resided in flat text files.

Table 2.1 gives an overview of the amount of frequent sets of different sizes found with frequency thresholds 0.01–0.2. The table shows that the number and the size of frequent sets increases quickly with a decreasing frequency threshold. Frequency threshold 0.2 corresponds to a set of at least 947 students, and 0.01 to 48 students, i.e., with threshold 0.01 all rules that hold for at least 48 students are found.

An overview of various characteristics of the same experiments is given in Table 2.2. On the top, the table shows the number of candidates and the time it took to generate them. Next, the number of frequent sets, their maximum size, and the time to compute the frequencies from the database are shown. The row "match ratio" shows how large fraction of candidate sets

Size	Frequency threshold					
	0.200	0.100	0.075	0.050	0.025	0.010
1	6	13	14	18	22	36
2	1	21	48	77	123	240
3	0	8	47	169	375	898
4	0	1	12	140	776	2 203
5	0	0	1	64	1 096	3 805
6	0	0	0	19	967	4 899
7	0	0	0	2	524	4 774
8	0	0	0	0	165	3 465
9	0	0	0	0	31	1 845
10	0	0	0	0	1	690
11	0	0	0	0	0	164
12	0	0	0	0	0	21
13	0	0	0	0	0	1

Table 2.1: Number of frequent sets of each size with different frequency thresholds.

was actually frequent. The lower parts of the table then show the number of rules generated from the frequent sets with different confidence thresholds, and the time it took to generate them.

The table shows that for reasonable frequency and confidence thresholds the time it takes to generate candidates is smaller by a magnitude than the time for the database pass, although our database is small. For large databases the candidate generation time can, in practice, be ignored. As was already noted, the number of frequent sets grows quickly when the threshold is lowered. The number of candidate sets grows almost identically, but the number of rules explodes: with frequency threshold 0.01 and confidence threshold 0.5 there are 1.8 million rules.

The table shows that the candidate generation method works well: the number of candidates is quite close to the number of frequent sets, especially with lower frequency thresholds. Table 2.3 shows in more detail the experiment with $min\_fr = 0.025$. In the first couple of iterations there is not much combinatorial information available, and subsequently there are over 100 non-frequent candidates. After that the candidate generation method works very effectively, and the match ratio is at least 90 %.

The execution times of the database passes of our experiments are roughly linear in the total number of items in the candidate collection. This is consistent with the result $\mathcal{O}(||r|| + l\,|r|\,|\mathcal{C}_l| + |R|)$ of Theorem 2.24. In our case $||r||$ is a fairly small constant since the database is small, and $|R|$ is even less significant. We tested the scale-up properties of the algorithms by producing 2, 4, 8, and 16 fold copies of the data set. Figure 2.3 presents the relative

	Frequency threshold					
	0.200	0.100	0.075	0.050	0.025	0.010
Candidate sets:						
Count	142	223	332	825	4 685	24 698
Generation time (s)	0.1	0.1	0.2	0.2	1.1	10.2
Frequent sets:						
Count	7	43	122	489	4 080	23 041
Maximum size	2	4	5	7	10	13
Database pass time (s)	0.7	1.9	3.5	10.3	71.2	379.7
Match ratio	5 %	19 %	37 %	59 %	87 %	93 %
Rules ($min\_conf = 0.9$):						
Count	0	3	39	503	15 737	239 429
Generation time (s)	0.0	0.0	0.1	0.4	46.2	2 566.2
Rules ($min\_conf = 0.7$):						
Count	0	40	193	2 347	65 181	913 181
Generation time (s)	0.0	0.0	0.1	0.8	77.4	5 632.8
Rules ($min\_conf = 0.5$):						
Count	0	81	347	4 022	130 680	1 810 780
Generation time (s)	0.0	0.0	0.1	1.1	106.5	7 613.62

Table 2.2: Characteristics of experiments with different frequency thresholds.

	Candidates		Frequent sets		Match
Size	Count	Time (s)	Count	Time (s)	ratio
1	127	0.05	22	0.26	17 %
2	231	0.04	123	1.79	53 %
3	458	0.04	375	5.64	82 %
4	859	0.09	776	12.92	90 %
5	1 168	0.21	1 096	18.90	94 %
6	1 058	0.30	967	18.20	91 %
7	566	0.24	524	9.69	93 %
8	184	0.11	165	3.09	90 %
9	31	0.04	31	0.55	100 %
10	3	0.01	1	0.15	33 %
11	0	0.00			
Total	4 685	1.13	4 080	71.19	87 %

Table 2.3: Details of candidates and frequent sets for each size with $min\_fr = 0.025$.

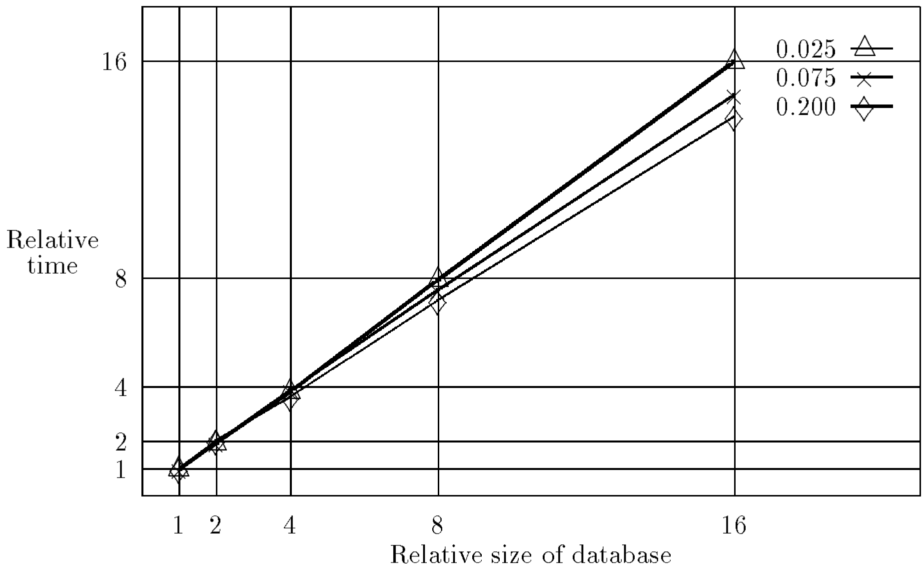


Figure 2.3: Results of scale-up tests.

running times for frequency thresholds 0.2, 0.075, and 0.025, including both candidate generation and database passes. As expected, the execution time is linear with respect to the number of rows in the database. Note also that the constant term of the candidate generation time does not essentially show in the figures.

2.5 Rule ranking

The number of association rules can be large, as was demonstrated by the experimental results with the course enrollment database in Section 2.4. Obviously, it would be useful to be able to rank the resulting rules into an approximate order of interest. We now briefly look at some possible ways to rank association rules.

The most simple choices are the confidence and frequency of rules. These are already known, so no extra computation is required. Confidence and frequecy also have clear interpretations. One of the major problems that remains is the redundancy of rules. For instance, if the frequency of A is high then the confidence of any rule $X \Rightarrow \{A\}$ is likely to be high (roughly equal to the frequency of A) even when X and A occur independently of each other. If the confidence of a rule $X \Rightarrow Y$ is close to the frequency of Y then X has no effect on Y, and the rule can hardly be interesting.

"Significance" Let us first consider a significance measure that closely corresponds to the statistical significance level.

As a simple example, consider an association rule $\{A\} \Rightarrow \{B\}$. Consider the frequencies $fr(A)$ and $fr(B)$ as given, and the size of $\mathcal{M}(\{A,B\})$ as a random variable $\mathbf{X}$. Recall that the confidence is obtained from $|\mathcal{M}(\{A,B\})|$ by just dividing it by $|\mathcal{M}(A)|$. Assuming that A and B are independent, $\mathbf{X}$ has a binomial probability distribution: you have $|\mathcal{M}(A)|$ Bernoulli trials each with success probability $fr(B)$.

The actual size of $\mathcal{M}(\{A,B\})$ is measured in the database (and it determines the confidence). Given the distribution of $\mathbf{X}$, one can compute the probability that the value of $\mathbf{X}$ deviates from its expected value at most as much as the size of $\mathcal{M}(\{A,B\})$ does. This probability can be used as a measure of significance or deviation from expectation for the rule. A value close to 1 means that the deviations would be smaller almost certainly if A and B were independent, i.e., there is strong evidence for dependence between A and B. A value close to 0 means that the deviation from the expected value could be caused by common random effects.

Couple of notes are in order. First, a significance value of s would be usually expressed as $1-s$ by statisticians. Second, you should be careful when talking about (statistical) significance: the statistical tradition is to formulate a hypothesis, test it, and obtain some significance measure, whereas in the case of association rules you test a huge number of hypothesis, and it is likely that some will look significant by random. There are methods like Boneferroni adjustment for taking this into account, but it is still not quite clear what the whole picture is if you compare the situation with more traditional statistical settings. So it is maybe safer to just talk about a significance measure, or a measure of deviation or interestingness.

J-measure The *J-measure*, introduced in [80], is an information theoretic measure for ranking rules. It measures, individually for each association rule $X \Rightarrow Y$, the information gained about Y over the situation where only the frequency of Y is known.

Expressed in terms of association rules, the J-measure is defined as

$$\begin{aligned} J(X \Rightarrow Y) \;=\; & fr(X)\left(conf(X \Rightarrow Y) \log\left(\frac{conf(X \Rightarrow Y)}{fr(Y)}\right) + \right. \\ & \left. (1 - conf(X \Rightarrow Y)) \log\left(\frac{1 - conf(X \Rightarrow Y)}{1 - fr(Y)}\right)\right). \end{aligned}$$

The first term, $fr(X)$, favours frequent rules, and the second term is known as the cross-entropy, the information gain of the rule. The J-measure has unique properties as a rule information measure and is in a certain sense a special case of Shannon's mutual information. From a practical viewpoint, the measure provides a useful and sound method for ranking rules in a balanced manner with respect to both the frequency and confidence of rules.

2.6 Theoretical analyses

The algorithms presented in the previous sections perform quite well in practice. For large databases and meaningful frequency and confidence thresholds the running time is dominated by the complexity $\mathcal{O}(||r|| + l\,|r|\,|\mathcal{C}_l| + |R|)$ of the database passes (Theorem 2.24).

2.6.1 A lower bound

The quantity $|\mathcal{C}|$ can be exponential in the number of items, as all itemsets can be frequent. If there are only a few frequent sets, the above algorithms still investigate several candidates. Next we show that this is to some degree inevitable. We give an information-theoretic lower bound for finding one association rule in a restricted model of computation where the only way of getting information from a database is by asking questions of the form "is the set X frequent". This model is realistic in the case the database is large and stored using a database system, and the model is also quite close to the one underlying the design of the original algorithm [1].

Theorem 2.25 Assume the database has $m = |R|$ items. In the worst case one needs at least

$$\log \binom{m}{h} \approx h \log(m/h)$$

questions of the form "is the set X frequent" to locate one maximal frequent set, where h is the size of the frequent set.

Proof Consider a database with exactly 1 maximal frequent set of size h. There are $\binom{m}{h}$ different possible answers to the problem of finding the maximal frequent set. Each question of the form "is the set X frequent" provides at most 1 bit of information. □

This lower bound is not optimal for small values of h. For example, assume that there is exactly one frequent set of size 1. Then any algorithm for finding this set has to use at least $\Theta(m)$ queries of the above type. However, the bound is fairly tight for larger values of h.

Loveland [57] has considered the problem of finding "critical sets". Given a function $f : \mathcal{P}(R) \to \{0, 1\}$ that is *upwards monotone* (i.e., if $f(Y) = 1$ and $Y \subseteq X$, then $f(X) = 1$), a set X is *critical* if $f(X) = 1$, but $f(Z) = 0$ for all subsets Z of X. Thus maximal frequent itemsets are complements of critical sets of the upwards monotone function

$$f(Y) = \begin{cases} 0 & \text{if } R \setminus Y \text{ is frequent} \\ 1 & \text{otherwise.} \end{cases}$$

For example for $h = m/2$, the lower bound above matches exactly the upper bound provided by one of Loveland's algorithms.

2.6.2 Probabilistic analysis of random databases

The number of frequent sets is an important factor influencing the running times of the algorithms. We now show that in one model of random databases all frequent itemsets have small size.

Consider a randomly generated database $r = \{t_1, \ldots, t_n\}$ over items $R = \{A_1, A_2, \ldots, A_m\}$; assume that each row t_k of the database contains any item A_j with probability q, and assume that the entries are independent. Then the probability that t_k contains A_j for all A_j in a given itemset X is q^h, where $h = |X|$. The number x of such rows has a binomial distribution with parameters n and q^h.

The Chernoff bounds [5, 36] state that for all $a > 0$ we have

$$Pr[x > nq^h + a] < e^{-2a^2/n}.$$

We thus obtain

$$Pr[x > sn] = Pr[x > nq^h + n(s - q^h)] < e^{-2n(s-q^h)^2}$$

where n is the number of rows and $s = \mathit{min\_fr}$ is the frequency threshold. Thus the expected number of frequent itemsets of size h is bounded by $m^h e^{-2n(s-q^h)^2}$, where m is the number of items. This is less than 1 provided $s > \sqrt{(h \ln m)/n} + q^h$. For large databases and thus for large n the first term is very small. Hence if $s > q^h$, the expected number of frequent sets of size h is small. (For $s = \mathit{min\_fr} = 0.01$ and $q = 0.1$, this means $h \geq 2$; for $s = 0.0001$ and $q = 0.1$, this means $h \geq 4$.) Thus a random database typically has only very few frequent itemsets. Of course, databases occurring in practice are not random.

2.6.3 Analysis of sampling

The running times of the algorithms depended linearly on the size of the database. One possibility of lowering this factor is to use only a sample of transactions.

We postpone the discussion until Chapter 9, where we show that relatively small samples give good approximations of frequencies and give a sampling-based method for finding association rules in one or two passes over the database.

2.7 Extensions and bibliographic notes

Since their introduction in 1993 [1], association rules have been researched a lot. In this section we discuss related work, some of which regards extensions to the basic framework of association rules. More remotely related research is discussed in Chapter 8.

Candidate generation Definition 2.13 and Algorithm 2.14 for candidate generation were presented independently in [3, 62]. We touch this subject again in Chapter 4, where we present in detail an algorithm that can deal, in addition to sets, also with multisets and ordered sets.

The frequent set discovery method presented in [1] also works in a generate and test fashion, but the candidate generation is quite different. An essential difference is that in [1] both candidate generation and evaluation are performed during the database pass. Most importantly, however, the method fails to take advantage of Proposition 2.11 and it generates candidates that cannot be frequent. Also in experimental comparisons the method of [1] has been shown to perform worse than the algorithms of this section [2, 3, 62].

Association rule generation The confidence of association rules is monotone decreasing with respect to moving items from the left-hand side of the rule to the right-hand side. This property can be used to reduce the number of potential rules to be tested in the rule generation in Algorithm 2.9. The idea is that for a set $X \in \mathcal{F}(r)$, the rule $X \setminus Y \Rightarrow Y$ is only tested if all rules $X \setminus Z \Rightarrow Z$ with $Z \subset Y$ have high enough confidence. Actually, the candidate generation Algorithm 2.18 can be used to construct candidates for rule right-hand sides [3].

Database pass For large databases it is important to minimize the database activity, i.e., improve on Algorithm 2.22. Matching a candidate X against a row in the database in the straightforward way results in repeating work that has been done previously. Namely, the subsets of X have been matched against the row already in the previous passes. Instead of starting from the scratch again, information about the matches of the subsets can be utilized. In particular, a candidate set X matches exactly those rows that are matched by any two subsets of size $|X| - 1$ [3]. One can create a new temporary 0/1 relation with an item for each candidate, and fill in the database during the database pass. Given a candidate X in the next pass, only two items representing subsets of X from the previous pass are needed to determine a match, and the old database is not needed at all. There is a trade-off: while matching is faster when reusing results, the size $\sum_{X \in \mathcal{C}_l} |\mathcal{M}(X)|$ of the temporary database may be even larger than the original database. Such reuse does not usually pay off in the first iterations: there are many candidates and they have high frequencies.

An alternative strategy for the database pass is to use inverted structures [40, 77]. The idea here is to organize the storage by the items of the original database rather than by its rows. The information per an item A is represented by the set $\mathcal{M}(\{A\})$ of (the identifiers of) the rows that contain A. With inverted structures, the database pass consists of computing intersections between these sets: the set $\mathcal{M}(X)$ of rows containing X is the intersection of the sets $\mathcal{M}(\{A\})$ for all A in X. For $|\mathcal{C}_l|$ candidates with l items each, the use of inverted structures results in somewhat smaller asymptotic worst-case time complexity $\mathcal{O}(|r|\,l\,|\mathcal{C}_l|)$ than the bound $\mathcal{O}(||r||+|r|\,l\,|\mathcal{C}_l|+|R|)$ of not using inverted structures. The main difference is that when using inverted structures, the non-frequent items need not be read at all. The above mentioned idea of reusing results from previous database passes can be implemented efficiently by storing the intersections representing frequent sets [40]. The size of the temporary database consisting of these inverted structures is then $\sum_{X \in \mathcal{F}_l(r)} |\mathcal{M}(X)|$.

An efficient method for the discovery of frequent sets takes advantage of the fact that with small databases the database needs to be read only once from the disk and can then remain in main memory. In the Partition method [77], a database too large to fit in main memory is partitioned, and each partition is analyzed separately in main memory. The first database pass consists of identifying in each part the collection of all locally frequent sets. For the second pass, the union of the collections of locally frequent sets is used as the candidate set. The first pass is guaranteed to locate a superset of the collection of frequent item sets; the second pass is needed to merely compute the frequencies of the sets.

Hashing has been used at least for two tasks in the discovery of frequent sets. In [3], hashing is used during the database pass to efficiently determine the collection of candidates that match a row. The method of [73], in turn, uses hashing to identify and prune non-frequent candidates before the database pass.

There is also a modification of the original algorithm of [1] that uses SQL for the discovery of frequent sets [42].

Item hierarchies Association rule algorithms have been generalized to work with items arranged to hierarchies or taxonomies [37, 40, 81]. Concept hierarchies exist often for the items: for instance, in the supermarket environment we know that *Samuel Adams* is a *beer* brand, that *beer* is a *beverage*, and so on. The idea is now to search for rules on several levels in the hierarchy, such as *beer* $\Rightarrow$ *chips* and *Samuel Adams* $\Rightarrow$ *chips*. The former rule gives useful information about beers in general, while the latter one may be interesting if its confidence differs significantly from the first one.

Non-binary data In the basic setting, association rules are found between sets of items. It would be useful to be able to search for associations between values of attributes in more general. Association rules between discrete values of different attributes can be found in a straightforward way by considering the (attribute, value) pairs as items. The number of items is then the sum of the sizes of the domains of all attributes. The candidate generation method can be modified so that it does not generate internally inconsistent candidate sets that contain items derived from the same attribute. Association rules for numeric attributes are considered in [27, 82, 88]. In addition to associations between single values, as above, the authors develop methods for automatic selection of useful value intervals to be used in the sets and rules.

Chapter 3

An example problem: Alarm correlation

In this part we consider a case-study in KDD: the problem of analyzing alarms from a telecommunication network. Telecommunication networks are large systems consisting of such complex components as switches, base stations, radio links, etc. Network elements produce large amounts of alarms about the faults in a network. Fully employing this valuable data is, however, difficult, due to the high volume and the fragmented nature of the information.

This part consists of 4 chapters. In this chapter we introduce the problem and outline one possible framework for knowledge discovery in alarm databases. In Chapter 4 we give a full and domain independent formulation for the framework and present algorithms for the task. In Chapter 5 we present an alternative approach and algorithms. The ideas have been implemented in a system called TASA, for Telecommunication Alarm Sequence Analyzer. We conclude this part in Chapter 6, where we look briefly at the KDD process in this particular case.

3.1 Fault management problems

Telecommunication networks are growing fast, and at the same time the task of identifying and correcting faults is becoming more difficult. The task is critical, since faults that interfere with the services offered by the network are costly for the operator. The quality of service plays also an important role in the growing competition between operators.

Telecommunication networks and alarms A telecommunication network can be viewed as consisting of a number of interconnected components: switches, transmission equipment, etc. Each component can in its turn contain several subcomponents. The number of components depends on the abstraction level used in viewing the system; a network operated by a local telephone company can be considered to contain 10–1000 components. We use the term *network element* to refer to the components under consideration.

An *alarm* is a message emitted by a network element, typically when a problem is encountered. Telecommunication systems are more or less fault tolerant, so the problem is not necessarily visible to the users. So faults are reported by network elements, and typically before the users are affected, but unfortunately a network element has a very narrow view to the network. A network element can therefore only report the symptoms of the fault from its limited viewpoint. On the other hand, one fault can result in a number of different alarms from several network elements.

Example 3.1 An actual alarm, as shown to the network manager, typically looks something like this:

```
1234  EL1  BTS  940926  082623  A1  Channel missing
```

The first field identifies the type of the alarm and the last field is the name of the alarm type. The second and third fields, `EL1` and `BTS`, idendify the network element that emitted the alarm and the type of that network element, respectively. The next two fields are the date and time when the alarm was sent. Finally, the sixth, `A1`, tells the priority level of the alarm on a scale from 1 to 5. □

The information contents of alarms vary a lot. Some alarms concern problems in logical concepts, such as virtual paths, some concern physical devices, such as power supplies. Some alarms report a distinct failure, e.g., that the incoming signal is missing, whereas some only report high error rate without any hint for the cause.

Even in a small telecommunication network there can be hundreds of different types of alarms. The number of alarms produced by a network varies greatly, of course, but typically there can be about 200–10000 alarms a day.

Processing of the alarm flow is a difficult task for the following reasons.

- The size of the networks and the diversity of alarm types mean that there are a lot of different situations that can occur.
- The alarms occur in bursts, and hence there is only little time for network managers to decide what to do with each alarm. However, when lots of alarms occur within a short time the operators should intervene.
- The hardware and software used in telecommunication networks develop fast. As new elements are added to the network or old ones are updated, the characteristics of the alarm sequences are changed. Thus the network managers do not have time to learn what the appropriate response to each situation is.

Alarm correlation A central problem of fault management is to determine which alarms are related, to combine the information in related alarms, and to make inferences about the faults and their locations. It is the task of a *network management center* to correlate the alarms as they are received from the network. *Correlating* alarms means combining the fragmented information they contain and interpreting the flow of alarms as a whole. Alarm correlation systems typically are expert systems performing operations such as (i) *removing* alarms carrying redundant information, (ii) *filtering out* low-priority alarms when higher-priority alarms are present, or (iii) *substituting* a set of alarms by some new information [45]. The goal of alarm correlation is to reduce the amount of information shown to the network managers, improve the usefulness of the information, and ultimately to identify the most probable faults that caused the alarms and to possibly even propose corrective actions. While the prediction of severe faults is a difficult task, the economic benefits that would be obtained from such predictions would be significant.

In addition to the alarms received from a network, several sources of background information are essential in alarm correlation. Knowledge about the topological relationships of network elements is crucial. A lot of other sources are also useful for interpreting alarms. For instance, knowledge about recent problems in the network may help to explain certain alarms.

While the use of alarm correlation systems is quite popular and methods for specifying the correlations are maturing, acquiring all the knowledge necessary for constructing an alarm correlation system for a network and its elements is difficult. The complexity and diversity of network elements and the large variation in the patterns of alarm occurrences pose serious problems for network management experts trying to build a correlation model.

Alarm correlation is typically based on looking at the active alarms within a time window, and interpreting them as a group. We adopt the following formal view to alarm correlation, similar to the one taken, e.g., in [44]. Abstractly, the input to a correlation system is an ordered sequence $(A_1, t_1), (A_2, t_2), \ldots$ of alarm occurrences. A *correlation pattern* describes a situation that can be recognized in an alarm sequence within a time window of a given length. Typically, a correlation pattern is an expression on the set of active alarms of, e.g., the last five minutes. If in a given window there is a set of alarms that matches the correlation pattern, then the set is said to be an occurrence of the pattern. Associated with each correlation pattern is a *correlation action*, which is to be executed when there is an occurrence of the corresponding pattern in a window.

Example 3.2 Consider a correlation pattern containing two alarms, "*link alarm* from X with severity 1" (alarm type A) and "*high fault rate* in X" (alarm type B), where the variable X may be replaced by the same network element in both alarms. Assume that when alarms of types A and B co-occur, they are known to be followed by fatal problems in the network element X. The correlation action could combine the information in the alarms to a high priority warning message "X will probably collapse within an hour", present it to the network manager, and filter out the original alarms. □

Given correlation patterns and actions, and a sequence s of alarms, an alarm correlation process continuously observes the incoming alarm sequence s, considers the last time window on s, and executes the actions associated with the patterns that occur in the window.

Alarm correlation and filtering systems for telecommunication network management have been presented, e.g., in [29, 30, 44, 45, 47, 71, 9]. Similar approaches have been used successfully also in process control tasks [66].

The problem Building a system for alarm correlation is a difficult task. Networks are large and network elements are complex. The number of correlation patters can be very large, and acquiring them from technical experts is a tedious task.

Correlations may pass unnoticed by the experts, for different reasons. It can be that an expert knows a correlation but did not come to think about it, or a correlation can be such that the expert did not even know there was a connection between certain alarms. Both networks and network elements evolve quickly over time, so a correlation system is never complete. It also takes time for the experts to learn new correlations and to modify existing ones.

3.2 Sketching a solution

Setting the goal Our high-level goal is to discover useful knowledge from telecommunication alarms, mainly to help in the construction of alarm correlation systems. From a sequence of alarms different types of knowledge can be discovered, for example neural networks, hazard models, or rule-based representations. If the goal would be just to obtain good predictive performance, a neural network could be useful. There is impressive evidence on the wide applicability of neural networks. However, in the current application one important goal is the comprehensibility of the discovered knowledge: the telecommunication operators do not wish to install any "black boxes" into their systems. This rules out the simple-minded use of neural networks.

From the statistical point of view, an alarm sequence can be viewed as a marked point process, and each event (i.e., an alarm) can be considered as a failure of a component. Therefore the hazard rate based methods of analysis of failure-time data for the data can be used. Combined with the Bayesian paradigm, the statistical machinery for these types of models is very powerful. However, these methods require a lot of human effort in the building of the

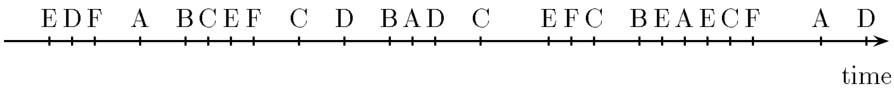


Figure 3.1: A sequence of alarms.

statistical models, as well as enormous amounts of computational resources. Currently they are not feasible for the analysis of hundreds of different types of alarms and their potential relations.

In this case rather simple rule-based formalisms are useful. Since the correlation systems are often based on looking at a time window of the alarm sequence at a time, it would be useful to use a similar concept in the discovered rules. So we choose as our goal the discovery of rules of the following form: "if certain alarms occur within a time window of a given width, then certain other alarm also occur within the time window". This type of rules are particularly useful for the following reasons.

- Comprehensibility: such rules are easy to understand and verify.
- "Standards": Correlation patterns closely resemble this rule format.
- Characteristics of the application domain: such rules can be representations of simple small causal relationships within the domain.

A first solution Let us now outline an exact representation of the problem. The alarms contain a number of fields, and the most important ones are the type of the alarm and the network element that sent the alarm. We first consider only the alarm type. The occurrence time is also essential: in the interest of considering different windows on the alarm sequence, we want to know the times of the alarms. Unfortunately though, the time stamps on the alarms can be inaccurate, due to unproper synchronization of clocks in the network elements. Anyway, we choose to view an occurrence of an alarm abstractly as a pair (A, t), where A is the *alarm type* and t is the *time* of the alarm. A sequence of alarms can now be represented as a sequence of alarm types, as in Figure 3.1. In the figure, A, B, C, D, E, and F are alarm types, and they have been marked on a time line.

What are the useful rules like more exactly? We want a set of alarms to occur close to each other, i.e., within a time window, as in the correlation patterns. Because of the poor synchronization of clocks, the rules should not be very sensitive to changes in the order of alarms. If we look at a window on the alarm sequence and consider the alarms in it as an (unordered) set, we obtain these goals at least partially. Additionally, the alarm sequence is merged from several sources, and therefore the rules should be insensitive to intervening events. We thus want to search for rules that only test the presense of alarms in a window and that do not require that other alarms are not present. If we look at an alarm sequence through a sliding window and find out which sets of alarms tend to occur together, we fulfill the requirements.

This task is obviously very similar to the task of finding frequent sets, and we can actually formulate a first solution as a modification of the frequent set discovery task. Consider a window at a given starting position and with a given width as a (multi) set of those alarm types that occur within the window. Denote now the set of all windows of the given width but over all possible starting positions by r, and denote the set of alarm types by R. Now r can be considered a binary database over R, and one can discover the collection $\mathcal{F}(r, \mathit{min\_fr})$ of frequent sets in r with respect to some frequency threshold *min_fr*. This way one finds all those sets of alarms that occur in a randomly chosen window of the given width with a probability at least *min_fr*. By discovering association rules one finds the kind of rules we were looking for: $X \Rightarrow Y$ means that "if the set X of alarms occurs within a time window of a given width, then alarms Y also occur in the time window".

A more general approach Let us have another look at the alarms in the sequence of Figure 3.1 and the kind of patterns that seem to occur frequently. Clearly, an alarm of type E is soon followed by F. One can also make the observation that whenever A and B occur (in either order), C occurs soon. The frequent alarm sets are interesting, but they do not take the order of alarms into consideration at all. What if we know that the order of alarms is important?

Instead of sets of alarms, we can consider partially ordered sets of alarms in general. Let us call such frequently occurring patterns *episodes*. We can draw episodes simply as directed acyclic graphs. Consider, for instance, episodes α, β, and γ in Figure 3.2. Episode α occurs in a sequence only if there are events of types E and F that occur in this order in the sequence. We call such totally ordered episodes *serial*. Episode β is a *parallel* episode: no constraints on the relative order of A and B are given. Episode γ is an example of non-serial and non-parallel episode: it occurs in a sequence if there are occurrences of A and B and these precede an occurrence of C; no constraints on the relative order of A and B are given.

It would not be very useful to search for all possible episodes with arbitrary partial orders. Even for small episodes there would be a large number of different partial orders, and frequent episodes would most probably contain a lot of redundancy. Instead, we assume that the user restricts the search to a certain class of episodes, in practice to either parallel or serial episodes. Note that even these classes have an infinite number of episodes if episodes are allowed to contain multiple similar events and if the size of episodes is not restricted.

Once the frequent episodes are known, they can be used to obtain rules, e.g., for prediction. For example, if we know that the episode β of Figure 3.2

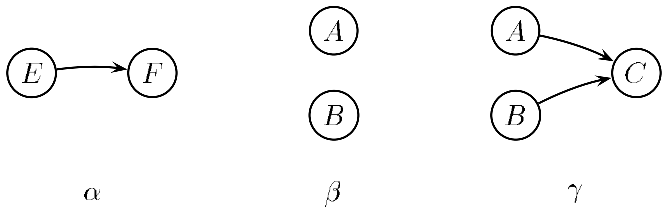


Figure 3.2: Episodes.

occurs in 4.2 % of the windows and that the superepisode γ occurs in 4.0 % of the windows, we can estimate that after seeing a window with A and B, there is a chance of about 0.95 that C follows in the same window. One can compute such rules and their confidences from the frequencies of episodes, just like confidences are computed for association rules.

Summary of the approach We propose automatic methods for the analysis of alarms, to aid in the knowledge acquisition phase of building an alarm correlation system. Briefly, the scenario for building alarm correlation systems is the following. First, a large database of alarms is analyzed off-line, and episode rules are discovered automatically. Then, for the construction of an alarm correlation system, the network management specialists have a collection of patterns of alarms at hand, and they can build the alarm correlation model from them. In the final step, the correlation rules are applied in real-time fault identification.

The discovery methods, although not directly applicable for on-line analysis themselves, can be used also in network surveillance. The analysis can be rerun or augmented, e.g., every day or every hour. Recent patterns may point to yet unnoticed faults in the network.

To sum up, the knowledge discovery task we consider is the following. Given a class of episodes, an input sequence of events, a window width, and a frequency threshold, find all episodes that are frequent in the windows on the event sequence. In the algorithm for solving this task we can use many ideas from the algorithm for discovering frequent sets. Additionally, episodes can be recognized efficiently in the database pass by "sliding" a window on the input sequence. Windows starting at successive positions have a lot of overlap and are therefore similar to each other. We can take advantage of this similarity: after recognizing episodes in a window, we can make incremental updates in our data structures to determine which episodes occur in the next window.

Chapter 4

Frequent episodes

In Chapter 3 we introduced the problem of analyzing telecommunication alarm sequences, and we briefly outlined a solution based on the idea of discovering frequent alarm sets in alarm sequences. Motivated by Chapter 3, we now give exact and domain-independent formulations for the ideas about finding all frequent episodes.

We define concepts related to event sequences and episodes in Section 4.1. In Section 4.2 we give algorithms for the discovery of all frequent parallel and serial episodes. We touch on the discovery of episodes with other types of partial orders, and on the rule generation process. Illustrative experimental results are presented in Section 4.3. Experiences are discussed in Chapter 6 in the context of the whole KDD process.

4.1 The framework

Let us now work out the approach in more detail. Since the framework is application independent—although motivated by fault management—we talk about events instead of alarms. We start by defining event sequences and windows.

Definition 4.1 Given a set R of *event types*, an *event* is a pair (A, t), where $A \in R$ is an event type and t is an integer, the *(occurrence) time* of the event.

An *event sequence* $\mathbf{s}$ on R is a triple (s, T_s, T_e), where $T_s < T_e$ are integers, T_s is called the starting time and T_e the ending time, and

$$s = \langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle$$

is an ordered sequence of events such that $A_i \in R$ and $T_s \leq t_i < T_e$ for all $i = 1, \ldots, n$, and $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n-1$. $\square$

Example 4.2 Figure 4.1 presents graphically the event sequence $\mathbf{s} = (s, 29, 68)$, where

$$s = \langle (E, 31), (D, 32), (F, 33), (A, 35), (B, 37), (C, 38), \ldots, (D, 67) \rangle.$$

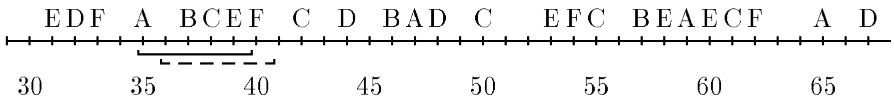


Figure 4.1: The example event sequence $\mathbf{s}$ and two windows of width 5.

Observations of the event sequence have been made from time 29 to just before time 68. For each event that occurred in the time interval $[29, 68)$, the event type and the time of occurrence have been recorded. □

Think now of looking at an event sequence through a narrow window, giving a view to the events within a relatively small time period. We define a window as a slice of an event sequence that is seen at any given time. We then discuss the case where one considers an event sequence as a sequence of partially overlapping windows.

Definition 4.3 A *window* on event sequence $\mathbf{s} = (s, T_s, T_e)$ is an event sequence $\mathbf{w} = (w, t_s, t_e)$, where $t_s < T_e, t_e > T_s$, and w consists of those pairs (A, t) from s where $t_s \leq t < t_e$. The time span $t_e - t_s$ is called the *width* of the window $\mathbf{w}$, and it is denoted $width(\mathbf{w})$. Given an event sequence $\mathbf{s}$ and an integer win, we denote by $\mathcal{W}(\mathbf{s}, win)$ the set of all windows $\mathbf{w}$ on $\mathbf{s}$ such that $width(\mathbf{w}) = win$. □

There is a small trick in the definition. The first and last windows on a sequence extend outside the sequence, so that the first window only contains the first time point of the sequence, and the last window only contains the last time point. With this definition an event close to either end of a sequence is observed in equally many windows to an event in the middle of the sequence. Given an event sequence $\mathbf{s} = (s, T_s, T_e)$ and a window width win, the number of windows in $\mathcal{W}(\mathbf{s}, win)$ is $T_e - T_s + win - 1$.

Example 4.4 Figure 4.1 shows two windows of width 5 on the sequence $\mathbf{s}$ of the previous example. A window starting at time 35 is shown in solid line, and the immediately following window, starting at time 36, is depicted with a dashed line. The window starting at time 35 is

$$(\langle (A, 35), (B, 37), (C, 38), (E, 39) \rangle, 35, 40).$$

Note that the event $(F, 40)$ that occurred at the ending time is not in the window. The window starting at 36 is similar to this one; the difference is that the first event $(A, 35)$ is missing and there is a new event $(F, 40)$ at the end.

The set of the 43 partially overlapping windows of width 5 constitutes $\mathcal{W}(\mathbf{s}, 5)$; the first window is $(\emptyset, 25, 30)$, and the last is $(\langle (D, 67) \rangle, 67, 72)$.

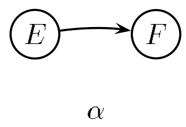


Figure 4.2: An episode.

Event $(D, 67)$ occurs in 5 windows of width 5, as does, e.g., event $(C, 50)$. If only windows totally within the sequence were considered, event $(D, 67)$ would occur only in window $(\langle(A, 65), (D, 67)\rangle, 63, 68)$. □

We now move on to define episodes formally. We also define when an episode is a subepisode of another; this relation is then used in the candidate generation for frequent episodes.

Definition 4.5 An *episode* α is a triple $(V, \leq, g)$ where V is a set of nodes, $\leq$ is a partial order on V, and $g : V \to R$ is a mapping associating each node with an event type. The interpretation of an episode is that the events in $g(V)$ have to occur in the order described by $\leq$. The *size* of α, denoted $|\alpha|$, is $|V|$. Episode α is *parallel* if the partial order $\leq$ is trivial (i.e., $x \not\leq y$ for all $x, y \in V$ such that $x \neq y$). Episode α is *serial* if the relation $\leq$ is a total order (i.e., $x \leq y$ or $y \leq x$ for all $x, y \in V$). Episode α is *injective* if the mapping g is an injection, i.e., no event type occurs twice in the episode. □

Example 4.6 An episode $\alpha = (V, \leq, g)$ can be drawn as a directed acyclic graph, where $g(x)$ is the label of node $x \in V$, and where edges indicate temporal ordering. Consider episode $\alpha = (V, \leq, g)$ in Figure 4.2. The set V contains two nodes; call them x and y. The mapping g labels these nodes with the event types that are seen in the figure: $g(x) = E$ and $g(y) = F$. An event of type E is supposed to occur before an event of type F, i.e., x precedes y, and we have $x \leq y$. Episode α is injective, since it does not contain duplicate event types; in a window where α occurs there may, however, be multiple events of types E and F. □

Definition 4.7 An episode $\beta = (V', \leq', g')$ is a *subepisode* of $\alpha = (V, \leq, g)$, denoted $\beta \preceq \alpha$, if there exists an injective mapping $f : V' \to V$ such that $g'(v) = g(f(v))$ for all $v \in V'$, and for all $v, w \in V'$ with $v \leq' w$ also $f(v) \leq f(w)$. An episode α is a *superepisode* of β if and only if $\beta \preceq \alpha$. We write $\beta \prec \alpha$ if $\beta \preceq \alpha$ and $\alpha \not\preceq \beta$. □

Example 4.8 Figure 4.3 presents two episodes, β and γ. From the figure we see that we have $\beta \preceq \gamma$ since β is a subgraph of γ. In terms of Definition 4.7,

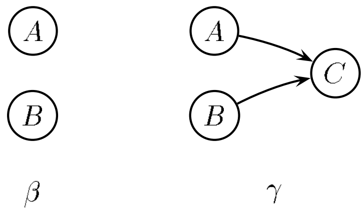


Figure 4.3: A subepisode and episode.

there is a mapping f that connects the nodes labeled A with each other and the nodes labeled B with each other, i.e., both nodes of β have (disjoint) corresponding nodes in γ. Since the nodes in episode β are not ordered, the corresponding nodes in γ do not need to be ordered, either, but they could be. □

Consider now what it means that an episode occurs in a sequence. The nodes of the episode need to have corresponding events in the sequence such that the event types are the same and the partial order of the episode is respected. Below we formalize this. We also define the frequency of an episode as the fraction of windows in which the episode occurs.

Definition 4.9 An episode $\alpha = (V, \leq, g)$ *occurs* in an event sequence

$$\mathbf{s} = (\langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle, T_s, T_e),$$

if there exists an injective mapping $h : V \rightarrow \{1, \ldots, n\}$ from nodes to events, such that $g(x) = A_{h(x)}$ for all $x \in V$, and for all $x, y \in V$ with $x \neq y$ and $x \leq y$ we have $t_{h(x)} < t_{h(y)}$. □

Example 4.10 The window $(w, 35, 40)$ of Figure 4.1 contains events A, B, C, and E, in that order. Both episodes β and γ of Figure 4.3 occur in the window. □

Definition 4.11 Given an event sequence $\mathbf{s}$ and a window width *win*, the *frequency* of an episode α in $\mathbf{s}$ is

$$fr(\alpha, \mathbf{s}, win) = \frac{|\{\mathbf{w} \in \mathcal{W}(\mathbf{s}, win) \mid \alpha \text{ occurs in } \mathbf{w}\}|}{|\mathcal{W}(\mathbf{s}, win)|},$$

i.e., the fraction of windows on $\mathbf{s}$ in which α occurs.

Given a *frequency threshold* *min_fr*, α is *frequent* if $fr(\alpha, \mathbf{s}, win) \geq min\_fr$. The collection of episodes that are frequent in $\mathbf{s}$ with respect to *win* and *min_fr* is denoted $\mathcal{F}(\mathbf{s}, win, min\_fr)$. The collection of frequent episodes of size l is denoted $\mathcal{F}_l(\mathbf{s}, win, min\_fr)$. □

Algorithm 4.13
Input: A set R of event types, an event sequence $\mathbf{s}$ over R, a set $\mathcal{E}$ of episodes, a window width *win*, and a frequency threshold *min_fr*.
Output: The collection $\mathcal{F}(\mathbf{s}, win, min\_fr)$ of frequent episodes.
Method:

```
compute C_1 := {α ∈ E | |α| = 1};
l := 1;
while C_l ≠ ∅ do
    // Database pass (Algorithms 4.19 and 4.21):
    compute F_l(s, win, min_fr) := {α ∈ C_l | fr(α, s, win) ≥ min_fr};
    l := l + 1;
    // Candidate generation (Algorithm 4.14):
    compute C_l := {α ∈ E | |α| = l, and β ∈ F_|β|(s, win, min_fr) for all
                                β ∈ E such that β ≺ α and |β| < l};
for all l do output F_l(s, win, min_fr);
```

$$\mathcal{C}_1 := \{\alpha \in \mathcal{E} \mid |\alpha| = 1\};\quad l := 1;$$
$$\textbf{while } \mathcal{C}_l \neq \emptyset \textbf{ do}$$
$$\mathcal{F}_l(\mathbf{s}, win, min\_fr) := \{\alpha \in \mathcal{C}_l \mid fr(\alpha, \mathbf{s}, win) \geq min\_fr\};\quad l := l+1;$$
$$\mathcal{C}_l := \{\alpha \in \mathcal{E} \mid |\alpha| = l, \text{ and } \beta \in \mathcal{F}_{|\beta|}(\mathbf{s}, win, min\_fr) \text{ for all } \beta \in \mathcal{E} \text{ such that } \beta \prec \alpha \text{ and } |\beta| < l\};$$
$$\textbf{for } \text{all } l \textbf{ do } \text{output } \mathcal{F}_l(\mathbf{s}, win, min\_fr);$$

We can now give an exact formulation of the discovery task at hand: given an event sequence $\mathbf{s}$, a set $\mathcal{E}$ of episodes, a window width *win*, and a frequency threshold *min_fr*, find $\mathcal{F}(\mathbf{s}, win, min\_fr)$.

4.2 Algorithms

In the algorithms of this section we assume that the search space $\mathcal{E}$ consists of all parallel or serial episodes constructed from the events in a given set R. Our main algorithm makes only the assumption that there are candidates and frequent episodes of each size up to the size of the largest frequent episodes; the more detailed algorithms for candidate generation and database pass, in turn, assume that episodes are etither parallel or serial. Towards the end of this section we briefly consider the discovery of episodes with more general partial orders.

4.2.1 Main algorithm

Algorithm 4.13 computes the collection $\mathcal{F}(\mathbf{s}, win, min\_fr)$ of frequent episodes. The algorithm has the familiar structure of alternation between candidate generation and database pass phases. The crucial point in the candidate generation is given by the following immediate lemma.

Lemma 4.12 If an episode α is frequent in an event sequence $\mathbf{s}$, then all subepisodes $\beta \preceq \alpha$ are frequent.

4.2.2 Generation of candidate episodes

We present in detail a candidate generation method which is a generalization of the candidate generation for frequent sets. The method can be adapted to

Algorithm 4.14
Input: A sorted array $\mathcal{F}_l$ of frequent parallel episodes of size l.
Output: A sorted array of candidate parallel episodes of size $l+1$.
Method:

1. $\mathcal{C}_{l+1} := \emptyset$;
2. $k := 0$;
3. **if** $l = 1$ **then for** $h := 1$ **to** $|\mathcal{F}_l|$ **do** $\mathcal{F}_l.block\_start[h] := 1$;
4. **for** $i := 1$ **to** $|\mathcal{F}_l|$ **do**
5. $current\_block\_start := k + 1$;
6.   **for** $(j := i; \mathcal{F}_l.block\_start[j] = \mathcal{F}_l.block\_start[i]; j := j+1)$ **do**
7.   // $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ have $l-1$ first event types in common,
8.   // build a potential candidate α as their combination:
9.   **for** $x := 1$ **to** l **do** $\alpha[x] := \mathcal{F}_l[i][x]$;
10. $\alpha[l+1] := \mathcal{F}_l[j][l]$;
11.   **for** $y := 1$ **to** $l-1$ **do**
12.   // Build and test subepisodes β that do not contain $\alpha[y]$:
13.   **for** $x := 1$ **to** $y-1$ **do** $\beta[x] := \alpha[x]$;
14.   **for** $x := y$ **to** l **do** $\beta[x] := \alpha[x+1]$;
15.   **if** β is not in $\mathcal{F}_l$ **then** continue with the next j at line 6;
16.   // All subepisodes are in $\mathcal{F}_l$, store α as candidate:
17. $k := k+1$;
18. $\mathcal{C}_{l+1}[k] := \alpha$;
19. $\mathcal{C}_{l+1}.block\_start[k] := current\_block\_start$;
20. output $\mathcal{C}_{l+1}$;

---

deal with parallel episodes (i.e., multisets of items), serial episodes (ordered multisets), and injective parallel and serial episodes (sets and ordered sets).

Algorithm 4.14 computes the candidates for parallel episodes. In the algorithm, an episode $\alpha = (V, \leq, g)$ is represented as a lexicographically sorted array of event types. The array is denoted by the name of the episode and the items in the array are referred to using square brackets. For example, a parallel episode α with events of types A, C, C, and F is represented as an array α with $\alpha[1] = A, \alpha[2] = C, \alpha[3] = C$, and $\alpha[4] = F$. Collections of episodes are also represented as lexicographically sorted arrays, i.e., the ith episode of a collection $\mathcal{F}$ is denoted by $\mathcal{F}[i]$.

Since the episodes and episode collections are sorted, all episodes that share the same first event types are consecutive in the episode collection. In particular, if episodes $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ of size l share the first $l-1$ events, then for all k with $i \leq k \leq j$ we have that $\mathcal{F}_l[k]$ shares also the same events. A maximal sequence of consecutive episodes of size l that share the first $l-1$ events is called a *block*. Potential candidates can be identified by creating all combinations of two episodes in the same block. For the efficient identification of blocks, we store in $\mathcal{F}_l.block\_start[j]$ for each episode $\mathcal{F}_l[j]$ the index i such that $\mathcal{F}_l[i]$ is the first episode in the block.

Theorem 4.15 Algorithm 4.14 works correctly.

Proof The crucial claim is that in the algorithm the pairs $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ of episodes generate all candidates. For a moment assume that for each episode $\mathcal{F}_l[j]$ the value of $\mathcal{F}_l.block\_start[j]$ is the index i such that $\mathcal{F}_l[i]$ is the first episode in the block. We show later that this assumption holds. —In the following we identify an episode with its index in the collection.

In the outer loop (line 4) variable i iterates through all episodes in $\mathcal{F}_l$, and in the inner loop (line 6) variable j iterates through those episodes in $\mathcal{F}_l$ that are in the same block with i but are not before i. Consider now any block b of episodes in $\mathcal{F}_l$. Variables i and j obviously iterate through all (unordered) pairs of episodes in block b, including the case where $i = j$.

Since i and j are in the same block, they have the same $l-1$ first event types. Conceptually we construct a new *potential candidate* α as the union of episodes (multisets) i and j. We build α by taking first the common $l-1$ events and the lth event from episode i (both done on line 9), and finally the event number $l+1$ from episode j (line 10). Then the events of a potential candidate are lexicographically sorted. Since the iteration of episodes proceeds in lexicographical order (over the sorted collection $\mathcal{F}_l$), the collection of candidates is also constructed in lexicographical order.

Next we show that the collection of potential candidates α contains all valid candidates γ of size $l+1$. All subepisodes of γ are frequent, and in particular those two subepisodes δ_1 and δ_2 of size l that contain all but the last and the second last events of γ, respectively. Since δ_1 and δ_2 are in $\mathcal{F}_l$ and they have $l-1$ items in common, they are in the same block. At some time in the algorithm we have $\mathcal{F}_l[i] = \delta_1$ and $\mathcal{F}_l[j] = \delta_2$, and γ is considered as a potential candidate in the algorithm.

We need to show that no false candidates are output. An episode of size $l+1$ has $l+1$ subepisodes β of size l, and for all of these we make sure that they are in $\mathcal{F}_l$. We obtain all these subepisodes by leaving out one of the events in α at a time (line 11). Note that the two subepisodes that were used for constructing α do not need to be checked again. Only if all subepisodes of size $l-1$ are in $\mathcal{F}_l$, is α correctly output as a candidate.

Finally we show that we have the correct value $\mathcal{F}_l.block\_start[j] = i$ for all j, i.e., the index i such that $\mathcal{F}_l[i]$ is the first episode in the block. For $l = 1$ the structure is built on line 3: all episodes of size 1 have at least 0 common events, so they are all in the same block, and $\mathcal{F}_1.block\_start[h] = 1$ for all h. For $l \geq 1$ and $\mathcal{F}_{l+1}$, a block b of $\mathcal{F}_{l+1}$ (or $\mathcal{C}_{l+1}$) has the property that all episodes in the block have been generated from the same episode $\mathcal{F}_l[i]$. This is due to the simple fact that the first l events have been copied directly from $\mathcal{F}_l[i]$ (line 9). We save for each i the index of the first candidate generated from it (line 5), and then use the saved value to set $\mathcal{C}_{l+1}.block\_start[k]$ correctly for all candidates k in the block (line 19). □

Algorithm 4.14 can be easily modified to generate candidate serial episodes. Now the events in the array representing an episode are in the order imposed by a total order $\leq$. For instance, a serial episode β with events of types C, A, F, and C, in that order, is represented as an array β with $\beta[1] = C$, $\beta[2] = A$, $\beta[3] = F$, and $\beta[4] = C$. Collections of episodes are still stored as lexicographically sorted arrays. The only change to the algorithm is to replace line 6.

Theorem 4.16 With the line

6. **for** $(j := \mathcal{F}_l.block\_start[i]; \mathcal{F}_l.block\_start[j] = \mathcal{F}_l.block\_start[i];$ $j := j + 1)$ **do**

Algorithm 4.14 works correctly for serial episodes.

Proof The proof is similar to the proof for Theorem 4.15; now, however, i and j iterate over all *ordered* pairs of episodes in each block. The (potential) candidates are ordered sequences of event types, not sorted arrays as before, but the candidate collection is still constructed in lexicographical order. The same arguments for the correctness of the candidate collection and the structure $\mathcal{F}_l.block\_start$ hold. □

There are further options with the algorithm. If the desired episode class consists of parallel or serial injective episodes, i.e., no episode should contain any event type more than once, simply add one line.

Theorem 4.17 With the line

6b. **if** $j = i$ **then** continue with the next j at line 6;

inserted after line 6, Algorithm 4.14 works correctly for injective parallel episodes (or injective serial episodes with the change of Theorem 4.16).

Proof Clearly, the effect of the inserted line is that some candidates are not generated. Consider now those excluded candidate episodes. First note that only candidates α that contain some event type at least twice are excluded. Either a candidate is excluded explicitly because $i = j$, or it is not generated because some of its subepisodes is not in $\mathcal{F}_l$. If α is excluded explicitly, then it contains the event type $\alpha[l] = \alpha[l + 1]$ twice. If, on the other hand, some tested subepisode β is not in the collection $\mathcal{F}_l$, then there must be a subepisode $\gamma \preceq \beta$ that has been excluded explicitly. Then α contains twice the event type $\gamma[|\gamma|]$.

Now note that no episode α with at least two occurrences of an event type is generated. Let A be an event type that occurs at least twice in α. Then for the episode γ of size 2 such that $\gamma[1] = A$ and $\gamma[2] = A$ we have $\gamma \preceq \alpha$, and thus α cannot be a candidate unless γ is frequent. However, γ has been excluded explicitly by the inserted line in an earlier iteration, and thus α is not a candidate. □

The time complexity of Algorithm 4.14 is polynomial in the size of the collection of frequent episodes and it is independent of the length of the event sequence.

Theorem 4.18 Algorithm 4.14 (with any of the above variations) has time complexity $\mathcal{O}(l^2 \, |\mathcal{F}_l|^2 \log |\mathcal{F}_l|)$.

Proof The initialization (line 3) takes time $\mathcal{O}(|\mathcal{F}_l|)$. The outer loop (line 4) is iterated $\mathcal{O}(|\mathcal{F}_l|)$ times and the inner loop (line 6) $\mathcal{O}(|\mathcal{F}_l|)$ times. Within the loops, a potential candidate (lines 9 and 10) and $l-1$ subcandidates (lines 11 to 14) are built in time $\mathcal{O}(l+1+(l-1)l) = \mathcal{O}(l^2)$. More importantly, the $l-1$ subsets need to be searched for in the collection $\mathcal{F}_l$ (line 15). Since $\mathcal{F}_l$ is sorted, each subcandidate can be located with binary search in time $\mathcal{O}(l \log |\mathcal{F}_l|)$. The total time complexity is thus $\mathcal{O}(|\mathcal{F}_l| + |\mathcal{F}_l| \, |\mathcal{F}_l| \, (l^2 + (l-1) \, l \, \log |\mathcal{F}_l|)) = \mathcal{O}(l^2 \, |\mathcal{F}_l|^2 \log |\mathcal{F}_l|)$. □

In practical situations the time complexity is likely to be close to $\mathcal{O}(l^2 \, |\mathcal{F}_l| \, \log |\mathcal{F}_l|)$, since the blocks are typically small.

4.2.3 Recognizing episodes in sequences

Let us now consider the implementation of the database pass. We give algorithms which recognize episodes in sequences in an incremental fashion. For two windows $\mathbf{w} = (w, t_s, t_s + win)$ and $\mathbf{w}' = (w', t_s + 1, T_s + win + 1)$, the sequences w and w' of events are similar to each other. We take advantage of this similarity: after recognizing episodes in $\mathbf{w}$, we make incremental updates in our data structures to achieve the shift of the window to obtain $\mathbf{w}'$.

The algorithms start by considering the empty window just before the input sequence, and they end after considering the empty window just after the sequence. This way the incremental methods need no other special actions at the beginning or end. For the frequency of episodes, only the windows correctly on the input sequence are, of course, considered.

Parallel episodes Algorithm 4.19 recognizes candidate parallel episodes in an event sequence. The main ideas of the algorithm are the following. For each candidate parallel episode α we maintain a counter $\alpha.\mathit{event\_count}$ that indicates how many events of α are present in the window. When $\alpha.\mathit{event\_count}$ becomes equal to $|\alpha|$, indicating that α is entirely included in the window, we save the starting time of the window in $\alpha.\mathit{inwindow}$. When $\alpha.\mathit{event\_count}$ decreases again, indicating that α is no longer entirely in the window, we increase the field $\alpha.\mathit{freq\_count}$ by the number of windows where α remained entirely in the window. At the end, $\alpha.\mathit{freq\_count}$ contains the total number of windows where α occurs.

To access candidates efficiently, they are indexed by the number of events of each type that they contain: all episodes that contain exactly a events of type A are in the list *contains(A,a)*. When the window is shifted and the contents of the window change, the episodes that are affected are updated. If, for instance, there is one event of type A in the window and a second

Algorithm 4.19
Input: A collection $\mathcal{C}$ of parallel episodes, an event sequence $\mathbf{s} = (s, T_s, T_e)$, a window width *win*, and a frequency threshold *min_fr*.
Output: The episodes of $\mathcal{C}$ that are frequent in $\mathbf{s}$ with respect to *win* and *min_fr*.
Method:

```
1.  // Initialization:
2.  for each α in C do
3.      for each A in α do
4.          A.count := 0;
5.          for i := 1 to |α| do contains(A, i) := ∅;
6.  for each α in C do
7.      for each A in α do
8.          a := number of events of type A in α;
9.          contains(A, a) := contains(A, a) ∪ {α};
10.     α.event_count := 0;
11.     α.freq_count := 0;
12. // Recognition:
13. for start := T_s − win + 1 to T_e do
14.     // Bring in new events to the window:
15.     for all events (A, t) in s such that t = start + win − 1 do
16.         A.count := A.count + 1;
17.         for each α ∈ contains(A, A.count) do
18.             α.event_count := α.event_count + A.count;
19.             if α.event_count = |α| then α.inwindow := start;
20.     // Drop out old events from the window:
21.     for all events (A, t) in s such that t = start − 1 do
22.         for each α ∈ contains(A, A.count) do
23.             if α.event_count = |α| then
24.                 α.freq_count := α.freq_count − α.inwindow + start;
25.             α.event_count := α.event_count − A.count;
26.         A.count := A.count − 1;
27. // Output:
28. for all episodes α in C do
29.     if α.freq_count/(T_e − T_s + win − 1) ≥ min_fr then output α;
```

---

one comes in, all episodes in the list *contains(A,2)* are updated with the information that both events of type A they are expecting are now present.

Theorem 4.20 Algorithm 4.19 works correctly.

Proof We consider the following two invariants. (1) For each event type A that occurs in any episode, the variable $A.count$ correctly contains the number of events of type A in the current window. (2) For each episode α, the counter $\alpha.event\_count$ equals $|\alpha|$ exactly when α occurs in the current window.

The first invariant holds trivially for the empty window starting at $T_s - win$, since counters $A.count$ are initialized to zero on line 4. Assume now that the counters are correct for the window starting at $start - 1$, and consider the computation for the window starting at $start$, i.e., one iteration of the loop starting at line 13. On lines 15 – 16, the counters are updated for each new event in the window; similarly, on lines 21 and 26, the counters are updated for events no longer in the window.

For the second invariant, first note that each set $contains(A, a)$ consists of all episodes that contain exactly a events of type A: the lists are initialized to empty on line 5, and then filled correctly for each event type in each episode on line 9. Now consider the counter $\alpha.event\_count$ for any episode α. In the beginning, the counter is initialized to zero (line 10). Given an event type A, denote by a the number of events of type A in α. The effect of lines 18 and 25 is that $\alpha.event\_count$ is increased by a exactly for the time when there are at least a events of type A in the window. Thus $\alpha.event\_count = |\alpha|$ exactly when there are enough events of each type of α in the window.

Finally note that at the end $\alpha.freq\_count$ is correct. The counter is initialized to zero (line 11). Given any number of consecutive windows containing α, by the invariant the index of the first window is stored in $\alpha.inwindow$ on line 19. After the last window of these, i.e., in the first window not containing α, the counter $\alpha.freq\_count$ is increased by the number of the consecutive windows containing α (line 24). Since the last window considered is the empty window immediately after the sequence, occurrences in the last windows on the sequence are correctly computed. On the last lines the frequent episodes are output. □

Serial episodes Serial candidate episodes are recognized in an event sequence by using state automata that accept the candidate episodes and ignore all other input. The idea is that there is an automaton for each serial episode α, and that there can be several instances of each automaton at the same time, so that the active states reflect the (disjoint) prefixes of α occurring in the window. Algorithm 4.21 implements this idea.

We initialize a new instance of the automaton for a serial episode α every time the first event of α comes into the window; the instance is removed when the same event leaves the window. When an automaton instance for α reaches its accepting state, indicating that α is entirely included in the window, and if there are no other instances for α in the accepting state already, we save the starting time of the window in $\alpha.inwindow$. When an automaton instance in the accepting state is removed, and if there are no other instances for α in the accepting state, we increase the field $\alpha.freq\_count$ by the number of windows where α remained entirely in the window.

It is useless to have multiple automaton instance in the same state, as they would only make the same transitions and produce the same information. It suffices to maintain the one that reached the common state last since it will be also removed last. There are thus at most $|\alpha|$ automaton instances for an episode α. For each instance we need to know when it should be removed. We can thus represent all the automaton instances for α with one array of size $|\alpha|$: the value of $\alpha.initialized[i]$ is the latest initialization time of an

Algorithm 4.21

Input: A collection $\mathcal{C}$ of serial episodes, an event sequence $\mathbf{s} = (s, T_s, T_e)$, a window width *win*, and a frequency threshold *min_fr*.

Output: The episodes of $\mathcal{C}$ that are frequent in $\mathbf{s}$ with respect to *win* and *min_fr*.

Method:

```
// Initialization:
for each α in C do
    for i := 1 to |α| do
        α.initialized[i] := 0;
        waits(α[i]) := ∅;
for each α ∈ C do
    waits(α[1]) := waits(α[1]) ∪ {(α, 1)};
    α.freq_count := 0;
for t := T_s − win to T_s − 1 do beginsat(t) := ∅;
// Recognition:
for start := T_s − win + 1 to T_e do
    // Bring in new events to the window:
    beginsat(start + win − 1) := ∅;
    transitions := ∅;
    for all events (A, t) in s such that t = start + win − 1 do
        for all (α, j) ∈ waits(A) do
            if j = |α| and α.initialized[j] = 0 then α.inwindow := start;
            if j = 1 then
                transitions := transitions ∪ {(α, 1, start + win − 1)};
            else
                transitions := transitions ∪ {(α, j, α.initialized[j − 1])};
                beginsat(α.initialized[j − 1]) :=
                                beginsat(α.initialized[j − 1]) \ {(α, j − 1)};
                α.initialized[j − 1] := 0;
                waits(A) := waits(A) \ {(α, j)};
    for all (α, j, t) ∈ transitions do
        α.initialized[j] := t;
        beginsat(t) := beginsat(t) ∪ {(α, j)};
        if j < |α| then waits(α[j + 1]) := waits(α[j + 1]) ∪ {(α, j + 1)};
    // Drop out old events from the window:
    for all (α, l) ∈ beginsat(start − 1) do
        if l = |α| then α.freq_count := α.freq_count − α.inwindow + start;
        else waits(α[l + 1]) := waits(α[l + 1]) \ {(α, l + 1)};
        α.initialized[l] := 0;
// Output:
for all episodes α in C do
    if α.freq_count/(T_e − T_s + win − 1) ≥ min_fr then output α;
```

automaton instance that has reached its ith state. Recall that α itself is represented by an array containing its events; this array can be used to label the state transitions.

To access and traverse the automata instances efficiently they are organized in the following way. For each event type $A \in R$, the instances that would next accept A are linked together to a list *waits(A)*. The list contains entries of the form (α, x) meaning that episode α is waiting for its xth event. When an event (A, t) enters the window during a shift, the list *waits(A)* is traversed. If an automaton instance reaches a common state with another instance, the earlier entry in the array $\alpha.initialized[]$ is simply overwritten.

The transitions made during one shift of the window are stored in a list *transitions*. They are represented in the form (α, x, t) meaning that episode α got its xth event, and the latest initialization time of the prefix of length x is t. Updates regarding the old states of the automaton instances are done immediately, but updates for the new states are carried out only after all transitions have been identified, in order to not overwrite any useful information. For easy removal of automaton instances when they go out of the window, the instances initialized at time t are stored in a list *beginsat(t)*. Note that only lists $beginsat(start-1)$ – $beginsat(start+win-1)$ are needed at any particular moment, and lists $beginsat(j)$ with $j < start - 1$ can be removed.

Theorem 4.22 Algorithm 4.21 works correctly.

Proof Let α be a serial episode in $\mathcal{C}$, j an integer such that $1 \leq j \leq |\alpha|$, and A an event type, and consider a window on the input sequence. Denote by $mpt(\alpha, j)$ the maximal time t in the window such that the prefix of length j of α occurs within the subsequence starting at time t and ending at where the window ends. Consider the following invariants.

1. We have $\alpha.initialized[j] = 0$, if the prefix does not occur in the window at all, or if $j < |\alpha|$ and $mpt(\alpha, j) = mpt(\alpha, j+1)$. Otherwise $\alpha.initialized[j] = mpt(\alpha, j)$.

2. For each time t in the window, we have $(\alpha, j) \in beginsat(t)$ if and only if $\alpha.initialized[j] = t$.

3. The list $waits(A)$ consists of entries (α, j) such that $\alpha[j] = A$ and either $j = 1$ or $\alpha.initialized[j-1] \neq 0$.

The first invariant holds trivially for the empty window in the beginning, as the data structures are initialized to zeros on line 4. Assume now that the data structures are correct for the window starting at $start - 1$, and consider the computation for the window starting at *start*. We show by induction that the computations are correct for all j. First, consider the case $j = 1$. When a new event comes to the window, it is always the latest prefix of length $j = 1$ for all episodes that start with the event type. The value of $\alpha.initialized[1]$ is correctly set to $start + win - 1$ for all such episodes α on lines 19 and 26. Assume now that $j > 1$, that $\alpha[j]$ comes into the window, and that $\alpha.initialized[k]$ is correct for all $k < j$. Now $mpt(\alpha, j)$ clearly equals the old value of $mpt(\alpha, j-1)$; the correct updates are done on lines 21 and 26 for $\alpha.initialized[j]$ and on line 23 for $\alpha.initialized[j-1]$. Note that the value of $\alpha.initialized[j-1]$ is set to non-zero later if $mpt(\alpha, j-1) > mpt(\alpha, j)$. Note also that when a prefix of length l is not in the window anymore, $\alpha.initialized[l]$ is correctly set to zero on line 33.

The second invariant holds also trivially in the beginning (line 9). Assuming that the data structures are correct for the window starting at $start - 1$, the correct additions to *beginsat* are done on line 27, and correct removals on line 22. (Removing lists $beginsat(t)$ with $t < start$ is not necessary.)

The third invariant holds for $j = 1$ for the whole algorithm: the *waits* lists are set correctly on line 7, and they are not altered during the algorithm. For larger prefixes correct additions to the *waits* lists are made on lines 19, 21, and 28, and correct removals are made when $\alpha.initialized[j-1]$ becomes zero (lines 24 and 32).

Based on these invariants, the index of the window is correctly stored in $\alpha.inwindow$ for the first of consecutive windows containing α (line 17), and $\alpha.freq\_count$ is correctly increased after the last of consecutive windows containing α (line 31). Finally, the frequent episodes are correctly output on the last lines of the algorithm. □

Analysis of time complexity For simplicity, suppose that the class of event types R is fixed, and assume that exactly one event takes place every time unit. Assume candidate episodes are all of size l, and let n be the length of the sequence. We go back to parallel episodes, and start the analysis from Algorithm 4.19.

Theorem 4.23 The time complexity of Algorithm 4.19 is $\mathcal{O}((n + l^2)\,|\mathcal{C}|)$.

Proof Initialization takes time $\mathcal{O}(|\mathcal{C}|\,l^2)$. Consider now the number of the operations in the innermost loops, i.e., accesses to $\alpha.event\_count$ on lines 18 and 25. In the recognition phase there are $\mathcal{O}(n)$ shifts of the window. In each shift, one new event comes into the window, and one old event leaves the window. Thus, for any episode α, $\alpha.event\_count$ is accessed at most twice during one shift. The cost of the recognition phase is thus $\mathcal{O}(n\,|\mathcal{C}|)$. □

In practice the size l of episodes is very small with respect to the size n of the sequence, and the time required for the initialization can be safely neglected. For injective episodes we have the following tighter result.

Theorem 4.24 The time complexity of recognizing injective parallel episodes in Algorithm 4.19 (excluding initialization) is $\mathcal{O}(\frac{n}{win}\,|\mathcal{C}|\,l + n)$.

Proof Consider *win* successive shifts of one time unit. During such sequence of shifts, each of the $|\mathcal{C}|$ candidate episodes α can undergo at most $2l$ changes: any event type A of α can have $A.count$ increased to 1 and decreased to 0 at most once. This is due to the fact that after an event of type A has come into the window, we have $A.count \geq 1$ for the next *win* time units. Reading the input takes time n. □

Compare this to a trivial non-incremental method where the sequence is pre-processed into windows, and then frequent sets are searched for. The time requirement for recognizing $|\mathcal{C}|$ candidate sets in n windows, plus the time required to read in n windows of size *win*, is $\mathcal{O}(n\,|\mathcal{C}|\,l + n \cdot win)$, i.e., larger by a factor of *win*.

Theorem 4.25 The time complexity of Algorithm 4.21 is $\mathcal{O}(n\,|\mathcal{C}|\,l)$.

Proof The initialization takes time $\mathcal{O}(|\mathcal{C}|\,l + win)$. In the recognition phase, again, there are $\mathcal{O}(n)$ shifts, and in each shift one event comes into the window and one event leaves the window. In one shift, the effort per an episode α depends on the number of automaton instances accessed; there are a maximum of l instances for each episode. The worst-case time complexity is thus $\mathcal{O}(|\mathcal{C}|\,l + win + n\,|\mathcal{C}|\,l) = \mathcal{O}(n\,|\mathcal{C}|\,l)$ (note that *win* is $\mathcal{O}(n)$). □

The input sequence consists in the worst case of events of only one event type, and the candidate serial episodes consist only of events of that particular type. Every shift of the window results now in an update in every automaton instance. This worst-case complexity is close to the complexity of the trivial non-incremental method $\mathcal{O}(n\,|\mathcal{C}|\,l + n \cdot win)$. In practical situations, however, the time requirement is considerably smaller, and we approach the savings obtained in the case of injective parallel episodes.

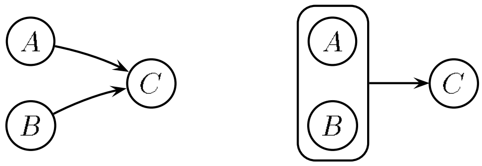
Theorem 4.26 The time complexity of recognizing injective serial episodes in Algorithm 4.21 (excluding initialization) is $\mathcal{O}(n\,|\mathcal{C}|)$.

Proof Each of the $\mathcal{O}(n)$ shifts can now affect at most two automaton instances for each episode: when an event comes into the window there can be a state transition in at most one instance, and at most one instance can be removed because the initializing event goes out of the window. □

4.2.4 General partial orders

So far we have only discussed serial and parallel episodes. We now discuss briefly the use of other partial orders in episodes. The recognition of an arbitrary episode can be reduced to the recognition of a hierarchical combination of serial and parallel episodes. For example, episode γ in Figure 4.4 is a serial combination of two episodes: δ', a parallel episode consisting of A and B, and δ'', an episode consisting of C alone. The occurrence of an episode in a window can be tested using such hierarchical structure: to see whether episode γ occurs in a window one checks (using a method for serial episodes) whether δ' and δ'' occur in this order; to check the occurrence of δ' one uses a method for parallel episodes to verify whether A and B occur.

There are, however, some complications one has to take into account. First, it is sometimes necessary to duplicate an event node to obtain a decomposition to serial and parallel episodes. Consider, for instance, the episode on the left in Figure 4.5. There is no hierarchical composition consisting only



γ δ' δ''

Figure 4.4: Recursive composition of a complex episode.

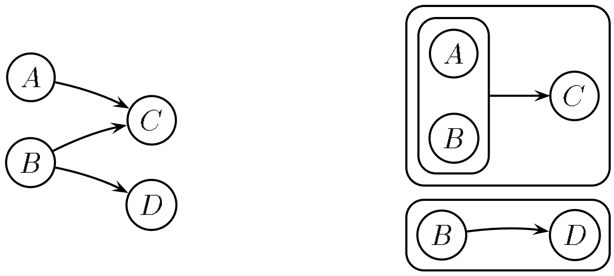


Figure 4.5: Recursive composition of a complex episode.

of serial and parallel episodes. In the composite episode on the right, the node labeled B has been duplicated. Such duplication works with injective episodes, but non-injective episodes need more complex methods. Another important aspect is that composite events have a duration, unlike the elementary events in R.

A practical alternative is to handle all episodes basically like parallel episodes, and to check the correct partial ordering only when all events are in the window. Parallel episodes can be located efficiently; after they have been found, checking the correct partial ordering is relatively fast.

Another interesting approach to the recognition of episodes is to use inverse structures. That is, for each frequent episode we store the identifiers of the windows in which the episode occurs. Then, in the recognition phase, for a candidate episode α we can compute the set of windows in which α occurs as the intersection of the sets of windows for two subepisodes of α. This holds for all but serial episodes, for which some additional information is needed.

4.2.5 Rule generation

Once the frequent episodes and their frequencies are known, one can generate episodes rules, similar to association rules. An episode rule can state, for instance, that if there are events of types A and B in a window, then an event of type C is also in the window (parallel episode rule), or that if there are events E and G in the window and in that order, then an event of type F is between them (serial episode rule). Serial episode rules can point forward or backward in time and, as illustrated by the example, the left-hand side can also have places that are filled by corresponding events on the right hand side. Episode rules can be generated from frequent episodes in the same way that association rules are generated from frequent sets. Extending the association rule methods to deal with multisets and ordered sets is fairly straightforward.

4.3 Experiments

We present experimental results obtained with two telecommunication network fault management databases. The first database $\mathbf{s}_1$ is a sequence of 73 679 alarms covering a time period of 7 weeks. The time granularity is one second. There are 287 different types of alarms with very diverse frequencies and distributions. On the average there is an alarm every minute. The alarms tend, however, occur in bursts: in the extreme cases there are over 40 alarms in one second. We present results from experiments with serial episodes and injective parallel episodes, i.e., the opposite extreme cases of the complexity of the recognition phase.

Performance overview Tables 4.1 and 4.2 give an overview of the discovery of frequent episodes. In Table 4.1, serial episodes and injective parallel episodes have been discovered in $\mathbf{s}_1$ with a fixed frequency threshold 0.003 and a varying window width; in Table 4.2, episodes have been discovered with a fixed window width of 60 seconds and a varying frequency threshold. These ranges for the parameter values have been given by experts of the alarm correlation domain.

The experiments show that the approach is efficient. Running times are between 5 seconds and 8 minutes, in which time hundreds of frequent episodes could be found. The methods are robust in the sense that a change in one parameter only adds or removes some frequent episodes, but does not replace any.

Quality of candidate generation Table 4.3 shows the number of candidate and frequent serial episodes per iteration, with frequency threshold 0.003, and averaged over test runs with window widths $10, 20, 40, 60, 80, 100$, and 120 seconds.

In the first iteration, for size 1, all 287 event types have to be checked. The larger the episodes become, the more combinatorial information there exists to take advantage of. From size 4 up, over one half of the candidates turned out to be frequent.

Window width (s)	Serial episodes Count	Time (s)	Injective parallel episodes Count	Time (s)
10	16	31	10	8
20	31	63	17	9
40	57	117	33	14
60	87	186	56	15
80	145	271	95	21
100	245	372	139	21
120	359	478	189	22

Table 4.1: Results of experiments with $\mathbf{s}_1$ using a fixed frequency threshold of 0.003 and a varying window width.

Frequency threshold	Serial episodes Count	Time (s)	Injective parallel episodes Count	Time (s)
0.1	0	7	0	5
0.05	1	12	1	5
0.008	30	62	19	14
0.004	60	100	40	15
0.002	150	407	93	22
0.001	357	490	185	22

Table 4.2: Results of experiments with $\mathbf{s}_1$ using a fixed window width of 60 s and a varying frequency threshold.

As can be seen from the table, a possible practical improvement is to combine iterations by generating candidate episodes for several iterations at once, and thus avoid reading the input sequence so many times. This pays off in the later iterations, where there are otherwise only few candidates to recognize, and where the match is good.

Scale-up We performed scale-up tests with 1 to 8 fold multiples of the sequence $\mathbf{s}_1$, i.e., sequences with approximately 74 000 to 590 000 events. The results in Figure 4.6 show that the time requirement is linear with respect to the length of the input sequence, as could be expected from the analysis.

Incremental recognition We also tested the efficiency of the database pass, in particular the effect of the incremental recognition. Figure 4.7 presents the ratio of times needed for trivial vs. incremental recognition of candidate episodes. The time required to generate the windows for the trivial

Episode size l	Number of episodes 287^l	Number of candidate episodes	Number of frequent episodes	Match
1	287	287.0	30.1	11 %
2	82 369	1 078.7	44.6	4 %
3	$2 \cdot 10^7$	192.4	20.0	10 %
4	$7 \cdot 10^9$	17.4	10.1	58 %
5	$2 \cdot 10^{12}$	7.1	5.3	74 %
6	$6 \cdot 10^{14}$	4.7	2.9	61 %
7	$2 \cdot 10^{17}$	2.9	2.1	75 %
8	$5 \cdot 10^{19}$	2.1	1.7	80 %
9	$1 \cdot 10^{22}$	1.7	1.4	83 %
10–		17.4	16.0	92 %

Table 4.3: Number of candidate and frequent serial episodes in $\mathbf{s}_1$ with frequency threshold 0.003 and averaged over window widths 10, 20, 40, 60, 80, 100, and 120 s.

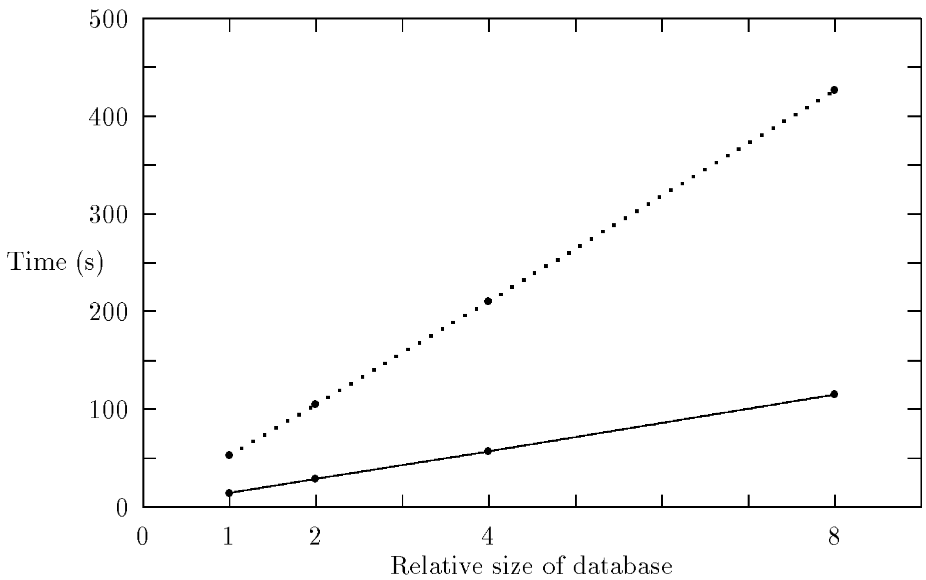


Figure 4.6: Scale-up results for serial episodes (dotted line) and injective parallel episodes (solid line) in $\mathbf{s}_1$ with window width 60 s and frequency threshold 0.01.

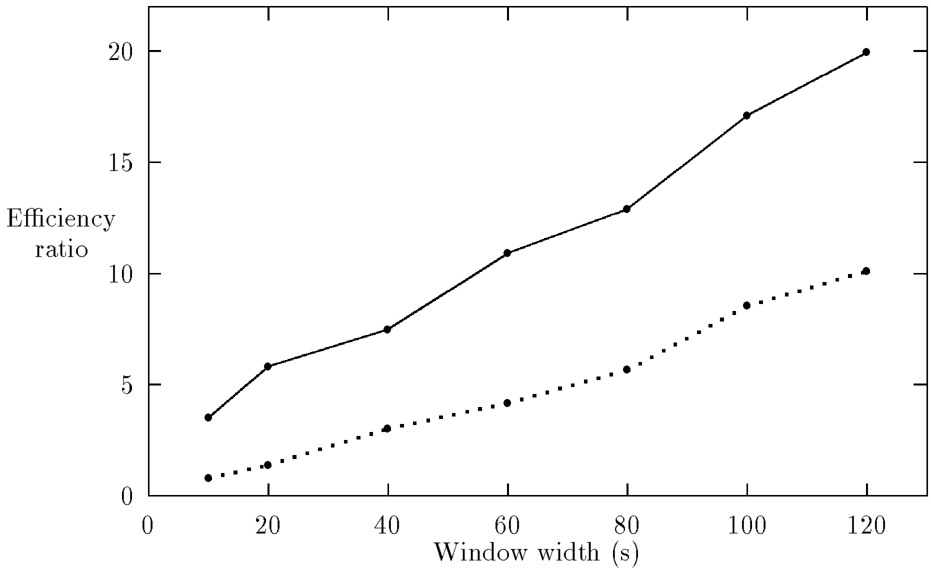


Figure 4.7: Ratio of times needed for trivial vs. incremental recognition methods in $\mathbf{s}_1$ for serial episodes (dotted line) and injective parallel episodes (solid line) as functions of window width.

method has been excluded from the results. The figure shows that the incremental methods are faster by a factor of 1–20, roughly linearly with respect to the window width of 10–120 seconds. This is consistent with the analysis of Algorithm 4.19: for injective parallel episodes the worst-case analysis gave a difference of a factor of *win*. The results indicate that the incremental recognition method is useful in practice also for non-injective serial episodes.

To analyze the effect of the incremental recognition in more detail we conducted the following more controlled tests. We used an alarm database $\mathbf{s}_2$ from a different network; this sequence contains 5000 events covering a time period of 6 days. We ignored the actual times of the events and assumed instead that one alarm had arrived in a time unit. There are 119 event types and the number of their occurrences ranges from 1 to 817. For these tests we considered only injective parallel episodes.

Table 4.4 presents results of test runs with different window widths and frequency thresholds. Results about the efficiency with respect to the number of frequent episodes and candidates are similar to the ones obtained with sequence $\mathbf{s}_1$. The frequency thresholds have in these experiments been higher since the data is dense: there is an event every time unit.

A central outcome of the analysis of the windowing methods was the effect of window width *win* on the time complexity. We examined it with tests where all other factors, in particular the candidate collection, were fixed. Our

Window width	Frequency threshold	Candidate episodes	Frequent episodes	Time (s)
10	0.05	444	84	16
20	0.10	463	161	27
40	0.20	632	346	71
60	0.30	767	488	84
80	0.40	841	581	112
100	0.50	755	529	90
120	0.60	578	397	49
160	0.70	633	479	64

Table 4.4: Results of experiments with sequence $\mathbf{s}_2$.

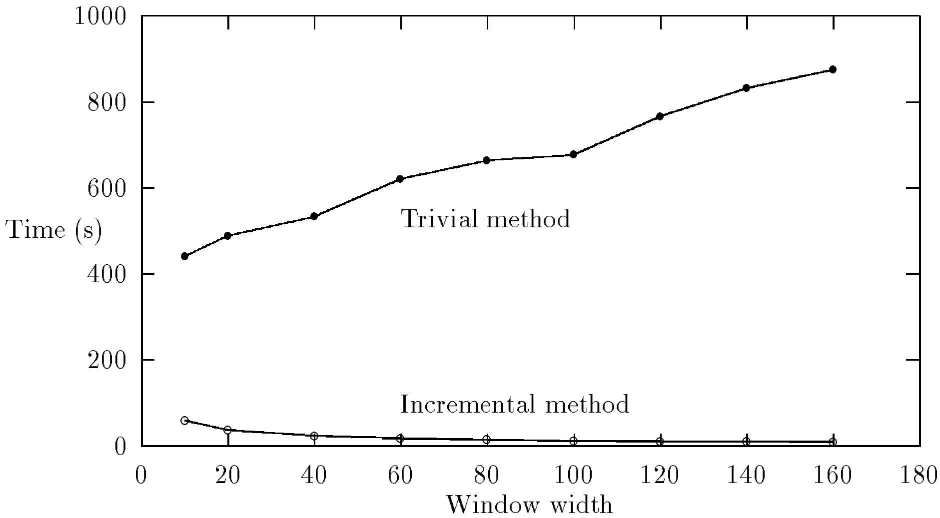


Figure 4.8: Time for the database pass over $\mathbf{s}_2$ as a function of the window width.

collection of candidates consists of 385 episodes of sizes 1 to 11. Figure 4.8 presents total times for recognizing these candidates in the input sequence $\mathbf{s}_2$. Within window widths of 10 to 160 time units, the total time with the trivial method doubles from 400 to 800 seconds. With the incremental method the time is in turn cut from 60 down to 10 seconds. The running time of the trivial method is approximately $3win + 420$, and for the incremental method $700/win + 5$. These results match the time complexity analysis given earlier. In particular, the time complexity of the trivial method is greater by a factor of the window width win; the approximating functions give a factor of $0.6win$. The efficiency ratio was in these experiments better than in the experiments described earlier: the ratio ranges from 6 up to 80.

Chapter 5

Minimal occurrences of episodes

In this chapter we describe an alternative approach to the discovery of episodes. Instead of looking at the windows and only considering whether an episode occurs in a window or not, we now look at the exact occurrences of episodes and the relationships between those occurrences. One of the advantages of this approach is that focusing on the occurrences of episodes allows us to more easily find rules with two window widths, one for the left-hand side and one for the whole rule, such as "if A and B occur within 15 seconds, then C follows within 30 seconds".

We give an introduction to the approach in Section 5.1. Sections 5.2 and 5.3 present the main ideas of the discovery of all frequent episodes and the rule generation, respectively. Finally, experiments are described in Section 5.4. In this chapter our aim is to convey the main ideas of an alternative approach, and we skip detailed proofs and algorithms.

5.1 Outline of the approach

The approach is based on minimal occurrences of episodes. Besides the new rule formulation, the use of minimal occurrences gives rise to the following new method for the recognition of episodes in the input sequence. For each frequent episode we store information about the locations of its minimal occurrences. In the recognition phase we can then compute the locations of minimal occurrences of a candidate episode α as a temporal join of the minimal occurrences of two subepisodes of α. In addition to being simple and efficient, this formulation has the advantage that the confidences and frequencies of rules with a large number of different window widths can be obtained quickly, i.e., there is no need to rerun the analysis if one only wants to modify the window widths. In the case of complicated episodes, the time needed for recognizing the occurrence of an episode can be significant; the use

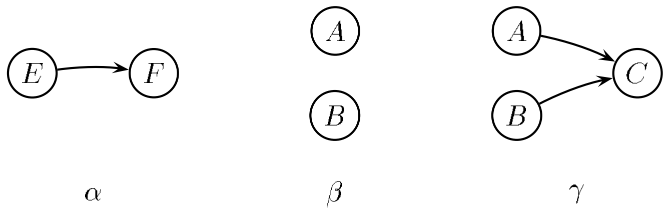


Figure 5.1: Episodes.

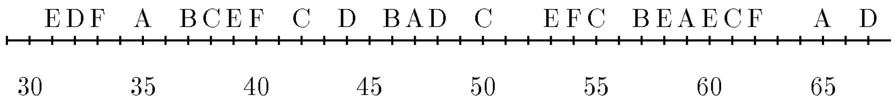


Figure 5.2: The example event sequence $\mathbf{s}$.

of stored minimal occurrences of episodes eliminates unnecessary repetition of the recognition effort.

We identify minimal occurrences with their time intervals in the following way. Given an episode α and an event sequence $\mathbf{s}$, we say that the interval $[t_s, t_e)$ is a *minimal occurrence* of α in $\mathbf{s}$, if (1) α occurs in the window $\mathbf{w} = (w, t_s, t_e)$ on $\mathbf{s}$, and if (2) α does not occur in any proper subwindow on $\mathbf{w}$, i.e., not in any window $\mathbf{w}' = (w', t'_s, t'_e)$ on $\mathbf{s}$ such that $t_s \leq t'_s$, $t'_e \leq t_e$, and $width(\mathbf{w}') < width(\mathbf{w})$. The *set of (intervals of) minimal occurrences* of an episode α in a given event sequence is denoted by $mo(\alpha)$: $mo(\alpha) = \{\, [t_s, t_e) \bigm| [t_s, t_e) \text{ is a minimal occurrence of } \alpha\}$.

Example 5.1 Consider the episodes in Figure 5.1 (reproduced from Figure 3.2) and the event sequence $\mathbf{s}$ in Figure 5.2 (reproduced from Figure 4.1). The parallel episode β consisting of event types A and B has four minimal occurrences in $\mathbf{s}$: $mo(\beta) = \{[35, 38), [46, 48), [47, 58), [57, 60)\}$. The partially ordered episode γ has the following three minimal occurrences: $[35, 39), [46, 51), [57, 62)$. □

An *episode rule* is an expression $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. The interpretation of the rule is that if episode β has a minimal occurrence at interval $[t_s, t_e)$ with $t_e - t_s \leq win_1$, then episode α occurs at interval $[t_s, t'_e)$ for some t'_e such that $t'_e - t_s \leq win_2$. Formally this can be expressed in the following way. Given win_1 and β, denote $mo_{win_1}(\beta) = \{[t_s, t_e) \in mo(\beta) \bigm| t_e - t_s \leq win_1\}$. Further, given α and an interval $[u_s, u_e)$, define $occ(\alpha, [u_s, u_e)) =$ true if and only if there exists a minimal occurrence $[u'_s, u'_e) \in mo(\alpha)$ such that $u_s \leq u'_s$

and $u'_e \leq u_e$. The confidence of an episode rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ is now

$$\frac{|\{[t_s, t_e) \in mo_{win_1}(\beta) \mid occ(\alpha, [t_s, t_s + win_2))\}|}{|mo_{win_1}(\beta)|}.$$

Example 5.2 Continuing Example 5.1, we have, e.g., the following rules and confidences. For the rule $\beta\,[3] \Rightarrow \gamma\,[4]$ we have three minimal occurrences $[35, 38), [46, 48), [57, 60)$ of β of width at most 3 in the denominator. Only one of them, $[35, 38)$, has an occurrence of α within width 4, so the confidence is 1/3. For the rule $\beta\,[3] \Rightarrow \gamma\,[5]$ the confidence is 1. □

Since in a rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ episode β is a subepisode of α, the rule right-hand side α contains information about the relative location of each event in the rule. Thus the "new" events in the rule right-hand side can actually be required to be positioned, e.g., between events in the left-hand side. There is also a number of possible definitions for the temporal relationship between the intervals. For instance, rules that point backwards in time can be defined in a similar way. For brevity, we only consider this one case.

In the previous chapter we defined the frequency of an episode as the fraction of windows that contain the episode. While frequency has a nice interpretation as the probability that a randomly chosen window contains the episode, the concept is not very useful with minimal occurrences: (1) there is no fixed window size, and (2) a window may contain several minimal occurrences of an episode. Instead of frequency we use the concept of *support*, the number of minimal occurrences of an episode: the support of an episode α in a given event sequence $\mathbf{s}$ is $|mo(\alpha)|$. Similarily to a frequency threshold, we now use a threshold for the support: given a support threshold min_sup, an episode α is *frequent* if $|mo(\alpha)| \geq min\_sup$.

The current episode rule discovery task can now be stated as follows. Given an event sequence $\mathbf{s}$, a class $\mathcal{E}$ of episodes, and a set W of time bounds, find all frequent episode rules of the form $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where $\beta, \alpha \in \mathcal{E}$ and $win_1, win_2 \in W$.

5.2 Finding minimal occurrences of episodes

In this section we describe algorithms that locate the minimal occurrences of frequent serial and parallel episodes. Let us start with some observations about the basic properties of episodes. Lemma 4.12 still holds: the subepisodes of a frequent episode are frequent. Thus we can use the main algorithm (Algorithm 4.13) and the candidate generation (Algorithm 4.14) as they are. We also have results about the minimal occurrences of an episode containing minimal occurrences of its subepisodes.

Lemma 5.3 Assume α is an episode and $\beta \preceq \alpha$ is its subepisode. If $[t_s, t_e) \in mo(\alpha)$, then β occurs in $[t_s, t_e)$ and hence there is an interval $[u_s, u_e) \in mo(\beta)$ such that $t_s \leq u_s \leq u_e \leq t_e$.

Lemma 5.4 Let α be a serial episode of size k, and let $[t_s, t_e) \in mo(\alpha)$. Then there are subepisodes α_1 and α_2 of α of size $k-1$ such that $[t_s, t_e^1) \in mo(\alpha_1)$ for some $t_e^1 < t_e$ and $[t_s^2, t_e) \in mo(\alpha_2)$ for some $t_s^2 > t_s$.

Lemma 5.5 Let α be a parallel episode of size k, and let $[t_s, t_e) \in mo(\alpha)$. Then there are subepisodes α_1 and α_2 of α of size $k-1$ such that $[t_s^1, t_e^1) \in mo(\alpha_1)$ and $[t_s^2, t_e^2) \in mo(\alpha_2)$ for some $t_s^1, t_e^1, t_s^2, t_e^2 \in [t_s, t_e]$, and furthermore $t_s = \min\{t_s^1, t_s^2\}$ and $t_e = \max\{t_e^1, t_e^2\}$.

The minimal occurrences of a candidate episode α are located in the following way. In the first iteration of the main algorithm, $mo(\alpha)$ is computed from the input sequence for all episodes α of size 1. In the rest of the iterations, the minimal occurrences of a candidate α are located by first selecting two suitable subepisodes α_1 and α_2 of α, and then computing a temporal join between the minimal occurrences of α_1 and α_2, in the spirit of Lemmas 5.4 and 5.5.

To be more specific, for serial episodes the two subepisodes are selected so that α_1 contains all events except the last one and α_2 in turn contains all except the first one. The minimal occurrences of α are then found with the following specification:

$$\begin{aligned} mo(\alpha) \;=\; \{\, [t_s, u_e) \;\big|\; & \text{ there are } [t_s, t_e) \in mo(\alpha_1) \text{ and} \\ & [u_s, u_e) \in mo(\alpha_2) \text{ such that } t_s < u_s, \\ & t_e < u_e, \text{ and } [t_s, u_e) \text{ is minimal } \}. \end{aligned}$$

For parallel episodes, the subepisodes α_1 and α_2 contain all events except one; the omitted events must be different. See Lemma 5.5 for the idea of how to compute the minimal occurrences of α.

The minimal occurrences of a candidate episode α can be found in a linear pass over the minimal occurrences of the selected subepisodes α_1 and α_2. The time required for one candidate is thus $\mathcal{O}(|mo(\alpha_1)| + |mo(\alpha_2)| + |mo(\alpha)|)$, which is $\mathcal{O}(n)$, where n is the length of the event sequence. To optimize the running time, α_1 and α_2 can be selected so that $|mo(\alpha_1)| + |mo(\alpha_2)|$ is minimized.

The space requirement of the algorithm is $\sum_i \sum_{\alpha \in \mathcal{F}_i} |mo(\alpha)|$, assuming the minimal occurrences of all frequent episodes are stored, or as $\max_i(\sum_{\alpha \in \mathcal{F}_i \cup \mathcal{F}_{i+1}} |mo(\alpha)|)$, if only the current and next levels of minimal occurrences are stored. The size of $\sum_{\alpha \in \mathcal{F}_1} |mo(\alpha)|$ is bounded by n, the number of events in the input sequence, as each event in the sequence is a minimal occurrence of an episode of size 1. In the second iteration, an event in the input sequence can start at most $|\mathcal{F}_1|$ minimal occurrences of episodes of size 2. The space complexity of the second iteration is thus $\mathcal{O}(|\mathcal{F}_1|n)$.

While minimal occurrences of episodes can be located quite efficiently from minimal occurrences, the size of the data structures can be even larger than the original database, especially in the first couple of iterations. A practical solution is to use in the beginning other pattern matching methods, e.g., similar to the ones described in Section 4.2, to locate the minimal occurrences.

5.3 Finding confidences of rules

We now show how the information about minimal occurrences of frequent episodes can be used to obtain confidences for various types of episode rules without looking at the data again.

Recall that we defined an episode rule as an expression $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$, where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. To find such rules, first note that for the rule to be frequent, the episode α has to be frequent. So rules of the above form can be enumerated by looking at all frequent episodes α, and then looking at all subepisodes β of α. The evaluation of the confidence of the rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ can be done in one pass through the structures $mo(\beta)$ and $mo(\alpha)$, as follows.

For each $[t_s, t_e) \in mo(\beta)$ with $t_e - t_s \leq win_1$, locate the minimal occurrence $[u_s, u_e)$ of α such that $t_s \leq u_s$ and $[u_s, u_e)$ is the first interval in $mo(\alpha)$ with this property. Then check whether $u_e - t_s \leq win_2$.

The time complexity of the confidence computation for a given episode and given time bounds win_1 and win_2 is $\mathcal{O}(|mo(\beta)| + |mo(\alpha)|)$. The confidences for all win_1, win_2 in the set W of time bounds can be found, using a table of size $|W|^2$, in time $\mathcal{O}(|mo(\beta)| + |mo(\alpha)| + |W|^2)$. For reasons of brevity we omit the details.

min_sup	Number of frequent episodes and informative rules							
	Time bounds W (s)							
	15, 30		30, 60		60, 120		15, 30, 60, 120	
50	1131	617	2278	1982	5899	7659	5899	14205
100	418	217	739	642	1676	2191	1676	3969
250	111	57	160	134	289	375	289	611
500	46	21	59	49	80	87	80	138

Table 5.1: Experimental results: number of episodes and rules

min_sup	Execution times (s)			
	Time bounds W (s)			
	$15, 30$	$30, 60$	$60, 120$	$15, 30, 60, 120$
50	158	210	274	268
100	80	87	103	104
250	56	56	59	58
500	50	51	51	52

Table 5.2: Experimental results: execution times

5.4 Experimental results

We have experimented with the methods using as test data a part of the WWW server log from the Department of Computer Science at the University of Helsinki. The log contains requests to see WWW pages at the department's server; such requests can be made by WWW browsers at any host in the Internet.

An event in the log can be seen as consisting of the attributes *page, host,* and *time*. The number of events in our data set is 116308, and it covers three weeks in February and March, 1996. In total, 7634 different pages are referred to from 11635 hosts. Requests for images have been excluded from consideration. For simplicity, we only considered the *page* and *time* attributes; we used relatively short time bounds to reduce the probability of unrelated requests contributing to the minimal occurrences.

We experimented with support thresholds *min_sup* between 50 and 500, and with time bounds between 15 s and 2 min. In three cases we used two time bounds, and in one case we searched simultaneously for all combinations of four time bounds in W. Episode rules discovered with these parameters should reveal the paths through which people navigate when they know where they want to go.

Table 5.1 shows the number of frequent episodes and the number of informative rules with confidence at least 0.2. (A rule $\beta\,[win_1] \Rightarrow \alpha\,[win_2]$ is considered informative if its confidence is higher than the confidence of all the rules $\beta\,[win_1'] \Rightarrow \alpha\,[win_2']$ with $win_1' > win_1$, or $win_1' = win_1$ and $win_2' < win_2$.)

The number of frequent episodes is in the range from 40 to 6000, and it seems to grow rather fast when the support threshold becomes lower. Our data is relatively dense, and therefore the effect of the time bounds on the number of frequent episodes is roughly linear. The largest frequent episodes consist of 7 events. Note that the method is robust in the sense that a change in one parameter extends or shrinks the collection of frequent episodes but does not replace any.

Table 5.2 shows the execution times for the experiments on a PC (90 MHz Pentium, 32 MB memory, Linux operating system). The data resided in a 3.0 MB flat text file. The execution times are between 50 s and 5 min. Note, in particular, that searching for episodes with several different time bounds (the right-most columns in the tables) is as fast as searching for episodes with only the largest time bound. Minimal occurrences are thus a very suitable representation for queries with different time bounds.

Following are some examples of the episode rules found (we use the titles of the pages here, or their English translations, which should be self-explanatory). All these rules show users going down in the hierarchy or pages.

- "Department Home Page", "Spring term 96" [15 s] $\Rightarrow$ "Classes in spring 96" [30 s] (confidence 0.83). In other words, in 83 % of the cases where the departmental home page and the spring term page had been accessed within 15 seconds, the classes page was requested within 30 seconds (that is, within 30 seconds from the request for the departmental home page).
- "Research at the department" $\Rightarrow$ "Staff of the department" [2 min] (confidence 0.29). (There is no time bound for the left-hand side since there is only one event.)
- "Department Home Page", "Department Home Page in Finnish", "Classes in spring 96", "Basic courses" [15 s] $\Rightarrow$ "Introduction to Document Preparation (IDP)", "IDP Course Description", "IDP Exercises" [2 min] (confidence 0.42).

Experiments with the alarm data set $\mathbf{s}_1$ of Chapter 4 show that—with comparable parameters—the present method is as fast or faster. The new method has, however, two important advantages: the rule formalism is more useful, and rules with several different time bounds can be found with the same effort.

5.5 Bibliographic notes

Most data mining and machine learning techniques are adapted towards the analysis of unordered collections of data. However, there are important application areas where the data to be analyzed has an inherent sequential structure. Examples of such data are telecommunications network alarms, user interface actions, crimes committed by a person, occurrences of recurrent illnesses, etc. Recently, interest in knowledge discovery from sequences of events has increased: see, e.g., [17, 56, 87, 69, 8]. Episodes were introduced in [64], but the first formulations and results on the discovery of parallel episodes were presented already in [63]. Applications of the methods have been described in [38, 39]. A similar approach to the automatic acquisition of network management knowledge from the existing data has been presented in [31].

Many ideas, for instance the candidate generation method, stem from the discovery of frequent sets and association rules. Various extensions to association rules apply directly or with minor modifications to episodes, too. For instance, these methods can be extended with an event taxonomy by a direct application of the similar extensions to association rules [37, 40, 81]. See Section 2.7 for extensions and work related to association rules.

Technical problems related to the recognition of episodes have been researched in several fields. Taking advantage of the slowly changing contents of the group of recent events has been studied, e.g., in artificial intelligence, where a similar problem in spirit is the many pattern/many object pattern match problem in production system interpreters [24]. Also, comparable strategies using a sliding window have been used, e.g., to study the locality of reference in virtual memory [16]. Our setting differs from these in that our window is a queue with the special property that we know in advance when an event will leave the window; this knowledge is used in the recognition of serial episodes. In the algorithm utilizing minimal occurrences, we take advantage of the fact that we know where subepisodes of candidates have occurred.

The recent work on sequence data in databases (see [78]) provides interesting openings towards the use of database techniques in the processing of queries on sequences. A problem similar to the computation of frequencies occurs also in the area of active databases. There triggers can be specified as composite events, somewhat similar to episodes. In [28] it is shown how finite automata can be constructed from composite events to recognize when a trigger should be fired. This method is not practical for episodes since the deterministic automata could be very large.

The methods for matching sets of episodes against a sequence have some similarities to the algorithms used in string matching (e.g., [32]). In particular, recognizing serial episodes in a sequence can be seen as locating all occurrences of subsequences, or matches of patterns with variable length don't care symbols, where the length of the occurrences is limited by the window width. Learning from a set of sequences has received considerable interest in the field of bioinformatics, where an interesting problem is the discovery of patterns common to a set of related protein or amino acid sequences. The classes of patterns differ from ours; they can be, e.g., substrings with fixed length don't care symbols [46]. Closer to our patterns are those considered in [87]. The described algorithm finds patterns that are similar to serial episodes; however, the patterns have a given minimum length, and the occurrences can be within a given edit distance. Recent results on the pattern matching aspects of recognizing episodes can be found in [12].

The work most closely related to ours is perhaps [4]. There multiple sequences are searched for patterns that are similar to the serial episodes with some extra restrictions and an event taxonomy. More recently, the pattern class has been extended with windowing, some extra time constraints, and an event taxonomy [83].—For a survey on patterns in sequential data, see [56].

In stochastics, event sequence data is often called a marked point process [48]. It should be noted that traditional methods for analyzing marked point processes are ill suited for the cases where the number of event types is large.

There are also some interesting similarities between the discovery of frequent episodes and the work done on inductive logic programming, ILP (see, e.g., [70]); a noticeable difference is caused by the sequentiality of the underlying data model, and the emphasis on time-limited occurrences. Some steps towards convergence have been published recently [14, 15, 60]. The problem of looking for one occurrence of an episode can, in turn, be viewed as a constraint satisfaction problem.

The class of patterns discovered can be easily modified in several directions. First, the methods can be used to analyze several sequences. If the sequences are short and windowing is not meaningful, simpler database passes are sufficient. If windowing is used there is actually a variety of choices for the definition of frequency of an episode in a set of sequences. Second, other windowing strategies could be used, e.g., considering only windows starting every win' time units for some win', or windows starting from every event, or for a serial episode with event types A and B, in this order, only windows starting with an event of type A could be taken into account. The use of minimal occurrences is actually an extension similar to this last idea. Third, other patterns could be searched for, e.g., substrings with fixed length don't care symbols.

Perhaps the most important extensions to the framework are facilities for rule querying and compilation, i.e., methods by which the user could specify the episode class in high-level language and the definition would automatically be compiled into a specialization of the algorithm that would take advantage of the restrictions on the episode class. Other open problems include the combination of episode techniques with intensity models.

Chapter 6

The knowledge discovery process

The process of knowledge discovery aims at the discovery of useful and interesting knowledge. In the alarm analysis, the task of finding frequent episodes in the alarm data is only a part of the KDD process, although a central one. In this chapter we discuss the KDD process in the context of this particular application. We also look at TASA, Telecommunication Alarm Sequence Analyzer, a system for discovering knowledge from telecommunication network alarm databases, and see how it supports the process.

6.1 A KDD framework

In the field of data mining or exploratory data analysis the goal is to discover previously unknown information. That is why it can be hard, or even impossible, to specify beforehand, what is interesting. This is particularly true with telecommunication alarms, because the networks are continuously updated. We concentrate here on *subjective* interestingness, and describe one way of trying to discover the most useful and interesting patterns. (Some simple objective interestingness measures were briefly considered in Section 2.5.)

A KDD process, at a coarse level, consists of the following phases:

1. Understanding the domain.

2. Collecting and cleaning data (selection, transformations etc.).

3. Discovery of patterns.

4. Presentation of the results.

5. Interpretation and utilization of the results.

We follow these general steps, with two distinctive characteristics:

1. In the pattern discovery phase, we aim to find *all* potentially interesting (= frequent) patterns according to rather loose riteria for frequency and confidence.

2. In the presentation phase, the discovered patterns can be explored iteratively and interactively.

Our motivation for discovering a lot of rules at once is that network management experts' requests for different viewpoints to the data can then be responded very quickly: a new pattern discovery phase is not necessary, but simply a new view to the already discovered patterns. By producing all rules at once different views on the data can be created very efficiently in the presentation phase. The idea is that the frequency and confidence thresholds filter effectively non-interesting patterns, and produce only potentially interesting ones. The decision of what is interesting is for the most part left to the domain expert to explore.

We next discuss the data preparation, presentation, and utilization phases of the KDD process.

6.2 Collecting and cleaning the data

The first step in the KDD process is collecting and cleaning the data. In some domains this step can take up to 80 % of the total time needed. In our application the information is already collected to the alarm log, and the data is usually of high quality.

Some problems still remain. One is related to the fact that the time stamps of the alarms are not reliable: there can be differences of up to 3–5 minutes in the synchronization of the clocks. As our goal is to locate regularities that are intimately connected to the temporal aspects of alarms, such errors are problematic. Parallel episodes are reasonably robust with respect to changes in the order of the alarms, but a more general solution would be to use preliminary data analysis for locating which components of the network are likely to have erroneous clocks.

Data transformations take a good deal of time: alarms need to be exported from a fault management database, the appropriate attributes have to be selected, and the data needs to be saved in a form suitable for input for TASA. Selection of relevant attributes is not such a trivial task as it might seem at first. Searching for frequent episodes consisting of alarm types is interesting, but there are other alternatives one should consider. For instance, one can derive a new set of event types by concatenating the information about the alarm type and the network element that sent the alarm. The new events can then look something like *1234_EL1*, meaning that element EL1 sent an alarm of type 1234. Frequent episodes now show connections between alarms from individual elements, e.g., that an alarm 1234 from EL1 is followed by an alarm 3333 from network element EL43. It is useful to try several such variations of the data set, as they give different views to the data and result in partially different findings.

We use the term *alarm predicate* to refer to the (properties of) alarms. Episodes can be discovered in sequences of alarm predicates, where the predicate considered can express the type of the alarm, or some other (combinations) of properties. Association rules can be searched for in unordered collections of alarms, if alarms are seen as rows and alarm predicates as items. For episode rules, the type of the alarm and the sender of the alarm are the most typical predicates. For association rules we consider also predicates such as the priority of the alarm, the day of the week, whether the alarm occurred during office hours or not, etc.

The task of acquiring the required background knowledge is in our case fairly easy. The background knowledge consists mostly of information about the network topology: how the network elements are connected and which subelements they contain. Additionally, the types of the network elements form an inheritance hierarchy that is useful in classifying and presenting the rules. This information is readily available from the telecommunication operators.

6.3 Presentation of results using templates

The presentation of discovered knowledge is a main part of this methodology. In this phase the interesting patterns should be located in large collections of potentially interesting patterns. But what is interesting? How to define it? Many of the patterns discovered in the alarm data are trivial or uninteresting for the network managers:

- **A rule can correspond to prior knowledge or expectations.** For instance, we might know from the network implementation that if an element sends an alarm A, it will also send an explanatory notice B.

- **A rule can refer to uninteresting attributes or attribute combinations.** If the user is trouble shooting a particular network element, then rules about unrelated parts of the network are probably not useful.

- **Rules can be redundant.** Rules may contain alarms of different abstraction levels but actually referring to the same fault.

For the most part, what is interesting depends on the case, and is highly based on the user's personal aims and perspective. Knowledge trivial to one expert may not be trivial to another, but with proper tools each expert may filter the rule collection based on his personal background knowledge, and to correspond to his current needs.

We now present methods for exploring large sets of association and episode rules. In TASA, the user can manipulate the set of patterns using selection and ranking operations, as well as more complex operations for including or excluding certain classes of rules. TASA supports the following types of operations:

1. *Focusing*: presentation only of a subset of rules, according to the templates the user specifies (see below).
2. *Sorting or ranking* of rules according to simple objective measures.
3. *Clustering*: grouping of rules into clusters of rules that have similar effects in the analyzed data set.

In TASA, rules can be selected or rejected from the view by templates, simple but powerful pattern expressions.

Definition 6.1 We define templates as regular expressions that describe, in terms of alarm predicates, the form of rules that are to be shown or not shown. More formally, a *template* is an expression $A_1, \ldots, A_k \Rightarrow A_{k+1}, \ldots, A_l$, where each A_i is either an alarm predicate, the name of an alarm predicate collection, or an expression $C+$ or $C*$, where C is a collection name. Here $C+$ and $C*$ correspond to one or more and zero or more instances of the collection C, respectively. A rule $B_1, \ldots, B_m \Rightarrow B_{m+1}, \ldots, B_n$ matches a template if the rule can be considered to be an instance of the pattern.

With templates, the user can explicitly specify both what is interesting and what is not. To be interesting, a rule has to match a *selective template*. If a rule, however, matches a *rejective template*, it is considered uninteresting. To be presented to the user, a rule must be considered interesting—i.e. match one of the selective templates—and it must not be uninteresting—i.e. not match with any of the unselective templates. □

Figure 6.1 shows the user interface to templates as implemented in TASA. Templates are specified using separate fields for the rule left-hand and right-hand sides. In addition to templates, additional bounds can be set, e.g., for frequency, confidence, and number of alarms in one rule. We next describe some illustrative scenarios that utilizing these ideas.

Example 6.2 Focus can be set to, e.g., day-time alarms by selecting only association rules that contain the predicate "office hours = yes". Or, episode rules containing alarms from separate subnetworks can be obtained by using templates that reject all rules where the senders are in the same subnetwork.

The template concept can be combined with thresholds for rule frequency, confidence, and significance (see Section 2.5). The user may state restrictions such as "rule frequency must be between 5 % and 30 %", "rule confidence must be at least 80 %", and "rule significance must be over 0.95". In this

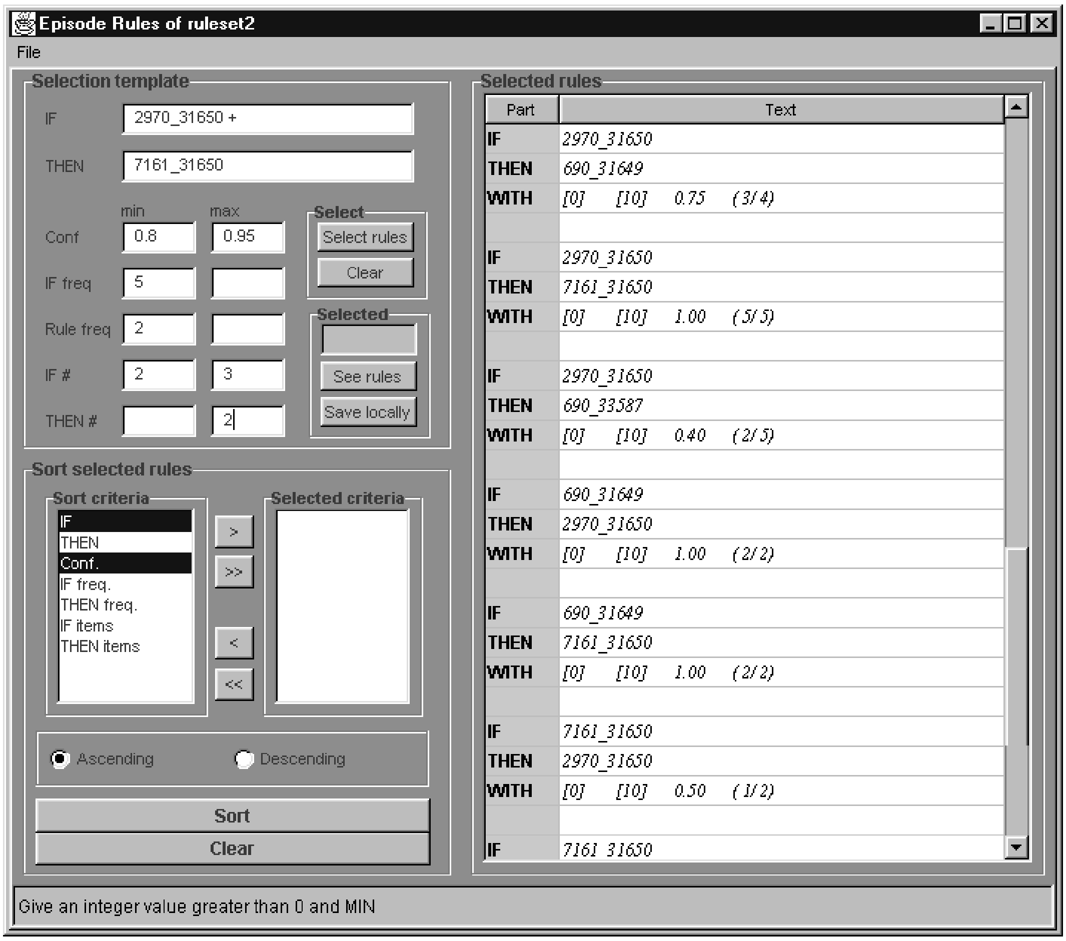


Figure 6.1: Rule Viewing window of TASA: template specification panel on the left, selected rules on the right.

case the user filters out very rare and reasonably frequent rules, and further on selects only those that are both strong and significant. □

Example 6.3 As an example of how the system can be used for off-line network surveillance, consider the following typical scenario. Assume the network manager has used TASA to discover association rules for the current month. First he might want to see what the alarms have been like during the current week, say week 30, so he uses a template to select rules with the predicate "week = 30" as the left-hand side.

The number of selected rules is still very large. The network manager decides to restrict the rule right-hand side to only contain one predicate, and he also sorts the rules by their confidences.

Looking at the selected rules, he sees the rule "if week = 30 then alarm type = *connection failure*" with confidence 0.12, and he infers that an unusually large fraction of alarms during the week has been of type *connection failure*. To see in more detail what the alarms have been like, he refines the template and selects rules with "week = 30 and alarm type = *connection failure*" as the left-hand side.

Looking at the new set of selected rules, the network manager sees that a lot of rules concern the network element *EL1*. That reminds him of maintenance undertaken in the beginning of the week that explains those rules. To remove the rules, he applies a rejective template with the predicate "network element = *EL1*".

The resulting set of rules shows nothing special, but just to make sure the network manager wants to compare the rules with the corresponding rules from some previous week. He opens a copy of the window, and changes the first template to "week = 29". If there is anything special or interesting, the viewing criteria can be refined or altered again. □

Clustering of rules aims at giving a larger picture of the behavior of the alarm sequence. In the data there are often various explanations for the occurrence of a particular alarm type, say *path unavailable*. Clustering methods can be used to assign rules to groups so that two rules with the right-hand side *path unavailable* belong to the same cluster if they often explain or predict *path unavailable* in similar situations. This can be useful in pointing out potentially related rules.

In addition to looking and manipulating the discovered knowledge, the users want to be able to use several types of views into the data. They want to see the discovered knowledge, but they also want to be able to see how that knowledge is actually supported by the original data. TASA links rules, alarms, and data together by hyperlinks.

Visualization of information is obviously an important part of KDD applications. For this the TASA system offers some simple facilities. As an example, consider the confidence of a rule. It is only a crude measure of how well the rule manages to predict the occurrence of the right-hand side, and a more complete picture of the interaction between the left and right-hand sides of a rule can be obtained by simply drawing a histogram showing the distance from each occurrence of the left-hand side to the nearest occurrence of the right-hand side. Such histograms are valuable guides for locating possible periodic relationships between the left and right-hand sides, as is demonstrated by Figure 6.2.

6.4 Experiences with TASA, episodes, and associations

Different versions of TASA have been in prototype use in four telecommunication companies since the beginning of 1995. TASA has been found useful

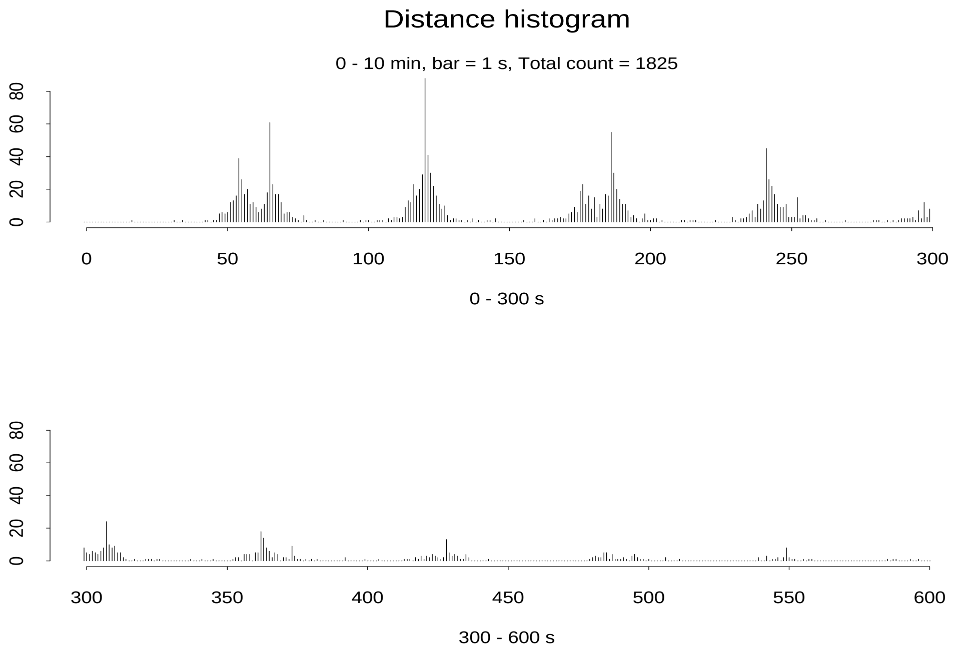


Figure 6.2: Distance histogram

in, e.g., finding long-term, rather frequently occurring dependencies, creating an overview of a short-term alarm sequence, and evaluating the alarm data base consistency and correctness.

Unexpected dependencies have been found, e.g., between network elements which are not closely connected in the network topology. An example of such a dependency is that when a remote device sends alarms, the fault is reflected to another corner of the network through several devices, and not always necessarily via the same routes and devices. So, just analyzing the neighboring devices might not reveal any strong relationships. However, when a larger region is analyzed, such a relationship can be detected. Beginning from the first tests, discovered rules have been integrated into alarm correlation systems.

On the other hand, many of the rules discovered by TASA are deemed trivial by the network managers. Some of the rules correspond to the knowledge that the network managers have about the behavior of the network, and some other rules reflect the assumed functioning of network devices. Luckily, much of the trivial knowledge can be expressed and removed with templates. Templates are also useful since the knowledge trivial to one expert may not be trivial to another, and with templates each expert may filter the rule collection based on his/her personal background knowledge.

The usability of discovery tools has an essential, often perhaps under-estimated role. The usability of an early version of TASA was tested in the usability laboratory of the Helsinki University of Technology. The tests contained, e.g., user tests taken by four fault management experts from telecommunication companies. In the tests, TASA was generally acknowledged as appealing. On the other hand, first-time users were unfamiliar with many concepts from the knowledge discovery field. Despite these problems with the terminology, the system as a whole got encouraging comments.

Overall, TASA has been considered useful. Episode rules are being used as first drafts of correlation rules, whereas association rules are more typically used for creating short-term overviews in off-line network surveillance. Telecommunication operators are integrating these methods to their alarm analysis and surveillance systems.

6.5 Bibliographic notes

The problem of locating a small set of truly interesting information is a generic problem in data mining (see, e.g., [74]): it is hard to know which aspects of the discovered knowledge really interest the user. Kloesgen [53], for instance, defines a number of criteria for interestingness, following the ones presented in [25]: evidence (statistical significance), non-redundancy, novelty (deviation from prior knowledge), simplicity (syntactical complexity of a finding), and generality (the fraction of the population the finding refers to).

The idea of discovering a large number of potentially interesting patterns can be contrasted with numerous methods, e.g., in machine learning, which are more focused and produce one or at most few patterns that match the given problem specification. These methods usually require that the searched or learned subject is quite carefully described in advance, and they leave any other potentially interesting phenomena hidden. The advantage of these systems is that the patterns they find are more expressive than the relatively simple association and episode rules, and focusing the pattern discovery is thus more important.

TASA uses association and episode rules, but the basic idea—iteration in the pattern presentation phase—can be applied to any formalisms that have some similar properties as association and episode rules:

- There is an algorithm that produces lots of potentially interesting patterns.
- The time requirement for discovering all potentially interesting patterns is not considerably longer than if the discovery was focused to a small subset of the potentially interesting patterns.
- The desired focus is not known definitely in advance.

This approach which allows the user to set the focus, and where the user has a total, explicit control over the resulting rule set, is in many ways similar to the declarative language bias used by some ILP (Inductive Logic Programming) systems, e.g., Claudien [76]. In ILP systems, however, the focus (bias) is usually set before the discovery.

Templates were introduced in [52]. In the Explora system, Hoschka and Kloesgen [41] already have used patterns similar to templates for defining what is interesting, and their ideas have influenced the work on templates. Their approach is based on few fixed statement types and a partial ordering of attributes, whereas templates are closer to regular expressions.

Sometimes considerable amounts of rules remain, even when the user has found the desired focus with the described methods. Automatic pruning, ordering, and structuring methods should at this point be available for invocation by the user, especially for removal of redundancy.

A method for pruning a set of strong association or episode rules by removing redundancy has been presented in [85]. The method is based on computing a *rule cover*. A rule cover is a subset of rules that has predictive power equal to the original set of rules, in the sense that whenever a rule of the original set mathces a row, then there is a rule in the cover that matches the row.

An approximate order of interestingness could be assigned to the discovered rules by giving weights to templates: positive weights to selective templates and negative weights to rejective templates. The ranking of rules would then correspond to the sum of weights of matched templates.

Clustering has been proposed as a method for structuring a set of association or episode rules [85]. The idea is to take rules with the same right-hand side, and to group the rules so that rules in the same cluster tend to match the same cases in the database. Clustering of rules has a remote connection to clustering of database rows.

The approaches taken by other KDD systems include, e.g., the following. Explora [41, 53] finds interesting instances of statistical patterns. In Explora, the pattern discovery phase is focused by the user. The system selects and presents the best patterns to the user, and, based on the results, the user can change the focus and repeat the pattern discovery. The patterns discovered by 49er [89] are contingency tables, equations, and logical equivalences. The user can interactively change the focus, e.g., independent and dependent variables, and require for a new pattern discovery. The Key Finding Reporter (Kefir) [65, 75] discovers and explains deviations, and gives recommendations for corrective actions. Applications of Kefir are tailored with a lot of domain knowledge to be aware of the interestingness criteria, corrective actions, etc., of the domain. Given a database from the domain, a Kefir-based application produces a report of the deviations without iteration.

Chapter 7

Discovery of all frequent patterns

In this part we consider a generalization of the problems of discovering all frequent sets or all frequent episodes. The problem we look at is the following: given a set of patterns, a selection criterion, and a database, find those patterns that satisfy the criterion in the database. In this chapter we present a framework for this problem and give a generic levelwise algorithm for solving it, e.g., in the case where the selection criterion is based on the frequency of patterns. We then analyze the algorithm and the problem in Chapter 8.

We start this chapter by describing the discovery task in Section 7.1. We give an algorithm for this task in Section 7.2. In Section 7.3 we give examples of the setting in various knowledge discovery tasks, and we show that, instead of frequency, other criteria can be used for selecting rules. Finally, in Section 7.4, we outline an extension of the setting and the algorithm for discovery in several database states.

7.1 The discovery task

We start by defining the knowledge discovery setting we consider in this chapter. Given a set of patterns, i.e., a class of expressions about databases, and a predicate to evaluate whether a database satisfies a pattern, the task is to determine which patterns are satisfied by a given database.

Definition 7.1 Assume that $\mathcal{P}$ is a set and q is a predicate $q : \mathcal{P} \times \{\mathbf{r} \mid \mathbf{r} \text{ is a database}\} \to \{\text{true}, \text{false}\}$. Elements of $\mathcal{P}$ are called *patterns* and q is a *selection criterion* over $\mathcal{P}$. Given a pattern φ in $\mathcal{P}$ and a database $\mathbf{r}$, we say that φ is *selected* if $q(\varphi, \mathbf{r})$ is true. Since the selection criterion is often based on the frequency of the pattern, we use the term *frequent* as a synonym for "selected". Given a database $\mathbf{r}$, the *theory* $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ of $\mathbf{r}$ with respect to $\mathcal{P}$ and q is $\mathcal{T}(\mathcal{P}, \mathbf{r}, q) = \{\varphi \in \mathcal{P} \mid q(\varphi, \mathbf{r}) \text{ is true}\}$. □

Example 7.2 The problem of finding all frequent item sets can be described as a task of discovering frequent patterns in a straightforward way. Given a set R, a binary database r over R, and a frequency threshold *min_fr*, the set $\mathcal{P}$ of patterns consists of all item sets, i.e., $\mathcal{P} = \{X \mid X \subseteq R\}$, and for the selection criterion we have $q(\varphi, r) = \text{true}$ if and only if $fr(\varphi, r) \geq min\_fr$. □

Note that we do not specify any satisfaction relation for the patterns of $\mathcal{P}$ in $\mathbf{r}$: this task is taken care of by the selection criterion q. For some applications, "$q(\varphi, \mathbf{r})$ is true" could mean that φ occurs often enough in $\mathbf{r}$, that φ is true or almost true in $\mathbf{r}$, or that φ defines, in some way, an interesting property or subgroup of $\mathbf{r}$. Obviously, the task of determining the theory of $\mathbf{r}$ is not tractable for arbitrary sets $\mathcal{P}$ and predicates q. If, for instance, $\mathcal{P}$ is infinite and $q(\varphi, \mathbf{r})$ is true for infinitely many patterns, an explicit representation of $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ cannot be computed.

In the discovery tasks considered here the aim is to find all patterns that are selected by a relatively simple criterion—such as exceeding a frequency threshold—in order to efficiently identify a space of potentially interesting patterns. Other criteria can then be used for further pruning and processing of the patterns. Consider as an example the discovery of association rules: first frequent sets are discovered, then all rules with sufficient frequency are generated, and a confidence threshold is used to further prune the rules.

The task of discovering frequent sets has two noteworthy properties. First, all frequent sets are needed for the generation of association rules. It is not sufficient to know just the largest frequent sets, although they determine the collection of all frequent sets. The second important property is that the selection criterion, i.e., frequency, is monotone decreasing with respect to expansion of the set. We consider only the situation where the predicate q is monotone with respect to a given partial order on the patterns.

Definition 7.3 Let $\mathcal{P}$ be a set of patterns, q a selection criterion over $\mathcal{P}$, and $\preceq$ a partial order on the patterns in $\mathcal{P}$. If for all databases $\mathbf{r}$ and patterns $\varphi, \theta \in \mathcal{P}$ we have that $q(\varphi, \mathbf{r})$ and $\theta \preceq \varphi$ imply $q(\theta, \mathbf{r})$, then $\preceq$ is a *specialization relation* on $\mathcal{P}$ with respect to q. If we have $\theta \preceq \varphi$, then φ is said to be *more special* than θ and θ to be *more general* than φ. If $\theta \preceq \varphi$ and not $\varphi \preceq \theta$ we write $\theta \prec \varphi$. □

Example 7.4 The set inclusion relation $\subseteq$ is a specialization relation for frequent sets. For instance, if the set $\{A, B, C\}$ is frequent, then its subset $\{A, C\}$ must also be frequent.

In more general, given two item sets $X, Y \subseteq R$, the set X is more general, $X \preceq Y$, if and only if $X \subseteq Y$. That is, $X \preceq Y$ implies that if the more specific set Y is frequent then the more general set X is frequent, too. □

For practical purposes the specialization relation has to be computable, i.e., given patterns φ and θ in $\mathcal{P}$, it must be possible to determine whether $\varphi \preceq \theta$. Typically, the specialization relation $\preceq$ is a restriction of the converse of the semantic implication relation: if $\theta \preceq \varphi$, then φ implies θ. If the predicate q is defined in terms of, e.g., statistical significance, then the semantic implication relation is not a specialization relation with respect to q: a pattern can be statistically significant even when a more general pattern is not. Recall that the predicate q is not meant to be the only way of identifying the interesting patterns; a threshold for the statistical significance can be used to further prune patterns found using q.

7.2 The generic levelwise algorithm

In this section we present an algorithm for the task of discovering all frequent patterns in the case where there exists a computable specialization relation between patterns. We use the following notation for the relative speciality of patterns.

Definition 7.5 Given a specialization relation $\preceq$ on patterns in $\mathcal{P}$, the *level* of a pattern φ in $\mathcal{P}$, denoted *level*(φ), is 1 if there is no θ in $\mathcal{P}$ for which $\theta \prec \varphi$. Otherwise *level*(φ) is $1+L$, where L is the maximum level of patterns θ in $\mathcal{P}$ for which $\theta \prec \varphi$. The collection of frequent patterns of level l is denoted by $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q) = \{\varphi \in \mathcal{T}(\mathcal{P}, \mathbf{r}, q) \mid \mathit{level}(\varphi) = l\}$. □

Algorithm 7.6, analogical to Algorithm 2.14, finds all frequent patterns. It works in a levewise or breadth-first manner, starting with the set $\mathcal{C}_1$ of the most general patterns, and then generating and evaluating more and more special candidate patterns. The algorithm prunes those patterns that cannot be frequent given all the frequent patterns obtained in earlier iterations.

The algorithm is generic: details depending on the specific types of patterns and data are left open, and instances of the algorithm must specify these. The levelwise algorithm aims at minimizing the number of evaluations of q on line 5. As with the frequent set discovery algorithm, the computation to determine the candidate collection does not involve the database at all.

Theorem 7.7 Algorithm 7.6 works correctly.

Proof We show by induction on l that $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$ is computed correctly for all l. For $l = 1$, the collection $\mathcal{C}_l$ contains all patterns of level one (line 1), and collection $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$ is then correctly computed (line 5).

For $l > 1$, assume the collections $\mathcal{T}_i(\mathcal{P}, \mathbf{r}, q)$ have been computed correctly for all $i < l$. Note first that $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q) \subseteq \mathcal{C}_l$. Namely, consider any pattern φ in $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$: we have *level*($\varphi$) $= l$ and thus for all patterns $\theta \prec \varphi$ we have *level*(θ) $< l$. Since $\mathcal{T}_i(\mathcal{P}, \mathbf{r}, q)$ has been computed for each $i < l$, each $\theta \prec \varphi$

Algorithm 7.6

Input: A database schema $\mathbf{R}$, a database $\mathbf{r}$ over $\mathbf{R}$, a finite set $\mathcal{P}$ of patterns, a computable selection criterion q over $\mathcal{P}$, and a computable specialization relation $\preceq$ on $\mathcal{P}$.

Output: The set $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ of all frequent patterns.

Method:

1. compute $\mathcal{C}_1 := \{\varphi \in \mathcal{P} \mid \mathit{level}(\varphi) = 1\}$;
2. $l := 1$;
3. **while** $\mathcal{C}_l \neq \emptyset$ **do**
4. // Database pass:
5. compute $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q) := \{\varphi \in \mathcal{C}_l \mid q(\varphi, \mathbf{r})\}$;
6. $l := l + 1$;
7. // Candidate generation:
8. compute $\mathcal{C}_l := \{\varphi \in \mathcal{P} \mid \mathit{level}(\varphi) = l$ and $\theta \in \mathcal{T}_{\mathit{level}(\theta)}(\mathcal{P}, \mathbf{r}, q)$ for all $\theta \in \mathcal{P}$ such that $\theta \prec \varphi\}$;
9. **for** all l **do** output $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$;

---

is correctly in $\mathcal{T}_{\mathit{level}(\theta)}(\mathcal{P}, \mathbf{r}, q)$, and so φ is put into $\mathcal{C}_l$ (line 8). The collection $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$ is then computed correctly on line 5.

Finally note that for every $\varphi \in \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ there is an iteration where the variable l has value $\mathit{level}(\varphi)$. By the definition of level, there are more general patterns $\theta \prec \varphi$ on every level less than $\mathit{level}(\varphi)$, and since $\preceq$ is a specialization relation they are all frequent, so the ending condition $\mathcal{C}_l = \emptyset$ is not true with $l \leq \mathit{level}(\varphi)$. □

The input specification of the algorithm states that the set $\mathcal{P}$ is finite. Actually, it does not always need to be finite: the algorithm works correctly as long as the number of candidate patterns is finite. There are some desirable properties for the specialization relation $\preceq$. An efficient method for accessing the more specific and more general patterns on neighboring levels is useful, or otherwise finding the collection of valid candidates may be expensive.

7.3 Examples

We now look at the applicability of Algorithm 7.6 for some knowledge discovery tasks. We consider three problems as tasks of discovering patterns that are selected by a given predicate: the discovery of association rules and frequent episodes, and the discovery of exact database rules.

7.3.1 Association rules

Recall that the task of discovering association rules can be split into two: the discovery of frequent sets, and the generation of rules. We summarize below the necessary specifications for discovering frequent sets with Algorithm 7.6.

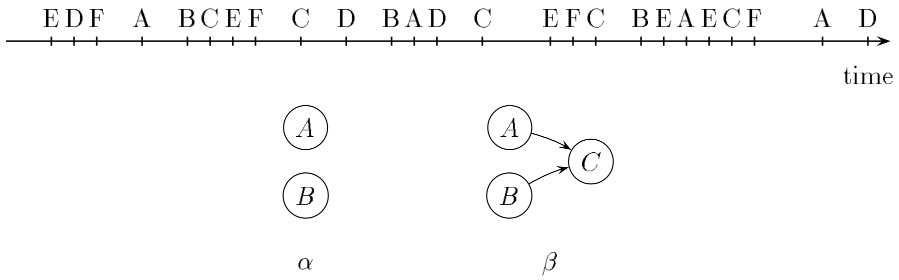


Figure 7.1: An event sequence and two episodes.

Given are a set R, a binary database r over R, and a frequency threshold *min_fr*. We want to find the frequent sets, so the set $\mathcal{P}$ of patterns consists of all item sets: $\mathcal{P} = \{X \mid X \subseteq R\}$. Our selection criterion q is based on the frequency of an item set, so that q is true for a set $X \in \mathcal{P}$ if and only if $fr(X, r) \geq min\_fr$. Obviously the set inclusion relation $\subseteq$ is monotone with respect to frequency, so it can be used as a specialization relation $\preceq$.

With these specifications Algorithm 7.6 proceeds as Algorithm 2.14. Note in particular that candidate generation on line 8 of Algorithm 7.6 gives a result equivalent to the candidate collection of Definition 2.13.

7.3.2 Episodes

Consider the problem of recognizing frequent episodes in sequences of events. Figure 7.1 depicts a sequence of events and two episodes that occur several times in the sequence. Episode α contains two events, A and B, but does not specify any order for them. Episode β contains additionally an event C, and states that C must occur after both A and B.

Recall from Chapter 4 that the task is to discover all episodes whose frequency exceeds a given threshold *min_fr*. Given a window width *win*, the frequency of α in a given event sequence $\mathbf{s}$ is defined as the fraction of windows of length *win* on $\mathbf{s}$ that contain an instance of α.

The discovery of all frequent episodes can be solved with the levelwise algorithm. The subepisode relation $\preceq$ is a specialization relation on episodes. That is, for $\alpha = (V, \leq, g)$ and $\beta = (V', \leq', g')$ we have $\beta \preceq \alpha$, if and only if (1) $V' \subseteq V$ (after possible renaming of nodes), (2) for all $v \in V'$ we have $g'(v) = g(v)$, and (3) for all $v, w \in V'$ with $v \leq' w$ also $v \leq w$. The relation $\alpha \preceq \beta$ holds for the episodes in Figure 7.1. In the case of arbitrary partial orders, Algorithm 7.6 is more conservative than Algorithm 4.13. Algorithm 7.6 considers a totally ordered episode consisting of events A and B as candidate only, if the trivially ordered episode consisting of A and B is frequent. Algorithm 4.13, in turn, was specified to increase the size of episodes by one in each iteration.

7.3.3 Exact database rules

Exact database rules [74] are a rule formalism that is somewhat more general than association rules: numerical and categorical attributes are considered. On the other hand, the confidence of exact rules must be 1; a small variation gives *strong rules* which can have a confidence less than 1. Before introducing exact database rules formally, we define the notion of a taxonomy on an attribute.

Definition 7.8 Given an attribute A, a *taxonomy* on A is a set $T(A)$ such that $Dom(A) \subseteq T(A)$, where $Dom(A)$ is the domain of A, and such that there is a partial order is-a on $T(A)$. We assume that is-a is reflexive and that there is a special member *any* in the taxonomy such that for all $a \in T(A)$ we have a is-a *any*. □

Example 7.9 Consider an attribute *department* for which the domain $Dom(department)$ is $\{dept\_1,\ dept\_2,\ \ldots,\ dept\_15\}$. Now a taxonomy $T(department)$ could consist of $Dom(department) \cup \{management\_dept,$ $production\_dept,\ sales\_dept,\ any\}$, i.e., of names of departments and their types. The partial order is-a could then classify each department to its type by defining that *dept_1* is-a *management_dept*, *dept_2* is-a *management_dept*, that for $i = 3, \ldots, 11$ *dept_i* is-a *production_dept*, and that for $i = 12, \ldots, 15$ *dept_i* is-a *sales_dept*. Additionally, for every *dept* in $T(department)$ we have *dept* is-a *dept* and *dept* is-a *any*. □

Definition 7.10 Let r be a relation over a relation schema R. Assume taxonomies are given for the non-numerical attributes in R. A *simple condition* on a row t in r is either of the form $a_1 \leq t[A] \leq a_2$, where $A \in R$ is a numerical attribute and $a_1, a_2 \in Dom(A)$, or of the form $t[A]$ is-a a, where $A \in R$ is non-numerical and has a taxonomy $T(A)$, and $a \in T(A)$.

An *exact database rule* is now an expression of the form $C_1 \Rightarrow C_2$, where both C_1 and C_2 are simple conditions. The rule $C_1 \Rightarrow C_2$ *holds* in r if C_2 is true on every row t of r that C_1 is true on. □

Example 7.11 Assume a relation r represents employees. Consider only attributes *department*, as in the previous example, and *age*, and assume that the domain of *age* is $\{18, 19, \ldots, 65\}$. The exact database rule

$$t[department] \text{ is-a } management\_dept \Rightarrow 43 \leq t[age] \leq 65$$

tells that employees in the management departments are at least 43 years old. □

Given a relation r and taxonomies for non-numerical attributes, the collection of exact database rules that hold in r is the theory $\mathcal{T}(\mathcal{P}, r, q)$, where the set $\mathcal{P}$ of patterns consists of all possible exact rules φ, and the predicate

$q(\varphi, r)$ is true if and only if φ holds in r. Next we show what is a specialization relation for exact rules.

Theorem 7.12 The following relation $\preceq$ is a specialization relation with respect to the set $\mathcal{P}$ of all possible exact database rules:

$$(C_1 \Rightarrow C_2) \preceq (C_1' \Rightarrow C_2') \text{ if and only if } C_1' \sqsubseteq C_1 \text{ and } C_2 \sqsubseteq C_2',$$

where $\sqsubseteq$ is a partial order on simple conditions defined as follows:

$$\begin{array}{l}(a_1 \leq t[A] \leq a_2) \sqsubseteq (b_1 \leq t[B] \leq b_2) \text{ if and only if} \\ A = B \text{ and } [b_1, b_2] \subseteq [a_1, a_2]\end{array}$$

and

$$(t[A] \textit{ is-a } a) \sqsubseteq (t[B] \textit{ is-a } b) \text{ if and only if } A = B \text{ and } b \textit{ is-a } a.$$

Proof Denote by $\mathcal{M}(C)$ the set of rows on which condition C is true. By the definition, the relation $\sqsubseteq$ on simple conditions has the following property: if C_1 is true on a row t, then every $C_2 \sqsubseteq C_1$ is true on t, i.e., $\mathcal{M}(C_1) \subseteq \mathcal{M}(C_2)$, and $\sqsubseteq$ is actually a specialization relation on simple conditions.

Assume the exact database rule $C_1' \Rightarrow C_2'$ holds, i.e., $\mathcal{M}(C_1') \subseteq \mathcal{M}(C_2')$. Consider now any more general rule $(C_1 \Rightarrow C_2) \preceq (C_1' \Rightarrow C_2')$. From the properties of $\sqsubseteq$ it follows that $\mathcal{M}(C_1) \subseteq \mathcal{M}(C_1')$ and $\mathcal{M}(C_2') \subseteq \mathcal{M}(C_2)$. Thus $\mathcal{M}(C_1) \subseteq \mathcal{M}(C_2)$, i.e., the rule $(C_1 \Rightarrow C_2)$ holds. $\square$

The proof shows that the specialization relation $\preceq$ is a restriction of the converse of the semantic implication: for any two patterns φ and θ, if we have $\varphi \preceq \theta$ then θ implies φ. Intuitively, the specialization relation means here that once we have an exact database rule that holds, we know that a modified rule where the left-hand side only matches a subset of rows must hold as well, and that if the right-hand side matches a superset of rows, the modified rule must also hold.

Algorithm 7.6 would start with those rules that are most likely to hold, and then loosen the conditions on the left-hand sides while tightening the conditions on the right-hand sides.

Example 7.13 Assume the relation r represents employees. Consider only attributes *department*, as in the previous example, and *age*, and assume that the domain of *age* is $\{18, 19, \ldots, 65\}$.

The most general patterns considered by Algorithm 7.6 are such as

$$39 \leq t[\mathit{age}] \leq 39 \Rightarrow t[\mathit{department}] \textit{ is-a } \mathit{any}$$

and

$$t[\mathit{department}] \textit{ is-a } \mathit{dept\_7} \Rightarrow 18 \leq t[\mathit{age}] \leq 65.$$

These and a number of other obvious rules hold. Later, when more meaningful rules are dealt with, the specialization relation prunes rules from consideration in the following way. If, for instance, the rule

$$t[\mathit{department}] \text{ is-a } \mathit{dept\_2} \Rightarrow 18 \leq t[\mathit{age}] \leq 40$$

does not hold, then rules such as

$$t[\mathit{department}] \text{ is-a } \mathit{dept\_2} \Rightarrow 18 \leq t[\mathit{age}] \leq 39$$

and

$$t[\mathit{department}] \text{ is-a } \mathit{management\_dept} \Rightarrow 18 \leq t[\mathit{age}] \leq 40$$

cannot hold. □

Note that the task of discovering exact database rules cannot be split into two phases like the discovery of association rules, where frequent sets, i.e., the rule components, are discovered first. Namely, in the case of exact rules there is no minimum threshold for the frequency of rules.

When strong, i.e., almost always correct rules are searched for, the "almost always correctness" needs to be carefully defined, or the partial order $\preceq$ given above is not a specialization relation. The following example demonstrates this.

Example 7.14 Consider the discovery of strong rules, and the use of a confidence threshold *min_conf*, defined as with association rules, as a means for determining whether a rule is strong or not. If the rule

$$t[\mathit{department}] \text{ is-a } \mathit{dept\_7} \Rightarrow 40 \leq t[\mathit{age}] \leq 50$$

has a confidence close to but below the threshold *min_conf*, then the rule

$$t[\mathit{department}] \text{ is-a } \mathit{any} \Rightarrow 40 \leq t[\mathit{age}] \leq 50$$

might actually be strong, e.g., if all employees in other departments than *dept_7* are between 40 and 50 years old. □

Algorithm 7.6 considers in each database pass a collection of candidate rules, where all the rules are on the same level. The KID3 algorithm [74] for discovering exact database rules, in turn, considers in one iteration all rules with the same attribute on the left-hand side. KID3 does not directly evaluate q on all those rules; instead, it stores some summary information from which rules that hold can be extracted. Both approaches have their drawbacks. The space requirement of the summaries in KID3 is in the worst case linear in the database size. Algorithm 7.6, in turn, does not take any advantage of the fact that rules close to each other in the specialization relation are similar, and could be evaluated efficiently together.

Almost always lots of redundant exact and strong rules hold. For exact rules, for instance, giving the most specific rules that hold would be sufficient, since the rest of the rules are implied by these. Recall, again, that the purpose is to find the rules that hold, and then use other methods to select the most useful ones—the specificity is certainly one criterion, but not the only one [74].

7.4 Discovery in several database states

Knowledge discovery is sometimes criticized for analyzing just one database state. The critics say that it does not give reliable information: it is often impossible to know if a regularity exists in the analyzed database only by chance, or if it is true in most database states. Next we describe how Algorithm 7.6 can be adopted to discover those patterns that are selected by the given criterion in most of the given database states. We define the global selection criterion of a pattern φ to depend on the number of database states where φ is selected.

Definition 7.15 Given a selection criterion q over a set $\mathcal{P}$ of patterns and a *frequency threshold* $\mathit{min\_fr}$, the *global selection criterion* Q is a predicate

$$Q : \mathcal{P} \times \{\boldsymbol{r} \mid \boldsymbol{r} \text{ is a set of databases}\} \to \{\text{true, false}\},$$

such that for any set $\boldsymbol{r} = \{\mathbf{r}_1, \ldots, \mathbf{r}_n\}$ of databases we have $Q(\varphi, \boldsymbol{r}) = \text{true}$ if and only if $|\{i \mid q(\varphi, \mathbf{r}_i)\}| \geq \mathit{min\_fr} \cdot n$. The theory $\mathcal{T}(\mathcal{P}, \boldsymbol{r}, Q)$ is called the *almost always true theory* of $\boldsymbol{r}$ with respect to $\mathcal{P}$, q, and $\mathit{min\_fr}$. $\square$

Note that any partial order $\preceq$ on $\mathcal{P}$ that is a specialization relation with respect to q is also a specialization relation with respect to Q. We have the following theorem.

Theorem 7.16 Let $\mathcal{P}, q$, and Q be as in Definition 7.15, and let $\preceq$ be a specialization relation with respect to q. Then $\preceq$ is a specialization relation also with respect to the global selection criterion Q.

Proof By definition the relation $\preceq$ is a specialization relation with respect to Q, if for all sets $\boldsymbol{r}$ of databases and all $\varphi, \theta \in \mathcal{P}$ we have that $Q(\varphi, \boldsymbol{r})$ and $\theta \preceq \varphi$ imply $Q(\theta, \boldsymbol{r})$. To see that this is the case, consider a pattern φ for which $Q(\varphi, \boldsymbol{r})$ holds. For each $\mathbf{r}_i$ in $\boldsymbol{r}$ for which $q(\varphi, \mathbf{r}_i)$ holds, $q(\theta, \mathbf{r}_i)$ must hold for all more general patterns $\theta \preceq \varphi$, and thus $Q(\theta, \boldsymbol{r})$ must hold. $\square$

Since $\preceq$ is a specialization relation with respect to the global selection criterion Q, Algorithm 7.6 can be applied directly for knowledge discovery from several database states; just use the global selection criterion Q instead of q. The evaluation of Q on $\boldsymbol{r}$ consists now of evaluating q on the individual database states $\mathbf{r}_i \in \boldsymbol{r}$. It turns out that we can use here the specialization relation $\preceq$ both locally and globally.

Consider Algorithm 7.6 running with a set $\boldsymbol{r} = \{\mathbf{r}_1, \ldots, \mathbf{r}_n\}$ of database states and the global selection criterion Q as its inputs. Candidate patterns φ in the algorithm will be such that all more general patterns than φ are globally selected. Such patterns are then evaluated in each database state, in order to find out if they are selected in by q in sufficiently many database states. However, it is possible that in some database state $\mathbf{r}_i \in \boldsymbol{r}$ patterns more general than φ are not selected by q, and correspondingly φ cannot be selected by q in $\mathbf{r}_i$. A key to a more efficient evaluation of the global selection criterion is thus to generate candidates also locally in each database state, and only to evaluate patterns that are candidates both globally and locally.

To be more specific, at level l the global candidate collection $\mathcal{C}_l$ contains those patterns that are potentially selected by Q in $\boldsymbol{r}$. The local candidate collections, denoted by $\mathcal{C}_l^i$, contain for each database state $\mathbf{r}_i$ those patterns that are potentially selected by q in $\mathbf{r}_i$. During the evaluation of Q, for each database state $\mathbf{r}_i$ we evaluate the predicate q on the intersection $\mathcal{C}_l \cap \mathcal{C}_l^i$ of global and local candidates.

By using information about the local candidates we can further eliminate evaluations of q. Namely, the global candidate collection $\mathcal{C}_l$ may contain such patterns that are not candidates in sufficiently many collections $\mathcal{C}_l^i$. This can be the situation for a pattern $\varphi \in \mathcal{C}_l$ when the more general patterns $\theta \prec \varphi$ are selected too often in disjoint database states. Such useless evaluation of candidates can be avoided by a simple check: a pattern $\varphi \in \mathcal{C}_l$ needs not to be evaluated if $|\{i \mid \mathcal{C}_l^i\}| < \mathit{min\_fr} \cdot n$. A similar check can be applied after each failed evaluation of $q(\varphi, \mathbf{r}_i)$, in order to prune a candidate as soon as it turns out that it cannot be globally selected.

In summary, using only information about the global selection criterion of patterns we would at level l investigate the patterns in $\mathcal{C}_l$ against each database state $\mathbf{r}_i$. Looking at each database state $\mathbf{r}_i$ locally would enable us to investigate the patterns in $\mathcal{C}_l^i$. Combining local and global information, we see that one has to investigate at most the patterns in $\mathcal{C}_l \cap \mathcal{C}_l^i$.

This method could be used to analyze, e.g., the same database over time, in order to see what regularities hold in most of the database states, or to analyze several similar databases, for instance to find out which association rules hold in most of the stores of a supermarket chain.

Chapter 8

Complexity of finding frequent patterns

We now analyze the complexity of finding all frequent patterns, and we also derive results for the complexity of discovering all frequent sets. In Section 8.1 we introduce the concept of the border between frequent and non-frequent patterns. This notion turns out to be useful in the analysis of the generic algorithm in Section 8.2. Inspired by this, we give in Section 8.3 a guess-and-correct algorithm for the task of finding all frequent patterns. We then analyze the task in Section 8.4. In Section 8.5 we return to the concept of border, and show that it has strong connections to transversals on hypergraphs.

8.1 The border

Consider the theory $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ of some set $\mathcal{P}$ of patterns. The whole theory can be specified by giving only the maximally specific patterns in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$: every pattern more general than any of those is selected by q, and the rest are not. The collection of maximally specific patterns in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and, correspondingly, the collection of *minimally* specific (i.e., maximally general) patterns *not* in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ are useful in the analysis of the generic algorithm and the problem. For this purpose we introduce the notion of border.

Definition 8.1 Let $\mathcal{P}$ be a set of patterns, $\mathcal{S}$ a subset of $\mathcal{P}$, and $\preceq$ a partial order on $\mathcal{P}$. Further, let $\mathcal{S}$ be closed downwards under the relation $\preceq$, i.e., if $\varphi \in \mathcal{S}$ and $\gamma \preceq \varphi$, then $\gamma \in \mathcal{S}$. The *border* $\mathcal{B}d(\mathcal{S})$ of $\mathcal{S}$ consists of those patterns φ such that all more general patterns than φ are in $\mathcal{S}$ and no pattern more specific than φ is in $\mathcal{S}$:

$$\begin{aligned}\mathcal{B}d(\mathcal{S}) \quad = \quad & \{\varphi \in \mathcal{P} \mid \text{for all } \gamma \in \mathcal{P} \text{ such that } \gamma \prec \varphi \text{ we have } \gamma \in \mathcal{S}, \text{ and} \\ & \text{for all } \theta \in \mathcal{P} \text{ such that } \varphi \prec \theta \text{ we have } \theta \notin \mathcal{S}\}.\end{aligned}$$

Those patterns φ in $\mathcal{B}d(\mathcal{S})$ that are in $\mathcal{S}$ are called the *positive border* $\mathcal{B}d^+(\mathcal{S})$,

$$\mathcal{B}d^+(\mathcal{S}) = \{\varphi \in \mathcal{S} \mid \text{for all } \theta \in \mathcal{P} \text{ such that } \varphi \prec \theta \text{ we have } \theta \notin \mathcal{S}\},$$

and those patterns φ in $\mathcal{B}d(\mathcal{S})$ that are not in $\mathcal{S}$ are the *negative border* $\mathcal{B}d^-(\mathcal{S})$,

$$\mathcal{B}d^-(\mathcal{S}) = \{\varphi \in \mathcal{P} \setminus \mathcal{S} \mid \text{for all } \gamma \in \mathcal{P} \text{ such that } \gamma \prec \varphi \text{ we have } \gamma \in \mathcal{S}\}.$$

□

In other words, the positive border consists of the most specific patterns in $\mathcal{S}$, the negative border consists of the most general patterns outside $\mathcal{S}$, and the border is the union of these two sets. Note that a set $\mathcal{S}$ that is closed downwards can be described by giving just the positive or the negative border. Consider, e.g., the negative border. No pattern θ such that $\varphi \preceq \theta$ for some φ in the negative border is in $\mathcal{S}$, while all other patterns are in $\mathcal{S}$.

A theory $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ is always closed downwards with respect to a specialization relation, and the concept of border can be applied on the set of all frequent patterns.

Example 8.2 Consider the discovery of frequent sets with items $R = \{A, \ldots, F\}$. Assume the collection $\mathcal{F}$ of frequent sets is

$$\{\{A\}, \{B\}, \{C\}, \{F\}, \{A, B\}, \{A, C\}, \{A, F\}, \{C, F\}, \{A, C, F\}\}.$$

The negative border of this collection contains now sets that are not frequent, but whose all subsets are frequent, i.e., minimal non-frequent sets. The negative border is thus

$$\mathcal{B}d^-(\mathcal{F}) = \{\{D\}, \{E\}, \{B, C\}, \{B, F\}\}.$$

The positive border, in turn, contains the maximal frequent sets, i.e.,

$$\mathcal{B}d^+(\mathcal{F}) = \{\{A, B\}, \{A, C, F\}\}.$$

□

Example 8.3 Consider the discovery of frequent episodes in a sequence over events $A, \ldots, D$. Assume that arbitrary partial orders are allowed in the episodes, and that the episodes in Figure 8.1 constitute the theory $\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr})$ of frequent episodes in some given event sequence $\mathbf{s}$.

There are only two maximally specific episodes, those presented in Figure 8.2. All other episodes in $\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr})$ are included in either of these two that constitute the positive border $\mathcal{B}d^+(\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr}))$. The negative border $\mathcal{B}d^-(\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr}))$ is depicted in Figure 8.3. It consists of the maximally general episodes that are not frequent. □

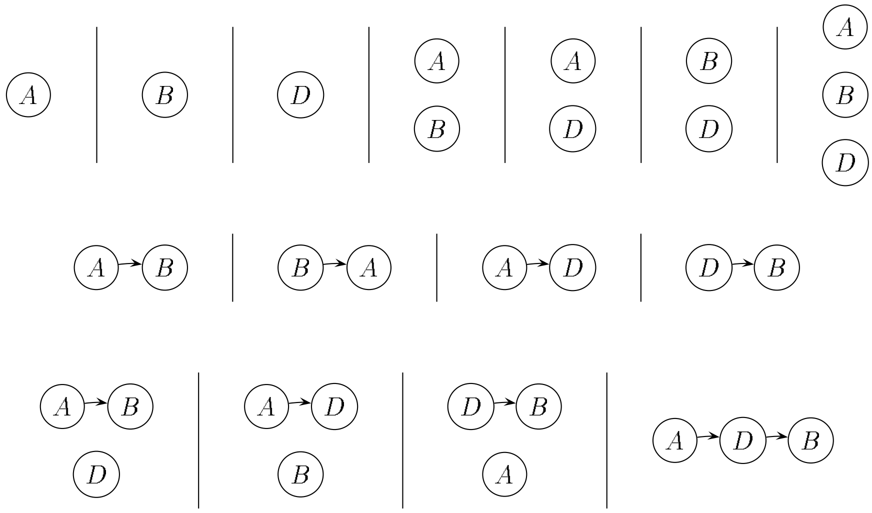


Figure 8.1: A collection $\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr})$ of frequent episodes.

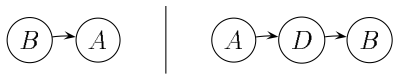


Figure 8.2: The positive border $\mathcal{B}d^+(\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr}))$.

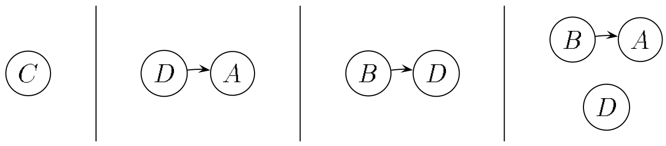


Figure 8.3: The negative border $\mathcal{B}d^-(\mathcal{F}(\mathbf{s}, \mathit{win}, \mathit{min\_fr}))$.

8.2 Complexity of the generic algorithm

Consider the complexity of discovering all frequent patterns, in terms of the number of evaluations of the selection criterion q. The trivial method for finding $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ is to test all patterns of $\mathcal{P}$, and hence use $|\mathcal{P}|$ evaluations of q. Algorithm 7.6 evaluates only patterns in the result, i.e., frequent patterns, and patterns in the negative border of the collection of frequent patterns.

Theorem 8.4 Let $\mathcal{P}, \mathbf{r}$, and q be as in Algorithm 7.6. Algorithm 7.6 evaluates the predicate q exactly on the patterns in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q) \cup \mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$.

Proof First note that no pattern is evaluated more than once: each pattern has a unique level and can therefore be at most in one collection $\mathcal{C}_l$. Recall now line 8 of Algorithm 7.6, the specification of the candidate collection:

8. compute $\mathcal{C}_l := \{\varphi \in \mathcal{P} \mid \mathit{level}(\varphi) = l$ and $\theta \in \mathcal{T}_{\mathit{level}(\theta)}(\mathcal{P}, \mathbf{r}, q)$ for all $\theta \in \mathcal{P}$ such that $\theta \prec \varphi\}$;

We show that $\bigcup_l \mathcal{C}_l = \mathcal{T}(\mathcal{P}, \mathbf{r}, q) \cup \mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$. First note that every pattern φ in $\mathcal{C}_l$ is in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q) \cup \mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$. If φ is selected by q, it is in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. If φ is not selected by q, it is in $\mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$, since all patterns more general than φ are in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$, and φ itself is not.

Now note that every pattern φ in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q) \cup \mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$ is in $\mathcal{C}_{\mathit{level}(\varphi)}$. When $l = \mathit{level}(\varphi)$, all more general patterns than φ have been evaluated in earlier iterations since their levels are less than $\mathit{level}(\varphi)$. All more general patterns $\theta \prec \varphi$ are in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and thus in $\mathcal{T}_{\mathit{level}(\theta)}(\mathcal{P}, \mathbf{r}, q)$. So when $l = \mathit{level}(\varphi)$, the pattern φ will be in $\mathcal{C}_{\mathit{level}(\varphi)}$.

Finally, it can be shown, as in the proof of Theorem 7.7, that $\mathcal{C}_{\mathit{level}(\varphi)}$ is constructed for each φ in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q) \cup \mathcal{B}d^-(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$. □

What the candidate generation step of Algorithm 7.6 basically does is to compute the negative border of frequent patterns found so far. Line 8 actually equals the following specification:

8. compute $\mathcal{C}_l := \mathcal{B}d^-(\bigcup_{i<l} \mathcal{T}_i(\mathcal{P}, \mathbf{r}, q)) \setminus \bigcup_{i<l} \mathcal{C}_i$;

That is, in each iteration the candidate collection generated is exactly the negative border of the patterns selected so far, minus patterns already found not to be selected. We can now use the concept of negative border to restate the complexity of the frequent set discovery algorithm in a compact form.

Corollary 8.5 Given a set R, a binary database r over R, and a frequency threshold *min_fr*, Algorithm 2.14 evaluates the frequency of sets in $\mathcal{F}(r, \mathit{min\_fr}) \cup \mathcal{B}d^-(\mathcal{F}(r, \mathit{min\_fr}))$.

Proof The claim follows directly from the fact that Algorithm 2.14 is an instance of Algorithm 7.6 and from Theorem 8.4. □

Example 8.6 Consider the discovery of frequent sets in a random relation over 20 attributes, where the probability that $t[A]$ has value 1 is p for all rows t and all attributes A. Table 8.1 presents the sizes of the theory and its positive and negative borders in experiments with two random relations, with $p = 0.2$ and $p = 0.5$, and with 1000 rows. In such relations the size of the border seems to be roughly 2 to 4 times the number of frequent sets.

$\mathcal{B}d(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$ can be small for a large theory $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. An experiment with frequent sets in a real database gave the following results. We discovered all frequent sets in a course enrollment database where there is a row per student, a column per each course offered, and a row has value 1 in a column

p	*min_fr*	$\lvert\mathcal{T}(\mathcal{P},\mathbf{r},q)\rvert$	$\lvert\mathcal{B}d^+(\mathcal{T}(\mathcal{P},\mathbf{r},q))\rvert$	$\lvert\mathcal{B}d^-(\mathcal{T}(\mathcal{P},\mathbf{r},q))\rvert$
0.2	0.01	469	273	938
0.2	0.005	1291	834	3027
0.5	0.1	1335	1125	4627
0.5	0.05	5782	4432	11531

Table 8.1: Experimental results with random data sets.

min_fr	$\lvert\mathcal{T}(\mathcal{P},\mathbf{r},q)\rvert$	$\lvert\mathcal{B}d^+(\mathcal{T}(\mathcal{P},\mathbf{r},q))\rvert$	$\lvert\mathcal{B}d^-(\mathcal{T}(\mathcal{P},\mathbf{r},q))\rvert$
0.08	96	35	201
0.06	270	61	271
0.04	1028	154	426
0.02	6875	328	759

Table 8.2: Experimental results with a real data set.

if the corresponding student took the corresponding course. There are 127 courses, and a student has taken 4.3 courses on average; the number of students is 4734. Table 8.2 shows that the size of the border behaves nicely with respect to the size of the theory. □

8.3 The guess-and-correct algorithm

Algorithm 7.6 starts by evaluating the selection predicate q on the most general sentences, and moves gradually to more specific sentences. As q is assumed to be monotone with respect to the specialization relation, this approach is safe in the sense that no statement satisfying q will be overlooked. However, the approach can be quite slow, if there are frequent patterns that are far from the bottom of the specialization relation, i.e., if there are patterns φ that are selected by q, but which appear in the candidate set $\mathcal{C}_i$ only for a large i. As every iteration of the algorithm requires an investigation of the database, this means that such sentences will be discovered slowly.

An alternative is to start the process of finding $\mathcal{T}(\mathcal{P},\mathbf{r},q)$ from an initial guess $\mathcal{S} \subseteq \mathcal{P}$, and then correcting the guess by looking at the database. The guess can be obtained, e.g., from computing the set $\mathcal{T}(\mathcal{P},\mathbf{r},q)$, where $\mathbf{s} \subseteq \mathbf{r}$ is a sample of $\mathbf{r}$. Algorithm 8.7 computes $\mathcal{T}(\mathcal{P},\mathbf{r},q)$ as follows. To begin with, the algorithm is given a guess $\mathcal{S} \subseteq \mathcal{P}$. The algorithm first evaluates the sentences in the positive border $\mathcal{B}d^+(\mathcal{S})$ and removes from $\mathcal{S}$ those that are not interesting. These evaluation and removal steps are repeated until the positive border only contains sentences satisfying q, and

Algorithm 8.7 The *guess-and-correct algorithm* for finding all potentially interesting sentences with an initial guess $\mathcal{S}$.
Input: A database $\mathbf{r}$, a language $\mathcal{P}$ with specialization relation $\preceq$, a selection predicate q, and an initial guess $\mathcal{S} \subseteq \mathcal{P}$ for $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. We assume $\mathcal{S}$ is closed under generalizations..
Output: The set $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$..
Method:

$\mathcal{C}^* := \emptyset$;
// correct $\mathcal{S}$ downward:
$\mathcal{C} := \mathcal{B}d^+(\mathcal{S})$;
while $\mathcal{C} \neq \emptyset$ **do**
    $\mathcal{C}^* := \mathcal{C}^* \cup \mathcal{C}$;
    $\mathcal{S} := \mathcal{S} \setminus \{\varphi \in \mathcal{C} \mid q(\mathbf{r}, \varphi) \text{ is false}\}$;
    $\mathcal{C} := \mathcal{B}d^+(\mathcal{S}) \setminus \mathcal{C}^*$;
od;
// now $\mathcal{S} \subseteq \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$; expand $\mathcal{S}$ upwards:
$\mathcal{C} := \mathcal{B}d^-(\mathcal{S}) \setminus \mathcal{C}^*$;
while $\mathcal{C} \neq \emptyset$ **do**
    $\mathcal{C}^* := \mathcal{C}^* \cup \mathcal{C}$;
    $\mathcal{S} := \mathcal{S} \cup \{\varphi \in \mathcal{C} \mid q(\mathbf{r}, \varphi) \text{ is true}\}$;
    $\mathcal{C} := \mathcal{B}d^-(\mathcal{S}) \setminus \mathcal{C}^*$;
od;
output $\mathcal{S}$;

---

thus we have $\mathcal{S} \subseteq \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. The variable $\mathcal{C}^*$ is used to avoid evaluating sentences twice. Then the algorithm expands $\mathcal{S}$ upwards, like Algorithm 7.6: it evaluates such sentences in the negative border $\mathcal{B}d^-(\mathcal{S})$ that have not been evaluated yet, and adds those that satisfy q to $\mathcal{S}$. Again, the algorithm repeats the evaluation and addition steps until there are no sentences to evaluate. Finally, the output is $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$.

The following results are straightforward.

Lemma 8.8 Algorithm 8.7 works correctly.

Theorem 8.9 Algorithm 8.7 uses at most

$$|(\mathcal{S} \triangle \mathcal{T}) \cup \mathcal{B}d(\mathcal{T}) \cup \mathcal{B}d^+(\mathcal{S} \cap \mathcal{T})|$$

evaluations of q, where $\mathcal{T} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$.

How to obtain good original guesses $\mathcal{S}$? One fairly widely applicable method is sampling. Take a small sample $\mathbf{s}$ from $\mathbf{r}$, compute $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and use it as $\mathcal{S}$. Applied to association rules this method produces extremely good results. Basically, with a high probability one can discover the association rules holding in a database using only a single pass through the database.

Another method for computing an initial approximation can be derived from the Partition algorithm [77]. The idea is to divide $\mathbf{r}$ into small datasets

$\mathbf{r}_i$ which can be handled in main memory, and to compute $\mathcal{S}_i = \mathcal{T}(\mathcal{P}, \mathbf{r}_i, q)$. In the case of frequent sets, use as the guess $\mathcal{S}$ the union $\bigcup_i \mathcal{S}_i$. The guess is a superset of $\mathcal{T}(\mathcal{P}, \mathbf{r}_i, q)$, and executing the first half of Algorithm 8.7 suffices.

8.4 Problem complexity

Let us now analyze the complexity of the problem of discovering all frequent patterns. Consider first the following verification problem: assume somebody gives a set $\mathcal{S} \subseteq \mathcal{P}$ and claims that $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. How many evaluations of q are necessary for verifying this claim? The following theorem shows that the whole border $\mathcal{B}d(\mathcal{S})$ must be inspected.

Theorem 8.10 Let $\mathcal{P}$ and $\mathcal{S} \subseteq \mathcal{P}$ be sets of patterns, $\mathbf{r}$ a database, q a selection criterion, and $\preceq$ a specialization relation. If the database $\mathbf{r}$ is accessed only using the predicate q, then determining whether $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ (1) requires in the worst case at least $|\mathcal{B}d(\mathcal{S})|$ evaluations of q, and (2) can be done in exactly $|\mathcal{B}d(\mathcal{S})|$ evaluations of q.

Proof We show that it is sufficient and in the worst case necessary to evaluate the border $\mathcal{B}d(\mathcal{S})$. The claims follow then from this.

First assume that patterns in the border are evaluated. If and only if every pattern in $\mathcal{B}d^+(\mathcal{S})$ and no pattern in $\mathcal{B}d^-(\mathcal{S})$ is selected by q, then $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$, by the definition of the border. If $\mathcal{S}$ and $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ do not agree on the border, then clearly $\mathcal{S} \neq \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$.

Now assume that less than $|\mathcal{B}d(\mathcal{S})|$ evaluations have been made, all consistent with the claim $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$; then there is a pattern φ in the border $\mathcal{B}d(\mathcal{S})$ for which q has not been evaluated. Now there is no way of knowing whether φ is in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ or not. The evaluations made for other patterns give no information about φ: since they were consistent with $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and φ is in the border, all more general patterns $\theta \prec \varphi$ are selected by the definition of border, none of the more specific patterns is selected, and the rest are irrelevant with respect to φ. In other words, any set in the negative border can be swapped to the positive border, and vice versa, without changing the truth of q for any other set. □

Corollary 8.11 Let $\mathcal{P}$ be a set of patterns, $\mathbf{r}$ a database, q a selection criterion, and $\preceq$ a specialization relation. Any algorithm that computes $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and accesses the data only with the predicate q must evaluate q on the patterns in $\mathcal{B}d(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))$.

Proof The claim follows directly from the proof of Theorem 8.10. □

Example 8.12 Continuing Example 8.2 consider the discovery of frequent sets in a relation over attributes $R = \{A, \ldots, F\}$. Assume a sample or some person tells us that the collection of frequent sets is

$$\mathcal{S} = \{\{A\}, \{B\}, \{C\}, \{F\}, \{A, B\}, \{A, C\}, \{A, F\}, \{C, F\}, \{A, C, F\}\}.$$

To verify whether the claim is true we have to evaluate, according to Corollary 8.11, the border $\mathcal{B}d(\mathcal{S})$. We thus have to evaluate $\mathcal{B}d^+(\mathcal{S}) = \{\{A, B\}, \{A, C, F\}\}$ and $\mathcal{B}d^-(\mathcal{S}) = \{\{D\}, \{E\}, \{B, C\}, \{B, F\}\}$ to determine whether $\mathcal{S}$ is the collection of frequent sets. □

We have further corollaries about the complexity of discovery tasks. Consider the discovery of frequent sets. Recall that Algorithm 2.14 evaluates also the frequency of the candidates that are not frequent, i.e., sets in $\mathcal{B}d^-(\mathcal{F}(r, \mathit{min\_fr}))$. The following corollary shows that in a limited but reasonable model of computation the evaluation of the non-frequent candidates is inevitable.

Corollary 8.13 Given a set R, a binary database r over R, and a frequency threshold *min_fr*, finding the collection $\mathcal{F}(r, \mathit{min\_fr})$ using only queries of the form "Is $X \subseteq R$ frequent in r" requires that sets in the negative border $\mathcal{B}d^-(\mathcal{F}(r, \mathit{min\_fr}))$ are evaluated.

Proof The claim follows directly from Corollary 8.11. □

Algorithm 2.14 actually evaluates the whole theory $\mathcal{F}(r, \mathit{min\_fr})$, not only its border. For the discovery of association rules this is in general necessary, since the exact frequencies are needed in the rule generation phase. Algorithm 2.14 is thus optimal under the simplifying restriction that the only way of controlling the algorithm is to use the information whether a set is frequent or not.

The advantage of Corollary 8.11 is that the border $\mathcal{B}d(\mathcal{S})$ can be small even for large $\mathcal{S}$. The drawback is that it can be difficult to determine the border. We return to this issue in Section 8.5 where we show a connection between the problem of finding the border and the hypergraph transversal problem.

8.5 Computing the border

We now return to the verification problem: given $\mathcal{P}$, $\mathbf{r}$, q, and a set $\mathcal{S} \subseteq \mathcal{P}$, verify whether $\mathcal{S} = \mathcal{T}(\mathcal{P}, \mathbf{r}, q)$. By Corollary 8.11 the border $\mathcal{B}d(\mathcal{S})$ must be inspected. Given $\mathcal{S}$, we can compute $\mathcal{B}d^+(\mathcal{S})$ without looking at the data $\mathbf{r}$ at all: simply find the most specific patterns in $\mathcal{S}$. The negative border $\mathcal{B}d^-(\mathcal{S})$ is also defined by $\mathcal{S}$, and can be determined without looking at the data, but finding the most general patterns in $\mathcal{P} \setminus \mathcal{S}$ can be difficult. We now show how minimal transversals of hypergraphs can be used to determine the negative border.

Definition 8.14 Let R be a set. A collection $\mathcal{H}$ of subsets of R is a *simple hypergraph* on R, if no element of $\mathcal{H}$ is empty and if $X, Y \in \mathcal{H}$ and $X \subseteq Y$ imply $X = Y$. The elements of $\mathcal{H}$ are called the *edges* of the hypergraph, and the elements of R are the *vertices* of the hypergraph. $\square$

Definition 8.15 Given a simple hypergraph $\mathcal{H}$ on a set R, a *transversal* T of $\mathcal{H}$ is a subset of R intersecting all the edges of $\mathcal{H}$: T is a transversal if and only if $T \cap X \neq \emptyset$ for all $X \in \mathcal{H}$. A *minimal transversal* of $\mathcal{H}$ is a transversal T such that no $T' \subset T$ is a transversal. We denote the collection of minimal transversals of $\mathcal{H}$ by $\mathcal{T}r(\mathcal{H})$. $\square$

For our purposes, hypergraphs and transversals apply almost directly on the pattern class of frequent sets. Let the items in R be the vertices and the *complements* of the sets in the positive border be the edges of a simple hypergraph $\mathcal{H}$. So, for each set X in the positive border we have the set $R \setminus X$ as an edge in $\mathcal{H}$. Consider now a set $Y \subseteq R$. If there is an edge $R \setminus X$ such that $Y \cap (R \setminus X) = \emptyset$, then $Y \subseteq X$, and Y is frequent. On the other hand, if there is no such edge that the intersection is empty, then Y cannot be frequent. That is, Y is not frequent if and only if Y is a transversal of $\mathcal{H}$. Minimal transversals are now the minimal non-frequent sets, i.e., the negative border.

In general, for using hypergraphs and transversals to determine the negative border, we need to represent patterns in $\mathcal{P}$ as sets. Frequent sets are such a representation themselves; next we give the requirements for the general case.

Definition 8.16 Let $\mathcal{P}$ be a set of patterns, $\preceq$ a specialization relation, and R a set. A function $f : \mathcal{P} \to \{X \mid X \subseteq R\}$ is a *set representation* of $\mathcal{P}$ and $\preceq$, if f is bijective, f and its inverse are computable, and for all $\theta, \varphi \in \mathcal{P}$ we have $\theta \preceq \varphi$ if and only if $f(\theta) \subseteq f(\varphi)$.

For notational convenience, given a collection $\mathcal{S}$ of sets, we write $f(\mathcal{S}) = \{f(X) \mid X \in \mathcal{S}\}$. $\square$

As was described above for frequent sets, minimal transversals of a suitably constructed hypergraph constitute the negative border.

Definition 8.17 Let $\mathcal{P}$ and $\mathcal{S} \subseteq \mathcal{P}$ be sets of patterns and let f be a set representation of $\mathcal{P}$. We denote by $\mathcal{H}(\mathcal{S})$ the simple hypergraph on R that contains as edges the complements of sets $f(\varphi)$ for $\varphi \in \mathcal{B}d^+(\mathcal{S})$, i.e., $\mathcal{H}(\mathcal{S}) = \{R \setminus f(\varphi) \mid \varphi \in \mathcal{B}d^+(\mathcal{S})\}$. $\square$

Now $\mathcal{H}(\mathcal{S})$ is a hypergraph corresponding to the set representation of $\mathcal{S}$, and $\mathcal{T}r(\mathcal{H}(\mathcal{S}))$ is the set representation of the negative border. The inverse function f^{-1} maps the set representations of the negative border to the patterns in the negative border. That is, the set $f^{-1}(\mathcal{T}r(\mathcal{H}(\mathcal{S})))$ is the negative border. Next we show this formally.

Theorem 8.18 Let $\mathcal{P}$ and $\mathcal{S} \subseteq \mathcal{P}$ be sets of patterns, and let f be a set representation of $\mathcal{P}$. Then $f^{-1}(\mathcal{T}r(\mathcal{H}(\mathcal{S}))) = \mathcal{B}d^-(\mathcal{S})$.

Proof We prove the claim in two steps. First we show that X is a transversal of $\mathcal{H}(\mathcal{S})$ if and only if $f^{-1}(X) \notin \mathcal{S}$:

$$\begin{array}{ll} & X \text{ is a transversal of } \mathcal{H}(\mathcal{S}) \\ \Leftrightarrow & X \cap Y \neq \emptyset \text{ for all } Y \in \mathcal{H}(\mathcal{S}) \\ \Leftrightarrow & X \cap (R \setminus f(\varphi)) \neq \emptyset \text{ for all } \varphi \in \mathcal{B}d^+(\mathcal{S}) \\ \Leftrightarrow & X \not\subseteq f(\varphi) \text{ for all } \varphi \in \mathcal{B}d^+(\mathcal{S}) \\ \Leftrightarrow & f^{-1}(X) \not\preceq \varphi \text{ for all } \varphi \in \mathcal{B}d^+(\mathcal{S}) \\ \Leftrightarrow & f^{-1}(X) \notin \mathcal{S}. \end{array}$$

Next we show that $\mathcal{T}r(\mathcal{H}(\mathcal{S})) = f(\mathcal{B}d^-(\mathcal{S}))$; the theorem then immediately follows.

$$\begin{array}{ll} & \mathcal{T}r(\mathcal{H}(\mathcal{S})) \\ = & \{X \mid X \text{ is a minimal transversal of } \mathcal{H}(\mathcal{S})\} \\ = & \{X \mid X \text{ is a minimal set such that } f^{-1}(X) \notin \mathcal{S}\} \\ = & \{X \mid f^{-1}(X) \notin \mathcal{S} \text{ and } f^{-1}(Y) \in \mathcal{S} \text{ for all } Y \subset X\} \\ = & \{f(\varphi) \mid \varphi \notin \mathcal{S} \text{ and } \theta \in \mathcal{S} \text{ for all } \theta \prec \varphi\} \\ = & f(\mathcal{B}d^-(\mathcal{S})). \end{array}$$

□

Thus for languages representable as sets, the notions of negative border and the minimal transversals coincide.

Example 8.19 Continuing Example 8.12, the discovery of frequent sets in a relation with attributes $R = \{A, \ldots, F\}$, we now compute the set $\mathcal{B}d^-(\mathcal{S})$ using the hypergraph formulation. For frequent sets we simply let f be the identity function: $f(X) = X$. We are given the positive border $\mathcal{B}d^+(\mathcal{S}) = \{\{A, B\}, \{A, C, F\}\}$, so we have

$$\mathcal{H}(\mathcal{S}) = \{R \setminus f(X) \mid X \in \mathcal{B}d^+(()\mathcal{S})\} = \{\{C, D, E, F\}, \{B, D, E\}\}.$$

The minimal transversals of $\mathcal{H}(\mathcal{S})$ are $\{D\}$, $\{E\}$, $\{B, C\}$, and $\{B, F\}$, thus

$$\mathcal{B}d^-(()\mathcal{S}) = f^{-1}(\mathcal{T}r(\mathcal{H}(\mathcal{S}))) = \mathcal{T}r(\mathcal{H}(\mathcal{S})) = \{\{D\}, \{E\}, \{B, C\}, \{B, F\}\}.$$

□

We showed in Section 8.4 that under some simplifying restrictions the negative border must be evaluated when discovering all frequent patterns. In this section we showed that for patterns representable as sets the notion of negative border corresponds to the minimal transversals of a suitably defined hypergraph. The advantage of this is that the wealth of material about transversals (see, e.g., [7]) can be used, e.g., in the design of algorithms or complexity analysis for specific knowledge discovery problems.

8.6 Bibliographic notes

Many of the concepts and ideas of this and the previous chapter are known in different contexts. These chapters are, however, among the first attempts to provide a unified viewpoint to several knowledge discovery tasks and a generic algorithm for those tasks. Parts of the material of these chapters have been published in [61].

Algorithm 7.6 is based on the breadth-first search strategy, and it uses a specialization relation for safely pruning branches of the search tree, both well known search methods. The idea of checking *all* more general patterns when generating candidates has, however, been missed, e.g., in the original algorithm for discovering frequent sets [1] and in the inference of inclusion dependencies [49, 58]. In the area of machine learning, the version spaces of Mitchell [68] are the first systematic use of specialization relations and concepts similar to our border. Mitchell's learning task is different from ours, but conceptually Mitchell's set S of the most specific consistent patterns is the same as our positive border. —A generic viewpoint to knowledge discovery algorithms, similar to ours, has been expressed in [79].

There are several knowledge discovery settings that can be seen as the discovery of all frequent patterns. We very briefly contrast the described approach with two important systems, Explora and Claudien.

Explora [53] is a system for discovering patterns describing outstanding subgroups of a given database. Explora employs specialization relations: it organizes patterns into hierarchies and lets the user specify which are specialization relations with respect to the domain and the user's interests. The algorithm repeatedly evaluates patterns along paths between the most general and the most specific patterns until the border is located. The specialization relations are used to prune from evaluation those patterns for which the truth of selection criterion is already known.

Claudien [13] discovers minimally complete first order clausal theories from databases; in our terms a minimally complete theory roughly corresponds to the positive border. Claudien discovers the positive border by finding those patterns that are not frequent, i.e., by a conversed strategy. Due to the large number of patterns, in particular the number of patterns in individual levels, the implementation of Claudien uses depth-first search in order to save space. Therefore Claudien cannot take full advantage of the specialization relation: it may generate and test candidate clauses that cannot hold given all more general patterns.

In general, the depth-first search strategy may be useful if the collections of frequent patterns on each level are very large, or if computing the selection criterion from the database is cheap. Pratt [46], a system for the discovery of patterns in protein sequences, is a good example of such an application. Since the total size of the analyzed sequences is not large, Pratt can store the data in main memory and even use index structures that are significantly larger than the original data. Evaluating the selection criterion is thus fast, possibly even faster than evaluating whether a potential candidate is a valid candidate or not.

Methods that use sampling in the discovery of frequent sets have been given in [84].

The problem complexity of these settings has not received much attention. Some lower bounds for the problem of finding all frequent sets are given in [2, 62]. The relevance of transversals to computing the theory of a model has been known in the context of finding functional dependencies [59] and several other specific problems [18]. The complexity of computing the minimal transversals of a hypergraph has long been open; see [34, 67] for recent results. It is, however, known that transversals can be computed in time $\mathcal{O}(n^{\mathcal{O}(\log n)})$, where n is the sum of the sizes of the edges of both the hypergraph and its minimal transversals [26, 50]. The connection of the border and transversals have been recently strengthened [33].

Chapter 9

Sampling large databases for frequent sets

The size of the data collection has an essential role in data mining. Large data sets are necessary for reliable results—unfortunately, however, the efficiency of mining algorithms depends significantly on the database. The time complexity of the frequent set discovery algorithm is linear with respect to the number of rows in the database. However, the algorithm requires multiple passes over the database, and subsequently the database size is the most influential factor in the execution time for large databases.

In this chapter we present algorithms that make only one or sometimes two passes over the database. The idea is to pick a random sample from the input database, use it to determine all sets that possibly are frequent in the whole database, and then to verify the results with the rest of the database. These algorithms thus produce a correct set of association rules in one full pass over the database. In those rare cases where our sampling method does not produce all frequent sets, the missing sets can be found in a second pass. The concept of negative border turns out to be useful in this task.

We describe our sampling approach for discovering association rules in Section 9.1. In Section 9.2 we analyze the goodness of the sampling method, e.g., the relation of sample size to the accuracy of results. In Section 9.3 we give variations of algorithms and experimental results. The results show that the methods reduce the disk activity considerably, making the approach attractive especially for large databases. This chapter is based on [84].

9.1 Sampling in the discovery of frequent sets

An obvious way of reducing the database activity of knowledge discovery is to use only a random sample of the database and to find approximate regularities. In other words, one can trade off accuracy against efficiency. This can be useful: samples small enough to be handled totally in main

Algorithm 9.1

Input: A binary database r over a set R, a frequency threshold *min_fr*, a sample size *s_size*, and a lowered frequency threshold *low_fr* $<$ *min_fr*.

Output: The collection $\mathcal{F}(r, min\_fr)$ of frequent sets and their frequencies, or its subset and a failure report.

Method:

compute s := a random sample of size *s_size* from r;
// Find frequent sets in the sample:
compute $\mathcal{S} := \mathcal{F}(s, low\_fr)$ in main memory using Algorithm 2.14;
// Database pass:
compute $\mathcal{R} := \{X \in \mathcal{S} \cup \mathcal{B}d^-(\mathcal{S}) \mid fr(X, r) \geq min\_fr\}$ using Algorithm 2.22;
// Output:
for all $X \in \mathcal{R}$ **do** output X and $fr(X, r)$;
if $\mathcal{R} \cap \mathcal{B}d^-(\mathcal{S}) \neq \emptyset$ **then** report that there possibly was a failure;

---

memory can give reasonably accurate results. Or, approximate results from a sample can be used to set the focus for a more complete discovery phase.

It is often important to know the frequencies and confidences of association rules exactly. In business applications, for example for large volumes of supermarket sales data, even small differences can be significant. When relying on results from sampling alone, one also takes the risk of losing valid association rules altogether because their frequency in the sample is below the threshold.

Using a random sample to get approximate results is fairly straightforward. Below we give bounds for sample sizes, given the desired accuracy of the results. We show further that exact frequencies can be found efficiently, by analyzing first a random sample and then the whole database as follows. Use a random sample to locate a superset $\mathcal{S}$ of the collection of frequent sets. A superset can be determined efficiently by applying Algorithm 2.14 on the sample in main memory, and by using a lowered frequency threshold. Then use $\mathcal{S}$ as the collection of candidates, and compute the exact frequencies of the sets from the rest of the database. This approach, when successful, requires only one full pass over the database, and two passes in the worst case.

Algorithm 9.1 presents the principle: search for frequent sets in the sample, but use a lower frequency threshold so that it is unlikely that frequent sets are missed.

The concept of negative border is useful here. As was noted in the previous chapter, the border has to be inspected when discovering frequent sets. It is thus not sufficient to locate a superset $\mathcal{S}$ of $\mathcal{F}(r, min\_fr)$ using the sample and then to evaluate $\mathcal{S}$ in r. Rather, the collection $\mathcal{F}(r, min\_fr) \cup \mathcal{B}d^-(\mathcal{F}(r, min\_fr))$ needs to be checked. Obviously $\mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$ is a superset of this collection if $\mathcal{S}$ is a superset of $\mathcal{F}(r, min\_fr)$, so we check the union. Sometimes, however, it happens that we find out that not all necessary sets have been evaluated.

Definition 9.2 There has been a *failure* in the sampling if all frequent sets are not found in one pass, i.e., if there is a frequent set X in $\mathcal{F}(r, min\_fr)$ that is not in $\mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$. A *miss* is a frequent set Y in $\mathcal{F}(r, min\_fr)$ that is in $\mathcal{B}d^-(\mathcal{S})$. □

If there are no misses, then the sampling has been successful. Misses themselves are not a problem: they are evaluated in the whole database, and thus they are not actually missed by the algorithm. Misses, however, indicate a potential failure. Namely, if there is a miss Y, then some superset of Y might be frequent but not in $\mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$. A simple way to recognize a potential failure is thus to check if there are any misses.

Theorem 9.3 Algorithm 9.1 works correctly.

Proof Clearly, on lines 5 and 7, a collection of frequent sets is computed and output. We need to show that if no failure report is given, then all frequent sets are found, and that if all frequent sets are not found, then a failure report is, in turn, given.

If there is no failure report, i.e., if $\mathcal{R}$ and $\mathcal{B}d^-(\mathcal{S})$ are disjoint, then $\mathcal{R} \subseteq \mathcal{S}$, and $\mathcal{B}d^-(\mathcal{R}) \subseteq \mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$. Thus the whole negative border $\mathcal{B}d^-(\mathcal{R})$ has been evaluated, and all frequent sets are found. If all frequent sets are not found, i.e., if there is a frequent set X that is not in $\mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$, then there exists a set Y in $\mathcal{B}d^-(\mathcal{S})$ such that $Y \subseteq X$ and Y is frequent. This set Y is thus in $\mathcal{R} \cap \mathcal{B}d^-(\mathcal{S})$, and a failure is reported. □

Example 9.4 Assume that we have a binary database r with 10 million rows over items $A, \ldots, F$, and that we want to find the frequent sets with the threshold 0.02. Algorithm 9.1 randomly picks a small fraction s of r, say 20 000 rows, and keeps this sample s in main memory. The algorithm can now, without any further database activity, discover efficiently the sets that are frequent in the sample.

To make it very probably that the collection of frequent sets in the sample includes all sets that really are frequent in r, the frequency threshold is lowered to, say, 0.015. So Algorithm 9.1 determines the collection $\mathcal{S} = \mathcal{F}(s, 0.015)$ from the sampled 20 000 rows. Let the maximal sets of $\mathcal{S}$, i.e., the positive border $\mathcal{B}d^+(\mathcal{S})$, be

$$\{A, D\}, \{B, D\}, \{A, B, C\}, \{A, C, F\}.$$

Since the threshold was lowered, $\mathcal{S}$ is likely to be a superset of the collection $\mathcal{F}(r, 0.02)$ of frequent sets. In the pass over the rest of the database r, the frequency of all sets in $\mathcal{S}$ and $\mathcal{B}d^-(\mathcal{S})$ is evaluated. That is, in addition to

Algorithm 9.5
Input: A binary database r over a set R, a frequency threshold *min_fr*, a subset $\mathcal{S}$ of $\mathcal{F}(r, min\_fr)$, and a subset $\mathcal{S}^-$ of $\mathcal{B}d^-(\mathcal{F}(r, min\_fr))$.
Output: The collection $\mathcal{F}(r, min\_fr)$ of frequent sets and their frequencies.
Method:
1. **repeat** compute $\mathcal{S} := \mathcal{S} \cup (\mathcal{B}d^-(\mathcal{S}) \setminus \mathcal{S}^-)$ **until** $\mathcal{S}$ does not grow;
2. compute $\mathcal{R} := \{X \in \mathcal{S} \mid fr(X, r) \geq min\_fr\}$;
3. **for** all $X \in \mathcal{R}$ **do** output X and $fr(X, r)$;

---

the sets that are frequent in the sample, we evaluate also those candidates that were not frequent, i.e., the negative border

$$\{E\}, \{B, F\}, \{C, D\}, \{D, F\}, \{A, B, D\}.$$

The goal is to discover the collection $\mathcal{F}(r, 0.02)$. Let the sets

$$\{A, B\}, \{A, C, F\}.$$

and their subsets be the frequent sets. All frequent sets are in $\mathcal{S}$, so they are evaluated and their exact frequencies are known after the full database pass. We also know that we have found all frequent sets since also sets

$$\{D\}, \{E\}, \{B, C\}, \{B, F\},$$

i.e., sets in the negative border of $\mathcal{F}(r, 0.02)$, were evaluated and found to be non-frequent.

Now assume a slightly different situation, where the set $\{B, F\}$ turns out to be frequent in r, that is, $\{B, F\}$ is a miss. The set $\{A, B, F\}$ could be frequent in r, since all its subsets are. In this case Algorithm 9.1 reports that there possibly is a failure. □

The problem formulation is now the following: given a database r and a frequency threshold *min_fr*, use a random sample s to determine a collection $\mathcal{S}$ of sets such that $\mathcal{S}$ contains with a high probability the collection of frequent sets $\mathcal{F}(r, min\_fr)$. For efficiency reasons, a secondary goal is that $\mathcal{S}$ does not contain unnecessarily many other sets.

In the fraction of cases where a possible failure is reported, all frequent sets can be found by making a second pass over the database. Algorithm 9.5 can be used to extend Algorithm 9.1 with a second pass in such a case. The algorithm simply computes the collection of all sets that possibly could be frequent. The parameter $\mathcal{S}$ is the collection of frequent sets found by Algorithm 9.1, and $\mathcal{S}^-$ is the collection of non-frequent candidates of Algorithm 9.1. The collection $\mathcal{B}d^-(\mathcal{S})$ can be located in a similar way as candidates are generated.

Theorem 9.6 Algorithm 9.5 works correctly.

Proof All sets computed and output on lines 2 and 3 are clearly frequent. To see that all frequent sets are output, consider any frequent set X and assume the contrary: X is not in $\mathcal{S}$ after line 1. Let $Y \subseteq X$ be the smallest subset of X that is not in $\mathcal{S}$. Then all subsets of Y are in $\mathcal{S}$, and Y must be in the negative border $\mathcal{B}d^-(\mathcal{S})$. The only possible reason for Y being excluded from $\mathcal{S}$ is that it is in $\mathcal{S}^-$. This is, however, a contradiction, since Y must be frequent. Thus all frequent sets are output. □

The number of candidates in the second pass can, in principle, be too large to fit in the main memory and to be handled in one database pass. This can happen when the sample is very bad and gives inaccurate results.

9.2 Analysis of sampling

Let us now analyze the relation of sample size to the accuracy of results. We first consider how accurate the frequencies computed from a random sample are. As has been noted before, samples of reasonable size provide good approximations for frequencies of sets [2, 62]. Related work on using a sample for approximately verifying the truth of arbitrary sentences of relational tuple calculus is considered in [51].

Definition 9.7 Given an item set $X \subseteq R$ and a random sample s of a binary database over R, the *error* $e(X, s)$ is the difference of the frequencies:

$$e(X, s) = |fr(X, s) - fr(X)|,$$

where $fr(X)$ is the frequency of X in the database from which s was drawn. □

To analyze the error, we consider sampling with replacement. The reason is that we want to avoid making other assumptions of the database size except that it is large. For sampling with replacement the size of the database has no effect on the analysis, so the results apply, in principle, on infinitely large databases. Note also that for very large databases there is practically no difference between sampling with and without replacement.

In the following we analyze the random variable $|\mathcal{M}(X, s)|$, that is, the number of rows in the sample s that contain X. The random variable has binomial distribution $B(|s|, fr(X))$, i.e., the probability of $|\mathcal{M}(X, s)| = c$, denoted $Pr[|\mathcal{M}(X, s)| = c]$, is

$$\binom{|s|}{c} fr(X)^c (1 - fr(X))^{|s|-c}.$$

First consider the necessary size of a sample, given requirements on the size of the error. The following theorem gives a lower bound for the size of

ε	δ	Sample size
0.01	0.01	27 000
0.01	0.001	38 000
0.01	0.0001	50 000
0.001	0.01	2 700 000
0.001	0.001	3 800 000
0.001	0.0001	5 000 000

Table 9.1: Sufficient sample sizes, given ε and δ.

the sample, given an error bound ε and a maximum probability δ for an error that exceeds the bound.

Theorem 9.8 Given an item set X and a random sample s of size

$$|s| \geq \frac{1}{2\varepsilon^2} \ln \frac{2}{\delta}$$

the probability that $e(X, s) > \varepsilon$ is at most δ.

Proof The Chernoff bounds give the result $Pr[|x - np| > a] < 2e^{-2a^2/n}$, where x is a random variable with binomial distribution $B(n, p)$ [5]. For the probability at hand we thus have

$$Pr[e(X, s) > \varepsilon] = Pr[|fr(X, s) - fr(X)| \cdot |s| > \varepsilon\,|s|] \leq 2e^{-2(\varepsilon\,|s|)^2/|s|} \leq \delta.$$

□

Table 9.1 gives values for the sufficient sample size $|s|$, for $\varepsilon = 0.01, 0.001$ and $\delta = 0.01, 0.001, 0.0001$. With the tolerable error ε around 0.01, samples of a reasonable size suffice. For instance, if a chance of 0.0001 for an error of more than 0.01 is acceptable, then a sample of size 50 000 is sufficient. For many applications these parameter values are perfectly reasonable. In such cases, approximate rules can be produced based on a sample, i.e., in constant time independent of the size of r. With tighter error requirements the sample sizes can be quite large.

The result above is for a given set X. The following corollary gives a result for a more stringent case: given a collection $\mathcal{S}$ of sets, with probability $1 - \Delta$ there is no set in $\mathcal{S}$ with error at least ε.

Corollary 9.9 Given a collection $\mathcal{S}$ of sets and a random sample s of size

$$|s| \geq \frac{1}{2\varepsilon^2} \ln \frac{2|\mathcal{S}|}{\Delta},$$

the probability that there is a set $X \in \mathcal{S}$ such that $e(X, s) > \varepsilon$ is at most Δ.

Proof By Theorem 9.8, the probability that $e(X, s) > \varepsilon$ for a given set X is at most $\frac{\Delta}{|\mathcal{S}|}$. Since there are $|\mathcal{S}|$ such sets, the probability in question is at most Δ. □

The Chernoff bound is not always very tight, and in practice the exact probability from the binomial distribution or its normal approximation are more useful.

Consider now the proposed approach to finding all frequent sets exactly. The idea was to locate a superset of the collection of frequent sets by discovering frequent sets in a sample with a lower threshold. Consider first the following simple setting: take a sample as small as possible but such that it is likely to contain all frequent sets at least once. What should the sample size be?

Theorem 9.10 Given a set X with $fr(X) \geq min\_fr$ and a random sample s of size

$$|s| \geq \frac{1}{min\_fr} \ln \frac{1}{\delta},$$

the probability that X does not occur in s is at most δ.

Proof We apply the following inequality: for every $x > 0$ and every real number b we have $(1 + \frac{b}{x})^x \leq e^b$. The probability that a frequent set X does not occur on a given row is at most $1 - min\_fr$. The probability that X does not occur on any row is then at most $(1 - min\_fr)^{|s|}$, which is further bounded by the inequality by $e^{-|s|\, min\_fr} \leq \delta$. □

The sample size given by the theorem is small, but unfortunately the approach is not very useful: a sample will include a lot of garbage, i.e., sets that are not frequent nor in the border. For instance, a single row containing 20 items has over a million subsets, and all of them would then be checked from the whole database. Obviously, to be useful the sample must be larger. It is likely that best results are achieved when the sample is as large as can conveniently be handled in the main memory.

We move on to the following problem. Assume we have a sample s and a collection $\mathcal{S} = \mathcal{F}(s, low\_fr)$ of sets. What can we say about the probability of a failure? We use the following simple approximation. Assuming that the sets in $\mathcal{B}d^-(\mathcal{S})$ are independent, an upper bound for the probability of a failure is the probability that at least one set in $\mathcal{B}d^-(\mathcal{S})$ turns out to be frequent in r.

This approximation tends to give too large probabilities. Namely, a set X in $\mathcal{B}d^-(\mathcal{S})$ that is frequent in r, i.e., an X that is a miss, does not necessarily indicate a failure at all. In general there is a failure only if the addition of X to $\mathcal{S}$ would add sets to the negative border $\mathcal{B}d^-(\mathcal{S})$; often several additions to $\mathcal{S}$ are needed before there are such new candidates. Note also that the assumption that sets in $\mathcal{B}d^-(\mathcal{S})$ are independent is unrealistic.

An interesting aspect is that this approximation can be computed on the fly when processing the sample. Thus, if an approximated probability of a failure needs to be set in advance, then the frequency threshold *low_fr* can be adjusted at run time to fit the desired probability of a miss. A variation of Theorem 9.8 gives the following result on how to set the lowered frequency threshold so that misses are avoided with a high probability.

Theorem 9.11 Given a frequent set X, a random sample s, and a probability parameter δ, the probability that X is a miss is at most δ when

$$low\_fr \leq min\_fr - \sqrt{\frac{1}{2|s|} \ln \frac{1}{\delta}}.$$

Proof Using the Chernoff bounds again—this time for one-sided error—we have

$$Pr[fr(X, s) < low\_fr] \leq e^{-2(\sqrt{\frac{1}{2|s|} \ln \frac{1}{\delta}}|s|)^2/|s|} = \delta.$$

□

Consider now the number of sets checked in the second pass by Algorithm 9.5, in the case of a potential failure. The collection $\mathcal{S}$ can, in principle, grow a lot. Each independent miss can in the worst case generate as many new candidates as there are frequent sets. Note, however, that if the probability that any given set is a miss is at most δ, then the probability of l independent misses can be at most δ^l.

9.3 Experiments

We now describe the experiments we conducted to assess the practical feasibility of the sampling method for finding frequent sets. In this section we also present variants of the method and give experimental results.

Test organization We used three synthetic data sets from [3] in our tests. These databases model supermarket basket data, and they have been used as benchmarks for several association rule algorithms [2, 3, 40, 73, 77]. The central properties of the data sets are the following. There are $|R| = 1\ 000$ items, and the average number T of items per row is 5, 10, or 20. The number $|r|$ of rows is approximately 100 000. The average size I of maximal frequent sets, i.e., sets in the positive border, is 2, 4, or 6. Table 9.2 summarizes the parameters for the data sets; see [3] for more details of the data generation.

We assume that the real data collections from which association rules are discovered can be much larger than the test data sets. To make the experiments fair we use sampling with replacement. This means that the

Data set name	$\lvert R\rvert$	T	I	$\lvert r\rvert$	Size (MB)
T5.I2.D100K	1 000	5	2	97 233	2.4
T10.I4.D100K	1 000	10	4	98 827	4.4
T20.I6.D100K	1 000	20	6	99 941	8.4

Table 9.2: Synthetic data set characteristics (T = row size on average, I = size of sets in the positive border on average).

Frequency threshold	Sample size			
	20 000	40 000	60 000	80 000
0.0025	0.0013	0.0017	0.0018	0.0019
0.0050	0.0034	0.0038	0.0040	0.0041
0.0075	0.0055	0.0061	0.0063	0.0065
0.0100	0.0077	0.0083	0.0086	0.0088
0.0150	0.0122	0.0130	0.0133	0.0135
0.0200	0.0167	0.0177	0.0181	0.0184

Table 9.3: Lowered frequency thresholds for $\delta = 0.001$.

real data collections could have been arbitrary large data sets such that these data sets represent their distributional properties.

We considered sample sizes from 20 000 to 80 000. Samples of these sizes are large enough to give good approximations and small enough to be handled in main memory. Since our approach is probabilistic, we repeated every experiment 100 times for each parameter combination. Altogether, over 10 000 trials were run.

Number of misses and database activity We experimented with Algorithm 9.1 with the above mentioned sample sizes 20 000 to 80 000. We selected the lowered threshold so that the probability of missing any given frequent set X is less than $\delta = 0.001$, i.e., given any set X with $fr(X) \geq min\_fr$, we have $Pr[fr(X, s) < low\_fr] \leq 0.001$. The lowered threshold depends on the frequency threshold and the sample size. The lowered threshold values are given in Table 9.3; in the computations of the lowered thresholds we used the exact probabilities obtained from the binomial distribution, not the Chernoff bounds.

Figure 9.1 shows the number of database passes for the three different types of algorithms: Algorithm 2.14, Partition, and the sampling Algorithm 9.1. The Partition algorithm [77] was discussed already shortly in Chapter 2. It is based on the idea of partitioning the database to several parts, each small enough to be handled in main memory. The algorithm

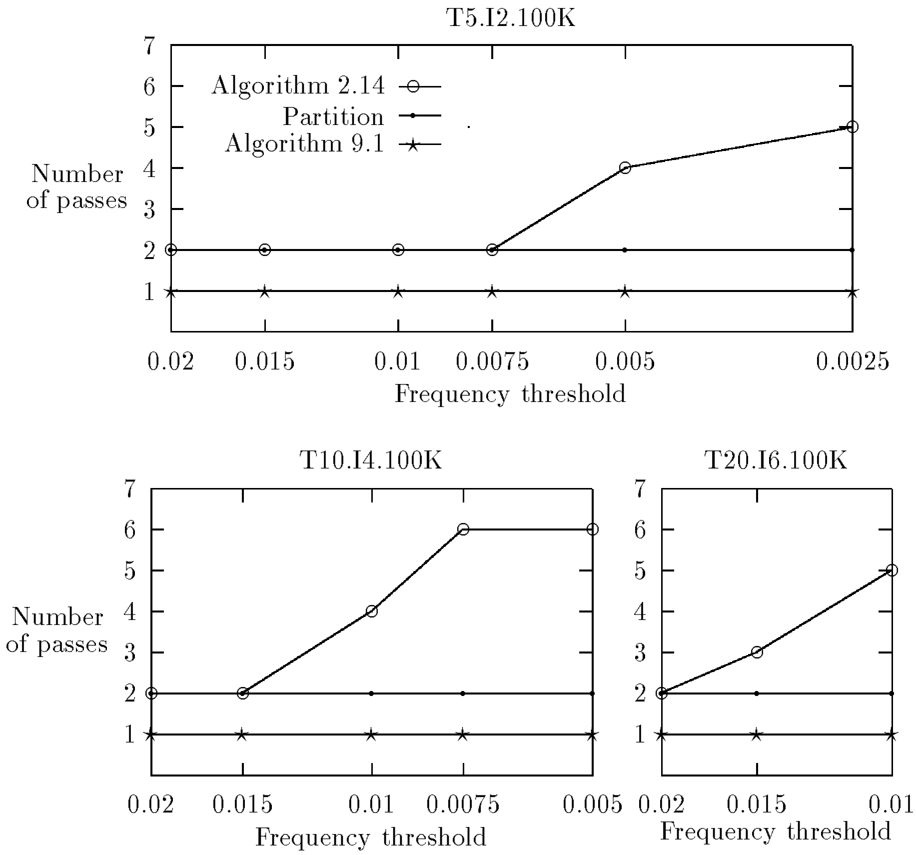


Figure 9.1: The number of database passes made by frequent set algorithms.

almost always makes two passes over the database: in the first pass, it uses a variant of Algorithm 2.14 to find frequent sets in each partition, and in the second pass it checks in the whole database all sets that were frequent in at least one partition. Each of the data points in the results shown for Algorithm 9.1 is the average value over 100 trials.

Explaining the results is easy. Algorithm 2.14 makes $L(+1)$ passes over the database, where L is the size of the largest frequent set. The Partition algorithm makes two passes over the database whenever there are any frequent sets. For Algorithm 9.1, the fraction of trials with misses is expected to be larger than $\delta = 0.001$, depending on how many frequent sets have a frequency relatively close to the threshold and are thus likely misses in a sample. The algorithm has succeeded in finding all frequent sets in one pass in almost all cases. The number of database passes made by Partition algorithm is practically twice that of Algorithm 9.1, and the number of passes of Algorithm 2.14 is up to six times that of Algorithm 9.1.

T5.I2.D100K

Frequency threshold	Sample size 20 000	40 000	60 000	80 000
0.0025	0	1	0	0
0.0050	0	1	0	1
0.0075	0	0	0	0
0.0100	0	0	0	0
0.0150	0	0	0	0
0.0200	0	0	0	0

T10.I4.D100K

Frequency threshold	Sample size 20 000	40 000	60 000	80 000
0.0050	0	2	1	1
0.0075	0	1	1	1
0.0100	1	0	1	1
0.0150	0	2	0	0
0.0200	0	0	0	0

T20.I6.D100K

Frequency threshold	Sample size 20 000	40 000	60 000	80 000
0.0100	0	0	0	0
0.0150	1	1	1	0
0.0200	0	1	0	2

Table 9.4: Number of trials with misses.

Table 9.4 shows the number of trials with misses for each data set, sample size, and frequency threshold. In each set of 100 trials, there have been zero to two trials with misses. The overall fraction of trials with misses was 0.0038. We repeated the experiment with $\delta = 0.01$, i.e., so that the probability of missing a given frequent set is at most 0.01. This experiment gave misses in fraction 0.041 of all the trials. In both cases the fraction of trials with misses was larger than δ, but of the same magnitude.

The actual amount of reduction in the database activity depends on the database storage structures. For instance, if the database has 10 million rows, a disk block contains on average 100 rows, and the sample size is 20 000, then the sampling phase could read up to 20 % of the database. An alternative for randomly drawing each row in separation is, of course, to draw whole blocks of rows to the sample. Depending on how randomly the rows have been assigned to the blocks, this method can give good or bad results. For the design and analysis of sampling methods see, e.g, [72]. The related problem of sampling for query estimation is considered in [35].

Frequency threshold	Sample size 20 000	40 000	60 000	80 000	Algorithm 2.14
0.0050	382 282	368 057	359 473	356 527	318 588
0.0075	290 311	259 015	248 594	237 595	188 024
0.0100	181 031	158 189	146 228	139 006	97 613
0.0150	52 369	40 512	36 679	34 200	20 701
0.0200	10 903	7 098	5 904	5 135	3 211

Table 9.5: Number of item sets considered for data set T10.I4.D100K.

The reduction in database activity is achieved at the cost of considering some item sets that Algorithm 2.14 does not generate and check. Table 9.5 shows the average number of sets considered for the data set T10.I4.D100K with different sample sizes and the number of candidate sets of Algorithm 2.14. The largest absolute overhead occurs with low thresholds, where the number of item sets considered has grown from 318 588 by 64 694 in the worst case. This growth is not significant for the total execution time since the item sets are handled entirely in main memory. The relative overhead is larger with higher thresholds, but since the absolute overheads are small the effect is negligible. Table 9.5 indicates that larger samples cause less overhead (with equally good results), but that for sample sizes from 20 000 to 80 000 the difference in the overhead is not significant.

To obtain a better picture of the relation of δ and the experimental number of trials with misses, we conducted the following test. We took 100 samples (for each frequency threshold and sample size) and determined the lowered frequency threshold that would have given misses in one out of the hundred trials. Figure 9.2 presents these results (as points), together with lines showing the lowered thresholds with $\delta = 0.01$ or 0.001, i.e., the thresholds corresponding to miss probabilities of 0.01 and 0.001 for a given frequent set. The frequency thresholds that give misses in fraction 0.01 of cases approximate surprisingly closely the thresholds for $\delta = 0.01$. Experiments with a larger scale of sample sizes give comparable results. There are two explanations for the similarity of the values. One reason is that there are not necessarily many potential misses, i.e., not many frequent sets with frequency relatively close to the threshold. Another reason that contributes to the similarity is that the sets are not independent.

In the case of a possible failure, Algorithm 9.5 generates iteratively all new candidates and makes another pass over the database. In our experiments the number of frequent sets missed—when any were missed—was one or two for $\delta = 0.001$, and one to 16 for $\delta = 0.01$. The number of candidates checked on the second pass was small compared to the total number of item sets checked.

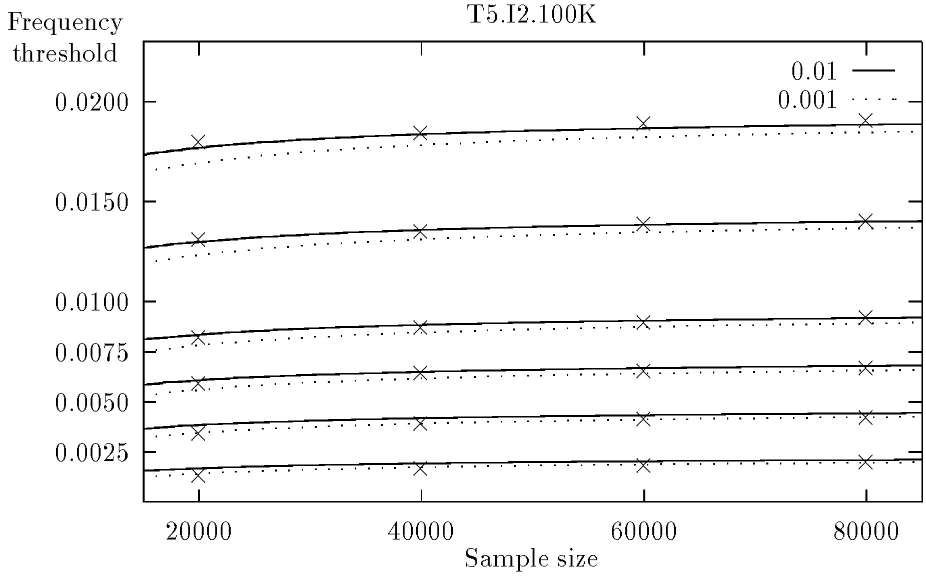


Figure 9.2: Frequency thresholds giving misses in 0.01 cases (points) and lowered thresholds with $\delta = 0.01$ and 0.001 (lines).

Approximate $1 - \Delta$ success probability Setting the lowered threshold for Algorithm 9.1 is not trivial: how to select it so that the probability of a failure is low but there are not unnecessarily many sets to check? An automatic way of setting the parameter would be desirable. Consider, for instance, an interactive mining tool. It would be useful to know in advance how long an operation will approximately take—or, in the case of mining association rules, how many database passes there will be.

We now present two algorithms that find all frequent sets in approximately fraction $1 - \Delta$ of the cases, where Δ is given by the user. Under the assumption that sets in the negative border are independent, the algorithms are actually guaranteed to find the correct sets at least in fraction $1-\Delta$ of the cases. The first algorithm uses a simple greedy principle to find the optimal lowered threshold under the independence assumption. The other algorithm is not as optimal, but its central phase is almost identical to Algorithm 2.14 and it is therefore easy to incorporate into existing implementations. We present experimental results with this latter algorithm.

The greedy Algorithm 9.12 starts with an empty set $\mathcal{S}$, and it then decreases the probability of failure by adding the most probable misses to $\mathcal{S}$ until the approximated probability of a potential failure is at most Δ.

Theorem 9.13 Algorithm 9.12 works correctly.

Algorithm 9.12

Input: A binary database r, a frequency threshold *min_fr*, a sample size *s_size*, and a miss probability Δ.

Output: The collection $\mathcal{F}(r, \mathit{min\_fr})$ of frequent sets and their frequencies at least in fraction $1-\Delta$ of the cases (assuming that the frequencies of any two sets X, Y are independent whenever $X \not\subseteq Y$ and $Y \not\subseteq X$), and a subset of $\mathcal{F}(r, \mathit{min\_fr})$ and a failure report in the rest of the cases.

Method:

1. compute $s :=$ random sample of size *s_size* from r;
2. $\mathcal{S} := \emptyset$;
3. // Find frequent sets in the sample:
4. **while** the estimated probability of a miss is larger than Δ **do**
5.   select $X \in \mathcal{B}d^-(\mathcal{S})$ with the highest probability of being a miss;
6. $\mathcal{S} := \mathcal{S} \cup \{X\}$;
7. // Database pass:
8. compute $\mathcal{R} := \{X \in \mathcal{S} \cup \mathcal{B}d^-(\mathcal{S}) \mid \mathit{fr}(X, r) \geq \mathit{min\_fr}\}$;
9. **for** all $X \in \mathcal{R}$ **do** output X and $\mathit{fr}(X, r)$;
10. **if** $\mathcal{R} \cap \mathcal{B}d^-(\mathcal{S}) \neq \emptyset$ **then** report that there possibly was a failure;

Proof See the proof of Theorem 9.3 for the correctness of the output and failure reports. From the assumption of independence of sets it follows, in particular, that sets in $\mathcal{B}d^-(\mathcal{S})$ are independent. Then the probabilities can be easily computed on lines 4 and 5, and the algorithm fails in less than fraction Δ of cases. Note also that a single miss does not always indicate a failure, and therefore the probability of a miss is an upper bound for the probability of a failure. $\square$

The assumption of independence of sets in the algorithm is unrealistic. For this reason Δ in practice only approximates (an upper bound of) the fraction of failures. Algorithm 9.14 is a simple variation of Algorithm 2.14. It utilizes also the failure probability approximation described in Section 9.2: it monitors the approximated probability of a miss and keeps the probability small by lowering the frequency threshold *low_fr*, when necessary, for the rest of the algorithm. When using Algorithm 2.14 for the discovery of frequent sets in a sample with the dynamic adjustment of the lowered threshold, the only modification concerns the phase where candidates are either added to the collection of frequent sets or thrown away. Every time there is a candidate X that is not frequent in the sample, compute the probability p that X is frequent. If the total probability P of a miss increases too much (see below), then lower the frequency threshold *low_fr* to the frequency of X in the sample for the rest of the algorithm. Thus X is eventually considered frequent in the sample, and so are all following candidate sets that would increase the overall probability of a miss at least as much as X.

We use the following heuristic to decide whether the possibility of a miss increases too much. Given a parameter γ in $[0, 1]$, the frequency threshold

Algorithm 9.14
Input: A binary database r over a set R, a sample size s_size, a frequency threshold min_fr, a miss probability Δ, and $\gamma \in [0,1]$.
Output: The collection $\mathcal{F}(r, min\_fr)$ of frequent sets and their frequencies at least in fraction $1 - \Delta$ of the cases (assuming that the frequencies of any two sets X, Y are independent whenever $X \not\subseteq Y$ and $Y \not\subseteq X$), and a subset of $\mathcal{F}(r, min\_fr)$ and a failure report in the rest of the cases.
Method:

1. compute $s :=$ random sample of size s_size from r;
2. $P := 0$;
3. $low\_fr := min\_fr$;
4. // Find frequent sets in the sample:
5. $\mathcal{C}_1 := \{\{A\} \mid A \in R\}$;
6. $l := 1$;
7. **while** $\mathcal{C}_l \neq \emptyset$ **do**
8. $\mathcal{R}_l := \emptyset$;
9.   **for** all $X \in \mathcal{C}_l$ **do**
10.   **if** $fr(X, s) < low\_fr$ **then**
11. $p := Pr[X$ is frequent in $r]$;
12.   **if** $p/(\Delta - P) > \gamma$ **then** $low\_fr := fr(X, s)$
13.   **else** $P := 1 - (1 - P)(1 - p)$;
14.   **if** $fr(X, s) \geq low\_fr$ **then** $\mathcal{R}_l := \mathcal{R}_l \cup \{X\}$;
15. $l := l + 1$;
16.   compute $\mathcal{C}_l := \mathcal{C}(\mathcal{R}_{l-1})$;
17. // Database pass:
18. compute $\mathcal{R} := \{X \in \bigcup_l \mathcal{C}_l \mid fr(X, r) \geq min\_fr\}$;
19. **for** all $X \in \mathcal{R}$ **do** output X and $fr(X, r)$;
20. **if** $\mathcal{R} \cap (\bigcup_l (\mathcal{C}_l \setminus \mathcal{R}_l)) \neq \emptyset$ **then** report that there possibly was a failure;

---

is lowered if the probability p is larger than fraction γ of the "remaining error reserve" $\Delta - P$. More complex heuristics for changing the frequency threshold could be developed, e.g., by taking into account the number of candidates on the level and whether the number of frequent sets per level is growing or shrinking. The observations made from Figure 9.2 hint that the lowered threshold can be set in the start-up to roughly correspond to the desired probability of a miss, i.e., for $\Delta = 0.01$ the lowered threshold could be set as for $\delta = 0.01$.

Theorem 9.15 Algorithm 9.14 works correctly.

Proof See the proof of Theorem 9.3 for the correctness of the output and failure reports. Consider the invariant that the variable P is the probability that any of the sets in the negative border is a miss (under the assumption that sets in the negative border are independent), and that $0 \leq P \leq \Delta$.

The invariant holds trivially in the beginning, where P is initialized to zero and no sets have been considered yet. We now show that the invariant continues to hold during the algorithm. On line 13, P is updated for each set X in the negative border in the sample, i.e., for each potential miss, to

T5.I2.D100K

Frequency Threshold	Sample size 20 000	40 000	60 000	80 000
0.005	3	3	0	2
0.010	0	0	0	0

T10.I4.D100K

Frequency threshold	Sample size 20 000	40 000	60 000	80 000
0.0075	1	4	2	1
0.0150	0	2	4	1

T20.I6.D100K

Frequency threshold	Sample size 20 000	40 000	60 000	80 000
0.01	2	1	1	1
0.02	1	3	1	3

Table 9.6: Number of trials with misses with $\Delta = 0.1$.

correctly correspond to the total probability of a miss. Given non-negative real numbers p and $\Delta - P$ such that $p/(\Delta - P) \leq \gamma \leq 1$, we have $p \leq \Delta - P$. Thus the new value of P is $1 - (1 - P)(1 - p) \leq 1 - (1 - P)(1 - (\Delta - P)) = \Delta + (P - \Delta)P$, and this is further bounded by Δ by the invariant itself. Thus the invariant continues to hold, and the probability of a miss when the program exits is $P \leq \Delta$. $\square$

Remember, again, that a single miss does not necessarily indicate a failure, and thus P is an upper bound for the probability of a failure. Since sets in the negative border are not necessarily independent, the upper bound is only an approximation.

We tested Algorithm 9.14 with maximum miss probability $\Delta = 0.1$ and dynamic adjustment parameter $\gamma = 0.01$ for two frequency thresholds for each data set. The number of trials with misses is shown in Table 9.6. The number successfully remained below $100\Delta = 10$ in each set of experiments. As Table 9.6 shows, the number of cases with misses was actually less than half of 10. The reason is that with a small γ the algorithm tends to be conservative and keeps a lot of space for the probability of a miss in reserve. This is useful when there can be very many candidates. The negligible trade-off is that the algorithm may consider unnecessarily many sets as frequent in the sample.

To summarize briefly, the experiments show that the proposed approach works well in practice: all frequent sets can actually be found in almost one pass over the database. For the efficiency of mining association rules in large databases the reduction of disk I/O is significant.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'93)*, pages 207 – 216, Washington, D.C., USA, May 1993. ACM.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307 – 328. AAAI Press, Menlo Park, CA, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB'94)*, pages 487 – 499, Sept. 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE'95)*, pages 3 – 14, Taipei, Taiwan, Mar. 1995.

[5] N. Alon and J. H. Spencer. *The Probabilistic Method.* John Wiley Inc., New York, NY, 1992.

[6] T. Anand. Opportunity explorer: Navigating large databases using knowledge discovery templates. *Journal of Intelligent Information Systems*, (4):27–37, 1995.

[7] C. Berge. *Hypergraphs. Combinatorics of Finite Sets.* North-Holland Publishing Company, Amsterdam, 1989.

[8] C. Bettini, X. S. Wang, and S. Jajodia. Testing complex temporal relationships involving multiple granularities and its application to data mining. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database and Knowledgebase Systems (PODS'96)*, pages 68 – 78, Montreal, Canada, June 1996.

[9] A. T. Bouloutas, S. B. Calo, A. Finkel, and I. Katzela. Distributed fault identification in telecommunication networks. *Journal of Network and Systems Management*, 3(3):295 – 312, 1995.

[10] R. J. Brachman, T. Khabaza, W. Kloesgen, G. Piatetsky-Shapiro, and E. Simoudis. Mining business databases. *Communications of the ACM*, pages 42–48, Nov. 1996.

[11] M. C. Burl, U. M. Fayyad, P. Perona, P. Smyth, and M. P. Burl. Automating the hunt for volcanoes on venus. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 302–309, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[12] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97)*, pages 12 – 27, Aarhus, Denmark, June 1997.

[13] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI–93)*, pages 1058 – 1053, Chambéry, France, 1993. Morgan Kaufmann.

[14] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 125–132. Springer-Verlag, 1997.

[15] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pages 30 – 36, New York, NY, Aug. 1998. AAAI Press.

[16] P. J. Denning. The working set model of program behavior. *Communications of the ACM*, 11(5):323 – 333, 1968.

[17] C. Dousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 166 – 172, Chambery, France, Aug. 1993.

[18] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278 – 1304, Dec. 1995.

[19] J. Elder IV and D. Pregibon. A statistical perspective on knowledge discovery in databases. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 83 – 113. AAAI Press, Menlo Park, CA, 1996.

[20] U. Fayyad, D. Haussler, and P. Stolorz. Mining scientific data. *Communications of the ACM*, pages 51–57, Nov. 1996.

[21] U. M. Fayyad, S. G. Djorgovski, and N. Weir. Automating the analysis and cataloging of sky surveys. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 471 – 494. AAAI Press, Menlo Park, CA, 1996.

[22] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1 – 34. AAAI Press, Menlo Park, CA, 1996.

[23] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, CA, 1996.

[24] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17 – 37, 1982.

[25] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 1 – 27. AAAI Press, Menlo Park, CA, 1991.

[26] M. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. Technical Report LCSR-TR-225, Rutgers University, 1994.

[27] T. Fukuda et al. Mining optimized association rules for numeric attributes. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database and Knowledgebase Systems (PODS'96)*, 1996.

[28] N. Gehani, H. Jagadish, and O. Shmueli. Composite event specification in active databases. In *Proceedings of the Eightteenth International Conference on Very Large Data Bases (VLDB'92)*, pages 327 – 338, Vancouver, Canada, Aug. 1992.

[29] J. Goldman, P. Hong, C. Jeromnimon, G. Louit, J. Min, and P. Sen. Integrated fault management in interconnected networks. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I*, pages 333 – 344. Elsevier, 1989.

[30] R. M. Goodman, B. E. Ambrose, H. W. Latin, and C. T. Ulmer. NOAA – An expert system managing the telephone network. In *Integrated Network Management IV*, pages 316 – 327. Chapman & Hall, London, 1995.

[31] R. M. Goodman and H. Latin. Automated knowledge acquisition from network management databases. In I. Krishnan and W. Zimmer, editors, *Integrated Network Management, II*, pages 541 – 549. Elsevier Science Publishers B.V (North-Holland), Amsterdam, The Netherlands, 1991.

[32] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33:113 – 120, 1989.

[33] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database and Knowledgebase Systems (PODS'97)*, pages 209 – 216, Tucson, Arizona, May 1997. ACM.

[34] V. Gurvich and L. Khachiyan. On generating the irredundant conjunctive and disjunctive normal forms of monotone boolean functions. Technical Report LCSR-TR-251, Rutgers University, 1995.

[35] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'92)*, pages 341 – 350, San Diego, CA, June 1992.

[36] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305 – 308, 1989/90.

[37] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 420 – 431, Zürich, Switzerland, Sept. 1995. Morgan Kaufmann.

[38] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen. Knowledge discovery from telecommunication network alarm databases. In S. Y. Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, pages 115 – 122, New Orleans, Louisiana, USA, Feb. 1996. IEEE Computer Society Press.

[39] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen. TASA: Telecommunication alarm sequence analyzer, or "How to enjoy faults in your network". In *Proceedings of the 1996 IEEE Network Operations and Management Symposium (NOMS'96)*, pages 520 – 529, Kyoto, Japan, Apr. 1996. IEEE.

[40] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 150 – 155, Montreal, Canada, Aug. 1995. AAAI Press.

[41] P. Hoschka and W. Kloesgen. A support system for interpreting statistical data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 325 – 345. AAAI Press, Menlo Park, CA, 1991.

[42] M. Houtsma and A. Swami. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, October 1993.

[43] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58 – 64, Nov. 1996.

[44] G. Jakobson and M. Weissman. Real-time telecommunication network management: Extending event correlation with temporal constraints. In *Integrated Network Management IV*, pages 290 – 301. Chapman & Hall, London, 1995.

[45] G. Jakobson and M. D. Weissman. Alarm correlation. *IEEE Network*, 7(6):52 – 59, Nov. 1993.

[46] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8):1587 – 1595, 1995.

[47] J. F. Jordaan and M. E. Paterok. Event correlation in heterogenous networks using OSI management framework. In H.-G. Hegering and Y. Yemini, editors, *Integrated Network Management, III*, pages 683 – 695. Elsevier Science Publishers B.V (North-Holland), Amsterdam, The Netherlands, 1993.

[48] J. D. Kalbfleisch and R. L. Prentice. *The Statistical Analysis of Failure Time Data*. John Wiley Inc., New York, NY, 1980.

[49] M. Kantola, H. Mannila, K.-J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7):591 – 607, Sept. 1992.

[50] R. Khardon. Translating between Horn representations and their characteristic models. *Journal of Artificial Intelligence Research*, 3:349–372, 1995.

[51] J. Kivinen and H. Mannila. The power of sampling in knowledge discovery. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'94)*, pages 77 – 85, Minneapolis, MN, USA, May 1994. ACM.

[52] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 401 – 407, Gaithersburg, MD, USA, Nov. 1994. ACM.

[53] W. Kloesgen. Efficient discovery of interesting statements in databases. *Journal of Intelligent Information Systems*, 4(1):53 – 69, 1995.

[54] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, Feb. 1994.

[55] A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in *e. coli* DNA. *Nucleic Acids Research*, 22:4768–4778, 1994.

[56] P. Laird. Identifying and using patterns in sequential data. In K. Jantke, S. Kobayashi, E. Tomita, and T. Yokomori, editors, *Algorithmic Learning Theory, 4th International Workshop*, pages 1 – 18, Berlin, 1993. Springer-Verlag.

[57] D. W. Loveland. Finding critical sets. *Journal of Algorithms*, 8:362 – 371, 1987.

[58] H. Mannila and K.-J. Räihä. *Design of Relational Databases*. Addison-Wesley Publishing Company, Wokingham, UK, 1992.

[59] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering*, 12(1):83 – 99, Feb. 1994.

[60] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 146 – 151, Portland, Oregon, USA, Aug. 1996. AAAI Press.

[61] H. Mannila and H. Toivonen. On an algorithm for finding all interesting sentences. In *Cybernetics and Systems, Volume II, The Thirteenth European Meeting on Cybernetics and Systems Research*, pages

973 – 978, Vienna, Austria, Apr. 1996. Austrian Society for Cybernetic Studies.

[62] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In U. M. Fayyad and R. Uthurusamy, editors, *Knowledge Discovery in Databases, Papers from the 1994 AAAI Workshop (KDD'94)*, pages 181 – 192, Seattle, Washington, USA, July 1994. AAAI Press.

[63] H. Mannila, H. Toivonen, and A. I. Verkamo. Finding association rules efficiently in sequential data. Technical Report C-1994-40, University of Helsinki, Department of Computer Science, P.O. Box 26, FIN-00014 University of Helsinki, Finland, July 1994.

[64] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 210 – 215, Montreal, Canada, Aug. 1995. AAAI Press.

[65] C. J. Matheus, G. Piatetsky-Shapiro, and D. McNeill. Selecting and reporting what is interesting. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 495 – 515. AAAI Press, Menlo Park, CA, 1996.

[66] R. Milne, C. Nicol, M. Ghallab, L. Trave-Massuyes, K. Bousson, C. Dousson, J. Quevedo, J. Aguilar, and A. Guasch. TIGER: Real-time situation assessment of dynamic systems. *Intelligent Systems Engineering*, 3(3):103 – 124, 1994.

[67] N. Mishra and L. Pitt. On bounded-degree hypergraph transversal. Manuscript, 1995.

[68] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203 – 226, 1982.

[69] R. A. Morris, L. Khatib, and G. Ligozat. Generating scenarios from specifications of repeating events. In *Second International Workshop on Temporal Representation and Reasoning (TIME-95)*, Melbourne Beach, Florida, Apr. 1995.

[70] S. Muggleton. *Inductive Logic Programming.* Academic Press, London, 1992.

[71] Y. A. Nygate. Event correlation using rule and object based techniques. In *Integrated Network Management IV*, pages 278 – 289. Chapman & Hall, London, 1995.

[72] F. Olken and D. Rotem. Random sampling from B^+ trees. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases (VLDB'89)*, pages 269 – 277, Amsterdam, Aug. 1989.

[73] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'95)*, pages 175 – 186, San Jose, California, May 1995.

[74] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 229 – 248. AAAI Press, Menlo Park, CA, 1991.

[75] G. Piatetsky-Shapiro and C. J. Matheus. The interestingness of deviations. In U. M. Fayyad and R. Uthurusamy, editors, *Knowledge Discovery in Databases, Papers from the 1994 AAAI Workshop (KDD'94)*, pages 25 – 36, Seattle, Washington, USA, July 1994. AAAI Press.

[76] L. D. Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99 – 146, 1997.

[77] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 432 – 444, Zürich, Switzerland, Sept. 1995.

[78] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design & implementation of a sequence database system. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, pages 99 – 110, Mumbay, India, Sept. 1996.

[79] A. Siebes. Data surveying, foundations of an inductive query language. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 269 – 274, Aug. 1995.

[80] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):301 – 316, Aug. 1992.

[81] R. Srikant and R. Agrawal. Mining generalized association rules. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 407 – 419, Zürich, Switzerland, Sept. 1995. Morgan Kaufmann.

[82] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM SIGMOD Conference on*

Management of Data (SIGMOD'96), pages 1 – 12, Montreal, Canada, June 1996.

[83] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology—5th International Conference on Extending Database Technology (EDBT'96)*, pages 3 – 17, Avignon, France, Mar. 1996.

[84] H. Toivonen. Sampling large databases for association rules. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, pages 134 – 145, Mumbay, India, Sept. 1996. Morgan Kaufmann.

[85] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hätönen, and H. Mannila. Pruning and grouping of discovered association rules. In G. N. Yves Kodratoff and C. Taylor, editors, *Workshop Notes of the ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*, pages 47 – 52, Heraklion, Crete, Greece, Apr. 1995. MLnet.

[86] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley Publishing Company, Reading, MA, 1977.

[87] J. T.-L. Wang, G.-W. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of ACM SIGMOD*, pages 115 – 125, Minneapolis, MN, May 1994.

[88] K. Yoda, T. Fukuda, and Y. Morimoto. Computing optimized rectilinear regions for association rules. In D. Heckerman, H. Mannila, D. Pregibon, and R. Uthurusamy, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, pages 96 – 103, Newport Beach, California, USA, Aug. 1997. AAAI Press.

[89] R. Zembowicz and J. M. Zytkow. From contingency tables to various forms of knowledge in databases. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 329 – 349. AAAI Press, Menlo Park, CA, 1996.