Department of Computer Science
Series of Publications A
Report A-1996-5

Discovery of Frequent Patterns in Large Data Collections

Hannu Toivonen

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1996-5

Discovery of Frequent Patterns in Large Data Collections

Hannu Toivonen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium III, Porthania, on December 5th, 1996, at 12 o'clock noon.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-7531-8
Computing Reviews (1991) Classification: H.3.1, I.2.6, F.2.2
Helsinki 1996
Helsinki University Printing House

Discovery of Frequent Patterns in Large Data Collections

Hannu Toivonen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Hannu.Toivonen@cs.helsinki.fi, http://www.cs.helsinki.fi/~htoivone/

PhD Thesis, Series of Publications A, Report A-1996-5
Helsinki, November 1996, 116 pages
ISSN 1238-8645, ISBN 951-45-7531-8

Abstract

Data mining, or knowledge discovery in databases, aims at finding useful regularities in large data sets. Interest in the field is motivated by the growth of computerized data collections and by the high potential value of patterns discovered in those collections. For instance, bar code readers at supermarkets produce extensive amounts of data about purchases. An analysis of this data can reveal useful information about the shopping behavior of the customers. Association rules, for instance, are a class of patterns that tell which products tend to be purchased together.

The general data mining task we consider is the following: given a class of patterns that possibly have occurrences in a given data collection, determine which patterns occur frequently and are thus probably the most useful ones. It is characteristic for data mining applications to deal with high volumes of both data and patterns.

We address the algorithmic problems of determining efficiently which patterns are frequent in the given data. Our contributions are new algorithms, analyses of problems, and pattern classes for data mining. We also present extensive experimental results. We start by giving an efficient method for the discovery of all frequent association rules, a well known data mining problem. We then introduce the problem of discovering frequent patterns in general, and show how the association rule algorithm can be extended to cover this problem. We analyze the problem complexity and derive a lower bound for the number of queries in a simple but realistic model. We then show how sampling can be used in the discovery of exact association rules, and we give algorithms that are efficient especially in terms of the amount of database processing. We also show that association rules with negation and disjunction can be approximated efficiently. Finally, we define episodes, a class of patterns in event sequences such as alarm logs. An episode is a combination of event types that occur often close to each other. We give methods for the discovery of all frequent episodes in a given event sequence.

The algorithm for the discovery of association rules has been used in commercial data mining products, the episode algorithms are used by telecommunication operators, and discovered episodes are used in alarm handling systems.

Computing Reviews (1991) Categories and Subject Descriptors:
H.3.1 Information Storage and Retrieval: Content Analysis and Indexing
I.2.6 Artificial Intelligence: Learning
F.2.2 Analysis of Algorithms and Problem Complexity: Nonnumerical Algorithms and Problems

General Terms:
Algorithms, Theory, Experimentation

Additional Key Words and Phrases:
Data mining, Knowledge discovery, Association rules, Episodes

Acknowledgements

I am most grateful to my advisor, Prof. Heikki Mannila, for guiding me through the doctoral studies. His deep involvement and continued support have been invaluable during the preparation of this dissertation. Large parts of the research reported here present results of joint efforts with him and Dr. Inkeri Verkamo. The work at hand has also benefitted from the insightful comments of Prof. Jyrki Katajainen and Prof. Jorma Tarhio.

This work has been carried out at the Department of Computer Science at the University of Helsinki. I am grateful to the Head of the Department, Prof. Martti Tienari, to Prof. Esko Ukkonen, and to all the personnel of the department for the inspiring and pleasant working environment. In particular, I would like to thank my team-mates in the Data Mining Group for collaboration in the research, and all my friends from the Coffee Room B440 for the refreshing breaks between the research. Financial support from the Academy of Finland, the 350th Anniversary Fund of the University of Helsinki, and the Helsinki Graduate School of Computer Science is gratefully acknowledged. Finally, I would like to thank my parents Orvokki and Tauno Toivonen for their support over the years.

Contents

Chapter 1

Introduction

Information is the currency of our era, but what is its value?
Information is not knowledge, and it surely is not wisdom.
– John Lippman

Data mining aims at the discovery of regularities in large data collections. The rapidly growing interest in the field is stimulated by the large amounts of computerized data available in business and also in science. For instance, supermarkets store electronic copies of millions of receipts, and banks and credit card companies maintain extensive collections of transaction histories. The goal of data mining, also called knowledge discovery in databases or KDD for short, is to analyze these data sets and to find regularities that provide new insight into the business or process that generates the data. A recent overview of research in data mining is given in [FPSSU96].

The current interest towards data mining is understandable: several companies have large collections of data describing their daily operations. For instance, in many countries retailers are required by the law to store copies of the receipts of their sales. Most stores do not use paper copies at all, except the one given to the customer. Instead, their cash registers immediately save the information in electronic format. Or, telecommunication operators store events from their networks for trouble shooting purposes. These companies are now realizing that new methods are emerging for the analysis of the databases they have accumulated more or less as a side product, and that by using these methods they possibly can gain a competitive advantage.

Discovery of association rules [AIS93] has been identified as an important data mining problem. The original motivation for searching association rules came from the need to analyze so called *supermarket basket data*, that is, to examine customer behavior in terms of the purchased products. Association rules describe how often items are purchased together: for instance, an association rule "beer $\Rightarrow$ chips (80 %)" states that four out of five customers that bought beer also purchased chips. Such rules can be useful for decisions concerning, e.g., product pricing, promotions, or store layout.

The discovery of association rules serves as the starting point of this thesis. In Chapter 2 we give an efficient method for finding all association rules that apply to at least a user specified number of rows of a given database, and whose confidence exceeds a threshold given by the user. We experiment with a university course enrollment database where we search for combinations of courses taken by the students.

Association rules are a simple form of frequent patterns. To see what we mean by "frequent", note that it is not interesting to find each and every association rule that logically holds in the analyzed database. Assume, for instance, that a supermarket has only sold one hammer during the analyzed time period, and that the hammer happened to be bought by a customer that also purchased a vase. The inference that every hammer has been sold with a vase is not very useful for any business decisions, nor is the association statistically significant. Rather, we are interested in patterns that occur several times in the database. How many times a pattern should occur to be considered frequent enough depends, of course, on the application domain. It is typical to require, for instance, that at least some small fraction of rows of a database matches the pattern.

In Chapter 3 we consider the general data mining task of discovering frequent patterns of a given class, and present a generic algorithm that solves the task. Actually, the discovery criterion does not need to be frequency: we show how so called exact database rules, inclusion dependencies, and functional dependencies can be discovered using the generic algorithm.

An often criticized aspect of knowledge discovery is that analyzing just one state of a database does not give reliable information: a regularity might exist in the analyzed database only by chance. In Chapter 3 we also describe how the presented method can be used in knowledge discovery from several similar databases or from several states of the same database.

Business data, such as sales information, has always been analyzed and searched for interesting knowledge. What is new is that the current state of technology allows efficient verification of a high number of hypothesis on large collections of data. This is perhaps the point where data mining can best be contrasted with its two close scientific neighbors: statistics and machine learning.

When compared to data mining, statistics is geared more towards analyzing or fitting relatively complicated models to the data. In statistics a great deal of effort is put into specifying the exact questions to be asked of the data, whereas data mining methods typically evaluate automatically a large number of relatively simple questions. Several aspects that are carefully dealt with in statistics, such as significance, variation, or explicit bias, have been largely ignored in data mining. A statistical perspective on data mining can be found in [EP96].

Many researchers in the machine learning community like to think that the field of machine learning covers also data mining. In any case, typical machine learning and typical data mining can be contrasted. Much of the research in machine learning is devoted to the discovery of a regularity between given input and output attributes. Data mining often differs from such tasks in two aspects: there is less focus, so regularities between unexpected attributes can be found, and the discovered regularities are simpler. As an example, consider the analysis of supermarket basket data. Given cat food as the target product, a machine learning method, such as the induction of decision trees, could probably find a complex expression identifying quite reliably those shopping baskets that contain cat food. For data mining, it would be more typical not to specify a target product, but to search for simple patterns, such as association rules, that are strong with respect to any product. In other words, data mining tends to trade the complexity of patterns for complexity in selecting the components of patterns.

We analyze in Chapter 4 the complexity of the problem of discovering all frequent patterns. We derive a lower bound for the number of queries of the form "What is the frequency of pattern φ" that are needed to find all frequent patterns, and we show that our algorithm for the discovery of association rules is optimal under some simplifying assumptions. Using the developed concepts, we also derive bounds for the well known problem of finding functional dependencies. The bounds for functional dependencies are not new, but the simplicity of their derivation demonstrates the power of the general formulation.

A contrast between machine learning and data mining is often seen also in the amount of the analyzed data. In data mining it is assumed that the data collections are large, up to millions or billions of rows. Large data sets are considered necessary for reliable results; unfortunately, however, the execution time of mining algorithms depends heavily on the database. Although the time complexity of discovering association rules is linear in the number of rows of the database, all the existing algorithms require multiple passes over the database. Subsequently the size of the database is the most influential factor of the execution time for large databases.

In Chapter 5 we develop a method that essentially reduces the database activity to one pass. We discuss the use of sampling in the discovery of association rules, and we analyze the accuracy of results obtained from a random sample. The main contributions are, however, new methods for the discovery of exact association rules. The methods first use a sample to initialize the discovery, and then make one database pass to compute the results. In terms of database activity, this method outperforms all known methods and is attractive especially for large databases. We give experimental results showing that the methods work well in practice.

So far we have been mainly interested in the efficiency issues of the discovery. It is time to note that the class of association rules is rather simple: only positive connections between sets of items can be expressed. In Chapter 6 we consider a more powerful rule formalism, the class of Boolean rules. Basically, Boolean rules are association rules with negation and disjunction, so rules such as "diet soda $\Rightarrow$ *not* chips" can be expressed. We demonstrate that the task of discovering *all* Boolean rules that hold in a database is not useful, nor always feasible. We then show that Boolean rules can be derived from the same information that is used to derive normal association rules. All the necessary information is not, however, available for all frequent Boolean rules. We show experimentally that good approximations for the frequencies and confidences can be computed from the available information.

There are several application areas in which the data is sequential in nature. Consider, for instance, monitoring some system, such as a telecommunication network, a World Wide Web server, or a computer application and its user. A lot of information about the activities can be recorded. As an example, the switches and other critical equipment in telecommunication networks produce large amounts of notifications for trouble shooting purposes. Such a sequence of events produced by a network contains detailed but fragmented information about how the network behaves, and about the relationships of the components. Discovery of regularities in such sequences would be useful, e.g., for the prediction of faults.

In Chapter 7 we present episodes, a class of frequent patterns in sequences of events. An episode is a combination of events occurring frequently together; once the frequent episodes are known, it is possible to generate rules that describe co-occurrences of events, and that can be used for prediction. We present methods for the discovery of episodes in sequences, and we give experimental results with telecommunication event databases.

Finally, in Chapter 8, we conclude with a short discussion.

The work described in this thesis consists on one hand of rather theoretical work on algorithms and concepts for the discovery of frequent patterns, and on the other hand of practical experiments that support the theory. Most of the results have been published before, many of them in joint articles with Prof. Heikki Mannila and Dr. Inkeri Verkamo: the algorithm for finding association rules [AMS+96, MTV94a], the generalized problem, algorithm, and analysis [MT96c], the methods for finding association rules using sampling [Toi96], the idea of approximate discovery of Boolean rules [MT96b], and the problem of discovering frequent episodes in sequences [MTV94b, MTV95]. All the material in this thesis represents original work, except for the definition of association rules and the basic rule generation method in the beginning of Chapter 2.

The other half of the work, which is not described in this thesis, has been the construction of a discovery system called Freddie, which implements the methods. The development of both the methods and Freddie has taken place in collaboration with data mining activities, where they have been used to analyze real databases. The development of methods for analyzing sequences has, in particular, benefitted from working with real data and real problems. The projects have gained from the algorithms and Freddie, too: regularities discovered in telecommunication databases have been incorporated into event handling software of telecommunication operators [HKM+96a]. TASA, a discovery system based on Freddie and on tools for browsing the discovered rules, is described in [HKM+96a, HKM+96b].

Chapter 2

Discovery of association rules

Predicting the future is easy. Getting it right is the hard part.
– Howard Frank

We start by studying the discovery of association rules, a simple but important case of frequent patterns. 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. Section 2.3 starts the contributions of this thesis. In that section we give an efficient method for the discovery of all frequent sets. Experiments with the method are described in Section 2.4. Finally, we review extensions and other related work in Section 2.5. Parts of the work described in this chapter have been published in [AMS$^+$96, MTV94a].

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 *binary database 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 denoted by $||r|| = \sum_{t \in r} |t|$. $\square$

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 binary database r over the set $R = \{A, \ldots, K\}$.

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

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

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 binary database 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 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 $fr(X, r) \geq min\_fr$. □

[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 binary database r in relational form over 0/1-valued attributes $\{A, \ldots, K\}$.

Example 2.4 Consider the binary database 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 "binary database". 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 user and depends on the application. For notational convenience, we introduce notations for collections of frequent sets.

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

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

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 binary database; we 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 binary database 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 $conf(X \Rightarrow Y, r)$, is $\frac{|\mathcal{M}(X \cup Y, r)|}{|\mathcal{M}(X, r)|}$. The *frequency* $fr(X \Rightarrow Y, r)$ of $X \Rightarrow Y$ in r is $fr(X \cup Y, r)$. We write simply $conf(X \Rightarrow Y)$ and $fr(X \Rightarrow Y)$ 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 $fr(X \Rightarrow Y, r) \geq min\_fr$ and $conf(X \Rightarrow Y, r) \geq min\_conf$. □

In other words, the confidence $conf(X \Rightarrow Y, r)$ 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 [AIS93] 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 binary database can be discovered in two phases [AIS93]. 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 [AIS93]) 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 subsection. Note that indentation is used in the algorithms to specify the extent of loops and conditional statements.

Algorithm 2.9
Input: A set R, a binary database 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 $\mathcal{F}(r, min\_fr) := \{X \subseteq R \mid fr(X, r) \geq min\_fr\}$;
3. // Generate rules:
4. **for** all $X \in \mathcal{F}(r, min\_fr)$ **do**
5.   **for** all $Y \subset X$ with $Y \neq \emptyset$ **do**
6.   **if** $fr(X)/fr(X \setminus Y) \geq min\_conf$ **then**
7.   output the rule $X \setminus Y \Rightarrow Y$, $fr(X)$, and $fr(X)/fr(X \setminus Y)$;

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 $fr(X \cup Y) \geq min\_fr$, 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 episodes 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, 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 $|X| - 1$.

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.

Definition 2.13 Given a collection $\mathcal{F}_l(r)$ of frequent sets of size l, the *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 that finds all frequent sets. The subtasks of the algorithm, for which only specifications are given, are described in detail in following subsections.

Algorithm 2.14
Input: A set R, a binary database 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:

1. $\mathcal{C}_1 := \{\{A\} \mid A \in R\}$;
2. $l := 1$;
3. **while** $\mathcal{C}_l \neq \emptyset$ **do**
4.   // Database pass (Algorithm 2.22):
5.   compute $\mathcal{F}_l(r) := \{X \in \mathcal{C}_l \mid \mathit{fr}(X, r) \geq \mathit{min\_fr}\}$;
6. $l := l + 1$;
7.   // Candidate generation (Algorithm 2.18):
8.   compute $\mathcal{C}_l := \mathcal{C}(\mathcal{F}_{l-1}(r))$;
9. **for** all l and **for** all $X \in \mathcal{F}_l(r)$ **do** output X and $\mathit{fr}(X, r)$;

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 $\mathcal{S} = \{X \subseteq R \mid |X| = l\}$ of sets of size l, there exists a binary database 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 $\mathit{min\_fr} = 1/|r|$. 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 binary database 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. □

Algorithm 2.14 could, in principle, take advantage of situations similar to the above examples. Such situations do not, however, occur 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 Candidate generation

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 the 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.

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 $X \cup Y$;

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). □

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 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 $\mathcal{C}(\mathcal{F}_l(r))$, $\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r)))$, $\mathcal{C}(\mathcal{C}(\mathcal{C}(\mathcal{F}_l(r)))), \ldots$ 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,

[2] Actually, the values of Y can be determined more efficiently with some extra bookkeeping information stored every time candidates are generated. A closely related method using this idea is presented in Section 7.3.

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

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

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. □

2.3.2 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.

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

// Initialization:
for all $A \in R$ **do** $A.\mathit{is\_contained\_in} := \emptyset$;
for all $X \in \mathcal{C}_l$ and **for** all $A \in X$ **do**
    $A.\mathit{is\_contained\_in} := A.\mathit{is\_contained\_in} \cup \{X\}$;
for all $X \in \mathcal{C}_l$ **do** $X.\mathit{freq\_count} := 0$;
// Database access:
for all $t \in r$ **do**
    for all $X \in \mathcal{C}_l$ **do** $X.\mathit{item\_count} := 0$;
    for all $A \in t$ **do**
        for all $X \in A.\mathit{is\_contained\_in}$ **do**
            $X.\mathit{item\_count} := X.\mathit{item\_count} + 1$;
            if $X.\mathit{item\_count} = l$ **then** $X.\mathit{freq\_count} := X.\mathit{freq\_count} + 1$;
// Output:
for all $X \in \mathcal{C}_l$ **do**
    if $X.\mathit{freq\_count}/|r| \geq \mathit{min\_fr}$ **then** output X and $X.\mathit{freq\_count}/|r|$;

For each item $A \in R$ we maintain a list $A.\mathit{is\_contained\_in}$ of candidates that contain A. For each candidate X we maintain two counters. Variable $X.\mathit{freq\_count}$ is used to count the number of rows that X matches, whereas variable $X.\mathit{item\_count}$ 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 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 $A.\mathit{is\_contained\_in} = \{X \mid X \in \mathcal{C}_l \text{ and } A \in X\}$. 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 $X.\mathit{item\_count}$. The value of $X.\mathit{item\_count}$ 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 $\mathcal{O}(|R| + l\,|\mathcal{C}_l| + |\mathcal{C}_l| + ||r|| + |r|\,|\mathcal{C}_l| + l\,|r|\,|\mathcal{C}_l| + l\,|\mathcal{C}_l|) = \mathcal{O}(||r|| + l\,|r|\,|\mathcal{C}_l| + |R|)$. $\square$

There are practical improvements for the algorithm. For instance, to determine whether a set $X \subseteq R$ is 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.

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.

2.4 Experiments

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 average 4.3 courses. The experiments in this and the following chapters have been run on a PC with 90 MHz Pentium processor and 32 MB 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

	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	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.

are shown. The row "match" shows how large fraction of candidate sets 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.

The table shows that 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 is at least 90 %.

The times of the database passes of our experiments are roughly linear

	Candidates		Frequent sets		
Size	Count	Time (s)	Count	Time (s)	Match
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 $min\_fr = 0.025$.

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 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 Extensions and related work

Since their introduction in 1993 [AIS93], 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 4.

Candidate generation Definition 2.13 and Algorithm 2.14 for candidate generation were presented independently in [AS94, MTV94a]. We touch this subject again in Chapter 7, where we present in detail an algorithm that can deal, in addition to sets, also with multisets and ordered sets.

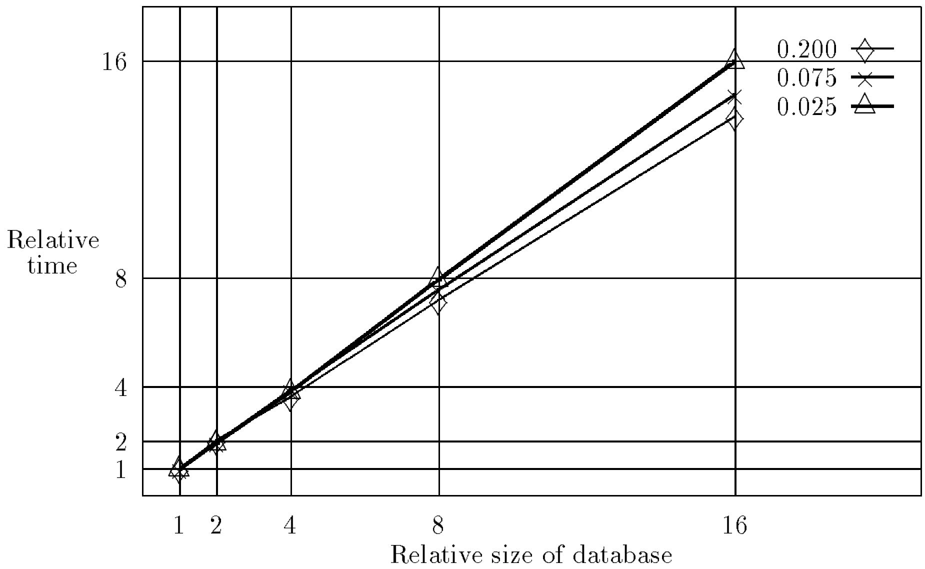


Figure 2.3: Results of scale-up tests.

The frequent set discovery method presented in [AIS93] also works in a generate and test fashion, but the candidate generation is quite different. An essential difference is that in [AIS93] 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 [AIS93] has been shown to perform worse than the algorithms of this section [AMS+96, AS94, MTV94a].

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 [AS94].

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 $|X|-1$ [AS94]. One can create a new temporary binary database with an item for each candidate, and to fill the database in 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 [HKMT95, SON95]. 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 [HKMT95]. 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 [SON95], 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 [AS94], hashing is used during the database pass to efficiently determine the collection of candidates that match a row. The method of [PCY95], 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 [AIS93] that uses SQL for the discovery of frequent sets [HS93].

Item hierarchies Association rule algorithms have been generalized to work with items arranged to hierarchies or taxonomies [HF95, HKMT95, SA95]. 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 [FMMT96, SA96a]. 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.

Rule selection Thousands or even millions of association rules may hold in a database, so thresholds for confidence and frequency are clearly not enough to point out the most useful rules. The problem of rule ranking is far from trivial. An interactive method for browsing rules is based on templates [KMR$^+$94], regular expressions that specify the rules that are to be selected or explicitly excluded. Automatic methods for pruning redundant rules and for clustering rules have been considered in [TKR$^+$95].

Each of the following chapters discusses some aspect strongly related to frequent sets. In Chapter 3 we present a generalization of Algorithm 2.14 for the discovery of different types of frequent patterns. The problem of discovering frequent patterns, including the discovery of frequent sets, is analyzed in Chapter 4. Methods that use sampling to reduce the amount of database processing are presented in Chapter 5. In Chapter 6, inferring association rules with negation and disjunction from frequent sets is discussed. Finally, Chapter 7 presents the problem of discovering frequent episodes in event sequences.

Chapter 3

Discovery of all frequent patterns

Do we need a solution—couldn't we just enjoy the problem?
– An anonymous PhD student

In this chapter we consider a generalization of the problem of discovering all frequent sets: given a set of patterns and a database, find those patterns that are frequent in the database. We give an algorithm for this task in Section 3.2; the algorithm is a direct generalization of Algorithm 2.14. In Section 3.3 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 3.4, we extend the setting and the algorithm for discovery in several database states. A review of related work is postponed to follow the analysis of the approach in Chapter 4. Parts of the material of this chapter have been published in [MT96c].

3.1 The discovery task

A significant fraction of the discussion in this and the following chapter is formalized in the framework of the relational data model, so we start by defining its basic concepts.

Definition 3.1 A *relation schema* R is a set $\{A_1, \ldots, A_m\}$ of *attributes*. Each attribute A_i has a *domain*, denoted by $Dom(A_i)$. A *row* over the schema R is a sequence $\langle a_1, \ldots, a_m \rangle$ such that $a_1 \in Dom(A_i)$ for all $i = 1, \ldots, m$. Given a row t, the ith value of t is denoted by $t[A_i]$. A *relation* over R is a set of rows over R. A *relational database* is a set of relations over a set of relation schemas, collectively called the *database schema*. □

We can now define 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 3.2 Assume that $\mathcal{P}$ is a set and q is a predicate $q : \mathcal{P} \times \{\mathbf{r} \mid \mathbf{r} \text{ is a database}\} \to \{\text{true, false}\}$. 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 3.3 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. From now on we consider only the situation where the predicate q is monotone with respect to a given partial order on the patterns.

Definition 3.4 Let $\mathcal{P}$ be a finite 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 *strength hierarchy* on $\mathcal{P}$ with respect to q. If we have $\theta \preceq \varphi$, then φ is said to be *stronger* than θ and θ to be *weaker* than φ. If $\theta \preceq \varphi$ and not $\varphi \preceq \theta$ we write $\theta \prec \varphi$. □

Example 3.5 The set inclusion relation $\subseteq$ is a strength hierarchy for frequent sets. Given two item sets $X, Y \subseteq R$, the set X is weaker, $X \preceq Y$, if and only if $X \subseteq Y$. That is, $X \preceq Y$ implies that if the stronger set Y is frequent then the weaker set X is frequent, too. □

For practical purposes the strength hierarchy 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 strength hierarchy $\preceq$ 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 strength hierarchy with respect to q: a pattern can be statistically significant even when a weaker 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.

3.2 A generic algorithm

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

Definition 3.6 Given a strength hierarchy $\preceq$ 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 $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q) = \{\varphi \in \mathcal{T}(\mathcal{P}, \mathbf{r}, q) \mid level(\varphi) = l\}$. □

Algorithm 3.7, analogical to Algorithm 2.14, finds all frequent patterns. It works in a breadth-first manner, starting with the set $\mathcal{C}_1$ of the weakest patterns, and then generating and evaluating stronger and stronger candidate patterns. The algorithm prunes those patterns that cannot be frequent given all the frequent patterns obtained in earlier iterations.

Algorithm 3.7
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 strength hierarchy $\preceq$ 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 $\mathcal{T}_l(\mathcal{P}, \mathbf{r}, q)$;

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 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 3.8 Algorithm 3.7 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 $\mathit{level}(\varphi) = l$ and thus for all patterns $\theta \prec \varphi$ we have $\mathit{level}(\theta) < l$. Since $\mathcal{T}_i(\mathcal{P}, \mathbf{r}, q)$ has been computed for each $i < l$, each $\theta \prec \varphi$ 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 weaker patterns $\theta \prec \varphi$ on every level less than $\mathit{level}(\varphi)$, and since $\preceq$ is a strength hierarchy 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 strength hierarchy $\preceq$. An efficient method for accessing the weaker and stronger patterns on neighboring levels is useful, or otherwise finding the collection of valid candidates may be expensive.

3.3 Examples

We now look at the applicability of Algorithm 3.7 for some well known knowledge discovery tasks. We formulate three problems as tasks of discovering patterns that are selected by a given predicate: the discovery of exact database rules, the inference of inclusion dependencies, and the inference of functional dependencies. The purpose is to show how the algorithm fits different discovery tasks, and to demonstrate how the selection criterion q can actually be based on other properties of patterns than the frequency.

3.3.1 Exact database rules

Exact database rules [PS91] 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 3.9 Given an attribute A, a *taxonomy* on A is a set $T(A)$ such that $Dom(A) \subseteq T(A)$ 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 3.10 Consider an attribute *department* for which the domain *Dom*(*department*) is {*dept_1*, *dept_2*, ..., *dept_15*}. Now a taxonomy *T*(*department*) could consist of *Dom*(*department*) ∪ {*management_dept*, *production_dept*, *sales_dept*}, 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 3.11 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. □

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 strength hierarchy for exact rules.

Theorem 3.12 The following relation $\preceq$ is a strength hierarchy 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 strength hierarchy 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 weaker 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. □

The proof shows that the strength hierarchy $\preceq$ 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 strength hierarchy 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 3.7 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 3.13 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 weakest patterns considered by Algorithm 3.7 are such as

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

and

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

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

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

does not hold, then rules such as

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

and

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

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 strength hierarchy. The following example demonstrates this.

Example 3.14 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

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

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

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

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

Algorithm 3.7 considers in each database pass a collection of candidate rules, where all the rules are on the same level. The KID3 algorithm [PS91] 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 3.7, in turn, does not take any advantage of the fact that rules close to each other in the strength hierarchy 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 [PS91].

3.3.2 Inclusion dependencies

Next we consider the discovery of inclusion dependencies [CFP84] that hold in a given database. In essence, an inclusion dependency $R[X] \subseteq S[Y]$ means that all the values of attributes X in a given relation over R are also values of attributes Y in a given relation over S. Practical databases have inclusion dependencies, since most of the data is interconnected. Discovery of inclusion dependencies is useful in database reverse engineering [KA95, KMRS92, MR92a].

Definition 3.15 Given a database schema $\mathbf{R}$, an *inclusion dependency* over $\mathbf{R}$ is an expression $R[X] \subseteq S[Y]$, where R and S are relation schemas of $\mathbf{R}$, and $X = \langle A_1, \ldots, A_k\rangle$ and $Y = \langle B_1, \ldots, B_k\rangle$ are equal-length sequences of attributes of R and S, respectively, that do not contain duplicates.

Given a database $\mathbf{r}$ over $\mathbf{R}$ with r and s the relations corresponding to R and S, respectively, the inclusion dependency $R[X] \subseteq S[Y]$ *holds* in $\mathbf{r}$ if for every row $t \in r$ there exists a row $u \in s$ such that $t[A_i] = u[B_i]$ for $i = 1, \ldots, k$. An inclusion dependency $R[X] \subseteq S[Y]$ is *trivial* if $R = S$ and $X = Y$. □

The problem is now the following: given a database $\mathbf{r}$ over a database schema $\mathbf{R}$, discover the collection of non-trivial inclusion dependencies that hold in $\mathbf{r}$. This collection is the theory $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$, where the set $\mathcal{P}$ consists of all non-trivial inclusion dependencies over $\mathbf{R}$, and q is true if and only if the inclusion dependency holds in $\mathbf{r}$. Next we show what is a suitable strength hierarchy.

Theorem 3.16 The following relation $\preceq$ is a strength hierarchy with respect to $\mathcal{P}$ consisting of all non-trivial inclusion dependencies over $\mathbf{R}$:

$$(R[X] \subseteq S[Y]) \quad \preceq \quad (R'[\langle A_1, \ldots, A_k \rangle] \subseteq S'[\langle B_1, \ldots, B_k \rangle])$$

if and only if $R = R'$, $S = S'$, and for some $\{i_1, \ldots, i_h\} \subseteq \{1, \ldots, k\}$ we have $X = \langle A_{i_1}, \ldots, A_{i_h} \rangle$ and $Y = \langle B_{i_1}, \ldots, B_{i_h} \rangle$.

Proof By definition, given any row $t \in r$ there exists a row $u \in s$ such that $t[A_i] = u[B_i]$ for all $i \in \{1, \ldots, k\}$. The claim holds trivially for any subset of $\{1, \ldots, k\}$, particularly the subset $\{i_1, \ldots, i_h\}$, and thus the dependency $R[X] \subseteq S[Y]$ holds. □

Again, the strength hierarchy $\preceq$ is a restriction of the converse of the semantic implication. With these specifications Algorithm 3.7 starts the search with inclusion dependencies $R[\langle A \rangle] \subseteq S[\langle B \rangle]$ over all $R, S \in \mathbf{R}$ and all $A \in R, B \in S$ (except the trivial ones with $R = S$ and $A = B$). It then expands the attribute lists in the dependencies until the dependencies do not hold.

Example 3.17 Consider two relation schemas $R = \{A, B, C, D\}$ and $S = \{E, F, G\}$ in $\mathbf{R}$. Among the first dependencies that are tested are, e.g., $R[\langle A \rangle] \subseteq R[\langle B \rangle]$ and $S[\langle F \rangle] \subseteq R[\langle C \rangle]$. Later, the dependency $R[\langle A, B, D \rangle] \subseteq S[\langle G, F, E \rangle]$ will be tested if and only if the following dependencies all hold: $R[\langle A, B \rangle] \subseteq S[\langle G, F \rangle], R[\langle A, D \rangle] \subseteq S[\langle G, E \rangle]$, and $R[\langle B, D \rangle] \subseteq S[\langle F, E \rangle]$. Here, as with frequent sets, for a pattern φ to be accepted as candidate it suffices that the selection criterion q is true for the weaker patterns on level $level(\varphi) - 1$. If the criterion is true for them all, it follows that the criterion is true for all weaker patterns, too, which is the requirement in Algorithm 3.7. □

The number of evaluations of q is smaller for Algorithm 3.7 than for the best known previous algorithms for the problem. Those algorithms take partial advantage of the strength hierarchy: for candidate patterns they require only that weaker patterns at level 1 are selected by q, not that *all* weaker patterns are selected [KMRS92].

In the same way that exact database rules can be generalized to strong rules, we could define *almost correct inclusion dependencies* by allowing for small inconsistencies in the database. This could be done, e.g., by defining

$q(R[X] \subseteq S[Y], \mathbf{r})$ to be true if and only if for at least a fraction c of the rows of r there exists a row of s with the desired properties, where r and s are relations corresponding to R and S. Note that the partial order $\preceq$ given above is a strength hierarchy also with this criterion.

3.3.3 Functional dependencies

Next we consider the discovery of functional dependencies [Cod70] between attributes in a database; this is another example of database reverse engineering by the discovery of integrity constraints.

Definition 3.18 Given a relation schema R, a *functional dependency* over R is an expression $X \to A$, where $X \subseteq R$ and $A \in R$. The dependency is *true* in a given relation r over R if for all pairs of rows $t, u \in r$ we have: if $t[B] = u[B]$ for all $B \in X$, then $t[A] = u[A]$. A functional dependency $X \to A$ is *trivial* if $A \in X$. □

Theorem 3.19 The following relation $\preceq$ is a strength hierarchy with respect to the set $\mathcal{P}$ consisting of all non-trivial functional dependencies over R: $(X \to A) \preceq (Y \to B)$ if and only if $A = B$ and $Y \subseteq X$.

Proof Consider a relation r over R and the set of pairs $t, u \in r$ such that $t[C] = u[C]$ for all $C \in Y$. Denote this set of pairs by $H(Y)$, and denote the corresponding set for X by $H(X)$. Since $Y \subseteq X$, we have $H(Y) \supseteq H(X)$.

Now, since $Y \to B$ holds, for each pair (t, u) in $H(Y)$ we have $t[B] = u[B]$; thus for each pair (t, u) in $H(X) \subseteq H(Y)$ we have $t[B] = u[B]$, and thus $X \to B$ must hold. □

Algorithm 3.7 starts from the dependencies $R \backslash \{A\} \to A$ for all $A \in R$. In each iteration, the size of the left-hand sides decreases by one. The number of database passes is $1 + |R \setminus X|$, where X is the smallest set such that $X \to A$ is true in r for some A. Note that for a large relation schema R it is likely that there will be many iterations, even though the answer might be representable succinctly as the set of dependencies with minimal left-hand sides.

Example 3.20 The problem of many iterations can be partially avoided by shifting the focus from the minimal left-hand sides of true functional dependencies to the maximal left-hand sides of *false* functional dependencies. That is, a conversed predicate q' and a conversed strength hierarchy $\preceq'$ could be defined as: $q'(X \to A, r)$ is true if and only if $q(X \to A, r)$ is not true, and $(X \to A) \preceq' (Y \to B)$ if and only if $(Y \to B) \preceq (X \to A)$.

The search for false functional dependencies starts with the empty set as the left-hand side, i.e., with dependencies such as $\emptyset \to A, \emptyset \to B$, and so on. In the next iteration, dependencies $A \to B$ are tested, over all $A \neq B$ in R such that $\emptyset \to B$ is not true. The left-hand sides grow by one in each iteration as long as the dependencies do not hold. For instance, the dependency $\{A, B, C\} \to D$ is considered as a candidate only if the dependencies $\{A, B\} \to D$, $\{A, C\} \to D$, and $\{B, C\} \to D$ are all false.

Also in this conversed case it can happen that many iterations are necessary, as there can be a large set of attributes that does not derive a target attribute. A useful output of the algorithm would in this conversed approach be the set of candidate patterns that are not selected by the predicate, i.e., those tested functional dependencies that are true. These have the minimal left-hand sides of true functional dependencies. □

The strength hierarchy given above is used by many of the most efficient known algorithms for discovering functional dependencies [MR92b, MR94]. These algorithms work more in a depth-first like manner, jumping in the strength hierarchy. They take advantage of the fact that, given an attribute sequence $\langle A_1, \ldots, A_k \rangle$ and an attribute B, the predicate q can be evaluated efficiently at the same time for all functional dependencies $\{A_1, \ldots, A_i\} \to B$ with $1 \leq i \leq k$.

3.4 Discovery in several database states

A criticized aspect of knowledge discovery is that analyzing just one database state 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 3.7 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 3.21 Given a selection criterion q over a set $\mathcal{P}$ of patterns and a *frequency threshold 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 *min_fr*. □

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

Theorem 3.22 Let $\mathcal{P}, q$, and Q be as in Definition 3.21, and let $\preceq$ be a strength hierarchy with respect to q. Then $\preceq$ is a strength hierarchy also with respect to the global selection criterion Q.

Proof By definition the relation $\preceq$ is a strength hierarchy 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 weaker patterns $\theta \preceq \varphi$, and thus $Q(\theta, \boldsymbol{r})$ must hold. □

Since $\preceq$ is a strength hierarchy with respect to the global selection criterion Q, Algorithm 3.7 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 strength hierarchy $\preceq$ both locally and globally.

Consider Algorithm 3.7 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 weaker 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 weaker 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 weaker 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 $|\{i \mid \mathcal{C}_l^i\}| < min\_fr \cdot n$. 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 4

Complexity of finding all frequent patterns

Now I have a solution—but does it fit the problem?
– The same, frustrated PhD student

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 4.1 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 4.2, as well as in the analysis of the problem of finding all frequent patterns in Section 4.3. Section 4.4 contains some notes about the complexity of evaluating selection criteria. In Section 4.5 we return to the concept of border, and show that it has strong connections to transversals on hypergraphs. Work related to the task of discovering all frequent patterns, to the generic algorithm, and to the analysis is reviewed in Section 4.6. Many results of this chapter have appeared in [MT96c].

4.1 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 strong patterns in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$: every pattern weaker than any of those is selected by q, and the rest are not. The collection of maximally strong patterns in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and, correspondingly, the collection of *minimally* strong 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 4.1 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 weaker patterns than φ are in $\mathcal{S}$ and no pattern stronger than φ is in $\mathcal{S}$:

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

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 strongest patterns in $\mathcal{S}$, the negative border consists of the weakest 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 $\mathcal{S}$.

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

Example 4.2 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.,

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

□

4.2 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 3.7 evaluates only patterns in the result, i.e., frequent patterns, and patterns in the negative border of the collection of frequent patterns.

Theorem 4.3 Let $\mathcal{P}, \mathbf{r}$, and q be as in Algorithm 3.7. Algorithm 3.7 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 3.7, the specification of the candidate collection:

8. compute $\mathcal{C}_l := \{\varphi \in \mathcal{P} \mid level(\varphi) = l$ and $\theta \in \mathcal{T}_{level(\theta)}(\mathcal{P}, \mathbf{r}, q)$ 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 weaker 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 $\mathcal{C}_{level(\varphi)}$. When $l = level(\varphi)$, all weaker patterns than φ have been evaluated in earlier iterations since their levels are less than $level(\varphi)$. All weaker patterns $\theta \prec \varphi$ are in $\mathcal{T}(\mathcal{P}, \mathbf{r}, q)$ and thus in $\mathcal{T}_{level(\theta)}(\mathcal{P}, \mathbf{r}, q)$. So when $l = level(\varphi)$, the pattern φ will be in $\mathcal{C}_{level(\varphi)}$.

Finally, it can be shown, as in the proof of Theorem 3.8, that $\mathcal{C}_{level(\varphi)}$ 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 3.7 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 4.4 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 $\mathcal{F}(r, min\_fr) \cup \mathcal{B}d^-(\mathcal{F}(r, min\_fr))$.

Proof The claim follows directly from the fact that Algorithm 2.14 is an instance of Algorithm 3.7 and from Theorem 4.3. □

4.3 Problem complexity

Let us 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 4.5 Let $\mathcal{P}$ and $\mathcal{S} \subseteq \mathcal{P}$ be sets of patterns, $\mathbf{r}$ a database, q a selection criterion, and $\preceq$ a strength hierarchy. 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 weaker patterns $\theta \prec \varphi$ are selected by the definition of border, none of the stronger 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 4.6 Let $\mathcal{P}$ be a set of patterns, $\mathbf{r}$ a database, q a selection criterion, and $\preceq$ a strength hierarchy. 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 4.5. □

This result gives corollaries about the complexity of discovery tasks. Consider first 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 4.7 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 4.6. □

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.

Another corollary gives a result about finding functional dependencies that in the more specific setting is not easy to find [MR92a, MR92b]. For simplicity, we present the result here for the case of finding the keys of a relation. We define keys first.

Definition 4.8 Given a relation r over a relation schema R, a subset X of R is a *key* of r if and only if the functional dependency $X \rightarrow A$ holds for all A in R, i.e., no two rows agree on X. A key X is *minimal* if no subset $Y \subset X$ is a key. The set of all keys of r is denoted by KEYS(r). □

Recall from Theorem 3.19 that the converse of set inclusion is a suitable strength hierarchy for the left-hand sides of functional dependencies, i.e., for keys in this case: $X \preceq Y$ if and only if $Y \subseteq X$. That is, a superset X of a key Y is always a key and the collection KEYS(r) consists of all supersets of the minimal keys. The set of minimal keys is the positive border $\mathcal{B}d^+(\mathrm{KEYS}(r))$.

Corollary 4.9 ([MR92b]) Given a relation r over a relation schema R, finding the minimal keys of r requires at least $|\mathrm{MAX}(r)|$ evaluations of the predicate "Is X a key in r", where MAX(r) is the set of all maximal subsets of R that do not contain a key.

Proof First note that $\mathrm{KEYS}(r) = \mathcal{T}(\mathcal{P}, r, q)$, where $\mathcal{P}$ is the power set of R and the predicate $q(X, r)$ is "Is X a key in r". The set of minimal keys is the positive border $\mathcal{B}d^+(\text{KEYS}(r))$. Recall that the positive border specifies the whole theory $\text{KEYS}(r)$. By Corollary 4.6, any algorithm computing the theory $\text{KEYS}(r)$ using only predicate "Is X a key in r" must evaluate the whole border $\mathcal{B}d(\text{KEYS}(r))$. Finally note that $\text{MAX}(r)$ is a subset of the border, namely the negative border $\mathcal{B}d^-(\text{KEYS}(r))$. Thus, $\text{MAX}(r)$ has to be evaluated. □

Example 4.10 Given a relation r over $R = \{A, B, C, D\}$, suppose somebody tells us that $\{A, B\}$ and $\{A, C\}$ are the minimal keys of r, i.e.,

$$\text{KEYS}(r) = \{X \subseteq R \mid \{A, B\} \subseteq X \text{ or } \{A, C\} \subseteq X\}.$$

To verify this, we have to check the border $\mathcal{B}d(\text{KEYS}(r))$. The positive border consists of the given minimal keys, $\mathcal{B}d^+(\text{KEYS}(r)) = \{\{A, B\}, \{A, C\}\}$. The negative border consists of the weakest patterns that are not frequent. This means that Y is in $\mathcal{B}d^-(\text{KEYS}(r))$ if and only if all proper supersets of Y are keys, but Y is not a key. Thus we have $\mathcal{B}d^-(\text{KEYS}(r)) = \{\{A, D\}, \{B, C, D\}\}$, and we have to inspect the sets $\mathcal{B}d(\text{KEYS}(r)) = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C, D\}\}$ to determine whether $\{A, B\}$ and $\{A, C\}$ and their supersets really are the only keys of r. □

The advantage of Corollary 4.6 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 4.5 where we show a connection between the problem of finding the border and the hypergraph transversal problem.

4.4 Complexity of computing selection criteria

Above we have analyzed the complexity of the discovery task in terms of the number of evaluations of the selection criterion q. Next we consider shortly the complexity of evaluating q.

In the case of frequent sets, a linear pass through the database is sufficient for finding whether a set $X \subseteq R$ is frequent. A linear pass suffices also for determining whether an exact database rule $C_1 \Rightarrow C_2$ holds or not.

For the integrity constraints considered, inclusion dependencies and functional dependencies, the situation is different. To verify whether an inclusion dependency $R[X] \subseteq S[Y]$ holds, one usually sorts the relations corresponding to relation schemas R and S. Thus the complexity is in the worst case of the order $\mathcal{O}(n \log n)$ for relations of size n. Sorting the relation or hashing it is also required in a comparison-based model of computation for verifying whether a functional dependency $X \rightarrow A$ holds.

The essential difference between finding frequent sets and finding integrity constraints is, however, not the difference between linear and $\mathcal{O}(n \log n)$ time complexities. When searching for frequent sets, several unrelated candidate sets can be evaluated simultaneously in one pass through the database. To verify the truth of a set of inclusion or functional dependencies requires in general as many passes through the database as there are dependencies. On the other hand, several *related* dependencies can be evaluated efficiently at the same time. For instance, algorithms for discovering functional dependencies evaluate full paths between the weakest and the strongest patterns at once. They record the place where the path crosses the border, and they make database passes until the crossings uniquely define the whole border. Note that each path of related functional dependencies can contain at most one pattern in the positive and one in the negative border. The number of passes is thus at least $|\mathcal{B}d(\mathcal{T}(\mathcal{P}, \mathbf{r}, q))|/2$.

4.5 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 4.6 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 strongest 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 weakest 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 4.11 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. □

Definition 4.12 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 $\mathcal{T}r(\mathcal{H})$. □

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 4.13 Let $\mathcal{P}$ be a set of patterns, $\preceq$ a strength hierarchy, 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 $f(\mathcal{S}) = \{f(X) \mid X \in \mathcal{S}\}$. □

Example 4.14 Consider functional dependencies $X \to A$ with a fixed right-hand side A. Recall from Theorem 3.19 that a suitable strength hierarchy $\preceq$ is the converse of set inclusion: if $X \to A$ is true, then $Y \to A$ is true for all supersets Y of X, and $(Y \to A) \preceq (X \to A)$.

Functional dependencies with a fixed right-hand side A have a set representation f where $f(X \to A)$ is the complement of X: $f(X \to A) = R\setminus X$. This representation is bijective, the representation and its inverse are computable, and $(Y \to A) \preceq (X \to A)$ if and only if $(R \setminus Y) \subseteq (R \setminus X)$. □

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

Definition 4.15 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., $\mathcal{H}(\mathcal{S}) = \{R \setminus f(\varphi) \mid \varphi \in \mathcal{B}d^+(\mathcal{S})\}$. □

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 4.16 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.

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

□

Example 4.17 Recall Example 4.10. The example deals with the keys in a given relation r: the set

$$\mathcal{B}d^+(\text{KEYS}(r)) = \{\{A, B\}, \{A, C\}\}$$

of minimal keys is given, and the task is to verify that this really is the positive border. For this we need to determine the negative border $\mathcal{B}d^-(\text{KEYS}(r))$, and we compute it now using the hypergraph formulation.

As with functional dependencies, the set representation of a key X is its complement, $f(X) = R \setminus X$. Hence the edges of the hypergraph $\mathcal{H}(\text{KEYS}(r))$ are complements of complements, i.e., the minimal keys themselves:

$$\mathcal{H}(\text{KEYS}(r)) = \{R \setminus f(X) \mid X \in \mathcal{B}d^+(\text{KEYS}(r))\} = \mathcal{B}d^+(\text{KEYS}(r)).$$

There are two minimal sets, $\{A\}$ and $\{B, C\}$, that intersect with both minimal keys $\{A, B\}$ and $\{A, C\}$, i.e., $\mathcal{T}r(\mathcal{H}(\text{KEYS}(r))) = \{\{A\}, \{B, C\}\}$. These minimal transversals are the set representation of the negative border, so we have $f^{-1}(\mathcal{T}r(\mathcal{H}(\text{KEYS}(r)))) = \{\{B, C, D\}, \{A, D\}\} = \mathcal{B}d^-(\text{KEYS}(r))$. □

We showed in Section 4.3 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., [Ber73]) can be used, e.g., in the design of algorithms or complexity analysis for specific knowledge discovery problems. The complexity of computing the minimal transversals of a hypergraph has long been open; 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 [FK94, Kha95].

4.6 Related work

Many of the concepts and ideas of this and the previous chapter are known in different contexts. The contributions of these chapters are in providing a unified viewpoint to several knowledge discovery tasks and a generic algorithm for those tasks, and in the analysis of these problems. For instance, Algorithm 3.7 is based on the breadth-first search strategy, and it uses a strength hierarchy for safely pruning branches of the search tree, both well known search methods. The idea of checking *all* weaker patterns when generating candidates has, however, been missed, e.g., in the original algorithm for discovering frequent sets [AIS93] and in the inference of inclusion dependencies [KMRS92, MR92a]. In the area of machine learning, the version spaces of Mitchell [Mit82] are the first systematic use of strength hierarchies 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 [Sie95].

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

Explora [Klö95] is a system for discovering patterns describing outstanding subgroups of a given database. Explora employs strength hierarchies: it organizes patterns into hierarchies and lets the user specify which are strength hierarchies with respect to the domain and the user's interests. The algorithm repeatedly evaluates patterns along paths between the weakest and the strongest patterns until the border is located. The strength hierarchies are used to prune from evaluation those patterns for which the truth of selection criterion is already known.

Claudien [DB93] 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 using the same conversed strategy that was described in Example 3.20 for functional dependencies. 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 strength hierarchy: it may generate and test candidate clauses that cannot hold given all weaker 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 [JCH95], 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.

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 [AMS+96, MTV94a]. A thorough analysis of the problem of discovering functional dependencies is given in [MR92a, MR92b]. For various algorithms for finding functional dependencies, see [Bel95, MR92a, MR92b, MR94, PK95]. The relevance of transversals to computing the theory of a model has been known in the context of finding functional dependencies [MR94] and several other specific problems [EG95].

Chapter 5

Sampling large databases for frequent sets

If enough data is collected, anything may be proven by statistical methods.
William's and Holland's Law

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 5.1. In Section 5.2 we analyze the goodness of the sampling method, e.g., the relation of sample size to the accuracy of results. In Section 5.3 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 [Toi96].

5.1 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 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 5.1 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.

Algorithm 5.1
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:

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

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 $\mathcal{F}(r, \mathit{min\_fr})$ using the sample and then to evaluate $\mathcal{S}$ in r. Rather, the collection $\mathcal{F}(r, \mathit{min\_fr}) \cup \mathcal{B}d^-(\mathcal{F}(r, \mathit{min\_fr}))$ 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 $\mathcal{F}(r, \mathit{min\_fr})$, so we check the union. Sometimes, however, it happens that we find out that not all necessary sets have been evaluated.

Definition 5.2 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 $\mathcal{F}(r, \mathit{min\_fr})$ that is not in $\mathcal{S} \cup \mathcal{B}d^-(\mathcal{S})$. A *miss* is a frequent set Y in $\mathcal{F}(r, \mathit{min\_fr})$ 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 5.3 Algorithm 5.1 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 5.4 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 5.1 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 5.1 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 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 5.1 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 $\mathcal{F}(r, \mathit{min\_fr})$. 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 5.5 can be used to extend Algorithm 5.1 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 5.1, and $\mathcal{S}^-$ is the collection of non-frequent candidates of Algorithm 5.1. The collection $\mathcal{B}d^-(\mathcal{S})$ can be located in a similar way as candidates are generated.

Algorithm 5.5
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 $fr(X, r)$;

Theorem 5.6 Algorithm 5.5 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.

5.2 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 [AMS+96, MTV94a]. Related work on using a sample for approximately verifying the truth of arbitrary sentences of relational tuple calculus is considered in [KM94].

Definition 5.7 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 the sample, given an error bound ε and a maximum probability δ for an error that exceeds the bound.

Theorem 5.8 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 $B(n, p)$ [AS92]. For the probability at hand we thus have

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

$\square$

Table 5.1 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.

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 5.1: Sufficient sample sizes, given ε and δ.

Corollary 5.9 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 5.8, 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. $\square$

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 5.10 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 $e^{-|s|\, min\_fr} \leq \delta$. $\square$

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 $\mathcal{S} = \mathcal{F}(s, \mathit{low\_fr})$ 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 5.8 gives the following result on how to set the lowered frequency threshold so that misses are avoided with a high probability.

Theorem 5.11 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

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

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

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

□

Consider now the number of sets checked in the second pass by Algorithm 5.5, 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

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

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.

5.3 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 [AS94] in our tests. These databases model supermarket basket data, and they have been used as benchmarks for several association rule algorithms [AMS+96, AS94, HKMT95, PCY95, SON95]. 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 5.2 summarizes the parameters for the data sets; see [AS94] 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 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.

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 5.3: Lowered frequency thresholds for $\delta = 0.001$.

Number of misses and database activity We experimented with Algorithm 5.1 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 5.3; in the computations of the lowered thresholds we used the exact probabilities obtained from the binomial distribution, not the Chernoff bounds.

Figure 5.1 shows the number of database passes for the three different types of algorithms: Algorithm 2.14, Partition, and the sampling Algorithm 5.1. The Partition algorithm [SON95] 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 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 5.1 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 5.1, 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 5.1, and the number of passes

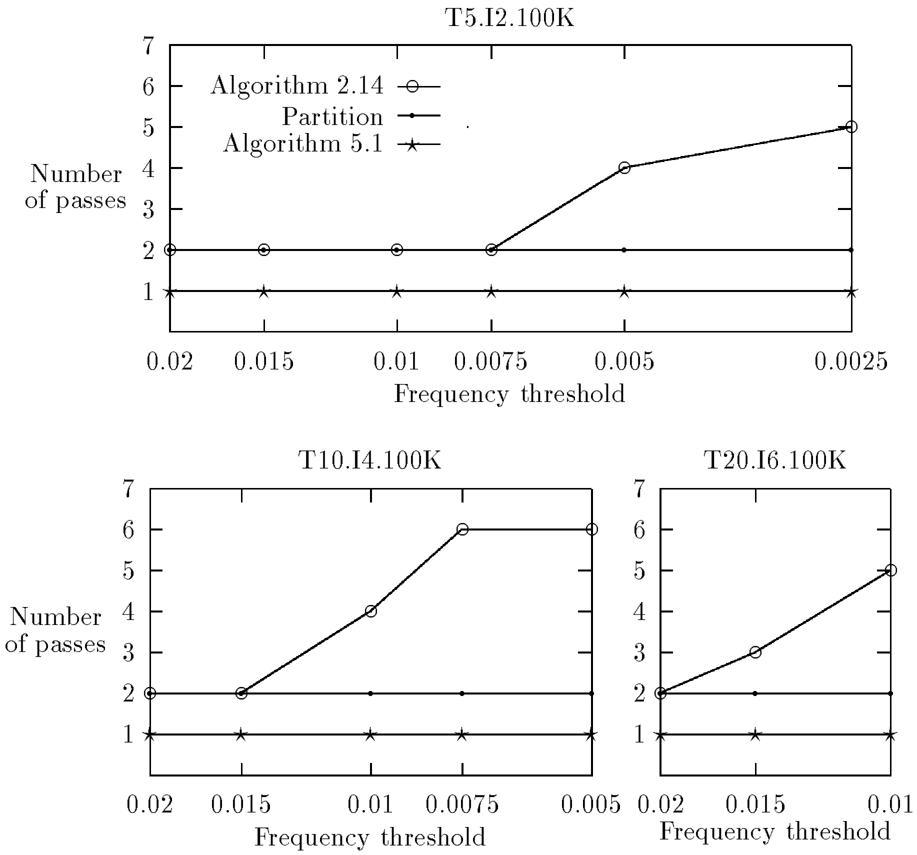


Figure 5.1: The number of database passes made by frequent set algorithms.

of Algorithm 2.14 is up to six times that of Algorithm 5.1.

Table 5.4 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

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 5.4: Number of trials with misses.

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, [OR89]. The related problem of sampling for query estimation is considered in [HS92].

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 5.5 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 over-

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

head is larger with higher thresholds, but since the absolute overheads are small the effect is negligible. Table 5.5 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 5.2 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 5.5 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.

Approximate $1 - \Delta$ success probability Setting the lowered threshold for Algorithm 5.1 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

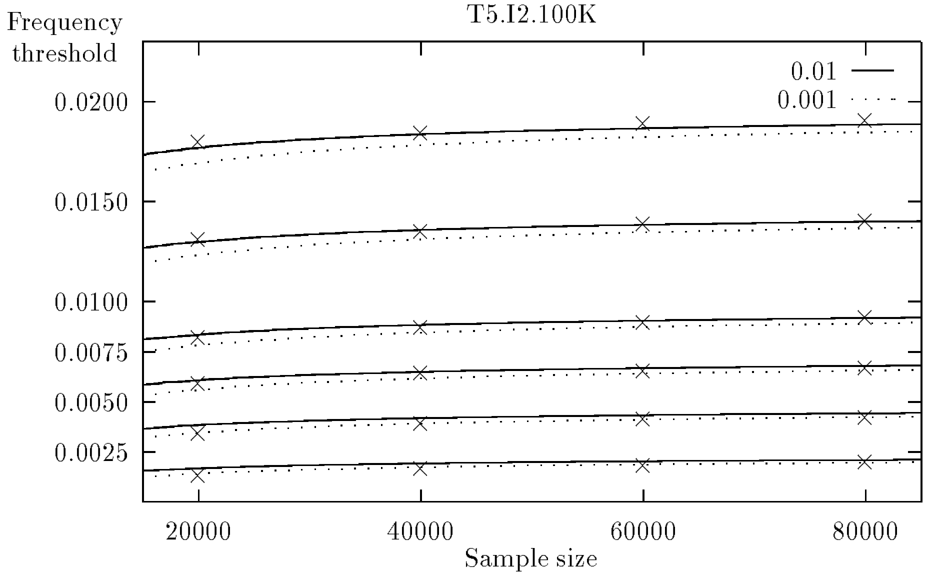


Figure 5.2: Frequency thresholds giving misses in 0.01 cases (points) and lowered thresholds with $\delta = 0.01$ and 0.001 (lines).

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 5.12 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.

Algorithm 5.12

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;

Theorem 5.13 Algorithm 5.12 works correctly.

Proof See the proof of Theorem 5.3 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. □

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 5.14 is a simple variation of Algorithm 2.14. It utilizes also the failure probability approximation described in Section 5.2: 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.

Algorithm 5.14

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 $\mathcal{F}(r, \mathit{min\_fr})$ 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:

```
compute $s$ := random sample of size $s\_size$ from $r$;
$P := 0$;
$low\_fr := min\_fr$;
// Find frequent sets in the sample:
$\mathcal{C}_1 := \{\{A\} \mid A \in R\}$;
$l := 1$;
while $\mathcal{C}_l \neq \emptyset$ do
    $\mathcal{R}_l := \emptyset$;
    for all $X \in \mathcal{C}_l$ do
        if $fr(X, s) < low\_fr$ then
            $p := Pr[X$ is frequent in $r]$;
            if $p/(\Delta - P) > \gamma$ then $low\_fr := fr(X, s)$
            else $P := 1 - (1 - P)(1 - p)$;
        if $fr(X, s) \geq low\_fr$ then $\mathcal{R}_l := \mathcal{R}_l \cup \{X\}$;
    $l := l + 1$;
    compute $\mathcal{C}_l := \mathcal{C}(\mathcal{R}_{l-1})$;
// Database pass:
compute $\mathcal{R} := \{X \in \bigcup_l \mathcal{C}_l \mid fr(X, r) \geq min\_fr\}$;
for all $X \in \mathcal{R}$ do output $X$ and $fr(X, r)$;
if $\mathcal{R} \cap (\bigcup_l (\mathcal{C}_l \setminus \mathcal{R}_l)) \neq \emptyset$ then report that there possibly was a failure;
```

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 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 5.2 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 5.15 Algorithm 5.14 works correctly.

Proof See the proof of Theorem 5.3 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

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

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 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 5.14 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 5.6. The number successfully remained below $100\Delta = 10$ in each set of experiments. As Table 5.6 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 shortly, 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.

Chapter 6

Discovery of Boolean rules using frequent sets

Eight out of ten people write with a ballpoint pen.
So what do the rest use the pen for?

The class of association rules is rather simple: only positive connections between sets of items can be expressed. In this chapter we consider a more powerful rule formalism, the class of Boolean rules. Informally, Boolean rules are association rules with negation and disjunction; we define them formally in Section 6.1. We also note that the task of discovering *all* Boolean rules that hold in a database is not very useful, nor very feasible. We list useful types of Boolean rules in Section 6.2, and show how their confidences can be computed from the frequencies of sets. We then show that the principle of inclusion and exclusion can be used to compute the frequency of any Boolean formula from the frequencies of sets. As it turns out, good approximations of frequencies and confidences of Boolean rules can be computed from the frequent sets alone. Experimental results on the accuracy of the estimated frequencies are given in Section 6.3. Parts of this chapter have been published in [MT96b].

6.1 Boolean formulas and rules

We generalize the notions of *set* and *association rule* to *Boolean formula* and *Boolean rule* as follows.

Definition 6.1 Let R be a set and r a binary database over R. A *literal* is an expression A or $\overline{A}$, where $A \in R$. A literal A is said to be *positive* and a literal $\overline{A}$ *negative*. Next we define recursively *Boolean formulas* and their truth.

A literal is a Boolean formula. A positive literal A is true on a row t in r if and only if $A \in t$. A negative literal $\overline{A}$ is true on a row t if and only if the literal A is not true on the row. Given a literal ψ, the set of rows in r on which ψ is true is denoted by $\mathcal{M}(\psi, r) = \{t \in r \mid \psi \text{ is true on } t\}$.

Let $\psi_1, \ldots, \psi_k$ be Boolean formulas. Then

- $(\psi_1 \wedge \ldots \wedge \psi_k)$ is a Boolean formula which is true on the rows in the set $\mathcal{M}(\psi_1 \wedge \ldots \wedge \psi_k, r) = \bigcap_{i=1,\ldots,k} \mathcal{M}(\psi_i, r)$. We use the following notational convention: a conjunction $(\psi_1 \wedge \ldots \wedge \psi_k)$ can also be written as the set $\{\psi_1, \ldots, \psi_k\}$.
- $(\psi_1 \vee \ldots \vee \psi_k)$ is a Boolean formula which is true on the rows in the set $\mathcal{M}(\psi_1 \vee \ldots \vee \psi_k, r) = \bigcup_{i=1,\ldots,k} \mathcal{M}(\psi_i, r)$.
- $\overline{\psi}$ is a Boolean formula which is true on the rows in the set $\mathcal{M}(\overline{\psi}, r) = r \setminus \mathcal{M}(\psi, r)$.

For notational convenience, we also consider the empty set $\emptyset$ to be a Boolean formula, namely the empty conjunction. It is true on all rows of r, i.e., $\mathcal{M}(\emptyset, r) = r$. □

Definition 6.2 Given a binary database r over a set R, a *Boolean rule* over R is an expression of the form $\psi \Rightarrow \omega$, where ψ and ω are Boolean formulas. The *frequency* of a Boolean formula ψ in r is

$$fr(\psi, r) = \frac{|\mathcal{M}(\psi, r)|}{|r|}.$$

The frequency of a Boolean rule $\psi \Rightarrow \omega$, denoted $fr(\psi \Rightarrow \omega, r)$, is $fr(\psi \wedge \omega, r)$. The *confidence* of the Boolean rule is

$$conf(\psi \Rightarrow \omega, r) = \frac{|\mathcal{M}(\psi \wedge \omega, r)|}{|\mathcal{M}(\psi, r)|}.$$

□

Item sets $X \subseteq R$ can be considered, according to these definitions and writing conventions, to be conjunctions of positive literals. This interpretation is consistent with the earlier definitions concerning item sets. We use the set notation in this chapter to emphasize that item sets are a subclass of Boolean formulas and association rules are a subclass of Boolean rules.

Next we consider the complexity of discovering Boolean rules that hold in a binary database. For simplicity, we look first at association rules with negation. Such rules could in principle be discovered with association rule algorithms by using a simple trick. Namely, given a binary database r over a set R, Boolean rules consisting only of literals and the conjunctive connective $\wedge$ can be found by association rule discovery algorithms as follows. Introduce $|R|$ new complement items $\overline{A_i}$, one for each item A_i in R, such that for all t in r, $\overline{A_i}$ is in t if and only if A_i is not in t. Then discover standard association rules using the extended set of items and the extended binary database.

If complement items are introduced this way, the size of the search space of frequent sets grows from $2^{|R|}$ to $3^{|R|}$. In addition to the growth of the search space, a direct application of the definition of candidate collection gives also candidates of the form $\{I_i, \overline{I_i}\}$; such impossible candidates are, however, only formed for size 2. The candidate generation can easily be modified to exclude such pairs from consideration. The discovery of such rules with negation is straightforward. It is also efficient in the sense that only frequent formulas and formulas in the negative border are evaluated. The problem is that the number of frequent formulas, i.e., the size of the output, explodes. The following rough analysis demonstrates the growth of the size of the output when negation is allowed in association rules.

Example 6.3 Assume all positive literals have a frequency of at most b. Consider now a Boolean formula $\overline{A_1} \wedge \ldots \wedge \overline{A_k}$ of k negative literals. Its frequency is at least $1-kb$; that is, the formula is frequent if $k \leq (1-\mathit{min\_fr})/b$. For instance, with $|R| = 100$ items, with maximum item frequency $b = 0.1$, and with frequency threshold $\mathit{min\_fr} = 0.05$, we have $k \leq 9.5$, meaning that *all* conjunctions of negative literals up to size 9 are frequent. There are thus at least $\binom{100}{9} > 10^{12}$ frequent formulas. □

Example 6.4 Consider now the class of Boolean formulas of the form $\overline{A_1 \wedge \ldots \wedge A_k}$. It turns out, with reasonable assumptions, that all formulas in the class are frequent. Assume that the frequency threshold $\mathit{min\_fr}$ is less than 0.5, and that $\mathit{fr}(X)$ is less than 0.5 for all $X \subseteq R$ (except the empty set). Consider now the subclass of Boolean formulas of the form $\overline{X}$, where $X \subseteq R$. Since we have $\mathit{fr}(X) < 0.5$ for every X, every $\overline{X}$ is frequent. There are $2^{|R|} - 1$ such frequent formulas. □

The task of discovering arbitrary Boolean rules is not very useful: the number of frequent Boolean rules can be enormous. Also, arbitrary rule forms are not likely to be valuable for applications. A more useful approach is that the user queries the system for specific rules or classes of rules.

6.2 Computation of frequencies

The frequencies of sets can be used to compute the frequency of any Boolean formula, and thus also the confidence of any Boolean rule. In particular, the frequencies can be computed using the principle of inclusion and exclusion on disjunctive normal forms of formulas. We now show how this is possible.

Definition 6.5 A boolean formula ψ is in *disjunctive normal form* if it is of the form $\psi = (\bigvee_{i=1,\ldots,k} \psi_i)$, where each ψ_i is, in turn, of the form $\psi_i = (\bigwedge_{j=1,\ldots,l_i} \psi_{i_j})$, where each ψ_{i_j} is a positive or negative literal. □

Proposition 6.6 For any Boolean formula ψ there is a formula ω in disjunctive normal form such that ψ is true if and only if ω is true.

Proof Denote by $R = \{A_1, \ldots, A_{|R|}\}$ the set of positive literals. A trivial disjunctive normal form for any Boolean formula ψ can be obtained by taking as terms of the disjunction all those conjunctions $\psi_i = (\bigwedge_{j=1,\ldots,|R|} \psi_{i_j})$ for which ψ is true, and where each ψ_{i_j} is either A_i or $\overline{A_i}$. □

The frequency of a formula ψ in disjunctive normal form can be computed with the principle of inclusion and exclusion. By Definition 6.1 the set $\mathcal{M}(\psi)$ of rows on which ψ is true is the union of the sets $\mathcal{M}(\psi_i)$; inclusion and exclusion is the standard principle for computing the size of a union from the sizes of different intersections.

Proposition 6.7 The frequency of a Boolean formula $\bigvee_{i=1,\ldots,k} \psi_i$ in disjunctive normal form is $fr(\bigvee_{i=1,\ldots,k} \psi_i) = 1 - \sum_{S\subseteq\{1,\ldots,k\}} (-1)^{|S|} fr(\bigwedge_{i\in S} \psi_i)$, i.e., a sum of frequencies of conjunctions of literals. (Remember that the empty Boolean formula $\emptyset$ is true on all rows, i.e, $fr(\emptyset) = 1$.)

Proof The proposition is a straightforward application of the principle of inclusion and exclusion, see, e.g., [PTW83]. □

Example 6.8 Consider the formula $\psi = ((A \wedge \overline{B}) \vee C \vee \overline{D})$ which is in disjunctive normal form. With the principle of inclusion and exclusion we can compute the frequency of the formula as follows:

$$\begin{aligned} fr(\psi) &= fr(\{A, \overline{B}\}) + fr(\{C\}) + fr(\{\overline{D}\}) \\ &\quad - fr(\{A, \overline{B}, C\}) - fr(\{A, \overline{B}, \overline{D}\}) - fr(\{C, \overline{D}\}) \\ &\quad + fr(\{A, \overline{B}, C, \overline{D}\}). \end{aligned}$$

□

The terms of the sum of Proposition 6.7 are frequencies of conjunctions of positive and negative literals. The principle of inclusion and exclusion can be applied also here, to express such a frequency as a sum of frequencies of conjunctions of positive literals, i.e., frequencies of item sets.

Proposition 6.9 The frequency of a conjunction $\{A_1, \ldots, A_l, \overline{B_1}, \ldots, \overline{B_k}\}$ of literals is

$$fr(\{A_1, \ldots, A_l, \overline{B_1}, \ldots, \overline{B_k}\}) = \sum_{S \subseteq \{1,\ldots,k\}} (-1)^{|S|} fr(\{A_1, \ldots, A_l\} \cup \{B_i \mid i \in S\}).$$

Proof Again, we apply the principle of inclusion and exclusion. Note that the conjunction $\{A_1, \ldots, A_l, \overline{B_1}, \ldots, \overline{B_k}\}$ is logically equivalent to $(A_1 \wedge \ldots \wedge A_l \wedge \overline{(B_1 \vee \ldots \vee B_k)})$. The frequency of the negated disjunction is one minus the frequency of the disjunction, and can thus be computed as in Proposition 6.7. Here we have to additionally restrict the disjunction to where $\{A_1, \ldots, A_l\}$ is true. □

Example 6.10 Consider the last term in the sum of Example 6.8, the frequency of $\{A, \overline{B}, C, \overline{D}\}$. By Proposition 6.9 we have

$$\begin{aligned} fr(\{A, \overline{B}, C, \overline{D}\}) \quad = \quad & fr(\{A, C\}) \\ & - fr(\{A, B, C\}) - fr(\{A, C, D\}) \\ & + fr(\{A, B, C, D\}). \end{aligned}$$

After we apply Proposition 6.9 on each of the terms in Example 6.8 and remove terms that sum to zero, we have

$$\begin{aligned} fr(\psi) \quad = \quad & 1 - fr(\{D\}) + fr(\{A, D\}) + fr(\{C, D\}) \\ & - fr(\{A, B, D\}) - fr(\{A, C, D\}) + fr(\{A, B, C, D\}). \end{aligned}$$

□

There are two problems with this approach of computing the frequency of a Boolean formula from the frequencies of sets. First, the size of the disjunctive normal form can be exponential with respect to the size of the original formula. For practical purposes the number of terms is not a problem: for most interesting cases few terms are needed, as is shown in examples below. The second and more difficult problem in computing frequencies in this way is that the frequencies are in practice only known down to a frequency threshold *min_fr*. We return to this problem briefly.

We now give examples of useful subclasses of Boolean rules. In the spirit of Definition 6.1, where the truth of all Boolean formulas was defined in terms of the truth of positive literals, we also show how the confidences can be computed from the frequencies of item sets.

Example 6.11 An *association rule* has the form $X \Rightarrow Y$, where $X, Y \subseteq R$, and the confidence is defined to be

$$\frac{fr(X \cup Y)}{fr(X)}.$$

□

Example 6.12 A rule with a *disjunctive right-hand side* has the form $X \Rightarrow Y \vee Z$ and expresses that if a row $t \in r$ contains X, then it contains Y or Z. The confidence of the rule is

$$\frac{fr(X \cup Y) + fr(X \cup Z) - fr(X \cup Y \cup Z)}{fr(X)}.$$

□

Example 6.13 Similarly, we can have rules with *disjunctive left-hand sides.* A rule of the form $X \vee Y \Rightarrow Z$ has the meaning: if a row $t \in r$ contains X or Y, then it also contains Z. The confidence of the rule is

$$\frac{fr(X \cup Z) + fr(Y \cup Z) - fr(X \cup Y \cup Z)}{fr(X) + fr(Y) - fr(X \cup Y)}.$$

□

Example 6.14 Rules can have *negation in the left-hand side.* Such a rule $X \wedge \overline{Y} \Rightarrow Z$ means that if a row $t \in r$ contains X but does not contain the whole Y, then it contains Z. The confidence of such a rule is

$$\frac{fr(X \cup Z) - fr(X \cup Y \cup Z)}{fr(X) - fr(X \cup Y)}.$$

□

Example 6.15 Rules with *negation in the right-hand side* have the form $X \Rightarrow Y \wedge \overline{Z}$ and mean that if a row $t \in r$ contains X, then it contains Y but does not contain all of Z. The confidence of the rule is

$$\frac{fr(X \cup Y) - fr(X \cup Y \cup Z)}{fr(X)}.$$

□

The frequency of a Boolean formula can be approximated in the following way: express the frequency of the formula as a sum of frequencies of sets, using, e.g., the principle of inclusion and exclusion, and then simply ignore the frequencies that are lower than a frequency threshold *min_fr*. Next we consider the error that results from ignoring some terms of the sum. First we define a shorthand notation for the sum obtained from the principle of inclusion and exclusion.

Definition 6.16 The *inclusion-exclusion frequency* of a set X of items is

$$ie(X) = 1 - \sum_{Y \subseteq X} (-1)^{|Y|} fr(Y) = \sum_{Y \subseteq X, Y \neq \emptyset} (-1)^{|Y|+1} fr(Y).$$

□

The frequencies of several interesting cases of Boolean formulas can be computed directly from the inclusion-exclusion frequencies. Note, in particular, the following three basic cases that hold for all sets $X = \{A_1, \ldots, A_k\}$ of items. The principle of inclusion and exclusion gives directly $fr(A_1 \vee \ldots \vee A_k) = ie(X)$ and $fr(\overline{A_1 \vee \ldots \vee A_k}) = fr(\overline{A_1} \wedge \ldots \wedge \overline{A_k}) = 1 - ie(X)$. Additionally, the frequency $fr(X)$ can be used directly to compute $fr(\overline{A_1 \wedge \ldots \wedge A_k}) = fr(\overline{A_1} \vee \ldots \vee \overline{A_k}) = 1 - fr(X)$.

Consider now approximating the inclusion-exclusion frequency. Given all frequent sets we do not know all terms in the sum. What we know are the terms greater than or equal to the frequency threshold.

Definition 6.17 Given a set X of items and the collection $\mathcal{F}(r)$ of frequent sets and their frequencies, the *truncated inclusion-exclusion frequency* $\widehat{ie}(X)$ is

$$\widehat{ie}(X) = \sum_{Y \subseteq X, \emptyset \neq Y \in \mathcal{F}(r)} (-1)^{|Y|+1} fr(Y).$$

The absolute error $err(X)$ of the truncation is $err(X) = |ie(X) - \widehat{ie}(X)|$. □

In Open Problem 6.18, below, we look for bounds for $err(X)$. The idea is to make use of the negative border $\mathcal{B}d^-(\mathcal{F}(r))$ that has been evaluated during the computation of $\mathcal{F}(r)$. Namely, the subsets of X that are in the negative border specify the sets that are ignored, and the frequencies of the border sets set constraints on the ignored frequencies.

Open Problem 6.18 Given a collection $\mathcal{B}d^-(\mathcal{F}(r))$ of sets in the negative border, frequencies of the sets, and an item set X, denote

$$bds(X) = \sum_{\substack{Y \subseteq X \text{ and} \\ Y \in \mathcal{B}d^-(\mathcal{F}(r))}} fr(Y).$$

What is the smallest positive constant C such that we have either $err(X) = 0$, if all subsets of X are in $\mathcal{F}(r)$, or $err(X) \leq C \cdot bds(X)$ otherwise? $\square$

We believe that a good bound can be computed from the sum of the frequencies of those ignored sets that are in the negative border. How to find a good value for C is, however, an open problem; several combinatorial aspects of the ignored sets could be used to determine the value for each case individually. It is obvious that the constant value $C = 1$ is correct for those simple cases where the ignored sets Y have size $|Y| \geq |X| - 1$. A counter-example for larger sets shows, however, that $C = 1$ is not correct in general. On the other hand, it is easy to see that $C = 2^s$, where $s = |\{Y \subseteq X \mid Y \in \mathcal{B}d^-(\mathcal{F}(r))\}|$, is sufficient.

Approximation of the inclusion-exclusion sum has been considered also in [KLS95, LN90], where it is shown that knowing the terms $(-1)^{|Y|} fr(Y)$ of the inclusion-exclusion formula for all Y with $|Y| \leq \sqrt{|X|}$ is sufficient for getting good approximations to the whole sum. Their results are not directly applicable to our problem, as the frequent sets do not necessarily provide the required subset of the terms of the inclusion-exclusion formula. Interestingly, the approximation is not formed by simply computing the truncated sum; rather, one uses a linear combination of the terms of the truncated sum.

6.3 Experiments

We experimented with the course enrollment data described in Section 2.4. After discovering frequent sets—and the negative border—with frequency threshold 0.05 we have, for instance, the following results for the frequency of disjunction of items.

- For the set $X_1 = \{$*Data Structures*, *Database Systems I*$\}$, all three terms in the inclusion-exclusion frequency are known and we have $fr(\textit{Data Structures} \vee \textit{Database Systems I}) = ie(X_1) = \widehat{ie}(X_1) = 0.246$ and $err(X_1) = 0$.

- For a set X_2 of three 'C' or Unix-related courses, 4 out of 7 terms are greater than the threshold and 2 are in the negative border. For the frequency we have $ie(X_2) = 0.227$, for the estimated frequency $\widehat{ie}(X_2) = 0.240$, and for the error $err(X_2) = 0.012$. The sum of the ignored frequencies of the border is $bds(X_2) = 0.017$.

- For a set X_3 of three different intermediate courses, *Theory of Computation*, *Artificial Intelligence*, and *Computer Graphics*, we have 3 out of 7 terms in the truncated inclusion-exclusion frequency, and another

Size of sets X	Average error $err(X)$	Average error $1 \cdot bds(X)$	Maximum error $err(X)$	Maximum error $1 \cdot bds(X)$	Relative error (%) $err(X)$	Relative error (%) $1 \cdot bds(X)$
2	0.017	0.017	0.048	0.048	6.8	6.8
3	0.036	0.048	0.090	0.105	10.3	13.7
4	0.052	0.104	0.145	0.178	12.4	24.4
5	0.063	0.182	0.166	0.278	12.5	37.3
6	0.075	0.300	0.212	0.560	14.3	57.0
7	0.086	0.459	0.202	0.845	14.0	77.4
8	0.105	0.667	0.273	1.041	15.3	100.8

Table 6.1: Errors for different sizes of random sets using frequency threshold 0.05.

3 are in the negative border. We have $ie(X_3) = 0.147$, $\widehat{ie}(X_3) = 0.212$, $err(X_3) = 0.065$, and $bds(X_3) = 0.079$.

- For a set X_4 consisting of six programming projects we have $ie(X_4) = 0.322$, $\widehat{ie}(X_4) = 0.314$, $err(X_4) = 0.009$, and $bds(X_4) = 0.012$. In this case only 7 out of 63 terms exceed the threshold, and another 3 are in the negative border.

The frequency threshold above is high if compared, for instance, to the supermarket domain, where reasonable frequency thresholds are assumed to be fractions of a per cent. By lowering the threshold to 0.005 we get the exact inclusion-exclusion frequency 0.147 for the set $X_3 = \{$*Theory of Computation*, *Artificial Intelligence*, *Computer Graphics*$\}$ and a more accurate estimate $\widehat{ie}(X_2) = 0.222$ for the set X_2 of three 'C' or Unix-related courses. To evaluate the accuracy of the truncated inclusion-exclusion and the sum in Open Problem 6.18 we generated random sets X and observed the error, in particular in relation to the computed estimation $\widehat{ie}(X)$. We used two frequency thresholds, 0.05 and 0.005, to see how strong the effect of the frequency threshold is on the error. First we computed the frequent sets and the negative border for these two test cases. We then generated random sets X such that each item in X is frequent. This decision is motivated by the assumption that the most interesting Boolean formulas consist of items that are frequent themselves, and by the fact that non-frequent items do not contribute any terms to the truncated inclusion-exclusion frequency. For each test case and sizes 2 to 8, we generated 100 random sets. Finally, we computed the errors and the sums $bds(X)$.

Results on the errors with frequency threshold 0.05 are presented in Table 6.1. We see first of all that the error is, on average, below 0.1 for all

Size of sets X	Average error $err(X)$	Average error $1 \cdot bds(X)$	Maximum error $err(X)$	Maximum error $1 \cdot bds(X)$	Relative error (%) $err(X)$	Relative error (%) $1 \cdot bds(X)$
2	0.001	0.001	0.004	0.004	1.6	1.6
3	0.003	0.003	0.007	0.009	2.9	3.1
4	0.004	0.006	0.012	0.014	3.3	4.3
5	0.006	0.011	0.017	0.025	3.8	6.2
6	0.008	0.019	0.020	0.037	3.8	7.8
7	0.010	0.027	0.026	0.048	3.9	9.5
8	0.009	0.038	0.026	0.068	3.2	12.1

Table 6.2: Errors for different sizes of random sets using frequency threshold 0.005.

sizes but 8. The sums $bds(X)$ are reasonably close to $err(X)$ for sizes 2 to 4. The maximum errors of the truncated sums range from 0.048 to 0.273; again, $bds(X)$ is reasonable for sizes 2 to 4. Finally, in the last rows on the right, we see that the relative error, i.e., the ratio of the error to the estimated inclusion-exclusion frequency, is on average below 16 % for all sizes.

A comparison to the results with $min\_fr = 0.005$, in Table 6.2, shows that when the frequency threshold is smaller the errors are smaller, especially for small sets. The difference is best visible in the average relative errors: they are below 4 % for all the experiments. In summary, frequencies of small Boolean formulas can be computed accurately from frequent sets. In both experiments, the average error is of the same order of magnitude as the frequency threshold, and the relative error becomes notably better with a lower threshold.

Chapter 7

Discovery of episodes in sequences

How many times do I have to tell you not to repeat yourself?
– A desperate PhD advisor

In several application domains one can collect sequences of events describing the behavior of users or systems. In this chapter we consider the problem of recognizing frequent episodes in such sequences of events. An episode is defined to be a partially ordered collection of events that occur within a time interval of a given length. Once the frequent episodes are known, one can produce rules for describing or predicting the behavior of the sequence.

We describe episodes informally in Section 7.1 and give the exact definitions in Section 7.2, where we also outline an algorithm for the discovery of all frequent episodes. In Sections 7.3 and 7.4 we give algorithms for the candidate generation and the database pass, respectively, for the most important types of episodes. In Section 7.5 we shortly discuss other types of episodes. Experimental results are presented in Section 7.6. Finally, we suggest extensions and review related work in Section 7.7. This chapter is mostly based on [MTV95], but the first formulations and results on the discovery of episodes were presented already in [MTV94b].

7.1 Event sequences and episodes

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. For instance, in telecommunication network monitoring or in empirical user interface studies it is easy to log a lot of information about the

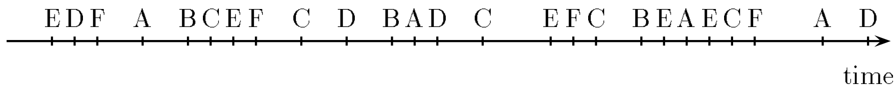


Figure 7.1: A sequence of events.

behavior and actions of the user and the system. Abstractly, such a log can be viewed as a sequence of events, where each event has an associated time of occurrence. An example of an event sequence is represented in Figure 7.1. Here A, B, C, D, E, and F are event types, e.g., different types of alarms from a telecommunication network, or different types of user actions, and they have been marked on a time line.

One basic problem in analyzing such a sequence is to find frequent *episodes*, i.e., collections of events occurring frequently together. For example, in the sequence of Figure 7.1, the episode "E is followed by F" occurs several times, even when the sequence is viewed through a narrow window. Episodes, in general, are partially ordered sets of events. From the sequence in the figure one can make, for instance, the observation that whenever A and B occur (in either order), C occurs soon.

Episodes can be described as directed acyclic graphs in the obvious way. Consider, for instance, episodes α, β, and γ in Figure 7.2. Episode α is a *serial episode*: it occurs in a sequence only if there are events of types E and F that occur in this order in the sequence. In the sequence there can be other events occurring between these two. Episode β is a *parallel episode*: no constraints on the relative order of A and B are given. In this thesis we only consider the discovery of serial and parallel episodes. The definitions we give apply, however, in more general. 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.

When discovering episodes in a telecommunication network alarm log, the goal is to find unknown relationships between alarms. Such relationships can then be used in an on-line analysis of the incoming alarm stream, e.g, to better explain the problems that cause alarms, to suppress redundant alarms, and to predict severe faults. The alarm sequence is merged from several sources, and therefore it is useful that episodes are insensitive to intervening events.

In the analysis of sequences we are interested in finding all frequent episodes from a class of episodes. Such a class can be, e.g., all serial episodes, all parallel episodes, or all episodes where the partial order matches some part of the network topology; one can even consider the class of all partial

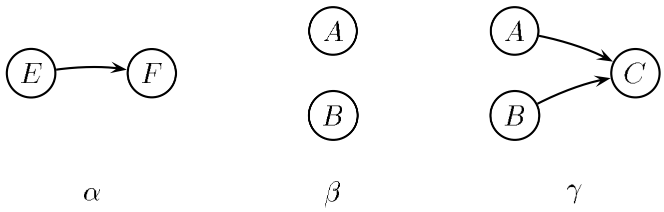


Figure 7.2: Episodes.

orders. To be considered interesting, the events of an episode must occur close enough in time. The user defines how close is close enough by giving the width of the *time window* within which the episode must occur. The user also specifies in how many windows an episode has to occur to be considered frequent.

Once the frequent episodes are known, they can be used to obtain rules for prediction. For example, if we know that the episode β of Figure 7.2 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.

In summary, we consider the following problem. 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 event sequence. The algorithm we give for solving this task is an instance of the generic Algorithm 3.7 for discovering all frequent patterns. The candidate generation is very similar to generating candidates for frequent sets. In the database pass we recognize episodes efficiently by "sliding" a window on the input sequence. Two adjacent windows have a lot of overlap and are therefore similar to each other. We take advantage of this similarity: after recognizing episodes in a window, we make incremental updates in our data structures to determine which episodes occur in the next window.

7.2 Definitions

Let us now return to the basic concepts and define them more formally. We start by giving definitions for event sequences and windows.

Definition 7.1 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.

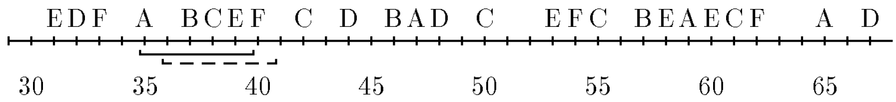


Figure 7.3: The example event sequence $\mathbf{s}$ and two windows of width 5.

An *event sequence* $\mathbf{s}$ on R is a triple $(T_\mathbf{s}, T^\mathbf{s}, s)$, where $T_\mathbf{s} < T^\mathbf{s}$ are integers, $T_\mathbf{s}$ is called the starting time and $T^\mathbf{s}$ the closing 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 $T_\mathbf{s} \leq t_i < T^\mathbf{s}$ for all $i = 1, \ldots, n$, and $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n-1$. □

Example 7.2 Figure 7.3 presents graphically the event sequence $\mathbf{s} = (29, 68, s)$, where

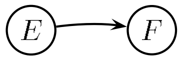
$$s = \langle (E, 31), (D, 32), (F, 33), (A, 35), (B, 37), (C, 38), \ldots, (D, 67) \rangle .$$

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. In the following we then discuss the case where one considers an event sequence as a sequence of partially overlapping windows.

Definition 7.3 A *window* on event sequence $\mathbf{s} = (T_\mathbf{s}, T^\mathbf{s}, s)$ is an event sequence $\mathbf{w} = (T_\mathbf{w}, T^\mathbf{w}, w)$, where $T_\mathbf{w} < T^\mathbf{s}, T^\mathbf{w} > T_\mathbf{s}$, and w consists of those pairs (A, t) from s where $T_\mathbf{w} \leq t < T^\mathbf{w}$. The time span $T^\mathbf{w} - T_\mathbf{w}$ 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 $width(\mathbf{w}) = win$. □

By 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 $\mathbf{s} = (T_\mathbf{s}, T^\mathbf{s}, s)$ and a window width win, the number of windows in $\mathcal{W}(\mathbf{s}, win)$ is $T^\mathbf{s} - T_\mathbf{s} + win - 1$.



α

Figure 7.4: An episode.

Example 7.4 Figure 7.3 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

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

Note that the event $(F, 40)$ that occurred at the closing 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 $(25, 30, \langle\rangle)$, and the last is $(67, 72, \langle (D, 67) \rangle)$. 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 $(63, 68, \langle (A, 65), (D, 67) \rangle)$. $\square$

We now move on to define episodes formally. We also define when an episode is a subepisode of another; this relation is then used as a strength hierarchy on episodes.

Definition 7.5 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. $\square$

Example 7.6 Consider episode $\alpha = (V, \leq, g)$ in Figure 7.4. The set V contains two nodes; call them x and y. The mapping g labels these nodes

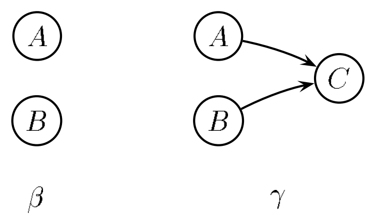


Figure 7.5: A subepisode and episode.

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 7.7 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 7.8 Figure 7.5 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 7.7, 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 7.9 An episode $\alpha = (V, \leq, g)$ *occurs* in an event sequence

$$\mathbf{s} = (T_\mathbf{s}, T^\mathbf{s}, \langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle),$$

if there exists an injective mapping $h : V \to \{1, \ldots, n\}$ 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 7.10 The window $(35, 40, w)$ of Figure 7.3 contains events A, B, C, and E, in that order. Both episodes β and γ of Figure 7.5 occur in the window. □

Definition 7.11 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 $\mathcal{F}_l(\mathbf{s}, win, min\_fr)$. □

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 $\mathcal{F}(\mathbf{s}, win, min\_fr)$. Algorithm 7.12 is an instantiation of the generic Algorithm 3.7 for computing 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; implementations of these steps are discussed in detail in the following sections.

Algorithm 7.12

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:

1. compute $\mathcal{C}_1 := \{\alpha \in \mathcal{E} \mid |\alpha| = 1\}$;
2. $l := 1$;
3. **while** $\mathcal{C}_l \neq \emptyset$ **do**
4.   // Database pass (Algorithms 7.18 and 7.20):
5.   compute $\mathcal{F}_l(\mathbf{s}, win, min\_fr) := \{\alpha \in \mathcal{C}_l \mid fr(\alpha, \mathbf{s}, win) \geq min\_fr\}$;
6. $l := l + 1$;
7.   // Candidate generation (Algorithm 7.13):
8.   compute $\mathcal{C}_l := \{\alpha \in \mathcal{E} \mid |\alpha| = l$, and $\beta \in \mathcal{F}_{|\beta|}(\mathbf{s}, win, min\_fr)$ for all $\beta \in \mathcal{E}, \beta \prec \alpha\}$;
9. **for** all l **do** output $\mathcal{F}_l(\mathbf{s}, win, min\_fr)$;

Instead of the level of a pattern (Definition 3.6), Algorithm 7.12 uses the sizes of episodes to assign episodes to iterations. For the classes of serial and parallel episodes the size of an episode is actually also the level of the episode in the class. Theorem 3.8 implies that Algorithm 7.12 works correctly, provided that the underlying subroutines work correctly.

The problem of discovering frequent injective parallel episodes is almost identical to searching for frequent sets. Injective parallel episodes are essentially subsets of R, and the collection of all windows on a sequence can be considered a binary database over the set R.

7.3 Generation of candidate episodes

We present now in detail a candidate generation method which is a generalization of the candidate generation for frequent sets. The method can be adapted to deal with parallel episodes (i.e., multisets of items), serial episodes (ordered multisets), and injective parallel and serial episodes (sets and ordered sets).

Algorithm 7.13 is a method for the computation of 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 with the square bracket notation. 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 $\mathcal{F}[i]$.

Algorithm 7.13
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.   // Build and test subepisodes β that do not contain $\alpha[y]$:
12.   **for** $y := 1$ **to** $l - 1$ **do**
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 $\mathcal{C}_{l+1}$;

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 i such that $\mathcal{F}_l[i]$ is the first episode in the block.

Theorem 7.14 Algorithm 7.13 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 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 12). 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 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 7.13 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 $\beta[4] = C$. The only change to the algorithm is to replace line 6.

Theorem 7.15 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 7.13 works correctly for serial episodes.

Proof The proof is similar to the proof for Theorem 7.14; 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 7.16 With the line

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

inserted after line 6, Algorithm 7.13 works correctly for injective parallel episodes (or injective serial episodes with the change of Theorem 7.15).

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. $\square$

The time complexity of Algorithm 7.13 is polynomial in the size of the collection of frequent episodes and it is independent of the length of the event sequence.

Theorem 7.17 Algorithm 7.13 (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 12 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 $\mathcal{O}(|\mathcal{F}_l| + |\mathcal{F}_l|\,|\mathcal{F}_l|\,(l^2 + (l-1)\,l\,\log |\mathcal{F}_l|)) = \mathcal{O}(l^2\,|\mathcal{F}_l|^2 \log |\mathcal{F}_l|)$. $\square$

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.

7.4 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 adjacent windows $\mathbf{w} = (T_i, T_i + win, w)$ and $\mathbf{w}' = (T_i + 1, T_i + win + 1, w')$, 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 7.18 recognizes candidate parallel episodes in an event sequence.

Algorithm 7.18
Input: A collection $\mathcal{C}$ of parallel episodes, an event sequence $\mathbf{s} = (T_{\mathbf{s}}, T^{\mathbf{s}}, s)$, 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 each A in α do
        A.count := 0;
        for i := 1 to |α| do contains(A, i) := ∅;
for each α in C do
    for each A in α do
        a := number of events of type A in α;
        contains(A, a) := contains(A, a) ∪ {α};
    α.event_count := 0;
    α.freq_count := 0;
// Recognition:
for start := T_s − win + 1 to T^s do
    // Bring in new events to the window:
    for all events (A, t) in s such that t = start + win − 1 do
        A.count := A.count + 1;
        for each α ∈ contains(A, A.count) do
            α.event_count := α.event_count + A.count;
            if α.event_count = |α| then α.inwindow := start;
    // Drop out old events from the window:
    for all events (A, t) in s such that t = start − 1 do
        for each α ∈ contains(A, A.count) do
            if α.event_count = |α| then
                α.freq_count := α.freq_count − α.inwindow + start;
            α.event_count := α.event_count − A.count;
        A.count := A.count − 1;
// Output:
for all episodes α in C do
    if α.freq_count/(T^s − T_s + win − 1) ≥ min_fr then output α;
```

The principles of the algorithm are the following. For each candidate parallel episode α we maintain a counter $\alpha.event\_count$ that indicates how many events of α are present in the window. When $\alpha.event\_count$ becomes equal to $|\alpha|$, indicating that α is entirely included in the window, we save the starting time of the window in $\alpha.inwindow$. When $\alpha.event\_count$ decreases again, indicating that α is no longer entirely in the window, we increase the field $\alpha.freq\_count$ by the number of windows where α remained entirely in the window. At the end, $\alpha.freq\_count$ 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 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 7.19 Algorithm 7.18 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 is 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 (note the set union operation). 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. $\square$

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 7.20 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 automaton is removed when the same event leaves the window. When an automaton for α reaches its accepting state, indicating that α is entirely included in the window, and if there are no other automata for α in the accepting state already, we save the starting time of the window in $\alpha.inwindow$. When an automaton in the accepting state is removed, and if there are no other automata 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 automata 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 $|\alpha|$ automata for an episode α. For each automaton we need to know when it should be removed. We can thus represent all the automata for α with one array of size $|\alpha|$: the value of $\alpha.initialized[i]$ is the latest initialization time of an automaton 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 efficiently they are organized in the following way. For each event type $A \in R$, the automata that 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 reaches a common state with another automaton, the earlier entry in the array $\alpha.initialized[]$ is simply overwritten.

Algorithm 7.20
Input: A collection $\mathcal{C}$ of serial episodes, an event sequence $\mathbf{s} = (T_\mathbf{s}, T^\mathbf{s}, s)$, 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 $\mathcal{C}$ **do**
  for $i := 1$ **to** $|\alpha|$ **do**
    $\alpha.initialized[i] := 0$;
    $waits(\alpha[i]) := \emptyset$;
for each $\alpha \in \mathcal{C}$ **do**
  $waits(\alpha[1]) := waits(\alpha[1]) \cup \{(\alpha, 1)\}$;
  $\alpha.freq\_count := 0$;
for $t := T_\mathbf{s} - win$ **to** $T_\mathbf{s} - 1$ **do** $beginsat(t) := \emptyset$;
// Recognition:
for $start := T_\mathbf{s} - win + 1$ **to** $T^\mathbf{s}$ **do**
  // Bring in new events to the window:
  $beginsat(start + win - 1) := \emptyset$;
  $transitions := \emptyset$;
  for all events (A, t) in s such that $t = start + win - 1$ **do**
    for all $(\alpha, j) \in waits(A)$ **do**
      if $j = |\alpha|$ and $\alpha.initialized[j] = 0$ **then** $\alpha.inwindow := start$;
      if $j = 1$ **then**
        $transitions := transitions \cup \{(\alpha, 1, start + win - 1)\}$;
      else
        $transitions := transitions \cup \{(\alpha, j, \alpha.initialized[j-1])\}$;
        $beginsat(\alpha.initialized[j-1]) :=$
          $beginsat(\alpha.initialized[j-1]) \setminus \{(\alpha, j-1)\}$;
        $\alpha.initialized[j-1] := 0$;
      $waits(A) := waits(A) \setminus \{(\alpha, j)\}$;
  for all $(\alpha, j, t) \in transitions$ **do**
    $\alpha.initialized[j] := t$;
    $beginsat(t) := beginsat(t) \cup \{(\alpha, j)\}$;
    if $j < |\alpha|$ **then** $waits(\alpha[j+1]) := waits(\alpha[j+1]) \cup \{(\alpha, j+1)\}$;
  // Drop out old events from the window:
  for all $(\alpha, l) \in beginsat(start - 1)$ **do**
    if $l = |\alpha|$ **then** $\alpha.freq\_count := \alpha.freq\_count - \alpha.inwindow + start$;
    else $waits(\alpha[l+1]) := waits(\alpha[l+1]) \setminus \{(\alpha, l+1)\}$;
    $\alpha.initialized[l] := 0$;
// Output:
for all episodes α in $\mathcal{C}$ **do**
  if $\alpha.freq\_count / (T^\mathbf{s} - T_\mathbf{s} + win - 1) \geq min\_fr$ **then** output α;

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 automata are done immediately, but updates for the new states are done only after all transitions have been identified, in order to not overwrite any useful information. For easy removal of automata when they go out of the window, the automata initialized at time t are stored in a list *beginsat(t)*.

Theorem 7.21 Algorithm 7.20 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.

Theorem 7.22 The time complexity of Algorithm 7.18 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 7.23 The time complexity of recognizing injective parallel episodes in Algorithm 7.18 (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 7.24 The time complexity of Algorithm 7.20 is $\mathcal{O}(n\,|\mathcal{C}|\,l)$.

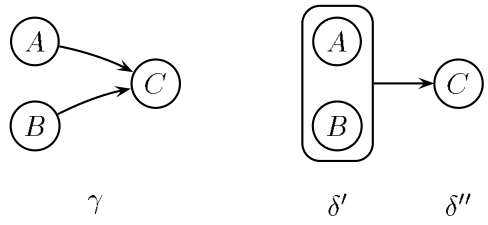


Figure 7.6: Recursive composition of a complex episode.

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 automata accessed; there are a maximum of l automata 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. 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 7.25 The time complexity of recognizing injective serial episodes in Algorithm 7.20 (excluding initialization) is $\mathcal{O}(n\,|\mathcal{C}|)$.

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

7.5 General partial orders

So far we have only discussed serial and parallel episodes. We next 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 7.6 is a serial combination of two episodes: δ', a parallel episode consisting of

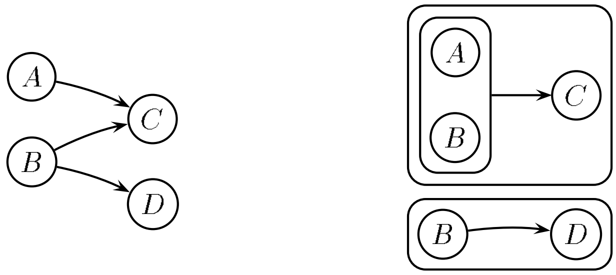


Figure 7.7: Recursive composition of a complex episode.

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 7.7. There is no hierarchical composition consisting only 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. In [MT96a], a related algorithm for the database pass is presented. The algorithm is actually based on storing minimal occurrences of episodes, and on using the minimal occurrences of two subepisodes to determine the minimal occurrences of a candidate.

Window width (s)	Serial episodes Count	Serial episodes Time (s)	Injective parallel episodes Count	Injective parallel episodes 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 7.1: Results of experiments with $\mathbf{s}_1$ using a fixed frequency threshold of 0.003 and a varying window width.

7.6 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 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 7.1 and 7.2 give an overview of the discovery of frequent episodes. In Table 7.1, 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 7.2, 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 handling 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.

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

Episode size	Number of episodes	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 7.3: 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.

Quality of candidate generation Table 7.3 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.

As can be seen from the table, a possible practical improvement is to

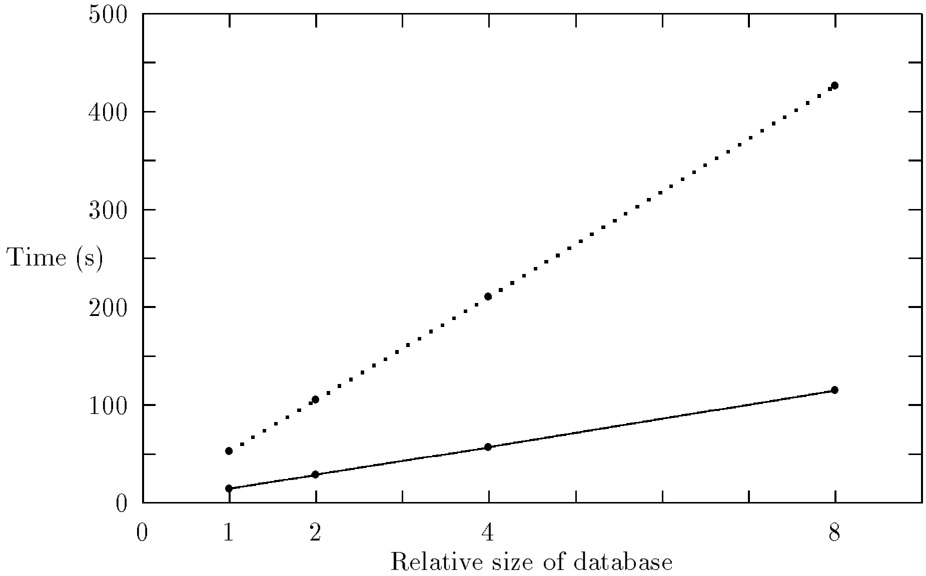


Figure 7.8: Scale-up results for serial episodes (dotted line) and injective parallel episodes (solid line) in $\mathbf{s}_1$ with window width 60 s and frequency threshold 0.01.

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 7.8 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 7.9 presents the ratio of times needed for trivial vs. incremental recognition of candidate episodes. The time required to generate the windows for the trivial 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 7.18: for injective parallel episodes the worst-case

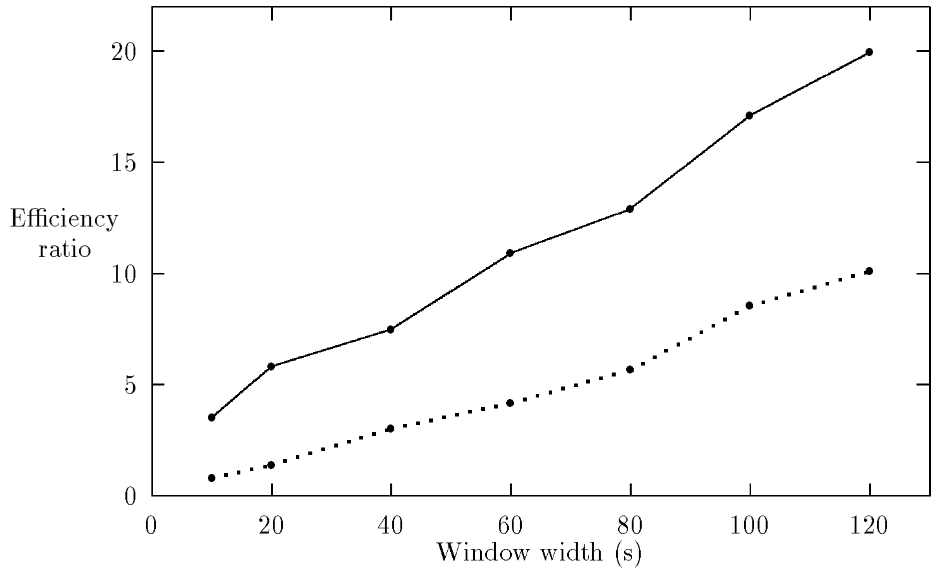


Figure 7.9: Ratio of times needed for trivial vs. incremental recognition methods in $\mathbf{s}_1$ for serial episodes (dotted line) and injective parallel episodes (solid line) as functions of window width.

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 7.4 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 7.4: Results of experiments with sequence $\mathbf{s}_2$.

collection of candidates consists of 385 episodes of sizes 1 to 11. Figure 7.10 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.

7.7 Extensions and related work

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 [HF95, HKMT95, SA95]. See Section 2.5 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 [For82]. In active databases

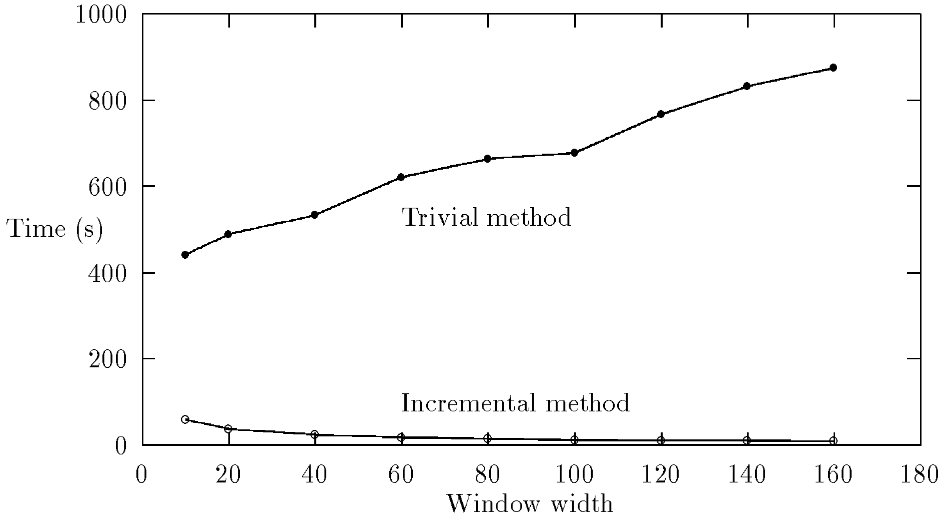


Figure 7.10: Time for the database pass over $\mathbf{s}_2$ as a function of the window width.

a related problem is the efficient detection of trigger events (see e.g. [GJS92]). Also, comparable strategies using a sliding window have been used, e.g., to study the locality of reference in virtual memory [Den68]. 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.

The methods for matching sets of episodes against a sequence have some similarities to the algorithms used in string matching (e.g., [GL89]). 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 [JCH95]. Closer to our patterns are those considered in [WCM$^+$94]. 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.

The problem of discovering so called sequential patterns is also closely related [AS95]. Sequential patterns are similar to serial episodes; they could also be roughly described as association rules discovered from ordered sets. More lately, the pattern class has been extended with windowing, some extra time constraints, and an event taxonomy [SA96b]. For a survey on patterns in sequential data, see [Lai93].

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. An extension similar to this last idea has been presented in [MT96a]. In that framework one can actually express episodes with two time bounds, such as "if A and B occur within 30 seconds, then C will follow within 2 minutes". Third, other patterns could be searched for, e.g., substrings with fixed length don't care symbols.

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.

As a final note it should be mentioned that the described methods have been used to analyze several alarm databases for telecommunication operators. Applications of the methods have been described in [HKM+96a, HKM+96b]. The goal of discovering new and useful knowledge has been achieved: episode rules discovered from alarm sequences have actually been taken into use in alarm handling software by the operators.

Chapter 8

Discussion

It gets harder the more you know.
Because the more you find out, the uglier everything seems.
– Frank Zappa

We have discussed the discovery of frequent patterns in large data collections. We presented an efficient method for the discovery of frequent sets and association rules in Chapter 2. We then generalized the problem setting and the algorithm in Chapter 3 for the discovery of frequent patterns in more general. We showed how exact database rules, inclusion dependencies, and functional dependencies can be discovered with the algorithm. We analyzed the problem of discovering all frequent patterns in Chapter 4, where we also showed that the algorithm for the discovery of frequent sets is optimal under some constraints. The knowledge discovery setting of Chapters 3 and 4, presented originally in [MT96c], is so far perhaps the only attempt to provide a unified view to data mining.

In Chapter 5 we considered the use of sampling in the discovery of association rules, and we gave methods that are efficient in terms of database activity. The idea was to determine from a random sample a probable superset of the collection of all frequent sets, and then to verify the result in one database pass. The approach was shown to work effectively for frequent sets. Such combination of sampling and verification can be used also in more general situations: random samples give good results at least when the selection criterion for patterns is based on the fraction of rows that match the pattern.

In Chapter 6 we looked at the discovery of Boolean rules, or association rules with negation and disjunction. We demonstrated that good approximations of Boolean rules can be computed from the frequent sets. Since the number of Boolean rules that hold in a database is usually huge, a useful approach is that the user queries the system for specific rules or classes of rules. Using the collection of frequent sets as a condensed representation [MT96b] of the database, the queries can be answered approximately without looking at the database. This approach has a strong connection to on-line analytical processing or OLAP: ad hoc queries for summaries from large data collections can be answered efficiently.

Finally, in Chapter 7, we introduced episodes, a novel class of regularities in event sequences. An episode is defined as a collection of events that tend to occur close to each other. From frequent episodes rules can be derived that describe connections between groups of events in the given sequence. Such rules can be used, e.g., to explain or predict events. We gave methods for the discovery of all frequent episodes; these methods have been successfully applied to the analysis of fault management data for telecommunication operators.

Combining the results from the last three chapters, i.e., sampling, Boolean rules, and episodes, is for the most part straightforward. Sampling from sequences is, however, not always useful: the sample size must be very small in order to outperform the presented incremental methods for the database pass.

In the larger picture of knowledge discovery, finding frequent patterns is just one part, although an important one. An essential ingredient of the whole discovery process is the identification of the most interesting or useful regularities. Thousands of association or episodes rules often hold in a data collection, and confidence and frequency alone are not enough to point out the most useful rules. The following interactive methodology for data mining—close to OLAP in spirit—has been suggested in [KMT96]: discover a large collection of rules or frequent sets at once, and provide tools with which the user can then efficiently query the rule set. This approach has the advantage over most methodologies for knowledge discovery and machine learning that changing the viewpoint does not result in a new and time consuming phase of discovering patterns.

In 1993, when the research leading to this thesis was started, data mining and knowledge discovery were concepts known only to few people, not including the author. The problem of discovering association rules was first introduced in the same year [AIS93]. Since then, data mining and knowledge discovery have become buzzwords within the database community. During these three years, about 50 research articles related to association rules alone have been published, including those that describe this research. Commercial interest, if not even hype, is now catching up quickly. Several companies have recently announced data mining products that, among other things, discover association rules and rules similar to episode rules.

Many interesting research problems remain open. How widely applicable is the generic algorithm for discovering all frequent sentences? Which classes of patterns can be searched reliably using sampling? What are useful condensed representations for different pattern types? How to locate the truly interesting patterns among all patterns that hold in a data collection?

References

[AIS93] 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 the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 207–216, Washington, D.C., 1993. ACM.

[AMS+96] 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.

[AS92] N. Alon and J. H. Spencer. *The Probabilistic Method.* John Wiley Inc., New York, NY, 1992.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, 1994. Morgan Kaufmann.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In P. Yu and A. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press.

[Bel95] S. Bell. Discovery and maintenance of functional dependencies by independencies. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 27–32, Montréal, Canada, 1995. AAAI Press.

[Ber73] C. Berge. *Hypergraphs. Combinatorics of Finite Sets.* North-Holland Publishing Company, Amsterdam, The Netherlands, 3rd edition, 1973.

[CFP84] M. A. Casanova, R. Fagin, and C. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28:29–59, 1984.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[DB93] 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.

[Den68] P. J. Denning. The working set model of program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[EG95] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

[EP96] J. F. Elder IV and D. Pregibon. A statistical perspective of 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.

[FK94] M. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. Technical Report LCSR-TR-225, Department of Computer Science, Rutgers University, Newark, NJ, 1994.

[FMMT96] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'96)*, pages 182–191, Montral, Canada, 1996. ACM.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[FPSSU96] 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.

[GJS92] N. Gehani, H. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*, pages 81–90, San Diego, CA, 1992. ACM.

[GL89] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33:113–120, 1989.

[HF95] 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, Zurich, Swizerland, 1995. Morgan Kaufmann.

[HKM+96a] 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 Twelth International Conference on Data Engineering (ICDE'96)*, pages 115–122, New Orleans, LA, 1996. IEEE Computer Society Press.

[HKM+96b] 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 *1996 IEEE Network Operations and Management Symposium (NOMS'96)*, pages 520–529, Kyoto, Japan, 1996. IEEE.

[HKMT95] 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, Montréal, Canada, 1995. AAAI Press.

[HS92] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In M. Stonebraker, editor, *Proceedings of* *the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*, pages 341–350, San Diego, CA, 1992. ACM.

[HS93] M. Houtsma and A. Swami. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Center, San Jose, CA, 1993.

[JCH95] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8):1587–1595, 1995.

[KA95] A. J. Knobbe and P. W. Adriaans. Discovering foreign key relations in relational databases. 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 94–99, Heraklion, Greece, 1995. MLnet.

[Kha95] R. Khardon. Translating between Horn representations and their characteristic models. *Journal of Artificial Intelligence Research*, 3:349–372, 1995.

[Klö95] W. Klösgen. Efficient discovery of interesting statements in databases. *Journal of Intelligent Information Systems*, 4(1):53–69, 1995.

[KLS95] J. Kahn, N. Linial, and A. Samorodnitsky. Inclusion-exclusion: exact and approximate. Manuscript, 1995.

[KM94] 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, May 1994. ACM.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 401–407, Gaithersburg, MD, 1994. ACM.

[KMRS92] 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, 1992.

[KMT96] M. Klemettinen, H. Mannila, and H. Toivonen. Interactive exploration of discovered knowledge: A methodology for interaction, and usability studies. Technical Report C-1996-3, Department of Computer Science, University of Helsinki, Finland, 1996.

[Lai93] 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, Germany, 1993. Springer-Verlag.

[LN90] N. Linial and N. Nisan. Approximate inclusion-exclusion. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing (STOC'90)*, pages 260–270, Baltimore, MD, 1990. ACM.

[Mit82] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[MR92a] H. Mannila and K.-J. Räihä. *Design of Relational Databases.* Addison-Wesley Publishing Company, Wokingham, United Kingdom, 1992.

[MR92b] H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40:237–243, 1992.

[MR94] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering*, 12(1):83–99, 1994.

[MT96a] 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, OR, 1996. AAAI Press.

[MT96b] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 189–194, Portland, OR, 1996. AAAI Press.

[MT96c] H. Mannila and H. Toivonen. On an algorithm for finding all interesting sentences. In R. Trappl, editor, *Cybernetics and Systems, Volume II, The Thirteenth European Meeting on Cybernetics and Systems Research*, pages 973–978, Vienna, Austria, 1996. Austrian Society for Cybernetic Studies.

[MTV94a] 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, WA, 1994. AAAI Press.

[MTV94b] H. Mannila, H. Toivonen, and A. I. Verkamo. Finding association rules efficiently in sequential data. Technical Report C-1994-40, Department of Computer Science, University of Helsinki, Finland, 1994.

[MTV95] 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, Montréal, Canada, 1995. AAAI Press.

[OR89] F. Olken and D. Rotem. Random sampling from B^+ trees. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases (VLDB'89)*, pages 269–277, Amsterdam, The Netherlands, 1989. Morgan Kaufmann.

[PCY95] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In M. Carey and D. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 175–186, San Jose, CA, 1995. ACM.

[PK95] B. Pfahringer and S. Kramer. Compression-based evaluation of partial determinations. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 234–239, Montréal, Canada, 1995. AAAI Press.

[PS91] 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.

[PTW83] G. Pólya, R. E. Tarjan, and D. R. Woods. *Notes on Introductory Combinatorics*. Birkhäuser, Boston, MA, 1983.

[SA95] 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, Zurich, Swizerland, 1995. Morgan Kaufmann.

[SA96a] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In H. Jagadish and I. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 1–12, Montréal, Canada, 1996. ACM.

[SA96b] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology—5th International Conference on Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, 1996. Springer-Verlag.

[Sie95] A. Siebes. Data surveying, foundations of an inductive query language. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 269–274, Montréal, Canada, 1995. AAAI Press.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in 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 432–444, Zurich, Swizerland, 1995. Morgan Kaufmann.

[TKR+95] 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, Greece, 1995. MLnet.

[Toi96] 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, 1996. Morgan Kaufmann.

[WCM+94] J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In R. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 115–125, Minneapolis, MI, 1994. ACM.

ISSN 1238-8645
ISBN 951-45-7531-8
Helsinki 1996
Helsinki University Printing House