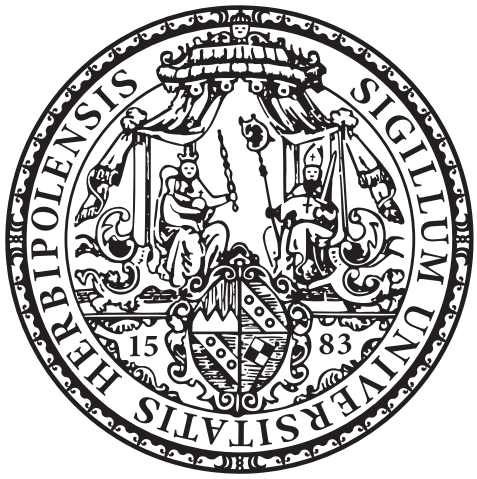
Julius-Maximilians-Universität Würzburg

Lehrstuhl für Künstliche Intelligenz
und Angewandte Informatik



Constraint Based Descriptive Pattern Mining

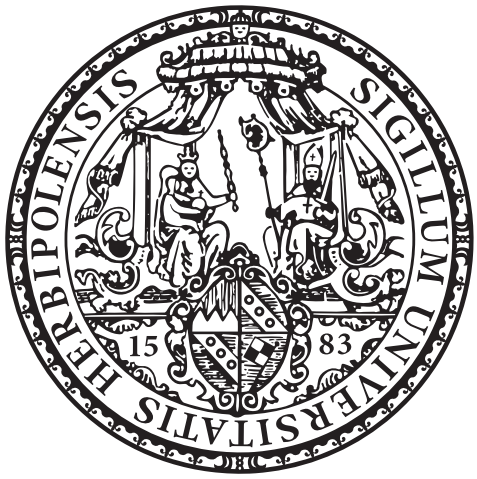
Datum:
13. Juli 2011

Diplomarbeit von
Martin Becker

Betreuer:
Prof. Dr. Andreas Hotho
Dr. Martin Atzmüller
Dipl. Inf. Florian Lemmerich

Julius-Maximilians-Universität Würzburg

Lehrstuhl für Künstliche Intelligenz
und Angewandte Informatik



Constraint Based Descriptive Pattern Mining

Datum:
13. Juli 2011

Diplomarbeit von
Martin Becker

Betreuer:
Prof. Dr. Andreas Hotho
Dr. Martin Atzmüller
Dipl. Inf. Florian Lemmerich

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Würzburg, 13. Juli 2011

Martin Becker

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Scope of the Work . . . 2
- 1.3 Structure . . . 3

2 Basics **5**
- 2.1 Pattern Mining . . . 9
  - 2.1.1 Descriptive Pattern Mining . . . 12
- 2.2 Constraints . . . 14
  - 2.2.1 Data Constraints . . . 16
  - 2.2.2 Description Constraints . . . 18
- 2.3 Valuation Basis . . . 20

3 Search **25**
- 3.1 Strategy . . . 25
  - 3.1.1 Pattern Discovery . . . 26
  - 3.1.2 Solution Discovery . . . 42
- 3.2 Optimizations . . . 45
  - 3.2.1 Pruning . . . 45
  - 3.2.2 Dynamic Constraints . . . 68
  - 3.2.3 Data Structures . . . 73

4 Constraints **93**
- 4.1 Pattern Constraints . . . 93
  - 4.1.1 Quality Functions and Optimistic Estimates . . . 93
  - 4.1.2 Descriptive Pattern Mining Classes . . . 96
  - 4.1.3 Condensed Itemset Mining . . . 102
- 4.2 Result Constraints . . . 106
  - 4.2.1 Dynamic Threshold . . . 108

5 Implementation **109**
5.1 The AnEMonE Algorithm 109
5.2 Framework 115

6 Evaluation **119**
6.1 Frequent Itemset Mining 121
6.1.1 Relative Frequency Threshold 122
6.1.2 Dynamic Frequency Threshold 133
6.1.3 ExAnte 138
6.2 Subgroup Mining 149
6.2.1 The *credit-g* Dataset and Branching Order 151
6.2.2 The *mushroom* Dataset and ExAnte 152
6.3 Community Mining 155

7 Conclusion **159**
7.1 Summary 159
7.2 Outlook 160

1

Introduction

Companies and institutions collect data not only for specific tasks but also for possible later usage. Enabled by practically boundless storage capabilities and communication technologies allowing sources from all over the world, tremendous amounts of data are amassed. The collected data ranges from transaction storages used by banks or retail organizations over customer profiles for insurance companies and friend graphs in social networks to unstructured data provided by internal documentation processes.

1.1 Motivation

[36, p. 3, par. 4] states that the mere variety, quantity and size of existing data collections renders manual data analysis a bottleneck. The field of Knowledge Discovery in Databases (KDD) rises to this challenge. Its community originated in the early 1990s (cf. [36, p. 2, par. 3]). In [25, p. 6, par 3] Fayyad et al. define KDD as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data". They also define Data Mining as part of the KDD process (see [25, p. 10]), which is responsible for "searching patterns of interest in a particular representational form or a set of such representations: classification rules, trees, regression, clustering, and so forth" (cf. [25, p. 10, par. 2]).

It is task specific which patterns found in a dataset are of interest. "Often, users have a good sense of which "direction" of mining may lead to interesting patterns and the "form" of the patterns or rules they would like to find" (cf. [32, p. 265-6]). One way to push such "intuition" and "expectations" into

the data mining process is by specifying constraints of varying types, for instance knowledge type, data, dimension, interestingness or rule constraints as listed by [32, p. 266, par. 2]. As these constraints confine the search space they potentially also improve the performance of the data mining process.

One particular, intensively studied constraint is the frequency constraint. "Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently" and mining such frequent patterns "has become an important data mining task and a focused theme in data mining research" (cf. [32, p. 227, par. 1]). Thus many efficient algorithms have emerged. One of the first efficient algorithms to find frequent itemsets in transaction databases was the Apriori algorithm introduced by [3]. Driven by the need for better performance and in the process of reducing costs introduced by candidate generation and database scans, the FP-Growth algorithm, as stated by [33], was developed (see [32, p. 243, par. 1-2]). Derivations of the FP-Growth algorithm were also applied to other related pattern mining tasks like subgroup mining (cf. [56, 7]) and mining descriptive community patterns (cf. [6]), e.g. in social networks.

Arguably these data mining tasks share a common structure. Additionally the possibility to apply derivation of the FP-Growth algorithm to solve them raises the question, if it is possible to specify a generic framework that covers their definition. An algorithm that solves the abstract problem stated by such a framework will be applicable to any of its instantiations seamlessly.

1.2 Scope of the Work

The focus of this work is to develop a pattern mining framework allowing to state the abstract problem of *pattern mining* which covers the definition of data mining tasks like frequent pattern mining (see [4]), subgroup mining (see [56, 7]) and mining descriptive community patterns (see [6]). Several optimizations and search strategies originating from these more specific problem settings are to be utilized as building blocks for deducing an algorithm efficiently solving the corresponding *pattern mining class*.

As stated above constraints are a basic utility to define which patterns of the search space are interesting. Thus the derived algorithm needs to support a variety of constraints allowing to specify different problem instantiations. Several possibilities of pruning the search space based on the properties of constraints are to be reviewed and incorporated into the algorithm. Especially anti-monotone and monotone constraints are important and well known constraint classes that allow efficient search space reduction. To further improve the efficiency of the search, specialized compact and fast data

structures will be revisited focusing on the FP-Tree from the FP-Growth algorithm.

1.3 Structure

Chapter 2 will define a framework making it possible to define *descriptive pattern mining*, a *pattern mining class* based on item-based *patterns*, which allows to formalize different pattern mining settings by defining constraints. It will furthermore introduce the notion of *valuation bases*, which are the core principle to derive an efficient algorithm to solve *descriptive pattern mining*.

Chapter 3 is to review different generally known search strategies and apply them to *descriptive pattern mining* in order to be able to formulate the exploitation of constraint properties for search space reduction abiding by the framework. It also revisits interchangeable data structures to be used by the search including the FP-Tree and bitset representation of datasets.

Chapter 4 formulates several data mining tasks like frequent pattern mining (see [4]), subgroup mining (see [56, 7]) and mining descriptive community patterns (see [6]) in the context of the framework allowing to apply the concepts reviewed in Chapter 3 and also lists other constraints common in the pattern mining community.

Chapter 5 combines the ideas from Chapter 3 to derive an algorithm solving the abstract problem of *descriptive pattern mining* and discusses its implementation.

Chapter 6 applies the algorithm from Chapter 5 to the *descriptive pattern mining* instances stated in Chapter 4 to illustrate and confirm the effects of the search strategy and its optimizations from Chapter 3 and to show the versatility of algorithms created to solve the generic problem of *descriptive pattern mining* defined using *valuation bases*.

Chapter 7 will give a summary of what has been achieved as well as perspectives for future work.

2

Basics

In data analysis typically a *population* $\mathbb{X}$ is studied [36, p. 33]. A *population* can contain a large amount of objects or *individuals*, so that usually only samples of the *population* are available. The data collected for each individual is often defined as values according to a fixed set of attributes [26, p. 579]. A more generic approach is to associate an *individual* with a *data instance* from an arbitrary *data domain* D according to field of interest. Such *data instances* can correspond to a wide range of actual data: the value of one or several variables, pictures, graphs, functions or any combination of the former. An example for a *population* is the set of all humans beings. If the field of interest is their height at a certain age, a *data instance* associated with an *individual* corresponds to a non-negative value being drawn from the the *data domain* $D = \mathbb{R}^+$. The *data domain* could also be defined as natural numbers $D = \mathbb{N}$, if only rough measurements are of interest. Note that based on the field of interest individuals cannot be distinguished if they are associated with the same *data instance*. Now, it is hardly possible to record the height of every human being. Nevertheless the *population* can be sampled. The *data instances* of a subset of *individuals* are recorded. The records of measured individuals are collected in a *data record set*. Each *data record* corresponds to an *individual* associated with a *data instance*.

Figure 2.1 shows an example of a *data record set*, where the height of human beings at a certain age is recorded. The *individuals* are identified by natural numbers. Definition 2.1 formally introduces the notion of *data record sets*. Example 2.1 formulates the *data record set* from Table 2.1 according to that definition.

individual	height in inches
11	28
1	30
17	40
111	52
33	28
34	28

Table 2.1: A *data record set* listing the height of human beings at a certain age.

Note 2.1 (Unique Identifiers and Natural Order)**.**
How individuals from a population $\mathbb{X}$ are identified in a data record set is arbitrary. In Table 2.1 the identifiers are chosen to be natural numbers, which can be interpreted to induce a natural order on individuals. Yet this order does not carry any meaning or information (e.g. order of recording). Instead an order can be introduced on the data level (for example by also recording the date of each recording). Any information carried by identifiers can be disregarded without loss of generality, because the identifiers and any possibly associated information can be pushed into the data domain. Thus all information is actually carried by a multiset of data instances if considering a single data record set independently of others. The reason for defining them otherwise is highlighted by Note 2.2.

Definition 2.1 (Data Record Set)**.**
*A **data record set** S is a tuple*

$$S = (R, \delta : \mathbb{X} \to D)$$

- *D is called **data domain** and is an arbitrary set. An element $d \in D$ is called a **data instance**.*
- *$\mathbb{X}$ is called **population** and is also an arbitrary set. An element $x \in \mathbb{X}$ is called an **individual**.*
- *$\delta : \mathbb{X} \to D$ is called **data recorder**, associating each individual with a data instance.*
- *$R \subseteq \mathbb{X}$ is a subset of the population $\mathbb{X}$ and is called the **record domain** of S.*
- *A tuple $(r, \delta(r))$, where $r \in R$ and $\delta(r) \in D$, is called a **data record**.*

A data recorder $\delta : \mathbb{X} \to D$ defines a set of data record sets

$$\Omega_\delta = \{(R, \delta) \mid R \subseteq \mathbb{X}\}$$

Set operations are defined on those data record sets with the same data recorder by projection onto their record domains (see example 2.2).

Definition 2.2 (Set Declarations)**.**

- *Let Δ_D denote the set of all data recorders on D.*
- *Let $\Omega_D = \bigcup_{\delta \in \Delta_D} \Omega_\delta$ denote the set of all data record sets recorded by all possible data recorders on D.*

Example 2.1 (Data Record Set)**.**
Based on Table 2.1 a data record set $S_{table} = (R, \delta : \mathbb{X} \to D)$ can be defined. To identify individuals the natural numbers are chosen $\mathbb{X} = \mathbb{N}$. The record domain $R \subseteq \mathbb{X}$ is $R = \{11, 1, 17, 11, 33, 34\}$. The data domain D represents all possible heights at a certain age. Assuming no decimal values, the domain of heights can be set to $D = \mathbb{N}$. Thus the data recorder δ can be defined as follows:

$$\begin{aligned} \delta : \quad & \mathbb{X} \to D \\ & i \mapsto \begin{cases} 28, \text{ if } i \in \{11, 33, 34\} \\ 30, \text{ if } i = 1 \\ 40, \text{ if } i = 17 \\ 52, \text{ if } i = 111 \\ 66, \text{ if } i = 12 \\ 0, \text{ otherwise} \end{cases} \end{aligned}$$

The fact that $\delta(12) = 66$, and that the data record $(12, 66)$ does not appear as an entry in Table 2.1, illustrates that data recorders are generally not bound to a specific data record set. They rather define a whole set of data record sets Ω_δ (see Definition 2.1) and S is merely one instance $S \in \Omega_\delta$ of the possibly recordable data record sets Ω_δ.

Example 2.2 (Set Operations on Data Record Sets)**.**
A few examples on set operations on data record sets are listed below. Let $S_1 = (R_1,\ \delta)$ and $S_2 = (R_2,\ \delta)$ be data record sets with the same data recorder δ.

- $S_1 \subseteq S_2 :\Leftrightarrow R_1 \subseteq R_2$
- $S_1 \cap S_2 := (R_1 \cap R_2,\ \delta)$
- $2^{S_1} := \{\ (R',\ \delta_1) \mid R' \in 2^{R_1}\}$
- $|S_1| := |R_1|$
- $r \in S_1 :\Leftrightarrow r \in R_1$

Note 2.2 (Unique Identifiers and Multi-Sets)**.**
As Note 2.1 states only the data instances according to the field of interest carry information and individuals cannot be distinguished if they are associated with the same data instance. Thus a data record set can also be defined as a multiset S_M. The data record set S_{table} from Table 2.1 for example can be expressed as

$$S_{M,table} = \{28, 30, 40, 52, 28, 28\}$$

Yet considering data record subsets

$$S_1 = (\{11, 1, 17, 111, 33\}, \delta)$$
$$S_2 = (\{11, 1, 17, 111, 34\}, \delta)$$

and their respective multiset representations

$$S_{M,1} = \{28, 30, 40, 52, 28\}$$
$$S_{M,2} = \{28, 30, 40, 52, 28\}$$

set operations are not well defined. Neither the join $\cup$ nor the multiset sum $\uplus$ return the original data record set $S_{M,table}$:

$$\begin{aligned} S_{M,1} \cup S_{M,2} &= \{28, 30, 40, 52, 28\} &&\neq S_{M,table} \\ S_{M,1} \uplus S_{M,2} &= \{28, 30, 40, 52, 28, 28, 28\} &&\neq S_{M,table} \end{aligned}$$

Furthermore vertical representations such as in Table 2.2 are not as straightforward to define. Thus it seems convenient to refrain from defining data record sets as multisets in order to maintain all information that is associated by collecting data allowing support for notions like vertical data formats or joining overlapping data record sets as illustrated above.

data instance	individuals the data instance is associated with
28	11, 33, 24
40	17
30	1
52	111

Table 2.2: Vertical format of the *data record set* from Table 2.1.

2.1 Pattern Mining

Fayyad et al. define knowledge discovery in databases as a process of identifying *patterns* in data and furthermore specify *patterns* as "expressions describing facts in a subset of data" (cf. [25]). This definition already imposes a meaning on *patterns*. Based on a more general view a *pattern* distinguishes *individuals* based on their *data instance*. Thus a *pattern* defines a class. An *individual* can either belong to the respective class or not. This notion is formalized in Definition 2.3. Note that because *individuals* associated with the same *data instance* cannot be distinguished, they always belong to the same set of classes.

Based on the *data record set* defined in Table 2.1 a *pattern* can for example describe all *individuals* with a height greater than 35 inducing the class "$height > 35$". Only the *individuals* 17 and 111 belong to that class. If other variables like hair color were also recorded, then "$height > 20 \wedge hairColor = brown$" would also define a class.

Definition 2.3 (Pattern)**.**
Let D be a data domain. A ***pattern*** *π on data domain D is a function*

$$\pi : D \to \{0, 1\}.$$

A data instance $d \in D$ is said to ***match*** *a pattern π on D, if and only if $\pi(d) = 1$. An individual $x \in \mathbb{X}$ recorded by data record set $S = (R, \delta : \mathbb{X} \to D)$ is said to* ***match*** *a pattern π on D, if and only if $\pi(\delta(x)) = 1$.*

Given a *data record set* S, the *individuals* matching a *pattern* represent *data record subsets*. Instead of storing the *individuals* for every *pattern* induced class separately, the respective *pattern* can be used to *project* a *data record set* onto the set of *individuals* by filtering those *individuals* not matching that *pattern* on demand. The notion of *projecting* a *pattern* and a *data record set* onto a *data record subset* is formalized by Definition 2.4.

Definition 2.4 (Pattern Projector)**.**
Let D be a data domain and let Ω_D be the set of all possible data record sets

on D (see Definition 2.2). Let $\pi : D \to \{0,1\}$ be a pattern on D. The function $\bar{\pi} : \Omega_D \to \Omega_D$ is called the ***pattern projector*** *of π on data domain D:*

$$\begin{aligned} \bar{\pi} : \quad & \Omega_D \to \Omega_D \\ & (R', \delta) \mapsto (\{i \in R \mid \pi(\delta(i)) = 1\},\ \delta). \end{aligned}$$

If S is a a data record set on D, then let S_π be an abbreviation for $\bar{\pi}(S)$. The data record subset S_π is said to be the ***projection*** *of pattern π onto the data record set S. The data record subset S_π is also called the* ***conditional data record set*** *of the data record set S based on pattern π. Thus S_π is created by* ***conditioning on*** *the pattern π.*

As mentioned above, *patterns* are convenient, because they

- save space, as they do not need to store all *individuals* separately to define a class,
- can be used as classifiers for *individuals* not part of the *data record set* and
- according to the definition by Fayyad et al. [25] carry meaning (e.g. "*height* > 35", which defines a class and also a *pattern*).

Note that in general *patterns* cannot describe all possible *data record subsets*, because *patterns* classify *individuals* based on their associated *data instance*. Thus *individuals* associated with the same *data instance* cannot be distinguished by a *pattern* as proven by Theorem 2.1. Without loss of generality this restriction can be circumvented by pushing an unique identifier into the *data domain* when recording. Also, as not the *data record set* itself, but the recorded data and its distribution is the focus of interest, this restriction is of no consequence during further investigations.

Theorem 2.1 (Sub Data Record Sets and Patterns)**.**
Let D be a data domain and $S = (R,\ \delta)$ be a data record set on D. Then there is no guarantee, that a pattern π exists for every data record subset $S' \subseteq S$, which projects S onto S', i.e. $\bar{\pi}(S) = S'$.

Proof. Let $R = \{x, y\}$ and let the data recorded for the identifiers x and y be the same, i.e. $\delta(x) = \delta(y)$. Furthermore let $S_x = (\{x\}, \delta)$ be a strict subset of S. $\pi(x) = 1$ must hold for a *pattern* π projecting onto S_x. Yet S_x is a strict subset of the projected *data record set* $\bar{\pi}(S)$, i.e. $S_x \subset \bar{\pi}(S) = S$. As a result *pattern* π does not project onto S_x. ↯. □

The number of possible *patterns* can be very large depending on the *data domain*. For example *data domains* $D = \{0,1\}^n$ featuring n binary variables induce 2^n *patterns*. Some of these *patterns* may be interesting, some may not according to the current data mining task. Rendering *patterns* interesting is usually based on which *individuals* are part of the *conditional data record subset* a *pattern* projects on. In a market basket database (each article is a binary variable), articles often bought together can be defined as interesting *patterns*. Each such *pattern* picks out those market baskets containing the articles it is associated with.

The task of finding constrained sets of *patterns* (i.e. solutions) based on a given *data record set* is called *pattern mining* and is formally introduced by Definition 2.6. The set of possible *patterns* potentially part of a solution is constrained by so called *pattern constraints* (see Definition 2.14) forming a set of valid *patterns*. The set of possible solutions is furthermore constrained by so called *result constraints* (see Definition 2.13).

Definition 2.5 (Pattern Mining Class)**.**
A ***pattern mining class*** *is a tuple* $\bar{M} = (D, \Pi, C_\Pi, C_R)$.

- D *is a data domain.*
- Π *is a set of patterns on* D.
- C_Π *is a set of constraints on patterns* Π *(pattern constraints).*
- C_R *is a set of constraints on sets of patterns* 2^Π *(result constraints).*

Definition 2.6 (Pattern Mining Instance)**.**
A ***pattern mining instance*** *is a tuple* $M = (\bar{M}, S)$.

- $\bar{M} = (D, \Pi, C_\Pi, C_R)$ *is a pattern mining class.*
- $S = (R,\ \delta : \mathbb{X} \to D)$ *is a data record set on data domain* D.

A pattern is called ***valid*** *if it satisfies every pattern constraint* $c \in C_\Pi$. $R \subset \Pi$ *is called a* ***solution*** *to* M*, if every pattern* $\pi \in R$ *is valid and if* R *satisfies every result constraint* $c \in C_R$.

Before specifying constraints more precisely the notion of *descriptive pattern mining* is introduced.

2.1.1 Descriptive Pattern Mining

A special type of *patterns* are *descriptive patterns*. Each *descriptive pattern* is associated with a set of items, that can be extracted from *data instances* of a *data domain*. Such items can be, but do not necessarily need to be, augmented with meaning like the value of a variable "$color = green$", a feature like "*data instance size* > 20" or even boolean formulas "$color = green \wedge$ *data instance size* > 20". A *data instance* matches a *descriptive pattern*, if the items associated with the *pattern* are a subset of those extracted from the *data instance*. In the case of market basket data, the items correspond to articles. A *descriptive pattern* is associated with a set of articles. An individual market basket matches a *descriptive pattern* if the market basket contains those items associated with the *descriptive pattern*.

To formally define *descriptive patterns* it is useful to introduce the notion of *data projectors* and *item projectors* first. *Data projectors* as stated in definition 2.7 project data into another, arbitrary set. Definition 2.8 defines a special kind of *data projector*, the *item projector*, which projects *data instances* onto a subset of a set of items. Definition 2.9 uses these notions to formally introduce the *descriptive pattern mining class*.

Definition 2.7 (Data Projector)**.**
Let D be a data domain and X be an arbitrary set. A ***data projector*** *ϕ on D is a function*

$$\phi : D \to X$$

Definition 2.8 (Item Projector)**.**
*Let I be a set of items, i.e. an arbitrary set. A data projector ϕ_I on data domain D is called an I-**item projector** on D, if its codomain $X \subseteq I$ is the powerset of the set of items I, i.e.*

$$\phi_I : D \to 2^I$$

Definition 2.9 (Descriptive Pattern)**.**
*Let D be a data domain and I be a set of items. Let $\phi_I : D \to 2^I$ be an I-item projector. With $P \subseteq I$, an I-**descriptive pattern** π_P on data domain D is a function*

$$\pi_P : \quad D \to \{0, 1\}$$
$$x \mapsto \begin{cases} 1, & \text{if } P \subseteq \phi_I(x) \\ 0, & \text{otherwise} \end{cases}$$

Let furthermore $\Pi_I = \{\pi_P \mid P \subseteq I\}$ *denote all possible* I*-descriptive patterns on a set of items* I*. Also when talking about descriptive patterns, it is equivalent to refer to either the pattern itself* π_P *or its associated itemset* P*, thus a descriptive pattern* π_P *can also be referred to as descriptive pattern* P *and adding an item* $i \in I$ *to the itemset* P *associated with* π_P *is equivalent to adding the item* i *to the descriptive pattern* P*.*

Before defining *descriptive pattern mining* in Definition 2.11, the notion of an *extension* of a *descriptive pattern* is introduced in Definition 2.10. It refers to the process of adding an item $i \in I$ to an I*-descriptive pattern* P, i.e. extending it by i. A *conditional data record set* based on a *descriptive pattern* P is a superset of the *conditional data records sets* based on any of its *extensions* P' as is proven by Theorem 2.2.

Definition 2.10 (Descriptive Pattern: Extension, Specialization, Reduction, Generalization)**.**
Let $P, P' \subseteq I$ *be* I*-descriptive patterns according to a set of items* I*.*

- P' *is called an* ***extension*** *or* ***specialization*** *of* P*,*
  if and only if $P \subseteq P'$*.*
- P' *is called a* ***reduction*** *or* ***generalization*** *of* P*,*
  if and only if $P' \subseteq P$*.*

Let E_P *be called* ***modification set*** *of the descriptive pattern* P*.* E_P *is a set of items* $E_P \subseteq I$*.*

- *If* E_P *is disjoint to* P*, i.e.* $E_P \cap P = \emptyset$*, it is called* ***extension set*** *and defines a set of extensions* $E_P^+ = \{P \cup E \mid E \subseteq E_P\} = P \cdot 2^{E_P}$*.*
- *If* E_P *is a subset of* P*, i.e.* $E_P \subseteq P$*, it is called a* ***reduction set*** *and defines a set of reductions* $E_P^- = \{P \setminus E \mid E \subseteq E_P\} = (P \setminus E_P) \cdot 2^{P \cap E_P}$*.*

Theorem 2.2.
Let π_P *and* $\pi_{P'}$ *be descriptive patterns on* D *and let* $S = (R, \delta : \mathbb{X} \to D)$ *be a data record set. Then*

$$P \subseteq P' \Rightarrow S_{\pi_{P'}} \subseteq S_{\pi_P}$$

Proof. Let ϕ_I be an I*-item projector.* Given Definition 2.4 and 2.9, the *conditional data record set* S_{π_P} based on a *descriptive pattern* π_P is defined as

$$S_{\pi_P} = (\{i \in R \mid P \subseteq \phi_I(\delta(i))\},\ \delta)$$

Thus

$$\begin{aligned} S_{\pi_{P'}} &= (\{i \in R \mid P' \subseteq \phi_I(\delta(i))\},\ \delta) \\ &= (\{i \in R \mid (P \cup P' \setminus P) \subseteq \phi_I(\delta(i))\},\ \delta) \\ &= (\{i \in R \mid P \subseteq \phi_I(\delta(i)) \wedge (P' \setminus P) \subseteq \phi_I(\delta(i))\},\ \delta) \\ &= (\{i \in R \mid P \subseteq \phi_I(\delta(i))\},\ \delta) \cap (\{i \in R \mid (P' \setminus P) \subseteq \phi_I(\delta(i))\},\ \delta) \\ &\subseteq (\{i \in R \mid P \subseteq \phi_I(\delta(i))\},\ \delta) \\ &= S_{\pi_P} \end{aligned}$$

□

Lemma 2.1 (Sequential Projection)**.**
Let π_P and $\pi_{P'}$ be descriptive patterns and let $S = (R, \delta)$ be a data record set. Then

$$P \subseteq P' \Rightarrow \bar{\pi}_{P'}(S) = \bar{\pi}_{P'}(\bar{\pi}_P(S))$$

or using a different notation

$$P \subseteq P' \Rightarrow S_{\pi_{P'}} = (S_{\pi_P})_{\pi_{P'}}$$

Definition 2.11 (Descriptive Pattern Mining Class)**.**
Given a set of items I and an I-item projector $\phi_I : D \to 2^I$, a pattern mining class

$$\bar{M} = (D, \Pi_I, C_{\Pi_I}, C_R)$$

is called a ***descriptive pattern mining class****, if*

$$\Pi_I = \{\pi_P \mid P \subseteq I\}$$

and can be written as

$$\bar{M} = (D, I, \phi_I, C_{\Pi_I}, C_R)$$

2.2 Constraints

As data mining processes can yield large amounts of *patterns* or rules, it is important to filter unrelated or uninteresting results with respect to the current task [32, p. 265]. *Constraints* are used to specify the scope of possible results to be mined. The general notion of a *constraint* is formalized by Definition 2.12.

Definition 2.12 (Constraint)**.**
Let X be an arbitrary set. Then

$$c : X \to \{0, 1\}$$

is called a ***constraint on*** *X. An element $x \in X$* ***satisfies*** *the constraint c, if and only if $c(x) = 1$. For compatibility with other work let furthermore*

$$\bar{c} = \{x \in X \mid c(x) = 1\}$$

As a result

$$c(x) = 1 \Leftrightarrow x \in \bar{c}$$

Concerning notation $\bar{c}$ can be written as c, if ambiguity is not an issue. Thus

$$c(x) = 1 \Leftrightarrow x \in c$$

Note 2.3 (Patterns as Constraints)**.**
By Definition 2.12 of constraints, patterns are constraints on data instances. This fact is exploited for formulating descriptive pattern mining as a constraint programming problem by [22].

In *pattern mining* there are two sets of *constraints*:

- *result constraints*: *constraints* C_R on sets of *pattern*, i.e. on the solutions of the *pattern mining* instance and
- *pattern constraints*: *constraints* on individual *patterns* C_Π, i.e. on the *patterns* that can be used to make up solutions.

In general the evaluation of *patterns* and solutions in *pattern mining* by *constraints* depends on the *data record set* which is given for the specific *pattern mining instance*. *Result constraints* are formally introduced in Definition 2.13. An example is given by Example 2.3. Definition 2.14 formalizes *pattern constraints* illustrated by Example 2.4.

Definition 2.13 (Result Constraint)**.**
Given a set of patterns Π and a data domain D, a ***result constraint*** *c_R is a constraint on $2^\Pi \times \Omega_D$:*

$$c_R : 2^\Pi \times \Omega_D \to \{0, 1\}$$

Example 2.3 (Result Constraint: Size Limitation)**.**
Solutions made of a lot of patterns can be hard to analyze. A few patterns

might be enough to satisfy the needs of the data mining task. Thus the amount of patterns in a solution can be limited, i.e. to a maximum of k patterns.

$$c_{R,size\leq k}: \quad 2^{\Pi} \times \Omega_D \to \{0,1\}$$
$$(R,S) \mapsto \begin{cases} 1, \text{ if } |R| \leq k \\ 0, \text{ otherwise} \end{cases}$$

Note that this result constraint is independent of the data record set given by the corresponding pattern mining instance.

Definition 2.14 (Pattern Constraint)**.**
Given a set of patterns Π *and a data domain* D*, a* ***pattern constraint*** *is a constraint on* $\Pi \times \Omega_D$*:*

$$c : \Pi \times \Omega_D \to \{0,1\}$$

Example 2.4 (Pattern Constraint: Frequent Patterns)**.**
Patterns often are only interesting, if they exist in a data record set, i.e. a data record set actually contains individuals, that match a pattern. Other mining tasks, like frequent itemset mining (cf. [4]), specify a threshold t*, that has to be surpassed by a pattern to be deemed interesting. The count of individuals in a data record set that match a pattern is referred to as* ***frequency*** *or* ***support****.*

Given a data record set S *and a set of patterns* Π*, the constraint* $c_{supp\geq t}$ *only renders patterns* $\pi \in \Pi$ *valid, that project onto data record subsets with a cardinality greater than or equal to a given threshold* t*. In other words, all valid* $\pi \in \Pi$ *with* $c_{supp\geq t}(\pi) = 1$ *are* ***frequent*** *patterns with respect to* t *or in other words satisfy the support threshold* t*:*

$$c_{supp\geq t}: \quad \Pi \times \Omega_D \to \{0,1\}$$
$$(\pi, S') \mapsto \begin{cases} 1, \text{ if } |S'_\pi| \geq t \\ 0, \text{ otherwise} \end{cases}$$

2.2.1 Data Constraints

As shown in Example 2.4, *pattern constraints* can solely be based on the given *data record set* S and the *conditional data record set* S_π based on the *pattern* π to evaluate. Information about the *pattern* itself is disregarded (such information include is e.g. the number of items associated with a *descriptive pattern*). These *pattern constraints* are called *data constraints* as formalized by Definition 2.15.

Definition 2.15 (Data Constraint)**.**
Let D be a data domain and let c be a constraint on $\Omega_D \times \Omega_D$:

$$c : \Omega_D \times \Omega_D \to \{0, 1\}$$

*A **data constraint** c_{data} is a pattern constraint based on c, defined as:*

$$\begin{aligned} c_{data} : \quad & \Pi \times \Omega_D \to \{0, 1\} \\ & (\pi, S) \mapsto c(S_\pi, S) \end{aligned}$$

Note 2.4 (Defining Data Constraints)**.**
Definition 2.15 makes clear, that defining a data constraint is equivalent to defining a constraint c on $\Omega_D \times \Omega_D$. Arguments will be used interchangeably: $c(\pi, S)$ and $c(S', S)$, where π is a pattern and S and S' are data record sets.

Example 2.5 (Data Constraint: Weight and Frequency)**.**
Data instances can be weighted by significance. For example not all recorded data corresponding to an individual might be measured with the same accuracy. Data that was measured very accurately is rated higher than data measured inaccurately. A pattern can be interpreted as inaccurate and thus as not valid, if the average weight of its projection onto a given data record set is below the average weight of the whole data record set.

Let D be data domain and $\phi_{weight} : D \to \mathbb{R}$ a data projector, that projects a data instance associated with an individual onto a weight corresponding to the accuracy of the recorded data. Only patterns that project onto data record subsets with an average weight greater than or equal to the average weight of the whole data record set satisfy the constraint $c_{avg(weight)}$:

$$\begin{aligned} c_{avg(weight)} : \quad & \Omega_D \times \Omega_D \to \{0, 1\} \\ & (S', S) \mapsto \begin{cases} 1, & \text{if} \quad \frac{\sum_{x \in S'} \phi_{weight}(\delta(x))}{|S'|} \geq \frac{\sum_{x \in S} \phi_{weight}(\delta(x))}{|S|} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Also, the pattern constraint $c_{supp \geq t}$ from example 2.4 can be identified as a data constraint:

$$\begin{aligned} c_{supp \geq t} : \quad & \Omega_D \times \Omega_D \to \{0, 1\} \\ & (S', S) \mapsto \begin{cases} 1, & \text{if} \quad |S'| \geq t \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

2.2.2 Description Constraints

In contrast to *data constraints*, a class of *constraints* can be identified, which is solely based on information associated with a *pattern* independent of the given *data record set*. A special kind of such *constraints* is defined on *descriptive patterns*. Given a set of items I, each *descriptive pattern* π_P is associated with a set of items $P \subseteq I$. Based on this "description", *description constraints* can render *descriptive patterns* valid or not. The formal definition is given by Definition 2.16. Examples are given by Example 2.6.

Definition 2.16 (Description Constraint)**.**
Let I be a set of items and let Π_I be the set of all I-descriptive patterns. Furthermore let c be a constraint on the powerset of I, i.e. 2^I:

$$c : 2^I \to \{0, 1\}$$

*A **description constraint** c_{desc} is a pattern constraint dependent on c and defined as follows:*

$$\begin{aligned} c_{desc} : \quad & \Pi_I \times \Omega_D \to \{0, 1\} \\ & (\pi_P, S) \mapsto c(P) \end{aligned}$$

Note 2.5 (Defining Description Constraints)**.**
Definition 2.16 makes clear, that, given a set of items I, defining a description constraint is equivalent to defining a constraint c on 2^I. Arguments will be used interchangeably: $c(\pi_P, S)$ and $c(P)$, where π_P is an I-descriptive pattern, S is a data record set and $P \subseteq I$ is a subset of the set of items I.

Items can be associated with certain properties. In frequent itemset mining for example items are sometimes associated with a price (cf. [4, 12]). Items can also be associated with categories or other properties. Definition 2.17 introduces *item properties* formally.

Definition 2.17 (Item Property)**.**
*Given a set of items $I = \{i_1, \ldots, i_n\}$, an **item property** is a function*

$$\phi : I \to X$$

where X is an arbitrary set.

Example 2.6 (Description Constraint: Price)**.**
As in frequent itemset mining [4, 12], a description constraint $c_{price \geq t}$ can be based on a price each item is associated with. Only itemsets P (and their

ID	itemset:weight
1	$\{a, b, c\} : 2$
2	$\{a, b\} : 2$

Table 2.3: A minimal *data record set* of itemsets with weights.

associated patterns π_P) are rendered valid ($c_{price \geq t}(P) = 1$), which sum up to a price equal to or above a certain threshold t.

Given a set of items I, let $\phi_{price} : I \to \mathbb{R}^+$ be an item property associating each item with a price. Let Π_I be the set of all I-descriptive patterns, then $c_{price \geq t}$ is defined as

$$c_{price \geq t} : \quad 2^I \to \{0, 1\}$$
$$P \mapsto \begin{cases} 1, & \text{if} \quad \sum_{i \in P} \phi_{price}(i) \geq t \\ 0, & \text{otherwise} \end{cases}$$

Long itemsets (and their associated patterns) can be hard to read and difficult to interpret. A description constraint can limit the itemset size a descriptive pattern is associated with to a maximal size of t. The corresponding description constraint $c_{size \leq t}$ is defined as

$$c_{size \leq t} : \quad 2^I \to \{0, 1\}$$
$$P \mapsto \begin{cases} 1, & \text{if} \quad |P| \leq t \\ 0, & \text{otherwise} \end{cases}$$

Note 2.6 (Data and Description Patterns)**.**
Patterns projecting onto the same data record sets will always yield the same evaluation for any data constraint c_{data}, i.e.

$$S_P = S_{P'} \Rightarrow c_{data}(\pi_P, S) = c_{data}(\pi_{P'}, S)$$

This is not necessarily true for description constraints.

For example let S be the data record set corresponding to Table 2.3. Let c_{weight} be the data constraint from Example 2.5 and let $c_{size \leq 1}$ be the description constraint from Example 2.6 with a threshold $t = 1$. Then

$$S_{\pi_{\{a\}}} = S_{\pi_{\{a,b\}}}$$

and

$$c_{weight}(\pi_{\{a\}}, S) = c_{weight}(\pi_{\{a,b\}}, S)$$

but

$$c_{size \leq 1}(\pi_{\{a\}}, S) = 1 \neq 0 = c_{size \leq 1}(\pi_{\{a,b\}}, S)$$

2.3 Valuation Basis

The information a *pattern constraint* uses to evaluate a pattern is twofold: it accesses information about the *pattern* by itself and in combination with an initial *data record set* S (see Definition 2.12). By applying the *pattern* π to the given *data record set* S, the *pattern constraint* c can also access the *conditional data record set* S_π. These two *data record sets* contain all information based on *data instances* available to the constraint c in order to evaluate the *pattern* π. The notion of *valuation bases* represents both *data record sets* as single objects, i.e. *valuation bases*. The basic building blocks of such objects are *data instances*. Consequently the most general *valuation basis* of a *data record set* is given by the corresponding multiset of *data instances*. Moreover each *data instance* can be interpreted as a *valuation basis*. Thus, more abstractly, the *valuation basis* of a *data record set* can be calculated by iterating over its *data instances* extracting the corresponding *valuation bases* and accumulating them. Following this line of thought the *valuation bases* associated with two distinct *data record subsets* can be accumulated to yield the *valuation basis* of their union. Figure 2.1 shows a schematic illustration of these concepts and Definition 2.18 formalizes them. Associating a *data instance* with a *valuation basis* is done by a *valuation basis projector* as introduced in Definition 2.19.

Definition 2.18 (Valuation Basis, Valuation Domain)**.**
Let V be an arbitrary set and $\oplus$ be a binary operator on V, i.e.

$$\begin{aligned} \oplus : \quad & V \times V \to V \\ & (a, b) \mapsto a \oplus b \end{aligned}$$

Then a ***valuation domain*** *$\mathbb{V} = (V, \oplus)$ is an abelian semigroup, i.e.*

- *V is closed under $\oplus$, i.e. $a, b \in V \Rightarrow a \oplus b \in V$*
- *$\oplus$ is associative, i.e. $a, b, c \in V \Rightarrow a \oplus (b \oplus c) = (a \oplus b) \oplus c$*
- *$\oplus$ is commutative, i.e. $a, b \in V \Rightarrow a \oplus b = b \oplus a$*

An element $v \in V$ is called a ***valuation basis****.*

Definition 2.19 (Valuation Basis Projector)**.**
Given a data domain D and a valuation domain $\mathbb{V} = (V, \oplus)$, a ***V-valuation basis projector*** *is a data projector*

$$\phi_V : D \to V$$

Furthermore let $\bar{\phi}_V$ *be defined as*

$$\begin{aligned}\bar{\phi}_V : \quad & \Omega_D \to \mathbb{N}_0^D \\ & S \mapsto \bigoplus_{r \in S} \phi_V(\delta(r))\end{aligned}$$

where $\mathbb{N}_0^D$ *is the set of all multisets over* D*. Note that data instances can occur multiple times in a multiset.* $\bar{\phi}_V$ *is used to project a data record set onto its valuation basis.* $\bar{\phi}_V$ *is also written as* ϕ_V *is ambiguity is not an issue.*

A simple *valuation basis* is the frequency or the support. The corresponding *valuation domain* $\mathbb{V}_{supp} = (\mathbb{N}, +_{\mathbb{N}})$ is the set of natural numbers $\mathbb{N}$ together with the addition operator $\oplus = +$. Each *individual* only occurs one time in the *data record set*, thus, it is associated with a support of one. The accumulated *valuation basis* for any *data record set* or *data record subset* is the sum of all supports. The same is applicable for a weight associated with each *data instance*. The corresponding *valuation domain* $\mathbb{V}_{weight} = (\mathbb{R}^+, +_{\mathbb{R}})$ can be the set of positive real numbers together with the addition operator $\oplus = +$. Now, some *constraints* might need the support for evaluation and some a sum of weights. Instead of associating several *valuation bases* with a single *data instance* an aggregate *valuation basis* is formed. In the case of support and a sum of weights, the tuple $(1, \phi_{\text{weight}}(\delta(i))) \in V_{supp} \times V_{weight}$ is the aggregated *valuation basis* for both *data instance*. The corresponding *valuation domain* is $\mathbb{V}_{combined} = (V_{supp} \times V_{weight}, \oplus_{+_{\mathbb{N}},+_{\mathbb{R}}})$, where $\oplus_{+_{\mathbb{N}},+_{\mathbb{R}}}$ applies the accumulation operators of each aggregated *valuation basis* componentwise, i.e.

$$\begin{aligned}\oplus_{+_{\mathbb{N}},+_{\mathbb{R}}} : \quad & (V_{supp} \times V_{weight}) \times (V_{supp} \times V_{weight}) \to V_{supp} \times V_{weight} \\ & ((a_{supp}, a_{weight}), (b_{supp}, b_{weight})) \mapsto (a_{supp} +_{\mathbb{N}} b_{supp}, a_{weight} +_{\mathbb{R}} b_{weight})\end{aligned}$$

Thus *valuation domains* can be combined at will by using the carthesian product. See Figure 2.2 illustrating the combination of support and weight in one *valuation basis*. If *valuation bases* share components a compressed aggregated representation is also possible. E.g. $\mathbb{V}_{combined}$ and $\mathbb{V}_{supp}$ trivially share the support component, where $\mathbb{V}_{combined}$ is the compressed representation of their aggregate.

As derived in the introduction of this section, the most general *valuation basis* is a multiset of *data instances*. Thus given a *data domain* D the corresponding *abelian semigroup* $\bar{\mathbb{V}} = (\mathbb{N}_0^D, \uplus)$ is the corresponding *valuation domain*, where $\mathbb{N}_0^D$ denotes set set of all multisets based on D and $\uplus$ the multiset sum, i.e. $\{1, 1, 2\} \uplus \{1, 1, 3\} = \{1, 1, 1, 1, 2, 3\}$ (see Note 2.2). A *data record* $(i, \delta(i))$ is associated with the corresponding *data instance* $\delta(i)$ itself

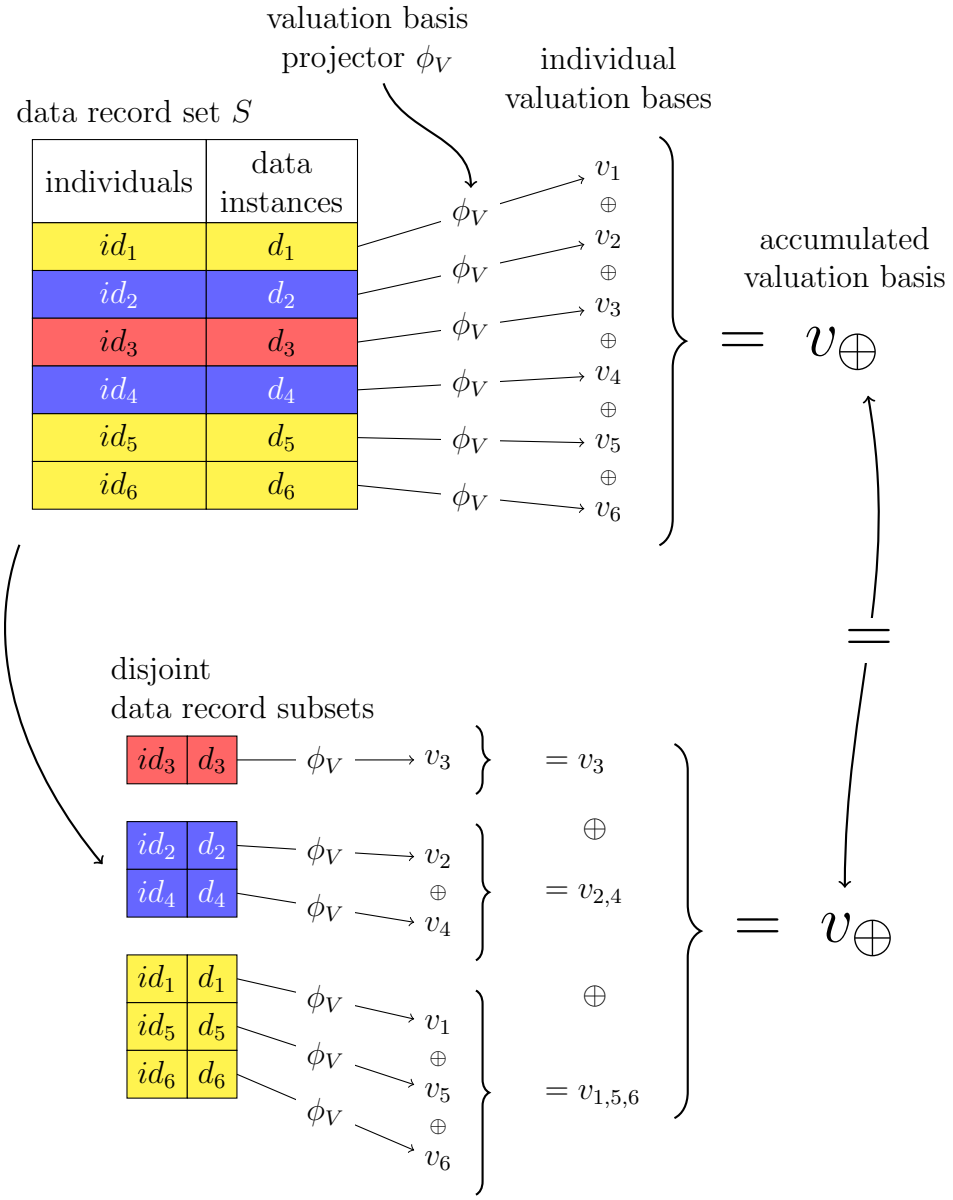


Figure 2.1: A schematic illustrations of *valuation bases* and their accumulation.

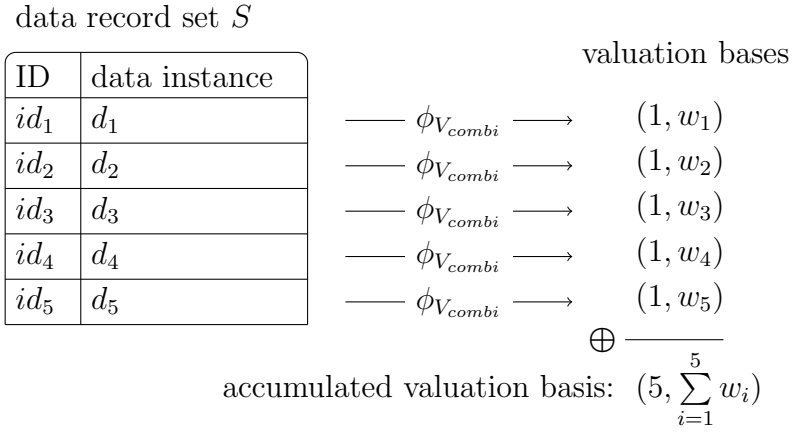


Figure 2.2: A schematic illustration of *valuation bases* consisting of frequency and a weight. The corresponding *valuation domain* is $\mathbb{V}_{combined} = (V_{supp} \times V_{weight}, \oplus_{+_{\mathbb{N}},+_{\mathbb{R}}})$.

(see Figure 2.3). Thus the $\bar{V}$*-valuation basis projector* $\phi_{\bar{V}}$ is

$$\begin{aligned} \phi_{\bar{V}} : \quad & D \to \mathbb{N}_0^D \\ & d \mapsto \{d\} \end{aligned}$$

This kind of a *valuation basis* subsumes all information contained by a *data record set* because, without loss of generality, the association with identifiers does not carry any information (see Note 2.1 and Note 2.2). As a result it is equivalent to use either *data record sets* or their respective *valuation bases* as input for *constraints*. Dependent on the *constraint*

- the *valuation basis* $\phi_V(S)$ of the whole *data record set* S,
- the *valuation basis* $\phi_V(S_\pi)$ of the *conditional data record set* S_π based on the *pattern* π to evaluate and
- the *valuation basis* $\phi_V(\bar{S}_\pi)$ corresponding to the *data record subset* $\bar{S}_\pi$ only containing *data records* not matching the *pattern* π to evaluate

are needed. If considering the *valuation basis* $\bar{\mathbb{V}}$ then $\phi_V(\bar{S}_\pi)$ can be derived from $\phi_V(S)$ and $\phi_V(\bar{S}_\pi)$. Consequently *constraints* can be reformulated taking *valuation bases* as arguments instead of the initial *data record set* given an appropriate *valuation domain*. Let $\mathbb{V} = (V, \oplus)$ be an appropriate *valuation domain*, then a *pattern constraint* c is defined as

$$c : \Pi \times V \times V \to \{0, 1\}$$

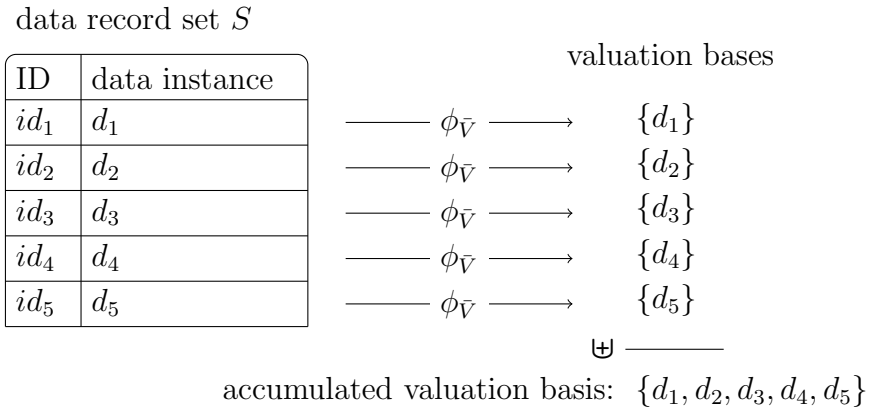


Figure 2.3: A schematic illustration of *valuation bases* consisting of multisets of *data instances*, which is the most general *valuation basis* to be specified. The corresponding *valuation domain* is $\bar{\mathbb{V}} = (\mathbb{N}_0^D, \uplus)$. Note that $d_i = d_j$ is possible for any pair of *data instances* with $i, j \in \{1, 2, 3, 4, 5\}$.

The third argument is always set to the *valuation basis* of the initial *data record set* of a *descriptive data mining instance*, i.e. evaluating a *pattern* π would be done by calling $c(\pi, \phi_V(S_\pi), \phi_V(S))$.

3

Search

A solution of a *pattern mining instance* $M = ((D, \Pi, C_\Pi, C_R), S)$ is a set of patterns $R \subseteq \Pi$. Searching solutions can be done exhaustively. It can also be sufficient to find a single solution. Search strategies depend on the objective and the *pattern mining class*. One class of *pattern mining* is the *descriptive pattern mining class* as introduced in Section 2.1.1, where *patterns* π_P are based on subsets $P \subseteq I$ of a set of items I. Section 3.1 states the general problem setting for any search strategy used to solve a *pattern mining instance*. It then reviews *bottom-up* and *top-down* enumeration of itemsets as an approach to solving *descriptive pattern mining instances*. The possibility of solution generation while enumerating *descriptive patterns* is also covered.

Considering that in *descriptive pattern mining* the space of patterns Π grows exponentially with respect to the amount of possible items $|I|$, it is important to optimize the search strategies in use. Section 3.2 builds upon the *bottom-up* approach from Section 3.1 and reviews several possibilities for optimization including pruning methods and efficient data structures.

3.1 Strategy

There are two prominent objectives when solving a *pattern mining instance*

$$M = ((D, \Pi, C_\Pi, C_R), S)$$

- One is searching for the complete set of *solutions*.
- The other is searching for a single *solution*.

Listing 3.1: *Pattern discovery.*

```
1 PROCEDURE patternDiscovery(Π, C_Π, S):
2     Π_valid ← ∅
3     forall π ∈ Π do
4         if (π, S) statisfies all c_Π ∈ C_Π do
5             Π_valid ← Π_valid ∪ {π}
6         endif
7     endfor
8     return Π_valid
```

Independent of the objective the vanilla way of searching for solutions can be divided into two steps, which are applied in sequence:

- **pattern discovery**: enumerate all possible *patterns* $\pi \in \Pi$ and filter them according to the *patten constraints* C_Π, so that only valid *patterns* Π_{valid} remain
- **solution discovery**: enumerate all possible sets of valid *patterns* $R \subseteq \Pi_{valid}$ and filter them according to the *result constraints* C_R, until only solutions remain, i.e. sets of valid *patterns* that satisfy all *result constraints* C_R

If it is sufficient to find only one solution, then the *solution discovery* step terminates as soon as one enumerated set of valid *patterns* satisfies all *result constraints* C_R. This Section focuses on the

- *descriptive pattern mining class* $\bar{M} = (D, I, \phi_I, C_{\Pi_I}, C_R)$
- searching for a *single* solution.

Section 3.1.1 reviews *bottom-up* and *top-down* generation of itemsets to enumerate *descriptive patterns*. A generative approach to *solution discovery* is outlined contrasting the more intuitive selective approach mentioned above in Section 3.1.2.

3.1.1 Pattern Discovery

Let $M = ((D, \Pi, C_\Pi, C_R), S)$ be a *pattern mining instance*. Then the *pattern discovery* step is responsible for filtering *patterns* Π, so that only valid *patterns* Π_{valid} remain, i.e. those patterns that satisfy all *pattern constraints* C_Π. As shown in Listing 3.1 the most basic variant of *pattern discovery* is to iterate over all *patterns* $\pi \in \Pi$ and check, if they satisfy every *patten constraint* $c_\Pi \in C_\Pi$.

Listing 3.2: *Descriptive pattern discovery.*

```
1 PROCEDURE patternDiscovery(I, C_{Π_I}, S):
2     Π_valid = ∅
3     forall P ∈ 2^I do // indirect enumeration of patterns by their associated itemsets
4         if (π_P, S) satisfies all c_{Π_I} ∈ C_{Π_I} do
5             Π_valid ← Π_valid ∪ {π_P}
6         endif
7     endfor
8     return Π_valid
```

Let $M = ((D, I, \phi_I, C_{\Pi_I}, C_R)$ be a *descriptive patter mining instance*. Then the *descriptive patterns* $\Pi_I = \{\pi_P \mid P \subseteq I\}$ are indirectly specified by the given set of items I. Thus *descriptive patterns* π_P can be enumerated by iterating over all subsets $P \subseteq I$ of the set of items I. Listing 3.2 shows the respectively modified version of the algorithm in Listing 3.1.

Enumeration Methods: Bottom-Up and Top-Down

The concept of *descriptive patterns* appears in a variety of different data mining tasks. In frequent itemset mining (cf. [4]) a *descriptive pattern* corresponds to an itemset. In subgroup mining (cf. [56]) subgroup descriptions are often defined as conjunctions of descriptive elements and community mining as introduced in [6] defines an equivalent concept of community descriptions. Not considering the individual *constraints*, the search space for any of these applications is the same, i.e. a powerset 2^I of items I. [39] states that an algorithm searching for patterns traverses a search lattice in most cases. Indeed the search space induced by *descriptive pattern mining* can be pictured as the powerset lattice based on the corresponding set of items (see Figure 3.1(a)). There are two dominating approaches to traversing the lattice of *descriptive patterns*:

- the *bottom-up* and
- the *top-down* approach.

Both, the *bottom-up* and the *top-down* traversal, have two important properties:

- they can be designed to avoid enumerating a single *descriptive pattern* twice and
- instead of traversing the whole lattice their characteristic way of enumerating *descriptive patterns* in combination with certain *constraints*

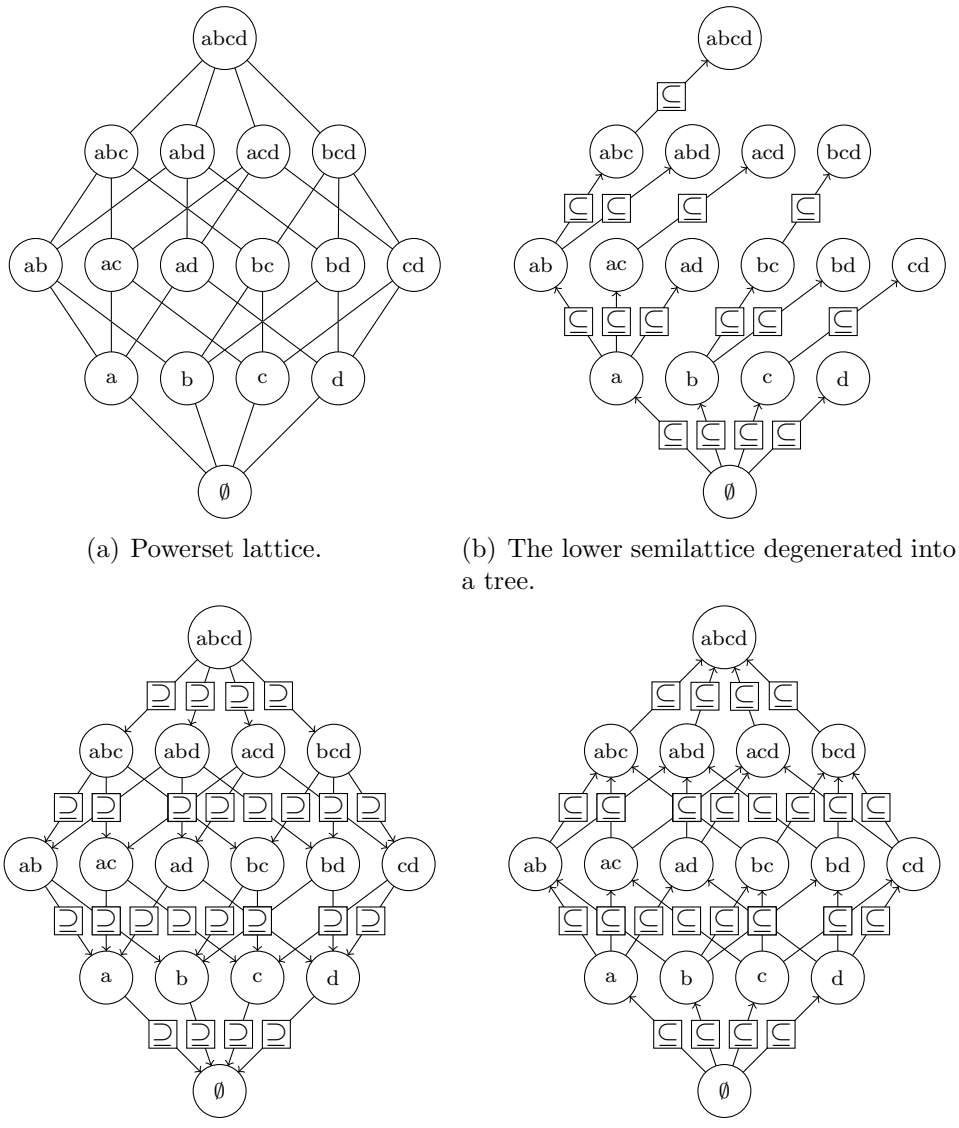
can be exploited in order to skip part of the *lattice* (see for example Section 3.2.1).

The *bottom-up* approach traverses the lattice from the bottom up starting with the empty set $\emptyset$, i.e. a single traversal path is based on the corresponding lower semilattice (see Figure 3.1(d)). It gradually adds items to the current itemset to traverse the lattice, thus it literally "grows" *descriptive patterns*. The *top-down* approach traverses the lattice from the top down starting with the full set I, i.e. a single traversal path is based on the corresponding upper semilattice (see Figure 3.1(c)). It gradually removes items from the current itemset to traverse the tree, thus it literally "shrinks" *descriptive patterns*. This section focuses on atomic modifications of the current itemsets, i.e. adding or removing one item at a time. Note that traversing the lattice is equivalent to atomically adding or removing a single item at a time.

Note 3.1 (Bottom-Up and Top-Down in Frequent Itemset Mining)**.** *In frequent itemset mining (cf. [4]) and related tasks both methods, bottom-up as well as top-down, are being applied. Apriori (see [4]) and FP-Growth (see [33]) based methods are usually working with bottom-up search. But there are algorithms that pursuit top-down methods or modifications and hybrids (cf. [45, 58]).*

Enumerating each *descriptive pattern* only once by traversing a semilattice is equivalent to only using a single incoming edge for each *descriptive pattern*. Thus the *semilattice* to traverse degenerates into a tree. Figure 3.1(b) shows how a lower semilattice degenerates into a tree. Figure 3.2(a) shows the corresponding tree based on the lower semilattice featuring a *bottom-up* enumeration and Figure 3.2(b) shows the same for the upper semilattice featuring the *top-down* approach. Figure 3.3 shows which items are added or removed independent of the traversal approach. To ensure the degeneration into a tree the notion of *modification sets* (see Definition 2.10) can be used.

An algorithm based on *modification sets* is shown by Listing 3.3. It uses the items from the *modification set* E_P of the current *descriptive pattern* π_P to atomically extend or reduce the set of items P to enumerate all *descriptive patterns*. If the respective *modification sets* are not chosen wisely, repetition is introduced. The worst case is for all *modification sets* E_P to be the full set of items (i.e. $E_P = I \setminus P$ (*bottom-up*) and $E_P = P$ (*top-down*)). Then every possible path defined by a semilattice is traversed introducing a large amount of replication. In Figure 3.1(d) for example every *descriptive pattern* at level one would be enumerated once, the *descriptive patterns* at level two twice, at level three thrice, etc. A more subtle scenario for a repetitive enumeration is given by Example 3.1. However, reducing the *modification*



(a) Powerset lattice.

(b) The lower semilattice degenerated into a tree.

(c) The upper semilattice.

(d) The lower semilattice.

Figure 3.1: The powerset lattice based on a set of items $I = \{a, b, c, d\}$ is an aggregate of an upper and a lower semilattice. The semilattices can degenerate into trees.

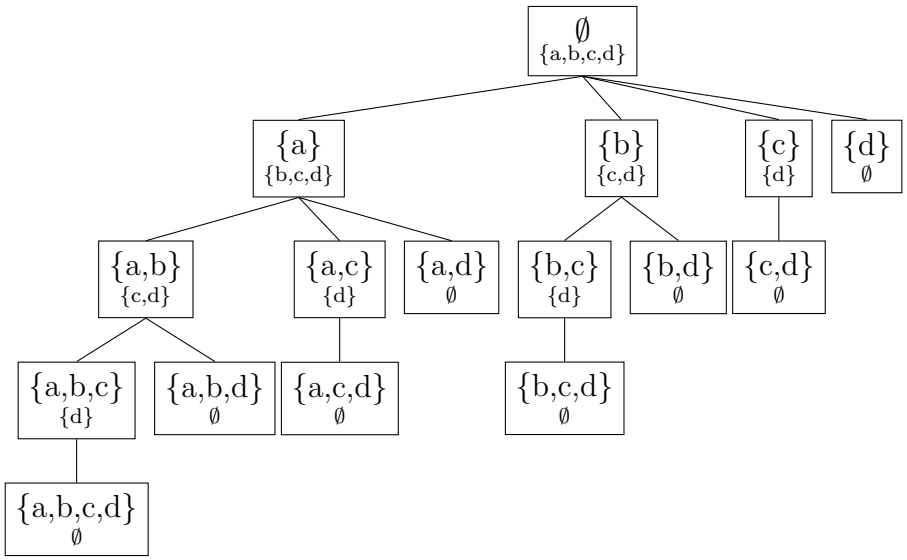
Listing 3.3: *Bottom-up descriptive pattern discovery* based on *modification sets*. The commented-out lines refer to the *top-down* variant.

```
1  VARIABLES:
2     Π_valid ← ∅
3     C_ΠI  // set implicitly
4     S  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_ΠI, S):
7     call patternDiscovery_rec(∅, E_∅)
8     // call patternDiscovery_rec(I, E_I) /* top-down */
9     return Π_valid
10
11 PROCEDURE patternDiscovery_rec(P, E_P):
12    forall i ∈ E_P do
13       P' ← P ∪ {i}
14       // P' ← P \ {i} /* (top-down) */
15
16       if (π_P', S) satisfies all c_ΠI ∈ C_ΠI do
17          Π_valid ← Π_valid ∪ {π_P'}
18       endif
19
20       call patternDiscovery_rec(P', E_P')
21    endfor
```

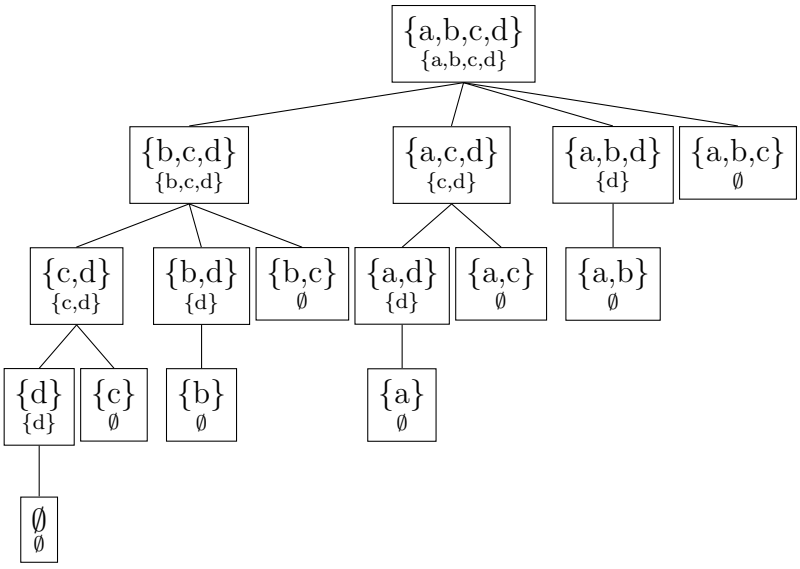
sets for each *descriptive pattern* can avoid repetition while still enumerating all *descriptive patterns*. The algorithm from Listing 3.3 would turn into a *depth-first* tree traversal algorithm (*depth-first* and *breadth-first* methods of traversing trees are reviewed in detail in Section 3.1.1). Figure 3.2(a) and 3.2(b) show examples for correctly chosen *modification sets*.

Example 3.1 (Repetition)**.**
Let $I = \{a, b, c\}$ *be a set of items. Let* $E_{\{a\}} = \{b, c\}$*. Thus* $\{a\}$ *can be grown into* $\{a, b\}$ *and* $\{a, c\}$*. If* $E_{\{a,b\}}$ *and* $E_{\{a,c\}}$ *are not further limited then* $E_{\{a,b\}} = \{c\}$ *and* $E_{\{a,c\}} = \{b\}$*. As a result* $\{a, b\}$ *can be grown into* $\{a, b, c\}$ *and* $\{a, c\}$ *can be grown into* $\{a, c, b\}$*. Thus the lattice of descriptive patterns does not degenerate into a tree and a enumeration based on the algorithm specified by Listing 3.3 would lead to a repetition during enumeration. Either* $E_{\{a,b\}}$ *and* $E_{\{a,c\}}$ *would need to be further limited and set to* $\emptyset$ *in this case.*

Note that algorithm 3.3 does not specify a method for calculating *modification sets*. As *modification sets* are solely dependent on the associated *descriptive pattern* there are many ways to choose them avoiding repetition. One way of choosing a *modification set* $E_{P'}$ for a *descriptive pattern* $\pi_{P'}$ is to introduce a dependency on the *modification set* E_P of its parent *descriptive pattern* π_P. For this purpose an order is imposed on each individual *modification set*. Such an order is called *item order* o_P^I and is introduced by Definition 3.1. Note that an order o_P^I is dependent on the *descriptive pattern*



(a) Bottom-up.



(b) Top-down.

Figure 3.2: *Bottom-up* and *top-down* enumeration of subsets of $I = \{a, b, c, d\}$ with a reversed lexical *item order* o_P^I $(a > b > \ldots)$ for every itemset $P \subseteq I$. The larger script indicates the current itemset P, the smaller script indicates the respective *modification set* E_P.

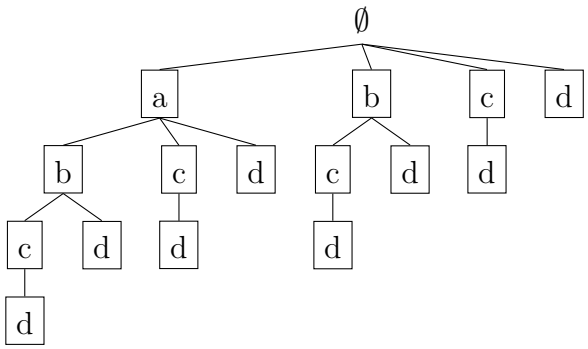


Figure 3.3: Each node represents the item added (*bottom-up*) or removed (*top-down*) from the parent *descriptive pattern* in order to create the given *descriptive pattern*. This figure is based on Figure 3.2, thus featuring the items $I = \{a, b, c, d\}$. Independently of the *descriptive patterns* the *item order* o_P^I is the reversed lexical order ($a > b > \ldots$).

π_P.

Definition 3.1 (Item Order)**.**
*Let I be a set of items and $P \subseteq I$. An **item order** o_P^I is a bijective function*

$$o_P^I : E_P \to \{1, \ldots, |E_P|\}$$

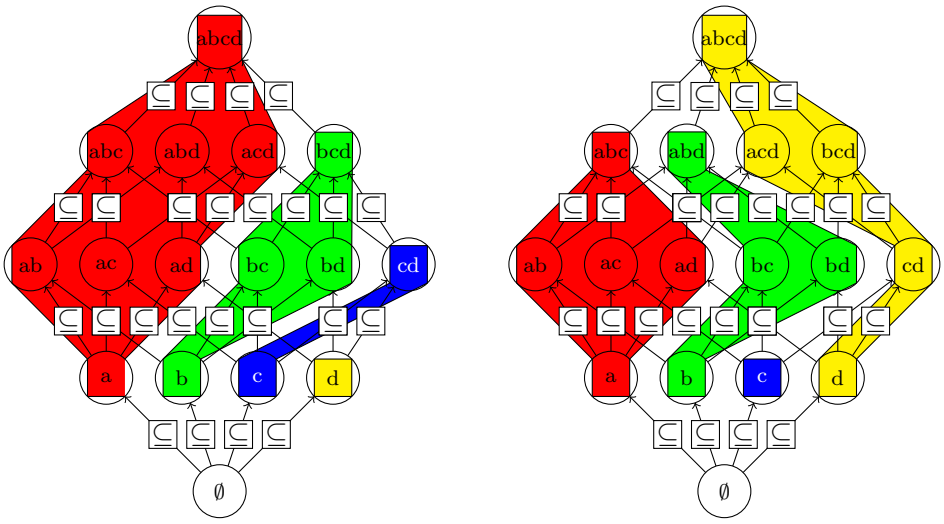
This function imposes an order on the modification set E_P of P.

Based on the atomic modifications implied by tree traversal, the *descriptive pattern* π_P is modified by adding or removing an item $i \in E_P$ to yield the *descriptive pattern* $\pi_{P'}$, i.e. $P' = P \cup \{i\}$. The *modification set* $E_{P\cup\{i\}}$ assigned to $P\cup\{i\}$ can be limited to items $j \in E_P$ smaller than i by the order imposed by an *item order* o_P^I, i.e. $o_P^I(j) < o_P^I(i)$. Thus the *modification set* $E_{P\cup\{i\}}$ can be calculated as follows:

$$E_{P\cup\{i\}} = \{j \mid j \in E_P \wedge o_P^I(j) < o_P^I(i)\} \tag{3.1}$$

The same holds for shrinking and the corresponding *modification set* $E_{P\setminus\{i\}}$.

Choosing *modification sets* in such a way ensures the degeneration of the *descriptive pattern* lattice into a tree, thus no repetition will occur during the enumeration. Based on a *descriptive pattern* π_P and its modification set E_P, this method limits the modifications $\pi_{P'}$ traversed starting at π_P to those from the associated set of modifications $E_P^{+/-}$ (extensions E_P^+ or reductions E_P^-, see Definition 2.10). In other words a prefix structure is induced. Given



(a) Item order based. (b) Arbitrary.

Figure 3.4: Powerset lower semilattices illustrating the modifications $\pi_{P'}$ traversed starting at the *descriptive patterns* associated with a single item (i.e. $\pi_{\{i\}}$ with $i \in I$), once for a *modification set* generation based on *item orders* and once for an arbitrary way of choosing *modification sets* that cannot be derived by using *item orders*.

the *bottom-up* traversal approach this means, given an item $i \in E_P$, any extension $\pi_{P'}$ based on the *descriptive pattern* $\pi_{P \cup \{i\}}$ and its *modification set* $E_{P \cup \{i\}}$ (i.e. $P' \in E^{+/-}_{P \cup \{i\}}$) does not contain greater items $j \in E_P$ (i.e. $j \notin P'$) with respect to the corresponding *item order* o^I_P, i.e. $o^I_P(i) < o^I_P(j)$. Figure 3.4 shows the modifications $\pi_{P'}$ traversed starting at the *descriptive patterns* associated with a single item (i.e. $\pi_{\{i\}}$ with $i \in I$), once for a *modification set* generation based on *item orders* and once for an arbitrary way of choosing *modification sets* that cannot be derived by using *item orders*.

Based on the algorithm from Listing 3.3 an adjusted version is given by Listing 3.4 using *item orders* to generate *modification stets*. For clarity, both, Figure 3.2(a) and Figure 3.2(b), apply a reversed lexical order o^I_P ($a > b > \ldots$) on the *modification set* E_P independent of the *descriptive pattern* π_P. Note again, that in general the orders can be different for each *descriptive pattern* without yielding repetition. Figure 3.5 shows how arbitrary *item orders* dependent on individual *descriptive patterns* can influence the search.

Listing 3.4: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders*. The commented-out lines refer to the *top-down* variant.

```
1  VARIABLES:
2     Π_valid ← ∅
3     S  // set implicitly
4     C_Π_I  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_Π_I, S):
7     call patternDiscovery_rec(∅, E_∅)
8     // call patternDiscovery_rec(I, ∅)
9     return Π_valid
10
11 PROCEDURE patternDiscovery_rec(P, E_P):
12    forall i ∈ E_P do
13
14       P' ← P ∪ {i}
15       // P' ← P \ {i} /* (top-down) */
16
17       if (π_P', S) satisfies all c_Π_I ∈ C_Π_I do
18          Π_valid ← Π_valid ∪ {π_P'}
19       endif
20
21       E_P' ← {j | j ∈ E_P ∧ o_P^I(j) < o_P^I(i)}  // modification set generation
22       call patternDiscovery_rec(P', E_P')
23    endfor
```

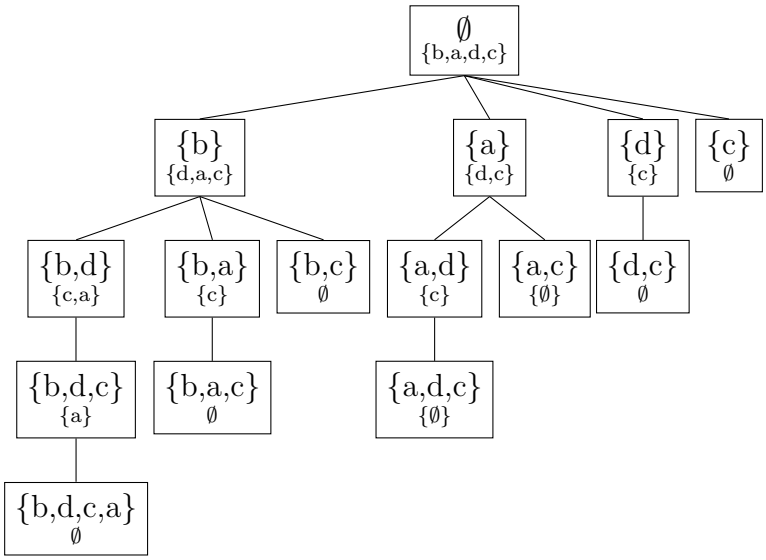


Figure 3.5: *Bottom-up* and *top-down* enumeration of subsets of $I = \{a, b, c, d\}$ with an individual *item order* for every itemset $P \subseteq I$. The larger script indicates the current itemset P, the smaller script indicates the respective *modification set* E_P.

Enumeration Order

By degenerating a semilattice of *descriptive patterns* into a tree, the enumeration of *descriptive patterns* without repetition can be viewed as traversing a tree. When enumerating all *descriptive patterns* $\pi_P \in \Pi_I$ during *pattern discovery* the resulting set of valid *descriptive patterns* $\Pi_{I,valid}$ is independent of the order in which the *patterns* are enumerated. In other scenarios this order can make a difference. For example in the case of a generative *solution discovery* (see Section 3.1.2) the *descriptive patterns* are not added to the set of valid *descriptive patterns* $\Pi_{I,valid}$, but to a solution directly. The final solution can differ based on the order *patterns* were enumerated. Additionally *dynamic constraints* can be present, which evaluate *patterns* dependent on *patterns* that were already found (see 3.2.2). In those cases, the order of adding *descriptive patterns* to the respective set matters. Adding *descriptive patterns* to the respective set is called to "**output**" that *pattern*. The following subsections review two general tree traversal methods yielding different output orders and examine how *branching orders* influence the output order of *descriptive patterns*.

Depth-First and Breadth-First Given *bottom-up* or *top-down* search there are two easily distinguishable ways to traverse the associated search tree (see Figure 3.2(a) and Figure 3.2(b) for examples on search trees):

- the *depth-first* and
- the *breadth-first* approach.

To keep things simple and as both methods can be applied to any tree, this section will focus on trees based on *bottom-up* search, without loss of generality.

The *depth-first* traversal method prioritizes modifying *descriptive patterns* recursively before considering alternative atomic *modifications* of a *descriptive pattern*. As soon as a *descriptive pattern* with an empty *modification set* is reached, it backtracks to the last *descriptive pattern* with items in the respective *modification set* that were not used for modification yet, and repeats the procedure. The enumerated *descriptive patterns* can either be returned before (*pre-order*) or after (*post-order*) all of their modifications have been generated. The former means, that, along a modification path, shorter *descriptive patterns* associated with less items are returned first, while the latter returns longer *descriptive patterns* first. For illustration a simplified version of the *depth-first* traversal algorithm based on tree nodes is given by Listing 3.5. The example algorithm for *bottom-up* enumeration of *descriptive*

Listing 3.5: *Depth-first* tree traversal.

```
1  PROCEDURE visit_depthFirst(Node n):
2
3      call ouput_pre-order(n)
4
5      forall c ∈ n.children do
6          call visit_depthFirst(c)
7      endfor
8
9      call output_post-order(n)
```

Listing 3.6: *Breadth-first* tree traversal.

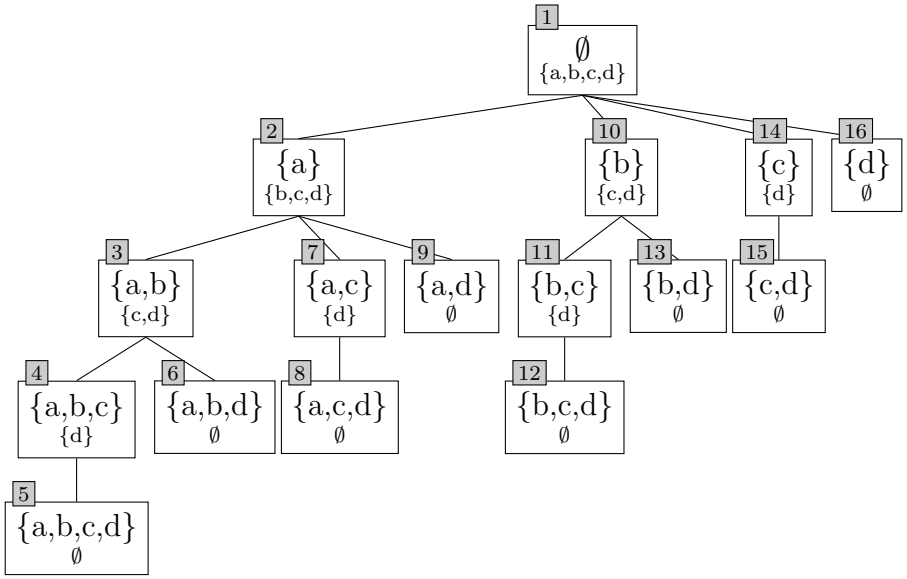
```
1  PROCEDURE visit_breadthFirst():
2      N ← {rootNode}
3
4      while N ≠ ∅ do
5        N' ← N
6        N ← ∅
7
8        forall n ∈ N' do
9          forall c ∈ n.children do
10           call output(c)
11           N ← N ∪ {c}
12         endfor
13       endfor
14     endwhile
```

patterns creating *modification sets* using *item orders* given by Listing 3.4 uses a *depth-first* approach based on *pre-order*. An example of enumerations based on *depth-first* traversal is given by Figure 3.6.

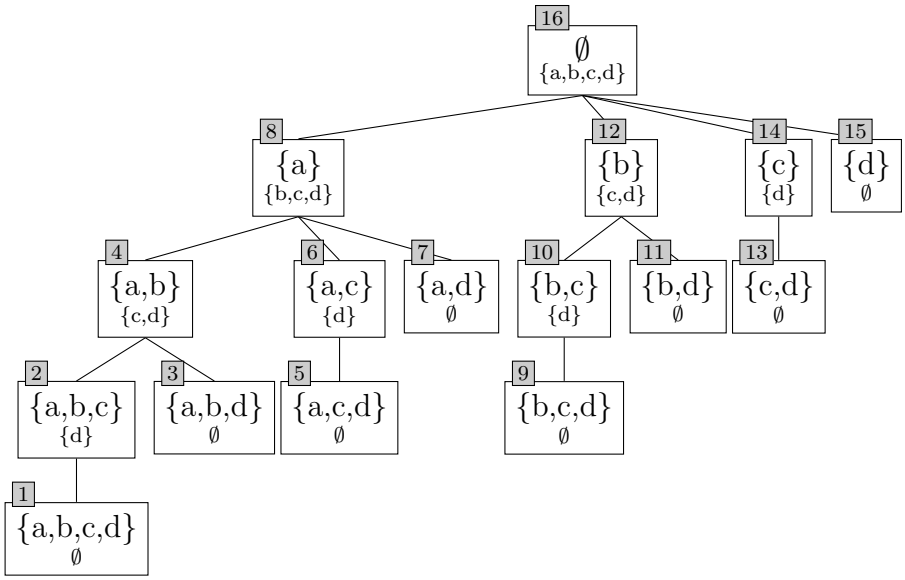
The *breadth-first* approach visits all nodes at a certain depth k, before visiting any node of depth $k+1$. For a *bottom-up* itemset enumeration this means that all *descriptive patterns* associated with k items (or a k-**itemset**) are enumerated before any *descriptive patterns* associated with $k+1$-*itemset*. A simple algorithm for *breadth-first* tree traversal is given by Listing 3.6. An example of the order of output is given by Figure 3.7(a).

Note 3.2 (Depth-First and Breadth-First in Frequent Pattern Mining)**.** *The Apriori based algorithms (cf. [3]) differ from FP-Growth based algorithms (cf. [33]) in their enumeration order. While both are based on a bottom-up approach, the Apriori based algorithms generally apply breadth-first methods while FP-Growth based algorithms mainly work in a depth-first manner.*

A hybrid of *breadth-first* and *depth-first* based on tree traversal is given by algorithm 3.7. The underlying traversal method is a *depth-first* approach

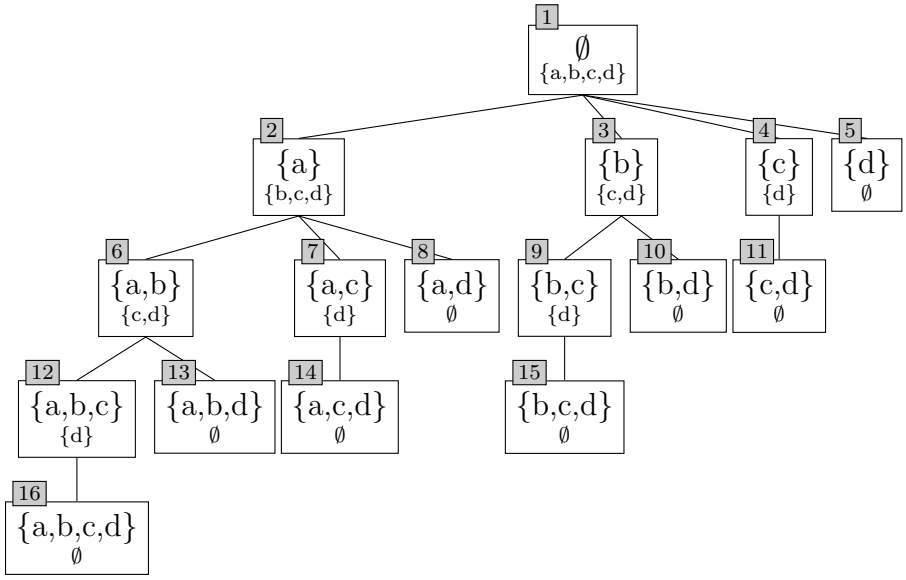


(a) Pre-Order.

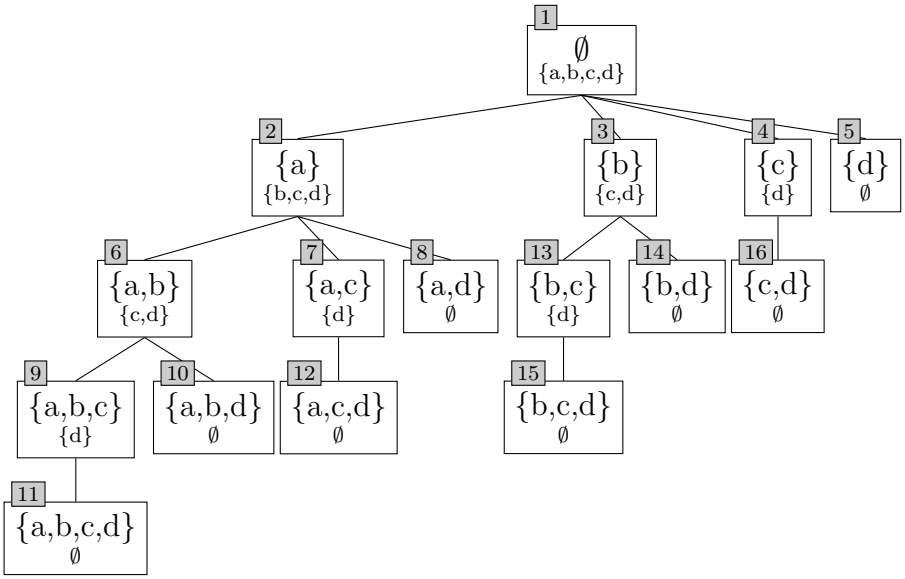


(b) Post-Order.

Figure 3.6: *Depth-first* enumeration of itemsets for the *bottom-up* search tree from Figure 3.2(a) using the lexical order ($a < b < \ldots$) on items to impose an order on visiting branches (smaller items are used for branching first).



(a) Breadth -First.



(b) Hybrid.

Figure 3.7: *Breadth-First* and *hybrid* enumeration of itemsets for the *bottom-up* search tree from Figure 3.2(a) using the lexical order ($a < b < \ldots$) on items to impose an order on visiting branches (smaller items are used for branching first).

Listing 3.7: *Depth-first* tree traversal with early node output.

```
1 PROCEDURE visit_early(Noden):
2
3     forall c ∈ n.children do
4         call output(c)
5     endfor
6
7     forall c ∈ n.children do
8         call visit_early(c)
9     endfor
```

based on *pre-order*, yet given a *descriptive pattern*, this algorithm outputs all atomic modifications before recursively modifying any of them. When using *modification sets* and given an itemset P, this corresponds outputting all *descriptive patterns* $P \cup \{i\}$, where $i \in E_P$, before recursively extending an itemset $P \cup \{i\}$ by $j \in E_{P\cup\{i\}}$. The *depth-first* algorithm from Listing 3.4 is modified accordingly by Listing 3.7, yielding an *bottom-up depth-first* algorithm, that adds *descriptive patterns* "early". An example of the output order is given by Figure 3.7(b).

Branching Order Besides the *depth-first* and *breadth-first* tree traversal methods and respective hybrids as introduced by Section 3.1.1 the *branching order* also influences the output order of *descriptive patterns*. Until now the *branching order* was not explicitly specified. The notion of *branching orders* is formally introduced by Definition 3.2. The order of modifying a *descriptive pattern* π_P using items $i \in E_P$ from the respective *modification set* is based on the respective *branching order* o_P^B. Given two items $i, j \in E_P$, i is used for modification before j, if the *branching order* assigns a lower number to i, i.e. $o_P^B(i) < o_P^B(j)$. This is illustrated by Figure 3.8.

Figure 3.9 shows a search tree using different *branching orders* for individual *descriptive patterns* based on a *bottom-up*, *depth-first* and *pre-ordered* enumeration approach contrasting the global lexical *branching order* from Figure 3.6(a). Listing 3.9 shows a modified version of the *bottom-up*, *depth-first* and *pre-ordered* algorithm from Listing 3.4 using *branching orders* besides *item orders*.

Definition 3.2 (Branching Order)**.**
Let I be a set of items and $P \subseteq I$. An ***item order*** *o_P^B is a bijective function*

$$o_P^B : E_P \to \{1, \ldots, |E_P|\}$$

This function imposes an order on a modification set E_P of P.

Listing 3.8: *Bottom-up*, *depth-first* and *pre-ordered descriptive pattern discovery*, based on *modification sets*, generated using *item orders*, adding *descriptive patterns* early to the result.

```
1  VARIABLES:
2     R ← ∅  // result
3     S  // set implicitly
4     C_ΠI  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_ΠI, S):
7     call patternDiscovery_rec(∅, I)
8     return R
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11    forall i ∈ E_P do
12       P' ← P ∪ {i}
13       if (π_P', S) satisfies all c_ΠI ∈ C_ΠI do
14          R ← R ∪ {π_P'}
15       endif
16    endfor
17
18    forall i ∈ E_P do
19       P' ← P ∪ {i}
20       E_P' ← {j | j ∈ E_P ∧ o^I_P(j) < o^I_P(i)}  // modification set generation
21       call patternDiscovery_rec(P', E_P')
22    endfor
```

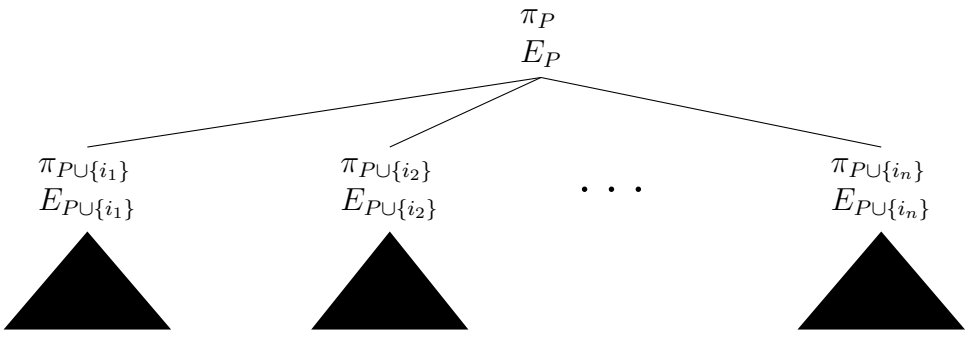


Figure 3.8: *Branching order* of the *descriptive pattern* π_P from left to right, where $o^B_P(i_j) = j$, thus $i_1 < i_2 < \ldots$ (smaller items are used for branching first).

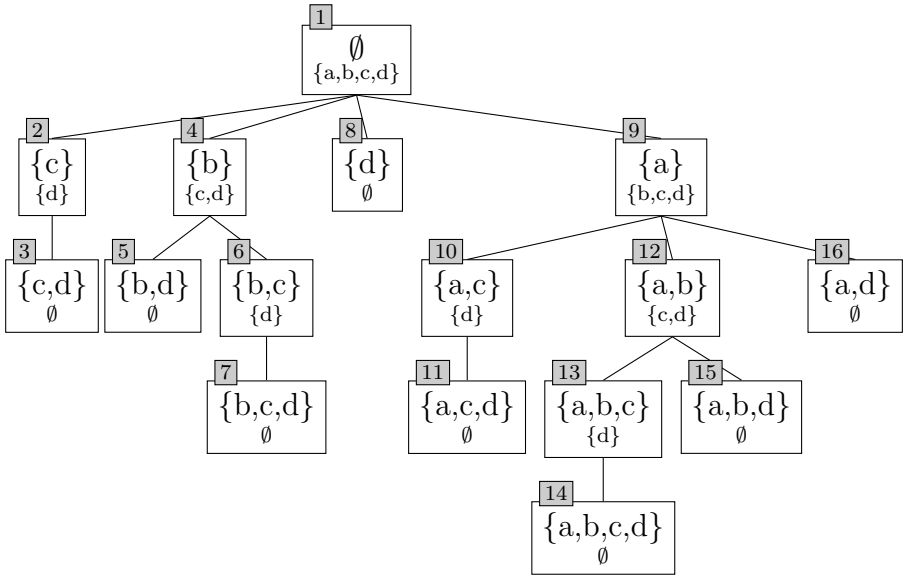


Figure 3.9: Enumeration order based on *bottom-up*, *depth-first* and *pre-ordered* search for *descriptive patterns* equivalently to Figure 3.6(a). In contrast to Figure 3.6(a) which uses a global lexical *branching order*, i.e. $a < b < \ldots$, this figure uses arbitrary *branching orders* dependent on each individual *descriptive pattern*.

Listing 3.9: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders* and *branching orders*

```
1  VARIABLES:
2     R ← ∅
3     S  // set implicitly
4     C_{Π_I}  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_{Π_I}, S):
7     call patternDiscovery_rec(∅, I)
8     return R
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    E_P[] ← sort(E_P, o_P^B)  // sort ascending by branching order
13    forall i ∈ E_P[] do
14       P' ← P ∪ {i}
15
16       if (π_{P'}, S) satisfies all c_{Π_I} ∈ C_{Π_I} do
17          R ← R ∪ {π_{P'}}
18       endif
19
20       E_{P'} ← {j | j ∈ E_P ∧ o_P^I(j) < o_P^I(i)}  // modification set generation
21       call patternDiscovery_rec(P', E_{P'})
22    endfor
```

Listing 3.10: Solution discovery: a selective approach.

```
1 PROCEDURE solutionDiscovery(Π_valid, C_R, S):
2     X = ∅ // solutions
3     forall R ∈ 2^Π_valid do
4         if (R, S) statisfies all c_R ∈ C_R do
5             X = X ∪ R
6         endif
7     endfor
8     return X
```

Note 3.3 (Branching Set)**.**
Note that a branching set can be separated as a subset of the modification set. The branching set further limits the branches that will be explored, i.e. the items that are used for modification of the current descriptive pattern. This only limits the immediate modification of a descriptive pattern, but does not influence the modification sets of the resulting descriptive patterns in contrast to limiting the current modification set.

3.1.2 Solution Discovery

The vanilla approach to *solution discovery* is the same as for *pattern discovery*: enumerating all possible subsets $R \subseteq \Pi_{\text{valid}}$ of valid patterns Π_{valid} and selecting only solutions, i.e. subsets R that pass all *result constraints* $c_R \in C_R$. The respective algorithm is given by Listing 3.10.

The requirement for this approach is to have access to all valid *patterns* $\pi \in \Pi$. In *descriptive pattern mining* the amount of possibly valid *descriptive patterns* $\pi_P \in \Pi_{I,valid}$ grows exponentially with respect to the size $|I|$ of the associated set of items I. Available space can easily be exhausted. Additionally the set of possible solutions grows exponentially with respect to the set of valid *patterns*. Thus the process of applying *pattern discovery* and *solution discovery* sequentially is often not feasible.

To avoid this problem a solution can be iteratively generated during *pattern discovery* instead of being selected in a second *solution discovery* step, i.e. a solution is discovered using a *generative* approach instead of a *selective* one. While one can imagine to build more than one solution during *pattern discovery*, the remainder of this work will assume that only one such solution is built.

The notion of subsequently building a solution is formalized by an *append function* α introduced by Definition 3.3. Listing 3.11 shows a modified version of the algorithm given by Listing 3.9 using an *append function* to generate a solution instead of returning all valid *descriptive patterns*. Note that not all

result constraints allow for a generative approach using an *append function* α, without storing all or at least a subset of all found *patterns*. Example 3.2 illustrates as much.

Definition 3.3 (Append Function)**.**
Let $M = ((D, \Pi, C_\Pi, C_R), S)$ be a pattern mining instance. Let

$$\Pi_{valid} = \{\pi \mid \pi \in \Pi \wedge \forall c_\Pi \in C_\Pi : c_\Pi(\pi) = 1\}$$

be the set of all valid patterns. Let furthermore a bijective function

$$o^\Pi : \Pi_{valid} \to \{1, \ldots, |\Pi_{valid}|\}$$

*be called a **pattern order** on the patterns Π_{valid}. If a solution to the pattern mining instance M exists, then an **append function** α is a function*

$$\alpha : \Pi \times \Omega_D \times 2^\Pi \to 2^\Pi$$

Let π_i denote the pattern π with $o^\Pi(\pi) = i$ and let

- $R^0 = \emptyset$ *and*
- $R^{i+1} = \alpha(\pi_{i+1}, S, R^i)$

The append function α must be defined such that $R^{|\Pi_{valid}|}$ is a solution to the pattern mining instance M. Note that the pattern order can influence the solution.

Example 3.2 (Limitations of the Generative Approach)**.**
*Let $M = (S, \Pi, C_\Pi, C_R)$ be a pattern mining instance and let $R \subseteq \Pi$ be a solution to M found during pattern discovery. The **coverage** S_R of S by R is defined as*

$$S_R = \bigcup_{\pi \in R} \bar{\pi}(S) = \bigcup_{\pi \in R} S_\pi$$

*The **coverage ratio** q_S^R is defined as*

$$q_S^R = \frac{|S_R|}{|S|}$$

Let the result constraint $c_{coverage}$ only be satisfied, if a given set of patterns R covers the whole data record set S, that is $S_R = S$ or equivalently $q_S^R = 1$:

$$c_{coverage} : \quad 2^\Pi \times \Omega_\delta \to \{0, 1\}$$
$$(R, S) \mapsto \begin{cases} 1, \textit{if } S_R = R \\ 0, \textit{else} \end{cases}$$

Listing 3.11: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders* and *branching orders*

```
1  VARIABLES:
2     R ← ∅
3     S  // set implicitly
4     C_ΠI  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_ΠI, S):
7     call patternDiscovery_rec(∅, I)
8     return R
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    E_P[] ← sort(E_P, o_P^B)  // sort ascending by branching order
13    forall i ∈ E_P[] do
14       P' ← P ∪ {i}
15
16       if (π_P', S) satisfies all c_ΠI ∈ C_ΠI do
17          R ← α(π_P', S, R)  // α contains information about result constraints
18       endif
19
20       E_P' ← {j | j ∈ E_P ∧ o_P^I(j) < o_P^I(i)}  // modification set generation
21       call patternDiscovery_rec(P', E_P')
22    endfor
```

Let furthermore $c_{size \leq k}$ be the result constraint from Example 2.3. If

$$C_R = \{c_{coverage}, c_{size \leq 2}\}$$

then it is impossible to find an appropriate append function for C_R. For example let the set of all valid patterns be

$$\Pi_{valid} = \{\pi_1, \pi_2, \pi_3, \pi_4\}$$

and let $R' = \{\pi_1, \pi_4\}$ be a solution such that the coverage constraint is satisfied (i.e. $c_{coverage}(R') = 1$). Let the patterns π_1, π_2, π_3, π_4 be enumerated in that order. After adding π_1 and π_2 to the result set R, any given append function α must decide to either discard the new pattern or remove either π_1 or π_2 from R in favor of π_3, due to the size constraint $c_{size,2}$. Without further information or caching, it is a random choice at this point and the append function might discard π_1, thus returning a set of patterns, which is not a solution. As a result an append function as defined in Definition 3.3 for the set of result constraints $C_R = \{c_{coverage}, c_{size \leq 2}\}$ does not exist. Instead the result when using a append function in such a scenario will be a local optimum rather than a global optimum, that is if random decisions are made.

3.2 Optimizations

Considering a *descriptive pattern mining class* $\bar{M} = (D, I, \phi_I, C_{\Pi_I}, C_R)$, the set of possible *descriptive patterns* $\Pi_I = \{\pi_P | P \in 2^I\}$ grows exponentially with respect to the size $|I|$ of the set of items I. With no optimizations each *pattern* needs to be enumerated and checked during the *pattern discovery* step. Section 3.2.1 reviews methods to skip parts of the *pattern* space in *descriptive pattern mining* to reduce the amount of enumerated *descriptive patterns* and corresponding *constraint* checking.

Result constraints introduce dependencies between *patterns*. Some *patterns* exclude other *patterns* from solutions. Especially in combination with *generative solution discovery* (see Section 3.1.2) information about already found *patterns* can be used in synergy with *result patterns* to further prune the search space. This notion is elaborated in Section 3.2.2.

When checking a *pattern* π against *constraints*, it is often necessary to project π onto the given *data record set* and filter items based on the current *modification set* (see Section 3.1.1). This process is costly as *data record sets* can be large. It is possible to gain performance by recursively building compact representations of such projections. Possible data structures are reviewed in Section 3.2.3.

3.2.1 Pruning

Let $M = ((D, I, \phi_I, C_{\Pi_I}, C_R), S)$ be the *descriptive pattern mining instance* referred to throughout this section. The set of patterns $\Pi_I = \{\pi_P | P \in 2^I\}$ grows exponentially with respect to the number of items $|I|$. To make *descriptive pattern mining* feasible, optimizations need to be applied in order to avoid enumerating the search space. In this section *descriptive patterns* are enumerated based on *bottom-up*, *depth-first* and *pre-ordered descriptive pattern* enumeration as introduced in Section 3.1.1. An example of a tree containing all possible *descriptive patterns* Π_I of a *descriptive pattern mining instance* is given by Figure 3.2(a). Each node N represents a *descriptive pattern* π_{P_N} and its *modification set* E_{P_N}. Let

$$\Pi_{I,invalid} = \{\pi \mid \exists c \in C_{\Pi_I} : c(S, \pi) = 0\}$$

denote all the invalid *descriptive patterns* with respect to a *descriptive pattern mining* instance M, i.e. all *descriptive patterns* that do not satisfy some *pattern constraint* given by M, and

$$\Pi_{I,valid} = \{\pi \mid \forall c \in C_{\Pi_I} : c(S, \pi) = 1\}$$

denote the valid *descriptive patterns*, i.e. all *descriptive patterns*, that satisfy every *pattern constraint* given by M.

When traversing the tree without any optimizations (as the algorithm in Listing 3.9 does for instance) then every single *descriptive pattern* $\pi_P \in \Pi_I$ is enumerated and checked against every single *pattern constraint* $c_\Pi \in C_\Pi$. A complete search tree with invalid *descriptive patterns* $\pi_{P,invalid} \in \Pi_{I,invalid}$ depicted as nodes with a darker background color is shown by Figure 3.10(a). The other *descriptive patterns* $\pi_{P,valid} \in \Pi_{I,valid}$ were able to satisfy every *descriptive constraint* $c_\Pi \in C_\Pi$.

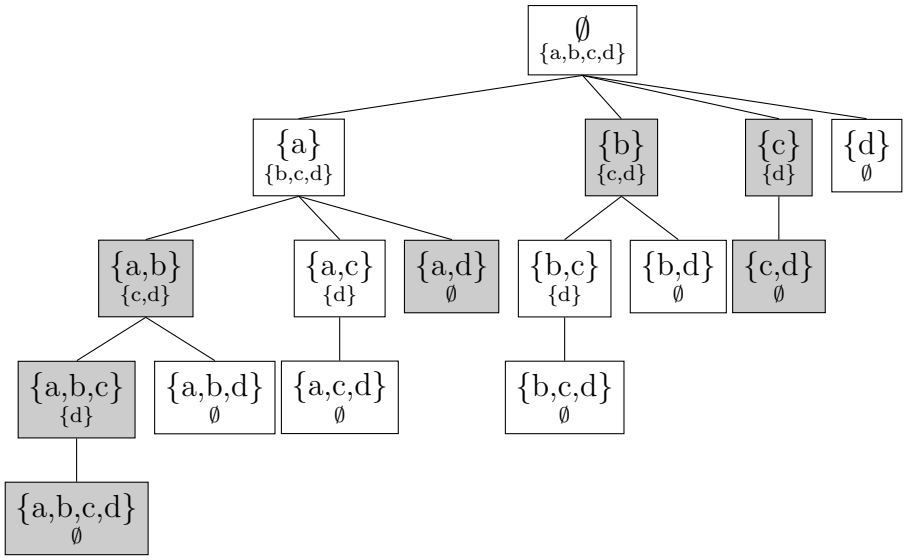
A fully optimized *bottom-up* enumeration of *descriptive patterns* $\pi_P \in \Pi_I$ is equivalent to traversing a tree only consisting of nodes representing valid *descriptive patterns* $\pi_{P,valid} \in \Pi_{I,valid}$. A fully optimized version of the tree depicted in Figure 3.10(a) is shown by 3.10(b). It only contains those nodes corresponding to valid *descriptive patterns*. To reduce a tree to only valid *patterns* or at least to reduce the amount of nodes referring to invalid *patterns*, information about constraints and how they influence each other needs to be exploited. Only recently, approaches were proposed supporting the exploitation of arbitrary constraints utilizing a constraint solver yet introducing considerable overhead (cf. [22]).

Based on a *bottom-up*, *depth-first*, *pre-ordered* search as introduced by Section 3.1.1, there are characteristic concepts to reduce the number of invalid *patterns* in a *descriptive pattern* search tree. Each concept identifies regions of the *descriptive pattern* space, that do not need to be explored. Thus the corresponding *descriptive patterns* will neither be enumerated nor checked against constraints. Some of these concepts manifest as

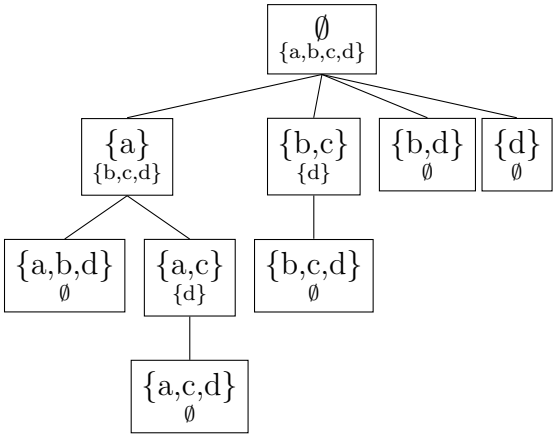
- *branch pruning*,
- *modification set pruning* or
- *direct modification*.

Given a node, *branch pruning* is a way to minimize branching. This approach is equivalent to a *branch-and-bound* search strategy. Before exploring a branch the corresponding *descriptive pattern* is examined and possibly discarded together with all its children. Figure 3.11 shows an appropriate example of the pruning effect. To be able to discard a branch, it must be guaranteed, that every *descriptive pattern* associated with a child of that branch are rendered invalid by some (not necessarily the same) *constraint*.

Example 3.3 (Branch Pruning: Minimal Price)**.**
Let $\phi_{price} : I \to \mathbb{R}^+$ be the item property introduced in Example 2.6 associating



(a) Invalid *descriptive patterns*.



(b) Tree resulting from removing invalid *descriptive patterns*.

Figure 3.10: *Descriptive pattern search trees*: one depicting invalid *descriptive patterns* as gray nodes and one showing the same tree with all the invalid *descriptive patterns* removed.

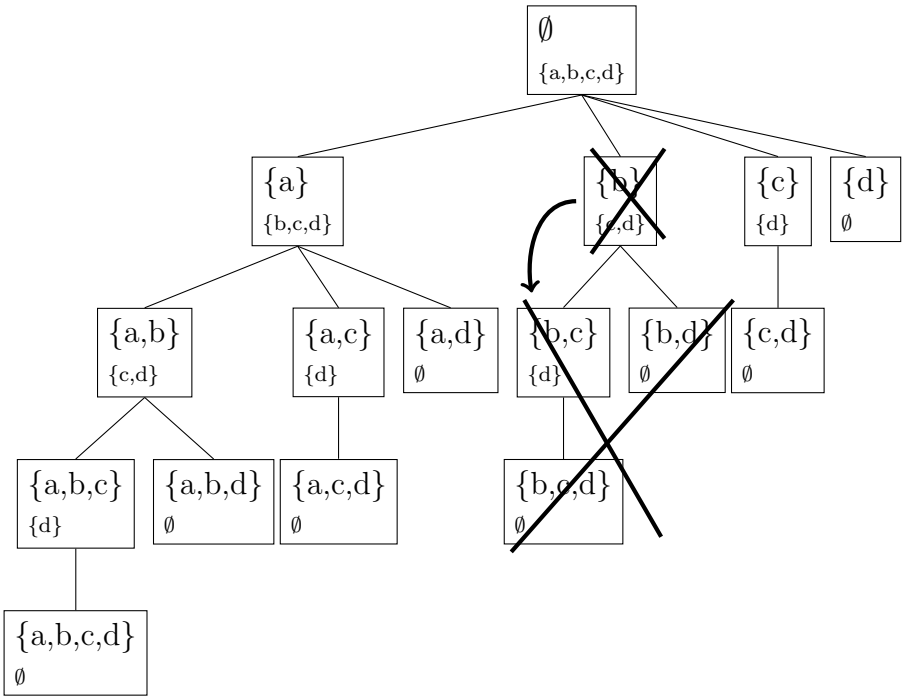


Figure 3.11: Based on the *bottom-up* search tree from Figure 3.2(a) *branch pruning* is being illustrated here. Upon branching on b the *descriptive pattern* $\{b\}$ is rendered invalid and the corresponding subtree is skipped. 3.2.1).

each item with a price. And let $c_{price \geq t}$ be the price constraint from the same example. Let

$$\phi_{price}(a) = 6$$
$$\phi_{price}(b) = 1$$
$$\phi_{price}(c) = 1$$
$$\phi_{price}(d) = 1$$

Looking at Figure 3.11 and given the constraint $c_{price \geq 7}$ starting at node $\emptyset$ the branch based on a modification by b is considered. $\pi_{\{b\}}$ is an invalid pattern $c_{price \geq 7}(\pi_{\{b\}}) = 0$, thus it is discarded. Furthermore the maximal sum of prices of any extensions of $\pi_{\{b\}}$ is also smaller than seven, i.e. $c_{price \geq 7}(\pi_{\{b,c,d\}}) = 0$. Thus the whole branch together with its children can be discarded. Note that the branches based on c and d will also be discarded.

Modification set pruning refers to the possibility to minimize the *modification set* E_P of a *descriptive pattern* π_P, thus recursively minimizing the branching possibilities of each of its extension $\pi_{P'} \in E_P^+$. The *modification set* E_P represents all items i, that are not part of the current *descriptive pattern* (i.e. $i \notin P$), but may occur in some extension $\pi'_P \in E_P^+$ (i.e. $i \in P'$). If an item can be removed from the *modification set* of the current *descriptive pattern*, then no *descriptive pattern*, being a child of the current node, can contain that item anymore. The effect of *modification set pruning* is illustrated by Figure 3.12. An example for a constraint enabling *modification set pruning* is given by Example 3.4.

Example 3.4 (Modification Set Pruning: Maximal Price)**.**
Let $\phi_{price} : I \to \mathbb{R}^+$ be the item property introduced in Example 2.6 associating each item with a price. And let $c_{price < t}$ be defined analogously to $c_{price \geq t}$ from the same example. Let

$$\phi_{price}(a) = 6$$
$$\phi_{price}(b) = 1$$
$$\phi_{price}(c) = 6$$
$$\phi_{price}(d) = 1$$

Looking at Figure 3.12 and given the constraint $c_{price < 10}$, at node $\{a\}$ the price is at 6. Adding c would raise the price sum to 12, rendering any descriptive pattern containing both a and c invalid. As a result c can be

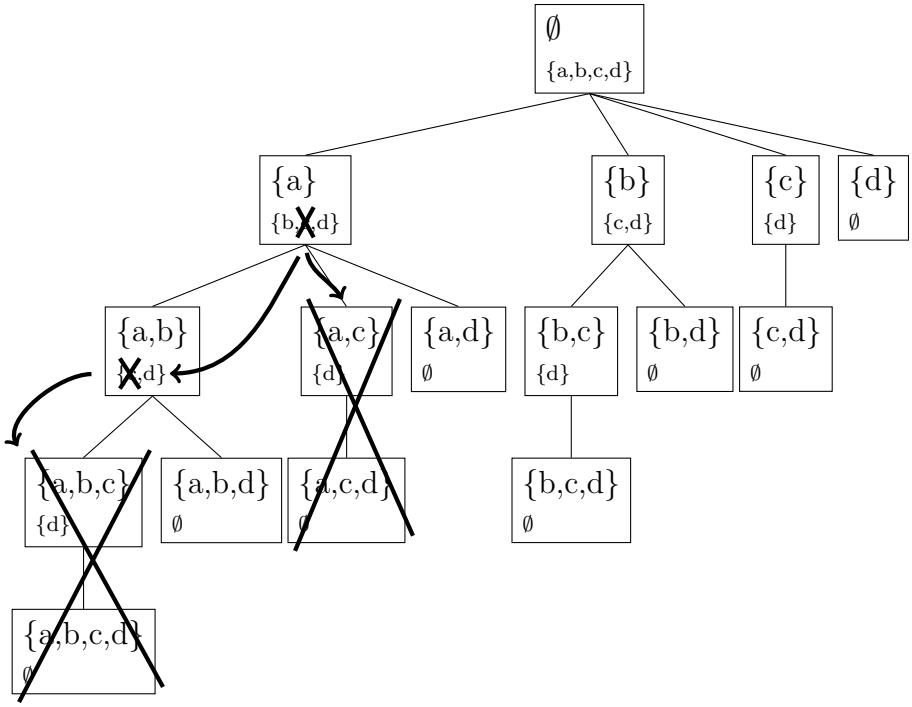


Figure 3.12: Item c is removed from the *modification set* $E_{\{a\}}$ of node $\{a\}$ resulting in the corresponding child being pruned. Item c is also removed from every child's *modification set*. Thus subtrees dependent on c are not only removed at the current node, but also at all corresponding children.

pruned from the modification set $E_{\{a\}} = \{b, c, d\}$ *resulting in the pruning as shown in the figure.*

Note 3.4 (Branch Pruning and Modification Set Pruning)**.**
Removing an item from the modification set of a descriptive pattern prunes all its extensions based on the current node, which contains that item. This includes direct children, i.e. branches. As a result modification set pruning subsumes branch pruning. Yet pruning a branch does not limit the set items used for extension by other branches. Thus it is not equivalent to modification set pruning. As mentioned in Note 3.3 a branching set can be defined alongside a modification set. See Example 3.3 for a situation were a constraint can be used for branch pruning but not for modification set pruning.

Direct modification discards all possible branches and directly extends a *descriptive pattern* by a subset of the current *modification set*. Figure 3.13 shows the effect of this pruning method and Figure 3.14 shows how the remodeled search tree would look like. Example 3.5 lists a *pattern constraint*, that can be used for *direct modification*. For *direct modification* it is necessary to derive that certain items from the *modification set* have to be part of any extension of *descriptive pattern*. Otherwise some (not necessarily the same) *constraint* would render the extension invalid.

Example 3.5 (Direct Modification: Minimum Price)**.**
This example uses the same constraint as Example 3.4. Let $\phi_{price} : I \to \mathbb{R}^+$ *be the item property introduced in Example 2.6 associating each item with a price. Let* $c_{price \geq t}$ *be the constraint from the same example. Let*

$$\phi_{price}(a) = 1$$
$$\phi_{price}(b) = 6$$
$$\phi_{price}(c) = 6$$
$$\phi_{price}(d) = 1$$

Looking at Figure 3.13 and given the constraint $C_{price \geq 12}$*, at node* $\emptyset$ *it can be concluded, that no descriptive pattern* π_P *can surpass threshold* $t \geq 12$ *without containing both, item b and item c. Thus both items can be directly added to the descriptive pattern. Figure 3.13 shows which children are accessed directly. The tree can also be rebuilt by modifying the current descriptive pattern and adjusting the corresponding modification set. The result is depicted by Figure 3.14. Note that this method is not restricted to the root node.*

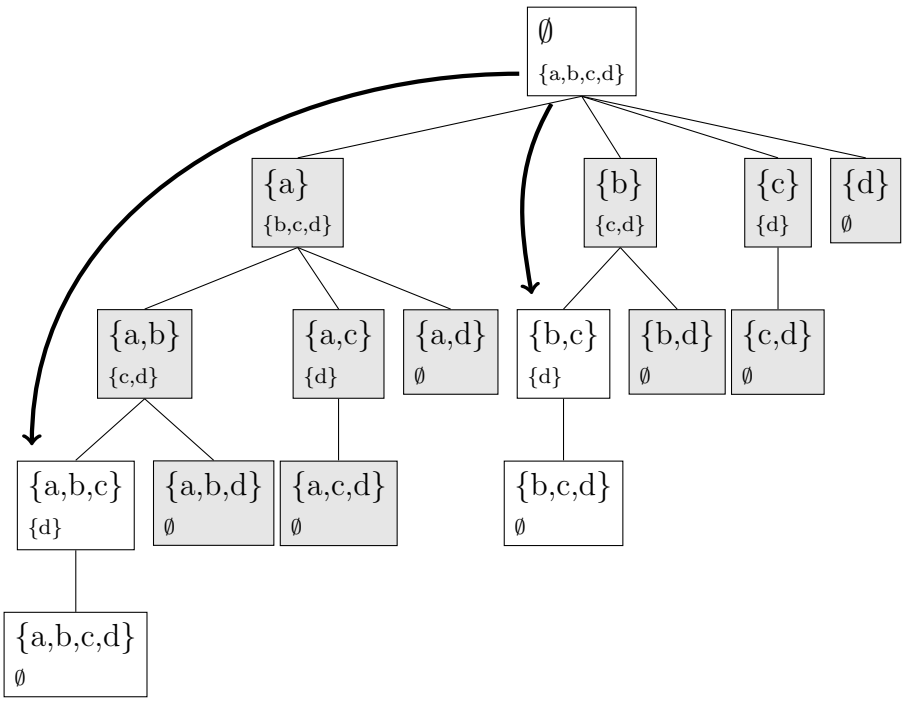


Figure 3.13: This figure illustrates *direct child access*. Instead of visiting every child along a path, child nodes further down the according subtree are accessed directly if the nodes in between only yield invalid *descriptive patterns*. Here only *descriptive patterns* π_P containing $\{b, c\} \subseteq P$ are valid. Thus extensions containing $\{b, c\}$ can be accessed directly. These extensions are $\{a, b, c\}$ and $\{b, c\}$. The white nodes are explored, the gray nodes are skipped. *Direct child access* is not restricted to starting at the root node as in this example.

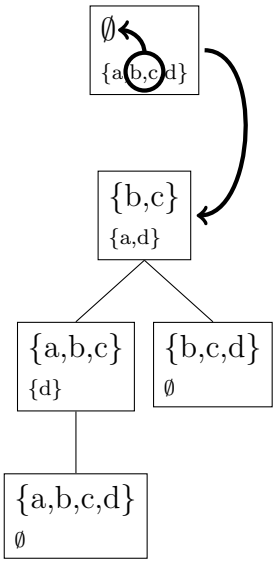


Figure 3.14: This figure shows how the search tree from Figure 3.13 is modified, when modifying the tree based on *direct child access* by adding {b,c} directly to the empty *descriptive pattern* ∅.

Note that exploiting the structure of the search as well as properties of the constraints often needs special processing besides checking *patterns* against *constraints* (see Example 3.3): the sum of all items in the *modification set* has to be calculated to derive the necessary information for skipping the branch. Information about the structure and status of the search can be utilized. While there exist approaches that support exploitation of any kind of constraint to some degree (cf. [22]), the following section will focus mainly on *anti-monotone constraints*, which can be used for *branch* and *modification set pruning* simply by checking *patterns* against *constraints*. To illustrate how synergies can be exploited, but also how extra processing is needed, is demonstrated by a property called ExAnte (cf. [12]), which uses *anti-monotone constraints* in synergy with *monotone constraints* for *modification set pruning*.

Note 3.5 (Constraint Context)**.**
By definition the evaluation of patterns by pattern constraints does not depend on search specific parameters like e.g. the modification set. If it would, different search strategies could yield different solutions, even if overall only one solution exists. Consider a descriptive pattern mining instance, where the only solution is the set of all valid descriptive patterns. The item order is not globally fixed and can be different at every node without yielding a differ-

ent overall result. It is possible to use a search strategy where the item order differs only at a single node. While the children of that node and their corresponding descriptive patterns stay the same, their modification sets differ. If a constraint would depend on modification sets, those different search strategies could yield different solutions for the same descriptive pattern mining instance.

Anti-monotone constraints

Anti-monotone constraints are *constraints* that allow to deduce that any extension $\pi_{P'}$ of a *descriptive pattern* π_P is rendered invalid if π_P itself is rendered invalid. This notion is formalized by Definition 3.4. Theorem 3.1 proofs that the *frequency constraint* introduced by Example 2.4 is an *anti-monotone constraint*.

Definition 3.4 (Anti-Monotone Constraint)**.**
Given an arbitrary data record set $S \in \Omega_D$ and a descriptive pattern π_P on D, a constraint c is called ***anti-monotone****, if*

$$\forall\ P' \subseteq P : (c(\pi_P, S) = 1) \Rightarrow (c(\pi_{P'}, S) = 1)$$

or equivalently

$$\forall\ P' \supseteq P : (c(\pi_P, S) = 0) \Rightarrow (c(\pi_{P'}, S) = 0)$$

Theorem 3.1 (Anti-Monotonicity of the Frequency Constraint)**.**
The frequency constraint $c_{freq \geq t}$ as introduced in Example 2.4 and modified to fit descriptive pattern mining is anti-monotone:

$$\begin{aligned} c_{freq \geq t} : \quad & \Pi \times \Omega_D \to \{0, 1\} \\ & (\pi_P, S') \mapsto \begin{cases} 1, \text{ if } |S'_{\pi_P}| \geq t \\ 0, \text{ else} \end{cases} \end{aligned}$$

Proof. Let $S = (R, \delta)$ be a *data record set* and let ϕ_I be an *I-item projector*. Given Definition 2.4 and 2.9, the projection S_{π_P} of a *descriptive pattern* π_P is defined

$$S_{\pi_P} = (\{i \in R \mid P \subseteq \phi_I(\delta(i))\},\ \delta)$$

then by Theorem 2.2 an extension $\pi_{P'}$ of π_P, i.e. $P \subseteq P'$, projects on a smaller *data record subset* than π_P, i.e. the frequency of the extension $\pi_{P'}$ of π_P is smaller than or equal to the frequency of π_P:

$$|S_{\pi_P}| \geq |S_{\pi_{P'}}|$$

individual	items		
1	a	c	
2	a	c	d
3	a	c	d
4	b		

Table 3.1: A transaction *data record set*, showing only the items an *individual* is associated with.

and consequently any extension $\pi_{P'}$ of π_P is rendered invalid by the frequency *constraint* $c_{freq \geq t}$, if π_P is rendered invalid:

$$(c_{freq \geq t}(\pi_P, S) = 0) \Rightarrow (c_{freq \geq t}(\pi_{P'}, S) = 0)$$

□

Note 3.6 (Anti-monotonicity and Itemset Mining)**.**
The concept of anti-monotone constraints is strongly incorporated into itemset mining as one of the first papers about itemset mining introduced the frequency constraint (cf. [4]). The frequency constraint itself is anti-monotone as shown by Theorem 3.1. Other instances of anti-monotone constraints are introduced for itemset mining (for instance by [12, 24]).

Anti-monotone constraints can be used for *branch pruning*. Based on the *bottom-up* search tree, any branch of a *descriptive pattern* π_P yields *extensions* $\pi_{P'}$ of the current *descriptive pattern*, i.e. $P \subseteq P'$. Thus if an *anti-monotone constraint* renders a *descriptive pattern* invalid, it will also render all its *extensions* invalid. Thus upon checking a branch and rendering the corresponding *descriptive pattern* invalid by a *anti-monotone constraint*, that branch can be discarded. Figure 3.15 shows the effect of a *descriptive pattern* $\pi_{\{b\}}$ being rendered invalid by an *anti-monotone constraint*. Example 3.6 gives a concrete example based on the *frequency constraint*. The algorithm shown in Listing 3.9 is extended to exploit the *anti-monotone* property by *branch pruning* and depicted by Listing 3.12.

Example 3.6 (Anti-Monotonicity: Support Threshold)**.**
Let $c_{supp \geq 2}$ be the support / frequency constraint as refined by Theorem 3.1. Consider the transaction data record set from Table 3.1.

The descriptive pattern $\pi_{\{b\}}$ is rendered invalid by $c_{supp \geq 2}$, i.e. $c_{supp \geq 2}(\pi_{\{b\}}) = 0$, because the conditional data record set based on $\pi_{\{b\}}$ is not large enough, i.e. $|S_{\pi_{\{b\}}}| < 2$. In other words the descriptive pattern $\pi_{\{b\}}$ does not pass the support threshold. Because the frequency constraint is anti-monotone, the branch based on b is discarded according to the concept of branch pruning.

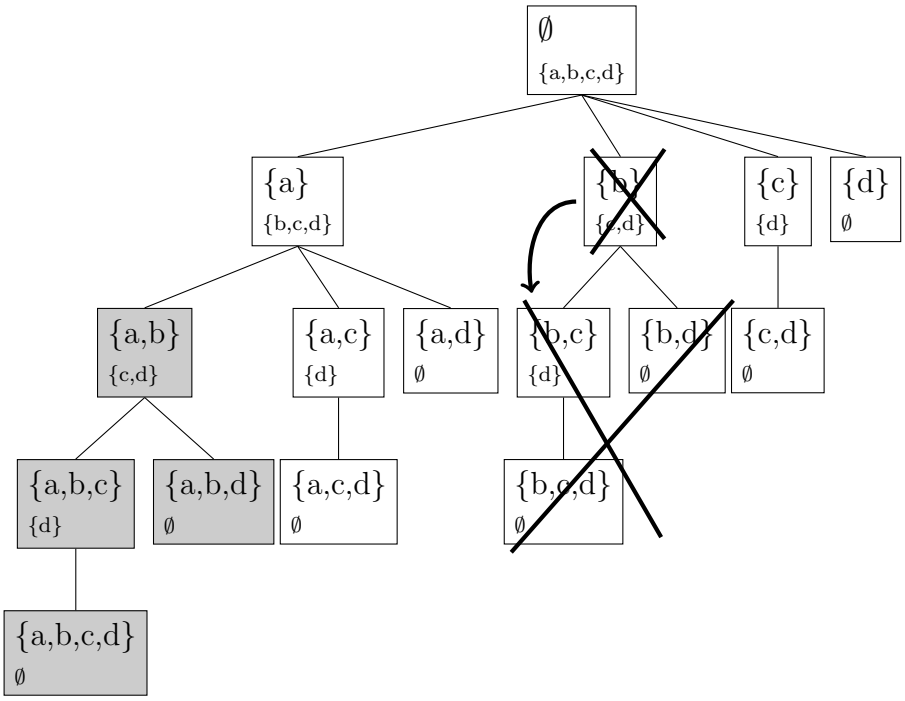


Figure 3.15: Using the *bottom-up* search tree from Figure 3.2(a), this figure illustrates *branch pruning* as used by the branch-and-bound paradigm. For example let node $\{b\}$ be pruned by an *anti-monotone constraints*, then the whole subtree dependent on $\{b\}$ can be discarded. The nodes with a gray background also represent invalid patterns π_P, if $\{b\}$ was pruned using an *anti-monotone constraint*, because $\{b\} \subseteq P$.

Listing 3.12: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders*, *branching orders* and an *append function* exploiting *anti-monotone constraints* by *branch pruning*.

```
1  VARIABLES:
2     Π_valid ← ∅
3     S  // set implicitly
4     C_Π_I  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_Π_I, S):
7     call patternDiscovery_rec(∅, I)
8     return Π_valid
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    forall i ∈ E_P do
13       P' ← P ∪ {i}
14
15       if (π_P', S) statisfies all anti-monotone c_Π_I ∈ C_Π_I do
16          if (π_P', S) statisfies all c_Π_I ∈ C_Π_I do
17             Π_valid ← Π_valid ∪ {π_P'}
18          endif
19
20          E_P' ← {j | j ∈ E_P ∧ o_P^I(j) < o_P^I(i)}  // modification set generation
21          call patternDiscovery_rec(P', E_P')
22       endif
23    endfor
```

Other descriptive patterns containing the item b will also be rendered invalid according to Theorem 2.2, which is indicated by gray nodes in the figure.

In Figure 3.15 the *descriptive pattern* $\pi_{\{b\}}$ was rendered invalid by an *anti-monotone constraint*. Thus any extension $\pi_{P'}$ of the *descriptive pattern* $\pi_{\{b\}}$ is also rendered invalid and not only those based on the corresponding *modification set* $E_{\{b\}}$, i.e. the children of $\pi_{\{b\}}$. The nodes with a gray background in Figure 3.15 represent those *descriptive patterns*. Thus the item b can be removed completely from the *modification set* $E_\emptyset$ of the current *descriptive pattern* $\pi_\emptyset$. The consequence is a reduction of any *modification set* derived from the current one. As a result a better exploitation of *anti-monotone constraints* is possible by using *modification set pruning*.

Figure 3.12 shows the resulting pruning effect. The algorithm in Listing 3.13 shows the general procedure of *modification set pruning* and the algorithm in Listing 3.14 exploits *anti-monotone constraints* by applying that procedure. Instead of checking each branch as it is traversed, the modifications based on the items from the *modification set* are checked before visiting any branch. The corresponding items are removed from the *modification set* of the current *descriptive pattern* if an *anti-monotone constraint* renders the associated modification invalid. This reduces the *modification set* and with

Listing 3.13: *Modification set pruning.*

```
1 PROCEDURE modsetPruning(P, E_P, C, S):
2   forall i ∈ E_P do
3     P' ← P ∪ {i} // generate modification
4     if (π_P', S) does not satisfy all anti-monotone constraints c ∈ C do
5       E_P ← E_P \ {i}
6     endif
7     return E_P
8   endfor
```

it any *modification set* derived from it. Note that *branch pruning* based on *anti-monotone constraints* becomes obsolete as stated by Note 3.4.

Note 3.7 (Anti-Monotone Constraints: Limits of Modification Set Pruning)**.** *Modification set pruning exploits the properties of anti-monotone constraints more than just branch pruning does. It removes extensions of a descriptive pattern which is rendered invalid. It still cannot totally prevent the generation of all extension of the pruned descriptive pattern as Figure 3.16 shows: based on the anti-monotonicity of the constraint rendering the extension* $\{b, c\}$ *invalid,* $\{a, b, c\}$ *and* $\{a, b, c, d\}$ *could also be rendered invalid immediately. Yet modification set pruning is limited to pruning those extensions that are created based on the current descriptive pattern and its modification set, i.e. it is limited to its children.*

ExAnte

[13] introduces *monotone constraints* (see Definition 3.5) as *constraints* which work the opposite way compared to *anti-monotone constraints*. If a *descriptive pattern* π_P is rendered invalid by a *monotone constraint*, then all the *descriptive patterns* $\pi_{P'}$ associated with a subset of items $P' \subseteq P$ are also rendered invalid. Thus any parent of a *descriptive pattern* in the search tree is rendered invalid, if any of its children is rendered invalid. As the *bottom-up* search grows *descriptive patterns*, i.e. creating extensions, *monotone constraints* are not directly applicable to pruning as *anti-monotone constraints* are. Figure 3.17 illustrates how *monotone constraints* influence the structure of the search space. Direct exploitation would involve keeping track of already found *descriptive patterns* rendered invalid by some (not necessarily the same) *monotone constraint*. See Example 3.7 for an example.

Definition 3.5 (Monotone Constraint)**.**
Given any data record set $S \in \Omega_D$ *and a descriptive pattern* π_P *on* D*, a pattern constraint* c *is called* ***monotone****, if*

$$\forall\ P' \supseteq P : (c(\pi_P, S) = 1) \Rightarrow (c(\pi_{P',S}) = 1)$$

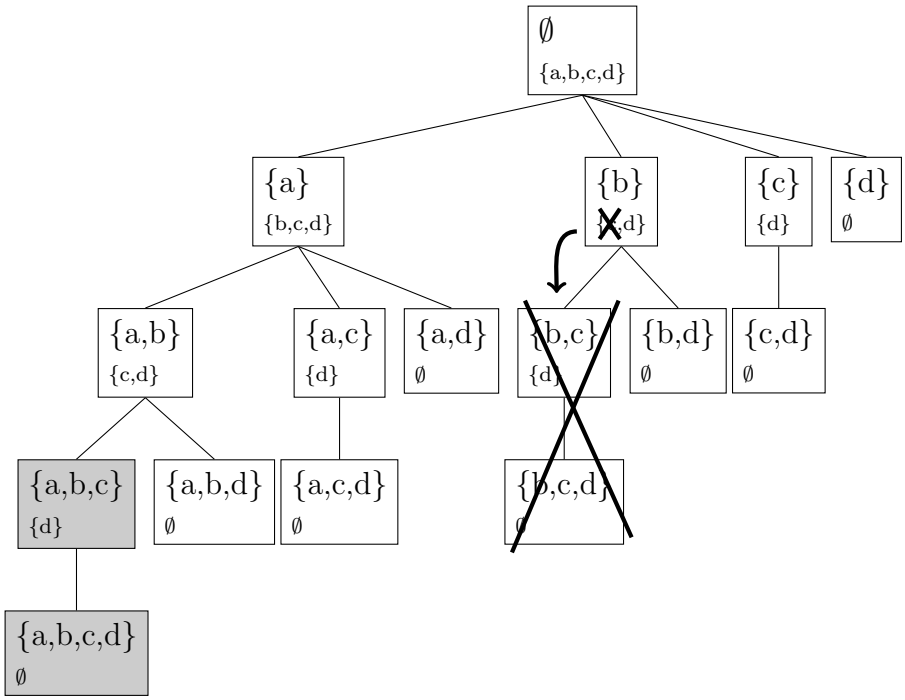


Figure 3.16: The *anti-monotonicity* property is used for *modification set pruning*. The item c is pruned from the *modification set* of the *descriptive pattern* $\{b\}$. *Modification set pruning* does not directly cover the removal of the *descriptive patterns* $\{a, b, c\}$ and $\{a, b, c, d\}$ which can be deduced to be invalid due to the *anti-monotonicity property* of the *constraint*.

Listing 3.14: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders*, *branching orders* and an *append function* exploiting *anti-monotone constraints* by *branch pruning*.

```
1  VARIABLES:
2     R ← ∅
3     S  // set implicitly
4     C_Π_I  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_Π_I, S):
7     call patternDiscovery_rec(∅, I)
8     return R
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    E_P ← modsetPruning(P, E_P, C_Π_I, S)
13
14    // branching
15    forall i ∈ E_P do
16       P' ← P ∪ {i}  // generate modification
17
18       if (π_P', S) satisfies all c_Π_I ∈ C_Π_I do //
19          R ← R ∪ {π_P'}
20       endif
21
22       E_P' ← {j | j ∈ E_P ∧ o_P^I(j) < o_P^I(i)}  // modification set generation
23       call patternDiscovery_rec(P', E_P')
24    endfor
```

or equivalently

$$\forall\ P' \subseteq P : (c(\pi_P, S) = 0) \Rightarrow (c(\pi_{P'}, S) = 0)$$

Example 3.7 (Monotone Constraint: Minimal Price)**.**
Let $\phi_{price} : I \to \mathbb{R}^+$ be the item property introduced in Example 2.6 associating each item with a price. Let $c_{price \geq t}$ be the constraint from the same example. $c_{price \geq t}$ is obviously monotone. Let furthermore

$$\phi_{price}(a) = 6$$
$$\phi_{price}(b) = 9$$
$$\phi_{price}(c) = 1$$
$$\phi_{price}(d) = 1$$

Looking at Figure 3.17 and given the constraint $c_{price \geq 9}$ the descriptive pattern $\pi_{\{a,c,d\}}$ and all its parents are pruned by $c_{price \geq 9}$. Because of the constraint's monotonicity any generalization of $\pi_{\{a,c,d\}}$ is invalid. Pattern $\pi_{\{c\}}$ is associated with the modification set $E_{\{c\}} = \{d\}$. Thus any extension of $\pi_{\{c\}}$ is

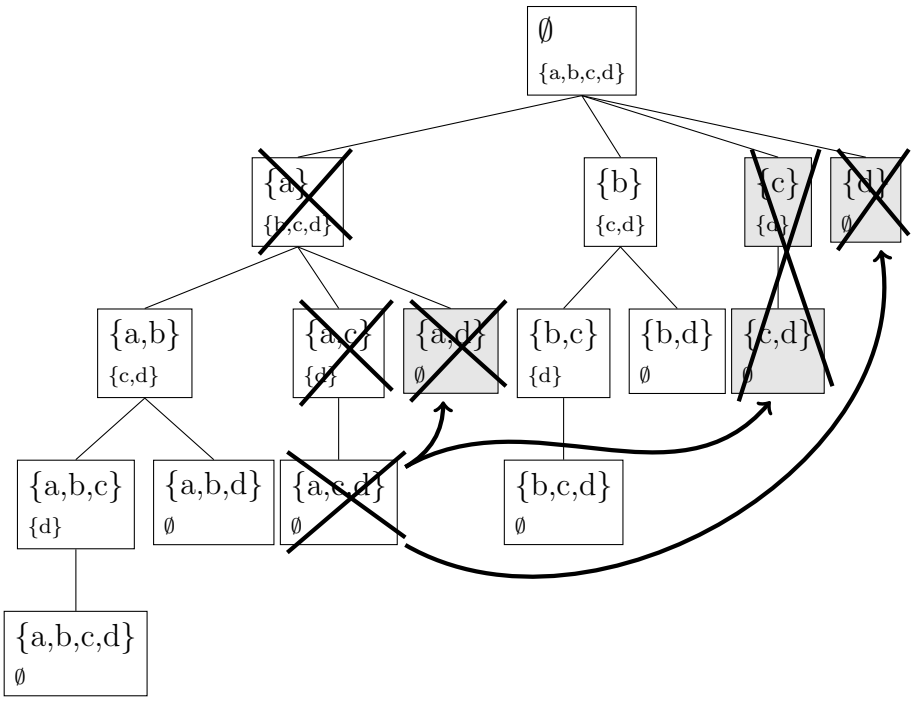


Figure 3.17: *Patterns* corresponding to $\{a\}$, $\{a, c\}$ and $\{a, c, d\}$ are found, yet rendered invalid by a *monotone constraint*. Thus any subset of $\{a, c, d\}$ is invalid, too, which results in $\{a, d\}$, $\{c\}$ and its subtree as well as $\{d\}$ being discarded.

a generalization of $\pi_{\{a,c,d\}}$. Consequently the subtree associated with $\pi_{\{c\}}$ can be skipped resulting in branch pruning based on monotone constraints. Note that this requires keeping track of patterns pruned by monotone constraints and also checking against them. Thus this type of pruning requires more information than just the results of evaluating constraints independently. See Notes 3.8 and 3.9 for further details.

Note 3.8 (Monotone Pruning: Efficiency)**.**
As mentioned before, monotone pruning needs to keeps track of descriptive patterns that were rendered invalid by monotone constraints, which costs additional space. Furthermore the pruning method introduced by Example 3.7 requires to check the current descriptive pattern against every monotonically pruned descriptive pattern to allow for pruning, which costs additional time.

Note 3.9 (Anti-Monotone and Monotone Trade-Off)**.**
Other work often mentions some kind of trade-off between monotone and anti-monotone pruning (cf. [12, 13, 23, 34, 11]). Using monotone constraints for pruning as in Example 3.7 also illustrates a trade-off. For monotone pruning it is favorable to know descriptive patterns associated with a lot of items pruned by monotone constraints. The more items, the higher the chance, that another descriptive pattern and its extensions based on its modification set are subsets of the descriptive pattern pruned by a monotone constraint. Yet anti-monotone constraints will skip all extensions of a descriptive pattern if it is rendered invalid by an anti-monotone constraint. Thus pruning anti-monotonically might skip descriptive patterns potentially pruned by monotone constraints, which could be used for monotone pruning.

Note 3.8 and 3.9 reason against exploiting *monotone constraints*. Nevertheless, there is another method to exploit a class of *monotone constraints*, i.e. *monotone description constraints* (see Definition 2.16), which has been reported to be efficient [12, 10]. The property that is exploited is called the *ExAnte property*. The *ExAnte property* is based on two other properties inherent to *anti-monotone data constraints* and *monotone description constraints*. Those properties are stated by Theorem 3.2 and 3.3 respectively.

The *ExAnte property* (see Definition 3.4, below) exploits both properties and enables *modification set pruning*.

Definition 3.6 (Data Consistent Data Record Subset)**.**
Let S be adata record set and S' data record subset, i.e. $S' \subseteq S$. S' is called ***data consists*** *(with respect to S) if and only if*

$$\forall r \in S, r' \in S' : \delta(r) = \delta(r') \Rightarrow r \in S'$$

In other words, if an individual r' associated with a data instance $\delta(r')$ is part of the data record subset S', then any individual r from the data record set associated with the same data instance $\delta(r) = \delta(r')$ is also part of the data record subset $r \in S'$.

Note that a data record subset $S'' \subseteq S'$, which is data consistent with respect to S', is also data consistent with respect to S. Also if $S'' \subseteq S'$ is data consistent with respect to S, it is also data consistent with respect to S'. As a result any conditional data record set based on a pattern is data consistent because projections are based on data instances.

Theorem 3.2 (Anti-monotone Consistency)**.**
Let S be a data record set and let $S'' \subseteq S' \subseteq S$ be data record subsets of S. Let furthermore S' and S'' be data consistent. Then S'' is rendered invalid if S' is rendered invalid by an anti-monotone data constraint $c_{a,data}$, i.e.

$$(c_{a,data}(S', S) = 0) \Rightarrow (c_{a,data}(S'', S) = 0)$$

Proof.

- Case $S' = S''$: The theorem is obviously true for $S' = S''$.

- Case $S' \neq S''$: Proof by contradiction: assume that the theorem stated is not true. Then an *anti-monotone constraint* $c_{a,data}$ and two *data consistent data record subsets* $S', S'' \subseteq S$ of $S = (R, \delta : \mathbb{X} \to D)$ with $S'' \subseteq S'$ exist, so that

  $$(c_{a,data}(S', S) = 0) \wedge (c_{a,data}(S'', S) = 1)$$

  Because $S' \neq S''$ there exist *data instances* $d \in D$, that occur in S' but not in S''. Let D'' be the set of these *data instances*:

  $$D'' = \{d \in D \mid (\exists r \in S' : \delta(r) = d) \wedge (\forall r \in S'' : \delta(r) \neq d)\}$$

  and let furthermore D' denote the *data instances* occurring in S but not in S':

  $$D' = \{d \in D \mid (\exists r \in S : \delta(r) = d) \wedge (\forall r \in S' : \delta(r) \neq d)\}$$

  Now let $\phi_{\{a,b\}}$ be an $\{a, b\}$-*item extractor* defined as follows:

  $$\phi_{\{a,b\}} : \quad \Omega_D \to 2^I$$
  $$d \mapsto \begin{cases} \{c\} & , d \in D' \\ \{a, b\} & , d \in D'' \\ \{a\} & , otherwise \end{cases}$$

Thus $\{a, b\} \supseteq \{a\}$ and

$$S_{\pi_{\{a,b\}}} \overset{\text{data consistency}}{=} S'' \overset{\text{Theorem 2.2}}{\subseteq} S' \overset{\text{data consistency}}{=} S_{\pi_{\{a\}}}$$

and using the *anti-monotone* property of $c_{a,data}$:

$$\begin{aligned} & c_{a,data}(S_{\pi_{\{a\}}}, S) = c_{a,data}(S', S) = 0 \\ \overset{\text{Definition 3.4}}{\Rightarrow} \; & c_{a,data}(S_{\pi_{\{a,b\}}}, S) = c_{a,data}(S'', S) = 0 \end{aligned}$$

Thus S'' is also rendered invalid by $c_{a,data}$, which is a contradiction to the assumption, that the theorem is false. ↯

□

Definition 3.7 (Monotone Description Projection)**.**
*Let $S = (R, \delta)$ be a data record set and π_P a descriptive pattern with the associated modification set E_P. Let furthermore ϕ_I be an I-item projector and $C_{m,desc}$ be a set of monotone description constraints. Then the data record set $S_{\pi_P}^{C_{m,desc}}$ is called **monotone description projection** and is defined as*

$$S_{\pi_P}^{C_{m,desc}} = (\{r \in S_\pi \mid \forall c \in C_{m,desc} : c(\phi_I(\delta(r)) \cap (P \cup E_P)) = 1\}, \delta)$$

A monotone description projection only contains those data records that are associated with itemsets (limited to those items corresponding to the current descriptive pattern and its modification set) that pass all monotone description constraints. Note that a monotone description projection is data consistent with respect to S, as the projection filtering data records is based on their data instances.

Theorem 3.3 (Monotone Description Consistency)**.**
Let $C_{m,desc}$ be a set of monotone description constraints, let π_P be a descriptive pattern and E_P its modification set. If an extension $\pi_{P'}$ of π_P based on E_P (i.e. $P' \in E_P^+$) is not rendered invalid by a monotone description constraint, it projects onto a data record set $S_{P'}$ that is a subset of the monotone description projection $S_{\pi_P}^{C_{m,desc}}$ of π_P, i.e.

$$\forall P' \subseteq P \cup E_P : (\forall c \in C_{m,desc} : c(\pi_{P'}) = 1) \Rightarrow S_{\pi_{P'}} \subseteq S_{\pi_P}^{C_{m,desc}}$$

Proof. Proof by contradiction: $S_{\pi_{P'}} \subseteq S_{\pi_P}$ by definition, thus if the theorem stated was false, then $S_{\pi_{P'}}$, with $\forall c \in C_{m,desc} : c(\pi_{P'}) = 1$, contains a *data record* $(r, \delta(r))$ that is associated with a set of items $\phi_I(\delta(r))$ limited to those

items in $P \cup E_P$, that is rendered invalid by some *monotone description constraint* $c_{m,desc} \in C_{m,desc}$, i.e.

$$\exists r \in S_{P'} : c_{m,desc}(\phi_I(\delta(r)) \cap (P \cup E_P)) = 0$$

Yet because $S_{\pi_{P'}}$ is a projection of $\pi_{P'}$ onto S and $\pi_{P'}$ is an extension of π_P based on its *modification set* E_P it holds

$$P' \subseteq \phi_I(\delta(r)) \cap (P \cup E_P)$$

and because of the *monotonicity* of $c_{m,desc}$

$$c_{m,desc}(P') = 0$$

which is a contradiction to the assumption $\lightning$.

□

Theorem 3.4 (ExAnte)**.**
Let $M = ((D, I, \phi_I, C_{\Pi_I}, C_R), S)$ *be a descriptive pattern mining instance with* $S = (R, \delta)$ *and let* $C_{m,desc} \subseteq C_{\Pi_I}$ *be the set of all monotone description constraints and* $C_{a,data} \subseteq C_{\Pi_I}$ *the set of all anti-monotone data constraints . Let* π_P *be a descriptive pattern and let* E_P *be its associated modification set. Let* $\pi_{P'}$ *denote an extension of* π_P *based on the modification set* E_P *and on a specific item* $i \in E_P$*, i.e.* $i \in P'$*. The* ***ExAnte property*** *states that*

$$(c_{a,data}(S_{\pi_P}^{C_{m,desc}} \cap S_{\pi_{P\cup\{i\}}}, S) = 0) \Rightarrow (\exists c \in C_{\Pi_I} : c(\pi_{P'}, S) = 0)$$

Proof. There are two cases:

1. $S_{\pi_P}^{C_{m,desc}} \cap S_{\pi_{P\cup\{i\}}} = S_{\pi_{P\cup\{i\}}}$: in this case the same *anti-monotone constraint* $c_{a,data}$ that rendered π_P invalid, renders $\pi_{P'}$ invalid, i.e.

$$P \cup \{i\} \subseteq P' \Rightarrow c_{a,data}(S_{\pi_{P'}}, S) = 0$$

2. $S_{\pi_P}^{C_{m,desc}} \cap S_{\pi_{P\cup\{i\}}} \neq S_{\pi_{P\cup\{i\}}}$: in this case again two cases have to be distinguished:

   (a) $S_{\pi_{P'}} \subseteq S_{\pi_P}^{C_{m,desc}}$: $S_{\pi_{P'}}$ and $S_{\pi_P}^{C_{m,desc}}$ are both *data consistent* with respect to S. Thus by Theorem 3.2 the same *anti-monotone constraint* $c_{a,data}$ that rendered $S_{\pi_P}^{C_{m,desc}} \cap S_{\pi_{P\cup\{i\}}}$ invalid, renders $\pi_{P'}$ invalid, i.e. $c_{a,data}(S_{\pi_{P'}}, S) = 0$.

   (b) $S_{\pi_{P'}} \not\subseteq S_{\pi_P}^{C_{m,desc}}$: in this case Theorem 3.3 states that a *monotone description constraint* $c_{m,desc}$ exists, that renders $\pi_{P'}$ invalid.

Listing 3.15: μ-Reduction.

```
1  PROCEDURE μ − reduction(π_P, E_P, C_{m,desc}, S):
2    for r ∈ S do
3      P' ← φ_I(δ(r)) ∩ (P ∪ E_P)
4
5      if P' does not satisfy a monotone description constraint c ∈ C_{m,desc} then
6        S ← S \ {r}
7      endif
8    endfor
9
10   return S
```

□

By definition the *ExAnte property* can be applied to *modification set* pruning. If an *extension* $\pi_{P\cup\{i\}}$ of a *descriptive pattern* π_P by an item $i \in E_P$ from the corresponding *modification set* E_P does not satisfy an *anti-monotone data constraint* based on the union of the *monotone description projection* $S_{\pi_P}^{C_{m,desc}}$ and the projection of the extension $S_{\pi_{P\cup\{i\}}}$, the item $i \in E_P$ used for extension can be removed from the *modification set* E_P of the current *descriptive pattern* π_P. Note that removing items from the *modification set* E_P influences the *monotone description projection* $S_{\pi_P}^{C_{m,desc}}$. Thus the process can be repeated to prune further items.

If a temporary *data record set* S_t is pictured, the *modification set* pruning based on the *ExAnte property* can be split into two steps as introduced by [12]: the *μ-reduction* and the *α − reduction*. Initially the temporary *data record set* is equal to the *conditional data record set* S_{π_P} based on π_P, i.e. $S_t = S_{\pi_P}$.

The *μ-reduction* sets the temporary *data record set* to the *monotone description projection* $S_{\pi_P}^{C_{m,desc}}$ based on the current *descriptive pattern* dependent on the *modification set* E_P (i.e. it removes *data records* reducing the *data record set* using the *monotonicity property*, thus the name*μ-reduction*). Listing 3.15 shows the corresponding algorithm.

Afterwards the *α-reduction* checks each extension π_{P_i} with $P_i = P \cup \{i\}, i \in E_P$ against every *anti-monotone data constraint* $c_{a,data} \in C_{a,data}$. If an extension π_{P_i} is rendered invalid ($c_{a,data}(S_{\pi_P}^{C_{m,desc}} \cap S_{\pi_{P_i}}, S) = 0$) the respective item $i \in E_P$ is removed from the *modification set*. The Listing 3.16 shows the corresponding algorithm. Note that α-reduction is very similar to *modification set pruning*, but only checks *anti-monotone data constraints*.

If some item $i \in E_P$ was removed the process can be restarted for further pruning as the *modification set* has changed. Example 3.8 illustrates the *modification set pruning* step based on the *ExAnte property*. Listing 3.17

Listing 3.16: α-Reduction.

```
1  PROCEDURE α − reduction(π_P, E_P, C_{a,data}, S):
2    forall i ∈ E_P do
3      P' ← P ∪ {i} // generate modification
4
5      if (π_{P'}, S) does not statisfy all anti-monotone data constraints c ∈ C_{a,data} do
6        E_P ← E_P \ {i}
7      endif
8    endfor
```

Listing 3.17: ExAnte loop.

```
1  PROCEDURE exAnte(π_P, E_P, C, S):
2    S_t ← S_{π_P}
3    do
4      S_t ← μ − reduction(π_P, E_P, C, S_t)
5      E_P ← α − reduction(π_P, E_P, C, S_t)
6    while E_P has changed
7    return (E_P, S_t)
```

shows the *ExAnte* loop and Listing 3.18 shows an algorithm which exploits the *ExAnte property* based on the algorithm from Listing 3.13 replacing *modification set pruning* by *ExAnte exploitation* (note that normal *modification set pruning* and *ExAnte* can be used in combination).

Example 3.8 (Modification Set Pruning Exploiting the ExAnte Property)**.** *This example is based on the example from [12]. The constraints being used are the support constraint $c_{supp \geq t_s}$, which is a anti-monotone data constraint and the price constraints $c_{price \geq t_p}$, which is an monotone description constraint. The support threshold is $t_s = 4$ and the price threshold is $t_p = 45$. Figure 3.18 shows an example of how ExAnte reduces data record set and the modification set. The current descriptive pattern π_P is arbitrary, the initial modification set is $E_P = \{a, b, c, d, e, f, g, h\}$. Note any table in this example only shows those items that are part of the current modification set. Table 3.18(a) shows the initial data record set ExAnte is applied to (not that Table 3.18(a) contains items not passing the support threshold; this does not happen, if general modification set pruning is applied first, which includes checking anti-monotone data constraints). The first μ-reduction removes individual 4 as is does not pass the price threshold (Table 3.18(b)). The first α-reduction removes items a, e, g and h from the modification set indicated by removing them from the table yielding Table 3.18(c). This reduces the items associated with each individual lowering their price sum. Thus the second μ-reduction removes individuals $2, 7$ and 9. This results in lower supports for b, d and g. Consequently the second α-reduction removes item g from the*

items		support after						
item	price	start	$1^{st}\mu$	$1^{st}\alpha$	$2^{nd}\mu$	$2^{nd}\alpha$	$3^{rd}\mu$	$3^{rd}\alpha$
a	5	4	3	-	-	-	-	-
b	8	7	7	7	4	4	4	4
c	14	5	5	5	5	5	4	4
d	30	7	7	7	5	5	4	4
e	20	4	3	-	-	-	-	-
f	15	3	3	-	-	-	-	-
g	6	6	5	5	3	-	-	-
h	12	2	2	-	-	-	-	-

Table 3.2: Information about items including supports for different ExAnte states according to the *data record set* in Figure 3.18.

modification set. The third μ-reduction yields a fixpoint by removing individual 5*. In this example the data record set as well as the current modification set are reduced by more than half.*

Note 3.10 (ExAnte and Data Structures)**.**
Instead of creating a conditional data record set from scratch at each node, a subsequently refined data record set can be maintained. The ExAnte property can be used to keep this conditional data record set as small as possible by replacing it by the corresponding monotone description projection.

3.2.2 Dynamic Constraints

Dynamic constraints are a generalization of *pattern constraints* that incorporate information about *result constraints* (which are usually reserved for *solution discovery*) into *pattern discovery*. Especially when generating a *solution* during *pattern discovery* this can be very efficient. *Dynamic constraints* exploit information about the search to evaluate *patterns* during *pattern discovery*. Information that can be exploited includes

- a generatively build solution,
- already found *patterns* in general or
- *modification sets* of already found *patterns*.

. This section will focus on *dynamic constraints* that only take into account the generatively build solution (see Section 3.1.2). These *dynamic constraints* are called *generative constraints* as introduced by Definition 3.8. A *generative constraint* is illustrated by Example 3.8.

individual	items	total price
1	b,c,d,g	58
2	a,b,d,e	63
3	b,c,d,g,**h**	70
4	a,e,g	**31**
5	c,d,**f**,g	65
6	a,b,c,d,e	77
7	a,b,d,**f**,g,**h**	76
8	b,c,d	52
9	b,e,**f**,g	49

(a) Initial *data record set*.

individual	items	total price
1	b,c,d,g	58
2	**a**,b,d,**e**	63
3	b,c,d,g,**h**	70
5	c,d,**f**,g	65
6	**a**,b,c,d,**e**	77
7	**a**,b,d,**f**,g,**h**	76
8	b,c,d	52
9	b,**e**,**f**,g	49

(b) *Data record set* after first μ-reduction.

individual	items	total price
1	b,c,d,g	58
2	b,d	**38**
3	b,c,d,g	58
5	c,d,g	50
6	b,c,d	52
7	b,d,g	**28**
8	b,c,d	52
9	b,g	**14**

(c) *Data record set* after first α-reduction.

individual	items	total price
1	b,c,d,**g**	58
3	b,c,d,**g**	58
5	c,d,**g**	50
6	b,c,d	52
8	b,c,d	52

(d) *Data record set* after second μ-reduction.

individual	items	total price
1	b,c,d	58
3	b,c,d	58
5	c,d	**44**
6	b,c,d	52
8	b,c,d	52

(e) *Data record set* after second α-reduction.

individual	items	total price
1	b,c,d	52
3	b,c,d	52
6	b,c,d	52
8	b,c,d	52

(f) *Data record set* after third μ-reduction.

Figure 3.18: *Data record set* being pruned using the *ExAnte property* with a support threshold of ≥ 4 and a price threshold of ≥ 45. The bold font depicts items and *data records* to be removed. For item information see Figure 3.2.

Listing 3.18: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders*, *branching orders* and an *append function* exploiting *anti-monotone constraints* by *branch pruning*.

```
1  VARIABLES:
2     R ← ∅
3     S  // set implicitly
4     C_{Π_I}  // set implicitly
5
6  PROCEDURE patternDiscovery(I, C_{Π_I}, S):
7     call patternDiscovery_rec(∅, I)
8     return Π_valid
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    (E_P, S) ← exAnte(P, E_P, C_{Π_I}, S)  // prunes E_P
13
14    // branching
15    forall i ∈ E'_P do
16       P' ← P ∪ {i}  // generate modification
17
18       if (π_{P'}, S) statisfies all c_{Π_I} ∈ C_{Π_I} do //
19          R ← R ∪ {π_{P'}}
20       endif
21
22       E_{P'} ← {j | j ∈ E_P ∧ o^I_P(j) < o^I_P(i)}  // modification set generation
23       call patternDiscovery_rec(P', E_{P'})
24    endfor
```

Definition 3.8 (Generative Constraint)**.**
Given a set of patterns Π *and a data domain* D*, a* ***generative constraint*** c_G *is a constraint on* $\Pi \times \Omega_D \times 2^\Pi$*:*

$$c_g : \Pi \times \Omega_D \times 2^\Pi \to \{0, 1\}$$

Note 3.11 (Dynamic Constraints: Consistency)**.**
Dynamic constraints must be designed to be consistent with the result constraints, i.e. using dynamic constraints must yield a solution if a solution exists. For example generative constraints cannot render patterns invalid that turn out to be needed for the solution as the search progresses. As a pattern is not enumerated twice, such a pattern is lost and no solution can be generated.

Note 3.12 (Dynamic Constraints: Properties)**.**
Dynamic constraints can have the same properties as normal pattern constraints, such as being a data or description constraint or being anti-monotone or monotone as long as they are consistent with the result (see Note 3.11). Example 3.9 lists an anti-monotone generative constraint.

Example 3.9 (Generative Constraint: Maximal Support)**.**
The goal is to find two descriptive pattern with maximal support, i.e. the descriptive pattern mining instance

$$M = ((D, I, \phi_I, C_{\Pi_I}, C_R), S)$$

is defined using

- S *as defined by Table 3.19(a)*
- $I = \{a, b, c\}$
- ϕ_I *also as defined by Table 3.19(a)*
- $C_{\Pi_I} = \emptyset$
- $C_R = \{c_{R,max2}\}$*, with* $c_{R,max2}$ *defined as*

$$c_{R,max2} : 2^{\Pi_I} \times \Omega_D \to \{0,1\}$$
$$(R, S) \mapsto \begin{cases} 1, & |R| = 2 \wedge \forall \pi \in R, \pi' \in \Pi_I \setminus R : |S_\pi| \geq |S_{\pi'}| \\ 0, & \textit{otherwise} \end{cases}$$

Instead of using a combination of pattern discovery and solution discovery, a generative approach as introduced in Section 3.1.2 can be taken by defining an append function α_{max2} *and a generative constraint* $c_{\Pi,max2}$*. The append function* α_{max2} *for generating the solution during pattern discovery is defined as*

$$\alpha_{max2} : \Pi_I \times \Omega_\delta \times 2^{\Pi_I} \to 2^{\Pi_I}$$
$$(\pi_P, S, R) \mapsto \begin{cases} (R \cup \{\pi_P\}) \setminus \{\underset{\pi \in R}{argmin}(|S_\pi|)\} & , |R| = 2 \\ R \cup \{\pi_P\} & , \textit{otherwise} \end{cases}$$

The corresponding generative constraint $c_{\Pi,max2}$ *is defined accordingly as*

$$c_{\Pi,max2} : \Pi_I \times \Omega_D \to \{0,1\}$$
$$(\pi_P, S) \mapsto \begin{cases} 1, & |R| < 2 \vee |S_{\pi_P}| > min\{|S_\pi| : \pi \in R\} \\ 0, & \textit{otherwise} \end{cases}$$

Note that $c_{\Pi,max2}$ *is equivalent to the frequency constraint (3.1) with a dynamically raising threshold based on the patterns that were found so far, thus it is anti-monotone.*

Now, consider the search tree in Figure 3.19(b). The algorithm for generative solution discovery is based on Listing 3.14 assuming a global reversed lexical item order and a and a global lexical branching order. As the solution R is empty to begin with, the dynamic support threshold t is zero, i.e. $t = 0$. The search starts at the root node. No item is pruned by modification set pruning because any extension $\pi_{\{i\}}$ (with $i \in E_\emptyset$) exceeds the threshold of $t = 0$. The algorithm proceeds stepwise:

- *The first item from the modification set to branch on is "a", yielding the descriptive pattern $\pi_{\{a\}}$. $\pi_{\{a\}}$ is added to the result because no patterns are in the result yet. The threshold stays at $t = 0$. Thus again, no item is pruned from the corresponding modification set $E_{\{a\}}$.*

- *Recursively the first item $b \in E_{\{a\}}$ is used for extension, yielding the descriptive pattern $\pi_{\{a,b\}}$. $\pi_{\{a,b\}}$ is added to the result, because the temporary result does not contain two patterns yet. The temporary result is now $R = \{\pi_{\{a,b\}}, \pi_{\{a\}}\}$. Consequently the dynamic threshold is now $t = 1$ because $|S_{\pi_{\{a,b\}}}| = 1$.*

- *The only item $c \in E_{\{a,b\}}$ is pruned by modification set pruning because of the generative constraint $c_{\Pi,max2}$.*

- *The search then backtracks to the node corresponding to pattern $\pi_{\{a\}}$ and adds the next (and last) item $c \in E_{\{a\}}$ to $\pi_{\{a\}}$ yielding $\pi_{\{a,c\}}$, which is added to the result, because it exceeds the current dynamic threshold $t = 1$. The temporary result is then $R = \{\pi_{\{a,b\}}, \pi_{\{a,c\}}\}$ and the dynamic threshold is raised to $t = 2$.*

- *The search then backtracks to the root node and adds the second item from $E_\emptyset$ yielding the descriptive pattern $\pi_{\{b\}}$, which exceeds the current dynamic threshold and is added to the temporary result, which is then $R = \{\pi_{\{a,b\}}, \pi_{\{b\}}\}$. The dynamic threshold is raised to $t = 3$. All items in the corresponding modification set $E_{\{b\}}$ are pruned.*

- *The last extension $\pi_{\{c\}}$ is not added to the result, because it does not exceed the current dynamic threshold $t = 3$.*

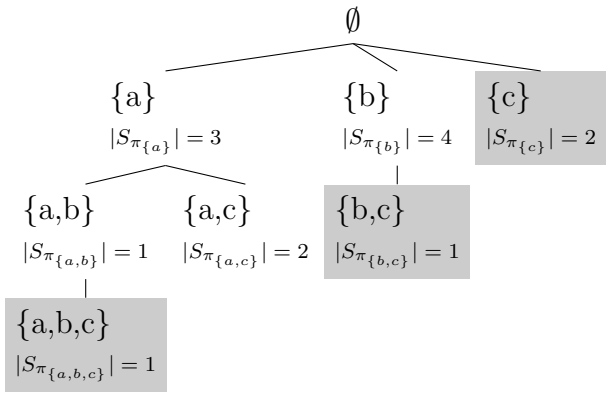
The final solution is $R = \{\pi_{\{a,b\}}, \pi_{\{b\}}\}$. Figure 3.19(b) depicts skipped descriptive patterns by a gray background.

Early Adding

When using *dynamic constraints* it cannot be assumed that a *pattern* will always be rendered valid by all *constraints* just because it was rendered valid

Items			
1	a	b	c
2	a		c
2	a		
6		b	
6		b	
7		b	

(a) The *data record set.*



(b) The search tree.

Figure 3.19: Shows a *data record set* and a corresponding search tree based on the set of items $I = \{a, b, c\}$.

once. For *generative constraints* for example, the evaluation might change with every *pattern* being added to the solution being built. Thus there are several places in the algorithm based on e.g. Listing 3.14 where *dynamic constraints* can be rechecked in order to increase efficiency. Checking more often can be costly, thus a trade-off is introduced. Listing 3.19 shows an algorithm using additional checking of *dynamic constraints*. By additionally utilizing the procedure of Listing 3.20 another optimization is introduced. By adding *descriptive pattern* not when actually branching but before any branch is considered, more information is available to *dynamic constraints* thus more efficient pruning is possible (also see Algorithm 3.7 and 3.8). Figure 3.20 shows the resulting search tree from Example 3.9 depicting the skipped *descriptive patterns* as gray nodes. Overall less *descriptive patterns* are enumerated to generate the same solution.

3.2.3 Data Structures

There are two factors that are costly during *pattern discovery* in *descriptive pattern mining*:

- the enumeration of an exponentially growing set of *descriptive patterns* with respect to the amount of items and
- the projection of the *pattern* π_P onto the *data record set* S in order to evaluate *constraints* (e.g. *data constraints*).

Listing 3.19: *Bottom-up descriptive pattern discovery* based on *modification sets* generated using *item orders* adding *patterns* early to exploit *dynamic constraints*.

```
1  VARIABLES:
2     R ← ∅
3     S
4     C_{Π_I}
5
6  PROCEDURE patternDiscovery(I, C_{Π_I}, S):
7     call patternDiscovery_rec(∅, I)
8     return R
9
10 PROCEDURE patternDiscovery_rec(P, E_P):
11
12    E_P ← modsetPruning(P, E_P, C_{Π_I}, S)  // prunes E_P
13    R ← earlyAdding(P, E_P, C_{Π_I}, S, R)  // adds patterns before branching
14
15    // here a second modification set pruning loop using dynamic anti-monotone constrains
16    // can exploit additional patterns added to the solution during early adding
17
18    // branching
19    forall i ∈ E'_P do
20       P' ← P ∪ {i}  // generate modification
21
22       // check dynamic anti-monotone constraints again
23       if (π_{P'}, S) does not satisfy all dynamic anti-monotone constraints c ∈ C_{Π_I} do
24          continue
25       endif
26
27       E_{P'} ← {j | j ∈ E_P ∧ o^I_P(j) < o^I_P(i)}  // modification set generation
28       call patternDiscovery_rec(P', E_{P'})
29
30       // at this point modification set pruning based on
31       // dynamic anti-monotone constraints could be applied
32       // for modification set pruning again
33    endfor
```

Listing 3.20: Adding *descriptive patterns* early.

```
1  PROCEDURE earlyAdding(P, E_P, C_{Π_I}, S, R):
2     forall i ∈ E'_P do
3        P' ← P ∪ {i}  // generate modification
4
5        if (π_{P'}, S) satisfies all constraints c_{Π_I} ∈ C_{Π_I} do
6           R ← α(π_{P'}, S, R)
7        endif
8     endfor
9
10    return R
```

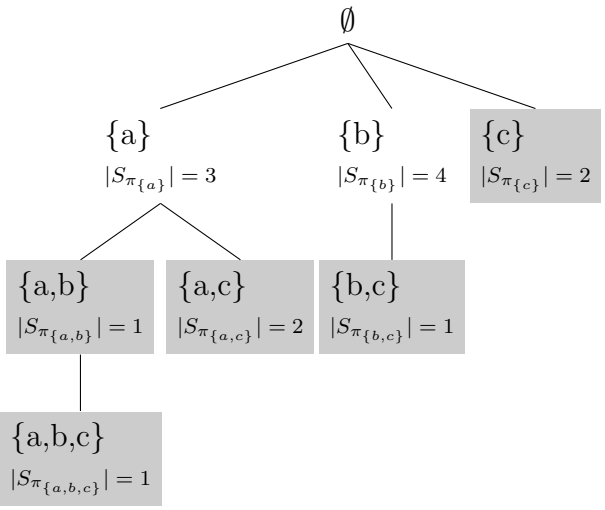


Figure 3.20: A search tree for a set of items $I = \{a, b, c\}$. The gray nodes depict *descriptive constraints*, that were not traversed.

The former was is addressed by the previous section exploiting properties of *constraints* to avoid enumerating invalid *descriptive patterns*. The latter will be elaborated on in this section.

When checking *descriptive patterns* against *constraints* like *data constraints*, the corresponding *conditional data record set* is required (see Definition 2.15). Creating such projections from scratch is expensive when *data record sets* are large. As an alternative Lemma 2.1 states that it is possible to maintain a subsequently refined *conditional data record subset* according to the current *descriptive pattern*. Furthermore it is not necessary to grant access to the actual *conditional data record set*, but merely to a few selected *valuation bases* if *pattern constraints* are defined on *valuation bases* as suggested by 2.3.

Given a *descriptive pattern* π_P different *valuation bases* are required dependent on the algorithm. Common *valuation bases* to require are

- the *valuation basis* $\phi_V(S_{\pi_P})$ of the current *descriptive pattern* π_P or
- the *valuation bases* $\phi_V(S_{\pi_{P\cup\{i\}}})$ of extensions $\pi_{P\cup\{i\}}$ of the current *descriptive pattern* π_P based on its *modification set* E_P, i.e. $i \in E_P$.

In the previous sections every algorithm requires the latter. Thus instead of maintaining a *conditional data record set* for a *descriptive pattern* a *reduced conditional data record set* is sufficient. A *reduced conditional data record set* contains only those *individuals* which will be part of an extension $\pi_{P\cup\{i\}}$ of

the current *descriptive pattern* π_P and its *modification set* E_P, i.e. $i \in E_P$. In other words they only contain *individuals* r associated with a set of items $\phi_I(\delta(r))$ which contains an item from the *modification set* $E_P \cap \phi_I(\delta(r)) \neq \emptyset$. Figure 3.21(a) and 3.21(b) highlight the differences between *conditional data record sets* and *reduced conditional data record sets*. Definition 3.9 introduces *reduced conditional data record sets* formally.

Definition 3.9 (Reduced Conditional Data Record Set)**.**
Given a data record set $S = (R, \delta)$, a descriptive pattern π_P and an I-item projector ϕ_I, a ***reduced conditional data record set*** *$S^-_{\pi_P}$ is defined as*

$$S^-_{\pi_P} = (\{r \in S_\pi \mid \phi_I(\delta(r)) \cap E_P \neq \emptyset\}, \delta)$$

where E_P is the modification set of π_P.

Until now, the actual *data record subsets* are used to access *valuation bases*. But it is also possible to transform *data record sets* such that only the absolutely necessary information is being stored, which is the *valuation bases*. Such a *data record set* is coined *component data record* and specified by Definition 3.10. It associates each *individual* with a *data instance* containing an *itemset component* and a *valuation basis component*. The former corresponds to the items an *individual* was originally associated with and the latter corresponds to the associated *valuation basis*. This representation is used instead of the original *data record set* representation to limit the stored data to a minimum.

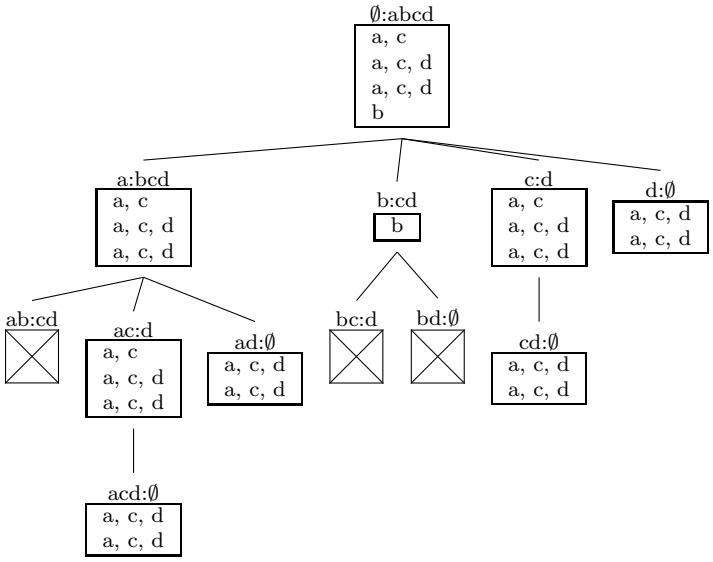
Definition 3.10 (Component Data Record Set)**.**
Given a data record set $S = (R, \delta)$, an I-item projector ϕ_I and a V-valuation basis projector ϕ_V, the corresponding ***component data record*** *S^V is specified on the data domain $D^V = 2^I \times V$ as*

$$S^V = (R, \delta^V : R \to D)$$

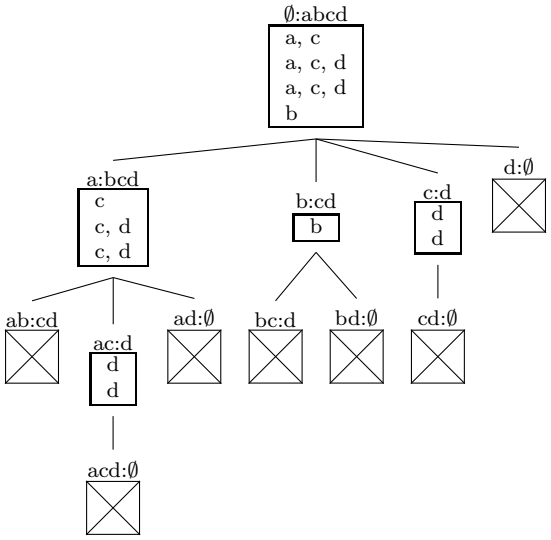
with δ^V being defined as

$$\begin{aligned} \delta^V : \quad & R \to 2^I \times V \\ & r \mapsto (\phi_I(\delta(r)), \phi_V(\delta(r))) \end{aligned}$$

Let $d = (I', v)$, $d \in D^V$ be a data instance from the data domain D^V, then I' is called ***itemset component*** *and v is called* ***valuation basis component*** *of d.*



(a) *Conditional data record sets* for each *descriptive pattern*.



(b) *Reduced conditional data record sets* for each *descriptive pattern*.

Figure 3.21: *Conditional data record sets* and *reduced conditional data record sets* for each *descriptive pattern*. *Descriptive patterns* and their *modification sets* divided by a colon are shown above each corresponding *(reduced) conditional data record set*. Only the items associated with *individuals* are depicted and not the *data instances*. *Reduced data record sets* only show those items part of the current *modification set*. Empty *projections* are depicted as crossed out squares. Only the first *pattern* projecting onto an empty *data record set* is shown.

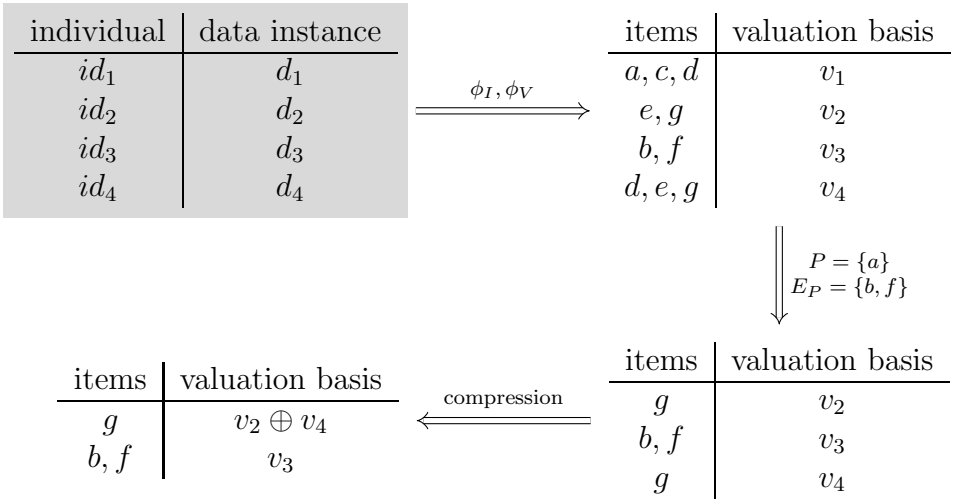


Figure 3.22: This figure depicts the general process of creating a *component data record set* from a *data record set* (upper left to upper right). It also shows how the *reduced conditional component data record set* according to the *descriptive pattern* $\pi_{\{a\}}$ and its *modification set* E_P is calculated (upper right to lower right). A possible merging step is also depicted (lower right to lower left). The identifiers for *individuals* are not explicitly listed for the *component data record sets*.

Now, as *descriptive patterns* select *individuals* by their associated set of items, those *data records* with equal *itemset components* can be merged. Furthermore given a *descriptive pattern* π_P and its *modification set* E_P, only items $i \in E_P$ need to be retained in the *item components* of the corresponding *conditional reduced component data record set* S_P^V for the same reason. Figure 3.22 depicts the general process of creating a *component data record set*, projecting it onto a *conditional reduced component data record set* and merging *data records* with the same *itemset components*.

Finally, let S_P^V be a *reduced conditional component data record set* of a *descriptive pattern* π_P. It supports all operations required by the optimizations introduced by previous sections:

- *atomic valuation basis extraction*: given an item $i \in E_P$, the valuation basis $\phi_V(S_{\pi_{P \cup \{i\}}})$ can be extracted by iterating over S_P^V and accumulating all *valuation bases* associated with *data instances* containing the item i.

- *ExAnte support*:

  - *μ-reduction*: let $(r, \delta(r))$ be a *data record* from the current *conditional data record set* and let $I_m = \phi_I(\delta(r)) \cap (P \cup E_P)$. If a *monotone description constraint* c_m renders I_m invalid ($c_m(I_m) = 0$), then the corresponding *data record* is removed from the *conditional data record set*. Such a *data record* is part of a potentially merged *data record* $(r', (I', v))$ from the corresponding *reduced conditional component data record set*. The required itemset I_m is equivalent to $I_m = P \cup I'$.
  - *α-reduction*: to represent reductions in the *modification set* E_P, the pruned items can be removed from each *itemset component* of the current *reduced conditional component data record set*

- *atomic sequential projection*: to calculate the next *reduced conditional component data record set* $S^V_{P\cup\{i\}}$ all *data records* not containing i in their *itemset component* are removed. Also, only items from the new *modification set* are retained in the itemset components. *Data records* with empty *itemset components* are removed. To create a maximally compressed *data record set* the *itemset components* of each *data record* need to be compared. *Data records* with the same *itemset component* are merged. This last step is usually not applied to avoid the cost of pairwise comparing the *data records* accepting the larger size of the resulting *conditional component data record set*.

To address the issue of generating maximally compressed *conditional component data record sets* by *atomic sequential projection* the notion of *GP-Trees* can be used as introduced by the next section.

GP-Tree

The FP-Tree structure was introduced by [33] in the context of frequent itemset mining. The FP-Tree consists of a prefix tree based on some order of items and an item table. The item table contains all items that occur in the tree. Each node of the prefix tree represents an item. The nodes representing the same item are linked. The head of such a link structure is stored in the item table together with the corresponding item for identification. The FP-Tree of a *data record set* or a *conditional data record set* is built by iterating over each *data record* of the *data record set* sorting the associated items (limited to those, part of the *modification set*) by the order defined by the tree and inserting the sorted list of items into the prefix tree. If a new node needs to be generated during the insertion, the link structure between nodes representing the item is adjusted accordingly.

For frequent itemset mining each node contains a frequency count. Each *individual* is associated with a frequency f (usually $f = 1$ in the initial *data record set*). If a node is created its count is initialized with the frequency associated with the *data record* that provided the items to be inserted into the tree. If a node already exists, the frequency of the *data record* being inserted is added to the frequency count of the existing node. Figure 3.23(b) shows an FP-Tree based on the *data record set* depicted by Figure 3.23(a). The *data domain* is the powerset 2^I of items I. In this case the order on the items to build the tree is $f > c > a > b > m > p$ according to their frequencies. Figure 3.23(c) shows the first few steps of building the FP-Tree from Figure 3.23(b). These examples are based on [33].

Note 3.13 (Compactness)**.**
The item order o^I for sorting the items influences the compactness of a FP-Tree greatly. [33] mentions that using a frequency order is a good heuristic to yield rather compact FP-Trees (items with a higher frequency are closer to the root). Yet it is also emphasized that in some cases more compact trees exist as Figure 3.24 illustrates.

Instead of just the frequency, any *valuation basis* can be stored and accumulated at the tree nodes, generalizing the data structure. The tree is then called a *GP-Tree* (generic pattern tree). Figure 3.23 shows how the *component data record set* is used as input for creating the FP-Tree. Thus, GP-Trees can be built from arbitrary *component data record sets*.

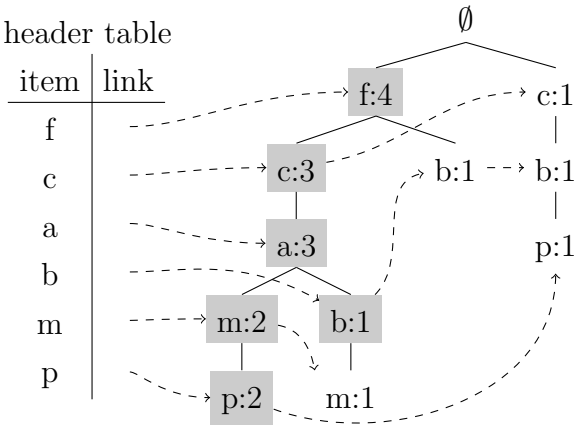
Now, let π_P be the current *descriptive pattern* and E_P is *modification set*. Also let S^V_P be the current *reduced conditional component data record set*. Instead of calculating the *reduced conditional component data record set* $S^V_{P\cup\{i\}}$ for an extension $\pi_{P\cup\{i\}}$ based on E_P directly from S^V_P, S^V_P is converted into a GP-Tree representation based on the *item order* o^I_P imposed on the current *modification set* E_P. Now $S^V_{P\cup\{i\}}$ can be calculated by iterating over all GP-Tree nodes associated with i. These nodes can quickly be accessed by the link structure between nodes with the same item. For each node a *data record* is added to the new *reduced conditional component data record set* $S^V_{P\cup\{i\}}$: the *item component* being those items associated with parents of the current node, and the *valuation basis component* being the *valuation basis* of the current node. Figure 3.25 illustrates the procedure.

Note 3.14 (Item Order and Modification Set)**.**
Note that the item order to build the GP-Tree must be identical with the item order to build modification sets. Otherwise the resulting component data record sets will not be consistent with the modification set. Thus when changing the modification set or the item order after building a GP-Tree,

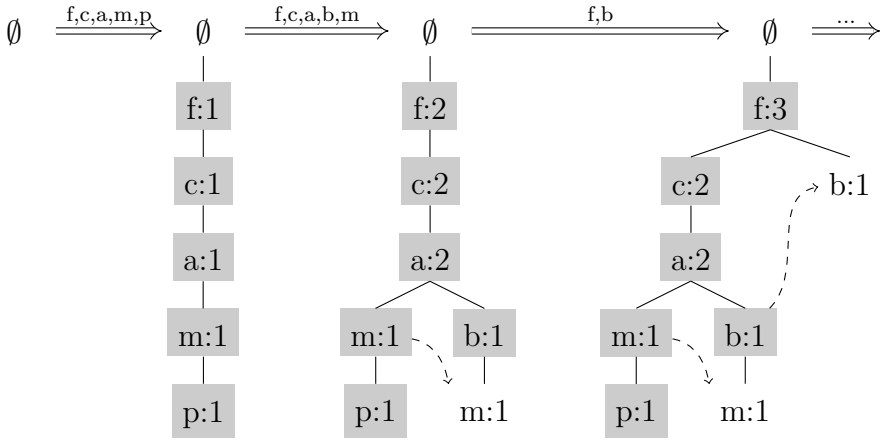
individual	items
100	f,c,a,m,p
200	f,c,a,b,m
300	f,b
400	c,b,p
500	f,c,a,m,p

items	valuation basis
f,c,a,m,p	1
f,c,a,b,m	1
f,b	1
c,b,p	1
f,c,a,m,p	1

(a) *Data record set* and matching *component data record set* with support as *valuation basis*.



(b) FP-Tree.

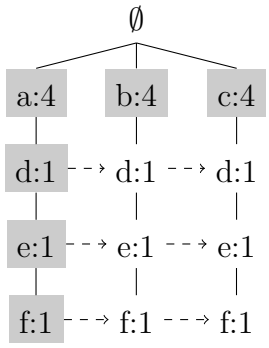


(c) Building an FP-Tree. The header table is left out. Upon adding an item, the according header node is stored in the header table.

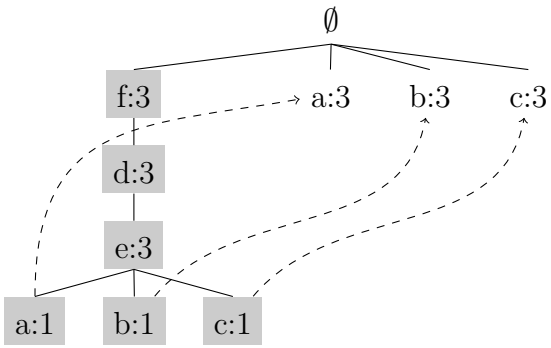
Figure 3.23: A FP-Tree according to a *data record set* and how it is build. The *data domain* of the *data record set* is the powerset 2^I of items $I = \{f, a, c, b, m, p\}$. The order used to build the tree is $f > c > a > b > m > p$ according to their frequencies. The gray nodes depict the head nodes stored in the header table.

individual	items
1	adef
2	bdef
3	cdef
4	a
5	a
6	a
7	b
8	b
9	b
10	c
11	c
12	c

(a) Data record set.



(b) FP-Tree built using frequency / support count node order. Node order: $a > b > c > d > e > f$



(c) FP-Tree built using custom node order. Node order: $f > d > e > a > b > c$

Figure 3.24: FP-Trees according to a *data record set* using different item orders. The *data domain* of the *data record set* is the powerset 2^I of items $I = \{a, b, c, d, e, f\}$.

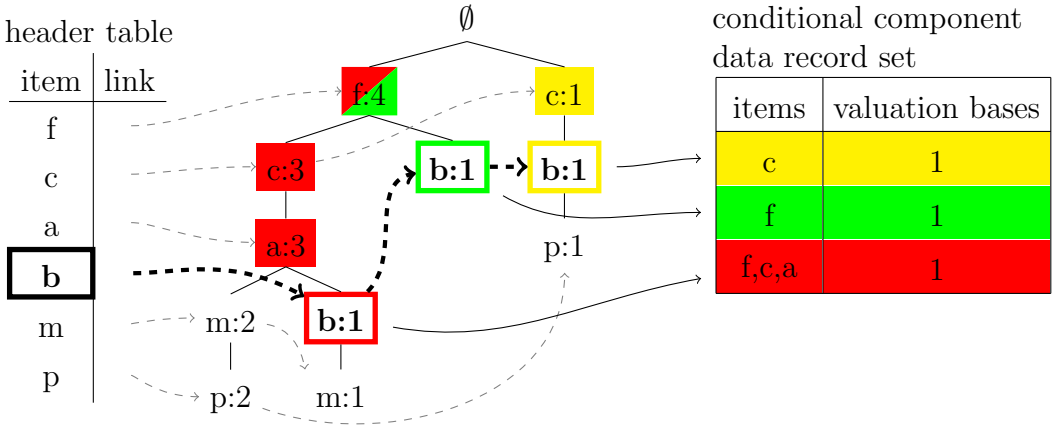


Figure 3.25: *GP-Tree* from Figure 3.23(b) used to build a *conditional component data record set*.

the tree has to be rebuilt to be able to yield consistent results. As building a GP-Tree is expensive this has to be avoided.

The advantage of using this method to create *reduced conditional component data record sets* is that it yields maximally compressed *component data record sets* (with respect to the *item order* applied) for every *conditional component data record set* created from the *GP-Tree*, i.e. those rows that share the same set of items are merged (see Figure 3.22 for an example).

Note 3.15 (Storage Complexity)**.**
Note that if the valuation bases are of constant size, GP-Trees are a very efficient way of storing data records because data records with the same set of items are combined in a single path of the GP-Tree and the size needed to store the accumulated valuation basis does not grow with additional data records sharing the same nodes. Furthermore because of the order of items imposed when creating the GP-Tree, data records share prefixes of items. Data records that share a prefix also share the nodes associated with it.

Yet valuation bases not being of constant size pose a problem. For example considering the most general valuation basis storing the data instances themselves, every node contains the data instances of the data records it is associated with. This will introduce an exponential growth on the size requirement of the tree. The GP-Tree equivalent to the FP-Tree from Figure 3.23(b) storing the data instances as valuation bases is depicted by Figure 3.26. Intelligent implementation of valuation bases can reduce this effect.

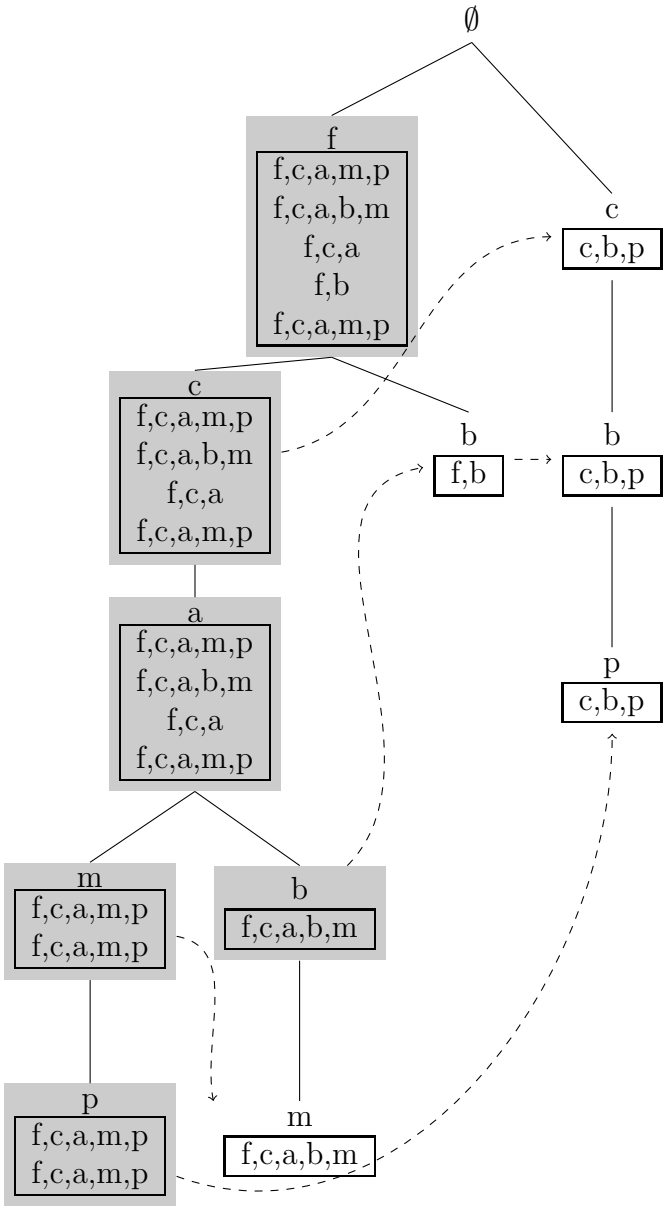


Figure 3.26: A *GP-Tree* storing a multiset of *data instances* as *valuation basis* (the most general *valuation basis*) based on Figure 3.23(b).

Single Prefix Path Exploiting a *single prefix path* is another method to improve the efficiency of the search. This method originates from using the *GP-Tree*, but can be generalized to other data structures at the cost of additional calculations. Given a *descriptive pattern* and its *modification set* the general idea is to avoid creating further *conditional data record sets* for corresponding extensions. Instead *valuation bases* calculated for the current items from the *modification set* are to be reused. The *conditional data record set* has to fulfill certain requirements to do so. If a *single prefix path* can be extracted from the *conditional data record set*, the *valuation bases* of extensions based on items associated with the *single prefix path* can be directly deduced from the *valuation bases* of these items. The notion of *single prefix paths* is introduced by Definition 3.11.

A *single prefix path* $E_P^{SPP} \subseteq E_P$ is a subset of the current *modification set* E_P. The items $i \in E_P^{SPP}$ of the *single prefix path* can be ordered by the frequency / support of their respective *conditional data record sets* $|S_{\pi_{P\cup\{i\}}}|$. If an item $i \in E_P^{SPP}$ is associated with a certain frequency then its associated *conditional data record set* $S_{\pi_{P\cup\{i\}}}$ is a subset of those items j associated with an equal or a higher frequency, i.e. $|S_{\pi_{P\cup\{i\}}}| \leq |S_{\pi_{P\cup\{j\}}}| \Rightarrow S_{\pi_{P\cup\{i\}}} \subseteq S_{\pi_{P\cup\{j\}}}$. Or in other words each *individual* $r \in S_{\pi_{P\cup\{i\}}}$ is associated with those items j of higher frequency, i.e. $|S_{\pi_{P\cup\{i\}}}| \leq |S_{\pi_{P\cup\{j\}}}| \Rightarrow \forall r \in |S_{\pi_{P\cup\{i\}}}| : j \in \phi_V(\delta(r))$. At the same time the size of each *conditional data record set* based on an item $i \in E_P^{SPP}$ from the *single prefix path* must be equal to or larger than those based on items $j \in E_P \setminus E_P^{SPP}$ not in the *single prefix path*, i.e. $|S_{\pi_{P\cup\{i\}}}| \geq |S_{\pi_{P\cup\{j\}}}|$.

Definition 3.11.
[Single Prefix Path]

Let S be a data record set and π_P a descriptive pattern with the modification set E_P. A subset $E_P^{SPP} \subseteq E_P$ of the modification set is called single prefix path if an item order o_P^I exists such that

$$\forall i, j \in E_P^{SPP} : |S_{\pi_{P\cup\{i\}}}| \leq |S_{\pi_{P\cup\{j\}}}| \Rightarrow S_{\pi_{P\cup\{i\}}} \subseteq S_{\pi_{P\cup\{j\}}}$$

and

$$\forall i \in E_P^{SPP}, j \in E_P \setminus E_P^{SPP} : S_{\pi_{P\cup\{i\}}} \supseteq S_{\pi_{P\cup\{j\}}}$$

Table 3.27(a) shows a *data record set* yielding a *single prefix path* E_P^{SPP} $\{a, b, c\}$ (with $P = \emptyset$ and $E_P = \{a, b, c, d, e\}$). A *GP-Tree* structure built upon an *item order* based on their frequencies results in a tree, where the *single prefix path* is represented by those nodes starting at (but not including) the root until (and this time including) the first node with more than one branch. Figure 3.27(b) shows the *GP-Tree* based on the frequency *item order*

according to the *data record set* from Table 3.27(a) using the frequency *valuation basis*. Figure 3.27(c) shows how the *single prefix path* cannot directly be deduced from the *GP-Tree* if a different order is being used.

The consequence is two-fold:

- Let $P' \subseteq E_P^{SPP}$ be a subset of the *single prefix path* E_P^{SPP} of the current *descriptive pattern* π_P and let $\pi_{P \cup P'}$ be the associated extension. Let furthermore $i \in P'$ denote the item with the lowest frequency in the subset P', i.e. $\forall j \in P' : |S_{\pi_{P \cup \{i\}}}| \leq |S_{\pi_{P \cup \{j\}}}|$.

  Then the *conditional data record set* $S_{\pi_{P \cup P'}}$ based on the extension $\pi_{P \cup P'}$ according to the subset $P' \subseteq E_P^{SPP}$ of the *single prefix path* E_P^{SPP} is equal to the *conditional data record set* $S_{\pi_{P \cup \{i\}}}$ based on the extension according to the item $i \in P'$ with the lowest frequency, i.e. $S_{\pi_{P \cup P'}} = S_{\pi_{P \cup \{i\}}}$.

- Let $P' \subseteq E_P$ be a subset of the *modification set* E_P of the current *descriptive pattern* π_P, with an item $k \in P'$, that does not belong to the *single prefix path* $k \notin E_P^{SPP}$. Let furthermore $i \in E_P^{SPP}$ be the item with the lowest frequency in the *single prefix path* E_P^{SPP}, i.e. $\forall j \in E_P^{SPP} : |S_{\pi_{P \cup \{i\}}}| \subseteq |S_{\pi_{P \cup \{j\}}}|$.

  Then the *conditional data record set* $S_{\pi_{P \cup P'}}$ based on the extension $\pi_{P \cup P'}$ according to the subset $P' \subseteq E_P$ of the *modification set* E_P is a subset of the *conditional data record set* $S_{\pi_{P \cup \{k\}}}$ based on the extension $\pi_{P \cup \{k\}}$ according to the item $i \in E_P^{SPP}$ with the lowest frequency, i.e. $S_{\pi_{P \cup P'}} \subseteq S_{\pi_{P \cup \{i\}}}$.

The original *FP-Growth* algorithms (cf. [33]) exploits both consequences by

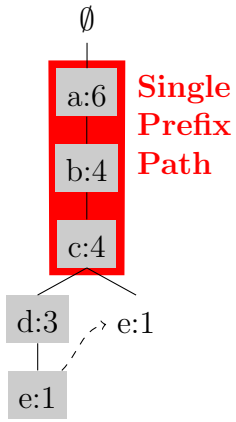
- first generating the powerset of the *single prefix path* using it to generate *descriptive patterns*, i.e. every element from the powerset is used to extend the current *descriptive pattern*. Afterwards all such generated *descriptive patterns* are added to the result using the *valuation basis* of the least frequent item for checking against constraints.

- Then the recursive search is called. Upon its return each *descriptive pattern* found during the recursive search is extended by all elements from the powerset generated during the first step and added to the result using the *valuation basis* of the *descriptive pattern* from the recursive search for checking against constraints.

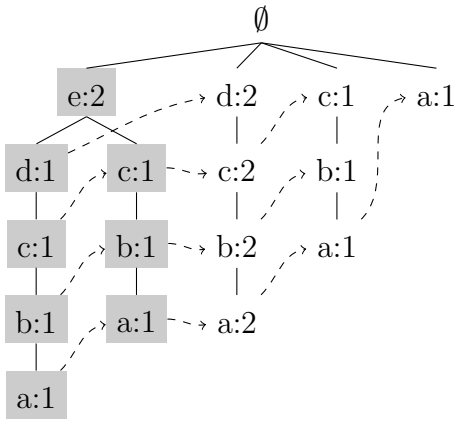
This method has a few disadvantages:

individual	items
1	a,b,c,d,e
2	a,b,c,d
3	a,b,c,d
4	a,b,c,e
5	a,b,c
6	a

(a) Data record set.



(b) FP-Tree built using frequency node order resulting in a *single prefix path*. Node order: $a > b > c > d > e$



(c) FP-Tree built using reverse frequency node order resulting on no *single prefix path*. Node order: $a < b < c < d < e$

Figure 3.27: FP-Trees according to a *data record set* using different item orders. Using the frequency order results in a *single prefix path*, using the reverse order does not. The *data domain* of the *data record set* is the powerset 2^I of items $I = \{a, b, c, d, e\}$.

- the powerset can be large and needs to be saved or generated again when the recursive search returns
- the amount of *descriptive patterns* from the recursive search may be also be large
- the *anti-monotone description* property is not used for pruning during powerset creation
- *descriptive patterns* are not added during but after the recursive search returns, which can lead to less efficient pruning when *dynamic constraints* are present

The former two disadvantages were irrelevant to [33], because all the *descriptive patterns* were part of the solution, thus capacity to store them all had to available anyways. The latter two were not an issue because neither *anti-monotone description* nor *dynamic constraints* were present.

Yet for example *top-k mining* can render extensions based on the cached powerset invalid, while it is still stored, thus the storage or the time to create it, is wasted. Also *top-k mining* is highly dependent on which *descriptive pattern* are added when to the result. In the more general setting of *descriptive pattern mining* these issues have to be considered.

To avoid most of the issues mentioned above, exploiting the *single prefix path* can be limited to cases, where the *single prefix path* E_P^{SPP} is equal to the *modification set* E_P. In such case the powerset of the *single prefix path* is build to create extensions of the current *descriptive pattern* π_P. To exploit *anti-monotone constraints* during this procedure a similar approach to the usual *pattern discovery* approach is taken (with the exception of exploiting the *ExAnte property* for reasons explained in Note 3.16). The only difference is, that instead of keeping a data structure containing *data records*, the *valuation bases* according to each item are kept. This method exploits the *single prefix path* by avoiding to generate *conditional data record sets* and at the same time it can exploit *anti-monotone constraints*. If all *anti-monotone constraints* are *data constraints* and no *dynamic constraints* are present, creating the powerset can be done more efficiently (instead of creating it recursively). Instead of starting a recursive search, the powerset can simply be enumerated.

Note 3.16 (Single Prefix Path and ExAnte)**.**
ExAnte exploitation uses monotone description constraints to reduce the current conditional data record set in order to allow anti-monotone data constraints to further prune items. If the single prefix path E_P^{SPP} *contains only*

items i that will yield valid extensions $\pi_{P\cup\{i\}}$ of the current descriptive pattern π_P, then the item $j \in E_P^{SPP}$ with the smallest frequency already yields a descriptive pattern $\pi_{P\cup\{j\}}$ that is valid according to all anti-monotone constraints. Thus any extension $\pi_{P\cup P'}$ using items $P' \subseteq E_P^{SPP}$ from the single prefix path E_P^{SPP} only yields valid descriptive patterns according to the anti-monotone data constraints. As a result using the ExAnte exploitation to reduce a modification set only containing items from a single prefix path will not yield any results.

In order to exploit cases where the *single prefix path* is not equal to the *modification set*, two approaches can be taken which will only be roughly outlined here:

- *Branching Set Pruning*: remove the items of the *single prefix path* from the *branching set* (items the current *descriptive pattern* is directly extended by) and enumerate their powerset exploiting *anti-monotone constraints* while using fixed *valuation bases*. Other than that. Proceed normally.

- *Optional Item Pushing*: a set of *optional items* is assumed. Remove items of the *single prefix path* from the *modification set* and add them to the *optional items*. Exploit the *optional items* like a *single prefix path*. The set of *optional items* is then given as an argument to the recursive search.

Bitsets

Given a *data record set* $S = (R, \delta)$ and a *I-item projector* ϕ_I, a data structure to efficiently store the *data record sets* are bitset representations as utilized by [40]. The idea is to transform the the *data record set* S into a binary table T_S based on the associated items I. Each row is associated with an item $i \in I$. Each column is associated with an *individual* $r \in R$. Each entry $T[k, l]$ is one if an item k is associated with an *individual* l as is defined as follows:

$$T[k,l] := \begin{cases} 1, i_k \in \phi_I(\delta(r_l)) \\ 0, else \end{cases}$$

The *data instance* or *valuation basis* of an *individual* is linked to the column and can be accessed. Figure 3.28 shows a *data record set* and its corresponding bitset representation.

The bitset representation can be seen as an implementation of a *reduced component data record set*. It provides the same features.

individual	items
100	f,c,a,m,p
200	f,c,a,b,m
300	f,b
400	c,b,p
500	f,c,a,m,p

(a) *Data record set* with the *data domain* be the powerset 2^I of items $I = \{a, b, c, f, m, p\}$

		individuals				
		100	200	300	400	500
items	a	1	1	0	0	1
	b	0	1	1	1	0
	c	1	1	0	1	1
	f	1	1	1	0	1
	m	1	1	0	0	1
	p	1	0	0	1	1

(b) The bitset representation of the *data record set* from Figure 3.28(a).

Figure 3.28: A *data record set* and its corresponding bitset representation.

- *atomic valuation basis extraction*: the *valuation basis* according to an item i_k is accumulated by adding up the *valuation bases* according to those *individuals* r_l being associated with that item i_k, i.e. $T[k, l] = 1$.
- *ExAnte support*:
  - *μ-reduction*: let $(r, \delta(r))$ be a *data record* from the current *conditional data record set* and let $I_m = \phi_I(\delta(r)) \cap (P \cup E_P)$. If a *monotone description constraint* c_m renders I_m invalid ($c_m(I_m) = 0$), then the corresponding *data record* is removed from the *conditional data record set*. Such a *data record* is part of a potentially merged *data record* $(r', (I', v))$ represented by a column $T[, l]$. The required itemset I_m is equivalent to $I_m = P \cup \{i_k | T[k, l] = 1\}$, if only rows corresponding to items from the *modification set* exist, otherwise $T[, l]$ needs to be adjusted using a vector corresponding to the items in the *valuation basis*.
  - *α-reduction*: to represent reductions in the *modification set* E_P, only those rows $T[k,]$ are retained which correspond to items $i_k \in E_P$ in the *modification set*.
- *atomic sequential projection*: each row $T_S[k,]$ is a binary vector associated with item i_k. A component-wise AND operation ($\wedge$) on two such vectors $T_S[k_1,]$ and $T_S[k_2,]$ returns a binary vector $T_S[k_1 \wedge k_2]$ that contains one-entries for all those *individuals* r_l, that are associated with both items i_{k_1} and i_{k_2}, i.e.

$$i_{k_1} \in \phi_I(\delta(r_l)) \wedge i_{k_2} \in \phi_I(\delta(r_l)) \Leftrightarrow \{i_{k_1}, i_{k_2}\} \subseteq \phi_I(\delta(r_l))$$

		individuals				
		100	200	300	**400**	500
items	a	1	1	0	**0**	1
	b	0	1	1	**0**	0
	c	1	1	0	**0**	1
	f	**1**	**1**	**1**	**0**	**1**
	m	1	1	0	**0**	1
	p	1	0	0	**0**	1

(a) Projection based on item f.

		individuals			
		100	200	300	500
items	a	1	1	0	1
	b	0	1	1	0
	c	1	1	0	1
	m	1	1	0	1
	p	1	0	0	1

(b) Compressed bitset representation by removing the item used for projection and *individuals* that are not associated with any remaining item (items from the *modification set*).

Figure 3.29: The projection based on item f of the bitset representation from Table 3.28(b) and the compressed version of that projection.

Thus the *atomic sequential projection* based on an item i_k is calculated by using the AND operation based on the vector $T[k,]$ on all rows. An example is given by Figure 3.29 showing the bitset representation of the *conditional data record set* $S_{\pi_{\{f\}}}$ by doing an *atomic sequential projection* based on item f. f is not associated with *individual* 400, thus every corresponding entry is set to zero. Furthermore only rows corresponding to items from the *modification set* are retained. Afterwards columns only containing zeros are removed. For merging *data records* columns need to be compared pairwise. If the column are identical they can be merged by removing one of them and accumulating the corresponding *valuation bases*. This process is again costly.

Note 3.17 (GP-Trees)**.**
Building a GP-Tree is expensive compared to maintaining a bitset representation of a data record set. On the other hand GP-Trees keep data record set representations as small as possible. As a result using GP-Trees are useful for sparse data record sets and bitset representations perform well on dense data. It might be favorable to use a combination, if it is possible to deduce a density measure from a data record set.

items		
	a	{100, 200, 500 }
	b	{ 200, 300, 400, }
	c	{100, 200, 400, 500 }
	f	{100, 200, 300, 500 }
	m	{100, 200, 500 }
	p	{100, 400, 500 }

Figure 3.30: The TID-Set representation of the *data record set* in Table 3.28(a).

Vertical

Vertical or TID-Set *data record set* representations keep track of *individuals* dependent on items. Instead of creating a binary table, the TID-Set representation directly keeps sets of "transaction ids (TID)", i.e. the *individuals* of *data records* for each item $i \in I$. Everything works just like when using bitset representations tables with the conjunction ($\cap$) replacing the AND operation ($\wedge$). Instead of iterating over the whole length of a bit vector to accumulate a *valuation basis* the identifiers from the resulting TID-set can be accessed directly. Table 3.30 shows the TID-Set representation of the *data record set* in Table 3.28(a).

4

Constraints

This section will introduce several common *constraints* known from data mining tasks that can be expressed as *(generative) descriptive pattern mining classes*. Those *constraints* will be formulated abiding by the framework from Section 2. One subsection covers *pattern constraints*, the other *result constraints*.

4.1 Pattern Constraints

The following sections will introduce a common way of formulating *data constraints*, i.e. by the notion of *quality functions*. Afterwards a set of basic *descriptive pattern mining classes* will be introduced abiding by the framework defined in Section 2. Then a class of *constraints* is revisited that reduces redundancy among *descriptive patterns*.

4.1.1 Quality Functions and Optimistic Estimates

A common way to express *data constraints* is by defining *quality functions*. *Quality functions* are functions that, given a *data record set*, map a *pattern* onto a real value as formalized by Definition 4.1. The most common *quality function* is the *frequency quality function* mapping a *pattern* onto the size of its projection as illustrated by Example 4.1. A *quality function* can be used to formalize a *pattern constraint* by defining a threshold. A *pattern* is only valid, if its quality is greater than or equal to the threshold. Such a *pattern*

constraint is called a *quality constraint* and is defined by Definition 4.2. Thus, defining a *quality function* and a threshold yields a *pattern constraint*.

Definition 4.1 (Quality Function)**.**
Given a set of patterns Π *and a data domain* D*, a* ***quality function*** q *is defined as*

$$q : \Pi \times \Omega_D \to \mathbb{R}$$

Example 4.1 (Quality Function: Frequency / Support)**.**
Given a data record set S*, the* ***frequency quality function*** *or* ***support quality function*** q_{supp} *is projecting a pattern* π *onto the size* $|S_\pi|$ *of its projection* S_π*:*

$$\begin{aligned} q_{supp} : \quad & \Pi \times \Omega_D \to \mathbb{R} \\ & (\pi, S) \mapsto |S_\pi| \end{aligned}$$

Definition 4.2 (Quality Constraint)**.**
Given a data domain D *and a set of patterns* Π*, as well as a* ***quality function*** $q : \Pi \times \Omega_D \to \mathbb{R}$*, a* ***quality constraint*** $c_{q \geq t}$ *according to a threshold* $t \in \mathbb{R}$ *is defined as*

$$\begin{aligned} c_{q \geq t} : \quad & \Pi \times \Omega_D \to \{0, 1\} \\ & (\pi, S) \mapsto \begin{cases} 1, \text{ if } q(\pi, S) \geq t \\ 0, \text{ otherwise} \end{cases} \end{aligned}$$

Considering the *descriptive pattern mining* setting the *frequency quality constraint* yielding from the *frequency quality function* (see Example 4.1) is *anti-monotone* (cf. Theorem 3.1). Yet not every *quality constraint* emits *anti-monotone* properties. To allow for *anti-monotone* pruning using a *quality constraint*, the corresponding *quality function* must be *anti-monotone* itself. A *quality function* is called *anti-monotone*, if it returns a smaller or equal value for an extension $\pi_{P'}$ of a *descriptive pattern* π_P than for the *pattern* π_P itself (see Definition 4.3). Thus, any extension $\pi_{P'}$ of a *descriptive pattern* π_P which is rendered invalid by a *quality constraint* $c_{q \geq t}$ using an *anti-monotone quality function* q is rendered invalid, too.

Definition 4.3 (Anti-Monotone Quality Function)**.**
Let Π_I *be a set of descriptive patterns and let* $q : \Pi_I \times \Omega_D \to \{0, 1\}$ *be a quality function. Given a data record set* S*,* q *is called* ***anti-monotone****, if and only if*

$$\forall P, P' \subseteq I : P \subseteq P' \Rightarrow q(\pi_P) \geq q(\pi_{P'})$$

To utilize a *quality function* for *anti-monotone* pruning even though it is not *anti-monotone*, an additional *quality constraint* based on a corresponding *optimistic estimate* needs to be formulated. An *optimistic estimate* of a *quality function* (as defined by Definition 4.4) is itself a *quality function* which estimates the maximum quality the respective *quality function* can still assign to an extension of a *descriptive pattern* (including the *descriptive pattern* itself). Thus, an *optimistic estimate* is always *anti-monotone* (cf. Theorem 4.1). If the quality $q(\pi_P, S)$ of a *descriptive pattern* π_P does not satisfy the threshold t defined by a *quality constraint* $c_{q \geq t}$, that *pattern* π_P is not added to the set of valid *patterns* $\Pi_{I,valid}$. If the *optimistic estimate* $e_q(\pi_P, S)$ of that *quality function* q does not satisfy the quality threshold t, extensions of that *descriptive pattern* can be discarded. The corresponding *constraint* $c_{e_q \geq t}$ is *anti-monotone*. Examples of *optimistic estimates* can be found in Section 4.1.2 (*subgroup mining* and *community mining*).

Definition 4.4 (Optimistic Estimate)**.**
*Let $q : \Pi_I \times \Omega_D \to \{0, 1\}$ be a quality function. A quality function e_q is called an **optimistic estimate** of q, if and only if*

$$\forall P, P' \subseteq I : P \subseteq P' \Rightarrow e(\pi_P) \geq q(\pi_{P'})$$

Theorem 4.1 (Anti-Monotonicity of Optimistic Estimates)**.**
An optimistic estimate e_q of a quality function q is always anti-monotone.

Proof. Assume that e_q is not *anti-monotone*. Then there is a pair of *descriptive patterns* $\pi_P, \pi_{P'} \in \Pi_I$, with $P \subseteq P'$ and

$$e_q(\pi_P) < e_q(\pi_{P'})$$

Yet, because e_q is an *optimistic estimate* of q, it is known that

$$e(\pi_P) \geq q(\pi_{P'})$$

thus also that

$$q(\pi_{P'}) \leq e_q(\pi_P) < e_q(\pi_{P'})$$

which is a contradiction to

$$e_q(\pi_{P'}) \geq q(\pi_{P'})$$

resulting from the definition of an *optimistic estimate* ↯. □

Valuation Bases

Just as *pattern constraints*, *quality constraints* can be formulated using *valuation bases*. Thus, in the context of *pattern mining* the same is true for *quality functions*. Given a *valuation domain* $\mathbb{V} = (V, \oplus)$ *quality functions* are defined as

$$q : \Pi \times V \times V \to \mathbb{R}$$

where the first V represents the *valuation basis* v_P of the evaluated *pattern's* π_P projection S_{π_P} and the second V represents the *valuation basis* $v_\emptyset$ of the initial *data record set* S. Every *quality function* requires a characteristic *valuation domain* to work on. In Example 4.2 the *support quality function* is redefined accordingly.

Example 4.2 (Quality Functions and Valuation Bases)**.**
The ***support valuation domain*** *is defined as* $\mathbb{V} = (\mathbb{N}, +)$*. The support of a single individual is always one. The support quality function* q_{supp} *from Example 4.1 can be reformulated using the* ***support valuation domain****:*

$$\begin{aligned} q_{supp} : \quad & \Pi \times \mathbb{N} \times \mathbb{N} \to \mathbb{R} \\ & (\pi, n, n_0) \mapsto n \end{aligned}$$

The ***relative support quality function*** *is defined as*

$$\begin{aligned} q_{supp} : \quad & \Pi \times \mathbb{N} \times \mathbb{N} \to \mathbb{R} \\ & (\pi, n, n_0) \mapsto \tfrac{n}{n_0} \end{aligned}$$

4.1.2 Descriptive Pattern Mining Classes

When the only solution of a *pattern mining class* is the set of all valid *patterns*, then it can be defined by solely specifying *pattern constraints*. To allow *pattern constraints* to be independent of the *data domain* of the *data record sets* used as input, they are defined on *valuation bases*. Accordingly, a compatible *valuation basis* needs to be defined for each *pattern constraint*. The different *valuation bases* are then either concatenated by the carthesian product or aggregated in a more complex manner yielding a compressed *valuation domain*. The following section introduces common *descriptive pattern mining classes* by defining their characteristic set of *pattern constraints* based on *valuation bases* using the concept of *quality functions*. The first class is *frequent pattern mining* (Section 4.1.2), the second is *subgroup mining* (Section 4.1.2) and the third is *community mining* (Section 4.1.2).

Frequent Pattern Mining

In literature, **frequent pattern mining** refers to finding sets of items in a transaction database. An application is retail market basket analyzation. The goal is to find all articles that are often bought together. Originally, "often" means that the count of transactions, a set of items occurs in, has to surpass a certain threshold. Thus, for *frequent pattern mining* the sole *constraint* is the *quality constraint* $c_{supp \geq t}$ based on the *frequency quality function* q_{supp} from Example 4.2 and a support threshold t. The corresponding *valuation basis* is

$$\mathbb{V} = (\mathbb{N}, +)$$

Each *data instance* $d \in D$ is mapped onto a one. i.e. the corresponding $\mathbb{V}$*-valuation basis projector* is

$$\begin{aligned} \phi : \quad & D \to \mathbb{N} \\ & d \mapsto 1 \end{aligned}$$

Subgroup Mining

The goal of **subgroup mining** or **subgroup discovery** is to find "interesting" subgroups of individuals. It helps to identify relations between a dependent (target) variable and usually many independent variables, e.g. "the subgroup of 16-25 year old men that own a sports car are more likely to pay high insurance rates than the people in the general population", cf. [7]. It can also be used for classification [38]. The interestingness of a subgroup is evaluated by a *quality function*. Common *subgroup mining* tasks are to find all subgroups that are equal to or greater than a certain quality threshold or to mine the k most interesting subgroups. The former will be introduced here and can be extended to the latter by using the methods introduced in Section 4.2.

Subgroup mining is defined in different ways [8, 7, 6, 56]. Yet the notion of *subgroup description* is essentially the same (see Definition 4.5).

Definition 4.5 (Subgroup Description)**.**
A ***subgroup description*** $sd = \{e_1, e_2, \ldots, e_n\}$ *is defined by the conjunction of a set of selection expressions. These selectors* $e_i = (a_i, V_i)$ *are selections on domains of attributes,* $a_i \in \Omega_A, V_i \subseteq dom(a_i)$*.* Ω_{sd} *is the set of all possible subgroup descriptions.*

With the description $P \subseteq I$ being a subset of selectors I, the definition of a *subgroup description* is equivalent to I*-descriptive patterns* (see definition

2.9), with I being the set of all possible selectors (or at least all selectors used in a particular subgroup mining instance).

While there are different target types, the most common is a binary one: a *data instance* $d \in D$ associated with an *individual* $r \in \mathbb{X}$ from a *population* $\mathbb{X}$ can either be a positive sample, i.e. $\phi_T(d) = 1$, or a negative one, i.e. $\phi_T(d) = 0$; where $\phi_T : D \to \{0, 1\}$ is a function discerning positive and negative samples according to the binary target T.

Now, given a data record set $S = (R, \delta)$, the common primitives for *quality functions* based on a binary target T are (cf. [8]):

- $n(S) = |S|$,
  count of all *data instances* in a *data record set* S
- $tp(S) = \sum_{i \in R} \phi_T(\delta(i))$,
  count of all positive *data instances* in a *data record set* S
- $fp(S) = \sum_{i \in R} (1 - \phi_T(\delta(i))) = n(S) - tp(S)$,
  count of all negative *data instances* in a *data record set* S
- $p(S) = \frac{tp(S)}{tp(S)+fp(S)} = \frac{tp(S)}{n(S)}$,
  coverage of T by all *data instances* in a *data record set* S

The primitives fp and p can be derived from n and tp, thus they can be subsumed by the *valuation domain*

$$\mathbb{V} = (\mathbb{N} \times \mathbb{N}, +)$$

Each *data instance* is associated with a zero or a one depending on being a positive or negative example and a count of one. The *V-valuation basis projector* is

$$\begin{aligned} \phi_V : \quad & D \to \mathbb{N} \times \mathbb{N} \\ & d \mapsto (\phi_T(d), 1) \end{aligned}$$

Thus, a *valuation basis* $v \in \mathbb{V}$ for a *data record set* S is represented by a tuple $v = (tp(S), n(S))$, holding the count of all positive *data instances* $tp(S)$ and the total count of *data instances* $n(S)$.

Three exemplary *quality functions* are the binomial test q_{bin} (cf. [35, 36]), the *weighted relative accuracy* q_{WRAcc} (cf. [38, 8]) and a variation of the latter, the *Piatetsky-Shapiro quality function* q_{PS} (cf. [30]).

Definition 4.6 (Subgroup Mining: Quality Functions)**.**
Let

$$p = \frac{tp}{tp + fp} = \frac{tp}{n}, \text{ with } fp = n - tp$$

and

$$p_0 = \frac{tp_0}{tp_0 + fp_0} = \frac{tp_0}{n_0}, \text{ with } fp_0 = n_0 - tp_0$$

tp_0 and n_0 represent the tp and p values for a reference valuation basis, i.e. the valuation basis for the initial data record set. Then

- *the **binomial test quality function** q_{bin} is defined as*

$$\begin{aligned} q_{bin} : \quad & \Pi \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{R} \\ & (\pi, (tp, n), (tp_0, n_0)) \mapsto \frac{p - p_0}{\sqrt{p_0(1-p_0)}} \sqrt{n} \sqrt{\frac{n_0}{n_0 - n}} \end{aligned}$$

- *the **weighted relative accuracy quality function** q_{WRAcc} is defined as*

$$\begin{aligned} q_{WRAcc} : \quad & \Pi \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{R} \\ & (\pi, (tp, n), (tp_0, n_0)) \mapsto \frac{n}{n_0}(p - p_0) \end{aligned}$$

- *and the **piatetsky-shapiro quality function** q_{PS} is defined as*

$$\begin{aligned} q_{PS} : \quad & \Pi \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{R} \\ & (\pi, (tp, n), (tp_0, n_0)) \mapsto n \cdot (p - p_0) \end{aligned}$$

It is possible to derive *optimistic estimates* from these *quality functions*. Definition 4.7 formalizes the *optimistic estimate* for the *Piatetsky-Shapiro quality function* from Definition 4.6 [56]. This *optimistic estimate* is not tight [30], i.e. the *optimistic estimate* of the current *pattern* is not necessarily equal to the quality of one of its extensions.

Definition 4.7 (Piatetsky-Shapiro Optimistic Estimate)**.**

$$\begin{aligned} e_{PS} : \quad & \Pi \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{R} \\ & (\pi, (tp, n), (tp_0, n_0)) \mapsto n \cdot (1 - p_0) \end{aligned}$$

Analogously *valuation bases* and *quality functions* can be formulated when using a continuous target variables as in [5] by replacing the *tp* value in the *valuation basis* with the sum of the values of the continuous target.

Community Mining

A community is intuitively introduced by [6] as a "a group of individuals C out of a population U such that members of C are densely "related" to one another but sparsely "related" to individuals in $U \backslash C$". An example are friend or interaction graphs defined on users of a social network. Communities can be described by features the individuals share, e.g. tags or topics the individuals are associated with. The task of community mining is to find descriptions for communities that have a high quality according to some *quality function*, e.g. are strongly connected, are of a certain size, etc.

These concepts are transfered into graph theory (cf. [6]). A *graph* is an ordered tuple $G = (U, E)$, where U is a set of *vertices* and E is a set of *edges* between *vertices*. An *undirected graph* defines *edges* $e \in E$ as subsets of U containing exactly two elements ($e \in E \Rightarrow e \subseteq U \wedge |e| = 2$). A *directed graph* defines edges as tuples of *vertices* $E \subseteq U \times U$. In both cases the notation is $(u, v) \in E$. The *degree* d of a *vertex* is the number of (outgoing) connections it has to other vertices, i.e. $d(v) = |\{(v, x) \mid (v, x) \in E\}|$. An *adjacency matrix* $A \in \mathbb{R}^{n \times n}$ is a binary matrix associated with a subset of vertices $U' \subseteq U$ of a *graph* $G = (U, E)$ with $|U'| = n$. An entry $A_{i,j} = 1$ if and only if a connection from *vertex* $i \in U'$ to *vertex* $j \in U'$ exists in the set of *edges* E, i.e.

$$A_{i,j} = \begin{cases} 1, i, j \in U' \wedge (i, j) \in E \\ 0, \mathit{else} \end{cases}$$

Further notations are:

- $n := |U|$
- $m := |E|$
- $n_C := |C|$
- $m_C := |\{(u, v) \in E | u, v \in C\}|$
- $\bar{m}_C := |\{(u, v) \in E | u, v \notin C\}|$

Using this formalization of communities, [6] states two quality functions for a given community C: the *inverse conductance* ($ICON$, cf. [41]) as introduced in Definition 4.8 and the *modularity* (MOD, cf. [43]) as introduced in Definition 4.9. [6] also derives *optimistic estimates* for both functions. The *optimistic estimate* of *modularity* is given by Definition 4.10. For the *optimistic estimate* of *conductance* one may refer to [6].

Definition 4.8 (Conductance)**.**

$$\begin{aligned} CON(C) = &\quad \tfrac{\bar{m}_C}{2m_C+\bar{m}_C} = 1 - \tfrac{2m_C}{\sum_{u\in C} d(u)} \\ ICON(C) = &\quad 1 - CON(C) = \tfrac{2m_C}{\sum_{u\in C} d(u)} \end{aligned}$$

Definition 4.9 (Modularity)**.**

$$MOD(C) = \frac{1}{2} \cdot \sum_{i\in C, j\in C} (A_{i,j} - \frac{d(i)\cdot d(j)}{2m}) = \frac{m_C}{m} - \sum_{i\in C, j\in C} \frac{d(i)\cdot d(j)}{4m^2}$$

Definition 4.10 (Modularity: Optimistic Estimate)**.**

$$oe(MOD(C)) = \begin{cases} 0.25 & , \textit{ if } m_C \geq \frac{m}{2} \\ \frac{m_C}{m} - \frac{m_C^2}{m^2} & , \textit{ otherwise} \end{cases}$$

The data used to obtain communities descriptions is the graph $G = (U, E)$, that connects individuals, and a function $f : U \to 2^I$, that assigns certain features $I' \subseteq I$ to the individuals from a set of features I. Those two data sources are merged to a *data record set*. Each *individual* in the *data record set* corresponds to an edge between individuals from the graph. The edge is furthermore associated with those features that the individuals it connects share. Thus, the corresponding *data domain* D is

$$D = E \times 2^I$$

The problem of mining community descriptions can now be formulated as a *descriptive pattern mining* instance. Let $S = (R, \delta)$ be a *data record set*, where each *data record* corresponds to an edge with the features attached that the corresponding individuals share. Thus, the *data domain* $D = E \times 2^I$ represents all possible edges together with all possible combinations of features. Those features correspond to items in *descriptive pattern mining*. The respective *I-item projector* $\phi_{I,\text{com}}$ is defined as

$$\begin{aligned} \phi_{I,com} : &\quad E \times 2^I \to 2^I \\ &\quad ((u, v), I) \mapsto I \end{aligned}$$

A *valuation basis* that can be used to calculate any of the above *quality functions* needs to provide the size of the current community m_C and the *degrees* of all its individuals $d(i)$. Thus, an appropriate *valuation domain* can be defined as the powerset of individuals 2^U in a population U. A single

valuation basis then corresponds to the set of all individuals in a community $C \in 2^U$. Thus, the *valuation domain* is

$$\mathbb{V} = (2^U, \cup)$$

The matching *V-valuation basis projector* $\phi_{V,com}$ is defined as

$$\begin{aligned} \phi_{V,com} : \quad & E \times 2^I \to 2^U \\ & ((u,v), I) \mapsto \{u, v\} \end{aligned}$$

The community quality functions *conductance* and *modularity* as well as its *optimistic estimate*, can be reformulated as *quality functions* based on *valuation bases* according to Section 4.1.1. Let U be the set of individuals. Then the first instance of 2^U is the individuals associated with the *descriptive pattern* π_P and second the instance of 2^U is a reference set of individuals, i.e. the set of all individuals U, then

$$\begin{aligned} q_{ICON} : \quad & \Pi \times 2^U \times 2^U \to \mathbb{R} \\ & (\pi, U, C) \mapsto \frac{2|C|}{\sum\limits_{u \in C} d(u)} \end{aligned}$$

$$\begin{aligned} q_{MOD} : \quad & \Pi \times 2^U \times 2^U \to \mathbb{R} \\ & (\pi, U, C) \mapsto \frac{|C|}{|U|} - \sum_{i \in C, j \in C} \frac{d(i) \cdot d(j)}{4|U|^2} \end{aligned}$$

$$\begin{aligned} e_{MOD} : \quad & \Pi \times 2^U \times 2^U \to \mathbb{R} \\ & (\pi, U, C) \mapsto \begin{cases} 0.25 & , \textit{ if } |C| \geq \frac{|U|}{2} \\ \frac{|C|}{|U|} - \frac{|C|^2}{|U|^2} & , \textit{ otherwise} \end{cases} \end{aligned}$$

Note 4.1 (Valuation Basis Size)**.**
The size of valuation bases grows with respect to the size of the individuals from the graph. When using the GP-Tree data structure, the required space grows exponentially.

4.1.3 Condensed Itemset Mining

[48, 57] state that *frequent pattern mining* often produces a huge number of *patterns*, which reduces efficiency and effectiveness of mining. [48] also states that approaches addressing this problem can be classified into two categories. One is using *constraints* to capture the user's focus. Works are mentioned

that push constraints deep into the mining process [37, 44, 49]. The other category is to explore concise representations of frequent *patterns*. Papers like [50, 57, 46] are listed. While the latter category is kept separate from the former, the search for concise representations of *patterns* can also be expressed as *constraints*. There are quite a few algorithms for mining more compact representation of itemsets or including them into their constraint repository [54, 50, 59, 55, 47, 53, 22]. The more general field of computing closure system also provides means to be applied to *closed descriptive pattern mining* [28]. This section introduces a notion of *closed descriptive patterns* abiding by the framework of *descriptive pattern mining*. Furthermore, it gives a short overview over other *condensed descriptive pattern* representations.

Closed Descriptive Patterns

A *descriptive pattern* π_P is considered *closed* if there exists no extension $\pi_{P'}$ (i.e. $P \subset P'$) with the same support (i.e. $|S_{\pi_P}| = |S_{\pi_{P'}}|$). Definition 4.11 formally states what a *closed descriptive pattern* is. Definition 4.12 identifies the *pattern constraint* c_{closed} for only mining *closed descriptive patterns*.

Definition 4.11 (Closed Descriptive Pattern)**.**
Given a data record set S, an I-descriptive pattern π_P is called ***closed****, if and only if*

$$\forall\ (P' \supset P) : |S_{\pi_{P'}}| < |S_{\pi_P}|$$

or given an item projector ϕ_I:

$$P = \bigcap_{i \in \bar{\pi}_P(S)} \phi_I(\delta(i))$$

Definition 4.12 (Closed Constraint)**.**
Given a set of items I, the pattern-data constraint c_{closed} is defined as follows:

$$c_{closed} : \quad \Pi_I \to \{0, 1\}$$
$$\pi_P \mapsto \begin{cases} 1, \textit{ if } \quad P = \bigcap\limits_{i \in \bar{\pi}_P(S)} \phi_I(\delta(i)) \\ 0, \textit{ otherwise} \end{cases}$$

The *closed constraint* can also be formulated in terms of *valuation bases*. The corresponding *valuation domain* is

$$\mathbb{V} = (2^I, \cap)$$

The *valuation basis* for a single *individual* is equivalent to the items it is associated with. Thus, given an *I-item projector* ϕ_I the respective $\mathbb{V}$-*valuation basis projector* ϕ_V is

$$\begin{aligned} \phi_{V,closed}: \quad & D \to 2^I \\ & d \mapsto \phi_I(d) \end{aligned}$$

The reformulated *pattern constraint* based on the *valuation domain* $\mathbb{V} = (2^I, \cap)$ is

$$\begin{aligned} c_{closed}: \quad & \Pi_I \times 2^I \times 2^I \to \{0,1\} \\ & (\pi_P, P', P_0) \mapsto \begin{cases} 1, \text{ if } & P = P' \\ 0, \text{ otherwise} \end{cases} \end{aligned}$$

As a result the *closed constraint* can be easily incorporated in any *descriptive pattern mining class*. But note that the size of such *valuation bases* generally grows with respect to the number of items in the *data record set*. Thus, the situation is similar to using the most general *valuation basis* based on multisets of *data instances* (see Section 2.3). For the *GP-Tree* data structure the required space grows exponentially. Additionally, the *valuation domain* is based on the union of itemsets. This operation can be expensive when used often.

As is, the *closed constraint* is not *anti-monotone*. Yet frequent itemset mining algorithms exist, that exploit other properties of *closed descriptive patterns* to optimize *closed descriptive pattern* mining. The TFP algorithm (cf. [54]) mainly uses two optimizations:

- *item merging*: upon encountering a *descriptive pattern* π_P and its *modification set* E_P, all items $i \in E_P$ that yield extensions $\pi_{P\cup\{i\}}$ with the same support (i.e. $|S_{\pi_P}| = |S_{\pi_{P\cup\{i\}}}|$) are removed from the *modification set* E_P and directly added to the *descriptive pattern* π_P. This yields a new *descriptive pattern* $\pi_{P\cup P'}$, where $P' = \{i \in E_P : |S_{\pi_P}| = |S_{\pi_{P\cup\{i\}}}|\}$. Any extension without these items would be rendered invalid by the *closed constraint* c_{closed}. This type of search space pruning was introduced by Section 3.2.1 as *direct modification*.

- *prefix-itemset skipping*: the current *descriptive pattern* π_P is checked against every *closed descriptive pattern*, that was already found. If a *closed descriptive pattern* $\pi_{P'}$ is found that is a superset of the current *descriptive pattern* π_P (i.e. $P \subseteq P'$) and has the same support (i.e. $|S_{\pi_P|} = |S_{\pi_{P'|}}$) then π_P and all its extensions are discarded. *Prefix-itemset skipping* can be formulated as an *dynamic anti-monotone constraint* thus supporting *branch pruning* and *modification set pruning*.

The former can be applied directly to the more general setting of *descriptive pattern mining* in combination with the *closed constraint* to speed up the search.

Prefix-itemset skipping is a more subtle matter and cannot be used for *descriptive pattern mining* in general without either using the *closed constraint* or limiting the *pattern constraint* types to be used. *Item merging* and *prefix-itemset skipping* were designed to work together without relying on the *closed constraint*. It is assumed that not all *closed descriptive patterns* are stored for *prefix-itemset skipping* to check against, but only those rendered valid with respect to the other *constraints* (otherwise it would be equivalent to first search for all *closed descriptive patterns* and the filter them with respect to the other *constraints*). This requires a restriction to *data constraints* and *monotone description constraints*, because some arbitrary *constraint* might render a *closed descriptive pattern* invalid which is needed for *prefix-itemset skipping*. In such a scenario it is possible that *descriptive patterns* are added to the result which are not *closed*. If only *data constraints* and *monotone description constraints* exist, then any reduction $\pi_{P'}$ of a discarded *pattern* π_P with the same support ($|S_{\pi_P}| = |S_{\pi_{P'}}|$) is either rendered invalid by the same *constraint* the *closed descriptive pattern* π_P was.

While storing all *closed descriptive patterns*, valid or not, is not an option, storing more of them can be an advantage: the more *closed descriptive patterns* are stored to check against, the more opportunities for search space pruning exist by virtue of *prefix-itemset skipping*. On the other hand, keeping the set of *closed descriptive patterns* to check during *prefix-itemset skipping* small is essential as each stored *pattern* needs to be compared to the newly created one. The TFP algorithm addresses this issue reducing the amount of *closed descriptive patterns* to check against by requiring a globally fixed *item* and *branching order*. This method is not favorable if the performance of the search depends on specialized *item* or *branching orders* (for example when using *branching orders* based on *optimistic estimates*, see Section 4.2). Furthermore, restricting the *constraint* types will restrict the respective algorithm to a subclass of *descriptive pattern mining*. As an alternative to these restrictions and in order to take advantage of the optimizations *prefix-itemset skipping* offers, nevertheless, the *closed constraint* can be used to ensure the correctness of the results. Yet, this comes at the cost of more complex *valuation bases* as was mentioned before.

Note 4.2 (Interpretation of Closed Itemsets)**.**
Closed descriptive pattern can be interpreted in different ways when combined with other constraints. One way is formalized by Definition 4.11: a descriptive pattern π_P is considered closed if if there exists no extension $\pi_{P'}$ (i.e.

*$P \subset P'$) with the same support (i.e. $|S_{\pi_P}| = |S_{\pi_{P'}}|$). Another way to interpret closed descriptive patterns is to define them with respect to the other constraints. In other words, a descriptive pattern π_P is considered closed, if there exists no **valid** extension $\pi_{P'}$ (i.e. $P \subset P'$) with the same support, i.e. $|S_{\pi_P}| = |S_{\pi_{P'}}|$; where "valid" refers to all but the closed constraint.*

Other Condensed Descriptive Pattern Representations

There are other *condensed descriptive pattern* types. A more general version of the *closed* and the *maximal descriptive patterns* are *δ-closed descriptive patterns* [21, 22] as formalized by Definition 4.13. *Closed descriptive patterns* are 0-*closed descriptive patterns*.

Definition 4.13 (δ-Closed Descriptive Pattern)**.**
*Given a data record set S, an I-descriptive pattern π_P is called **closed**, if*

$$\forall\ (P' \supset P) : |S_{\pi_{P'}}| < (1 - \delta) \cdot |S_{\pi_P}|$$

Another *condensed descriptive pattern* type are *maximal descriptive patterns* [9] as introduced by definition 4.14. While any reduction of a *maximal descriptive patterns* is frequent their extensions are not. Thus, *maximal descriptive patterns* constitute a border between frequent and infrequent *descriptive patterns* (cf. [22]).

Definition 4.14 (Maximal Descriptive Pattern)**.**
*Given a data record set S and a frequency threshold t, a I-descriptive pattern π_P is called **maximal**, if*

$$\forall\ (P' \supset P) : |S_{\pi_{P'}}| < t$$

It is to be evaluated if *maximal* and *δ-closed descriptive pattern mining* can utilize anything but the most general *valuation domain* of multisets of *data instances* (see Section 2.3).

Other condensed representations like free [14, 15], simple disjunction-free [16, 17] and non-derivable [18] itemsets exist that are used to estimate or calculate the frequency of given itemsets, i.e. answer frequency queries [20]. [20] also mentions a unifying framework of the mentioned representations (cf. [19]). Further studies might help to integrate these approaches into the domain of *descriptive pattern mininig* or the other way around.

4.2 Result Constraints

Result constraints are *constraints* that restrict the *solutions* of a *pattern mining instance* beyond generating the set of all valid *patterns*. One example

are *coverage constraints* [22]. An instance is given by Example 3.2 which does not allow a *generative approach* to *solution discovery*. Other *result constraints* such as *top-k mining* do. *Top-K mining* is always associated with a *quality function* as introduced in Section 4.1.1. There are several interpretations of *top-k mining*. Let $\Pi_{valid} = \{\pi_1, \pi_2, \pi_3\}$ be a set of valid *patterns* and let their qualities be defined as $q(\pi_1, S) = 2$, $q(\pi_2, S) = 2$ and $q(\pi_3, S) = 1$, then

- one interpretation of *top-k mining* is finding a set R of k valid *patterns* such that no other valid *pattern* exists that has a higher quality than any of the *patterns* in the set, i.e. for top-1 mining $R_1 = \{\pi_1\}$ and $R_2 = \{\pi_2\}$ are solutions.
- A variation of the first interpretation is finding a tuple (R, X), where R is a set of k valid *patterns* such that no other valid *pattern* exists that has a higher quality than any of the *patterns* in the set R and X contains all *patterns* with a quality equal to the least quality in R, i.e. the solution of top-1 mining is $(R_1, X_1) = (\{\pi_1\}, \{\pi_1, \pi_2\})$ and the solution of top-2 mining is $(R_2, X_2) = (\{\pi_1, \pi_2\}, \emptyset)$.
- Another interpretation of *top-k mining* to find a set of patterns with the top k highest qualities, i.e. for top-1 mining $R = \{\pi_1, \pi_2\}$ is the only solution.

The former is by definition not well-defined. Several solution can exist. The latter two define a single solution. Nevertheless, the former solution is reviewed in the remainder of this section as it specifies a definite number of *patterns* in the solution guaranteeing a compact result. The other two *tok-k mining* variants cannot assure such a bound.

To formulate *top-k mining* in the context of *generative solution discovery* an *append function* α must be defined. The *append function* α_k for *top-k mining* is introduced by Definition 4.15.

Definition 4.15 (Append Function: Top-k and top-$\bar{k}$ mining)**.**
Given a quality function $q : \Pi \times \Omega_D \to \mathbb{R}$*, let*

$$q_{min}(R, S) := \underset{\pi \in R}{argmin}(q(\pi, S))$$

Then the append function α_k *for top-k mining is defined as*

$$\alpha_k : \quad \Pi \times \Omega_D \times 2^{\Pi} \to 2^{\Pi}$$

$$(\pi, S, R) \mapsto \begin{cases} R \cup \{\pi\} & , |R| < k \\ (R \setminus \{q_{min}(R, S)\}) \cup \{\pi\} & , q(q_{min}(R, S)) < q(\pi, S) \\ R & , otherwise \end{cases}$$

Just like *quality functions append function* can also be defined using the concept of *valuation bases*. Yet note that it is necessary to either store the quality of the *patterns* in the solution being built or the corresponding *valuation basis* to calculate the associated qualities in order to be able to derive the minimum quality in the current result.

4.2.1 Dynamic Threshold

So far *quality constraints* (see Section 4.1.1) have defined a *static threshold*. In combination with *top-k mining* such a threshold can be raised dynamically. The result are *generative constraints* (see Section 3.2.2). Such a *constraint* takes the the *solution* into account, that is being generated by using an *append function*. Upon reaching the size limit k the threshold is set to the lowest quality in the current *solution*. Only *patterns* with a higher quality will be added to the solution. *Generative quality constraints* based on *top-k mining* are formalized in Definition 4.16.

Definition 4.16 (Generative Quality Constraints)**.**
Given a solution R being built by a append function α and a quality function q, then a ***generative quality constraint*** *is a pattern constraint defined as*

$$c_{>R}: \quad \Pi_I \times \Omega_D \to \{0,1\}$$
$$(\pi_P, S) \mapsto \begin{cases} 1, \ \textit{if} \quad q(\pi, S) > \min\limits_{\pi \in R}(R, S) \\ 0, \ \textit{otherwise} \end{cases}$$

Just like *quality constraints*, *generative quality constraint* can be *anti-monotone* and consequently favor *branch* or *modification set* pruning. As the *append function* already takes care of building the correct solution, *generative quality constraint* are solely used for pruning. If a *quality function* used by a *top-k append function* is not *anti-monotone*, it is possible to use its *optimistic estimate* to formulate a *generative quality constraint*.

When using a *generative quality constraints* for *anti-monotone pruning* the efficiency of a search highly depends on the order in which *patterns* are being added. The sooner *patterns* with a high quality are added, the faster the threshold raises and consequently the search space can be pruned in larger quantities. A heuristic to speed up threshold raising is to define the *branching order* o_P^B (see Section 3.1.1) for each *descriptive pattern* π_P based on the *optimistic estimate* $e(\pi_{P\cup\{i\}})$ of the extension $\pi_{P\cup\{i\}}$ corresponding to the items $i \in E_P$ in the *modification set* E_P. The higher the *optimistic estimate*, the sooner the branch based on the corresponding item i is explored. This heuristic is mentioned in literature (cf. [5, 56]).

5

Implementation

This chapter initially describes the actual algorithm being implemented incorporating many features introduced in Chapter 3 and 4. The second part of this chapter deals with technical details and how the framework (Chapter 2) used to formulate the algorithm is implemented.

5.1 The AnEMonE Algorithm

Chapter 3 and Chapter 4 introduce many concepts and optimizations. The algorithm actually implemented tries to cover as many as possible while allowing a reasonable spectrum of configuration, i.e. which optimizations are to be enabled or disabled. The algorithm is based on *bottom-up*, *depth-first*, *pre-ordered* search as introduced in Section 3.1. The "baseline" algorithm is given by Listing 5.1. First all implemented features are described. Optional features are highlighted. The algorithm in Listing 5.1 id coined *AnEMonE* for **An**ti-Monotone, **E**xAnte and **Mo**dificatio**n** Set **E**xploitation. It has the following features:

- explicit usage of a combination of *component data record sets* and *GP-Trees* (see Section 3.2.3),
- exploiting the *single prefix path* (see Section 3.2.3): limited to *GP-Trees* that are *single prefix paths*, i.e. trees only containing nodes with a single child,
- *modification set pruning* based on *anti-monotone constraints* (see Section 3.2.1),

- exploiting the *ExAnte property* (see Section 3.2.1): as part of the *modification set pruning* procedure and
- *early adding* (see Section 3.2.2): as part of the *modification set pruning* procedure.

Listing 5.1: The *AnEMonE* algorithm.

```
1  VARIABLES:
2     R   // result
3     v_∅  // valuation basis of the initial data record set
4     C_Π_I  // all pattern constraints, implicitly set by initial call
5     o^I  // used to derive item order; implicitly set by initial call
6     o^B  // used to derive branching order; implicitly set by initial call
7     α  // append function; implicitly set by initial call
8     C_A  // anti-monotone pattern constraints
9     C_D  // dynamic pattern constraints
10
11 PROCEDURE anemone(C_Π_I, o^I, o^B, α, S, φ_I, φ_V):
12    R ← ∅
13    C_A ← extractAntimonotoneConstraints(C_Π_I)
14    C_D ← extractDynamicConstraints(C_Π_I)
15
16    (P, E_P, S_π_P, v_∅) ← createComponentDataRecordSet(S, φ_I, φ_V)
17    (E_P, S_π_P) ← modificationSetPruning(P, E_P, S_π_P)  // including: earlyAdding, exAnte
18    call anemone_rec(P, E_P, S_π_P)
19    return R
20
21 PROCEDURE anemone_rec(P, E_P, S_π_P):
22    o^I_P ← o^I(P, E_P, S_π_P, v_∅)  // derive item order
23    o^B_P ← o^B(P, E_P, S_π_P, v_∅)  // derive branching order
24
25    T ← buildGPTree(S_π_P, o^I_P)
26    (E_P, T) ← exploitSinglePrefixPath(E_P, T)  // only if the tree IS a single prefix path
27
28    E_P[] ← sort(E_P, o^B_P)  // set branching order
29    for i ∈ E_P[] do // branching abiding by branching order
30       P' ← P ∪ {i}  // create extension
31       v_P' ← getConditionalValuationBasis(S_π_P, i)
32       if !isValid(P', v_P', v_∅, C_A ∩ C_D) do // additional branch pruning (only dynamic constraints)
33          continue
34       endif
35
36       E_P' ← getConditionalModificationSet(E_P, o^I_P, i)
37       S_π_P' ← getConditionalComponentDataRecordSet(T, i)
38       (E_P', S_π_P') ← modificationSetPruning(P', E_P', S_π_P')  // including: earlyAdding, exAnte
39       call anemone_rec(P', E_P', S_π_P')
40    endfor
```

The algorithm from Listing 5.1 is called by invoking the

$$anemone(C_{\Pi_I}, o^I, o^B, \alpha, S, \phi_I, \phi_V)$$

procedure (Line 11). It requires several input values:

- C_{Π_I}, a set of *pattern constraints* (see Section 2.2 and Section 4.1.1 for *pattern constraints* based on *valuation bases*); note that *pattern constraints* can be *generative* (see Section 3.2.2),
- o^I, a function able to derive an *item order* $o^I(P, E_P, S_{\pi_P}) = o^I_P$ (see Section 3.1.1) based on the current *descriptive pattern* π_P, its *modification set* E_P, the associated data structure S_{π_P} and the *valuation basis* $v_\emptyset$ of the initial *data record set*,
- o^B, a function able to derive a *branching order* $o^B(P, E_P, S_{\pi_P}) = o^B_P$ (see Section 3.1.1 based on the current *descriptive pattern* π_P, its *modification set* E_P, the associated data structure S_{π_P} and the *valuation basis* $v_\emptyset$ of the initial *data record set*,
- α, an *append function* as defined in Section 3.1.2,
- $S = (R, \delta : \mathbb{X} \to D)$, the initial *data record set* on a *data domain* D (see Section 2),
- ϕ_I, an *I-item projector* on *data domain* D (see Section 2.1.1) and
- ϕ_V, an *V-valuation basis projector* on *data domain* D and *valuation domain* $\mathbb{V} = (V, \oplus)$ according to the *pattern constraints* in C_{Π_I} (see Section 2.3).

The called *anemone* procedure (Line 11) first initializes the result R, and extracts *anti-monotone* and *dynamic constraints* from the given *constraints* C_{Π_I}. Some of the input arguments are implicitly stored in the global variables with the same name. Line 16 prepares the initial *descriptive pattern* P (or rather its associated itemset), its *modification set* E_P, the corresponding *component data record set* S_{π_P} and the *valuation basis* $v_\emptyset$ of the initial *data record set* $S_{\pi_\emptyset}$. Here the initial *descriptive pattern* P is the empty *descriptive pattern* $P = \emptyset$. Note at this point, that the *Anemone* algorithm does not (as of yet) support arbitrary data structures, but relies on a combination of *component data records* and the *GP-Tree* structure as introduced in Section 3.2.3. The following line prunes the generated *modification set* (see Section 3.2.1) including *early adding* (see Section 3.2.2) and the exploitation of the *ExAnte property* (see Section 3.2.1). Three minor modifications might be interesting at this point

- incorporate the procedure

$$modificationSetPruning(P, E_P, S_{\pi_P})$$

into

$$createComponentDataRecordSet(S, \phi_I, \phi_V)$$

This allows the exploitation of the *ExAnte property* to be applied to the initial *data record set* S. Thus the *initial component data record set* S_{π_P} can be kept smaller. Also sharing other information might be favorable.

- skip further processing altogether if the initial *descriptive pattern* π_P does not satisfy some *anti-monotone constraints*
- add the empty *descriptive pattern* $\pi_\emptyset$ to the result if it satisfies every *pattern constraint* $c \in C_{\pi_I}$

Line 18 finally calls the recursive search. Afterwards the built solution R is returned.

The recursive search is represented by the procedure

$$anemone_{rec}(P, E_P, S_{\pi_P})$$

Its arguments are

- P, the itemset associated with the current *descriptive pattern* π_P,
- E_P, the (not pruned and not sorted) *modification set* E_P of the current *descriptive pattern* π_P and
- S_{π_P}, the *component data record set* (see Section 3.2.3) according to the current *descriptive pattern* π_P.

While other recursion layouts are possible (see Note 5.1), the $anemone_{rec}$ procedure begins with preparing the branching loop by deriving *item* and *branching order* according to the current *descriptive pattern* P, its *modification set* E_P and the corresponding *component data record set* S_{π_P} (to have access to the *valuation bases* $v_{\pi_{P\cup\{i\}}}$ of extensions $\pi_{P\cup\{i\}}$ used to derive an order on the items $i \in E_P$). The *GP-Tree* structure is being used, thus a common *item order* is the descending *frequency order*. When using *quality functions* a common *branching order* is based on the *optimistic estimates* of the extensions corresponding to *items* to sort. The next step is to prepare for the generation of *conditional component data record sets* by creating the *GP-Tree* T from the current *component data record set* by using the derived *item order* o_P^I. If the *GP-Tree* generated is a *single prefix path*, i.e. only contains nodes with a single child, it is exploited. Other exploitations of the *single prefix path* are possible, but not implemented here (see Section 3.2.3). Then the pruned *modification set* is sorted and used for the *branching loop*.

Note 5.1 (Recursion Layout)**.**
There are two prominent alternative recursion layouts besides the one chosen in the AnEMonE algorithm (see Listing 5.1).

- *One is to start the recursion with the branching loop without preparations, i.e. move Lines 22 through 28 to right before Line 39. Thus the inner loop is taking care of branching preparation such as modification set pruning, branching order, etc. before calling the recursion. This was avoided to keep the initial anemone(...) procedure as well as the the arguments passed to $anemone_{rec}(...)$ simpler.*

- *The other prominent layout is to relocate all preparation done in the branching loop located before the recursive call towards the beginning of the recursion, i.e. moving Lines 30 through 38 to right after Line 21. The disadvantage of this approach is that the conditional data like component data record set and valuation basis need to be extracted at the beginning of the recursion (i.e. before branching) requiring knowledge about the item currently used for extension. This would mean, that the branching loop needs to be replicated in the initial anemone(...) procedure.*

The current layout is a compromise between both variants. It also allows subvariants. It is possible to relocate the $modificationSetPruning(P, E_P, S_{\pi_P})$ *from the branching loop to the beginning of the recursion, i.e. move Line 38 to right after 21. This would save the call to modification set pruning in the initial anemone(...) procedure (Line 17). Yet as a result the incorporating the*

$$modificationSetPruning(P, E_P, S_{\pi_P})$$

into the

$$createComponentDataRecordSet(S, \phi_I, \phi_V)$$

procedure would not be possible without redundancy. Furthermore it is possible to enable the exploitation of the ExAnte property only for the initial data record set. Thus this setting was avoided to be consistent with possible implementations.

The branching loop (Lines 30 through 39) starts off with extending the current *descriptive pattern* P with the conditioning item i yielding the extended *descriptive pattern* P' and calculating / accessing its *valuation basis* $v_{P'}$. Lines 31 through 34 are an optimization when *dynamic anti-monotone constraints* are present. It is likely that in a previous loop cycle *descriptive patterns* were added to the result. Thus an extension based on the current

item i, that was not pruned from the *modification set* might now be invalid according to some *dynamic anti-monotone constraint*. Thus checking again here can help to improve the performance by *branch pruning*. Alternatively *modification set pruning* based on *dynamic constraints* can be used after each recursive call, i.e. after Line 39. Note that both optimizations introduce additional *constraint* checking. The latter more than the former. Additionally the latter can include *ExAnte property exploitation*, which potentially modifies the *component data record set* requiring the *GP-Tree* to be rebuilt. Thus the *AnEMonE* implementation goes with the former as a compromise. Afterwards the *conditional modification set* $E_{P'}$ is generated by applying the *item order* o_P^I to the current *modification set* E_P based on the item i used for extension (see Section 3.1.1). The next step is to generate the *conditional component data record* $S_{\pi_{P'}}$ from the *GP-Tree* T based on the item i used for extension. The new *modification set* $E_{P'}$ is then pruned and the *component data record set* adjusted accordingly using the extension P' and the corresponding *conditional component data record* $S_{\pi_{P'}}$ as input. Finally the $anemone_{rec}$ is called on the conditional data.

The following options are available:

- disable exploiting the *ExAnte property* (included in *modification set pruning*): can be done without side effects
- disable *single prefix path* exploitation: can be done without side effects
- disable *early adding* (included in *modification set pruning*): *descriptive patters* need to be added after *branch pruning* in the branching loop. This requires adding the code from Listing 5.2 after Line 34 checking the current extension P' against all constraints and adding it to the result, if it is rendered valid. As *static anti-monotone constraints* were already checked during *modification set pruning* they could be left out at this point to avoid redundant checking.

Listing 5.2: Code needed when *early adding* is disabled.

```
1 if isValid(P', v_P', v_0, C_Π_I) do
2   R ← α(P', v_P', v_0, R) // add pattern to result
3 endif
```

- disable *modification set pruning*: the *modification set pruning* procedures fall away completely (i.e. Lines 17 and 38 are removed) and is replaced by *branch pruning* based on *anti-monotone constraints*. This is implemented by checking all *anti-monotone pattern constraints* C_A instead of just *dynamic* ones at Line 32 and by replacing Line 38 by the code from Listing 5.2.

5.2 Framework

The implementation is done in Java (*http://java.com*), version 1.6. It is based on the framework introduced in Chapter 2 and uses the notion of *valuation bases*. Data access is granted by the interface *IDataRecordSet* corresponding to the notion of a *data record set*, which allows iteration over the *IDataInstance* interface. The *IDataProjector* interface allows projections of a *IDataInstance* onto arbitrary objects. Projections onto *valuation bases* (interface *IValuationBasis*) and items (interface *IItem*) are available through the interfaces *IValuationBasisProjector* and *IItemProjector*. Several implementations according to the mining task have been written. A schematic class diagram is given by Figure 5.1.

To specify a mining task, several interface were defined. The most basic interfaces are *IPatternConstraint* and *IResult*. The *IPatternConstraint* interface refers to *pattern constraints* that also taking projected *data record sets* in the form of *valuation bases* as an argument for evaluation as discussed in Section 3.2.3. The initial *data record set* in the form of its *valuation basis* has to be set for each constraint before the search starts. The *IResult* interface holds the solution of the mining task after the search. It incorporates the *append function* introduced in Section 3.1.2. Figure 5.2 depicts both interfaces. Several interfaces and classes were defined and implemented to easily formulate *constraints* including the *IQualityFunction* interface and the *QualityThresholdConstraint* class according to Section 4. The two most prominent implementations of *IResult* are the *EveryResult* and *TopKResult* classes also depicted in Figure 5.2.

The implementation of the *Anemone* algorithm is a straight forward translation of the algorithm from Section 5.1 using the interfaces and classes introduced above. It can be configured to take any number of *pattern constraints* and an arbitrary *item order* and as well as an arbitrary *branching order*. Also it allows to turn features like *branch pruning*, *modification set pruning*, *ExAnte* and *single prefix path exploitation* on and off in appropriate combinations. After the configuration step, the algorithm is called by providing a result *IResult* and a *DataComponents* class. The latter contains an *IDataRecordSet*, an *IItemProjector* and an *IValuationBasisProjector* and is translated into a *component data record set*, that is a *IDataRecordSet* containing *ComponentDataInstances*. *ComponentDataInstances* contain a list of *IItems* and a *IValuationBasis* according to Section 3.2.3. A schematic class diagram is given by Figure 5.3. As of now the data structure used in the algorithm itself is not abstracted as suggested by Section 3.2.3, but uses a combination of *component data record sets* and a *GP-Tree* for a data struc-

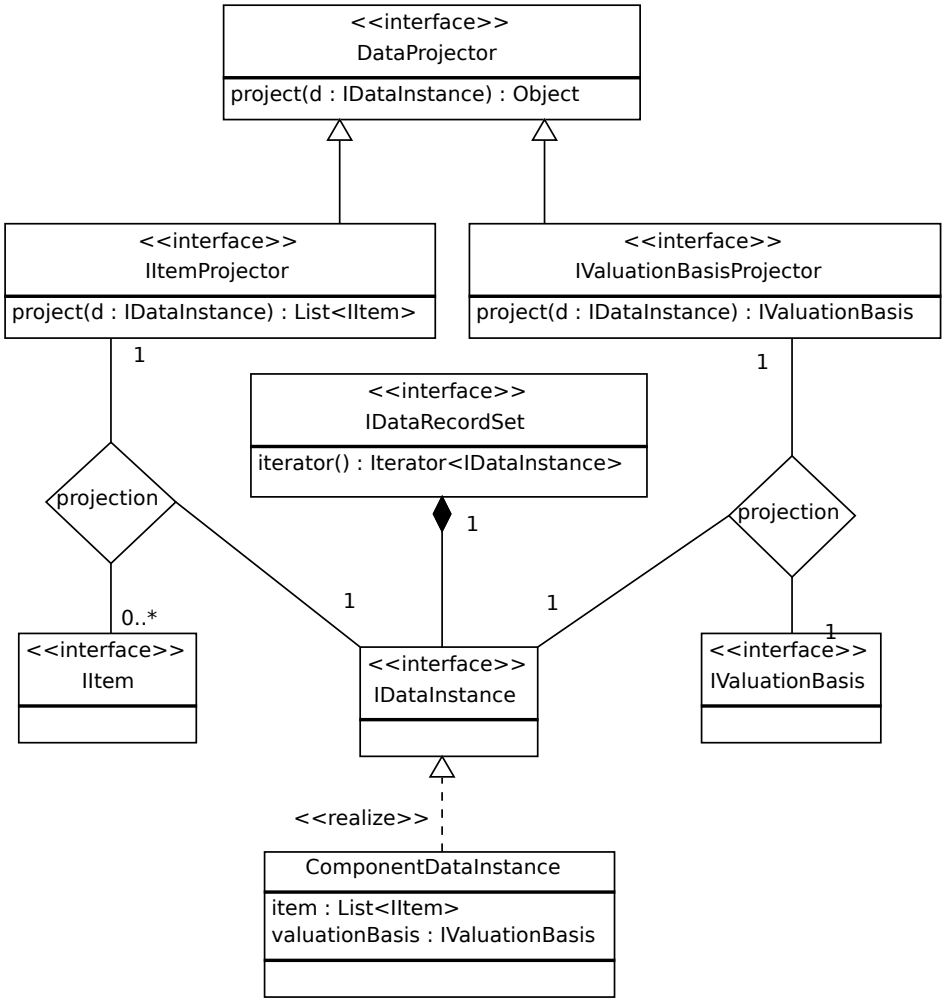


Figure 5.1: A schematic class diagram of the basic interfaces.

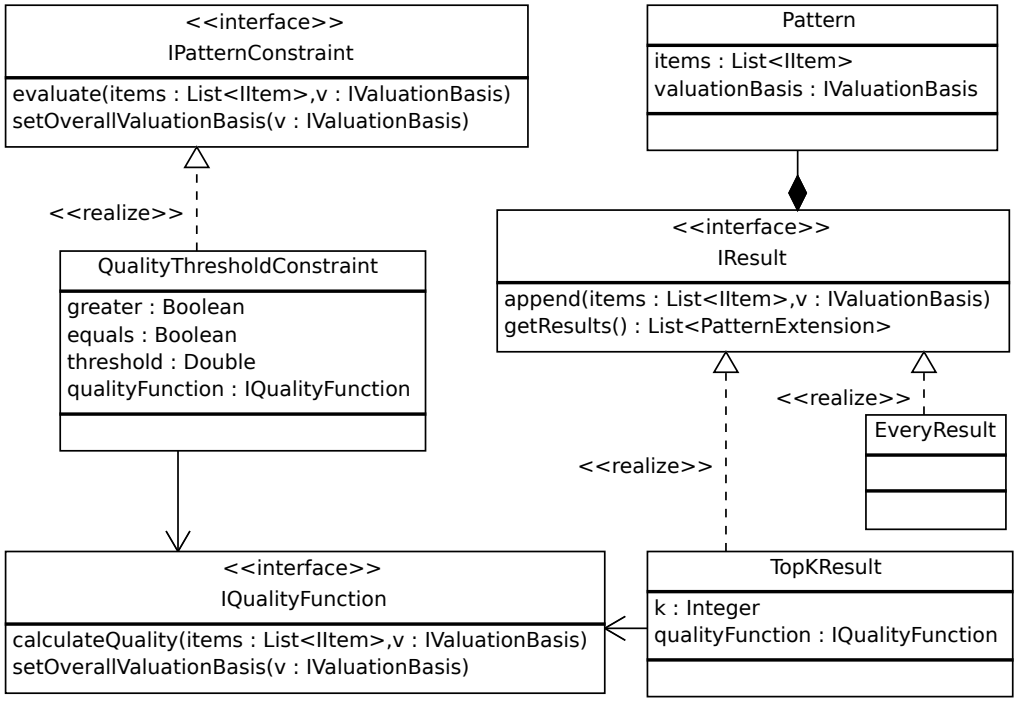


Figure 5.2: A schematic class diagram of search specific interfaces and classes.

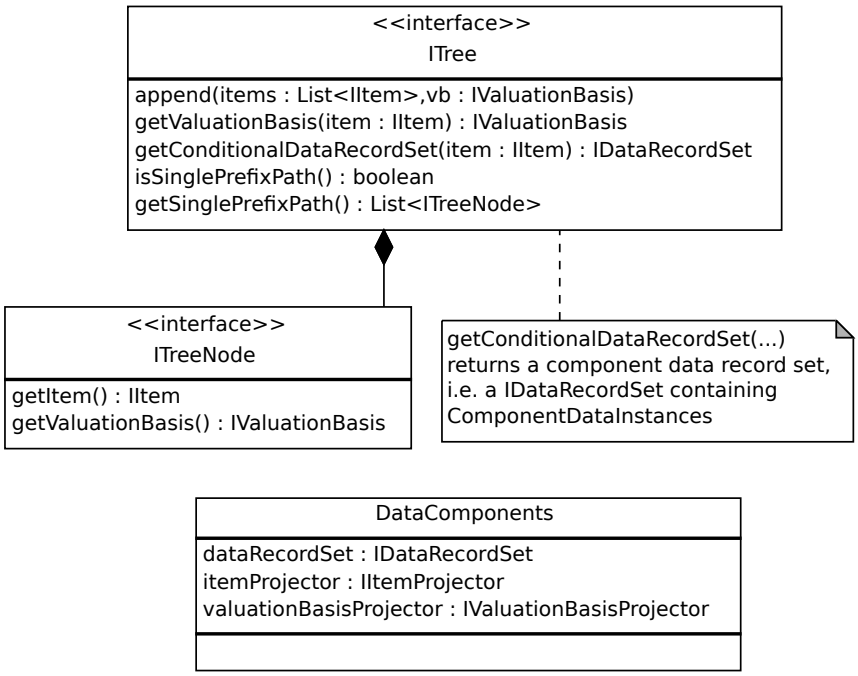


Figure 5.3: A schematic class diagram of data specific interfaces and classes.

ture. *ITree* provides the interface required by a *GP-Tree* implementation as given by Figure 5.3.

As the framework is implemented now, no Generics are being used. This results in many castings done by classes that need certain kinds of *IDataInstances*, *IItems* and *IValuationBases*, such as *IDataProjectors* or *IQualityFunctions*. Thus also no type checking can be applied beforehand. For more efficiency and better configuration error detection the framework needs to be adjusted accordingly, i.e. Generics need to be introduced.

6

Evaluation

By specifying the *AnEMonE* algorithm to solve the generic problem of *descriptive pattern mining* it can be applied to any *descriptive pattern mining instance* abiding by the framework introduced in Chapter 2. The *AnEMonE* algorithm at its minimal optimization level only uses *branch pruning* based on *anti-monotone constraints* (see Section 3.2.1). It uses a reversed frequency order as *item* as well as *branching order*, i.e. given a *descriptive pattern* π_P and its *modification set* E_P, items $i \in E_P$ associated with an extensions $\pi_{P\cup\{i\}}$ of higher frequency are assigned smaller *modification sets* $E_{P\cup\{i\}}$ and are branched on earlier. This setup of the *AnEMonE* algorithm is labeled *min*. The same algorithm, but exploiting *anti-monotone* properties using *modification set pruning* instead of *branch pruning* is labeled *mod*. The other options mentioned in Section 5.1 and their combination with the different settings of the algorithm are listed in Table 6.1.

This chapter will compare the different algorithm settings. They are applied to the *descriptive pattern mining classes* defined by Section 4.1.2, i.e. *frequent pattern mining* [4], *subgroup mining* [56, 8] and *community mining* [6]. This includes the possibility to exploit *ExAnte property* for *subgroup mining* which has not been done before.

The main characteristics to be compared in order to measure the performance differences when using different combinations of optimization methods are the runtime and the number of conditional trees (the tree built from the initial *data record set* is not conditional) being generated during the search. At this point, note that any runtime depicted in this section is based on a single run. Characteristic runs have been selected. At times the Java's JIT compilation feature randomly introduced large deviations from the ex-

<table>
<tr><th></th><th>min</th><th>mod</th></tr>
<tr><td>spp</td><td colspan="2">single prefix path exploitation as introduced in Section 3.2.3 (only for GP-Trees which are a single prefix path as described by Section 5.1)</td></tr>
<tr><td>early</td><td>early adding is not used together with branch pruning because when adding descriptive patterns early, all constraints need to be checked including anti-monotone ones; thus modification set pruning can be applied without extra cost</td><td>early adding as introduced by Section 3.2.2; instead of adding descriptive patterns to the result at their respective nodes they are added right after modification set pruning in order to provide information for dynamic constraints faster</td></tr>
<tr><td>exInit</td><td rowspan="3">ExAnte property exploitation is not used in combination with branch pruning as exploiting the ExAnte property requires checking the extensions of the items from the current modification set against anti-monotone constraints thus enabling modification set pruning</td><td>exploits the ExAnte property (see Section 3.2.1) on the initial data record set only</td></tr>
<tr><td>exRec</td><td>exploits the ExAnte property (see Section 3.2.1) on all but the initial data record set</td></tr>
<tr><td>exFull</td><td>always exploits the ExAnte property (see Section 3.2.1)</td></tr>
<tr><td>estim</td><td colspan="2">top-k mining (see Section 4.2) defines a quality function; the corresponding optimistic estimate can be used for branch sorting; those items associated with patterns assigned a higher optimistic estimate value are branched on earlier</td></tr>
</table>

Table 6.1: Options of the *AnEMonE* algorithm used in this chapter.

pected runtimes. In order to verify the correctness of the conclusions being drawn, several tests were run with the JIT compilation feature turned off. A performance loss up to a factor of 20 has been observed. Nevertheless it was possible to derive accurate statements due to the relative nature of comparing the different settings.

The algorithm was run on a 2.2GHz Dual Core system with 4GB of RAM. The operating system in use was a 64bit version of Ubuntu 11.04 with a SWAP partition of 3.9GB. The algorithm was implemented using Java 1.6.9_22 (the results of the `java -version` command is stated by Listing 6.1). The virtual memory for a JVM instance was set to `2048m`.

Listing 6.1: Version of the Java Virtual Machine used for experiments.

```
1 java version "1.6.0_22"
2 OpenJDK Runtime Environment (IcedTea6 1.10.1) (6b22-1.10.1-0ubuntu1)
3 OpenJDK 64-Bit Server VM (build 20.0-b11, mixed mode)
```

Section 6.1 will cover the settings represented by the first five rows from Table 6.1. Section 6.2 evaluates the last row, i.e. sorting branches by their *optimistic estimates*. After covering two *descriptive pattern mining classes* Section 6.3 illustrates how a third *descriptive pattern class* can benefit from a selected set of optimizations.

6.1 Frequent Itemset Mining

If the original definition [4] is used, *frequent pattern mining*, as introduced in Section 4.1.2, is one of the simplest *descriptive pattern mining* classes. It only introduces a single constraint: the *frequency constraint*, i.e. the projection of a *descriptive pattern* has to be equal to or exceed a certain size to be considered valid. The goal of *frequent pattern mining* can either be to find all *descriptive patterns* exceeding a fixed threshold or to find the top most frequent *descriptive patterns* resulting in a dynamically increasing threshold. The latter setup is equivalent to *top-k mining* using a *generative quality constraint* based on the *frequency quality function* (see Section 4 and Section 4.2 in particularly). Both scenarios are explored (see Section 6.1.1 and 6.1.2).

This section will focus on comparing *branch pruning* with *modification set pruning* additionally taking *single prefix path exploitation* into account. Also, when specifying *dynamic constraints* (e.g. when searching for the top most frequent *descriptive patterns*) adding *patterns* to the result early can influence the performance of the search. The corresponding effects are highlighted by Section 6.1.2. Finally a *monotone description constraint* is introduced and the exploitation of the *ExAnte property* is evaluated (see Section 6.1.3). Considering Table 6.1 this section covers rows one through five.

mushroom dataset	
attributes	22
instances	8124
attribute-value pairs (items)	119

(a) Information about the *mushroom* dataset.

retail dataset	
instances	88162
items	16470

(b) Information about the *retail* dataset.

Figure 6.1: Information about the *mushroom* and the *retail* dataset.

Two datasets are used by the experiments listed in this section. These are the *mushroom* and the *retail* dataset. The *mushroom* dataset is from the UCI Machine Learning Repository [27] and is an attribute-value based dataset. It contains descriptions of mushrooms based on attributes including a classification rendering them "edible", "poisonous" or "unknown". Information about its dimensions is given by Figure 6.1(a). The *retail* dataset is from the Frequent Itemset Mining Dataset Repository [1] and is item-based, i.e. it contains transactions each corresponding to a set of items. The data corresponds to retail market basket data from a Belgian retail store. Information about its dimensions is given by Figure 6.1(b).

6.1.1 Relative Frequency Threshold

The experiments in this section compare *branch pruning* with *modification set pruning* in the presence of a fixed frequency threshold. The frequency threshold is chosen relatively to the size of the dataset (amount of individuals). In addition the effect of exploiting the *single prefix path* is evaluated.

The *mushroom* Dataset

Figure 6.2 shows the number of valid *descriptive patterns* at different relative frequency thresholds for the *mushroom* dataset. Note the logarithmic scale! When decreasing the threshold linearly, the amount of results grows exponentially.

Figure 6.3 shows the amount of conditional trees (all *GP-Trees* except the first one) generated by the algorithm. It also depicts their accumulated size at different relative frequency thresholds. The amount of conditional trees is mainly affected by the exploitation of the *single prefix path*. This is the case even though the exploitation of *single prefix paths* is limited to trees that are *single prefix paths*, i.e. trees only containing nodes being part of a *single prefix path*. This is due to the fact that the number of conditional trees generated from a *single prefix path*, without exploiting it, grows exponentially

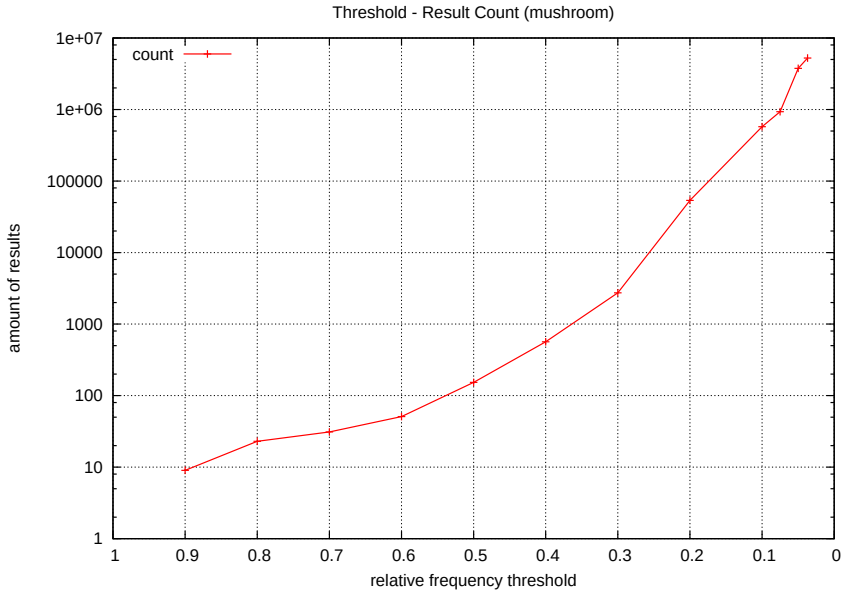
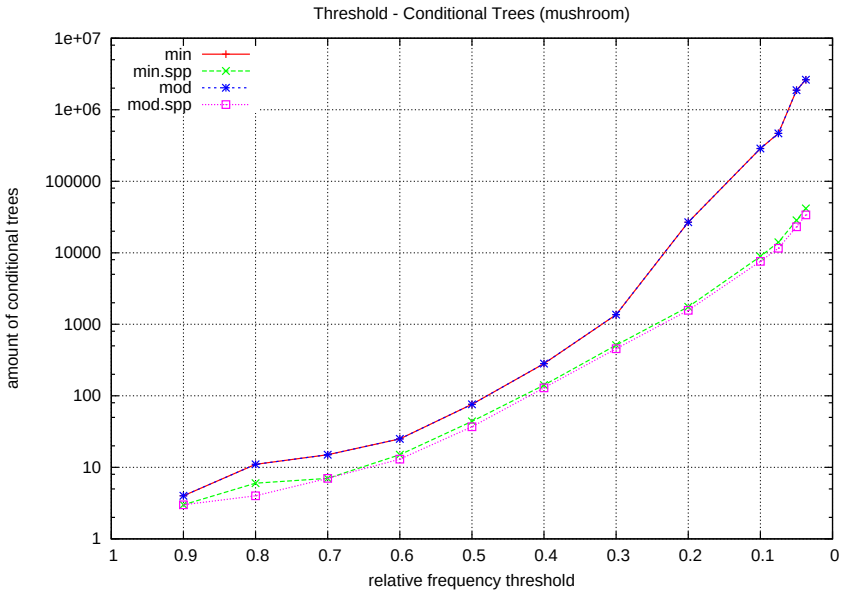


Figure 6.2: Number of frequent *descriptive patterns* in the *mushroom* dataset at different relative frequency thresholds.

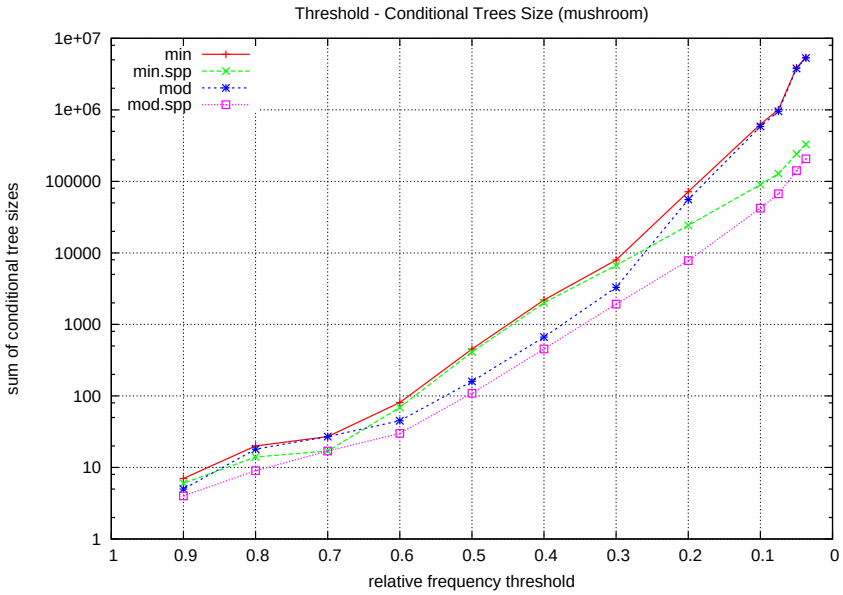
with respect to its size. Thus, skipping those trees in large scales can account for better runtimes as Figure 6.4 depicts. In general *modification set pruning* also helps to reduce the amount of generated conditional trees. In this case the effect is only marginal.

Note the development of the sum of the conditional tree sizes for the *min.spp* and the *mod* setting. Depending on the threshold, the accumulated size of all conditional trees is sometimes larger for *min.spp* and sometimes for *mod*. The change is especially significant between thresholds 0.2 and 0.3 (note the logarithmic scale). At this point the amount of exploited *single prefix paths* and the consequently saved exponential amount of conditional trees significantly surpasses the effect of *modification set pruning* alone. Such characteristic developments are generally due to the structure of the dataset. The dominance of the effect of *single prefix path* exploitation might hint at a dataset were single items occur very often and if less often, then likely together with a more probable item, thus, raising the probability for *single prefix paths*. This hypothesis is backed by the low amount of different attribute-value pairs considering the size of the dataset (the attribute-value pairs to dataset size ratio is around 1.4 percent).

Figure 6.4 shows the runtimes of the algorithm on the *mushroom* dataset for different relative frequency thresholds. As expected from the analysis



(a) Amount of conditional trees.



(b) Sum of the sizes of the conditional trees.

Figure 6.3: Amount of conditional trees and the sum of their respective sizes generated by the *AnEMonE* algorithm on the *mushroom* dataset at different relative frequency thresholds.

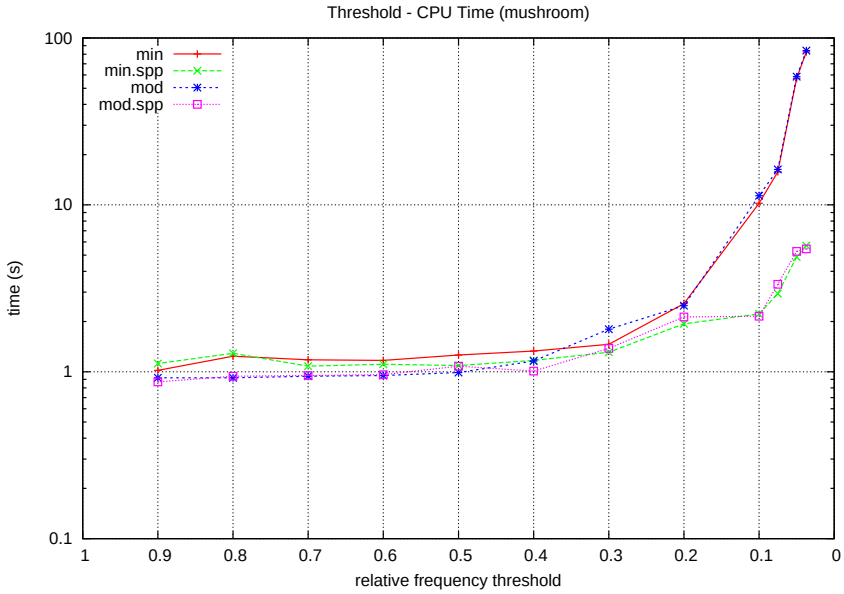


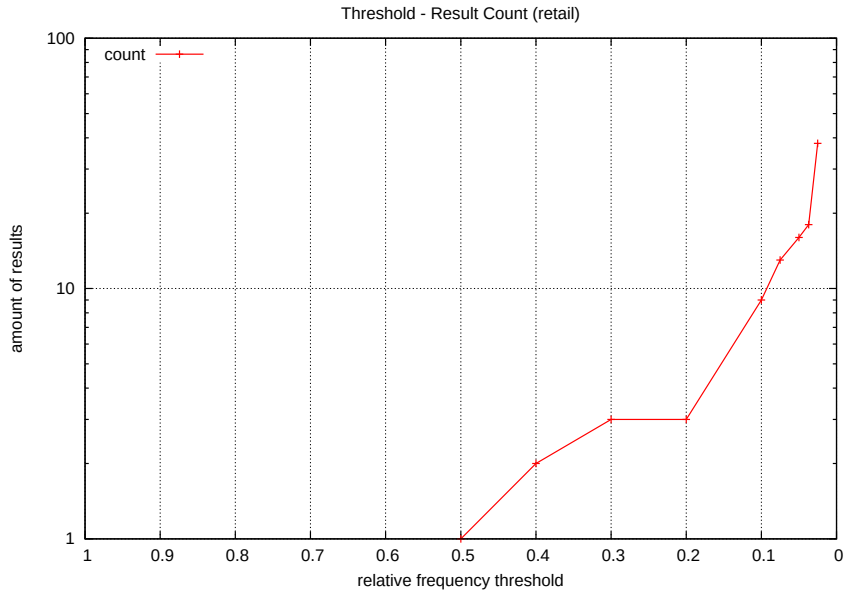
Figure 6.4: Runtimes of the *AnEMonE* algorithm on the *mushroom* dataset at different relative frequency thresholds.

above, *branch pruning* and *modification set pruning* hardly differ, while exploiting *single prefix paths* greatly improves the performance at lower threshold levels. Not expecting the implementation to be too efficient, the runtimes can still compare to, yet not quite compete with, the results of Frequent Itemset Mining Implementations 2003 and 2004 [29, 2], when exploiting *single prefix paths*.

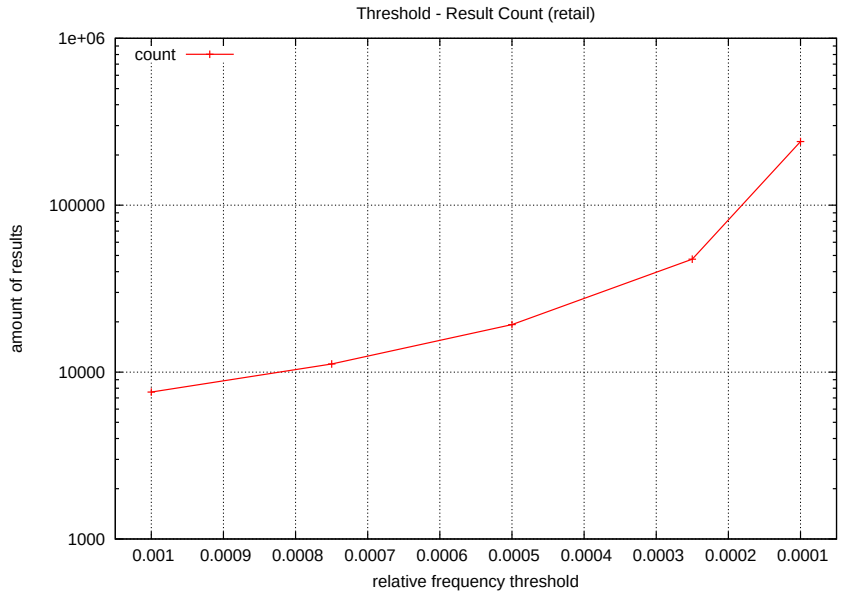
The *retail* Dataset

Figure 6.5 shows the amount of frequent *descriptive patterns* for the *retail* dataset at different relative frequency thresholds. Note that for the *retail* dataset no results are found up to a threshold of 0.6. On the other hand the number of results grows rapidly for thresholds smaller than 0.025. Thus, the threshold range was extended and split into two threshold ranges for better analysis.

In contrast to the *mushroom* data set, the effect of *modification set pruning* is the dominating factor for reducing the number of generated conditional trees on the *retail* dataset as is shown in Figure 6.6. Because of the amount of results for higher frequency thresholds backed by the fact that there are a lot of different items compared to the size of the dataset (the item to dataset



(a) Thresholds 0.025 through 0.9



(b) Thresholds 0.0001 through 0.001

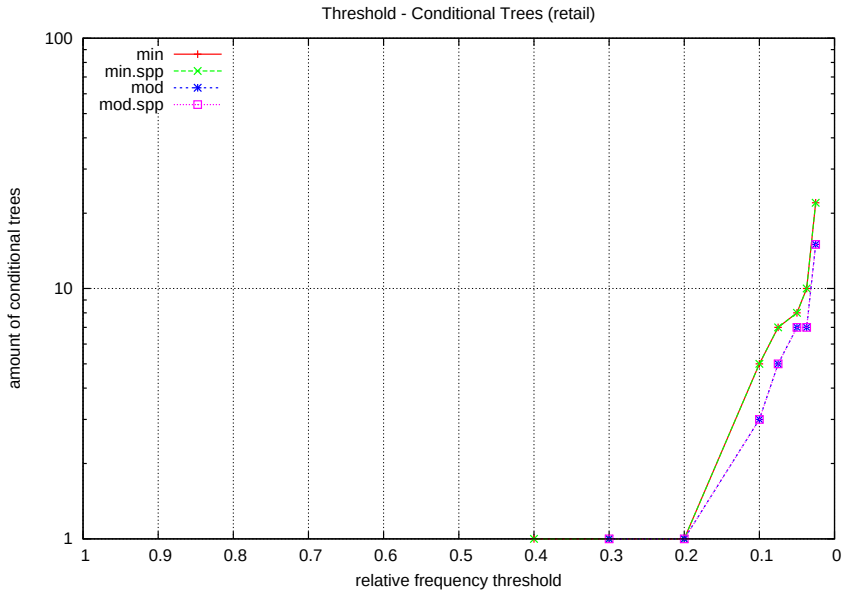
Figure 6.5: Number of frequent *descriptive patterns* in the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.

size ratio is at roughly 19 percent), it is possible to deduce that single items are quite rare in the dataset and even more so their combinations with other items. On the other hand the low amount of frequent *patterns* found at thresholds 0.5 through 0.025 induces a few, yet exceptionally frequent items. Such a distribution does not favor the existence of *single prefix paths* because, like items themselves, combinations of items do not occur very often.

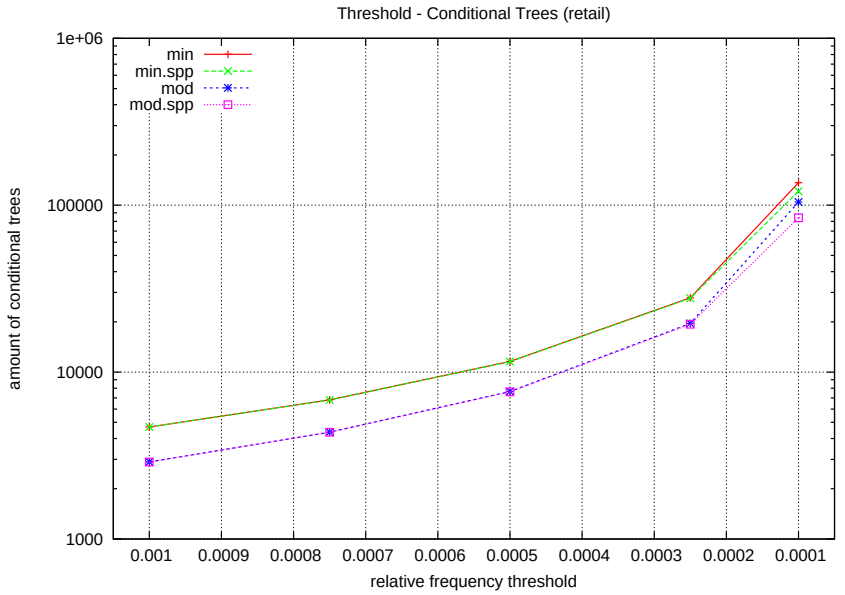
In such a scenario *modification set pruning* can be very effective. By removing infrequent items before adding them to a (conditional) tree or using them for extending *patterns*, trees are kept small (see Figure 6.7) and actual extensions sparse (see Figure 6.6). The low probability of items occurring together favors this optimization mechanism. Note that as the threshold decreases, less items can be pruned by *modification set pruning*. The amount of items to consider increases and the difference between using *modification set pruning* or not decreases, both in amount of generated conditional trees and their respective sizes. Contrariwise with lower thresholds the probability of *single prefix paths* raises. The beginnings of this effect can be observed at threshold 0.0001 in Figure 6.6 and Figure 6.7 respectively. The same effect at much higher thresholds was observed for the *mushroom* dataset. Figure 6.10 shows the initial tree's size. The effect of *modification set pruning* can be seen clearly. Yet it also becomes apparent how lower thresholds limit pruning possibilities.

The runtimes of the algorithm on the *retail* dataset at different thresholds are shown by Figure 6.8. As expected from the analysis above the main performance gain is introduced by using *modification set pruning* instead of solely depending on *branch pruning* whereas exploiting *single prefix paths* does hardly show any advantage in both threshold ranges. Note that very strong fluctuations occur for the *min.spp* and *mod.spp* settings. At the cost of considerably longer runtimes these random variations did not occur when using Java without its JIT compilation feature. Nevertheless, the overall distribution of runtimes was the same.

For the *retail* dataset the runtimes are considerably worse compared to the results of Frequent Itemset Mining Implementations 2003 and 2004 [29, 2]. Better performance can probably be achieved by more efficient implementations of internal procedures. An example is the most likely inefficient implementation of the tree structure and its building procedure, which can be derived from the fact that building the initial tree accounts for a relatively large amount of the overall runtime (see Figure 6.9), even considering that the initial tree is obviously the largest (comparing Figures 6.7 and 6.10).

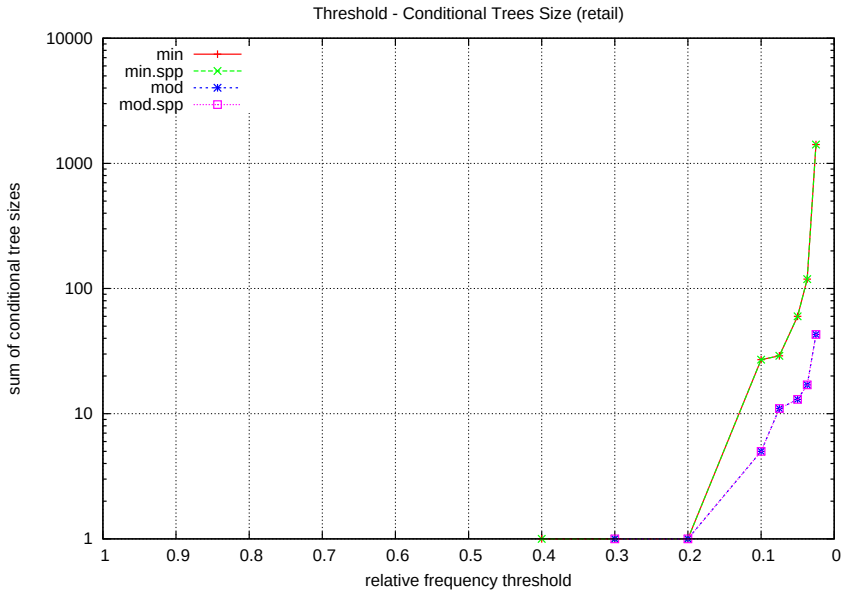


(a) Thresholds 0.025 through 0.9

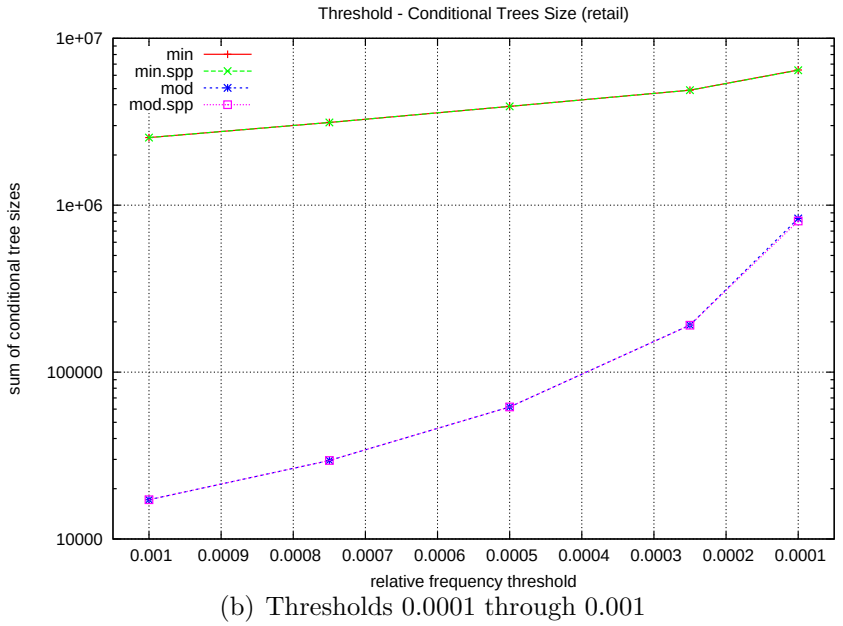


(b) Thresholds 0.0001 through 0.001

Figure 6.6: Amount of conditional trees generated by the *AnEMonE* algorithm on the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.

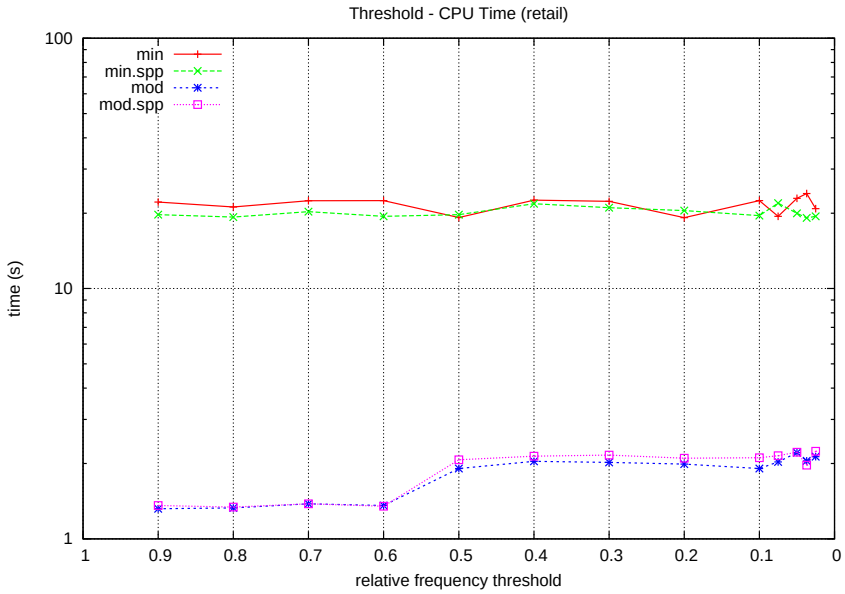


(a) Thresholds 0.025 through 0.9

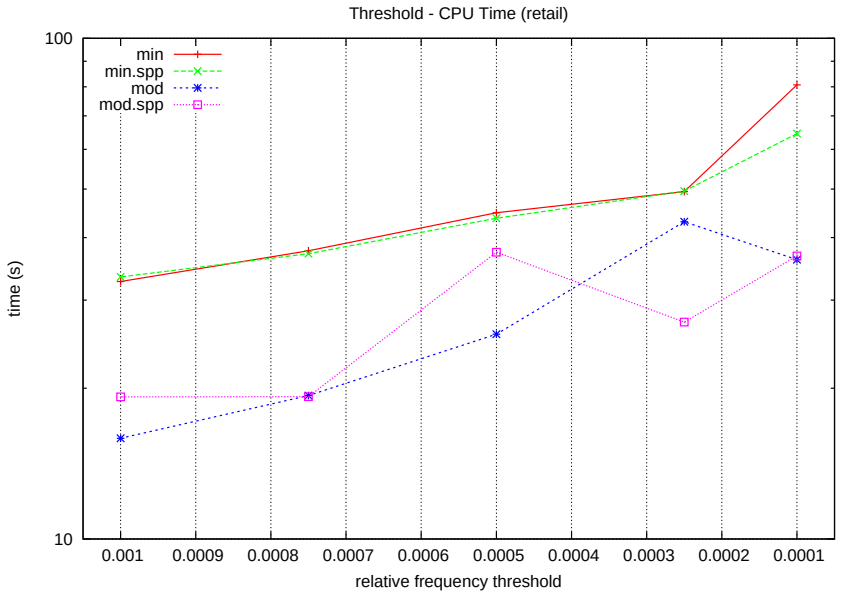


(b) Thresholds 0.0001 through 0.001

Figure 6.7: Sum of the sizes of conditional trees generated by the algorithm *AnEMonE* on the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.

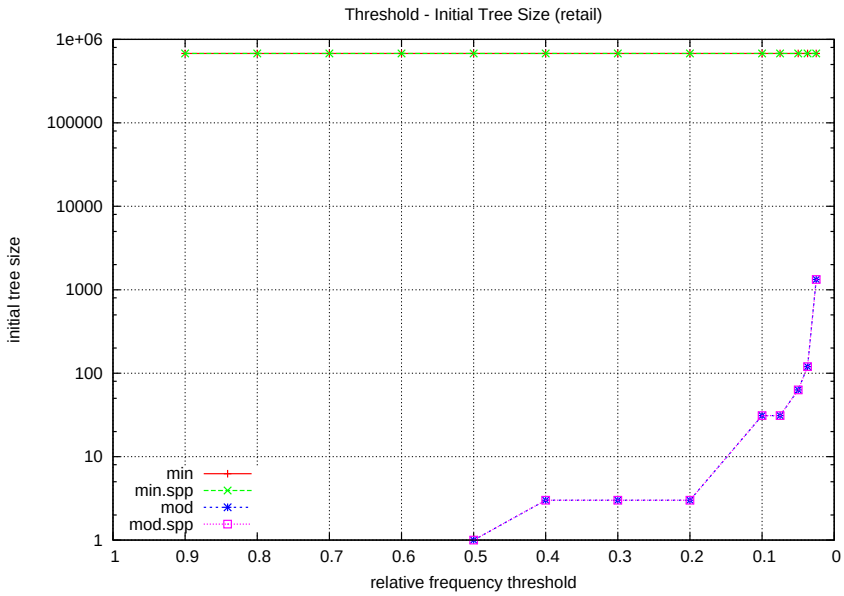


(a) Thresholds 0.025 through 0.9

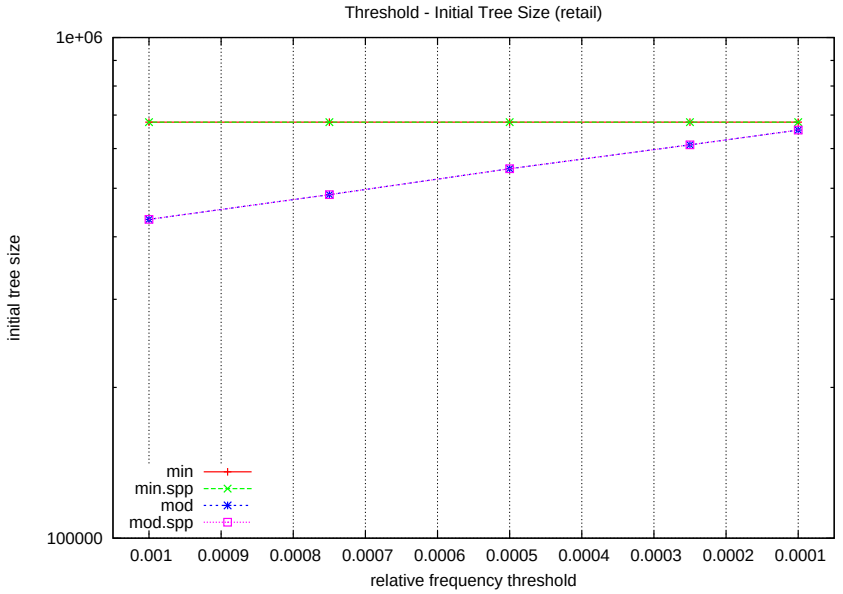


(b) Thresholds 0.0001 through 0.001

Figure 6.8: Runtime of the *AnEMonE* algorithm on the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.

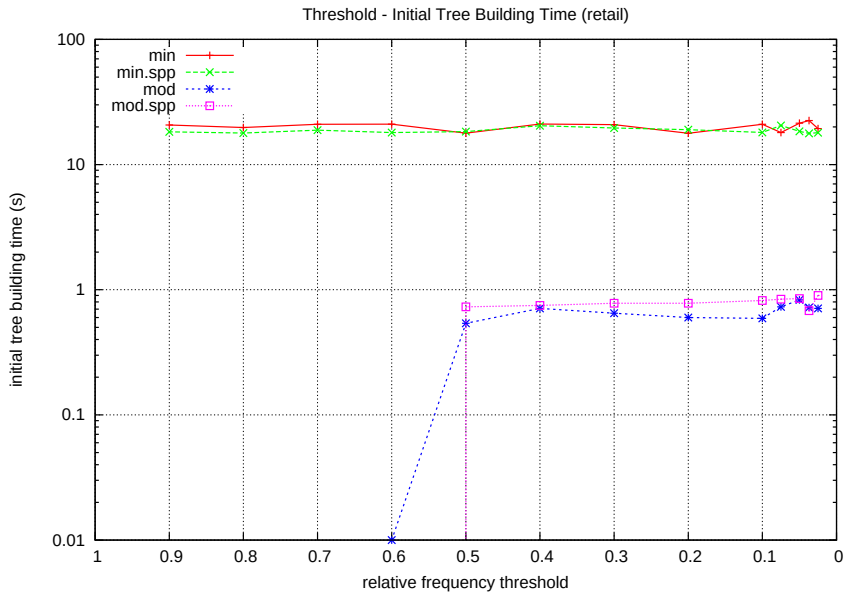


(a) Thresholds 0.025 through 0.9

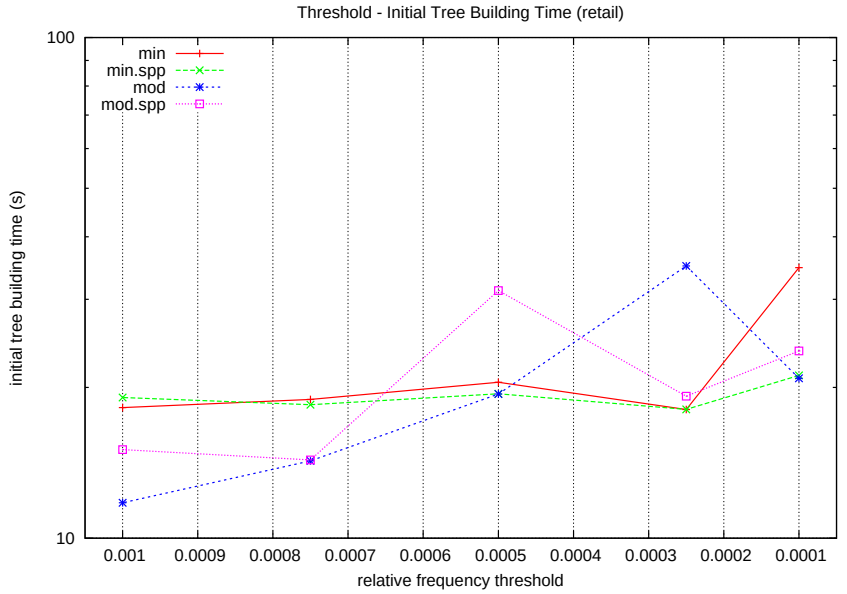


(b) Thresholds 0.0001 through 0.001

Figure 6.9: Size of the initial tree on the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.



(a) Thresholds 0.025 through 0.9



(b) Thresholds 0.0001 through 0.001

Figure 6.10: Time for building the initial tree on the *retail* dataset at different relative frequency thresholds. The subfigures show different threshold ranges.

6.1.2 Dynamic Frequency Threshold

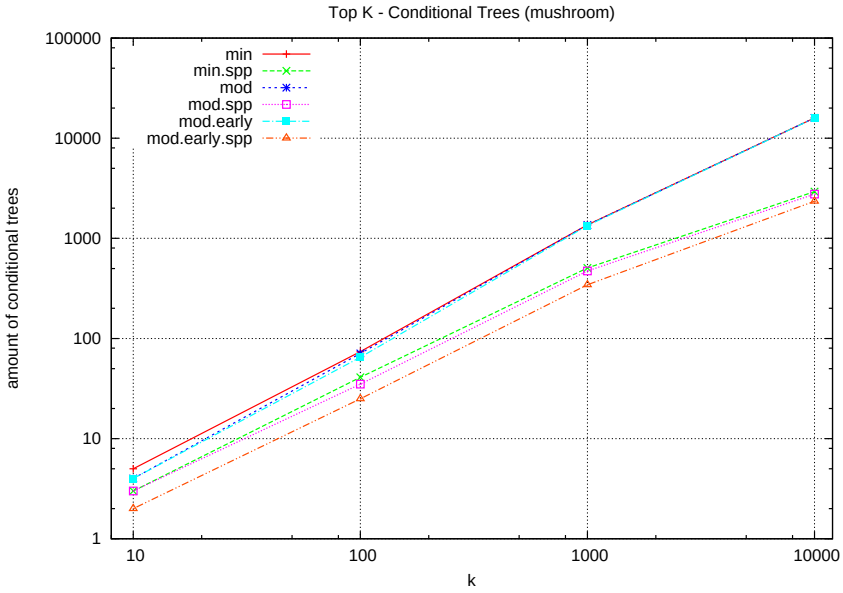
Instead of specifying a fixed threshold the number of frequent *descriptive patterns* to be part of the result can be limited to a fixed number of top most frequent *descriptive patterns* (see Section 4.2). Thus, the frequency threshold raises dynamically dependent on the *patterns* already added to the result. Again the experiments compare *branch pruning* with *modification set pruning* optionally exploiting *single prefix paths*. Additionally *early adding* is evaluated (see Section 3.2.2).

Note 6.1 (Early Adding)**.**
Adding pattern early can be done at different times. It can be done after or during the evaluation of extensions based on the items in the modification set (modification set pruning). The latter allows to have more information at each subsequent extension to be checked. After all descriptive patterns are added another pass over all extensions is needed to finish modification set pruning as the gained information from adding the descriptive patterns is only fully available at this point. Other possibilities were tested. The difference was insignificant.

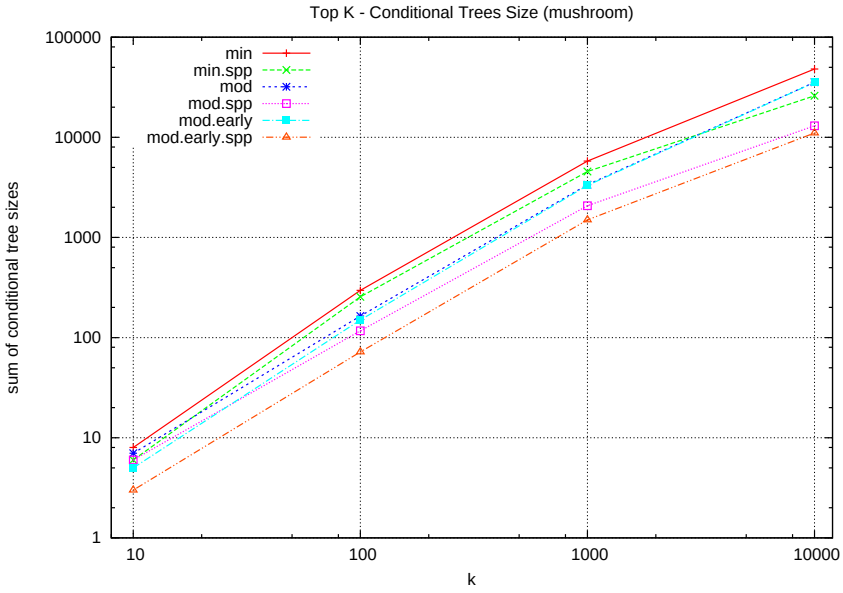
The *mushroom* Dataset

Figure 6.11 shows the amount of conditional trees generated during the search as well as the sum of their sizes. Similar to the relative threshold case, exploiting the *single prefix path* is the main factor in reducing the amount of conditional trees generated. Additionally adding *patterns* to the result early during the search raises the threshold faster, thus, allowing more efficient pruning of the *modification set* causing less and smaller trees as Figure 6.11 shows. Moreover, as in the relative threshold case, the overall size of conditional trees is lower for the *mod* settings compared to the *min.spp* setting at first. Yet, this relation changes between the result limit 1000 and 10000 which corresponds to a relative frequency threshold of 0.3 (see Figure 6.2 and 6.3 for comparison). Adding *patterns* early in combination with *single prefix path* exploitation achieves the overall least number of smallest conditional trees.

Figure 6.12 shows the runtimes of the algorithm for several result set limits. The runtimes are very close together, thus, an accurate comparison is not sensible. Yet, for higher limits the benefits of the optimizations and their respective combinations become apparent.



(a) Amount of conditional trees.



(b) Sum of the sizes of the conditional trees.

Figure 6.11: Amount of conditional trees and the sum of their respective sizes generated by the *AnEMonE* algorithm on the *mushroom* dataset at different result set limits.

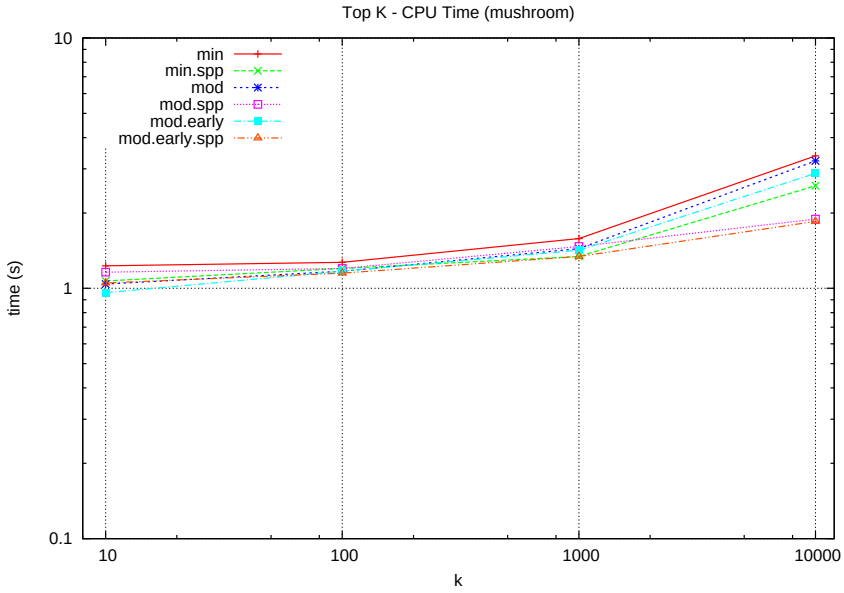


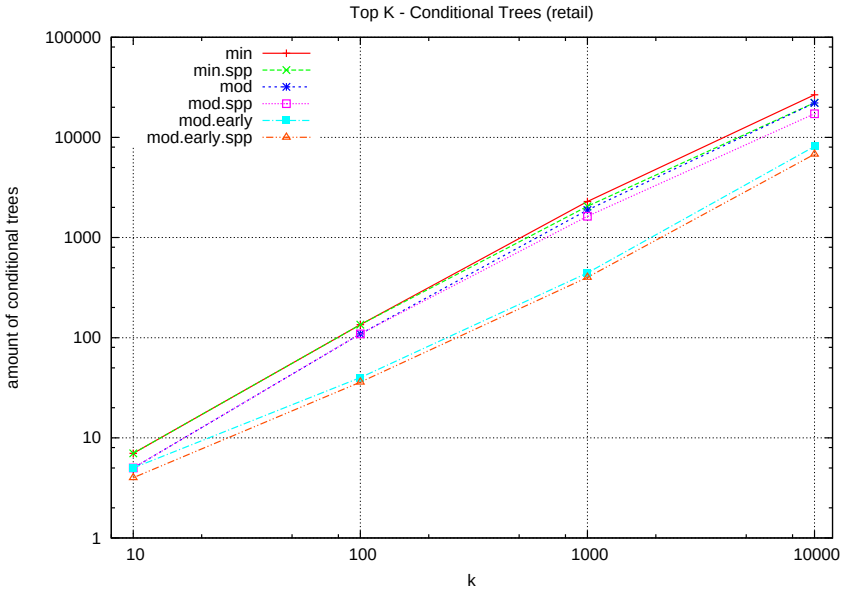
Figure 6.12: The runtime of the algorithm on the *mushroom* dataset at different result set limits.

The *retail* Dataset

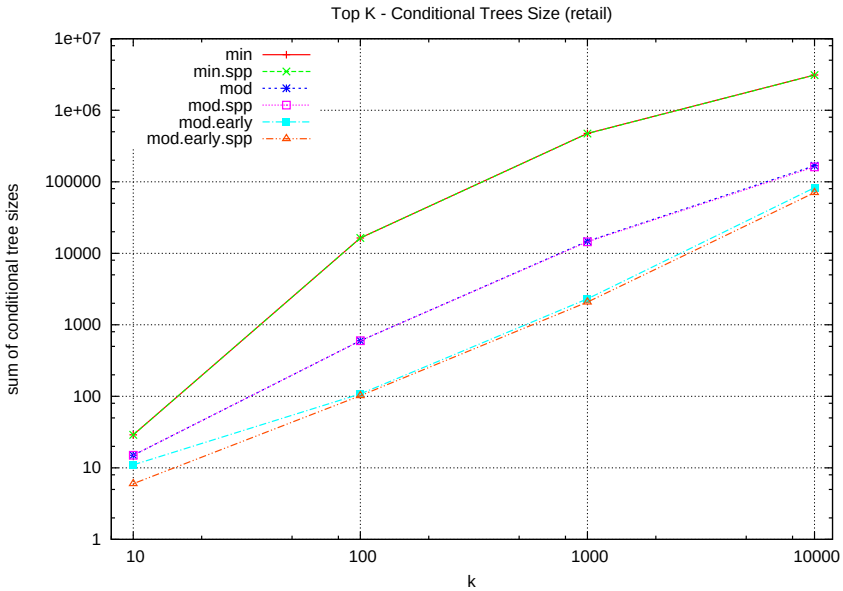
Figure 6.13 shows the amount of conditional trees as well as their accumulated size generated during the search for the *descriptive patterns* with the top k frequencies. Adding *patterns* early (*mod.early*) generates by far the least amount of conditional trees. *Modification set pruning* results in slightly less conditional trees than *min*. As in the case of relative thresholds exploiting *single prefix paths* shows small advantages. The effect was stronger on the *mushroom* dataset. The advantage of adding *patterns* to the result early (*early*) is due to the fact that initially the frequency threshold starts at zero. As a result, it is less likely to prune items at the first few search nodes until more *patterns* are added to the result.

The tree sizes differ greatly between *min*, *mod* and *early*. *Mod* generates smaller trees than *min* because it prunes items early, which are then not part of the tree. *Mod.early* raises the threshold fast being able to prune items even earlier than *mod* resulting in the smallest trees.

Figure 6.15 shows the runtimes for several result limits on the *retail* dataset. As expected, the runtimes for adding results early to raise the threshold quickly yields the best results. Note that again, compared to the initial tree building time, the overall runtime is not much higher. This is explainable by the fact that the size of all conditional trees combined (which

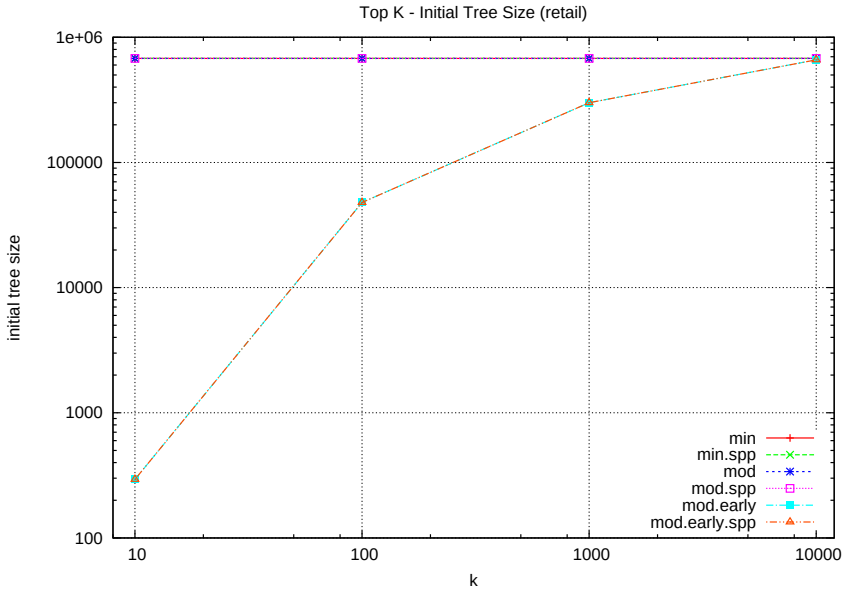


(a) Amount of conditional trees.

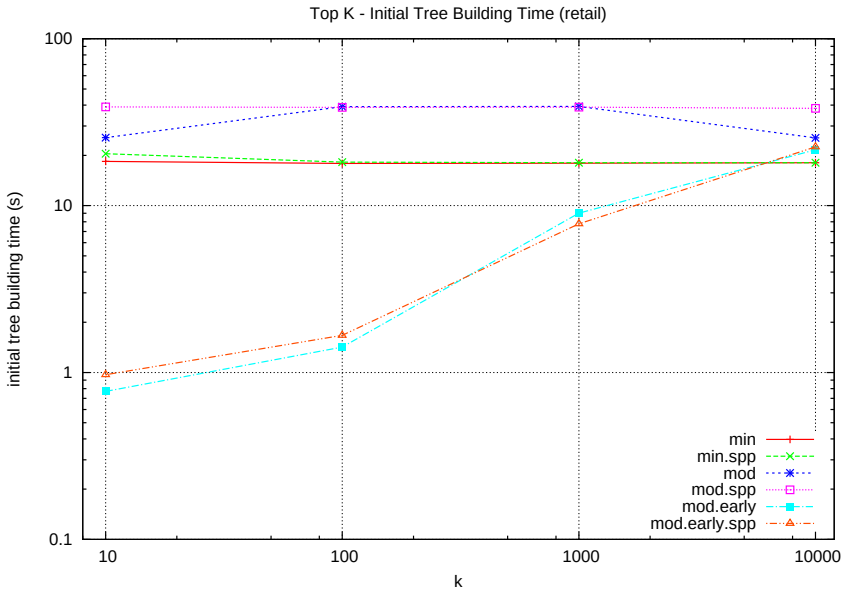


(b) Accumulated size of conditional trees.

Figure 6.13: The amount of generated conditional trees and their accumulated size on the *retail* dataset at different result set limits.



(a) Initial tree size.



(b) Initial tree building time.

Figure 6.14: The size of the initial tree and its building time on the *retail* dataset at different result set limits.

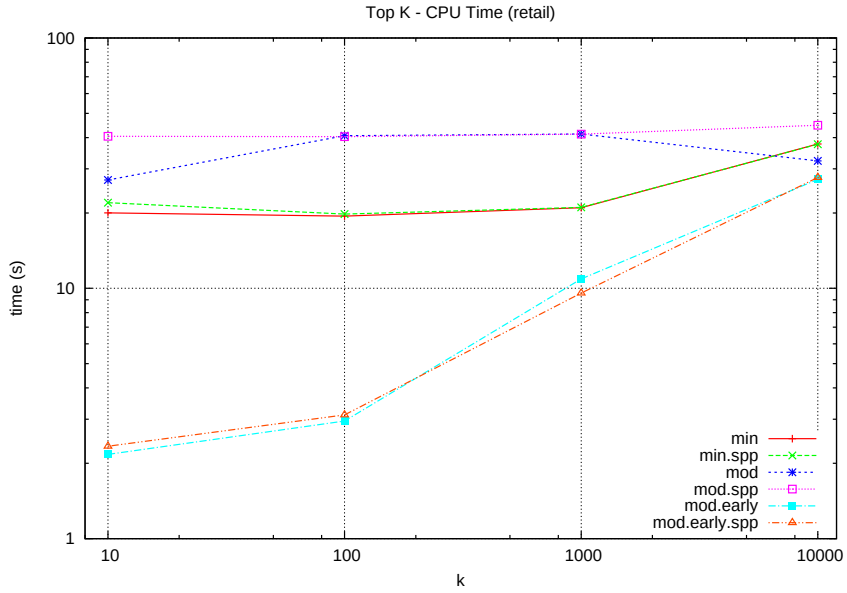


Figure 6.15: The runtime of the *AnEMonE* algorithm on the *retail* dataset at different result set limits.

does not include the initial tree) is smaller than the initial tree size in all cases.

The considerable runtime difference (factor two) between *min* and *mod* should not occur as the main time spent by the algorithm is when creating the initial tree. As no threshold is set when building the initial tree, *mod* does not have an opportunity to prune. Thus, the trees are of the same size (as Figure 6.14(a) shows). The same set of tests was run without Java's Just-In-Time compilation feature yielding the correct proportions, yet, tremendously longer runtimes (see Figure 6.16).

6.1.3 ExAnte

In this section a *monotone description constraint* is added to *frequent pattern mining*. This enables the exploitation the *ExAnte property*. Different modes are available as explained by Table 6.1. Note that *exInit* is equivalent to a preprocessing step as suggested by the original *ExAnte* paper [12]. *exRec* and *exFull* take the exploitation of the *ExAnte property* a step further pushing it into the recursion (also suggested by [12] and implemented by [10]). Note that the corresponding method introduced in [12] and reviewed in Section 3.2.1 requires additional database passes at each search node introducing a

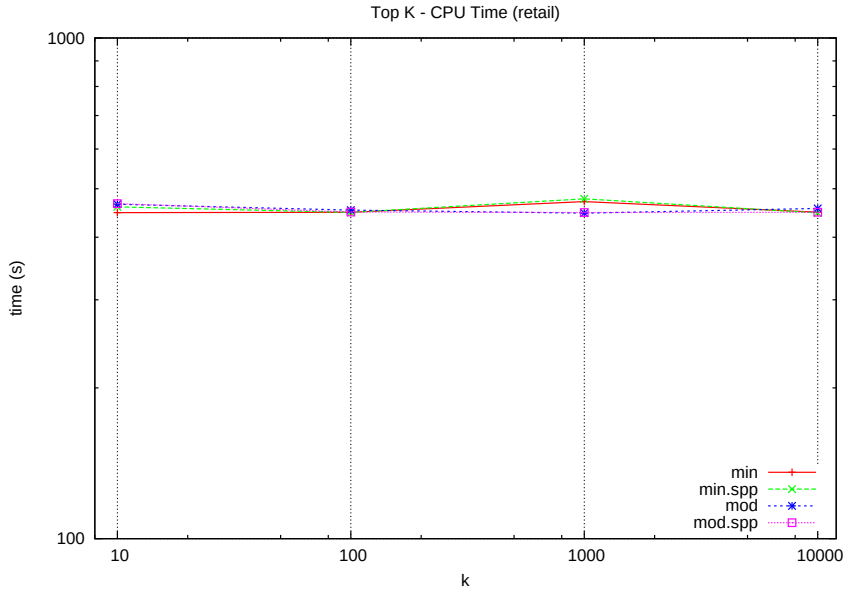


Figure 6.16: The runtime of the *AnEMonE* algorithm on the *retail* dataset at different result set limits with Java's JIT feature turned off.

trade-off between the effect of pruning and costs of the pruning method.

The data used for the corresponding experiments is *retail* dataset. It contains market basket data, thus, each item was sold for a specific price. Instead of looking for frequent itemsets only, it is possible to search frequent itemsets associated with a minimum overall price. This defines a *monotone description constraint* (for more *monotone description constraints* see [12]). As no real world data has been available, a price between 0 and 10 was assigned randomly to each item in the dataset. For a *descriptive pattern* to be valid the accumulated price of corresponding set of items must reach or surpass a predefined threshold p.

Two test scenarios are explored: exploiting the *ExAnte property* in the context of relative frequency thresholds and in the context of dynamic frequency thresholds. Only *modification set pruning* is being used in combination with exploiting the *ExAnte property* because *modification set pruning* has proven to be more efficient than *branch pruning* in the previous sections. Also exploiting the *ExAnte property* requires checking *anti-monotone (data) constraints*, thus, skipping *modification set pruning* in favor of *branch pruning* introduces redundancy in checking *anti-monotone constraints*.

The following section will focus on the lower relative frequency threshold range known from the previous sections, because more interesting develop-

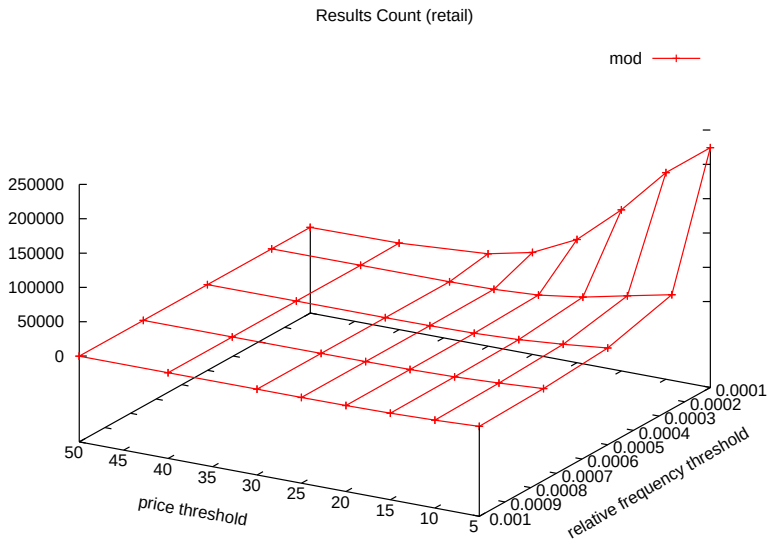


Figure 6.17: The amount results on the *retail* dataset with the price threshold constraint.

ments were observed. Moreover the effect of exploiting *single prefix paths* was shown before and will only be mentioned briefly in the following analysis. Overall this section covers rows three through five intersected by the second column from Table 6.1.

Relative Frequency Threshold

Figure 6.17 shows the amount of frequent *patterns* satisfying different price thresholds. For higher price thresholds the number of results drops tremendously. Thus, exploiting the *monotone description constraint* can potentially show significant effects on generated conditional trees and the overall tree size.

Figures 6.18 and 6.19 show the amount of generated conditional trees and their overall size for different algorithm settings at different thresholds. By reducing the size of *conditional data record sets* as well as the amount of items used for extensions, exploiting the *ExAnte property* keeps trees small. Additionally the number of conditional trees decreases according to the number of results. The effect of *ExAnte* on the initial *data record set* is marginal (see Figure 6.20).

Exploiting the *ExAnte property* pays its reduction of tree numbers and

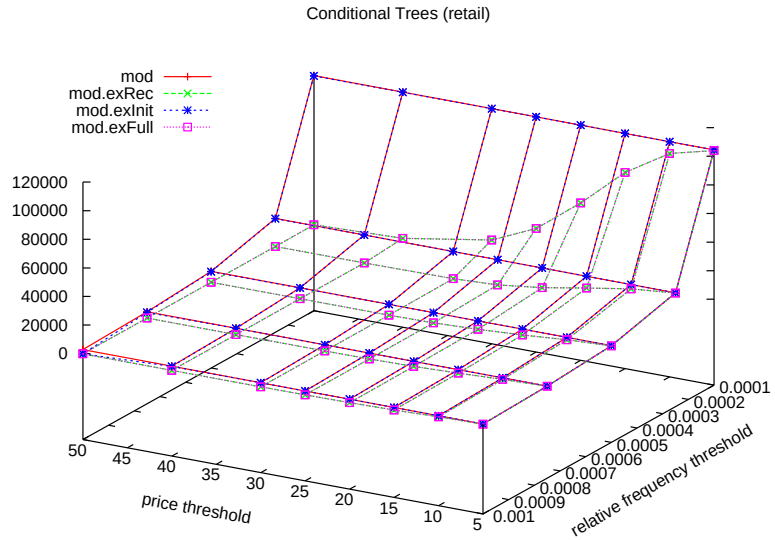


Figure 6.18: The amount of generated conditional trees on the *retail* dataset exploiting the ExAnte property at different price and relative frequency thresholds.

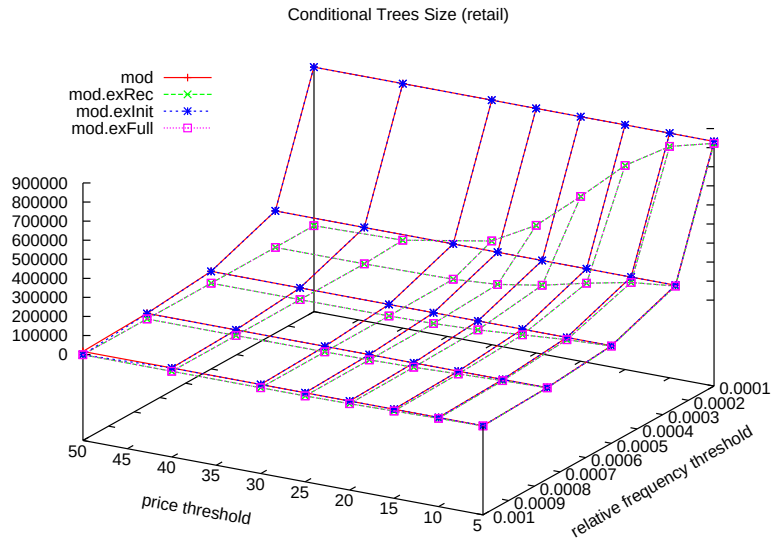


Figure 6.19: The accumulates size of all conditional trees on the *retail* dataset exploiting the ExAnte property at different price and relative frequency thresholds.

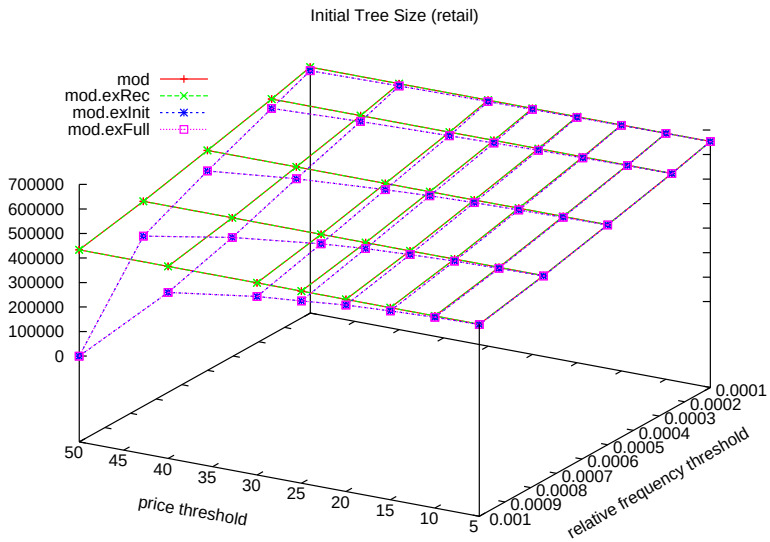


Figure 6.20: Initial tree size for the search on the *retail* dataset at different price and relative frequency thresholds.

sizes with additional *(conditional) data record set* passes. The reference amount of *data records* passed during a search is given by only using *modification set pruning* (*mod*) (see Figure 6.21). Note how exploiting the *ExAnte property* increases the number of passed *data records* by large quantities for lower price thresholds, but also how this relation changes as price thresholds increase. Low frequency thresholds and higher price thresholds denote more items and *data records* to prune by *monotonicity*. As the number of *conditional data record set* passes decreases, and with every pruned item, the *ExAnte* property starts to be more efficient for higher price thresholds than solely exploiting *anti-monotonicity* by *modification set pruning*. Note how this effect does not occur, if exploiting the *ExAnte property* is only applied to the initial *data record set*, i.e. is only used as a preprocessing step.

The runtimes (see Figure 6.22) reflect the analysis from above thereby illustrating how less and smaller trees weigh against additional *data record set* passes. Using *ExAnte property* exploitation only on the initial *data record set* reduces tree numbers and sizes marginally, thus, introducing *data record set* passes that are less efficient than others considering the pruned items and *data record* relative to the *data record set* size. Additionally, the initial passes are costly as more *data records* and items are present in the initial *data record set*. The result is a considerably worse runtime compared with

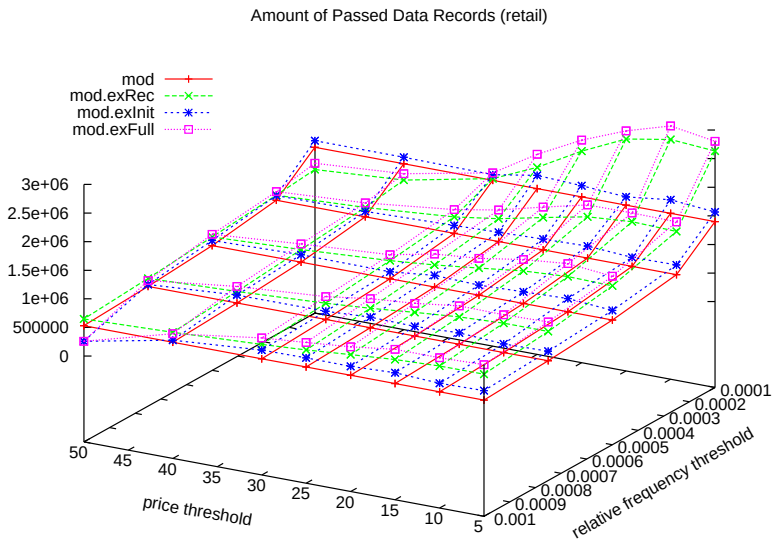


Figure 6.21: Number of *data record sets* passed during the search on the *retail* dataset at different price and relative frequency thresholds.

using *modification set pruning* alone. An exception is seen at the highest frequency and the highest price threshold.

Using *ExAnte property* exploitation on all but the initial *data record set* shows a strong effect due to its significant tree number and size reduction even though a lot more *data records* are passed overall. This is because the *conditional data record sets* which the exploitation method of the *ExAnte property* is working on are built from conditional trees being small compared to the initial *data record set* (compare the initial tree size 6.20 with the accumulated size of all conditional trees 6.19). The runtime of using full *ExAnte property* exploitation ($exFull$) results directly from combining both the $exInit$ and the $exRec$ settings. Overall, the effect of exploiting the *ExAnte property* is strongly dependent on the characteristics of the analyzed dataset. Denser datasets would result in larger conditional trees making the exploitation more costly.

The effect of exploiting *single prefix paths* is shown in Figure 6.23 depicting the amount of conditional trees generated. Figure 6.24 shows the corresponding runtimes. The amount of conditional trees drops according to the exploitation of single prefix paths. The runtime is marginally effected. The proportions basically stay the same.

Figure 6.25 shows how initial pruning based on the *ExAnte property* is

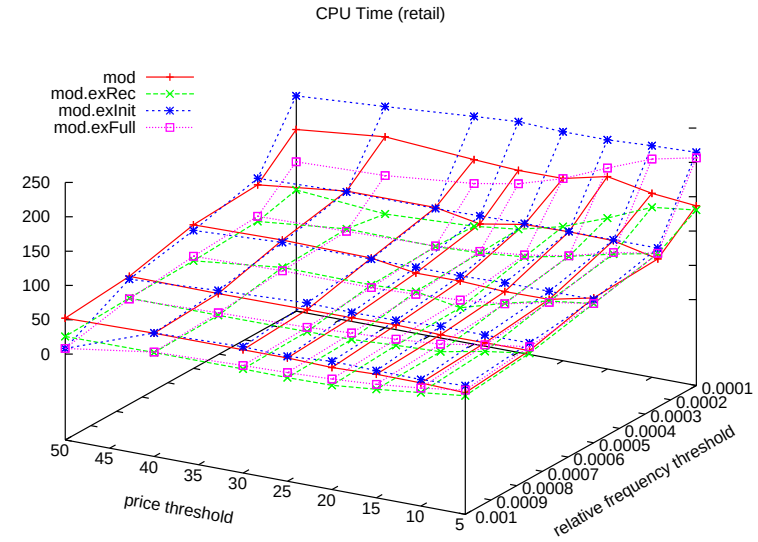


Figure 6.22: Runtimes for the search on the *retail* dataset at different price and relative frequency thresholds.

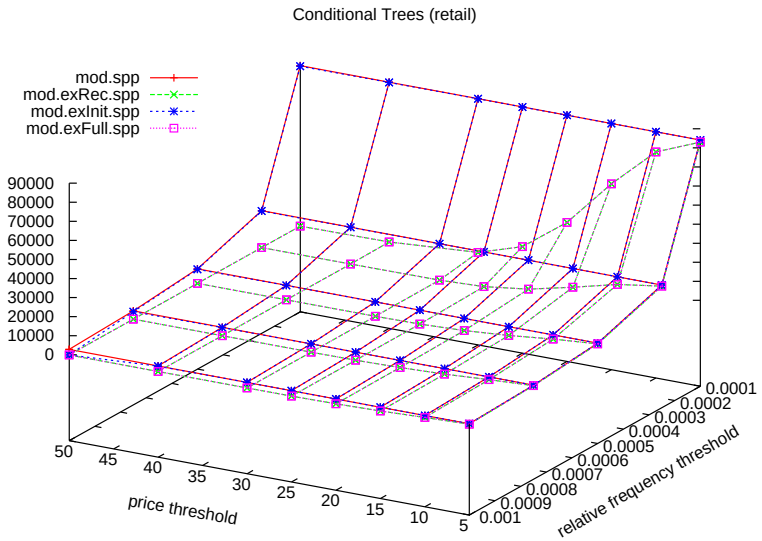


Figure 6.23: Amount of conditional trees for the search on the *retail* dataset at different price and relative frequency thresholds exploiting *single prefix paths*.

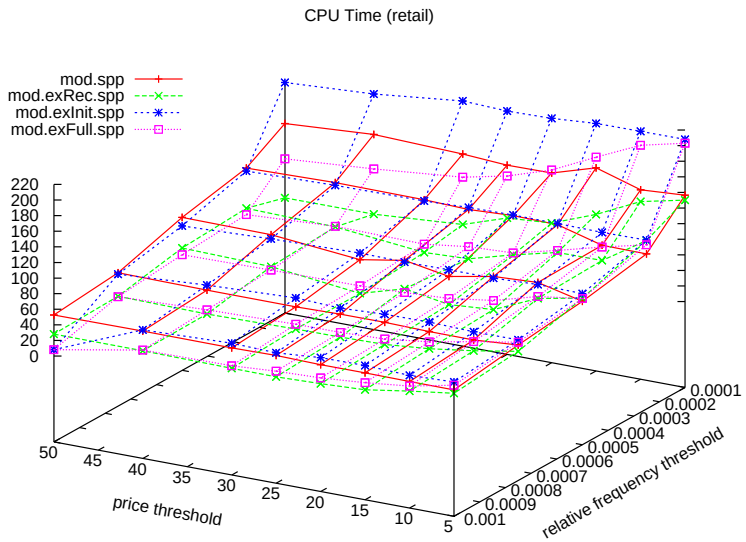


Figure 6.24: Runtimes for the search on the *retail* dataset at different price and relative frequency thresholds exploiting single prefix paths.

more efficient for higher frequency thresholds in relation to smaller ones but still does not surpass the *exRec* setting.

Dynamic

The first thing to notice is that using the *monotone* price *constraint* together with a dynamically raised frequency threshold based on a limited set of *patterns* has a strong negative effect on how many conditional trees are being generated (and also how many candidates are being considered) during the search compared to the case using fixed relative thresholds (compare Figures 6.6 and 6.18 with Figures 6.13 and 6.26). When using a relative frequency threshold the set of considered *patterns* is fixed, dependent on the frequency threshold and the price *constraint* further limits the set of results. The relative frequency threshold is at its maximum from the beginning. On the other hand, when using a dynamic threshold based on the currently found set of most frequent *patterns*, the limitation of valid *patterns* by the price *constraint* results in less *patterns* being added to the result at lower levels of the search. Thus, the frequency threshold is raised more slowly resulting in less *anti-monotone* pruning. It takes time and the exploration of many branches for the threshold to reach its maximum. As a result of this, more

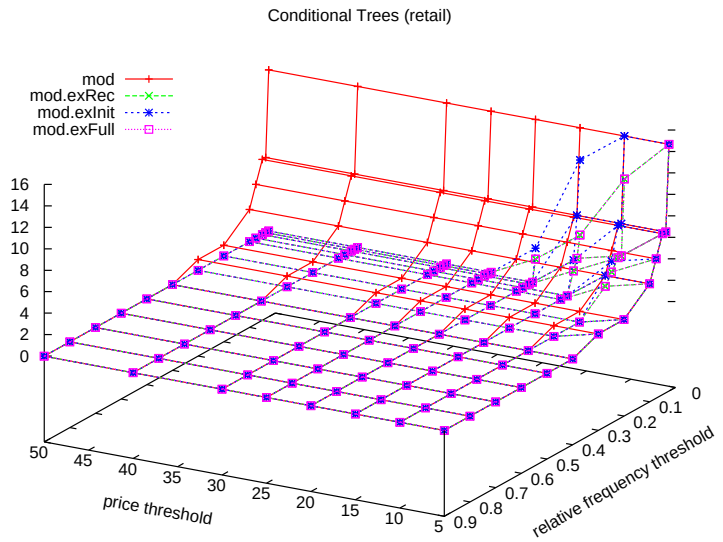


Figure 6.25: Amount of conditional trees for the search on the *retail* dataset at different price and relative frequency thresholds illustrating how using ExAnte on the initial data is more effective at higher frequency thresholds.

conditional trees are produced.

The performance gain achieved by adding *patterns* early (*mod.early*) to the result in order to raise the frequency threshold quickly only shows a small effect on the number of generated conditional trees as the price threshold increases. This is due to the large number of trees being generated when a *monotone constraint* is present as is shown in Figure 6.26. As expected, the corresponding effect is marginally, yet obviously, visible when looking at the runtimes (see Figure 6.27). The remainder of this section will focus on the slightly slower variant not adding *patterns* early to allow direct comparison with the static experiments.

Figure 6.28 shows the amount of conditional trees for the different *ExAnte* settings introduced by Table 6.1. As when using the relative frequency threshold, fully exploiting the *ExAnte property* (*exFull*) as well as by using it on *conditional data* only (*exRec*) reduces the number of conditional trees greatly. The *ExAnte property* exploitation used only on the initial *data record set* (*exInit*; equivalent to preprocessing) does not yield considerable results. The latter even increases the number of conditional trees. This effect can be explained by the fact that exploiting the *ExAnte property* influences the order of branching (currently the reversed *frequency order*) by removing *data*

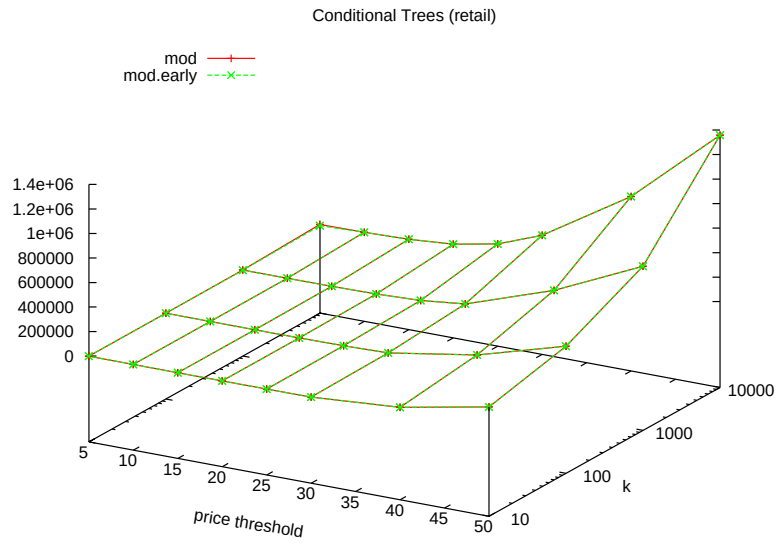


Figure 6.26: Amount of conditional trees for the search on the *retail* dataset at different price thresholds and result set limits illustrating how adding *patterns* early compares to *modification set* pruning alone.

records considered for extensions. Thus, depending on the characteristics of the dataset and the applied *branching order*, those changes can influence the search either positively or negatively.

Figure 6.29 shows the number of *data records* passed according to the different thresholds and settings. Only using *modification set pruning* (*mod*) the amount of conditional trees generated increases for large result set limits and high price thresholds, thus accordingly, the amount of passed *data records* in order to build those trees increases. Using *ExAnte property* exploitation on the initial *data record set* alone (*mod.exInit*) does not show a significant increase or decrease of passed *data records* compared to using *modification set pruning* only (*mod*). Those algorithm settings that exploit the *ExAnte property* on conditional data (*mod.exRec* and *mod.exFull*), pass more *data records* for lower price thresholds. As the threshold increases more *data records* and items can be pruned exploiting the *monotone description constraint*. This effect, as well as the resulting smaller amount of conditional trees, reduces the passed *data records* to an extend were less *data records* are passed compared with settings not applying the *ExAnte property* on conditional data.

Figure 6.30 shows the runtimes according to the different thresholds and

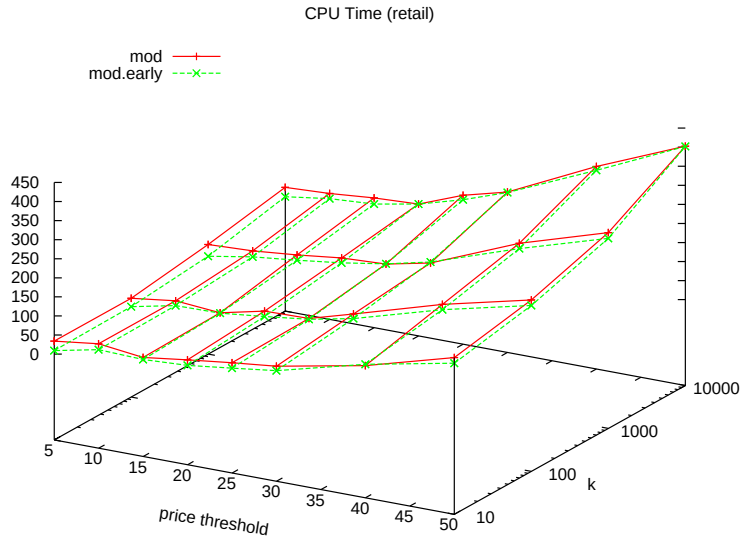


Figure 6.27: Runtimes for the search on the *retail* dataset at different price thresholds and result set limits how adding *patterns* to the result early influences the runtime positively.

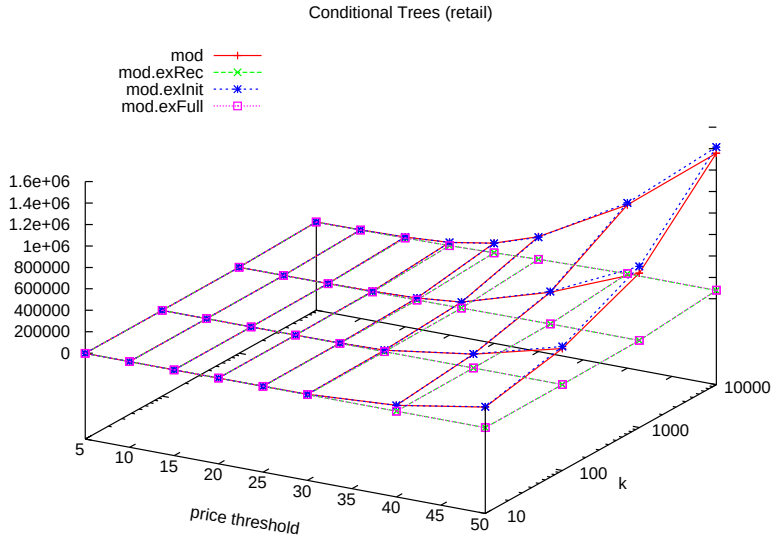


Figure 6.28: Amount of conditional trees for the search on the *retail* dataset at different price thresholds and result set limits.

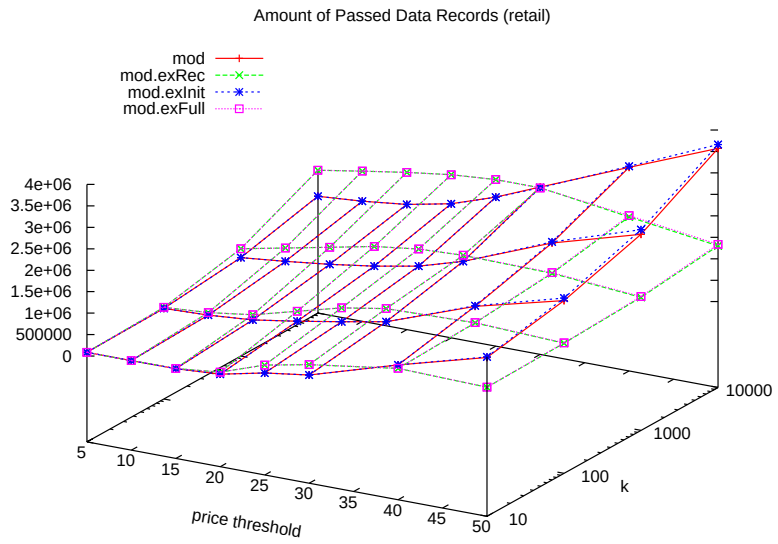


Figure 6.29: *Data records* passed during the search on the *retail* dataset at different price thresholds and result set limits.

settings. The runtime behavior is clearly dominated by using *ExAnte property* exploitation on conditional data (*mod.exRec* and *mod.exFull*). The proportions reflect the number of conditional trees generated (see Figure 6.28).

While the effect of exploiting *single prefix paths* was visible when using relative frequency thresholds in generated conditional trees, the runtime was hardly affected. The same effect is much stronger for the dynamically raised frequency threshold, as can be seen by comparing Figures 6.31 and 6.32 with Figures 6.28 and 6.30. The number of conditional trees is cut by more than half resulting in much better runtimes (see Figure 6.32). The proportions of the different settings for conditional trees and the runtimes stay the same.

6.2 Subgroup Mining

Subgroup mining [56, 8] was formulated to fit the definition of *descriptive pattern mining* in Section 4.1.2. Like *frequent pattern mining* it defines a single constraint: a *quality constraint*. In contrast to *frequent pattern mining* the corresponding *quality functions* depend on a target variable. Different *quality functions* can be derived based on such a target variable. Dependent

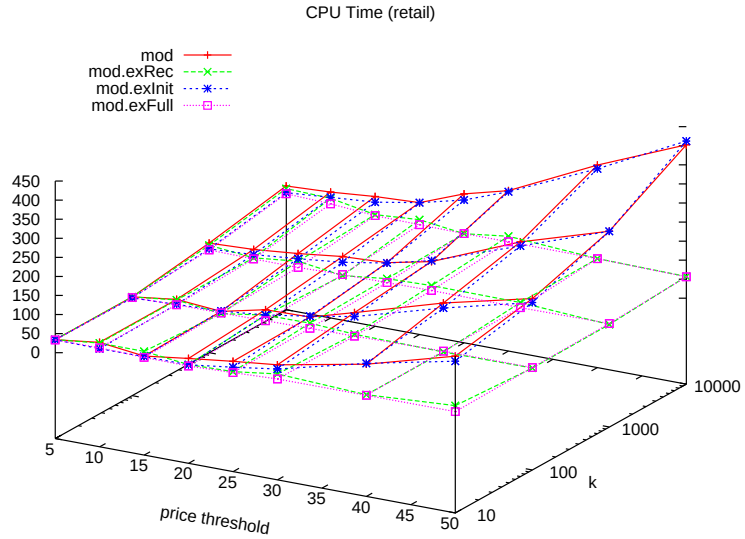


Figure 6.30: Runtimes for the search on the *retail* dataset at different price thresholds and set result set limits.

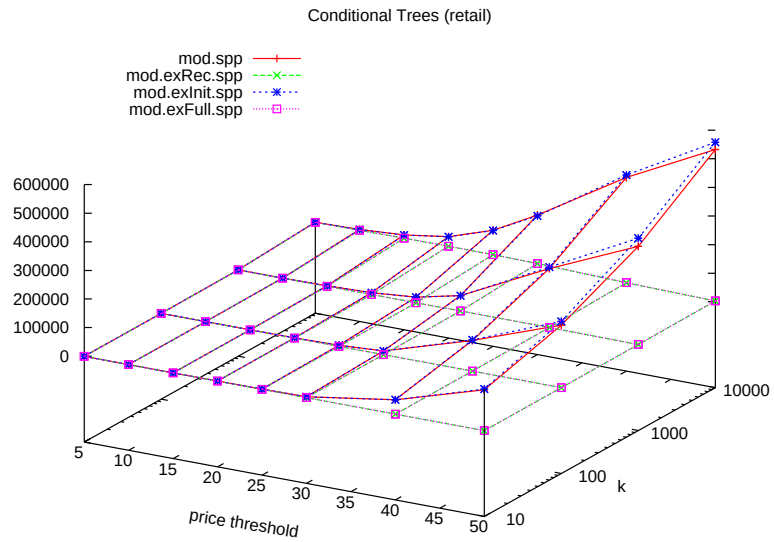


Figure 6.31: Amount of conditional trees for the search on the *retail* dataset at different price thresholds and result set limits using *single prefix path exploitation*.

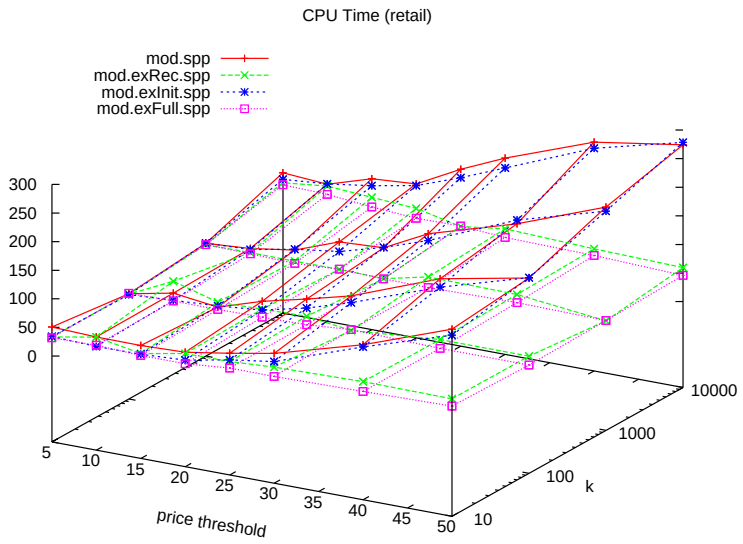


Figure 6.32: Runtimes for the search on the *retail* dataset at different price thresholds and result set limits using *single prefix path exploitation*.

on the assigned qualities a fixed threshold or a set of the *descriptive patterns* with the largest qualities, values can be the objective of the search. This section will focus on the latter, thus, doing *top-k mining* (see Section 4.2). In this context the order of adding *descriptive patterns* to the result plays an important role. One heuristic for positively influencing the corresponding order is sorting by *optimistic estimate*. So, additionally to using the *optimistic estimate* for pruning, it is being used for specifying the branch order (*estim*), i.e. those items associated with extensions assigned larger *optimistic estimates* values will be used for branching first. The effects are highlighted by Section 6.2.1. Furthermore all optimizations introduced by Section 3.2 can be used for any *descriptive pattern mining class* or respective variations. Section 6.2.2 demonstrates how exploiting the *ExAnte property* can be applied to an extended version of *subgroup mining* which has not been done before.

6.2.1 The *credit-g* Dataset and Branching Order

The first subgroup mining instance is based on the dataset *credit-g*. This dataset is part of the UCI Machine Learning Repository [27] and is originally called "Statlog (German Credit Data) Data Set". It has been dis-

Credit-g	
attributes	21
instances	1000
attribute-value pairs	98

Table 6.2: Information about the *credit-g* dataset.

cretized using WEKA's (version 3.6.4) [31] unsupervised discretization utility (`weka.filters.unsupervised.attribute.Discretize`). It contains data classifying individuals as "good" or "bad" concerning credits. Individuals are described by a set of attributes. Information about the dataset dimensions are found in Table 6.2.

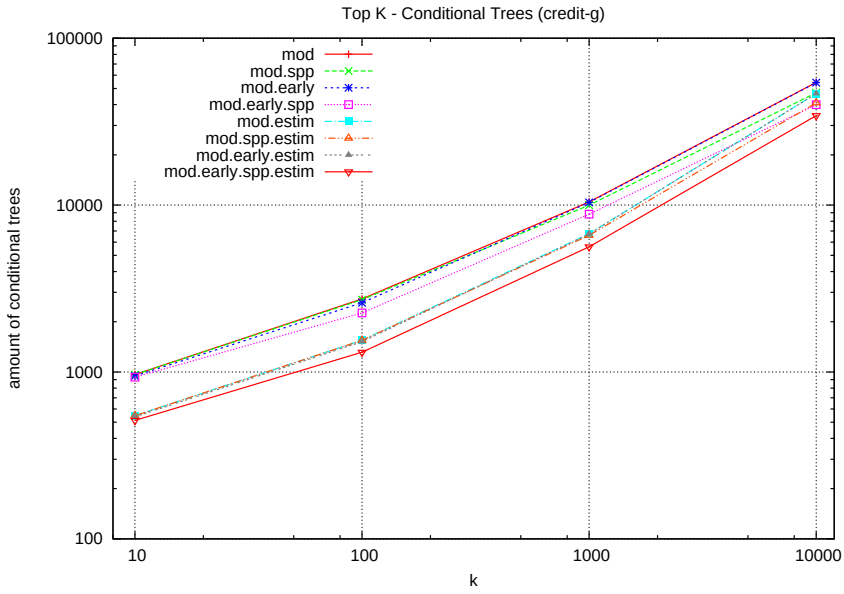
The experiments are based on finding a set of *descriptive patterns* with the best qualities. The *quality function* used is the *Piatetsky-Shapiro quality function* as introduced in Section 4.1.2. Pruning is based on the corresponding *optimistic estimate* from the same section. The setting *estim* is sorting items to branch on by the *optimistic estimate* of their corresponding extensions instead of their frequency. This has been proven to be a well working heuristic (cf. [56, 30, 5]).

Figure 6.33 shows the amount of conditional trees generated during subgroup mining. The results are as expected. Sorting by *optimistic estimate* generally produces the better results. Exploiting *single prefix paths* gains importance as the amount of *patterns* to search increases (i.e. the associated relative quality threshold decreases) and even yields better results than relying on sorting by *optimistic estimate* alone. The other optimizations emit the same behavior as shown in previous sections. Combining all optimizations yields the best results without exception. The same is true for the runtimes.

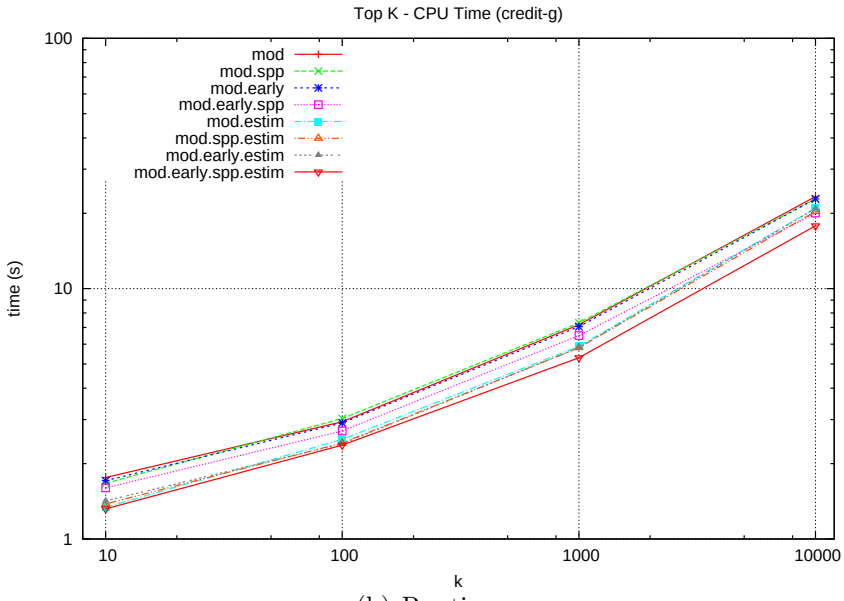
6.2.2 The *mushroom* Dataset and ExAnte

The *mushroom* has been used for subgroup mining before (c.f. [30]). The original task is to find *patterns* of features (attribute-value pairs) that describe edible mushrooms. The amount of resulting *descriptive patterns* is limited to those with best qualities according to a *quality function* judging the edibility of mushrooms described by the classifying *descriptive pattern* and based on the given data. Just like in the previous section the *Piatetsky-Shapiro quality function* is used.

Currently lacking an application of *monotone description constraints* in the field of subgroup mining, the *mushroom* dataset is extended by assigning a value of interest to each feature (attribute-value pair). This value of interest can be high, for example, if a certain feature turns out to specify mushrooms



(a) Amount of conditional trees.



(b) Runtime.

Figure 6.33: The amount of generated conditional trees and the runtime on the *credit-g* dataset for *subgroup mining* with the nominal target "class=good" at different result set limits.

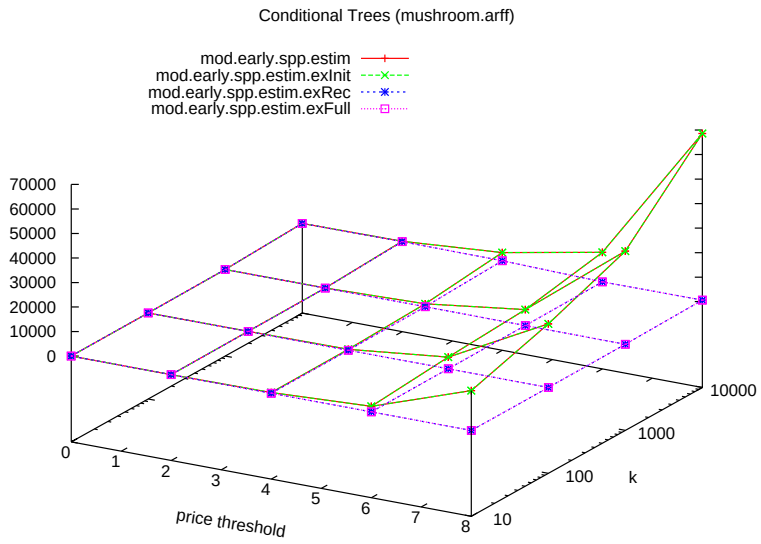


Figure 6.34: Amount of conditional trees for the search on the *mushroom* dataset for *subgroup mining* with a *monotone description constraint* being exploited the *ExAnte property*.

that taste very well, i.e. red dots indicate a good taste. Given that, the more features indicating good taste, the better the mushroom tastes (independent of the truth of this statement). The *monotone description constraint* can limit the valid *patterns* describing mushrooms to those that surpass a threshold according to the value of interest. A value of interest from a range between 0 and 1 was assigned randomly to each attribute-value pair. Several threshold levels are used throughout this section.

This set of experiments shows how exploiting *monotone descriptive constraints* by the *ExAnte property* effects the subgroup search. Figures 6.34 and 6.35 show the effect of exploiting the *ExAnte property* by comparing the amount of generated conditional trees and runtimes. Both the number of generated conditional trees as well as the runtimes are lower for higher thresholds according to the value of interest.

The conclusion drawn from this section is that exploiting the *ExAnte property* in the domain of *subgroup mining* can be seamlessly applied and potentially improves the performance of the search, if *monotone description constraints* are present.

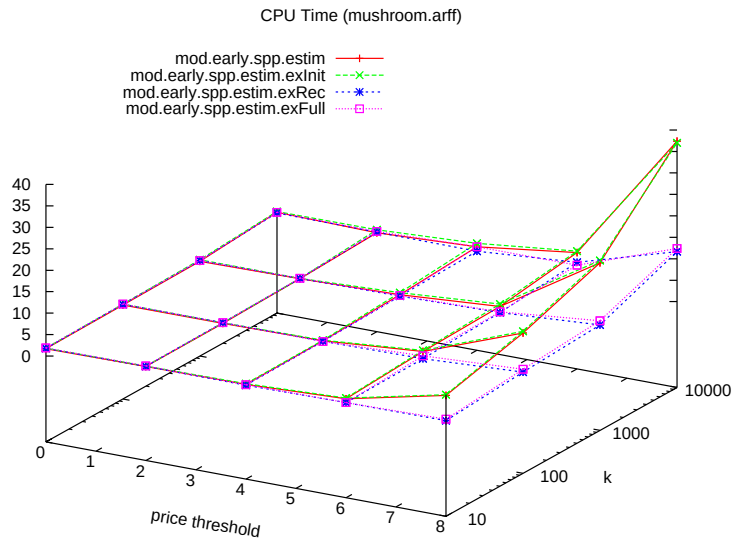


Figure 6.35: Runtimes for the search on the *mushroom* dataset for *subgroup mining* with a *monotone description constraint* exploited by the *ExAnte property*.

6.3 Community Mining

Community mining as introduced by [6] and defined as a *descriptive pattern mining class* by Section 4.1.2 uses a *dataset* built from a graph as input. The *individuals* are associated with edges each assigned a set of items derived from the intersection of terms corresponding to each node. A community is a subgraph described by a set of *items*, i.e. a *descriptive pattern* selecting a set of edges. A *descriptive pattern* is valid if it is equal to or exceeds a threshold according to some *quality function* based on (sub)graphs. This section demonstrates another *pattern mining problem* which can be mapped onto a *descriptive pattern mining class*. A limited set of optimization has been chosen to highlight how the previously evaluated optimization methods also apply to this new problem setting.

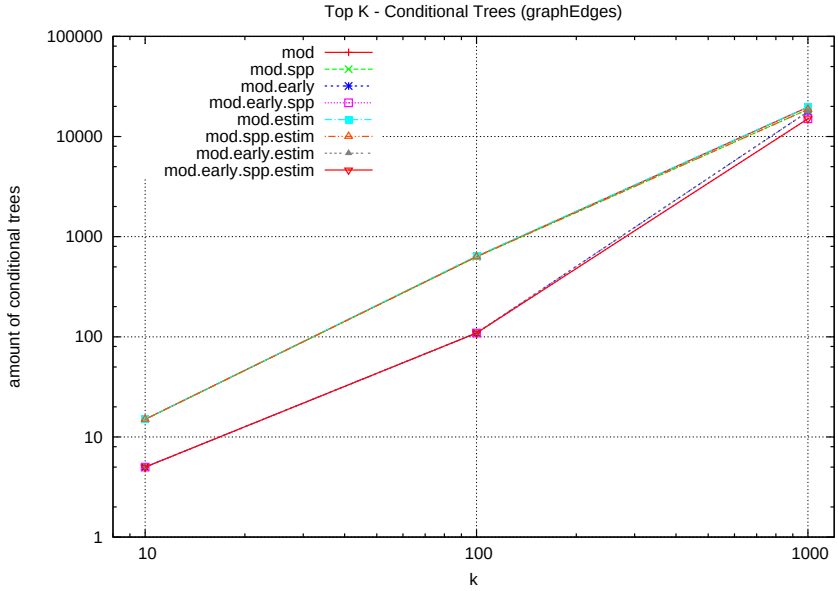
Community mining is based on a *quality constraint*. The objective of the search in this section was chosen to be *top-k mining*. The corresponding *quality function* is the *modularity quality function* featuring an *optimistic estimate* (see Section 4.1.2). The dataset being used is called *graphEdges* and is based on the visit-graph G_V as well as the user/topic relation featuring 100 topics as mentioned in [6]. Information about the dimensions of the resulting

GraphEdges	
instances	5543
topic (items)	100

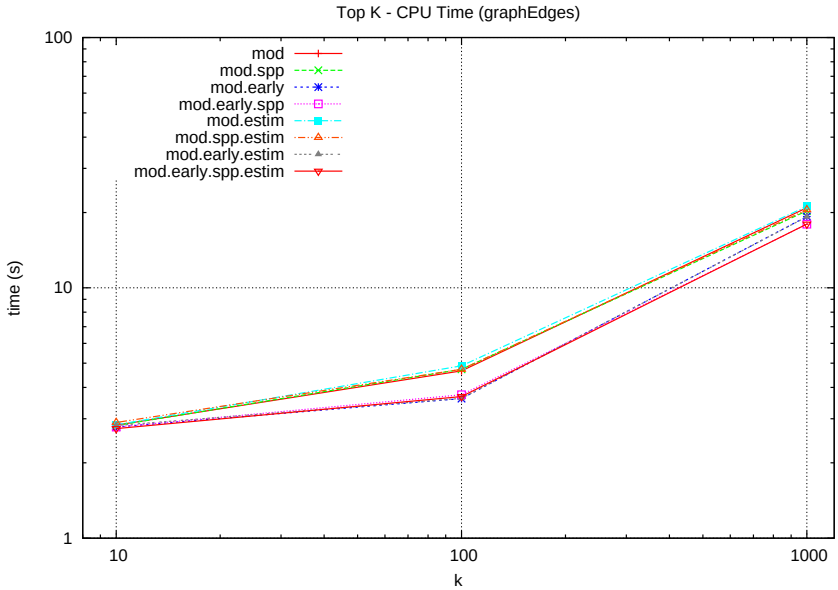
Table 6.3: Information about the *graphEdges* dataset.

dataset are given by Table 6.3.

Figure 6.36 shows the number of conditional trees generated during the search. The dominating factor in optimizing the search is adding *descriptive pattern* early to the result (*early*). The characteristics of the dataset and of the constraints do not seem to favor any other sort of optimization in particular like exploiting the *single prefix path* (*spp*) or sorting branches by *optimistic estimate* (*estim*). Figure 6.36 contains many algorithm settings. To confirm the dominance of the *early adding* (*early*), Figure 6.37 shows the same figure, but only for the *mod* and the *mod.early* setting.

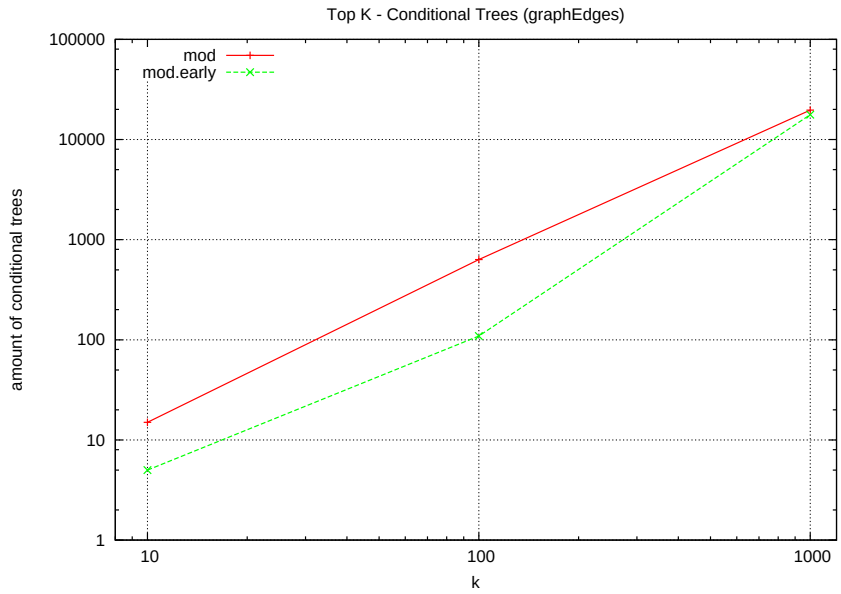


(a) Amount of conditional trees.

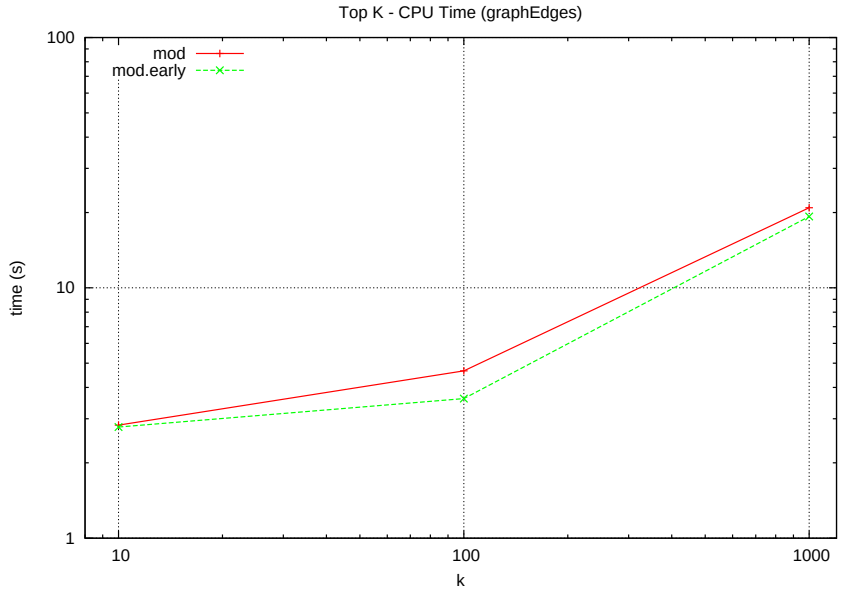


(b) Runtime.

Figure 6.36: The amount of generated conditional trees and the runtime on the *graphEdges* dataset for *community mining* at different result set limits.



(a) Amount of conditional trees.



(b) Runtime.

Figure 6.37: The amount of generated conditional trees and the runtime on the *graphEdges* dataset for *community mining* at different result set limits.

7

Conclusion

The last chapter summarizes the results and contributions of this work and lists perspectives for future work.

7.1 Summary

A framework was introduced allowing to formalize the problem of *pattern mining* and its specialization *descriptive pattern mining*. Classes of these problems can be specified by defining *constraints* utilizing the notion of *valuation bases*. *Valuation bases* express useful properties of the data which patterns are evaluated against.

General search strategies were reviewed and applied to *descriptive pattern mining*. Optimizations based on *constraints* derived from more specific *pattern mining* tasks like *frequent pattern mining* were incorporated into the search such as *anti-monotone pruning* and the exploitation of the *ExAnte property*. To allow for a more efficient search, data structures common to the *pattern mining* community were revisited and it was discussed how they can be applied to the general problem setting defined by the framework. The FP-Tree data structure turned out to be highly compatible with the concept of *valuation bases* keeping the stored data to a minimum under the condition that *valuation bases* are of constant size.

Several *pattern mining instances* were defined using the framework by specifying a set of constraints including frequent pattern mining (see [4]), subgroup mining (see [56, 7]) and mining descriptive community patterns (see [6]). The advantage of abiding by the framework specifying *descriptive*

pattern mining instances by *constraints* and *valuation bases* lies in the extendability by adding arbitrary constraints while any optimization introduced to algorithms solving the *descriptive pattern mining* problem are directly applicable to any specific problem setting.

The introduced strategies and optimizations were compiled into the *AnEMonE* algorithm featuring pruning based on *anti-monotone constraints*, exploitation of the *ExAnte property* as well as *single prefix paths*. *AnEMonE* was evaluated against several *data mining classes* on different datasets comparing the different optimization methods. The results met the expectations based on their counterparts applied to the more specific problem they were originally designed for. This raises the expectation to combine more optimizations into a single general algorithm.

7.2 Outlook

The generic approach of the introduced framework provides a layout for future algorithms that cope with a wide variety of different problem settings in the *descriptive pattern mining* domain. The *AnEMonE* algorithm is a first step into that direction. The *pattern mining* community knows many optimizations for specialized problem settings. It will be interesting to see which ones can be ported to the abstract problem of *descriptive pattern mining* especially focusing on the synergetic effect of different constraints as mentioned in [13, 22]. But there are also other concepts that are worth exploring.

The concept of *closed descriptive pattern* descriptions and how it is to be formulated by using the *descriptive pattern mining* framework was reviewed in Section 4.1.3. Only minimal optimizations from [54] were utilized to improve the search still requiring *valuation bases* of inconstant size. Yet due to its confined constraint settings [54] was able to introduce a way to check patterns without the use of such a *valuation bases*. It may be possible to derive an algorithm applying a similar approach to the more general problem setting of *descriptive pattern mining*. As such methods come at certain overhead the performance variations are difficult to guess. There are also improvements on the TFP algorithm from [54] that are worth to be taken into consideration (cf. [51]).

Community mining was also formalized using *valuation bases* that are not of constant size (see Section 4.1.2). In combination with the *GP-Tree* structures this leads to exponential growth of space requirements. To solve this problem it may be possible to use a link structure between *valuation bases* associated with parent and child nodes in the tree.

Further optimizations concerning the underlying data structures are possible. For example bitset representations of datasets [40] yield a better performance on dense data while the *GP-Tree* structure works well on sparse data. Thus a combination of both can improve the efficiency of respective algorithms. Also, as datasets grow the data structure instances can exceed the size of available memory. According measures need to be taken.

As performance is an important issue in *descriptive pattern mining* it is inevitable to develop mechanisms that allow for parallelization. [42] is a particularly interesting approach due to its simplicity and will be further investigated. As known from other work, like [52], dynamic constraints pose the main problem especially due to dynamic *branching order*. Yet first inquiries showed that it is possible to derive an algorithm guaranteeing sequential runtime with minimal communication costs.

When introducing dynamic constraints the *item* and *branching orders* are important to process a *descriptive pattern mining instance* efficiently. Other optimizations like exploiting the *ExAnte property* can also increase or decrease efficiency depending on the data being provided. Thus a challenging subject of further work is to create an algorithm that learns how to set such parameters dependent on the given data by taking the *descriptive pattern mining instances* it has processed into account.

A similar approach to define a generic framework for *pattern mining* has been proposed by [39], yet lacks an equivalent to the notion of *valuation bases* and does not propose an efficient algorithm to solve the generally stated *pattern mining* problem. On the other hand many models for pattern evaluation are reviewed. This raises the question whether these problem settings can be formulated using *valuation bases* and how efficiently they can be solved by the *AnEMonE* algorithm or respective enhancements.

Bibliography

[1] Frequent itemset mining dataset repository, 2004.

[2] Frequent itemset mining implementations, 2004.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the 20th international conference on Very Large Data Bases (VLDB'94)*, pages 478–499. Morgan Kaufmann, September 1994.

[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22:207–216, June 1993.

[5] Martin Atzmueller and Florian Lemmerich. Fast subgroup discovery for continuous target concepts. In *Proceedings of the 18th International Symposium on Foundations of Intelligent Systems*, ISMIS '09, pages 35–44, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Martin Atzmueller and Folke Mitzlaff. Efficient descriptive community mining. In R. Charles Murray and Philip M. McCarthy, editors, *FLAIRS Conference*, 2011.

[7] Martin Atzmueller and Frank Puppe. SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery. In *Proc. 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2006)*, pages 6–17, 2006.

[8] Martin Atzmuller. *Knowledge-Intensive Subgroup Mining: Techniques for Automatic and Interactive Discovery*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2007.

[9] Roberto J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 85–93, New York, NY, USA, 1998. ACM.

[10] F. Bonchi and B. Goethals. Fp-bonsai: the art of growing and pruning small fp-trees. In *Proceedings of the Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, volume 3056 of *LNCS*, pages 155–160. Springer, 2004.

[11] Francesco Bonchi, Fosca Giannotti, Alessio Mazzanti, and Dino Pedreschi. Adaptive constraint pushing in frequent pattern mining. In Nada Lavrac, Dragan Gamberger, Ljupco Todorovski, and Hendrik Blockeel, editors, *Knowledge Discovery in Databases: PKDD 2003*, volume 2838 of *Lecture Notes in Computer Science*, pages 47–58. Springer Berlin / Heidelberg, 2003.

[12] Francesco Bonchi, Fosca Giannotti, Alessio Mazzanti, and Dino Pedreschi. Exante: A preprocessing method for frequent-pattern mining. *IEEE Intelligent Systems*, 20(3):25–31, 2005.

[13] Francesco Bonchi and Claudio Lucchese. Extending the state-of-the-art of constraint-based pattern discovery. *Data Knowl. Eng.*, 60:377–399, February 2007.

[14] Jean-François Boulicaut, Artur Bykowski, and Christophe Rigotti. Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Min. Knowl. Discov.*, 7:5–22, January 2003.

[15] Jean-Francois Boulicaut, Artur Bykowski, and Christophe Rigotti. Approximation of frequency queris by means of free-sets. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '00, pages 75–85, London, UK, 2000. Springer-Verlag.

[16] Artur Bykowski and Christophe Rigotti. A condensed representation to find frequent patterns. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 267–273, New York, NY, USA, 2001. ACM.

[17] Artur Bykowski and Christophe Rigotti. Dbc: a condensed representation of frequent patterns for efficient mining. *Inf. Syst.*, 28:949–977, December 2003.

[18] Toon Calders and Bart Goethals. Mining all non-derivable frequent itemsets. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *Principles of Data Mining and Knowledge Discovery*, volume 2431 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 2002.

[19] Toon Calders and Bart Goethals. Minimal k-free representations of frequent sets. In Nada Lavrac, Dragan Gamberger, Ljupco Todorovski, and Hendrik Blockeel, editors, *Knowledge Discovery in Databases: PKDD 2003*, volume 2838 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin / Heidelberg, 2003.

[20] Toon Calders, Christophe Rigotti, and Jean-François Boulicaut. A survey on condensed representations for frequent sets. In Jean-François Boulicaut, Luc De Raedt, and Heikki Mannila, editors, *Constraint-Based Mining and Inductive Databases*, volume 3848 of *Lecture Notes in Computer Science*, pages 64–80. Springer Berlin / Heidelberg, 2006.

[21] James Cheng, Yiping Ke, and Wilfred Ng. Tolerance closed frequent itemsets. *Data Mining, IEEE International Conference on*, 0:139–148, 2006.

[22] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 204–212, New York, NY, USA, 2008. ACM.

[23] Luc De Raedt and Stefan Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 2*, pages 853–859, 2001.

[24] Luc De Raedt and Albrecht Zimmermann. Constraint-based pattern set mining. In *Proceedings of the Seventh SIAM International Conference on Data Mining*, pages 1–12. SIAM, SIAM, 2007.

[25] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Advances in knowledge discovery and data mining. chapter From data mining to knowledge discovery: an overview, pages 1–34. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[26] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in knowledge discovery and*

data mining. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[27] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[28] Stum Gerd, Rafik Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal. Computing iceberg concept lattices with titanic. *Data Knowl. Eng.*, 42:189–222, August 2002.

[29] Bart Goethals and Mohammed J. Zaki. Advances in frequent itemset mining implementations: report on fimi'03. *SIGKDD Explor. Newsl.*, 6:109–117, June 2004.

[30] Henrik Grosskreutz, Stefan Rüping, and Stefan Wrobel. Tight optimistic estimates for fast subgroup discovery. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ECML PKDD '08, pages 440–456, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[32] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[33] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8:53–87, 2004.

[34] Baptiste Jeudy and Jean-François Boulicaut. Optimization of association rule mining queries. *Intell. Data Anal.*, 6:341–357, September 2002.

[35] Willi Klösgen. Explora: a multipattern and multistrategy discovery assistant. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in knowledge discovery and data mining*, pages 249–271. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[36] W. Klösgen and J. Zytkow. *Handbook of data mining and knowledge discovery*. Oxford University Press, Oxford, 2002.

[37] Laks V. S. Lakshmanan, Raymond Ng, Jiawei Han, and Alex Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 157–168, New York, NY, USA, 1999. ACM.

[38] Nada Lavrač, Branko Kavšek, Peter Flach, and Ljupčo Todorovski. Subgroup discovery with cn2-sd. *J. Mach. Learn. Res.*, 5:153–188, December 2004.

[39] Dennis Leman, Ad Feelders, and Arno Knobbe. Exceptional model mining. In *Proceedings of the European conference on Machine Learning and Knowledge Discovery in Databases - Part II*, ECML PKDD '08, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Florian Lemmerich, Mathias Rohlfs, and Martin Atzmüller. Fast discovery of relevant subgroup patterns. In Hans W. Guesgen and R. Charles Murray, editors, *FLAIRS Conference*. AAAI Press, 2010.

[41] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Technical Report arXiv:0810.1355, Oct 2008. Comments: 66 pages, a much expanded version of our WWW 2008 paper.

[42] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 107–114, 2008.

[43] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, Feb 2004.

[44] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 13–24, New York, NY, USA, 1998. ACM.

[45] Zaki Parthasarathy Ogihara, M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *In 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.

[46] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In Catriel Beeri and Peter Buneman, editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1999.

[47] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25 – 46, 1999.

[48] Jian Pei, Guozhu Dong, Wei Zou, and Jiawei Han. On computing condensed frequent pattern bases. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 378–, Washington, DC, USA, 2002. IEEE Computer Society.

[49] Jian Pei, Jiawei Han, and Laks V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. *Data Engineering, International Conference on*, 0:0433, 2001.

[50] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[51] Andrea Pietracaprina and Fabio Vandin. Efficient incremental mining of top-k frequent closed itemsets. In *Proceedings of the 10th international conference on Discovery science*, DS'07, pages 275–280, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] Mathias Rohlfs, Frank Puppe, Martin Atzmüller, and Florian Lemmerich. Techniken zur effizienten und verteilten Subgruppenentdeckung in großen Datenmengen. Master's thesis, Bayerische Julius-Maximilians-Universität Würzburg, 2009.

[53] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver.3: collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, OSDM '05, pages 77–86, New York, NY, USA, 2005. ACM.

[54] Jianyong Wang, Jiawei Han, Ying Lu, and Petre Tzvetkov. Tfp: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 17:652–664, 2005.

[55] Jianyong Wang, Jiawei Han, and Jian Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *KDD*, pages 236–245, 2003.

[56] Stefan Wrobel. An algorithm for multi-relational discovery of subgroups. In *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, PKDD '97, pages 78–87, London, UK, 1997. Springer-Verlag.

[57] Mohammed J. Zaki. Generating non-redundant association rules. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 34–43, New York, NY, USA, 2000. ACM.

[58] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12:372–390, May 2000.

[59] Mohammed Javeed Zaki and Ching-Jiu Hsiao. Charm: An efficient algorithm for closed itemset mining. In Robert L. Grossman, Jiawei Han, Vipin Kumar, Heikki Mannila, and Rajeev Motwani, editors, *SDM*. SIAM, 2002.

Danksagung

Danke an alle, die mich tatkräftig unterstützt haben.