VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MINING MULTI-LEVEL SEQUENTIAL PATTERNS

DOLOVÁNÍ VÍCEÚROVŇOVÝCH SEKVENČNÍCH VZORŮ

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. MICHAL ŠEBEK
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. JAROSLAV ZENDULKA, CSc.
SUPERVISOR

BRNO 2016

## Abstrakt

Dolování sekvenčních vzorů je důležitá oblast získávání znalostí z databází. Stále více průmyslových a obchodních aplikací uchovává data mající povahu sekvencí, kdy je dáno pořadí jednotlivých transakcí. Toho může být využito například při analýze po sobě jdoucích nákupů zákazníků.
Tato práce se zabývá využitím hierarchického uspořádání položek při dolování sekvenčních vzorů. V rámci práce jsou řešeny dvě základní oblasti – dolování víceúrovňových sekvenčních vzorů s křížením a bez křížení úrovní hierarchií. Dolovací úlohy pro obě oblasti jsou v práci formalizovány a následně navrženy algoritmy hGSP a MLSP pro jejich řešení. Experimentálně bylo ověřeno, že především algoritmus MLSP dosahuje výborných výkonnostních vlastností a stability. Význam nově získaných vzorů je ukázán na dolování reálných produkčních dat.

## Abstract

Mining sequential patterns is a very important area of the data mining. Many industrial and business applications save sequential data where the ordering of transactions is defined. It can be used for example for analysis of consecutive shopping transactions.
This thesis deals with the using of concept hierarchies of items for mining sequential patterns. This thesis focuses on two basic approaches – mining level-crossing sequential patterns and mining multi-level sequential patterns. The approaches for the both data mining tasks are formalized and there are proposed data mining algorithms hGSP and MLSP to solve these tasks. Experiments verified that mainly the MLSP has good performance and stability. The usability of newly obtained patterns is shown on the real-world data mining task.

## Klíčová slova

získávání znalostí z databází, dolování sekvenčních vzorů, konceptové hierarchie, uzavřené vzory, víceúrovňové sekvenční vzory

## Keywords

data mining, mining sequential patterns, concept hierarchies, closed patterns, level-crossing sequential patterns, multi-level sequential patterns

## Citace

Michal Šebek: Dolování víceúrovňových sekvenčních vzorů, disertační práce, Brno, FIT VUT v Brně, 2016

## Dolování víceúrovňových sekvenčních vzorů

## Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením pana doc. Ing. Jaroslava Zendulky, CSc.

. . . . . . . . . . . . . . . . . . . . . .
Michal Šebek
23. června 2016

## Poděkování

Rád bych poděkoval mému školiteli doc. Ing. Jaroslavu Zendulkovi, CSc. za vedení, pomoc a poskytnutí cenných rad během mého výzkumu a při psaní této práce. Dále bych rád poděkoval firmám AVG Technologies CZ, s.r.o., v rámci níž vznikala výzkumná část této práce, a firmě VOPI s.r.o. za poskytnutí dat pro experimentální ověření metod. V neposlední řadě patří dík i mé rodině za morální podporu po celou dobu mého studia.

# Contents

# Chapter 1

# Introduction

Nowadays, the total amount of stored data in different kinds of databases is growing. Many different applications save data about each transaction. For example, data about merchant transactions are saved for billing purposes. If the data are collected from a high number of customers, new dependencies about the customers' behavior can be formed. Another example is that data about insurance events can be stored. If the reporting period is long enough, the data can be used for the risk analysis of new contracts. Retrieving of such new knowledge from data is called *Data Mining* (or *Knowledge Discovery)* introduced in early 90th of 20th century.

An established definition is that *Data Mining is an extraction or "mining" of hidden knowledge from large amounts of data* [19]. Data Mining is a complex process where the application of the data mining algorithm is only one step of the process. The process is composed of following steps: data pre-processing (data cleaning, integration, transformation and reduction), data mining, pattern evaluation and knowledge presentation.

Various types of databases require different data mining tasks and provide different kinds of patterns. Examples of main data mining tasks and types of data to be mined are following:

- Classes Characterization and Discrimination – data are associated with classes and characterized and/or compared.
- Mining Frequent Patterns, Associations, and Correlations – frequent patterns are such patterns that occur frequently in data (in other words, data which occur in the dataset in a number which is higher than the given threshold value).
- Classification and Prediction – data are labeled into classes. The algorithms of classification try to find a model which describes each class and can be applied on new (unlabeled) data. The prediction has a continuous target attribute.
- Cluster Analysis – clustering algorithms try to find a model which can divide data into a specific number of groups. Clustering can be used for an initial labeling of data.

- Outlier Analysis – algorithms for outlier analysis reveal data records which have different values than the majority. Such problem is typically used for fraud detection.

This thesis deals with *mining sequential patterns*. Mining sequential patterns is a special case of mining frequent patterns with a defined order of transactions. It is used for many applications such as the analysis of customer patterns, web log data purchase, security applications, etc. The goal is to find sequential patterns that occur in the database frequently. Market basket analysis is a typical application example where the sequential patterns like $\langle PC\_minitower\ ink\_printer\rangle$ can be discovered. The pattern says that many people buy a minitower PC and then, later, they also buy an ink printer.

Items in the database can be assorted and categorized into one or more taxonomies. An example of taxonomies of items is shown on Figure 1.1. Taxonomies can be used to find patterns which items are on the different levels of hierarchy. We demonstrate it on the example of customer purchase analysis. Following the sequential pattern example above, the pattern $\langle PC\ printer\rangle$ can be found by replacing all items by items on a higher hierarchy level. Unfortunately, the amount of such patterns can grow enormously, but many of the patterns can be considered as useless. For instance, the pattern $\langle PC\ printer\rangle$ does not bring any new information if the number of its occurrence in the database is the same as a number of $\langle PC\_minitower\ ink\_printer\rangle$.

**Example 1.** For better illustration of the practical impact of the issue being solved, the thesis uses a simple real world example from a PC shop. There is an illustration of several categories representing products of the shop on Figure 1.1.

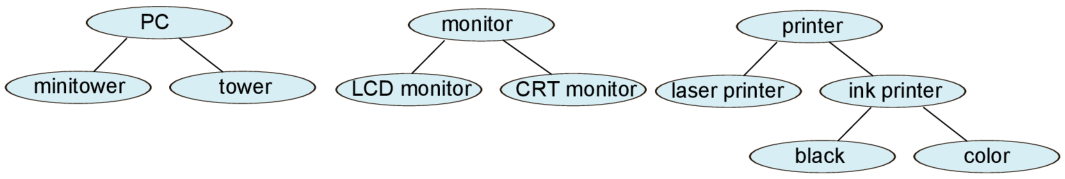

Figure 1.1: Example of the products structure in the shop.

When items are categorized in taxonomies, the sequential patterns can be divided into the following two categories:

- *multi-level* (known also as *intra-level*)
- and *level-crossing* (known also as *inter-level*) [17].

All items of *multi-level* sequential patterns are at the same level of hierarchy, whereas levels of items of *level-crossing* can be different. In Chapter 3 it is shown that very few algorithms deal with the problem of mining multi-level sequential patterns. The thesis deals with the both categories of sequential patterns and examines how to mine them effectively.

## 1.1 Goals of the Thesis

The Ph.D. thesis deals with the mining sequential patterns where taxonomies are defined over items in a sequence database. The hypothesis of the thesis is following:

*"The existence of taxonomies makes it possible to find a new type of sequential patterns and a new method for mining it effectively can be developed."*

The goal of the thesis is to verify the hypothesis. The goal is decomposed into the following three sub-goals.

1. To design and formally define the problem of mining sequential patterns with items in taxonomies.
2. To design and formally define a new method(s) or algorithm(s) which can solve the defined data mining problem effectively.
3. To experimentally evaluate properties of the developed method(s) or algorithm(s) and to compare it (them) with the existing methods.

## 1.2 Thesis Contribution

The main contributions of the thesis are as follows.

- The both multi-level and level-crossing categories of mining sequential patterns with items in taxonomies are discussed. There is proposed a new type of multi-level sequential patterns task which reduces redundant (useless) patterns using new constraints.
- New methods for mining of level-crossing and especially multi-level sequential patterns are introduced. There are proposed new optimization techniques which significantly increase the speed of the multi-level mining algorithm. The properties of algorithms are experimentally verified.

## 1.3 Structure of the Thesis

Pattern Mining is introduced in Chapter 2. The first part is focused on frequent pattern mining. Then the sequential pattern mining is defined and a problem is extended by the existence of taxonomies. The state of the art, especially existing algorithms for frequent or sequential pattern mining, is described in Chapter 3. The core of the thesis is described in Chapter 4. The first part deals with level-crossing sequential patterns and the second part is focused on the research of multi-level patterns. Here, a new data mining task is defined and algorithms to solve them are proposed. The experiments and their results are described in Chapter 5. First, the performance is compared on synthetic data, then, the practical results are shown. Finally, the results are summarized and possible following research is suggested in Chapter 6.

# Chapter 2

# Pattern Mining

Mining of frequent patterns is the most common of the data analysis and data mining tasks [19]. Basic concepts of mining frequent patterns and association rules are introduced first. Then, mining sequential patterns and mining with defined concept hierarchy are described. In addition, the chapter gives basic formal background to the pattern mining.

## 2.1 Mining Frequent Patterns

Mining frequent patterns was firstly studied by Agrawal et al. (1993) in the paper [2]. The main objective was to find such sets of items (shortly itemsets) which occur in transactions of input database more frequently than a given threshold. It is widely used to discovery of associations and correlations among input items. It produces simply understandable model of data and, therefore, the task is usually used for initial data analysis of an unknown dataset.

The task became very popular for industry and business, especially for decision making applications and marketing applications. The typical example of usage of the association rules mining is a market basked analysis. The goal is to find items which are usually purchased together. The example can be a typical computer shop which sells items such as computers, notebooks, monitors, printers, keyboards etc. The frequent pattern mining task can reveal that computers are usually purchased together with monitors and keyboards, but notebooks are purchased just alone. The task association rules analysis brings the results in the form of implication. It means that if a customer buys a computer, he probably buys also a keyboard.

### 2.1.1 Problem Definition

Here, the problem is described formally.

**Definition 1. (Itemset)** Let $I = \{i_1, i_2, i_3, \dots, i_k\}$ be a nonempty finite set of items. Then an *itemset* $T$ is a non-empty set of items $I$, such that $T \subseteq I$.

**Definition 2. (Frequent Itemset)** Let $I$ be a set of items, $\mathcal{D}$ be set of transactions, such that each transaction $T$ is $T \subseteq I$, and $A \subseteq I$ be an itemset. The transaction $T$

*contains* an itemset $A$ iff $A \subseteq T$. A relative *support* of the itemset $A$ is a percentage of transactions in $\mathcal{D}$ that contain $A$. Given the minimal support threshold value $min\_sup$, the itemset $A$ is called *frequent itemset* if its support is more than or equal to $min\_sup$.

**Definition 3. (Association Rule, Support of Association Rule, Confidence of Association Rule)** Let $I$ be a set of items, $\mathcal{D}$ be set of transactions and let $A$ and $B$ be itemsets such that $A, B \subseteq I$ and $A \cap B = \emptyset$. Then an *association rule* is the implication $A \Rightarrow B$. The *support of the association rule* $A \Rightarrow B$ is a percentage of transactions of $\mathcal{D}$ that contain $A \cup B$. The *confidence of the rule* $A \Rightarrow B$ is a percentage of transactions in $\mathcal{D}$ containing $A$ which contain also $B$. This means

$$\begin{aligned} support(A \Rightarrow B) &= P(A \cup B), & (2.1)\\ confidence(A \Rightarrow B) &= P(B|A) = \frac{support(A \cup B)}{support(A)}. & (2.2) \end{aligned}$$

**Definition 4. (Mining Frequent Patterns)** Given a database $\mathcal{D}$ and a minimal support threshold $min\_sup$, the task of finding of the complete set of frequent itemsets is called the *mining frequent patterns.*

### 2.1.2 Mining Maximal and Closed Frequent Itemsets

The huge number of result itemsets can be reduced using the maximal and closed restrictions of frequent itemsets [14, 30].

**Definition 5. (Maximal Frequent Itemset)** Let $F$ be a set of frequent itemsets. A frequent itemset $x \in F$ is called *maximal frequent itemset* if it is not a proper subset of any other frequent itemset $x' \in F$.

**Definition 6. (Frequent Closed Itemset)** Let $F$ be a set of frequent itemsets. A frequent itemset $x \in F$ is called *closed frequent itemset* if it is not a proper subset of any other frequent itemset $x' \in F$ such that $support(x) = support(x')$.

**Definition 7. (Mining Maximal/Closed Frequent Patterns)** Given a database $\mathcal{D}$ and a minimal support threshold $min\_sup$, the task of finding of the complete set of maximal (closed) itemsets is called the *frequent maximal (closed) patterns mining.*

## 2.2 Mining Sequential Patterns

Sequential pattern mining was introduced by Agrawal and Srikant in 1995 [1]. The sequence is defined as an ordered list of transactions (itemsets) of one customer. The example of usage of the sequential pattern mining can be also demonstrated on market basket analysis. It can be expected that customers return for further purchases. Therefore, the sequential patterns over purchased items can be found. The example of such sequence can be, that a customer buys the computer with a monitor in the one purchase and, later, the customer returns and buys a printer. If a sequence occurs in database more than a given threshold, it is called the sequential pattern.

### 2.2.1 Problem Definition

In this section the problem of mining sequential patterns is formalized. Firstly, the basic terms such as item, itemset, sequence and sequence database should be defined.

**Definition 8. (Sequence)** A *sequence* is an ordered list of itemsets. A sequence $s$ is denoted by $\langle s_1 s_2 s_3 \dots s_n \rangle$, where $s_j$ for $1 \leq j \leq n$ is an itemset. The itemset $s_j$ is also called an *element* of the sequence. The *length* of a sequence is defined as the number of instances of items in the sequence. A sequence of length $l$ is called an *l-sequence*. The sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$ is a *subsequence* of the sequence $\beta = \langle b_1 b_2 \dots b_m \rangle$ where $n \leq m$ if there exist integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$. We say that the sequence $\alpha$ is contained in the sequence $\beta$. We denote it $\alpha \sqsubseteq \beta$ and $\beta$ is a *supersequence* of $\alpha$ .

**Definition 9. (Sequence database)** A sequence database $\mathcal{D}$ is a set of tuples $\langle SID, s \rangle$, where SID is a sequence identifier and $s$ is a sequence.

**Definition 10. (Sequence Support)** Given sequence database $\mathcal{D}$, the support of a sequence $s_1$ in $\mathcal{D}$ is defined as the number of sequences in $\mathcal{D}$ containing a subsequence $s_1$. Formally stated, the support of a sequence $s_1$ in $\mathcal{D}$ is $support(s_1) = |\{\langle SID, s \rangle | (\langle SID, s \rangle \in D) \land (s_1 \sqsubseteq s)\}|$.

**Definition 11. (Sequential Pattern, Mining Sequential Patterns)** Given sequence database $\mathcal{D}$ and *minimal support threshold min_supp*, a *frequent sequence* is such a sequence $s$ whose $support(s) \geq min\_sup$. A frequent sequence is called *a sequential pattern*. For a given sequence database $D$ and a minimal support *min_supp*, the goal of *mining sequential patterns* is to find all frequent sequences in $\mathcal{D}$.

**Example 2. (Item, Element, Sequence, Sequence Database, Sequential Pattern)** For better understandability, the examples in the thesis are based on the *sequence database* in Table 2.1. The set of *items* for the example is the set $I = \{a_{11}, a_{12}, a_1, a_2, b_1, b_2, c_1, d_1, e_1, f_1, f_2, g_1, g_2, h_1, h_2\}$. The table represents a sequence database with sequences in the sequence column. Let's focus on the first row containing the sequence $s = \langle (c_1 d_1)(a_{12} b_1 c_1)(a_1 b_2 f_1)(a_{11} c_1 d_1 f_1) \rangle$. The sequence length is twelve, therefore the sequence is called *12-sequence*. The sequence consists of four *elements* (*itemsets*): $(c_1 d_1)$, $(a_{12} b_1 c_1)$, $(a_1 b_2 f_1)$ and $(a_{12} c_1 d_1 f_1)$. Note that if element contains only one item, than the parentheses around the itemset can be omitted, e.g. element $e_1$ on the second row. Items denoted by the same letter, which differ only in indexes, belong to one taxonomy. This notation will be described later in Example 4 on page 11.

For the following examples we assume the minimal support threshold *min_supp*=2, unless otherwise stated. The *support* of item $b_2$ is 3, denoted as $\langle b_2 \rangle : 3$, because the item is included in three sequences with $SID$ 1, 2 and 3. Therefore, the 1-sequence

$\langle b_2 \rangle$ is frequent and is called *sequential pattern*. In contrast, the support of 1-sequence $\langle g_1 \rangle$ is 1 and it is not frequent. Further, the 2-sequences$\langle d_1 b_1 \rangle : 2$ , $\langle d_1 f_1 \rangle : 2$ and $\langle (b_2 f_2) \rangle : 2$ are *sequential patterns* of length 2 with the support 2, e.g. the first sequence $\langle d_1 b_1 \rangle : 2$ is the subsequence of sequences 1 and 4 of the sequence database. In the sequence database there is no any sequential pattern longer than two.

Table 2.1: A sequence database $\mathcal{D}$ containing items on different taxonomy levels.

| SID | Sequence |
|---|---|
| 1 | $\langle (c_1\, d_1)(a_{12}\, b_1\, c_1)(a_1\, b_2\, f_1)(a_{11}\, c_1\, d_1\, f_1) \rangle$ |
| 2 | $\langle (a_{12}\, b_2\, f_2)\; e_1 \rangle$ |
| 3 | $\langle (a_2\, b_2\, f_2) \rangle$ |
| 4 | $\langle a_{11}\; (d_1\, g_1\, h_1)(b_1\, f_1)(a_2\, g_2\, h_2) \rangle$ |

Table 2.2 contains items without indexes called *root nodes* (*root nodes* of taxonomies are described later in Section 10). Now, it will be compared the result of the same mining task using the database $\mathcal{D}_{root}$ of root items. The support of item $b$ is 4 because it is contained in elements of all sequences in database. Note that the support $b_2$ was only 3. Moreover, the longest sequential patterns in $\mathcal{D}_{root}$ are of length 4, for example the sequential pattern $\langle a\; (bf)\; a \rangle : 2$ which is contained in the sequences 1 and 4 of the database $\mathcal{D}_{root}$.

Table 2.2: A sequence database $\mathcal{D}_{root}$ of items without indexes from Table 2.1 for running example.

| SID | Sequence |
|---|---|
| 1 | $\langle (c\, d)(a\, b\, c)(a\, b\, f)(a\, c\, d\, f) \rangle$ |
| 2 | $\langle (a\, b\, f)\; e \rangle$ |
| 3 | $\langle (a\, b\, f) \rangle$ |
| 4 | $\langle a\; (d\, g\, h)(b\, f)(a\, g\, h) \rangle$ |

### 2.2.2 Mining Maximal and Closed Sequential Patterns

Restrictions over the sequence length and support can be defined. In general, there are two restrictions similar to those introduced in Def. 5 and Def. 6 – maximal sequential patterns and closed sequential patterns. In the case of maximal sequential patterns we are interested in sequences whose supersequences are not frequent (simply, algorithms find the longest sequences). This problem was deeply studied by Wang et al. [45]. In the case of closed sequential patterns proposed by Agrawal in [1], the change of support of the supersequences is important and it also bring us some information. In this case, we omit only subsequences whose support is the same as theirs supersequences.

**Definition 12.** (Closed Sequential Pattern) Given sequence database $\mathcal{D}$ and a frequent sequence $s$. If there is no proper supersequences of $s$ with the same support, i.e. $\nexists s'$ such that $s \sqsubset s'$ and $support(s) = support(s')$, the sequence $s$ is called *closed sequential pattern*.

**Definition 13.** (Maximal Sequential Pattern) Given sequence database $\mathcal{D}$, a frequent sequence $s$ and *minimal support threshold min_supp*. If there is no proper frequent supersequence of $s$, i.e. $\nexists s'$ such that $s \sqsubset s'$ and $support(s') \geq min\_supp$, the sequence $s$ is called *maximal sequential pattern*.

**Example 3.** One of the longest sequential patterns in Table 2.2. is $\langle a\ (bf)\ a \rangle : 2$ which is also a closed and maximal sequential pattern. The sequential pattern $\langle aba \rangle$ : 2 is neither the *closed*, nor *maximal* sequential pattern because its support is equal to supersequence $\langle a\ (bf)\ a \rangle : 2$.

## 2.3 Concept Hierarchy and Taxonomies

Concept hierarchy allows describing relations between concepts (values of attributes) in database. The usage of concept hierarchy for data mining is summarized in [19] and [10]. In general, the concept hierarchies define relations between lower (more specific) and higher (more general) concepts. Formally, the concept hierarchy is a partially (or totally) ordered set of concepts. A special case of concept hierarchy is a hierarchy of items referred as taxonomy.

**Definition 14. (Concept Hierarchy)** A Concept Hierarchy $\mathcal{CH}$ is a partially ordered set $(\mathbf{CH}, \preceq)$, or respectively a totally ordered set $(\mathbf{CH}, \prec)$, where $\mathbf{CH}$ is a finite set of concepts, and $\preceq$ and $\prec$ are partial and total order over $\mathbf{CH}$, respectively.

The concept hierarchy can be defined explicitly or can exist implicitly in the database. In the case of implicitly defined concept hierarchy, the levels of hierarchy are encoded by the database schema. For example for *location* dimension, the database schema could contain attributes *street*, *city*, *province* and *country*. Moreover, the *location* concept hierarchy is totally ordered as following *street*$\prec$*city*$\prec$*province*$\prec$*country*. In contrast, we can expect a partially ordered concept hierarchy in *time* dimension such as (1) *day*$\prec$*month*$\prec$*quarter*$\prec$year or (2) *day*$\prec$*week*$\prec$year. The partial order can be represented by lattice shown on Figure 2.1.

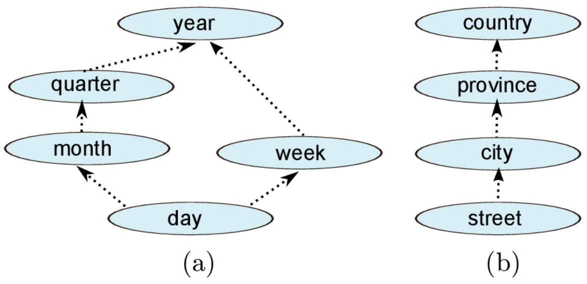

Figure 2.1: The hierarchy of concepts for (a) time dimension and (b) location.

**Definition 15. (Taxonomy)** The taxonomy structure of an itemset $V$ (abbr. *taxonomy*) and edges $E$ is a rooted tree $\tau = (V, E)$ with a root $r \in V$. In the context of the tree, we refer to $V$ as a set of nodes representing items. For each node $v$ in the tree, let $UP(v)$ be a simple unique path from $v$ to $r$. If $UP(v)$ has exactly $k$ edges then the *level* of $v$ is $k$ for $k \geq 0$. The level of the root is 0. The *height* of a taxonomy is the greatest level in the tree. The *parent* of $v \neq r$, formally $parent(v)$, is the neighbor of $v$ on $UP(v)$, and for each node $v \in V, v \neq r$ there exists a set of its *ancestors* defined as:

$$ancestors(v) = \{x | x \in UP(v), x \neq v\}. \tag{2.3}$$

The parent of $r$ and the ancestors of $r$ are not defined. If $v$ is the parent of $u$ then $u$ is a *child* of $v$. A *leaf* is a node having no child [28].

In every taxonomy there exists a *is-a relation* which is defined as follows:

$$is - a : V \times V :\equiv \{(a, b) | b \in ancestors(a)\}. \tag{2.4}$$

Let $\iota = \{I_1, \ldots, I_m\}$ be a partition of a nonempty finite set of items $I$. Then a set of taxonomy structures of items $I$ is a nonempty set of taxonomy structures $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ corresponding to $\iota$ such that $\tau_i = (I_i, E_i)$ where $I_i \in \iota$ for $1 \leq i \leq m$. It means that each item $i \in I$ appears in exactly one taxonomy structure $\tau_i \in \mathcal{T}$. It should be noted that we do not require that items need to be only leaf nodes. Items $ancestors(i)$ will be referred to as *generalized* items of $i$.

**Example 4. (Taxonomy of Items, Taxonomy Level, Parent, Ancestor, Generalized Item.)** The tree structures on Figure 2.2 are called *Taxonomies of Items* which are used in the running example. The *root* symbols are alone letters from $a$ to $h$ which are called *root items*. Then, all descendants are denoted by down-indexes. By the definition, the *level* of item is the number of edges from item to root item, for example the *level* of $a_{12}$ is 2. Note, that the count of down-index digits denotes the level of the item and the digit value denotes ordering of the item on the current taxonomy level.

Now, we focus on *relations* between items in the taxonomies. The $a_1$ is a *parent* of both $a_{11}$ and $a_{12}$, denoted as $parent(a_{11}) = a_1$. Each item has almost one parent item. In contrast, the *ancestors* are a set for each item laying on the path to root, for example $a_{11}$ has two ancestors $a_1$ and $a$, denoted as $ancestor(a_{11}) = \{a_1, a\}$. The *generalized items* of item $a_{11}$ are both $a_1$ or $a$.

### 2.3.1 Mining Multi-Level and Level-Crossing

The necessity of mining association rules on different concept levels has been firstly mentioned by Agrawal et. al. in [4]. It is important to deal with multiple level pattern mining because association rules over leaf items may not satisfy minimal support but association rules over more general items in the taxonomy may satisfy it.

Therefore, the task of mining association rules is extended to the form of the generalized association rules [38] by the Def. 16.

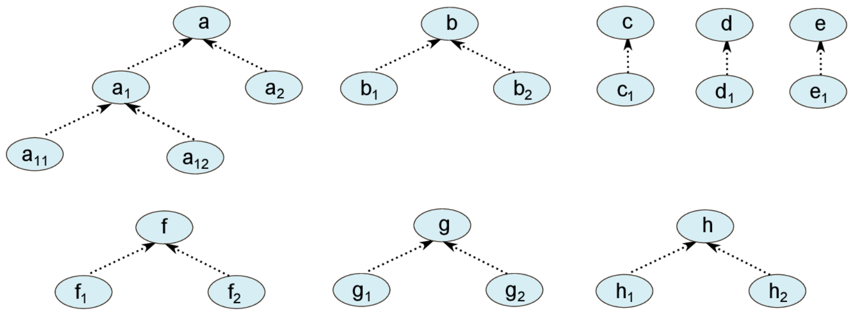

Figure 2.2: Visualization of taxonomies over items from the Example .

**Definition 16. (Generalized Association Rule)** Let $\mathcal{D}$ be a set of transactions, $\mathcal{T}$ be a set of taxonomies and $I$ the set of all items, where each transaction $T$ is a set of items such that $T \subseteq I$. A transaction $T$ *supports* an item $x \in I$ if $x \in T$ or $x$ is an ancestor of some item in $T$. A transaction $T$ *supports* a set $X \subseteq I$ if $T$ supports every item in $X$. Then, a *generalized association rule* is an implication $A \Rightarrow B$, where $A, B \subseteq I, X \cap Y = \emptyset$ and no item in $Y$ is an ancestor of any item in $X$. The support of the generalized association rule $A \Rightarrow B$ is a percentage of transactions in $\mathcal{D}$ which contain $A \cup B$ according to the *support* defined in this definition. The confidence of the generalized association rule $A \Rightarrow B$ is percentage of transactions in $\mathcal{D}$ supporting $A$ that also support $B$.

**Definition 17. (Mining Generalized Association Rules)** Let $\mathcal{D}$ be a set of transactions and $\mathcal{T}$ be a set of taxonomies. The task of mining generalized association rules is to discover all rules that have support and confidence greater (or equal) than the user specified minimal support and minimal confidence values.

The item ancestor definition can be extended to itemsets by Def. 18 by Srikant et al. [38].

**Definition 18. (Ancestors of Itemset, Ancestor of Rule, Close Ancestor of Rule)** Let $\mathcal{D}$ be a set of transactions, $\mathcal{T}$ be a set of taxonomies and $I$ the set of all items. The $\hat{X} \subseteq I$ is called the *ancestor of an itemset* $X \subseteq I$ if we can get $\hat{X}$ from $X$ by replacing one or more items in $X$ by their ancestors and $|X| = |\hat{X}|$. Rules $X \Rightarrow \hat{Y}$,$\hat{X} \Rightarrow Y$ and $\hat{X} \Rightarrow \hat{Y}$ are all the *ancestors of rule* $X \Rightarrow Y$. Given a set of rules, we call $\hat{X} \Rightarrow \hat{Y}$ a *close ancestor of rule* of $X \Rightarrow Y$ if there is no rule $X' \Rightarrow Y$' such that $X' \Rightarrow Y$' is an ancestor of $X \Rightarrow Y$ and $\hat{X} \Rightarrow \hat{Y}$ is an ancestor of $X' \Rightarrow Y$'.

From the perspective of the structure of association rules, two different approaches to pattern mining with taxonomies are distinguished:

- Multi-level (multiple level) mining – all items of result patterns (or association rules) are on the same level of taxonomies $l$.
- Level-crossing Mining – items of result patterns could be on different levels of taxonomies.

There are several observations related to the phenomenon of the generalization of mining association rules and frequent patterns [38].

1. Let $x$ and $y$ be items $x, y \in I$, $\hat{x}$ be an ancestor of $x$ and $\hat{y}$ be an ancestor of $y$. If a set $\{x, y\}$ satisfies a minimal support *min_supp*, so do all $\{x, \hat{y}\}$, $\{\hat{x}, \hat{y}\}$ and $\{\hat{x}, \hat{y}\}$.

2. If a rule $X \Rightarrow Y$ satisfies a minimal support and minimal confidence, only the rule $X \Rightarrow \hat{Y}$ satisfies a minimal support and a minimal confidence automatically. The rules $\hat{X} \Rightarrow \hat{Y}$ and $\hat{X} \Rightarrow Y$ satisfies minimal support, but they may not satisfy the minimal confidence threshold.

3. The support of an item $x$ in the taxonomy is not equal to the sum of the supports of all its children because several children of $x$ can occur in one transaction in $\mathcal{D}$ together. Therefore, the support of non-leaf items of taxonomy cannot be simply derived from the support of theirs leaf children items.

4. Mining of generalized association rules by the Definition 17 produces many redundant rules and frequent patterns. Then, the analyst is glutted by many related rules (or patterns).

### Interesting rules

The measure of *interest* has been formulated to deal with the *redundant rules* and to filter them. The definition of mining generalized association rules by the Def. 17 is reformulated to *mining interesting association rules.*

The first definition of interesting measure for generalized association rules was formulated in [33] by Piatetsky-Shapiro. There was suggested to filter not interesting rules whose support is expectable

$$support(A \Rightarrow B) \approx support(A) \times support(B). \tag{2.5}$$

However, by Srikant et al. in [38] the interesting measure in equation 2.5 is not sufficient for real world datasets. They demonstrated that only 1% of rules become not interesting in this sense. Therefore, it was introduced a new method for the interesting measure. Consider a rule $A \Rightarrow B$ and $C = A \cup B$, denoted as $C = \{c_1, \ldots, c_n\}$, and its ancestor $\hat{C}$ in the form of $\hat{C} = \{\hat{c_1}, \ldots, \hat{c_j}, c_{j+1}, \ldots, c_n\}$. Then, the expected value of probability $\mathcal{P}(C)$ for given $\mathcal{P}(\hat{C})$ is

$$E_{\hat{C}}\left[P(C)\right] = \frac{P(c_1)}{P(\hat{c_1})} \times \cdots \times \frac{P(c_j)}{P(\hat{c_j})} \times P(\hat{C}). \tag{2.6}$$

Similarly, let $A \Rightarrow B$ be a rule in the form of $B = \{b_1, \ldots, b_n\}$, and $\hat{B}$ be its ancestor in the form $\hat{B} = \{\hat{b_1}, \ldots, \hat{b_j}, b_{j+1}, \ldots, b_n\}$. Then the expected confidence of rule $A \Rightarrow B$ given $\hat{A} \Rightarrow \hat{B}$ is defined

$$E_{\hat{A} \Rightarrow \hat{B}}\left[P(B|A)\right] = \frac{P(b_1)}{P(\hat{b_1})} \times \cdots \times \frac{P(b_j)}{P(\hat{b_j})} \times P(\hat{B}|\hat{A}). \tag{2.7}$$

The a rule $A \Rightarrow B$ is called $R$-interesting if the support of the rule $A \Rightarrow B$, where $C = A \cup B$, is $R$ times the expected support $E_{\hat{C}}[P(C)]$ or the confidence is $R$ times the expected confidence $E_{\hat{A} \Rightarrow \hat{B}}[P(B|A)]$. Using the redefined interesting measure, the interesting rules are defined in Def. 19.

**Definition 19. (Interesting rules)** Given a minimal interest $R$, a rule $A \Rightarrow B$ is called *interesting* if it has no ancestors or it is $R$-interesting w.r.t. its close ancestor rules among its interesting ancestors rules.

# Chapter 3

# State of the Art

Various approaches and algorithms have been developed to solve data mining tasks introduced in Chapter 2. There are two basic approaches to mining frequent and sequential patterns – candidate generation and pattern growth. The candidate generation approach algorithms iteratively generate $(n + 1)$-length patterns from the $n$-length ones. Also, it creates high number of non-frequent candidate sequences during the mining process. The pattern growth approach, by contrast, generates final frequent patterns directly. This chapter is organized into three sections. The first section describes algorithms for mining frequent patterns. The second section is focused on methods for mining sequential patterns. The third section explains the methods for mining more complex sequential patterns. Selected principles are used within new methods for mining multi-level sequential patterns later in Chapter 4.

## 3.1 Mining Frequent Patterns

Frequent patterns are such patterns which occur in a dataset more than a given threshold value. In following subsections, basic approaches to mining frequent patterns are described.

### 3.1.1 Candidate Generation Approach

The first method is based on the candidate generation. It means that during the mining process the candidates are generated and then tested if they are frequent. The basic algorithm for mining frequent patterns is the *Apriori* algorithm presented by Agrawal in [4]. The algorithm works iteratively. The following algorithm assumes that itemsets in $L_i$ and $C_i$ are implemented as lists and itemsets are sorted in lexicographic order. Originally, frequent itemsets were called as *large*, therefore they are often denoted by letter $L$. The operator $x[i]$ is the index operator returning $i$-position item of the sorted list $x$. The algorithm processes the database iteratively. The support of all items in the database is counted in the first iteration. Items with sufficient support form the initial frequent 1-itemsets denoted as $L_1$. Next iterations consist of two phases:

1. In the first *candidate generation phase* of $k-1$ iteration, the candidate itemsets $C_k$ are generated from the frequent itemsets $L_{(k-1)}$. All itemsets from the $L_{(k-1)}$ are tested with all others if they are joinable. The procedure is called the *join step*. The itemsets $l_1 \in L_{(k-1)}$ and $l_2 \in L_{(k-1)}$ are joined into a new candidate itemset $c = \{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$

   $$\text{if} \quad l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \cdots \wedge l_1[k-2] = l_2[k-2] \wedge l_1[k-1] < l_2[k-1]. \quad (3.1)$$

   Then, the “Apriori” theorem is tested in the *pruning step*. The theorem is formulated as that all the $(k-1)$-itemsets of the frequent $k$-itemset are frequent. Therefore, the candidate $k$-itemsets, which contain at least one non-frequent $(k-1)$-subset, are deleted.

2. In the second phase called *counting phase*, the algorithm makes a pass through the database and the support values are counted for all candidate itemsets in the $c \in C_k$. Finally, if the support of the candidate itemset is sufficient according to minimal support threshold, the itemset is marked as frequent one and added into $L_k$.

Different modifications of the Apriori algorithm were published. The *AprioriTid* algorithm [4] counts the support of itemsets of size two and greater from the special data structure $\overline{C_k}$. The data structure is the reduced projection of the database $\mathcal{D}$ which contains potentially frequent itemsets. Therefore, the size of the structure $\overline{C_k}$ may be smaller than $\mathcal{D}$.

Other interesting research issue is the *constrained association rules* defined in [39]. The constrained association rules are focused on the situation when the analyst is interested only in the subset of all association rules which contain a specific item or children of a specific item in a hierarchy.

### 3.1.2 Pattern Growth Approach

The main disadvantages of the Apriori method, described in previous Section 3.1.1, are high number of generated candidates and number of database scans during the mining process. The solution was described by Han et al. in [21] where authors proposed the two-pass algorithm *FP-growth* (*Frequent-Pattern growth*). It does not generate candidates. The algorithm is based on the tree structure called the *FP-tree*. The algorithm runs in the following steps:

1. The first step is similar to the Apriori algorithm. The database is scanned and support of all items is counted.

2. In the second step, the FP-tree is constructed. The algorithm starts with one root node tree called *null*. During the second database scan, the transactions are transformed to the representation in the FP-tree. Items of each transaction are sorted in descendant support value order and they are added as nodes into the FP-tree. The first item of the transaction is added as a child of the root node *null*, the second item is added as a child of the node of the first item, etc.

The support value of each node is initialized to value 1. Because the items are in descendant support order, items with higher support are in upper levels of the tree. If there exists some prefix of items in the tree, the support of nodes is incremented by 1 and only nodes for new items are added into a new sub-tree of the last common item. Finally, the nodes for each item are linked in the list started in the Header table. The example of FP-tree is shown on Figure 3.1.

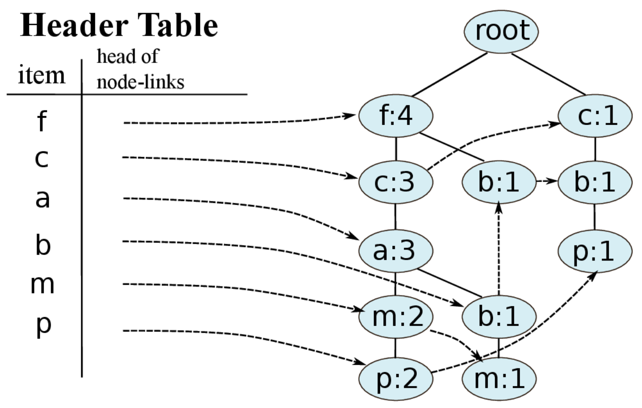

Figure 3.1: The example of a FP-tree data structure (taken from [21]).

- The last step is generating of frequent patterns from the FP-tree structure. The conditional FP-tree is constructed for each frequent item of tree. The conditional FP-tree by item base $x$, denoted as $x$-conditional FT-tree, is constructed similarly to basic FP-tree from the set of prefix paths of the item $x$ in the FP-tree. The process of the construction of conditional FP-tree is executed recursively on the $x$-conditional FP-tree until the conditional FP-tree contains only one path. The set of frequent patterns is generated by enumeration of all combinations of the subpaths of such conditional FP-trees.

### 3.1.3 Mining Maximal and Closed Frequent Itemsets

Maximal and closed frequent itemsets mining tasks reduce the number of generated frequent patterns without losing the interesting patterns. Efficient methods for mining such patterns are based on the early pruning of search space.

#### MaxMiner Algorithm

The *MaxMiner* algorithm, introduced in [8], uses the breadth-first search method. The method uses the Apriori property for pruning the infrequent itemsets and a new prune condition for non-maximal frequent itemsets. The algorithm produces larger itemsets gradually.

### GenMax Algorithm

The *GenMax* algorithm, presented in [14, 15], is based on the backtracking (depth-first search) method. In this case, larger itemsets are not generated gradually but the whole subtree of item node is searched for maximal frequent itemsets. The maximal itemsets condition remains the same.

The disadvantage of maximal frequent itemsets is that the information about support of subsets of maximal frequent itemsets is lost. The closed frequent itemsets, by contract, have the same power as mining frequent itemsets, however, the result set may be smaller [29].

### CLOSET Algorithm

Pei et al. presented an algorithm for mining frequent closed itemsets called CLOSET [30] which is based on the construction of the conditional FP-trees. The algorithm CLOSET divides the search space by the list $f\_list = (i_1, i_2, \ldots, i_n)$, which is a list of frequent items in the transaction database in descendant order of support. First, only $i_n$-conditional databases are mined. The $i$-conditional database is a subset of transactions containing an item $i$, and all occurrences of $i$ and items after $i$ in $f - list$ are omitted. Therefore, all frequent itemsets are divided into subsets based on $f - list$. Then, the process runs recursively on each conditional database and $X$-conditional databases are constructed (transactions containing all items in itemset $X$). Frequent itemsets $X$ are mined from $X$-conditional databases and tested if they are *closed*: if $X$ is closed frequent itemset, then there is no item appearing in all transactions of $X$-conditional database.

## 3.1.4 Mining Multi-level Frequent Patterns

There can be defined taxonomies over items. Moreover, some association rules may not satisfy the minimal support or the minimal confidence threshold on the lowest level of the hierarchy but it may satisfy them on higher levels.

The issue was discussed by Han et al. in [18]. They allowed that the minimal support and the minimal confidence values can differ for different levels of taxonomy. Authors proposed a basic method and several optimization variants for mining multi-level association rules. We can expect that children of all nodes of taxonomy are in lexicographic order without loss of generality. Methods are based on a encoded transaction tables $\mathcal{D}^{\mathcal{T}}[x]$ containing tuples in the form $\langle TID, Itemset\rangle$ which all items are encoded as follows. For item on level $l$, the code contains $l$ numbers. Each number is the ordering index of a child node in the taxonomy. For example, the code 124 code for an item $a$ means that the item $a$ is the fourth child on level 3 of the second node on level 2 of the first node of the top level. Then, the method finds large $k$-itemsets on each level $l$ of taxonomy denoted as $\mathcal{L}[k, l]$.

### Algorithm ML_T2L1

The first algorithm proposed in [18] is the ML_T2L1 variant. The algorithm is based on the top-down approach which finds frequent itemsets on the highest level and goes down in hierarchy. The pseudocode of the method is in the Algorithm 3.1. First, the transformed table $\mathcal{D}^{\mathcal{T}}[1]$ is created and 1-itemsets on the level 1 are counted. Using such itemsets, the filtered encoded table $\mathcal{D}^{\mathcal{T}}[2]$ contains only frequent items. Then, the algorithm runs iteratively in the $k$ iterations. The itemsets of size $k$ are created from the frequent itemsets of size $k-1$. This iterative procedure is run for all levels. Finally, all result frequent itemsets are returned.

**Algorithm 3.1** Method *ML_T2L1*($\mathcal{D}^{\mathcal{T}}[1]$, $min\_supp[l]$)

```
1:  for $(l = 1; \mathcal{L}[l,1]) \neq \emptyset \wedge l \leq max\_level; l++)$ do
2:      if $l = 1$ then
3:          $\mathcal{L}[l,1] = \text{get\_large\_1\_itemsets}(\mathcal{D}^{\mathcal{T}}[1], l)$
4:          $\mathcal{D}^{\mathcal{T}'} = \text{get\_filtered\_table}(\mathcal{D}^{\mathcal{T}}[1], \mathcal{L}[l,1])$
5:      else
6:          $\mathcal{L}[l,1] = \text{get\_large\_1\_itemsets}(\mathcal{D}^{\mathcal{T}}[2], l)$
7:      end if
8:      for $(k = 2; \mathcal{L}[l,k-1] \neq \emptyset; k++)$ do
9:          $C_k = \text{get\_candidates}(\mathcal{L}[l,k-1])$
10:         for all $t \in \mathcal{D}^{\mathcal{T}}[2]$ do
11:             Increment support for all itemsets in $C_k$ which are subsets of $t$
12:         end for
13:         $L[l,k] = \{c \in C_k | support(c) \geq min\_supp[l]\}$
14:     end for
15:     $\mathcal{LL}[l] = \bigcup_k \mathcal{L}[l,k]$
16: end for
17: return Multi-level frequent itemsets $\bigcup_l \mathcal{LL}[l]$
```

### Algorithms ML_T1LA, ML_TML1, ML_T2LA

Authors in [18] proposed several modifications of the algorithm ML_T2L1.

- The ML_T1LA avoids generating of second transformed table $\mathcal{D}^{\mathcal{T}}[2]$ and generates table $\mathcal{L}[l,1]$ for all levels in the one scan.
- The ML_TML1 uses transformed tables for each level $\mathcal{D}^{\mathcal{T}}[l]$.
- The ML_T2LA uses two transformed tables like the ML_T2L1 variant, but generates table $\mathcal{L}[l,1]$ for all levels in the one scan.

It was shown that performance of variants differs for different datasets and parameters.

#### Algorithm Cumulate

Srikant and Agrawal in [38] deal with the redundancy of multi-level association rules defined in Def. 19. They proposed an algorithm for mining multi-level association rules called *Cumulate*. Note that they don't distinguish between *multi-level* and *level-crossing* terms. They recommended using an extended database which contains all ancestors of items in the transactions. It allows counting support without traversing the taxonomy tree each time. In general, the algorithm is based on Apriori candidate generation method.

The algorithm is improved by several optimizations:

- use of a hash-tree for counting support,
- use of an extended transaction database,
- removal of all candidates which contain both item and its ancestor (redundant itemsets),
- removal of all ancestors that are not contained in any candidate.

## 3.2 Mining Sequential Patterns

Basic algorithms for mining frequent patterns were described in the previous section. Similar concepts were adapted to mining sequential patterns. Moreover, the algorithms for mining sequential patterns have to deal with an ordering of transactions of customers. This section contains an overview of approaches to the sequential patterns mining. The algorithms based on candidate generation are described first and, then, the efficiency improvements based on pattern-growth approach are introduced.

### 3.2.1 AprioriAll, AprioriSome and DynamicSome Algorithms

The first algorithms for mining (maximal) sequential patterns called *AprioriAll*, *AprioriSome* and *DynamicSome* were proposed by Agrawal et al. in paper [1]. The algorithms are based on the Apriori theorem known from frequent patterns mining. The algorithms consist of the following phases:

1. *Sort Phase* – the database is sorted by the transaction arrival time. In the first phase, the database of transactions $\mathcal{D}$ is sorted primary by the customer id and secondary by the transaction time. The customer sequences are implicitly created in this phase.

2. *Litemset Phase* – in this phase, the algorithm finds all *litemsets* (frequent itemsets, also called *large itemsets*). The set of initial 1-sequences are found. The authors recommended the special mapping of litemsets – each k-itemset is mapped to a single unique integer label. Such mapping allows comparing two litemsets for equality in the constant time and it reduces the time for checking if the sequence is contained in a customer sequence.

3. *Transformation Phase* – the mapping created in the second phase is used to transformation of sequences of the database $\mathcal{D}$. Each transaction is replaced by the all litemsets contained in the transaction. Therefore, the non-frequent items are removed from the transformed database. If the sequence does not contain any litemset, it is removed from the transformed database but it is counted to the total count of transactions. The transformed database is called $\mathcal{D}_T$.

4. *Sequence Phase* – in this phase, sequential patterns are generated and added into the result set.

5. Maximal Phase – this phase performs the post-processing operation which removes the non-maximal sequential patterns from the result set. In some algorithms, the phase is integrated directly in the Sequence Phase.

### AprioriAll Algorithm

*AprioriAll* is an iterative algorithm. Initially, the frequent 1-sequences are mined from the database. Then, in each iteration, the set $C_k$ of candidate $k$-sequences are created from the large $(k-1)$-sequences from the previous iteration. Generated candidate sequences $c \in C_k$, such that their all $(k-1)$-subsequences are not in the set $L_{(k-1)}$, are deleted (Apriori property check). Finally, the support values for all candidate $k$-sequences are counted in the step called counting step and the set $L_k$ of frequent $k$-sequences generated. The complete procedure is shown in Algorithm 3.2.

Candidate $k$-sequences are joined from sequences $s_1 \in L_{(k-1)}$ with $s_2 \in L_{(k-1)}$. Given a pair of sequences $s_1 = \langle s^1_1 s^1_2 s^1_3 \dots s^1_{(k-1)} \rangle$ and $s_2 = \langle s^2_1 s^2_2 s^2_3 \dots s^2_{(k-1)} \rangle$, if $s^1_1 = s^2_1 \wedge s^1_2 = s^2_2 \wedge s^1_3 = s^2_3 \wedge \dots \wedge s^1_{(k-2)} = s^2_{(k-2)}$, the sequences are joined into a new sequence $s = \langle s^1_1 s^1_2 s^1_3 \dots s^1_{(k-1)} s^2_{(k-1)} \rangle$ .

**Algorithm 3.2** Method *AprioriAll($\mathcal{D}$, min_supp)*

```
1: $L_1$={frequent 1-sequences}
2: for $(k = 2; L_{k-1} \neq \emptyset; k++)$ do
3:     $C_k$ = New candidates generated from $L_{k-1}$
4:     for $\forall s \in \mathcal{D}$ do
5:         Increment support of all $c \in C_k$ which are contained in $s$
6:     end for
7:     $L_k$= Candidates from $C_k$ which satisfy $min\_supp$
8: end for
9: return Maximal Sequences in $\bigcup_k L_k$
```

### AprioriSome Algorithm

The disadvantage of the AprioriAll is the post-processing phase for pruning of non-maximal sequential patterns. Therefore, two other algorithms were proposed in the research paper [1]. *AprioriSome* algorithm includes two phases – *forward* one and *backward* one.

- The time-consuming *counting step* (counting support of candidates) is omitted in specified iterations of *forward phase*. Authors proposed an evaluation technique that decides which forward counting steps should be omitted. They defined a ratio value $hit_k = |L_k|/|C_k|$. Using the value $hit_k$ is determined how many next forward counting steps will be skipped and $L_k$ will not be generated. However, the AprioriSome algorithm uses the same candidate generation procedure as the AprioriAll algorithm. The $L_{k-1}$ is used for generation of $C_k$ in the AprioriSome algorithm if previous counting step was not skipped according to $hit_k$, otherwise the $C_{k-1}$ must be used.

- Then, in the *backward phase*, all $L_k$ sequential patterns skipped in forward phase are evaluated. Also, the non-maximal sequences can be simply identified and removed because all the longer large sequences are known.

#### DynamicSome Algorithm

The third modification of the algorithm is called *DynamicSome*. Generated $C_k$ candidate sequences are determined by the parameter *step*. The algorithm is composed of four steps for a given *step* value:

- *Initialization phase* – all the candidate sequences of length up to value *step* are counted. For example, for the *step* = 4, the candidate sequences of lengths 1, 2, 3 and 4 are counted.

- *Forward phase* – The candidate sequences, whose lengths are multiples of step, are counted. For the *step* = 4, the sequences of lengths $k = 8$, 12, 16, etc. are counted until any sequence of the length $k$ exists.

- *Intermediate phase* – unlike the AprioriSome algorithm, several lengths of candidate sequences are skipped in the forward phase. Candidates of such lengths are counted in the intermediate phase from $k = k_{max} - 1$ decreasingly, where $k_{max}$ is the maximal length of candidate sequences in the forward phase.

- *Backward phase* – the same phase to the AprioriSome algorithm. The non-maximal sequences are removed from the result set.

The results in the paper [1] show that the AprioriAll and the AprioriSome have a similar performance result. The DynamicSome algorithm gives the worst performance results.

### 3.2.2 GSP Algorithm

The algorithms AprioriAll and AprioriSome described in the previous section allow mining of non-constrained sequential patterns and maximal sequential patterns using post-processing procedure. Srikant introduced a new mining algorithm called *Generalized Sequential Patterns (GSP)* in [37]. The GSP allows different types of constraints of sequential patterns such as:

- Taxonomies – can be defined over items in a sequence database. In this case, an itemset $s_i$ contains an item $x \in I$ if $x$ is in $s_i$ or $x$ is an ancestor of some item in $s_i$. Then, the itemset $s_i$ contains an itemset $X \subseteq I$ if $e_i$ contains all items in $X$. The rest of the *contains* relation remains the same. The rest of the thesis focuses on this problem.

- Sliding windows – allow only such sequences, in which a maximum distance between the first and the last item of sequence in the sequence database is less than the window size. A sequence $d = \langle d_1, \ldots, d_m \rangle$ in database contains a sequence $s = \langle s_1, \ldots, s_n \rangle$ if there exist integers $l_i \leq u_1 \leq l_2 \leq u_2 < \cdots < l_n \leq u_n$ such that $s_i$ is contained in $\bigcup_{k=l_i}^{u_i} d_k, 1 \leq i \leq n$ and

  - $transaction\_time(d_{u_i}) - transaction\_time(d_{l_i}) \leq window\_size$ where $1 \leq i \leq n$.

- Time constraints – extend the sliding windows adding the *min_gap* and *max_gap* conditions. Such constraints allow only patterns with minimally and maximally limited gap between itemsets. First part of definition is the same for sliding windows. Further, it defines the following two restrictions:

  - $transaction\_time(d_{l_i}) - transaction\_time(d_{u_{i-1}}) > min\_gap$ where $2 \leq i \leq n$ and
  - $transaction\_time(d_{u_i}) - transaction\_time(d_{l_{i-1}}) \leq max\_gap$ where $2 \leq i \leq n$.

The algorithm is important for this work and, therefore, it will be described in detail. The main steps of the GSP algorithm are:

- *Candidate Generation* – new candidate sequences are generated.

- *Counting Candidates* -- the support values of new candidates are counted.

The algorithm works iteratively. It makes a pass over the sequence database in all iterations:

1. Initially, the support of items is counted in the first database pass. 1-sequences are created from items with higher support value than a minimal support *min_sup.* Such 1-sequences are inserted into a partial result set $L_1$ containing all *frequent 1-sequences.*

2. Then the following steps are processed iteratively until none $k$-sequential pattern is generated:

   (a) The *Candidate Generation* step generates $C_k$ candidate sequences.

   (b) The *Counting Candidates* step filters the frequent sequences into the $L_k$ sets.

3. The result set of sequential patterns is $\bigcup_k L_k$.

The *Candidate Generation* runs in Join and Prune steps.

1. In the *Join step*, a set of *candidate sequences* $C_k$ is generated from sequential patterns in $L_{k-1}$. A pair of sequences $s_1, s_2 \in L_{k-1}$ can be joined if subsequences, generated by omitting of the first item of $s_1$ and the last item of $s_2$, are equal. Then, the candidate $k$-sequence is formed by adding the last item of the $s_2$ at the end of the sequence $s_1$ as:

   (a) the last new element containing one item $x$ if $x$ was in a separate element in $s_2$;

   (b) as a next item of the last element in $s_1$ otherwise.

   (c) When joining $x \in L_1$ with $y \in L_1$, both sequences $<(y)(x)>$ and $<(yx)>$ are generated as candidate sequences.

2. The *Prune step* removes candidates whose any $(k-1)$-subsequence is not frequent.

In the *Counting Candidates step*, the database is passed and the support of each candidate sequence is counted. Candidates with a support greater than $min\_supp$ are added into the set $L_k$ of sequential patterns. The *contains test*, checking if a sequence $s$ of the sequence database contains a candidate sequence $s_c$, is used for support evaluation. Because of the time constraints, the *contains test* consists of two phases. Given an input data sequence $d$ and a candidate sequence $s$, the procedure works as follows:

1. *Forward phase* – the algorithm finds one-after-one elements of $s$ in the $d$ until the gap between the start position of the next element of $s$ and the position of the previous item of $s$ is less than the $min\_gap$ constraint. If the gap is greater than the $max\_gap$, the algorithm switches into the backward phase. If the next item of $s$ is not found in the searched part of $d$, the algorithm returns with result that the $s$ is not contained in the $d$ with specified $max\_gap$ constraint.

2. *Backward phase* – the algorithm backtracks previous elements and *pulls-up* them. The backward phase for $s_i$ tries to find the first occurrence of $s_{i-1}$ such that $end\_time(s_i) - start\_time(s_{i-1}) \leq max\_gap$. Then, the algorithm moves back and pulls-up the $s_{i-2}$ because $s_{i-1}$ may not satisfy the $max\_gap$ constraint. The backward phase is switched back to forward phase after $x$-steps when the $max\_gap$ constraint could not be satisfied for $s_{i-x}$. If any element could not be pulled-up, the algorithm returns result that $s$ is not contained in $d$ with specified $max\_gap$ constraint.

The pseudo code of the basic steps of GSP is shown on Algorithm 3.3. Authors of the GSP also presented the proof of completeness of the algorithm in [37]. The proof idea is that the join step for construction of $C_k$ sequences from $L_{k-1}$ is equivalent

**Algorithm 3.3** The GSP algorithm

```
1: procedure GSP($\mathcal{D}, min\_supp$)
2:     $L_k$ =frequent 1-sequential patterns
3:     while $L_k = \emptyset$ do
4:         $C_{k+1} = CandidateGeneration(L_k)$
5:         for $\forall s \in \mathcal{D}$ do
6:             Increment support of all candidates that are contained in $s$
7:         end for
8:         $L_{k+1} = \{s \in C_{k+1} | support(s) \geq min\_supp\}$
9:     end while
10:    return $\bigcup_k L_k$
11: end procedure
```

Table 3.1: The sequential patterns with the $min\_supp = 2$ mined from the sequence database of root items in Table 2.2.

| # | **1-seq. patterns** | **2-seq. patterns** | **3-seq. patterns** | **4-seq. patterns** |
|---|---|---|---|---|
| 1. | $\langle d \rangle : 2$ | $\langle da \rangle : 2$ | $\langle dba \rangle : 2$ | $\langle d\ (bf)\ a \rangle : 2$ |
| 2. | $\langle a \rangle : 4$ | $\langle db \rangle : 2$ | $\langle d\ (bf) \rangle : 2$ | $\langle a\ (bf)\ a \rangle : 2$ |
| 3. | $\langle b \rangle : 4$ | $\langle df \rangle : 2$ | $\langle dfa \rangle : 2$ | |
| 4. | $\langle f \rangle : 4$ | $\langle ad \rangle : 2$ | $\langle (abf) \rangle : 3$ | |
| 5. | | $\langle aa \rangle : 2$ | $\langle aba \rangle : 2$ | |
| 6. | | $\langle (ab) \rangle : 3$ | $\langle a\ (bf) \rangle : 2$ | |
| 7. | | $\langle ab \rangle : 2$ | $\langle afa \rangle : 2$ | |
| 8. | | $\langle (af) \rangle : 3$ | $\langle (bf)\ a \rangle : 2$ | |
| 9. | | $\langle af \rangle : 2$ | | |
| 10. | | $\langle ba \rangle : 2$ | | |
| 11. | | $\langle (bf) \rangle : 4$ | | |
| 12. | | $\langle fa \rangle : 2$ | | |

to extending all $L_{k-1}$ with each frequent item followed by deleting those sequences which $(k-1)$-subsequences with deleted the first item are not in $L_{k-1}$.

Sliding windows and gap time constraints are not considered for the rest of the thesis.

**Example 5.** This example explains the GSP procedure using the running example data from the Example 2. Assume the sequence database $\mathcal{D}_{root}$ in Table 2.2. The procedure works as follows.

1. The GSP algorithm reads the sequence database and counts the support for all items in the database. The support values for items are:$a : 4$, $b : 4$, $c : 1$, $d : 2$, $e : 1$, $f : 4$, $g : 1$, $h : 1$. Then, the 1-sequential patterns are created from the frequent items $\{a, b, d, f\}$ only: $\langle a \rangle : 4$, $\langle b \rangle : 4$, $\langle d \rangle : 2$ and $\langle f \rangle : 4$.

2. In the next phase, the candidate 2-sequences are joined from the 1-sequential patterns. Three candidate sequences are generated using each pair of 1-sequential patterns, for example from pair $\langle b \rangle$ and $\langle f \rangle$ join procedure creates sequences $\langle bf \rangle$, $\langle fb \rangle$ and $\langle (bf) \rangle$ (note that $\langle (fb) \rangle$ is equal to $\langle (bf) \rangle$). Similarly, all candidate 2-sequences are created. Finally, the counting step is performed using next database pass: $\langle bf \rangle : 1$, $\langle fb \rangle : 0$ and $\langle (bf) \rangle : 4$. Assuming the minimal support equal to 2, only the $\langle (bf) \rangle : 4$ from this pair is a frequent pattern. Remaining sequences are removed. The complete set of 2-sequential patterns is shown in Table 3.1.

3. Then in the next phase, the algorithm continues with join step of 2-sequential patterns into candidate 3-sequences. The algorithm must verify if the join of the pair of sequential patterns is possible. For example: the pair $\langle (bf) \rangle : 4$ and $\langle (ab) \rangle : 3$ can be joined into the sequence $\langle (abf) \rangle$ because there exists the common 1-subsequences$\langle (_b) \rangle$ and $\langle (b\_) \rangle$. In this case, the items are joined into one element. The pair $\langle (bf) \rangle : 4$ and $\langle ab \rangle : 2$ can be also joined into another sequence $\langle a(bf) \rangle$. Note that the common 1-subsequences in this case are$\langle _b \rangle$ and $\langle (b\_) \rangle$, therefore the sequence with 2 elements is created. Finally, the pair $\langle (bf) \rangle : 4$ and $\langle aa \rangle : 2$ cannot be joined because there is no common subsequence. As in the previous iteration, the counting step, making a next database pass and removing of non-frequent sequences, is finally performed. In the example above, 3-sequential pattern $\langle (abf) \rangle$ has support 3.

4. The algorithm continues generating 4-sequential patterns. In our example, the pair $\langle d\ (bf) \rangle : 2$ and $\langle (bf)\ a \rangle : 2$ is used to create the candidate sequence $\langle d\ (bf)\ a \rangle : 2$ and the pair $\langle a\ (bf) \rangle : 2$ and $\langle (bf)\ a \rangle : 2$ to create the candidate sequence $\langle a\ (bf)\ a \rangle : 2$.

5. The pair of 4-sequential patterns could not be joined and no candidate sequence is created. Therefore, the GSP algorithm finishes. The complete result of mining the database $\mathcal{D}_{root}$ is shown in Table 3.1.

#### 3.2.2.1 Variants of Sequential Pattern Mining

By Shrikant [37] there exist several extensions of basic sequential pattern mining problem. In general, extensions are based on redefinition of *contains* subsequence function.

### 3.2.3 SPAM Algorithm

Algorithms described in previous sections were based on the *breadth first search* strategy [6]. It means that in each iteration the algorithms generate sequences longer by one item than in the previous one. In contrast, the SPAM algorithm presented in [7] is the representative of the *depth first search* [6] strategy. In general, this strategy tries to extend the sequence being processed immediately.

First, the lexicographical ordering $\leq$ over items is defined in the database $\mathcal{D}$. For given items $i, j \in I$, if the item $i$ lexicographically precedes item $j$, then we denote

$i \leq j$. The same ordering is applied to sequences. Given sequences $s_1$ and $s_2$, we denote $s_{1\leq}s_2$ if $s_1$ is a subsequence of $s_2$. Then, the sequence tree, denoted as $T$, is arranged from nodes as follows. Nodes of level $l$ represent only the sequences of the length $l$. The root of the sequence tree is a node $n = \langle\rangle$. Then, recursively, if the node $n$ is the node in the tree, the children of $n$ are all nodes $n'$ such that $n \leq n'$ and $\forall m \in T : n' \leq m \Rightarrow n \leq m$. The sequences are generated by sequence extension steps (*S-step*; a new item is appended to the sequence as a new element) and itemset extension steps (*I-step*; a new item is append to the sequence into the last element). Each node $n$ in the tree is associated with two sets: $S_n$ is the set of candidate items with possible $S$-step extension and $I_n$ is the set of candidate items for possible $I$-step extension. The tree is traversed by the depth first search traversal approach – for each node all its children are processed before going to next sibling.

Because of the large search space, two optimization pruning rules are applied during traversing the tree. The algorithm pseudocode is in Algorithm 3.4.

## S-Step Pruning

Suppose sequence $s$ of the node $n$, $i_j, i_k \in I$. Its *S-Step* extensions are $s_a = \langle s, i_j\rangle$, which is frequent, and $s_b = \langle s, i_k\rangle$, which is not frequent. According to the Apriori theorem, both extensions $\langle s, i_j, i_k\rangle$ and $\langle s, (i_j, i_k)\rangle$ are not frequent and, therefore, $i_k$ can be removed from both $S_m$ and $I_m$ sets of items, where $m$ is any child node of $n$ corresponding to frequent sequence.

## I-Step Pruning

Suppose sequence $s = \langle s', (i_1, \ldots, i_n)\rangle$ of the node $n$, $i_j, i_k \in I$ and its *I-Step* extensions $s_a = \langle s', (i_1, \ldots, i_n, i_j)\rangle$, which is frequent, and $s_b = \langle s', (i_1, \ldots, i_n, i_k)\rangle$, which is not frequent. According to the Apriori theorem, $\langle s', (i_1, \ldots, i_n, i_j, i_k)\rangle$ is not frequent and $i_k$ can be removed from $I_m$. Notice that the $S_m$ set can be pruned by the same items as in the S-Step.

## Vertical database format

A vertical bitmap database format enables an effective *S-Step* and *I-Step* and effective support counting. The *vertical bitmap table* is in format $\mathcal{B}_i = (SID, TID, B)$, where $SID$ is sequence identifier, $TID$ is a unique transaction identifier and $B$ is a list of bits $b_i$ , where the bit for item $i$ is set to one, if the transaction $TID$ of sequence $SID$ contains $i$, or zero otherwise. Then, the bitmap can be also evaluated for a sequence, for example for 2-sequence $s = \langle i, j\rangle$, the bitmap is evaluated as a logical *AND operator* applied over bitmaps for item $i$ and item $j$. Bitmap tables are divided into groups by lengths. The group contains sequences from the $2^k + 1$ length up to $2^{k+1}$ for $k = 1, \ldots, 5$. Therefore, the maximal sequence length is 64, longer sequences are ignored. Authors say that longer sequences occur minimally. Then, the support counting only checks, if any bit of the corresponding bitmap partition for a sequence is set to one.

**Algorithm 3.4** The SPAM algorithm

```
 1: procedure SPAM(node $n = \langle s_1, \ldots, s_k \rangle, S_n, I_n$)
 2:     $S_{temp} = \emptyset$
 3:     $I_{temp} = \emptyset$
 4:     for $\forall i \in S_n$ do
 5:         if $\langle s_1, \ldots, s_k, \{i\} \rangle$ is frequent then
 6:             $S_{temp} = S_{temp} \cup \{i\}$
 7:         end if
 8:     end for
 9:     for $\forall i \in S_{temp}$ do
10:         $SPAM(\langle s_1, \ldots, s_k, \{i\} \rangle, S_{temp}, \{j \in S_{temp} | i < j\}$
11:     end for
12:     for $\forall i \in I_n$ do
13:         if $\langle s_1, \ldots, s_k \cup \{i\} \rangle$ is frequent then
14:             $I_{temp} = I_{temp} \cup \{i\}$
15:         end if
16:     end for
17:     for $\forall i \in I_{temp}$ do
18:         $SPAM(\langle s_1, \ldots, s_k \cup \{i\} \rangle, S_{temp}, \{j \in I_{temp} | i < j\}$
19:     end for
20: end procedure
```

The main advantage is in the way how the S-Step and I-Step are performed. The I-Step simply performs logical AND over a sequence bitmap and a bitmap of appended item. The S-Step needs transformation step.

### 3.2.4 PrefixSpan Algorithm

The *PrefixSpan* proposed by Pet et al. [32],[31] is a representative of the *pattern-growth algorithms.* The algorithm does not use the time-consuming generating of candidate sequences. The algorithm is based on the projected databases [20]. Without loss on generality, the algorithm assumes that the items of elements are sorted in lexicographical order.

For the purpose of the algorithm, the prefix and suffix terms were introduced. Given sequences $\alpha = \langle s_1 s_2 \ldots s_n \rangle$ and $\beta = \langle s'_1 s'_2 \ldots s'_m \rangle$ where $(m < n)$. Sequence $\beta$ is a *prefix* of sequence $\alpha$ iff $s'_i = s_i$ for $(i \leq m-1)$ and $e'_m \subseteq e_m$ and all items in $(e_m - e'_m)$ are lexicographically after all items in $e'_m$.

Given a sequence $\alpha = \langle s_1 s_2 \ldots s_n \rangle$ and $\beta = \langle s_1 s_2 \ldots e_{m-1} s'_m \rangle$ $(m < n)$ be a prefix of $\alpha$. A sequence $\gamma = \langle e''_m e_{m+1} \ldots e_n \rangle$ is a *suffix* of $\alpha$ w.r.t. a prefix $\beta$. The suffix is denoted as $\gamma = \alpha \backslash \beta$ and $\alpha = \beta.\gamma$. The *projected database* for a prefix sequence $\alpha$, denoted as $S_{|\alpha}$, is defined as a set of suffixes of sequences in $S$ with regards to prefix$\alpha$. The *length of prefix* $\alpha$ is denoted as $l$.

The PrefixSpan algorithm works as follows. In the first scan, the algorithm finds all 1-sequential patterns in the sequence database (the prefix is empty). Then, projected database construction and PrefixSpan procedure are applied for each 1-sequential patterns. Constructed projected databases are searched for a set of local frequent items again. Output sequential patterns are constructed by joining a prefix with all local frequent items. Finally, sequential patterns represent new prefixes and the PrefixSpan is run recursively. The pseudo code of the PrefixSpan is in the Algorithm 3.5.

**Algorithm 3.5** The PrefixSpan algorithm

```
1: procedure PrefixSpan($\alpha, l, S_{|\alpha}$)
2:     Find all frequent items $X$ in the projected database $S_{|\alpha}$
3:     for each item $x \in X$ do
4:         Construct $(l+1)$-sequential pattern $\alpha'$ appending $x$ to prefix $\alpha$
5:         Output sequential pattern $\alpha'$
6:         Construct projected database for $\alpha'$ and run PrefixSpan($\alpha', l+1, S_{|\alpha'}$)
7:     end for
8: end procedure
```

The construction of projected databases is inefficient. Therefore, optimization called pseudo-projection was proposed. The projected databases are not constructed by copying subsequences but they are constructed by pointing to the sequences in $\mathcal{D}$. The only condition is that the sequence database $\mathcal{D}$ has to be loaded in the main memory.

**Example 6.** The PrefixSpan example shows the pattern-growth approach. Given the sequence database in Table 2.2 and the $min\_supp = 2$ the algorithm works in following steps.

1. First, the algorithm finds the frequent items during the first database scan: $a, b, d, f$.

2. Projected databases are created using the frequent items from the first step during the second full database scan. Projected databases with length 1 prefixes are shown in Table 3.2. Note, that an element of form $\langle(\_x)\dots\rangle$ denotes that the last element of the prefix is part of this element.

3. Each projected database in Table 3.2 is scanned for 2-sequential patterns and the procedure is run recursively for such sequential patterns, e.g. the $\langle b\rangle$ projected database contains sequential patterns $\langle ba\rangle : 2$ and $\langle(bf)\rangle : 4$, projected databases $\langle ba\rangle$ and $\langle(bf)\rangle$ are constructed and PrefixSpan is called recursively for those projected databases.

### 3.2.5 Other Sequential Pattern Mining Methods

Several other algorithms and their variants for mining sequential patterns were discovered. Some important principles are described in this section.

Table 3.2: Projected databases and final sequential patterns.

| **Prefix** | **Projected database** | **Sequential patterns (mined recursively)** |
|---|---|---|
| $\langle a\rangle : 4$ | $\langle(\_bc)(abf)(acdf)\rangle$, $\langle(\_bf)\ e\rangle$, $\langle(\_bf)\rangle$, $\langle(dgh)(bf)(agh)\rangle$ | $\langle a\rangle : 4$, $\langle ad\rangle : 2$, $\langle aa\rangle : 2$, $\langle(ab)\rangle : 3$, $\langle ab\rangle : 2$, $\langle(af)\rangle : 3$, $\langle af\rangle : 2$, $\langle(abf)\rangle : 3$, $\langle aba\rangle : 2$, $\langle a\ (bf)\rangle : 2$, $\langle afa\rangle : 2$, $\langle a\ (bf)\ a\rangle : 2$ |
| $\langle b\rangle : 4$ | $\langle(\_c)(abf)(acdf)\rangle$, $\langle(\_f)\ e\rangle\langle(\_f)\rangle$, $\langle(\_f)(agh)\rangle$ | $\langle b\rangle : 4$, $\langle ba\rangle : 2$, $\langle(bf)\rangle : 4$, $\langle(bf)\ a\rangle : 2$ |
| $\langle d\rangle : 2$ | $\langle(\_f)\rangle$, $\langle(\_gh)(bf)(agh)\rangle$ | $\langle d\rangle : 2$, $\langle da\rangle : 2$, $\langle db\rangle : 2$, $\langle df\rangle : 2$, $\langle dba\rangle : 2$, $\langle d\ (bf)\rangle : 2$, $\langle dfa\rangle : 2$, $\langle d\ (bf)\ a\rangle : 2$ |
| $\langle f\rangle : 4$ | $\langle(acdf)\rangle$, $\langle e\rangle$, $\langle(agh)\rangle$ | $\langle f\rangle : 4$,$\langle fa\rangle : 2$ |

### SPADE

The vertical database format was firstly used for mining sequential patterns in the SPAM algorithm described in Section 3.2.3. The improved vertical database algorithm is the SPADE algorithm presented by Zaki et al. in [50]. Sequences are divided into equivalent classes by atoms, which are initially all frequent items in the sequence database. Used principle is in general similar to the SPAM algorithm; however, it uses different internal representation. The SPADE does not use the bitmap representation but it stores the pairs of $(SID, EID)$ for each sequence, where the $SID$ is a sequence identifier and $EID$ is a timestamp of the event (transaction ID).

The algorithm uses two ways how to extend sequences using vertical databases of sequences:

- *temporal join* – a new item is appended as a new element to a base sequence, the temporal join can be performed if the item to be appended has its $EID$ greater than the base sequence has.

- *equality join* – a new item is appended as a part of the last element of a base sequence, the equality join can be performed if the item to be appended has its $EID$ equal to the base sequence has.

### LAPIN-SPAM

The *LAPIN-SPAM* algorithm, presented in [47], is an algorithm also based on the vertical database format. It improves the SPAM [7] algorithm. The $S$-step and $I$-step and a bitmap representation is adopted from the SPAM algorithm. It improves the counting step using a temporal table for each sequence of sequence database $\mathcal{D}$ which keeps information about the last occurrence of items in sequences of $\mathcal{D}$. It enables to skip several AND operations when the last occurrence of the candidate item is behind the current prefix position.

## 3.2.6 Mining Closed Sequential Patterns

The problem of mining closed sequential patterns is defined in Def. 12. In following subsections, major algorithms for mining closed sequential patterns are described.

The advantage of closed sequential patterns is that the result set of sequential patterns is significantly smaller. In general, algorithms use techniques of early pruning to avoid exploring of non-closed sequential patterns.

### BIDE

The first representative for mining closed sequential patterns is the *BIDE (Bi-Directional Extension)* algorithm presented in [45]. The algorithm is based on the principle of projected databases used in PrefixSpan. It uses several optimization techniques for early pruning and fast check if the sequence can be closed. Authors of algorithm formulated the following lemma: If there exists neither forward-extension event nor backward-extension event w.r.t. to a prefix sequence $\alpha$, then $\alpha$ is a closed sequence, otherwise, $\alpha$ must be non-closed. The forward-extension and backward-extension events check the following conditions:

1. *Forward-extension event checking* – The sequence database contains an extension of prefix sequence $\alpha = \langle e_1 \dots e_n \rangle$ extended by item $i$ in following form $\alpha' = \langle e_1 \dots e_n i \rangle$, such that $support(\alpha) = support(\alpha')$. The check of the condition can be done by counting local frequent items in the projected database with prefix $\alpha$.

2. *Backward-extension event checking* – The sequence database contains an extension of prefix sequence $\alpha = \langle e_1 \dots e_n \rangle$ extended by item $i$ in following two forms $\alpha' = \langle e_1 \dots e_k i \, e_{k+1} \dots e_n \rangle$ or $\alpha' = \langle i \, e_1 \dots e_n \rangle$, such that $support(\alpha) = support(\alpha')$. The check finds all items which are included in all prefixes of sequence database $\mathcal{D}$ which supports a current prefix $\alpha$. Formally, backward-extension events are items which occur in all a *maximum periods of a prefix sequence* in $D$. Details are described in [45].

Finally, the procedure for early pruning method called *BackScan* is defined. It analyses if there is a chance to get any closed frequent sequences with prefix $\alpha$. The authors presented that the algorithm outperforms PrefixSpan and SPADE algorithms.

### CloSpan

The algorithm *CloSpan* is next representative of family of pattern growth algorithms. It uses projected databases similarly to the *PrefixSpan* and a *lexicographic sequence tree* [46]. The algorithm generates a superset of all closed sequential patterns in the first step and the non-closed sequential patterns are eliminated in the second step. The optimization of the algorithm is based on the usage of early termination by equivalence of the projection databases: given sequences $s$ and $s'$ and its projected databases such that $s \sqsubseteq s'$ and $size(S_{|s}) = size(S_{|s'})$, then $\forall i \in \mathcal{I} : support(\langle s \, i \rangle) = support(\langle s' \, i \rangle)$. Therefore, many countings over projected database can be skipped.

Such method generates also non-closed sequential patterns. Therefore, authors added the post-processing step for removing such patterns inspired by a method developed by Zaki in [49] for mining closed itemsets. The method uses the hash table for fast access indexed by support of sequences.

**ClaSP**

One of recent algorithms for mining closed sequential patterns is the algorithm *ClaSP* presented by Gomariz et al. in [13]. The algorithm combines some previously presented principles – the vertical format of the database with the depth first search method. The algorithm adopts the pruning method from the CloSpan algorithm and saves the sequential patterns into a lexicographic sequence tree. Finally, a hash table is used for fast determination of possible redundant (non-closed) sequences.

### 3.2.7 Sequential Generator Patterns

The sequential pattern generator is in the opposite to closed sequential patterns. Whereas the mining closed sequential patterns finds the longest sequential patterns, the mining sequential generator patterns finds the shortest ones w.r.t. equal support. The sequential generator is defined by the Definition 20.

**Definition 20. (Sequential Pattern Generator)** A sequential pattern $s_a$ is said to be a generator if there is no other sequential pattern $s_b$, such that $s_b \sqsubseteq s_a$, and their supports are equal.

There exists several algorithms to solve this problem such as GenMiner [24], FEAT [12], FSGP [48]. However, the problem is very time consuming because of large search space. It was proposed a depth first search method *VGEN* based on a vertical format database for mining sequential pattern generators in our paper [11]. The experiments show that the algorithm VGEN works from one to two orders of magnitude faster than the algorithms mentioned above.

## 3.3 Mining Multi-Level Sequential Patterns

Mining multi-level and level-crossing sequential patterns is a big challenge. Similarly to the multi-level frequent itemsets described in Section 3.1.4, the multi-level sequential patterns mining algorithms try to find items on higher taxonomy levels. The task is very difficult according to the large search space of the possible results. The straightforward solutions bring very unclear results with many redundant patterns. In this subsection, current approaches are described, however, none of them provides sufficient solution. This section is closely related to the core of the thesis.

**Example 7.** The example explains the benefit of multi-level sequential patterns in contrast to sequential patterns. Remind the sequence database in Table 2.1 on page 9. Given the $min\_supp = 2$ and the sequence database in Table 2.1. The sequential patterns, found by the GSP method, are shown in Table 3.3. Only three 2-sequential patterns are found. Now, the sequence database containing only root items in Table 2.2 is mined. The resulting sequential patterns are presented in Table 3.1 on page 25. In that case, 22 sequential patterns of length 2, 3 and 4 are found. Sequential patterns of the root items are multi-level sequential patterns on the highest level. Here it was shown, that new sequential patterns can be found on higher taxonomy levels. The rest of the thesis deals with the methods how to get effectively different types of multi-level sequential patterns.

Table 3.3: The sequential patterns with the $min\_supp = 2$ mined from the sequence database in Table 2.1.

| # | **1-seq. patterns** | **2-seq. patterns** |
|---|---|---|
| 1. | $\langle d_1 \rangle : 2$ | $\langle d_1 b_1 \rangle : 2$ |
| 2. | $\langle a_{12} \rangle : 2$ | $\langle d_1 f_1 \rangle : 2$ |
| 3. | $\langle b_1 \rangle : 2$ | $\langle (b_2 f_2) \rangle : 2$ |
| 4. | $\langle b_2 \rangle : 3$ | |
| 5. | $\langle f_1 \rangle : 2$ | |
| 6. | $\langle a_{11} \rangle : 2$ | |
| 7. | $\langle f_2 \rangle : 2$ | |
| 8. | $\langle a_2 \rangle : 2$ | |

### 3.3.1 Approach Based on Extended Sequences

The first method to deal with taxonomies over sequential patterns was described in [34]. The method is called *Uniform sequential approach* [34]. It allows using a common sequential patterns mining algorithm for mining multi-dimensional and multi-level sequential patterns. This intuitive approach is based on the extending of sequence database $\mathcal{D}$. Each sequence $s \in \mathcal{D}$ is replaced by a new sequence $s'$ called *extended sequence* where each item of $s$ is replaced by all its ancestors within an element. Then, the presented AprioriSome and GSP algorithms find level-crossing sequential patterns. There were proposed 2 optimizations in the [37].

1. Pre-compute the ancestors of each item and drop ancestors which are not in any of the candidates being counted before.

2. Do not count the sequential patterns which contain both an item and it's any ancestor within one element. The support of such sequence is the same as one of the sequence containing the item.

Han et al. used in [16] such *extended sequences* for mining multi-level sequential patterns by means of the PrefixSpan algorithm. The modification of the sequence database was used in the principle adopted from the mining multi-level frequent patterns algorithms proposed in [22]. Authors encoded items by $n$-tuples representing their order in different levels of taxonomies. Moreover, they assumed fuzzy membership of items in taxonomies.

**Example 8.** Given a sequence database in Table 2.1 on page 9. The extended sequence database is shown in Table 3.4. The modification is straightforward – all items in the sequence database are expanded by all their ancestors (w.r.t. taxonomies on Figure 2.2). E.g. if the element contains item $a_{12}$, then items $a_1$ and $a$ are added into element. Therefore, multi-level sequential patterns can be mined using the casual data mining algorithms for mining sequential patterns with extended sequences. However, many redundant sequential patterns are mined. For example, all generalized patterns $\langle db_1 \rangle$, $\langle d_1 b \rangle$ and $\langle db \rangle$ are found together with sequence$\langle d_1 b_1 \rangle$.

Table 3.4: An extended sequence database to sequence database in Table 2.1.

| **Sequence_ID** | **Sequence** |
|---|---|
| 1 | $\langle (c_1\, c\, d_1\, d)(a_{12}\, a_1\, a\, b_1\, b\, c_1\, c)(a_1\, a\, b_2\, b\, f_1\, f)(a_{11}\, a_1\, a\, c_1\, c\, d_1\, d\, f_1\, f) \rangle$ |
| 2 | $\langle (a_{12}\, a_1\, a\, b_2\, b\, f_2\, f)\ (e_1\, e) \rangle$ |
| 3 | $\langle (a_2\, a\, b_2\, b\, f_2\, f) \rangle$ |
| 4 | $\langle (a_{11}\, a_1\, a)(d_1\, d\, g_1\, g\, h_1\, h)(b_1\, b\, f_1\, f)(a_2\, a\, g_2\, g\, h_2\, h) \rangle$ |

### 3.3.2 Generalization Based Methods

The topic of mining multi-level sequential patterns was deeply studied by Plantevit et al. [36] and [35]. They proposed several methods for mining different kinds of multidimensional and multi-level sequential patterns. The multidimensional database contains items from $n$ distinct dimensions $D_i$. Then items for data mining tasks are $n$-tuples $i = (d_1, \dots, d_n)$, where $d_i \in dom(D_i) \cup \{*\}$ and star symbol * denotes all items of domain $D_i$, called multidimensional items.

They proposed an algorithm called HYPE (HierarchY Pattern Extension). They described an idea of generalization and specialization of sequential patterns [36]. The algorithm runs in two phases. In the first phase, the algorithm creates the most *specific* (items in leaf nodes of taxonomies) $n-$multidimensional items $i = (d_1, \dots, d_n)$ denotes any item. The construction gradually replaces star symbols by specific items of domains by joining pairs of compatible hierarchical items – two items over $n-$dimensions are compatible, if items share $n-2$ items. In the second phase, the sequential patterns are iteratively mined using an Apriori theorem.

The disadvantage of the HYPE is that only two levels of hierarchy are assumed – the most specific one (items) and most general (star symbols) level. The algorithm was improved in [35] where more levels of taxonomies were used. Authors defined a specificity relation $\preceq_I$ over items and $\preceq_S$ over sequences where $s_1 \preceq s_2$ denotes that the $s_2$ is more specific or equal to $s_1$. The algorithm firstly creates the most specific MAF *(maximal atomic frequent)* sequences which are the most specific frequent 1-sequences. For mining longer sequences, authors recommended to use any existing algorithm such as SPADE. However, the algorithm does not perform generalization operations after an initialization of MAF-sequences and it does not reveal all most specific sequences with size greater than one.

# Chapter 4

# Level-Crossing and Multi-level Sequential Pattern Mining

Basic concepts of mining sequential patterns were described in previous Chapters 2 and 3. There were introduced the field of mining level-crossing and multi-level sequential patterns. It was indicated that the usage of taxonomies of items can help to find new results and new knowledge. The analysis of the state-of-the-art exposed that such mining task is important and challenging, however, there does not exist any satisfying solution.

In this chapter, my research over level-crossing and multi-level sequential pattern mining is described. The main idea of my research is that the generalization of sequence items can be performed when the subsequence support does not reach the minimal support value. I have studied both sub-problems – level-crossing and, also, multi-level sequential patterns. The naive solution for mining the level-crossing sequential patterns uncovered the main issues of the task which are the huge search space and the large result set. Therefore, there were proposed the constrains for mining multi-level sequential patterns which simplified the complexity of the mining process. The chapter summarizes facts published in research papers [41], [42] and [43]. All those research papers were presented as results of TAČR research project TA01010858: "*Improving Security of the Internet by Using System for Analyzing of Malicious Code Spreading*" .

## 4.1 Introduction

The existence of taxonomies can be used for mining level-crossing and multi-level sequential patterns. The advantage is shown in Examples 5 and 7 in the previous chapter. The result contains only three sequential patterns of the length two. The result of root sequential patterns, by contrast, contains 22 sequences of lengths from two up to four. These examples compare two extreme situations – the situation without the generalization and the situation with the maximal generalization of items. The simple mining method based on an extended database produces totally 77 sequential patterns. Such result contains high number of unimportant sequential patterns (e.g.

$\langle f_1\ a\rangle$ and also $\langle f\ a\rangle$ ). Note that in this chapter will be used the notations from the Example 4 on page 11.

For the rest of the thesis two basic tasks are distinguished:

- Level-crossing sequential patterns – items of sequential patterns can be generalized to any level of taxonomy.
- Multi-level sequential patterns – items of sequential patterns have to have the same level of taxonomy.

**Example 9.** The difference between the complexity of level-crossing and multi-level sequential patterns mining is shown on Figure 4.1. The multi-level mining approach creates only the most bottom and top sequences: $\langle a_1\ (b_2\ f_2)\rangle$ and $\langle a\ (b\ f)\rangle$ because levels of items of those sequential patterns must be the same. However, there are more 5 level-crossing sequential patterns between the pair of multi-level sequential patterns of length three. With the length of the sequential patterns, the number of level-crossing patterns increases.

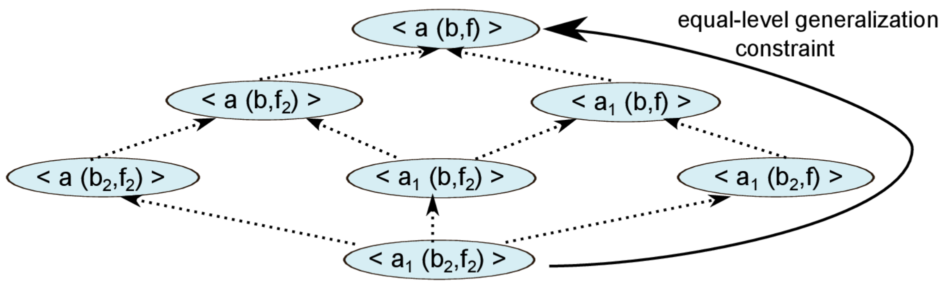

Figure 4.1: Difference between level-crossing and multi-level generalization of sequence $\langle a_1\ (b_2 f_2)\rangle$.

For multi-level sequential patterns mining, the definition of the *support* must be modified. The basic Def. 10 of the support takes into account only items of the sequences in the sequence database. Multi-level and level crossing sequential patterns can contain the items of the taxonomies and their ancestors which are not directly contained in the sequences. Therefore, using of Def. 10 is not possible.

A new support measure called a *generalized support* is introduced. The *generalized support gen_supp* is based on Def. 10 but the sequence subset relation is replaced by the *generalized subset relation* $\subseteq_g$ for the generalized support. Then, the generalized support must test if the subsequence contains an item or any of its descendants.

**Definition 21. (Generalized Support)** Given elements $e_1, e_2 \subseteq I$, the *generalized subset relation* $\subseteq_g$ is defined as

$$\begin{aligned} e_1 \subseteq_g e_2 \quad \Leftrightarrow \quad & \forall i \in e_1 : i \in e_2 \vee \\ & \exists j \in e_2 : i \in ancestors(j). \end{aligned} \tag{4.1}$$

A sequence $\alpha = \langle a_1 a_2 \dots a_n\rangle$ is a *generalized subsequence* of a sequence $\beta = \langle b_1 b_2 \dots b_m\rangle$ if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 \subseteq_g$

Table 4.1: Table shows differences between support and generalized support of ancestors of sequence $\langle(b_2\, f_2)\rangle$. The $SID$ column indicates sequences which support the sequence $s$. The Type column denotes the type of the sequence $s$: common sequence (S), level-crossing sequence (L-C) and multi-level sequence (M-L).

| Sequence $s$ | $support(s)$ | $gen\_supp(s)$ | $SID$ | Type |
|---|---|---|---|---|
| $\langle(b_2\, f_2)\rangle$ | 2 | 2 | 2,3 | S, M-L, L-C |
| $\langle(b_2\, f)\rangle$ | – | 3 | 1,2,3 | L-C |
| $\langle(b\, f_2)\rangle$ | – | 2 | 2,3 | L-C |
| $\langle(b\, f)\rangle$ | – | 4 | 1,2,3,4 | M-L, L-C |

$b_{j_1}, a_2 \subseteq_g b_{j_2}, \ldots, a_n \subseteq_g b_{j_n}$. We denote $\alpha \sqsubseteq_g \beta$. Formally, the definition of *the generalized support* of a sequence $s_1$ is

$$gen\_supp(s_1) = |\{\langle SID, s\rangle | (\langle SID, s\rangle \in D) \wedge (s_1 \sqsubseteq_g s)\}|. \tag{4.2}$$

**Example 10.** For given sequence database in Table 2.1, this example explains the method of generalized support counting for sequence $\langle(b_2\, f_2)\rangle$. The ancestors are the following: $\langle(b\, f_2)\rangle, \langle(b_2\, f)\rangle, \langle(b\, f)\rangle$. The support and generalized support of the sequence $\langle(b_2\, f_2)\rangle$ are equal to 2. The sequence $\langle(b_2\, f)\rangle$ is supported by sequences which contain $\langle(b_2\, f_2)\rangle$ or $\langle(b_2\, f_1)\rangle$. Therefore, the generalized support value of sequence $\langle(b_2\, f)\rangle$ is 3.

The rest of the thesis will use shortened term *support* instead of *generalized support.*

**Definition 22. (Generalization Procedure)** The *generalization procedure* (shortly *generalization*) is a procedure whose input is the node $n$ of the taxonomy $\tau$ and the output is the subset $N_a$ of ancestors of node $n$:

$$N_a \subseteq \tau \wedge n \in \tau \wedge N_a \subseteq ancestors(n). \tag{4.3}$$

## 4.2 Mining Level-Crossing Sequential Patterns

Sequential patterns are such subsequences which occur frequently in a sequence database. Level-crossing sequential patterns allow items to be on different levels of taxonomies. On the other hand, the search space significantly grows for level-crossing sequential patterns.

This section is based on facts published in [41] and [42]. First, the problem of mining level-crossing sequential patterns is formalized. Especially, the relations as parents and ancestors are defined for level-crossing sequences. Then, the algorithm for mining level-crossing sequential patterns is proposed and it is explained on the complex example.

### 4.2.1 Problem Definition

**Definition 23. (Element Parents)** Given an element $e = \{i_1, i_2, \ldots, i_n\}$, an *element parents* of the element $e$ is a set of the elements which are the same as $e$ except one of the items which is generalized. Formally,

$$\begin{aligned} parents_{el}(e) &= \{e \setminus \{i_k\} \cup \{parent(i_k)\} | i_k \in e \\ &\wedge parent(i_k) \notin e \wedge 1 \leq k \leq n\}. \end{aligned} \tag{4.4}$$

Note that items inside elements can be linearly ordered without lost of generality.

Now, the sequence parents and sequence ancestors can be defined using the element parent definition.

**Definition 24. (Sequence Parents)** Given a sequence $s = \langle e_1 e_2 \ldots e_n \rangle$, where $e_k$ are elements. The *sequence parents* of $s$ is the set of sequences that are the same as the sequence $s$ except one of their elements which is replaced by one of its element parents. Formally,

$$\begin{aligned} parents_{seq}(s) &= \{\langle f_1 f_2 \ldots f_n \rangle | f_k \in parents_{el}(e_k) \\ &\wedge 1 \leq k \leq n \\ &\wedge \forall l \neq k, 1 \leq l \leq n : e_l = f_l\}. \end{aligned} \tag{4.5}$$

**Definition 25. (Root Sequence)** Given taxonomy $\tau$, a *root sequence* is a sequence consisting of elements with items corresponding to root nodes only. The set of sequence parents of a root sequence is an empty set.

**Definition 26. (Sequence Ancestors)** Given the sequence $s$, the *sequence ancestors* of the sequence $s$ is defined as follows:

$$\begin{aligned} ancestors_{seq}(s) &= M_i, \text{ for such } i \text{ that } M_{i+1} = M_i, where \\ M_0 &= parents_{seq}(s) \\ M_{i+1} &= M_i \cup \{parents_{seq}(x) \mid x \in M_i\} \\ &\quad \text{for } i \geq 0. \end{aligned} \tag{4.6}$$

Example of sequence parents and sequence ancestors of sequence are in Example 11.

**Example 11.** For a given sequence $\langle a_{12}\, a_{11} \rangle : 1$, a set of parent sequences is the set of two sequences $\{\langle a_{12} a_1 \rangle : 1, \langle a_1\, a_{11} \rangle : 1\}$. The set of ancestors of the sequence $\langle a_{12}\, a_{11} \rangle : 1$ is the set of sequences $\{\langle a_{12}\, a_1 \rangle : 1, \langle a_1\, a_{11} \rangle : 1, \langle a_1\, a_1 \rangle : 1, \langle a_1\, a \rangle : 2, \langle a\, a_1 \rangle : 1, \langle a\ a \rangle : 2\}$. The sequence $\langle a\ a \rangle : 2$ is the root sequence and it has no parent and ancestor sequences.

### 4.2.2 The hGSP Algorithm

In this section, the algorithm *hGSP (hierarchical-GSP)* for mining level crossing sequential patterns is introduced. The algorithm is based on GSP [37] described in Section 3.2.2. In contrast to the method based on "extended-sequences", the hGSP algorithm reduces the number of redundant patterns. If a sequence $s$ is frequent and $s_1 \in ancestors_{seq}(s)$, then $s_1$ must be also frequent. Therefore, the sequence $s_1$ is *redundant* because it does not contain any new information. Due to the observation our algorithm does not generate all possible generalizations of frequent sequences. It performs generalization only when the sequence would be pruned. The hGSP is based on the idea of *concreteness* of each sequence. The concreteness measure is evaluated using information theory explained in the following subsection.

#### Concept of hGSP Algorithm

The main idea of the hGSP algorithm is that if a sequence $s$ has support $gen\_supp(s)$, there can exist a generalized sequence $s_g \in parents_{seq}(s)$ such that $gen\_supp(s_g) > gen\_supp(s)$. This can be applied repeatedly. Note that $\forall s_g \in parents_{seq}(s) : gen\_supp(s) \leq gen\_supp(s_g)$.

Generally, more specific sequence $s$ is more important result than its generalized form $s_g$ because the generalized $s_g$ is more expectable in the result set. It corresponds to the concept of information content.

**Definition 27. (Shannon information content)** The Shannon *information content* [25] of a value $x$ with probability $p(x)$ is defined as

$$h(x) = \log_2 \frac{1}{p(x)}. \tag{4.7}$$

The probability $p(s)$ that a sequence $s$ occurs in a sequence database $D$ is

$$p(s) = \frac{gen\_supp(s)}{|D|}. \tag{4.8}$$

The *information content of* $s$ in $D$ is

$$h(s) = \log_2 \frac{1}{\frac{gen\_supp(s)}{|D|}} = -\log_2 \frac{gen\_supp(s)}{|D|}. \tag{4.9}$$

For a sequence $s$, the dependence between information content $h(s)$ and generalized support $gen\_supp(s)$ causes that if the generalization from $s$ to $s_g$ is performed and $gen\_supp(s_g) > gen\_supp(s)$, then $h(s_g) < h(s)$. Some information is lost during generalization. Therefore, the generalization should be performed only if the candidate sequence is not frequent (i.e. $gen\_supp(s) < min\_supp$) or the GSP algorithm cannot perform *join* of two candidate sequences with joinable sequence ancestors.

**Definition 28. (Concreteness)** A sequence $s_1$ is more *concrete* than a sequence $s_2$ if $(h(s_2) < h(s_1)) \wedge (ancestors_{seq}(s_1) \cup s_1) \cap (ancestors_{seq}(s_2) \cup s_2) \neq \emptyset$.

It means that $s_1$ must have higher information content then $s_2$ to be more concrete and both sequences must have at least one common ancestor or one sequence is ancestor of the other.

## Algorithm details

The hGSP algorithm uses modified *join step* and *pruning step* of the GSP algorithm. The rest of the algorithm remains the same. The hGSP algorithm implementation assumes that items in elements are in lexicographic order.

The *join step* is modified for generating candidates of length $k \geq 3$. Let $s_1$ and $s_2$ be a pair of frequent sequences of length $k-1$. The join can be performed if subsequences of $s_1$ after omitting the first item and $s_2$ after omitting the last one have a common *sequence ancestor*. Then the joined sequence of length $k$ is composed from the first item of $s_1$, the most concrete sequence ancestor of common part and the last item of $s_2$. The last item is added as in GSP.

Support of candidates is counted similarly to original GSP. The only difference is that we use $gen\_supp(s)$ defined in Def. 21 instead of support. Therefore, only the procedure for checking, if a candidate is a subsequence of sequences in a given sequence database, is modified.

The modification of the *pruning step* is shown in Algorithm 4.2. The algorithm uses a method for finding the approximation of the most concrete generalization set of sequences which is described in Algorithm 4.1. The hGSP algorithm is based on the *greedy optimization* technique [9]. The method $FindGeneralization(s)$ returns the set $G_s$ of most concrete generalizations of the sequence $s$ with higher information value. Then the *hGSP* algorithm checks, if each sequence in $G_s$ is frequent. If so, it is added into set of sequential patterns, otherwise the candidate sequence is generalized again. Therefore, the algorithm finds only sequences corresponding to the local optimum of concreteness measure. Finding of a global optimum would be extremely computationally complex. It is not necessary to evaluate information content using logarithm functions but it is sufficient to compare ratios of supports of sequences and theirs generalized forms.

Given sequence $s$ and its generalized form $s_1$, the information contents of these sequences are $h(s) = -\log_2 \frac{gen\_supp(s)}{|D|}$ and $h(s_1) = -\log_2 \frac{gen\_supp(s_1)}{|D|}$. The information lost during generalization of $s$ to $s_1$ is $\Delta h = h(s) - h(s_1)$. It follows that

$$\Delta h = \log_2 \left( \frac{\frac{gen\_supp(s_1)}{|D|}}{\frac{gen\_supp(s)}{|D|}} \right) = \log_2 \left( \frac{gen\_supp(s_1)}{gen\_supp(s)} \right). \tag{4.10}$$

The generalization of $s$ with the smallest information loss is found because then the sequences will be the most concrete. Therefore, the algorithm minimizes ratio $\frac{gen\_supp(s_1)}{gen\_supp(s)}$.

Generalized sequences which contain ancestor item of another item in the same element are redundant and they are discarded.

### 4.2.3 Complex Example

In this section, the basic principles of hGSP are described in the Example 12 based on the dataset in Table 2.1 on page 9.

**Example 12.** The algorithm hGSP is running in Phases. In each Phase the database pass is made.

#### Phase 1

In the first phase, the algorithm finds all sequences of length 1. First, the support of all items are counted in the sequence database $\mathcal{D}$. The resulting 1-sequential patterns are 9 sequences: $L_1 = \{\langle a_{11}\rangle : 2, \langle a_{12}\rangle : 2, \langle a_1\rangle : 3, \langle a_2\rangle : 2, \langle b_1\rangle : 2, \langle b_2\rangle : 3, \langle d_1\rangle : 2, \langle f_1\rangle : 2, \langle f_2\rangle : 2\}$.

#### Phase 2

The algorithm continues by the second phase taking all pairs $s_1 \in L_1, s_2 \in L_1$ and trying to join them into new level-crossing sequential patterns.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1\rangle, s_2 = \langle b_2\rangle$ – Sequences are joined into new candidate sequences $s' = \langle d_1 b_2\rangle$ and $s'' = \langle (b_2 d_1)\rangle$ and sequences are added into candidate set $C_2$.
  **Counting Step:** The support for sequence $s'$ is counted $s' = \langle d_1 b_2\rangle : 1$. Such sequence does not satisfies the minimal support threshold and the generalization is performed $s'_{g_1} = \langle d_1\ b\rangle : 2$ and $s'_{g2} = \langle d\ b_2\rangle : 1$. The first generalization is enough and the candidate sequence $s'_{g_1}$ is added into result set $L_2$. The second generalized sequence $s'_{g2}$ does not satisfy the minimal support and it is removed. The second candidate sequence $s''$ is not frequent and cannot be generalized to any frequent ancestor, therefore, it is removed.

The hGSP algorithm finds 29 level-crossing sequential patterns $L_2$ of length 2.

#### Phase 3

In the next **Phase 3**, the algorithm finds 19 sequential patterns $L_3$ of length 3.

#### Phase 4

In **Phase 4**, the hGSP algorithm finds the following four sequential patterns $L_4 = \{\langle d_1\ (b\ f_1)\ a\rangle : 2\}, \langle d\ (b\ f)\ a\rangle : 2, \langle a_1\ (b\ f_1)\ a\rangle : 2\}, \langle a_1\ (b\ f)\ a\rangle : 2\}\}$.

#### Phase 5

Finally, during the **Phase 5** algorithm cannot create any candidate sequence and finishes.

The hGSP algorithm creates 200 candidate sequences totally for mining process and it reveals 61 level-crossing sequential patterns. The number of generated candidate sequences is still high. Such result is better than in case of using extended sequences in original GSP but improvements are still required.

**Algorithm 4.1** Method FINDGENERALIZATION()

```
procedure $FindGeneralization(s)$
    $G_s = \{\}$
    $min\_supp\_ratio = +\infty$
    for all $p_s \in parents_{seq}(s)$ do
        $ratio = gen\_supp(p_s)/gen\_supp(s)$
        if $(gen\_supp(p_s) \neq gen\_supp(s) \wedge ratio < min\_supp\_ratio$ then
            $G_s = \{p_s\}$
            $min\_supp\_ratio = ratio$
        else if $(ratio = min\_supp\_ratio$ then
            $G_s = G_s \cup \{p_s\}$
        end if
    end for
    return $G_s$
end procedure
```

**Algorithm 4.2** Pseudocode of hGSP Pruning Step

```
procedure HGSP$(C_k, min\_supp)$
    $L_k = \{\}$
    for all $s_c \in C_k$ do
        $C'_k = \{s_c\}$
        $sequence\_added = false$
        while $sequence\_added = false \wedge |C'_k| > 0$ do
            $G_s = \{\}$
            for all $s \in C'_k$ do
                if $gen\_supp(s) \geq min\_supp$ then
                    $L_k = L_k \cup \{s\}$
                    $sequence\_added = true$
                else
                    $G_s = G_s \cup$ FINDGENERALIZATION$(s)$
                end if
            end for
            $C'_k = G_s$
        end while
    end for
    return $L_k$
end procedure
```

## 4.3 Mining Hierarchically-Closed Multi-Level Sequential Patterns

This section presents the core result of my research work. It deals with the formal definition of the newly formulated task of *mining hierarchically-closed sequential patterns*, and then it describes a new algorithm for mining such sequential patterns. For better intelligibility, all steps are explained on detailed examples.

The level-crossing kind of sequential patterns introduced in previous section is the natural taxonomical (hierarchical) extension of the sequential patterns. However, the mining process of such patterns is very difficult and computationally expensive. Therefore, the simplification of the problem was introduced in the research paper [43]. The most computationally expensive task of mining level-crossing patterns is the number of various searched paths during the generalization. Moreover, the number of searched paths increases dramatically with the length of sequential patterns and the heights of taxonomies. The proposed solution is inspired by adding a new stronger constraint [18]. The constraint uses a multi-level approach. Generalizations are performed for all items in itemsets simultaneously. Therefore, the improvement is based on the multi-level sequential patterns concept. The main idea is to find only patterns containing items of the same level. It reduces the number of searched paths during the mining process.

The difference is explained on the following example which uses the taxonomies of a shop from Example 1 on page 4. The possible result of mining level-crossing sequential patterns can contain e.g. sequences like $\langle PC\_minitower\ ink\_printer\rangle$, $\langle PC\_minitower\ printer\rangle$ *or* $\langle PC\ printer\rangle$ because there is no constraint for the combination of the level of items. The multi-level sequential patterns, by contrast, must not contain the sequential pattern as $\langle PC\_minitower\ printer\rangle$ because the levels of items *PC_minitower* and *printer* are different.

The hierarchically-closed sequential patterns follow the idea of the hGSP algorithm which reveals only the most concrete patterns using the information content measure. It was observed, that the result becomes more clear and revealing if the closed patterns are used [45], [1]. In our example, the analyst could be overloaded by redundant patterns if the result contains all $\langle (PC\_minitower\ LCD\_monitor)\ ink\_printer\rangle$, $\langle PC\_minitower\ ink\_printer\rangle$, $\langle LCD\_monitor\ ink\_printer\rangle$, $\langle (PC\_minitower\ LCD\_monitor)\rangle$, etc. Moreover, no information is lost if the non-closed patterns are omitted and the longest sequential patterns with the equal support are found. On the other hand, the mining of the close patterns are more complicated, because result patterns must be retroactively pruned. In our work, the "close" problem is applied to the process of generalization. It leads to the similar type of redundant patterns like in mining closed sequential patterns – only the most specific patterns with the no support change are revealing. Ancestors of the frequent hierarchically-closed multi-level sequential patterns are always also frequent. However, the change of the support during the generalization is important. Therefore, we focus on the mining the hierarchically–closed instead of the hierarchically–maximal sequential patterns.

### 4.3.1 Problem Definition

This section deals with the formal basics of the mining hierarchically-closed multi-level sequential patterns. It follows the definitions Def. 1 (Itemset), Def. 8 (Sequence), Def. 9 (Sequence Database), Def. 14 (Concept Hierarchy), Def. 15 (Taxonomy of Items) and Def. 21 (Generalized Support).

First, the multi-level (ML) extensions of element, sequence, parent and ancestors must be defined. The definitions of ML-element and the ML-sequence are derived from definitions of element and sequence. The Definition 29 extends the element and the sequence definitions using items from nodes of taxonomies where the level of all items must be the same. The rest of the definition remains unchanged.

**Definition 29. (ML-element, ML-sequence)** Let $l \in \boldsymbol{N}$ be a level of items in a taxonomy $T \in \tau$. Then an *ML-sequence* is an ordered list of itemsets $s_{ML} = \langle s_1 s_2 s_3 \dots s_n \rangle$ such that the levels of all items of the itemsets are equal to $l$. The itemset of the ML-sequence is called an *ML-element*. The *length, subsequence and supersequence* of an ML-sequence is defined analogously to the ones in Definition 8.

**Example 13.** Next examples will be based on running example Example 2 on page 8. Assume three sequential patterns $\langle a_1\ (bf_1)\rangle : 2$ , $\langle a_1\ (bf)\rangle : 2$ and $\langle a\ (bf)\rangle : 2$. The first sequence $\langle a_1\ (bf_1)\rangle$ is not a ML-sequence because the level of items differs in the element $(bf_1)$ – the level of $b$ is 0 and the level of $f_1$is 1. The second sequence $\langle a_1\ (bf)\rangle$ is not a ML-sequence too because the level of items differs between elements. Finally, only the third sequence $\langle a\ (bf)\rangle$ is a ML-sequence because it satisfies Def. 29. Therefore, only the third sequence may be included in the result of the mining multi-level sequential patterns. Note that the sequence $\langle a\ (bf)\rangle$ is the root sequence because all its items are the root items.

Here, it is possible to define taxonomic relations between ML-sequences. The *ML-element parent* can be simply obtained by replacing all items of a ML-element by their parent items. Then for the *ML-sequence parent*, all its ML-elements are replaced by their ML-element parents. Note that the parent of a level-crossing sequence is a set of sequences but a ML-sequence has only one ML-sequence parent. The *ML-sequence ancestors* are the union of ML-sequence parents recursively up to the root of a ML-sequence. These statements are formalized in the following definitions.

**Definition 30. (ML-element parent)** Given an ML-element $e = \{i_1, i_2, \dots, i_n\}$, an *ML-element parent* of the ML-element $e$ is an element whose items are obtained by replacing all items of their parents. This is defined as

$$parent_{el}(e) = \{parent(i_k) | 1 \leq k \leq n \wedge i_k \in e\}. \tag{4.11}$$

**Definition 31. (ML-sequence parent, ML-sequence ancestors)** Given an ML-sequence $s = \langle e_1 e_2 \dots e_n \rangle$, where $e_k$ is a ML-element on a position $k$, the *ML-sequence parent* of $s$ is an ML-sequence such that all ML-elements of $s$ are replaced by their ML-element parents. Formally,

$$parent_{seq}(s) = \langle f_1 f_2 \dots f_n \rangle,\ f_k = parent_{el}(e_k), 1 \leq k \leq n. \tag{4.12}$$

**Definition 32. (ML-sequence ancestors)** For a given set of taxonomies $\tau$, a *root ML-sequence* is an ML-sequence consisting of ML-elements with items corresponding to root nodes of taxonomies. The ML-sequence parent of a root ML-sequence is not defined. Based on the definition of the ML-sequence parent, the *ML-sequence ancestors* of an ML-sequence $s$, $ancestors_{seq}(s)$ is defined recursively as follows:

$$\begin{aligned} ancestors_{seq}(s) &= M_i, \text{ for such } i \text{ that } M_{i+1} = M_i, \textit{where} \\ M_0 &= \{parent_{seq}(s)\} \\ M_{i+1} &= M_i \cup \{parent_{seq}(x) \mid x \in M_i\} \text{ for } i \geq 0. \end{aligned} \tag{4.13}$$

**Example 14.** For a given ML-sequence $\langle a_{12}\ a_{11}\rangle : 1$, the ML-sequence parent is $\langle a_1\ a_1\rangle : 1$. The set of ML-sequence ancestors of $\langle a_{12}\ a_{11}\rangle : 1$ is the set of two ML-sequences $\{\langle a_1\ a_1\rangle : 1, \langle a\ a\rangle : 2\}$. The ML-sequence $\langle a\ a\rangle : 2$ is the root sequence and it has no ML-sequence parent and ML-sequence ancestors (see Fig. 4.2). Note that if the input sequence is an ML-sequence, then the result parent sequence and ancestor sequences are ML-sequences too because of the principle of their construction.

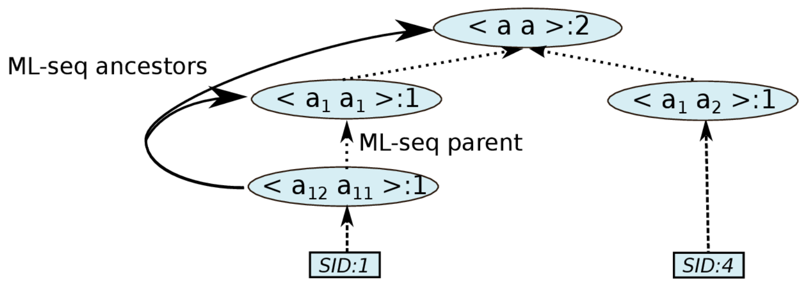

Figure 4.2: The example of the ML-sequence parent and ML-sequence ancestors. The most bottom line in squares are $SID$s of the origin sequences from the sequence database which increment support.

The multi-level approach reduces the search space of the data mining task. Moreover, we try to reduce the number of redundant (unimportant) patterns. Recall the term *closed* in *closed sequential pattern mining*. The *closed* means that if a sequence $s$ and a supersequence of $s$ have the same support, then the result set will contain only a supersequence of $s$. In this case, any omitted subsequence can be derived from the result set.

In the case of mining multi-level sequential patterns, the closeness property can be applied for taxonomic relations. If a ML-sequence $s$ and the ML-sequence ancestor of $s$ have the same support, then the result set will contain only the ML-sequence $s$. A new data mining task is called **mining hierarchically-closed multi-level sequential patterns**. It has the following two fundamental properties:

- Only ML-sequences are revealed. It ensures fulfillment of equal-level of all items in the sequential patterns. The generalization (level changes) is allowed during the mining process, however, all newly constructed sequences are ML-sequences.

- Sequences are filtered for the hierarchically-closed condition. If some ML-sequences are in the ancestor relation and have the same value of the generalized support, then only the most-bottom sequences in the meaning of taxonomies are revealed. The generalized support must be used because the generalized ML-sequences are supported by their more specific variants.

Let's summarize all three basic constraints for the task of mining hierarchically-closed multi-level sequential patterns:

- **Constraint 1 (C1)**: A sequential pattern $s$ must have sufficient support.
- **Constraint 2 (C2)**: A sequential pattern $s$ must be an ML-sequence.
- **Constraint 3 (C3)**: A sequential pattern $s$ must be hierarchically-closed.

The mining problem is formalized in the Definition 33 and explained in Example 15.

**Definition 33. (Mining hierarchically-closed multi-level sequential patterns)** The *set of hierarchically-closed ML-sequences* is such a set of ML-sequences which *does not* contain any ML-sequence $s$ and its ML-sequence ancestor with *equal* generalized supports. Then, the problem of **mining hierarchically-closed multi-level sequential patterns** (hereinafter *ML-sequential patterns*) for a given input sequence database $D$ and minimal generalized support threshold $min\_supp$ is to find a set $L_{ML}$ of all ML-sequences in $\mathcal{D}$ such that:

$$\begin{aligned} L_{ML} \;=\; & \{s_{ML} \sqsubseteq s | \langle SID, s\rangle \in \mathcal{D} \wedge gen\_supp(s_{ML}) \geq min\_supp \\ & \wedge \nexists s_x \sqsubseteq_g s[gen\_supp(s_x) \geq min\_supp \\ & \quad \wedge gen\_supp(s_x) = gen\_supp(s_{ML}) \\ & \quad \wedge s_{ML} \in ancestor_{seq}(s_x)]\}. \end{aligned} \tag{4.14}$$

**Example 15.** In this example, different situations of *mining hierarchically-closed multi-level sequential patterns* are described. The example describes the meaning of the constraints *C1-C3* and their implications while mining two root ML-sequences $\langle a\ d\rangle$ and $\langle (a\ b)\rangle$ and their descendants. Given the minimal support value $min\_supp = 2$ and the sequence database $\mathcal{D}$ from the Example 2 on page 8. The example shows interesting hierarchical relations among constructed sequences during the mining process.

First case. The root ML-sequence $\langle a\ d\rangle$ is supported by two sequences in sequence database $\mathcal{D}$: $SID = 1$ and $SID = 4$. The items of sequences in $D$ counted for support are marked in bold:

- $SID\ 1 : \langle (c_1\ d_1)(\mathbf{a_{12}}\ b_1\ c_1)(\mathbf{a_1}\ b_2\ f_1)(a_{11}\ c_1\ \mathbf{d_1}\ f_1)\rangle$
- $SID\ 4 : \langle \mathbf{a_{11}}\ (\mathbf{d_1}\ g_1\ h_1)(b_1\ f_1)(a_2\ g_2\ h_2)\rangle$

Descendants of the root ML-sequence $\langle a\ d\rangle$ are shown on Fig. 4.3 and a related mining hierarchically-closed ML-sequential patterns process is explained below.

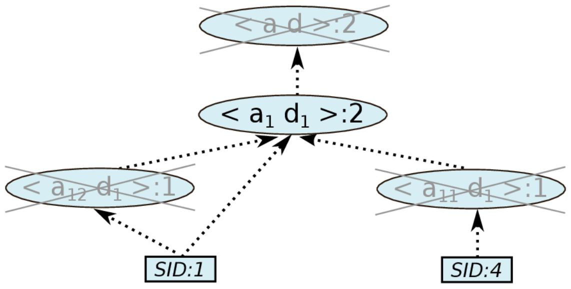

Figure 4.3: Example of the generalization process of sequences $\langle a_{12}\ d_1 \rangle$ and $\langle a_{11}\ d_1 \rangle$ in which constraints for hierarchically-closed ML-sequential patterns are applied.

- The sequence $\langle a_{12}\ d_1 \rangle$ **does not satisfy** following constraints:
  - C1: Only a sequence with $SID = 1$ in the sequence database supports it, therefore, it does not satisfy *min_supp*. The support of the sequence is $gen\_supp(\langle a_{12}\ d_1 \rangle) = 1$.
  - C2: The sequence is not a ML-sequence.

  The sequence must be generalized to the closest ML-sequence $\langle a_1\ d_1 \rangle$.

- The sequence $\langle a_{11}\ d_1 \rangle$ **does not satisfy** following constraints:
  - C1: Only a sequence with $SID = 4$ in the sequence database supports it, therefore, it does not satisfy *min_supp*. The support of the sequence is $gen\_supp(\langle a_{11}\ d_1 \rangle) = 1$.
  - C2: The sequence is not a ML-sequence.

  The sequence must be generalized to the closest ML-sequence $\langle a_1\ d_1 \rangle$.

- The sequence $\langle a_1\ d_1 \rangle$ **satisfies** all constraints C1, C2 and C3. Two different approaches can be used for construct the sequence:
  - by generalization from $\langle a_{12}\ d_1 \rangle$ and/or $\langle a_{11}\ d_1 \rangle$,
  - directly from the sequence database from $SID = 1$.

  Sequence $\langle a_1\ d_1 \rangle$ is supported by sequences $SID = 1$ and $SID = 4$, therefore, the support of the sequence is $gen\_supp(\langle a_1\ d_1 \rangle) = 2$. The sequence $\langle a_1\ d_1 \rangle$ is a hierarchically-closed multi-level sequential pattern.

- The root sequence $\langle a\ d \rangle$ **does not satisfy** following constraint:
  - C3: The sequence $\langle a\ \ d \rangle$ is not hierarchically-closed because $gen\_supp(\langle a_1\ d_1 \rangle) = gen\_supp\langle a\ d \rangle)$.

  The sequence $\langle a\ d \rangle$ will not be included into result.

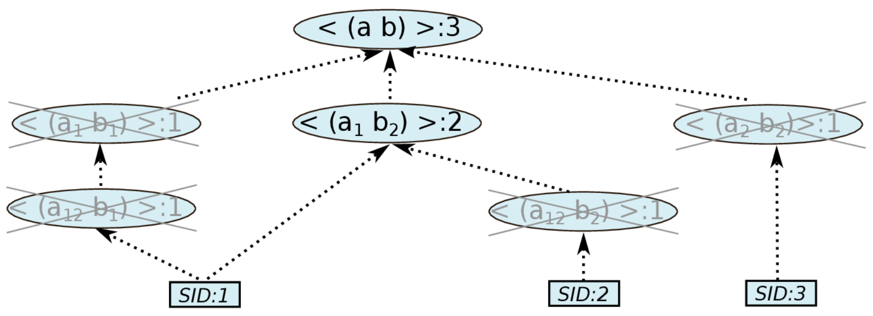

Figure 4.4: Example of the generalization process of sequences $\langle a_{12}\ b_1\rangle$, $\langle a_{12}\ b_2\rangle$ and $\langle a_2\ b_2\rangle$ in which constraints for hierarchically-closed ML-sequential patterns are applied.

Second case. The root sequence $\langle(a\ b)\rangle$ is supported by three sequences in sequence database $\mathcal{D}$: $SID = 1$, $SID = 2$ and $SID = 3$.:

- $SID\ 1:\ \langle(c_1\ d_1)(\mathbf{a_{12}\ b_1}\ c_1)(\mathbf{a_1\ b_2}\ f_1)(a_{11}\ c_1\ d_1\ f_1)\rangle$
- $SID\ 2:\ \langle(\mathbf{a_{12}\ b_2}\ f_2)\ e_1\rangle$
- $SID\ 3:\ \langle(\mathbf{a_2\ b_2}\ f_2)\rangle$

Descendants of the root ML-sequence $\langle(a\ b)\rangle$ are shown on Fig. 4.4 and a related mining hierarchically-closed ML-sequential patterns process is explained below.

- The sequence $\langle(a_{12}\ b_1)\rangle$ **does not satisfy** following constraints:
  - C1: Only a sequence with $SID = 1$ in the sequence database supports it, therefore, it does not satisfy *min_supp*. The support of the sequence is $gen\_supp(\langle(a_{12}\ b_1)\rangle) = 1$.
  - C2: The sequence is not a ML-sequence.

  The sequence must be generalized to the closest ML-sequence$\langle(a_1\ b_1)\rangle$.

- The sequence $\langle(a_1\ b_1)\rangle$ **does not satisfy** following constraint:
  - C1: Only a sequence with $SID = 1$ in the sequence database supports it, therefore, it does not satisfy *min_supp*. The support of the sequence is $gen\_supp(\langle(a_1\ b_1)\rangle) = 1$.

  The sequence must be generalized to the closest ML-sequence parent $\langle(a\ b)\rangle$.

- The sequence $\langle(a_{12}\ b_2)\rangle$ **does not satisfy** following constraints:
  - C1: Only a sequence with $SID = 2$ in the sequence database supports it, therefore, it does not satisfy *min_supp*. The support of the sequence is $gen\_supp(\langle(a_{12}\ b_2)\rangle) = 1$.

  - C2: The sequence is not a ML-sequence.

  The sequence must be generalized to the closest ML-sequence $\langle a_1\ b_2 \rangle$.

- The sequence $\langle (a_1\ b_2) \rangle$ **satisfies** all constraints C1, C2 and C3.
  - Sequence $\langle (a_1\ b_2) \rangle$ is supported by sequences $SID = 1$ and $SID = 2$, therefore, the support of the sequence is $gen\_supp(\langle (a_1\ b_2) \rangle) = 2$.
  - The sequence $\langle (a_1\ b_2) \rangle$ is a hierarchically-closed multi-level sequential pattern.

- The sequence $\langle (a_2\ b_2) \rangle$ **does not satisfy** following constraint:
  - C1: Only a sequence with $SID = 3$ in the sequence database supports it, therefore, it does not satisfy $min\_supp$. The support of the sequence is $gen\_supp(\langle (a_2\ b_2) \rangle) = 1$.

  The sequence must be generalized to the closest ML-sequence parent $\langle (a\ b) \rangle$.

- The root sequence $\langle (a\ b) \rangle$ **satisfies** all constraints C1, C2 and C3.
  - The root sequence $\langle (a\ b) \rangle$ is supported by three sequences of sequence database $\mathcal{D}$: $SID = 1$, $SID = 2$ and $SID = 3$, therefore, the support of the sequence is support $gen\_supp(\langle (a\ b) \rangle) = 3$.
  - The sequence $\langle (a\ b) \rangle$ is a hierarchically-closed multi-level sequential pattern. In contrast to the case with sequence $\langle a\ d \rangle$, the sequential pattern $\langle (a\ b) \rangle$ has got greater support than all its children and, therefore, it satisfies also the C3.

**Example 16.** Let's consider that we have a method for *mining hierarchically-closed multi-level sequential patterns* by Def. 33 and let's consider the sequence database $\mathcal{D}$ from Example 2 on page 8 and given parameter $min\_supp = 2$. Then the outputs of such method are sequential patterns shown in Table 4.2.

Results of all approaches to the mining of sequential patterns with defined taxonomies presented in this thesis that were applied on the sequential database from Example 2 on page 8 are summarized in Table 4.3. Case 1 shows the parameters of the result mining basic sequential patterns. Total number of patterns is 11 with maximal length only 2. In the case, much information, which is contained in sequences on higher taxonomy levels, is lost during the mining process. On the other hand, Case 2 reveals a high number of sequences of the maximum length 4. However, it contains also redundant sequential patterns (e.g. a sequence and its parent with the same support) which do not bring any new information to the analyst. Case 3 is opposite to Case 1. It contains only the top-most items. Discovered sequential patterns in Case 3 contain only very general elements. Finally, Case 5 reveals sequential patterns by Def. 33 which are between extreme Cases 1 and 3. It finds only important patterns to the analyst. Case 4 is an intermediate step to Case 5 – it reduces the number of redundant patterns but it is still computationally expensive.

Table 4.2: The result of mining hierarchically-closed multi-level sequential patterns in the sequence database $\mathcal{D}$.

| # | **1-seq. patterns** | **2-seq. patterns** | **3-seq. patterns** | **4-seq. patterns** |
|---|---|---|---|---|
| 1. | $\langle a \rangle : 4$ | $\langle aa \rangle : 2$ | $\langle (abf) \rangle : 3$ | $\langle a\ (bf)\ a \rangle : 2$ |
| 2. | $\langle a_1 \rangle : 3$ | $\langle (ab) \rangle : 3$ | $\langle aba \rangle : 2$ | $\langle d\ (bf)\ a \rangle : 2$ |
| 3. | $\langle a_2 \rangle : 2$ | $\langle (a_1 b_2) \rangle : 2$ | $\langle a(bf) \rangle : 2$ | |
| 4. | $\langle a_{11} \rangle : 2$ | $\langle ab \rangle : 2$ | $\langle afa \rangle : 2$ | |
| 5. | $\langle a_{12} \rangle : 2$ | $\langle a_1 d_1 \rangle : 2$ | $\langle (bf)a \rangle : 2$ | |
| 6. | $\langle b \rangle : 4$ | $\langle (af) \rangle : 3$ | $\langle dba \rangle : 2$ | |
| 7. | $\langle b_1 \rangle : 2$ | $\langle a_1 f_1 \rangle : 2$ | $\langle d(bf) \rangle : 2$ | |
| 8. | $\langle b_2 \rangle : 3$ | $\langle ba \rangle : 2$ | $\langle dfa \rangle : 2$ | |
| 9. | $\langle d_1 \rangle : 2$ | $\langle (bf) \rangle : 4$ | | |
| 10. | $\langle f \rangle : 4$ | $\langle (b_2 f_2) \rangle : 2$ | | |
| 11. | $\langle f_1 \rangle : 2$ | $\langle da \rangle : 2$ | | |
| 12. | $\langle f_2 \rangle : 2$ | $\langle d_1 b_1 \rangle : 2$ | | |
| 13. | | $\langle d_1 f_1 \rangle : 2$ | | |
| 14. | | $\langle fa \rangle : 2$ | | |

## 4.3.2 The MLSP Algorithm

Han et al. in their book [19] characterized the sequential pattern mining by following words: *"Sequential pattern mining is computationally challenging because such mining may generate and/or test combinatorial explosive number of intermediate subsequences."* The task of *mining hierarchically-closed multi-level sequential patterns* is even more difficult because of the traversing taxonomies and the result pruning. As the research result, the algorithm MLSP (*Multi-Level Sequential Patterns* algorithm) was proposed in [43] for the effective data mining of multi-level sequential patterns.

The algorithm MLSP is based on the candidate generation principle (adapted from the GSP, see Section 3.2.2) combined with the on-demand generalization. The algorithm works in phases.

### The first phase

The algorithm passes through the sequence database and the generalized support is counted for all items. Unlike GSP, the MLSP continues the first phase by generalization procedure. Candidate 1-sequences are created from all items in the sequence database $\mathcal{D}$. Candidate sequences are processed as follows:

1. The set of candidate 1-sequences is expanded by their all ML-sequence ancestors.

2. The value of the generalized support is counted for all candidate 1-sequences.

3. All hierarchically-closed 1-sequences with the sufficient support are added into the set of sequential patterns.

Table 4.3: Summarization of the results of approaches to the mining of sequential patterns with defined taxonomies over items.

| Case | Type | Patt. Count / Max Length | Redundant Patterns Count | Reference |
|---|---|---|---|---|
| 1 | Sequential Patterns | 11/2 | - - | Example 7 |
| 2 | Sequential Patterns in Extended Sequences | 90/4 | 54 | Example 8 |
| 3 | Sequential Patterns Containing Root Items Only | 26/4 | 0 | Example 5 |
| 4 | Level-Crossing Sequential Patterns | 61/4 | 25 | Example 12 |
| 5 | **Hierarchically-closed Multi-level Sequential Patterns** | **36/4** | **0** | Example 16 |

Sequential patterns of length 1 are outputted by the algorithm and passed to the second phase.

### The next phases

The next phases of the algorithm run repeatedly until any new sequential pattern is generated. There are two steps during each phase:

1. *candidate generation step,*
2. *counting candidates step.*

### Candidate Generation Step

The *candidate generation step* is based on the *join* and *prune* principles. In the *join* procedure, all pairs of $k$-length ML-sequential patterns are taken. They are tested if they are joinable to the $(k+1)$-length candidate ML-sequences. Similarly to GSP, a $k$-length ML-sequential pattern $s_1$ can be joined with a $k$-length ML-sequential pattern $s_2$ if the subsequence created by removing the first item of $s_1$ and the subsequence created by removing the last item of $s_2$ are equal. Moreover in MLSP, the ML-sequences are also joinable if it is possible to perform such generalization of both subsequences of sequences $s_1$ and $s_2$, in which a common ML-sequence ancestor can be found. The MLSP algorithm tries to find the common ML-sequence ancestor of the candidate ML-subsequences in a *bottom-up way.* If a common ML-subsequence ancestor exists, then the generalized ML-sequences are joined into the new candidate ML-sequence, otherwise, no candidate is generated. Levels of ML-sequences $s_1$ and $s_2$ can be different, but the levels of items of the generated ML-sequence are the same. Finally, the prune principle is applied. The pruning is based on the Apriori theorem of the possible frequent sequences. For the multi-level sequential patterns, the **Apriori theorem** must be modified as follows (further referred as to MLSP Apriori Rule): *All ML-subsequences and their ancestors of a frequent ML-sequence are frequent too.*

The procedure for candidate generation is shown in Algorithm 4.3. The necessity of the generalization during the join step is demonstrated in the real-world Example 17. Finally, the whole procedure is explained in the running Example 18,

**Algorithm 4.3** Method GENERATECANDIDATEMLSEQUENCES()

```
1: procedure GENERATECANDIDATEMLSEQUENCES($L_{k-1}, k$)
2:     $C_k = \emptyset$
3:     for all $s_1, s_2 \in L_{k-1}$ do
4:         if ML-subsequences MLSP join condition is fulfilled for $s_1$ and $s_2$ then
5:             Join sequences $s_1$ and $s_2$ to a new ML-sequence $s'$
6:             if the MLSP Apriori Rule is fulfilled for $s'$ then
7:                 Add $s'$ into $C_k$.
8:             end if
9:         end if
10:    end for
11:    return $C_k$
12: end procedure
```

**Example 17.** Assume a pair of 2-ML-sequences ⟨*PC_minitower CRT_monitor*⟩ and ⟨*LCD_monitor laser_printer*⟩ which cannot be joined in GSP because subsequences ⟨*CRT_monitor*⟩ and ⟨*LCD_monitor*⟩ are not equal. However for MLSP, they have a common parent *monitor*. Therefore, the items should be generalized to the common subsequence ⟨*monitor*⟩ and the sequences are joined to a 3-ML-sequence⟨*PC, monitor, printer*⟩ by MLSP.

**Example 18.** Assume the join of the following two 2-length multi-level sequential patterns: $\langle ba \rangle : 2$ and $\langle a_1 f_1 \rangle : 2$. These two ML-sequences are firstly tested if the join is possible. Because the ML-subsequences $\langle a \rangle$ and $\langle a_1 \rangle$ have a common ancestor $\langle a \rangle$ they are joinable. The second ML-sequence is generalized to $\langle af \rangle$ and then ML-sequences are joined into a new ML-sequence $\langle baf \rangle$. Finally, the ML-sequence is tested for MLSP Apriori Rule. The ML-subsequence $\langle bf \rangle$ and it's any ancestor are not frequent, therefore, the ML-sequence $\langle baf \rangle$ is also not frequent and the ML-sequence is not added to the set of candidate sequences. In another case, assume the join of ML-sequential patterns $\langle (ab) \rangle : 3$ and $\langle (bf) \rangle : 4$. The join condition is fulfilled and a new ML-sequence $\langle (abf) \rangle$ is created. The Apriori test verifies that all ML-subsequences $\langle (ab) \rangle$, $\langle (bf) \rangle$ and finally $\langle (af) \rangle$ are frequent . The ML-sequence $\langle (abf) \rangle$ is added to the set of candidate sequences.

### Counting Candidates Step

When all candidate ML-sequences are generated, the frequent sequential patterns are filtered by the support value. The counting step consists of two substeps:

- test and generalization procedure,
- pruning of not hierarchically-closed sequential patterns.

The idea of the test and generalization substep is to read the sequence database and count the generalized support of all candidate ML-sequences $s_c \in C_k$. For each $s_c$, one of the following results is possible:

1. The generalized support value **satisfies** the minimal support threshold and the ML-sequence is marked as frequent one, denoted as $s_c^f$.
2. The generalized support value **does not satisfy** the minimal support threshold and then the generalization procedure is performed. The generalization of the ML-sequence tries to find a ML-sequence ancestor with the greatest sequence level **which satisfies** the minimal support threshold. The on-demand *bottom-up* generalization procedure GetFirstFrequentAncestor() is shown in Algorithm 4.4. Upper-level ML-sequence is tested recursively until the ancestor is found or the generalization procedure reach the root.

**Algorithm 4.4** Method GetFirstFrequentAncestor()

```
1: procedure GetFirstFrequentAncestor($s, min\_supp$)
2:     repeat
3:         if $gen\_supp(s) \geq min\_supp$ then
4:             return $s$
5:         end if
6:         $s \leftarrow parent_{seq}(s)$
7:     until $s$ is root sequence
8:     return null
9: end procedure
```

**Example 19.** The length 2 ML-sequence $\langle a_1 a_1 \rangle$ is generated in the running example from the 1-sequence $\langle a_1 \rangle : 3$ by the Candidate Generation step. All subsequences are frequent, therefore, the sequence may be frequent. However, after Counting Candidates, the generalized support of the sequence is 1 which does not satisfy the minimal support threshold value. Therefore, the generalization is performed by the MLSP algorithm and the ML-sequence parent $\langle a\, a \rangle$ is formed. The Counting Candidates step evaluates the generalized support to 2. The ML-sequence ancestor (ML-sequence parent) $\langle a\ a \rangle : 2$ of ML-sequence $\langle a_1 a_1 \rangle : 1$ is frequent and it is a ML-sequential pattern. Moreover, the hierarchically-close condition is satisfied and the sequence is hierarchically-closed multi-level sequential pattern by our definition.

### The MLSP Algorithm Summarization

The algorithm MLSP has two inputs: a sequence database $\mathcal{D}$ with a taxonomy (or taxonomies) defined for its items and a minimal support threshold value.

The algorithm output is the set of hierarchically-closed multi-level sequential patterns. The algorithm MLSP runs in the phases. The sequence database $\mathcal{D}$ is passed once in each phase. The first phase generates 1-length hierarchically-closed multi-level sequential patterns. Next phases generate $(k+1)$-length hierarchically-closed multi-level sequential patterns from the $k$-length sequential patterns. Because there can exist candidate ML-sequences that are not hierarchically-closed, it is necessary to verify that there is no child of the candidate ML-sequence with the same generalized support. The procedure for the effective check of this constraint is described in Section 4.3.5. The algorithm runs until any hierarchically-closed multi-level sequential patterns are generated. The algorithm generates the complete set of hierarchically-closed multi-level sequential patterns. The complete MLSP algorithm is formalized in Algorithm 4.5.

**Algorithm 4.5** The pseudocode of the MLSP algorithm

```
1: procedure MLSP($\mathcal{D}, min\_supp$)
2:     $k \leftarrow 1$                                          ▷ First phase.
3:     $I \leftarrow$Insert all items and all their ancestors $i$ in $\mathcal{D}$ and count their support
   $gen\_supp(i)$
4:     $C_1 \leftarrow$Add all 1-ML-sequences for all items $i$ from $I$
5:     $L_1 \leftarrow \{\}$
6:     for all $s_c \in C_1$ do
7:         if $gen\_supp(s_c) \geq min\_supp$ and $s_c$ is hierarchically-closed then
8:             $L_1 \leftarrow L_1 \cup \{s_c\}$
9:         end if
10:    end for
11:    while $L_k \neq \emptyset$ do                             ▷ Next iterative phases.
12:        $k \leftarrow k+1$
13:        $C_k \leftarrow$GenerateCandidateMLSequences($L_{k-1}, k$)
14:        Count support $gen\_supp(s)$ in $\mathcal{D}$ for all candidate ML-sequences and their
   ML-sequence ancestors $s \in \cup_{s_c \in C_k} ancestor_{seq}(s_c) \cup \{s_c\}$
15:        $L_{TMP} \leftarrow \{\}$
16:        for all $s_c \in C_k$ do
17:            $L_{TMP} \leftarrow L_{TMP} \cup$ GetFirstFrequentAncestor($s_c, min\_supp$)
18:        end for
19:        $L_k \leftarrow \{\}$
20:        for all $s \in L_{TMP}$ do
21:            if $s$ is hierarchically-closed then
22:                $L_k \leftarrow L_k \cup \{s\}$
23:            end if
24:        end for
25:    end while
26:    return $\bigcup_{i=1}^{k} L_i$
27: end procedure
```

### 4.3.3 Optimization 1: Is-generalized-subsequence Check in Linear Time-Complexity

The algorithm often performs "is-generalized-subsequence" test (e.g. for the generalized support counting). It uses the *generalized subset relation* $\sqsubseteq_g$. The test can be optimized to the linear time-complexity if a suitable complete ordering exists over items. The simple lexicographical ordering cannot be used for MLSP because the simple lexicographical ordering cannot be used because of the generalization which changes the order. Therefore, MLSP uses two step ordering.

1. It sorts taxonomies lexicographically by their roots. It provides for a grouping of items within elements by taxonomy.

2. Items within the taxonomy must be sorted unambiguously. The bottom-up order is suitable, because such ordering can be used for join step comparison and searching of the minimal necessary generalization. Suitable order type is a post-order walk [5] (see Example 20). The post-order makes traversal in the following steps:

   (a) Traverse the left subtree by recursively calling the post-order.

   (b) Traverse the right subtree by recursively calling the post-order.

   (c) Return current element.

3. It guarantees that it is possible to check for an ideal mapping to ancestors in linear time complexity.

The complete method for the "Is a generalized subsequence" test is shown in Algorithm 4.6. The procedure IsGeneralizedSubsequence() tests if a sequence $s_{sub}$ is the generalized subset of a sequence $s_{super}$: $s_{sub} \sqsubseteq_g s_{super}$. The maximal time complexity of the procedure is $m+n$ where $m$ is a number of elements in sequence $s_{sub}$ and $n$ is the number of elements in sequence $s_{super}$. The same is for each tested element in sub-procedure ContainsGeneralizedElement(). Finally, IsAncestor() runs with constant time-complexity using a hash table or with linear time complexity using tree traversal. Therefore, the whole procedure keeps linear time-complexity w.r.t. to lengths of sequences $s_{sub}$ and $s_{super}$.

**Example 20.** The post-order walk for taxonomy $a$ is shown on Figure 4.5.

### 4.3.4 Optimization 2: Hash Table Pre-Check for Is-generalized-subsequence Check

The majority of "is-generalized-subsequence" tests return *false*. Such major *false* case can be optimized by the pre-check. The $s_{sub} \sqsubseteq_g s_{super}$ is *true* if all items of subsequence $s_{sub}$ are contained in the set of items and their ancestors of supersequence $s_{super}$. If it is false, then the test results in false too. Then, a set of all items and

**Algorithm 4.6** Methods ContainsGeneralizedElement() and IsGeneralizedSubsequence()

```
procedure IsAncestor($item_1$, $item_2$)
    return $true$ if $item_1$ is ancestor or equal to $item_2$, $false$ otherwise
end procedure
procedure ContainsGeneralizedElement($e_{sub}$, $e_{super}$)
    $ptr_{super} \leftarrow 0$
    $ptr_{sub} \leftarrow 0$
    $found \leftarrow false$
    for $ptr_{sub} \leftarrow 0$; $ptr_{sub} <$ Size($e_{sub}$); $ptr_{sub} = ptr_{sub} + 1$ do
        do
            if $Size(e_{super}) \leq ptr_{super}$ then
                return $false$
            end if
            $found \leftarrow$IsAncestor($e_{sub}[ptr_{sub}]$, $e_{super}[ptr_{super}]$)
            $ptr_{super} = ptr_{super} + 1$
        while $found = false$
    end for
    return $(ptr_{sub} = Size(e_{sub}) \wedge found = true)$
end procedure
procedure IsGeneralizedSubsequence($s_{sub}$, $s_{super}$)
    $ptr_{sub} \leftarrow 0$
    $ptr_{super} \leftarrow 0$
    $found \leftarrow false$
    for $ptr_{sub} \leftarrow 0$; $ptr_{sub} < ElementCount(s_{sub})$; $ptr_{sub} = ptr_{sub} + 1$ do
        do
            if $ElementCount(s_{super}) \leq ptr_{super}$ then
                return $false$
            end if
            $found \leftarrow$ContainsGeneralizedElement($s_{sub}[ptr_{sub}]$, $s_{super}[ptr_{super}]$)
            $ptr_{super} = ptr_{super} + 1$
        while $found = false$
    end for
    return ($ptr_{sub} =$ ElementCount($s_{sub}$) $\wedge$ $found = true$)
end procedure
```

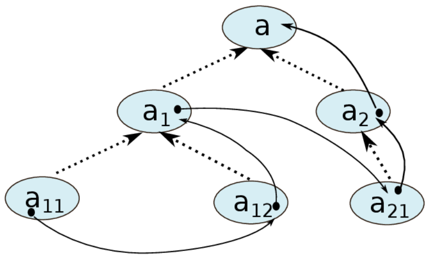

Figure 4.5: Post-order walk for taxonomy of $a$ is a list: $(a_{11}, a_{12}, a_1, a_{21}, a_2, a)$.

their ancestors is constructed for each sequence. Finally, the procedure IsGeneralizedSubsequence() can be completed by the fast pre-check for false result. The procedure is denoted in Algorithm 4.7 and it may be placed as a part of Algorithm 4.6 before the Line 23. The procedure assumes that there exists a simple function `GetAllItems()` which returns the set of all items in all elements of the sequence.

Such set is organized (stored) as a hash table in a main memory because its search time complexity is equal to 1 (details about generic hash table algorithms and their properties are in [5]). Maximal number of searches in the hash table is equal to the length of the sequence $s_{sub}$ . Final time-complexity of the whole pre-check is maximally linear too but it speeds-up the check for the most cases (see experiments in the next chapter).

**Algorithm 4.7** Pre-check procedure pseudocode for method IsGeneralizedSubsequence()

```
1: for all  $i \in GetAllItems(s_{sub})$ do
2:     if  $i \notin GetAllItems(s_{super}) \wedge i \notin \bigcup_{x \in GetAllItems(s_{super})} ancestor(x)\}$ then
3:         return $false$
4:     end if
5: end for
```

### 4.3.5 Optimization 3: Is-redundant Fast Check

Sequential patterns created by the join and generalization algorithm steps may not be hierarchically-closed. Then, the post-processing (filtering) is necessary. A naive approach compares each pair of sequential patterns, if one ML-sequence is an ML-sequence ancestor of the other and prunes them, if so. Nevertheless, it is possible to utilize the *Counting Candidates Step* procedure to mark sequential patterns which are redundant.

- First, we associate a new helper indexed list of counters called a *redundant base* to all candidate ML-sequences before the counting step. During the counting step of a candidate ML-sequence $s_c$ , the algorithm increments by one the redundant base counter on index $s_c^f$ to all ancestors: $S = ancestors_{seq}(s_c^f)$ when the generalization sub-procedure finds the most specific frequent sequential pattern $s_c^f \in ancestor_{seq}(s_c) \cup \{s_c\}$. The redundant base of a ML-sequence $x \in S$ on index $s_c^f$ is denoted as $\mathcal{RB}_x[s_c^f]$.

- Finally, the prune condition can be formulated as follows:
  - If there **exists any redundant base counter** with value equal to the value of the **generalized support** of the ML-sequence $s_c^f$, then the ML-sequence $s_c^f$ **is redundant** and is pruned,
  - **else**, $s_c^f$ **is hierarchically-closed multi-level sequential pattern**.

### 4.3.6 Complex Example

The MLSP algorithm contains several principles which are harder to understand without a demonstration. Therefore, the whole algorithm is explained on a complex step-by-step Example 21.

**Example 21. (Complex step-by-step example)** The example uses the sequence database from the running Example 2 on page 8. For better readability of example, the table from Example 2 is copied here to Table 4.4.

Table 4.4: A sequence database $\mathcal{D}$ containing items on different taxonomy level.

| SID | Sequence |
|---|---|
| 1 | $\langle(c_1\, d_1)(a_{12}\, b_1\, c_1)(a_1\, b_2\, f_1)(a_{11}\, c_1\, d_1\, f_1)\rangle$ |
| 2 | $\langle(a_{12}\, b_2\, f_2)\ e_1\rangle$ |
| 3 | $\langle(a_2\, b_2\, f_2)\rangle$ |
| 4 | $\langle a_{11}\ (d_1\, g_1\, h_1)(b_1\, f_1)(a_2\, g_2\, h_2)\rangle$ |

#### Phase 1

The initial step of the algorithm is scanning of the sequence database $\mathcal{D}$ and counting the values of generalized support for all items and their ancestors:

- The first processed sequence of $\mathcal{D}$ is $SID$=1 and the first read item is $c_1$ which is supported only by one sequence with $SID = 1$(shortly as $SID$ 1). Also, the item $c_1$ is generalized to its parent $c$ which is also supported only by $SID$ 1.

- Item $d_1$ is the next processed item. Its generalized support value is initiated to 1 because of support by $SID$ 1. Also, the item is generalized to its parent $d$. Then, the support of $d_1$ is incremented during the scan of the $SID$ 4 to the value 2. The same for $d$.

- The value of the generalized support of the item $a_{12}$ is initiated to 1. Also, the item is generalized to both ancestors $a_1$ and $a$ and their supports are also initiated to 1. During the scan of $SID$ 2, all support values of items $a_{12}$, $a_1$, $a$ are incremented to 2. Next, during the scan of the sequence $SID$ 3, the

Table 4.5: Counted values of generalized support of all items of the sequence database $\mathcal{D}$.

| **Item** | **Supported by *SID*s** | **Support** | | **Item** | **Supported by *SID*s** | **Support** |
|---|---|---|---|---|---|---|
| $c_1$ | 1 | 1 | | **$a_{11}$** | **2,3** | **2** |
| $c$ | 1 | 1 | | **$f_2$** | **2,3** | **2** |
| **$d_1$** | **1,4** | **2** | | $e_1$ | 2 | 1 |
| **$d$** | **1,4** | **2** | | $e$ | 2 | 1 |
| **$a_{12}$** | **1,2** | **2** | | **$a_2$** | **3,4** | **2** |
| **$a_1$** | **1,2,4** | **3** | | $g_1$ | 4 | 1 |
| **$a$** | **1,2,3,4** | **4** | | $g$ | 4 | 1 |
| **$b_1$** | **1,4** | **2** | | $h_1$ | 4 | 1 |
| **$b$** | **1,2,3,4** | **4** | | $h$ | 4 | 1 |
| **$b_2$** | **1,2,3** | **3** | | $g_2$ | 4 | 1 |
| **$f_1$** | **1,2** | **2** | | $h_2$ | 4 | 1 |
| **$f$** | **1,2,3,4** | **4** | | | | |

algorithm increments the support of item $a$ because it contains its descendant $a_2$. Finally, during the scan of $SID$ 4, values of generalized support of items $a_1$ and $a$ are incremented because $SID$ 4 contains $a_{11}$ ($a_2$, respectively).

- Then the algorithm processes similarly all items of sequences $SID$ 1, 2, 3 and 4. The result of the scan is shown in Table 4.5. If any ancestor of a scanned item is already initiated by the algorithm, its support is not initiated again and it is incremented (i.e. for $a_{11}$).

- Finally, found and counted 1-sequences are filtered by support and hierarchically-closeness and added into the result set of sequential patterns $L_1$. Both $\langle c_1 \rangle : 1$ and $\langle c \rangle : 1$ are not frequent and they are not added into $L_1$. Next, the generalized support values of $\langle d_1 \rangle : 2$ and $\langle d \rangle : 2$ are both sufficient and they are frequent. However, the support is same for both sequences where one is the parent of the other. Therefore, the 1-sequence containing the item $d$ is not hierarchically-closed – it is redundant. Only the sequence $\langle d_1 \rangle : 2$ is added into the result set of sequential patterns $L_1$. Further, the generalized support values of items of $a$-taxonomy: $a_{11} : 2$, $a_{12} : 2$, $a_1 : 3$, $a_2 : 2$ and $a : 4$ differ among ancestors and their descendants. Therefore, all sequences $\langle a_{11} \rangle : 2$ , $\langle a_{12} \rangle : 2$ , $\langle a_1 \rangle : 3$, $\langle a_2 \rangle : 2$, $\langle a \rangle : 4$ are hierarchically-closed multi-level sequential patterns and they are all added into $L_1$.

As a result, the set of hierarchically-closed multi-level sequential patterns of length 1 is a set of 12 ML-sequences $L_1 = \{\langle a_{11} \rangle : 2, \langle a_{12} \rangle : 2, \langle a_1 \rangle : 3, \langle a_2 \rangle : 2, \langle a \rangle : 4, \langle b_1 \rangle : 2, \langle b_2 \rangle : 3, \langle b \rangle : 4, \langle d_1 \rangle : 2, \langle f_1 \rangle : 2, \langle f_2 \rangle : 2, \langle f \rangle : 4\}$.

**Phase 2**

Because $L_1$ is not empty, the algorithm continues by generating ML-sequences of length 2. The algorithm takes all pairs $s_1 \in L_1, s_2 \in L_1$. The algorithm is demonstrated on only several pairs. Notice, that the example is ordered by cases, not by real processing order. The real processing order is by MLSP algorithm: candidate generation step and counting step (and hierarchically-closed sequences filtering sub-step).

- **Candidate Generation Step:** Pair $s_1 = \langle d_1 \rangle, s_2 = \langle d_1 \rangle$ – the algorithm creates one candidate ML-sequence $s' = \langle d_1 d_1 \rangle$, the MLSP Apriori Rule is met and the sequence is added into $C_2$. The second ML-sequence $s'' = \langle (d_1 d_1) \rangle$ has not allowed form because of the duplicate occurrence of on item in one element.
**Counting Step:** During the counting step of $C_2$, the support is counted for the ML-sequence $s'$ and for all its ML-sequence ancestors, concretely, it is generalized to $s'_{g1} = \langle d\, d \rangle$. Moreover, the redundant base is set: $\mathcal{RB}_{\langle d\, d \rangle}[\langle d_1 d_1 \rangle] = 1$ because $\langle d_1 d_1 \rangle$ is contained in *SID* 1 and, so, $\langle d\, d \rangle$. The support of both ML-sequences is 1: $\langle d_1 d_1 \rangle : 1, \langle d\, d \rangle : 1$. ML-sequences $\langle d_1 d_1 \rangle : 1, \langle d\, d \rangle : 1$ are not frequent and are removed.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1 \rangle, s_2 = \langle a \rangle$ – it is not possible to directly *join* ML-sequences $s_1$and $s_2$ because $level(s_1) \neq level(s_2)$. Therefore, $s_1$ must be generalized to $s'_1 = \langle d \rangle$. Then two ML-sequences are created by *join* $s' = \langle d\ a \rangle$ and $s'' = \langle (a\ d) \rangle$ (notice that the items in the element are lexicographically reordered). The MLSP Apriori Rule for both ML-sequences is met and both are added into the set of candidate ML-sequences $C_2$.
**Counting Step:** The support values for ML-sequences are counted: $s' = \langle da \rangle : 2$, $s'' = \langle (ad) \rangle : 1$. The ML-sequence $s'$ is supported by sequences *SID* 1,4. The generalized support of $gen\_supp(s') = 2$ is sufficient. The redundant base of $s'$ contains two indexes:$\mathcal{RB}_{\langle d\, a \rangle}[\langle d_1 a_1 \rangle] = 1$,$\mathcal{RB}_{\langle d\, a \rangle}\{\langle d_1 a_2 \rangle] = 1$. The redundant base does not violate the hierarchical-closeness and $s'$ is added into $L_2$. The ML-sequence $s''$ is supported only by sequence *SID* 1, the minimal support threshold condition is not met and the candidate ML-sequence is removed.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1 \rangle, s_2 = \langle a_1 \rangle$ – this case is closely related to the previous one. The join creates two candidate ML-sequences $s' = \langle d_1\, a_1 \rangle$ and $s'' = \langle (a_1\, d_1) \rangle$ and they are added into $C_2$.
**Counting Step:** The support values for ML-sequences are counted: $s' = \langle d_1\, a_1 \rangle : 1$, $s'' = \langle (a_1\, d_1) \rangle : 1$. The ML-sequence $s'$ has not sufficient support and, therefore, it is generalized to $s'_{g1} = \langle d\, a \rangle$. It will not be added to $L_2$ because it has already been added by the previous join pair.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1 \rangle, s_2 = \langle b_1 \rangle$ – the pair is joined into two candidate ML-sequences $s' = \langle d_1\, b_1 \rangle$ and $s'' = \langle (b_1\, d_1) \rangle$ and they are added into $C_2$.
**Counting Step:** The support values for ML-sequences are counted: $s' = \langle d_1\, b_1 \rangle : 2$ , $s'' = \langle (b_1\, d_1) \rangle : 0$. The ML-sequence $s' = \langle d_1\, b_1 \rangle : 2$ is frequent

and hierarchically-closed (the redundant base is empty). Therefore, the $s'$ is sequential pattern and it is added into $L_2$. Notice that during the counting step, the redundant base is counted for all ML-sequence parents of $s'$. The ML-sequence $s''$ has zero support and is generalized into $s''_{g_1} = \langle(b\,d)\rangle$ whose support is also zero and both candidate ML-sequences are removed.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1\rangle, s_2 = \langle b_2\rangle$ the pair is joined into two candidate ML-sequences $s' = \langle d_1\, b_2\rangle$ and $s'' = \langle(b_2\, d_1)\rangle$ and they are added into $C_2$.
  **Counting Step:** The support values for ML-sequences are counted :$s' = \langle d_1\, b_2\rangle : 1$, $s'' = \langle(b_2\, d_1)\rangle : 0$. The ML-sequence $s' = \langle d_1\, b_2\rangle : 1$ is not frequent and MLSP algorithm tries to do the generalization to $s'_{g_1} = \langle d\, b\rangle$. The generalized support value of the generalized ML-sequence $s'_{g_1}$ is 2 $s'_{g_1} = \langle db\rangle : 2$. Such ML-sequence $s'_{g_1}$ is frequent but the sequence is not hierarchically-closed because the redundant base on index $\langle d_1\, b_1\rangle$ is $\mathcal{RB}_{\langle d\,b\rangle}[\langle d_1 b_1\rangle] = 2$ which is equal to $gen\_supp(s'_{g_1}) = 2$. Therefore, the ML-sequence $s'_{g_1}$ is not added into $L_2$ and it is removed.

The processing of other pairs is similar. The final result set $L_2$ after processing of all pairs is shown in Table 4.2. It contains 14 hierarchically-closed multi-level sequential patterns of length 2.

### Phase 3

The algorithm continues by the third phase taking all pairs $s_1 \in L_2, s_2 \in L_2$ and trying to join them into new hierarchically-closed multi-level sequential patterns of length 3. There are also described only several cases of the running example to demonstrate the algorithm principles. Mainly, the join step differs from the second phase where the *join* was possible in all cases.

- **Candidate Generation Step:** Pair $s_1 = \langle d\; a\rangle, s_2 = \langle d\; a\rangle$ – First, the test if the join step is possible is needed. The part of ML-sequence $s_{x_1} = \langle a\rangle$ and the part of ML-sequence $s_{x_2} = \langle d\rangle$ are not equal and there is no common ML-sequence ancestors of $s_{x_1}$ and $s_{x_2}$. Therefore, the join step is not possible and none new candidate ML-sequence could be generated.

- **Candidate Generation Step:** Pair $s_1 = \langle d\; a\rangle, s_2 = \langle a\; a\rangle$ – The join step is possible for this pair of ML-sequences because $s_{x_1} = \langle a\rangle$ is equal to $s_{x_2} = \langle a\rangle$. The Apriori condition is met for all ML-subsequences and the candidate ML-sequence $s' = \langle d\; a\; a\rangle$is added into $C_3$.
  **Counting Step:** The support for ML-sequence $s'$ is counted $s' = \langle d\; a\; a\rangle : 1$ which indicates that the ML-sequence is not frequent. The candidate sequence cannot be generalized because it is a root ML-sequence and is removed.

- **Candidate Generation Step:** Pair $s_1 = \langle da\rangle, s_2 = \langle(ab)\rangle$ – The pair is joined into the ML-sequence $s' = \langle d\,(a\, b)\rangle$. Because all ML-subsequences are frequent

(namely $\langle d\, b\rangle$) are frequent, the Apriori condition is met. The candidate ML-sequence is added into $C_3$.
**Counting Step:** The ML-sequence $s' = \langle d\ (a\ b)\rangle : 1$ is supported only by sequence $SID$ 1. Here, the generalization is not possible because it is a root ML-sequence and the candidate ML-sequence is removed.

- **Candidate Generation Step:** Pair $s_1 = \langle a\ a\rangle, s_2 = \langle a_1\ d_1\rangle$ – The join step is not directly possible because $s_{x_1} = \langle a\rangle$ is not equal to $s_{x_2} = \langle a_1\rangle$, however, there exists a common ML-sequence ancestor for the second sequence:$\langle a\rangle$ – the second sequence must be generalized by 1 level to ML-sequence $s^{x_2}_{g_1} = \langle a\ d\rangle$. The candidate ML-sequence $s' = \langle a\ a\ d\rangle$ meets MLSP Apriori Rule and it is added into $C_3$.
**Counting Step:** The ML-sequence $s' = \langle a\ a\ d\rangle : 1$ is supported only by sequence $SID$ 1. The candidate ML-sequence is removed.

- **Candidate Generation Step:** Pair $s_1 = \langle(a_1\, b_2)\rangle, s_2 = \langle(b_2\, f_2)\rangle$ – The join step is possible because $s_{x_1} = \langle b_2\rangle$ is equal to $s_{x_2} = \langle b_2\rangle$. New ML-sequence $s' = \langle(a_1\, b_2\, f_2)\rangle$ is created, however, the MLSP Apriori Rule is not met for ML-subsequence: $\langle(a_1\, f_2)\rangle \notin L_2$. Notice that the ML-sequential pattern $\langle(a\, b\, f)\rangle$ is created by another join step by the MLSP algorithm.

- **Candidate Generation Step:** Pair $s_1 = \langle d_1\, b_1\rangle, s_2 = \langle b\, a\rangle$ – The join step is possible after the generalization of the common part to a new candidate ML-sequence $s' = \langle d\, b\, a\rangle$. All ML-subsequences $\langle d\, b\rangle, \langle b\, a\rangle, \langle d\, a\rangle$ are frequent and $s'$ is added into $C_3$.
**Counting Step:** The counting step evaluates the generalized support to $s' = \langle d\, b\, a\rangle$:2 because it is supported by $SID$ 1,4. The redundant base of the ML-sequence is empty. Therefore, the ML-sequence is added into $L_3$.

Finally, the algorithm creates the result set $L_3$ which contains eight hierarchically-closed multi-level sequential patterns. All found sequential patterns are shown in Table 4.2. Note that as ML-sequences get longer, they are much general because it is harder to meet the minimal support threshold.

### Phase 4

Because there were created new sequential patterns during the third phase, the MLSP algorithm continues by the next phase in which it creates ML-sequences of length four.

- **Candidate Generation Step:** Pair $s_1 = \langle a\ (b\ f)\rangle, s_2 = \langle(b\ f)\ a\rangle$ – ML-sequences can be joined into the new candidate ML-sequence $s' = \langle a\ (b\ f)\ a\rangle$. All ML-subsequences of $s'$: $\langle a\ (b\ f)\rangle, \langle(b\ f)\ a\rangle, \langle a\ b\ a\rangle, \langle a\ f\ a\rangle \in L_3$ and the candidate ML-sequence is added into $C_4$.
**Counting Step:** The counting step evaluates the value of the generalized support of the ML-sequence $s' = \langle a\ (b\ f)\ a\rangle : 2$ because it is supported by sequences $SID$ 1,4 – it satisfies the minimal support threshold. The redundant base of the $s'$ is empty and the $s' = \langle a\ (b\ f)\ a\rangle : 2$ is added into $L_4$.

- **Candidate Generation Step:** Pair $s_1 = \langle d\ (b\ f)\rangle, s_2 = \langle (b\ f)\ a\rangle$ – ML-sequences are joined into the new candidate ML-sequence $s' = \langle d\ (b\ f)\ a\rangle$. The rest od the case is the same as the previous one and the $s' = \langle d\ (b\ f)\ a\rangle : 2$ is added into $L_4$.

- All others cases do not lead to ML-sequential patterns of length four.

### Phase 5

In the Phase 4, there were created two hierarchically-closed multi-level sequential patterns. In Phase 5, the MLSP algorithm cannot create any candidate ML-sequence because the patterns cannot be joined. The algorithm finishes.

Result of the running example using the MLSP algorithm is shown in Table 4.2. Totally, it was generated 123 candidate ML-sequences during the mining process. Finally, the algorithm finds 36 hierarchically-closed multi-level sequential patterns.

Scalability of the algorithm and other properties are discussed in the next chapter.

# Chapter 5

# Experimental Evaluation

The issue of mining sequential patterns is generally computationally expensive. If we imagine that the sequence length is the horizontal dimension, then the mining multi-level sequential patterns adds a new vertical dimension over the patterns. The complexity of the problem growths because the algorithm must deal with relations between different multi-level sequences.

This chapter deals with a comparison of different multi-level approaches and algorithms to solve them. The first section of experiments is focused on time comparison of mining different algorithms on synthetic datasets. The advantage of synthetic datasets is the possibility to define specific probabilistic properties. The second section is focused on mining in real-world data. Mining in the real world dataset is an important evaluation because it shows if the algorithms can be used and if revealed results are useful. Commonly used real world testing dataset AdventureWorks [26] by Microsoft is absolutely inappropriate because it does not contain a long-period order history. Therefore, the five year order history of on-line e-shop VOPI [40] is used for the real world evaluation.

## 5.1 Evaluation on Synthetic Datasets

The synthetic dataset allows changing only a specific property of the dataset without changing others if necessary. There was no generator for multi-level sequential patterns. This section describes a generator of multi-level or level-crossing sequence datasets developed by the author of this thesis published in [44].

The time complexity of algorithms is affected mainly by the set of dataset parameters:

- the dataset size,
- the relative or absolute number of sequential patterns,
- the length of sequences,
- the number of taxonomies,
- and the number of levels of taxonomies.

The complete set of parameters of sequence databases with defined taxonomies is shown in Table 5.2. The general methodology of experiments is following. All parameters of generated datasets are fixed except one. Then, the effect of the changes of such dataset or algorithm parameter is evaluated.

Experiments were performed on a PC with CPU i5 3.3GHz, 8GB RAM, OS MS Windows 10. Because there is no algorithm for mining multi-level sequential patterns, results of our algorithms are compared with GSP and PrefixSpan. Authors of the GSP recommended using their algorithm over an extended database for mining sequential patterns with taxonomies. Algorithms GSP and PrefixSpan must use post-processing filtering to get complete set of hierarchically-closed multi-level sequential patterns. All algorithms were implemented in C# on .NET platform using the MS SQL Server database.

The following algorithms are compared in experiments:

- GSP over the database of extended sequences,
- GSP over the database of extended sequences (optimized by Optimization 2, see Section 4.3.4),
- PrefixSpan over the database of extended sequences,
- hGSP – level-crossing sequential patterns,
- MLSP (without Optimization 2, see Section 4.3.4) – hierarchically-closed multi-level sequential patterns,
- MLSP (with Optimization 2, see Section 4.3.4) – hierarchically-closed multi-level sequential patterns.

### 5.1.1 Generating Synthetic Datasets

A well-known generator of synthetic datasets for mining association rules and sequential patterns is *IBM Quest Synthetic Data Generator* [23]. The last available version supports three types of datasets: association rules, multi-level association rules and sequential patterns. Parameters of each type are shown in Table 5.1.

Generator details for association rules datasets were described in [4]. Firstly, the sizes of all transactions in $|C|$ are initialized by picking random numbers with a *Poisson distribution* (see Def. 34) with mean equal to the generator parameter average transaction size $|T|$.

**Definition 34. (Poisson distribution)** The random variable $X$ that equals the number of counts in the interval is a Poisson random variable with parameter $\lambda > 0$ with probability mass function $f(x)$ such that

$$f(x) = \frac{e^{-\lambda}\lambda^x}{x!} \qquad x = 0, 1, 2, \ldots \tag{5.1}$$

Table 5.1: Parameters of IBM Quest Synthetic Data Generator for Association Rules (AR), Sequential Patterns (SP) and Multi-level Association Rules (Tax).

| **Parameter** | **AR** | **SP** | **Tax** |
|---|---|---|---|
| Number of Transactions/Sequences | $\|C\|$ | $\|\mathcal{D}\|$ | $\|C\|$ |
| Number of Transactions | | $\|C\|$ | |
| Avg. size of Transaction | $\|T\|$ | $\|T\|$ | $\|T\|$ |
| Avg. size of potentially freq. itemsets | $\|I\|$ | $\|I\|$ | $\|I\|$ |
| Number of potentially freq. itemsets | $\|L\|$ | $N_I$ | $\|L\|$ |
| Number of items | $N$ | $N$ | $N$ |
| Avg. length of potentially freq. sequences | | $\|S\|$ | |
| Number of potentially freq. sequences | | $N_S$ | |
| Number of roots | | | $\|R\|$ |
| Avg. depth of items in transactions | | | $d$ |

The *Poisson random variable* $X$ with parameter $\lambda$ has the mean and variance equal to $\lambda$, i.e. $E(X) = \lambda$, $D(X) = \lambda$. A random variable $X$ with Poisson distribution with parameter $\lambda$ is denoted as $Po(\lambda)$ [27].

Then, the algorithm generates $|L|$ potentially frequent itemsets $\mathcal{T}$ of average size $|I|$. For each itemset, the probability of occurrence is randomly generated (notice that the sum of probabilities must be equal to 1). Finally, each transaction is filled with a selected subset of itemsets from $\mathcal{T}$. In addition, the algorithm reflects other real-world dataset properties such as corruption of itemsets or inter-itemset similarity. It is impossible to set some important parameters such as expected threshold of minimal support for frequent itemsets. Instead, the support is inversely related to the number of $|L|$. In case of taxonomy generation, the only difference is that items are firstly assigned to taxonomies. The concept of generating association rules datasets described in [4] was extended to generation of sequential patterns datasets in [3].

### 5.1.2 Parameters of the Generator

The list of parameters of our generator is shown in Table 5.2. The parameters differ to parameters of generator presented in [4] and [3] are in bold. The main difference is that our generator allows specifying of average support $Supp_S$ of frequent sequences in the output dataset. Note, that the dataset must have transactions of enough average length otherwise the average support is wrong (e.g. frequent sequences of length $|S|$ cannot have support 60% if the average length of sequences is equal to $|S|$).

Further changes are in parameters of taxonomies. Number of children of each taxonomy node is counted w.r.t. to the parameters $R_h$, $|R|$ and $N$ (details are described below). The levels of items inside frequent sequences and other sequences are managed by parameters $P^S_{ch}$ and $P^I_{ch}$.

### 5.1.3 The method of the generator

The generator works basically in two phases:

Table 5.2: Parameters of our hierarchical sequence generator.

| **Parameter** | **ML-Seq.Patt.** |
|---|---|
| Dataset Size | $\|\mathcal{D}\|$ |
| Avg. number of elements of sequences | $\|C\|$ |
| Avg. size of elements | $\|T\|$ |
| Avg. size of frequent elements | $\|I\|$ |
| Number of items | $N$ |
| Avg. length of (frequent) sequential patterns | $\|S\|$ |
| Number of sequential patterns | $N_S$ |
| **Avg. support of sequential patterns** | $Supp_S$ |
| Number of taxonomies (roots) | $\|R\|$ |
| **Avg. taxonomy height** | $R_h$ |
| **Probability of children (general items)** | $P^I_{ch}$ |
| **Probability of children** (items of freq. sequences) | $P^S_{ch}$ |

1. Preparation of the result model – in this phase, all frequent sequences with the probability of occurrence are prepared.

2. Generating of the output dataset – in this phase, sequences into the output sequence database are generated.

In the following paragraphs, each step of generator is described in detail. First and second step belong to the phase *Preparation of the result model*, third and fourth steps belong to the phase *Generating of the output dataset.*

**1. Create Hierarchies.** The first step of the generator is to create the set of $|R|$ taxonomies and the set of all possible items in the dataset. The generator assigns to each taxonomy in $\tau \in R$ the count of items of the taxonomy $N_{R_i} = Po(\frac{|N|}{|R|})$. The count of all items $N^{rand} = \sum_{R_i \in R} N_{R_i}$ generated by the random generator is probably not equal to the required number of items $N$, therefore the numbers $N_{R_i}$ are normalized from interval $1, \ldots, N^{rand}$ to interval $1, \ldots, N$. Next, the number of children $ch$ of each node is evaluated expecting the tree $R$ is a full $ch$-ary tree. Note, that the total count of nodes in the full $ch$-ary tree of height $R_h$ is $N_{R_i} \approx (ch^{R_h+1} - 1)/(ch - 1)$. Finally, starting with the root node in $R$, a set of $Po(ch)$ children are generated for each tree node recursively.

**2. Generate hierarchical sequential patterns model.** The sequential patters that should be contained in the dataset are prepared in this step. The generator prepares $N_S$ sequences and initializes their lengths by numbers got from a random variable with distribution $Po(|S|)$. Then, elements and items of sequences are generated. Firstly, the generator randomly selects the taxonomy $R_i$. Then the concrete item from $R_i$ is selected randomly by top-down traversing a tree. Next, child-node is selected with probability $P^I_{ch}$ during traversing if any child exists, otherwise actual item is added into the last element of sequence. If the number of items in the actual element reaches the number defined by $Po(|I|)$, the element is added into the sequence and process repeats for each element of sequence. Finally, the probability of selection is generated for each sequential pattern.

**3. Generate the result dataset.** The generator generates $|\mathcal{D}|$ dataset sequences. The length of each dataset sequence is set to a random number $Po(|S|)$. The dataset sequence can contain items of some sequential pattern or a noise (random items). When the sequence is generated, it is decided, whether the next generated item should be noise or next sequential pattern. The probability of generating sequential pattern is related to the support of sequence – if the support is higher, the noise probability should be lower. The probability of noise in our generator is $P_{NOISE} = 1 - \frac{N_S * Supp_S * |S|}{|T|}$. The $1 - P_{NOISE}$ probability ensures reservation of the necessary number of items to reach sufficient support of sequential patterns. The sequential pattern to be generated is selected randomly w.r.t. probabilities of sequential patterns. When the process of generating sequential pattern is started, the items of the sequential pattern are copied into the sequence. If the current selected item has a child, the item is replaced by its child with probability $P^I_{ch}$ or with $P^S_{ch}$ for items of patterns, recursively. This allows generating various hierarchical sequence datasets.

In the following sections describing experiments, the dataset properties are shortened to 6-parameters notation C$x_1$T$x_2$S$x_3$I$x_4$N$x_5$R$x_6$ where$x_1$ is a dataset size, $x_2$ is a length of sequences, $x_3$ is a length of sequential patterns, $x_4$ is average frequent element size, $x_5$ is a number of sequential patterns, $x_6$ is a number of taxonomies.

### 5.1.4 Experiment 1: Dataset Size – Scalability

The first experiment is focused on scalability of the algorithms. The methodology of the experiment is to measure the dependency of execution time on a dataset size. For example, the values of fixed parameters denoted as C4T1.2S3I1.2N15%|Ð|R1k are explained in Table 5.3. The suffix 'k' of a number means that the value is ×1000.

Table 5.3: Dataset Parameters for Experiment 1

| Dataset | $\|C\|$ | $\|T\|$ | $\|S\|$ | $\|I\|$ | $\|N\|$ | $\|R\|$ | $P^I_{ch}, P^S_{ch}$ | $Supp_S$ |
|---|---|---|---|---|---|---|---|---|
| C4T1.2S3I1.2N150kR1k | 4 | 1.2 | 3 | 1.2 | 15% $\|\mathcal{D}\|$ | $1k$ | 0.9 | 0.045 |

The variable is the dataset size. The dataset size is set to different number of sequences $|\mathcal{D}| \in \{100\,000, 250\,000, 500\,000, 750\,000, 1\,000\,000\}$ where each sequence is of the average length 4 – it results in about from 400 000 up to 4 000 000 items in the synthetic datasets. The number of items $|N|$ cannot be set statically but it must be related to $|\mathcal{D}|$ because the small number of items increases their support in the dataset if the dataset naturally grows. Number of frequent sequences $N_S = 5$ with average support is $Supp_S = 0.045$ (4.5 %).

The execution time in seconds was measured for the evaluation. Lower execution time represents better scalability. Results are shown on Figures 5.1 and 5.2. The slowest is basic GSP algorithm. Moreover, for the $|D| = 1\,000\,000$ the run does not finish. Results of GSP can be better using the hash optimization for fast is-generalized-subsequence pre-check. Similarly the time complexity of our algorithm hGSP is computationally hard and is comparable to optimized GSP using the hash is-subsequence check.

Algorithms PrefixSpan and MLSP have better results. PrefixSpan is approximately 7× faster than MLSP without optimizations. However, the MLSP algorithm can be improved using the optimized *Is-generalized-subsequence Check*. The optimized MLSP (with hash is-generalized-subsequence check, denoted by “hash” suffix) is the fastest of all the algorithms. It is in average 4× faster than the second PrefixSpan. Results are shown in Table 5.4. The log scale plot on Figure 5.2 shows that execution times of all algorithms growth exponentially with the dataset size. They differ in the parameter of the exponential function only.

The execution time is related to the number of candidates which are generated during the mining process. Counts of candidate sequences are shown in Table 5.6. The GSP generates a high number of candidates because it generates all combinations for all extended items. It leads to a high number of redundant sequential patterns in the result. A better situation is in case of the hGSP which does not generate all combinations of candidates, however, it still combines a high number of level crossing sequences. Finally, the MLSP reduces the number of candidate sequences and the number of sequential patterns into non-redundant ones only.

Table 5.5 shows the memory requirements of algorithms depending on dataset size. The experiment shows that the highest memory complexity has the PrefixSpan algorithm because of the construction of projected tables. The candidate generating algorithms have approx. 10 times lower memory complexity. It is also shown that the fast hash table increases memory requirements by 60 % in the case of MLSP.

Table 5.4: The dependency of execution times in [s] on the dataset size $|\mathcal{D}|$ (in thousands). The N/A means that the algorithm do not finish for the task.

| $\lvert D\rvert$ | **GSP** | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP** | **MLSP (hash)** |
|---|---|---|---|---|---|---|
| $100k$ | 3753 | 470 | 36 | 933 | 140 | **12** |
| $250k$ | 1514 | 131 | **19** | 1330 | 276 | 24 |
| $500k$ | 10568 | 959 | 145 | 4406 | 849 | **66** |
| $750k$ | 31391 | 6987 | 779 | 3518 | 408 | **58** |
| $100k$ | N/A | 1630 | **197** | N/A | 3190 | 199 |

Table 5.5: The dependency of memory requirements in [MB] on the dataset size.

| $\lvert D\rvert$ | **GSP** | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP** | **MLSP (hash)** |
|---|---|---|---|---|---|---|
| $100k$ | 3 | 4 | 17 | 3 | 1 | 1 |
| $250k$ | 2 | 3 | 41 | 5 | 3 | 3 |
| $500k$ | 3 | 4 | 166 | 9 | 6 | 10 |
| $750k$ | 3 | 4 | 39 | 12 | 6 | 13 |
| $100k$ | N/A | 4 | 87 | N/A | 11 | 12 |

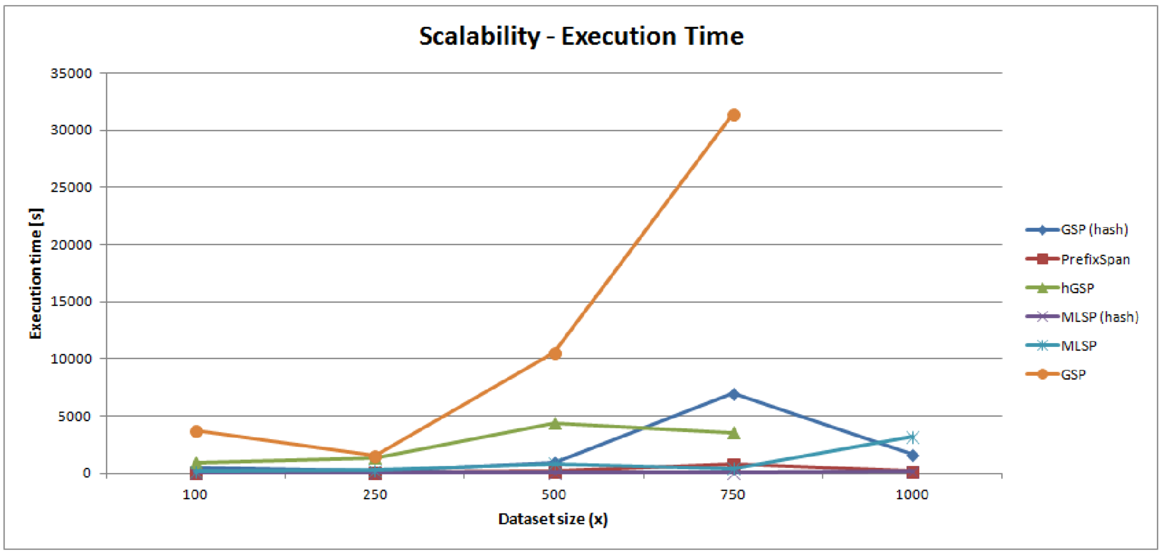

Figure 5.1: Comparison of execution time w.r.t. dataset size *1000.

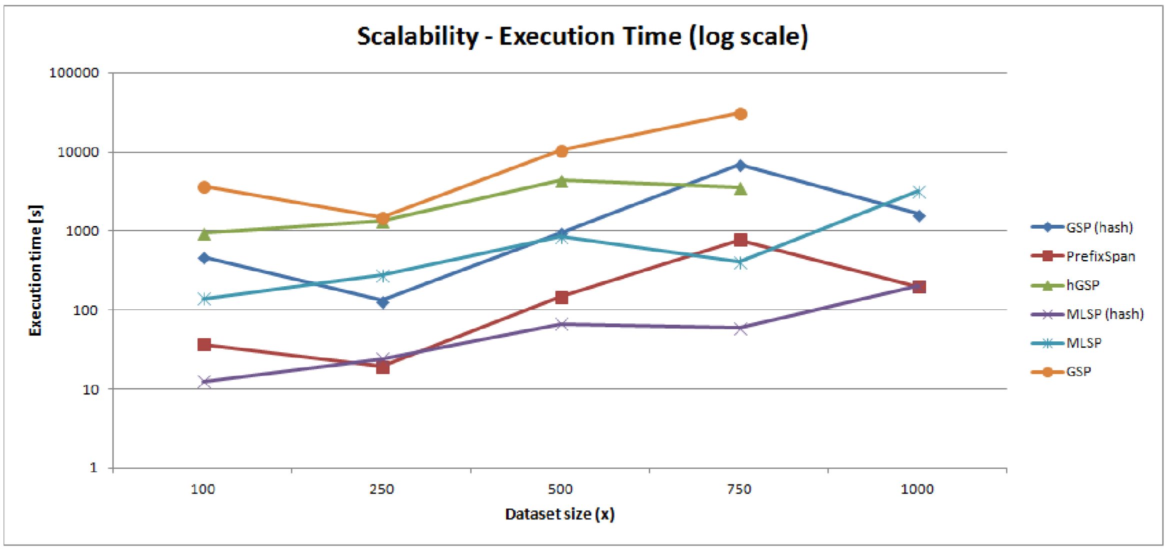

Figure 5.2: Comparison of execution time w.r.t. dataset size *1000 (in logarithmic scale).

Table 5.6: The first part of the table shows the number of candidate sequences generated during the mining process of candidate generation algorithms GSP, hGSP, MLSP). The second part shows the number of sequential patterns. The extended sequences are common for GSP and PrefixSpan. The extended sequences can be pruned into non-redundant hierarchically-closed multi-level sequential patterns.

| | **Candidate Sequences** | | | **Sequential Patterns** | | |
|---|---|---|---|---|---|---|
| $\lvert D\rvert$ | **GSP** | **hGSP** | **MLSP** | **Ext. Seq.** | **L-C Seq.** | **HC ML Seq.** |
| 100$k$ | 11409 | 2535 | **359** | 9119 | 236 | 67 |
| 250$k$ | 2371 | 1927 | **505** | 583 | 102 | 52 |
| 500$k$ | 6548 | 3281 | **660** | 4011 | 281 | 85 |
| 750$k$ | 17466 | 1374 | **258** | 16445 | 313 | 51 |
| 100$k$ | 7569 | N/A | **1343** | 1881 | N/A | 84 |

### 5.1.5 Experiment 2: Changes of Minimal Support Threshold

The Experiment 1 showed that the *Is-generalized-subsequence check* brings important speed-up of the algorithms GSP and MLSP. Therefore, next Experiments uses only optimized variants for both GSP and MLSP. All further experiments were limited by maximal execution time up to 3 600 seconds (1 hour) which should be sufficient according to average execution times of Experiment 1.

In practice, the optimal minimal support threshold is not known on the beginning of the analysis. The optimal minimal support is usually determined experimentally when the data mining starts with the high minimal support threshold value and it is gradually decreased until sequential patterns are found. The decreasing of the minimal support increases the number of generated candidate sequences and sequential patterns while the dataset remains the same.

The setup of this experiment is following. The dataset parameters are C4T1.2S3I1.2N15kR1k, $|\mathcal{D}| = 100\,000$. All algorithms were run with several values of minimal support threshold $min\_supp \in \{0.025, 0.035, 0.045, 0.055, 0.065\}$. Results of this experiment are similar to Experiment 1.

Table 5.7 and Figure 5.3 show that the execution times of PrefixSpan and MLSP algorithms are similar for high values of the minimal support. While the MLSP keeps the stable execution times, the performance of the PrefixSpan gets worse with the decreasing value of the minimal support value parameter. The main reason is that the PrefixSpan must be applied on extended databases to mine multi-level sequential patterns. Therefore, it must analyze high number of sequences and construct large projected databases. Numbers of sequential patterns and candidate sequences are shown in Table 5.8. The GSP and hGSP algorithms are much slower in all cases.

Table 5.7: The dependency of the execution time on the minimal support threshold value.

| $\lvert min\_supp \rvert$ | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP (hash)** |
|---|---|---|---|---|
| 0.025 | 767 | 81 | 1563 | **18** |
| 0.035 | 597 | 43 | 1220 | **18** |
| 0.045 | 470 | 36 | 933 | **12** |
| 0.055 | 158 | **15** | 945 | **15** |
| 0.065 | 160 | **14** | 1051 | 15 |

Table 5.8: Numbers of candidate sequences and numbers of sequential patterns for cases different values of minimal support threshold values.

| | **Candidate Sequences** | | | **Sequential Patterns** | | |
|---|---|---|---|---|---|---|
| $\lvert min\_supp \rvert$ | **GSP** | **hGSP** | **MLSP** | **Ext. Seq.** | **L-C Seq.** | **HC ML Seq.** |
| 0.03 | 12814 | 3128 | 663 | 10142 | 283 | 82 |
| 0.04 | 11471 | 2571 | 539 | 9272 | 245 | 69 |
| 0.05 | 11409 | 2535 | 539 | 9199 | 236 | 67 |
| 0.06 | 3324 | 2008 | 423 | 1553 | 96 | 47 |
| 0.07 | 3324 | 2008 | 423 | 1553 | 96 | 47 |

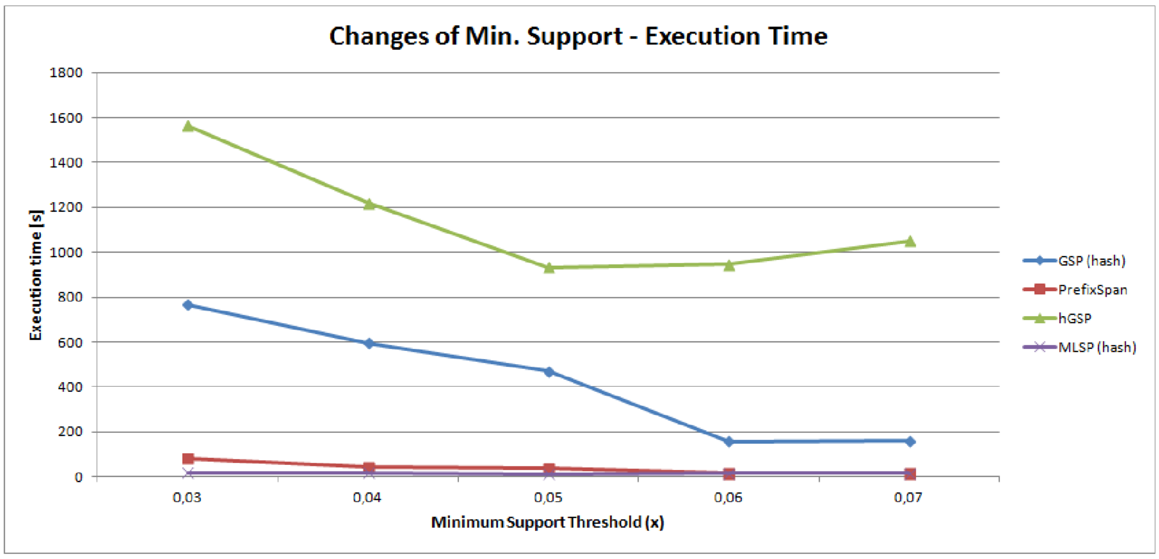

Figure 5.3: Comparison of execution time w.r.t. the minimal support threshold value.

Table 5.9: The dependency of the execution time on the average length $|C|$ of sequences in the database.

| $\lvert C \rvert$ | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP (hash)** |
|---|---|---|---|---|
| 4 | 229 | 11 | 137 | **5** |
| 6 | 633 | 79 | 2660 | **52** |
| 8 | 3253 | **136** | N/A | 333 |
| 10 | 967 | **34** | N/A | 367 |

### 5.1.6 Experiment 3: Length of Sequential Patterns

Next parameter which can affect the performance of the algorithm is the average length of sequential patterns because the length of sequential patterns leads to a higher number of frequent subsequences and candidate sequences during the mining process. The Experiment 3 is focused on the gradually increasing length of the sequential patterns from $|S| \in \{3, 5, 7, 9\}$. Number of elements of sequences in the database $\mathcal{D}$ was determined to value $|C| = 10$. Note that the longer sequences result into higher number of items in the database. Therefore the Experiment 3 is divided into two parts.

First part analyses the dependence of the execution time on the length of sequences in $\mathcal{D}$ because the average sequence length must be at least the average length of sequential patterns. The experiment uses average number of elements of sequences $|C| \in \{4, 6, 8, 10\}$. The fixed dataset parameters are T1.2I1.2N15kR1k, $|\mathcal{D}| = 100\,000$ and $|N_S| = 3$ . The results are shown in Table 5.9. The fastest algorithm on such databases is the PrefixSpan. The MLSP is slower mainly for longer sequences. The reason is that the combinations of candidate sequences which grows massively (there are 116 candidate sequences for $|C| = 4$ and 17250 candidate sequences for $|C| = 10$). Nevertheless the MLSP is up to $10\times$ faster than the GSP algorithm. The hGSP does not finish for longer sequences because of the large search space.

The second part analyses the dependence of execution time on the length of sequential patterns. Experiment results are shown in Table 5.10. The experiment shows the strongest point of the algorithm MLSP. The longer sequential patterns lead to high number of candidate sequences and projected databases of the algorithms based on extended databases. In contrast, the performance of the MLSP is not affected by the length of sequential patterns but only by the number of final sequential patterns (see Table 5.11). Therefore, the MLSP is the only algorithm which is able to finish on all test cases. The other algorithms PrefixSpan and GSP do not finish for the cases $|S| \in \{7, 9\}$ the time limit of one hour.

### 5.1.7 Experiment 4: Number of Sequential Patterns

This experiment analyses the dependence of the execution time on the number of sequential patterns. The experiment setup is following: fixed dataset parameters are $C4$T1.2S3I1.2N15kR1k, $|\mathcal{D}| = 100\,000$, $|N_S| = 3$ and variable is $|N_S| \in \{3, 5, 7, 9, 30\}$.

Table 5.10: The dependency of the execution time on the average length of sequential patterns $|S|$.

| $\|S\|$ | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP (hash)** |
|---|---|---|---|---|
| 3 | 967 | **34** | N/A | 367 |
| 5 | N/A | 1229 | N/A | **301** |
| 7 | N/A | N/A | N/A | **487** |
| 9 | N/A | N/A | N/A | **254** |

Table 5.11: The dependency of the number of candidate sequences and sequential patterns on the average length of sequential patterns $|S|$.

| | **Candidate Sequences** | | | **Sequential Patterns** | | |
|---|---|---|---|---|---|---|
| $\|N_S\|$ | **GSP** | **hGSP** | **MLSP** | **Ext. Seq.** | **L-C Seq.** | **HC ML Seq.** |
| 3 | 39743 | N/A | 17250 | 1742 | N/A | 177 |
| 5 | N/A | N/A | 16268 | N/A | N/A | 335 |
| 7 | N/A | N/A | 16883 | N/A | N/A | 560 |
| 9 | N/A | N/A | 4893 | N/A | N/A | 577 |

The results of experiment are shown in Table 5.12 and the numbers of generated pattern are shown in next Table 5.13. The best results of the experiment are achieved by the PrefixSpan algorithm. The MLSP algorithm gives also satisfactory results. GSP and hGSP algorithms provide by the order of magnitude worse results.

Finally, the experiment also tests the behavior of the algorithms when number of sequential patterns is higher $|N_S| = 30$. In that case the results are similar to previous cases. Therefore, we can say, that the number of sequential patterns does not negatively affect the execution time of the algorithms, especially examined MLSP and hGSP algorithms.

Table 5.12: The dependency of the execution time on the total count of sequential patterns $|N_S|$.

| $\|N_S\|$ | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP (hash)** |
|---|---|---|---|---|
| 3 | 201 | 9 | 94 | **5** |
| 5 | 8 | **1** | 82 | 3 |
| 7 | 17 | **1** | 219 | 5 |
| 9 | 217 | **11** | 893 | 16 |
| 30 | 57 | **1** | 889 | 16 |

Table 5.13: The dependency of the number of candidate sequences and sequential patterns on the total count of sequential patterns $|N_S|$.

| | **Candidate Sequences** | | | **Sequential Patterns** | | |
|---|---|---|---|---|---|---|
| $\|N_S\|$ | **GSP** | **hGSP** | **MLSP** | **Ext. Seq.** | **L-C Seq.** | **HC ML Seq.** |
| 3 | 4393 | 503 | 116 | 4100 | 203 | 34 |
| 5 | 648 | 579 | 263 | 104 | 30 | 23 |
| 7 | 1306 | 1230 | 365 | 129 | 41 | 33 |
| 9 | 6777 | 3837 | 907 | 3179 | 180 | 72 |
| 30 | 5618 | 5533 | 1420 | 203 | 61 | 61 |

Table 5.14: The dependency of the execution time on the number of taxonomy levels $|R_h|$

| $\|R_h\|$ | **GSP (hash)** | **Prefix Span** | **hGSP** | **MLSP (hash)** |
|---|---|---|---|---|
| 2 | 4.1 | **0.5** | 35.6 | 2.7 |
| 6 | 4.7 | **0.7** | 34.4 | 2.2 |

### 5.1.8 Experiment 5: Taxonomy Height

The last experiment on the synthetic dataset analyses the dependency of the execution time on the average taxonomy height (the total number of levels of all taxonomies). The fixed parameters of the experiment are $C$4T1.2S3I1.2N15kR1k, $|\mathcal{D}| = 100\,000$, $|N_S| = 3$. The variable parameter is $|R_h| \in \{2, 6\}$. The first case of the average height 2 shows algorithms behavior on low item categorizations. Second case is run on dataset with the taxonomies of average height 6. The results are shown in Table 5.14. It is shown that the height of taxonomies do not affect the execution time and the complexity of the run of the algorithms. The results are the same for the both test cases.

### 5.1.9 Experiments Summary

Experiments on synthetic datasets compare algorithms GSP, PrefixSpan, hGSP and MLSP. Best results are given by the algorithms MLSP and PrefixSpan. The other algorithms are over a magnitude worse. The performance of the PrefixSpan is significantly slower for mining long sequential patterns. Only the algorithm MLSP finishes all the experiments and it proved very good results for mining hierarchically-closed multi-level sequential patterns.

.

## 5.2 Evaluation on Real-World Datasets

The previous experiments verified the behavior of algorithm MLSP however they did not deal with usability on a real world dataset. The real world data are much more suitable to test the usability. The dataset of orders history of the e-shop VOPI is used for the experiment. First, the dataset is described from general and statistical points of view. Second, the sequential patterns obtained by the MLSP algorithm are discussed. Note, that some kind of data were anonymized or marked as N/P (not presentable).

### 5.2.1 Dataset Description

The real world mining task modeled in this experiment is to analyze the fidelity of customers of the e-shop depending on the kinds of ordered products. In other words, the task is to mine the sequential patterns of ordered items (or their categories) of the same customer. For example, we expect sequential patterns like $\langle a\, b \rangle$ one, which means that customers who buy product $a$, return in future and buy product $b$.

The dataset statistical properties are shown in Table 5.15. The total number of different ordered items is $|N| = 13000$. The items are assorted to totally 32 different root categories. The tree taxonomies in the problem are formed from the main categories, their sub-categories and the products as leafs. The average taxonomies height is $R_h = 4$. The very important parameter is $|C| = 1.13$, which means the average number of transactions of a customer. From one point of view, such number is low, however, it reflects the real highly competitive business environment. The average transaction size $|T| = 1.39$ means that customers usually order more than one product by one order.

The product catalogue contains two main kinds of products: 1. car carpets / accessories and 2. home carpets. The products for cars are tailored for each model of a car. For example, the items for Audi are different from ones for Skoda. Therefore, the product catalogue for car accessories is very large.

Table 5.15: The Real-World dataset parameters.

| **Dataset** | $\lvert\mathcal{D}\rvert$ | $\lvert C\rvert$ | $\lvert T\rvert$ | $\lvert S\rvert$ | $\lvert I\rvert$ | $\lvert N\rvert$ | $\lvert R\rvert$ | $R_h$ | $\lvert N_S\rvert$ |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{D}$ | N/P | 1.13 | 1.39 | N/A | N/A | 13000 | 32 | 4 | N/A |

### 5.2.2 Mining Results

The mining process was tested by MLSP algorithm in two runs with the values of minimal support threshold $min\_supp \in \{0.07\,\%, 0.10\,\%\}$. Numbers of mined sequential patterns of different lengths are shown in Table 5.16. The main point is that the support of sequential patterns of leaf products is very low. Sequential patterns are formed on higher taxonomy levels. Therefore, mining without taxonomies would not probably discover any interesting sequential patterns.

Table 5.16: Counts of sequential patterns of different length depending on minimal support value.

| | **Number of sequential patterns** | | | |
|---|---|---|---|---|
| *min_supp* | $length = 1$ | $length = 2$ | $length = 3$ | $length = 4$ |
| 0.07 % | 448 | $\underline{85}$ | 11 | 1 |
| 0.10 % | 334 | 58 | 7 | 0 |

Selected important sequential patterns are shown in Table 5.17. Arrows in Table show whole path from root to the found item in multi-level sequential patterns, for example, $a \to b \to c$ means that root item $a$ is a parent of $b$ and $b$ is a parent of $c$. Sequential patterns would contain only the item $c$. Following text interprets the mining results:

- Rows 1, 10, 12 show that customers return for *Car Carpets* products.
- Rows 2, 3 are in relation such that the row 2 is the parent of row 3.
- Rows 4, 5, 8, 9, 11 show that the customers return for popular home products.
- Rows 6, 7 show mark-based orders for car carpets.
- Rows 13-16 show orders, where elements contain more than one frequent product. Such patterns were observed for car products. Note that sequences on rows 15 and 16 contain only one element.

It was shown that the MLSP produces new sequential patterns on the real-world datasets. The execution time of the MLSP on the real world dataset is in minutes depending on parameters settings. Therefore, we can say that the algorithm is fully usable for real world data mining problems.

Table 5.17: Examples of mined sequential patterns on real world dataset.

| # | Pattern Length | Support [%] | Element 1 | Element 2 | Element 3 | Element 4 |
|---|---|---|---|---|---|---|
| 1 | 2 | 2.16 | Car Carpets | Car Carpets | | |
| 2 | 2 | 0.30 | Car Carpets | Rubber Car Mats | | |
| 3 | 2 | 0.12 | Car Carpets $\rightarrow$ Tailored Textile Carpets | Rubber Car Mats$\rightarrow$Plastic Car Mats | | |
| 4 | 2 | 0.13 | Indoor Doormats | Indoor Doormats | | |
| 5 | 2 | 0.13 | Outdoor Doormats | Outdoor Doormats | | |
| 6 | 2 | 0.13 | Car Carpets $\rightarrow$ Tailored Textile Carpets$\rightarrow$ VW Carpets | Car Carpets $\rightarrow$ Tailored Textile Carpets$\rightarrow$ VW Carpets | | |
| 7 | 2 | 0.11 | Car Carpets $\rightarrow$ Tailored Textile Carpets$\rightarrow$ Ford Carpets | Car Carpets $\rightarrow$ Tailored Textile Carpets$\rightarrow$ Ford Carpets | | |
| 8 | 2 | 0.76 | Modern Home Carpets | Modern Home Carpets | | |
| 9 | 2 | 0.40 | Modern Home Carpets$\rightarrow$ Eton Carpet | Modern Home Carpets$\rightarrow$ Eton Carpet | | |
| 10 | 3 | 0.25 | Car Carpets | Car Carpets | Car Carpets | |
| 11 | 3 | 0.13 | Modern Home Carpets | Modern Home Carpets | Modern Home Carpets | |
| 12 | 4 | 0.07 | Car Carpets | Car Carpets | Car Carpets | Car Carpets |
| | | | | | | |
| 13 | 3 | 0.11 | (Car Carpets, Plastic Car Mats) | | Car Carpets | |
| 14 | 3 | 0.09 | Plastic Car Mats | (Car Carpets, Plastic Car Mats) | | |
| 15 | 3 | 0.10 | (Car Accessories, Car Covers, Car Carpets) | | | |
| 16 | 3 | 0.16 | (Car Accessories, Car Carpets, Plastic Car Mats) | | | |

# Chapter 6

# Conclusions

Mining sequential patterns, especially mining multi-level sequential patterns, is a challenging task. The main goal of the thesis was to confirm the hypothesis that taxonomies lead to find out new patterns and a new method to mine them effectively can be formulated.

In my research, I focused on two main approaches of dealing with items in taxonomies. The first approach is to find out patterns called level-crossing sequential patterns. A new algorithm called hGSP was proposed but the level-crossing approach came out as extremely time-consuming. The second approach adds some special constraints which simplify the task while keeping important patterns in the result. It leads to the definition of a new type of data mining task called mining hierarchically-closed multi-level sequential patterns. Mining hierarchically-closed multi-level sequential patterns produces results without redundant patterns useful for the analyst. These research results confirm the first part of the thesis hypothesis.

The thesis introduces a new algorithm called MLSP designed for mining hierarchically-closed multi-level sequential patterns. Both the hGSP and MLSP algorithms prefer the generalization of a sequence to dropping it. The performance of the algorithms was evaluated in several experiments. The experiments were focused on comparison of the performance in dependence on the dataset size (scalability), sequential patterns length and size and the taxonomies sizes. The best results are provided by MLSP and PrefixSpan algorithms. The other algorithms were more than over a magnitude slower. It was shown that the average length of sequential patterns has a significant effect on the execution time. The PrefixSpan did not finish for sequences containing 7 and more elements. Only the MLSP algorithm finished in all runs. The usability of the MLSP algorithm was shown on real dataset where the algorithm has found some new useful knowledge. This confirms the second part of the hypothesis related to a new data mining method.

As a result, both parts of the hypothesis were confirmed. Therefore, the thesis hypothesis was completely confirmed and the goal of the thesis was fulfilled.

It was shown that mining hierarchically-closed multi-level sequential patterns is suitable for tasks of the analysis of customer behavior. But there are some other domains where this type of mining task can be useful, for example security analysis of Domain Name System because domains are also organized in taxonomies.

The future work can be focused on the research of other constrains while mining level-crossing or multi-level sequential patterns. In the field of mining methods, research may continue exploring other optimizations.

# Bibliography

[1] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, Mar 1995, pp. 3–14.

[2] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.

[3] R. Agrawal and R. Srikant, "Mining sequential patterns," IBM Research Division. Almaden Research Center. 1994, Tech. Rep.

[4] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[5] A. V. Aho, J. E. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983, 427 p.

[6] M. Atallah, *Algorithms and Theory of Computation Handbook*, ser. Chapman & Hall/CRC Applied Algorithms and Data Structures series. CRC Press, 1998, 950 p.

[7] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 429–435.

[8] R. J. Bayardo, Jr., "Efficiently mining long patterns from databases," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98. New York, NY, USA: ACM, 1998, pp. 85–93.

[9] T. Cormen, *Introduction to Algorithms*. MIT Press, 2009, 1312 p.

[10] M. E. M. Di Beneditto and L. N. de Barros, "Using concept hierarchies in knowledge discovery," in *Advances in Artificial Intelligence–SBIA 2004*. Springer, 2004, pp. 255–265.

[11] P. Fournier-Viger, A. Gomariz, M. Šebek, and M. Hlosta, "Vgen: Fast vertical mining of sequential generator patterns," in *Data Warehousing and Knowledge*

*Discovery*, ser. Lecture Notes in Computer Science, L. Bellatreche and M. Mohania, Eds. Springer International Publishing, 2014, vol. 8646, pp. 476–488.

[12] C. Gao, J. Wang, Y. He, and L. Zhou, "Efficient mining of frequent sequence generators," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 1051–1052.

[13] A. Gomariz, M. Campos, R. Marin, and B. Goethals, "Clasp: An efficient algorithm for mining frequent closed sequences," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, J. Pei, V. Tseng, L. Cao, H. Motoda, and G. Xu, Eds. Springer Berlin Heidelberg, 2013, vol. 7818, pp. 50–61.

[14] K. Gouda and M. Zaki, "Efficiently mining maximal frequent itemsets," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, 2001, pp. 163–170.

[15] K. Gouda and M. Zaki, "Genmax: An efficient algorithm for mining maximal frequent itemsets," *Data Mining and Knowledge Discovery*, vol. 11, no. 3, pp. 223–242, 2005.

[16] J.-W. Han, J. Pei, and X.-F. Yan, "From sequential pattern mining to structured pattern mining: A pattern-growth approach," *Journal of Computer Science and Technology*, vol. 19, no. 3, pp. 257–279, 2004.

[17] J. Han and A. Fu, "Mining multiple-level association rules in large databases," *IEEE Trans. on Knowledge and Data Engineering*, vol. 11, no. 5, pp. 798–805, 1999.

[18] J. Han and Y. Fu, "Discovery of multiple-level association rules from large databases," in *VLDB*, vol. 95, 1995, pp. 420–431.

[19] J. Han and M. Kamber, *Data mining: concepts and techniques*, ser. The Morgan Kaufmann series in data management systems. Elsevier, 2006, 800 p.

[20] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "Freespan: Frequent pattern-projected sequential pattern mining," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 355–359.

[21] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[22] T.-K. Huang, "Developing an efficient knowledge discovering model for mining fuzzy multi-level sequential patterns in sequence databases," in *New Trends in Information and Service Science, 2009. NISS '09. International Conference on*, June 2009, pp. 362–371.

[23] IBM, “IBM quest synthetic data generator,” [cit. 2016-01-15, on-line], 2010. [Online]. Available: http://sourceforge.net/projects/ibmquestdatagen/

[24] D. Lo, S.-C. Khoo, and J. Li, “Mining and ranking generators of sequential patterns.” in *SDM.* SIAM, 2008, pp. 553–564.

[25] D. MacKay, *Information Theory, Inference, and Learning Algorithms.* Cambrifge University Press, 2003, 640 p.

[26] Microsoft, “Microsoft sql server product samples database,” [cit. 2016-01-15, on-line], 2016, published on: http://msftdbprodsamples.codeplex.com/.

[27] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, 4th ed. John Wiley & Sons, May 2006.

[28] S.-I. Nakano, “Efficient generation of plane trees,” *Inf. Process. Lett.*, vol. 84, no. 3, pp. 167–172, nov. 2002.

[29] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, “Discovering frequent closed itemsets for association rules,” in *Database Theory - ICDT’99*, ser. Lecture Notes in Computer Science, C. Beeri and P. Buneman, Eds. Springer Berlin Heidelberg, 1999, vol. 1540, pp. 398–416.

[30] J. Pei, J. Han, R. Mao *et al.*, “Closet: An efficient algorithm for mining frequent closed itemsets.” in *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, vol. 4, no. 2, 2000, pp. 21–30.

[31] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, “Mining sequential patterns by pattern-growth: the prefixspan approach,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, no. 11, pp. 1424–1440, Nov 2004.

[32] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, “Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth,” in *Proc. of the 17th International Conference on Data Engineering.* IEEE Computer Society, 2001, pp. 0215–0215.

[33] G. Piatetsky-Shapiro, “Discovery, analysis and presentation of strong rules,” *Knowledge discovery in databases*, pp. 229–238, 1991.

[34] H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal, “Multi-dimensional sequential pattern mining,” in *Proceedings of the Tenth International Conference on Information and Knowledge Management*, ser. CIKM ’01. New York, NY, USA: ACM, 2001, pp. 81–88.

[35] M. Plantevit, A. Laurent, D. Laurent, M. Teisseire, and Y. W. Choong, “Mining multidimensional and multilevel sequential patterns,” *ACM Trans. Knowl. Discov. Data*, vol. 4, no. 1, pp. 4:1–4:37, Jan. 2010.

[36] M. Plantevit, A. Laurent, and M. Teisseire, "Hype: Mining hierarchical sequential patterns," in *Proceedings of the 9th ACM International Workshop on Data Warehousing and OLAP*, ser. DOLAP '06. New York, NY, USA: ACM, 2006, pp. 19–26.

[37] R. Srikant and R. Agrawal, *Mining sequential patterns: Generalizations and performance improvements.* IBM Research Division, 1996, 29 p., research Report.

[38] R. Srikant and R. Agrawal, "Mining generalized association rules," *Future Generation Computer Systems*, vol. 13, no. 2, pp. 161–180, 1997, data Mining.

[39] R. Srikant, Q. Vu, and R. Agrawal, "Mining association rules with item constraints." in *KDD*, vol. 97, 1997, pp. 67–73.

[40] VOPI, "Vopi.cz," [cit. 2016-01-15, on-line], 2016, published on: http://www.vopi.cz/.

[41] M. Šebek, M. Hlosta, J. Kupčík, J. Zendulka, and T. Hruška, "Multi-level sequence mining based on gsp," in *Proceedings of the Eleventh International Conference on Informatics INFORMATICS'2011*, ser. 1. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2011, pp. 185–190.

[42] M. Šebek, M. Hlosta, J. Kupčík, J. Zendulka, and T. Hruška, "Multi-level sequence mining based on gsp," *Acta Electrotechnica et Informatica*, no. 2, pp. 31–38, 2012.

[43] M. Šebek, M. Hlosta, J. Zendulka, and T. Hruška, "Mlsp: Mining hierarchically-closed multi-level sequential patterns," in *Advanced Data Mining and Applications*, ser. Lecture Notes in Computer Science, H. Motoda, Z. Wu, L. Cao, O. Zaiane, M. Yao, and W. Wang, Eds. Springer Berlin Heidelberg, 2013, vol. 8346, pp. 157–168.

[44] M. Šebek and J. Zendulka, "Generator of synthetic datasets for hierarchical sequential pattern mining evaluation," in *Proceedings of the Twelfth International Conference on Informatics 2013.* The University of Technology Košice, 2013, pp. 289–292.

[45] J. Wang and J. Han, "Bide: efficient mining of frequent closed sequences," in *Data Engineering, 2004. Proceedings. 20th International Conference on*, March 2004, pp. 79–90.

[46] X. Yan, J. Han, and R. Afshar, "Clospan: Mining closed sequential patterns in large datasets," in *In SDM*, 2003, pp. 166–177.

[47] Z. Yang and M. Kitsuregawa, "Lapin-spam: An improved algorithm for mining sequential pattern," in *Data Engineering Workshops, 2005. 21st International Conference on*, April 2005, pp. 1222–1222.

[48] S. Yi, T. Zhao, Y. Zhang, S. Ma, and Z. Che, “An effective algorithm for mining sequential generators,” *Procedia Engineering*, vol. 15, pp. 3653–3657, 2011.

[49] M. J. Zaki and C.-J. Hsiao, “Charm: An efficient algorithm for closed itemset mining.” in *SDM*, vol. 2. SIAM, 2002, pp. 457–473.

[50] M. J. Zaki, “Spade: An efficient algorithm for mining frequent sequences,” *Machine Learning*, vol. 42, no. 1-2, pp. 31–60, 2001.