Efficient Mining of Frequent Closures with Precedence Links and Associated Generators

Laszlo Szathmary, Petko Valtchev, Amedeo Napoli

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Mining of Frequent Closures with Precedence Links and Associated Generators

Laszlo Szathmary — Petko Valtchev — Amedeo Napoli

N° 6657

May 2008

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6657--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche NANCY - GRAND EST

Efficient Mining of Frequent Closures with Precedence Links and Associated Generators

Laszlo Szathmary*†, Petko Valtchev† , Amedeo Napoli*

Thème SYM — Systèmes symboliques
Équipes-Projets Orpailleur

Rapport de recherche n° 6657 — May 2008 — 58 pages

Abstract: The effective construction of many association rule bases require the computation of frequent closures, generators, and precedence links between closures. However, these tasks are rarely combined, and no scalable algorithm exists at present for their joint computation. We propose here a method that solves this challenging problem in two separated steps. First, we introduce a new algorithm called *Touch* for finding frequent closed itemsets (FCIs) and their generators (FGs). *Touch* applies depth-first traversal, and experimental results indicate that this algorithm is highly efficient and outperforms its levelwise competitors. Second, we propose another algorithm called *Snow* for extracting efficiently the precedence from the output of *Touch*. To do so, we apply hypergraph theory. *Snow* is a generic algorithm that can be used with any FCI/FG-miner. The two algorithms, *Touch* and *Snow*, provide a complete solution for constructing iceberg lattices. Furthermore, due to their modular design, parts of the algorithms can also be used independently.

Key-words: algorithm, data mining, itemset search, association rule bases, closed itemsets, generators, concept lattice, iceberg lattice

Szathmary.L@gmail.com, **valtchev.petko@uqam.ca**, **napoli@loria.fr**

* LORIA UMR 7503, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
† Dépt. d'Informatique UQAM, C.P. 8888, Succ. Centre-Ville, Montréal H3C 3P8, Canada

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Une méthode efficace de recherche de motifs fréquents fermés et générateurs et de la relation d'ordre entre fermés

Résumé : En fouille de données, la construction effective de la plupart des bases de règles d'association nécessite le calcul des motifs fermés fréquents, des générateurs et des relations de subsomption entre motifs fermés, ce qui donne d'ailleurs l'ordre du treillis des concepts sous-jacent. Ces trois tâches, qui sont entremêlés, sont pourtant rarement combinées et il n'existe aucun algorithme qui propose une telle combinaison qui soit efficace et qui passe à l'échelle. Nous proposons dans ce rapport une façon de résoudre ce problème important, qui s'appuie sur deux étapes principales. D'abord, nous introduisons un nouvel algorithme appelé *Touch* procède en profondeur, et qui recherche les motifs fermés fréquents (FCIs) et les motifs générateurs. Des résultats expérimentaux montrent que l'algorithme *Touch* est très efficace et qu'il a de très bonnes performances comparé à ses homologues qui procèdent par niveaux. Ensuite, nous proposons l'algorithme appelé *Snow* qui calcule la relation de subsomption entre les motifs fermés produit par *Touch*, en faisant appel à la théorie des hypergraphes. L'algorithme *Snow* est générique et peut être utilisé avec n'importe quel algorithme de recherche de FCIs/FGs. Les deux algorithmes, *Touch* et *Snow*, apportent l'ensemble des éléments nécessaires à la construction des treillis iceberg. En outre, la conception modulaire de ces algorithmes fait qu'ils peuvent être utilisés indépendamment les uns des autres.

Mots-clés : algorithmes pour la fouille de données, motifs fréquents, motifs fermés, motifs générateurs, règles d'association, bases, treillis de concepts, treillis iceberg

Table of Contents

Chapter 1

Frequent Itemsets and Formal Concept Analysis

In this chapter, we present the basic concepts of **(i)** frequent itemset search, and **(ii)** formal concept analysis. The chapter is organized as follows. Section 1.1 presents the basic concepts related to frequent itemset search. Section 1.2 sums up the definitions and properties of formal concept analysis (FCA). Section 1.3 points out the close relation between data mining and formal concept analysis.

1.1 Frequent Itemsets and Frequent Association Rules

Frequent Itemsets. Below we use standard definitions of data mining. We consider a set of *objects* or *transactions* $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$, a set of *attributes* or *items* $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, and a relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$, where $\mathcal{R}(o, a)$ means that the object o has the attribute a. In formal concept analysis [GW99] the triple $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is called a *formal context*. A set of items is called an *itemset* or a *pattern*. Each transaction has a unique identifier (*tid*), and a set of transactions is called a *tidset*.[1] The *length* of an itemset is the cardinality of the itemset, i.e. the number of items included in the itemset. An itemset of length i is called an i-long itemset, or simply an i-itemset[2]. An itemset P is said to be *larger* (resp. *smaller*) than Q if $|P| > |Q|$ (resp. $|P| < |Q|$). We say that an itemset $P \subseteq \mathcal{A}$ is *included* in an object $o \in \mathcal{O}$, if $(o, p) \in \mathcal{R}$ for all $p \in P$. Let f be the function that assigns to each itemset $P \subseteq \mathcal{A}$ the set of all objects that include P: $f(P) = \{o \in \mathcal{O} \mid o \text{ includes } P\}$. The set of objects including the itemset is also known as the *image* of the itemset.[3] The (absolute) *support* of an itemset P indicates how many objects include the itemset, i.e. $supp(P) = |f(P)|$. The support of an itemset P can also be defined in relative value, which corresponds to the proportion of objects including P, with respect to the whole population of objects. An itemset P is called *frequent*, if its support is not less than a given *minimum support* (denoted by *min_supp*), i.e. $supp(P) \geq min\_supp$.

Definition 1.1 (generator) *An itemset G is called* generator *if it has no proper subset H ($H \subset G$) with the same support.*

---

[1] For convenience, we will use separator-free set notations throughout the paper, e.g. AB stands for $\{A, B\}$, 13 stands for $\{1, 3\}$, etc.

[2] For instance, ABE is a 3-itemset.

[3] For instance, in dataset $\mathcal{D}$ (Table 2.1), the image of AB is 23.

Definition 1.2 (closed itemset) *An itemset X is called* closed *if it has no proper superset Y ($X \subset Y$) with the same support.*

The *closure* of an itemset X (denoted by $\gamma(X)$) is the largest superset of X with the same support. Naturally, if $X = \gamma(X)$, then X is closed. The task of frequent (closed) itemset mining consists of generating all (closed) itemsets with supports greater than or equal to a specified *min_supp*.

Equivalence Classes. Two itemsets $P, Q \subseteq \mathcal{A}$ are said to be *equivalent* ($P \cong Q$) iff they belong to the same set of objects (i.e. $\gamma(P) = \gamma(Q)$). From this definition it follows that *equivalent itemsets have the same support values.* The set of itemsets that are equivalent to an itemset P (P's *equivalence class*) is denoted by $[P] = \{Q \subseteq \mathcal{A} \mid P \cong Q\}$. Generators are *minimal elements* in their equivalence classes (w.r.t. set inclusion), i.e. a generator $G \in [G]$ has no proper subset in $[G]$. An equivalence class has at least one generator. Closed itemsets are *maximal elements* in their equivalence classes (w.r.t. set inclusion), i.e. a closed itemset $X \in [X]$ has no proper superset in $[X]$. An equivalence class has exactly one closed itemset, which means that closed itemsets are unique elements in their equivalence classes. If an equivalence class has only one element, then the equivalence class is called *singleton*. The only element of a singleton equivalence class is closed as well as generator.

Frequent Association Rules. An association rule is an expression of the form $P_1 \to P_2$, where P_1 and P_2 are arbitrary itemsets ($P_1, P_2 \subseteq \mathcal{A}$), $P_1 \cap P_2 = \emptyset$ and $P_2 \neq \emptyset$. The left side, P_1 is called *antecedent*, the right side, P_2 is called *consequent*. The support of an association rule r: $P_1 \to P_2$ is defined as: $supp(r) = supp(P_1 \cup P_2)$. An association rule r is called *frequent*, if its support is not less than a given *minimum support* (denoted by *min_supp*), i.e. $supp(r) \geq min\_supp$. The *confidence* of an association rule r: $P_1 \to P_2$ is defined as the conditional probability that an object includes P_2, given that it includes P_1: $conf(r) = supp(P_1 \cup P_2)/supp(P_1)$. An association rule r is called *confident*, if its confidence is not less than a given *minimum confidence* (denoted by min_conf), i.e. $conf(r) \geq min\_conf$. An association rule r with $conf(r) = 1.0$ (i.e. 100%) is an *exact* association rule, otherwise it is an *approximate* association rule. A frequent association rule is *valid* (their set is denoted by $\mathcal{AR}$) if it is both frequent and confident, i.e. $supp(r) \geq min\_supp$ and $conf(r) \geq min\_conf$. The problem of mining frequent association rules in a database $\mathcal{D}$ consists of finding all frequent valid rules in the database.

1.2 Formal Concept Analysis

We describe here the basic notions of FCA [GW99]. Formal concept analysis considers a triple $\mathbb{K} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ called context, where $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is a set of objects, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is a set of attributes, and the binary relation $\mathcal{R}(o, a)$ means that the object o has the attribute a. Two operators, both denoted by $'$, connect the power sets of objects $2^{\mathcal{O}}$ and attributes $2^{\mathcal{A}}$ as follows:

$$' : 2^{\mathcal{O}} \to 2^{\mathcal{A}},\ Z' = \{o \in \mathcal{O} \mid \forall a \in Z,\ \mathcal{R}(o, a)\}$$

The operator $'$ is dually defined on attributes. The pair of $'$ operators induces a Galois connection between $2^{\mathcal{O}}$ and $2^{\mathcal{A}}$. The composition operators $''$ are closure operators: they are idempotent, extensive, and monotonous.

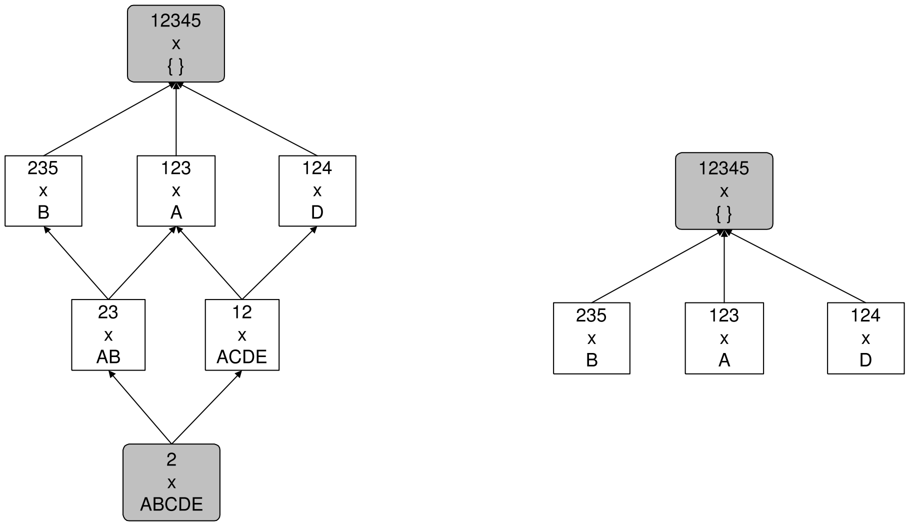

Figure 1.1: The complete concept lattice **(left)** and an iceberg concept lattice **(right)** of the formal context of Table 2.1.

A formal concept of the context $\mathbb{K} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ is a pair $(\mathtt{X}, \mathtt{Y}) \subseteq \mathcal{O} \times \mathcal{A}$, where $\mathtt{X}' = \mathtt{Y}$ and $\mathtt{Y}' = \mathtt{X}$. $\mathtt{X}$ is called *extent*, and $\mathtt{Y}$ is called *intent*. A concept $(\mathtt{X}_1, \mathtt{Y}_1)$ is a subconcept of a concept $(\mathtt{X}_2, \mathtt{Y}_2)$ if $\mathtt{X}_1 \subseteq \mathtt{X}_2$ (or dually $\mathtt{Y}_2 \subseteq \mathtt{Y}_1$) and we write $(\mathtt{X}_1, \mathtt{Y}_1) \leq (\mathtt{X}_2, \mathtt{Y}_2)$. The concept whose extent contains all the objects is called the *top* concept. The concept whose intent contains all the attributes is called the *bottom* concept.

Definition 1.3 (concept lattice) *The set $\mathfrak{B}$ of all concepts of a formal context $\mathbb{K}$ together with the partial order relation $\leq$ forms a lattice and is called the (complete)* concept lattice *of $\mathbb{K}$.*

A concept lattice can be visualized with a Hasse diagram. For instance, the concept lattice associated to the formal context of Table 2.1 is shown in Figure 1.1 (left).

A concept is called frequent if its intent is frequent.

Definition 1.4 (iceberg lattice [STB+02]) *The set of all* frequent *concepts of a context $\mathbb{K}$ together with the partial order relation $\leq$ forms a so-called* iceberg concept lattice *of $\mathbb{K}$.*

Note that an iceberg lattice is an order filter of the complete concept lattice and in general only a join semi-lattice. However, adding a bottom element makes it a lattice again. For instance, the iceberg lattice of the formal context of Table 2.1 by $min\_supp = 3$ is shown in Figure 1.1 (right).

Let $(\mathtt{X}_\mathtt{j}, \mathtt{Y}_\mathtt{j})$ be a subconcept of $(\mathtt{X}_\mathtt{i}, \mathtt{Y}_\mathtt{i})$, i.e. $(\mathtt{X}_\mathtt{j}, \mathtt{Y}_\mathtt{j}) \leq (\mathtt{X}_\mathtt{i}, \mathtt{Y}_\mathtt{i})$. If there is no concept $(\mathtt{X}_\mathtt{k}, \mathtt{Y}_\mathtt{k})$ such that $(\mathtt{X}_\mathtt{j}, \mathtt{Y}_\mathtt{j}) \leq (\mathtt{X}_\mathtt{k}, \mathtt{Y}_\mathtt{k}) \leq (\mathtt{X}_\mathtt{i}, \mathtt{Y}_\mathtt{i})$, then $(\mathtt{X}_\mathtt{i}, \mathtt{Y}_\mathtt{i})$ is said to *cover* $(\mathtt{X}_\mathtt{j}, \mathtt{Y}_\mathtt{j})$ *from above* (and dually, $(\mathtt{X}_\mathtt{j}, \mathtt{Y}_\mathtt{j})$ is said to *cover* $(\mathtt{X}_\mathtt{i}, \mathtt{Y}_\mathtt{i})$ *from below*). For instance, in Figure 1.1 the concept $(123, A)$ covers the concept $(12, ACDE)$ from above (and dually, $(12, ACDE)$ covers $(123, A)$ from below).

Definition 1.5 (upper cover) *The upper cover of a concept C is a set of concepts where each element of the set covers C from above.*

The *lower cover* of a concept is defined dually. For instance, in Figure 1.1 the upper cover of $(12, ACDE)$ is $\{(123, A), (124, D)\}$, and the lower cover of $(123, A)$ is $\{(23, AB), (12, ACDE)\}$.

In other words, the upper cover contains all the "direct parents", and the lower cover contains all the "direct children" of a concept.

When the upper cover (or dually, the lower cover) for each concept is discovered, we say that the *order* (or *covering relation*) is found among the concepts.

1.3 Relation Between Data Mining and Formal Concept Analysis

In this subsection we review the relation between data mining (DM) and formal concept analysis (FCA). The intents of a formal concept lattice are closed itemsets. The intents of an iceberg lattice are *frequent* closed itemsets [STB$^+$02]. For constructing the Hasse diagram of a concept lattice, extents are not necessary, i.e. the order among concepts can be discovered based upon the intents only.

Discovering the set of all intents in FCA is equivalent to the DM problem of finding all closed itemsets by $min\_supp = 0$. Most DM algorithms only concentrate on itemsets, thus the order must be established in an additional step. When $min\_supp > 0$, DM algorithms will extract frequent closed itemsets. Finding the order among them results in an iceberg concept lattice.

As a summary, we can state that the general schema for using DM algorithms for building formal concept lattices (either complete or iceberg) consists of two main steps: **(1)** extracting (frequent) closed itemsets, and then **(2)** finding the order among them.

Chapter 2

Frequent Itemset Search with Vertical Algorithms

In this chapter, we present the common parts of three vertical algorithms namely *Eclat*, *Charm*, and *Talky-G*. The detailed description of these algorithms can be found in Appendix B. This chapter mainly relies on [ZPOL97], [Zak00], and [ZH02].

Overview of Vertical Algorithms

Eclat was the first successful algorithm proposed to generate all frequent itemsets in a depth-first manner. *Charm* is a modification of *Eclat* to explore frequent closed itemsets only. *Talky-G* is an adaptation of *Eclat* and *Charm*. *Talky-G* extracts frequent generators only. All three algorithms use a vertical layout of the database. In this way, the support of an itemset can be easily computed by a simple intersection operation.

Consider the following dataset $\mathcal{D}$ (Table 2.1) that we will use for our examples throughout the paper.

	A	B	C	D	E
1	x		x	x	x
2	x	x	x	x	x
3	x	x			
4				x	
5		x			

Table 2.1: A sample dataset ($\mathcal{D}$) for the examples.

Basic concepts. Here we would like to present the necessary notions specific to *Eclat*, *Charm*, and *Talky-G*, using the terminology of Zaki. Let $\mathcal{I}$ be a set of items, and $\mathcal{D}$ a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. The set of all tids is denoted as $\mathcal{T}$. A set of items is called an *itemset*, and a set of transactions is called a *tidset*. For convenience, we write an itemset $\{A, B, E\}$ as ABE, and a tidset $\{2,3\}$ as 23. For an itemset X, we denote its corresponding tidset as $t(X)$, i.e. the set of all tids that contain X as a subset. For a tidset Y, we denote its corresponding itemset as $i(Y)$, i.e. the set of items

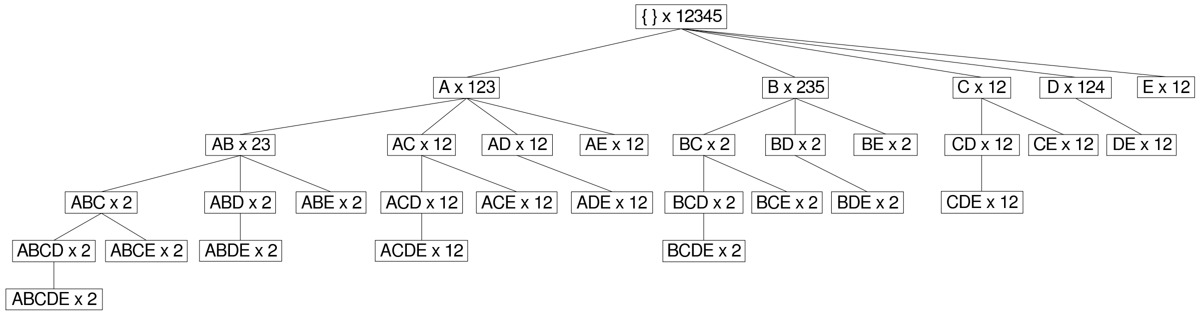

Figure 2.1: IT-tree: Itemset-Tidset search tree of dataset $\mathcal{D}$ (Table 2.1).

common to all the tids in Y. Note that $t(X) = \bigcap_{x \in X} t(x)$, and $i(Y) = \bigcap_{y \in Y} i(y)$. For instance, using our dataset $\mathcal{D}$ (Table 2.1), $t(ABE) = t(A) \cap t(B) \cap t(E) = 1235 \cap 1345 \cap 1345 = 135$ and $i(23) = i(2) \cap i(3) = AC \cap ABCE = AC$. The support of an itemset X is equal to the cardinality of its tid-list, i.e. $supp(X) = |t(X)|$.

Lemma 2.1 *Let X and Y be two itemsets. Then, $X \subseteq Y \Rightarrow t(X) \supseteq t(Y)$.*

Proof. Follows from the definition of support. $\square$

Itemset-tidset search tree and prefix-based equivalence classes. Let $\mathcal{I}$ be the set of items. Define a function $p(X, k) = X[1 : k]$ as the k length prefix of X, and a *prefix-based* equivalence relation θ_k on itemsets as follows: $\forall X, Y \subseteq \mathcal{I}, X \equiv_{\theta_k} Y \iff p(X, k) = p(Y, k)$. That is, two itemsets are in the same k-class if they share a common k-length prefix.

Eclat (resp. *Charm* and *Talky-G*) performs a search for frequent itemsets (resp. frequent closed itemsets and frequent generators) over a so-called IT-tree search space, as shown in Figure 2.1. While most previous methods exploit only the itemset search space, *Eclat*, *Charm*, and *Talky-G* simultaneously explore both the itemset space *and* the transaction space. Each node in the IT-tree, called an IT-node, represented by an itemset-tidset pair, $X \times t(X)$, is in fact a prefix-based equivalence class (or simply a prefix-based class). All the children of a given node X belong to its prefix-based class, since they all share the same prefix X. We denote a prefix-based class as $[P] = \{l_1, l_2, \ldots l_n\}$, where P is the parent node (the prefix), and each l_i is a single item, representing the node $Pl_i \times t(Pl_i)$. For example, the root of the tree corresponds to the class $[\,] = \{A, B, C, D, E\}$. The left-most child of the root consists of the class $[A]$ of all itemsets containing A as the prefix. Each class member represents one child of the parent node. A class represents items that the prefix can be extended with to obtain a new frequent node. Clearly, no subtree of an infrequent prefix has to be examined. The power of the prefix-based class approach is that it breaks the original search space into independent sub-problems. When all direct children of a node X are known, one can treat it as a completely new problem; one can enumerate the itemsets under it and simply prefix them with the item X, and so on.

Lemma 2.1 states that if X is a subset of Y, then the cardinality of the tid-list of Y (i.e. its support) must be less than or equal to the cardinality of the tid-list of X. A practical and important consequence of this lemma is that the cardinalities of intermediate tid-lists shrink as we descend in the IT-tree. This results in very fast intersection and support counting.

Vertical layout. It is necessary to access the dataset in order to determine the support of a collection of itemsets. Itemset mining algorithms work on binary tables, and such a database can be represented by a binary two-dimensional matrix. There are two commonly used layout for the implementation of such a matrix: *horizontal* and *vertical* data layout. Levelwise algorithms use horizontal layout. *Eclat*, *Charm*, and *Talky-G* use instead vertical layout, in which the database consists of a set of items and their tid-lists. To count the support of an itemset X using the horizontal layout, we need one full database pass to test for every transaction T if $X \subseteq T$. For a large collection of itemsets, this can be done at once using the trie data structure. The vertical layout has the advantage that the support of an itemset can be computed by a simple intersection operation. In [Zak00], it is shown that the support of any k-itemset can be determined by intersecting the tid-lists of any two of its $(k-1)$-long subsets. A simple check on the cardinality of the resulting tid-list tells us whether the new itemset is frequent or not. It means that in the IT-tree, only the lexicographically first two subsets at the previous level are required to compute the support of an itemset at any level. One layout can be easily transformed to the other layout on-the-fly (see Appendix C for more details).

Other Optimizations

Element reordering. As pointed out in [Goe03] and [CG05], *Eclat* does not fully exploit the monotonocity property. It generates a candidate itemset based on only two of its subsets, thus the number of candidate itemsets is much larger as compared to breadth-first approaches such as *Apriori*. *Eclat* essentially generates candidate itemsets using only the join step of *Apriori*, since the itemsets necessary for the prune step are not available due to the depth-first search. A technique that is regularly used is to reorder the items in support ascending order, which leads to the generation of less candidates. In *Eclat*, *Charm*, and *Talky-G* such reordering can be performed on the children of a node N when all direct children of N are discovered. Experimental evaluations show that item reordering results in significant performance gains in the case of all three algorithms.

Support count of 2-itemsets. It is well known that many itemsets of length 2 turn out to be infrequent. A naïve implementation for computing the frequent 2-itemsets requires $n(n-1)/2$ intersection operations, where n is the number of frequent 1-items. Considering that 1-items have the largest tid-lists (see Lemma 2.1), these operations are quite expensive. Here we present a method that can be used not only for depth-first, but for breadth-first algorithms too, such as *Apriori*. First, the database must be transformed in horizontal format (see Appendix C). Second, through a database pass on the horizontal layout, an upper-triangular 2D matrix is built containing the support values of 2-itemsets [ZH02] (see Appendix D for a detailed description and an example).

Diffsets for further optimizing memory usage. Recently, Zaki proposed a new approach to efficiently compute the support of an itemset using the vertical data layout [ZG03]. Instead of storing the tidset of a k-itemset P in a node, the difference between the tidset of P and the tidset of the $(k-1)$-prefix of P is stored, denoted by the *diffset* of P. To compute the support of P, we simply need to subtract the cardinality of the diffset from the support of its $(k-1)$-prefix. Support values can be stored in each node as an additional information. The diffset of an itemset $P \cup \{i, j\}$, given the two diffsets of its subsets $P \cup \{i\}$ and $P \cup \{j\}$, with $i < j$, is computed as follows: $\text{diffset}(P \cup \{i, j\}) \leftarrow \text{diffset}(P \cup \{j\}) \setminus \text{diffset}(P \cup \{i\})$. Diffsets also shrink as larger itemsets are found. Diffsets can be used together with the other optimizations presented above. This technique can significantly reduce the size of memory required to store intermediate results. Diffsets can be used for *Eclat*, *Charm*, and *Talky-G* resulting in *dEclat*, *dCharm*, and *dTalky-G*, respectively. Note that we have not used diffsets in our implementations yet.[4]

Conclusion

In this chapter we presented the common parts of *Eclat*, *Charm*, and *Talky-G*. Detailed description of *Eclat* can be found in Appendix B.1. *Charm* is presented in Appendix B.2. *Talky-G* is detailed in Appendix B.3.

[4]We plan to investigate this technique as a future perspective.

Chapter 3

The Touch Algorithm

In this chapter, we introduce a very efficient new algorithm called *Touch* that can find frequent equivalence classes in a dataset. The chapter is organized as follows. First, we present the motivation and contribution of the algorithm. This is followed by the description of the three main features of *Touch*. We then present the pseudo code of the algorithm and give a running example. Next, we provide experimental results for comparing the efficiency of *Touch* to some other algorithms. Finally, we draw conclusions.

3.1 Motivation and Contribution

Finding association rules is one of the most important tasks in data mining. Generating valid association rules from frequent itemsets (FIs) often results in a huge number of rules, which limits their usefulness in real life applications. To solve this problem, different concise representations of association rules have been proposed, e.g. generic basis [BTP$^+$00b], informative basis [BTP$^+$00b], minimal non-redundant association rules [BTP$^+$00b], representative rules [Kry98], Duquennes-Guigues basis [GD86], Luxenburger basis [Lux91], proper basis [PBTL99a], structural basis [PBTL99a], etc. A very good comparative study of these bases can be found in [Kry02], where it is stated that a rule representation should be *lossless* (should enable derivation of all valid rules), *sound* (should forbid derivation of rules that are not valid), and *informative* (should allow determination of rules parameters such as support and confidence).

In this present work, we are more interested in finding minimal non-redundant association rules ($\mathcal{MNR}$). Rules in this set have the following form: $P \to Q \setminus P$, where $P \subset Q$ and P is a *generator* and Q is a *closed itemset*. These rules are particularly interesting because of the following reasons. First, this set of rules is *lossless*, *sound*, and *informative*. In [Kry02], it is shown that with the so-called cover operator, which is an inference mechanism, all valid rules can be restored from these rules with their proper support and confidence values. Second, among rules with the same support and same confidence, these rules contain the most information and these rules can be the most useful in practice [Pas00].

The minimal non-redundant association rules were introduced in [BTP$^+$00b]. In [BTP$^+$00a], Bastide *et al.* presented the *Pascal* algorithm and claimed that $\mathcal{MNR}$ can be extracted with this algorithm. However, to obtain $\mathcal{MNR}$ from the output of *Pascal*, one has to do a lot of computing. First, from the form of $\mathcal{MNR}$ it can be seen that frequent closed itemsets must *also* be known. Second, frequent generators must be *associated* to their closures. Recently we proposed the algorithm *Zart*, an extension of *Pascal*, that gives a solution for these two extra needs [SNK07]. With the output of *Zart*, one can easily construct the set of $\mathcal{MNR}$.

Pascal is very efficient among *levelwise* frequent itemset mining algorithms. This is due to its pattern counting inference mechanism that can significantly reduce the number of expensive database passes. In [SNK07] we showed that *Zart*, in spite of its additional features, is almost as efficient as *Pascal*. Furthermore, as it was argued in [Sza06], the idea introduced in *Zart* can be generalized, and thus it can be applied to *any* frequent itemset mining algorithm.

However, *Zart* has a drawback too. It traverses the whole set of frequent itemsets in order to filter generators and closed itemsets. It can be a serious waste in dense, highly correlated datasets, in which the number of FCIs and the number of FGs is usually *much less* than the number of FIs. That is, if we only want the frequent equivalence classes with their minimal and maximal elements (FGs and FCIs), it is no good enumerating the whole set of FIs.

Here we present an algorithm that gives an elegant solution for this problem. The algorithm *Touch* reduces the search space to frequent generators and frequent closed itemsets *only*. Then, we use a novel method for associating the frequent generators to their closures, thus the algorithm produces exactly the same output as *Zart*. Experimental results show that our new method is very efficient, especially on dense, highly correlated datasets.

3.2 Main Features of Touch

Touch has three main features, namely **(1)** extracting frequent closed itemsets, **(2)** extracting frequent generators, and **(3)** associating frequent generators to their closures, i.e. identifying frequent equivalence classes.

3.2.1 Finding Frequent Closed Itemsets

This part of *Touch* relies on *Charm*. *Charm* is a vertical algorithm that can find FCIs very efficiently in a database. *Charm* traverses the IT-tree in a depth-first manner in a pre-order way, from left-to-right. As detailed in Appendix B.2, *Charm* collects FCIs in the main memory in a special hash data structure. This data structure is taken over from *Charm* by *Touch*. That is, the hash structure containing all FCIs is one of the two inputs of *Touch*.

Charm builds this hash structure for filtering non-closed itemsets. When a new candidate IT-node is created (let us call it *current* node), *Charm* checks if a proper superset with the same support is already stored in the hash. If yes, then the current node is not closed. *Charm* performs this test the following way. First, it assigns a hash key to the current node. It takes the sum of the tids in the tidset, and then it calculates the modulo of the sum with the size of the hash table. The resulting hash key gives a position in the hash table, where a list of closed itemsets is stored. Finally, *Charm* checks if this list includes a proper superset of the current node with the same support. For a running example, please consult Appendix B.2. The hash structure of *Charm* has the following important property:

Property 3.1 *If the hash key of an itemset is calculated as above (i.e. taking the sum of the tids in the tidset of the itemset, and calculating the modulo of the sum with the size of the hash table), then itemsets with the same image are to be stored in the same slot of the hash table.*

Example. Let us see the left part of Figure 3.1 (*Charm*) that depicts the hash structure of FCIs extracted from dataset $\mathcal{D}$ (Table 2.1). For this example, the size of the hash table is set to four. To demonstrate Property 3.1, let us investigate the itemsets $ABCDE$ and BD (see also Figure B.2). The two itemsets are included in transaction 2 (see also Table 2.1). *Charm* finds $ABCDE$ first. Its hash key is 2, thus it is stored in the hash structure at position 2. Later,

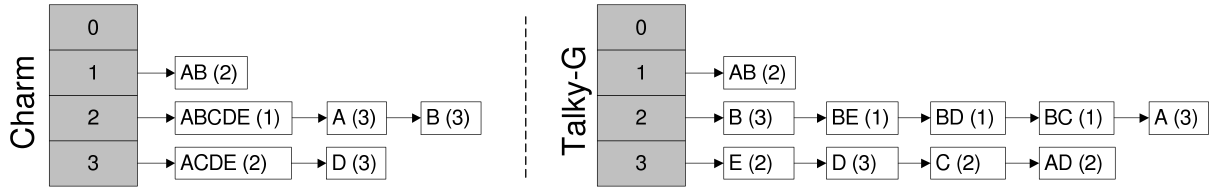

Figure 3.1: Hash tables for dataset $\mathcal{D}$ (Table 2.1) by $min\_supp = 1$. **Left:** hash table of *Charm* containing all FCIs. **Right:** hash table of *Talky-G* containing all FGs.

in another branch of the IT-tree, *Charm* finds BD, whose hash key is also 2 by Property 3.1. *Charm* checks the list of itemsets at position 2 in the hash structure, but since the list already includes a proper superset of BD with the same support ($ABCDE$), BD is not inserted in the hash since it it not closed.

3.2.2 Finding Frequent Generators

This part of *Touch* relies on *Talky-G*. *Talky-G* is a vertical algorithm that can find FGs very efficiently in a database. *Talky-G* traverses the IT-tree in a depth-first manner in a reverse pre-order way, from right-to-left. As detailed in Appendix B.3, this traversal has the advantage that when an itemset X is found, all subsets of X are treated *before* X itself. *Talky-G* collects FGs in the main memory in a special hash data structure. This data structure is taken over from *Talky-G* by *Touch*. That is, the hash structure containing all FGs is the second input of *Touch*.

Talky-G builds this hash structure for filtering non-generator itemsets. When a candidate IT-node is created (let us call it *current* node), *Talky-G* checks if a proper subset with the same support is already stored in the hash. If yes, then the current node is not generator. *Talky-G* performs this test the following way. First, it assigns a hash key to the current node. It takes the sum of the tids in the tidset, and then it calculates the modulo of the sum with the size of the hash table. The resulting hash key gives a position in the hash table, where a list of generators is stored. Finally, *Talky-G* checks if this list includes a proper subset of the current node with the same support. For a running example, please consult Appendix B.3.

Since *Talky-G* calculates the hash key of an itemset exactly the same way as *Charm* does (see Section 3.2.1), Property 3.1 *also holds* for *Talky-G*. That it, itemsets having the same image are to be stored in the same slot of the hash table.

Example. Let us see the right part of Figure 3.1 (*Talky-G*) that depicts the final state of the hash structure of FGs extracted from dataset $\mathcal{D}$ (Table 2.1). For this example, the size of the hash table is set to four. To demonstrate that Property 3.1 is also true for *Talky-G*, let us investigate the itemsets BD and ABD (see also Figure B.5). The two itemsets are included in transaction 2 (see also Table 2.1). *Talky-G* finds BD first because of the reverse pre-order strategy. Its hash key is 2, thus it is stored in the hash structure at position 2. Later, in another branch of the IT-tree, *Talky-G* finds ABD, whose hash key is also 2 by Property 3.1. *Talky-G* checks the list of itemsets at position 2 in the hash structure, but since the list already includes a proper subset of ABD with the same support (BD), ABD is not inserted in the hash since it it not generator.

Algorithm 1 (Touch):

Description: finds frequent equivalence classes

```
hashFCI ←(call Charm and get its hash data structure);   // see Appendix B.2
hashFG ←(call Talky-G and get its hash data structure);   // see Appendix B.3
loop over the FCIs in hashFCI (c)
{
    i ←(index position of c);
    c.generators ← ∅;   // empty set
    loop over the list of FGs in hashFG at index i (g)
    {
        if (c.support = g.support) {
            if g is a subset of c, then
                c.generators ← c.generators ∪ {g};   // g is a generator of c
        }
    }
}
```

3.2.3 Associating Frequent Generators to Their Closures

In the previous two steps (Sections 3.2.1 and 3.2.2) we managed to extract FCIs and FGs. There is one more step to do namely associating frequent generators to their closures, i.e. identifying the equivalence classes. If we had two simple sets of FCIs and FGs, the following naïve method could be used for instance. Enumerate all FCIs, and find their FG subsets with the same support. Unfortunately, this approach is very expensive.

In our method, instead of simple sets, we take over FCIs and FGs in a special hash data structure. The reason is that we use these hash structures for the association process. The idea is the following:

Property 3.2 *If both* Charm *and* Talky-G *use the* same size *for the hash tables (i.e., same number of slots), and if both algorithms use the same image-based hashing method (i.e., sum of tids in the tidset modulo with the size of the hash table), then from Property 3.1 and from the definition of equivalence classes it follows that a frequent closed itemset and its generators are in different tables but they are at the same index position.*

Example. Let us see Figure 3.1 that depicts the two inputs of *Touch*, i.e. the hash structure of *Charm* and the hash structure of *Talky-G*. Say we want to determine the generators of the closed itemset $ACDE$. $ACDE$ is stored at position 3 in the hash structure of *Charm*. By Property 3.2, its generators are stored in the list at position 3 in the other hash structure, i.e. in the hash structure of *Talky-G*. In this list, there are three itemsets that are subsets of $ACDE$ and that have the same support values: E, C, and AD. It means that these three itemsets are the generators of $ACDE$.

FCI	support	associated FG(s)
AB	2	AB
$ABCDE$	1	BE; BD; BC
A	3	A
B	3	B
$ACDE$	2	E; C; AD
D	3	D

Table 3.1: Output of *Touch* on dataset $\mathcal{D}$ by $min\_supp = 1$.

3.3 The Algorithm

3.3.1 Pseudo Code

The pseudo code of *Touch* is given in Algorithm 1. Line 1 corresponds to Section 3.2.1. Line 2 is explained in Section 3.2.2. The block between lines 3 and 14 is detailed in Section 3.2.3.

3.3.2 Running Example

Here we demonstrate the execution of *Touch* on dataset $\mathcal{D}$ with $min\_supp = 1$ (20%). First, *Charm* and *Talky-G* are called on $\mathcal{D}$ with the given min_supp value. As a result, they return FCIs and FGs in two hash structures, as shown in Figure 3.1. Then, *Touch* associates generators to their closures the following way. *Processing AB.* The frequent closed itemset AB has one subset with the same support in the other hash structure at index 1 (AB). This means that AB is a closed itemset *and* a generator at the same time, i.e. the equivalence class of AB has one element only (singleton equivalence class). *Processing ABCDE.* The frequent closed itemset $ABCDE$ has three generators: BE, BD, and BC. Etc. The output of *Touch* is shown in Table 3.1. Recall that due to the property of equivalence classes, the support of a generator is equal to the support of its closure.

3.4 Experimental Results

We evaluated *Touch* against *Zart* [SNK07] and *A-Close* [PBTL99b]. The algorithms were implemented in Java in the Coron data mining platform [SN05].[5] The experiments were carried out on a bi-processor Intel Quad Core Xeon 2.33 GHz machine running under Ubuntu GNU/Linux operating system with 4 GB of RAM. For the experiments we have used the following datasets: T20I6D100K, C20D10K, and Mushrooms. The T20I6D100K[6] is a sparse dataset, constructed according to the properties of market basket data that are typical weakly correlated data. The C20D10K is a census dataset from the PUMS sample file, while the Mushrooms[7] describes mushrooms characteristics. The last two are highly correlated datasets.

Table 3.2 contains detailed information about the execution of *Touch*. The first three columns correspond to the three main steps of *Touch* namely **(1)** getting FCIs using *Charm*, **(2)** getting FGs using *Talky-G*, and **(3)** associating FGs to their closures. Column 4 indicates the total execution time of the algorithm including input and output. In the sparse dataset T20I6D100K, almost all frequent itemsets are closed and generators at the same time. It means that most

[5] `http://coron.loria.fr`

[6] `http://www.almaden.ibm.com/software/quest/Resources/`

[7] `http://kdd.ics.uci.edu/`

min_supp	execution time (sec.) get FCIs (*Charm*)	get FGs (*Talky-G*)	associate FCIs and FGs	**total time** (with I/O)	# FCIs	# FGs	(# FIs)	$\frac{\#FCIs}{\#FIs}$	$\frac{\#FGs}{\#FIs}$
T20I6D100K									
1%	19.07	2.16	0.03	22.76	1,534	1,534	1,534	100.00%	100.00%
0.75%	24.06	2.65	0.05	28.32	4,710	4,710	4,710	100.00%	100.00%
0.5%	35.21	5.01	0.14	42.45	26,208	26,305	26,836	97.66%	98.02%
0.25%	94.59	20.71	0.50	121.60	149,217	149,447	155,163	96.17%	96.32%
C20D10K									
30%	0.20	0.29	0.02	1.06	951	967	5,319	17.88%	18.18%
20%	0.34	0.41	0.03	1.42	2,519	2,671	20,239	12.45%	13.20%
10%	0.71	0.70	0.07	2.27	8,777	9,331	89,883	9.76%	10.38%
5%	1.13	1.06	0.11	3.37	21,213	23,051	352,611	6.02%	6.54%
MUSHROOMS									
30%	0.12	0.21	0.02	0.82	425	544	2,587	16.43%	21.03%
20%	0.19	0.27	0.02	0.98	1,169	1,704	53,337	2.19%	3.19%
10%	0.43	0.46	0.04	1.57	4,850	7,585	600,817	0.81%	1.26%
5%	0.80	0.81	0.08	2.53	12,789	21,391	4,137,547	0.31%	0.52%

Table 3.2: Detailed execution times of *Touch* and other statistics (number of FCIs, number of FGs, number of FIs, proportion of the number of FCIs to the number of FIs, proportion of the number of FGs to the number of FIs). Note that the number of FIs is shown for comparative reasons only. *Touch* does not need to extract all FIs.

equivalence classes are singletons. It is known that *Charm* is less efficient on sparse datasets. The reason is that *Charm* performs four tests on each candidate for reducing the IT-tree. However, in sparse datasets the number of FCIs is almost equivalent to the number of FIs, thus the search space cannot be reduced significantly. Consequently, the four tests give some overhead to the algorithm. *Talky-G* is also less efficient on sparse datasets, but still much faster than *Charm*. It can be due to the less number of tests on a candidate. In dense, highly correlated datasets (C20D10K and MUSHROOMS), both *Charm* and *Talky-G* are very efficient, even at low minimum support values. Since the number of FCIs and FGs is much less than the number of FIs, the two algorithms can take advantage of exploring a much smaller search space. The association of FCIs and FGs is done extremely efficiently in all cases. That is, the association step gives absolutely no overhead to *Touch*.

Table 3.3 contains the experimental evaluation of *Touch* against *Zart* and *A-Close*. All times reported are real, wall clock times as obtained from the Unix *time* command between input and output. We have chosen *Zart* and *A-Close* because they represent two efficient algorithms that produce exactly the same output as *Touch*. *Zart* and *A-Close* are both levelwise algorithms. *Zart* is an extension of *Pascal*, i.e. first it finds all FIs, then it filters FCIs, and finally the algorithm associates FGs to their closures. *A-Close* reduces the search space to FGs only, then it calculates the closure for each generator. Both algorithms have advantages and disadvantages. *Zart*, due to its pattern counting inference, can enumerate FIs very efficiently. As shown in [SNK07], the extra computations of *Zart* give no significant overhead to the algorithm. Thus, when the number of FIs is not too high, *Zart* proves to be quite efficient. *A-Close*, on the other hand, requires much less memory since it reduces the search space to FGs only. This reduction of the memory usage is more spectacular in the case of dense, highly correlated datasets. However, the way *A-Close* computes the closures of generators is very expensive because of the huge number of intersection operations. Thus, in spite of reducing the search space, *A-Close* is less efficient than *Zart* in most cases. *Touch*, just like *A-Close*, reduces the search space to the strict minimum, i.e. it only extracts what it really needs namely the set of FCIs and the set of FGs. Then, *Touch* associates

min_supp	execution time (sec.) Touch	Zart	A-Close
T20I6D100K			
1%	22.76	7.33	31.25
0.75%	28.32	14.96	39.49
0.5%	42.45	45.52	100.60
0.25%	121.60	159.78	285.41
C20D10K			
30%	1.06	8.17	15.78
20%	1.42	15.84	29.88
10%	2.27	36.66	59.41
5%	3.37	75.28	94.18
Mushrooms			
30%	0.82	3.65	7.17
20%	0.98	10.69	15.28
10%	1.57	75.36	36.83
5%	2.53	641.54	63.37

Table 3.3: Response times of *Touch*, compared to *Zart* and *A-Close*.

the two sets in a very efficient way. Since *Touch* is based on *Charm* and *Talky-G*, the algorithm is very efficient on dense, highly correlated datasets. We must admit however that levelwise algorithms like *Zart* are sometimes more suitable for sparse datasets.

3.5 Conclusion

In this chapter we presented a new algorithm called *Touch* for exploring the frequent equivalence classes in a dataset. The algorithm has three main features namely **(1)** extraction of FCIs, **(2)** extraction of FGs, and **(3)** association of FGs to their closures. The algorithm is very competitive because each one of the three steps is solved in a very efficient way. First, FCIs are extracted with *Charm*, and *Charm* is considered to be one of the most efficient FCI-mining algorithms. Second, for the extraction of FGs we propose a new algorithm called *Talky-G*. *Talky-G* concentrates on FGs only in the search space. Third, we found a very efficient solution for associating FCIs and FGs together that gives *no overhead at all* to the algorithm. That is, the association can be done in almost no time. Moreover, the memory footprint of *Touch* is low since, unlike *Zart* for instance, the algorithm does not need to extract all frequent itemsets. Experimental results prove the interest of *Touch*, especially on dense, highly correlated datasets.

To sum up, *Touch* is a very efficient new algorithm that has two original parts: **(1)** the algorithm *Talky-G* for finding FGs in a depth-first manner with a reverse pre-order traversal, and **(2)** the association of FCIs and FGs for identifying frequent equivalence classes.

Chapter 4

The Snow and Snow-Touch Algorithms

In this chapter, we present the *Snow* and *Snow-Touch* algorithms. *Snow* can find order among concepts in a formal lattice, and *Snow-Touch* is a combination of two algorithms namely *Snow* and *Touch*. The chapter is organized as follows. Section 4.1 presents the basic concepts of hypergraphs. This short overview of hypergraph theory is necessary for understanding the *Snow* algorithm that is detailed in Section 4.2. Section 4.2 also includes the pseudo code of *Snow* and *Snow-Touch*. Experimental results of the two algorithms are provided in Section 4.3. Finally, we draw conclusions in Section 4.4.

4.1 Basic Concepts of Hypergraphs

In this subsection we mainly rely on [EG95]. Hypergraph theory [Ber89] is an important field of discrete mathematics with many relevant applications in applied computer science. A hypergraph is a generalization of a graph, where edges can connect arbitrary number of vertices. Formally:

Definition 4.1 (hypergraph) *A* hypergraph *is a pair* $(V,\mathcal{E})$ *of a finite set* $V = \{v_1, v_2, \ldots, v_n\}$ *and a family* $\mathcal{E}$ *of subsets of* V*. The elements of* V *are called vertices, the elements of* $\mathcal{E}$ *edges.*

Note that some authors, e.g. [Ber89], state that the edge-set as well as each edge must be non-empty and that the union of all edges results in the vertex set.

Definition 4.2 (partial hypergraph) *Let* $\mathcal{H} = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m\}$ *be a hypergraph. The partial hypergraph* $\mathcal{H}_i$ *of* $\mathcal{H}$ $(i = 1, \ldots, n)$ *is the hypergraph that contains the first* i *edges of* $\mathcal{H}$*, i.e.* $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\}$.

A hypergraph is *simple* if none of its edges is contained in any other of its edges. Formally:

Definition 4.3 (simple hypergraph) *A hypergraph is called* simple *if it satisfies* $\forall \mathcal{E}_i, \mathcal{E}_j \in \mathcal{E} : \mathcal{E}_i \subseteq \mathcal{E}_j \Rightarrow i = j$.

Example. The hypergraph $\mathcal{H}$ in Figure 4.1 is not simple because the edge $\{a\}$ is contained in the edge $\{a, c, d\}$.

Definition 4.4 *Let* $\mathcal{H} = (V, \mathcal{E})$ *be a hypergraph. Then* $\min(\mathcal{H})$ *denotes the set of minimal edges of* $\mathcal{H}$ *w.r.t. set inclusion, i.e.* $\min(\mathcal{H}) = \{E \in \mathcal{E} \mid \nexists E' \in \mathcal{E} : E' \subset E\}$*, and* $\max(\mathcal{H})$ *denotes the set of maximal edges of* $\mathcal{H}$ *w.r.t. set inclusion, i.e.* $\max(\mathcal{H}) = \{E \in \mathcal{E} \mid \nexists E' \in \mathcal{E} : E' \supset E\}$.

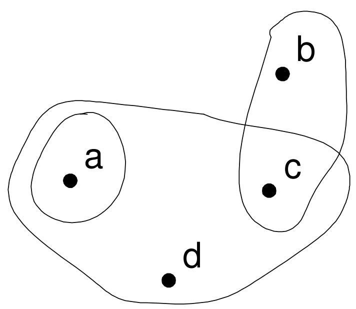

Figure 4.1: A sample hypergraph $\mathcal{H}$, where $V = \{a, b, c, d\}$ and $\mathcal{E} = \{\{a\}, \{b, c\}, \{a, c, d\}\}$.

Clearly, for any hypergraph $\mathcal{H}$, $min(\mathcal{H})$ and $max(\mathcal{H})$ are simple hypergraphs. Moreover, every partial hypergraph of a simple hypergraph is simple, too.

EXAMPLE. In the case of hypergraph $\mathcal{H}$ in Figure 4.1, $\min(\mathcal{H}) = \{\{a\}, \{b, c\}\}$ and $\max(\mathcal{H}) = \{\{b, c\}, \{a, c, d\}\}$.

The problem that is of high interest for us concerns hypergraph transversals. A transversal of a hypergraph $\mathcal{H}$ is a subset of the vertex set of $\mathcal{H}$ which intersects each edge of $\mathcal{H}$. A transversal is *minimal* if it does not contain any transversal as proper subset. Formally:

Definition 4.5 (transversal) *Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. A set $T \subseteq V$ is called a* transversal *of $\mathcal{H}$ if it meets all edges of $\mathcal{H}$, i.e. $\forall E \in \mathcal{E} : T \cap E \neq \emptyset$. A transversal T is called* minimal *if no proper subset T' of T is a transversal.*

Note that Pfaltz and Jamison call transversal (resp. minimal transversal) as *blocker* (resp. *minimal blocker*) in [PJ01]. Outside hypergraph theory, a transversal is usually called a *hitting set*.

EXAMPLE. The hypergraph $\mathcal{H}$ in Figure 4.1 has two minimal transversals: $\{a, b\}$ and $\{a, c\}$. For instance, the sets $\{a, b, c\}$ and $\{a, c, d\}$ are transversals but they are not minimal.

Definition 4.6 (transversal hypergraph) *The family of all minimal transversals of $\mathcal{H}$ constitutes a simple hypergraph on V called the* transversal hypergraph *of $\mathcal{H}$, which is denoted by $Tr(\mathcal{H})$.*

EXAMPLE. Considering the hypergraph $\mathcal{H}$ in Figure 4.1, $Tr(\mathcal{H}) = \{\{a, b\}, \{a, c\}\}$. See Appendix E for an algorithm and further examples.

The following propositions capture important relations between a hypergraph and its transversal hypergraph (for proofs see [Ber89]).

Proposition 4.1 *Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. Then $Tr(\mathcal{H})$ is a simple hypergraph, and $Tr(\mathcal{H}) = Tr(min(\mathcal{H}))$.*

Proposition 4.2 *Let $\mathcal{G}$ and $\mathcal{H}$ be two simple hypergraphs. Then $\mathcal{G} = Tr(\mathcal{H})$ if and only if $\mathcal{H} = Tr(\mathcal{G})$.*

Corollary 4.1 *Let $\mathcal{G}$ and $\mathcal{H}$ be two simple hypergraphs. Then $Tr(\mathcal{G}) = Tr(\mathcal{H})$ iff $\mathcal{G} = \mathcal{H}$.*

Corollary 4.2 (duality property) *Let $\mathcal{H}$ be a simple hypergraph. Then $Tr(Tr(\mathcal{H})) = \mathcal{H}$.*

Corollary 4.2 states that calculating the transversal hypergraph $\mathcal{H}'$ of a simple hypergraph $\mathcal{H}$, and calculating once again the transversal hypergraph $\mathcal{H}''$ of $\mathcal{H}'$, we get back the original hypergraph $\mathcal{H}$, i.e. $\mathcal{H}'' = \mathcal{H}$.

Example. Consider the hypergraph $\mathcal{H}$ in Figure 4.1. Since $\mathcal{H}$ is not simple, let $\mathcal{G} = \min(\mathcal{H}) = \{\{a\}, \{b, c\}\}$. Then,

$$\mathcal{G}' = Tr(\mathcal{G}) = Tr(\{\{a\}, \{b, c\}\}) = \{\{a, b\}, \{a, c\}\}$$

$$\mathcal{G}'' = Tr(\mathcal{G}') = Tr(\{\{a, b\}, \{a, c\}\}) = \{\{a\}, \{b, c\}\} \ .$$

That is, $\mathcal{G}'' = \mathcal{G}$. □

4.2 Detailed Description of Snow

The goal of the *Snow* algorithm is to find order among concepts, i.e. provided a set of concepts, construct the corresponding concept lattice. As shown in Section 1.3, the order is uniquely determined by the set of concept intents. In a concept, in addition to its intent, we also need its associated generators (see Def. 1.1). It is important to note that *Snow* does not require concept extents. To give an overview of the algorithm, *Snow* performs the following operations. As input, it takes a set of concepts, where a concept contains the intent and its generators. As output, *Snow* provides a concept lattice, i.e. it finds the order among concepts.

For the examples of *Snow* we will use the concept lattice depicted in Figure 4.2. In a concept, the following details are provided: intent, support of the intent (in parentheses), and below the list of generators of the intent (separated by hashmarks). For instance, the bottom concept's intent is $ABCDE$, and it has three generators namely BC, BD, and BE. Since they belong to the same equivalence class, the support of all these four itemsets is the same, which is 1 in this example.

Now, let us see the theoretical background of *Snow*.

Definition 4.7 (face [Pfa02]) *Let C_1 and C_2 be two concepts where C_2 covers C_1 from above. The difference between the intents of C_1 and C_2 is called a face of C_1.*

A concept has as many faces as the number of concepts in its upper cover.

Example. Let us consider the bottom concept in Figure 4.2. This concept has two faces: $F_1 = ABCDE \setminus AB = CDE$, and $F_2 = ABCDE \setminus ACDE = B$.

A basic property of the generators of a closed itemset X states that they are the minimal blockers of the family of faces associated to X [Pfa02]:

Theorem 4.1 *Assume a closed itemset X and let $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$ be its family of associated faces. Then a set $Z \subseteq X$ is a minimal generator of X iff X is a minimal blocker of $\mathcal{F}$.*

As indicated in Section 4.1, a minimal blocker of a family of sets is an identical notion to a minimal transversal of a hypergraph. This trivially follows from the fact that each hypergraph $(V,\mathcal{E})$ is nothing else than a family of sets drawn from $\wp(V)$. Now following Theorem 4.1,

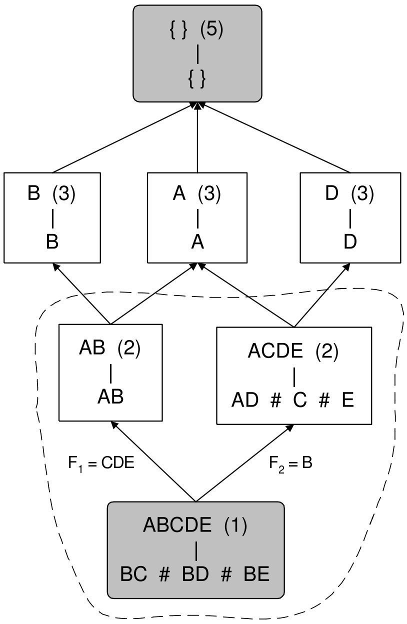

Figure 4.2: Concept lattice of the context of Table 2.1. Concepts are labeled with their generators.

we conclude that given a closed itemset X, the associated generators compose the transversal hypergraph of its family of faces $\mathcal{F}$ seen as the hypergraph $(X,\mathcal{F})$.

Next, further to the basic property of a transversal hypergraph, we conclude that $(X,\mathcal{F})$ is necessarily simple. In order to apply Proposition 4.2, we must also show that the family of generators associated to a closed itemset, say $\mathcal{G}$, forms a simple hypergraph. Yet this holds trivially due to the definition of generators. We can therefore advance that both families represent two mutually corresponding hypergraphs.

Property 4.1 *Let X be a closure and let $\mathcal{G}$ and $\mathcal{F}$ be the family of its generators and the family of its faces, respectively. Then, for the underlying hypergraphs it holds that **(a)** $Tr(X,\mathcal{G}) = (X,\mathcal{F})$ and **(b)** $Tr(X,\mathcal{F}) = (X,\mathcal{G})$.*

The *Snow* algorithm is based on part (a) of Property 4.1, which says that computing all the minimal transversals of the generators of a concept, one obtains the family of faces of the concept. By taking the difference between the concept intent and its family of faces, we get the intents of the concepts in the upper cover. This is the *principal idea* of *Snow*.

Example 4.1 *Let us consider the bottom concept in Figure 4.2 whose intent is $ABCDE$. First, we compute the transversal hypergraph of its generators: $Tr(\{BC, BD, BE\}) = \{CDE, B\}$. This means that the bottom concept has two faces namely $F_1 = CDE$ and $F_2 = B$. The number of faces indicates that the bottom concept is covered by two concepts C_1 and C_2 from above. By Def. 4.7, $intent(C_1) = ABCDE \setminus CDE = AB$, and $intent(C_2) = ABCDE \setminus B = ACDE$. Applying this procedure for all concepts, the order relation among concepts can easily be discovered.* □

Note that part (b) of Property 4.1 can be applied to determine the generators of concepts in a concept lattice (that is, order is known this time). First, the family of faces $\mathcal{F}$ of a concept C must be computed by using Def. 4.7. Then, calculating all the minimal transversals of $\mathcal{F}$ results in the family of generators of C.

4.2.1 The Snow-Touch Algorithm

By *Snow* we refer to the algorithm that finds the order among concepts, i.e. the input is a set of concepts[8] and the output is a concept lattice (either complete or iceberg). By *Snow-Touch* we refer to our *complete* solution as described in this paper. In addition to *Snow*, *Snow-Touch* also includes the call of *Touch* for extracting the set of concepts, i.e. the input of *Snow-Touch* is a context and a minimum support value, and the output is a concept lattice (either complete or iceberg).

4.2.2 Pseudo Code

The pseudo code of *Snow* (lines 3–12) and *Snow-Touch* (lines 1–12) is given in Algorithm 2.

Algorithm 2 (Snow and Snow-Touch):

Description (Snow): build a concept lattice from a set of concepts (lines 3–12)
Description (Snow-Touch): build a concept lattice from a context (lines 1–12)

```
 1) setOfItemsets ← getItemsets(context, min_supp);   // only for Snow-Touch
 2) setOfConcepts ← convertToConcepts(setOfItemsets);   // only for Snow-Touch
 3) identifyOrCreateTopConcept(setOfConcepts);   // Snow starts here
 4) loop over the elements of setOfConcepts (c)   // find the upper cover for each concept
 5) {
 6)     setOfFaces ← getMinTransversals(c.generators);
 7)     intentsInUpperCover ← getIntentsInUpperCover(c.intent, setOfFaces);
 8)     upperCover ← getCorrespondingConcepts(intentsInUpperCover);
 9)     loop over the concepts in upperCover (p) {
10)         createLink(c, p);
11)     }
12) }
```

`getItemsets` function: this method calls an itemset mining algorithm that returns (frequent) closed itemsets *and* their associated generators. In our implementation we used the *Touch* algorithm for this purpose (see Chapter 3 for more details). Table 3.1 depicts a sample output of *Touch*. This function is specific for *Snow-Touch*.

`convertToConcepts` function: this method converts the set of itemsets obtained from *Touch* to a set of concepts. A sample output of *Touch* is shown in Table 3.1. A concept is an object with the following properties: intent, support, list of generators, list of references to concepts in the upper cover, list of references to concepts in the lower cover. The lists of references are initialized empty; the other properties are provided by *Touch*. This function is specific for *Snow-Touch*.

`identifyOrCreateTopConcept` procedure: this method tries to identify the top concept in the set of concepts. If it fails to find it, then it creates the top concept. The top concept is somewhat special, that is why it requires special attention. By definition, the intent of the top concept is the itemset that is present in all the objects of the input dataset, i.e. whose support is 100%. If there is no full column in the input context, then this itemset is the empty set, because by definition its support is 100%. The empty set is *always* a generator. If the input context has a closed

[8]Recall that concept intents are labeled by their generators.

itemset X (different from the empty set) with support 100%, then X is the closure of the empty set; otherwise the empty set is a closed itemset. If the closure of the empty set is itself (like this is the case in Figure 4.2), then it is considered to be a trivial case, thus data mining algorithms often omit the empty set in their output (like *Touch* for instance in Table 3.1). However, the top concept is important in the case of concept lattices, thus the `identifyOrCreateTopConcept` procedure performs the following task. It enumerates the concepts looking for a concept whose support is 100%. If it does not find such a concept, then it creates a concept whose intent and generator is the empty set. The support value of this newly created concept is set to 100% (see Figure 4.2 for an example).

`getMinTransversals` function: this method computes the transversal hypergraph of a given hypergraph. More precisely, given the family of generators of a concept c, the function returns the family of faces of c (by Property 4.1). For a detailed description of this function, see Algorithm 9 in Appendix E.

`getIntentsInUpperCover` function: this method calculates the differences between the intent of the current concept c and the family of faces of c (by Def. 4.7). The function returns a set of itemsets that are the intents of the concepts that form the upper cover of c.

`getCorrespondingConcepts` function: this method gets a set of intents as input, and it returns their concept objects. This correspondence can be easily done with a hash table where intents are keys and concept objects are values.

`createLink` procedure: this method links together the current concept and its upper cover. Links are established with references in both directions.

For a running example, see Example 4.1.

4.3 Experimental Results

Experiments were carried out on three different databases, namely T20I6D100K, C20D10K and Mushrooms. Detailed description of these datasets and the test environment can be found in Appendix A. It must be noted that T20I6D100K is a sparse, weakly correlated dataset imitating market basket data, while the other two datasets are dense and highly correlated.

Response time of Snow. The first two columns of Table 4.1 contain the following information: **(1)** number of concepts, and **(2)** execution time of *Snow* for finding the order among the concepts. Recall that the execution time of *Snow* does not include the extraction of concepts, i.e. we suppose that the set of concepts is already given.

As can be seen, *Snow* is able to discover the order very efficiently in sparse as well as in dense datasets. That is, the efficiency of *Snow* is independent of the density of the input dataset. This is due to the following reason. Looking at Algorithm 2, we can see that *Snow* only has one expensive step namely the computation of all minimal transversals of a given hypergraph (line 6). As seen before, *Snow* considers the family of generators of a concept as a hypergraph. Thus, the response time of *Snow* mainly depends on how efficiently it can compute the transversal hypergraphs. In Appendix E, we presented an optimized version of Berge's original algorithm called *BergeOpt*. *BergeOpt* is very efficient when the input hypergraph does not contain too many edges. Figure 4.3, 4.4, and 4.5 indicate the distribution of hypergraph sizes in the three different datasets.[9] These figures show that most hypergraphs only have 1 edge, which is a trivial

[9]For instance, the dataset T20I6D100K by $min\_supp = 0.25\%$ contains 149,019 1-edged hypergraphs, 171 2-edged hypergraphs, 25 3-edged hypergraphs, 0 4-edged hypergraphs, 1 5-edged hypergraph, and 1 6-edged

min_supp	# concepts (including top)	***Snow*** **(finding order)**	total time (*Snow-Touch* with I/O)
T20I6D100K			
1%	1,535	0.04	22.80
0.75%	4,711	0.11	29.02
0.5%	26,209	0.36	44.75
0.25%	149,218	3.24	137.06
C20D10K			
30%	951	0.03	1.22
20%	2,519	0.07	1.67
10%	8,777	0.29	3.07
5%	21,213	0.54	4.93
Mushrooms			
30%	425	0.02	0.87
20%	1,169	0.05	1.17
10%	4,850	0.17	1.15
5%	12,789	0.47	2.26

Table 4.1: Response times of *Snow* and *Snow-Touch*.

case for *BergeOpt*, and large hypergraphs are rare. As a consequence, *BergeOpt*, and thus *Snow* can work very efficiently. Note that we obtained similar hypergraph-size distributions on two further datasets namely T25I10D10K and C73D10K.

Response time of Snow-Touch. The last column of Table 4.1 contains the execution time of the *Snow-Touch* algorithm. These response times include the following operations: reading the input dataset; extracting the set of concepts with *Touch*; finding order among concepts with *Snow*; writing the description of the lattice in a file.

Table 4.1 shows that the efficiency of *Snow* is similar on both sparse and dense datasets. As seen in Chapter 3, *Touch* is very efficient on dense, highly correlated datasets, and performs well on sparse datasets too. From this it follows that *Snow-Touch* represents a very efficient solution, especially on dense datasets.

4.4 Conclusion

In this paper, we presented a *complete* and *efficient* solution for constructing a concept lattice from a given context. With our approach, one can build not only complete, but iceberg lattices too. In addition to most lattice-building algorithms, our method labels concept intents with their generators, thus the resulting concept lattice can easily be used to generate interesting association rules for instance.

Our combined and complete algorithm *Snow-Touch* is a combination of two algorithms namely *Snow* and *Touch*. *Touch* is a new algorithm that extracts frequent equivalence classes, i.e. frequent closed itemsets and their associated generators. This way *Touch* provides the concepts of the corresponding formal lattice. The second algorithm, *Snow*, can find the order

hypergraph.

among the previously found concepts using hypergraph theory. Experimental results show that the combination of the two algorithms gives a very efficient solution.

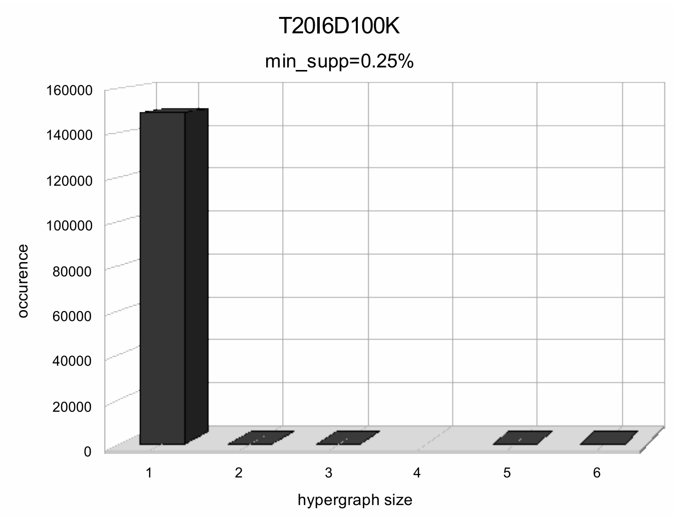

Figure 4.3: Distribution of hypergraph sizes for T20I6D100K.

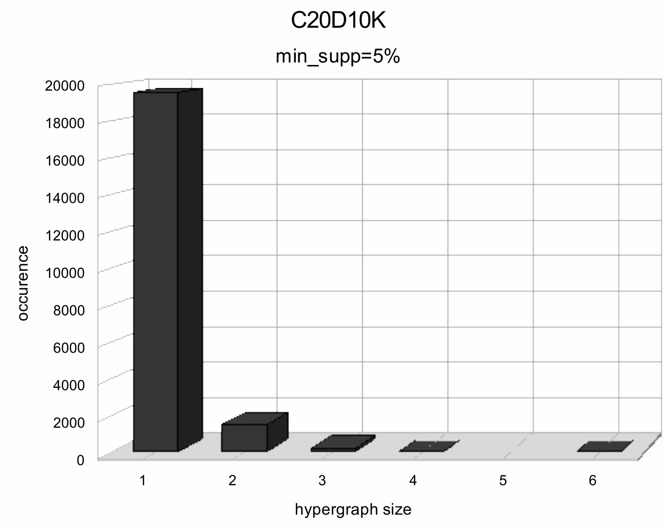

Figure 4.4: Distribution of hypergraph sizes for C20D10K.

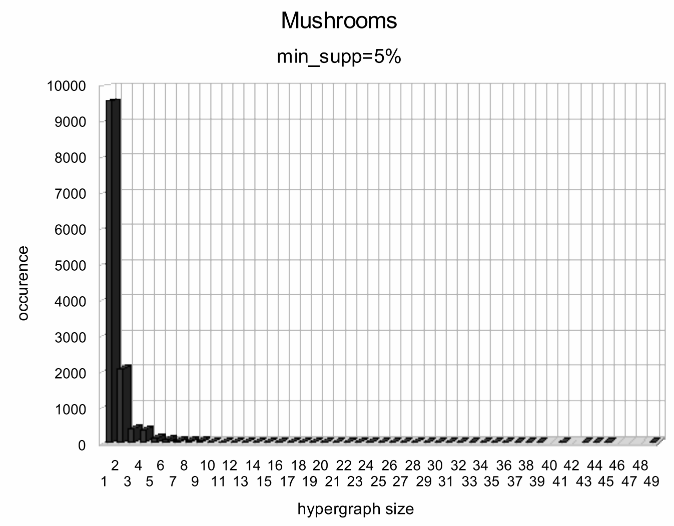

Figure 4.5: Distribution of hypergraph sizes for Mushrooms.

Appendix A

Test Environment

Test Platform

All the experiments in this paper were carried out on the same bi-processor Intel Quad Core Xeon 2.33 GHz machine running under Ubuntu GNU/Linux operating system with 4 GB of RAM. All the experiments were carried out with the Coron system, which is a domain and platform independent, multi-purposed data mining toolkit implemented entirely in Java.[10] All times reported are real, wall clock times, as obtained from the Unix *time* command between input and output. Time values are given in seconds.

Test Databases

For testing and comparing the algorithms, we chose several publicly available real and synthetic databases to work with. Table A.1 shows the characteristics of these datasets, i.e. the name and size of the database, the number of transactions, the number of different attributes, the average transaction length, and the largest attribute in each database.

database name	database size (bytes)	# of transactions	# of attributes	# of attributes in average	largest attribute
T20I6D100K	7,833,397	100,000	893	20	1,000
T25I10D10K	970,890	10,000	929	25	1,000
C20D10K	800,020	10,000	192	20	385
C73D10K	3,205,889	10,000	1,592	73	2,177
Mushrooms	603,513	8,416	119	23	128

Table A.1: Database characteristics.

The T20I6D100K and T25I10D10K[11] are sparse datasets, constructed according to the properties of market basket data that are typically sparse, weakly correlated data. The number of frequent itemsets is small, and nearly all the FIs are closed. The C20D10K and C73D10K are census datasets from the PUMS sample file, while the Mushrooms[12] describes the characteristics of various species of mushrooms. The latter three are dense, highly correlated datasets. Weakly correlated data, such as synthetic data, constitute easy cases for the algorithms that extract frequent itemsets, since few itemsets are frequent. On the contrary, correlated data constitute

[10] `http://coron.loria.fr`

[11] `http://www.almaden.ibm.com/software/quest/Resources/`

[12] `http://kdd.ics.uci.edu/`

far more difficult cases for the extraction due to the large number of frequent itemsets. Such data are typical of real-life datasets.

Appendix B

Detailed Description of Vertical Algorithms

B.1 Eclat

This appendix is based on Chapter 2, where we presented *Eclat* [ZPOL97, Zak00] in a general way. As seen, *Eclat* is a vertical algorithm that traverses a so-called itemset-tidset search tree (IT-tree) in a depth-first manner in a pre-order way, from left-to-right. The IT search tree of dataset $\mathcal{D}$ (Table 2.1) is depicted in Figure 2.1. The goal of *Eclat* is to find all frequent itemsets in this search tree. *Eclat* processes the input dataset in a vertical way, i.e. it associates to each attribute its tidset pair. Here we present the algorithm in detail through an example.

The Algorithm

Eclat uses a special data structure for storing frequent itemsets called IT-search tree. This structure is composed of IT-nodes. An IT-node is an itemset-tidset pair, where an itemset is a set of items, and a tidset is a set of transaction identifiers. That is, an IT-node shows us which transactions (or objects) include the given itemset.

Pseudo code. The main block of the algorithm[13] is given in Algorithm 3. First, the IT-tree is initialized, which includes the following steps: the root node, representing the empty set, is created. By definition, the support of the empty set is equal to the number of transactions in the dataset (100%). *Eclat* transforms the layout of the dataset in vertical format, and inserts under the root node all frequent attributes. After this the dataset itself can be deleted from the main memory since it is not needed anymore. Then we call the `extend` procedure recursively for each child of the root. At the end, all frequent itemsets are discovered in the IT-tree.

`addChild` procedure: this method inserts an IT-node under the current node.

`save` procedure: this procedure has an IT-node as its parameter. This is the method that is responsible for processing the itemset. It can be implemented in different ways, e.g. by simply printing the itemset and its support value to the standard output, or by saving the itemset in a file, in a database, etc.

`delete` procedure: this method deletes a node from the IT-tree, i.e. it removes the reference on the node from its parent, and frees the memory that is occupied by the node.

[13] Note that the main block of Charm is exactly the same.

`getCandidate` function: this function has two nodes as its parameters (*curr* and *other*). The function creates a new candidate node, i.e. it takes the union of the itemsets of the two nodes, and it calculates the intersection of the tidsets of the two nodes. If the support of the candidate is less than the minimum support, it returns "null", otherwise it returns the candidate as an IT-node. In Chapter 2 we presented an optimization method for the support count of 2-itemsets. This technique can be used here: if the itemset part of *curr* and *other* consists of one attribute only, then the union of their itemsets is a 2-itemset. In this case, instead of taking the intersection of their tidsets, we consult the upper-triangular matrix to get its support. Naturally, this matrix had been built before in the initialization phase.

`sortChildren` procedure: this procedure gets an IT-node as parameter. The method sorts the children of the given node in ascending order by their support values. This step is highly recommended since it results in a much less number of non-frequent candidates (see also Chapter 2).

Algorithm 3 (main block of Eclat & Charm):

```
root.itemset ← ∅;   // root is an IT-node whose itemset is empty
root.tidset ← {all transaction IDs};   // the empty set is present in every transaction
root.support ← |O|;   // where |O| is the total number of objects in the input dataset
root.parent ← null;   // the root has no parent node
loop over the vertical representation of the dataset (attr) {
    if (attr.supp ≥ min_supp) then root.addChild(attr);
}
delete the vertical representation of the dataset;   // free memory, not needed anymore
sortChildren(root);   // optimization, results in a less number of non-frequent candidates

while root has children
{
    child ← (first child of root);
    extend(child);
    save(child);   // process the itemset
    delete(child);   // free memory, not needed anymore
}
```

Algorithm 4 ("extend" procedure of Eclat):

Method: extend an IT-node recursively (discover FIs in its subtree)
Input: *curr* – an IT-node whose subtree is to be discovered

```
loop over the "brothers" (other children of its parent) of curr (other)
{
    candidate ← getCandidate(curr, other);
    if (candidate ≠ null) then curr.addChild(candidate);
}

sortChildren(curr);   // optimization, results in a less number of non-frequent candidates

while curr has children
{
    child ← (first child of curr);
    extend(child);
    save(child);   // process the itemset
    delete(child);   // free memory, not needed anymore
}
```

Running example. The execution of *Eclat* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 2$ (40%) is illustrated in Figure B.1. The execution order is indicated on the left side of the nodes in circles. For the sake of easier understanding, the element reordering optimization is not applied.

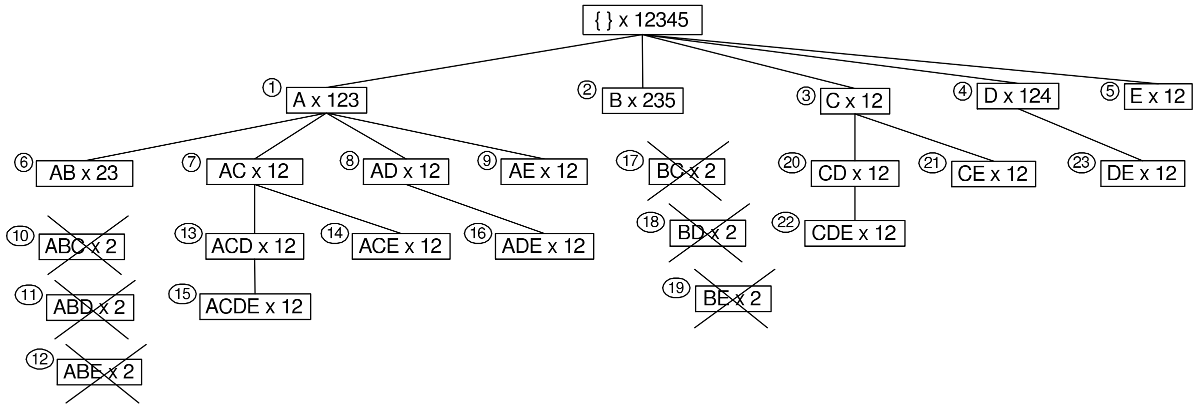

Figure B.1: Execution of *Eclat* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 2$ (40%).

The algorithm first initializes the IT-tree with the root node, which is the smallest itemset, the empty set, that is present in each transaction, thus its support is 100%. Using the vertical representation of the dataset, frequent attributes with their tidsets are added directly under the root. The children of the root node are extended recursively one by one in a pre-order way, from left-to-right. Let us see the prefix-based equivalence class of attribute A. This class includes all frequent itemsets that have A as their prefix. 2-long supersets of A are formed by using the "brother" nodes of A (nodes that are children of the parent of A, i.e. B, C, D, and E). As AB, AC, AD, and AE are all frequent itemsets, they are added under A. The *extend* procedure is now called recursively on AB. It turns out that none of its 3-long supersets are frequent, thus ABC, ABD, and ABE are discarded. Then, *extend* is called on AC. Its 3-long supersets are frequent, thus they are added in the IT-tree. Extending ACD results in another frequent itemset, $ACDE$. Finally, *extend* is called on AD and ADE is added in the IT-tree. With this, the subtree of A is completely discovered. After processing the nodes, this subtree can be deleted from main memory. Extension of nodes continues with B, etc. When the algorithm stops, all frequent itemsets are discovered.

Conclusion

We presented here the frequent itemset mining algorithm *Eclat* that is based on a different approach. *Eclat* is not a levelwise, but a depth-first, vertical algorithm. As such, it makes only one database scan. *Eclat* requires no complicated data structures, like trie, and it uses simple intersection operations to generate candidate itemsets (candidate generation *and* support counting happen in a single step). *Apriori* has been followed by lots of optimizations, extensions. The same is true for *Eclat*. Experimental results show that *Eclat* outperforms levelwise, frequent itemset mining algorithms. It also means that *Eclat* can be used on such datasets that other levelwise algorithms cannot simply handle.

B.2 Charm

This appendix is based on Chapter 2, where we presented the common parts of *Eclat* and *Charm* [ZH02]. *Eclat* was designed to find all frequent itemsets in a dataset. *Charm* is a modification of *Eclat*, allowing one to find frequent *closed* itemsets only. Since *Charm* is based on *Eclat*, reading Appendix B.1 is highly recommended for an easier understanding. *Charm* is a vertical algorithm that traverses the itemset-tidset search tree (IT-tree) in a depth-first manner in a pre-order way, from left-to-right. The IT search tree of dataset $\mathcal{D}$ (Table 2.1) is depicted in Figure 2.1. The goal of *Charm* is to find frequent closed itemsets only in this search tree. *Charm*, just like *Eclat*, processes the input dataset in a vertical way, i.e. it associates to each attribute its tidset pair. Here we present the algorithm in detail. This subsection mainly relies on [ZH02], where the proof of Theorem B.2 can also be found.

Basic Properties of Itemset-Tidset Pairs

There are four basic properties of IT-pairs that *Charm* exploits for efficient exploration of closed itemsets. Assume that we are currently processing a node $P \times t(P)$, where $[P] = \{l_1, l_2, \ldots, l_n\}$ is the prefix class. Let X_i denote the itemset Pl_i, then each member of $[P]$ is an IT-pair $X_i \times t(X_i)$.

Theorem B.2 *Let $X_i \times t(X_i)$ and $X_j \times t(X_j)$ be any two members of a class $[P]$, with $X_i \leq_f X_j$, where f is a total order (e.g. lexicographic or support-based). The following four properties hold:*

1. *If $t(X_i) = t(X_j)$, then $\gamma(X_i) = \gamma(X_j) = \gamma(X_i \cup X_j)$*
2. *If $t(X_i) \subset t(X_j)$, then $\gamma(X_i) \neq \gamma(X_j)$, but $\gamma(X_i) = \gamma(X_i \cup X_j)$*
3. *If $t(X_i) \supset t(X_j)$, then $\gamma(X_i) \neq \gamma(X_j)$, but $\gamma(X_j) = \gamma(X_i \cup X_j)$*
4. *If $t(X_i) \neq t(X_j)$, then $\gamma(X_i) \neq \gamma(X_j) \neq \gamma(X_i \cup X_j)$*

The Algorithm

Pseudo code. The main block of the algorithm is exactly the same as the main block of *Eclat* (see Algorithm 3), thus we do not repeat it here. The difference is in the `extend` procedure (Algorithm 5). While *Eclat* finds all frequent itemsets in the subtree of a node, *Charm* concentrates on frequent closed itemsets only.

The initialization phase is equivalent to *Eclat*'s: first the root node is created that represents the empty set. By definition, the empty set is included in every transaction, thus its support is equal to the number of transactions in the dataset (100%). *Charm* transforms the layout of the dataset in vertical format, and inserts under the root node all frequent attributes. After this, the dataset itself can be deleted from main memory since it is not needed anymore. Then the `extend` procedure is called recursively for each child of the root. At the end, all frequent *closed* itemsets are discovered in the IT-tree.

The following methods are equivalent to the methods of *Eclat* with the same name: `addChild`, `delete`, `getCandidate`, `sortChildren`. Their description can be found in Appendix B.1.

`replaceInSubtree` procedure: it has two parameters, an IT-node (*curr*), and an itemset X (the itemset part of another node). The method is the following: let Y be the union of X and the itemset part of *curr*. Then, traverse recursively the subtree of *curr*, and replace everywhere the itemset of *curr* (as a sub-itemset) with Y.

`save` procedure: this procedure is a bit different from the procedure described in *Eclat*. First, it must be checked whether the current itemset is closed or not. It can be done by testing if a proper superset of the current node with the same support was found before. If yes, then the current node is not closed. If the test is negative, i.e. the current itemset is closed, we can process the itemset as we want (print it to the standard output, save it in a database, etc.).

Algorithm 5 ("extend" procedure of Charm):

Method: extend an IT-node recursively (discover FCIs in its subtree)
Input: *curr* – an IT-node whose subtree is to be discovered

```
loop over the "brothers" (other children of its parent) of curr (other)
{
    if (curr.tidset = other.tidset) {   // Property 1 of Theorem B.2
        replaceInSubtree(curr, other.itemset);
        delete(other);
    }
    else if (curr.tidset ⊂ other.tidset) {   // Property 2 of Theorem B.2
        replaceInSubtree(curr, other.itemset);
    }
    else if (curr.tidset ⊃ other.tidset) {   // Property 3 of Theorem B.2
        candidate ← getCandidate(curr, other);
        delete(other);
        if (candidate ≠ null) then curr.addChild(candidate);
    }
    else {   // if (curr.tidset ≠ other.tidset)   // Property 4 of Theorem B.2
        candidate ← getCandidate(curr, other);
        if (candidate ≠ null) then curr.addChild(candidate);
    }
}

sortChildren(curr);   // optimization, results in a less number of non-frequent candidates

while curr has children
{
    child ← (first child of curr);
    extend(child);
    save(child);   // process the itemset
    delete(child);   // free memory, not needed anymore
}
```

Running example. The execution of *Charm* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 1$ (20%) is illustrated in Figure B.2. The execution order is shown in circles. Numbers on the left side indicate the step when the node is inserted in the IT-tree, while numbers on the right side signal when the node is removed. For instance, C is inserted at step 3, and it is removed from the IT-tree at step 7/b. For the sake of easier understanding, the element reordering optimization is not applied.

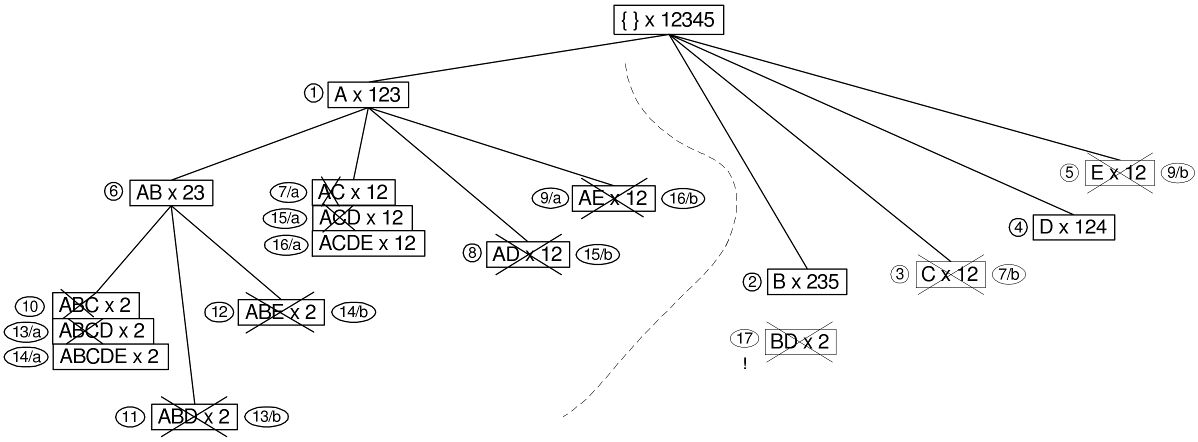

Figure B.2: Execution of *Charm* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 1$ (20%).

The algorithm first initializes the IT-tree with the root node, and adds all frequent attributes under it. The children of the root node are extended recursively one by one in a pre-order way, from left-to-right. *Extending A.* The tidsets of A and B have no relation (Property 4), thus AB is generated and inserted under A (step 6). The tidset of A is a proper superset of the tidset of C (Property 3), thus the following tasks are executed: first, AC is inserted under A (step 7/a), and then C is removed from the IT-tree (step 7/b). Step 8 is similar to step 6, and step 9 is like step 7. Now, *extend* is called on AB. By Property 4, ABC, ABD, and ABE are inserted under AB. The tidset of ABC is equivalent to the tidset of ABD (Property 1), thus in the subtree of node ABC the sub-itemset "ABC" is replaced by "ABCD" everywhere (step 13/a). Since the subtree of node ABC consists of one node only, this replacement only concerns one node ($ABC \times 2$ becomes $ABCD \times 2$). After the replacement, the node ABD is removed (step 13/b). The following three steps (steps 14, 15, and 16) are similar to step 13. With step 16 the subtree of A is completely discovered, i.e. all frequent closed itemsets are found in this subtree. *Extending B.* After applying Property 4, we get the itemset BD (remember, node C was removed in step 7). With BD there is a "problem": although it is frequent, this itemset is not closed because we already found a proper superset of it with the same support ($ABCDE$), thus BD is not added to the IT-tree. *Extending D.* Since this node has no "brother" nodes (node E was removed in step 9), the algorithm stops. At this point, all frequent closed itemsets are discovered.

Fast Subsumption Checking

Let X_i and X_j be two itemsets. We say that X_i *subsumes* X_j (or X_j is *subsumed by* X_i), iff $X_j \subset X_i$ and $supp(X_i) = supp(X_j)$. Recall that in the `save` procedure, before adding an itemset X to the set of closed itemsets, *Charm* checks if X is subsumed by a previously found closed itemset. In other words, *Charm* can find itemsets that are actually *not* closed. It might seem to be a problem, but Zaki managed to find a very efficient way to filter these non-closed itemsets.

Zaki proposes a hash structure for storing FCIs in order to perform fast subsumption checking. It also means that *Charm* stores the found frequent closed itemsets in the main memory. The idea is the following. *Charm* computes a hash function on the tidset and stores in the hash table a closed set with its support value. Let $h(X_i)$ denote the hash function on the tidset of X_i. This hash function has one important criteria: it must return the same value for itemsets that are included by the same set of objects. Several hash functions could be possible, but *Charm* uses the sum of the tids in the tidset (note that this is not the same as support, which is the cardinality of the tidset). Itemsets having the same hash value are stored in a list at the same position of the hash. Before adding X to the set of closed itemsets, we retrieve from the hash table all entries with the hash key $h(X)$. For each element C in this list, check if $supp(X) = supp(C)$. If yes, check if $X \subset C$. If yes, then X is subsumed by C, and we do not register X in the hash table.

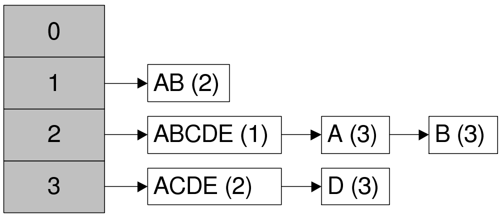

Figure B.3: Hash table for the IT-tree in Figure B.2.

Example. Let us see Figure B.3 that depicts the hash structure of the IT-tree in Figure B.2. For this example, the size of the hash table is set to four.[14] At each position of the hash table there are pointers to lists. In each list we can find itemsets that have the same hash key. In the running example we saw that BD is not closed. Using the hash table it can be determined the following way. First, compute the sum of the tids in its tidset (its tidset has one element only, so the sum is 2); then modulo this sum by the size of the hash table to get its hash value: 2 mod 4 = 2. Traverse the list of the hash table at position 2. We find that $ABCDE$ has the same support value as BD, thus check if BD is a proper subset of $ABCDE$. As the answer is positive, it means that BD is not closed.

Experimental Results

Experimental results of *Charm* are reported in Chapter 3 together with the *Touch* algorithm.

[14] In our implementation, we set the size of the hash table to 100,000.

B.3 Talky-G

This appendix is based on Chapter 2, where we presented the common parts of vertical algorithms. *Talky-G* is an adaptation of *Eclat* and *Charm*, allowing one to find frequent *generators* only. *Talky-G* is a vertical algorithm that traverses the itemset-tidset search tree (IT-tree) in a depth-first manner, in a reverse pre-order way. The IT search tree of dataset $\mathcal{D}$ (Table 2.1) is depicted in Figure 2.1. The goal of *Talky-G* is to find frequent generators only in this search tree. *Talky-G*, just like *Eclat*, processes the input dataset in a vertical way, i.e. it associates to each attribute its tidset pair. Here we present the algorithm in detail.

Reverse Pre-Order Traversal

While *Eclat* traverses the IT-tree in a pre-order way, *Talky-G* uses a so-called *reverse pre-order* traversal. In [CG05], Calders and Goethals made the following important observation. Let X be the itemset of a node in the IT-tree. The nodes of the subsets of X are either on the path from the root node to the node of X, or are in a branch that comes *after* X, never in a branch that comes before the branch of X. Hence, the traversal can be changed as follows: the same tree is processed, still in a depth-first manner, but from *right-to-left*. This order is called reverse pre-order. As pointed out in [CG05], using reverse pre-order traversal, all subsets of X are handled before X itself. Though no name was given in [CG05] to this modified version of *Eclat*, we will refer to this algorithm as *Talky*. That is, *Talky* is *Eclat* with reverse pre-order traversal. As we will see, our algorithm *Talky-G* also uses the reverse pre-order strategy.

The reverse pre-order traversal does not have any drawback on performance or memory usage. Furthermore, all the optimization methods described in Chapter 2 can be combined with the reverse pre-order traversal namely **(1)** element reordering, **(2)** support count of 2-itemsets, and **(3)** diffsets.

Example. See Figure B.4 for a comparison between the two traversals namely pre-order with *Eclat* (left) and reverse pre-order with *Talky* (right). The order of traversal is indicated in circles on the left side of the nodes.

Basic Property of Generators

As defined in Def. 1.1, an itemset is called *generator* if it has no proper subset with the same support. *Talky-G* exploits the following property of generators in order to reduce the search space in the IT-tree:

Property B.2 *If an itemset is not generator, then none of its supersets are generators [BTP$^+$00a].*

The Algorithm

Pseudo code. The main block of the algorithm is given in Algorithm 6. First, the IT-tree is initialized, which includes the following steps: the root node, representing the empty set, is created. By definition, the support of the empty set is equal to the number of transactions in the dataset (100%). *Talky-G* transforms the layout of the dataset in vertical format, and inserts under the root node all frequent attributes. In addition to *Eclat*, *Talky-G* has to determine whether an attribute is generator or not. If the support of an attribute is equal to the total number of transactions in the database, then the attribute is not generator since it has a proper subset

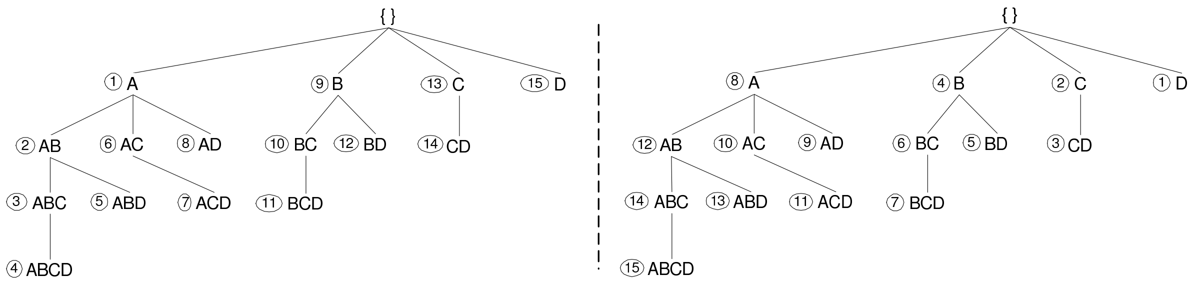

Figure B.4: **Left:** pre-order traversal with *Eclat*. **Right:** reverse pre-order traversal with *Talky*.

(the empty set) with the same support. After the insertion of all frequent 1-long generators, the dataset itself can be deleted from the main memory since it is not needed anymore. Then, the `extend` procedure is called recursively for each child of the root in a reverse pre-order way from right-to-left. While *Eclat* finds all frequent itemsets in the subtree of a node, *Talky-G* concentrates on frequent generators only. At the end, all frequent generators are discovered in the IT-tree.

The following methods are equivalent to the methods of *Eclat* with the same name: `addChild`, `sortChildren`, and `delete`. The description of these methods can be found in Appendix B.1.

`extend` procedure: this method (see Algorithm 7) discovers all frequent generators in the subtree of the current node. First, the procedure forms new generators with the "brother" nodes of the current node. Then, these generators are added below the current node and are extended recursively in a reverse pre-order way from right-to-left.

`getNextGenerator` function: this method (see Algorithm 8) is a bit different from the `getCandidate` function of *Eclat*. The function has two nodes as input parameters. The function either returns a new frequent generator, or "null" if no frequent generator can be produced from the two input nodes. A candidate node is created by taking the union of the itemsets of the two input nodes, and by calculating the intersection of their tidsets. The input nodes are also called as the *parents* of the candidate. Then, the candidate undergoes a series of tests. *First*, a frequency test is used to eliminate non-frequent itemsets. *Second*, the candidate is compared to its parents. If its tidset is equivalent to the tidset of one of its parents, then the candidate is not generator by Def. 1.1. If an itemset passed these two tests, it is still not sure that it is generator. As seen before, the reverse pre-order traversal *guarantees* that when an itemset is reached in the IT-tree, all its subsets are handled before. *Talky-G* collects frequent generators in a "list" too (see also the `save` procedure). The *third* test checks if the candidate has a proper subset with the same support in this "list". If yes, then the candidate is not generator by Def. 1.1. This last step might seem to be a very expensive step, but as we will see later, it can be done very efficiently with a special hash data structure. If a candidate survives all the tests, then it is sure that the candidate is a frequent generator. If a candidate fails a test, then it is not added to the IT-tree, and thus none of its supersets are generated (see Prop. B.2). This way the search space is reduced to frequent generators *only*.

In Chapter 2 we presented an optimization method for the support count of 2-itemsets. This technique can be used here: if the itemset parts of the parents of a candidate consist of one attribute only, then the union of their itemsets is a 2-itemset. In this case, instead of taking the intersection of their tidsets, we consult the upper-triangular matrix to get its support. Naturally, this matrix had been built before in the initialization phase. For the sake of easier understanding, this optimization is not included in Algorithm 8.

Algorithm 6 (main block of Talky-G):

```
root.itemset ← ∅;   // root is an IT-node whose itemset is empty
root.tidset ← {all transaction IDs};   // the empty set is present in every transaction
root.support ← |O|;   // where |O| is the total number of objects in the input dataset
root.parent ← null;   // the root has no parent node
loop over the vertical representation of the dataset (attr) {
    if ((attr.supp ≥ min_supp) and (attr.supp < |O|)) {   // frequent and generator
        root.addChild(attr);
    }
}
delete the vertical representation of the dataset;   // free memory, not needed anymore
sortChildren(root);   // optimization, results in a less number of non-frequent candidates

loop over the children of root from right-to-left (child) {
    save(child);   // process the itemset
    extend(child);   // discover and then destroy the subtree below child
}
delete the children of root;   // only 1-long itemsets are left below the root
delete root;   // destroy the IT-tree
```

Algorithm 7 ("extend" procedure of Talky-G):

Method: extends an IT-node recursively (discovers FGs in its subtree)
Input: *curr* – an IT-node whose subtree is to be discovered

```
loop over the "brothers" (other children of its parent) of curr from left-to-right (other) {
    generator ← getNextGenerator(curr, other);
    if (generator ≠ null) then curr.addChild(generator);
}
sortChildren(curr);   // optimization, results in a less number of non-frequent candidates

loop over the children of curr from right-to-left (child) {
    save(child);   // process the itemset
    extend(child);   // discover and then destroy the subtree below child
}
delete the children of curr;   // free memory
```

`save` procedure: this method has an IT-node as its parameter representing a frequent generator. The method has two tasks to perform. First, this is the method that is responsible for processing the itemset. It can be implemented in different ways, e.g. by simply printing the itemset and its support value to the standard output, or by saving the itemset in a file, in a database, etc. Second, the `save` procedure will also store the current frequent generator in a "list". Due to the reverse pre-order traversal, it can be possible that later on *in another branch* of the IT-tree we will find a proper superset (call it X) of the current itemset with the same support. According to Def. 1.1, X is not generator, and thanks to the "list" X can be pruned.

Algorithm 8 ("getNextGenerator" function of Talky-G):

Method: create a new frequent generator
Input: two IT-nodes (*curr* and *other*)
Output: a frequent generator or null

```
cand.tidset ← curr.tidset ∩ other.tidset;
if (cardinality(cand.tidset) < min_supp) {   // test 1
      return null;   // not frequent
}
// else, if it is frequent
if ((cand.tidset = curr.tidset) or (cand.tidset = other.tidset)) {   // test 2
      return null;   // not generator (its supp. is equal to the supp. of one of its proper subset)
}
// else, if it is a potential generator
cand.itemset ← curr.itemset ∪ other.itemset;
if (cand has a proper subset with the same support in the hash) {   // test 3
      return null;   // not generator
}
// if cand passed all the tests then cand is a frequent generator
return cand;
```

When the algorithm stops, all frequent generators are stored in the "list". That is, *Talky-G* finds all frequent generators and nothing else.

Running example. The execution of *Talky-G* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 1$ (20%) is illustrated in Figure B.5. The execution order is indicated on the left side of the nodes in circles. For the sake of easier understanding, the element reordering optimization is not applied.

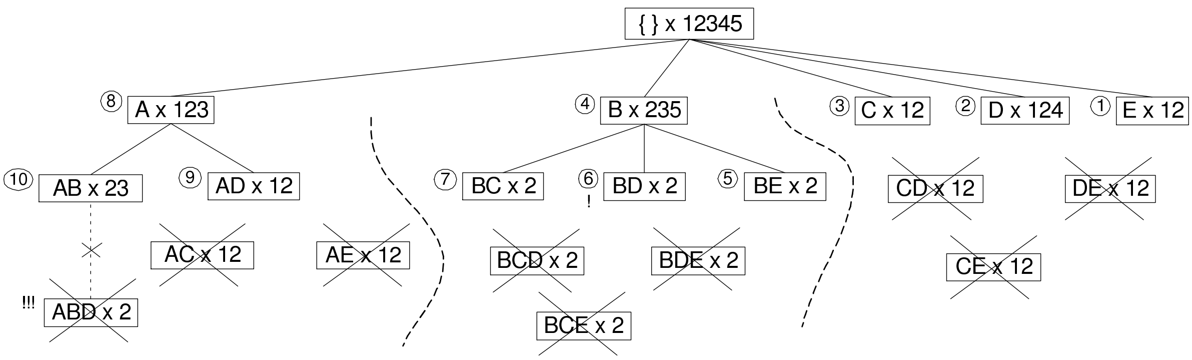

Figure B.5: Execution of *Talky-G* on dataset $\mathcal{D}$ (Table 2.1) with $min\_supp = 1$ (20%).

The algorithm first initializes the IT-tree with the root node. Since there is no full column in the input dataset, all attributes are generators, thus they are added under the root. The children of the root node are extended recursively one by one from *right-to-left*. Node E has no "brother" nodes on its right side, thus it cannot be extended. Node D is extended with E, but the resulting itemset DE is not generator since its support is equal to the support of its parent E. Node C is extended with D and E, but neither CD nor CE is generator because of the previous reason. 2-long supersets of B are formed by using its "brother" nodes on its right side. When a new candidate is created, it is also tested if it has a previously found proper subset with the same support. If the test is negative, then the candidate is added to the IT-tree and to the "list" of generators. Thus, the nodes BC, BD, and BE are added below B. Node BE cannot be extended. The extensions of BD and BC produce no new generators. 2-long supersets of A are added to the IT-tree. The candidates AC and AE are not generators because of C and E, respectively. The combination of AB and AD produces the candidate ABD, which represents a special case. Its support is different from its parents, but we already found a proper subset of it with the same support (BD) *in a previous branch*. According to Def. 1.1, ABD is not generator, thus it is not added to the IT-tree. When the algorithm stops, all frequent generators (and *only* frequent generators) are inserted in the IT-tree *and* in the "list" of generators.

Fast Subsumption Checking

Let X_i and X_j be two itemsets. We say that X_i *subsumes* X_j (or X_j is *subsumed by* X_i), iff $X_j \subset X_i$ and $supp(X_i) = supp(X_j)$. X_i (resp. X_j) is also known as *subsumer* (resp. *subsumee*). By Def. 1.1, if an itemset subsumes another itemset, then the subsumer is not generator. Recall that in the `getNextGenerator` function, when a new candidate itemset C is created, *Talky-G* checks if C subsumes a previously found generator. If the test is positive, then clearly C is not generator. This subsumption test might seem to be an expensive step, but we found a very efficient way to filter these non-generator itemsets.

In *Charm*, Zaki proposed a hash structure for storing FCIs in order to perform fast subsumption checking. In *Talky-G* we adapted this hash structure for storing frequent generators. Our motivation is the same: we want to perform subsumption checks very efficiently. It also means that *Talky-G*, just like *Charm*, stores the found frequent generators in the main memory (see also the `save` procedure). The idea is the following. *Talky-G* computes a hash function on the tidset and stores in the hash table a generator with its support value. Let $h(X_i)$ denote the hash function on the tidset of X_i. This hash function has one important criteria: it must return the same value for itemsets that are included by the same set of objects. Several hash functions could be possible, but *Talky-G* uses the sum of the tids in the tidset (note that this is not the same as support, which is the cardinality of the tidset). Itemsets having the same hash value are stored in a list at the same position of the hash. For the subsumption check of a candidate itemset C, we retrieve from the hash table all entries with the hash key $h(C)$. For each element G in this list, test if $supp(C) = supp(G)$. If yes, test if $C \supset G$. If yes, then C subsumes G, i.e. C is not generator. If C subsumes no entries in the list, then C is generator, thus C is added to the end of the list.

Example. Let us see Figure B.6 that depicts the hash structure of the IT-tree in Figure B.5. This hash table contains all frequent generators of dataset $\mathcal{D}$. For this example, the size of the hash table is set to four.[15] At each position of the hash table there are pointers to lists. In each list we can find itemsets that have the same hash key. In the running example we saw that ABD is not generator. Using the hash table it can be determined the following way. First, compute the sum of the tids in its tidset (its tidset has one element only, so the sum is 2); then modulo this sum by the size of the hash table to get its hash value: 2 mod 4 = 2. Traverse the list of

[15] In our implementation, we set the size of the hash table to 100,000.

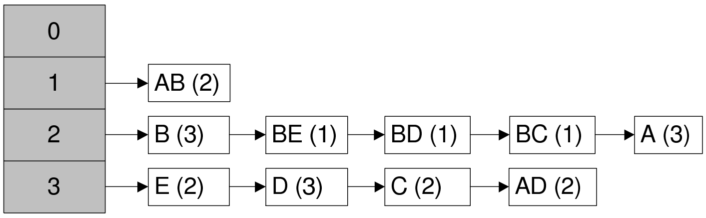

Figure B.6: Hash table for the IT-tree in Figure B.5.

the hash table at position 2. The support of B differs from the support of ABD, thus we take the next element of the list. We find that BE has the same support value as ABD, but ABD is not a proper superset of BE, thus we continue. BD has the same support as ABD, and ABD is a proper superset of BD, thus ABD is not generator. At this point the traversal of the list is finished.

Pre-Order Traversal

One might ask the question if it is possible to produce the same result with *Eclat*, i.e. with a pre-order traversal of the IT-tree. The answer is positive, but this approach has several drawbacks.[16] First, in the case of pre-order traversal, all subsets of the current node are not yet available, i.e. it cannot be decided whether a candidate is a generator or not. However, this problem can be solved the following way. Candidate generators that passed the first two tests in the `getNextGenerator` function are added to the hash structure. In the hash such itemsets are stored that are *likely* to be generators, but their generator status is not yet sure. Hence, we call them candidate generators. When a new candidate C is found, non-generators that are proper supersets of C with the same support are removed from the hash, and C is added to the end of the list. Note that the hash can contain *several* subsumers of C, thus all the entries in the corresponding list must be compared to C. Furthermore, each potential generator C must be added to the hash. The second drawback is that the search space is larger. This is due to the fact that the IT-tree can contain non-generators too. Third, the hash contains candidate generators whose generator status only becomes clear when the algorithm terminates. However, with the reverse pre-order strategy, all these drawbacks can be eliminated in *Talky-G*.

Experimental Results

Experimental results of *Talky-G* are reported in Chapter 3 together with the *Touch* algorithm.

Conclusion

Here we presented a new algorithm for extracting frequent generators from a dataset. The proposed algorithm called *Talky-G* is a vertical, depth-first algorithm that traverses its IT-tree in a reverse pre-order way. *Talky-G* has several advantages: **(1)** Due to the reversed traversal, it is guaranteed that when an itemset X is found, all its subsets are handled *before* X. This property allows us to quickly eliminate non-generators. **(2)** As non-generators are eliminated, only frequent generators are added to the IT-tree, i.e. the search space is reduced to the minimum. **(3)** At each time, the hash structure contains the already found frequent generators. Once an

[16] We refer to this algorithm as *Eclat-G*. That is, *Eclat-G* produces the same result as *Talky-G* by using an *Eclat*-like traversal.

itemset G is added to the hash, it is sure that G is generator. When the algorithm terminates, all frequent generators are collected in the hash.

As a summary, we can say that *Talky-G* is a very efficient solution for finding the family of frequent generators.

Appendix C

Horizontal and Vertical Data Layouts

Most itemset mining algorithms use a *horizontal* database layout, such as the one shown in Figure C.1 (left), consisting of a list of transactions (or objects), where each transaction has an identifier followed by a list of items that are included in that transaction. Some algorithms, like *Eclat*, *Charm*, or *Talky-G*, use a *vertical* database layout, such as the one shown in Figure C.1 (right), consisting of a list of items (or attributes), where each item is associated with a list of transactions that include the given item. One layout can easily be converted into the other on-the-fly, with very little cost. This process requires only a trivial amount of overhead.

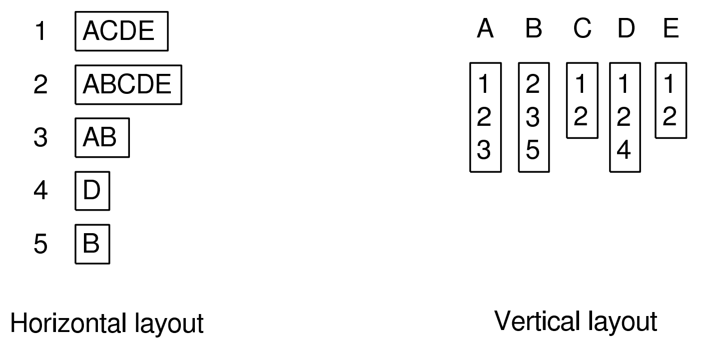

Figure C.1: Horizontal and vertical layouts of dataset $\mathcal{D}$ (Table 2.1).

Horizontal to Vertical Database Transformation

For each transaction t, we read its item list. During the transformation process, we build an array that is indexed by items of the database. We insert the ID of t in those positions of the array that are indexed by the items present in the associated list of t.

Example. Consider the item list of transaction 1, shown in Figure C.1 (left). We read its first item, A, and insert 1 in the array indexed by item A. We repeat this process for all other items in the list and for all other transactions. Figure C.2 shows the transformation process step by step.[17]

[17] In the figure, "tid" stands for "transaction ID".

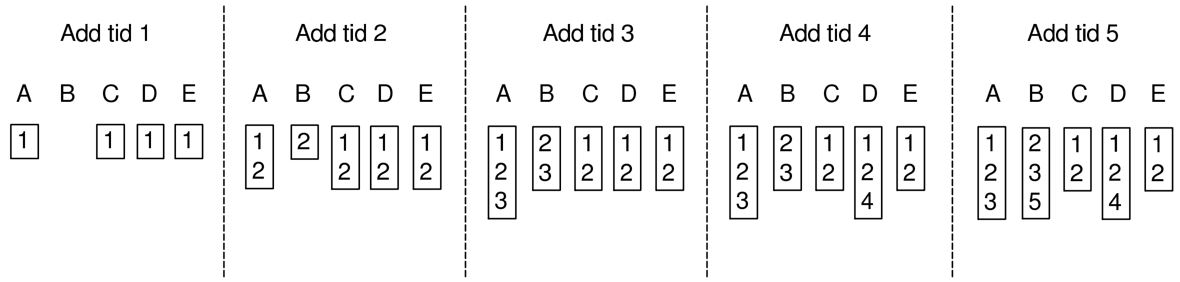

Figure C.2: Horizontal to vertical database transformation.

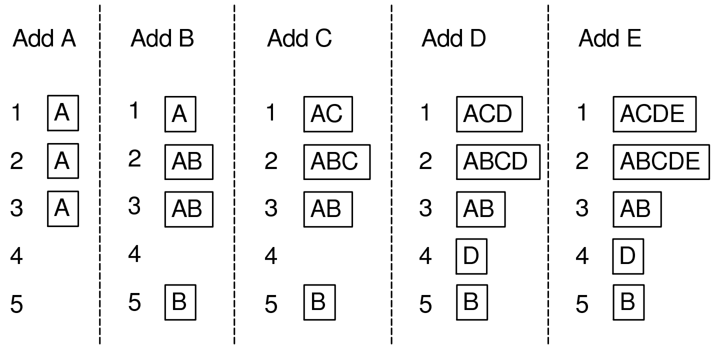

Figure C.3: Vertical to horizontal database transformation.

Vertical to Horizontal Database Transformation

For each item i, we read its transaction list. During the transformation process, we build an array that is indexed by transaction IDs. We insert item i in those positions of the array that are indexed by the transactions present in the associated list of i.

Example. Consider the transaction list of item A, shown in Figure C.1 (right). We read its first transaction ID, 1, and insert A in the array indexed by transaction 1. We repeat this process for all other transaction IDs in the list and for all other items. Figure C.3 shows the transformation process step by step.

Appendix D

Efficient Support Count of 2-itemsets

Here we present an optimization of the support count of 2-itemsets. This method was proposed by Zaki in [ZH02] for the *Charm* algorithm. However, this optimization can also be used with *Eclat* and *Talky-G*, as well as with breadth-first algorithms, such as *Apriori*.

In the case of vertical algorithms (e.g. *Eclat*, *Charm*, *Talky-G*), this method significantly reduces the number of intersection operations. The idea is that the support of 2-itemsets is calculated and stored in a matrix. Then, an intersection operation is performed *only* if it surely results in a frequent itemset.

In the case of levelwise algorithms, with this method we can read the support from the matrix directly, and we do not need to use a trie for finding the subsets of each transacation in C_2 (where C_2 contains the potentially frequent 2-itemsets). Note that this optimization only concerns the support count of 2-itemsets in C_2. The support values of larger candidates are determined by a trie.

The Algorithm

The algorithm requires that the database be in horizontal format. In the case of vertical algorithms it means that first the database must be transformed (see Appendix C). If the database has n attributes, then an $(n-1) \times (n-1)$ upper triangular matrix is built, such as the one shown in Figure D.1. This matrix will contain the support values of 2-itemsets, thus its entries are initialized by 0. A row (transaction) of the database is decomposed into a list of 2-itemsets. For each element in this list, the value of its corresponding entry in the matrix is incremented by 1. This process is repeated for each row of the database.

	E	D	C	B
A	0	0	0	0
B	0	0	0	
C	0	0		
D	0			

Figure D.1: Initialized upper triangular matrix for counting the support of 2-itemsets.

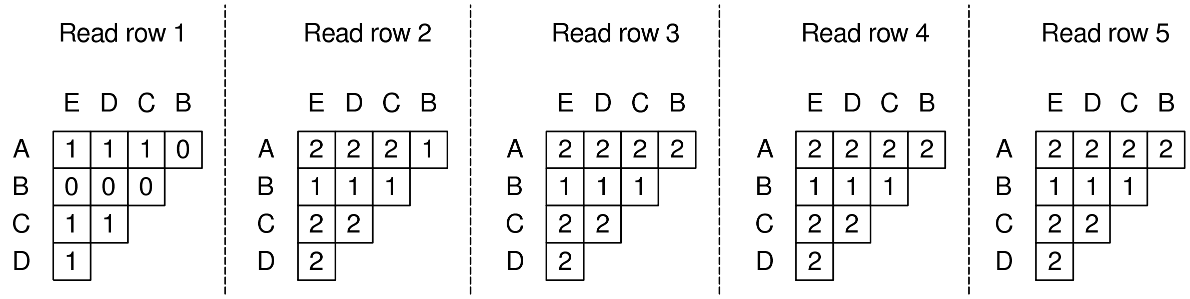

Figure D.2: Support count of 2-itemsets of dataset $\mathcal{D}$ (Table 2.1) with an upper triangular matrix.

Example. The first row of dataset $\mathcal{D}$ (Table 2.1) includes the itemset $ACDE$. This itemset is decomposed into the following list of 2-itemsets: $\{AC, AD, AE, CD, CE, DE\}$. We read the first element of this list, AC, and increment its entry in the triangular matrix. We repeat this process for all other itemsets in the list and for all other rows of the database. Figure D.2 shows the process step by step.

Appendix E

Computing the Transversal Hypergraph

This appendix presents an optimized version of Berge's algorithm [Ber89] for solving the transversal hypergraph problem (see Def. 4.6). Before presenting the algorithm, we show the correspondance between itemsets and hypergraphs, and we also review Berge's algorithm.

Relation Between Itemsets and Hypergraphs

In this paper we present several itemset mining algorithms, e.g. *Eclat*, *Charm*, *Talky-G*, etc. These algorithms extract specific subsets of frequent itemsets from a given context. Here we show that a family of itemsets can be treated as a hypergraph, and vice versa. As seen in Def. 4.1, a hypergraph $\mathcal{H}$ is a pair $(V, \mathcal{E})$, where V is a finite set $\{v_1, v_2, \ldots, v_n\}$ and $\mathcal{E}$ is a family of subsets of V. The elements of V are called vertices, the elements of $\mathcal{E}$ edges. In Section 1.1 we saw that a formal context is a triple $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is a set of objects, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is a set of items, and $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ is a relation between $\mathcal{O}$ and $\mathcal{A}$, where $\mathcal{R}(o, a)$ means that the object o has the item a. A set of items is called an itemset.

The set $\mathcal{A}$ can be considered as a set of vertices V. An itemset corresponds to an edge $E \in \mathcal{E}$. From this it follows that a set of itemsets can be considered as a family of edges $\mathcal{E}$.

Example. Consider the hypergraph $\mathcal{H}$ in Figure 4.1, where $V = \{a, b, c, d\}$ and $\mathcal{E} = \{\{a\}, \{b, c\}, \{a, c, d\}\}$. This hypergraph corresponds to the following set of itemsets: $\{\{a\}, \{b, c\}, \{a, c, d\}\}$. For convenience, we will use separator-free set notations, and we will indicate itemsets with capital letters. That is, the hypergraph $\mathcal{H}$ can be considered as the following set of itemsets: $\{A, BC, ACD\}$. This holds in the other direction too, i.e. the hypergraph representation of the family of itemsets $\{A, BC, ACD\}$ is depicted in Figure 4.1.

In the rest of the paper, we will treat a family of itemsets as a hypergraph and vice-versa if there is no danger of ambiguity. Thus for a set of itemsets $\{A, BC, ACD\}$, we write "the *hypergraph* $\{A, BC, ACD\}$", etc.

The Algorithm of Berge

In this subsection we review the basic algorithm of Berge [Ber89], which is the most simple and direct scheme for generating all minimal transversals of a hypergraph. First, let us see two useful operations on hypergraphs:

Definition E.8 *Let $\mathcal{H} = \{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ and $\mathcal{G} = \{\mathcal{E}'_1, \ldots, \mathcal{E}'_{n'}\}$ be two hypergraphs. Then,*

$$\mathcal{H} \cup \mathcal{G} = \{\mathcal{E}_1, \ldots, \mathcal{E}_n, \mathcal{E}'_1, \ldots, \mathcal{E}'_{n'}\}, \text{ and}$$

$$\mathcal{H} \vee \mathcal{G} = \{\mathcal{E}_i \cup \mathcal{E}'_j,\ i = 1, \ldots, n,\ j = 1, \ldots, n'\}.$$

The first operation is the *union* of $\mathcal{H}$ and $\mathcal{G}$, i.e. the hypergraph whose edges are the edges of both hypergraphs. The second operation is very similar to the *Cartesian product*, i.e. the union of all possible pairs of edges, where one element of a pair is from the first hypergraph, and the other element is from the second hypergraph.

Proposition E.3 ([Ber89]) *Let $\mathcal{H}$ and $\mathcal{G}$ be two simple hypergraphs. Then,*

$$Tr(\mathcal{H} \cup \mathcal{G}) = min(Tr(\mathcal{H}) \vee Tr(\mathcal{G})).$$

Let $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\},\ i = 1, \ldots, n$ be the partial hypergraph of the hypergraph $\mathcal{H}$. It holds that $\mathcal{H}_i = \mathcal{H}_{i-1} \cup \{\mathcal{E}_i\}$, for all $i = 2, \ldots, n$, where $\mathcal{H}_1 = \{\mathcal{E}_1\}$ and $\mathcal{H}_n = \mathcal{H}$. Thus, $Tr(\mathcal{H}_i) = Tr(\mathcal{H}_{i-1} \cup \{\mathcal{E}_i\})$, and by Prop. E.3,

Equation E.1

$$\begin{aligned} Tr(\mathcal{H}_i) &= min(Tr(\mathcal{H}_{i-1}) \vee Tr(\{\mathcal{E}_i\})) \\ &= min(Tr(\mathcal{H}_{i-1}) \vee \{\{v\},\ v \in \mathcal{E}_i\}) \end{aligned}$$

The algorithm of Berge is based on this equation. The algorithm computes all minimal transversals of a given hypergraph $\mathcal{H}$ in two steps. First, it computes the minimal transversals of the partial hypergraph $\mathcal{H}_{i-1}$ and then it calculates the Cartesian product of the set $Tr(\mathcal{H}_{i-1})$ by the i^{th} edge $\mathcal{E}_i$ of $\mathcal{H}$. Finally, non-minimal elements are removed. Thus, the algorithm starts with the computation of $Tr(\mathcal{H}_1)$, which is a trivial case ($\mathcal{H}_1$ has one edge only, $\mathcal{E}_1$, whose minimal transversals are its vertices). Then, the algorithm adds one by one the rest of the edges, computing at each step the set of minimal transversals of the new partial hypergraph. The algorithm terminates when the last edge $\mathcal{E}_n$ is added. The algorithm of Berge outputs at the end all minimal transversals of the input hypergraph $\mathcal{H}$ [Ber89].

BergeOpt: An Optimized Version of Berge's Algorithm

In the previous subsection we reviewed the algorithm of Berge, which implements the most simple and direct approach for calculating the minimal transversals of a hypergraph. Here we present an optimized version of Berge's algorithm that we call *BergeOpt*.

In [LFFM$^+$03], Le Floc'h *et al.* presented an algorithm called *JEN* whose goal is to efficiently extract generators from a concept lattice for mining exact and approximate association rules. As part of *JEN*, the aforementioned authors presented a simple algorithm without a name for calculating all the minimal transversals of a hypergraph. In the rest of this appendix we present this algorithm in an extended and completed way. In addition to [LFFM$^+$03], **(i)** we show that this algorithm is actually an optimization of Berge's original algorithm (hence the name *BergeOpt*), and **(ii)** we provide a proposition (see Prop. E.4) and its proof.

Optimization idea. One drawback of Berge's algorithm is that after calculating the Cartesian product of the set $Tr(\mathcal{H}_{i-1})$ by the i^{th} edge $\mathcal{E}_i$ of $\mathcal{H}$ (see Equation E.1), it stores the resulting elements together in the same set, i.e. it has no information whether an element is minimal or not. As a consequence, the filtering of non-minimal elements can be quite expensive when the resulting set has a large number of elements because the algorithm must test the minimality of all elements, including also such elements that are actually minimal.

Our optimization is based on the idea to separate minimal and potentially minimal transversals in two different lists L_1 and L_2, respectively. This way, our optimized algorithm only has to check the minimality of the potentially minimal elements in L_2. As a result, the number of expensive subset checks can be reduced.

The *BergeOpt* algorithm exploits the following proposition:

Proposition E.4 *In the* BergeOpt *algorithm, the potentially minimal transversals stored in the list L_2 form a simple hypergraph, i.e. L_2 has no two elements e_i and e_j such that $e_i \subseteq e_j$.*

Proof. Assume $X, Y \subseteq V$ are two distinct subsets in $Tr(\mathcal{H}_{i-1}) - Tr(\mathcal{H}_i)$, i.e., they are minimal transversals of $\mathcal{H}_{i-1}$ that lost this status in the i-th partial hypergraph. Assume also that $X \cup \{a\}$ and $Y \cup \{b\}$ are two candidates for $Tr(\mathcal{H}_i)$ produced by the algorithm (i.e., $\{a, b\} \subseteq \mathcal{E}_i$ whereby $a \notin X$ and $b \notin Y$).

Notice that any element of the L_2 list will have the form $X \cup \{a\}$ for some X and a.

Now, without loss of generality we can hypothesize $X \cup \{a\} \subseteq Y \cup \{b\}$, and show this leads to a contradiction. First, notice that $a \neq b$, otherwise we would have $X \subseteq Y$ hence a contradiction with the minimal transversal status. Next, we deduce that $Y = \bar{X} \cup \bar{Y}$ where $\bar{X} = X - \{b\}$ hence $X = \bar{X} \cup \{b\}$. Yet this means that $b \in X \cap \mathcal{E}_i$ which contradicts $X \notin Tr(\mathcal{H}_i)$. □

Pseudo code. The pseudo code of the algorithm is given in Algorithm 9. Let $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\}$, $i = 1, \ldots, n$ be the partial hypergraph of the hypergraph $\mathcal{H}$. It holds that $\mathcal{H}_i = \mathcal{H}_{i-1} \cup \{\mathcal{E}_i\}$, for all $i = 2, \ldots, n$, where $\mathcal{H}_1 = \{\mathcal{E}_1\}$ and $\mathcal{H}_n = \mathcal{H}$. Let $\mathcal{MT}_{\mathcal{H}_i}$ denote the set of all minimal transversals of the partial hypergraph $\mathcal{H}_i$.

As input, we have a set of itemsets that we treat as a hypergraph. The goal is to compute all the minimal transversals of this hypergraph. The algorithm performs this task in an incremental way. First, the algorithm takes the first itemset $\mathcal{E}_1$ of the input and it calculates its minimal transversals. This is a trivial case; we only have to decompose the itemset into its 1-long subsets. For instance, the itemset ABC has three minimal transversals namely A, B, and C. Then, the algorithm takes the next itemset $\mathcal{E}_i$ of the input and it updates the list of minimal transversals $\mathcal{MT}_{\mathcal{H}_{i-1}}$ if necessary. This is done the following way. Each minimal transversal m found so far, i.e. each element of $\mathcal{MT}_{\mathcal{H}_{i-1}}$, is tested if it has a common part with the current itemset $\mathcal{E}_i$. If it has, then m is a minimal transversal of $\mathcal{H}_i$ too, thus m is added to the list L_1. In the list L_1 we collect those itemsets that are minimal transversals of the partial hypergraph processed so far, including the current itemset $\mathcal{E}_i$ too. Prop. 4.1 guarantees that L_1 has no two elements e_1 and e_2 such that $e_1 \subseteq e_2$. If the test was negative, i.e. m has no common part with the current itemset $\mathcal{E}_i$, then it means that m is not a transversal of $\mathcal{E}_i$, thus m must be extended to have an intersection with $\mathcal{E}_i$ (in other words, m is a transversal of $\mathcal{H}_{i-1}$, but not a transversal of $\mathcal{H}_i$). This can be done by decomposing $\mathcal{E}_i$, and generating the one-size larger supersets of m using the 1-long subsets of $\mathcal{E}_i$ (Cartesian product of m with the vertices of $\mathcal{E}_i$). For instance, if $\mathcal{E}_i = BCH$, and the minimal transversal to be updated is AD, then the following potentially minimal transversals are generated: ABD, ACD, and ADH. We call these itemsets "potentially minimal transversals", because with this extension it is guaranteed that they became transversals of $\mathcal{H}_i$, but it is not sure that they are *minimal*, thus they are put in another list L_2. It can be possible that they have subsets among the minimal transversals in L_1. When all elements of $\mathcal{MT}_{\mathcal{H}_{i-1}}$ are tested against the current itemset $\mathcal{E}_i$, the lists L_1 and L_2 are filled. At this point, there are three possibilities (lines 20–23 of Algorithm 9): **(1)** L_1 is non-empty and L_2 is empty, or **(2)** L_1 is empty and L_2 is non-empty, or **(3)** both L_1 and L_2 are non-empty. In the *first case*, L_1 contains all the minimal transversals of $\mathcal{H}_i$. By Prop. 4.1, L_1 is a simple hypergraph. In the *second case*, L_2 contains all the minimal transversals of $\mathcal{H}_i$. Since L_1 is empty, all elements in L_2 are minimal. Moreover, from Prop. E.4 it follows that L_2 is a simple hypergraph. In the *third case*, the list L_2 must be cleaned first, i.e. if an element e_1 in L_1 is a subset of an element e_2 in L_2, then e_2 must be removed because e_2 is *not minimal*. Prop. E.4 guarantees that the elements of L_2 are not comparable w.r.t. set inclusion. Then, taking the union of the lists L_1 and L_2, we have all the minimal transversals of $\mathcal{H}_i$.

The algorithm continues by taking the next itemset of the input set (next current itemset) and it updates again the list of minimal transversals. The algorithm terminates when all elements of the input set are processed. At this point, the algorithm collected all the minimal transversals of the input set, i.e. it calculated the transversal hypergraph of the input hypergraph.

`cleanSupersets` procedure: this method removes non-minimal transversals from the list L_2, i.e. itemsets that have subsets in L_1. The procedure works as follows. It enumerates all elements of L_2. If the current element e_2 in L_2 has a subset in L_1, then e_2 is removed from L_2. When the procedure terminates, L_2 only contains *minimal* transversals.

Running example. Consider the following hypergraph $\mathcal{H} = \{ACD, ACH, BCD, DF, FH\}$. Let $\mathcal{E}_i$ denote the i^{th} element (edge) of the hypergraph, i.e. $\mathcal{E}_1 = ACD, \mathcal{E}_2 = ACH, \ldots, \mathcal{E}_5 = FH$. Let $\mathcal{H}_i$ denote the partial hypergraph that contains the first i elements of $\mathcal{H}$, i.e. $\mathcal{H}_1 = \{ACD\}$, $\mathcal{H}_2 = \{ACD, ACH\}$, …, $\mathcal{H}_5 = \{ACD, ACH, BCD, DF, FH\} = \mathcal{H}$. The notation $\mathcal{MT}_{\mathcal{H}_i}$ denotes the set of all minimal transversals of the partial hypergraph $\mathcal{H}_i$.

The execution of the algorithm is depicted in Table E.1. First, the algorithm takes $\mathcal{E}_1$ (ACD) and computes its minimal transversals that are A, C, and D. The algorithm continues with processing $\mathcal{E}_2$ (ACH). Each time when a new element of $\mathcal{H}$ is handled, the already found minimal transversals are tested. The itemsets A and C have common parts with $\mathcal{E}_2$, thus they are minimal transversals of ACH, so they are added to the list L_1. However, D has no common part with $\mathcal{E}_2$, which means that D is a minimal transversal of $\mathcal{H}_1$, but not a transversal of $\mathcal{H}_2$. In order to make D a transversal of $\mathcal{H}_2$, D is extended with the 1-long subsets of ACH, thus the following candidates are generated: AD, CD, and DH. These three itemsets are put in the list L_2. Then, the algorithm removes itemsets from L_2 that have subsets in L_1 since they are not minimal transversals (AD and CD). The union of L_1 and L_2, which is stored in the list $\mathcal{MT}_{\mathcal{H}_2}$, gives all the minimal transversals of $\mathcal{H}_2$. The same steps are repeated with the other elements of $\mathcal{H}$ ($\mathcal{E}_3$, $\mathcal{E}_4$, and $\mathcal{E}_5$). When the algorithm terminates, all minimal transversals of the hypergraph $\mathcal{H}$ are discovered. In this example, the transversal hypergraph of $\mathcal{H}$ is $Tr(\mathcal{H}) = \{DH, CF, ABF, ADF\}$.

Conclusion. In this appendix we presented an optimization of Berge's original algorithm [Ber89] called *BergeOpt* that can significantly reduce the number of expensive inclusion tests. Since Berge's algorithm several other, more efficient algorithms have been introduced. As pointed out in [KS05] for instance, the simple method of Berge needs exponential many steps to produce the whole output. It generates the first minimal transversal near the end of the procedure and its high memory requirements make it suitable only for small problem cases. However, as we pointed

Algorithm 9 ("getMinTransversals" function):

Description: BergeOpt algorithm (computing all minimal transversals of a hypergraph)
Input: a hypergraph ($\mathcal{H}$)
Output: all minimal transversals of $\mathcal{H}$ ($\mathcal{MT}$)

```
MT ← ∅;   // initialisation; no minimal transversals are found yet
loop over the elements of H (E_i)   // an element of H is an edge (an itemset)
{
    if (E_i is the first element of H) {
        MT ← {vertices of E_i};   // decomposition (1-itemsets of E_i)
    }
    else
    {
        L_1 ← ∅; L_2 ← ∅;   // two empty lists
        loop over the elements of MT (m)
        {
            if (m ∩ E_i ≠ ∅) {   // m has a common vertex with E_i
                L_1 ← L_1 ∪ m;   // m is a minimal transversal of E_i
            }
            else {
                S ← {one-size larger supersets of m using the vertices of E_i};
                L_2 ← L_2 ∪ S;
            }
        }
        if (L_1 ≠ ∅ and L_2 ≠ ∅) {
            cleanSupersets(L_1, L_2);   // removing non-minimal transversals from L_2
        }
        MT ← L_1 ∪ L_2;
    }
}

return MT;
```

out in the *Snow* algorithm in Section 4.2, our hypergraphs are usually very small, thus we did not have to face these efficiency problems. Experimental results show that *BergeOpt* provides a very efficient solution for the problem instance that we had to deal with in the *Snow* algorithm.

$\mathcal{E}_1 = ACD$	$\mathcal{MT}_{\mathcal{H}_1} = \{A, C, D\}$
$\mathcal{E}_2 = ACH$	$L_1 = \{A, C\}$
	$L_2 = \{\cancel{AD}, \cancel{CD}, DH\}$
	$\mathcal{MT}_{\mathcal{H}_2} = \{A, C, DH\}$
$\mathcal{E}_3 = BCD$	$L_1 = \{C, DH\}$
	$L_2 = \{AB, \cancel{AC}, AD\}$
	$\mathcal{MT}_{\mathcal{H}_3} = \{C, DH, AB, AD\}$
$\mathcal{E}_4 = DF$	$L_1 = \{DH, AD\}$
	$L_2 = \{CD, CF, \cancel{ABD}, ABF\}$
	$\mathcal{MT}_{\mathcal{H}_4} = \{DH, AD, CD, CF, ABF\}$
$\mathcal{E}_5 = FH$	$L_1 = \{DH, CF, ABF\}$
	$L_2 = \{ADF, \cancel{ADH}, \cancel{CDF}, \cancel{CDH}\}$
	$\mathcal{MT}_{\mathcal{H}_5} = \{DH, CF, ABF, ADF\} = \mathcal{MT}_{\mathcal{H}} = Tr(\mathcal{H})$

Table E.1: Incremental computation of the transversal hypergraph of $\mathcal{H} = \{ACD$, ACH, BCD, DF, $FH\}$ with the *BergeOpt* algorithm.

Bibliography

[Ber89] C. Berge. *Hypergraphs: Combinatorics of Finite Sets.* North Holland, Amsterdam, 1989.

[BTP+00a] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explor. Newsl.*, 2(2):66–75, 2000.

[BTP+00b] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining Minimal Non-Redundant Association Rules Using Frequent Closed Itemsets. In J. Lloyd *et al.*, editor, *Proc. of the Computational Logic (CL '00)*, volume 1861 of *LNAI*, pages 972–986. Springer, 2000.

[CG05] T. Calders and B. Goethals. Depth-first non-derivable itemset mining. In *Proc. of the SIAM Intl. Conf. on Data Mining (SDM '05), Newport Beach, USA*, Apr 2005.

[EG95] T. Eiter and G. Gottlob. Identifying the Minimal Transversals of a Hypergraph and Related Problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

[GD86] J. L. Guigues and V. Duquenne. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Mathématiques et Sciences Humaines*, 95:5–18, 1986.

[Goe03] B. Goethals. Survey on Frequent Pattern Mining. Manuscript, 2003.

[GW99] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations.* Springer, Berlin/Heidelberg, 1999.

[Kry98] M. Kryszkiewicz. Representative Association Rules. In *Proc. of the 2nd Pacific-Asia Conf. on Research and Development in Knowledge Discovery and Data Mining (PAKDD '98)*, pages 198–209, London, UK, 1998. Springer-Verlag.

[Kry02] M. Kryszkiewicz. Concise Representations of Association Rules. In *Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery*, pages 92–109, 2002.

[KS05] D. J. Kavvadias and E. C. Stavropoulos. An Efficient Algorithm for the Transversal Hypergraph Generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

[LFFM+03] A. Le Floc'h, C. Fisette, R. Missaoui, P. Valtchev, and R. Godin. JEN : un algorithme efficace de construction de générateurs pour l'identification des règles d'association. *Numéro spécial de la revue des Nouvelles Technologies de l'Information*, 1(1):135–146, 2003.

[Lux91] M. Luxenburger. Implications partielles dans un contexte. *Mathématiques, Informatique et Sciences Humaines*, 113:35–55, 1991.

[Pas00] N. Pasquier. Mining association rules using formal concept analysis. In *Proc. of the 8th Intl. Conf. on Conceptual Structures (ICCS '00)*, pages 259–264. Shaker-Verlag, Aug 2000.

[PBTL99a] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Closed set based discovery of small covers for association rules. In *Proc. 15emes Journees Bases de Donnees Avancees (BDA)*, pages 361–381, 1999.

[PBTL99b] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *Lecture Notes in Computer Science*, 1540:398–416, 1999.

[Pfa02] J. L. Pfaltz. Incremental Transformation of Lattices: A Key to Effective Knowledge Discovery. In *Proc. of the First Intl. Conf. on Graph Transformation (ICGT '02)*, pages 351–362, Barcelona, Spain, Oct 2002.

[PJ01] J. L. Pfaltz and R. E. Jamison. Closure Systems and their Structure. *Information Sciences*, 139(3–4):275–286, 2001.

[SN05] L. Szathmary and A. Napoli. CORON: A Framework for Levelwise Itemset Mining Algorithms. In B. Ganter, R. Godin, and E. Mephu Nguifo, editors, *Supplementary Proc. of the 3rd Intl. Conf. on Formal Concept Analysis (ICFCA '05), Lens, France*, pages 110–113, Feb 2005.

[SNK07] L. Szathmary, A. Napoli, and S. O. Kuznetsov. ZART: A Multifunctional Itemset Mining Algorithm. In *Proc. of the 5th Intl. Conf. on Concept Lattices and Their Applications (CLA '07)*, pages 26–37, Montpellier, France, Oct 2007.

[STB+02] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal. Computing Iceberg Concept Lattices with TITANIC. *Data and Knowledge Engineering*, 42(2):189–222, 2002.

[Sza06] L. Szathmary. *Symbolic Data Mining Methods with the Coron Platform.* PhD Thesis in Computer Science, University Henri Poincaré – Nancy 1, France, Nov 2006.

[Zak00] M. J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[ZG03] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge discovery and data mining (KDD '03)*, pages 326–335, New York, NY, USA, 2003. ACM Press.

[ZH02] M. J. Zaki and C.-J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *SIAM Intl. Conf. on Data Mining (SDM' 02)*, pages 33–43, Apr 2002.

[ZPOL97] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *Proc. of the 3rd Intl. Conf. on Knowledge Discovery in Databases*, pages 283–286, August 1997.

Index

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399