Efficient Vertical Mining of Frequent Closures and Generators

Laszlo Szathmary, Petko Valtchev, Amedeo Napoli, Robert Godin

Efficient Vertical Mining of Frequent Closures and Generators

Laszlo Szathmary[1], Petko Valtchev[1], Amedeo Napoli[2], and Robert Godin[1]

[1] Dépt. d'Informatique UQAM, C.P. 8888,
Succ. Centre-Ville, Montréal H3C 3P8, Canada
Szathmary.L@gmail.com, {valtchev.petko, godin.robert}@uqam.ca
[2] LORIA UMR 7503, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
napoli@loria.fr

Abstract. The effective construction of many association rule bases requires the computation of both frequent closed and frequent generator itemsets (FCIs/FGs). However, only few miners address both concerns, typically by applying levelwise breadth-first traversal. As depth-first traversal is known to be superior, we examine here the depth-first FCI/FG-mining. The proposed algorithm, *Touch*, deals with both tasks separately, i.e., uses a well-known vertical method, *Charm*, to extract FCIs and a novel one, *Talky-G*, to extract FGs. The respective outputs are matched in a post-processing step. Experimental results indicate that *Touch* is highly efficient and outperforms its levelwise competitors.

1 Introduction

The discovery of meaningful associations is a key data mining discipline [1]. An association miner typically proceeds in two steps: **(i)** extract all frequent patterns X of a database, and **(ii)** break each X into a *premise* Y, and a *conclusion* $X \setminus Y$ parts to form a rule $Y \rightarrow X \setminus Y$. Interestingness measures, such as support and confidence, are applied to prune the set of extracted association rules. However, the number of the remaining rules is usually way too high to be practical. As a remedy, various concise representations of the family of interesting association rules have been proposed [2,3,4], whereas others have been imported from related fields such as concept analysis [5,6]. A good survey can be found in [7].

In this paper we focus on the computation of frequent closed itemsets (FCIs) and frequent generators (FGs), on which are based the minimal non-redundant association rules ($\mathcal{MNR}$) for instance. Following [2], these are rules with the form $P \rightarrow Q \setminus P$, where $P \subset Q$, P is a *(minimal) generator* (a.k.a. key-set or free-set) and Q is a *closed itemset*. In other terms, in such rules the premise is minimal and the conclusion is maximal. As shown in [7], $\mathcal{MNR}$ is a *lossless*, *sound*, and *informative* representation of all valid rules. Moreover, further restrictions can be imposed on the rules in $\mathcal{MNR}$, leading to more compact representations such as the *generic basis* or the *proper basis* (see [7] for a complete list).

From a computational point of view, constructing $\mathcal{MNR}$ or its sub-structures requires the family of frequent closed itemsets (FCIs) and their generators (FGs),

and possibly the precedence order between FCIs. A few methods for extracting both FCIs and FGs have been published in the mining literature, e.g. *A-Close* [8] or *Titanic* [9]. Generators have been targeted within the concept analysis field as well [10], e.g. by the *Zart* algorithm [11]. Well-known FCI/FG-miners exclusively apply levelwise strategies, although the levelwise itemset miners are known to be outperformed by depth-first methods (e.g. *Charm* [12] and *Closet* [13]) on a broad range of dataset profiles, especially on dense ones. Hence the idea of designing a depth-first FCI/FG-miner. Our method, called *Touch*, tackles the component tasks separately: while the state-of-the-art algorithm *Charm* extracts FCIs, FG-mining is performed by *Talky-G*, an original method following a set inclusion-compliant order in the traversal of the itemset lattice. At a post-processing step of *Touch*, FGs are associated to their respective FCIs, thus providing the necessary starting point for the production of $\mathcal{MNR}$. Experimental results show that *Touch* outperforms two other efficient competitors, *A-Close* [8] and *Zart* [11], especially on dense and highly correlated datasets. Thus, the contributions of our study lay mainly in the design of an efficient method, *Touch*, for constructing the aforementioned rule bases. Additionally, *Talky-G* is a stand-alone algorithm for extracting FGs.

The paper is organized as follows. Section 2 lists the basic concepts of frequent itemset mining and presents the vertical depth-first mining strategy of *Charm*. In Section 3, we introduce a new FG-miner algorithm called *Talky-G*. The *Touch* algorithm that combines the results of *Charm* and *Talky-G* is introduced in Section 4. Finally, conclusions and future work are discussed in Section 5.

2 Background

Consider the following 5×5 sample dataset: $\mathcal{D} = \{(1,\ ACDE),\ (2,\ ABCDE),\ (3,\ AB),\ (4,\ D),\ (5,\ B)\}$. Throughout the paper, we will refer to this example as **"dataset $\mathcal{D}$"**.

2.1 Basic Concepts From Pattern 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 $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$. A set of items is called an *itemset*. Each transaction has a unique identifier (*tid*), and a set of transactions is called a *tidset*. The tidset of all transactions sharing a given itemset X is its *image*, denoted $t(X)$. For instance, the image of $\{A, B\}$ in $\mathcal{D}$ is $\{2, 3\}$, i.e., $t(AB) = 23$ in our separator-free set notation. The *length* of an itemset X is $|X|$, whereas an itemset of length i is called an i-itemset. The (absolute) *support* of an itemset X, denoted by $supp(X)$, is the size of its image, i.e. $supp(X) = |t(X)|$. Moreover, X is *frequent*, if its support is not less than a given *minimum support* threshold, min_supp, i.e. $supp(X) \geq min\_supp$. An equivalence relation is induced by t on the power-set of items $\wp(\mathcal{A})$: equivalent itemsets share the same image ($X \cong Z$ iff $t(X) = t(Z)$) [14]. In [12], a subsumption relation is defined as well: X *subsumes* Z, iff $X \supset Z$ and $supp(X) = supp(Z)$.

Consider the equivalence class of X, denoted $[X]$, and its extremal elements w.r.t. set inclusion. $[X]$ has a unique maximum (a *closed* itemset), and a set of minima (*generator* itemsets). The following definition thereof exploits the monotony of support upon set inclusion in $\wp(\mathcal{A})$.

Definition 1. *An itemset X is* closed (generator) *if it has no proper superset (subset) with the same support.*

A *closure* operator underlies the set of closed itemsets; it assigns to X the maximum of $[X]$ (denoted by $\gamma(X)$). Naturally, $X = \gamma(X)$ for closed X. Generators, a.k.a. *key-sets* in database theory, represent a special case of free-sets [15].

By Def. 1, if Z *subsumes* X, then Z cannot be a generator. The following property, which is widely known in the domain, generalizes this observation. It basically states the generator family is a downset within the Boolean lattice $\langle \wp(\mathcal{A}), \subseteq \rangle$.

Property 1. Given $X \subseteq \mathcal{A}$, if X is a generator, then $\forall Y \subseteq X$, Y is a generator, whereas if X is not a generator, $\forall Z \supseteq X$, Z is not a generator.

2.2 Vertical Itemset Mining

Miners from the literature, whether for plain FIs or FCIs, can be roughly split into breadth-first and depth-first ones. Breadth-first algorithms, more specifically the *Apriori*-like [1] ones, apply levelwise traversal of the pattern space exploiting the anti-monotony of the frequent status. Depth-first algorithms, e.g., *Closet* [13], in contrast, organize the search space into a prefix-tree (see Figure 1) thus factoring out the effort to process common prefixes of itemsets. Among them, the *vertical* miners use an encoding of the dataset as a set of pairs (item, tidset), i.e., $\{(i, t(i)) | i \in \mathcal{A}\}$, which reportedly allows the costly database re-scans to be avoided.

Eclat [16] was the first FI-miner to combine the vertical encoding with a depth-first traversal of a tree structure, called IT-tree, whose nodes are $X \times t(X)$ pairs. *Eclat* traverses the IT-tree in a depth-first manner in a pre-order way, from left-to-right [16,17] (see Figure 1).

Charm adapts the computing schema of *Eclat* to the exclusive construction of the FCIs [12]. The key challenges it faces are parsimony in generating the closedness candidates and efficiency of closedness tests on those candidates. To avoid examining the entire IT-tree of the FIs, *Charm* relies on a technique that, given a node $X \times t(X)$, looks for a Z subsuming X by combining X to Y, where $Y \times t(Y)$ is any right sibling node in the tree. Due to the specific traversal discipline, all Z are such that X is a prefix thereof (hence not all X expand to the closure of X).

To certify a candidate Z as closed, it should be checked that no set can subsume Z. Again, the traversal ensures that potential subsumers can only precede Z in the traversal-induced order on $\mathcal{A}$, hence at the moment Z is tested all of them are already processed and the actual closure is known. Thus, the closedness test amounts to a lookup in the working memory for a set Y such that

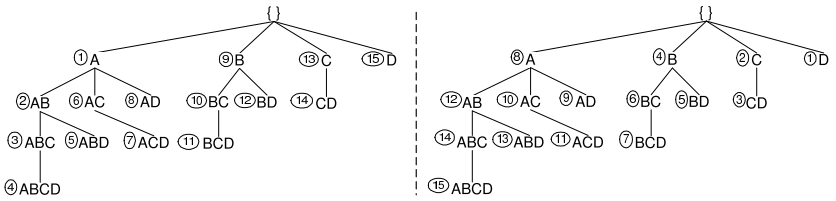


Fig. 1. Left: pre-order traversal with *Eclat*; **Right:** reverse pre-order traversal with *Talky-G*. The direction of traversal is indicated in circles

$t(Z) = t(Y)$, absence meaning that Z is the closure of $[Z]$. To avoid extensive search through the known part of the FCI family, *Charm* employs a hashing on $t(Z)$ (hashing is discussed in Section 3). For a more detailed presentation of vertical itemset miners, please refer to the report [18].

Charm is known to be one of the fastest FCI-miners, hence we adopt it in our own FCI/FG-miner. A natural question is whether a similar strategy could be defined for FGs. Several generators within a class mean that the pure image-based tests will not work as the existence of a generator with the same image does not disqualify a candidate X. Indeed, beside image equality, the existing generator must be a subset of X in order to invalidate X. Thus, X can only be certified "generator" if no stored generator is a subset thereof with the same image.

Moreover, hidden in the testing principles is a different traversal order: in fact, for the test to be effective, all subsets of a candidate X must be processed before X itself. Only then all generator subsets of X will be known and hence could be used in correctly (in)validating its own generator status. Although such a concern is typically addressed through a breadth-first traversal, the corresponding order could also be achieved with a depth-first one, yet with a different listing order on the items, as discussed in the next section.

3 Talky-G

Talky-G is a vertical FG-miner following a depth-first traversal of the IT-tree and a right-to-left order on sibling nodes.

3.1 Reverse Pre-Order Traversal

Talky-G applies an inclusion-compatible traversal: it goes down the IT-tree while listing sibling nodes from right-to-left and not the other way round as in *Eclat* and *Charm*. The resulting order on itemsets is exactly the order on their numerical representations (e.g., E is 1 and ABD is 26 in our dataset $\mathcal{D}$) that is frequently used in combinatorial generation algorithms. This strategy is used in *Next-Closure* [19] under the name of *lectic order*.

Algorithm 1 (main block of Talky-G):

```
 1) root.itemset ← ∅;   // root is an IT-node whose itemset is empty
 2) root.tidset ← {all transaction IDs};   // the empty set is included in every tr.
 3) loop over the vertical representation of the dataset (attr) {
 4)       if ((attr.supp ≥ min_supp) and (attr.supp < |O|)) {
 5)            // |O| stands for the total number of objects in the database
 6)            root.addChild(attr);   // attr is frequent and generator
 7)       }
 8) }
 9) loop over the children of root from right-to-left (child) {
10)       save(child);   // process the itemset
11)       extend(child);   // discover the subtree below child
12) }
```

Algorithm 2 ("extend(*curr*)" procedure of Talky-G):

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

 1) loop over the right siblings of curr from left-to-right (other) {
 2)       generator ← getNextGenerator(curr, other);
 3)       if (generator ≠ null) then curr.addChild(generator);
 4) }
 5) loop over the children of curr from right-to-left (child) {
 6)       save(child);   // process the itemset
 7)       extend(child);   // discover the subtree below child
 8) }
```

The authors of [20] explored that order for mining calling it *reverse pre-order*. They observed that for any itemset X its subsets appear in the IT-tree in nodes that lay either higher on the same branch as $(X, t(X))$ or on branches to the right of it. Hence, depth-first processing of the branches from right-to-left would perfectly match set inclusion, i.e., all subsets of X will be met before X itself.

While the algorithm in [20] extracts the so-called non-derivable itemsets, our *Talky-G* algorithm uses this traversal to find the set of frequent generators. See Figure 1 for a comparison of *Eclat* and its "reversed" version.

3.2 The Algorithm

Pseudo code. Algorithm 1 provides the main block of *Talky-G*. First, the IT-tree is initialized, which involves the creation of the root node, representing the empty set (of 100% support, by construction). *Talky-G* then transforms the layout of the dataset in vertical format, and inserts below the root node all 1-long

Algorithm 3 ("getNextGenerator(*curr*, *other*)" function of Talky-G):

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

```
 1) cand.tidset ← curr.tidset ∩ other.tidset;
 2) if (cardinality(cand.tidset) < min_supp) {   // test 1
 3)       return null;   // not frequent
 4) }
 5) // else, if it is frequent
 6) if ((cand.tidset = curr.tidset) or (cand.tidset = other.tidset)) {   // test 2
 7)       return null;   // not generator
 8) }
 9) // else, if it is a potential generator
10) cand.itemset ← curr.itemset ∪ other.itemset;
11) if (cand has a proper subset with the same support in the hash) {   // test 3
12)       return null;   // not generator
13) }
14) // if cand passed all the tests then cand is a frequent generator
15) return cand;
```

frequent itemsets. Such a set is an FG whenever its support is less than 100%. At this point, the dataset is no more needed since larger itemsets can be obtained as unions of smaller ones while for the images intersection must be used.

In the core processing, the `extend` procedure is called recursively for each child of the root in a right-to-left order. At the end, the IT-tree contains all FGs. The `addChild` procedure inserts an IT-node under a node. The `save` procedure stores an FG in a dedicated "list" data structure. The `extend` procedure (see Algorithm 2) discovers all FGs in the subtree of a node. First, new FGs are tentatively generated from the right siblings of the current node. Then, certified FGs are added below the current node and later on extended recursively in a right-to-left order.

The `getNextGenerator` function (see Algorithm 3) takes two nodes and returns a new FG, or "null" if no FG can be produced from the input nodes. First, a candidate node is created by taking the union of both itemsets and the intersection of their respective images. The input nodes are thus the candidate's *parents*. Then, the candidate undergoes a frequency test. If successful, the candidate is compared to its parents: if its tidset is equivalent to a parent tidset, then the candidate cannot be a generator. Even with both outcomes positive, an itemset may still not be a generator as a subsumed subset may lay elsewhere in the IT-tree. Due to the traversal strategy in *Talky-G*, all generator subsets of the current candidate are already detected and the algorithm has stored them in a "list" structure (see the `save` procedure). Thus, the ultimate test checks whether the candidate has a proper subset with the same support in this "list". A posi-

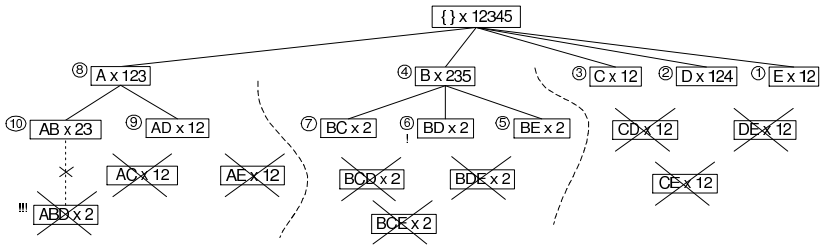


Fig. 2. Execution of *Talky-G* on dataset $\mathcal{D}$ with $min\_supp = 1$ (20%)

tive outcome disqualifies the candidate. The test exploits a hash structure that enhances the one used in *Charm* to perform the search for FG subsets efficiently.

Candidates surviving the final test are declared FG and added to the IT-tree. An unsuccessful candidate X is discarded which ultimately prevents any itemset Y having X as a prefix to be generated as candidate and hence substantially reduces the overall search space. When the algorithm stops, all frequent generators (and *only* frequent generators) are inserted in the IT-tree *and* in the "list" of generators.

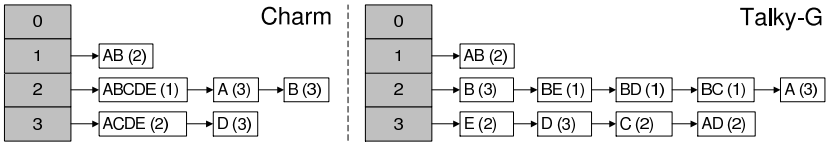
RUNNING EXAMPLE. The execution of *Talky-G* on dataset $\mathcal{D}$ with $min\_supp = 1$ (20%) is illustrated in Figure 2. Circles beside tree nodes show traversal ranks.

The IT-tree root node is first created and, as there is no full column in the dataset, all items become FGs, thus they are inserted below the root. These nodes are recursively extended in a *right-to-left* order. The rightmost node E has no right sibling, thus it cannot be extended. In contrast, D is extended with E. The result, DE, is discarded since of equal support to its parent E. C is extended with both D and E, but both CD and CE are discarded for this same reason. The processing of the B-branch, in short, yields FGs BC, BD, and BE. As to the 2-long supersets of A, AC and AE fail the second test because of C and E, respectively, while AB and AD succeed. The combination of the latter, ABD, although of strictly smaller support than its parents, fails because of a subsumed FG in the list (BD).

3.3 Fast Subsumption Checking

Recall that in the `getNextGenerator` function, when a new candidate itemset C is created, *Talky-G* checks whether C subsumes a previously found generator. If the test is positive, then clearly C is not a generator. Subsumption might seem expensive here, yet an efficient way to filter non-generators exists.

To that end the hash structure of *Charm* was adapted to the storage of frequent generators. Actually, *Talky-G* hashes the itemsets upon the tidset while storing generators with their support values. Consequently, equivalent itemsets get the same hash value and end up in the same list in the hash structure. In the testing of a candidate Z, the entire list corresponding to its hash value $h(Z)$



FCI (supp)	FGs
AB (2)	AB
$ABCDE$ (1)	BE; BD; BC
A (3)	A

FCI (supp)	FGs
B (3)	B
$ACDE$ (2)	E; C; AD
D (3)	D

Fig. 3. Top: hash tables for dataset $\mathcal{D}$ with $min\_supp = 1$. **Top left:** hash table of *Charm* containing all FCIs. **Top right:** hash table of *Talky-G* containing all FGs. **Bottom:** output of *Touch* on dataset $\mathcal{D}$ with $min\_supp = 1$

is retrieved. Whenever there is a set G in the list such that $supp(Z) = supp(G)$ and $Z \supset G$, Z is discarded, otherwise Z is declared an FG.

Example. Figure 3 (top right) depicts the hash structure of the IT-tree in Figure 2, which contains all FGs of $\mathcal{D}$. Each entry of the table is a list of (itemset, support) pairs. Here, the size of the hash table is set to four.

Assume we need to test ABD whose absolute support is 1. First, the sum of the tids in its tidset is 2 which, modulo the size of the hash table, is again 2. When traversing the list at position 2 of the table, B is more frequent than ABD while BE, although of the same support, is not a subset of ABD. Yet the next set, BD, is both of identical support and a proper subset of the candidate, hence ABD is discarded.

4 Touch

The algorithm has three steps: **(1)** extracting FCIs, **(2)** extracting FGs, and **(3)** associating FGs to their FCIs.

4.1 The Algorithm

While the above tasks **(1)** and **(2)** are solved by *Charm* and *Talky-G*, respectively, the appropriate associations between the respective outputs of both algorithms, task **(3)**, require some additional effort. Yet as both algorithms provide an additional hash structure (see Figure 3), the problem admits an efficient solution.

The exact method is based on a generalization of the storage strategy for FGs in *Talky-G* to FCIs. Indeed, observe that just as all FGs of the same equivalence class are forced to belong to the same list within the hash structure, their respective closure, whenever hashed to the FGs table would fall into the same list too. Conversely, if hashed against the FCI structure, each FG would fall

precisely in the list where its closure lays. In both cases, the same hash value is guaranteed by the shared image.

Yet an effective re-hashing of FCIs or FGs is not necessary: with tables properly sized, i.e. of the *same dimension*, and with *identical hash functions*, the lists from both tables can be directly matched. To that end, FCIs from the list at position n in the closure table should be compared only to FGs from the list at the same n position in the generator table.

Pseudo code. The algorithm *Touch* starts by calling *Charm* and *Talky-G* and taking over their hash structures. Then, *Touch* matches the two hash tables: for each FCI X, it looks up in the hash table of *Talky-G* at the same index position all subsets of X that have the same support.

EXAMPLE. Consider the hash structures of *Charm* and *Talky-G* in Figure 3. Assume the generators of the closed itemset $ACDE$ are sought. As $ACDE$ is stored at position 3 in the hash structure of *Charm*, its generators will also be at position 3 in the hash structure of *Talky-G*. Three members of the corresponding list are subsumed by $ACDE$: E, C, and AD, hence they are the target generators. For the FCI A, the only subsumed FG of the list at index 2 is A, meaning that A is the unique member of its equivalence class $[A]$. The output of *Touch* is shown in Figure 3 (bottom).

4.2 Experimental Results

We evaluated *Touch* against *Zart* [11] and *A-Close* [8]. All the algorithms were implemented in Java in the CORON data mining platform [21].[1] The experiments were carried out on a bi-processor Intel Quad Core Xeon 2.33 GHz machine running under Ubuntu GNU/Linux with 4 GB of RAM. For the experiments we have used the following datasets: T20I6D100K[2], C20D10K, and MUSHROOMS[3]. The T20 is a sparse dataset, constructed according to the properties of market basket data that are typical weakly correlated data. The C20 is a census dataset from the PUMS[4] sample file, while the MUSHROOMS describes mushrooms characteristics. The last two are highly correlated datasets.

Table 1 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 T20, almost all frequent itemsets are closed and generators at the same time. It means that most equivalence classes are singletons. It is known that *Charm* is less efficient on sparse datasets. This is due to the fact that *Charm* performs four tests on candidates for reducing the IT-tree. However, in sparse datasets the number of FCIs is almost

[1] http://coron.loria.fr
[2] http://www.almaden.ibm.com/software/quest/Resources/
[3] http://kdd.ics.uci.edu/
[4] Public Use Microdata Sample

Table 1. Detailed execution times of *Touch* and data-related statistics: number of FCIs, of of FGs, and of FIs (for comparison only, *Touch* does not work with all FIs), ratio of FCIs to FIs, ratio of FGs to FIs.

min_supp	execution time (sec.)				# FCIs	# FGs	(# FIs)	$\frac{\#FCIs}{\#FIs}$	$\frac{\#FGs}{\#FIs}$
	get FCIs (*Charm*)	get FGs (*Talky-G*)	associate FCIs and FGs	**total time** (with I/O)					
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%

equivalent to the number of FIs, thus the search space cannot be reduced significantly. *Talky-G* is also less efficient on sparse datasets. However, in dense, highly correlated datasets (C20 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 very efficiently in all cases. That is, the association step gives absolutely no overhead to *Touch*.

Table 2 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* [14], i.e. first it finds all FIs using pattern-counting inference, 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. The way *A-Close* computes the closures of generators is quite expensive because of the huge number of intersection operations. *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 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 are sometimes more suitable for sparse datasets.

Table 2. Response times of *Touch*, compared to *Zart* and *A-Close*

T20I6D100K			
min.	execution time (sec.)		
supp.	Touch	Zart	A-Close
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			
min.	execution time (sec.)		
supp.	Touch	Zart	A-Close
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			
min.	execution time (sec.)		
supp.	Touch	Zart	A-Close
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

5 Conclusions and Future Work

Mining FGs has so far been done largely in a levelwise manner as the breadth-first traversal fits the down-set structure of the FG family. Yet depth-first algorithms have shown superior efficiency in many situations, whence the motivation of our study of depth-first FCI/FG-mining.

As a contribution to this problem, we presented *Touch*, an algorithm that splits the general problem into three tasks: **(1)** FCI-mining, **(2)** FG-mining, and **(3)** association of FGs to their closures (FCIs). While **(1)** is solved by reusing an existing algorithm, *Charm*, the two others generate innovative solutions. Hence the *Talky-G* vertical FG-miner used in **(2)** is an original contribution on its own. As all three solutions are highly optimized, the algorithm performs well against comparable levelwise miners. Numerous concise representations of valid association rules can be readily derived from the method's output.

The study led to a range of exciting questions that are currently investigated. Thus, from an algorithmic point of view, it would be interesting to merge steps **(1)** and **(2)**, e.g. by using the output of *Talky-G* (i.e., the IT-tree of all FGs) as a starting point for the FCI-mining, hence avoiding step **(3)**. A further challenge lays in the computation of the FCI precedence order that underlies some of the association rule bases from the literature. We plan to join *Touch* with our previous algorithm *Snow* [22]. *Snow* allows us to easily compute precedence order using hypergraph theory. Once we have a concept lattice whose nodes are labeled with generators, it is possible to produce all kinds of $\mathcal{MNR}$ rules, including approximate association rules too.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proc. of the 20th Intl. Conf. on Very Large Data Bases (VLDB '94), Morgan Kaufmann Publishers Inc. (1994) 487–499
2. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining Minimal Non-Redundant Association Rules Using Frequent Closed Itemsets. In: Proc. of the 1st Intl. Conf. on Computational Logic (CL '00). Volume 1861 of LNAI., Springer (2000) 972–986
3. Kryszkiewicz, M.: Representative Association Rules. In: Proc. of the 2nd Pacific-Asia Conf. on Research and Development in Knowledge Discovery and Data Mining (PAKDD '98). Volume 1394 of LNCS., Springer-Verlag (1998) 198–209

4. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Closed Set Based Discovery of Small Covers for Association Rules. In: Proc. 15emes Journees Bases de Donnees Avancees (BDA). (1999) 361–381
5. Duquenne, V.: Contextual Implications Between Attributes and Some Representational Properties for Finite Lattices. In: Beitraege zur Begriffsanalyse, B.I. Wissenschaftsverlag, Mannheim (1987) 213–239
6. Luxenburger, M.: Implications partielles dans un contexte. Mathématiques, Informatique et Sciences Humaines **113** (1991) 35–55
7. Kryszkiewicz, M.: Concise Representations of Association Rules. In: Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery. (2002) 92–109
8. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering Frequent Closed Itemsets for Association Rules. In: Proc. of the 7th Intl. Conf. on Database Theory (ICDT '99). (1999) 398–416
9. Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., Lakhal, L.: Computing Iceberg Concept Lattices with TITANIC. Data and Knowledge Engineering **42**(2) (2002) 189–222
10. Valtchev, P., Missaoui, R., Godin, R.: Formal Concept Analysis for Knowledge Discovery and Data Mining: The New Challenges. In: Proc. of the 2nd Intl. Conf. on Formal Concept Analysis, Springer Verlag (2004) 352–371
11. Szathmary, L., Napoli, A., Kuznetsov, S.O.: ZART: A Multifunctional Itemset Mining Algorithm. In: Proc. of the 5th Intl. Conf. on Concept Lattices and Their Applications (CLA '07). (2007) 26–37
12. Zaki, M.J., Hsiao, C.J.: ChARM: An Efficient Algorithm for Closed Itemset Mining. In: SIAM Intl. Conf. on Data Mining (SDM' 02). (2002) 33–43
13. Pei, J., Han, J., Mao, R.: CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. (2000) 21–30
14. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining Frequent Patterns with Counting Inference. SIGKDD Explor. Newsl. **2**(2) (2000) 66–75
15. Boulicaut, J.F., Bykowski, A., Rigotti, C.: Free-Sets: A Condensed Representation of Boolean Data for the Approximation of Frequency Queries. Data Mining and Knowledge Discovery **7**(1) (2003) 5–22
16. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New Algorithms for Fast Discovery of Association Rules. In: Proc. of the 3rd Intl. Conf. on Knowledge Discovery in Databases. (1997) 283–286
17. Zaki, M.J.: Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering **12**(3) (2000) 372–390
18. Szathmary, L., Valtchev, P., Napoli, A.: Efficient Mining of Frequent Closures with Precedence Links and Associated Generators. Research Report RR-6657, INRIA (2008) (http://hal.inria.fr/inria-00322798/en).
19. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Berlin/Heidelberg (1999)
20. Calders, T., Goethals, B.: Depth-First Non-Derivable Itemset Mining. In: Proc. of the SIAM Intl. Conf. on Data Mining (SDM '05), Newport Beach, USA. (2005)
21. Szathmary, L.: Symbolic Data Mining Methods with the Coron Platform. PhD Thesis in Computer Science, Univ. Henri Poincaré – Nancy 1, France (2006)
22. Szathmary, L., Valtchev, P., Napoli, A., Godin, R.: Constructing Iceberg Lattices from Frequent Closures Using Generators. In: Proc. of the 11th Intl. Conf. on Discovery Science (DS '08). Volume 5255 of LNAI., Springer (2008) 136–147