Available online at www.sciencedirect.com
ScienceDirect
Discrete Applied Mathematics 156 (2008) 924–949
DISCRETE APPLIED MATHEMATICS
www.elsevier.com/locate/dam

A framework for incremental generation of closed itemsets

Petko Valtchev[a,c,*], Rokia Missaoui[b], Robert Godin[c]

[a] *DIRO, Université de Montréal, CP 6128, Succ. Centre-Ville, Montréal, Qué., Canada H3C 3J7*
[b] *Département d'informatique et d'ingénierie, UQO, C.P. 1250, succursale B, Gatineau, Qué., Canada J8X 3X7*
[c] *Département d'Informatique, UQAM, C.P. 8888, succ. "Centre Ville", Montréal, Qué., Canada H3C 3P8*

Received 2 May 2002; received in revised form 24 June 2007; accepted 11 July 2007
Available online 1 October 2007

Abstract

Association rule mining from a transaction database (TDB) requires the detection of frequently occurring patterns, called frequent itemsets (*FIs*), whereby the number of *FIs* may be potentially huge. Recent approaches for *FI* mining use the *closed* itemset paradigm to limit the mining effort to a subset of the entire *FI* family, the frequent *closed* itemsets (*FCIs*). We show here how *FCIs* can be mined incrementally yet efficiently whenever a new transaction is added to a database whose mining results are available. Our approach for mining *FIs* in dynamic databases relies on recent results about lattice incremental restructuring and lattice construction. The fundamentals of the incremental *FCI* mining task are discussed and its reduction to the problem of lattice update, via the *CI* family, is made explicit. The related structural results underlie two algorithms for updating the set of *FCIs* of a given TDB upon the insertion of a new transaction. A straightforward method searches for necessary completions throughout the entire *CI* family, whereas a second method exploits lattice properties to limit the search to *CIs* which share at least one item with the new transaction. Efficient implementations of the parsimonious method is discussed in the paper together with a set of results from a preliminary study of the method's practical performances.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Frequent closed itemsets; Incremental data mining; Galois lattices; Formal concept analysis

1. Introduction

Association rule mining from a transaction database (TDB) [2] is a classical data mining topic, whereby the most challenging problem is the detection of frequent patterns (*itemsets*) in the transaction set [1,6,22]. A major difficulty with association rules is the prohibitive number of frequent itemsets (FIs) (and hence rules) that can be generated even from a reasonably sized data set. The frequent *closed* itemsets (*FCIs*) research topic [27,29,43,44,46,47] constitutes a promising solution to the problem of reducing the number of the reported associations. Yet another difficulty arises with dynamic databases where the transaction set is frequently updated. Although the necessity of processing volatile data in an incremental manner has been repeatedly emphasized in the general data mining literature (e.g., in [20]), a few incremental mining algorithms have been reported so far [3,13,14,17,30,34]. All these studies

* Corresponding author. DIRO, Université de Montréal, CP 6128, Succ. Centre-Ville, Montréal, Qué., Canada H3C 3J7.
E-mail address: Petko.Valtchev@UMontreal.CA (P. Valtchev).

0166-218X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2007.08.004

have highlighted the need for some additional storage and/or some supplementary database passes to cope with data set evolution.

Our own approach to incremental *FI* mining is motivated by the belief that *FCIs* provide the key to compact rule sets and low storage requirements. Therefore, we have been investigating the potential benefits of using concept analysis as a formal basis for the resolution of the *FCIs* mining problem. The ultimate goal of our study is the definition of a complete framework for that problem, i.e., a consistent body of theoretical knowledge that underlies a set of high-level algorithms which are in turn realized by effective mining tools. It is to be completed by experimental evidence about its strength drawn from practical studies on association mining test-beds.

In this paper, we present an initial, but complete version of our framework: the core theoretical part is presented together with some further structural results that pertain to a minimalist approach toward the incremental *FI* generation. The approach consists in reducing the dynamic *FI* mining problem to a one-by-one insertion of transactions into an available database. The resulting incremental *FI* problem is successfully mapped into a lattice maintenance problem [35] and the underlying theory is adapted to the association rule paradigm so that efficient algorithms can be designed based on the structural properties. First, we establish the correspondence between basic elements of both frameworks. Then, we present a way to transform a recent enhancement of a classical lattice algorithm into an *FCI*-miner. As the resulting approach relies on extensive exploration of the temporary *FCI* family upon each update, we investigate possible pruning strategies to reduce the set of examined *FCIs*. To that end, we provide a set of characteristic properties of the involved lattice substructures and embody them into a new method that only processes *relevant* lattice nodes. Efficient implementation of the method yields an incremental *FCI*-miner whose performances are compared to those of a known batch procedure [29]. The paper starts with summaries on association rule mining (Section 2) and on lattices and dedicated algorithms (Section 3). The theoretical foundations of our framework are presented in Section 4. Our generic mining approach, Galicia, is outlined in Section 5 together with two methods, a straightforward one and an optimized one. Section 6 presents an implementation of the second method. Section 7 lists related work and Section 8 discusses theoretical worst-case complexity as well as preliminary results about the practical performances of our methods.

2. Association rule mining problem

The association rule mining problem targets all strong and significant associations between items within a TDB. Let $\mathscr{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct items. A transaction T contains a set of items in $\mathscr{I}$, and has an associated unique identifier called *TID*. A subset X of $\mathscr{I}$ where $k = |X|$ is referred to as a k-itemset (or simply an itemset), and k is called the length of X. A *TDB*, say $\mathscr{D}$, is a set of transactions. The number of transactions in $\mathscr{D}$ that contains an itemset X is called the *absolute support* of X, whereas the fraction of these is called its *relative support* (both denoted by $supp(X)$). For example, the support of *efh* in Table 1 is 33%. Thus, an itemset is frequent (or large) when $supp(X)$ reaches at least a user-specified minimum threshold called *minsupp*.

As a running example, let us consider Table 1 which shows a supermarket database with a sample set of transactions $\mathscr{D} = \{1, \ldots, 9\}$ involving items from the set $\mathscr{I} = \{a, \ldots, h\}$. The itemsets whose support is higher than 30% of $|\mathscr{D}|$ are given in Table 1(B).

2.1. Association rule generation

An association rule is an implication of the form $X \Rightarrow Y$, where X and Y are subsets of $\mathscr{I}$, and $X \cap Y = \emptyset$ (e.g., $e \Rightarrow h$). The support of a rule $X \Rightarrow Y$ is defined as $supp(X \cup Y)$ while its confidence is computed as the ratio $supp(X \cup Y)/supp(X)$. For example, the support and confidence of $e \Rightarrow h$ are 33% and 75%, respectively.

The problem of mining association rules with given minimum support and confidence (called *minconf*) can be split into two steps:

- Detecting all *FIs*, i.e., having support $\geqslant$*minsupp*.
- Generating *strong* association rules from large itemsets, i.e., with confidence $\geqslant$*minconf*.

The second step is relatively straightforward. However, the first step presents a great challenge because the set of FIs may grow exponentially with $|\mathscr{I}|$.

Table 1

TID	Items
(A) A sample transaction database	
1	*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*
2	a, b, c, e, f
3	**c**, **d**, **f**, **g**, **h**
4	*e*, *f*, *g*, *h*
5	*g*
6	*e*, *f*, *h*
7	*a*, *b*, *c*, *d*
8	*b*, *c*, *d*
9	*d*

i-set	Supp.	i-set	Supp.	i-set	Supp.
(B) Itemsets *X* of support greater than 30%					
a	3	*b*	4	*c*	5
d	5	*e*	4	*f*	5
g	4	*h*	4		
ab	3	*ac*	3	*bc*	4
bd	3	*cd*	4	*cf*	3
ef	4	*eh*	3	*fg*	3
fh	4	*gh*	3		
abc	3	*bcd*	3	*efh*	3
fgh	3				

2.2. Frequent closed itemsets

Since the most time consuming operation in association rule generation is the computation of *FIs*, recent work has concentrated on the benefits of keeping *FCIs* only [27,46]. An itemset X is closed if adding an arbitrary item i from $\mathscr{I} - X$ to X yields an itemset of strictly lower support (see Section 3 for a detailed discussion):

$$\forall i \in \mathscr{I} - X, \quad supp(X \cup \{i\}) < supp(X).$$

The following table provides the set of all *CIs*, both frequent (support greater than 30%) and infrequent ones, relative to the TDB of the previous example (see Table 1).

Set of CI	Closed itemsets
FCI	*c*, *d*, *g*, *f*, *bc*, *cd*, *cf*, *ef*, *fh*, *abc*, *bcd*, *efh*, *fgh*
CI –*FCI*	*abcd*, *abcef*, *cdfgh*, *efgh*, *abcdefgh*

A key property in the *CI* framework states that any itemset has the same support as its closure, hence it is just as frequent. For example, the closure of the itemset *b* is *bc* and both have a support of 4. Previous work [20,27] has shown that *FCIs* constitute a compact lossless encoding of all *FIs* whose retrieval requires no further access to the TDB. Moreover, similar representations for association rules can be derived from either the *FCIs* [33] or the lattice of *CIs* [46].

2.3. Incremental generation

In real-life situations, data sets tend to be very large and volatile. One way to deal with these features in data mining is to process data incrementally yet efficiently. In association rule mining, this means that the available *FCIs* need to be updated without restarting the mining task from scratch each time new transactions are added.

As an introductory example, let us consider the following data set. Assume that the *initial* TDB, $\mathscr{D}$, includes only transactions $\{1, 2, 4, \ldots, 9\}$ while the *increment* is made of transaction 3. Thus, the augmented TDB $\mathscr{D}^+$ is the union of $\mathscr{D}$ and the increment. The following table provides the sets of *CI* for both the initial TDB and the increment.

Set of CI	Closed itemsets
CI	*d*, *g*, *bc*, *ef*, *abc*, *bcd*, *efh*, *abcd*, *abcef*, *efgh*, *abcdefgh*
Increment	*c*, *f*, *cd*, *cf*, *fh*, *fgh*, *cdfgh*

A batch algorithm would have to start mining the *CIs* in $\mathscr{D}^+$ from scratch. In contrast, an incremental method will use both the new transaction and the existing set of *CIs* from $\mathscr{D}$ to compute the new *CIs* and thus compose the entire family corresponding to $\mathscr{D}^+$.

2.4. Mining FCI within a dynamic database

We argue that there is a significant potential to combine the benefits from a flexible computation method with a theoretical framework that insures more compact results. Just like in the general case of *FIs*, there is clearly a room for incremental techniques which maintain efficiently the *FCIs* upon the insertion of new transactions.

In its most general form, the problem of dynamic *FCI* mining would require the "merge" of two families of *FCIs* corresponding to two TDB that share the same global set of items $\mathscr{I}$. In terms of concept lattices, this could be modeled as the assembly of two upper semi-lattices (obtained from both *FCI* families). In a recent work, we already addressed the problem of assembling entire Galois lattices [42] and the extension of the proposed approach on truncated lattices, also called *iceberg* lattices [32], is under investigation.

However, as a starting point for the present study of connections between maintaining a Galois lattice and computing dynamically a *FCI* family, we have chosen a less complex version of both problems which only considers adding one transaction to the data set. This allows a body of available results about lattices to be explored and enhanced to yield the definition of an complete framework for *FCI* mining. Indeed, the results presented here already provide a solution for the general problem of dynamic *FCI* mining. Moreover, following our work on lattices, we are currently generalizing them to the multi-transaction case.

In summary, the rest of the paper presents an incremental *FCI* mining approach, which, to the best of our knowledge, has no equivalent in the literature.

3. Background on Galois/concept lattices

We recall basic results on Galois lattice [4] and formal concept analysis (FCA) [19,45], which constitute the basis of our approach toward incremental generation of *FCIs*.

3.1. The basics of ordered structures

$P = \langle G, \leqslant_P \rangle$ is a *partial order* (poset) over a *ground* set G and a binary relation $\leqslant_P$ if $\leqslant_P$ is reflexive, antisymmetric and transitive. For a pair of elements a, b in G, if $b \leqslant_P a$ we shall say that *a succeeds* (is greater than) b and, inversely, *b precedes a*. If neither $b \leqslant_P a$ nor $a \leqslant_P b$, then a and b are said to be *incomparable*. All common successors (predecessors) of a and b are called *upper* (*lower*) *bounds*. The *precedence* relation $\prec_P$ in P is the transitive reduction of $\leqslant_P$, i.e., $a \prec_P b$ if $a \leqslant_P b$ and all c such that $a \leqslant_P c \leqslant_P b$ satisfy $c = a$ or $c = b$. Given such a pair, a will be referred to as an *immediate predecessor* of b and b as an *immediate successor* of a. Usually, P is represented by its *covering graph* $Cov(P) = (G, \prec_P)$. In this graph, each element a in G is connected to any of its immediate predecessors and immediate successors. A poset is visualized by its Hasse diagram, that is the line diagram of the covering graph where an element is located "below" all its successors.

A lattice $L = \langle G, \leqslant_L \rangle$ is a poset where any pair of elements a, b has a unique *greatest lower bound* (GLB) and a unique *least upper bound* (LUB). GLB and LUB define binary operators on G called, respectively, *join* ($a \bigvee_L b$) and *meet* ($a \bigwedge_L b$). In a complete lattice L, for all finite $A \subseteq G$, $\bigvee_L A$ and $\bigwedge_L A$ exist. In particular, for finite G,

	a	b	c	d	e	f	g	h
1	X	X	X	X	X	X	X	X
2	X	X	X		X	X		
4					X	X	X	X
5							X	
6					X	X		X
7	X	X	X	X				
8		X	X	X				
9				X				
3			X	X		X	X	X

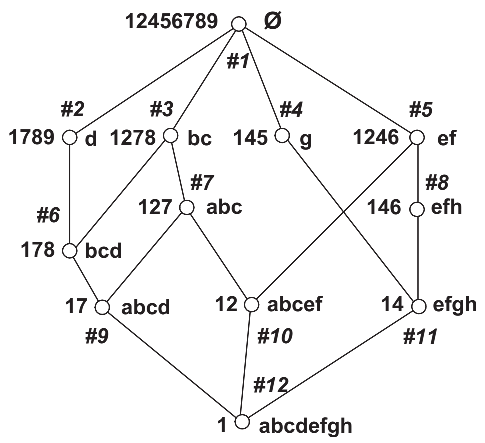


Fig. 1. Left: Binary table $\mathcal{K} = (O = \{1, 2, 4, \ldots, 9\}, A = \{a, b, \ldots, h\}, I)$ and the object 3 to be added. Right: The Hasse diagram of the Galois lattice derived from $\mathcal{K}$.

there are unique maximal (top, $\top$) and minimal (bottom, $\bot$) elements in the lattice. A structure with only one of the above operations is called semi-lattice, e.g., the existence of a unique GLB for any couple (set) of elements implies a (complete) *meet* semi-lattice structure.

3.2. Fundamental results about Galois/concept lattices

The focus is on the partially ordered structure [15] induced by a binary relation I over a pair of sets, O (*objects*) and A (*attributes*). Already discussed in the work of Öre [26] and Birkhoff [8], nowadays the structure is known as the *Galois lattice* [4] or, more popularly, *concept lattice* [45]. For example, Fig. 1 on the left shows the binary relation $\mathcal{K} = (O, A, I)$ (or *context*) drawn from the TDB of Table 1 where transactions are taken as objects, items as attributes, and oIa is to be read as "transaction o has the item a". Two *derivation* operators, f and g, summarize the links between object and attribute subsets induced by I.

Definition 1. The function f maps a set of objects into a set of common attributes, whereas g is the dual for attribute sets:

- $f : \wp(O) \to \wp(A)$, $f(X) = X' = \{a \in A | \forall o \in X, oIa\}$,
- $g : \wp(A) \to \wp(O)$, $g(Y) = Y' = \{o \in O | \forall a \in Y, oIa\}$.

For example, w.r.t. the table in Fig. 1, $f(14) = efgh$ and $g(abc) = 127$. Hereafter, following a standard FCA notation, both f and g are expressed by $'$, whereas sets are given in a separator-free form. Both $'$ operators define a *Galois connection* between the Boolean lattices 2^O and 2^A. Consequently, the compound operators $''$ represent *closure* operators over $\wp(O)$ and $\wp(A)$, respectively. This means, in particular, that $Z \subseteq Z''$ and $(Z'')'' = Z''$ for any $Z \in \wp(A)$ or $Z \in \wp(O)$. Thus, each of the $''$ operators induces a family of *closed* subsets, further denoted $\mathcal{C}^a_{\mathcal{K}}$ (from *attributes*) and $\mathcal{C}^o_{\mathcal{K}}$ (from *objects*), respectively. With the example in Fig. 1, the attribute sets in $\mathcal{C}^a_{\mathcal{K}}$, represent the *CIs* in the TDB $\mathcal{D}$ as described in the previous section. It is noteworthy that $\mathcal{C}^a_{\mathcal{K}}$ induces an equivalence relation on $\wp(A)$ whereby all the closed attribute sets are the maxima of their respective equivalence classes. A well-known result states that $\mathcal{C}^o_{\mathcal{K}}$ and $\mathcal{C}^a_{\mathcal{K}}$, ordered by set-theoretical inclusion, form two complete semi-lattices which are: (i) sub-semi-lattices of 2^O and 2^A, respectively, (ii) complete lattices, since finite semi-lattices. Moreover, both $'$ mappings between $\mathcal{C}^o_{\mathcal{K}}$ and $\mathcal{C}^a_{\mathcal{K}}$ are bijective and represent dual isomorphisms between the underlying lattices. This yields a unique structure comprising all pairs of mutually corresponding closures.

Definition 2. A *concept* is a pair of sets (X, Y) where $X \in \wp(O)$, $Y \in \wp(A)$, $X = Y'$ and $Y = X'$. X is called the *extent* and Y the *intent* of the concept.

For example, (178, *bcd*) is a concept, but (16, *e*) is not. In mining terms, a concept comprises closed itemset *Y* and the (closed) set *X* of all transactions including *Y*, i.e., the supporting *TID* set. Furthermore, the set $\mathscr{C}_{\mathscr{K}}$ of all concepts of $\mathscr{K} = (O, A, I)$ is partially ordered by intent/extent inclusion:

$$(X_1, Y_1) \leqslant_{\mathscr{K}} (X_2, Y_2) \quad \Leftrightarrow \quad X_1 \subseteq X_2, Y_2 \subseteq Y_1.$$

The partial order $\langle \mathscr{C}_{\mathscr{K}}, \leqslant_{\mathscr{K}} \rangle$ actually forms a complete lattice, called *Galois* or *concept* lattice whereby the lattice operators are given in the following property (see [4,45]).

Theorem 3. *$\mathscr{L} = \langle \mathscr{C}_{\mathscr{K}}, \leqslant_{\mathscr{K}} \rangle$ is a complete lattice with* join *and* meet *defined as follows*:

- $\bigvee_{i=1}^{k}(X_i, Y_i) = ((\bigcup_{i=1}^{k} X_i)'', \bigcap_{i=1}^{k} Y_i)$,
- $\bigwedge_{i=1}^{k}(X_i, Y_i) = (\bigcap_{i=1}^{k} X_i, (\bigcup_{i=1}^{k} Y_i)'')$.

The Hasse diagram of the lattice $\mathscr{L}$ drawn from $\mathscr{K} = (\{1, 2, 4, \dots, 9\}, A, I)$ is shown on the right side of Fig. 1 where itemsets and transaction sets are drawn on both sides of a node representing a concept. For example, the join and the meet of the concepts $c_1 = (178, bcd)$ and $c_2 = (127, abc)$ are (1278, *bc*) and (17, *abcd*), respectively.

The lattice provides a hierarchical organization of all concepts which may be used to speed-up their computation and subsequent retrieval. This is particularly useful when the set of concepts is to be generated incrementally, a problem which is addressed in the remainder of this section.

3.3. Incremental lattice update

The incremental construction of a lattice $\mathscr{L}$ is an iterative process starting by $\mathscr{L}_0 = \langle \{(\emptyset, A)\}, \emptyset \rangle$. At the *i*th iteration, the lattice $\mathscr{L}_i$ corresponding to $\mathscr{K}_i = (O_i = \{o_1, \dots, o_i\}, A, I \cap O_i \times A)$ is obtained by incorporating the object o_i into $\mathscr{L}_{i-1}$. To that end, a set of modifications of the data structure storing $\mathscr{L}_{i-1}$, typically the graph of its Hasse diagram, are performed.

The generic incremental problem amounts to incorporating the structures generated by a new object *o* into the lattice $\mathscr{L}$ of a context $\mathscr{K} = (O, A, I)$. The original solution in [21] exploits the basic fact that a closure family is itself closed under set intersection [4]. Thus, the restructuring boils down to closing the family $\mathscr{C}^a_{\mathscr{K}} \cup \{o'\}$ for intersection. To that end, all possible intersections of o' with the intents in $\mathscr{C}^a_{\mathscr{K}}$ are produced. Part of these are new, i.e., represent non-closed sets in $\mathscr{K}$, whereas the remainder are already in $\mathscr{C}^a_{\mathscr{K}}$. Consequently, one of the goals is to identify all concepts whose intents correspond to new closures (in the family $\mathscr{C}^{a+}$ of the extended context $\mathscr{K}^+ = (O \cup \{o\}, A, I \cup \{o\} \times o')$) and integrate them into (the data structure storing) $\mathscr{L}$.

The reshuffling of $\mathscr{L}$ is based on the recognition of two subsets of concepts, the *modified* and the *genitors*, denoted by $\mathbf{M}(o)$ and $\mathbf{G}(o)$, respectively. Modified concepts correspond to intersections that are already closed in $\mathscr{K}$ and transform into homologous concepts in $\mathscr{L}^+$ after a minimal fix: their extents expand to include *o*. The genitor concepts—a hint to their parental role—serve both as seeds and as milestones for the creation of new concepts. Thus, on the one hand, both the extent and the intent of a new concept are derived from the corresponding elements of its genitor (e.g., new extent is the genitor one plus *o*). On the other hand, a genitor marks the place of a new concept in the lattice cover relation. Finally, the *old*, or *unchanged*, concepts (denoted by $\mathbf{U}(o)$), i.e., neither modified nor genitors, are identical in both $\mathscr{L}$ and $\mathscr{L}^+$.

In summary, a reconstruction algorithm needs to identify $\mathbf{M}(o)$ and $\mathbf{G}(o)$ within $\mathscr{L}$, physically create the new concepts ($\mathbf{N}^+(o)$), and subsequently integrate these in the existing lattice structure. A theoretical framework for these tasks has been initially proposed in [41] and then enhanced in [37] (see Section 4.1 for a summary).

Algorithm 1 provides an overview of the basic steps from the incremental method in [21]. Hereafter, details about the lattice order updates (primitive UPDATE-ORDER) are skipped as irrelevant. Technically speaking, the concepts are first sorted by increasing intent sizes thus yielding a (decreasing) linear extension of the lattice order (line 3). Each concept is then examined in order to identify its actual category (lines 4–11). Modified concepts have intents that are included in the description of the new object, i.e., $\{o\}'$ (line 6). A remaining concept is potentially old unless the corresponding intersection with $\{o\}'$ is met for the first time, in which case the concept is a genitor and a new concept is created. A property which remains implicit in the code states that a genitor is the *maximum* of the set of concepts which generate

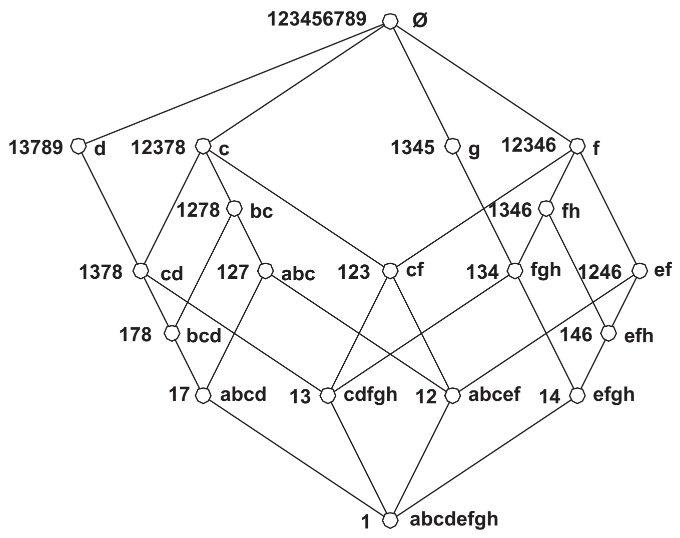


Fig. 2. The Hasse diagram of the lattice derived from $\mathscr{K}$ with $O = \{1, 2, 3, \ldots, 9\}$.

a new intent (since the first one to meet down the decreasing order). The previously mentioned dependancy between the respective intent and extent in a genitor and in its corresponding new may be observed here in action (line 10).

Algorithm 1. Update of a Galois (concept) lattice upon an insertion of a new object.

```
1:  procedure ADD-OBJECT(In: $\mathscr{L}$ a lattice, $o$ an object)
2:
3:  SORT($\mathscr{L}$)   {in ascending order of intent sizes}
4:  for all $\bar{c}$ in $\mathscr{L}$ do
5:     if $Intent(\bar{c}) \subseteq \{o\}'$ then
6:        ADD($Extent(\bar{c}), o$)   {($\bar{c}$) is a modified concept}
7:     else
8:        $Int \leftarrow Intent(\bar{c}) \cap \{o\}'$   {($\bar{c}$) is an old concept}
9:        if not $(Int', Int) \in \mathscr{L}$ then
10:          $c \leftarrow$ NEW-CONCEPT($Extent(\bar{c}) \cup \{o\}, Int$)   {($\bar{c}$) is a genitor}
11:          UPDATE-ORDER($c, \bar{c}$) ; ADD($\mathscr{L}, c$)
```

As an illustration, assume that the object 3 is to be inserted into the lattice induced by $O = \{12456789\}$ (in Fig. 1, on the right). Following Algorithm 1, the relevant concept sets are $\mathbf{U}(o) = \{c_{\#7}, c_{\#9}\}$, $\mathbf{M}(o) = \{c_{\#1}, c_{\#2}, c_{\#4}\}$, and $\mathbf{G}(o) = \{c_{\#3}, c_{\#5}, c_{\#6}, c_{\#8}, c_{\#10}, c_{\#11}, c_{\#12}\}$. The new concepts correspond to the intents $c, f, cd, cf, fh, fgh,$ and $cdfgh$. Fig. 2 depicts the updated lattice.

In the remainder, we focus on an adaptation of the above technique to the incremental generation of *CI* families which is rooted in a formalization of the manipulated lattice substructures.

4. Lattice-based framework for incremental itemset mining

The aforementioned concept sets are formally defined and translated into subsets of *CIs* whose role in the mining process is investigated.

4.1. Definitions

We need to formally define concept categories in $\mathscr{L}^+$ as Godin et al. only considered their homologous elements in $\mathscr{L}$. To avoid confusion, operators $'$ hereafter will be denoted so as to reflect the underlying context, i.e., $\_^I$ for $\mathscr{K}$ and $\_^J$ for $\mathscr{K}^+$ (we assume $J = I \cup \{o\} \times o'$). First, two maps are defined between contexts linking concepts that share one dimension or both.

Definition 4. Let $\sigma : \mathscr{C} \to \mathscr{C}^+$; $\sigma(X, Y) = (Y^J, Y)$ and $\gamma : \mathscr{C}^+ \to \mathscr{C}$; $\gamma(X, Y) = (X_1, X_1^I)$, where $X_1 = X - \{o\}$.

In other terms, σ preserves the concept intent, whereas γ preserves extents up to a difference of $\{o\}$. Now new concepts are exactly those whose extents comprise o but whenever o is removed, the result is a valid extent, whereas for a modified extent the removal does not yield an extent.

Definition 5. The set of new concepts in $\mathscr{L}^+$ is

$$\mathbf{N}^+(o) = \{(X, Y)|(X, Y) \in \mathscr{C}^+; o \in X; (X - \{o\})^{JJ} = X - \{o\}\}.$$

$\mathbf{M}(o)$ and $\mathbf{M}^+(o)$ are characterized by intent preserving upon swaps of o in the extent.

Definition 6. The sets of modified concepts in $\mathscr{L}^+$ and in $\mathscr{L}$ are, respectively:

- $\mathbf{M}^+(o) = \{(X, Y)|(X, Y) \in \mathscr{C}^+; o \in X; (X - \{o\})^J = Y\}$,
- $\mathbf{M}(o) = \{(X, Y)|(X, Y) \in \mathscr{C}; (X \cup \{o\})^J = Y\}$.

Genitor definition is the reverse of the new concept one: The genitor of a concept (X, Y) in $\mathbf{N}^+(o)$ has an extent $X - \{o\}$ in both $\mathscr{K}$ and $\mathscr{K}^+$.

Definition 7. The sets of genitor concepts in $\mathscr{L}^+$ and in $\mathscr{L}$ are:

- $\mathbf{G}^+(o) = \{(X, Y)|o \notin X; (X \cup \{o\})^{JJ} = X \cup \{o\}\}$;
- $\mathbf{G}(o) = \{(X, Y)|(X \cup \{o\})^{JJ} = X \cup \{o\}; Y \neq (X \cup \{o\})^J; \}$.

A definition closer to the one used by Godin et al. states that genitor intents are the *closures* of the underlying intersections with $\{o\}^J$ while themselves not included in $\{o\}^J$:

Proposition 8. *The set of genitors in* $\mathscr{L}$ *is* $\mathbf{G}(o) = \{(X, Y)|Y \not\subseteq \{o\}^J; Y = (Y \cap \{o\}^J)^{II}\}$.

To sum up, both genitors and modified intents in $\mathscr{L}$ represent the *closures* of their own intersections with o^J, whereby the intersection is an intent in $\mathscr{K}$ for a modified but not for a genitor.

4.2. Bridging the gap between concepts and CIs

We start with some notations to support further discussion. Recall that the family of *CIs* of a TDB $\mathscr{D}$ roughly corresponds the set of intents in the equivalent context representation $\mathscr{K}_{\mathscr{D}}$. In fact, the only notorious difference is that under no circumstances will the target structure for mining comprise the empty set (should it be closed) as it carries no useful information. The following table presents a summary of the mathematical notations used in the remainder of the text. These slightly diverge from the standard notations in data mining literature.

Symbol	Stands for
T_n	The new transaction
$\mathscr{D}$	The current database
$\mathscr{C}^a_{\mathscr{D}}$	The set of *CIs* in $\mathscr{D}$
$\delta\mathscr{C}^a$	Set difference $\mathscr{C}^a_{\mathscr{D}^+} - \mathscr{C}^a_{\mathscr{D}}$
I_n	The itemset of T_n
$\mathscr{D}^+$	$\mathscr{D}$ augmented with T_n
$\mathscr{C}^a_{\mathscr{D}^+}$	The set of *CIs* in $\mathscr{D}^+$
$\mu\mathscr{C}^a$	*CIs* from $\mathscr{D}$ included in I_n

The following storage structures for *CIs* appear in algorithmic code:

- *FamilyCI*: the data structure for $\mathscr{C}^a_{\mathscr{D}}$,
- *NewCI*: the data structure for $\delta\mathscr{C}^a$.

Moreover, the nodes *e* of *FamilyCI* will have fields `itemset`, `support`.

Further to Section 3.3, the updating of $\mathscr{C}^a_{\mathscr{D}}$ upon the arrival of a new transaction T_n amounts to computing all intersections of existing *CI* with its itemset I_n which further fall into two cases: itemsets already in $\mathscr{C}^a_{\mathscr{D}}$ (hence closed and in $\mu\mathscr{C}^a$) and *new CIs*, i.e., closed only in $\mathscr{C}^a_{\mathscr{D}^+}$ (hence in $\delta\mathscr{C}^a$). A straightforward method could generate all intersections and add the new ones to $\mathscr{C}^a_{\mathscr{D}}$.

Step two is the calculation of the support for the *CIs* in $\mathscr{C}^a_{\mathscr{D}^+}$. Intuitively, for any intersection Y_T in $\delta\mathscr{C}^a \cup \mu\mathscr{C}^a$, its absolute support in $\mathscr{D}^+$ is exactly the support of Y_T in $\mathscr{D}$ plus one. However, the latter is not directly available as Y_T may be obtained by more than one intersection, e.g., in Fig. 1, *c* is the result of intersecting *cdfgh* with *bc* or *abc*. Here we recall a key property from Section 2: Y_T shares the support value with its closure Y_T^{II} (since a transaction comprising Y_T also comprises Y_T^{II}). This suggests the following procedure for support computation: (i) find the closure Y_T^{II}, (ii) extract its support in $\mathscr{D}$, and (iii) add one (two more operations necessary if support is relative). The procedure stresses the importance within the *CI* framework of the notions of genitor and modified intent which, as indicated in Section 4.1, represent the respective closures for intersections in $\delta\mathscr{C}^a \cup \mu\mathscr{C}^a$. The next concern is the efficient detection of the closure of Y_T with its support. It is addressed below through an alternative definition of closures that is readily embodied into an algorithm.

4.3. Extremal status of closures

Back to concept analysis we focus on the structure that a new transaction induces on the current *CI* family. It is rooted in a basic property of closures stating that, set in our mining terminology, given an intersection Y_T, its closure in $\mathscr{D}$ is the smallest *CI* comprising Y_T. In the case of genitor and modified *CIs*, the property can be strenghtened using further constructs, a set-valued function and the equivalence it induces on $\mathscr{C}^a_{\mathscr{D}}$. Both are part of a homogeneous characterization for genitor and modified concepts that generalizes the initial results of Godin et al. (see [37]). First, the function $\mathscr{Q}$ maps $\mathscr{C}^a_{\mathscr{D}}$ into $\wp(\mathscr{I})$ by computing the respective intersections with I_n.

Definition 9. The function $\mathscr{Q} : \mathscr{C}^a_{\mathscr{D}} \to \wp(\mathscr{I})$ computes: $\mathscr{Q}(e) = e.itemset \cap I_n$.

$\mathscr{Q}$ induces an equivalence relation on $\mathscr{C}^a_{\mathscr{D}}$ whereby each class $[]_{\mathscr{Q}}$ has a unique minimal element for set inclusion, which is exactly the closure of the respective $\mathscr{Q}$ value.

Proposition 10. $\forall e \in \mathscr{C}^a_{\mathscr{D}}, \exists\, \bar{e} = \min([e]_{\mathscr{Q}})$, *where by* $\bar{e}.itemset = (e.itemset \cap I_n)^{II}$.

Let $\mathbf{E}(T_n)$ be the set of class minima. From Proposition 8 and the trivial fact $\mathbf{M}(T_n) \subseteq \mathbf{E}(T_n)$, we conclude that class minima are exactly the genitor and modified *CIs*.

Proposition 11. *The set of all class minima in* $\mathscr{C}^a_{\mathscr{D}}$ *is* $\mathbf{E}(T_n) = \mathbf{G}(T_n) \cup \mathbf{M}(T_n)$.

To sum up, the target *CIs* lay at the bottom of their respective classes. A key observation here is that the overall bottom of the closures is also the minimum of the class induced by the smallest Y_T. If $Y_T = \emptyset$, i.e., practically in every realistic case, the class will not require any processing, albeit covering a large proportion of the current *CIs*, since the empty itemset is of no value.

5. Incremental generation of FCIs with Galicia

The Galicia (for GAlois Lattice-based Incremental Closed Itemset Approach) mining approach is presented below together with two high-level methods (low-level design is discussed in Section 6): a first one mirroring the method in [21], used for comparison purposes here (see [39] for details), and a second method using a novel search strategy to spot relevant *CIs*.

5.1. Straightforward incremental method

Algorithm 1 may be turned into a *CI* miner by limiting processing to relevant concept elements.

5.1.1. Principles of the approach

Our aim is, given a TDB $\mathscr{D}$ and its *CI* family $\mathscr{C}^a_{\mathscr{D}}$, to construct $\mathscr{C}^a_{\mathscr{D}^+}$ reflecting $\mathscr{D}^+ = \mathscr{D} \cup \{T_n\}$ only by looking at T_n and $\mathscr{C}^a_{\mathscr{D}}$. A straightforward strategy consists in examining every *CI* in $\mathscr{C}^a_{\mathscr{D}}$ and computing its intersection with I_n. Instead of keeping an intersection at each *CI*, every known intersection Y_T could keep track of the (so far) smallest *CI* that produced it. This requires no direct comparison of *CIs* since supports are just as good: the minimum of a class has also the highest support. Thus, the traversal of $\mathscr{C}^a_{\mathscr{D}}$ can also yield the closure of each Y_T.

A less exhaustive approach could process differently modified and new *CIs*. Thus, while new intersections require explicit storage and support computation, the existing ones are almost done once detected: The *CI* that such an intersection represents is in $\mathscr{C}^a_{\mathscr{D}}$ and it only needs a support increase by one (once during the entire traversal). This implies a more complex control structure with lookups within $\mathscr{C}^a_{\mathscr{D}}$ (to check whether Y_T is inside) and in $\delta\mathscr{C}^a$ (actually, the known part thereof). Section 5.1.2 presents a method which applies this strategy.

Finally, let us observe that incremental methods are *forced to work with the entire set* of *CIs*, including infrequent ones. First, the frequent status depends on the portion of the TDB already processed: some of the *FCIs* at a given time point may become infrequent after some further insertions, and vice versa. Then, discarding infrequent genitors will prevent the discovery of the respective new *CIs* which may well be frequent. For instance, assume transactions (10, *abcd*) and (11, *abcde*) are added to $\mathscr{D}$ (see Table 1). *FCIs cf*, *efh*, *fgh* become now infrequent *CIs* (27%) while *abcd* is frequent (36%). If *abcef* was initially discarded as infrequent *CI* (22%), then the new *CI abce* would have been missed.

5.1.2. Algorithmic design

Algorithm 2 hereafter preserves the main control structure of its lattice counterpart: each *CI* of the current collection (*FamilyCI*) is examined to establish its specific category (*modified*, *old* or *genitor* of a new *CI*). Modified *CIs* simply get their support increased (line 9) and old ones remain unchanged (line 11). Processing genitors diverges from the lattice version since no particular order is assumed on *FamilyCI*. Indeed, as one cannot rely on a specific ordering within that collection, a class minimum can be established only after the traversal of the entire class, which, in turn, cannot be reliably assumed as fully accomplished before every member of *FamilyCI* has been examined. To that end, it is enough to keep track of the minimal generating *CI* for every new intersection along the global traversal. In doing that, set inclusion tests can be advantageously replaced by support comparison. Actually, each new *CI* is stored together with the maximal support already reached for that *CI*. Thus, each time the *CI* is generated (lines 13–17), the support is tentatively updated. Furthermore, the storage of new *CIs* is organized separately (collection *NewCI*) so that unnecessary tests can be avoided.

Algorithm 2. Update of the *CI* family upon a new transaction arrival.

```
1: procedure UPDATE-CLOSED(In: $T_n$ a transaction, FamilyCI a collection of itemsets)
2:
3: Local : NewCI a collection of itemsets
4:
5: NewCI ← ∅ ; $I_n \leftarrow T_n.itemset$
6: for all $e$ in FamilyCI do
7:   $I_e \leftarrow e.itemset$
8:   if $I_e \subseteq I_n$ then
9:     $e.support{+}{+}$    {$e$ is modified, update its support}
10:  else
11:    $Y \leftarrow I_e \cap I_n$ ; $e_Y \leftarrow$ lookup(FamilyCI, $Y$)    {$e$ may be old or genitor}
12:    if $e_Y$ = NULL then
13:      $e_Y \leftarrow$ lookup(NewCI, $Y$)    {$e$ is a potential genitor}
14:      if $e_Y$ = NULL then
15:        $node \leftarrow$ create-CI($Y$, $e.support + 1$) ; NewCI ← NewCI ∪ {$node$}
16:      else
17:        $e_Y.support \leftarrow \max(e.support + 1, e_Y.support)$
18: FamilyCI ← FamilyCI ∪ NewCI
```

The above computation yields the correct supports at the end of the global traversal. This fact is strongly reinforced by an implementation proposal which utilizes trie structures in order to reduce redundancy in both storage and update of the *CI* family.

It is noteworthy that the gains due to the use of a linear extension of the lattice order in the immediate detection of genitors (as in [21]) are offset by the overhead of the order maintenance.

5.1.3. *Limitations of the exhaustive traversal*

Although the utilization of advanced data structures may lead to some gains both in memory consumption and efficiency, the complete exploration of the *CI* family upon each insertion may still prove too expensive for large databases (see [39] for details). This observation emerged from our preliminary experimental studies. In fact, in large and sparse databases, the insertion of a new transaction requires the processing of only a limited set of existing *CIs* (modified and genitors). The size of this set is usually far smaller than the size of the entire *CI* family $\mathscr{C}^a_{\mathscr{K}}$ (down to 0.1%). Thus, the overwhelming number of computations done by the exhaustive algorithm will not trigger any modification of the *CI* family.

This fact motivates the design of improved search strategies, e.g., one that only processes potential genitors and modified while skipping a large portion of the old *CIs* as discussed below.

5.2. *Narrowing the set of examined CIs*

In the ideal case, the incremental algorithm should be able to pinpoint members of $\mathbf{E}(T_n)$ with a minimal search outside that set. For instance, the method in [41] relies on lattice order to move quickly within an equivalence class toward its maximal concept, i.e., minimal intent, while skipping many of its non-maximal members. However, as order has been deliberately excluded here, a different criterion for eliminating old *CIs* is necessary. Let us now observe that all non-empty genitor and modified *CIs* share at least one item with T_n. Thus, a possible superset of $\mathbf{E}(T_n)$ to target with an algorithm is the set of existing *CIs* having a non-empty intersection with I_n, denoted $\mathbf{S}(T_n)$:

$$\mathbf{S}(T_n) = \{e \in \mathscr{C}^a_{\mathscr{K}} | e.itemset \cap I_n \neq \emptyset\}.$$

In fact, unless the data set is very dense, $\mathbf{S}(T_n)$ will be orders of magnitude smaller than $\mathscr{C}^a_{\mathscr{K}}$.

In the light of the above arguments, the task of a parsimonious update of $\mathscr{C}^a_{\mathscr{K}}$ could be split into subtasks as follows: (i) detecting all the elements of $\mathbf{S}(T_n)$; (ii) computing the value of $\mathscr{Q}$ for each c in $\mathbf{S}(T_n)$; (iii) partitioning of $\mathbf{S}(T_n)$ into classes with respect to $\mathscr{Q}$; (iv) detecting the minimal element in each class, i.e., $\mathbf{E}(T_n)$; (v) determining the category of each minimum; (vi) performing the updates in the *CI* family.

Task (i) requires an efficient means for the detection of $\mathbf{S}(T_n)$, e.g., an indexing structure associating to each item i the set of *CIs* that share i. Thus, the exploration of $\mathscr{C}^a_{\mathscr{K}}$ can be limited to *CIs* that belong to at least one list of an item from T_n. Moreover, this enables gradual computing of intersections, i.e., by adding the current item to partial intersections (see below). The identification of distinguished *CIs* involves support comparisons within classes hence $\mathbf{S}(T_n)$ must be split to form these classes.

5.3. *Parsimonious CI mining*

The overall control structure of a method focusing exclusively on $\mathbf{S}(T_n)$ splits into three steps: (i) traversal of $\mathbf{S}(T_n)$ with simultaneous intersection calculation, (ii) partitioning of $\mathbf{S}(T_n)$ into classes, and (iii) detection of class minima with the subsequent updates. The main improvement here is the removal of the entire class generated by $\mathscr{Q} = \emptyset$ from the search space. The efficiency gain thereof, as compared to the one presented in [39], is particularly high with sparse transaction sets where each item is shared by a small proportion of all transactions while many *CIs* generate empty intersections with I_n.

The new method uses *CI* storage similar to the one in Section 5.1. In addition, it stores with a *CI* e the (partial) intersection of its itemset with I_n. Its computation relies on *ItemIndex*, an index of *CIs* by their member items (see example below).

Algorithm 3 starts with a traversal of all *CI* lists corresponding to the items from I_n (lines 6–8). At each *CI* e from a list i, the item i is added to the current value of the intersection. Next, examined *CIs* are split into classes following the intersections (line 9). Then, class minima are computed and their status is determined followed by the respective updates (support increase, new *CI* creation, etc.). At any creation of a new *CI* $\hat{e}$ it is added to each of the lists corresponding to an item from $\hat{e}.itemset$ (lines 15–16). It is noteworthy that as intersections are stored at each *CI*, there is no need for a global structure to host them.

Algorithm 3. Parsimonious update of the *CI* family of a transaction database.

```
1:  procedure UPDATE-CLOSED-BIS (In: $T_n$ a transaction, In/Out: FamilyCI a collection of
    CIs, ItemIndex an indexed set of CI lists)
2:
3:  Local : Qclasses a trie indexing sets of CIs
4:
5:  $I_n \leftarrow T_n.itemset$
6:  for all $i \in I_n$ do
7:     for all $e \in ItemIndex(i)$ do
8:        add($e.intersection, i$)   {gradually construct all non-trivial intersections}
9:  Qclasses ← SEPARATECLASSES(FamilyCI)   {separate the classes in $\mathscr{C}^a$}
10: for each class $\Theta \in$ Qclasses do
11:    $e \leftarrow$ MIN($\Theta$)   {extracts the node of highest support in $\Theta$}
12:    if $e.itemset \subseteq I_n$ then
13 :      $e.support++$
14 :   else
15:       $\hat{e} \leftarrow$ create-CI($e.itemset \cap I_n, e.support + 1$)
16:       add(FamilyCI,$\hat{e}$)
17:       for all $i \in \hat{e}.itemset$ do
18:          add(ItemIndex($i$),$\hat{e}$)
```

To illustrate the algorithm, assume the TDB $\mathscr{D} = \{1, 2, 3, \dots, 9\}$ from Section 3.3. The *CIs* of $\mathscr{D}$ are provided below together with their respective IDs and supports:

CI id	i-set	Supp.
#1	*abcdefgh*	1
#2	*abcd*	2
#3	*cdfgh*	2
#4	*abcef*	2
#5	*efgh*	2
#6	*bcd*	3
#7	*efh*	3
#8	*cd*	4
#9	*abc*	3
#10	*cf*	3
#11	*fgh*	3
#12	*ef*	4
#13	*bc*	4
#14	*fh*	4
#15	*d*	5
#16	*c*	5
#17	*g*	4
#18	*f*	5

We consider the insertion of a transaction $(10, bcgh)$ into $\mathscr{D}$. The following table illustrates the *CI* lists associated to items in I_n within *ItemIndex*.

Item	*CI* id lists
b	#1, #2, #4, #6, #9, #13
c	#1, #2, #3, #4, #6, #8, #9, #10, #13, #16
g	#1, #3, #5, #11, #17
h	#1, #3, #5, #7, #11, #14

The content of *Qclasses* after the sorting step (line 10) is presented in the table below, together with the indication of the class minimum and its respective status (**gen**itor or **mod**ified).

$\mathscr{Q}(c)$	$[]_{\mathscr{Q}}$	min.	Status
c	#8, #10, #16	#16	**mod**
g	#17	#17	**mod**
h	#7, #14	#14	**gen**
bc	#2, #4, #6, #9, #13	#13	**mod**
gh	#5, #11	#11	**gen**
cgh	#3	#3	**gen**
bcgh	#1	#1	**gen**

As a result, at the end of the traversal of *Qclasses* (lines 10–18) the new *CIs* are created, whereas modified *CIs* have their support increased. These are provided by the following two tables:

New *CIs* (δC^a)

CI id	#19	#20	#21	#22
Itemset	*bcgh*	*cgh*	*gh*	*h*
Genitor	#1	#3	#11	#14
Support	2	3	4	5

Modified *CIs* (μC^a)

CI id	#13	#16	#17
Support	5	6	5

Finally, the new state of the relevant lists in the index structure is as follows:

Item	*CI* id lists
b	#1, #2, #4, #6, #9, #13, #19
c	#1, #2, #3, #4, #6, #8, #9, #10, #13, #16, #19, #20
g	#1, #3, #5, #11, #17, #19, #20, #21
h	#1, #3, #5, #7, #11, #14, #19, #20, #21, #22

6. Implementing the parsimonious incremental method

Algorithm 3 already uses an indexing structure on all *CIs* to carry out steps (i) and (ii) simultaneously. We propose to extend the simultaneous processing to cover steps (iii) and (iv) as well, which amounts to yielding $\mathbf{E}(T_n)$ at the end of the initial traversal of $\mathbf{S}(T_n)$.

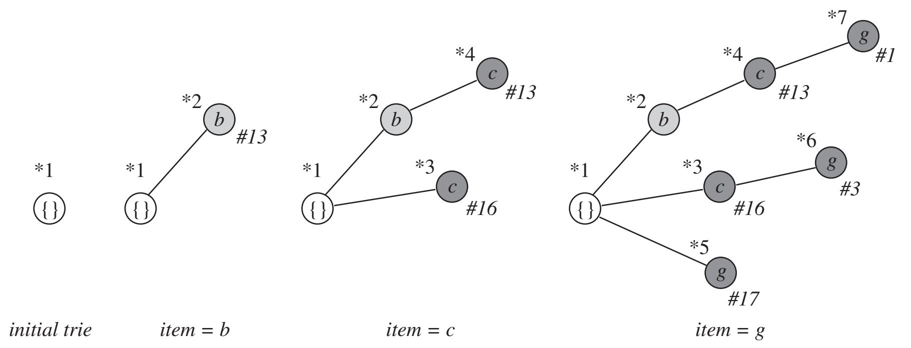


Fig. 3. The evolution of *Intersections*: state of the trie after each of the first three steps. Terminal nodes are drawn in dark gray.

6.1. Principles

To enable an even more limited search, we use a compact storage of the produced intersections: Instead of keeping a copy of its intersection locally, each *CI c* stores only a reference to a global trie structure, *Intersections*, that represents intersections and their shared prefixes only once.

Tries [23] provide a good trade-off between storage requirements and manipulation cost, hence they are frequently used to store large data sets, e.g., collections of words over an alphabet. In its basic form, a trie is a tree with letters assigned to vertices (or to edges), so that each word corresponds to a unique path (see Fig. 3). Nodes corresponding to the end of a word, called *terminal* nodes, are distinguished from the rest.

Tries compact the information since all prefixes common to two or more words are stored only once in the trie. Such factorization not only reduces the storage space, but also provides for more efficient operations, e.g., search or insertion of a word into the trie. Tries where words represent sets—as in our case—provide very efficient operations which can be carried out in a time linear in the size of the alphabet, regardless of the size of the trie.

In *Intersections*, a node is a record with fields `item`, `successors`, `current-min` and `nb-refs`. The `successors` field is a sorted, indexed and extendible collection. The third field is a pointer to the current element of maximal support, while the fourth one reflects the number of *CIs* pointing at the node via their `last-item` fields. A *CI* record has fields `id`, `support`, and `last-item`. The last one points at the node *n* from *Intersections* such that `path`(*n*) (see below) corresponds to the already processed part of the intersection.

Technically speaking, a *CI c* stores a pointer to the terminal node in the trie which corresponds to a path labeled by the intersection $\mathcal{Q}(c)$. As the intersections are computed gradually during the traversal of the item index, the trie also grows with every item from I_n. At any moment, the pointer indicates a node labeled by the last item in $\mathcal{Q}(c)$ that has been examined so far. In this way, at the end of the traversal, the pointer of a *CI c* is directed at the last node of the path representing the $\mathcal{Q}(c)$. Consequently, the class $[c]_{\mathcal{Q}}$ is implicitly represented as the set of all *CIs* pointing at the trie node with the last item of $\mathcal{Q}(c)$ (see Section 6.4). This reduced view on classes nevertheless yields both class minima and the associated $\mathcal{Q}(c)$ value.

A class minimum emerges by keeping track of all *CIs* that point, or have pointed, to a node, and then selecting the most frequent one. For simplicity, it will be referred to by a reverse pointer stored at a trie node and updated during the traversal. $\mathcal{Q}(c)$ is represented by the path from the trie root up to the terminal node, denoted *path*(). A counting mechanism helps spot the terminal trie nodes which are otherwise hard to detect as the last item of a *CI* is not available through the index structure. Thus, nodes with a strictly positive number of references from *CIs* are terminal, whereas the others are not.

To separate genitors from modified (subtask (v) in Section 5.2), the *CI c* indicated by the reverse pointer at a node *n* is retrieved and its size is compared to the size of $\mathcal{Q}(c)$ which is the length of *path*(*n*). Equality of sizes witnesses modified *CIs* and inequality genitors. In summary, after the first stage, class minima are pointed at by trie nodes marking the end of the intersection path. Stage two recognizes categories and carries out updates in a way similar to Algorithm 3. A noteworthy difference, trie paths are traversed to form the new *CIs*.

6.2. Algorithmic design

At its first stage the method (see Algorithm 4) loops over the items in I_n (outer loop, line 6–11), then over the list of *CIs* associated to an item i (inner loop). Within the inner loop, the intersection of the current *CIs* with I_n is updated by adding i to it. This may or may not involve a creation of a node labeled i in *Intersections*, depending on whether the underlying path exists in the trie or not. Moreover, for already encountered *CIs* (line 9), the lookup for i starts from the last trie node in the current intersection path which is actually referred to by the `last-item` field. If i is the first item of the intersection, i.e., if `last-item` is NULL, then the search starts at the root of the trie (line 11).

Algorithm 4. Update of the *CI* family of a transaction database.

```
1:  procedure UPDATE-FAMILYCI(In: $T_n$ a transaction, In/Out: $FamilyCI$ a collection of
    $CIs$, $ItemIndex$ an indexed set of $CI$ lists)
2:
3:  Local : $Intersections$ a trie of itemsets indexing $CIs$
4:
5:  $I_n \leftarrow T_n.itemset$
6:  for all $i \in I_n$ do
7:    for all $e \in ItemIndex(i)$ do
8:      if $e.last\text{-}item = \text{NULL}$ then
9:        UPDATE-INTERSECTION($e$, $i$, root($Intersections$))
10:     else
11:       UPDATE-INTERSECTION($e$, $i$, $e.last\text{-}item$)
12: for each terminal node $n \in Intersections$ do
13:   $e \leftarrow n.\ current\text{-}min$ {returns the minimum the class corresponding to path($n$)}
14:   if $e.size = n.depth$ then
15:     $e.support{+}{+}$   {modified $CI$}
16:   else
17:     $\hat{e} \leftarrow$ create-CI(path($n$), $e.support + 1$)   {creates a new $CI$}
18:     add($FamilyCI$, $\hat{e}$)
19:     for all $i \in$ path($n$) do
20:       add($ItemIndex(i)$, $\hat{e}$)
```

The lookup/insertion (primitive UPDATE-INTERSECTION, Algorithm 5) first checks the existence of a node i among the successors of the current trie node (line 3). A negative outcome triggers the creation of such a node (lines 5–6) with appropriate field values. For existing nodes, the current minimal *CI* is tested for possible update (lines 8–9). The number of references in both the previous and the current last items for the *CI* is properly modified (line 10).

Algorithm 5. Update of the intersection trie nodes.

```
1:  procedure UPDATE-INTERSECTION(In: $e$ a $CI$, $i$ an item, $n$ a trie node)
2:
3:  $\hat{n} \leftarrow$ get-successor($n$, $i$)
4:  if $\hat{n} = \text{NULL}$ then
5:    $\hat{n} \leftarrow$ create-node($i$, $\emptyset$, $e$, 0)   {initializes $item$, $successors$, $current\text{-}min$, $nb\text{-}refs$}
6:    add-successor($n$,$\hat{n}$)
7:  else
8:    if $\hat{n}.current\text{-}min.support\text{-}max < e.support$ then
9:      $\hat{n}.current\text{-}min \leftarrow e$
10: $\hat{n}.nb\text{-}ref{+}{+}$; $n.nb\text{-}ref{-}{-}$
11: $e.last\text{-}item \leftarrow \hat{n}$;
```

The next stage of Algorithm 4 is the partition of minima into modified and genitors and the modification of the data structures. Minimal *CIs* are retrieved from the terminal nodes of *Intersections* (line 12). Depth of a node, i.e., the length of the path from the root, is used to recognize modified *CIs*. Moreover, new *CI* are recovered by effectively traversing the path from the terminal node down to the trie root (line 17). Finally, the index is updated with new *CIs* which are added to the appropriate lists (lines 19–20).

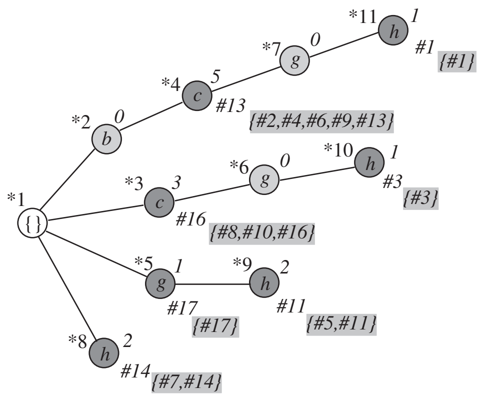


Fig. 4. Final state of the trie induced by the insertion of *bcgh*. The value of the `nb-ref` counter is visualized above each node. Terminal nodes are given with the set of all *CIs* that point to them.

6.3. Example

As an illustration, assume again that (10, *bcgh*) must be inserted into $\mathscr{D} = \{1, 2, 3, \ldots, 9\}$. Since the content of *ItemIndex* does not differ from what was given in Section 5.3, we only show the evolution of *Intersections* and the related `last-item` field in the extended *CI* structure.

The execution trace below follows the steps of the external **for** loop: It provides the `last-item` values and the content of *Intersections* at the end of each step. Pointers to trie nodes are given in tabular form, while the subsequent modifications of *Intersections* are visualized in Figs. 3 and 4. To ease distinction between *CIs* and trie nodes while keeping the notations succinct, the latter nodes are denoted by a $*$ sign preceding a numeric identifier, i.e., from $*1$ up.

As an initialization, the fields `last-item` in *FamilyCI* nodes are set to NULL ($\times$ below), whereas *Intersections* is set to a root node whose item field is void ({}) as depicted in Fig. 3. The processing of *b* yields a new successor of the root. The exploration of *CIs* associated with *b*—#1, #2, #4, #6, #9, #13—assigns the *id* of the current node, $*2$, to the respective `last-item` fields. The *id* of the current minimal *CI* and the number of *CIs* pointing to the node $*2$ are stored as well. At its end, step *b* of the loop yields the results in the next table (see also Fig. 3).

id	`last`
#1	$*2$
#2	$*2$
#3	$\times$
#4	$*2$
#5	$\times$
#6	$*2$
#7	$\times$
#8	$\times$
#9	$*2$
#10	$\times$
#11	$\times$
#12	$\times$
#13	$*2$
#14	$\times$
#15	$\times$
#16	$\times$
#17	$\times$
#18	$\times$

The step focusing on c examines *CIs* #1, #2, #3, #4, #6, #8, #9, #10, #13, #16 and extends both trie paths, i.e., {} and b, with a node c each. In addition, all the above *CIs* point now to nodes c. In particular, the number of references to the node b is decreased to 0, since all *CIs* that previously pointed to $*2$ have been reset to $*4$. This reflects the fact that bc is the closure of b. This step of the loop leaves the trie in a state given in Fig. 3, whereas the relevant fields in the *CIs* are given by the next table.

id	`last`
#1	$*4$
#2	$*4$
#3	$*3$
#4	$*4$
#5	$\times$
#6	$*4$
#7	$\times$
#8	$*3$
#9	$*4$
#10	$*3$
#11	$\times$
#12	$\times$
#13	$*4$
#14	$\times$
#15	$\times$
#16	$*3$
#17	$\times$
#18	$\times$

The links from the *CIs* to the trie nodes after the processing of g are depicted in the next table. Fig. 3, again, provides the content of the trie itself.

id	`last`
#1	$*7$
#2	$*4$
#3	$*6$
#4	$*4$
#5	$*5$
#6	$*4$
#7	$*8$
#8	$*3$
#9	$*4$
#10	$*3$
#11	$*9$
#12	$\times$
#13	$*4$
#14	$*8$
#15	$\times$
#16	$*3$
#17	$*5$
#18	$\times$

The values of `last-item` after the processing of h are given in the table below.

id	`last`
#1	$*11$
#2	$*4$
#3	$*10$
#4	$*4$
#5	$*9$
#6	$*4$
#7	$*8$
#8	$*3$
#9	$*4$
#10	$*3$
#11	$*9$
#12	$\times$
#13	$*4$
#14	$*8$
#15	$\times$
#16	$*3$
#17	$*5$
#18	$\times$

Fig. 4 illustrates the final state of *Intersections* which is to be explored by the second phase of the method. For each terminal node n, the figure shows the *CIs* from the class induced by $\mathcal{Q}$.

6.4. Soundness results

The correctness of the minima computation, i.e., the recognition of their status and category, is rooted in the following two properties. First, we prove that at the end of stage one, for any terminal node n, the *CI* pointed by `current-min` is exactly the minimal element of the class generated by the itemset corresponding to $path(n)$.

Proposition 12. *Given a trie node n, and let n be terminal. If c is the CI that n.* `current-min` *points at, then $e = \min([e]_{\mathcal{Q}})$ and $path(n) = \mathcal{Q}(e)$.*

Proof (*Sketch*). The key idea is to show that given a *CI* e with its class $[e]_{\mathcal{Q}}$ and the corresponding node n, such that $path(n) = \mathcal{Q}(e)$, the minimal element in $[e]_{\mathcal{Q}}$, say $\hat{e} = \min([e]_{\mathcal{Q}})$ is more frequent than any other *CI* $\bar{e}$ such that $\mathcal{Q}(e)$ is a prefix of $\mathcal{Q}(\bar{e})$. □

To complete the proof of Proposition 12 one must show that that terminal nodes can be correctly recognized, i.e., that at the end of stage one (lines 6–11 of Algorithm 4), only trie nodes n such that $path(n) = \mathcal{Q}(e)$ for some e have a non-zero value for `nb-refs`. In fact, the property below reflects the way the trie is constructed, hence it holds.

Proposition 13. *Given a node n in the trie and a CI e, at the end of the computation, e.*`last-item` *$= n$ if and only if $path(n) = \mathcal{Q}(e)$.*

7. Related work

Frequent pattern mining is a key step in many data mining tasks such as the discovery of association rules, sequential patterns, and episodes. In the following, we report in a non-exhaustive way related work on incremental *FI* mining or extraction of *CIs* as well as on connected aspects of FCA algorithmic practice, especially on concept computation and lattice construction.

7.1. Concept computation

Early algorithms calculating the set of concepts (under different names) may be found in [12,25], but the first one dedicated to the task is NEXTCLOSURE [18]. It uses a classical listing technique for combinatorial objects based on an order, called *lectic*, on attribute sets and can actually compute any closure family provided with an explicit closure operator. NEXTCLOSURE explores $\wp(A)$ with closure tests on candidates while listing closures in the lectic order based on a canonical representation thereof, i.e., a lectically minimal generating prefix. The bases of the incremental concept formation approach were laid in the work of Godin et al. [21]. GALOIS [11] is another pioneering incremental method. Recently, deeper insights into the incremental update mechanisms founded the design of smarter methods, e.g., an off-spring of a data partitioning framework [38] or a bottom-up traversal strategy for the lattice graph [41]. Finally, [37] proposed a generic algorithmic scheme for the incremental lattice computation.

7.2. FI mining

Historically, the reference *FI* mining algorithm is APRIORI [2]. It performs a level-wise generation of *FIs* within the powerset lattice $2^{\mathscr{I}}$, starting with singleton sets and moving upwards and level-wise in $2^{\mathscr{I}}$. At level $i+1$, the candidates are generated by joining *FIs* from level i that differ by a single element. Candidates having at least one infrequent subset are pruned *a priori*, i.e., without looking at the database to calculate frequencies. APRIORI was followed by a variety of competing mining approaches. A large part of them aim at improving the efficiency of the basic method [22], whereby the key difficulty is the potentially huge number of *FIs*.

As a remedy, characteristic subsets of the *FI* family were brought to light, most prominently, the closed *FIs* [47,27,5,29] and the maximal *FIs* [7,10], i.e., *FIs* having only infrequent supersets (*MFIs*). It is noteworthy that while there can be exponentially more *CFIs* than *MFIs*, *CFI* encode the family of *FIs* faithfully, whereas *MFIs* only retrieve *FIs* without supports. On the complexity axis, it was shown in [9] that listing all *MFIs* is an NP-hard problem while listing *CFIs* can be done in incremental polynomial time (see [16] for a description of complexity classes).

7.3. Batch FCI mining

ACLOSE and CLOSE [28] are among the first *FCI* miners. Like APRIORI, ACLOSE performs level-wise traversal of $2^{\mathscr{I}}$, but exploits the generators, i.e., the minimal itemsets producing a *CI* by means of $''$. Generators replace candidates in the APRIORI framework; they guide the *FCI* lookup in the database. TITANIC [32] improves ACLOSE, in that it relies on further properties of generators, e.g., easy computing of closures through supersets, to avoid redundant computation.

CHARM [47] is another closed pattern miner which generates *FCIs* in a tree organized by inclusion. Closure and support computations rely on storage and intersection of *TID-sets*. To speed-up closure computation, CHARM uses *diffsets*, the set difference on the *TID-sets* of a given node and of its unique parent node in the tree.

CLOSET and its recent improvement CLOSET+ [43] both generate FCIs as maximal branches of a *FP-tree*, a structure that is basically a prefix tree (or *trie*) augmented with transversal lists of pointers. The global FP-tree of a database is projected into a set of conditional FP-trees that organize patterns sharing the same suffix. Support values are compared in order to compute the closure of a given branch in the FP-tree. BAMBOO [44] is an improved version of CLOSET+ producing a reduced result set and with better performance and scalability features than the latter. BAMBOO exploits various pruning and optimization techniques to accelerate the mining process. For instance, the length-decreasing support constraint defines the minimal support as a non-increasing function of itemset length based on the observation that short itemsets may be interesting if they have a high support, while long itemsets may still be relevant even when their support is below but close to *minsupp*. However, the key problem of defining a good length-decreasing support function for a specific data set has not been tackled by the authors.

7.4. Incremental FI mining

On-line mining algorithms were introduced to cope with data evolution at low cost, i.e., without starting from scratch. Early incremental *FI* miners were based on the APRIORI framework. FUP [13] (for *F*ast *U*pdate with *P*runing) updates the set of association rules whenever some new transactions are added. The candidates for the incremental transaction set are generated with respect to their frequencies in the initial database which are in turn deduced from some pre-computed support values for the TDB. FUP-2, the sequel of FUP, admits a larger set of operations on the database, including insertion, removal and modification of transactions.

An alternative on-line paradigm relies on the notion of *negative border* [24], i.e., the infrequent itemsets that are minimal for inclusion (see [17,34] for concrete methods). In [3], the UWEP incremental algorithm performs a look-ahead pruning by discarding as early as possible itemsets that will become infrequent. A recent work reported in [30] extends the limits of incremental approaches by allowing changes to the basic parameters of the mining process such as support threshold, and analyzing the impact of the increment on the mining process.

7.5. Incremental FCI mining

Based on the criteria described in [30], we believe that our approach has the following attractive features: (i) it is incremental, (ii) it allows flexible changes to the support threshold, and (iii) it helps capture the effects of the update. The last feature is enabled by the *Intersections* structure (see Section 6) which contains the newly discovered *FCIs* and the modified ones. Such a structure can be explored, for instance in a market basket analysis framework, in order to analyze the impact of some actions (e.g., new marketing strategies) taken between a previous mining process of a TDB and the current one (i.e., the mining of the increment only).

It is nevertheless a difficult task to combine strict computation of *FCIs* with the incremental mode. The key obstacle for the direct application of the genitor-modified-new framework, regardless of the concrete method, is a phenomenon that may be qualified as *CI drift*: Some *CIs* may bounce back and forth between the frequent and the unfrequent part of the *CI* family all along the incremental construction. In particular, infrequent genitors may give rise to frequent new *CIs* while modified concepts may show instability around the cut-off point by "crossing" it upon a single increment just to be back some steps later. In [31], we presented the schema of an incremental *FCI* miner. The study of various implementations thereof has shown that invariably the most expensive task in a single update step is the management of the drifting *CIs*. To limit its cost, we have designed a method exploiting the order between *CIs*. As order is not provided for in the above algorithmic proposition, further research will be necessary to determine the appropriate trade-off between parsimony in *FCI* traversal and drifting *CI* recovery.

8. Performance results of the incremental methods

In the light of previous work on the performance of *CI* and *FCI* miners, we discuss here the way our work compares to existing methods and known complexity results.

8.1. Complexity analysis

Before tackling the complexity of Algorithm 4, a relevant fact to recall is the potentially exponential growth of the lattice in the number of objects/attributes. Consequently, lattice methods cannot be polynomial, unless the size of the initial lattice becomes a factor in the complexity function. However, as the ultimate goal is to compare our algorithm to other methods, we shall consider it as a step of the larger process of batch computing of the *CI* family and stick to classical evaluation, i.e., ignore, if possible, the size of the intermediate results.

To start with the complexity order, let m, k and l be the sizes of $\mathscr{D}$, $\mathscr{I}$ and $\mathscr{C}^a_{\mathscr{D}}$, respectively. First, the cost of Algorithm 4 is split into two additive factors. The first one reflects the trie construction effort (lines 6–11). The outer loop is executed as many times as the size of I_n (bound by k). The inner loop iterates on the set of *CIs* having a given item, and will thus be executed at most l times. Line 8 is carried out in constant time, as are lines 9 and 11, since the corresponding trie operations take O(1) time. Recall that in the *Intersections* trie all nodes of the same label i are inserted during the traversal of the list associated to i in *ItemIndex*. Hence, the current item i can only be located at the end of the successor list at each node of the trie (since all other successors are indexed by items preceding i in I_n). Therefore, a lookup for a successor indexed by i, or the creation of such a successor, takes a constant time (one pointer access) provided a list pointer is maintained to this end. In short, trie construction is in $O(lk)$.

The second factor reflects the cost of creating the representation of new *CIs* and their integration into *ItemIndex* (lines 12–20). The outer loop is executed once for each intersection, whereby for modified ones only support update is carried out. For a new *CI*, the corresponding structure is created (constant time) and the node is inserted in all the

CI lists of the belonging items. The constitution of the new itemset amounts to a root-bound traversal of the trie and therefore has a $O(k)$ cost. The insertion in each list in *ItemIndex* has a constant time cost. Consequently, the second cost factor is also in $O(lk)$ since there are at most l modified and new concepts.

The global cost of UPDATE-FAMILYCI is thus in $O(lk)$. Based on this observation, a rough cost assessment for the complete construction of the *CI* family from scratch, i.e., by repeated transaction insertions, would put the result to $O(lkm)$ (as UPDATE-FAMILYCI is called m times). However, a finer estimation brings the cost to the less expensive $O(ln)$ where n is the size of the incidence relation ($n = |R|$), i.e., the total number of Xs in the context.

To come to this figure, the global cost is again split into the previous pair of factors which are now summed up along the set of object insertions. Thus, the cost of all m trie constructions is a function of the number of "hits" that will trigger a call of UPDATE-INTERSECTION. Given a *CI* t with an itemset I_t and its corresponding concept c in the context of $\mathscr{D}$, this number can be assessed as follows. Let $c = (I_t', I_t)$, then the number of times c will be "hit" in the *ItemIndex* is exactly the total number of Xs in the columns of the items from I_t, $\sum_{i \in I_t} |i'|$. Indeed, a concept is hit by exactly those objects that share at least one item with I_t, i.e., those having Xs in the columns of I_t. Moreover, each o of this category hits c a number of times equal to $|o' \cap I_t|$, hence the figure. Finally, the number $\sum_{i \in I_t} |i'|$ is clearly bounded by n. Thus, the global cost of all trie constructions is in $O(ln)$ instead of $O(lkm)$.

To assess the second factor, recall that a modified *CI* is processed in constant time while the creation of a new one has an $O(k)$ cost. *CIs* are created once, so globally they cost $O(lk)$. In contrast, a concept $c = (X, Y)$ is processed as modified a number of times that nears its extent size: $|X| - 1$, i.e., once for each non-creating object. Thus, the total time spent in modifying is in $O(lm)$, hence the second cost factor is in $O(l(k + m))$. Consequently, the cost of the *CI* family computation is in $O(ln)$. To the best of our knowledge, this is the lowest complexity figure for a *CI* mining algorithm, be it batch or incremental. Algorithm 2 is clearly of lesser interest since its complexity cannot be put below $O(lm)$ in the general case, hence an $O(lmk)$ total complexity of the *CI* mining. Indeed, whatever the data structures used in its implementation, these cannot help examining each existing *CI* at least once. The complexity of the plain intersection being evaluated to $O(m)$, this yields $O(lm)$ cost for Algorithm 2.

Taken as a *CI* listing procedure, UPDATE-FAMILYCI is polynomial in its input (with respect to complexity classes described in [16]), provided the initial *CIs* are considered as data. Otherwise, it is clearly output polynomial since it may take up to exponential time (in m) before the first new or modified *CI* is produced by the algorithm (because exponentially many *CIs* from the initial family may be hit by an object o). In contrast, the global construction of all *CIs* cannot be polynomial under any assumption. However, it is clearly incremental polynomial since the processing of a new transaction, i.e., UPDATE-FAMILYCI, takes polynomial time in the size of the current solution set, i.e., the *CI* family. Following the unbound initial delay of UPDATE-FAMILYCI the global algorithm cannot be polynomial delay.

As far as maintenance of the *FCIs* is concerned, a basic fact is that their number is not bound to any of the above parameters. Actually, depending on the minimal support value, the size of the *FCI* family may vary between zero and l. Hence, it is hard to assess the cost of Algorithm 4 with respect to the number of *FCIs*. However, as the experimental results in the next section indicate, even in the cases of large discrepancies in the sizes of both families, i.e., *CIs* and *FCIs*, it may still be more advantageous to run our tool a small number of times with the entire set of closures than to construct once from scratch the frequent ones with a batch miner.

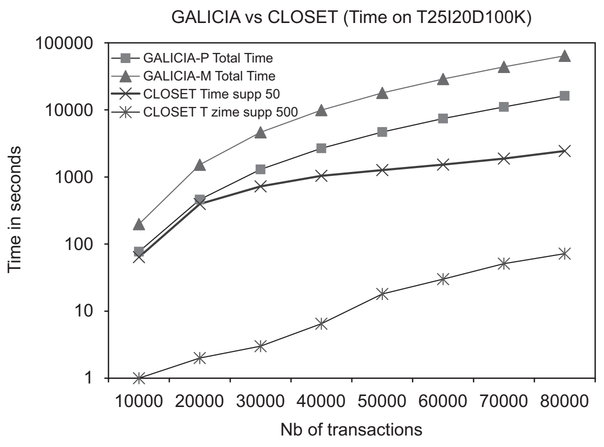
8.2. Experimental results

We conducted a set of tests in which both variants of GALICIA, further called GALICIA-M (a direct implementation of Algorithm 3) and GALICIA-P (Algorithm 4), have been compared to CLOSET [29]. The latter was chosen as a reference method for two reasons: First, like the algorithms in the GALICIA family, it works with *FCIs*. Then, both GALICIA-P and CLOSET use trie-like data structures. Moreover, CLOSET was chosen since it is one of the most efficient algorithms for *FCIs* generation. To ensure fair comparison, our own implementation was preferred to the available executable version of CLOSET. Thus, all three algorithms were implemented in `Java`$^{\text{TM}}$ and on top of the same low-level data structures, whereas CLOSET was enhanced by adding an additional trie structure that supports the inclusion tests between confirmed and candidate *FCIs*. The following experiments were carried out on a Windows 2000 platform (1.3 GHz AMD TB processor with 1.2 GB RAM).

Our performance study involves two synthetic databases, T25.I20.D100K and T25.I10.D10K, randomly generated by the tool described in [2]. T25.I20.D100K has 100,000 transactions over 10,000 items, hence it represents a sparse

Table 2

TDB size	nb. *CIs*	nb. *FCIs* supp. 50	nb. *FCIs* supp. 500
(A) Evolution of the numbers of *CIs* and *FCIs* for T25.I10.D10K (suport of 50)			
2000	281,209	544	
4000	826,114	2275	
6000	1,562,211	6977	
8000	2,479,770	14,701	
10,000	3,530,786	23,852	
(B) Evolution of the numbers of *CIs* and *FCIs* for T25.I20.D100K (supports of 50 and 500)			
10,000	420,144	22,326	11
20,000	1,148,803	73,851	52
⋮	⋮	⋮	⋮
90,000	10,895,757	271,074	22,998
100,000	12,868,438	313,409	27,112



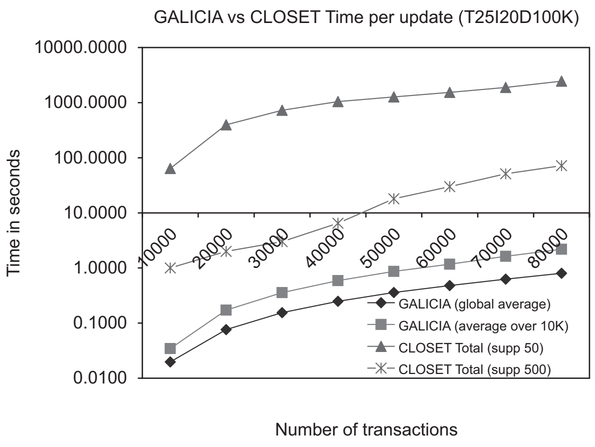
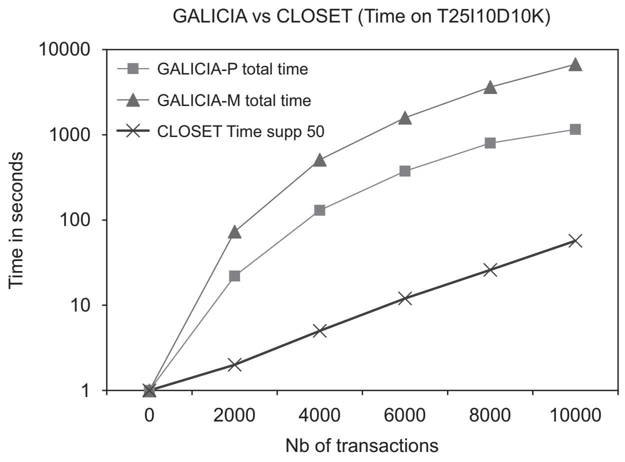


Fig. 5. Left: Total CPU-time for Galicia-P, Galicia-M, and Closet for increasing prefixes of T25.I20.D100K, with *min-supp* fixed to absolute values (50 and 500). Right: CPU-time for the insertion of a single transaction with Galicia-P, average over both the total set and the current batch of 10,000 transactions compared to the CPU-time for running Closet on the entire transaction set.

data set. Transactions have 25 items on average, and the average size of the maximal potentially FIs is 20. There are about 12.8M *CIs* in the lattice of T25.I20.D100K with some 0.3M of them being more frequent than 0.05% (50 transactions) and 27.1K more frequent than 0.5% (500 transactions). T25.I10.D10K, contains 10,000 transactions over 1000 items with average values of 25 and 10 for transaction and maximal FI sizes, respectively. It is therefore considerably smaller but *denser* than T25.I20.D100K. It generates a total of some 3.5M *CIs*, whereby only 23.8K of them have a support larger than 0.5% (50 transactions). A dynamic picture of the evolution in the *CI* and *FCI* figures is provided in Table 2 which follows a series of increasing subsets of the entire data sets. Technically speaking, the data sets were cut into segments of equal length, 2000 transactions for T25.I10.D10K and 10,000 for T25.I20.D100K, and the respective numbers recorded at the end of each data set prefix, i.e., set of consecutive segments.

CPU-time was measured for three types of tasks performed with every combination of algorithm and data set: processing a single new transaction, processing an entire segment of new transactions, and processing the entire data set. Moreover, Closet used support thresholds of 50 for both data sets plus an additional 500 value for T25.I20.D100K.

Performances were compared under two viewpoints, each involving a specific set of metrics. The first one sees all the algorithms as batch *CI* miners processing various prefixes of a data set. Hence it compares the total CPU-time for every prefix. The graphs on the left-hand side of Figs. 5 (for T25.I20.D100K) and 6 (for T25.I10.D10K) indicate that both Galicia-P and Galicia-M are dominated by Closet in this setting. More precisely, for reasonable values of the



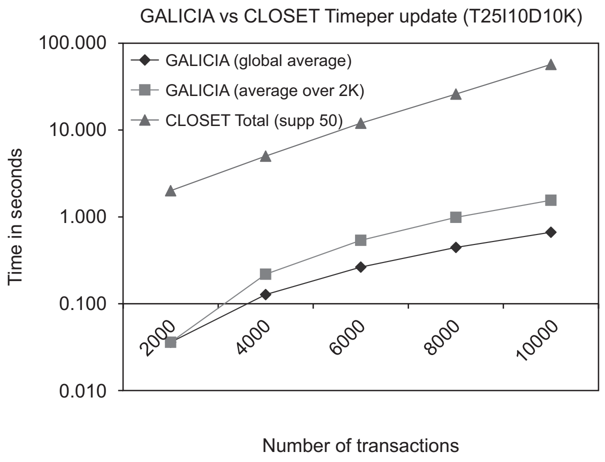


Fig. 6. Left: Total CPU-time for both GALICIA-P and CLOSET for increasing subsets of T25.I10.D10K, with *min-supp* fixed to an absolute value of 50. Right: CPU-time for the insertion of a single transaction, average over both the total set and the current increment of 2000 transactions, compared to the CPU-time for running CLOSET on the entire transaction set.

support threshold (comparable to an absolute value of 500 transactions), CLOSET proved to be up to 8 times faster on T25.I20.D100K and up to 20 times faster on T25.I10.D10K. Only tiny support values that force almost all the *CIs* to be retrieved tend to favor our method.

The second way of analyzing test results is oriented toward on-line processing. Indeed, it opposes the performance of GALICIA-P (GALICIA-M was ignored since the differences were insignificant) as an incremental *FCI* miner, i.e., the cost of integrating a transaction/segment, to the cost of re-running CLOSET on the whole updated database. The corresponding graphs are shown on the right-hand side of Figs. 5 (for T25.I20.D100K) and 6 (for T25.I10.D10K). Both diagrams show important trends. First, while the total time taken by CLOSET might lay orders of magnitude lower than that of GALICIA, it is also orders of magnitude higher than the update time for a single new transaction. For example, when T25.I20.D100K is considered, the processing of half the database, i.e., 50,000 transactions, may well take 1.5 h for GALICIA and only 20 min for CLOSET (see Fig. 5 on the left). In the same time, the insertion of a single transaction in GALICIA "costs" just below a second (0.4 s, Fig. 5 on the right). Next, with the sparse data set, the average insertion cost for GALICIA and the total mining cost for CLOSET are quasi-linear functions of the data set size.

Memory consumption in run-time was also considered for comparison purposes within this study. In a very general manner, we have registered a surge in the storage space required by GALICIA. For example, its consumption in the case of T25.I20.D100K exceeded the available 1 GB[1] for roughly 80K transactions with GALICIA-P (85K for GALICIA-M). The following table summarizes the total memory consumption of both algorithms on the various settings:

Data set	GALICIA-M	GALICIA-P	CLOSET supp. 50	CLOSET supp. 500
T25.I10.D10K	456 MB	368 MB	63 MB	–
T25.I20.D100K	1 GB	1 GB	823 MB	389 MB
	(swap after 85K trans.)	(swap after 80K trans.)		

8.3. Discussion

The above facts provide some evidence to support the benefits of the incremental approach. In fact, running CLOSET once with an augmented data set may cost up to 100 times more than the time spent for inserting a single transaction with GALICIA. In other words, one may run, say, several hundreds of insertions with GALICIA while CLOSET is working on the entire data set. Of course, this does not make our algorithm more efficient for the whole task as the total execution time

[1] This seems to be the maximally allowed RAM allocation for the Java VM on our platform.

remains too high. However, with a dynamic database, the mining process is spread over the entire database life-cycle (usually long) so that the main question becomes the establishment of a proper trade-off between the update costs and the urgent need for intermediate results.

When taken as a whole, the experimental results suggest that the benefits of the parsimonious update strategy in GALICIA-P, as opposed to the more straightforward one in GALICIA-M, are more substantial with sparse data sets than with dense ones. An important factor behind those gains, although probably not the only one, is the difference in complexity order for both algorithms. To spell it in figures, while GALICIA-M depends more heavily on the size of the context seen as a matrix, i.e., on the number of items/transactions (see Section 8.1), GALICIA-P only depends on the total number of items in the database. One may express the underlying dependency as follows: execution time gains of GALICIA-P with respect to GALICIA-M inversely depend on the ratio between the number of examined elements and the total size of the *CI* family. Clearly, the denser the data set, the larger the ratio. Therefore, we hypothesize that sparse data sets will generally tend to favor the parsimonious update in GALICIA-P.

9. Conclusion

Incrementality is a major challenge in data mining. The proposed framework for incremental FCI mining is a first step toward achieving that goal. The framework is based on FCA and lattices whose benefits for the association rule mining problem have already been demonstrated. Two concrete mining algorithms have been devised within the framework, one straightforward and the other one using a pruning mechanism, with an additional valuable feature which is the low-cost response to readjustments in the support threshold.

Appropriate implementation of the basic algorithms have been discussed and their respective practical performances were compared to those of a major batch algorithm. The results of a preliminary experimental study on two synthetic data sets of contrasted profiles revealed some potential benefits but also important limitations in the incremental paradigm. When taken as a whole, they seem to suggest that a straightforward incremental approach of the kind described here will most probably prove inefficient in purely static databases when the target support threshold is known beforehand. However, the approach will certainly be more appealing for database applications and data mining tasks where data stores are very dynamic and the mining task is carried out in an exploratory manner. More precisely, incremental mining procedures may be very helpful in environments where the user may want to frequently: (i) modify the support threshold of *FIs* for a given TDB and/or (ii) process new transactions in dynamic databases and analyze their impact on the mining result.

The scalability of our incremental approach is clearly obstructed by the necessity of maintaining the whole set of *CIs*. Therefore, we are now focusing on parsimonious updating only of its frequent part. As a first step, the MAGALICE algorithm [31] was devised. It helped identify the *CI* drift phenomenon and its high computational cost. Hence there is still room for further research on parsimonious traversal of *FCIs*. Another promising track seems to reside in the joint use of GALICIA and an efficient *FCI* miner, e.g., CLOSET, ACLOSE or CHARM. The latter could extract the *FCIs* plus their border from the known part of a data set while leaving the subsequent maintenance of the result to GALICIA. The idea reflects to a different yet somewhat related aspect of our lattice-based framework, i.e., the incremental integration of batches of transactions by lattice merge procedures (see [42]). The underlying framework offers a large choice of operations reflecting updates in the data set such as the removal of transaction batches.

Clearly other data mining tasks could be translated into the FCA framework. For example, the computation of non-redundant bases for association rules has already been researched from the perspective of the implication extraction in FCA [33,36]. A discussion of the potential benefits of applying FCA to a wide range of data mining tasks may be found in [40].

Acknowledgments

The research described here was partially funded by the Canadian NSERC (National Science and Engineering Research Council) grants held by the authors as well as the team grant from FQRNT (*Fonds de Recherche sur la Nature et les Technologies*) of Quebec. The authors would like to thank anonymous referees for their critical remarks that helped greatly improve the paper.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. Verkamo, Fast discovery of association rules, in: U. Fayyad, G. Piatetsky-Shapiro, P. Smyth (Eds.), Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, CA, USA, 1996, pp. 307–328.
[2] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94), Santiago, Chile, 1994, pp. 487–499.
[3] N. Ayan, A. Tansel, M. Arkun, An efficient algorithm to update large itemsets with early pruning, in: Proceedings, KDD-99, ACM Press, San Diego, CA, USA, 1999, pp. 287–291.
[4] M. Barbut, B. Monjardet, Ordre et Classification: Algèbre et combinatoire, Hachette, Paris, 1970.
[5] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, L. Lakhal, Mining frequent patterns with counting inference, SIGKDD Explorations 2 (2) (2000) 66–75.
[6] R. Bayardo, R. Agrawal, Mining the most interesting rules, in: Proceedings, KDD-99, ACM Press, San Diego, CA, USA, 1999, pp. 145–154.
[7] R.J. Bayardo, Efficiently mining long patterns from databases, in: Proceedings of the ACM SIGMOD 1998 Conference, 1998, pp. 85–93.
[8] G. Birkhoff, Lattice Theory, AMS Colloquium Publications, third ed., vol. XXV, AMS, Providence, RJ, 1967.
[9] E. Boros, V. Gurvich, L. Khachiyan, K. Makino, On the complexity of generating maximal frequent and minimal infrequent sets, in: Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 2285, Springer, Berlin, 2002, pp. 133–141.
[10] D. Burdick, M. Calimlim, J. Gehrke, MAFIA: a maximal frequent itemset algorithm for transactional databases, in: Proceedings of the 17th IEEE ICDE Conference (ICDE'01), Heidelberg, Germany, 2001, pp. 443–452.
[11] C. Carpineto, G. Romano, A lattice conceptual clustering system and its application to browsing retrieval, Mach. Learning 24 (2) (1996) 95–122.
[12] M. Chein, Algorithme de recherche des sous-matrices premières d'une matrice, Bull. Math. Soc. Sci. R.S. Roumanie 13 (1969) 21–25.
[13] D.W. Cheung, J. Han, V. Ng, C. Wong, Maintenance of discovered association rules in large databases: an incremental updating technique, in: Proceedings, ICDE-96, New Orleans, LA, USA, 1996, pp. 106–114.
[14] D.W. Cheung, S.D. Lee, B. Kao, A general incremental technique for maintaining discovered association rules, in: Proceedings, DASFAA-97, Melbourne, Australia, 1997, pp. 185–194.
[15] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, second ed., Cambridge University Press, Cambridge, 2002.
[16] T. Eiter, G. Gottlob, Identifying the minimal transversals of a hypergraph and related problems, SIAM J. Comput. 24 (6) (1995) 1278–1304.
[17] R. Feldman, Y. Aumann, A. Amir, H. Mannila, Efficient algorithms for discovering frequent sets in incremental databases, in: Proceedings of ACM SIGMOD Workshop DMKD'97, Tucson, AZ, USA, Avon Books, NY, 1997, pp. 59–70.
[18] B. Ganter, Two basic algorithms in concept analysis, preprint 831, Technische Hochschule, Darmstadt, 1984.
[19] B. Ganter, R. Wille, Formal Concept Analysis, Mathematical Foundations, Springer, Berlin, 1999.
[20] R. Godin, R. Missaoui, An incremental concept formation approach for learning from databases, Theoret. Comput. Sci. 133 (1994) 378–419.
[21] R. Godin, R. Missaoui, H. Alaoui, Incremental concept formation algorithms based on Galois (concept) lattices, Comput. Intell. 11 (2) (1995) 246–267.
[22] J. Hipp, U. Guentzer, G. Nakhaeizadeh, Algorithms for association rule mining—a general survey and comparison, SIGKDD Explorations 2 (1) (2000) 58–64.
[23] D.E. Knuth, The Art of Computer Programming, Sorting and Searching, vol. 3, second ed., Addison-Wesley, Reading, MA, 1998.
[24] H. Mannila, H. Toivonen, A. Verkamo, Efficient algorithms for discovering association rules, in: U. Fayyad, R. Uthurusamy (Eds.), Proceedings, AAAI Workshop on Knowledge Discovery in Databases, AAAI Press, Seattle, WA, USA, 1994, pp. 181–192.
[25] E.M. Norris, An algorithm for computing the maximal rectangles in a binary relation, Rev.Roumaine Math. Pures Appl. 23 (2) (1978) 243–250.
[26] O. Öre, Galois connections, Trans. Amer. Math. Soc. 55 (1944) 493–513.
[27] N. Pasquier, Y. Bastide, T. Taouil, L. Lakhal, Efficient mining of association rules using closed itemset lattices, Inform. Systems 24 (1) (1999) 25–46.
[28] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Proceedings, ICDT-99, Jerusalem, Israel, 1999, pp. 398–416.
[29] J. Pei, J. Han, R. Mao, CLOSET: an efficient algorithm for mining frequent closed itemsets, in: Proceedings, ACM SIGMOD Workshop DMKD'00, Dallas, TX, USA, 2000, pp. 21–30.
[30] V. Pudi, J.R. Haritsa, Quantifying the utility of the past in mining large databases, Inform. Systems 25 (5) (2000) 323–343.
[31] M.H. Rouane, K. Nehme, P. Valtchev, R. Godin, On-line maintenance of iceberg concept lattices, in: Contributions to the 12th ICCS, Shaker Verlag, Huntsville, AL, 2004, pp. 14.
[32] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, L. Lakhal, Computing iceberg concept lattices with Titanic, Data Knowledge Eng. 42 (2) (2002) 189–222.
[33] R. Taouil, N. Pasquier, Y. Bastide, L. Lakhal, Mining bases for association rules using closed sets, in: Proceedings, ICDE-00, IEEE Computer Society, San Diego, CA, USA, 2000, p. 307.
[34] S. Thomas, S. Bodagala, K. Alsabti, S. Ranka, An efficient algorithm for the incremental updation of association rules in large databases, in: Proceedings, KDD-97, New Port Beach, CA, USA, 1997, pp. 263–266.
[35] P. Valtchev, An algorithm for minimal insertion in a type lattice, Comput. Intell. 15 (1) (1999) 63–78.
[36] P. Valtchev, V. Duquenne, Implication-based methods for the merge of factor concept lattices, 32pp, submitted.
[37] P. Valtchev, M.R. Hacene, R. Missaoui, A generic scheme for the design of efficient on-line algorithms for lattices, in: A. de Moor, W. Lex, B. Ganter (Eds.), Proceedings of the 11th International Conference on Conceptual Structures (ICCS'03), Lecture Notes in Computer Science, vol. 2746, Springer, Berlin, DE, 2003, pp. 282–295.

[38] P. Valtchev, R. Missaoui, Building concept (Galois) lattices from parts: generalizing the incremental methods, in: H. Delugach, G. Stumme (Eds.), Proceedings of the ICCS'01, Lecture Notes in Computer Science, vol. 2120, Springer, Berlin, 2001, pp. 290–303.
[39] P. Valtchev, R. Missaoui, R. Godin, A framework for incremental generation of frequent closed itemsets, in: Proceedings of the First International Workshop on Discrete Mathematics and Data Mining, Washington, DC, USA, 2002.
[40] P. Valtchev, R. Missaoui, R. Godin, Formal concept analysis for knowledge discovery and data mining: the new challenges, in: P. Eklund (Ed.), Concept Lattices: Proceedings of the Second International Conference on Formal Concept Analysis (FCA'04), Lecture Notes in Computer Science, vol. 2961, Springer, Berlin, 2004, pp. 352–371.
[41] P. Valtchev, R. Missaoui, R. Godin, M. Meridji, Generating frequent itemsets incrementally: two novel approaches based on Galois lattice theory, J. Experimental Theoret. Artificial Intell. 14 (2–3) (2002) 115–142.
[42] P. Valtchev, R. Missaoui, P. Lebrun, A partition-based approach towards building Galois (concept) lattices, Discrete Math. 256 (3) (2002) 801–829.
[43] J. Wang, J. Han, J. Pei, CLOSET+: searching for the best strategies for mining frequent closed itemsets, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03), Washington, DC, USA, 2003, pp. 236–245.
[44] J. Wang, G. Karypis, Bamboo: accelerating closed itemset mining by deeply pushin the length-decreasing support constraint, in: International SIAM Conference on Data Mining, 2004.
[45] R. Wille, Restructuring lattice theory: an approach based on hierarchies of concepts, in: I. Rival (Ed.), Ordered Sets, Reidel, Dordrecht, Boston, 1982, pp. 445–470.
[46] M. Zaki, Generating non-redundant association rules, in: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00), Boston, MA, USA, 2000, pp. 34–43.
[47] M. Zaki, C.-J. Hsiao, CHARM: an efficiently algorithm for closed itemset mining, in: R. Grossman, J. Han, V. Kumar, H. Mannila, R. Motwani (Eds.), Proceedings of the Second SIAM International Conference on Data Mining (ICDM'02), 2002.