Theoretical Computer Science 133 (1994) 387419
Elsevier

An incremental concept formation approach for learning from databases*

Robert Godin and Rokia Missaoui

Département de Mathématiques et d'Informatique, Université du Québec à Mont&al, C.P. 8888, succursale "Centre Ville", Montréal, Canada, H3C 3P8

Abstract

Godin, R. and R. Missaoui, An incremental concept formation approach for learning from databases, Theoretical Computer Science 133 (1994) 353–385.

This paper describes a concept formation approach to the discovery of new concepts and implication rules from data. This machine learning approach is based on the Galois lattice theory, and starts from a binary relation between a set of objects and a set of properties (descriptors) to build a *concept lattice* and a set of rules. Each node (concept) of the lattice represents a subset of objects with their common properties.

In this paper, some efficient algorithms for generating concepts and rules are presented. The rules are either in conjunctive or disjunctive form. To avoid the repetitive process of constructing the concept lattice and determining the set of implication rules from scratch each time a new object is introduced in the input relation, we propose an algorithm for incrementally updating both the lattice and the set of generated rules. The empirical behavior of the algorithms is also analysed.

The implication problem for these rules can be handled based on the well-known theoretical results on functional dependencies in relational databases.

1. Introduction

Recent work in the field of databases shows an increasing interest in knowledge discovery from data [1, 2, 43]. The basic motivations for such an interest are: (i) in many organizations, databases are information mines that can be usefully exploited to discover concepts, patterns and relationships, (ii) the discovered knowledge may be

Correspondence to: R. Godin, Département Mathematiques et Informatique, Université du Quebec, C.P. 8888, Succ. A, Montreal, Que., H3C 3P8, Canada. Email: godin@info.uqam.ca.

*A preliminary version of this paper appeared in the Proceedings of the Workshop on Formal Methods in Databases and Software Engineering, Springer-Verlag, London.

0304-3975/94/$07.00 © 1994—Elsevier Science B.V. All rights reserved
SSDI 0304-3975(94)00057-P

efficiently used for many purposes such as business decision making, database schema refinement, integrity enforcement, and intelligent query handling.

Third generation database systems are expected to handle data, objects and rules, manage a broader set of applications [11], and deal with various kinds of queries such as intensional ones which are evaluated using the semantics of the data [40]. In databases (DB), there are two kinds of information: extensional information (data or facts) which represents real world objects, and intensional information which reflects the meaning, the structure (in terms of properties) and the relationships between properties and/or objects. In deductive databases, the intensional information takes the form of deduction rules defining new relations in terms of existing ones, integrity constraints expressing predicates the facts are assumed to verify, and sometimes class hierarchies, describing generalization/specialization relationships.

Research about the discovery of rules and concepts from large databases is relatively recent and is ranked among the most promising topics in the field of DBs for the 1990s [44]. According to [16], knowledge discovery is "the nontrivial extraction of implicit, previously unknown, and potentially useful information from data". Knowledge discovery techniques as they currently stand cannot be applied to many database applications. There are at least two reasons for this. One is the fact that DBs are generally complex, voluminous, noisy and continually changing. Two is the fact that the overhead due to the application of discovery techniques may be high. That is why researchers in this area [43] recommend that discovery algorithms for database applications be *incremental,* sufficiently *efficient* to have at most a quadratic growth with respect to the size of input, and *robust* enough to *cope* with noisy data.

The system RX [5] is one of the early works in knowledge discovery. It uses artificial intelligence techniques to guide the statistical analysis of medical collected data. Borgida and Williamson [7] uses machine learning techniques to detect and accommodate exceptional information that may occur in a database. Cai et al. [S] presents an induction algorithm which extracts classification and characterization rules from relational databases by performing a step by step generalization on individual attributes. Classification rules discriminate the concepts of one class from that of the others, while characteristic rules characterize a class independently from the other classes. In [27], the discovery process is incremental and includes two consecutive steps: conceptual clustering, and rule generation using the classification obtained at the first step. Ioannidis et al. [28] uses two machine learning algorithms, *viz.* COBWEB and UNMEM [18], to generate concept hierarchies from queries addressed to a database. The extracted knowledge is used for physical and logical database reorganization. Kaufman et al. [29] describes the INLEN system which integrates a relational database, a knowledge base as well as machine learning tools for manipulating data and knowledge, and for discovering rules, concepts and equations. In [30], the authors propose algorithms for abstracting class definitions from a set of instances. In [41], a survey of methods, theories and implementations of *inductive logic programming* (ILP) is given. ILP is a new descipline defined as the convergence of inductive learning and logic programming. Learning in that discipline

starts from examples and background knowledge to inductively build first-order clausal theories.

The main purpose of this paper is to present algorithms for generating implication rules from the Galois (concept) lattice structure of a binary relation. This article extends our previous work on knowledge discovery [38, 39]. Our approach is similar to the work done by [27] since it is incremental and based on a conceptual clustering procedure. However, the classification produced by [27] is a tree rather than a lattice. Like in [8], our approach helps learn characteristic rules (i.e. data summarization) as well as classification rules. The rules are either in conjunctive or disjunctive form.

The remainder of this paper is organized as follows. In the next section we give a background on the concept lattice theory and its relationship with machine learning techniques. Section 3 provides definitions for implication rules. Algorithms for rule and concept generation are presented in Section 4. Section 5 gives details about the empirical analysis of the algorithms. Finally, a brief discussion on further refinements is proposed.

2. **The concept lattice**

2.1. *Preliminaries*

From the context $(\mathcal{O}, \mathcal{D}, \mathcal{R})$ describing a set $\mathcal{O}$ of objects, a set $\mathcal{D}$ of properties and a binary relation $\mathcal{R}$ (Table 1) between $\mathcal{O}$ and $\mathcal{D}$, there is a unique ordered set which describes the inherent lattice structure defining natural groupings and relationships among the objects and their properties (Fig. 1). This structure is known as a concept lattice [45] or Galois lattice [4]. In the following we will use both terminologies interchangeably. The notation x $\mathcal{R}$ x' will be used to express the fact that an element x from $\mathcal{O}$ is related to an element x' from $\mathcal{D}$. Each element of the lattice $\mathcal{L}$ derived from the context $(\mathcal{O}, \mathcal{D}, \mathcal{R})$ [45] is a couple, noted (X, X'), composed of an object set X of the power set $\mathcal{P}(\mathcal{O})$ and a property (or descriptor) set $X' \in \mathcal{P}(\mathcal{D})$. Each couple (called *concept* by Wille [45]) must be a complete couple with respect to $\mathcal{R}$, which means that the following two properties are satisfied:

(i) $X' = f(X)$ where $f(X) = \{x' \in \mathcal{D} \mid \forall x \in X, x \mathcal{R} x'\}$,

(ii) $X = f'(X')$ where $f'(X') = \{x \in \mathcal{O} \mid \forall x' \in X', x \mathcal{R} x'\}$.

X is the largest set of objects described by the properties found in X', and symetrically, X' is the largest set of properties common to the objects in X. From this point of view, it can be considered as a kind of a maximally specific description [34]. The couple of functions (f, f') is a *Galois connection* between $\mathcal{P}(\mathcal{O})$ and $\mathcal{P}(\mathcal{D})$, and the Galois lattice $\mathcal{L}$ for the binary relation is the set of all complete couples [4, 45] with the following partial order.

Given $C_1 = (X_1, X'_1)$ and $C_2 = (X_2, X'_2)$, $C_1 \leqslant C_2 \Leftrightarrow X'_1 \subset X'_2$. There is a dual relationship between the X and X' sets in the lattice, i.e., $X'_1 \subset X'_2 \Leftrightarrow X_2 \subset X_1$ and therefore, $C_1 \leqslant C_2 \Leftrightarrow X_2 \subset X_1$. The partial order is used to generate the graph in the

following way: there is an edge from C_1 to C_2 if $C_1 < C_2$ and there is no other element C_3 in the lattice such that $C_1 < C_3 < C_2$. In that case, we say that C_1 is covered by C_2. The graph is usually called a Hasse diagram and the precedent covering relation means that C_1 is parent of C_2. When drawing a Hasse diagram, the edge direction is either downwards or upwards. Given, $\mathscr{C}$, a set of elements from the lattice $\mathscr{L}$, $\inf(\mathscr{C})$ and $\sup(\mathscr{C})$ will denote respectively the infimum (or meet) and the supremum (or *join*) of the elements in $\mathscr{C}$.

The fundamental theorem on concept lattices [45]. ***Let*** $(\mathscr{O}, \mathscr{D}, \mathscr{R})$ ***be a context. Then*** $\langle \mathscr{L}; \leqslant \rangle$ ***is a complete lattice for which infimum and supremum of any subset of*** $\mathscr{L}$ ***are given by***[1]

$$\bigwedge_{i \in I} (X_i, X'_i) = \left(f'f\left(\bigcup_{i \in I} X_i \right), \bigcap_{i \in I} X'_i \right),$$

$$\bigvee_{i \in I} (X_i, X'_i) = \left(\bigcap_{i \in I} X_i, ff'\left(\bigcup_{i \in I} X'_i \right) \right).$$

Many algorithms have been proposed for generating the elements of the lattice [6, 10, 15, 17, 32, 42]. However none of these algorithms incrementally update the lattice and the corresponding Hasse diagram, which is necessary for many applications. In [23], we have presented a basic algorithm for incrementally updating the lattice and Hasse diagram. More details about the basic algorithm and several variants are found in [19, 22]. When there is a constant upper bound on $\|f(\{x\})\|$ which is usually the case in practical applications, the basic algorithm and variants have an $0\ \|\mathscr{O}\|$ time complexity for adding a new object. Although some variants of the basic algorithm show a subtantial saving in time, the asymptotical behavior remains $0\ \|\mathscr{O}\|$. Extensive testing with several applications and simulated data has supported the linear growth with respect to $\|\mathscr{O}\|$ for the complexity of the incremental algorithms [3, 23, 25]. Surprisingly, our current experiments [3] on four existing algorithms for lattice construction show that, in most cases, our incremental algorithm is the most efficient and is always the best asymptotically. In this article, one efficient variant of the basic algorithm has been enriched to embody the generation of rules without increasing the time complexity (see Section 4 for more details).

2.2. A machine learning approach.

The concept lattice is a form of ***concept hierarchy*** where each node represents a subset of objects (extent) with their common properties (intent) [13, 45]. The Hasse diagram of the lattice represents a generalization/specialization relationship between

[1] Since we shall focus on generating rules for descriptors rather than for objects, the partial order as well as infimum and supremum definitions are given with respect to descriptors instead of objects as in Wille.

Table 1
The input relation

Objects	Attributes				
	F	*R*	***E***	***M***	*S*
Tiger	f_1	r_1	e_1	m_1	s_1
Horse	f_2	r_1	e_3	m_1	s_1
Sheep	f_2	r_1	e_3	m_1	s_0
Penguin	f_3	r_2	e_1	m_0	s_1
Frog	f_3	r_2	e_1	m_0	s_2
Rat	f_1	r_1	e_2	m_1	s_1

The attributes and their values have the following meaning:
F = Feet-f, = claw, f_2 = hoof, f_3 = web;
R = Ears $- r_1$ = external, r_2 = middle;
E = Eats-e, = meat, e_2 = grain, e_3 = grass;
M = Gives milk-m, = no milk; m_1 = milk;
S = Swims $- s_0$ = unable, s_1 = able, s_2 = well.

the concepts. Therefore, building the lattice and Hasse diagram corresponding to a set of objects, each described by some properties, can be used as an effective tool for symbolic data analysis and knowledge acquisition [24, 46].

The task of inducing a concept hierarchy in an incremental manner is called *incremental concept formation* [18]. Concept formation is similar to conceptual clustering which also builds concept hierarchies [35]. However, the former approach is partially distinguished from the latter in that the learning is incremental. Concept formation falls into the category of *unsupervised learning* also called learning from observation [9] since the concepts to learn are not predetermined by a teacher, and the instances are not pre-classified with respect to these concepts. As opposed to *explanation-based learning* methods [14], this approach falls into the class of *empirical inductive learning* [35] since no background knowledge is needed.

For illustration, a part of the well-known relation describing animals [27] will be used (see Table 1). The corresponding concept lattice is shown in Fig. 1. More details about the construction of the lattice are given in Section 4 and in [22].

3. Learning rules from the concept lattice

In addition to being a technique for classifying and defining concepts from the data, the concept lattice may be exploited to discover dependencies among the objects and the properties. The process may be undertaken in two different ways, depending on the peculiarities of the DB under consideration and the needs of the users: (i) scan the whole lattice or part of it in order to generate a set of rules that can be later used in

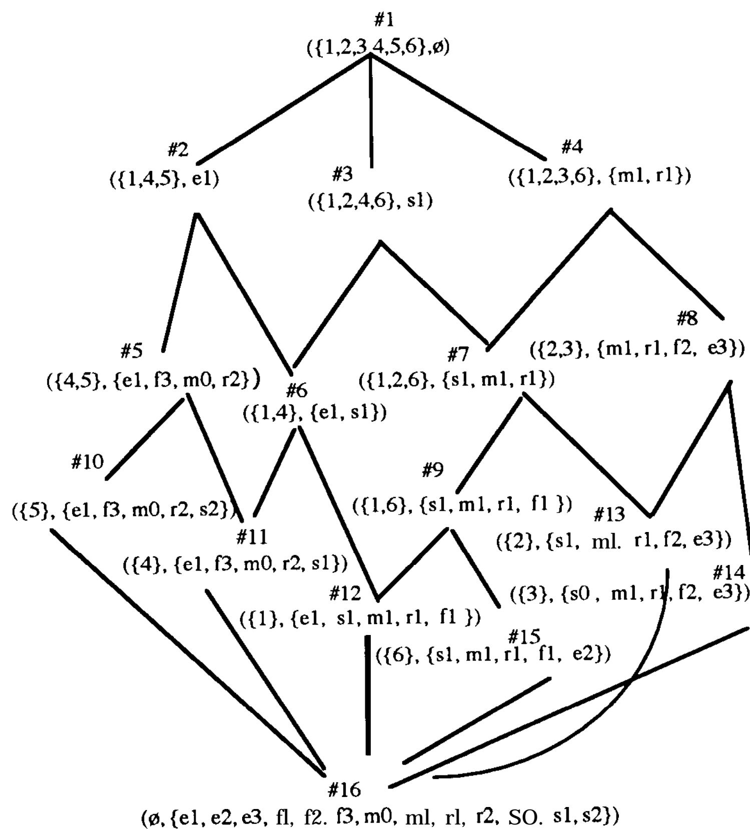

Fig. 1. The concept lattice.

a knowledge-based system, (ii) browse the lattice to check if a given rule holds, without necessarily generating the whole set of rules, but rather by looking for a node with some specific description. The first kind of operation is useful in a knowledge-based environment and helps enrich the knowledge base with the new generated rules and infer new facts. The motivation behind the second usage is that it often happens that one wants to confirm a hypothesis or invalidate a claim based on the analysis of input data. As an illustration of the two kinds of usage, suppose that from a DB application about divestment of units in a firm, the manager wants to confirm the following hypothesis: "*If* both the divested unit and the firm to which it belongs had a performance rate higher than the average in the industry, and the divested unit was created by internal development, *then* the motif of the divestment is a strategic reorientation." In the first case, we will try to show that this hypothesis, say d, can be derived from the set Σ of rules discovered from the data (i.e. $\Sigma \models d$). In the second case, we do not need to generate rules from the lattice (built from data), but rather we have to look for the smallest node (w.r.t. descriptors) containing the premise components

of that hypothesis, and check if the conclusion components also occur in the intent of that node.

In the first case, the learning process is as follows.

Input. A relation or a view of the database.
Output. (i) The corresponding concept lattice.
(ii) A set of conjunctive implication rules.
Method
Step 1. Construct the concept lattice of the binary relation.
Step 2. Generate a set of conjunctive rules from the lattice.
Step 3. Remove redundant rules.

Step 2 of the learning process can eiher be handled independently from (but as a sequel to) Step 1 as in Algorithms 4.1 and 4.2 or be integrated with Step 1 as in Algorithm 4.3.

In the following we use P, Q, R, . , Z to denote sets of properties, while we use lower-case letters p, q, r, to name atomic properties (descriptors). The notation pq is a simplification of the notation $\forall x$ *p(x) A q(x)* meaning that each object has properties p and q. In the sequel, we shall take the freedom of using either the logical notation or the set-oriented notation, depending on the context under consideration.

The general format of a rule is $P \Rightarrow Q$, where P and Q represent either a set of objects or a set of properties. Four cases can be considered:

(i) implication rules for descriptors (IRDS) in which both P and Q belong to $\mathscr{P}(\mathscr{D})$;
(ii) implication rules for objects (IROS) in which both P and Q belong to $\mathscr{P}(\mathscr{O})$;
(iii) discriminant rules for objects (DROS) where $P \in \mathscr{P}(\mathscr{D})$ and $Q \in \mathscr{P}(\mathscr{O})$;
(iv) discriminant rules for descriptors (DRDS) where $P \in \mathscr{P}(\mathscr{O})$ and $Q \in \mathscr{P}(\mathscr{D})$.

In the following, we give definitions for implication rules only. Then, we show that the rule generation problem is NP-hard. However, under a reasonable assumption, the problem becomes tractable. In Section 4 we propose a set of efficient algorithms for implication rule generation.

3.1. *Implication rules*

We define implication rules (IR) as ones such that P and Q are both subsets of either $\mathscr{O}$ or $\mathscr{D}$. Due to the nature of IRs, the inference system for functional dependencies holds also for IRs. The following definitions are borrowed from the implication theory on functional dependencies [31] and apply to implication rules as well.

Given a set Σ of IR, the closure Σ^+ is the set of rules implied by Σ by application of the inference axioms. Two sets Σ and Σ' are equivalent if they have the same closure.

- $P \Rightarrow Q$ is *redundant* in the set Σ of IRs if $\Sigma - \{P \Rightarrow Q\} \models P \Rightarrow Q$.
- $P \Rightarrow Q$ is *full* (or left-reduced) if $\nexists P' \subset P$ such that (C- $\{P \Rightarrow Q\}) \cup \{P' \Rightarrow Q\} \equiv \Sigma$.
- $P \Rightarrow Q$ is *right-reduced* if $\nexists Q' \subset Q$ such that $(\Sigma - \{P \Rightarrow Q\}) \cup \{P \Rightarrow Q'\} \equiv \Sigma$.
- $P \Rightarrow Q$ is *reduced* if it is left-reduced and right-reduced, and Q $\neq \emptyset$.

- $P \Rightarrow Q$ is *trivial* if $Q \subseteq P$.
- P-Q is *elementary* if it is full and $\| Q \| = 1$ and $Q \not\subseteq P$.

The closure P^+ of a set P according to Σ is defined by: $P^+ = P \cup \{ Q \mid Z \subseteq P^+$ and $Z \Rightarrow Q \in \Sigma\}$.

3.1.1. *Conjunctive implication rules*

The problem of generating conjunctive rules can be stated as starting from a finite model where each atom in the model is denoted by p(a), and finding Horn clauses of the form:

$$\forall x\, p_1(x) \wedge p_2(x) \wedge \cdots \wedge p_m(x) \Rightarrow q(x).$$

Each $p_i(x)$ is a property (descriptor) predicate indicating that x may or may not have property p_i. A more accurate notation for p_i would be $a_i(x) =$ v, where $a_i(x)$ is an attribute predicate, and v is a value from the domain of the attribute a_i. For example (see Table 1), $Feet(x) =$ "f_3" is a predicate about webbed animals that we shall write simply as $f_3(x)$ or as f_3.

Two alternative notations can be used to express conjunctive implication rules: the *compact* form and the *Horn clause* form. Each (compact) rule $P \Rightarrow Q$ may have a conjunction of properties in the conclusion part, and can equivalently be expressed as $\| Q \|$ separate Horn clauses $P \Rightarrow q_i$. The compact notation is commonly used in the literature about machine learning and knowledge discovery [35, 43], while the Horn clause notation is frequently adopted in deductive databases and logic programming studies [37, 41].

Definition 1. A *conjunctive* implication rule for *descriptors* (IRD) is an IR of the form D-D' where D and D' are subsets of 9. A context $(\mathcal{O}, \mathcal{D}, \mathcal{R})$ satisfies the IRD $D \Rightarrow D'$ if for every object x in $\mathcal{O}$, whenever x is characterized by all the properties found in D it has necessarily the whole set D' of properties, i.e.,

$$f(x) \supseteq D \Rightarrow f(x) \supseteq D'.$$

Proposition 1. $D \Rightarrow D'$ is a conjunctive IRD $\Leftrightarrow [[(O'', D'') = \inf \{(X, X') \in \mathcal{L} \mid D \subset X'$ and $X \neq \emptyset\}] \Rightarrow D' \subset D'']$.

In other words, the rule $D \Rightarrow D'$ holds if and only if the smallest concept (w.r.t. properties) containing D as a part of its intent is also described by D'.

Proof. ($\Rightarrow$) If $D \Rightarrow D'$ is computed based on the node described by some couple (O'', D''), it means that the objects in 0" have the properties found in D and D', and therefore $D \cup D' \subseteq D''$. In particular, $D' \subset D''$. To ensure that every $x \in \mathcal{O}$ that has the properties in D is also characterized by D', the couple (O'', D'') must be the smallest concept containing D.

($\Leftarrow$) Whenever the couple (0", D") is the smallest concept containing D, it also contains D'. This corresponds exactly to the definition of the IRD: $D \Rightarrow D'$. □

Example 1. From the lattice shown in Fig. 1, one can generate the IRD $f_1 \Rightarrow m_1 r_1 s_1$ meaning that if an animal has claws, then it gives milk, has external ears, and is able to swim. This rule is valid since, when starting from the node at the top (i.e. $\inf(\mathscr{L})$) of the lattice, the first encountered node containing the property f_1, which is node #9, contains also the properties m_1, r_1 and s_1. However, the IRD $s_1 \Rightarrow m_1$ does not hold since node #3 is described by the property s_1 alone.

Definition 2. A *conjunctive* implication rule for *objects* (IRO) is an IR of the form $O \Rightarrow O'$ where 0 and 0' are subsets of 0. A context $(\mathscr{O}, \mathscr{D}, \mathscr{R})$ satisfies the IRO $O \Rightarrow O'$ if for every descriptor x' in $\mathscr{D}$, whenever x' is associated with the objects in 0 it is also associated with the objects in O', i.e.,

$$f'(x') \supseteq O \Rightarrow f'(x') \supseteq O'.$$

For example, the rule $dog \Rightarrow \{bulldog$, poodle$\}$ means that bulldogs and poodles have at least the properties attached to dogs, i.e. they are dogs with possibly additional characterizations. The following proposition is the dual of Proposition 1.

Proposition 2. $O \Rightarrow O'$ is a conjunctive IRO $\Leftrightarrow$ [[(0", D'') $= \sup\{(X, X') \in \mathscr{L} \mid O \subset X$ and $X' \neq \emptyset\}] \Rightarrow O' \subset O''$].

In other words, $O \Rightarrow O'$ if and only if the biggest node (with respect to descriptors) containing 0 as a part of its extent is also described by 0'.

Proof. A reasoning similar to the proof of Proposition 1 can be done to demonstrate the correctness of Proposition 2. □

Proposition 3. $D \Rightarrow D^+$ is a full (or left-reduced) conjunctive IRD if and only if $\forall Q \subseteq \mathscr{D}\; Q \subset D \Rightarrow Q^+ \subset D^+$.

Proof. ($\Rightarrow$) If $D \Rightarrow D^+$ is a full rule, it means that $\nexists Q \subset D$ such that $Q \Rightarrow D^+$. Since the closure of Q is by definition the set of properties that are implied by Q, then $Q \Rightarrow Q^+$ and therefore $Q^+ \subset D^+$.

($\Leftarrow$) Suppose that $\forall Q \subseteq \mathscr{D}\; [Q \subset D \Rightarrow Q^+ \subset D^+]$ holds. As a consequence, $Q \Rightarrow D^+$ does not hold, and therefore $D \Rightarrow D^+$ is a full rule. □

Definition 3. $P \Rightarrow Q$ is an *existence* implication rule if

$$\nexists Z \subset P \text{ such that } Z^+ \subset P^+.$$

A *composite* rule is one for which the precedent condition does not hold.

For example, $m_1 \Rightarrow r_1$ is an existence rule (see the lattice in Fig. 1) while $e_1 m_1 \Rightarrow r_1 s_1 f_1$ is a full composite rule since there exists $Z = \{m_1\}$ in the premise $P = \{e_1 m_1\}$ of the second rule such that $Z^+ \subset P^+$, i.e., $\{m_1 r_1\} \subset \{e_1 m_1 r_1 s_1 f_1\}$. Intuitively, an algorithm that computes existence rules associated with a node H ignores any rule $P \Rightarrow Q$ whenever there exists at least a subset Z of P such that Z^+ is the intent of an ancestor node of H.

NP-hardness of the rule generation problem

It is known that the set-covering problem is an optimization problem that generalizes and models many NP-complete problems [12].

An instance $(Y, \mathscr{F})$ of the set-covering problem contains a finite set Y and a collection $\mathscr{F}$ of subsets $S_1, \ldots, S_m$ of Y, such that every element of Y exists in at least one subset in $\mathscr{F}$:

$$Y = \bigcup_{S_i \in \mathscr{F}} S_i$$

The problem is to find a minimum-size subset $Z \subseteq \mathscr{F}$ such that its members cover the set Y:

$$Y = \bigcup_{S_i \in \mathscr{Z}} S_i$$

This well-known problem can be useful for proving NP-completeness (and hardness) of the rule generation problem which can be stated as starting from a finite model where each atom in the model is denoted by p(a), and finding Horn clauses of the form:

$$\forall x \; p_1(x) \wedge p_2(x) \wedge \cdots \wedge p_m(x) \Rightarrow q(x).$$

The set Y to be covered corresponds to objects for which property q is false. A set S_i in $\mathscr{F}$ represents objects for which property p_i is false.

$$Y = \mathcal{O} - f'(q) = \{x \in \mathcal{O} \mid q(x) = False\}.$$

$$S_i = \{x \in Y \mid p_i(x) = False\}, \quad 1 \leqslant i \leqslant m.$$

The minimum size cover Z corresponds to the smallest-size union of S_i that covers Y. In other words,

$$Z = \{x \in Y \mid p_1(x) = False \;\vee\; \cdots \;\vee\; p_m(x) = False\}, \quad \text{or}$$

$$Z = \{x \in Y \mid \neg p_1(x) \;\vee\; \cdots \;\vee\; \neg p_m(x)\}.$$

Therefore, we get the following implication:

$$\forall x \neg q(x) \Rightarrow \neg p_1(x) \vee \cdots \vee \neg p_m(x).$$

The contraposition of the above logical expression leads to the following Horn clause:

$$\forall x \; p_1(x) \wedge p_2(x) \wedge \cdots \wedge p_m(x) \Rightarrow q(x).$$

Since we have shown that the rule generation problem can be reduced to the set-covering problem (which is NP-complete), the rule generation problem is therefore as hard as finding the minimum-size cover, and hence is an NP-hard problem.

Example: Let us check if $e_1(x) \& m_1(x) \Rightarrow f_1(x)$ holds for each object x in Table 1.

$$Y = \{x \in \mathcal{O} | f_1(x) = False\} = \{2, 3, 4, 5\}.$$

The minimum size cover for Y is Z= $S_1 \cup S_2$, where:

$$S_1 = \{x \in Y | e_1(x) = False\} = \{2, 3\}, \text{ and } S_2 = \{x \in Y | m_1(x) = False\} = \{4, 5\}.$$

Therefore,

$$\forall x \, e_1(x) \wedge m_1(x) \Rightarrow f_1(x).$$

If we assume there exists an upper bound K on the number of descriptors per object, i.e. $\|f(\{x\})\| \leqslant K$, we limit the size of any rule to contain at most K atoms, and therefore the problem of rule generation becomes tractable. This assumption is true in the context of databases since the number of pairs *(attribute, value)* per row (object) is bounded. Without this restriction, the general problem of rule generation is NP-hard as demonstrated before.

3.1.2. *Disjunctive implication rules*

Disjunctive implication rules are rules such that either their left-hand side (LHS) or their right-hand side (RHS) contains a disjunctive expression $Q_1 \vee \cdots \vee Q_i \vee \vee Q_m$, where Q_i is possibly a conjunction of atomic properties. There is a mapping between conjunctive and disjunctive rules. The conjunctive IRD $d_j \Rightarrow Q$ can be computed from the right-hand disjunctive IRD $d_j \Rightarrow Q_1 \vee \cdots \vee Q_m$ by setting Q to the properties common to $Q_1, \ldots, Q_m$. The left-hand disjunctive IRD $Q_1 \vee \cdots \vee Q_m \Rightarrow d_i$ means that $Q_1, \ldots, Q_m$ are the alternative (sets of) properties which subsume d_i, and can be computed from the set of conjunctive IRDs (see Algorithm 4.5).

3.2. *Characteristic and classification rules*

As mentioned earlier, *classification* rules discriminate the concepts of one class (e.g. a carnivore) from that of the others, while *characteristic* rules characterize a class independently from the other classes. The first kind is a *sufficient* condition of the class under consideration while the second type is a *necessary* condition of the class (see [8] for more details).

Implication rules in either conjunctive or disjunctive form can express these two kinds of rules. The RHS disjunctive IRD $d_j \Rightarrow Q_1 \vee \cdots \vee Q_m$ may be useful for defining the *characteristic* rule for objects having the property d_j, when there is no conjunctive IRD with d_j as a premise. The LHS disjunctive IRD $Q_1 \vee \cdots \vee Q_m \Rightarrow d_i$ can be used to define the *classification* rule for objects with the property d_i. E.g., the classification rule for animals having claws (property f_1) is $e_2 \Rightarrow f_1$ while the characteristic rule for carnivora (property e_1) can be expressed by the RHS disjunctive IRD $e_1 \Rightarrow f_3 s_1 \vee f_3 s_2 \vee f_1$.

4. Implication rule determination

In [26, 46], the authors deal with the problem of extracting rules from the concept lattice structure. However, they do not propose any algorithm to determine those rules. In this section we propose a set of algorithms for rule generation. For ease of exposition, we limit ourselves to IRDs. However, owing to the symmetry of the lattice structure, the definitions and algorithms can be adapted without difficulty to IROs.

Algorithm 4.1 computes a complete set of (existence as well as composite) IRDs from an already built lattice while Algorithms 4.1 a and 4.2 compute subsets of the output of Algorithm 4.1. In fact, Algorithm 4.1a determines the set of *reduced* IRDs while Algorithm 4.2 computes the set of *left-reduced existence* rules only. Algorithm 4.3 leads to the same output as Algorithm 4.1. However, it incrementally builds the lattice and computes the set of rules in one shot.

Based on Proposition 1 defined before, an obvious method for rule generation that immediately comes to mind is to systematically generate at each node $H=(X, X')$ the power set of X', and for each set P in $2^{X'}$, make sure that its value is not included in the intent of the parent nodes of H. The rules generated by this algorithm can be *composite* ones. Each rule of the form $P\Rightarrow Q$ can be converted into a set of Horn clauses $P\Rightarrow q_i$, for $1 \leqslant i \leqslant \|Q\|$.

Algorithm 4.1.
Input: A lattice $\mathscr{L}$.
Output: A set Σ of conjunctive IRDs: $P\Rightarrow Q$ (not necessarily reduced), and the array *Rules* $[1\cdots\|\mathscr{L}\|]$, where an element *Rules*$[H]$ represents the set of IRDs associated with the node H.

function *GenerateRulesForNode*$(N=(X, X'))$
/*Returns the complete set of rules generated from the node N*/
begin
A:=@;
if X $\neq\emptyset$ and $\|$X'$\| > 1$ **then** /* discard some trivial rules such as $P\Rightarrow\emptyset$ */
  For each nonempty set $P\in\{\mathscr{P}(X')-X'\}$ in ascending $\|P\|$ **do**
    if $\nexists M=(Y, Y')$ parent on N such that $P\subseteq Y'$ **then**
      if $\nexists P'\Rightarrow Q\in\Delta$ such that $P'\subset P$ **then**
        $A:=\Delta\cup\{P\Rightarrow X'-P\}$
      endif
    endif
  endfor
end if
return(A)
end {*GenerateRulesForNode*}

Table 2
Rules generated from the example lattice in Fig. 1

Node number	Rules generated by Algorithm 4.1	Reduced rules generated by Algorithm 4.1 a	Rules generated by Algorithm 4.2
4	$m_1 \Rightarrow r_1$	$m_1 \Rightarrow r_1$	$m_1 \Rightarrow r_1$
	$r_1 \Rightarrow m_1$	$r_1 \Rightarrow m_1$	$r_1 \Rightarrow m_1$
5	$f_3 \Rightarrow r_2, e_1, m_0$	$f_3 \Rightarrow r_2, e_1, m_0$	$f_3 \Rightarrow r_2, e_1, m_0$
	$m_0 \Rightarrow f_3, r_2, e_1$	$m_0 \Rightarrow f_3, r_2, e_1$	$m_0 \Rightarrow f_3, r_2, e_1$
	$r_2 \Rightarrow f_3, e_1, m_0$	$r_2 \Rightarrow f_3, e_1, m_0$	$r_2 \Rightarrow f_3, e_1, m_0$
7	$r_1, s_1 \Rightarrow m_1$		
	$m_1, s_1 \Rightarrow r_1$		
8	$f_2 \Rightarrow r_1, e_3, m_1$	$f_2 \Rightarrow e_3, m_1$	$f_2 \Rightarrow r_1, e_3, m_1$
	$e_3 \Rightarrow f_2, r_1, m_1$	$e_3 \Rightarrow f_2, m_1$	$e_3 \Rightarrow f_2, r_1, m_1$
9	$f_1 \Rightarrow r_1, m_1, s_1$	$f_1 \Rightarrow m_1, s_1$	$f_1 \Rightarrow r_1, m_1, s_1$
10	$s_2 \Rightarrow e_1, m_0, f_3, r_2$	$s_2 \Rightarrow f_3$	$s_2 \Rightarrow e_1, m_0, f_3, r_2$
11	$f_3, s_1 \Rightarrow r_2, e_1, m_0$		
	$r_2, s_1 \Rightarrow f_3, e_1, m_0$		
	$m_0, s_1 \Rightarrow f_3, r_2, e_1$		
12	$r_1, e_1 \Rightarrow f_1, m_1, s_1$	$r_1, e_1 \Rightarrow f_1$	
	$e_1, m_1 \Rightarrow f_1, r_1, s_1$	$e_1, m_1 \Rightarrow f_1$	
	$f_1, e_1 \Rightarrow r_1, m_1, s_1$		
13	$f_2, s_1 \Rightarrow r_1, e_3, m_1$		
	$e_3, s_1 \Rightarrow f_2, r_1, m_1$		
14	$s_0 \Rightarrow f_2, r_1, e_3, m_1$	$s_0 \Rightarrow f_2$	$s_0 \Rightarrow f_2, r_1, e_3, m_1$
15	$e_2 \Rightarrow f_1, r_1, m_1, s_1$	$e_2 \Rightarrow f_1$	$e_2 \Rightarrow f_1, r_1, m_1, s_1$

begin
$\Sigma := \emptyset$; /* the cumulative set of IRDs */
for each node $H = (X, X') \in \mathscr{L}$ in ascending $\|X'\|$ **do**
  begin
    Rules$[H]$:= *GenerateRulesForNode(H)*
    $\Sigma := \Sigma \cup Rules[H]$
  end
endfor
return(C)
end

Table 2 shows the rules generated from the lattice illustrated by Fig. 1.

The correctness of the algorithm follows from Propositions 1 and 3. The first proposition is expressed by the first test inside the *For* loop in the *GenerateRulesForNode* function, and ensures that $P \Rightarrow Q$ is valid by checking if the current node is the smallest concept containing P in its intent. Proposition 3 helps discard non full IRDs attached to a same node. In the *For* loop of the *GenerateRulesForNode* function, the possible subsets P in $\mathscr{P}(X')$ are considered in their ascending size to make sure that the IRDs with the smallest LHS part are produced first. The inner *If* test detects and

discards non full rules appearing in a same node. The algorithm also discards some trivial IRDs such as $P \Rightarrow \emptyset$ (which happens when $\|X'\| = 1$) and $P \Rightarrow P$ (which occurs when ***P=X').*** Since $P \Rightarrow P$ can be inferred by the reflexivity axiom, any valid rule produced from a concept (X, X') will have a ***RHS*** of the form $X'-P$.

Complexity analysis

The following lemmas will be useful for the complexity analysis of the algorithms.

Lemma 1. ***The number of elements in*** $\mathscr{L}$ ***is bounded by*** $2^K \times \|\mathscr{O}\|$, ***where K is an upper bound on*** $\|f(\{x\})\|$.

Lemma 2. ***For every node other than*** $sup(\mathscr{L})$, ***the number of ancestors is bounded by*** 2^K ***where K is an upper bound on*** $\|f(\{x\})\|$.

The proofs for these lemmas are trivial. A more detailed complexity anlysis of the lattice can be found in [20, 21]. If the upper bound ***K*** grows with $\|\mathscr{O}\|$, we obtain an exponential upper bound on $\|\mathscr{L}\|$ with respect to $\|\mathscr{O}\|$. However, in practical applications such as for databases, there is always an upper bound ***K*** on $\|f(\{x\})\|$ which is independent of $\|\mathscr{O}\|$, and therefore the lattice has a linear upper bound with respect to $\|\mathscr{O}\|$. In the case of relational databases, the number of attributes per relation is fixed and bounded by a constant. Furthermore, even though the exponential factor in ***K*** may seem to be a problem, experience with many applications and theoretical analysis using a uniform distribution hypothesis [24] has shown that $\|\mathscr{L}\|$ is usually less than $k \times \|\mathscr{L}\|$ where ***k*** is the average size of $\|f(\{x\})\|$. Therefore the number of iterations in the main loop is $0(\|\mathscr{O}\|)$. Each iteration invokes the *GenerateRulesFromNode* function. For almost all nodes $H = (X, X')$ in $\mathscr{L}$, we need to compute the power set $\mathscr{P}(X')$ and compare each element in it with the intent of the parents of ***H.*** Based on the assumption of a constant upper bound K, the number of iterations of the ***For*** loop in the function, the number of parents of a node and the size of $Rules[H]$ per node are all bounded by the constant 2^K. As a consequence, $\|\Sigma\|$ is $0(\|\mathscr{O}\|)$, and the time complexity of Algorithm 4.1 is $0(\|\mathscr{O}\|)$. These results are supported by the empirical study (see Figs. 3 and 8).

The *If* tests inside the *GenerateRulesFromNode* function help eliminate redundant rules produced by a same node. However, the rules generated from Algorithm 4.1 are not necessarily reduced and non-redundant. Redundancy may occur between rules generated from two different nodes of the lattice, and can be removed partly by Algorithm 4.1a or completely using the nonredundant cover algorithm [31].

The following algorithm takes the output of Algorithm 4.1 (or 4.3) and produces a complete set of reduced IRDs. The ***Reduce*** function aims at searching the ancestors of the current node to check if a subset of the ***RHS*** of the current rule can be inferred from already existing rules. After visiting the ancestors, if the new value of ***RHS*** is $\emptyset$, then the current rule is redundant and has to be removed.

Algorithm 4.1a

Input: A lattice $\mathscr{L}$, and the array Rules $[1 \ldots \|\mathscr{L}\|]$ generated by Algorithm 4.1 (or 4.3), where an element $Rules[H]$ represents the set of rules associated with the node H in $\mathscr{L}$.

Output: A complete set Σ of conjunctive and ***reduced*** IRDs: $P \Rightarrow Q$

```
function Reduce(H) /* Returns a set of reduced rules corresponding to node H*/
  function CheckParents(N, LHS, RHS)
  begin
    for each parent M = (Y, Y') of N such that LHS∩Y' ≠ ∅ do
      for each rule P⇒Q in Rules[M] do
        if Pc{LHS∪RHS} then
          RHS:= RHS-Q /* the common part to RHS and Q is redundant */
        endif
      endfor
      CheckParents(M, LHS, RHS)
    endfor
  end {CheckParents}
begin {Reduce}
  Δ := ∅; /* The set of IRDs for the current node */
  for each rule P⇒Q∈Rules[H] do
    CheckParents(H, P, Q);
    If Q ≠ ∅ then /* Q = ∅ means that the initial rule is redundant */
      Δ := Δ∪{P⇒Q}
      endif
  endfor
  return(d)
end {Reduce}
begin
  Σ := ∅; /* the cumulative set of IRDs*/
  for each node H = (X, X')∈ℒ do
    Σ := Σ∪Reduce(H)
  endfor
  return(C)
end
```

Complexity analysis

Based on Lemmas 1 and 2, the number of iterations in the main procedure is linear with respect to the size of 0, and each call of the function ***Reduce*** is done in a constant time since the number of ancestors of a node is bounded by 2^K. Therefore, the time complexity of Algorithm 4.1 a is 0 ($\|\mathscr{O}\|$), which is supported by the empirical analysis presented in Section 5.

Table 2 shows the rules generated from the lattice in Fig. 1 based on applying Algorithm 4.1 and then Algorithm 4.1a. As an example, the IRD $: e_1 m_1 \Rightarrow f_1 r_1 s_1$ is generated from node # 12 based on Algorithm 4.1. Using Algorithm 4.1a, this IRD will be reduced to $e_1 m_1 \Rightarrow f_1$ since $f_1 \Rightarrow m_1 r_1 s_1$ holds from node # 9. Rules generated by Algorithm 4.1 from nodes # 7, # 11, and # 13 are redundant, and therefore discarded by Algorithm 4.1a.

Algorithm 4.2 generates a subset of the whole set of IRDs discovered by Algorithm 4.1. This subset includes *existence* rules that are full but not necessarily right-reduced. Algorithm 4.2 uses a slightly modified description of nodes in $\mathscr{L}$. Each node H, instead of being a couple (X, X') is a triple (X, X', X") where X and X' have the same meaning as earlier, and X" is the set of properties encountered for the first time in the node H. Formally, X" is defined as follows: $X'' = X' - \{\bigcup_{i \in I} X'(N_i)\}$, where $X'(N_i)$ stands for the intent of the parent node N_i of H.

Algorithm 4.2

Input: A modified version of nodes in $\mathscr{L}$ where $H = (X, X', X'')$.

Output: A set Σ of conjunctive *existence* IRDs: $P \Rightarrow Q$, and the array *Rules* $[1 \ldots \| \mathscr{L} \|]$.

begin
  $\Sigma := \emptyset$; $P := \emptyset$;
  for each node $H = (X, X', X'') \in \mathscr{L}$ **do**
    begin
      Rules $[H] := \emptyset$;
      if $X \neq \emptyset$ and $X'' \neq \emptyset$ and $\| X' \| > 1$ **then**
        for each $x' \in X''$ **do**
          $Rules[H] := Rules[H] \cup \{\{x'\} \Rightarrow (X' - \{x'\})\}$
        endfor
        $P := P \cup X''$
        $\Sigma := \Sigma \cup$ *Rules* $[H]$
        if $P = \mathscr{D}$ **then return(C)**
        endif
      endif
    end
  endfor
return(C)
end

Complexity analysis

The number of iterations of the outer *For* loop is $\| \mathscr{L} \|$ which is $0(\| (\mathscr{O}) \|)$ as indicated earlier. Therefore, the time complexity of Algorithm 4.2 is $0(\| (\mathscr{O}) \|)$. Compared to Algorithm 4.1, Algorithm 4.2 has the same time complexity order but needs slightly more space to store the lattice nodes, and produces a subset of the output of Algorithm 4.1.

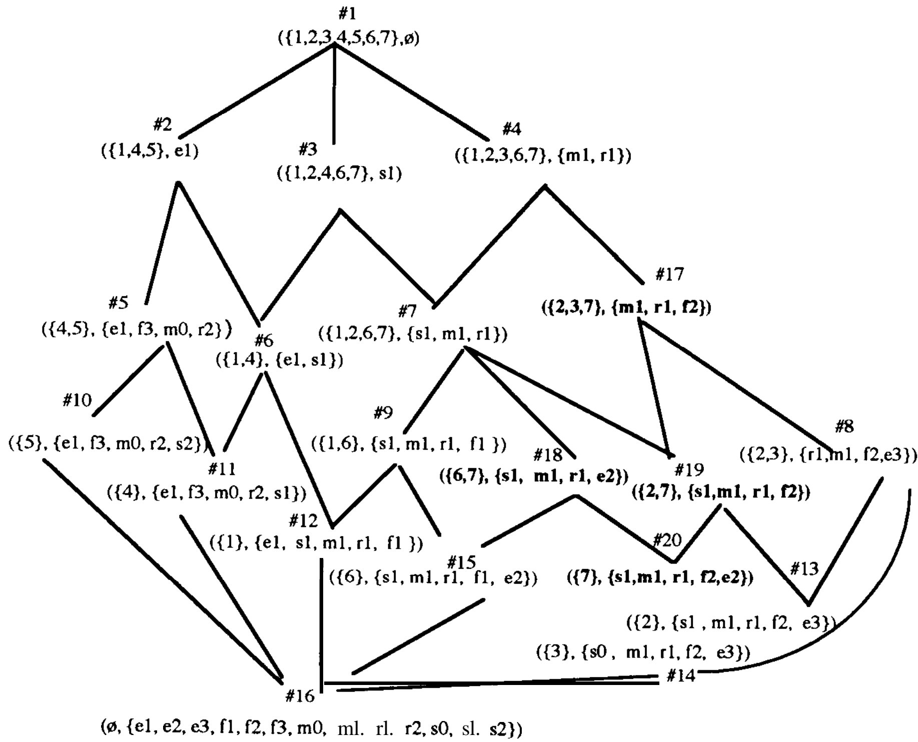

Fig. 2. The new concept lattice once the object #7 is added

The reader may notice in Table 2 that the rule $e_1 m_1 \Rightarrow s_1 f_1$ produced by Algorithm 4.1 is missing from the output of Algorithm 4.2 because it is a composite rule.

As opposed to semantic integrity constraints (e.g. functional dependencies) which can be viewed as invariant properties whatever the state of the database is, implication rules are dependencies that are inductively generated from, and hold for a particular database relation. Therefore, they have to be updated whenever the state of the database changes. To avoid the repetitive process of constructing the concept lattice $\mathscr{L}$ and determining the set Σ of implication rules from scratch each time a new object is introduced in the input relation, we propose an algorithm for incrementally updating both the lattice and the set Σ of generated rules.

Fig. 2 shows the new concept lattice once the object #7 is added.

Algorithm 4.3. Incremental update of concepts in $\mathscr{L}$ and rules in Σ when a new object x^* is introduced.

Input: A lattice $\mathscr{L}$, a set Σ of IRDs, and an array *Rules* $[1..\|\mathscr{L}\|]$ before the insertion of the new element x^* characterized by $f^*(\{x^*\})$.

Output: The Galois lattice $\mathscr{L}'$ for the new binary relation, the new value for Σ, and the array $Rules[1...\|\mathscr{L}\|]$.

```
Procedure SelectAndClassifyNodes
   Procedure Search(H = X, X'))
   begin
      Mark H as visited and add H to C[||X'(H)||];
      for each H_d child of H
         if H_d is not marked as visited
            Search(H_d)
         endif
      endfor
   end {Search}
   function GenerateRulesForNode (N=(X, X')) /* See Algorithm 4.1 */

   end {GenerateRulesForNode}
begin
   Mark in f(ℒ) as visited and add in f(ℒ) to C[||X'(in f(ℒ))||]
   for each x'∈f*({x*})
      Search (P_x')
   endfor
end {SelectAndClassifyNodes}
```

```
begin
 1. Adjust sup(ℒ) for new elements in 𝒟'
 2. if sup(ℒ)=(∅,∅) then
 3.    Replace sup(ℒ) by: H=(x*,f*({x*}));
 4.    for each x'∈f*({x*}) do
          Make P_x' point to H
       endfor
 5.    Rules[H] := GenerateRulesForNode[H]
 6.    Σ := Σ ∪ Rules [H]
 7. else
 8.    if f*({x*})⊈X'(sup(ℒ)) then
 9.       if X(sup(ℒ))=∅ then
10.          for each x'∈f*({x*}) such that x'∉(sup(ℒ)) do
                Make P_x' point to sup(ℒ)
             endfor
11.          X'(sup(ℒ)):=X'(sup(ℒ)) ∪ f*({x*})
12.       else
13.          Add a new node H=(∅,X'(sup(ℒ))∪f*({x*})) /* H becomes sup(ℒ) */
14.          Add a new edge ⟨sup(ℒ),H⟩
15.       endif
16.    endif
17.    SelectAndClassifyNodes;
```

```
18.  for i=0 to maximum cardinality do
       C'[i]:=∅; /* Initialize the C' sets */
     endfor
19.  (Treat each bucket in ascending cardinality order}
20.  for i=0 to maximum cardinality do
21.    for each node H∈C[i] do /* C[i]={H | ||X'(H)||=i} */
22.      if X'(H)⊆f*({x*}) then /* H is then a modified node */
23.        Add x* to X(H);
24.        Add H to C'[i];
25.        if X'(H)=f*({x*}) then exit endif
26.      else /* H is then an old node */
27.        Int := X'(H)∩f*({x*});
28.        if ∄ H_1 ∈ C' [||int||] such that X'(H_1) = Int then /* H is a generator node */
29.        Create New node H_n=(X(H)∪{x*}, int) and add it to C'[||int||];
30.         Add edge <H_n, H>;
31.         for each x'∈Int do
             if P_x' points to H
               Make P_x' point to H_n
             endif
           endfor
32.         {Modify edges}
33.         for j=0 to ||int||- 1 do
34.          for each H_a∈C'[j] do
35.           if X'(H_a)⊂ int then /* H_a is a potetial parent of H_n */
36.            parent := true;
37.            for each H_d child of H_a do
38.              if X'(H_d)⊂ int then
                   parent := false;
                   exit the for loop
                 endif
39.            endfor;
40.            if parent then
41.              if H_a is a parent of H then
                   eliminate edge (H,, H)
                 endif
42.              Add edge ⟨H_a, H_n⟩
43.            endif
44.           endif
45.          endfor
46.         endfor;
47.         {Modify rules for H}
48.         Σ := Σ - Rules[H]∪GenerateRulesForNode[H]));
49.         Rules[H] := GenerateRulesForNode[H]);
```

50. $Rules[H_n] := GenerateRulesForNode(H_n)$;
51. $\Sigma := \Sigma \cup Rules[H_n]$;
52. **if** $int = f^*(\{x^*\})$ **then** exit **endif**
53. endif
54. endif
55. endfor
56. endfor
57. endif
end

Table 3 shows the results of the incremental modification of the lattice in Fig. 1 and the generated rules when the new object #7 is added. New nodes and rules are indicated in bold style, while deleted rules are underlined.

The lattice update process

Algorithm 4.3 is a refinement of the incremental algorithm for updating the lattice proposed in [22]. The following gives details on the algorithm operation. First we

Table 3
Incremental update of the lattice and rules

Node numbers	Rules generated by algorithm 4.3 for 6 objects	Incremental modifications once object #7 is added
4	$m_1 \Rightarrow r_1$	$m_1 \Rightarrow r_1$
	$r_1 \Rightarrow m_1$	$r_1 \Rightarrow m_1$
	$f_3 \Rightarrow r_2, e_1, m_0$	$f_3 \Rightarrow r_2, e_1, m_0$
	$m_0 \Rightarrow f_3, r_2, e_1$	$m_0 \Rightarrow f_3, r_2, e_1$
	$r_2 \Rightarrow f_3, e_1, m_0$	$r_2 \Rightarrow f_3, e_1, m_0$
7	$r_1, s_1 \Rightarrow m_1$	$r_1, s_1 \Rightarrow m_1$
	$m_1, s_1 \Rightarrow r_1$	$m_1, s_1 \Rightarrow r_1$
8	$\underline{f_2 \Rightarrow r_1, e_3, m_1}$	$e_3 \Rightarrow f_2, r_1, m_1$
	$e_3 \Rightarrow f_2, r_1, m_1$	
9	$f_1 \Rightarrow r_1, m_1, s_1$	$f_1 \Rightarrow r_1, m_1, s_1$
10	$s_2 \Rightarrow e_1, m_0, f_3, r_2$	$s_2 \Rightarrow e_1, m_0, f_3, r_2$
11	$f_3, s_1 \Rightarrow r_2, e_1, m_0$	$f_3, s_1 \Rightarrow r_2, e_1, m_0$
	$r_2, s_1 \Rightarrow f_3, e_1, m_0$	$r_2, s_1 \Rightarrow f_3, e_1, m_0$
	$m_0, s_1 \Rightarrow f_3, r_2, e_1$	$m_0, s_1 \Rightarrow f_3, r_2, e_1$
12	$r_1, e_1 \Rightarrow f_1, m_1, s_1$	$r_1, e_1 \Rightarrow f_1, m_1, s_1$
	$e_1, m_1 \Rightarrow f_1, r_1, s_1$	$e_1, m_1 \Rightarrow f_1, r_1, s_1$
	$f_1, e_1 \Rightarrow r_1, m_1, s_1$	$f_1, e_1 \Rightarrow r_1, m_1, s_1$
13	$\underline{f_2, s_1 \Rightarrow r_1, e_3, m_1}$	$e_3, s_1 \Rightarrow f_2, r_1, m_1$
	$e_3, s_1 \Rightarrow f_2, r_1, m_1$	
14	$s_0 \Rightarrow f_2, r_1, e_3, m_1$	$s_0 \Rightarrow f_2, r_1, e_3, m_1$
15	$\underline{e_2 \Rightarrow f_1, r_1, m_1, s_1}$	$\mathbf{f_1, e_2 \Rightarrow r_1, m_1, s_1}$
17		$\mathbf{f_2 \Rightarrow r_1, m_1}$
18		$\mathbf{e_2 \Rightarrow r_1, m_1, s_1}$
19		$\mathbf{f_2, s_1 \Rightarrow r_1, m_1}$
20		$\mathbf{f_2, e_2 \Rightarrow r_1, m_1, s_1}$

explain how the lattice is incrementally updated, an then we look at the rule updating. The lattice $\mathscr{L}'$ can be obtained from $\mathscr{L}$ by taking all the nodes in $\mathscr{L}$ and modifying the X part of the nodes for which $X' \subseteq f^*(\{x^*\})$ by adding x^*. The nodes that remain unchanged are called *old* nodes (nodes #2, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, and 16 n Fig. 2) and the other ones are called *modified* nodes in $\mathscr{L}'$ (nodes # 1, 3, 4, and 7). In addition, new nodes are created (nodes # 17, 18, 19, 20). These new nodes are always in the form $(Y \cup \{x^*\}, Y' \cap f^*(\{x^*\}))$ for some node (Y, Y') in $\mathscr{L}$. They correspond to new X' sets with respect to $\mathscr{L}$. The related (Y, Y') node in $\mathscr{L}$ is called the *generator* for this new node. If the generators can be characterized in some manner, the new nodes can be generated from them. Proposition 4 is one possible characterization, and is used implicitly in Algorithm 4.3. This proposition is rather obvious but formal proofs are found in [22].

Proposition 4. *If* $(X, X') = \inf\{(Y, Y') \in \mathscr{L} \mid X' = Y' \cap f^*(\{x^*\})\}$ *for some set X' and there is no node of the form (Z, X') in* $\mathscr{L}$ *then (Y, Y') is the generator of a new node* $(X = Y \cup \{x^*\}, X' = Y' \cap f^*(\{x^*\})) \in \mathscr{L}'$.

Any new X' set in $\mathscr{L}'$ will have to be the result of intersecting $f^*(\{x^*\})$ with some Y' set already present in the lattice $\mathscr{L}$. There may be many nodes in $\mathscr{L}$ that give a particular new intersection in this manner. For example, in Fig. 2, the new X' set, $\{m_1, r_1, f_2\}$ corresponding to the new node # 17 can be formed by intersecting $f^*(\{x^*\}) = \{s_1, m_1, r_1, f_2, e_2\}$ with the Y' set of node #8 which is $\{r_1, m_1, f_2, e_3\}$, or the Y' set of node # 14 which is $\{s_0, m_1, r_1, f_2, e_3\}$. However, there is only one of these nodes that is the generator of the new node. It corresponds to the smallest old node that produces the intersection. Unicity of the greatest lower bound is guaranteed by the fondamental theorem (see Section 2). In Fig. 2, the generator nodes for the new nodes # 17, # 18, # 19, and #20 are #8, # 15, # 13, and # 16 respectively.

One minor problem is for the case of the generator being $\sup(\mathscr{L}) = (\emptyset, \mathscr{D})$. If the new instance x^* contains new features not contained in $\mathscr{D}$, there will be no generators for the node $(\{x^*\}, f^*(\{x^*\}))$. In practical applications, we may want $\mathscr{D}$ to grow as new features are encountered. This is easily taken into account by simply adding the new features to $\mathscr{D}$ as a first step in the algorithm, and therefore the characterization remains valid.

Besides updating the nodes, the edges of the Hasse diagram also have to be taken care of. First, the generator of a new node will always be a child to the new node in the Hasse diagram. The children of old nodes do not change. The parents of generator nodes, however, have to be changed. The generator is the only old node that becomes a child of a new node. There may be another child but it will be a new node. For example in Fig. 2, there is an edge from the new node # 17 to its generator # 8 and there is another child # 19 which is a new node. The parents of old nodes that are not generators remain unchanged.

Moreover, the parents of modified nodes never change. However, the children of some modified nodes may change. There may be new children that are new nodes. This implies that some old children have to be removed if the new children fall in between the old child and the modified node. This is the case when the new node falls between a generator and one of its parents. The result is that the edge from that parent to the generator (e.g., edge (# 7, # 13) in Fig. 2) is replaced by two edges, one from the parent to the new node (i.e., edge (#7, # 19)) and one from the new node to the generator (i.e., edge (# 19, # 13)).

Table 4 is a summary of the modifications resulting from the update process with respect to our categorization of nodes. The impact of the update on the X set, the X' set, the parents and children are shown in the four columns of the table. The first three categories (rows) represent the nodes that are in $\mathscr{L}$ and remain in $\mathscr{L}'$ with possibly some modifications. Cases 1 and 2 need the update of nodes or edges. The fourth case concerns new nodes.

The lattice is initialized with one element: $(\emptyset, \emptyset)$. This means that $\mathscr{O} = \mathscr{D} = \emptyset$. The algorithm updates $\mathscr{O}$ and $\mathscr{D}$ as new elements are added. If we assume that $\mathscr{O}$ and $\mathscr{D}$ contain in advance every element with an empty $\mathscr{R}$, the lattice would be initialized with the two elements: $(\mathscr{O}, \emptyset)$ and $(\emptyset, \mathscr{D})$. This would slightly simplify the algorithm because adjusting $\mathscr{D}$ by adding new elements from $f^*(\{x^*\})$ would not be necessary.

Lines 1-16 of the main procedure essentially take into account the case when new properties appear by adjusting $\mathscr{D}$ in $\sup(\mathscr{L})$. Line 17 calls the *SelectAndClassifyNodes* procedure. This procedure does the following tasks:

(1) First, it selects a subset of nodes from the lattice for the updating process, i.e. the nodes which have at least one property in common with the new object because the other nodes have no effect on the update process. The result is a huge saving as opposed to searching the whole lattice. To perform the selection without having to scan the whole lattice, for each x' in $\mathscr{D}$ a pointer $P_{x'}$ on the smallest node containing x' is maintained and these pointers are used as entry points for a top-down depth-first search starting with every x' in $f^*(\{x^*\})$. This guarantees that any node encountered will have at least x' in common with $f^*(\{x^*\})$. Maintaining these pointers is expressed in lines 4, 10 and 31 of the algorithm.

(2) Second, the nodes are sorted into buckets $C[\| X' \|]$ of same cardinality $\| X' \|$ because the following part of the algorithm needs to work level by level based on $\| X' \|$.

The main loop (lines 20-56) iterates on the nodes selected in *SelectAndClassifyNodes* by going through the C buckets in ascending $\| X' \|$. Lines 22-25 process the modified nodes. When the condition in line 25 holds, the rest of the treatment is skipped because the nodes under consideration cannot be generators. New nodes are obtained by systematically trying to generate a new intersection from each pair (Y, Y') already in the lattice by intersecting Y' with $f^*(\{x^*\})$ (line 27). Verifying that this intersection is not already present is done by looking at the sets already encountered

which are subsets off* $(\{x^*\})$ (line 28). These sets are kept in C' (line 24, line 29). This is valid only because the nodes are treated in ascending $\|X'\|$. Furthermore, the first node encountered which gives a new intersection is the generator of the new node because it is necessarily the infimum. Thus, we compute the X set of the new node by adding x^* to the generator's X set (line 29). Also, there is automatically an edge from the new node to the generator as explained earlier (line 30). When a new node is added, some edges have to be added from modified or other new nodes to the new node. The candidates are necessarily in the C' sets since their X' set must be a subset off* $(\{x^*\})$. These parents of the new node are determined by examining the nodes in C' (lines 32–46), testing if the X' sets are subsets of the X' set of the new node (line 35) and verifying that no child of the potential parent has this property (lines 36-39). It is necessary to eliminate an edge between the new parent and the generator when there is such an edge (lines 41 and 42).

The rule update process

The rule updating process is very simple compared to the lattice updating part. Algorithm 4.3 computes the same set of rules as Algorithm 4.1 and uses the same procedure for finding rules of a node, that is ***GenerateRulesForNode.*** As in Algorithm 4.1, the rules are related to the node which generates them, and represented by an array $Rules[1 \ldots \|\mathcal{L}\|]$. The *GenerateRulesForNode* procedure, as previously explained, finds the rules by looking at the parents of the current node. So the non new nodes which might have their rule set altered by the update operation are the generator nodes since their parents are altered (see Table 4). The rules for generator (old) nodes are treated in lines 47-49 where the old set of rules is replaced by the new

Table 4
Summary of the modifications of the update process

Type of node	X set	X' set	Parents	Children
1. Modified node (Y, Y')$\in G$	Add x^*	No change	No change	Add new nodes in some cases. Remove a generator when a new node in between
2. Old node generator of N	No change	No change	Add new node N and remove parent when N is in between this parent and the generator	No change
3. Old node non-generator of N	No change	No change	No change	No change
4. New node having generator (Y, Y')	$Y \cup \{x^*\}$	$Y' \cap f(\{x^*\})$	Old nodes and new nodes	Generator and possibly new node

set computed from the *GenerateRulesForNode* function. There are also new rules which might be generated from the new nodes. This is done in lines 50 and 51 of the algorithm using the same *GenerateRulesForNode* function. Lines 5 and 6 take care of the special case for the first object added to the lattice.

Complexity analysis

The time complexity of iterating on the nodes for creating the intersections and verifying the existence of the intersection in C' is the major factor in analyzing the complexity of the algorithm. Although the linking process is a bit tedious, the number of nodes affected is bounded by 2^K and this part is only done when a generator node is encountered. This is why we give a fairly straight-forward algorithm for this process. The rule generation is done only for generators and new nodes. Given the upper bound K, the number of nodes treated is $0(\|\mathcal{O}\|)$, $\|C'\|$ is bounded by 2^K, the number of generators and new nodes are also bounded by 2^K and the rule generation using *GenerateRulesForNode* is also bounded by a constant as explained earlier. Therefore the total process is $0(\|\mathcal{O}\|)$.

We have proposed so far algorithms for generating rules in conjunctive forms. In the following we present procedures aimed at detecting rules in (exclusive) *disjunctive* forms as well.

Algorithm 4.4.
Input: A descriptor d_j in $\mathcal{D}$, and a lattice $\mathcal{L}$.
Output: A disjunctive RHS rule of the form $d_j \Rightarrow Q_1 \vee \cdots \vee Q_m$.
begin
RHS := *True;*
  for each parent node N_i of $\sup(\mathcal{L})$ such that $X(\sup(\mathcal{L}))=\emptyset$ **do**
    if $d_j \subseteq X'(N_i)$ **then** /* $X'(N_i)$ stands for the intent of N_i */
      $RHS := RHS \vee (X'(N_i) - \{d_j\})$
    endif
  endfor
return$(d_j \Rightarrow RHS)$
end

To collect the disjunction of the different conjunctions of descriptors that d_j implies, Algorithm 4.4 selects all the parent nodes N_i of $\sup(\mathcal{L})$ such that these nodes include d_j in their intent, and then takes the descriptors other than d_j. This algorithm is $0\,\|\mathcal{O}\|$ and is particularly useful when there is no (nontrivial) conjunctive IRD with d_j as a premise. E.g., the characteristic rule for carnivora is $e_1 \Rightarrow f_3 m_0 r_2 s_1 \vee f_3 m_0 r_2 s_2 \vee f_1 s_1 m_1 r_1$ which can be simplified (using existing IRDs) into $e_1 \Rightarrow f_3 s_1 \vee f_3 s_2 \vee f_1$, meaning that carnivora are either webbed animals able to swim, or animals with claws.

Algorithm 4.5
Input: A descriptor d_i in $\mathscr{D}$, and a set Σ of conjunctive IRDs.
Output: The disjunctive LHS rule $Q_1 \vee \cdots \vee Q_m \Rightarrow d_i$.

```
begin
Old := ∅; New := d_i; LHS := True;
  while Old ≠ New or Σ ≠ ∅ do
    begin
    Old:= New
      for each {P⇒Q} ∈ Σ do
        if ∃S ∈ New such that Q ⊆ S then
        begin
          Σ := Σ − {P⇒Q}
          New := New ∪ {P ∪ (S−Q)}
          LHS := LHS ∨ {P ∪ (S−Q)}
        end
        endif
      endfor
    end
  endwhile
return (LHS⇒d_i)
end
```

This algorithm implicitly uses the same inference axioms as those related to functional dependencies to derive all the alternative Q_j that imply a discriptor d_i. For example, if $d_j = f$ and $\Sigma = \{a \Rightarrow b;\ bc \Rightarrow d; k \Rightarrow e; d \Rightarrow e; e \Rightarrow f\}$, then Algorithm 4.5 will produce $e \vee d \vee k \vee bc \vee ac \Rightarrow f$.

Complexity analysis

The While needs about $\|\mathscr{D}\|$ iterations while the *For* loop is executed $\|\Sigma\|$ times. Therefore, the overall complexity is $O(\|\mathscr{D}\| \times \|\mathscr{O}\|)$ which is reduced to $O(\|\mathscr{O}\|)$ if the assumption of a fixed bound K on the number of descriptors per object is retained.

5. Empirical analysis of the algorithms

The algorithms described in Section 4 have been implemented in Smalltalk within the environment of ObjectWorks (release 4.0). The prototype runs both on SUN SPARC work-stations and MacIntosh micro-computers equipped with 16 Megabytes RAM. The empirical comparison of the algorithms has been undertaken for a real-life application. The application concerns a data repository for a large database describing more than four thousand of attributes by means of a set of keywords, and

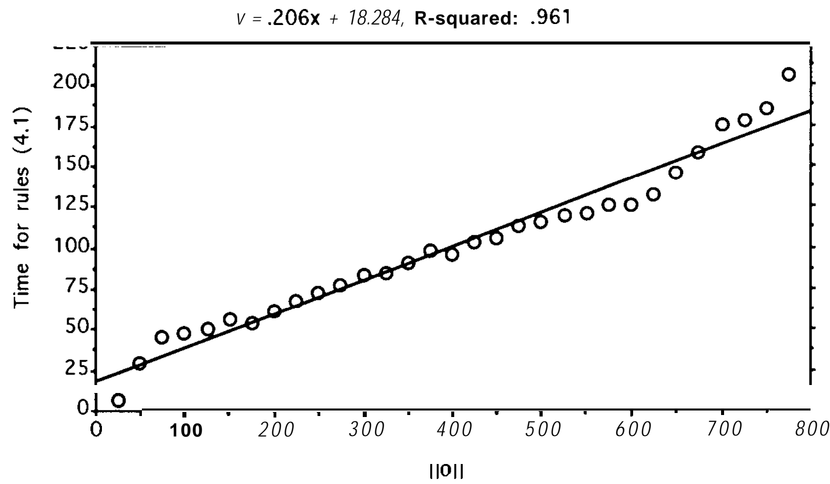

Fig. 3. Time for generating rules from the lattice using Algorithm 4.1.

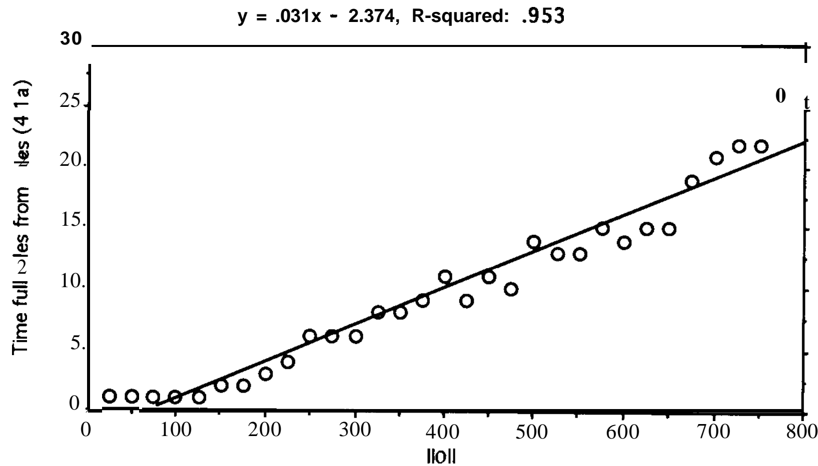

Fig. 4. Time for generating the left-reduced rules from the rules discovered by Algorithm 4.1 (or 4.3).

more than one hundred entities with their corresponding attributes. Only a sample of five to seven hundred attributes has been considered.

For lack of space, only the most significant results are given. Fig. 3 shows the time (in seconds) for generating rules from an existing lattice using Algorithm 4.1. The rules were generated for a varying number of objects by increments of 25. As expected, the growth is linear in $\|\mathcal{O}\|$. The value 0.961 for the R-squared is very significant. Obviously, given the unpredictable distributions of properties in a real world application, there are some variations which depend on how each new batch of objects relates to the previous ones. Fig. 4 shows the time for reducing the rules using Algorithm

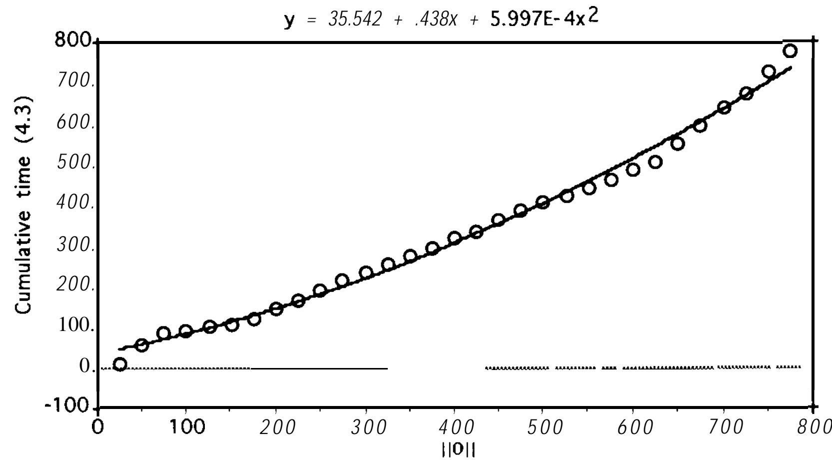

Fig. 5. Cumulative time for the incremental generation of the lattice and the rules using Algorithm 4.3.

4.1a. This process can be applied to the rules produced either by Algorithm 4.1 or Algorithm 4.3 since they both produce the same set of rules. Also as expected, the growth is linear in $\|\mathcal{O}\|$. A careful look at Figs. 3 and 4 indicates that the additional time needed to reduce rules is quite small compared to the rule generation itself. The R-squared value of 0.953 is very significant.

Fig. 5 shows the time for incremental update using Algorithm 4.3. The algorithm was applied repeatedly for each new object, and the total time was accumulated from the beginning of processing. Given the $O(\|\mathcal{O}\|)$ complexity of adding a new object, we should expect the total time to be $O(\|\mathcal{O}\|^2)$. This is supported by the empirical behavior and second order polynomial regression.

In order to see the advantage of using the incremental algorithm against the non incremental algorithm, we have generated the lattice incrementally using basically Algorithm 4.3 without the rule generation part, and after each new object was added to lattice, Algorithm 4.1 was applied for rule generation. The total cumulative time was measured including both the incremental generation of the lattice and the rule generation using Algorithm 4.1. Fig. 6 shows the results.[2] A comparison of Fig. 5 and 6 shows that the saving in computing resources is very substantial when using the incremental algorithm.

In this comparison, the underlying hypothesis is that the rules are generated each time there is a new object added. But suppose the rules are rarely needed. In such a case, it might be better to generate all the rules upon request for some batch of objects. In this context, it would be interesting to compare the following costs:

[2] Here, $\|\mathcal{O}\|$ is limited to 450 due to the overflow of the 16 bit integer time counter in Smalltalk

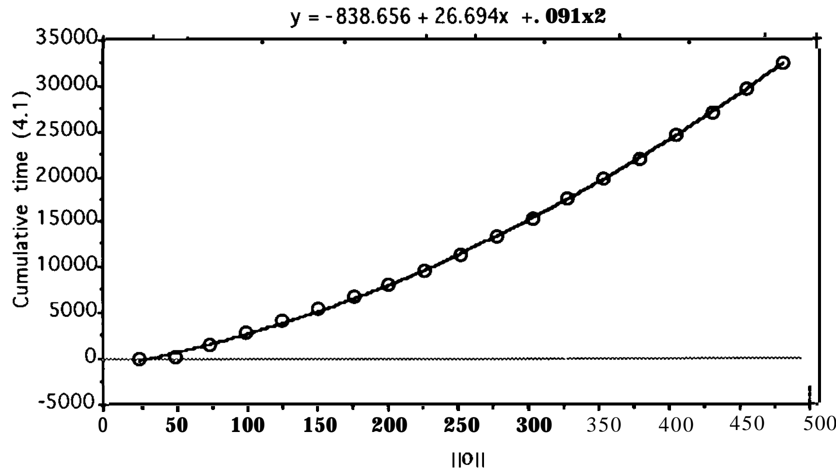

Fig. 6. Cumulative time for the incremental generation of the lattice plus the generation of rules using Algorithm 4.1 for each new object.

(1) The cost of generating the lattice plus the whole set of rules (using Algorithm 4.1) in one shot for some number of objects $\|\mathcal{O}\|$.

(2) The total cumulative cost for incrementally maintaining the lattice plus the cost for generating the whole set of rules (using Algorithm 4.1) for the same set of objects.

(3) The total cumulative cost for incremental generation of the lattice and rules (using Algorithm 4.3) for the same set of objects.

Based on the evidence of the relative performance of currently known batch algorithms versus our incremental algorithm for generating the lattice [3], surprisingly, the incremental algorithm outperforms the batch algorithms in most cases of practical interest when counting the total cumulative time for incremental update. It is therefore as efficient or better to use the incremental algorithm even for one shot batch generation. Case (1) can therefore be taken into account by case (2) where we think of the lattice batch update being done by the lattice incremental algorithm.

Fig. 7 compares case (2) and (3). One can notice that using the batch Algorithm 4.1 might be advantageous if the rules are rarely needed or if we only want to see the rules for a one shot analysis of some fixed set of objects. However, given a batch of new objects, when is it better to use the incremental algorithm for the rules versus using the regeneration from scratch? Fig. 8 helps answering this question. For the lattice update, in both cases we can use the same incremental algorithm. For rule generation, we isolated, in Algorithm 4.3, the additional time needed for incremental rule updating and the result is compared to the batch rule updating using Algorithm 4.1. For example, when there are 400 objects already treated, if we want to generate the whole rule set using Algorithm 4.1, it takes 96 seconds. The additional time in Algorithm 4.3 for incrementally updating the rule set is 15 seconds. Therefore, if we want to add a batch of 7 objects, the additional time for incrementally updating the rules would be

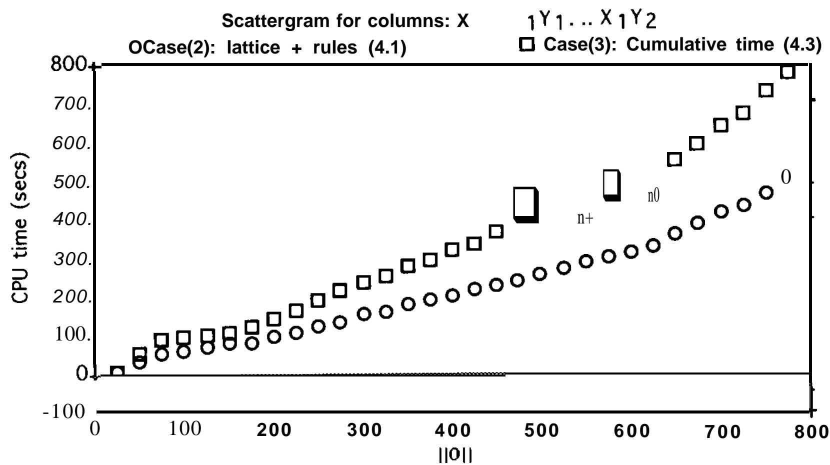

Fig. 7. Incremental generation of the lattice plus batch generation of rules using Algorithm 4.1. versus incremental generation of the lattice and rules.

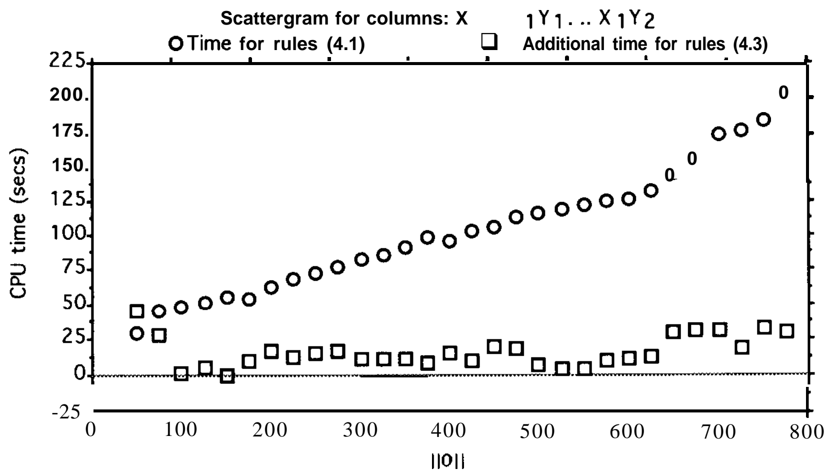

Fig. 8. Incremental versus batch comparison when isolating the rule processing.

about 10.5 (= 7*15) seconds compared to 96 seconds for regenerating the whole set of rules. It therefore becomes preferable to use the batch updating of the rules if we want to add about 7 or more objects before looking at the rules when there are 400 objects in the lattice. This number would become larger as the lattice grows because the growth factor of the batch generation of rules is larger than for the incremental update as can be observed from Fig. 8.

Finally, Fig. 9 compares the time for generating the rules using Algorithm 4.1 (composite and existence rules) and Algorithm 4.2 (existence rules only). If the subset generated by Algorithm 4.2 is useful enough, the savings are significant. For

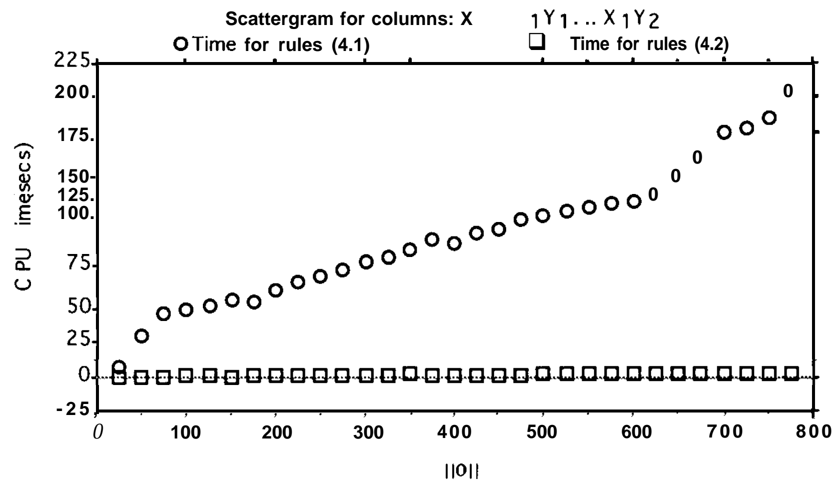

Fig. 9. Time for generating the rules from the lattice using Algorithm 4.1 versus Algorithm 4.2.

Algorithm 4.2, the time is less than two seconds as opposed to [6...205] range for Algorithm 4.1.

6. Conclusion

We have proposed an approach based on the concept lattice structure to discover concepts and rules related to the objects and their properties. This approach has been tested on many data sets found in the literature and has been proved to be as efficient and effective as some works related to knowledge mining [8, 43]. For example, all rules that can be generated by algorithms in [8, 27] are also produced by our algorithms. The algorithms presented in this paper have also been tested on real-life applications [25].

Our approach to rule generation can be very useful in database applications for discovering semantic integrity constraints such as implication dependencies and functional dependencies which are very common in DB applications. The knowledge discovered may be helpful for future learning and a better understanding of the semantics of the data. It may also be helpful in making more effective decisions with regard to scheme refinement, integrity enforcement, and semantic query optimization.

However, databases are basically used to store and retrieve a large amount of data. The schema of real-life applications is most likely complex in terms of the entities, the attributes, and the relationships among entities. Moreover, there may be a great number of possible modalities for the attributes. To overcome this complexity in the size and the structure of data, we believe that two kinds of pruning can be undertaken before or during the process of knowledge mining: input pruning and search space

pruning. The first one consists of discarding some input data in order to avoid both the processing of potentially useless data and the generation of more likely irrelevant concepts and rules. The second pruning happens once the concept lattice is produced, and consists into bypassing some concepts and ignoring some rules. There are some studies done on lattice pruning [33, 25, 36] which could be applied in this context. To deal with input pruning, we suggest the use of sampling techniques to reduce the size of the observation set, and exploratory data analysis techniques to get hints about attributes and objects that play a significant role in discriminating objects. In that way, only the objects and attributes that are most likely relevant and representative are selected. The search space pruning includes also the confirmation of a hypothesis $P \Rightarrow Q$ by selecting the smallest node (in the lattice) with a discription P without necessarily generating the whole set of rules. This task can be done in a constant time.

Our current research in the area of knowledge discovery includes: (i) generalizing the Galois lattice nodes structure to allow richer knowledge representation schemes such as conceptual graphs, (ii) dealing with complex objects, (iii) and testing the potential of these ideas in different application domains such as software reuse, database design, and intensional query answering.

Acknowledgements

We are greatful to the anonymous referees for their valuable comments and suggestions that helped a lot in improving the original paper. This research has been supported by NSERC (the Natural Sciences and Engineering Research Council of Canada) under grants Nos. OGP0041899 and OGP0009184.

References

[1] R. Agrawal, S. Ghosh, T. Imelinski, B. Iyer and A. Swami, An interval classifier for database mining applications, in: *Proc. 18th VLDB Conf. (1992)* 560–573.

[2] R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases, in: *Proc. ACM SIGMOD'93 Conf.* (1993) 2077216.

[3] H. Alaoui, Algorithmes de manipulation du treillis de Galois d'une relation binaire et applications, these de maitrise, Department de Mathematiques et d'Informatique, Université du Quebec à Montreal, 1992.

[4] M. Barbut and B. Monjardet, *Ordre et Classification. Algèbre et Combinatoire*, Tome II (Hachette, Paris, 1970).

[5] R.L. Blum, Discovery confirmation and incorporation of causal relationships from a large time-oriented clinical data base: The RX Project, *Computers Biomedical Research 15 (1982) 164-l 87.*

[6] J.P. Bordat, Calcul pratique du treillis de Galois d'une correspondance, *Mathématiques et Sciences Humaines 96 (1986) 3 l-47.*

[7] A. Borgida and K.E. Williamson, Accommodating exceptions in databases, and refining the schema by learning from them, in: *Proc. 11th Conf. On Very Large Data Bases,* Stockholm (1985) 72–81.

[8] Y. Cai, N. Cercone and J. Han, Attribute-oriented induction in relational databases, in: G. Piatetsky-Shapiro and W.J. Frawley, eds., *Knowledge Discovery from Databases* (AAAI Press/MIT Press, Menlo Park, CA, 1991) 213–228.

[9] J.G. Carbonell, Introduction: Paradigms for machine learning, in: J.G. Carbonell, ed., Machine *Learning: Paradigms and Methods,* (MIT Press, Cambridge, MA, 1990) 1–9.
[10] M. Chein, Algorithme de recherche des sous-matrices premières d'une matrice, Bull. Math. *Soc. Sci. R.S.* Roumanie 13 (1969) 21-25.
[11] Committee for Advanced DBMS Function, Third Generation Database System Manifesto, *SIGMOD* RECORD 19 (1990) 31–44.
[12] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Lattices and Order* (Cambridge University Press, Cambridge, 1990).
[13] B.A. Davey and H.A. Priestley, *Introduction to Algorithms* (McGraw-Hill. New York. 1990).
[14] T. Ellman, Explanation-based learning: A survey of programs and perspectives, *ACM Comput. Surveys* 21 (1989) 162–222.
[15] G. Fay, An algorithm for finite Galois connexions, *J.* Comput. *Linguistic and Languages* 10 (1975) 99-123.
[16] W.J. Frawley, G. Piatetsky-Shapiro and C.J. Matheus, Knowledge discovery in databases: An overview, in: G. Piatetsky-Shapiro and W.J. Frawley, eds., *Knowledge Discovery from Databases* (AAAI Press/MIT Press, Menlo Park, CA, 1991) 1-27.
[17] B. Ganter, Two basic Algorithms in Concept Analysis. Preprint #831, Technische Hochschule Darmstadt, 1984.
[18] J.H. Gennari, P. Langley and D. Fisher, Models of incremental concept formation, in: J.G. Carbonell, ed., *Machine learning: Paradigms and methods,* (MIT Press, Cambridge, MA, 1990) 1 1–62.
[19] R. Godin, L'utilisation de treillis pour Paccès aux systèmes d'information, Ph.D. Thesis, Université de Montréal, 1986.
[20] R. Godin, E. Saunders and J. Gecsei, Lattice model of browsable data spaces, *Inform. Sci.* 40 (1986) 89-116.
[21] R. Godin, Complexité de structures de treillis, *Annales des* Sciences *Mathématiques du Québec,* **13**(1) (1989) 19–38.
[22] R. Godin, R. Missaoui and H. Alaoui, Incremental algorithms for updating the Galois lattice of a binary relation, Tech. Rep. # 155, Département de Mathématiques et d'Informatique, Université du Québec à Montréal, 1991.
1231 R. Godin, R. Missaoui, and H. Alaoui, Learning algorithms using a Galois lattice structure, in: *Proc. Third Int. Conf. on* **Tools** *for Artificial Intelligence,* San Jose, CA (1991) 22–29.
[24] R. Godin, R. Missaoui and A. April, Experimental comparison of Galois lattice browsing with conventional information retrievel methods, *Internat. J. Man-Machine Studies, 38* (1993) 747–767.
[25] R. Godin, G. Mineau and R. Missaoui, Rapport de la phase 2 du projet Macroscope pour le volet Réutilisation, 1993.
[26] J.L. Guigues and V. Duquenne, Familles Minimales d'Implications Informatives Résultant d'un Tableau de Données Binaries, *Mathématiques et Sciences Humaines 95* (1986) *5-1 8.*
1271 J. Hong and C. Mao, Incremental discovery of rules and structure by hierarchical and parallel clustering, in: G. Piatetsky-Shapiro and W. J. Frawley, eds., *Knowledge Discovery from Databases,* (AAAI Press/MIT Press, Menlo Park, CA, 1991) 177-194.
[28] Y.E. Ioannidis, T. Saulys and A.J. Whitsitt, Conceptual learning in database design, *ACM Trans. Information Systems* 10 (1992) 265-293.
[29] K.A. Kaufman, R.S. Michalski, and L. Kerschberg, Mining for knowledge in databases: goals and general description of the INLEN system, in: G. Piatesky-Shapiro and W.J. Frawley, eds., *Knowledge Discovery from Databases* (AAAI Press/MIT Press, Menlo Park, CA, 1991) 449462.
[30] K.J. Liecerherr, P. Bergstein and I. Silve-Lepe, Abstraction of object-oriented data models, in: Proc. of International Conf. on Entity-Relationship (Elsevier, Lausanne, 1990) 81-94.
[31] D. Maier, *The Theory* **of** *Relational Databases* (Computer Science Press, Rockville, MD, 1983).
[32] Y. Malgrange, Recherche des Sous-Matrices Premieres d'une Matrice à Coefficients Binaires; Applications à Certains Problèmes de Grpahes, in: *Proc. Deuxième Congrès de l'AFCALTI.* (Gauthier-Villars, Paris, 1962) 231–242.
[33] E. Mephy Nguifo, Concevoir une Abstraction à Partir de Ressemblances, Doctorat, Université Montpellier II Sciences et Techniques du Languedoc, 1993.
[34] R.S. Michalski, J. Carbonell and T. Mitchell, *Machine learning: an artificial intelligence Approach* (Tioga Palo Allo, CA, 1983).

[35] R.S. Michalski and Y. Kodratoff, Research in machine learning: Recent progress, classification of methods and future directions, in: Y. Kodratoff and R.S. Michalski, eds., ***Machine learning. An artificial intelligence approach*** (Morgan Kaufmann, San Mateo, CA, 1990) l-30.

[36] G. Mineau and R. Godin, Automatic structuring of knowledge bases by conceptual clustering, IEEE ***Trans. on Knowledge and Data Engineering,*** accepted for publication.

[37] J. Minker ed., ***Foundations of Deductive Databases and Logic Programming*** (Morgan Kaufmann, Los Altos, CA, 1988).

[38] R. Missaoui and R. Godin, An incremental concept formation approach for learning from databases, in: V.S. Alagar, VS. Lakshmanan and F. Sadri eds., ***Workshop on formal methods in databases and software engineering,*** Montreal, May 15-16, 1992 (Springer, London, 1993) 39-53.

[39] R. Missaoui and R. Godin, An expert system for discovering and using the semantics of databases, in: *Proc.* ***The World Congress on Expert Systems,*** Orlando, FL (Pergamon Press, Oxford, 1991) 1732–1740.

[40] A. Motro, Using integrity constraints to provide intensional answers to relational queries, in: *Proc.* ***Fifteenth International*** *Conf.* ***On Very Large Data Bases.*** Amsterdam (1989) 2317246.

[41] S. Muggleton and L. De Raedt, Inductive logic programming: Theory and methods, *J.* ***Logic Programming*** (1993).

[42] E.M. Norris, An algorithm for computing the maximal rectangles in a binary relation, ***Revue Roumaine Math.*** *Pures Appl.* ***23 (1978) 243-250.***

[43] ***G.*** Piatetsky-Shapiro and W.J. Frawley, Eds., ***Knowledge*** *discovery* ***in databases*** (AAAI Press/MIT Press, Menlo Park, CA, 1991).

[44] A. Silbershatz, M. Stonebraker and J.D. Ullman, Database systems: Achievements and comm. opportunities, ACM 34 (1991) 110-120.

[45] R. Wille, Restructuring lattice theory: an approach based on hierarchies of concepts, in: I. Rival, ed., ***Ordered sets*** (Reidel, Dordrecht, 1982) 4455470.

[46] R. Wille, Knowledge acquisition by methods of formal concept analysis, in: E. Diday, ed., ***Data analysis, learning symbolic and numeric knowledge*** (Nova Science, New York, 1989) 3655380.