Université de Montréal

Galois Lattice Dynamics

par

Alexi Ventzeslavov Popov

Departement d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des Études Supérieures

en vue de l'obtention du grade de Maîtrise ès Sciences

en Informatique

Janvier 2003

Mémoire accepté le 20 mai 2003

Université de Montréal
Faculté des Études Supérieures

Ce mémoire intitulé

Galois Lattice Dynamics

présenté par
Alexi Ventzeslavov Popov

a été évalué par un jury composé des personnes suivantes:

Jean-Yves Potvin
président rapporteur

Geňa Hahn
membre du jury

Patrice Marcotte
membre du jury

Sommaire

Les recherches théoriques du présent mémoire, sur les mutations aux treillis de Galois induites par des changements de contexte, sont une base pour la dynamique des treillis à fermeture en général. Les algorithmes en émanant pour incrémenter, décrémenter, construire graduellement et assembler un treillis de Galois ont la complexité minimale connue et utilisent des structures de données originales chaînées réciproquement. Ils sont tous applicables au domaine des treillis à fermeture en général, dont l'usage en informatique, sciences naturelles, et génie électrique est plus répandu.

Mots clés: treillis de Galois, algorithmique, structures de données

Abstract

Theoretical inquiry into the mutations of Galois lattices induced by changes in their context lays a comprehensive basis for studying the dynamics of lattices with known closure in general. The resulting algorithms for incremental and decremental update, gradual building and assembly of a Galois lattice have the minimal known complexity and are supported by innovative reciprocally interlinked data structures. All of them are also adaptable to the general field of lattices with known closure, which have a wider range of applications in computer science, electrical engineering and the physical sciences.

Keywords: Galois lattices, algorithms, data structures

Contents

List of Tables

List of Figures

Acknowledgements

This research and all of its results were possible exclusively through the sole support of the programme Smile and Be Happy! by my wife An Lili within the framework No Copyright, No Logo, No Opression! and ensuing ⊗ programmes.

Chapter 1

Introduction

The theoretical results leading to Theorem 3.1 of the present thesis, and the theorem itself, are a new ground for further study of lattice dynamics, whose significance lies not only in the abstract mathematical interest, but also in the fact that Galois lattices are widely used for a hierarchical representation of mostly dynamic information in computer science, natural and social sciences, which requires the structures to easily adapt with changes in said information. This is also true for the additional use in physical sciences and electrical engineering of other lattices with known closure, to which results are easily transferrable. The study is well supplemented by theory on precedence dynamics [18, 19]. Minimal complexity achieved recently [20] for building lattices with known closure applied exclusively to their static computation from a given basis. Inherent dynamic changes in the latter, however, cannot be translated into appropriate changes in the lattice through that approach without rebuilding anew. The algorithm introduced on Table 3.2 allows dynamic building, incremental update and assembly of any two lattices, while the one in Table 3.3 is the original introduction of decremental update, both algorithms performing all of these operations with the same minimal complexity, which is lower than that

achieved for similar operations until recently [16, 17, 18, 21, 22]. It is also worth noting that former work on algorithms for gradual building of Galois lattices, with a single known exception [17], considered a partially static context, since either the set of objects G or of attributes M was left invariant as in a very recent algorithm [19] of minimal complexity. Such approach is inappropriate for the truly dynamic nature of information underlying the practical need for the lattice structures. Finally, reciprocally interlinked data structures on Figure 4.2, developed here to insure minimal complexity for the algorithm in Table 3.3, allow updating a set of fully interdependent, interlinked and unordered lists with complexity linear in the size of a single one, rather than the usual quadratic polynomial. A short review of pertinent literature on lattice theory [1, 2, 3, 4, 5, 6, 7] and Galois lattices [12, 13, 14, 15, 16, 17, 18, 19] from formal concept analysis [8, 9, 10, 11] in the next chapter is followed by the theory on dynamics introduced by the present thesis with ensuing algorithmic results and a further chapter on algorithm implementation. Appendices on original proofs of existing theoretical results, implementation details and Java programme code conclude the thesis.

Chapter 2

Lattice Overview

Intuitive notions of context and concept are needed to enter the field of concept or Galois lattices. A classroom may be taken containing some professor, student, monkey, and book, each of them characterised as appropriate through living, human, teaching, listening and old. The concept of a living human in the classroom context is composed of all entities sharing the characteristics of living and human, which in this example are, intuitively, said professor and student, both assumed to be alive in a classroom, whether they look or not so. More precisely, a formal context may be viewed as a finite set of objects, a finite set of attributes, and a relation between the two. This is a simplified version of what is context in ordinary language, as data allowing for the proper interpretation of words. A formal concept is the subset of all objects sharing a particular subset of attributes, and similarly for a subset of the latter. Again, this is a simplified version of what is often vaguely called a concept.

Since lattice theory is based on the notions of ordered sets and order, a formal beginning with the latter is appropriate. For a set S, an **order** or **partial** order is a binary relation $\leq$, which is **reflexive** as $x \leq x$, **antisymmetric** as $x \leq y$ and $y \leq x$ imply $x = y$, and **transitive** as $x \leq y$ and $y \leq z$ imply $x \leq z$ for

		living	human	teaching	listening	old
		v	w	x	y	z
monkey	1	+		+		
student	2	+	+			
professor	3	+	+		+	+
book	4					+

Table 2.1: In the common classroom context, the objects are assigned numbers, the attributes letters, and the incidence is shown by the + signs in place.

all $x, y, z \in S$. A set S with a partial order on it is an **ordered set**. It is easy to notice that **inclusion** $\subseteq$ on the subsets of a given set S is an order on the **power set** $\mathcal{P}(S)$ of the latter. The binary relation $\geq$, defined as $x \geq y$ if and only if $y \leq x$ is also an order. The relations are **dual** and by the **duality principle** any statement applicable to either is, mutatis mutandis, applicable to the other. Further, $x < y$ and dually $y > x$ if and only if $x \leq y$ with $x \neq y$.

For $R \subseteq S$, an **infimum**, if it exists, $\bigwedge R \in S$ of the subset R is a **lower bound** of the subset, as $\bigwedge R \leq r$ for all $r \in R$, and it is the **greatest** lower bound, with **supremum** $\bigvee R \in S$ being the **least upper** bound. As an example, for the **family** or set of sets $\{Q, T\}$, the subset $Q \cap T = Q \wedge T$ is the infimum, while $Q \vee T = Q \cup T$ is the supremum, with inclusion order on $\mathcal{P}(Q \cup T)$. If $R \subseteq S$ is such that, $x \leq r$ implies $x \in R$ for every $r \in R$ and all $x \in S$, then R is an **order ideal** in S, with **order filter** dually defined for $x \geq r$. The subsets

$$y_{\downarrow} = \{x \in S \mid x \leq y\} \text{ and } y^{\uparrow} = \{x \in S \mid x \geq y\}$$

are order ideal and order filter with respective supremum and infimum y. The binary relation **precedence** $\prec$ is such that $x \prec y$ if and only if $x < y$ and there is no z with $x < z < y$, where x is a **predecessor** of y and y is an **successor** of x. As an example, the empty set is a predecessor of any singleton subset of a set S, with inclusion order on $\mathcal{P}(S)$, and any of the latter subsets is a successor

to the former. The binary relation $\succ$ is dually defined. A **formal context** K is a triple (G, M, I) with a set of **objects** G, a set of **attributes** M, and a binary relation their **incidence** $I \subseteq G \times M$. In the example on Table 2.1, the professor may be characterised as living, human, listening and old, the student only as living and human, properly, without mention of listening, the monkey as living and, indeed, teaching with, finally, the book as old only. The classroom thus becomes a formal context, with the set of characteristics as M, while said professor, student, monkey and book together are G. All is given on Table 2.1.

The right-side **incidence operator** provides the subset of all objects Y' sharing a given subset of attributes Y and the subset of all attributes X' shared by a given subset of objects X or

$$X' = \{a \in M \mid (o, a) \in I \textit{ for all } o \in X\},$$

$$Y' = \{o \in G \mid (o, a) \in I \text{ for all } a \in Y\}.$$

For simplicity, it is preferable to use v or 123 instead of $\{v\}$ or $\{1, 2, 3\}$, and $\{1\}'$ or $\{v, w, x\}'$ become $1'$ or vwx'. Thus $1' = vx$ and $vwy' = 3$ from the example in Table 2.1, or more clearly, the monkey 1 is characterised as living and teaching vx, while the one characterised, among other, as living, human and listening vwy is the professor 3. Further, $X \subseteq X''$ and $X' = X'''$ for any subset of objects X or of attributes Y, as shown by Lemma 2.1.

Lemma 2.1 *If K is a context with $X_i, X_j \in \mathcal{P}(G)$, then $X_i \subseteq X_i''$ and $X_i' = X_i'''$, while for all $n \leq |\mathcal{P}(G)|$,*

$$(\bigcup_{j=1}^{n} X_j)' = \bigcap_{j=1}^{n} X_j',$$

and $X_i \subseteq X_j$ implies $X_i' \supseteq X_j'$, with the same for $Y_i, Y_j \in \mathcal{P}(G)$, plus $\bigvee \mathfrak{B}_K = (G, G')$ and $\bigwedge \mathfrak{B}_K = (M', M)$ for the Galois or concept lattice $\mathfrak{B}_K$ from K.

Proof: *easily available in literature on concept lattices [8, 9, 10].*

A **formal concept** of a context K is a pair (X, Y) with **extent** $X \subseteq G$ such that $X = Y'$ and **intent** $Y \subseteq M$ such that $Y = X'$. Otherwise said,

it is an ordered pair (X'', X') or (Y', Y''). In the preceding example, $1' = vx$ and $vx' = 1$ gives the concept $(1, vx)$, while $vwy' = 3$ but $3' = vwyz$ and the proper concept is $(3, vwyz)$, not $(3, vwy)$. In fact, $(\emptyset, vwxyz)$, $(1, vx)$, $(3, vwyz)$, $(23, vw)$, $(34, z)$, $(123, v)$, $(1234, \emptyset)$ are all the concepts in the example from Table 2.1. It is also proper to underline that, for every attribute $a \in M$, there is a maximum concept (a', a'') containing a and, for every object $o \in G$, there is a minimum concept (o'', o') containing it. These are known as the **attribute concept** of a and the **object concept** of o. Continuing with Table 2.1, the object concept for the student 2 is $(23, vw)$, which is, accidentally, also the attribute concept for human w. The composed incidence operator is a **closure operator** for the sets of extents or intents as (X'', X') is a concept for every $X \subseteq G$ or dually for every $Y \subseteq M$.

A **complete lattice** is an ordered set S such that, for any subset $R \subseteq S$, $\bigwedge R \in S$ and $\bigvee R \in S$ exist. The duality principle for the relations $\leq$ and $\geq$ of

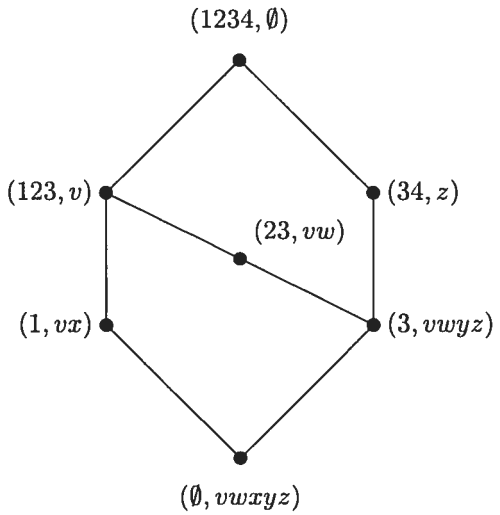

Figure 2.1: In this representation, known as Hasse diagram, of the lattice from the context in Table 2.1, concepts are points, of which, as an example, the middle one has attributes living and human vw shared by objects professor and student 23, with lines connecting predecessors to successors above them.

course applies. For a context K, a **concept lattice** is the ordered family $\mathfrak{B}_K$ of all concepts from K with a partial order $\leq$, by which concepts $(X_i, Y_i) \leq (X_j, Y_j)$ if and only if $X_i \subseteq X_j$ with $Y_j \subseteq Y_i$. The lattice corresponding to the context of Table 2.1 is given on Figure 2.1. The following theorem is considered fundamental [2, 8, 9, 10] for concept lattices and formal concept analysis.

Theorem 2.2 *The lattice $\mathfrak{B}_K$ is a complete lattice, in which infimum and supremum for any subfamily of concepts $R = \{(X_1, Y_1) \ldots (X_r, Y_r)\}$ are*

$$\bigwedge R = (\bigcap_{i=1}^{r} X_i, (\bigcup_{i=1}^{r} Y_i)'') \text{ and } \bigvee R = ((\bigcup_{i=1}^{r} X_i)'', \bigcap_{i=1}^{r} Y_i).$$

Proof: *easily available in literature on concept lattices [2, 8, 9, 10].*

Concept lattices are traditionally called **Galois lattices** [3] and, for simplicity, all further lattices here are Galois unless explicitly stated otherwise. For an attribute a, the context $(a', a, I \cap (a' \times a))$ gives rise to a **singleton lattice** where infimum and supremum coincide, and dually for an object o with o'. Further, a **basis** $\mathcal{B}(M)$, sometimes called **base** [26], from which the closed family may be generated, contains all pairs a, a' with $a \in M$ providing the extents of all attribute concepts, with the same for all objects and $\mathcal{B}(G)$. This definition is an adaptation to concept lattices of some more general definition for basis of a lattice [25] or a family [26]. The columns of Table 2.1 can be seen as an attribute basis and the rows as an object one for the lattice of Figure 2.1. An interesting property is that any $\mathfrak{B}_K$ contains in fact two complete lattices, one composed of extents with supremum G and another of intents with supremum M. Both are intersection closed with respective inclusion order on the powerset of G and dual order on the powerset of M. Intersection of intents is accompanied by union of extents and vice versa under the conditions of the theorem above.

The following theorem for computing predecessors [20] is an important result for reducing the complexity of algorithms for building lattices. More precisely, with $\mathbf{g} = |G|$, $\mathbf{m} = |M|$, $\mathfrak{b} = |\mathfrak{B}_K|$, the above allows for a procedure [20] on Table 2.2 computing the predecessors of a given concept with complexity

algorithm **computeLattice**()

1. **for all** $a \in \mathcal{B}(M)$ **do**
2. **for all** $(X_i, Y_i) \in \mathfrak{B}_K$ **do**
3. **if** $(a' \cap X_i, (a' \cap X_i)') \notin \mathfrak{B}_K$ **then** $\mathfrak{B}_K \cup= (a' \cap X_i, a \cup Y_i)$
4. **else** $(a' \cap X_i)' \cup= (a \cup Y_i)$
5. $G \cup= a'$ **and if** $(a', a'') \notin \mathfrak{B}_K$ **then** $\mathfrak{B}_K \cup= (a', a)$
6. 1ex] computePredecessors $(\mathfrak{B}_K)$

procedure **computePredecessors** (linkableConcepts)

1. **for all** $(X_i, Y_i) \in$ linkableConcepts **do**
2. **for all** $a \in M \setminus Y_i$ **do**
3. $(a' \cap X_i, (a' \cap X_i)')$'s count++
4. (X_i, Y_i)'s resettableCounts $\cup= (a' \cap X_i, (a' \cap X_i)')$
5. **if** $|(a' \cap X_i)'| = |Y_i| + (a' \cap X_i, (a' \cap X_i)')$'s count **then**
6. **do** $(a' \cap X_i, (a' \cap X_i)') \prec (X_i, Y_i)$
7. **for all** $(X_j, Y_j) \in (X_i, Y_i)$'s resettableCounts **do** (X_j, Y_j)'s count $= 0$

Table 2.2: The **computeLattice** () algorithm [20] computes the family $\mathfrak{B}_K$ from data in the basis and predecessors by Theorem 2.3, while $G \cup= a'$ is for $G = G \cup \{a'\}$. Notice that setting precedence on line 6 of the procedure is simple addition of pointers at the beginning of linked lists, while concepts are retrieved by their extents or intents from tries on lines 3 and 4 of the algorithm, which is performed in $\mathcal{O}(\mathbf{m} + \mathbf{g})$ and similarly on line 3 of the procedure. This gives complexity for the algorithm in $\mathcal{O}(\mathfrak{b}\mathbf{m}(\mathbf{g} + \mathbf{m}))$, or $\mathcal{O}(\mathfrak{b}\mathbf{mg})$ if $\mathbf{m} < \mathbf{g}$.

$\mathcal{O}(\mathbf{m}(\mathbf{g} + \mathbf{m}))$, giving $\mathcal{O}(\mathfrak{b}\mathbf{m}(\mathbf{g} + \mathbf{m}))$ for all concepts in the lattice. The complexity is $\mathcal{O}(\mathfrak{b}\mathbf{mg})$ if $\mathbf{m} < \mathbf{g}$ and this is the minimal known one for computing the predecessors in a lattice and for building the lattice from the basis or context data [20].

Theorem 2.3 *If $(X_i, Y_i), (X_j, Y_j)$ are concepts of lattice $\mathfrak{B}_K$, then the first precedes the second if and only if the former is pair infimum $(X_i, Y_i) = (X_j, Y_j) \wedge (a', a'')$ for all a in the difference of their intents, or*

$$(X_i, Y_i) \prec (X_j, Y_j) \text{ if and only if } X_j \cap a' = X_i \text{ for all } a \in Y_i \setminus Y_j,$$

and dually for all objects $o \in X_j \setminus X_i$.

Proof: *available, mutatis mutandis, in the original publication [20].*

As an example of its application on the lattice from Figure 2.1, let the first concept be $(3, vwyz)$ and the second $(34, z)$. The difference in their intents is

vwy and $v' = 123$, $w' = 23$, $y' = 3$ from Table 2.1. The intersection of v', w' and y' with 34 all give 3, which is the intent of $(3, vwyz)$ and it precedes $(34, z)$. If $(123, v)$ had been used instead of the latter concept, the difference would have been wyz and, for $w' = 23$, the intersection with 123 is not 3 but 23, whence not all attributes in the difference of intents give the same result, and $(3, vwyz)$ does not precede $(123, v)$. In what follows $M \setminus a$ is simplified form for $M \setminus \{a\}$ and other similarly so, with

$$K \setminus a = (G \setminus 'a, M \setminus a, I \cap ((G \setminus 'a) \times (M \setminus a))),$$

and dually for **subcontext** $K \setminus o$. Follows a theorem [19], which is here proven, on the dynamics of precedence or on partially inferring the precedence relation in the **superlattice** $\mathfrak{B}_K$ [19] from that in the **sublattice** $\mathfrak{B}_{K \setminus a}$ [18, 19].

Theorem 2.4 *If* $(X_i, Y_i) \prec (X_j, Y_j)$ *in* $\mathfrak{B}_{K \setminus a}$, *then* $(X_i, X_i') \prec (X_j'', X_j')$ *in* $\mathfrak{B}_K$, *except that, if* $(X_j \cap a', Y_j \cup a) \prec (X_j, Y_j)$ *in the superlattice and* $X_j \cap a' \neq X_i$, *then* $(X_i'', X_i') \not\prec (X_j'', X_j')$, *and dually with* $o \in G$.

Proof: *given as Proof A.3 in appendix.*

With $a_\downarrow$ being $(a', a'')_\downarrow$ and so for $o^\uparrow$, below is a lemma [16], which is here proven and is necessary for the proof of Theorem 2.4 as seen in appendix, about the successors from $\mathfrak{B}_K \setminus a_\downarrow$ of concepts in $a_\downarrow$. Application examples for said theorem and the lemma are to be seen on Figure 3.2 in next chapter upon introducing general theory on the dynamics of Galois lattices beyond the precedence evolution.

Lemma 2.5 *Let* $(X_i, Y_i) = \bigwedge R$ *for* $R = \{(X, Y) \in \mathfrak{B}_K \setminus a_\downarrow \mid X \cap a' = X_i \cap a'\}$ *and there is no* $(X_m, Y_m) \in \mathfrak{B}_{K \setminus a}$ *with* $X_i \cap a' = X_m$, *then* $(X_i \cap a', Y_i \cup a) \prec \bigwedge R$ *and no other concept in* $a_\downarrow$ *precedes any in* R, *plus dually for an object* o, $o^\uparrow$, *and the supremum.*

Proof: *given as Proof A.2 in appendix.*

Chapter 3

Lattice Dynamics

Original results of the thesis start with Theorem 3.1 giving the dynamic evolution of the family $\mathfrak{B}$ from $K \setminus a$ to K. Here the **isolation** operator provides the subset of all objects $'Y$ or all attributes $'X$ with respective incidence Y and X,

$$'X = \{a \in M \mid (o,a) \in I \textit{ if and only if } o \in X\},$$

$$'Y = \{o \in G \mid (o,a) \in I \textit{ if and only if } a \in Y\}.$$

Notation is simplified as for the incidence operator and, in Table 2.1, only the monkey $'1 = x$ is characterised as teaching, while the one characterised exclusively as living and human $'vw = 2$ is the student.

Theorem 3.1 *If* $(X,Y) \in \mathfrak{B}_{K\setminus a}$, *then* $(X'' \cap a', (X'' \cap a')') \in a_\downarrow$ *and, if* $X'' \neq X'' \cap a'$, *then* $(X'', X') \in \mathfrak{B}_K \setminus a_\downarrow$ *and* $(X'', X') = (X'', Y)$, *else* $(X'', X') = (X'', Y \cup a)$, *where*

$$a_\downarrow = \{(X'' \cap a', (X'' \cap a')') \in \mathfrak{B}_K \mid (X,Y) \in \mathfrak{B}_{K\setminus a} \textit{ or } X = G\}$$

with $\mathfrak{B}_K \setminus a_\downarrow = \{(X'', Y) \in \mathfrak{B}_K \mid (X,Y) \in \mathfrak{B}_{K\setminus a} \textit{ or } Y = \emptyset\}$ *and, if* $(X,Y) = (G \setminus 'a, \emptyset)$ *with* $'a \neq \emptyset$, *then* $(G \setminus 'a)'' = G$ *in* $\mathfrak{B}_K$, *while in all other cases,* $X'' = X$ *in the superlattice, and dually for* $o^\uparrow$.

Proof: *given as Proof A.1 in appendix.*

Otherwise said, all extents of concepts in the sublattice remain unchanged as extents of concepts in the superlattice, except for that of the sublattice

		living	human	teaching	listening			
		v	w	x	y			
monkey	1	+		+		+		
student	2	+	+			+		
professor	3	+	+		+	+	4	book
						z		
						old		

Table 3.1: The initial classroom context, before the addition of the attribute z with proper incidence to the subcontext $(123, vwxy, I \cap (123 \times vwxy))$ or $K \setminus z$, is missing not only z itself but also object 4 or $'z$.

supremum if it is $(G \setminus 'a, \emptyset)$ and if $'a \neq \emptyset$. The theorem further states that $(X_i, Y_i) \in \mathfrak{B}_K$ is in $a_\downarrow$ if and only if there is $(X_j, Y_j) \in \mathfrak{B}_{K \setminus a}$ or $(X_j, Y_j) = \bigvee \mathfrak{B}_K$ with the former being infimum of the pair $(X_i, Y_i) = (X_j'', X_j') \wedge (a', a'')$ and, if $(X_i, Y_i) = (X_j'', X_j')$ in the superlattice, then $Y_i = Y_j \cup a$. It is to consider X_j'' here within the statement on extents of concepts in the sublattice. Similarly, the family $\mathfrak{B}_K \setminus a_\downarrow$ contains all concepts from the sublattice whose extent is not included in a', plus $(G, \emptyset)$ if present in $\mathfrak{B}_K$. The theorem allows to obtain $a_\downarrow$ and $\mathfrak{B}_K \setminus a_\downarrow$, whence the whole **superfamily**, from the **subfamily** $\mathfrak{B}_{K \setminus a}$.

As an example in Table 3.1 representing $K \setminus z$ for K in Table 2.1, any subset of attributes $vwxy$ shares only a subset of objects 123 because $4 = 'z$ exclusively and z is still to be added. The concepts in $\mathfrak{B}_{K \setminus z}$ are $(\emptyset, vwxy)$, $(3, vwy)$, $(1, vx)$, $(23, vw)$ and $(123, v)$. Since $z' = 34$ and the extent of $(\emptyset, vwxy)$ and $(3, vwy)$ is included in z', then these two concepts with z added to intents, plus (z', z) form $z_\downarrow$ in $\mathfrak{B}_K$. In this case, all extents of concepts are unchanged in the superlattice as $G \setminus 'z \neq \emptyset$ and the special case mentioned in Theorem 3.1 does not apply. For concepts from the sublattice whose intent is added by z to form a superlattice concept, the graphical representation on Figure 3.1 remains the same with the

evolution of the lattice, beyond changes in the intent listing. As an example, the sublattice concept of living human and listening, otherwise said the professor $(3, vwy) \in \mathfrak{B}_{K\setminus z}$, in the superlattice is replaced by $(3, vwyz) \in \mathfrak{B}_K$, which is still the same professor although in a different context, whence the representation $(3, vwy{+}z)$ on Figure 3.1 with $+$ indicating the evolution from sublattice to superlattice, equally observable for the infimum $(\emptyset, vwxy{+}z)$. The concepts concerned by the matter are, in general, all those from $\mathfrak{B}_{K\setminus a}$ whose extent is included in a', but the same is also to be considered for the particular case of $(G\setminus {}'a, \emptyset)$ when ${}'a \neq \emptyset$. Of course, the graphical representation of concepts from the sublattice, which also belong to the superlattice, remains the same.

Before introducing an algorithm based on the last theorem, it is proper to mention that gradual or dynamic lattice building [16] starts from a null context and concept family. The first step is inevitably adding a with a' to the

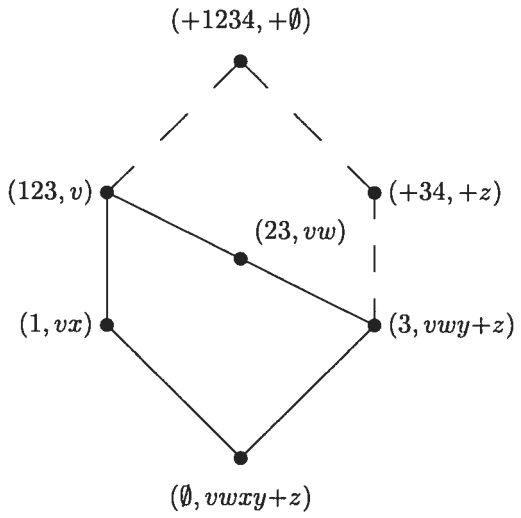

Figure 3.1: This is the graphical representation with solid lines of the lattice for the subcontext from Table 3.1, and upon the attribute z with proper incidence being added to it, the resulting portion of the superlattice is represented with dashed lines, while elements after the $+$ signs are added by the evolution from sublattice to superlattice as for $(3, vwy) \in \mathfrak{B}_{K\setminus a}$ and $(3, vwyz) \in \mathfrak{B}_K$ represented by $(3, vwy{+}z)$.

null context and obtaining the singleton lattice with $K \setminus a = \emptyset$. The process continues, and an augmented context with an additional attribute plus proper incidence results from each further step, together with a new superlattice. In this dynamic process, it must be kept in mind that each superlattice so obtained, in the next step, if any, is the sublattice $\mathfrak{B}_{K\setminus a}$ for the next attribute a being

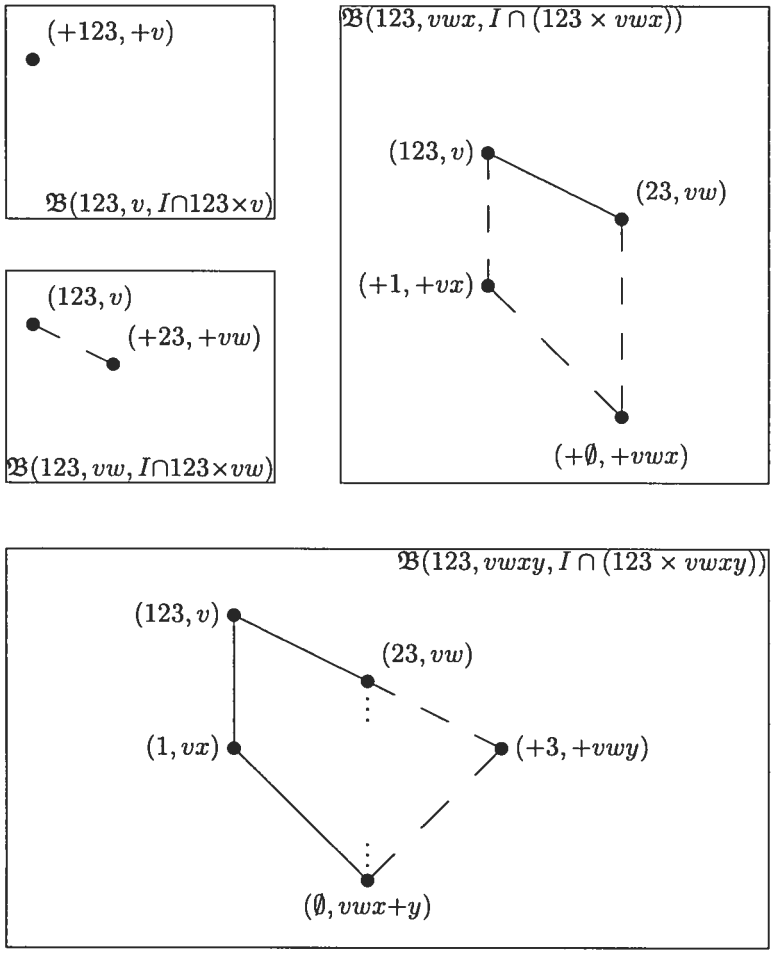

Figure 3.2: Illustrating lattice building for the context in Table 2.1, the + signs indicate evolution from sublattice to superlattice as in Figure 3.1, the loss of a predecessor by Theorem 2.4 is shown with dots, while $(+3, +vwy) \prec (23, vw)$ instead of preceding $(123, v)$ is given by Lemma 2.5, all finalised on Figure 3.1.

algorithm **updateLattice** (moreAttributes)

1. **for all** $a \in$ moreAttributes **do**
2. **if** $'a \neq \emptyset$ **then if** $(G \setminus 'a)' = \emptyset$ **then** $G \setminus 'a \cup\!= 'a$ **else**
3. $\mathfrak{B}_K \cup\!= (G, G')$ **and do** $(G \setminus 'a, (G \setminus 'a)') \prec (G, G')$
4. updateConcepts (a, (G, G'))
5. computePredecessors ($\{(X, Y) \in a_{\downarrow} \mid (X, X') \notin \mathfrak{B}_{K \setminus a}\}$)

procedure **updateConcepts** (a, (X, Y))

1. $\mathfrak{B}_K \cup\!= (X, Y)$ **and for all** $(X_j, Y_j) \prec (X, Y)$ **do**
2. **if** $(X_j, Y_j) \notin \mathfrak{B}_K$ **then** updateConcepts (a, (X_j, Y_j))
3. **if** $(a' \cap X, (a' \cap X)') \notin \mathfrak{B}_K$ **then**
4. $\mathfrak{B}_K \cup\!= (a' \cap X, a \cup Y)$ **and do** $(a' \cap X, a \cup Y) \prec (X, Y)$
5. **for all** $(X_j, Y_j) \prec (X, Y)$ **do if** $a \in Y_j$ **then do** $(X_j, Y_j) \not\prec (X, Y)$
6. **else if** $a' \cap X = X$ **then** $Y \cup\!= a$

Table 3.2: The **updateLattice** (...) algorithm updates a lattice by gradually adding a set of attributes to its context and computing, for each a of them the superlattice $\mathfrak{B}_K$ from $\mathfrak{B}_{K \setminus a}$ as per Theorem 3.1 with predecessors by Theorem 2.4 and Theorem 2.3. Notice that extent, intent and trie operations on lines 3 to 5 or on line 6 of the procedure above have complexity in $mathcal{O}(\mathbf{g} + \mathbf{m})$. The complexity of the procedure on line 5 of the algorithm is in $mathcal{O}\mathbf{m}(\mathbf{g} + \mathbf{m})$ per concept and, given the definition of its arguments, it is executed exactly once on no more than the number of concepts in the final superlattice. This gives complexity for the algorithm in $\mathcal{O}(\mathfrak{b}\mathbf{m}(\mathbf{g} + \mathbf{m}))$, or $\mathcal{O}(\mathfrak{b}\mathbf{m}\mathbf{g})$ if $\mathbf{m} < \mathbf{g}$ equalising the minimal known one, with $Y \cup\!= a$ for $Y = Y \cup \{a\}$ and other similarly.

added. Taking Table 2.1, lattice building can be shown by adding attribute v, of course with v', to the null context for $K \setminus wxyz = (123, v, I \cap (123 \times v))$. Further steps from successfully adding w, x, y are on Figure 3.2, with $K \setminus z$ on Table 3.1 and Figure 3.1 in the next chapter. The reasoning is dually applicable to objects. Gradual lattice building with a partially static context, where either G or M is invariant has been more widely studied [16, 17, 18, 19], including a theory on precedence dynamics [18, 19].

The last two theorems introduce on Table 3.2 an algorithm for incremental lattice update upon adding attributes to the subcontext, or dynamic lattice building, with complexity in $\mathcal{O}(\mathfrak{b}\mathbf{m}(\mathbf{g} + \mathbf{m}))$ giving $\mathcal{O}(\mathfrak{b}\mathbf{g}\mathbf{m})$ with $\mathbf{m} < \mathbf{g}$, which is the least known complexity for the purpose and where $\mathfrak{b} = |\mathfrak{B}_K|$ for the final

algorithm **reduceLattice** (someAttributes)

1. extractBasis $(\bigwedge \mathfrak{B}_K)$
2. **for all** $a \in \mathcal{B}(M) \cap$ someAttributes **do**
3. **for all** $(X,Y) \in \mathfrak{B}_K$ **do** $\mathfrak{B}_{K\setminus a} \cup= (X,Y)$ **and if** $a \in Y$ **then**
4. **if** $((Y \setminus a)', Y \setminus a) \notin \mathfrak{B}_K$ **then** $Y \setminus= a$ **else**
5. $\mathfrak{B}_{K\setminus a} \setminus= (X,Y)$ **and for all** $(X_j,Y_j) \prec (X,Y)$ **do** $(X_j,Y_j) \not\prec (X,Y)$
6. **for all** $(X_k,Y_k) \prec ((Y \setminus a)', Y \setminus a)$ **do** $(X_k,Y_k) \not\prec ((Y \setminus a)', Y \setminus a)$
7. **if** $(X,Y) = \bigvee \mathfrak{B}_K$ **and** $'a \neq \emptyset$ **then**
8. **if** $(G \setminus 'a, (G \setminus 'a)') \notin \mathfrak{B}_{K\setminus a}$ **then** $X \setminus= 'a$ **else** $\mathfrak{B}_{K\setminus a} \setminus= (X,Y)$
9. computePredecessors $(\{((Y \setminus a)', Y \setminus a) \in \mathfrak{B}_K \mid (X,Y) \in a_\downarrow \setminus \mathfrak{B}_K\})$

procedure **extractBasis** $((X,Y))$

1. visitedConcepts $\cup= (X,Y)$ **and for all** $(X_j,Y_j) \succ (X,Y)$ **do**
2. **if** $(X_j,Y_j) \notin$ visitedConcepts **then** extractBasis $((X_j,Y_j))$
3. **for all** $a \in (M \setminus \mathcal{B}(M)) \cap Y$ **do** $\mathcal{B}(M) \cup= a$

Table 3.3: The **reduceLattice** (...) algorithm updates a lattice by gradually removing a set of attributes from its context and computing, for each a of them the sublattice $\mathfrak{B}_{K\setminus a}$ from $\mathfrak{B}_K$ as per Theorem 3.1 with predecessors by Theorem 2.4 and Theorem 2.3. The complexity of the procedure above is in $\mathcal{O}(\mathfrak{bm})$, insignificant to that of the algorithm. Notice that extent, intent and trie operations on lines 4 to 6 have complexity in $\mathcal{O}(\mathbf{g}+\mathbf{m})$. The complexity of the procedure on line 5 of the algorithm is in $\mathcal{O}(\mathbf{m}(\mathbf{g}+\mathbf{m}))$ per concept and, given the definition of its arguments, it is executed exactly once on no more than the number of concepts in the initial superlattice. This gives complexity for the algorithm in $\mathcal{O}(\mathfrak{bm}(\mathbf{g}+\mathbf{m}))$, or $\mathcal{O}(\mathfrak{bmg})$ if $\mathbf{m} < \mathbf{g}$ equalising the minimal known one, with $\mathfrak{B}_{K\setminus a} \setminus= (X,Y)$ for $\mathfrak{B}_{K\setminus a} = \mathfrak{B}_{K\setminus a} \setminus \{(X,Y)\}$ and other similarly. Finally, notice that only the algorithms in Table 2.2 and Table 3.2 are implemented here, not the present one.

superlattice computed upon adding all attributes. The Java implementation of these algorithms is expalined in next chapter. Reciprocally interlinked data structures are introduced and used there, with an expansion of the implementation in mind, to insure the minimal known complexity above for an algorithm on decremental update as the one on Table 3.3.

If two lattices with disjoint bases have to be assembled, the first can be updated with the basis of the second and complexity will be as for the updating algorithm on said Table 3.2. If necessary for the purpose, a basis can be

extracted from a lattice through the procedure in Table 3.3, whose complexity in $\mathcal{O}(\mathfrak{bm})$ or $\mathcal{O}(\mathfrak{bg})$ is insignificant to the algorithm for update and assembly. In fact, since every concept except the supremum is part of the order ideal $a_{\downarrow}$ for at least one $a \in M$, it is easy to reduce the problem of assembling two sublattices with disjoint $\mathcal{B}(M)$ to the operations, as in Theorem 3.1, involving one of them and the family of attribute concepts (a', a'') of the second lattice, which are suprema of the order ideals in the latter. This is, however, equivalent to the approach of update and gradual building of Table 3.2 using a' from the basis, and dually for o' and $\mathcal{B}(G)$. Finally, Lemma 3.2 below may allow for a more practical version of the algorithm on Table 3.2 by reducing the size of intersection computations, although without immediate effect on worst case complexity.

Lemma 3.2 *If* $(X_i, Y_i), (X_j, Y_j) \in \mathfrak{B}_{K\setminus a}$, $(X_j, Y_j) \leq (X_i, Y_i)$, *then the set* $X_i \cap a' = X_i \cap (X_j \cap a')$ *since* $X_i \subseteq X_j$ *and similarly in* $\mathfrak{B}_{K\setminus o}$.

Proof: *given as Proof A.4 in appendix.*

A second part of Theorem 2.2, which is not provided here but is easily available in literature on concept lattices [2, 8, 9, 10], deals with isomorphism between them and complete lattices in general. It allows, thus, generalising to the latter, which have a wider array of applications in natural sciences, electrical engineering and computer science, results obtained here for the former.

Chapter 4

Implementation and Tests

The algorithms for static building on Table 2.2 and for dynamic or gradual building on Table 3.2 are implemented using the lattice and concept structures on Figure 4.2, basis on Figure 4.3, plus trie and set auxiliaries on Figure 4.1,

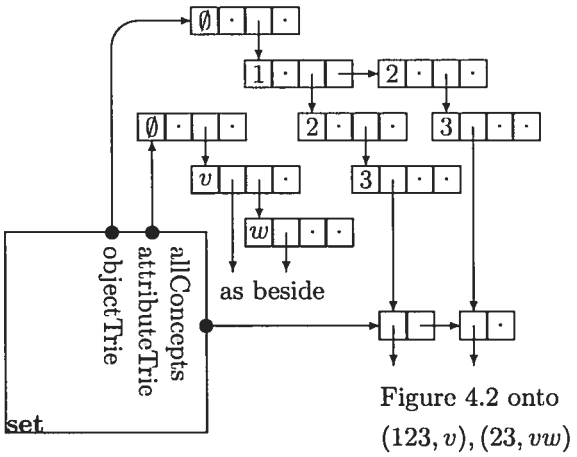

Figure 4.2 onto
$(123, v), (23, vw)$

Figure 4.1: The set structure comprises pointers to the object and attribute tries, and to a list of pointers to all concepts, while everything is represented at the point upon computing $\mathfrak{B}_{K \setminus xyz}$ when the list linkableConcepts is reset to null and omitted here to improve readability, and the pointers from attributeTrie lead to the elements of allConcepts exactly as those from objectTrie.

preserving names from algorithms. Summary of execution results is also shown.

The implementation of the algorithms presented thus far is based on a lattice structure seen on Figure 4.2, which is composed of pointers to the supremum and infimum concepts, the attribute and object basis and to a set structure. The latter is a seen on Figure 4.1 with pointers to attribute and object tries, to a list of pointers towards all concepts and one of pointers towards those pending predecessor computation. The tries are linked lists shown on the latter figure and allow the retrieval of concepts by their intent or extent, in keeping pointers indirectly through the list of all concepts to the latter. Said list is very desirable for processing with linear complexity in the size of the family. A concept is a structure shown on Figure 4.2. It comprises Boolean variables to mark the concept from the sublattice as visited upon traversal and modified upon adding

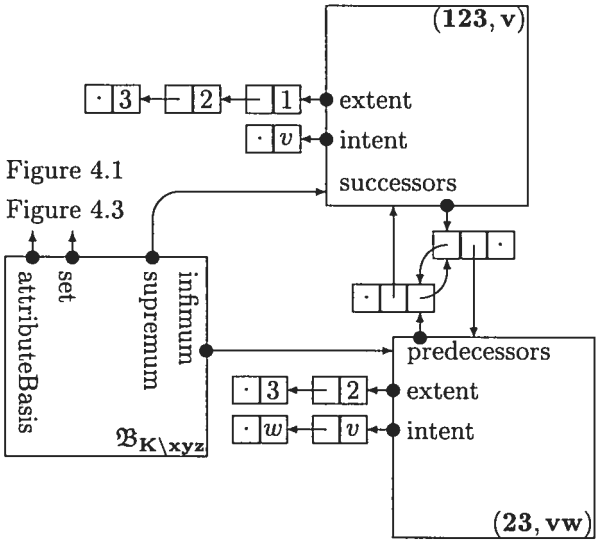

Figure 4.2: The lattice data structure upon computing $\mathfrak{B}_{K\setminus xyz}$ of K in Table 2.1 with pointers to infimum and supremum concepts, linked lists of intent, extent, predecessors and successors,, while elements, which are null or uncomputed at that step as the objectBasis plus the successors and predecessors of supremum and infimum respectively or the Boolean and numerical variables in the concepts, are omitted to improve readability.

a to its intent or dually o, numerical variables for extent and intent length, pointers to extent and intent lists, plus pointers to the reciprocally interlinked lists for predecessors and successors. The latter are pointers to concepts, such that the one to a predecessor of a concept (X, Y) will also point to an element in the successor list of said predecessor. Exactly this very element reciprocally points to (X, Y) and to the pointer mentioned in the predecessor list.

The structure for basis is presented on Figure 4.3. It is proper to consider its form in a dynamic perspective. When the lattice building from the basis begins, the latter is a linked list of attributes, where each element additionally points to a simple list of objects or a' for its attribute a. For all a added

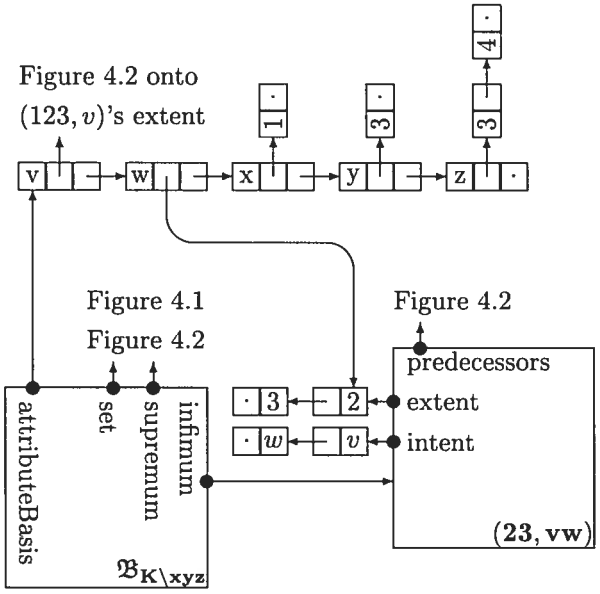

Figure 4.3: While building $\mathfrak{B}_K$ from attributeBasis, the incidence a' of each attribute a corresponds to the extent of its attribute concept (a', a''), and the basis structure contains an extent list for every attribute not yet added or a pointer to the extent of the attribute concept otherwise, as seen for v and w upon computing $\mathfrak{B}_{K\setminus xyz}$ and dually for the objectBasis, which here is yet to be extracted from the final lattice, and is omitted mention for readability.

density 0.25	5	10	15	20	25
computeLattice	0.847	1.033	1.335	2.906	6.063
updateLattice	0.847	0.990	1.306	2.466	4.634

density 0.50	5	10	15	20	25
computeLattice	0.847	1.437	5.810	48.503	304.970
updateLattice	0.861	1.345	3.933	22.179	104.013

density 0.75	5	10	15	20	25
computeLattice	0.873	1.449	17.082	1761.218	-
updateLattice	0.849	1.363	10.267	951.166	-

Table 4.1: Summary of execution results in seconds for the algorithms on Table 2.2 and Table 3.2, on the machine gle21.iro.umontreal.ca with 400 MHz processor plus 128 MB system memory operating on Linux kernel 2.4.18-27.7, shows that the latter one clearly outperforms the former. Columns above indicate the number of attributes and objects in each randomly generated test context, while density of the context is used here in reference to the ratio between size of the average incidence o' or a' and size of G or M respectively. Execution results were not taken for density 0.75 on context with 25 objects and attributes because the lattice grows exponentially with G and M at high density, which leads to large execution times.

to the context, a' becomes the extent of some concept. The pointer form the attribute basis still points to it, but there is now an additional pointer from the concept (a', a''), which is well visible on the figure for w and $w' = 23$, being also the same for v and 123. The latter are the only attributes in the context $K \setminus xyz$ represented, and all other elements of the attribute basis retain their original version, pending processing in contexts $K \setminus yz, K \setminus z, K$. Of course, all is dually applicable when building or updating the lattice with objects. Finally, performance tests comparing the algorithms on Table 2.2 and Table 3.2 are summarised on Table 4.1. They are discussed with more detail in appendix and indicate that the algorithm for dynamic lattice building of the present thesis clearly outperforms the one for static building, both of them being with the same minimal known complexity.

Chapter 5

Conclusion

Having introduced results on the dynamics of concept lattices and algorithms arising therefrom with the minimal known complexity for their task, it is proper to look at the extension of the work into the field of general lattice dynamics with proper manipulation and use. The results so far allow only for the disjoint update or assembly of lattices, where attributes or objects with their incidence being added to contexts are not yet present there. Mutations without that restriction are something that remains to be explored. So is the internal update of a lattice, by which some object loses its incidence relation to an attribute and another one gains the same, or further variations thereof including their dual version. Of course, having inquired into the assembly of two sublattices, need arrives for disassembly. Present algorithms can be made more rational through further observations on the predecessors of concepts exclusive to the superlattice, and it is also worth exploring the practical applications of theory.

Bibliography

[1] G. Birkhoff, *Lattice Theory*, American Mathematical Society, 1940.

[2] B. A. Davey, H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[3] M. Barbut, B. Monjardet, *Ordre et Classification: Algèbre et Combinatoire*, Hachette, 1970.

[4] S. MacLane, G. Birkhoff, *Algebra*, Mcmillan, 1979.

[5] M. L. Dubreuil-Jacotin, L. Lesieur, R. Grisot, *Leçons sur la Théorie des Treillis des Structures Algébriques Ordonnées et des Treillis Géométriques*, Gauthier-Villers, 1953.

[6] H. Hermes, *Einführung in die Verbandstheorie*, Springer, 1955.

[7] G. Szász, *Introduction to Lattice Theory*, Academic Press, 1963.

[8] R. Wille, Restructuring the Lattice Theory: an Approach Based on Hierarchies of Concepts, in *Ordered Sets*, I. Rival Ed., pp. 445 - 470, Kluwer, 1982.

[9] B. Ganter, R. Wille, *Formale Begriffsanalyse: Mathematische Grundlagen*, Springer, 1996.

[10] B. Ganter, R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer, 1999.

[11] F. Vogt, *Formale Begriffsanalyse mit C++: Datenstrukturen und Algorithmen*, Springer, 1996.

[12] J. P. Bordat, Calcul Pratique du Treillis de Galois d'une Correspondance, in *Mathématiques et Sciences Humaines*, vol. 96, pp 31 - 47, 1986.

[13] B. Ganter, K. Reuter, Finding All Closed Sets: a General Approach, in *Order*, vol. 8, pp. 283 - 290, 1991.

[14] A. Guénoche, Construction du Treillis de Galois d'une Relation Binaire, in *Mathématiques et Sciences Humaines*, vol. 109, pp. 41 - 53, 1990.

[15] R. Wille, Subdirect Product Construction of Concept Lattices, in *Discrete Mathematics*, vol. 63(2-3), pp. 305 - 313, 1987.

[16] R. Godin, R. Missaoui, An Incremental Concept Formation Approach for Learning from Databases, in *Theoretical Computer Science*, vol. 133(2) 387 - 419, 1994.

[17] R. Godin, R. Missaoui, H. Alaoui, Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices, in *Computational Intelligence*, vol. 11(2), pp. 246 - 267, 1995.

[18] P. Valtchev, R. Missaoui, P. Lebrun, A Partition-Based Approach towards Constructing Galois (Concept) Lattices, in *Discrete Mathematics*, vol. 256(3), pp. 801 - 829, 2000.

[19] P. Valtchev, R. Missaoui, R. Godin, M. Meridji, Generating Frequent Itemsets Incrementally: Two Novel Approaches Based on Galois Lattice Theory, in *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 14(2-3), pp. 115 - 142, 2002.

[20] L. Nourine, O. Reynaud, A Fast Algorithm for Building Lattices, in *Information Processing Letters*, vol. 71(5-6), pp. 199 - 204, 1999.

[21] C. Jard, G. V. Jourdan, J. X. Rampon, Computing On-Line the Lattice of Maximal Antichains of Posets, in *Order*. vol. 11(2), pp. 197 - 210, 1994.

[22] P. Valtchev, An Algorithm for Minimal Insertion in a Type Lattice, in *Computational Intelligence*, vol. 15(1), pp. 63 - 78, 1999.

[23] G. Stumme, Free Distributive Completions of Partial Complete Lattices, in *Order*, vol. 14(2), pp. 179 - 189, 1997.

[24] S. O. Kuznetsov, On Computing the Size of a Lattice and Related Decision Problems, in *Order*, vol. 18(4), pp. 313 - 321, 2001.

[25] J. von Neumann, *Continuous Geometry*, Princeton University Press, 1960.

[26] G. Buskes, A. van Rooij, *Topological Spaces: from Distance to Neighborhood*, Springer, 1997.

Appendix A

Original Proofs

Proof A.1 (Theorem 3.1) *If* $(X,Y) \in \mathfrak{B}_{K\setminus a}$, *then* $(X'' \cap a', (X'' \cap a')') \in a_\downarrow$ *and, if* $X'' \neq X'' \cap a'$, *then* $(X'', X') \in \mathfrak{B}_K \setminus a_\downarrow$ *and* $(X'', X') = (X'', Y)$, *else* $(X'', X') = (X'', Y \cup a)$, *where*

$$a_\downarrow = \{(X'' \cap a', (X'' \cap a')') \in \mathfrak{B}_K \mid (X,Y) \in \mathfrak{B}_{K\setminus a} \text{ or } X = G\}$$

with $\mathfrak{B}_K \setminus a_\downarrow = \{(X'', Y) \in \mathfrak{B}_K \mid (X,Y) \in \mathfrak{B}_{K\setminus a} \text{ or } Y = \emptyset\}$ *and, if* $(X,Y) = (G \setminus {}'a, \emptyset)$ *with* ${}'a \neq \emptyset$, *then* $(G \setminus {}'a)'' = G$ *in* $\mathfrak{B}_K$, *while in all other cases,* $X'' = X$ *in the superlattice, and dually for* $o^\uparrow$.

Proof: *For the latter statement on* $(X,Y) \in \mathfrak{B}_{K\setminus a}$ *when* $(X,Y) \neq (G \setminus {}'a, \emptyset)$, *if* $X \subseteq a'$ *with* $X = \{x_1 \dots x_n\}$, $Y = \{y_1 \dots y_m\}$, *then*

$$X' = \bigcap_{j=1}^{n} x_j' = Y \cup a \text{ and } (Y \cup a)' = \bigcap_{j=1}^{m} y_j' = X$$

in the superlattice, otherwise X', Y' *are unchanged and either way* $X'' = X$. *In the case of* $(G \setminus {}'a, \emptyset)$, *the above reasoning applies if* ${}'a = \emptyset$, *otherwise* $(G \setminus {}'a)' = a$ *or* $(G \setminus {}'a)' = \emptyset$ *in the superlattice and* ${}'a \in (G \setminus {}'a)''$. *Starting from the beginning, since* $(X'', X') \in \mathfrak{B}_K$ *for all* $X \subseteq G$ *and* $(a', a'') \in \mathfrak{B}_K$, *then their pair infimum* $(X'' \cap a', (X'' \cap a')')$ *also is in the superlattice. If* $(X'', X') \in a_\downarrow$, *then* $(X'', X') \leq (a', a'')$ *and* $X \subseteq a'$. *When the extent is not a subset of* a', *or* $X'' \neq X'' \cap a'$, *then the concept must be in* $\mathfrak{B}_K \setminus a_\downarrow$, *with* $X' = Y$ *in the superlattice for* $(X,Y) \in \mathfrak{B}_{K\setminus a}$ *by the reasoning starting the proof, which also shows that* $X' = Y \cup a$ *if* $X \subseteq a$. *If* $(X_m, Y_m) \in a_\downarrow$, *then* $((Y_m \setminus a)', (Y_m \setminus a)'') \in \mathfrak{B}_{K\setminus a}$ *or* $((Y_m \setminus a)', (Y_m \setminus a)'') = (G, \emptyset)$ *and, in both cases,* $X_m = ((Y_i \setminus a) \cup a)' = (Y_i \setminus a)' \cap a'$ *in* $\mathfrak{B}_K$ *by Lemma 2.1. This gives, with* (X,Y) *for* $((Y_m \setminus a)', (Y_m \setminus a)'')$, *the definition of the order ideal above. If* $(X_i, Y_i) \in \mathfrak{B}_K \setminus a_\downarrow$, *then* $Y_i \subseteq M \setminus a$ *and* $Y_i' = X_i$ *in the sublattice, unless*

$(X_i, Y_i) = (G, \emptyset)$ with $'a \neq \emptyset$, in which case $Y_i' = G \setminus 'a$ in the sublattice. This defines, with (X, Y) for (Y_i', Y_i), the family $\mathfrak{B}_K \setminus a_\downarrow$ as above in the theorem.

Proof A.2 (Lemma 2.5) *Let $(X_i, Y_i) = \bigwedge R$ for $R = \{(X, Y) \in \mathfrak{B}_K \setminus a_\downarrow \mid X \cap a' = X_i \cap a'\}$ and there is no $(X_m, Y_m) \in \mathfrak{B}_{K \setminus a}$ with $X_i \cap a' = X_m$, then $(X_i \cap a', Y_i \cup a) \prec \bigwedge R$ and no other concept in $a_\downarrow$ precedes any in R, plus dually for an object o, $o^\uparrow$, and the supremum.*

Proof: *The difference of intents between $\bigwedge R$ and $(X_i \cap a', Y_i \cup a)$ is only a with the latter being infimum for the pair of the former and (a', a''), whence the precedence by Theorem 2.3. Notice that, in this case, $\bigwedge R \in R$ and, if $(X_m, Y_m) \in a_\downarrow$ with $X_m \neq X_i \cap a'$ preceded some $(X, Y) \in R$, then $X \cap a' = X_m \neq X_i \cap a'$, which is a contradiction and $(X_m, Y_m) \not\prec (X, Y)$.*

Proof A.3 (Theorem 2.4) *If $(X_i, Y_i) \prec (X_j, Y_j)$ in $\mathfrak{B}_{K \setminus a}$, then $(X_i, X_i') \prec (X_j'', X_j')$ in $\mathfrak{B}_K$, except that, if $(X_j \cap a', Y_j \cup a) \prec (X_j, Y_j)$ in the superlattice and $X_j \cap a' \neq X_i$, then $(X_i'', X_i') \not\prec (X_j'', X_j')$, and dually with $o \in G$.*

Proof: *If $(X_i, Y_i), (X_j, Y_j) \in \mathfrak{B}_K \setminus a_\downarrow$, which is to say $(X_i, X_i') = (X_i, Y_i)$ and $(X_j'', X_j') = (X_j, Y_j)$ or $(X_j'', X_j') = (X_j \cup 'a, Y_j)$ in the superlattice, where the latter is the case of $(X_j, Y_j) = (G \setminus 'a, \emptyset)$, then the difference of their intents remains the same and so does the precedence. If $(X_i, X_i'), (X_j'', X_j') \in a_\downarrow$, then $(X_i, X_i') = (X_i, Y_i \cup a)$ with $(X_j'', X_j') = (X_j, Y_j \cup a)$ or $(X_j'', X_j') = (a', Y_j \cup a) = (G, Y_j \cup a)$ in the superlattice, as per Theorem 3.1, where the latter case is for $(X_j'', X_j) = (G \setminus 'a, \emptyset)$ with $'a \neq \emptyset$ and $a' = G$, whence the difference in intents remains again the same from sublattice to superlattice and so does the precedence. If $(X_j, Y_j) \in \mathfrak{B}_K \setminus a_\downarrow$ and $(X_j, Y_j \cup a) \in a_\downarrow$, the precedence relation remains valid in the superlattice if $X_i \cap a' = X_j$, otherwise*

$$(X_j, Y_j) \prec (X_i \cap a', Y_i \cup a) \prec (X_i, Y_i)$$

as shown already by Lemma 2.5 above.

Proof A.4 (Lemma 3.2) *If $(X_i, Y_i), (X_j, Y_j) \in \mathfrak{B}_{K \setminus a}$, $(X_j, Y_j) \leq (X_i, Y_i)$, then the set $X_i \cap a' = X_i \cap (X_j \cap a')$ since $X_i \subseteq X_j$ and similarly in $\mathfrak{B}_{K \setminus o}$.*

Proof: *By Theorem 2.2, $X_j \subseteq X_i$ implies $X_j \cap a' \subseteq X_i \cap a'$, whence $X_j \cap a' = X_j \cap (X_i \cap a')$ and dually for $\mathfrak{B}_{K \setminus o}$.*

Appendix B

Illustrative Execution

The programme runs on a data file, supplied by the user, containing strictly increasing numbers separated by spaces on each line, with all lines being independent of each other. It is instructed, by calling options, to interpret line numbers of the data file as either objects or attributes and line contents as the proper incidence. Two test files with the object and with the attribute basis of the context from Table 2.1 are used in the demonstration and listed below.

```
7:26:26 more class.obj
1 3
1 2
1 2 4 5
5
7:26:37 more class.att
1 2 3
2 3
1
3
4 5
7:26:41
```

The execution results for static and for dynamic or gradual building of the lattice from the object basis, using respective options 0 or 2 at the end of the programme call, or from the attribute basis with options 1 and 3 follow. The different order of computing concepts with each building approach is visible in the order of their display and the same applies also on updating their successors. Notice that, unlike in the text, attributes are assigned numbers exactly

as objects and are distinguished by their position on the display of a concept, where objects are on the left and attributes on the right.

```
7:27:40 java -cp run.jar Run class.obj 0
objectBasis
1 : 1,3
2 : 1,2
3 : 1,2,4,5
4 : 5

infimum
null : 1,2,3,4,5
successors
3 : 1,2,4,5
1 : 1,3

supremum
1,2,3,4 : null
predecessors
1,2,3 : 1
3,4 : 5

concept
3,4 : 5
successors
1,2,3,4 : null
predecessors
3 : 1,2,4,5

concept
3 : 1,2,4,5
successors
3,4 : 5
2,3 : 1,2
predecessors
null : 1,2,3,4,5

concept
2,3 : 1,2
successors
1,2,3 : 1
predecessors
3 : 1,2,4,5

concept
1,2,3 : 1
successors
1,2,3,4 : null
predecessors
1 : 1,3
2,3 : 1,2

concept
1 : 1,3
successors
1,2,3 : 1
predecessors
null : 1,2,3,4,5

7:28:34 java -cp run.jar Run class.obj 2
objectBasis
1 : 1,3
2 : 1,2
3 : 1,2,4,5
4 : 5

concept
```

```
3,4 : 5
successors
1,2,3,4 : null
predecessors
3 : 1,2,4,5

supremum
1,2,3,4 : null
predecessors
3,4 : 5
1,2,3 : 1

concept
3 : 1,2,4,5
successors
3,4 : 5
2,3 : 1,2
predecessors
null : 1,2,3,4,5

concept
2,3 : 1,2
successors
1,2,3 : 1
predecessors
3 : 1,2,4,5

concept
1,2,3 : 1
successors
1,2,3,4 : null
predecessors
2,3 : 1,2
1 : 1,3

infimum
null : 1,2,3,4,5
successors
3 : 1,2,4,5
1 : 1,3

concept
1 : 1,3
successors
1,2,3 : 1
predecessors
null : 1,2,3,4,5

7:30:11 java -cp run.jar Run class.att 1
attributeBasis
1 : 1,2,3
2 : 2,3
3 : 1
4 : 3
5 : 4,5

supremum
1,2,3,4,5 : null
predecessors
4,5 : 5
1,2,3 : 1

concept
4,5 : 5
successors
1,2,3,4,5 : null
predecessors
null : 1,2,3,4,5
```

```
concept
3 : 1,2,4
successors
2,3 : 1,2
predecessors
null : 1,2,3,4,5

concept
1 : 1,3
successors
1,2,3 : 1
predecessors
null : 1,2,3,4,5

infimum
null : 1,2,3,4,5
successors
1 : 1,3
3 : 1,2,4
4,5 : 5

concept
2,3 : 1,2
successors
1,2,3 : 1
predecessors
3 : 1,2,4

concept
1,2,3 : 1
successors
1,2,3,4,5 : null
predecessors
1 : 1,3
2,3 : 1,2

7:31:17 java -cp run.jar Run class.att 3
attributeBasis
1 : 1,2,3
2 : 2,3
3 : 1
4 : 3
5 : 4,5

concept
4,5 : 5
successors
1,2,3,4,5 : null
predecessors
null : 1,2,3,4,5

supremum
1,2,3,4,5 : null
predecessors
4,5 : 5
1,2,3 : 1

concept
3 : 1,2,4
successors
2,3 : 1,2
predecessors
null : 1,2,3,4,5

concept
1 : 1,3
successors
```

```
1,2,3 : 1
predecessors
null : 1,2,3,4,5

infimum
null : 1,2,3,4,5
successors
4,5 : 5
3 : 1,2,4
1 : 1,3

concept
2,3 : 1,2
successors
1,2,3 : 1
predecessors
3 : 1,2,4

concept
1,2,3 : 1
successors
1,2,3,4,5 : null
predecessors
1 : 1,3
2,3 : 1,2

7:32:11
```

The algorithms on Table 2.2 and Table 3.2 were tested on the machine gle21.iro.umontreal.ca, which has a 400 MHz processor plus 128 MB system memory and operates on Linux kernel 2.4.18-27.7. The set of test files comprised randomly generated contexts with 5, 10, 15, 20 or 25 objects and attributes. The size of the average incidence o' or a', as compared to the size of G or M respectively, was set to 0.25, 0.5 and 0.75 for the tests. This ratio may be considered context density as it indicates how much a representation of the context like Table 2.1 is filled with + or not. A level of density is manually chosen as command line option 1, 2 or 3, for the respective density levels above, to the generator of random test files listed below.

```
11:02:53 more Ran.java
import java.io.*;
import java.util.*;
public final class Ran {
 private static BufferedWriter init (int g) {
  try { return new BufferedWriter (new FileWriter ("" + g)); }
  catch (IOException o) { return null; }
 }
 private static void load (int n) {
 BufferedWriter b[] = { init (1), init (2), init (3), init (4), init (5) };
 StringBuffer s = new StringBuffer ();
 Random r = new Random ();
  for (int k = 1; k <= 5; k ++) {
   for (int j = 1; j <= k * 5; j ++) {
    for (int i = 1; i <= k * 5; i ++)
```

```
    if (r.nextFloat () < n * 0.25) s.append ("" + i + " ");
   try { s.setCharAt(s.length () - 1, '\n'); }
   catch (IndexOutOfBoundsException e) { ; }
  }
  try {
   b[k - 1].write (s.toString ());
   b[k - 1].flush ();
   s.setLength (0);
  } catch (IOException u) { ; }
 }
}
public static void main (String[] argv) {
 try {
  int m = Integer.parseInt (argv[0]);
  if (m < 1 || m > 3) throw new NumberFormatException ();
  else load (m);
 } catch (Exception x) {
  System.out.println ("\nto user : use command line java Ran 1..3");
  System.out.println ("options : 1 - averages density of 25/100");
  System.out.println ("        : 2 - averages density of 50/100");
  System.out.println ("        : 3 - averages density of 75/100");
  System.out.println ("results : files named 1, 2, 3, 4, 5 with");
  System.out.println ("        : 5..25 basis units respectively");
  System.out.println ("warning : overriting files 1, 2, 3, 4, 5\n");
 }
}
}
11:03:35
```

Transcript of the test session follows. The first command creates context files with density 0.25, follows execution of the algorithm from Table 2.2 and, after it, the one from Table 3.2 on contexts of increasing size. All is repeated upon the creation of context files with density 0.5 and 0.75 too. Results are summarised on Table 4.1 in the text.

```
8:15:12 java Ran 1
8:15:23 time java -cp run.jar Run 1 0 >> temp.out
0.847u 0.078s 0:00.96 94.7%     0+0k 0+0io 1725pf+0w
8:15:34 time java -cp run.jar Run 1 2 >> temp.out
0.847u 0.068s 0:00.96 93.7%     0+0k 0+0io 1725pf+0w
8:15:41 time java -cp run.jar Run 2 0 >> temp.out
1.033u 0.072s 0:01.18 93.2%     0+0k 0+0io 1733pf+0w
8:15:52 time java -cp run.jar Run 2 2 >> temp.out
0.990u 0.111s 0:01.14 96.4%     0+0k 0+0io 1733pf+0w
8:15:58 time java -cp run.jar Run 3 0 >> temp.out
1.335u 0.148s 0:01.53 96.0%     0+0k 0+0io 1735pf+0w
8:16:09 time java -cp run.jar Run 3 2 >> temp.out
1.306u 0.101s 0:01.45 96.5%     0+0k 0+0io 1735pf+0w
8:16:16 time java -cp run.jar Run 4 0 >> temp.out
2.906u 0.203s 0:03.15 98.4%     0+0k 0+0io 1735pf+0w
8:16:29 time java -cp run.jar Run 4 2 >> temp.out
2.466u 0.199s 0:02.69 98.5%     0+0k 0+0io 1735pf+0w
8:16:39 time java -cp run.jar Run 5 0 >> temp.out
6.062u 0.369s 0:06.47 99.2%     0+0k 0+0io 1735pf+0w
8:16:59 time java -cp run.jar Run 5 2 >> temp.out
4.634u 0.335s 0:05.01 99.0%     0+0k 0+0io 1735pf+0w
8:17:10 java Ran 2
8:17:21 time java -cp run.jar Run 1 0 >> temp.out
0.847u 0.089s 0:00.97 94.8%     0+0k 0+0io 1725pf+0w
8:17:31 time java -cp run.jar Run 1 2 >> temp.out
0.861u 0.076s 0:00.96 96.8%     0+0k 0+0io 1725pf+0w
```

```
8:17:37 time java -cp run.jar Run 2 0 >> temp.out
1.437u 0.126s 0:01.60 96.8%     0+0k 0+0io 1735pf+0w
8:17:46 time java -cp run.jar Run 2 2 >> temp.out
1.345u 0.128s 0:01.51 96.6%     0+0k 0+0io 1735pf+0w
8:17:53 time java -cp run.jar Run 3 0 >> temp.out
5.810u 0.330s 0:06.18 99.3%     0+0k 0+0io 1735pf+0w
8:18:11 time java -cp run.jar Run 3 2 >> temp.out
3.933u 0.312s 0:04.29 98.8%     0+0k 0+0io 1735pf+0w
8:18:22 time java -cp run.jar Run 4 0 >> temp.out
48.503u 1.431s 0:50.17 99.5%    0+0k 0+0io 1735pf+0w
8:19:23 time java -cp run.jar Run 4 2 >> temp.out
22.179u 1.250s 0:23.95 97.7%    0+0k 0+0io 1735pf+0w
8:19:56 time java -cp run.jar Run 5 0 >> temp.out
304.970u 4.705s 5:12.26 99.1%   0+0k 0+0io 1735pf+0w
8:25:18 time java -cp run.jar Run 5 2 >> temp.out
104.013u 2.773s 1:46.86 99.9%   0+0k 0+0io 1735pf+0w
8:27:15 java Ran 3
8:27:33 time java -cp run.jar Run 1 0 >> temp.out
0.873u 0.074s 0:00.99 94.9%     0+0k 0+0io 1726pf+0w
8:27:49 time java -cp run.jar Run 1 2 >> temp.out
0.849u 0.082s 0:00.98 93.8%     0+0k 0+0io 1726pf+0w
8:27:56 time java -cp run.jar Run 2 0 >> temp.out
1.449u 0.109s 0:01.60 96.2%     0+0k 0+0io 1735pf+0w
8:28:05 time java -cp run.jar Run 2 2 >> temp.out
1.363u 0.107s 0:01.52 96.0%     0+0k 0+0io 1735pf+0w
8:28:14 time java -cp run.jar Run 3 0 >> temp.out
17.082u 0.695s 0:17.83 99.6%    0+0k 0+0io 1735pf+0w
8:28:39 time java -cp run.jar Run 3 2 >> temp.out
10.267u 0.599s 0:10.93 99.2%    0+0k 0+0io 1735pf+0w
8:28:56 time java -cp run.jar Run 4 0 >> temp.out
1761.218u 12.779s 29:41.55 99.5%        0+0k 0+0io 1735pf+0w
8:59:24
8:59:24 time java -cp run.jar Run 4 2 >> temp.out
951.166u 6.984s 16:02.06 99.5%  0+0k 0+0io 1735pf+0w
9:15:55
```

The Java programme code, based on the structures introduced in the text and used in a Java executable file named run.jar in the examples, is given in next appendix. It displays all calling options in the case of inappropriate call, or relevant information in the case of errors encountered. The algorithmic code is split among the different classes but easily identifiable by the names used in Table 2.2 and Table 3.2.

Appendix C

Java Source

```
import java.io.*;
public final class Run {
 private ObjectBase objectRead (double i, StreamTokenizer s) throws IOException {
  ObjectBase temp = null;
  temp = i == StreamTokenizer.TT_NUMBER ?
   new ObjectBase (s.lineno (), null, intentRead (i, s)) : null;
  if (temp != null) temp.next = objectRead (s.nextToken (), s);
  else if (i != StreamTokenizer.TT_EOF) throw new IOException ();
  return temp;
 }
 private AttributeBase attributeRead (double i, StreamTokenizer s) throws IOException {
  AttributeBase temp = null;
  temp = i == StreamTokenizer.TT_NUMBER ?
   new AttributeBase (s.lineno (), null, extentRead (i, s)) : null;
  if (temp != null) temp.next = attributeRead (s.nextToken (), s);
  else if (i != StreamTokenizer.TT_EOF) throw new IOException ();
  return temp;
 }
 private Extent extentRead (double i, StreamTokenizer s) throws IOException {
  Extent temp = null;
  if (i == StreamTokenizer.TT_NUMBER) {
   double j = s.nval;
   double k = s.nextToken ();
   if (k == StreamTokenizer.TT_NUMBER && s.nval <= j) throw new IOException ();
   temp = new Extent ((int) j, extentRead (k, s));
  } else if (i == StreamTokenizer.TT_WORD) throw new IOException ();
  return temp;
 }
 private Intent intentRead (double i, StreamTokenizer s) throws IOException {
  Intent temp = null;
  if (i == StreamTokenizer.TT_NUMBER) {
   double j = s.nval;
   double k = s.nextToken ();
   if (k == StreamTokenizer.TT_NUMBER && s.nval <= j) throw new IOException ();
   temp = new Intent ((int) j, intentRead (k, s));
  } else if (i == StreamTokenizer.TT_WORD) throw new IOException ();
  return temp;
 }
 private void load (String g, int i) {
  StreamTokenizer s;
  ObjectBase temp;
  AttributeBase other;
```

```
  Lattice l = new Lattice ();
  try {
   s = new StreamTokenizer (new BufferedReader (new FileReader (g)));
   s.eolIsSignificant (true);
   if (i == 0 || i == 2) {
    temp = objectRead (s.nextToken (), s);
    System.out.println("objectBasis\n" + temp.show ());
    if (i == 0) l.computeLattice (temp);
    else l.updateLattice (temp);
   } else {
    other = attributeRead (s.nextToken (), s);
    System.out.println("attributeBasis\n" + other.show ());
    if (i == 1) l.computeLattice (other);
    else l.updateLattice (other);
   }
   if (l.set.allConcepts != null) System.out.print (l.set.allConcepts.show());
  } catch (FileNotFoundException n) {
   System.out.println ("\nerror! problem accessing the input file!\n");
  } catch (IOException m) {
   System.out.println ("\nerror! problem with input file provided!");
   System.out.println ("insure increasing integers in each line!\n");
  }
 }
 public static void main (String[] argv) {
  Run r = new Run ();
  try {
   int i = Integer.parseInt (argv[1]);
   if (argv[0] == null || i > 3 || i < 0) throw new NumberFormatException ();
   else r.load (argv[0], i);
  } catch (Exception n) {
   System.out.println ("\nto user : java -cp run.jar Run file 0..3");
   System.out.println ("options : 0 - none of other oprions 1..3");
   System.out.println ("          1 - attribute basis input file");
   System.out.println ("          2 - gradually building lattice");
   System.out.println ("          3 - both of other options 1..2");
   System.out.println ("exemple : java -cp run.jar Run go.file 2");
   System.out.println ("in file : every line increasing integers\n");
  }
 }
}

final class Lattice {
 protected ObjectBase objectBasis;
 protected AttributeBase attributeBasis;
 protected Concept infimum, supremum;
 protected Set set;
 protected Lattice () { this.set = new Set (); }
 protected void computeLattice (ObjectBase b) {
  Concept temp, down;
  Intent up = b.side;
  down = new Concept (null, null);
  objectBasis = b;
  try { objectBasis.computeLattice (this, down, up); }
  catch (NullPointerException n) { ; }
  temp = set.intentFind (down.intent);
  if (temp == null) {
   temp = down;
   set.add (temp);
  }
  infimum = temp;
  supremum = set.intentFind (up);
  try { set.linkableConcepts.computeSuccessors (this); }
  catch (NullPointerException n) { ; }
  set.allConcepts.reset ();
  set.linkableConcepts = null;
 }
 protected void computeLattice (AttributeBase b) {
  Concept temp, up;
```

```
  Extent down = b.side;
  up = new Concept (null, null);
  attributeBasis = b;
  try { attributeBasis.computeLattice (this, up, down); }
  catch (NullPointerException n) { ; }
  temp = set.extentFind (up.extent);
  if (temp == null) {
   temp = up;
   set.add (temp);
  }
  supremum = temp;
  infimum = set.extentFind (down);
  try { set.linkableConcepts.computePredecessors (this); }
  catch (NullPointerException n) { ; }
  set.allConcepts.reset ();
  set.linkableConcepts = null;
 }
 protected void updateLattice (ObjectBase b) {
  objectBasis = b;
  try { objectBasis.updateLattice (this); } catch (NullPointerException n) { ; }
 }
 protected void updateLattice (AttributeBase b) {
  attributeBasis = b;
  try { attributeBasis.updateLattice (this); } catch (NullPointerException n) { ; }
 }
}

final class Set {
 protected ObjectRoot oroot;
 protected AttributeRoot aroot;
 protected List allConcepts, linkableConcepts;
 protected Set () {
  this.oroot = new ObjectRoot (null);
  this.aroot = new AttributeRoot (null);
 }
 protected void add (Concept c) {
  allConcepts =
   oroot.add (c.extent, aroot.add (c.intent, new List (c, allConcepts, null, null)));
  if (allConcepts.next != null) allConcepts.next.previous = allConcepts;
  linkableConcepts = new List (c, linkableConcepts, null, null);
  c.extentLength = c.extent != null ? c.extent.length () : 0;
  c.intentLength = c.intent != null ? c.intent.length () : 0;
 }
 protected Concept extentFind (Extent e) {
  List temp = oroot.find (e);
  return temp != null ? temp.concept : null;
 }
 protected Concept intentFind (Intent e) {
  List temp = aroot.find (e);
  return temp != null ? temp.concept : null;
 }
 protected void extentUpdate (Extent e, Concept c) {
  int temp = e.length ();
  if (temp > c.extentLength) {
   (oroot.add (e, oroot.remove (c.extent))).concept.extent = e;
   c.extentLength = temp;
  }
 }
 protected void intentUpdate (Intent e, Concept c) {
  int temp = e.length ();
  if (temp > c.intentLength) {
   (aroot.add (e, aroot.remove (c.intent))).concept.intent = e;
   c.intentLength = temp;
  }
 }
}

final class List {
```

```
protected Concept concept;
protected List next;
protected List previous;
protected List reciprocal;
protected List (Concept c, List n, List p, List r) {
 this.concept = c;
 this.next = n;
 this.previous = p;
 this.reciprocal = r;
}
protected String show () {
 String n = "";
 if (concept.predecessors == null || concept.successors == null) {
  if (concept.successors == null) n = "supremum\n";
  if (concept.predecessors == null) n = n + "infimum\n";
 } else n = "concept\n";
 return next != null ? n + concept.show () + concept.more () + next.show () :
  n + concept.show () + concept.more ();
}
protected String more () {
 return next != null ? concept.show () + next.more () : concept.show ();
}
protected void reset () {
 concept.visited = false;
 concept.modified = false;
 concept.count = 0;
 concept.resettableConcepts = null;
 try { next.reset (); } catch (NullPointerException n) { ; }
}
protected void computeLattice (Extent e, Intent t, Lattice l, Concept c)
 throws NullPointerException {
 Intent temp = t.intersect (concept.intent);
 Extent other = e.unite (concept.extent);
 Concept further = l.set.intentFind (temp);
 if (further != null) l.set.extentUpdate (other, further);
 else l.set.add (new Concept (other, temp));
 c.extent = further == concept ? c.extent : concept.extent != null ?
  concept.extent.intersect (c.extent) : null;
 next.computeLattice (e, t, l, c);
}
protected void computeLattice (Intent e, Extent t, Lattice l, Concept c)
 throws NullPointerException {
 Extent temp = t.intersect (concept.extent);
 Intent other = e.unite (concept.intent);
 Concept further = l.set.extentFind (temp);
 if (further != null) l.set.intentUpdate (other, further);
 else l.set.add (new Concept (temp, other));
 c.intent = further == concept ? c.intent : concept.intent != null ?
  concept.intent.intersect (c.intent) : null;
 next.computeLattice (e, t, l, c);
}
protected void computeSuccessors (Lattice l) throws NullPointerException {
 try { l.objectBasis.computeSuccessors (l, concept, concept.extent); }
 catch (NullPointerException n) { ; }
 concept.resettableConcepts =
  new List (concept, concept.resettableConcepts, null, null);
 concept.resettableConcepts.reset ();
 next.computeSuccessors (l);
}
protected void computePredecessors (Lattice l) throws NullPointerException {
 try { l.attributeBasis.computePredecessors (l, concept, concept.intent); }
 catch (NullPointerException n) { ; }
 concept.resettableConcepts =
  new List (concept, concept.resettableConcepts, null, null);
 concept.resettableConcepts.reset ();
 next.computePredecessors (l);
}
protected void updateLattice (ObjectBase b, Lattice l) throws NullPointerException {
```

```
  if (concept.visited != true) concept.updateLattice (b, l);
  next.updateLattice (b, l);
 }
 protected void updateLattice (AttributeBase b, Lattice l) throws NullPointerException {
  if (concept.visited != true) concept.updateLattice (b, l);
  next.updateLattice (b, l);
 }
 protected void objectClean (Concept c) throws NullPointerException {
  if (concept.modified == true) c.removeUpper (this);
  next.objectClean (c);
 }
 protected void attributeClean (Concept c) throws NullPointerException {
  if (concept.modified == true) c.removeLower (this);
  next.attributeClean (c);
 }
}

final class Concept {
 protected Extent extent;
 protected Intent intent;
 protected int extentLength, intentLength, count;
 protected List successors, predecessors, resettableConcepts;
 protected boolean visited, modified;
 protected Concept (Extent e, Intent i) {
  this.extent = e;
  this.intent = i;
 }
 protected String show () {
  String g, s;
  g = extent != null ? extent.show () : "null";
  s = intent != null ? intent.show () : "null";
  return g + " : " + s + "\n";
 }
 protected String more () {
  String g, s;
  g = successors != null ? "successors\n" + successors.more () : "";
  s = predecessors != null ? "predecessors\n" + predecessors.more () : "";
  return g + s + "\n";
 }
 protected void addUpper (Concept c) {
  successors = new List (c, successors, null, null);
  if (successors.next != null) successors.next.previous = successors;
  c.predecessors = new List (this, c.predecessors, null, successors);
  if (c.predecessors.next != null) c.predecessors.next.previous = c.predecessors;
  successors.reciprocal = c.predecessors;
 }
 protected void addLower (Concept c) {
  predecessors = new List (c, predecessors, null, null);
  if (predecessors.next != null) predecessors.next.previous = predecessors;
  c.successors = new List (this, c.successors, null, predecessors);
  if (c.successors.next != null) c.successors.next.previous = c.successors;
  predecessors.reciprocal = c.successors;
 }
 protected void removeUpper (List l) {
  if (l.previous != null) l.previous.next = l.next;
  else successors = l.next;
  if (l.next != null) l.next.previous = l.previous;
  if (l.reciprocal.previous != null) l.reciprocal.previous.next = l.reciprocal.next;
  else l.concept.predecessors = l.reciprocal.next;
  if (l.reciprocal.next != null) l.reciprocal.next.previous = l.reciprocal.previous;
 }
 protected void removeLower (List l) {
  if (l.previous != null) l.previous.next = l.next;
  else predecessors = l.next;
  if (l.next != null) l.next.previous = l.previous;
  if (l.reciprocal.previous != null) l.reciprocal.previous.next = l.reciprocal.next;
  else l.concept.successors = l.reciprocal.next;
  if (l.reciprocal.next != null) l.reciprocal.next.previous = l.reciprocal.previous;
```

```
 }
 protected void updateLattice (ObjectBase b, Lattice l) {
  visited = true;
  try { successors.updateLattice (b, l); } catch (NullPointerException n) { ; }
  Intent temp = b.side.intersect (intent);
  Extent other = (new Extent (b.object, null)).unite (extent);
  Concept further = l.set.intentFind (temp);
  if (further != null) {
   if (further == this) {
    l.set.extentUpdate (other, this);
    modified = true;
   }
  } else {
   try { successors.objectClean (this); } catch (NullPointerException n) { ; }
   further = new Concept (other, temp);
   l.set.add (further);
   if (successors == null) l.supremum = further;
   addUpper (further);
  }
 }
 protected void updateLattice (AttributeBase b, Lattice l) {
  visited = true;
  try { predecessors.updateLattice (b, l); } catch (NullPointerException n) { ; }
  Extent temp = b.side.intersect (extent);
  Intent other = (new Intent (b.attribute, null)).unite (intent);
  Concept further = l.set.extentFind (temp);
  if (further != null) {
   if (further == this) {
    l.set.intentUpdate (other, this);
    modified = true;
   }
  } else {
   try { predecessors.attributeClean (this); } catch (NullPointerException n) { ; }
   further = new Concept (temp, other);
   l.set.add (further);
   if (predecessors == null) l.infimum = further;
   addLower (further);
  }
 }
}

abstract class LinkedStructure {
 private LinkedStructure next;
}

final class Intent extends LinkedStructure {
 protected int attribute;
 protected Intent next;
 protected Intent (int a, Intent n) {
  this.attribute = a;
  this.next = n;
 }
 protected String show () {
  return next != null ? "" + attribute + "," + next.show () : "" + attribute;
 }
 protected int length () {
  try { return next.length () + 1; } catch (NullPointerException n) { return 1; }
 }
 private Intent duplicate () {
  try { return new Intent (attribute, next.duplicate ()); }
  catch (NullPointerException n) { return new Intent (attribute, null); }
 }
 protected Intent intersect (Intent e) {
  try {
   return attribute == e.attribute ? new Intent (attribute, next.intersect (e.next)) :
    attribute < e.attribute ? next.intersect (e) : intersect (e.next);
  } catch (NullPointerException n) {
   return e != null && attribute == e.attribute ? new Intent (attribute, null) : null;
```

```
 }
}
protected Intent substract (Intent e) {
 try {
  return attribute == e.attribute ? next.substract (e.next) : attribute < e.attribute ?
   new Intent (attribute, next.substract (e)) : substract (e.next);
 } catch (NullPointerException n) {
  return e == null ? duplicate () : attribute < e.attribute ?
   new Intent (attribute, null) : null;
 }
}
protected Intent unite (Intent e) {
 try {
  return attribute == e.attribute ? new Intent (attribute, next.unite (e.next)) :
   attribute < e.attribute ? new Intent (attribute, next.unite (e)) :
   new Intent (e.attribute, unite (e.next));
 } catch (NullPointerException n) {
  return e == null ? duplicate () : attribute < e.attribute ?
   new Intent (attribute, e.duplicate ()) : e.next != null ?
   new Intent (attribute, e.next.duplicate ()) : new Intent (attribute, null);
 }
}
protected List addNext (AttributeRoot o, List l) {
 try {
  if (o.next == null || attribute < o.next.attribute)
   o.next = new AttributeTrie (attribute, null, o.next);
  return attribute == o.next.attribute ? next.addNext (o.next, l) : addSide (o.next, l);
 } catch (NullPointerException n) {
  o.next.concept = l;
  return o.next.concept;
 }
}
private List addSide (AttributeTrie o, List l) {
 try {
  if (o.side == null || attribute < o.side.attribute)
   o.side = new AttributeTrie (attribute, null, o.side);
  return attribute == o.side.attribute ? next.addNext (o.side, l) : addSide (o.side, l);
 } catch (NullPointerException n) {
  o.side.concept = l;
  return o.side.concept;
 }
}
protected List removeNext (AttributeRoot o) {
 List temp;
 try {
  return o.next == null || attribute < o.next.attribute ? null :
   attribute == o.next.attribute ? next.removeNext (o.next) : removeSide (o.next);
 } catch (NullPointerException n) {
  temp = o.next.concept;
  o.next.concept = null;
  return temp;
 } finally { if (o.next.next == null && o.next.concept == null) o.next = o.next.side; }
}
private List removeSide (AttributeTrie o) {
 List temp;
 try {
  return o.side == null || attribute < o.side.attribute ? null :
   attribute == o.side.attribute ? next.removeNext (o.side) : removeSide (o.side);
 } catch (NullPointerException n) {
  temp = o.side.concept;
  o.side.concept = null;
  return temp;
 } finally { if (o.side.next == null && o.side.concept == null) o.side = o.side.side; }
}
protected List findNext (AttributeRoot o) {
 try {
  return o.next == null || attribute < o.next.attribute ? null :
   attribute == o.next.attribute ? next.findNext (o.next) : findSide (o.next);
```

```
  } catch (NullPointerException n) {
   return o.next.concept == null ? null : o.next.concept;
  }
 }
 private List findSide (AttributeTrie o) {
  try {
   return o.side == null || attribute < o.side.attribute ? null :
    attribute == o.side.attribute ? next.findNext (o.side) : findSide (o.side);
  } catch (NullPointerException n) {
   return o.side.concept == null ? null : o.side.concept;
  }
 }
}

final class Extent extends LinkedStructure {
 protected int object;
 protected Extent next;
 protected Extent (int o, Extent n) {
  this.object = o;
  this.next = n;
 }
 protected String show () {
  return next != null ? "" + object + "," + next.show () : "" + object;
 }
 protected int length () {
  try { return next.length () + 1; } catch (NullPointerException n) { return 1; }
 }
 private Extent duplicate () {
  try { return new Extent (object, next.duplicate ()); }
  catch (NullPointerException n) { return new Extent (object, null); }
 }
 protected Extent intersect (Extent e) {
  try {
   return object == e.object ? new Extent (object, next.intersect (e.next)) :
    object < e.object ? next.intersect (e) : intersect (e.next);
  } catch (NullPointerException n) {
   return e != null && object == e.object ? new Extent (object, null) : null;
   }
 }
 protected Extent substract (Extent e) {
  try {
   return object == e.object ? next.substract (e.next) : object < e.object ?
    new Extent (object, next.substract (e)) : substract (e.next);
  } catch (NullPointerException n) {
   return e == null ? duplicate () : object < e.object ? new Extent (object, null) : null;
  }
 }
 protected Extent unite (Extent e) {
  try {
   return object == e.object ? new Extent (object, next.unite (e.next)) :
    object < e.object ? new Extent (object, next.unite (e)) :
    new Extent (e.object, unite (e.next));
  } catch (NullPointerException n) {
   return e == null ? duplicate () : object < e.object ?
    new Extent (object, e.duplicate ()) : e.next != null ?
    new Extent (object, e.next.duplicate ()) : new Extent (object, null);
  }
 }
 protected List addNext (ObjectRoot o, List l) {
  try {
   if (o.next == null || object < o.next.object)
    o.next = new ObjectTrie (object, null, o.next);
   return object == o.next.object ? next.addNext (o.next, l) : addSide (o.next, l);
  } catch (NullPointerException n) {
   o.next.concept = l;
   return o.next.concept;
  }
 }
```

```
 private List addSide (ObjectTrie o, List l) {
  try {
   if (o.side == null || object < o.side.object)
    o.side = new ObjectTrie (object, null, o.side);
   return object == o.side.object ? next.addNext (o.side, l) : addSide (o.side, l);
  } catch (NullPointerException n) {
   o.side.concept = l;
   return o.side.concept;
  }
 }
 protected List removeNext (ObjectRoot o) {
  List temp;
  try {
   return o.next == null || object < o.next.object ? null : object == o.next.object ?
    next.removeNext (o.next) : removeSide (o.next);
  } catch (NullPointerException n) {
   temp = o.next.concept;
   o.next.concept = null;
   return temp;
  } finally {
   if (o.next.next == null && o.next.concept == null) o.next = o.next.side;
  }
 }
 private List removeSide (ObjectTrie o) {
  List temp;
  try {
   return o.side == null || object < o.side.object ? null : object == o.side.object ?
    next.removeNext (o.side) : removeSide (o.side);
  } catch (NullPointerException n) {
   temp = o.side.concept;
   o.side.concept = null;
   return temp;
  } finally {
   if (o.side.next == null && o.side.concept == null) o.side = o.side.side;
  }
 }
 protected List findNext (ObjectRoot o) {
  try {
   return o.next == null || object < o.next.object ? null : object == o.next.object ?
    next.findNext (o.next) : findSide (o.next);
  } catch (NullPointerException n) { return o.next.concept == null ? null :
   o.next.concept; }
 }
 private List findSide (ObjectTrie o) {
  try {
   return o.side == null || object < o.side.object ? null : object == o.side.object ?
    next.findNext (o.side) : findSide (o.side);
  } catch (NullPointerException n) {
   return o.side.concept == null ? null : o.side.concept;
  }
 }
}

final class AttributeBase extends LinkedStructure {
 protected int attribute;
 protected AttributeBase next;
 protected Extent side;
 protected AttributeBase (int o, AttributeBase n, Extent s) {
  this.attribute = o;
  this.next = n;
  this.side = s;
 }
 protected String show () {
  return next != null ? "" + attribute + " : " + side.show () + "\n" + next.show () :
   "" + attribute + " : " + side.show () + "\n";
 }
 protected void computeLattice (Lattice l, Concept c, Extent e)
  throws NullPointerException {
```

```
 Intent temp = new Intent (attribute, null);
 Concept other = null;
 c.extent = side.unite (c.extent);
 c.intent = temp.unite (c.intent);
 e = side.intersect (e);
 try { l.set.allConcepts.computeLattice (temp, side, l, c); }
 catch (NullPointerException x) { ; }
 other = l.set.extentFind (side);
 if (other == null) {
  l.set.add (new Concept (side, temp));
  c.intent = temp.intersect (c.intent);
 } else side = other.extent;
 next.computeLattice (l, c, e);
}
protected void computePredecessors (Lattice l, Concept c, Intent e) {
 Concept temp;
 if (e == null || attribute < e.attribute) {
  temp = l.set.extentFind (side.intersect (c.extent));
  c.resettableConcepts = new List (temp, c.resettableConcepts, null, null);
  temp.count += 1;
  if (temp.intentLength - temp.count == c.intentLength) c.addLower (temp);
  next.computePredecessors (l, c, e);
 } else next.computePredecessors (l, c, e.next);
}
protected void updateLattice (Lattice l) throws NullPointerException {
 Extent temp;
 Concept further = null;
 if (l.supremum != null) {
  if (side.substract (l.supremum.extent) != null) {
   temp = side.unite (l.supremum.extent);
   if (l.supremum.intent == null) l.set.extentUpdate (temp, l.supremum);
   else {
    further = new Concept (temp, null);
    further.addLower (l.supremum);
    l.supremum = further;
    l.set.add (further);
    l.set.linkableConcepts = null;
   }
  }
  l.supremum.updateLattice (this, l);
  try { l.set.linkableConcepts.computePredecessors (l); }
  catch (NullPointerException n) { ; }
  l.set.allConcepts.reset ();
  l.set.linkableConcepts = null;
 } else {
  l.set.add (new Concept (side, new Intent (attribute, null)));
  l.supremum = l.set.allConcepts.concept;
  l.infimum = l.supremum;
  l.set.linkableConcepts = null;
 }
 next.updateLattice (l);
}
}

class AttributeRoot extends LinkedStructure {
 protected AttributeTrie next;
 protected List concept;
 protected AttributeRoot (AttributeTrie n) {
  this.next = n;
  this.concept = null;
 }
 protected String show () {
  String s = concept != null ? "null\n" : "";
  return next != null ? s + next.show ("") : s + "";
 }
 protected List add (Intent e, List l) {
  if (e == null) concept = l;
  return e != null ? e.addNext (this, l) : concept;
```

```
 }
 protected List remove (Intent e) {
  List temp;
  temp = e == null ? concept : e.removeNext (this);
  if (e == null) concept = null;
  return temp;
 }
 protected List find (Intent e) { return e == null ? concept : e.findNext (this); }
}

final class AttributeTrie extends AttributeRoot {
 protected int attribute;
 protected AttributeTrie side;
 protected AttributeTrie (int a, AttributeTrie n, AttributeTrie s) {
  super (n);
  this.attribute = a;
  this.side = s;
 }
 protected String show (String s) {
  String t = s + attribute;
  String g = concept == null ? "" : t + "\n";
  return next != null && side != null ? g + next.show (t + ",") + side.show (s) :
   next != null ? g + next.show (t + ",") : side != null ? g + side.show (s) : g;
 }
}

final class ObjectBase extends LinkedStructure {
 protected int object;
 protected ObjectBase next;
 protected Intent side;
 protected ObjectBase (int o, ObjectBase n, Intent s) {
  this.object = o;
  this.next = n;
  this.side = s;
 }
 protected String show () {
  return next != null ? "" + object + " : " + side.show () + "\n" + next.show () :
   "" + object + " : " + side.show () + "\n"; }
 protected void computeLattice (Lattice l, Concept c, Intent e)
  throws NullPointerException {
  Extent temp = new Extent (object, null);
  Concept other = null;
  c.intent = side.unite (c.intent);
  c.extent = temp.unite (c.extent);
  e = side.intersect (e);
  try { l.set.allConcepts.computeLattice (temp, side, l, c); }
  catch (NullPointerException x) { ; }
  other = l.set.intentFind (side);
  if (other == null) {
   l.set.add (new Concept (temp, side));
   c.extent = temp.intersect (c.extent);
  } else side = other.intent;
  next.computeLattice (l, c, e);
 }
 protected void computeSuccessors (Lattice l, Concept c, Extent e)
  throws NullPointerException {
  Concept temp;
  if (e == null || object < e.object) {
   temp = l.set.intentFind (side.intersect (c.intent));
   c.resettableConcepts = new List (temp, c.resettableConcepts, null, null);
   temp.count += 1;
   if (temp.extentLength - temp.count == c.extentLength) c.addUpper (temp);
   next.computeSuccessors (l, c, e);
  } else next.computeSuccessors (l, c, e.next);
 }
 protected void updateLattice (Lattice l) throws NullPointerException {
  Intent temp;
  Concept further;
```

```
  if (l.infimum != null) {
   if (side.substract (l.infimum.intent) != null) {
    temp = side.unite (l.infimum.intent);
    if (l.infimum.extent == null) l.set.intentUpdate (temp, l.infimum);
    else {
     further = new Concept (null, temp);
     further.addUpper (l.infimum);
     l.infimum = further;
     l.set.add (further);
     l.set.linkableConcepts = null;
    }
   }
   l.infimum.updateLattice (this, l);
   try { l.set.linkableConcepts.computeSuccessors (l); }
    catch (NullPointerException n) { ; }
   l.set.allConcepts.reset ();
   l.set.linkableConcepts = null;
  } else {
   l.set.add (new Concept (new Extent (object, null), side));
   l.infimum = l.set.allConcepts.concept;
   l.supremum = l.infimum;
   l.set.linkableConcepts = null;
  }
  next.updateLattice (l);
 }
}

class ObjectRoot extends LinkedStructure {
 protected ObjectTrie next;
 protected List concept;
 protected ObjectRoot (ObjectTrie n) {
  this.next = n;
  this.concept = null;
 }
 protected String show () {
  String s = concept != null ? "null\n" : "";
  return next != null ? s + next.show ("") : s + "";
 }
 protected List add (Extent e, List l) {
  if (e == null) concept = l;
  return e != null ? e.addNext (this, l) : concept;
 }
 protected List remove (Extent e) {
  List temp;
  temp = e == null ? concept : e.removeNext (this);
  if (e == null) concept = null;
  return temp;
 }
 protected List find (Extent e) { return e == null ? concept : e.findNext (this); }
}

final class ObjectTrie extends ObjectRoot {
 protected int object;
 protected ObjectTrie side;
 protected ObjectTrie (int o, ObjectTrie n, ObjectTrie s) {
  super (n);
  this.object = o;
  this.side = s;
 }
 protected String show (String s) {
  String t = s + object;
  String g = concept == null ? "" : t + "\n";
  return next != null && side != null ? g + next.show (t + ",") + side.show (s) :
   next != null ? g + next.show (t + ",") : side != null ? g + side.show (s) : g;
 }
}
```