Université de Montréal

Mise à jour de la famille des générateurs minimaux des treillis de concepts et des icebergs

par

Kamal Nehme

Département 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
en informatique

Décembre, 2004

Université de Montréal

Faculté des études supérieures

Ce Mémoire intitulé:

Mise à jour de la famille des générateurs minimaux
des treillis de concepts et des icebergs

présenté par:
Kamal Nehme

a été évalué par un jury composé des personnes suivantes:

Gena Hahn	président-rapporteur
Petko Valtchev	directeur de recherche
Robert Godin	codirecteur
Victor Ostromoukhov	membre du jury

Mémoire accepté le: 3 mars 2005

Résumé

Une règle d'association est une implication de la forme (prémisse ⇒ conséquence) ayant pour but d'identifier des co-occurrences entre les sous-ensembles d'items de la base de données, appelés motifs. L'approche classique d'extraction de règles d'association consiste, d'abord, à déterminer l'ensemble des motifs fréquents, ensuite, à générer les règles d'association à partir de ces motifs. Le grand nombre de motifs fréquents extraits donne naissance à un grand nombre de règles d'une forte proportion de redondance.

L'analyse formelle de concepts (AFC) construit, à partir d'un ensemble d'objets, tous les sous-ensembles maximaux qui se caractérisent par le partage de collections d'attributs. Les couples d'ensembles fermés (objets, attributs), appelés concepts, sont organisés dans un treillis. L'AFC a été utilisé avec succès pour limiter le nombre des motifs pour l'extraction des règles d'association. En effet, seules les règles maximalement informatives, c.-à-d., avec une prémisse minimale et une conclusion maximale sont générées: la conséquence d'une telle règle est un motif fermé et la prémisse est un générateur minimal de ce motif. Un générateur minimal est un sous-ensemble minimal de son motif fermé dont la présence au sein d'une transaction entraîne celle du motif entier.

Notre étude a pour objectif de maintenir, d'une manière efficace, les familles des générateurs minimaux lors de l'évolution dans la base de données, par exemple, quand un nouvel attribut est ajouté. Nous proposons un ensemble de propriétés structurelles qui sous-tendent notre méthode. Ces propriétés sont transformées en des méthodes algorithmiques pour la maintenance de la famille des générateurs minimaux. Les analyses empiriques de l'implémentation de nos méthodes comparées avec celles de la littérature ont démontré la supériorité de notre approche.

Mots clés: analyse formelle des concepts, motifs fermés fréquents, règles d'association, méthodes incrémentales.

Abstract

An association rule is an implication of the form "premise $\Rightarrow$ consequence". It is used to identify significant co-occurrences of subsets of items, called itemsets, in a transaction database. The classical problem of extracting such rules consists in, on the one hand, determining the frequently occurring itemsets, and on the other hand, drawing associations between parts of the discovered itemsets. This method gives rise to a large amount of rules with a strong proportion of redundancy.

Formal concept analysis (FCA) builds, from a set of objects described by a collection of attributes, all the maximum subsets sharing some attributes. The couples of closed sets (objects, attributes), called concepts, are organized in a lattice. FCA has been successfully used to limit the number of itemsets for association rule mining. Indeed, only only sets with minimal premise and maximal conclusion are generated. The premise of such a rule is a minimal generator, that is a minimal generating subset of its respective closed itemset. It is therefore interesting to study the efficient computation of the minimal generator families.

Our study aims at efficiently maintaining the minimal generators families upon evolution in the dataset, e.g., when a new attribute is added. We propose a set of theoretical results that characterize the evolution and especially the involved lattice substructures. The provided properties are transformed into a set of algorithmic methods that maintain the minimal generator family. The empirical analysis of the implementation of our methods as compared to those of the literature showed the superiority of our approach.

Key Words: formal concept analysis, frequent closed itemsets, association rules, incremental methods.

Table des matières

Liste des tableaux

Table des figures

Remerciements

Je tiens à remercier tout particulièrement mon directeur de recherche, Monsieur Petko Valtchev, professeur ajoint au DIRO[1] de l'Université de Montréal (UdM) pour sa courtoisie, sa disponibilité, ses conseils et pour l'intérêt qu'il a manifesté pour mon travail.

Je remercie également mon codirecteur, Monsieur Robert Godin, professeur titulaire au département d'informatique de l'Université du Québec à Montréal (UQÀM) pour son aide et sa gentillesse.

Je tiens à remercier Céline du laboratoire LATECE[2] pour son aide et sa bonne humeur et Amine du DIRO pour sa collaboration et son amitié.

Enfin, mille merci à mes parents qui, malgré la distance, m'ont toujours soutenu et encouragé.

1. Département d'Informatique et de Recherche Opérationnelle
2. Laboratoire de recherche sur les TEchnologies du Commcerce Electronique

Chapitre 1

Introduction

1.1 Problématique

1.1.1 Fouille de données

La *fouille de données*, ou encore *l'extraction de connaissances* à partir des données, est un domaine de recherche des plus actifs de nos jours dont le but principal est d'exploiter les grandes quantités de données (e.g., transactions commerciales) collectées dans les divers champs d'application de l'informatique. Cette exploitation consiste dans la plupart des cas à extraire diverses régularités[PsF91], en particulier, les *règles d'association*. Une règle d'association est une implication de la forme (prémisse $\Rightarrow$ conséquence) ayant pour but d'identifier des relations significatives entre sous-ensembles d'items de la base de données, appelés "*motifs*". Un motif est une conjonction de propriétés élémentaires, appelés "*attributs*" ou *items*. Comme une base de données peut conduire à l'extraction d'un grand nombre de motifs, des mesures d'intérêt sont souvent introduites permettant ainsi le filtrage des motifs considérés comme significatifs. Lorsque cette mesure porte sur la fréquence d'occurence, relativement à un seuil fixé, alors les motifs significatifs sont appelés *motifs fréquents*.

1.1.2 Problème de l'extraction des règles d'association

L'approche classique d'extraction de règles d'association consiste, d'abord, à déterminer l'ensemble des motifs fréquents, ensuite, à générer les règles d'association à partir de ces motifs. La complexité de la tâche d'extraction des motifs fréquents est de nature exponentielle en fonction des attributs ce qui donne naissance à un grand nombre de règles d'une forte proportions de redondance. La génération d'un ensemble de taille réduite des règles maximisant l'information convoyée est un problème important pour la pertinence et l'utilité des algorithmes d'extraction des règles d'association.

1.2 Motivation

Ce travail consiste à utiliser l'Analyse Formelle de Concept (AFC) pour l'extraction des motifs fréquents.

Des travaux antérieurs basés sur l'AFC ont conduit à une approche qui, au lieu d'extraire la totalité des motifs fréquents, se limite à un ensemble restreint et ceci sans perte d'information, ce qui montre l'efficacité de cette approche. Pour ce faire, l'AFC s'appuie sur la notion de fermeture des motifs identifiés dans une relation binaire liant les objets (transactions) aux attributs (items). Ceci permet de générer toutes les règles à prémisse minimale et conséquence maximale. La conséquence d'une règle est un motif fermé et la prémisse un générateur minimal de ce motif. Les générateurs minimaux sont des sous-ensembles d'attributs d'un motif fermé dont la présence au sein d'une transaction entraîne celle du motif entier. Finalement, la base de règles recherchées est composée des règles extraites de ces motifs fréquents.

De son côté, l'évolution des bases de données a suscité la conception de l'approche incrémentale pour la maintenance des motifs fermés suite à l'ajout/supression d'un(e) nouvel(le) transaction/item à la base. Les résulats encourageant des travaux sur l'incrémentatlité de l'ensemble des motifs fermés ont ouvert la voie à son application au problème des règles d'association. Le but est de maintenir simultanément les motifs fermés et leurs générateurs.

Dans ce travail, on se propose de trouver des réponses aux questions suivantes:

- Comment calculer les générateurs des motifs fermés suite à l'ajout d'un nouvel objet/attribut ,
- Comment limiter le calcul sur les motifs fermés fréquents.

Le travail est présenté en deux parties, à savoir, l'incrémentalité par objet et l'incrémentalité par attribut. Chaque partie traite, d'abord, le problème particulier de l'ajout d'un seul objet (attribut), ensuite, la généralisation vers l'ajout d'un amas d'objets (attributs). La table 1.1 récapitule les différentes tâches reliées à ce sujet ainsi que les noms des étudiants ayant contribué à la mise en oeuvre de la solution.

	Incrémentalité par objet		Incrémentalité par attribut	
Tâches	**Ajout 1 obj**	**Ajout n obj**	**Ajout 1 att**	**Ajout m att**
MAJ des motifs fermés	C.Frambourg [a]	C.Frambourg		
MAJ d'un iceberg	J.Jing [b]			
MAJ des générateurs	C.Frambourg	C.Frambourg		
MAJ (iceberg + générateurs)	C.Frambourg			

TAB. 1.1 – *Application de l'approche incrémentale pour la maintenance des icebergs et des générateurs des motifs fermés fréquents.*

[a] Céline Frambourg [Fra04]
[b] Jun Jing [Jin04]

1.3 Contribution

Le but de ce travail est d'apporter une contribution au problème de l'extraction des règles d'association de manière incrémentale par ajout d'un nouvel attribut au contexte. Nous espérions par là réaliser des meilleures performances que les algorithmes existants.

En effet, la construction incrémentale du treillis par ajout d'objet s'est avérée plus performante par rapport aux méthodes batch dans les bases éparses [KO02]. Du côté de l'incrémentalité par attribut, il est connu que dans les bases de données, le nombre des attributs est largement plus petit que le nombre des objets ce qui rend la construction du treillis par ajout d'attribut prône à une meilleure efficacité. Notre étude a pour objectif de cerner l'évolution des familles des générateurs minimaux lors de l'ajout d'un nouvel attribut afin d'en faciliter l'intégration au sein des algorithmes existants de maitien du treillis/famille des fermés. Nous proposons donc un cadre théorique et algorithmique pour effectuer la tâche.

1.4 Plan du mémoire

Le mémoire est organisé en deux grandes parties. La première partie est consacrée à l'état de l'art de l'extraction des règles d'association et de l'AFC. La deuxième partie présente nos travaux sur les problèmes sous-jacents. La structure du mémoire est comme suit:

- Le chapitre 2 introduit les notions de bases de l'AFC et présente les deux approches batch et incrémentale pour la construction du treillis.
- Le chapitre 3 discute l'approche classique d'extraction des règles d'association ainsi que l'usage classique de l'AFC à résoudre ce problème.
- L'approche incrémentale pour la maintenance des icebergs fait l'objet du chapitre 4.
- Le chapitre 5 expose un nouveau cadre pour la construction incrémentale des générateurs.
- Les performances des algorithmes proposés sont illustrées au chapitre 6 montrant les avantages et les inconvénients de notre approche par rapport aux méthodes par lots.

Chapitre 2

Analyse Formelle de Concepts

L'Analyse Formelle de Concepts (AFC) [GW99] a ses racines dans la théorie des treillis dont les fondements mathématiques ont été posés par Birkhoff [Bir40] et repose sur les connexions de Galois formellement décrites dans [BM70]. L'AFC permet d'identifier des groupements d'objets ayant des attributs en commun et de les organiser en une hiérarchie conceptuelle (*treillis*).

2.1 Les treillis

Les définitions et les propriétés que nous présentons dans cette section sont extraites de [DP92]

Définition 1 (Relation binaire). *Une relation binaire d'un ensemble E vers un ensemble F est une partie R de $E \times F$. Si $(x,y) \in R$, on dit que x est en relation avec y et on note xRy.*

Définition 2 (Ordre partiel). *Soient E un ensemble, $\leq$, est une relation binaire sur E. On dit que $\leq$ est une relation d'ordre si elle vérifie les propriétés de réflexivité, d'antisymétrie et de transitivité pour tout élément x, y et $z \in E$.*

- *Réflexivité :* $x \leq x$
- *Antisymétrie :* $x \leq y$ *et* $y \leq x \Rightarrow x = y$
- *Transitivité :* $x \leq y$ *et* $y \leq z \Rightarrow x \leq z$.

Définition 3 (Ensemble ordonné). *Un couple $(E, \leq)$ formé d'un ensemble E et d'une relation d'ordre $\leq$ est appelé ensemble ordonné.*

Définition 4 (Relation de couverture). *Soient $(E,\leq)$ un ensemble ordonné et $a \leq b \in E$. Si pour tout $x \in E$, soit $x \leq a$, soit $b \leq x$, on dit alors que a et b sont consécutifs, que a est couvert par b ou que b couvre a. On écrit alors $a \prec b$ (ou $b \succ a$).*

Exemple Soit E l'ensemble formé par les diviseurs de 30. $E = \{1,2,3,5,6,10,15,30\}$ et $\leq$ la relation "divise". La figure 2.1 montre le graphe associé à $\leq$.

	1	2	3	5	6	10	15	30
1	x	x	x	x	x	x	x	x
2		x			x	x		x
3			x		x		x	x
5				x		x	x	x
6					x			x
10						x		x
15							x	x
30								x

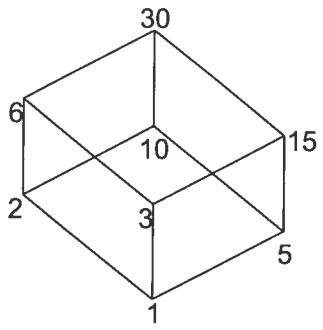

FIG. 2.1 – *le graphe associé à (E,divise).*

Définition 5 (Majorant - Minorant). *Soient $(E,\leq)$ un ensemble ordonné et S une partie non vide de E. On dit qu'un élément a est un majorant (respectivement un minorant) de S, si pour tout x de S, $x \leq a$ (respectivement $x \geq a$). A noter que S peut avoir plusieurs majorants (respectivement minorants).*

Définition 6 (infimum - supremum). *Soient $(E,\leq)$ un ensemble ordonné et S une par-*

tie non vide de E. L'infimum noté $\bigwedge$[1] (respectivement supremum noté $\bigvee$[2]) de S est le plus grand (respectivement le plus petit) élément, s'il existe, de l'ensemble des minorants (resp. des majorants) de S.

Définition 7 (Relation inverse). *Soit $\leq$ une relation binaire d'un ensemble E vers un ensemble F. On appelle relation inverse de $\leq$ et on note $\geq$ la relation binaire de F vers E définie par: $(\forall(x,y) \in E \times F)$, $x \leq y \Leftrightarrow y \geq x$.*

Principe de la dualité Notons $\geq$ la relation inverse de $\leq$. Une assertion impliquant $\leq$, $\vee$, $\wedge$ reste vraie en remplaçant $\leq$ par $\geq$ et en permuttant $\vee$ et $\wedge$. Il s'agit du principe de la dualité.

Définition 8 (Treillis). *Un ensemble ordonné $(E,\leq)$ est un treillis si pour tout x, $y \in E$ $x \bigwedge y$ et $x \bigvee y$ existent.*

Par exemple, dans le cas de l'ensemble des diviseurs de 30, $\bigvee$ = pgcd[3] et $\bigwedge$ = ppcm[4]

Définition 9 (Treillis complet). *Le treillis E est complet si pour toute partie S non vide de E, $\bigvee S$ et $\bigwedge S$ existent.*

Notons que, si E est un treillis fini alors il est un treillis complet.

2.2 Treillis de Galois

À partir d'une relation binaire, on peut construire un treillis. Dans la terminologie de l'AFC [GW99], une relation binaire entre les objets et les attributs est appelée contexte.

Définition 10 (Contextes Formels). *: Un contexte formel est un triplet $\mathcal{K} = (O,A,I)$ où O et A sont deux ensembles (respectivement, ensemble des objets formels et des attributs formels) et $I \subseteq O \times A$ est une relation modélisant quels attributs sont associées à chaque objet: la notation oIa signifie que a est un attribut de l'objet o.*

1. appelé en anglais meet
2. appelé en anglais join
3. plus grand commun diviseur
4. plus petit commun multiple

Considérons l'exemple de la figure 2.2 qui va nous accompagner tout au long de ce mémoire à titre illustratif.

	a	b	c	d	e	f	g	h
1		X	X	X			X	X
2	X		X					
3					X	X		X
4							X	
5				X	X	X	X	X

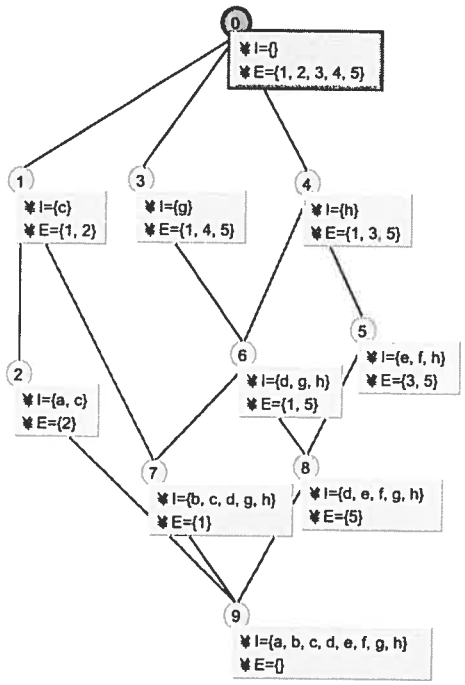

FIG. 2.2 – ***Gauche:*** *la table binaire* $\mathcal{K}$ *=*$(O = \{1,2,...,5\}, A = \{a,b,...,h\}, I)$. ***Droite:*** *Le diagramme de Hasse du treillis dérivé à partir de* $\mathcal{K}$.

Correspondance de Galois

Soient P et Q deux ensembles ordonnés, f une application de P dans Q et g une application de Q dans P. On dit que le couple (f,g) d'applications établit une *correspondance de Galois* entre P et Q si la condition suivante est satisfaite:

$$\forall X \in P, \forall X \in Q \; f(X) \leq Y \Leftrightarrow g(Y) \leq X$$

Définition 11. *Soient f la fonction qui associe à un ensemble d'objets X, l'ensemble Y des attributs partagés (ou communs) par tous les objets de X et g la fonction duale pour les ensembles d'attributs. Formellement, on a:*

$f : \mathcal{P}(O) \rightarrow \mathcal{P}(A)$, $f(X) = \{a \in A | \forall o \in X, oIa\}$

$g : \mathcal{P}(A) \rightarrow \mathcal{P}(O)$, $g(Y) = \{o \in O | \forall a \in Y, oIa\}$

Par la suite, les deux fonctions f et g seront notées par $'$.

Exemple Dans le contexte de la figure 2.2, $g(\{c\}) = \{1,2\}$ et $f(\{1,2\}) = \{c\}$.
$g(\{a\}) = \{2\}$ et $f(\{2\}) = \{a,c\}$.

Propriété 1. *Le couple (f,g) de la définition 11 définit une correspondance de Galois entre 2^O et 2^A du contexte $\mathcal{K} = (O,A,I)$.*

Opérateur de Fermeture

Soit $(P, \leq)$ un ensemble ordonné. On appelle *opérateur de fermeture* sur l'ensemble P toute application φ de P dans lui même qui vérifie les trois propriétés suivantes:

1. Extensive: $(\forall\ x \in P)$, $x \leq \varphi(x)$
2. Monotone: $(\forall\ x, y \in P)$, $x \leq y \Rightarrow \varphi(x) \leq \varphi(y)$
3. Idempotente: $(\forall\ x \in P)$, $(\varphi \circ \varphi)(x) = \varphi(x)$

Propriété 2. *Les compositions des fonctions f et g (de la définition 11) $f \circ g : (2^A, \subseteq) \rightarrow (2^A, \subseteq$) et $g \circ f : (2^O, \subseteq) \rightarrow (2^O, \subseteq)$ sont les opérateurs de fermeture sur 2^A et 2^O respectivement.*

Par la suite, les deux fonctions composées seront notées par $''$.

Définition 12 (fermé). *Etant donné un opérateur de fermeture φ sur un ensemble ordonné $(E, \leq)$, un élément $x \in E$ est un élément fermé si l'image de x par l'opérateur de fermeture est lui même: $\varphi(x) = x$. Ainsi, dans notre contexte $\mathcal{K} = (O,A,I)$, $X \in \mathcal{P}(O)$ (respectivement $Y \in \mathcal{P}(A)$) est fermé si $X'' = X$ (respectivement $Y'' = Y$).*

Exemple Dans le contexte de la figure 2.2, c est fermé puisque $c'' = c$ alors que a n'est pas fermé puisque $a'' = ac$ (ac signifie $\{a,c\}$; dans la suite, nous n'utilisons aucun séparateur entre les éléments des ensembles).

Ainsi, l'opérateur $''$ $(g \circ f)$ génère des sous-ensembles fermés d'objets tandis que $''$ $(f \circ g)$ des sous-ensembles fermés d'attributs. Un des résultats fondamentaux de la théorie des treillis de Galois stipule que ces deux familles de sous-ensembles fermés d'objets $\mathcal{C}_O$ et d'attributs $\mathcal{C}_A$ constituent deux treillis complets $\mathcal{L}_O$ et $\mathcal{L}_A$ lorsque la relation d'ordre choisie est l'inclusion ensembliste. D'autre part, la fonction $'$ (f)(resp. g) est une bijection entre les familles $\mathcal{C}_O$ et $\mathcal{C}_A$ (resp. $\mathcal{C}_A$ et $\mathcal{C}_O$) et définit un isomorphisme entre les deux treillis complets $\mathcal{L}_O$ et $\mathcal{L}_A$ (resp. $\mathcal{L}_A$ et $\mathcal{L}_O$). Ainsi, chacun des ensembles fermés d'objets (resp. attributs) du treillis $\mathcal{L}_O$ (resp. $\mathcal{L}_A$) possède un homologue unique parmi les ensembles fermés d'attributs (resp. objets) dans $\mathcal{L}_A$ (resp. $\mathcal{L}_O$). Le couple formé d'un ensemble fermé d'objets et de son ensemble fermé correspondant d'attributs (motifs fermés) constitue un concept.

Définition 13 (Concept formel). *Un concept formel d'un contexte $\mathcal{K}$ est une paire (X,Y) où $X \in \mathcal{P}(O)$, $Y \in \mathcal{P}(A)$, $X = Y'$ et $Y = X'$. L'ensemble X est appelé extension et Y l'intension du concept formel.*

Exemple Dans le contexte de la figure 2.2, $(15,dgh)$ est un concept alors que $(15,d)$ ne l'est pas.

Hiérarchie conceptuelle

La famille des concepts formels d'un contexte $\mathcal{K} = (O,A,I)$, notée $\mathcal{C}_\mathcal{K}$, est partiellement ordonnée par la relation d'inclusion $\subseteq$ entre intensions/extensions.

Définition 14. *Étant donnés (X_1,Y_1) et (X_2,Y_2) deux concepts formels d'un contexte $\mathcal{K} = (O,A,I)$. Le concept (X_1,Y_1) est un* ***sous-concept*** *de (X_2,Y_2) (d'une manière équivalente, (X_2,Y_2) est un* ***super-concept*** *de (X_1,Y_1)) si et seulement si $X_1 \subseteq X_2$ (d'une manière équivalente, $Y_2 \subseteq Y_1$). On utilise le signe $\leq$ pour exprimer cette relation. On obtient par conséquent:*

$$(X_1,Y_1) \leq_\mathcal{K} (X_2,Y_2) \Leftrightarrow X_1 \subseteq X_2 (Y_2 \subseteq Y_1).$$

Définition 15 (idéal et filtre générés par un concept). *L'idéal et le filtre d'un concept c, notés respectivement par $\downarrow c$ et $\uparrow c$ sont définis comme suit:*

$$\downarrow c = \{\underline{c} \in \mathcal{C}_\mathcal{K} / \underline{c} \leq c\} \qquad \uparrow c = \{\bar{c} \in \mathcal{C}_\mathcal{K} / \bar{c} \geq c\}$$

Par exemple, dans la figure 2.2, $\downarrow c_{\#1} = \{c_{\#2}, c_{\#7}, c_{\#9}\}$ et $\uparrow c_{\#6} = \{c_{\#3}, c_{\#4}, c_{\#0}\}$

Propriété 3. *L'ensemble de tous les concepts $\mathcal{C}_\mathcal{K}$ muni de la relation d'ordre $\leq_\mathcal{K}$ forme un treillis complet où les infimums et les supremums sont définies comme suit:*

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

Ce treillis est noté $\mathcal{L} = \langle \mathcal{C}_\mathcal{K}, \leq_\mathcal{K} \rangle$ et est appelé le **treillis de Galois** [BM70] (ou **treillis de concepts** [Wil82]) du contexte $\mathcal{K} = (O,A,I)$.

Définition 16 (Concept objet - Concept attribut). *Étant donné le treillis $\mathcal{L} = \langle \mathcal{C}_\mathcal{K}, \leq_\mathcal{K} \rangle$, alors:*

- *à chaque objet $o \in O$, le concept-objet, noté $\mu(o)$, est donné par (o'',o'),*
- *à chaque attribut $a \in A$, le concept-attribut, noté $\nu(a)(o)$, est donné par (a',a'').*

Par exemple (voir figure 2.2 à droite), le concept-objet de l'objet 5 est le concept 8. Le concept-attribut de g est le concept 3.

Propriété 4. *La fonction μ (resp ν) associe à un objet o (respectivement un attribut a) le plus petit (en termes de son extention) (respectivement le plus grand) concept du treillis ayant l'objet o (respectivement l'attribut a) dans son extention (respectivement intention).*

Treillis dual Si $\mathcal{K} = (O,A,I)$ est un contexte, alors $\mathcal{K}^{-1} = (A,O,I^{-1})$ est aussi un contexte. En appliquant le principe de la dualité, si on échange les rôles des objets et des attributs, on obtient alors le treillis dual.

Après avoir présenté les treillis de Galois, nous pouvons nous attarder à leur construction. La section suivante décrit deux approches dédiées à cette tâche.

2.3 Construction du treillis de Galois

Plusieurs algorithmes de construction des treillis ont été proposés et ont fait l'objet d'études comparatives. Toutefois, il faut tenir compte des deux facteurs suivants: la construction de la relation de couverture entre les concepts et la technique de mise à jour des données d'entrée utilisée (incrémentale ou par lots).

- **Les méthodes par lots (*batch*)**: les données sont connues à l'avance. Par conséquent, l'ajout d'un nouvel objet/attribut au contexte nécessite une construction de nouveau de tout le treillis.
- **Les méthodes incrémentales**: les données sont ajoutées au fur et à mesure. Dans ce cas on a plutôt une maintenance ou mise à jour du treillis et non pas une re-construction. En effet, lors de l'ajout d'un nouvel objet/attribut, la structure du treillis sera localement mise à jour; certains concepts seront touchés alors qu'une partie du treillis serait inchangée.

L'approche incrémentale présente plusieurs avantages entre autres:

- Dans les bases de données, les transactions sont ajoutées au fur et à mesure. Vu ce caratère dynamique des données, il peut être plus efficace de maintenir le treillis déjà construit que de le reconstruire chaque fois qu'une modification est survenue sur la base correspondante.
- Pouvoir ajouter et supprimer des attributs à une vue partielle sur la base offre aussi la possibilité d'explorer les données et/ou la navigation dans le treillis correspondant. Le but est souvent de comprendre l'impact d'un attribut sur le reste des attributs ou la structure conceptuelle dérivée.

2.3.1 Approche par lots

Nous décrivons, dans cette section, NEXT-CLOSURE [Gan84] un algorithme classique pour le calcul des motifs fermés.

Étant donné le contexte $\mathcal{K} = (O,A,I)$, on voudrait avoir tous les motifs fermés. Ainsi, il faudrait calculer pour chaque $Y \subset A$ sa fermeture soit Y''. Le problème qui se pose est que

plusieurs sous ensembles peuvent avoir la même fermeture (à savoir les générateurs d'un motif fermé que nous étudierons dans le chapitre suivant). NEXT-CLOSURE calcule les fermés en réduisant le nombre des parties de A auxquelles est appliqué l'opérateur de fermeture ($''$) qui, lorsque calculé sur la table entière peut être très coûteux.

2.3.1.1 Bases théoriques de Next-Closure

Soient un ensemble $M = \{1,2,\cdots,m\}$ trié suivant l'ordre total naturel et A, B deux sous ensembles de M. On définit l'ordre *lectique* noté $\leq_l$ entre A et B comme suit:

$$A \leq_l B \Leftrightarrow max(A - B) \leq max(B - A)$$

NEXT-CLOSURE énumère les fermés dans l'ordre et parcourt l'espace 2^A en appliquant l'opérateur de fermeture. Dans le cadre de ce travail, nous reprenons la version de l'algorithme décrite dans [Tao00].

Définition 17. *Soit le contexte $\mathcal{K} = (O,A,I)$. Pour $Y \subseteq A$ et $i \in A - Y$, on définit:*

$$inc(Y,i) = Y \cup \{i\} - \{1,2,\cdots,i-1\}$$

Par exemple, en considérant $Y = efh$, nous avons $inc(Y,a) = aefh$, $inc(Y,g) = gh$.

Propriété 5. *Si i est le plus petit élément de $A - Y$ tel que $max((inc(Y,i))'' - Y) \leq i$, alors $(inc(Y,i))''$ est le plus petit ensemble fermé plus grand (au sens de l'ordre lectique) que Y.*

La propriété 5 peut être transformé en un pseudo code pour donner l'algorithme 1.

On commence par le plus petit fermé à savoir $Y = \emptyset''$ (ligne 7). Les itérations suivantes (lignes 9 - ligne 20) permettent de calculer tous les fermés comme suit:

Étape 1: Le premier élément i dans $S = A - Y$, dans l'ordre lectique, est déterminé (ligne 12) et enlevé de S (ligne14). Ensuite, l'ensemble $I = inc(Y,i)$ qui contient i est plus grand que Y est déterminé (ligne 15) et la fermeture de I est calculée (ligne 16). Ensuite, on teste si le dernier élément de l'ensemble $(I'' - Y)$ et plus petit ou égal à i (ligne 17). Si cette condition est vérifiée, alors I'' est ajoutée à l'ensemble des fermés (ligne 19).

Sinon, on choisit le i suivant dans $S = A - Y$ et on répète l'étape 1 jusqu'à la condition (ligne 17) soit vérifiée.

L'algorithme s'arrête quand tous les ensembles fermés ont été calculés, c.à.d, lorsque $Y = A$.

```
1: NEXT-CLOSURE(In: K = (O,A,I) : formal context ; Out: Fh : set of closed itemsets.)
2:
3: Y, S, I, I'' ⊆ A
4: i  formal Attribute
5: h  closure operator
6:
7: Y ← h(∅)
8: Fh ← {Y}
9: while (Y ≠ A) do
10:    lower ← false
11:    S ← A − Y
12:    while (¬lower) do
13:       i ← GET-FIRSTELEMENT(S)
14:       S ← S − {i}
15:       I ← inc(Y,i)
16:       I'' ← h(I)
17:       if (GET-LASTELEMENT(I'' − Y) ≤ i) then
18:          lower ← true
19:          Fh ← Fh ∪ I''
20:          Y ← I''
21: return Fh
```

Algorithm 1: L'algorithme de Ganter pour le calcul des fermés

Exemple Considérons le contexte de la figure 2.2. On souhaite claculer les fermés sur A. Notons que pour cet exemple, $\emptyset'' = \emptyset$. L'exécution de NEXT-CLOSURE sur ce contexte est donnée par la table 2.1. À l'étape 28 de l'algorithme, par exemple, on a le fermé $Y = dgh$. Il y a donc cinq candidats possibles pour i à savoir a,b,c,e, ou f: $inc(Y,a) = adgh$, $inc(Y,b) = bdgh$, $inc(Y,c) = cdgh$, $inc(Y,e) = egh$, $inc(Y,f) = fgh$. On a $(adgh)'' = abcdefgh$ et ainsi $max((inc(Y,a))'' - Y) = f \not\leq = a$, donc $abcdefgh$ n'est pas le successeur immédiat (au sens lectique) de dgh. De même,

$(bdgh)'' = bcdgh$ et ainsi $max((inc(Y,a))'' - Y) = c \not\leq = b$, donc $bcdgh$ n'est pas le successeur immédiat de dgh. Par contre, $(cdgh)'' = bcdgh$ et ainsi $max((inc(Y,a))'' - Y) = c \leq = c$; $bcdgh$ est donc le successeur immédiat de dgh.

L'avantage de NEXT-CLOSURE est qu'il n'a besoin que du dernier fermé et de la relation binaire. Donc il n'a pas besoin de mémoriser tous les fermés déjà générés.

2.3.1.2 Construction du diagramme de Hasse

Comme mentionné au début du chapitre, certains algorithmes construisent juste la famille des intentions des concepts alors que d'autres construisent le treillis des concepts et la relation de couverture entre les concepts. Afin de pouvoir étudier le coût de la construction de la relation de couverture entre les concepts, nous proposons d'implémenter une version de l'algorithme NEXT-CLOSURE qui génère le treillis des concepts et le diagramme de Hasse qui y est associé. Pour ces fins, nous adoptons la méthode de Nourine et Raynaud [NR99]. Cette méthode est basée sur la propriété 6 pour déterminer les prédecesseurs immédiats des concepts.

Propriété 6. : $c \prec \underline{c} \iff |intent(\underline{c})| - |intent(c)| = |\{a \in A \mid \{a\}' \cap extent(\underline{c}) = extent(c)\}|$

Cette propriété stipule qu'un concept c est un prédecesseur immédiat du concept $\underline{c}$ si et seulement si c est généré un nombre de fois égal à la taille de la différence des intentions des deux concepts. On sait que a' est l'extention du concept attribut $\nu(a) = (a',a'')$ et que l'intersection de deux extentions est une extention (celle du concept infimum des deux concepts d'après la section précédente). Ainsi, la méthode consiste à calculer l'infimum du concept $\underline{c}$ et chacun des $\nu(a)$ pour tout attribut a n'appartenant pas à son intention. On associe à chaque concept un compteur afin de tester le nombre de fois qu'il soit généré.

Le pseudo-code de cette méthode est présenté dans l'Algorithm 2. Pour chaque concept c et pour chaque attribut a n'appartenant pas à son intention, on calcule l'extention formée par l'intersection des deux extentions a' et $extent(\underline{c})$ (ligne 8) et on cherche le concept correspondant à l'extention générée (ligne 9). On incrémente le compteur associé au concept respectif (ligne 10) et on applique la propriété 6 (lignes 13-14). Une fois tous les prédecesseurs trouvés, on

Itération	Y	i	$inc(Y,i)$	$(inc(Y,i))''$	$max((inc(Y,i))'' - Y) \leq i$
1	$\emptyset$	a	a	ac	non
2		b	b	bcdgh	non
3		c	c	c	oui
4	c	a	ac	ac	oui
5	ac	b	bc	bcdgh	non
6		d	d	dgh	non
7		e	e	efh	non
8		f	f	efh	non
9		g	g	g	oui
10	g	a	ag	abcdefgh	non
11		b	bg	bcdgh	non
12		c	cg	bcdgh	non
13		d	dg	dgh	non
14		e	eg	defgh	non
15		f	fg	defgh	non
16		h	h	h	oui
17	h	a	ah	abcdefgh	non
18		b	bh	bcdgh	non
19		c	ch	bcdgh	non
20		d	dh	dgh	non
21		e	eh	efh	non
22		f	fh	efh	oui
23	efh	a	aefh	abcdefgh	non
24		b	befh	abcdefgh	non
25		c	cefh	abcdefgh	non
26		d	defh	defgh	non
27		g	gh	dgh	oui
28	dgh	a	adgh	abcdefgh	non
29		b	bdgh	bcdgh	non
30		c	cdgh	bcdgh	oui
31	bcdgh	a	abcdgh	abcdefgh	non
32		e	egh	defgh	non
33		f	fgh	defgh	oui
34	defgh	a	adefgh	abcdefgh	non
35		b	bdefgh	abcdefgh	non
36		c	cdefgh	abcdefgh	oui

TAB. 2.1 – *Exécution de* NEXT-CLOSURE *sur le contexte* $\mathcal{K}$ *de la figure 2.2*

remet à zéro les compteurs (lignes 15-16) et on passe au concept suivant.

```
1: NR-GEN-HASSEDIAGRAM(In: C_K, conceptAttribute: array of ν(a) ; Out: Hasse diagram.)
2:
3: conceptCandidates : set
4: e : extent
5: candidate : concept
6: for all (c_i ∈ C_K) do
7:    conceptCandidates ← ∅
8:    for all (a ∈ A − intent(c_i)) do
9:       e ← Extent(conceptAttribute[a]) ∩ extent(c_i)
10:      candidate ← LOOKUP(e,C)
11:      candidate.counter + +
12:      if (candidate.counter = 1) then
13:         conceptCandidates ← conceptCandidates ∪ {candidat}
14:      if (candidate.counter + |intent(c_i)| = |intent(candidate)|) then
15:         NEW-LINK(c_i,candidate)
16:   for all (c_j ∈ ConceptCandidates) do
17:      c_j.counter ← 0
```

Algorithm 2: Construction du diagramme de Hasse associé au treillis de Galois

Exemple Considérons une version de NEXT-CLOSURE qui génère la famille des concepts au lieu des motifs fermés (seulement les intentions). Il suffit de garder les extentions lors du calcul de la fermeture (ligne 11 de l'algorithm 1) pour le calcul de l'ensemble $\mathcal{C}_{\mathcal{K}}$.

Les concepts tel qu'ils ont été générés par NEXT-CLOSURE sont donnés dans la table 2.2:

conceptId	*extent*	*intent*	*conceptId*	*extent*	*intent*	*conceptId*	*extent*	*intent*
$c_{\#0}$	12345	$\emptyset$	$c_{\#1}$	12	c	$c_{\#2}$	2	ac
$c_{\#3}$	145	g	$c_{\#4}$	135	h	$c_{\#5}$	35	efh
$c_{\#6}$	15	dgh	$c_{\#7}$	1	$bcdgh$	$c_{\#8}$	5	$defgh$
$c_{\#9}$	$\emptyset$	$abcdefgh$						

TAB. 2.2 – *Les concepts générés par* NEXT-CLOSURE

La table 2.3 montre à titre d'exemple le calcul des prédecesseurs immédiats du concept $c_{\#4}$ qui sont $c_{\#5}$ et $c_{\#6}$.

att	$\nu(att)$	$\nu(att).extent \cap c_{\#4}.extent$	$candidat$	$candidat.compteur$	$candidat.compteur + \lvert intent(c_{\#4})\rvert = \lvert intent(candidat)\rvert$	$Candidats$
a	$c_{\#2}$	$\emptyset$	$c_{\#9}$	1	*non*	$\{c_{\#9}\}$
b	$c_{\#7}$	1	$c_{\#7}$	1	*non*	$\{c_{\#9},c_{\#7}\}$
c	$c_{\#1}$	1	$c_{\#7}$	2	*non*	$\{c_{\#9},c_{\#7}\}$
d	$c_{\#6}$	15	$c_{\#6}$	1	*non*	$\{c_{\#9},c_{\#7},c_{\#6}\}$
e	$c_{\#5}$	35	$c_{\#5}$	1	*non*	$\{c_{\#9},c_{\#7},c_{\#6},c_{\#5}\}$
f	$c_{\#5}$	35	$c_{\#5}$	2	*oui*	$\{c_{\#9},c_{\#7},c_{\#6},c_{\#5}\}$
g	$c_{\#3}$	15	$c_{\#6}$	2	*oui*	$\{c_{\#9},c_{\#7},c_{\#6},c_{\#5}\}$

TAB. 2.3 – *Génération des prédecesseurs de* $c_{\#4}$

2.3.1.3 Complexité

Sur l'exemple présenté plus tôt dans ce chapitre, NEXT-CLOSURE nécessite 36 itérations au lieu de $2^8 = 256$. La complexité de NEXT-CLOSURE est calculée de la manière suivante:

Soient $m = |A|$, $n = |O|$ et l le nombre de tous les fermés calculés . Il n'y a pas de moyen d'estimer l à partir de m et n. Comme l peut être exponentiel en m ou n, alors on le met dans l'estimation du côut ou alternativement on parle de coût par concept.

Pour calculer à partir d'un fermé son prochain fermé (au sens lectique), on peut parcourir au plus m attributs pour calculer la fermeture. D'après sa définition, l'opérateur de fermeture $''$ a la complexité $n \cdot m$ et ainsi la complexité de trouver un fermé est $m \cdot (n \cdot m) = n \cdot m^2$. Par conséquent, la complexité de NEXT-CLOSURE est de l'ordre: $O(l \cdot n \cdot m^2)$.

Pour la découverte de la relation de couvertures entre les concepts, la complexité de NR-GEN-HASSEDIAGRAM est de l'ordre de $O(l \cdot m \cdot n)$. En effet, dans l'algorithme 2, pour chaque concept, on parcourt au plus m attributs pour calculer les concepts-attributs, pour chacun

de ces concepts, on calcule l'intersection des extentions qui est de l'ordre de n, le LOOKUP du concept à partir de son extention est aussi de l'ordre de n et les autres instructions sont constantes (dans $O(1)$).

2.3.2 Approche incrémentale pour la maintenance du treillis de Galois

Les méthodes incrémentales construisent le treillis $\mathcal{L}$ en commençant par $\mathcal{L}_0 = \langle\{(\emptyset,A)\},\emptyset\rangle$ (respectivement $\mathcal{L}_0 = \langle\{(O,\emptyset)\},\emptyset\rangle$) et progressivement incorporent un nouveau objet o_i (respectivement attribut a_i) au treillis $\mathcal{L}_{i-1}$ qui correspond au contexte $\mathcal{K}_{i-1} = (\{o_1,...,o_{i-1}\},A,I)$ (respectivement $\mathcal{K}_{i-1} = (O,\{a_1,...,a_{i-1}\},I)$). A chaque étape, des opérations de mise à jour de la structure sont exécutées [GMA95]. Les algorithmes incrémentaux décrits dans les sections suivantes construisent dans une même procédure les concepts et leurs couvertures.

2.3.2.1 Incrémentalité par objet

Soient les deux contextes $\mathcal{K}_1$ et $\mathcal{K}_2$ obtenus, respectivement, avant et après l'insertion du nouvel objet o. $\mathcal{K}_1 = (O_1,A,I_1)$ et $\mathcal{K}_2 = (O_2,A,I_2)$ avec $O_2 = O_1 \cup \{o\}$ et $I_2 = I_1 \cup (\{o\} \times o')$.

L'approche incrémentale qui a été initialement décrite dans [GMA95] repose sur la propriété suivante:

Propriété 7. *: La famille des intentions des concepts, notée par C^a, est fermée par l'intersection.*

En d'autres termes, si $c = (X,Y)$ est un concept de $\mathcal{L}_1$ alors $Y \cap o'$ est fermée et correspond à l'intention d'un concept du nouveau treillis $\mathcal{L}_2$. Ainsi, l'intégration du nouvel objet o vise, principalement, le calcul de tous les concepts dont l'intension est l'intersection de o' avec les intensions des concepts existants dans $\mathcal{L}_1$. On distingue deux types d'intersection: les intersections existantes et celles qui sont nouvelles. De ce fait, les concepts sont répartis en trois groupes: les concepts *modifiés*, dénotés $M(o)$, qui sont les fruits des intersections existantes, les concepts *inchangés*, dénotés $U(o)$, qui demeurent complètement inchangés, et les concepts

géniteurs, dénotés $G(o)$, qui donnent naissance à de nouveaux concepts, dénotés $N(o)$ (nouvelles intersections).

Dorénavant, nous adopterons les notations suivantes: Les ensembles G_1, U_1 et M_1 font référence au treillis $\mathcal{L}_1$ associé au contexte $\mathcal{K}_1$. Les ensembles G_2, U_2, M_2 et N_2 font référence au treillis $\mathcal{L}_2$ associé au context $\mathcal{K}_2$. De même, les fonctions dérivées ($'$) seront indexées par $i = 1{,}2$ suivant le contexte $\mathcal{K}_i$.

Fondements théoriques On cherche à:

- identifier les concepts géniteurs afin de construire les nouveaux concepts,
- identifier les concepts modifiés pour mettre à jour leurs extentions,
- trouver les couvertures des nouveaux concepts.

Dans cette section, on rappelle les aspects théoriques définis dans [VMGM02] et [VHM03] nécessaires pour la compréhension des algorithmes présentés ultérieurement dans ce chapitre.

La figure 2.3 montre les deux treillis $\mathcal{L}_1$ et $\mathcal{L}_2$ obtenus avant et après l'ajout de l'objet 5 au contexte de notre exemple.

Définitions

Soit o l'objet ajouté. Pour tout concept c de $\mathcal{L}_1$, on a $intent(c) \cap o^2$ est l'intention du concept formé par $c \vee \mu(o)$. Comme plusieurs concepts peuvent générer la même intersection, on définit les fonctions $\mathcal{Q}_i$: $\mathcal{C}_i \rightarrow \mathcal{P}(A)$ par $\mathcal{Q}_i(c) = intent(c) \cap o^2$.

La fonction $\mathcal{Q}_i$ induit une relation d'équivalence sur $\mathcal{C}_i$, et ainsi la classe d'équivalence de c dans $\mathcal{Q}_i$, notée $[c]_{\mathcal{Q}_i}$, est donnée par:

$$[c]_{\mathcal{Q}_i} = \{\bar{c} \in \mathcal{C}_i \mid \mathcal{Q}_i(c) = \mathcal{Q}_i(\bar{c})\}$$

Par exemple, lors de l'ajout de l'objet (5,$defgh$), $\mathcal{Q}_1(c_{\#4}) = \mathcal{Q}_1(c_{\#2}) = \mathcal{Q}_1(c_{\#3}) = \emptyset$. (voir figure 2.3 à gauche) et par conséquent $[c_{\#4}]_{\mathcal{Q}_1} = \{c_{\#4}, c_{\#2}, c_{\#3}\}$.

De plus, lors de l'ajout d'un nouvel objet, on devrait pouvoir relier les concepts des deux treillis (ancien et nouveau). Pour cela, les fonctions de correspondance entre $\mathcal{L}_1$ et $\mathcal{L}_2$ sont

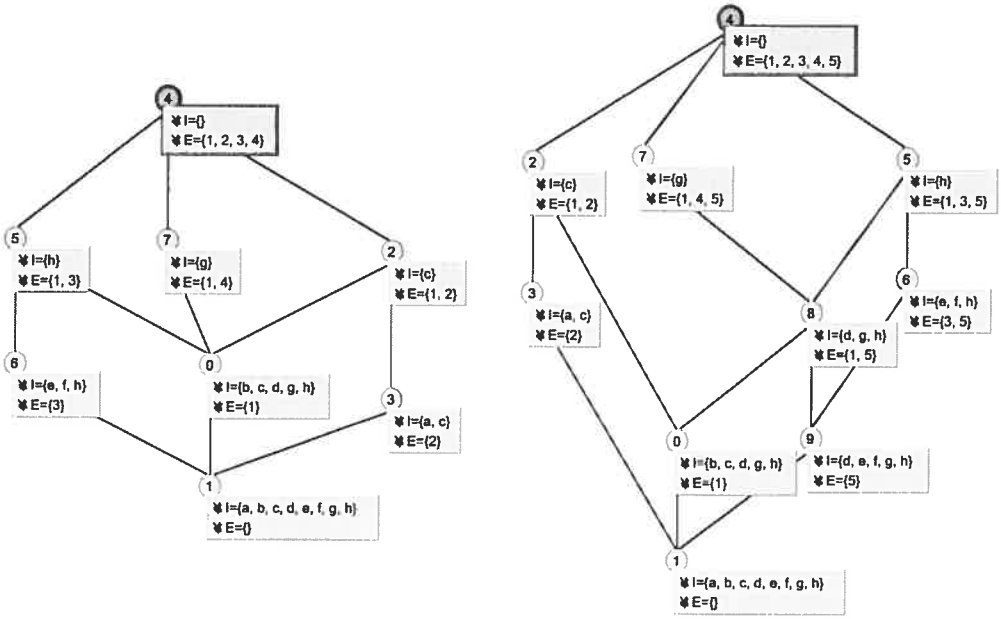

FIG. 2.3 – ***Gauche:*** *le treillis* $\mathcal{L}_1$ *associé à* $\mathcal{K}_1$ $=(O_1 = \{1,2,3,4\}, A = \{a,b,\cdots,h\}, I_1)$ ***Droite:*** *le treillis* $\mathcal{L}_2$ *associé à* $\mathcal{K}_2$ $=(O_2 = O_1 \cup \{5\},\ A = \{a,b,\cdots,h\}, I_1 \cup \{5\} \times \{5\}')$

définies comme suit:

- σ_o: $C_1 \rightarrow C_2$, $\sigma_o(X,Y) = (Y^2,Y)$: retourne le concept dans $\mathcal{L}_2$ ayant la même intention que le concept c de $\mathcal{L}_1$. Par exemple, $\sigma_o(c_{\#7}) = c_{\#7}$ (dans la figure 2.3).
- γ_o: $C_2 \rightarrow C_1$, $\gamma_o(X,Y) = (X_1,X_1^1)$ tel que $X_1 = X - \{o\}$: retourne pour un concept c de $\mathcal{L}_2$ le concept dans $\mathcal{L}_1$ ayant l'extention égale à celle de c modulo o. Par exemple, $\gamma_o(c_{\#8})$ $= c_{\#0}$ (dans la figure 2.3),
- χ_o^2: $C_1 \rightarrow C_2$, $\chi_o^2(X,Y) = (Y_1^2,Y_1)$, tel que $Y_1 = Y \cap \{o\}^2$: retourne pour un concept c de $\mathcal{L}_1$ l'élément maximal de sa classe d'équivalence dans $\mathcal{L}_2$.

A partir de ces fonctions, on peut définir formellement les géniteurs, les modifiés et les nouveaux concepts.

Pour un nouveau concept dans $\mathcal{L}_2$, si on enlève le nouvel objet o de son extention, le concept résultant est le concept géniteur dans $\mathcal{L}_1$. Ainsi, l'ensemble $N_2(o)$ est défini par:

$$N_2(o) = \{c = (X,Y) \mid c \in C_2,\ o \in X; (X - \{o\})^{22} = X - \{o\}\}$$

A titre d'exemple, $c_{\#8} = (15,dgh) \in N_2(5)$. $X - 5 = 1$ et $(X-5)^{22} = ((X-5)^2)^2 = (bcdgh)^2 = 1$.

Pour un concept modifié dans $\mathcal{L}_2$, on sait que son intention est la même que celle de son correspondant dans $\mathcal{L}_1$ mais que son extension se distingue de celui-ci par l'ajout de o. Ainsi, les ensembles des concepts modifiés dans $\mathcal{L}_1$ et $\mathcal{L}_2$ sont définis comme suit:

- $M_2(o) = \{c = (X,Y) | c \in \mathcal{C}_2, o \in X; (X - \{o\})^1 = Y\}$,
- $M_1(o) = \{c = (X,Y) | c \in \mathcal{C}_1, \exists\ \bar{c} \in M_2(o); c = \gamma_o(\bar{c})\}$.

Si on considère, à titre d'exemple, les concepts $c_{\#7} = (145,g)$ dans $\mathcal{L}_2$ et $c_{\#7} = (14,g)$ dans $\mathcal{L}_1$. On remarque que $(145-5)^1 = (14)^1 = g$ et $\gamma_o(c_{\#7}) = c_{\#7}$.

Pour un concept géniteur, d'après la définition de $N_2(o)$, on a $X - o$ est un fermé et par suite le concept associé à cette extention est le géniteur dans $\mathcal{L}_2$. Ainsi, les ensembles des concepts géniteurs dans $\mathcal{L}_1$ et $\mathcal{L}_2$ sont:

- $G_1(o) = \{c = (X,Y) | Y \not\subseteq \{o\}^2, o \in X; (X \cup \{o\})^{22} = X \cup \{o\}\}$,
- $G_2(o) = \{c = (X,Y) | o \notin X; (X \cup \{o\})^{22} = X \cup \{o\}\}$.

Par exemple, le concept $c_{\#0} = (1,bcdgh)$ dans $\mathcal{L}_2$ est un concept géniteur. $(\{1 \cup 5\})^{22} = (15)^{22} = (dgh)^2 = 15$. De plus, le concept $c_{\#0} = (1,bcdgh)$ dans $\mathcal{L}_1$ appartient à l'ensemble $G_1(5)$ puisque $bcdgh \not\subseteq defgh$ et $(15)^{22} = 15$.

Détection des géniteurs et modifiés

Maintenant qu'on sait comment définir les concepts géniteurs, modifiés et nouveaux, nous pouvons les identifier (géniteurs et modifiés).

Propriété 8. *: L'ensemble des géniteurs $G_1(o)$ dans $\mathcal{L}_1$ est donné par:*

$$G_1(o) = \{c = (X,Y) | Y \not\subseteq \{o\}^2; Y = (Y \cap \{o\}^2)^{11}\}$$

Preuve.: En effet, $Y \cap \{o\}^2 = (X \cup o)^2$ (d'après la formule $(A \cup B)' = A' \cap B'$). En outre, $\gamma_o(X \cup \{o\}, Y \cap \{o\}^2) = (X,X^1) = (X,Y)$. Donc $(Y \cap \{o\}^2)^{11} = Y^{11} = Y$. ■

La propriété suivante montre que $G_1(o)$ et $M_1(o)$ sont les concepts maximaux (en terme de taille d'intention) de leurs classes d'équivalence.

Propriété 9. : *Soit $E(o)$ l'ensemble de tous les éléments maximaux des classes d'équivalence dans $\mathcal{L}_1$. On a alors: $E(o) = G_1(o) \cup M_1(o)$*

La Propriété 9 permet l'identification des concepts géniteurs et modifiés.

Relation de couverture dans $\mathcal{L}_2$

L'étape suivante consiste à définir les couvertures des nouveaux concepts dans $\mathcal{L}_2$.

On définit la relation $\leq_{/\mathcal{Q}}$ entre les classes d'équivalence $[]_{\mathcal{Q}}$ comme suit:

$$[c_1]_{\mathcal{Q}} \leq_{/\mathcal{Q}} [c_2]_{\mathcal{Q}} \iff \mathcal{Q}(c_2) \subseteq \mathcal{Q}(c_1)$$

Propriété 10. *La structure partiellement ordonné $\mathcal{L}_{/\mathcal{Q}} = \langle \mathcal{C}_{/\mathcal{Q}}, \leq_{/\mathcal{Q}} \rangle$ est un treillis complet.*

La propriété suivante montre que si la classe d'un géniteur c_2 appartient à la couverture supérieure d'une autre classe ayant c_1 comme géniteur, il existe alors un concept de la couverture supérieure de c_1 qui appartient à $[c_2]_{\mathcal{Q}}$.

Propriété 11. : $\forall c_1, c_2 \in G_1(o)$, *si* $[c_1]_{\mathcal{Q}} \leq_{/\mathcal{Q}} [c_2]_{\mathcal{Q}}$ *alors* $\exists\ \bar{c} \in [c_2]_{\mathcal{Q}}$ *telque* $c_1 \prec \bar{c}$.

Ainsi, on peut trouver les couvertures supérieures d'un nouveau concept en considérant la couverture supérieure de son géniteur et en prenant les minimas de leurs images par χ^2.

Propriété 12. : $\forall c \in N_2(o)$, $\bar{c} \in \mathcal{C}_1$, $c \prec_2 \bar{c} \iff \bar{c} \in min(\{\chi_o(\hat{c}) \ / \ \gamma_o(c) \prec \hat{c}\})$.

On sait que le nouveau concept appartient à la couverture supérieure de son géniteur. La propriété suivante montre que le seul concept de $\sigma(\mathcal{C}_1)$ qui est couvert par un nouveau concept dans $\mathcal{L}_2$ est l'homologue de son géniteur.

Propriété 13. : $\forall \bar{c} \in \mathcal{C}^2 - N_2(o)$, $\bar{c} \leq^2 c \iff \bar{c} \leq^2 \gamma_o(\sigma_o(c))$.

en regroupant toutes ses propriétés, le calcul des couvertures supérieures dans le nouveau treillis $\mathcal{L}_2$ est donnée par:

Propriété 14. : $$\prec_2 = \{(c_1,c_2) | (\gamma_o(c_1), \gamma_o(c_2)) \in \prec_1\} \cup \{(\sigma_o(\gamma_o(c)), c) | c \in N_2(o)\}$$

$$\cup \{(c,\bar{c}) | c \in N_2(o), \bar{c} \in \ min(\{\chi_o(\hat{c}) \mid \gamma_o(c) \prec_1 \hat{c}\}\}$$
$$- \{(c_1,c_2)/\gamma_o(c_1) \in G_1(o), \gamma_o(c_2) \in M_1(o)\}.$$

Ainsi, l'ordre dans $\mathcal{L}_2$ se résume comme suit:

- Pour un nouveau concept c_n, son géniteur c_g est un prédecesseur immédiat et tous ses autres prédecesseurs, s'ils existent, sont des nouveaux concepts.
- Tous les successeurs immédiats d'un nouveau concept c_n sont des concepts nouveaux et/ou modifiés.
- Les prédecesseurs immédiats d'un concept modifié c_m diffèrent de ceux de $\gamma(c_m)$ dans $\mathcal{L}_1$ par:
  (i) l'ajout d'un ensemble de nouveaux concepts,
  (ii) la supression de tous les concepts appartenant à $G_2(o)$.

En somme, la mise à jour du treillis par ajout d'un nouvel objet consiste à:

- Identifier les classes d'équivalence,
- Trouver l'élément maximal de chaque classe et découvrir son statut (géniteur ou modifié),
- Créer les nouveaux concepts et mettre à jour l'extention des concepts modifiés.
- Générer les nouvelles couvertures (les liens des nouveaux noeuds du diagramme de Hasse).

La méthode incrémentale Nous décrirons la méthode proposée dans [VHM03]. Comme le calcul des classes est une tâche coûteuse, et qu'on a besoin seulement de l'élément maximal de chaque classe, il faut trouver une méthode efficace qui permet d'identifier cet élément sans avoir à construire la classe en entier. La propriété suivante montre que pour un concept c qui n'est pas maximal dans sa classe, il existe toujours un successeur de c appartenant aussi à $[c]_{\mathcal{Q}}$. Ainsi, comme le parcours du treillis se fait de manière descendante, le statut de c peut être généré en comparant c avec tous ses successeurs (ces derniers sont visités avant c).

Propriété 15. *:* $\forall\ c, \bar{c}, \underline{c} \in \mathcal{C}, \underline{c} \leq c \leq \bar{c}$ *et* $[\bar{c}]_{\mathcal{Q}} = [\underline{c}]_{\mathcal{Q}} \Rightarrow [\bar{c}]_{\mathcal{Q}} = [c]_{\mathcal{Q}}$

En ce qui concerne le calcul des couvertures supérieures d'un nouveau concept c notées

$Cov^u(c)$[5], on sait que si $\bar{c_1} = (\bar{X_1},\bar{Y_1})$ et $\bar{c_2} = (\bar{X_2},\bar{Y_2})$ appartiennent à $Cov^u(c)$ alors $\bar{X_1} \cap \bar{X_2} = X$.

Donc, la méthode de calcul des couvertures supérieures du nouveau concept c consiste à prendre les minima des images des couvertures supérieures du géniteur par la valeur de $\mathcal{Q}_2$. $\gamma_o(c)$ $(\{\chi^2(\bar{c})/\bar{c} \in Cov^u(\gamma_o(c))\})$ et de calculer l'intersection de tous les couples possibles. Une méthode plus efficace pour le calcul des intersections serait de calculer l'intersection de chaque extention d'un candidat avec l'extention, appelée *face*, formée par la réunion de celles qui ont été considérés avec les candidats précédents (d'après $A_1 \cap (A_2 \cup A_3... \cup A_n) = (A_1 \cap A_2) \cup (A_1 \cap A_3)... \cup (A_1 \cap A_n)$). Pour chaque *candidat*, on calcule l'intersection des deux extentions *extent(candidat)* et *face*. Si cette intersection est égale à l'extention du concept c alors *candidat* est un successeur de c et la valeur de *face* est mise à jour (*face* = *face* $\cup$ *extent(candidat)*). Toutefois, si *candidat* est modifié, il faut supprimer le lien entre $\gamma_o(c)$ et *candidat* (d'après la Propriété 14).

En s'appuyant sur toutes ces propriétés, on définit une méthode(voir Algorithm 3) pour la mise à jour d'un treillis suite à l'ajout d'un nouvel objet.

On commence par trier les concepts par ordre croissant de l'intention afin de permettre le parcours descendant (ligne 3). La ligne 5 est une implémentation de la propriété 15. Elle permet de calculer les intersections générées par les successeurs du concept c et d'appliquer la fontion ARGMAX pour trouver la plus grande valeur (en taille de l'intention). Si cette valeur maximale est égale à $\mathcal{Q}(c)$ alors ce dernier n'est pas maximal dans sa classe et par conséquent c est un concept inchangé. Par contre, si les deux valeurs sont différentes alors c est le concept maximal de sa classe. Dans ce cas, on détermine son statut:

- si la taille de $\mathcal{Q}(c)$ est égale à celle de *intent*(c) (ligne 7) alors c est un concept modifié (*intent*$(c) \subseteq o'$). Dans ce cas, l'extention de c est mise à jour (ligne 8), c est ajouté à l'ensemble des concepts modifiés (ligne 9) et l'élément maximal de la classe est mis à jour

5. u pour indiquer upper en anglais

```
1: procedure ADD-OBJECT(In/Out: 𝓛 = ⟨𝒞, ≤⟩ a lattice; In: o an object)
2:
3: SORT(𝒞)
4: for all c in 𝒞 do
5:    new-max ← ARGMAX({|𝒬(c̄)| | c̄ ∈ Cov^u(c)})
6:    if |𝒬(c)| ≠ |𝒬(new-max)| then
7:       if |𝒬(c)| = |Intent(c)| then
8:          Extent(c) ← Extent(c) ∪ {o}      {c is modified}
9:          M(o) ← M(o) ∪ {c}
10:         new-max ← c
11:      else
12:         ĉ ← NEW-CONCEPT(Extent(c) ∪ {o},𝒬(c))      {c is genitor}
13:         Candidates ← {ChiPlus(c̄) | c̄ ∈ Cov^u(c)}
14:         for all c̄ in MIN-CLOSED(Candidates) do
15:            NEW-LINK(ĉ,c̄)
16:            if c̄ ∈ M(o) then
17:               DROP-LINK(c,c̄)
18:         new-max ← ĉ
19:         𝓛 ← 𝓛 ∪ {ĉ}
20:   ChiPlus(c) ← new-max
```

Algorithm 3: Mise à jour du treillis suite à l'ajout d'un nouvel objet au contexte.

(ligne 10) (deux tâches nécessaires pour la mise à jour de l'ordre dans le nouveau treillis).

- Sinon, c est un concept géniteur. Premièrement, le nouveau concept $\hat{c}$ est généré (ligne 12), deuxièmement, les nouveaux liens sont crées (ligne 13-17) selon la propriété 14, et finalement le maximal de la classe est mis à jour (ligne 18).

Exemple: Illustrons le fonctionnement de l'algorithm 3 sur le contexte de la figure 2.2. Soient $\mathcal{K}_1$ le contexte formé par $(O = \{1,\ldots,4\}, A = \{a,\ldots,h\}, I)$ et $\mathcal{K}_2 = (O \cup \{5\}, A, I \cup \{5\} \times \{5\}')$. Le tableau suivant montre pour chaque concept c la valeur de $\mathcal{Q}(c)$, le statut[6] de c et le nouveau concept maximal de $[c]_{\mathcal{Q}}$.

c	$\mathcal{Q}(c)$	$\chi^+(c)$	Statut	c	$\mathcal{Q}(c)$	$\chi^+(c)$	Statut
$c_{\#4}$	$\emptyset$	$c_{\#4}$	m	$c_{\#5}$	h	$c_{\#5}$	m
$c_{\#7}$	g	$c_{\#7}$	m	$c_{\#2}$	$\emptyset$	$c_{\#4}$	i
$c_{\#3}$	$\emptyset$	$c_{\#4}$	i	$c_{\#6}$	efh	$c_{\#6}$	m
$c_{\#0}$	dgh	$c_{\#8}$	g	$c_{\#3}$	$\emptyset$	$c_{\#4}$	i
$c_{\#1}$	$defgh$	$c_{\#9}$	g				

Prenons à titre d'exemple le calcul des nouveaux liens du concept $c_{\#9}$. L'ensemble des successeurs de son géniteur ($c_{\#1}$) est donné par: $\{c_{\#3}, c_{\#0}, c_{\#6}\}$ et par conséquent $Candidates = \{c_{\#6}, c_{\#8}, c_{\#4}\}$. La table 2.4 montre les tâches effectuées pour cette mise à jour de l'ordre.

2.3.2.2 Incrémentalité par attribut

En appliquant le principe de la dualité, la maintenance du treillis par ajout d'un nouvel attribut a se fait en appliquant les duales des propriétés démontrées dans la section précédente. Ainsi, l'intégration du nouvel attribut a consiste alors à calculer tous les concepts dont l'extension est l'intersection de a' avec les extensions des concepts existants dans $\mathcal{L}_1$. Les concepts sont réparties en trois catégories: *inchangés*, *modifiés* et *géniteurs*. Dans le cas de l'incrémentalité

6. m = modifié, g = géniteur et i = inchangé.

face	c_i	$extent(c_i) \cap face$	$extent(c_i) \cap face = extent(c_{\#9})$	Opérations de MAJ de l'ordre
5	$c_{\#6}$	5	OUI	Nouveau lien entre $c_{\#9}$ *et* $c_{\#6}$- Suppression du lien entre $c_{\#6}$ *etc*$_{\#1}$
35	$c_{\#8}$	5	OUI	Nouveau lien entre $c_{\#9}$ *et* $c_{\#8}$
135	$c_{\#4}$	135	NON	

TAB. 2.4 – *Calcul de la couverture du concept* $c_{\#9}$ *dans* $\mathcal{L}_2$

attribut, $G_1(a)$ et $M_1(a)$ sont les concepts minimaux (en terme de taille d'extention) de leurs classes d'équivalence.

Dans ce qui suit, on donne les définitions qui seront utiles pour le Chapitre 5.

Soient $\mathcal{L}_1$, $\mathcal{L}_2$ les deux treillis obtenus, respectivement, avant et après l'insertion du nouvel attribut a et deux concepts $c_1 = (X_1,Y_1)$ et $c_2 = (X_2,Y_2)$. Les fonctions $\mathcal{R}_i$ sont définies comme suit: $\mathcal{R}_i$: $\mathcal{C}_i \rightarrow 2^O$ tel que $\mathcal{R}_i(c) = extent(c) \cap a^2$. La classe d'équivalence de c dans $\mathcal{R}_i$, notée $[c]_{R_i}$, est donnée par: $[c]_{R_i} = \{\bar{c} \in \mathcal{C}_i \ / \mathcal{R}_i(c) = \mathcal{R}_i(\bar{c})\}$.

Les fonctions de correspondance entre $\mathcal{L}_1$ et $\mathcal{L}_2$ sont données par:

- σ_a: $C_1 \rightarrow C_2$, $\sigma_a(X,Y) = (X,X^2)$: retourne le concept dans $\mathcal{L}_2$ ayant la même extention que le concept c de $\mathcal{L}_1$.
- γ_a: $C_2 \rightarrow C_1$, $\gamma_a(X,Y) = (Y_1^1,Y_1)$ tel que $Y_1 = Y - \{a\}$: retourne pour un concept c de $\mathcal{L}_2$ le concept dans $\mathcal{L}_1$ ayant l'intention égale à celle de c modulo a.
- χ_a^2: $C_1 \rightarrow C_2$, $\chi^2(X,Y) = (X_1,X_1^2)$ tel que $X_1 = X \cap a^2$: retourne pour un concept dans $\mathcal{L}_1$ l'élément minimal de sa classe d'équivalence $[]_{\mathcal{R}}$ dans $\mathcal{L}_2$.

Le pseudo-code de la maintenance du treillis suite à l'ajout d'un nouvel attribut est présenté dans l'algorithm 4.

Exemple Considérons cette fois l'ajout de l'attribut h au contexte de la figure 2.2.

```
1: procedure ADD-ATTRIBUTE(In/Out: L = ⟨C, ≤⟩ a lattice; In: a an attribute)
2:
3: SORT(C)
4: for all c in C do
5:    new-min ← ARGMIN({|R(c̄)| | c̄ ∈ Cov^u(c)})
6:    if |R(c)| ≠ |R(new-min)| then
7:       if |R(c)| = |Extent(c)| then
8:          Intent(c) ← Intent(c) ∪ {a}      {c is modified}
9:          M(a) ← M(a) ∪ {c}
10:         new-min ← c
11:      else
12:         ĉ ← NEW-CONCEPT(R(c),Intent(c) ∪ {a})      {c is genitor}
13:         Candidates ← {ChiPlus(c̄) | c̄ ∈ Cov^u(c)}
14:         for all c̄ in MIN-CLOSED(Candidates) do
15:            NEW-LINK(ĉ,c̄)
16:            if c̄ ∈ M(a) then
17:               DROP-LINK(c,c̄)
18:         new-min ← ĉ
19:         L ← L ∪ {ĉ}
20:   ChiPlus(c) ← new-min
```

Algorithm 4: Mise à jour du treillis suite à l'ajout d'un nouvel attribut au contexte.

Le tableau suivant montre pour chaque concept c la valeur de $\mathcal{R}(c)$, le statut de c et le nouveau concept minimal de $[c]_{\mathcal{R}}$.

c	$\mathcal{R}(c)$	$\chi^+(c)$	Statut	c	$\mathcal{R}(c)$	$\chi^+(c)$	Statut
$c_{\#1}$	135	$c_{\#1}$	g	$c_{\#4}$	1	$c_{\#3}$	i
$c_{\#8}$	15	$c_{\#5}$	i	$c_{\#7}$	35	$c_{\#7}$	m
$c_{\#0}$	$\emptyset$	$c_{\#2}$	i	$c_{\#5}$	15	$c_{\#5}$	m
$c_{\#3}$	1	$c_{\#3}$	m	$c_{\#6}$	5	$c_{\#6}$	m
$c_{\#2}$	$\emptyset$	$c_{\#2}$	m				

Comme il y a un seul nouveau concept ($c_{\#9}$), décrivons la trace de ses nouvelles couvertures.

L'ensemble des succcesseurs de $\gamma_a(c_{\#9})$ est formé par :$\{c_{\#4},c_{\#8},c_{\#7}\}$ et ainsi l'ensemble des prédécesseurs candidats de $c_{\#9}$ est égal à $\{c_{\#7},c_{\#5},c_{\#3}\}$. Pour $face = h$ ($intent(c_{\#9})$) $intent(c_{\#7}) \cap face = h = intent(c_{\#9})$. Donc, $c_{\#7}$ est un prédéccesseur de $c_{\#9}$ et $face$ devient efh. En outre, comme $c_{\#7}$ est un concept modifié alors le lien entre $c_{\#1}$ et $c_{\#7}$ sera supprimé. $intent(c_{\#5}) \cap face = h = intent(c_{\#9})$ et par conséquent $c_{\#5}$ est aussi un prédéccesseur de $c_{\#9}$. Et enfin, pour $face = defgh$, $intent(c_{\#3}) \cap face = dgh \neq intent(c_{\#9})$.

2.3.2.3 Complexité

Nous allons expliquer la complexité de l'algorithme 3 décrite dans [VHM03]. Soient $l = |\mathcal{L}_2|$, $m = |A|$, $n = |O|$ et $\Delta l = |C_2| - |C_1|$. Le tri des concepts (ligne 3) est linéaire dans la taille du treillis car en effet il s'agit d'une comparaison entre les tailles des intentions qui sont bornées par m. Pour la boucle (ligne 4-20), nous avons un facteur de l. Le calcul de $\mathcal{Q}(c)$ est borné par m, car il s'agit d'intersection d'intention. La couverture supérieure d'un concept $c = (X,Y)$ contient un nombre de concepts qui est borné par $n - |X|$, en effet un concept $c_i = (X_i,Y_i)$ sera un successeur de c si $X_i = X \cup O_i$ tel que $O_i \subseteq O - X$ et ainsi le calcul de la complexité de l'opération ARGMAX($\{|\mathcal{Q}(\bar{c})| \mid \bar{c} \in Cov^u(c)\}$) appartient à $O(m+n)$. En ce qui concerne les concepts modifiés, les lignes (8-10) sont linéaires en fonction de la taille de $M(o)$ et peuvent être négligés. L'autre opération coûteuse est la partie lié aux concepts géniteurs. En effet, le

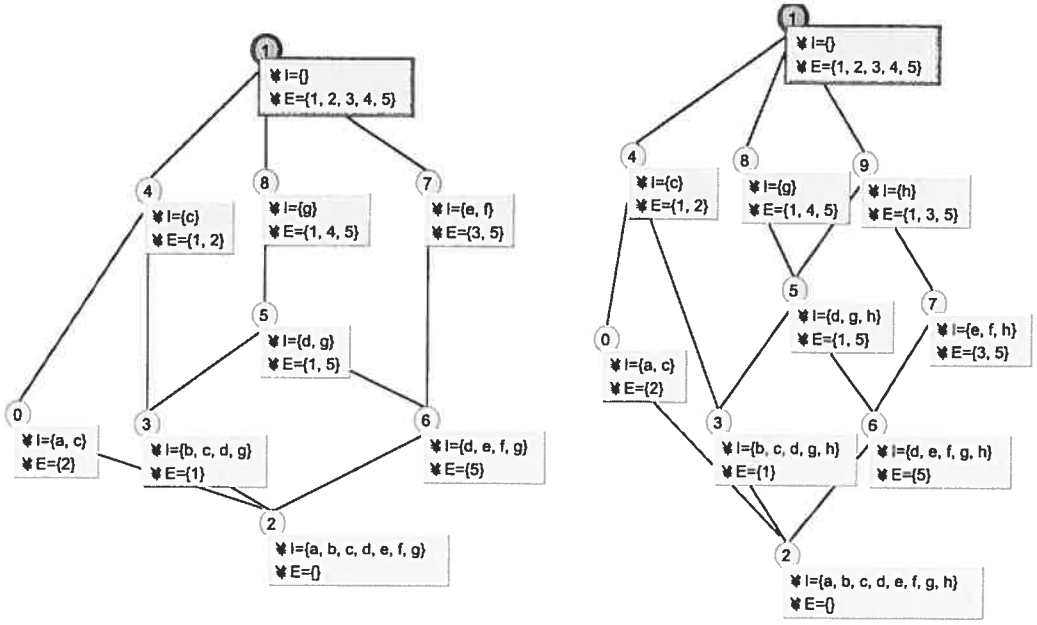

FIG. 2.4 – ***Gauche:*** *le treillis* $\mathcal{L}_1$ *associé à* $\mathcal{K}_1$ $=(O = \{1,2,3,4,5\}, A_1 = \{a,b,\cdots,g\}, I_1)$. ***Droite:*** *le treillis* $\mathcal{L}_2$ *associé à* $\mathcal{K}_2$ $=(O = \{1,2,3,4,5\}, A_2 = \{a,b,\cdots,h\}, I_2)$.

nombre de géniteurs est Δl. La mise à jour de l'ordre, effectuée pour chaque géniteur, a un coût de $O(n^2)$ qui peut s'expliquer par le parcours de la couverture supérieure dont on a vu ci-dessus que sa taille est en $O(n)$ et pour chaque élément de la couverture, un calcul d'intersection sur les extentions également en $O(n)$. Les autres opérations sont à temps constant. Donc, la partie liée aux géniteurs a un coût global en $O(\Delta l \cdot n^2)$. Le coût total de l'ajout d'un objet est en $O(l \cdot (m+n) + \Delta l \cdot n^2)$.

En somme, la complexité de l'algorithme pour l'ajout de tous les objets est de l'ordre de $O(nl \cdot (m+n))$.

Avec le même raisonnement, la complexité de l'algorithme 4 est donnée par:

Le tri des concepts est linéaire dans la taille du treillis car en effet il s'agit d'une comparaison entre les tailles des intentions qui sont bornées par m. Le calcul de $\mathcal{R}(c)$ est borné par n, car il s'agit d'intersection d'extention. La couverture supérieure d'un concept $c = (X,Y)$ contient un nombre de concepts qui est borné par $m - |Y|$, en effet un concept $c_i = (X_i,Y_i)$ sera un

successeur de c si $Y_i = U \cup A_i$ tel que $A_i \subseteq A - Y$ et ainsi le calcul de la complexité de l'opération ARGMIN($\{|\mathcal{R}(\bar{c})| \mid \bar{c} \in Cov^u(c)\}$) appartient à $O(n+m)$. La mise à jour de l'ordre, effectuée pour chaque géniteur, a un coût de $O(m^2)$ qui peut s'expliquer par le parcours de la couverture supérieure ($O(m)$) et pour chaque élément de la couverture, un calcul d'intersection sur les intentions également en $O(m)$. Les autres opérations étant négligeables à celle là (elles sont à temps constant). Donc, la partie liée aux géniteurs a un coût global en $O(\Delta l \cdot n^2)$. Le coût total de l'ajout d'un objet est en $O(l \cdot (n+m) + \Delta l \cdot m^2)$. La complexité de l'algorithme pour l'ajout de tous les attributs est de l'ordre de $O(ml \cdot (m+n))$.

2.4 Conclusion partielle

Dans ce chapitre, nous avons présenté les fondements des treillis de Galois (ou de concepts) et nous avons décrit deux approches pour leur construction. L'algorithme de traitement par lots pour la génération des motifs fermés NEXT-CLOSURE est un algorithme efficace vu sa complexité théorique et jumelé avec la méthode de Nourine et Reynaud pour le calcul de la relation de couverture entre les concepts, sa complexité du pire cas ne change pas. De son côté, l'approche incrémentale par ajout d'un nouvel attribut présente la meilleure complexité ce qui nous pousse à l'exploiter afin d'améliorer le temps de réponse d'un système de ARM par application de l'AFC.

Chapitre 3

Fouille de données et Analyse Formelle de Concepts

Les techniques d'extraction de connaissances (knowledge discovery) occupent actuellement une place de plus en plus importante, tant dans la littérature que dans le champ des applications. Un problème classique de ce champ, celui de l'**extraction des règles d'association (ARM[1])** a attiré l'attention des chercheurs et plusieurs travaux ont été effectués dans ce domaine. Dans ce chapitre, nous présentons l'approche classique de l'ARM et les travaux de l'AFC dans la découverte des règles d'association.

3.1 Extraction de Connaissances à partir de Données

Le développement des outils informatiques a provoqué un véritable déluge d'informations stockées dans de grandes bases de données scientifiques, économiques, financières, médicales, etc. La croissance permanente de ces données a produit un besoin de développer des techniques et d'outils qui peuvent intelligemment et automatiquement transformer ces données en informations et connaissances utiles. Par conséquent, l'exploitation de données est devenue un

1. le terme en anglais Association Rule Mining

secteur de recherche d'une extrême importance. L'Extraction de Connaissances à partir de Données (ECD) est un processus d'extraction des informations implicites, précédemment inconnues et potentiellement utiles à partir des données [PsF91]. L'ECD a emprunté ses outils à divers domaines, tels les systèmes de base de données, l'intelligence artificielle, les bases de données spatiales et la visualisation de données. Plusieurs types de connaissance peuvent être découverts dans une base de données notamment les règles d'association.

3.2 Extraction des règles d'association

Développée dans le cadre de l'extraction des connaissances à partir des transactions de vente afin de faire des prédictions sur les achats [AS94], l'ARM est appliquée, de nos jours, à tout domaine s'intéressant à regrouper des produits ou des services. Étant donné une base de données de transactions d'achats, il est souhaitable de découvrir les associations importantes entre les articles tel que la cooccurence de certains articles dans une transaction implique la présence d'autres articles dans la même transaction.

3.2.1 Quelques définitions de l'ARM

Soient $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ un ensemble de m attributs distincts, $\mathcal{T}$ une transaction contenant un ensemble d'attributs (motifs) dans $\mathcal{I}$ et $\mathcal{D}$ l'ensemble formé par les transactions $\mathcal{T}$. Le support d'un motif X (noté par $supp(X)$) mesure la fréquence de X dans la base $\mathcal{D}$. Une règle d'association entre deux motifs disjoints X et Y de la forme $r : X \Rightarrow Y$ avec X, Y des sous-ensembles de $\mathcal{I}$. Etant donné une règle $r : X \Rightarrow Y$, on définit:

- son support (noté par $supp(r)$) comme étant $supp(X \cup Y)$. Il mesure la probabilité que X et Y soient simultanément présents dans $\mathcal{D}$.
- sa confiance (notée par $conf(r)$) comme étant $supp(X \cup Y)$ / $supp(X)$. Elle mesure la probabilité que Y se produise lorsque X est présent dans $\mathcal{D}$.

La confiance dénote la force de l'implication et le support indique les fréquences d'occurrence des motifs dans la règle.

3.2.2 Problématique

Etant donnés les seuils minimaux de support, appelé *minsupp* et de confiance, appelé *minconf*, définis par l'usager, la découverte des règles d'association se décompose aux deux sous-problèmes suivants:

- Extraire les motifs fréquents, c.à.d ceux ayant un support $\geq$ *minsupp*,
- A partir des motifs extraits précédemment, générer les règles fortes, c.à.d celles ayant une confiance $\geq$ *minconf*.

Agrawal et *al* ont proposé dans [AS94] le premier algorithme efficace pour l'ARM dans les bases transactionnelles nommé APRIORI. Comme les algorithmes batch pour la construction des motifs fermés sont une amélioration d'APRIORI et utilisent ainsi ses procédures, nous présentons cet algorithme dans la section suivante.

3.2.2.1 Découverte des motifs fréquents

Un k-motif est défini comme un motif de taille k. APRIORI est un algorithme conçu pour la génération des motifs fréquents; c.à.d les motifs de taille k sont générés à partir de ceux de taille $(k-1)$. Pour cela, le treillis des motifs est exploré en largeur d'abord. On sait qu'un k-motif est formé par la jointure de deux $(k-1)$-motifs ayant un même préfixe de taille $(k-2)$. Pour diminuer le nombre des candidats lors du calcul de la jointure, APRIORI se base sur les deux propriétés suivantes:

Propriété 16. : *Tous les sous-ensembles d'un motif fréquent sont fréquents.*

Propriété 17. : *Tous les sur-ensembles d'un motif infréquent sont infréquents.*

3.2.2.2 Apriori

Les notations suivantes sont utilisées dans le pseudo code:

- C_k: l'ensemble des k-motifs fréquents candidats,
- F_k: l'ensemble des k-motifs fréquents,
- $\bowtie$: jointure.

```
1: APRIORI(In: base D, le support minimal minsupport ; Out: F ensemble des motifs fréquents)
2: F_1 ← {1 − motifsfrequents}
3: for (k ← 2; F_{k−1} ≠ ∅; k + +) do
4:    C_k ← APRIORI-GEN(F_{k−1})
5:    for all (t ∈ D) do
6:       C_t ← SUBSET(C_k,t)
7:       for all (c ∈ C_t) do
8:          c.support + +
9:    F_k ← {c ∈ C_k|c.support ≥ minsupport}
10: F ← ⋃_k F_k
11: return F
```

Algorithm 5: Extraction des motifs fréquents avec APRIORI

```
1: APRIORI-GEN(In: F_{k−1} ; Out: C_k)
2: c motif candidat
3: C_k ← ∅
4: for all (p ∈ F_{k−1}) do
5:    for all (q ∈ F_{k−1}) do
6:       if (p[1] = q[1] ∧ p[2] = q[2] ∧ ··· ∧ p[k − 2] = q[k − 2] ∧ p[k − 1] < q[k − 1]) then
7:          c ← p ⋈ q
8:          if (∀ s ⊂ c | s (k − 1) − motifs  s ∈ F_{k−1}) then
9:             C_k ← C_k ∪ {c}
10: return C_k
```

Algorithm 6: Génération des motifs candidats avec APRIORI-GEN

Durant la première itération (ligne 2), la base $\mathcal{D}$ est parcourue afin de trouver tous les attributs (1-motifs) fréquents. La boucle (ligne 3-9) consiste à générer à chaque niveau k les k-motifs

fréquents. Pour ce faire, la première étape vise à calculer les k-candidats (ligne 4). La fonction APRIORI-GEN génère les candidats de taille k en joignant deux $k-1$ motifs ayant un $k-2$ préfixe en commun. Les candidats ayant tous leurs sous ensembles de taille $k-1$ fréquents seront gardés, les autres seront éliminés (en vertu de la propriété 17). Une fois C_k- l'ensemble des candidats de taille k- calculé, la base de transactions est parcourue d'une manière séquentielle afin de déterminer le support de chaque candidat. La fonction SUBSET(C_k,t) recherche parmi les candidats de C_k ceux qui sont contenus dans la transaction t. Le support de chacun de ces candidats est incrémenté (ligne 8). Parmi tous les candidats seuls les fréquents sont gardés dans l'ensemble F_k (ligne 9).

Illustrons le fonctionnement de l'algorithme APRIORI sur le contexte de notre exemple avec un $minsupp = 30\%$. (voir les détails dans la figure 3.1).

Étape 1: Les 1-motifs candidats sont générés d'abord. Un premier balayage de la base sert à calculer leur support. Les motifs infréquents sont élagués pour produire l'ensemble des 1-motifs fréquents.

Étape 2: APRIORI-GEN est appliquée sur l'ensemble des 1-motifs fréquents pour générer l'ensemble des 2-motifs fréquents candidats. Un second balayage de la base permet de calculer les supports des candidats afin de produire l'ensemble des 2-motifs fréquents.

On répète l'étape 2 avec l'ensemble des 2-motifs fréquents pour générer les 3-motifs fréquents.

On répète l'étape 2 avec l'ensemble des 3-motifs fréquents. Comme l'ensemble des 4-motifs candidats est vide, alors l'algorithme s'arrête.

Le treillis des motifs fréquents est présenté dans la figure 3.2.

3.2.2.3 Génération des règles d'association

Pour générer les règles d'association, on considère l'ensemble F des motifs fréquents calculé dans l'étape précédente. Pour chaque motif fréquent l, on calcule tous ses sous ensembles (fréquents d'après la Propriété 16) et on génère toutes les règles fortes de la forme:

$$r : c \rightarrow (l - c) \; telque \; c \subset l$$

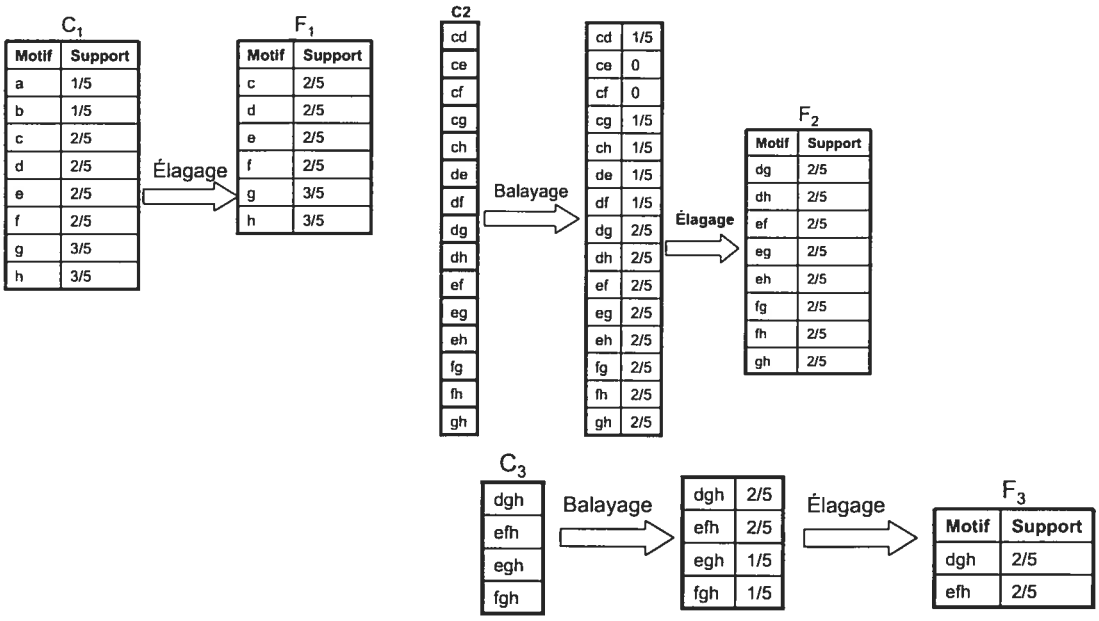

FIG. 3.1 – *Génération des motifs fréquents par application d'*APRIORI

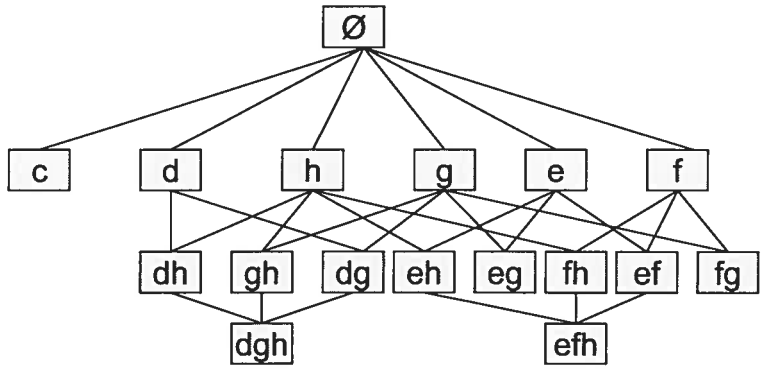

FIG. 3.2 – *Treillis des motifs fréquents*

avec $conf(r) \geq minconf$ ($conf(r) = support(l)/support(c)$).

Dans [AS94], les auteurs proposent une méthode permettant de réduire le nombre d'opérations réalisées pour la génération des règles. Cette méthode est basée sur la propriété suivante:

Propriété 18. : *Soit l un motif fréquent, nous avons :* $\forall c \subset l$, $support(c) \geq support(l)$

Étant donné trois motifs m_1, m_2, m_3 tel que $m_1 \subset m_2 \subset m_3$. D'après la propriété 18, on a : $support(m_1) \geq support(m_2) \geq support(m_3)$ et ainsi la confiance de la règle $r_1 : l_1 \rightarrow (l_3 - l_1)$ est plus petite que celle de la règle $r_2 : l_2 \rightarrow (l_3 - l_2)$. Par conséquent, si la règle r_2 n'est pas valide alors la règle r_1 ne l'est pas aussi.

Comme notre travail n'apporte pas des modifications à la génération des règles d'association, nous ne décrivons pas les algorithmes respectifs mais nous nous concentons sur l'extraction des motifs et sur la génération de prémisses de taille minimale.

3.2.3 Complexité d'Apriori

La génération des motifs fréquents représente la phase la plus coûteuse à cause des deux raisons suivantes:

- Le nombre des itérations effectuées correspond à la taille des plus grands motifs fréquents. Dans les bases fortement correlées, ce nombre est assez grand et par conséquent le temps d'accès à la base pour le calcul des supports devient coûteux.
- Le nombre important des candidats générés dans chaque itération constitue la majeure partie du temps de calcul étant donné que l'algorithme se base sur des manipulations de nature combinatoire de ces candidats pour la détermination des motifs fréquents.

En outre, le nombre de règles générées est exponentiel en la taille des motifs fréquents mais cette phase est directe alors que la première peut être améliorée.

3.2.4 Amélioration d'Apriori

L'approche classique pour la génération des motifs fréquents et la génération des règles d'association présente d'une part un problème d'efficacité pour l'extraction des règles et d'autre part un problème de la pertinence et de l'utilité de l'ensemble des règles générées. Durant les dernières années, plusieurs travaux de recherche ont été menés afin de résoudre ces problèmes. Parmi ces travaux, on trouve des variantes d'`Apriori` comme `Apriori-TID`, `Partition` [SON95], `DIC` [BMUT97], etc. Ces méthodes ont pour but de générer d'une manière efficace les motifs fréquents. Toutefois, le problème des règles pertinentes n'est pas pris en compte. C'est pourquoi d'autres alternatives ont été proposées notamment la génération des motifs fermés fréquents [PBTL99, PHM00, ZH99, Zak00]. Cette approche permet non seulement d'améliorer l'efficacité de l'extraction mais aussi de réduire considérablement le nombre de règles redondantes.

3.3 Application de l'AFC à l'ARM

D'après sa définition, un motif fermé est un ensemble maximal d'attributs communs à un ensemble d'objets. En d'autres termes, il n'a aucun sur-ensemble ayant le même support. Un motif fermé X est fréquent si pour un seuil minimal de support *minsupp*, $supp(X) \geq minsupp$. L'intérêt de la génération de motifs fermés est basé sur la propriété suivante:

Propriété 19. *: Pour tout motif* X, $supp(X) = supp(X'')$.

La propriété 19 montre que si l'on dispose des motifs fermés fréquents, alors on dispose de tous les motifs fréquents et leurs supports [Pas00] et par suite on peut extraire les règles d'association. En effet, en énumérant tous les sous-ensembles des motifs fermés fréquents, on retrouve l'ensemble des motifs fréquents.

Toutefois, le problème des règles redondantes n'est pas résolu. Pour cela, une nouvelle approche est définie pour l'extraction des règles d'association [Pas00]. Le problème de l'extraction

des règles d'association est réduit ainsi aux deux sous-problèmes suivants:

- Déterminer l'ensemble des motifs fermés fréquents,
- Extraire les règles d'association à partir des motifs générés dans la phase précédente.

l'ARM basée sur l'AFC consiste à calculer les motifs fermés fréquents dans le cas de la construction du treillis des motifs fermés ou les concepts fréquents dans le cas de la construction du treillis des concepts. À savoir que le support d'un concept $c = (X,Y)$ est donné par: $support(c) = |X| / |O|$.

Comme nous avons juste besoin des motifs fermés (ou concepts) fréquents, la construction du treillis complet des motifs fermés (ou des concepts) devient ainsi initule. La section suivante traite ce problème en appliquant la notion de demi-treillis (iceberg) au cas des motifs fermés fréquents.

3.3.1 Icebergs

Définition 18 (demi-treillis). *Un sup-demi-treillis (resp inf-demi- treillis) est un ensemble S ordonné dans lequel tout couple d'éléments (a,b) a un suprémum $a \vee b$ (resp un infimum $a \wedge b$)*

Définition 19 (demi-treillis complet). *Un sup-demi-treillis (resp inf-demi- treillis) S est dit sup-complet (resp inf-complet) si toute partie de S admet un suprémum (resp un infimum) dans S.*

Les sups-complets de Galois, connus aussi sous le nom d'*icebergs* sont utilisés en classification conceptuelle comme une méthode de visualisation de bases de données de grande taille et un moyen de représentation condensée des motifs fréquents [STB+02].

Etant donné un seuil minimal $\alpha \in [0,1]$, le treillis complet $\mathcal{L} = \langle \mathcal{C}, \leq_{\mathcal{K}} \rangle$ peut être coupé horizontalement en deux parties: la supérieure notée $\bar{\mathcal{L}}^{\alpha} = \langle \bar{\mathcal{C}}^{\alpha} \leq_{\mathcal{K}} \rangle$ par les concepts fréquents, et l'inférieure notée $\underline{\mathcal{L}}^{\alpha} = \langle \underline{\mathcal{C}}^{\alpha}, \leq_{\mathcal{K}} \rangle$ par les concepts infréquents,.

Formellement:

Définition 20 (iceberg). *Etant donné $\alpha \in [0,1]$, le sous-ordre $\bar{\mathcal{L}}^{\alpha}$ d'un treillis complet $\mathcal{L}$ tel*

que $(\forall\ c \in \mathcal{L},\ c \in \bar{\mathcal{L}}^{\alpha} \Leftrightarrow support(c) \geq \alpha)$ *est appelé le* α*-iceberg.*

Propriété 20. $\bar{\mathcal{L}}^{\alpha}$ *est un sup-complet.*

A l'instar du chapitre précédent, nous étudions les deux approches pour la construction/maintenance des icebergs, à savoir l'approche par lots et celle incrémentale. Comme la maintenance des icebergs fait partie de notre contribution, elle fera l'objet du chapitre suivant.

La figure 3.3 montre à titre d'exemple l'iceberg associé au contexte $\mathcal{K}$ du chapitre 2 pour un seuil $\alpha = 0.3$

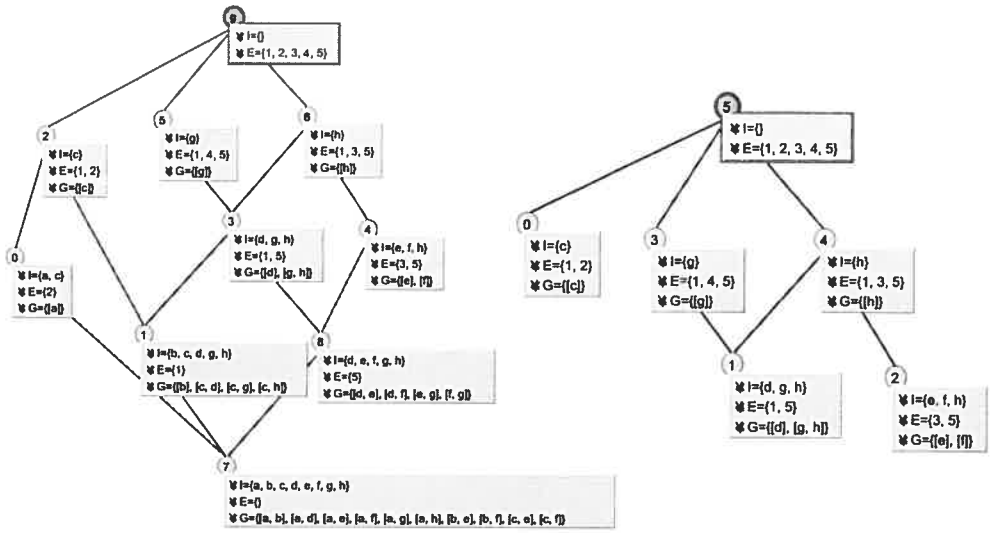

FIG. 3.3 – ***Gauche:*** *le treillis* $\mathcal{L}$ *dérivé du contexte* $\mathcal{K}$ ***Droite:*** *l'iceberg* $\mathcal{L}^{0.3}$ *dérivé du contexte* $\mathcal{K}$

3.3.2 Approche par lots pour la construction d'un iceberg

Dans cette section, nous décrirons TITANIC [STB+02], un algorithme récent classé parmi les algorithmes itératifs les plus performants. Comme ses ancêtres CLOSE et ACLOSE [Pas00], TITANIC génère les motifs fermés fréquents (FCI[2]) en calculant la fermeture des générateurs de ces motifs. Avant d'aller plus loin, définissons les générateurs.

2. Frequent Closed Itemsets terme anglais pour désigner les motifs fermés fréquents.

Définition 21. *Un générateur G d'un motif fermé X est le sous ensemble minimal de X (selon la relation d'inclusion) tel que $G'' = X$.*
L'ensemble des générateurs d'un concept $c = (X,Y)$ noté $Gen(c)$ est défini comme suit:

$$Gen(c) = \{G \subseteq Y \ / \ G'' = Y \text{ et } \nexists \ F \subset G \mid F'' = Y\}.$$

Dans l'iceberg de la figure 3.3, le concept $c_{\#1} = (15,dgh)$ a deux générateurs minimaux à savoir $Gen(c_{\#1}) = \{d,gh\}$.

L'algorithme CLOSE calcule à chaque itération k la fermeture de chaque k-générateur et son support. Pour chaque k-générateur fréquent, sa fermeture est rajoutée à l'ensemble des motifs fermés fréquents. L'ensemble des $(k+1)$-générateurs est formé en joignant deux k générateurs fréquents ayant un préfixe commun de taille $(k-1)$ (à la APRIORI). CLOSE est très couteux, surtout pour les bases éparses, à cause du nombre d'accès au contexte. En effet, à chaque itération un parcours du contexte est effectué afin de calculer les supports et fermetures des générateurs. On rappelle qu'une fermeture correspond à l'intersection de tous les objets dans lequel le motif (générateur) apparait. ACLOSE a été proposé afin d'améliorer les performances de CLOSE. ACLOSE ne calcule pas les fermetures des motifs durant les itérations mais lors d'un seul balayage à la fin des itérations. Il reste coûteux à cause du calcul des supports. Pour cela, nous cherchons un algorithme ayant pour but d'optimiser:

- le nombre de calcul des fermetures,
- le nombre de calcul des supports.

3.3.3 Titanic - Algorithme itératif pour la génération des FCI

Soit le contexte $\mathcal{K}$ =(O,A,I). TITANIC consiste à calculer les fermetures des motifs à partir de leurs supports. La propriété suivante montre la relation entre les fermetures et les supports des motifs.

Propriété 21. *: Soit $X \subset A$. $X" = X \cup \{m \in A - X$ tel que $support(X) = support(X \cup \{m\})\}$*

Par exemple, en appliquant la propriété 21 on obtient $(gh)'' = dgh$. En effet, pour $m = a$, $support(agh) = 0/5$ pour $m \in \{b,c,e,f\}$, $support(gh \cup \{m\}) = 1/5$ et pour $m = d$, $support(dgh) = 2/5 = support(gh)$.

Etant donné deux motifs X, Y , $X'' = Y''$ définit une relation d'équivalence notée θ. La classe d'équivalence $[X]_\theta$ de X dans θ est donnée par: $[X]_\theta = \{Y \in A,\ X'' = Y''\}$

Si X et Y sont deux motifs appartenant à la même classe d'équivalence, alors ils ont le même support. De plus, deux motifs de support égal et dont l'un est inclus dans l'autre font partie de la même classe d'équivalence.

Propriété 22. *:*

1. $X \subset Y \Rightarrow support(X) \geq support(Y)$,
2. $X'' = Y'' \Rightarrow support(X) = support(Y)$,
3. $X \subset Y$ *et* $support(X) = support(Y) \Rightarrow X'' = Y''$.

Puisque l'algorithme procède par niveau, les premiers motifs d'une classe d'équivalence rencontrés sont minimaux au sens de l'inclusion ou plus précisement sont les générateurs minimaux. D'une manière générale, si on a déterminé le support d'un motif X et qu'on trouve plus tard un motif Y appartenant à $[X]_\theta$, il est inutile d'accéder au contexte pour calculer son support. Donc, la tâche consiste d'abord à identifier les générateurs minimaux; ensuite calculer leurs fermetures. Ainsi, on minimize le nombre de calcul des fermetures de tous les motifs au nombre des fermetures des générateurs minimaux.

Identification des générateurs minimaux

La propriété suivante montre qu'un motif est un générateur si et seulement si son support est différent des supports de ses prédecesseurs immédiats. Elle est donnée par:

Propriété 23. *: Soit* $X \subset A$.

X *est un générateur minimal* $\iff$ $support(X) \neq min\{support(X - \{m\}) \text{ — } m \in X\}$.

Par exemple, gh est un générateur minimal puisque $support(gh) = 2/5$ et min{support($gh - \{m\}$) — m $\in X$} = min{support(g),support(h)} = min{3/5,3/5} =3/5.

Calcul de la fermeture d'un générateur minimal

Pour calculer la fermeture d'un générateur minimal, on applique la Propriété 21. Donc, on a besoin des supports de ses successeurs immédiats. Il se peut que ces derniers ne soient pas des générateurs. Si un motif n'est pas un générateur cela veut dire qu'on a déjà trouvé au moins un minimal de sa classe d'équivalence. La propriété suivante montre que si un motif n'est pas un générateur minimal alors son support est égal à la valeur minimale des supports de ceux qui y sont inclus.

Propriété 24. *: Si X n'est pas un générateur minimal alors support(X) = min{support(G), $G \subset X$ et G générateur minimal}.*

Au fait, on n'a pas besoin de parcourir tous ses générateurs minimaux. Il suffit de vérifier les générateurs les plus larges (en taille) car on sait que pour deux motifs A et B, si $A \subseteq B$ alors support(A) $\geq$ support(B).

De plus, cette propriété permet de minimiser le nombre de calcul des supports puisqu'en effet, on a besoin juste de calculer le support des générateurs minimaux, les supports des autres motifs seront déduits de ceux des générateurs minimaux.

Et enfin, on peut montrer la propriété suivante:

Propriété 25. *Si g est un générateur minimal alors $\forall\ f \subset g$, f est un générateur minimal.*

Si g n'est pas un générateur minimal alors $\forall\ f \supset g$, f n'est pas un générateur minimal.

La propriété 25 permet d'appliquer la méthode d'élagage utilisée dans l'algorithme APRIORI.

Le pseudo-code de TITANIC est présenté dans l'algorithme 7.

TITANIC commence par le plus petit générateur, à savoir $\emptyset$ (ligne 4). Durant la première itération, tous les générateurs candidats de taille 1 sont calculés (lignes 6-8). La boucle (lignes 9-18) permet de calculer à chaque niveau k les k-générateurs candidats en appliquant la méthode TITANIC-GEN (ligne 18). Cette méthode n'est autre que APRIORI-GEN(section 3.2.2.2) et consiste à joindre deux $k-1$ générateurs ayant un $k-2$ prefixe commun et à ne garder que les candidats ayant tous leurs $k-1$ sous ensembles fréquents. Une fois l'ensemble des k-générateurs candidats déterminé, le support de chaque candidat est calculé en appliquant

la méthode WEIGH (ligne 10). L'étape suivante consiste à calculer les fermetures des $k-1$ générateurs en appliquant la méthode CLOSURE. Si le support est égal à -1 alors la fermeture est l'ensemble des attributs A. Sinon, on cherche pour chaque attribut m n'appartenant pas au candidat x le support de $X \cup m$. Si ce support est égal à celui de X alors m appartient à la fermeture de X. Enfin, si l'ensemble des générateurs de taille k est généré (ligne 13). Si cet ensemble est vide, alors l'algorithme se termine, sinon on passe à la prochaine itération.

```
1: TITANIC(In: Contexte K ; Out: l'ensemble des motifs fermés fréquents)
2: boolean ok ← true
3: integer k ← 1
4: ∅.s ← 1
5: K_0 ← {∅}
6: K'_0 ← {∅}
7: for all (m ∈ A) do
8:    {m}.p_s ← ∅.s
9: C ← {{m} | m ∈ A}
10: while (C ≠ ∅ and ok) do
11:    WEIGH(C)
12:    for all (X ∈ K_{k-1}) do
13:       X.closure ← CLOSURE(X)
14:    K_k ← {X ∈ C | X.s ≠ X.p_s}
15:    K'_k ← {X ∈ K_k | X.s = 1}
16:    if (||K_k|| = ||K'_k||) then
17:       ok ← false
18:    k++
19:    C ← TITANIC-GEN(K_{k-1})
20: return ⋃_{i=0}^{k-1} {X.closure | X ∈ K_i}
```

Algorithm 7: Génération des FCI avec TITANIC

Exemple Illustrons l'exécution de TITANIC sur le contexte du chapitre 2 avec un *minsupport*= 30%.

Montrons, à titre d'exemple, le calcul de la fermeture de $X = d$. On considère $m \in \{a,b,c,e,f,g,h\}$. Soit Y la fermeture de X; $Y = d$. Pour $m = a$, $X \cup m = ad$ n'est pas un générateur et par suite

```
1: TITANIC-GEN(In: K_{k-1} ; Out: C)
2: X candidat
3: C ← ∅
4: for all (p ∈ K_{k-1}) do
5:    for all (q ∈ K_{k-1}) do
6:       if (p[1] = q[1]∧p[2] = q[2]∧···∧p[k−2] = q[k−2]∧p[k−1] < q[k−1]) and (p[1],...,p[k−1]).s >
         −1 and (p[1],...,p[k − 1]).s > −1 then
7:          X ← p ⋈ q
8:          if (∀ S ⊂ X | S ∈ K_{k-1}) then
9:             C ← C ∪ {X}
10:            X.p_s ← min(X.p_s,S.s)
11: return C
```

Algorithm 8: Génération des candidats avec TITANIC-GEN

```
1: CLOSURE(In: générateur X ; Out: Y la fermeture de X)
2: if (X.s = −1) then
3:    return A
4: Y ← X
5: for all (m ∈ X) do
6:    Y ← Y ∪ (X − {m}).closure
7: for all (m ∈ A − Y) do
8:    if (X ∪ {m} ∈ C) then
9:       s ← (X ∪ {m}).s
10:   else
11:      s ← min{Z.s | Z is a generator and Z ⊆ (X ∪ {m})}
12:   if (s = X.s) then
13:      Y ← Y ∪ {m}
14: return Y
```

Algorithm 9: Fermetures des générateurs avec CLOSURE

$\underline{k = 0}$: $X = \emptyset$ et $X.s = 5/5$.

$\underline{k = 1}$:

X	$X.p\_s$	$X.s$	$X \in \mathcal{K}_K$	X	$X.p\_s$	$X.s$	$X \in \mathcal{K}_K$
a	5/5	-1	OUI	b	5/5	-1	OUI
c	5/5	2/5	OUI	d	5/5	2/5	OUI
e	5/5	2/5	OUI	f	5/5	2/5	OUI
g	5/5	3/5	OUI	h	5/5	3/5	OUI

CLOSURE$(\emptyset) = \emptyset$.

$\underline{k = 2}$:

X	$X.p\_s$	$X.s$	$X \in \mathcal{K}_K$	X	$X.p\_s$	$X.s$	$X \in \mathcal{K}_K$	X	$X.p\_s$	$X.s$	$X \in \mathcal{K}_K$
cd	2/5	-1	OUI	ce	2/5	-1	OUI	cf	2/5	-1	OUI
cg	2/5	-1	OUI	ch	2/5	-1	OUI	de	2/5	-1	OUI
df	2/5	-1	OUI	dg	2/5	2/5	NON	dh	2/5	2/5	NON
ef	2/5	2/5	NON	eg	2/5	2/5	NON	eh	2/5	2/5	NON
fg	2/5	-1	OUI	fh	2/5	2/5	NON	gh	3/5	2/5	OUI

CLOSURE$(c) = c$ CLOSURE$(d) = dgh$ CLOSURE$(e) = efh$

CLOSURE$(f) = efh$ CLOSURE$(g) = g$ CLOSURE$(h) = h$

$\underline{k = 3}$:

$\mathcal{K}_3 = \emptyset$.

CLOSURE$(gh) = dgh$

```
1: WEIGH(In: set X ; Out: X.s le support de G)
2: for all (x ∈ X) do
3:    X.s ← 0
4: for all (o ∈ O) do
5:    for all (X ∈ SUBSETS(o',X)) do
6:       X.s + +
7: for all (x ∈ X) do
8:    x.s ← x.s/|O|
```

Algorithm 10: Calcul des supports avec WEIGH

$s = min\{Z.s \mid Z \text{ generateur et } Z \subseteq (X \cup \{m\})\} = min\{a.s,d.s\} = -1 \neq X.s = 2/5$. De même, pour $m = b$, $X \cup m = bd$ n'est pas un générateur et par suite $s = min\{b.s,d.s\} = -1 \neq X.s$. Pour $m = c$, $X \cup m = cd$ est un générateur et par suite $s = cd.s = -1 \neq X.s = 2/5$. Pour $m = e$, $X \cup m = de$ est un générateur et par suite $s = de.s = -1 \neq X.s = 2/5$. De même, pour $m = f$, $X \cup m = df$ est un générateur et par suite $s = df.s = -1 \neq X.s = 2/5$. Pour $m = g$, $X \cup m = dg$ n'est pas un générateur et par suite $s = min\{d.s,g.s\} = 2/5 = X.s = 2/5$ et ainsi $Y = Y \cup m = dg$. De même, pour $m = h$, $X \cup m = dh$ n'est pas un générateur et par suite $s = min\{d.s,h.s\} = 2/5 = X.s = 2/5$ et ainsi $Y = Y \cup m = dgh$.

3.4 Efficacité de l'AFC appliquée à l'ARM

Dans cette section, nous montrons comment les bases de règles informatives permettent de résoudre le problème des règles redondantes.

En effet, deux règles sont redondantes si elles ont les mêmes valeurs de support et de confiance. Ainsi si deux règles sont redondantes, celle qui a une conséquence plus petite apporte mois d'information. Celle qui a une prémisse plus grande est inutile puisqu'elle a besoin d'un nombre d'attributs plus grand que le necessaire. La prémisse minimale est un générateur du motif fermé car si on prend deux règles d'association redondantes tel que la prémisse de la première notée p_1 est un sous ensemble de celle de la deuxième, notée p_2 nous aurons $support(p_1) = support(p_2)$ (d'après la définition des supports égaux des deux règles). D'après la propriété 22, on a $p_1'' = p_2''$

et dans ce cas $p_1 \in [p_2]$. Enfin, si p_1 est minimal alors il est un générateur de $[p_2]$. Le motif fermé fréquent est la conséquence maximale car si l'on considère deux règles d'association redondantes tel que la conséquence de l'une est incluse dans la conséquence de l'autre alors ces règles sont de la forme: $r_1 : p_1 \rightarrow p_2 - p_1$ et $r_2 : p_1 \rightarrow p_3 - p_1$ tel que $p_2 \subset p_3$. Dans ce cas, on a: $support(p_2) = support(p_3)$ et par conséquent p_2 et p_3 sont dans la même classe d'équivalence. p_3 est maximale s'il est maximal dans sa classe, c.à.d un motif fermé fréquent.

Chapitre 4

Maintenance évolutive d'un treillis iceberg

Dans le chapitre précédent, nous avons défini les icebergs, montré leur application à l'*ARM* et décrit une méthode batch pour leur construction. Ce chapitre est dédié à l'étude de la mise à jour des icebergs après l'ajout d'un nouvel objet/attribut. Ceci consistue une approche alternative de construction qui peut être très intéressante du point de vue des performances.

4.1 Incrémentalité par objet d'un iceberg

Soient $\mathcal{L}_1^\alpha$ et $\mathcal{L}_2^\alpha$ les deux icebergs obtenus respectivement avant et après l'insertion du nouvel objet o. L'application directe de la méthode ADD-OBJECT (section 2.3.2.1) sur $\mathcal{L}_1^\alpha$ mène à un résultat incomplet. En effet, dans le cas de l'iceberg, lors de l'ajout d'un nouvel objet:

- certains concepts de $\mathcal{L}_1^\alpha$ deviennent infréquents dans $\mathcal{L}_2^\alpha$ (des concepts inchangés ayant une fréquence proche de α dans $\mathcal{L}_1^\alpha$),
- inversement, des concepts, que nous appelerons *jumpers*, infréquents dans $\mathcal{L}_1^\alpha$ deviennent fréquents dans $\mathcal{L}_2^\alpha$ (des concepts modifiés et/ou de nouveaux concepts produits par des géniteurs infréquents dans $\mathcal{L}_1^\alpha$).

4.1.1 Fondements théoriques

Soient $\mathbf{H}_2(o)$ l'ensemble des *jumpers* et $\mathbf{V}_2(o)$ l'ensemble des concepts c de $\mathcal{L}_2^\alpha$ ayant leur correspondant $\gamma(c)$ fréquent dans $\mathcal{L}_1^\alpha$.

Définition 22. $\mathbf{H}_2(o) = \{(X,Y) \in \uparrow\mu(o) \mid |X - \{o\}| < \alpha|O_1|, |X| \geq \alpha|O_2|\}$.

$\mathbf{V}_2(o) = \{(X,Y) \in \uparrow\mu(o) \mid |X| \geq \alpha * |O_1| + 1\}$.

D'après sa définition, $\mathbf{H}_2(o)$ peut s'écrire sous la forme:

$\mathbf{H}_2(o) = \{(X,Y) \in \uparrow\mu(o) \mid \alpha.|O_1| + \alpha \leq |X| < \alpha.|O_1| + 1\}$ (*donc* $|X| \in [\alpha.|O_1| + \alpha, \alpha.|O_1| + 1[$)

En effet, $|X - \{o\}| < \alpha|O_1|$ implique $|X| - |\{o\}| < \alpha|O_1|$ qui implique $|X| < \alpha|O_1| + 1$. En outre, $|X| \geq \alpha|O_2|$ implique $|X| \geq \alpha|(O_1 + 1)|$ et par conséquent $|X| \geq \alpha|O_1| + \alpha$.

Comme dans l'intervalle $[\alpha.|O| + \alpha, \alpha.|O| + 1[$, il existe au plus un entier, on peut conclure que tous les *jumpers* ont la même taille d'extention. Ce résultat indique que *les successeurs immédiats d'un jumper appartiennent à l'ensemble* $\mathbf{V}_2(o)$. Ainsi, la méthode d'identification des *jumpers* consiste à calculer les prédecesseurs immédiats des concepts *visibles*. D'après la méthode de Nourine et Raynaud [NR99], on sait que pour calculer les prédecesseurs immédiats du concept $\underline{c}$, on trouve l'infimum du concept $\underline{c}$ et chacun des concepts attributs des attributs n'appartenant pas à $intent(\underline{c})$.

Donc, pour chaque concept c *visible*, et pour chaque attribut a appartenant à l'ensemble des attributs candidats formé par $\{o'\} - intent(c)$, on calcule l'extention du concept infimum $(\nu(a) \wedge c)$, noté $\wedge_a c$ tel que $extent(\wedge_a c) = extent(c) \cap a'$. Si $|extent(\wedge_a c)| \in [\alpha.|O| + \alpha, \alpha.|O| + 1[$ alors le concept $\wedge_a c$ est un *jumper*. L'intention du concept $\wedge_a c$ est égale à $intent(c)$ à laquelle on ajoute tous les attributs $\bar{a}$ vérifiant $extent(c) \cap a' = extent(c) \cap \bar{a}'$. (d'après la Propriété 6)

En outre, comme le calcul des intersections des extentions est coûteux, on cherche à trouver une méthode efficace permettant de réduire le nombre des attributs candidats. En effet, le concept $\nu(a) \wedge c$ peut être soit un concept infréquent, soit un concept *jumper*, soit un concept *visible*.

Comment éviter le calcul des prédécesseurs visibles ? Le but est de trouver un sur-ensemble de l'intention d'un concept visible c pour le remplacer dans l'équation: $attributs = \{o'\} - intent(c)$.

En effet, si les concepts visibles sont triés par ordre décroissant de leur intention, on est sûr de ne pas traiter un concept avant ses prédécesseurs. Ainsi, il suffit de considérer l'ensemble formé par les attributs des prédécesseurs.

Formellement, pour un concept visible c, on définit la trace de c, notée $T^h(c)$, par: $T^h(c) = \{a | a \in Int(\hat{c}), \hat{c} \in \downarrow c \cap \mathbf{V}_2(o) - Cov^l(c) \cap \mathbf{H}_2(o)\}$.

On remarque, toutefois, que cette tâche est incomplète. En effet, on ne considère pas les jumpers qui ont été découverts par les prédécesseurs.

Propriété 26. $T^h(c) = \bigcup_{\hat{c} \in Cov^l(c) \cap \mathbf{V}_2(o)\}} (T^h(\hat{c}) \cup h(\hat{c}))$.

Ainsi, pour un concept visible c, l'ensemble des candidats pour générer les jumpers est formé par $o' - T^h(c)$.

4.2 La méthode Maglice-O

Nous présentons dans cette section, l'algorithme de mise à jour des icebergs MAGALICE-O(MAintaining GALois ICEberg-Object) [RNVG04].

4.2.1 Description de Magalice-O

La première partie consiste à mettre à jour l'iceberg en le considérant comme un treillis complet. Ainsi, on applique la méthode ADD-OBJECT décrite dans le chapitre précédent. La deuxième partie consiste à supprimer les concepts devenus infréquents dans le nouvel iceberg. La dernière partie consiste à trouver les concepts *jumpers* et de les lier à leurs successeurs.

```
1: procedure MAGALICE-O(In: 𝓛 an iceberg lattice, α a minimum support, o a new object, Context an
   indexed set of objects)
2:
3: ADD-OBJECT(𝓛,o)
4: for all c in 𝓛 do
5:    if support(c) < α then
6:       DROP(𝓛,c)
7: FIND-FREQUENT-LOWER-COVERS(𝓛,α,Context,c,o)
```

Algorithm 11: Iceberg update upon the insertion of new object within Magalice

4.2.2 Calcul des Jumpers

La procédure FIND-FREQUENT-LOWER-COVERS est l'application des propriétés de la section précédente. On cherche à générer les successeurs immédiats fréquents des concepts visibles. Ces derniers sont triés par ordre décroissant de la taille des intentions. Pour chaque concept (visible) c, et pour chaque attribut a appartenant à la trace de c, on calcule le concept $\wedge_a c$ à partir de son extention. Si cette extention est fréquente plusieurs cas se présentent:

- Si le concept $\wedge_a c$ a été déjà généré par un autre concept alors il suffit de mettre à jour les liens entre les deux concepts $\wedge_a c$ et c. En outre, comme les autres attributs de l'intention de $\wedge_a c$ vont générer la même intersection et par suite ne vont rien ajouter, ils seront éliminés de la liste des attributs candidats (pour accélérer le calcul) (ligne 20).
- Sinon, on cherche le concept $\wedge_a c$ localement. En d'autres termes, on construit l'intention de $\wedge_a c$ au fur et à mesure de chaque attribut appartenant à cette intention.
  - Si l'extention du concept $\wedge_a c$ a été généré par un autre attribut, il suffit de mettre à jour l'intention du concept $\wedge_a c$ (ligne 24).
  - Sinon, il suffit de créer le candidat $\wedge_a c$ avec une intention égale à celle de c à laquelle on ajoute l'attribut a (ligne 26).

Une fois que tous les attributs candidats ont été traités, on crée tous les concepts candidats et on les lie au concept c.

```
 1: procedure FIND-FREQUENT-LOWER-COVERS(In/Out: $\bar{\mathcal{L}}^{\alpha}$ an iceberg lattice, Context an indexed
    set of objects, o the new object)
 2:
 3: Local : Jumpers : set of concepts
 4: Local : Th, Pool, h : set of attributes
 5: Local : Extent : set of objects
 6: Local : Candidates : set of pairs of tuples ⟨X,Y⟩
 7:
 8: Jumpers ← ∅
 9: SORT(V+(o))      {in descending order of the Intent}
10: for all c ∈ V+(o) do
11:    Candidates ← ∅; Th ← Int(c); h ← ∅
12:    for all c̄ ∈ Cov^l(c) do
13:       Th ← Th ∪ T^h(c̄) ∪ h(c̄)
14:    Pool ← o' - Th
15:    while not Pool = ∅ do
16:       a ← extract-first(Pool); Extent ← Ext(c) ∩ a'
17:       if α * |O| + α ≤ |Extent| < α * |O| + 1 then
18:          c̄ ← LOOKUP(Extent, Jumpers)
19:          if c̄ ≠ NULL then
20:             Cov^l(c) ← Cov^l(c) ∪ {c̄}; h ← h ∪ Int(c̄); Pool ← Pool - Int(c̄)
21:          else
22:             can ← LOOKUP(Extent, Candidates)
23:             if can ≠ NULL then
24:                can.Y ← can.Y ∪ {a}
25:             else
26:                can ← ⟨Extent, (Int(c) ∪ {a}) ⟩; Candidates ← Candidates ∪ {can}
27:             h ← h ∪ {a}
28:    for all can ∈ Candidates do
29:       c̄ ← NEWCONCEPT(can.X, can.Y); Cov^l(c) ← Cov^l(c) ∪ {c̄}
30:       Jumpers ← Jumpers ∪ {c̄}
31:    T^h(c) ← Th; h(c) ← h
```

Algorithm 12: Calcul des jumpers d'un concept visible.

Exemple Illustrons le calcul des prédecesseurs du concept $c_{\#4}$ de l'iceberg de la figure . Comme ce dernier n'a pas de prédecesseurs, alors $pool(c_{\#4}) = defg$. En outre, $jumpers$ qui représente $\mathbf{H}^{+}(5)$ est vide ($c_{\#4}$ est le premier concept visible traité).

Pour l'attribut d, $Extent(\nu(d)) \cap Extent(c_{\#4}) = 15 \cap 135 = 15$ est fréquente. Comme $jumpers$ est vide, on cherche l' extention 15 localement. Puisqu'il n'existe pas dans l'ensemble $candidates$, alors on crée un nouveau candidat formé par le couple (15,dh).

Pour l'attribut e, $Extent(\nu(e)) \cap Extent(c_{\#4}) = 35 \cap 135 = 35$ est fréquente. On crée un nouveau candidat formé par le couple (35,eh).

Pour l'attribut f, $Extent(\nu(f)) \cap Extent(c_{\#4}) = 35 \cap 135 = 35$ est fréquente. Comme un candidat local ayant une extention 35 existe, on complète la partie correspondante à l'intention. Par suite, le candidat devient (35,efh).

Pour l'attribut g, $Extent(\nu(g)) \cap Extent(c_{\#4}) = 145 \cap 135 = 15$ est fréquente. Comme un candidat local ayant une extention 15 existe, on complète la partie correspondante à l'intention. Par suite, le candidat devient (15,dgh).

On crée deux concepts correspondants aux deux candidats et on met à jour les liens entre ces concepts et leur successeur $c_{\#4}$.

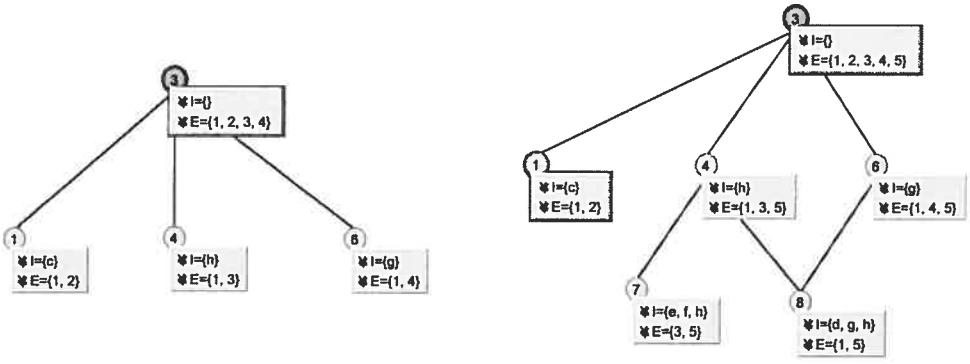

FIG. 4.1 – ***Gauche:*** *l'iceberg* $\mathcal{L}_1^{0.3}$ *associé à* $\mathcal{K}_1$ =($O_1 = \{1,2,3,4\}$,$A = \{a,b,\cdots,h\}$,I_1) ***Droite:*** *l'iceberg* $\mathcal{L}_2^{0.3}$ *associé à* $\mathcal{K}_2$ =($O_2 = \{1,2,3,4,5\}$,$A = \{a,b,\cdots,h\}$,I_2)

4.3 Incrémentalité par attribut d'un iceberg

Contrairement au cas de l'incrémentalité par objet, les opérations effectuées lors de la maintenance de l'iceberg par ajout d'un nouvel attribut sont presque les mêmes que pour le treillis complet. Ceci est dû à la constance dans la fréquence des concepts. En effet, une fois un concept est créé dans l'iceberg, son statut fréquent demeure inchangé par la suite. Pour le montrer, considérons un concept c de $\mathcal{L}_1^\alpha$ alors $freq(\sigma_a(c))$ dans $\mathcal{L}_2^\alpha = \frac{|Extent(\sigma_a(c))|}{|O|}$. Comme $Extent(\sigma_a(c)) = Extent(c)$ et le nombre des objets ne varie pas lors du passage de $\mathcal{L}_1^\alpha$ à $\mathcal{L}_2^\alpha$ on aura alors $freq(c) = freq(\sigma_a(c))$.

La seule différence entre la mise à jour de l'iceberg et celle du treillis complet réside dans le choix des valeurs de $\mathcal{R}$. En effet, comme ces valeurs peuvent être les extentions des nouveaux concepts, il faut limiter le calcul à celles qui sont fréquentes. Le parcours "top-down" de l'iceberg offre une solution efficace pour cette tâche comme le montre la propriété suivante.

Propriété 27. $\forall c \in \mathcal{L}_1^\alpha$, *si* $|\mathcal{R}_1(c)| \leq \alpha|O|$ *alors* $\forall \underline{c}$ — $\underline{c} \prec c$, $|\mathcal{R}_1(\underline{c})| \leq \alpha|O|$.

Preuve. *: En effet,* $\underline{c} \prec c$ *implique que* $\underline{X} \subset X$. *Par suite,* $(\underline{X} \cap a') \subset (X \cap a')$ *et par conséquent* $|\underline{X} \cap a'| \leq |X \cap a'| \leq \alpha$ ■

4.3.1 La méthode Magalice-A

Ajoutons le filtrage des concepts à la méthode ADD-ATTRIBUTE décrite dans le chapitre précédent. On obtient le pseudo-code (voir algorithme 13) de la mise à jour de l'iceberg suite à l'ajout d'un nouvel attribut.

Pour chaque concept c, on vérifie en premier s'il n'est pas un successeur d'un concept ayant produit une intersection infréquente (en vertu de la propriété 27). Pour cela, on le cherche dans l'ensemble *unfrequent* (ligne 7). S'il n'y est pas, on calcule alors la valeur de $\mathcal{R}(c)$ et on teste sa fréquence (ligne 10). Si elle est fréquente, on applique les opérations effectuées sur le treillis complet (ligne 11-25). Sinon, on ajoute tous les successeurs de c à l'ensemble *unfrequent*(lignes 27-28) afin de limiter le nombre de calcul des intersections à celles qui sont fréquentes.

```
1: procedure MAGALICE-A(In/Out: 𝓛 = ⟨𝒞, ≤⟩ an iceberg; In: a an attribute, α a minimum support)
2:
   Local : unfrequent : set
   Local : E : extent
   Local : ĉ : concept
3:
4: SORT(𝒞)
5: for all c in 𝒞 do
6:   E ← Extent(c)
7:   ĉ ← lookup(unfrequent, E)
8:   if (ĉ = NULL) then
9:     if |ℛ(c)| ≥ α · |O| then
10:      new-min ← ARGMIN({|ℛ(c̄)| | c̄ ∈ Cov^l(c)})
11:      if |ℛ(c)| ≠ |ℛ(new-min)| then
12:        if |ℛ(c)| = |Extent(c)| then
13:          Intent(c) ← Intent(c) ∪ {a}      {c is modified}
14:          M(a) ← M(a) ∪ {c}
15:          new-min ← c
16:        else
17:          ĉ ← NEW-CONCEPT(ℛ(c),Intent(c) ∪ {a})      {c is genitor}
18:          Candidates ← {ChiPlus(c̄) | c̄ ∈ Cov^l(c)}
19:          for all c̄ in MIN-CLOSED(Candidates) do
20:            NEW-LINK(ĉ,c̄)
21:            if c̄ ∈ M(a) then
22:              DROP-LINK(c,c̄)
23:          new-min ← ĉ
24:          𝓛 ← 𝓛 ∪ {ĉ}
25:          ChiPlus(c) ← new-min
26:   else
27:     for all c̄ in Cov^u(c) do
28:       unfrequent ← unfrequent ∪ {c̄}
```

Algorithm 13: Mise à jour de l'iceberg suite à l'ajout d'un nouvel attribut au contexte.

Exemple Soient $\mathcal{K}_1$ et $\mathcal{K}_2$ les deux contextes obtenus avant et après l'ajout de l'attribut h de l'exemple du chapitre 2 et le seuil $\alpha = 0.2$. La table 4.1 montre l'exécution de la partie de l'algorithme dédiée au filtrage des concepts.

Concept(c)	$\mathcal{R}(c)$	$\lvert\mathcal{R}(c)\rvert \geq 1$	Statut (c)	*unfrequent*
$\{c_{\#1}\}$	135	OUI	géniteur	$\emptyset$
$\{c_{\#3}\}$	1	OUI	inchangé	$\emptyset$
$\{c_{\#7}\}$	15	OUI	inchangé	$\emptyset$
$\{c_{\#6}\}$	35	OUI	modifié	$\emptyset$
$\{c_{\#0}\}$	$\emptyset$	NON	inchangé	$\{c_{\#0}\}$
$\{c_{\#4}\}$	15	OUI	modifié	$\{c_{\#0}\}$
$\{c_{\#2}\}$	1	OUI	modifié	$\{c_{\#0}\}$
$\{c_{\#5}\}$	5	OUI	modifié	$\{c_{\#0}\}$

TAB. 4.1 – *Trace du filtrage des concepts de l'iceberg $\mathcal{L}_1^{0.2}$ lors de l'ajout de l'attribut h.*

4.4 Complexité

Calculons la complexité de MAGALICE-O. Soient $l = |\mathcal{L}_2|$, $m = |A|$, $n = |O|$ et $\Delta l = |\mathcal{L}_2| - |\mathcal{L}_1|$. L'algorithme principal (algorithme 11) est composé de trois parties: la première a pour complexité $O(l \cdot (m+n) + \Delta l \cdot n^2)$ (d'après le chapitre 2). La deuxième partie consiste à éliminer les concepts devenus infréquents. Elle est de l'ordre $O(l \cdot n)$ puisqu'on visite au plus tous les concepts du treillis et si on trouve un concept infréquent, on devrait éliminer ses liens avec ses parents qui sont bornés par n. Calculons la complexité de la dernière partie reliée à la génération des jumpers. Le tri des concepts est linéaire dans la taille du treillis. Les concepts visibles sont bornés par la taille du treillis. Les couvertures inférieures d'un concept sont bornés par m (raisonnement par analogie aux couvertures supérieures) et la réunion des traces est limité à m (les traces sont des intentions). Le calcul des intersections des extentions est borné

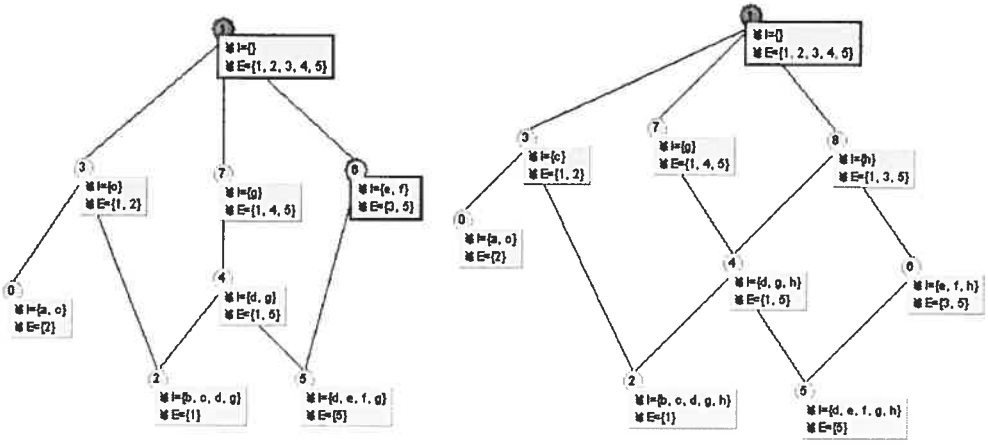

FIG. 4.2 – ***Gauche:*** *l'iceberg* $\mathcal{L}_1^{0.2}$ *associé à* $\mathcal{K}_1 = (O = \{1,2,3,4,5\}, A_1 = \{a,b,\cdots,g\}, I_1)$. ***Droite:*** *l'iceberg* $\mathcal{L}_2^{0.2}$ *associé à* $\mathcal{K}_2 = (O = \{1,2,3,4,5\}, A_2 = \{a,b,\cdots,h\}, I_2)$.

par n et la recherche des extentions peut se faire en $O(n)$. Comme les jumpers sont limités par le nombre des attributs, la dernière boucle (ligne 28) est de l'ordre de m. Ainsi la complexité de FIND-LOWER-COVERS est de l'ordre: $O(m^2 + mn) \cdot l) = O(m \cdot (m + n) \cdot l)$.

En somme, la complexité de MAGALICE-O est: $O(l \cdot (m+n) + \Delta l \cdot n^2) + O(l \cdot n) + O(m \cdot (m+n) \cdot l)$ $= O(\Delta l \cdot n^2 + m \cdot (m + n) \cdot l)$.

En ce qui concerne MAGALICE-A, sa complexité est la même que celle du treillis complet. En effet, la fonction LOOKUP (ligne 7) est de l'ordre de n, le test de la fréquence de $\mathcal{R}$ est de l'ordre $O(1)$ (ligne 9) et l'ajout des successeurs d'un concept à l'ensemble des concepts infréquents est de l'ordre de m. Et par conséquent, en faisant la somme total des complexités de toutes les fonctions de l'algorithme 13, on obtient la complexité de l'algorithme de la mise à jour du treillis complet qui est $O(ml \cdot (m + n))$ (section 2.3.2.3).

Chapitre 5

Maintenance de la famille des générateurs

Comme les générateurs jouent un rôle important dans le calcul des motifs fermés et présentent les prémisses des règles d'association, on s'intéresse à à leur évolution lors de la mise à jour des treillis par ajout d'un nouvel objet/attribut. Dans [VMRG03], une approche est présentée permettant de traiter le cas de l'ajout d'un nouvel objet. Dans ce chapitre, nous présentons notre approche [NVRG05] pour la mise à jour du treillis des concepts et de leurs générateurs suite à l'ajout d'un nouvel attribut au contexte.

On s'intéresse à l'évolution des générateurs de chaque concept c lors du passage de $\mathcal{L}_1$ à $\mathcal{L}_2$. Puisque seules les intentions des nouveaux concepts et des concepts modifiés sont affectées par l'ajout du nouvel attribut a, on cherche à calculer leurs générateurs respectifs dans $\mathcal{L}_2$.

Afin de mieux voir les modifications apportées au nouveau treillis, nous considérons un nouvel exemple dans ce chapitre dans lequel on trouve plus d'objets et d'attributs que le contexte précédent (chapitre 2). La table 5.1 montre la relation binaire et le treillis associé aux 7 premiers attributs.

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

TAB. 5.1 – *table binaire* $\mathcal{K}_1 = (O = \{1,2,...,8\}, A_1 = \{a,b,...,g\}, I_1)$ *et l'attribut* h.

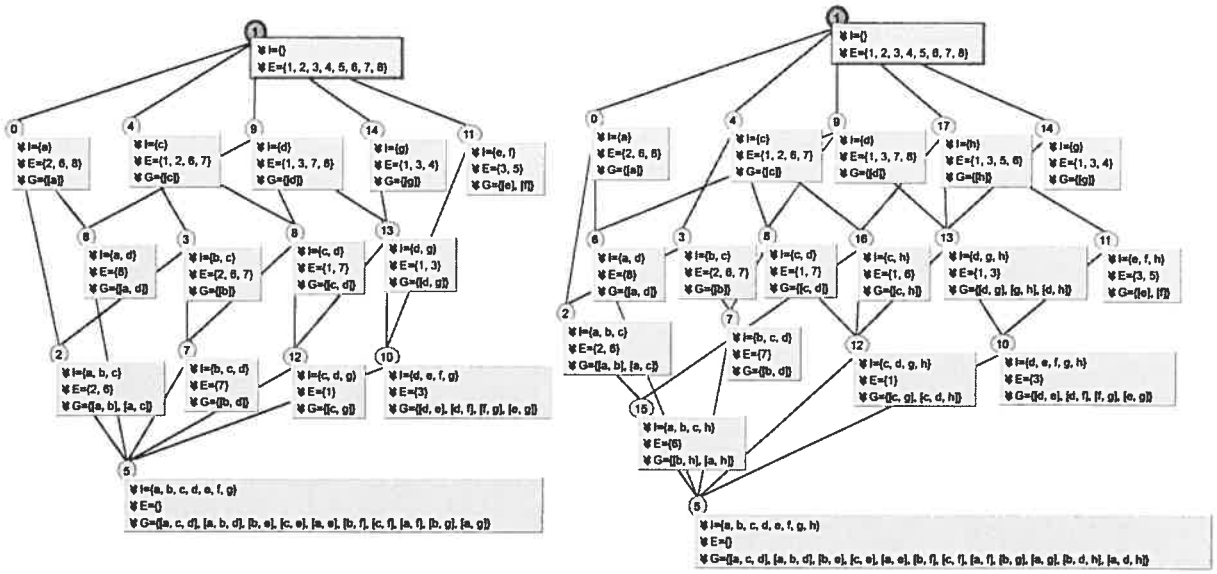

FIG. 5.1 – ***Gauche:****Le treillis* $\mathcal{L}_1$ *associé à* $\mathcal{K}_1$. ***Droite:*** *Le treillis* $\mathcal{L}_2$ *associé à* $\mathcal{K}_2 = (O, A_1 \cup \{h\}, I_1 \cup \{h\}' \times \{h\})$.

5.1 Fondements théoriques

Dans un premier temps, trouvons la relation entre les générateurs d'un concept c de $\mathcal{L}_1$ et ceux de son correspondant $\sigma_a(c)$ dans $\mathcal{L}_2$.

La propriété suivante stipule que les générateurs de c sont aussi des générateurs de $\sigma_a(c)$.

Propriété 28. *:* $\forall\ c = (X,Y) \in \mathcal{L}_1, Gen_1(c) \subseteq Gen_2(\sigma_a c)$

Preuve. En effet, d'après les définitions des fonctions dérivées (^{1} et ^{2}), on a: $G^1 = \{o \in O | \forall a \in Y, oI_1a\}$ et $G^2 = \{o \in O | \forall a \in Y, oI_2a\}$. Comme l'extention du concept $\sigma_a(c)$ dans $\mathcal{L}_2$ est la même que celle de c dans $\mathcal{L}_1$ on aurait $G^1 = G^2$.

Soit $G \in Gen_1(c)$. On a: $G^1 = X$, $X = \bar{X}$ $G^1 = G^2$ et alors $G^2 = \bar{X}$ et par la suite $G^{22} = \bar{X}^2 = \bar{Y}$. De plus G est minimal pour $\bar{Y}$ sinon il ne serait pas générateur de c. ■

A titre d'exemple, considérons le concept $c_{\#12} = (1,cdg)$ dans $\mathcal{L}_1$ (dans la figure 5.1 à gauche). On voit que son générateur cg est aussi générateur de $\sigma_a(c_{\#12}) = (1,cdgh)$ dans $\mathcal{L}_2$ (dans la figure 5.1 à droite)

Nous avons considéré, dans l'exemple, le cas d'un concept modifié car pour le cas d'un concept *géniteur* ou *inchangé*, les générateurs de c sont exactement les mêmes que ceux de $\sigma_a(c)$.

Propriété 29. *:* $\forall\ c = (X,Y) \in \mathcal{L}_1$, *si* $Intent(\sigma_a(c)) = Intent(c)$ *alors* $Gen_1(c) = Gen_2(\sigma_a(c))$

Preuve. $Gen_1(c) \subseteq Gen_2(\sigma_a(c))$ (d'après la Propriété 28). L'inclusion $Gen_2(\sigma_a(c) \subseteq Gen_1(c))$ découle du fait que si $G \in Gen_2(\sigma_a(c)$ et $Intent(c) = Intent(\sigma_a(c))$ alors $G^1 = G^2$ et $G^{11} = G^{22} = Y$. De plus, G minimal (par rapport à c) sinon il ne serait pas générateur de $\sigma_a(c)$. ■

Par exemple, considérons le concept $c_{\#2} = (26,abc)$ dans $\mathcal{L}_1$. L'ensemble $\{ab,ac\}$ de ces générateurs est égal à celui de $\sigma_a(c_{\#2}) = c_{\#2}$ dans $\mathcal{L}_2$.

A partir des deux Propriétés 28 et 29, on en déduit:

Propriété 30. *: $Gen(\mathcal{L}_1) \subseteq Gen(\mathcal{L}_2)$*

Preuve. Soit $c = (X,Y) \in \mathcal{L}_1$. Dans $\mathcal{L}_2$, $\sigma_a(c)$ peut avoir un des trois statuts suivants: *ancien*, *géniteur* ou *modifié*. Pour les trois cas , on a $Gen_1(c) \subseteq Gen_2(\sigma_a(c))$. ■

La propriété 30 montre qu'un générateur de $\mathcal{L}_1$ reste générateur dans $\mathcal{L}_2$. Ainsi, notre tâche consiste à calculer les nouveaux générateurs dans $\mathcal{L}_2$. Comme nous cherchons à calculer les générateurs des nouveaux concepts et des concepts modifiés, et comme on le sait déjà (d'après la section 2.3.2.2) leurs correspondants dans $\mathcal{L}_1$ sont les éléments minimaux de leurs classes d'équivalence, étudions l'évolution de ces classes de $\mathcal{L}_1$ à $\mathcal{L}_2$. En effet, la classe d'un nouveau concept dans $\mathcal{L}_2$ est exactement la classe de son géniteur dans $\mathcal{L}_1$ à laquelle on ajoute le nouveau concept et la classe d'un concept modifié est exactement la même dans $\mathcal{L}_1$ et $\mathcal{L}_2$.

Corrolaire 1. *:*

- *Si* $c \in N_2(a)$ *alors* $[c]_{\mathcal{R}_2} = \gamma_a([c]_{\mathcal{R}_1}) \cup \{c\}$
- *Si* $c \in M_2(a)$ *alors* $[c]_{\mathcal{R}_2} = \gamma_a([c]_{\mathcal{R}_1})$

Cette propriété montre que les classes d'équivalence $[]_{\mathcal{R}_2}$ diffèrent au plus par un élément de leurs contre-parties $[]_{\mathcal{R}_1}$ qui est le nouvel élément minimal de la classe dans $\mathcal{L}_2$.

On se demande s'il n'existe pas de lien entre les générateurs des concepts de la classe et ceux de son élément minimal. En effet, soit $c = (X,Y)$ le concept minimal d'une classe d'équivalence et $\bar{c} = (\bar{X},\bar{Y})$ un concept de cette classe ayant $\bar{G}$ comme générateur. D'après la définition de $[c]_R$, on a: $\bar{X} \cap a^2 = X$ et par la suite $\bar{G}^2 \cap a^2 = Y^2$. En outre, on sait que pour A, $B \in 2^O$, on a $(A \cup B)^2 = A^2 \cap B^2$. $\bar{G}^2 \cap a^2 = Y^2$ s'écrit alors $(\bar{G} \cup a)^2 = Y^2$ et par suite $(\bar{G} \cup a)^{22} = Y^{22}$.

Ainsi, on vient de montrer que pour chaque générateur $\bar{G}$ d'un concept $\bar{c}$ appartenant à $[\bar{c}]_{\mathcal{R}_1}$, la fermeture de $(\bar{G} \cup a)$ est égale à l'intention du concept minimal de la la classe $[\sigma_a(\bar{c})]_{\mathcal{R}_2}$

On va montrer dans ce qui suit que $(\bar{G} \cup a)$ est un générateur du concept c si et seulement si $\bar{G}$ est minimal (en terme d'inclusion) dans $[\gamma_a(c)]_{\mathcal{R}_1}$.

Propriété 31. *: $\forall\, c \in M_2(a)$, on a:*

$$Gen_2(c) = Gen_1(\gamma_a(c)) \cup min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1} \text{ et } \hat{c} \neq \gamma_a(c)} Gen_1(\hat{c})) \times \{a\}$$

La preuve de cette propriété est donnée dans l'annexe A.

Exemple Calculons les nouveaux générateurs du concept $c_{\#13} = (13,dgh)$ dans $\mathcal{L}_2$. $\mathcal{R}(c_{\#13})$ = 13 et $[c_{\#13}]_{\mathcal{L}2} = \{c_{\#9},c_{\#14},c_{\#13}\}$. Evidemment, $c_{\#13}$ est minimal dans sa classe, et plus précisement, un concept modifié. Pour $\hat{c} = c_{\#9}$, $Gen_1(\hat{c}) = d$ et pour $\hat{c} = c_{\#14}$, $Gen_1(\hat{c}) = g$. Comme les deux générateurs d and g sont incomparables, l'ensemble des nouveaux générateurs de $c_{\#13}$ est $\{dh,gh\}$ et par conséquent $Gen_2(c_{\#13})$ est l'ensemble formé par $\{dg,dh,gh\}$.

Calculons maintenant les générateurs d'un nouveau concept c dans $\mathcal{L}_2$. La propriété suivante montre que les générateurs de c sont formés par l'ajout du nouvel attribut a à tous les générateurs minimaux (en terme de la taille) des concepts appartenant à $[\gamma_a(c)]_{R_1}$.

Remarque : Si $c_n = (X_n,Y_n) \in \mathbf{N_2}(a)$ alors $\forall\ G \in Gen_2(c_n)$ on a $a \in G$.

En effet, soit $G \in Gen_2(c_n)$ et $a \notin G$. On a alors $G^1 = G^2$. De plus, on sait que $\gamma_a(c_n)$ est le concept géniteur du nouveau concept c_n. $G^1 = (Y_n - a)^1 \Rightarrow G^{22} = (Y_n - a)^{22} = Y_n - a \subset Y_n$ ($G^2 = G^1 = extent(\gamma_a(c_n))$ et par suite $G^{22} = (extent(\gamma_a(c_n)))^2 = Y_n - a$). Ce qui contredit $G \in Gen^2(c_n)$.

Propriété 32. *: $\forall c \in N_2(a)$. On a:*

$$Gen_2(c) = min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1}} Gen_1(\hat{c})) \times \{a\}$$

La preuve de cette propriété est donnée dans l'annexe A.

Exemple Calculons les générateurs du nouveau concept $c_{\#15} = (6,abch)$ dans $\mathcal{L}_2$. $\mathcal{R}(c_{\#15}) =$ 6 et $[c_{\#15}]_{\mathcal{L}2} = \{c_{\#0},c_{\#3},c_{\#2},c_{\#15}\}$. $[\gamma_a(c_{\#15})]_{\mathcal{L}^1} = \{c_{\#0},c_{\#3},c_{\#2}\}$. Pour $\hat{c} = c_{\#0}$, $Gen_1(\hat{c}) = a$, pour $\hat{c} = c_{\#3}$, $Gen_1(\hat{c}) = b$ et pour $\hat{c} = c_{\#2}$, $Gen_1(\hat{c}) = \{ab,ac\}$.

Ainsi, $min(\bigcup_{\hat{c}\in[(6,abc)]_R} Gen_1(\hat{c})) = min\{a,b,ab,ac\} = \{a,b\}$ et par conséquent $Genc_{\#15} = \{ah,bh\}$.

Comme les nouveaux générateurs dans $\mathcal{L}_2$ sont ceux des concepts modifiés et nouveaux, d'après les Propriétés 31 et 32, on en déduit:

Corrolaire 2. : $Gen(\mathcal{L}_2) - Gen(\mathcal{L}_1) \subseteq Gen(\mathcal{L}_1) \times \{a\}$.

À partir de ces propriétés, nous décrivons, la méthode INC-A-GEN dédiée à la mise à jour du treillis et des générateurs de ces concepts suite à l'ajout du nouvel attribut.

5.2 Description de INC-A-GEN

D'après la section précédente, les générateurs des concepts modifiés et nouveaux sont calculés à partir des générateurs de leurs classes d'équivalence appropriées. Contrairement à la mise à jour du treillis seulement, on est obligé de construire explicitement les classes d'équivalence durant cette mise à jour. Une classe, représentée par l'enregistrement θ dans l'algorithme 14, est composée de deux champs: le concept minimal, noté *min-concept* et l'ensemble des générateurs minimaux (en taille), noté *min-gen*. Toutes les classes sont stockées dans une structure, notée *Classes*, indexées par la valeur $\mathcal{R}_1$.

La procédure COMPUTE-CLASSES (voir algorithme 15) a pour but de calculer les classes d'équivalence, c.à.d maintenir les deux variables *min-concept* et *min-gen*. Pour chaque concept c dans $\mathcal{L}_1$, on calcule la valeur de $\mathcal{R}(c) = extent(c) \cap a^2$ (ligne 4) et on cherche la classe θ correspondante dans *Classes* (ligne 5). Si θ n'existe pas encore, elle est alors créée en initialisant le *min-concept* à c et *min-gen* à $\emptyset$ (ligne 6-9). Sinon, il faudrait mettre à jour *min-concept* en lui affectant la valeur de c (ligne 11). Et enfin, *min-gen* est mis à jour (ligne 12) en lui ajoutant les générateurs de c qui sont minimaux (par rapport à leur taille) dans *min-gen*.

L'algorithme 14 présente la MAJ du treillis et des générateurs. La première étape consiste à partitionner les concepts dans les classes d'équivalence (ligne 4). La mise à jour du treillis

consiste l'étape suivante (ligne 5-15): pour chaque classe, on calcule le statut de son concept minimal; s'il est un géniteur alors on créé le nouveau concept et on met à jour l'ordre; sinon son intention est mise à jour. Une fois le minimal de la classe calculé dans le nouveau treillis, on met à jour ses générateurs (ligne 16).

```
1:  procedure INCA-GEN(In/Out: 𝓛 = ⟨𝒞, ≤⟩ a Lattice, In: a an attribute)
2:  Local : Classes : an indexed structure of classes
3:
4:  COMPUTE-CLASSES(𝒞,a)
5:  for all θ in Classes do
6:     c ← θ.min-concept
7:     if |ℛ(c)| = |extent(c)| then
8:        intent(c) ← intent(c) ∪ {a}      {c is modified}
9:     else
10:       ĉ ← newConcept(ℛ(c),Intent(c) ∪ {a})      {c is genitor}
11:       𝓛 ← 𝓛 ∪ {ĉ}
12:       updateOrder(c,ĉ)
13:       gen(ĉ) ← ∅
14:       θ.min-concept ← ĉ
15:    c ← θ.min-concept
16:    gen(c) ← gen(c) ∪ θ.min-gen × {a}
```

Algorithm 14: MAJ du treillis et des générateurs par ajout d'un nouvel attribut

Exemple détaillé Dans le but d'illustrer le mécanisme de l'algorithme 14, considérons l'ajout de l'attribut h au contexte de la figure 5.1. La trace de l'algorithme est donnée dans la table 5.2

$Concept(c)$	$\mathcal{R}(c)$	$[c]_R.min\text{-}concept$	$Statut(c)$	$[c]_R.min\text{-}gen$	Explication
$\{c_{\#1}\}$	1356	$\{c_{\#1}\}$	géniteur	$\emptyset$	une nouvelle classe est créée. Comme $\{c_{\#1}\}$ est un géniteur, l'ensemble des générateurs du nouveau concept est égal à $\emptyset \cup \{h\} = \{h\}$

$\{c_{\#0}\}$	6	$\{c_{\#0}\}$	inchangé	$\{a\}$	une nouvelle classe est créée
$\{c_{\#4}\}$	16	$\{c_{\#4}\}$	géniteur	$\{c\}$	une nouvelle classe est créée. Comme $\{c_{\#4}\}$ est un géniteur, l'ensemble des générateurs du nouveau concept est égal à $\{c\} \cup \{h\} = \{ch\}$
$\{c_{\#9}\}$	13	$\{c_{\#9}\}$	inchangé	$\{d\}$	une nouvelle classe est créée
$\{c_{\#14}\}$	13	$\{c_{\#14}\}$	inchangé	$\{d,g\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour
$\{c_{\#11}\}$	35	$\{c_{\#11}\}$	modifié	$\emptyset$	une nouvelle classe est créée. Comme $\{c_{\#11}\}$ est un concept modifié et unique dans sa classe d'équivalence, l'ensemble de ses générateurs ne change pas.
$\{c_{\#6}\}$	$\emptyset$	$\{c_{\#6}\}$	inchangé	$\{ad\}$	une nouvelle classe est créée
$\{c_{\#3}\}$	6	$\{c_{\#3}\}$	inchangé	$\{a,b\}$	Comme la classe existe déjà, son concept minimal et l'enssemble des générateurs sont mis à jour
$\{c_{\#8}\}$	1	$\{c_{\#8}\}$	inchangé	$\{cd\}$	une nouvelle classe est créée

$\{c_{\#13}\}$	13	$\{c_{\#13}\}$	modifié	$\{d,g\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour. Puisque $\{c_{\#13}\}$ est un concept modifié, l'ensemble de ses nouveaux générateurs est formé par $\{dh,gh\}$ et par conséquent le nouvel ensemble des générateurs de $\{c_{\#13}\}$ est égal à $\{dg,dh,gh\}$
$\{c_{\#2}\}$	6	$\{c_{\#2}\}$	géniteur	$\{a,b\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour. En outre, les générateurs ab et ac du concept $\{c_{\#2}\}$ ne sont pas minimaux dans leur classe ce qui n'entraîne aucun changement dans cet ensemble. Et ainsi, l'ensemble des générateurs du nouveau concept est égal à $\{ah,bh\}$
$\{c_{\#7}\}$	$\emptyset$	$\{c_{\#7}\}$	inchangé	$\{ad,bd\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour.

$\{c_{\#12}\}$	1	$\{c_{\#12}\}$	modifié	$\{cd\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour. Puisque $\{c_{\#12}\}$ est un concept modifié, l'ensemble de ses nouveaux générateurs est formé par cdh et par conséquent le nouvel ensemble des générateurs de $\{c_{\#12}\}$ est égal à $\{cg,cdh\}$
$\{c_{\#10}\}$	3	$\{c_{\#10}\}$	modifié	$\{de,df,fg,eg\}$	une nouvelle classe est créée. Comme $\{c_{\#10}\}$ est un concept modifié et unique dans sa classe d'équivalence, l'ensemble de ses générateurs ne change pas
$\{c_{\#5}\}$	$\emptyset$	$\{c_{\#5}\}$	modifié	$\{ad,bd\}$	Comme la classe existe déjà, son concept minimal et l'ensemble des générateurs sont mis à jour. Comme $\{c_{\#5}\}$ est modifié, l'ensemble de ses nouveaux générateurs est formé par $\{adh,bdh\}$

Table 5.2: Exécution de la mise à jour (treillis et générateurs).

Rappelons que dans le cas des règles d'association, on construit l'iceberg et non pas le treillis complet. Pour cela, la section suivante est consacrée à l'étude de la mise à jour des générateurs dans le cas d'un iceberg.

```
1: procedure COMPUTE-CLASSES(In/Out: C concept set, In: a an attribute)
2:
3: for all c in C do
4:    E ← extent(c) ∩ a'
5:    θ ← lookup(Classes, E)
6:    if (θ = NULL) then
7:       θ ← newClass()
8:       θ.min-concept ← c
9:       put(Classes, θ, E)
10:   if (θ.min-concept < c) then
11:      θ.min-concept ← c
12:   θ.min-gen ← Min(θ.min-gen ∪ gen(c))
```

Algorithm 15: Calcul des classes d'équivalence $[]_{\mathcal{R}}$ dans le treillis initial

5.3 La variante iceberg

De la section précédente, il en sort que la mise à jour des générateurs est limitée à un seul concept par classe (le minimum de la classe). Or, dans le cas de l'iceberg, $\mathcal{R}(c)$ peut être soit fréquente, soit infréquente. Dans le dernier cas, $[c]_R$ sera divisée en deux parties non vides l'une fréquente et l'autre infréquente respectivement . Mais, comme les modifications sont rapportées sur le minimum de la classe, la partie infréquente ne pose pas un problème car en effet l'élément minimal, infréquent dans ce cas, ne sera pas traité. On ne traite l'élément minimal que s'il est fréquent. Or si l'élément minimal est fréquent alors tous les éléments de la classe le sont. Donc à chaque fois qu'un recalcul de générateurs s'impose, tous les concepts de la classe d'équivalence appropriée sont présents dans l'iceberg et les deux Propriétés 31 et 32 sont toujours valides pour les icebergs aussi.

En somme, tout comme dans le cas du treillis complet, lors de la mise à jour de l'iceberg (concepts et leurs générateurs), il faudrait construire les classes d'équivalence explicitement. Comme on l'a déjà vu dans le chapitre 4, la mise à jour de l'iceberg (juste les concepts) se fait presque de la même manière que celle du treillis complet. Il suffit de filtrer les concepts pour ne traiter que ceux qui sont fréquents.

Ainsi, pour le cas de la mise à jour de l'iceberg (concepts et leurs générateurs), le filtrage des concepts est ajouté dans la phase de regroupement des concepts dans les classes d'équivalence. Ainsi, dans la nouvelle méthode COMPUTE-F-CLASSES (voir Algorithm 16), les concepts sont rangés dans une file. A chaque concept c traité, on calcule la valeur de $\mathcal{R}(c)$. Si cette valeur est fréquente, on exécute les mêmes tâches que celles du treillis complet et on ajoute tous les successeurs de c à la file. Cette procédure est répétée jusqu'à ce que la file soit vide.

```
1: procedure COMPUTE-F-CLASSES(In/Out: L_α an iceberg lattice, In: a an attribute)
2: Local : cQ : a queue of concepts
3:
4: in(cQ, top(L_α))
5: while nonempty(cQ) do
6:    c ← out(cQ)
7:    E ← extent(c) ∩ a'
8:    if (|E| ≥ α|O|) then
9:       θ ← lookup(Classes, E)
10:      if (θ = NULL) then
11:         θ ← newClass()
12:         θ.min-concept ← c
13:         put(Classes, θ, E)
14:      if (θ.min-concept > c) then
15:         θ.min-concept ← c
16:      θ.min-gen ← Min(θ.min-gen ∪ gen(c))
17:      for all ĉ ∈ Cov^u(c) do
18:         in(cQ, ĉ)
```

Algorithm 16: Calcul des classes d'équivalence fréquentes $[]_{\mathcal{R}}$ dans le treillis initial

Exemple Soit $\mathcal{L}_1^{0.2}$ l'iceberg associé à l'ajout des 7 premiers attributs du contexte de la table 5.1. Le filtrage des concepts de $\mathcal{L}_1^{0.2}$ est montré dans la table 5.3 et les deux icebergs obtenus avant et après l'insertion de h sont donnés dans la figure 5.2.

Concept(c)	$\mathcal{R}(c)$	$\lvert\mathcal{R}(c)\rvert \geq 2$	Statut (c)	cQ
$\{c_{\#1}\}$	1356	OUI	géniteur	$\{c_{\#0}, c_{\#4}, c_{\#6}, c_{\#9}, c_{\#7}\}$
$\{c_{\#0}\}$	6	NON		$\{c_{\#4}, c_{\#6}, c_{\#9}, c_{\#7}\}$
$\{c_{\#4}\}$	16	OUI	géniteur	$\{c_{\#6}, c_{\#9}, c_{\#7}, c_{\#3}, c_{\#5}\}$
$\{c_{\#6}\}$	13	OUI	inchangé	$\{c_{\#9}, c_{\#7}, c_{\#3}, c_{\#5}, c_{\#8}\}$
$\{c_{\#9}\}$	13	OUI	inchangé	$\{c_{\#7}, c_{\#3}, c_{\#5}, c_{\#8}\}$
$\{c_{\#7}\}$	35	OUI	modifié	$\{c_{\#3}, c_{\#5}, c_{\#8}\}$
$\{c_{\#3}\}$	6	NON		$\{c_{\#5}, c_{\#8}\}$
$\{c_{\#5}\}$	1	NON		$\{c_{\#8}\}$
$\{c_{\#8}\}$	13	OUI	modifié	$\emptyset$

TAB. 5.3 – *Trace du filtrage des concepts de l'iceberg $\mathcal{L}_1^{0.2}$ lors de l'ajout de l'attribut h.*

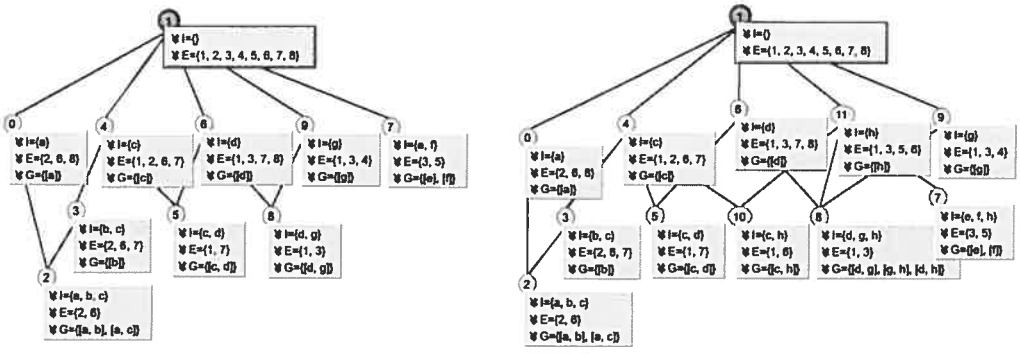

FIG. 5.2 – ***Gauche:*** *l'iceberg $\mathcal{L}_1^{0.2}$ associé à $\mathcal{K}_1$ =$(O, A = \{a, b, \cdots, g\}, I_1)$* ***Droite:*** *l'iceberg $\mathcal{L}_2^{0.2}$ associé à $\mathcal{K}_2$ =$(O, A = \{a, b, \cdots, h\}, I_2)$*

5.4 Complexité

La partie la plus coûteuse en terme de complexité, dans l'algorithme de mise à jour du treillis, avec maintien des générateurs, est celle du calcul des générateurs d'une classe d'équivalence. L'instruction dominante est la ligne 12 de l'algorithme 15 qui consiste à comparer chaque générateur minimal d'un concept donné c avec ceux de sa classe d'équivalence déjà calculés.

Soient $g(c)$ le nombre maximal des générateurs par concept et $Gen(\mathcal{L}_1)$ le nombre maximal des générateurs du treillis $\mathcal{L}_1$, cc le coût d'une comparaison entre deux générateurs, nc le nombre de comparaisons et $ct = cc \times nc$ le coût total de toutes les comparaisons.

- Calculer nc d'un générateur consiste à trouver le nombre des minimas de la classe d'équivalence du concept correspondant. On note *temp* l'ensemble de ces générateurs minimas. Comme le treillis $\mathcal{L}_1$ est parcouru de manière descendante, en largeur d'abord, chaque générateur ajouté à *temp* est définitivement minimal. En effet, lorsqu'un concept est ajouté à une classe d'équivalence, la taille de son intention est plus grande ou égale à celles de ceux déjà dans la même classe (le nouvel arrivé dans la classe n'est inclus dans aucun qui est déjà présent dans la classe). On peut prouver que ses générateurs sont de taille plus grande ou égale à celles des générateurs présents dans *temp* (sinon, on aurait une inclusion automatique des intentions respectives d'après la propriété de la monotonie de l'opérateur de fermeture ce qui contredit notre parcours du treillis). Ainsi, la taille maximale de *temp* est bornée par le nombre de générateurs minimaux de la classe d'équivalence qui correspond au nombre des générateurs minimaux du concept minimal de la classe. Finalement, nc est bornée par $g(c)$.
- Comparer deux générateurs est linéaire en fonction de leurs tailles, donc cc est borné par $|A|$.

En somme, le coût du traitement d'un générateur ct est borné par $|A| \cdot g(c)$. La complexité de la mise à jour de tous les générateurs est de l'ordre de $O(|A| \cdot g(c) \cdot Gen(\mathcal{L}_1))$.

Il est difficile de trouver une borne supérieure acceptable pour $g(c)$ et $Gen(\mathcal{L}_1)$. Toute-

fois, d'après [EG02], on sait que les générateurs sont les transversaux minimaux des hypergraphes [Ber89] des faces[1] respectives. De plus, le calcul des transversaux minimaux des hypergraphes est exponentiel en la taille des attributs [MR92].

Dans le cas de l'iceberg (Algorithme 16), comme les concepts ayant les plus grandes intentions ne sont pas calculés (infréquents), l'opération de mise à jour des générateurs est automatiquement moins coûteuse que celle du treillis complet (dans ce cas, la taille des faces est plus petite).

1. Étant donné un concept $c = (X,Y)$ et $Successeur_i(c)$ le ième successeur immédiat du concept c dans le treillis. La ième face du concept c correspond à la différence entre son intention et celle de son ième successeur immédiat

Chapitre 6

Expérimentations

Dans le but de mesurer les performances pratiques des algorithmes proposés, nous avons conduit une série de tests et de comparaisons. Ces algorithmes ont été implémentés au sein de GALICIA[1] [VGRH03], une plate-forme Java destinée à la manipulation des treillis de Galois et de leurs structures dérivées (icebergs, sous-hiérarchies de Galois).

Un facteur de comparaison entre les algorithmes est la sélection de jeux de données. Les bases de données denses se distinguent par le support élevé des motifs (les ensembles d'objets associés aux attributs sont de grandes tailles), alors que les bases éparses, sont caractérisées par des attributs de faible support (les ensembles d'objets associés aux attributs sont de tailles réduites). Dans [KO02], une étude comparative entre plusieurs algorithmes batch et incrémentaux a montré l'efficacité de l'approche incrémentale par objet dans le cas des données éparses tandis que les algorithmes batch se sont avérés plus aventageux dans le cas des données denses.

Dans ce chapitre, nous considérons une version batch de l'algorithme de maintenance d'un iceberg avec le maintien des générateurs. Le contexte est donné au départ et l'iceberg est construit en ajoutant les attributs un à un. Nous allons comparer cette implémentation à celle de TITANIC (considéré comme l'un des algorithmes batch les plus performants). Concernant l'implémentation de TITANIC, nous avons demandé une version à ses auteurs. Une fois testée

1. disponible sur le site http://galicia.sourceforge.net

Nom	Nbr d'objets	taille moyenne des objets	Nbr d'attributs	Description de la base
soyBean	455	36	151	contient des informations sur la plante soya. Les attributs décrivent de divers états de croissance de la plante et le type de maladie l'affectant.
Mushroom	8124	23	119	contient des informations sur les champignons. Les attributs sont les caractéristiques des champignons.
$T25I10D10k$	10000	25	10000	une base transactionnelle. Les attributs sont les produits offerts.

TAB. 6.1 – *Caractéristiques des jeux de données*

avec nos bases de données, nous avons constaté qu'elle n'était pas optimale. Pour cela, nous avons adapté à la plateforme GALICIA une version performante, fournie par Laszlo Szathmary[2].

6.1 Les jeux de données

Nous avons sélectionné des jeux de données fréquemment utilisés afin de comparer les algorithmes de l'ECD. Les caractéristiques de ces données sont présentées dans la table 6.1

soyBean[3] et *Mushroom*[4] sont des bases denses et $T25I10D10k$ [5] est éparse.

6.2 Première série de tests

Elle concerne la comparaison des algorithmes de construction de treillis complets, à savoir NEXT-CLOSURE (pour la construction des motifs fermés), INC-O (pour la maintenance du treillis par ajout d'un nouvel objet) et INC-A (pour la maintenance du treillis par ajout d'un

2. Etudiant en PhD au Loria
3. disponible sur le site http://www.cse.iitb.ac.in/dbms/Data/Software-Other/MLCPP-1.3.1/db
4. disponible sur le site http://deptinfo.unice.fr/ pasquier
5. disponible sur le site http://deptinfo.unice.fr/ pasquier

Nb Trans	Nb Concepts	Next-Closure	INC-A	INC-O
1000	32514	21	2	57
2000	58983	60	7	465
3000	80902	110	17	1300
4000	104105	196	30	4320

TAB. 6.2 – *Temps d'exécution (en minutes) et nombre des concepts sur des préfixes de la base Mushroom*

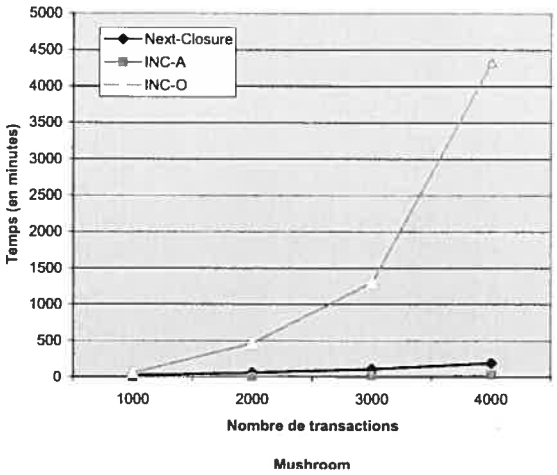

FIG. 6.1 – *Temps d'exécution pour la construction du treillis sur des préfixes de Mushroom*

nouvel attribut). Nous avons implémenté une version de NEXT-CLOSURE pour la construction du treillis de concepts. Nous avons remarqué que la partie destinée à la création de la relation de couverture entre concepts est négligeable par rapport à celle de la construction des concepts. Pour cela, nous n'avons pas considéré cette méthode dans les comparaisons.

6.2.1 Résultats

Les résultats des tests effectués sur *Mushroom* sont présentés dans la table 6.2

Nb Trans	Nb Concepts	INC-O	INC-A
1000	10814	30	75
1500	18601	65	185
2000	28309	135	400

TAB. 6.3 – *Temps d'exécution (en minutes) et nombre des concepts sur des préfixes de la base* $T25I10D10k$

En termes de temps d'exécution, NEXT-CLOSURE est efficace car il génère une seule fois les motifs fermés. Il examine les motifs susceptibles d'être fermés. Nous avons constaté dans nos tests que le nombre des motifs examinés dépend de la nature de la relation binaire. En effet, même avec un préfixe de la base $T25I10D10k$ formé des 1000 premières transactions, NEXT-CLOSURE a pris beaucoup de temps ce qui nous a conduit à interrompre son exécution. Pour cela les résultats sont absents. Avec la base *Mushroom*, NEXT-CLOSURE a été plus performant que INC-O.

Dans la base $T25I10D10k$, les algorithmes incrémentaux sont plus efficaces vu le petit nombre de nouveaux concepts générés lors du passage du treillis $\mathcal{L}_i$ (état initial) au treillis $\mathcal{L}_{i+1}$ (état final) par ajout d'un nouvel objet/attribut. À titre d'exemple, nous avons considéré la maintencance du treillis par ajout des 2000 premières transactions. Dans les deux algorithmes (INC-O et INC-A), le nombre maximal de concepts générés lors du passage de l'état initial à l'état final ne dépasse pas les 20 concepts. La table 6.3 montre les résultats des deux algorithmes incrémentaux sur un préfixe de la base $T25I10D10k$.

Les résultats de la table 6.3 montre que INC-O est plus efficace que INC-A. En effet, le nombre des attributs est plus grand que celui des objets et le calcul des classes d'équivalence dans le cas de l'incrémentalité objet se fait par l'intersection des intentions des concepts (qui ont une taille maximale d'une vingtaine d'attributs dans cet exemple) alors que dans le cas de l'incémentalité attribut, l'intersection des extentions est plus coûteuse (les extentions comportent des milliers d'objets).

Support	Nb Concepts	Titanic	Magalice-A-GEN	Magalice-O
1%	25043	1260	180	18000
2%	14454	540	120	7500
3%	9999	300	90	5040
4%	7399	180	60	4200

TAB. 6.4 – *Temps d'exécution (en secondes) sur une partie de la base Mushroom avec variation des supports*

Dans la base *Mushroom*, le nombre de nouveaux concepts générés est très grand (voir table 6.2). Ainsi, INC-O est moins efficace à cause du nombre de répétitions de ses opérations et de la mise à jour de l'ordre (le nombre des concepts de la couverture supérieure est borné par le nombre d'objets). La rapidité de la méthode INC-A est justifiée par un nombre moins grand d'itérations et par un temps de mise à jour de l'ordre meilleur (la taille de la couverture inférieure dans ce cas est bornée par le nombre des attributs).

6.3 Deuxième série de tests

La deuxième série de tests concerne la comparaison des algorithmes de construction des icebergs, c.à.d TITANIC (pour la construction des motifs fermés fréquents), MAGALICE-O (pour la maintenance des icebergs par ajout d'un nouvel objet) et MAGALICE-A-GEN (pour la maintenance des icebergs et des générateurs par ajout d'un nouvel attribut).

6.3.1 Résultats

Nous avons lancé les trois algorithmes dans GALICIA. Pour la base *Mushroom*, nous avons considéré les 4000 premières transactions et nous avons varié les supports minimaux. Les résultats sont présentés dans la table 6.4

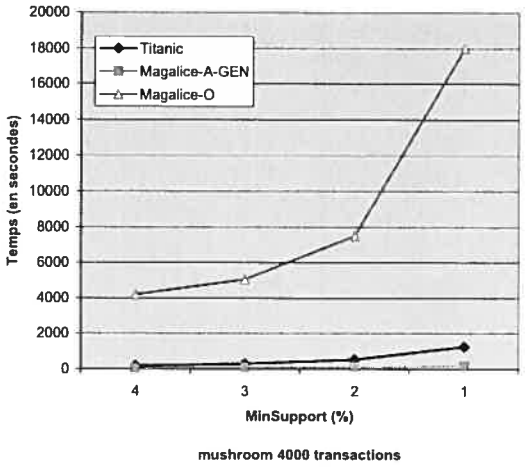

FIG. 6.2 – *Graphe du temps d'exécution des icebergs lancés sur la base dense*

Support	Nb Concepts	Nb Générateurs	Magalice-A-GEN	Titanic
20%	168216	248245	7260	19200
30%	61582	81105	1020	5460
40%	21190	25015	175	1560
50%	6058	6768	59	420
60%	1490	1601	8	60

TAB. 6.5 – *Temps d'exécution (en secondes) sur la base soyBean avec variation des supports*

Comme notre but est de comparer la maintenance des générateurs, nous avons lancé les deux algorithmes MAGALICE-A-GEN et TITANIC dans la plateforme sur la base *soyBean* et nous avons varié le support minimal de 30 à 60%. Les résultats sont présentés dans la table 6.5. Ensuite, nous avons implémenté une version indépendante de MAGALICE-A-GEN et nous l'avons comparé avec celle de TITANIC sur toute la base *Mushroom* en construisant le treillis complet (le cas où l'on a tous les générateurs). La table 6.6 montre les résultats issus de cette comparaison.

Pour la base T25I10D10K, nous présentons seulement un résultat (voir table 6.7) pour montrer la grande différence entre les temps d'exécution de MAGALICE-A-GEN et TITANIC.

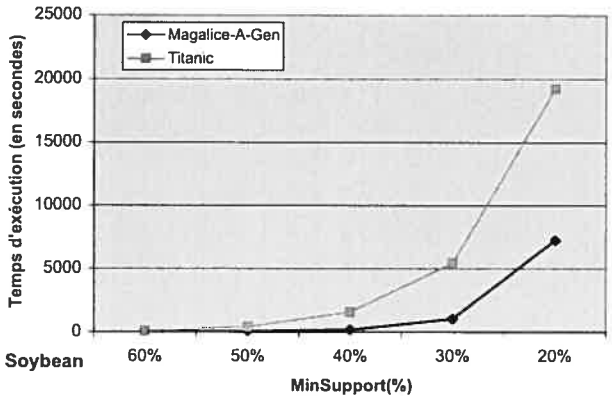

FIG. 6.3 – *Graphe du temps d'exécution des icebergs lancés sur la base dense*

Nb Objets	Nb Concepts	Nb Générateurs	Magalice-A-GEN	Titanic
2000	58983	120820	51	1500
4000	104105	211423	180	5100
6000	156574	326783	550	12700
8124	238710	503016	2243	19200

TAB. 6.6 – *Temps d'exécution (en secondes) sur la base Mushroom avec variation des supports*

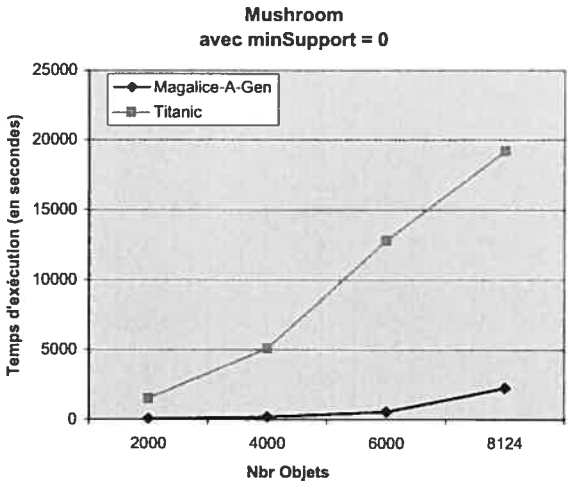

FIG. 6.4 – *Graphe du temps d'exécution des icebergs lancés sur la base dense*

Support	Nb Concepts	Titanic	Magalice-A-GEN
1%	4949	15840	240
2%	128	30	13

TAB. 6.7 – *Temps d'exécution (en secondes) sur la base* T25I10D10K *avec variation des supports*

Pour un support inférieur à 1 %, TITANIC s'avère très lent, et la disparité dans le temps d'exécution des deux algorithmes devient marquante.

6.3.2 Interprétation des résultats

Les variances des performances des algorithmes suivant les types de données pourront être expliquées comme suit:

Pour la base $T25I10D10k$ qui est éparse, le nombre des générateurs minimaux est sensiblement le même que celui des motifs fermés. L'inefficacité de TITANIC provient du grand nombre des attributs à considérer dans le calcul de la fermeture d'un générateur.

Les algorithmes incrémentaux sont avantageux dans cette base. En effet, pour MAGALICE-O, comme le nombre des concepts nouveaux et modifiés générés lors de la première phase (de l'algo) est faible, alors la procédure FIND-FREQUENT-LOWER-COVERS n'est pas très coûteuse. Contrairement à TITANIC, le nombre des attributs à considérer dans le calcul est petit. Ce qui explique l'avantage de cette méthode. Pour MAGALICE-A-GEN, la mise à jour de l'iceberg opère un filtrage des concepts en plus de la mise à jour du treillis complet (un simple test). L'efficacité de cet algorithme réside dans le petit nombre de concepts créés à chaque ajout d'un attribut. En ce qui concerne la maintenance des générateurs, puisque leur nombre est presque égal à celui des concepts, alors le nombre de comparasion à faire entre un générateur et ceux de sa classe d'équivalence déjà calculé est très petit. Ce qui explique une meilleure performance de MAGALICE-A-GEN par rapport à TITANIC.

Le résultat le plus impressionnant est celui de la performance de MAGALICE-A-GEN pour les deux bases *Mushroom* et *soyBean* qui sont denses. L'efficacité de cet algorithme et son

avantage sur TITANIC peut être expliqué comme suit:

Dans MAGALICE-A-GEN, on calcule les concepts d'abord, et ensuite on calcule leurs générateurs. L'avantage de cette méthode réside dans le fait qu'un concept est généré une seule fois durant toute la maintenance de l'iceberg. TITANIC, quant à lui, calcule les concepts en faisant la fermeture de leurs générateurs correspondants. Puisqu'un concept peut avoir plusieurs générateurs, il sera, dans ce cas, généré un nombre de fois égal au nombre de ses générateurs.

De plus, dans les deux bases denses, le nombre de classes d'équivalence est grand, la taille des générateurs minimaux est petite et le nombre des générateurs par concept n'est pas grand. Le premier concept trouvé dans une classe a la taille de son intention la plus petite (dans la classe) et par la suite ses générateurs sont aussi minimaux de la classe. À chaque concept ajouté à sa classe, chacun de ses générateurs devrait être comparé aux générateurs de la classe. Pour le concept ajouté, La taille de ses générateurs, le nombre de ses générateurs et le nombre de générateurs de la classe assurent une opération de comparaison entre deux générateurs non coûteuse.

Pour TITANIC, le calcul de la fermeture d'un générateur est plus coûteuse à cause du nombre de ses successeurs à considérer. Vu le nombre des générateurs, le temps total du calcul de toutes les fermetures effectuées par TITANIC devient grand.

Pour MAGALICE-O, la procédure FIND-FREQUENT-LOWER-COVERS est coûteuse. En effet, le grand nombre de concepts (modifiés et nouveaux) générés dans la première phase de l'algorithme principal entraîne un calcul important d'intersection d'extentions de concepts même après la réduction du nombre des attributs candidats. Ce qui rend son temps d'exécution moins efficace que les deux autres algorithmes.

Chapitre 7

Conclusion

La quantité de redondance dans l'extraction des règles d'association a conduit à la création de bases informatives par application de l'AFC et plus précisement du calcul des générateurs d'un motif fermé. Les algorithmes existants actuellement pour cette opération ont des performances pas toujours intéressantes ce qui nous a poussé à développer une nouvelle méthode de calcul des générateurs.

Récemment, l'approche incrémentale par ajout d'un objet a été développée et testée. Les résultats ayant montré que les algorithmes batch sont plus performants même considérant une reconstruction complète, nous avons essayé de développer l'approche duale.

D'un point de vue théorique, cette approche surpasse l'approche incrémentale par objet et rejoint les algorithmes batchs. Du point de vue pratique, les tests que nous avons réalisés ont montré que notre approche a de meilleures performances que les algorithmes incrémentaux objets et batchs.

Notre contribution s'insère dans les lignes de recherche décrites à la table 1.1. La table résultante est présentée dans la table 7.1

Plusieurs perspectives peuvent être envisagées, par exemple, l'exploitation des résultats de C. Frambourg sur la fusion des treillis [VML02] pour concevoir une méthode efficace pour la mise à jour des générateurs par la fusion de treillis suite à l'ajout de nouveaux attributs au contexte

	Incrémentalité par objet		Incrémentalité par attribut	
Tâches	**Ajout 1 obj**	**Ajout n obj**	**Ajout 1 att**	**Ajout m att**
MAJ des motifs fermés	C.Frambourg [a]	C.Frambourg	K.Nehmé [b]	C.Frambourg
MAJ d'un iceberg	J.Jing [c] + K.Nehmé		K.Nehmé	
MAJ des générateurs	C.Frambourg	C.Frambourg	K.Nehmé	
MAJ (iceberg + générateurs)	C.Frambourg		K.Nehmé	

TAB. 7.1 – *Application de l'approche incrémentale pour la maintenance des icebergs et des générateurs des motifs fermés fréquents à l'ARM.*

[a] Céline Frambourg [Fra04]
[b] Kamal Nehmé
[c] Jun Jing [Jin04]

ou l'adaptation des travaux de fusion aux icebergs.

Annexe A

Preuves des propriétés de la MAJ de la famille des générateurs

Propriété 31

$\forall\ c \in \mathrm{M}_2(\mathrm{a})$, on a:

$$Gen_2(c) = Gen_1(\gamma_a(c)) \cup min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1}\ et\ \hat{c} \neq \gamma_a(c)} Gen_1(\hat{c})) \times \{a\}$$

Preuve. ($\subseteq$). Soit $G \in Gen_2(c)$. 2 cas se présentent:

- $a \notin G$ montrons alors que $G \in Gen_1(\gamma_a(c))$. Raisonnons par l'absurde. Supposons que $G \notin Gen_1(\gamma_a(c))$. Il existe alors $F \subset G$ t.q $\mathrm{F}^{11} = Y$. En outre, on sait que $\mathrm{F}^1 = \mathrm{F}^2 = X$ (d'après $a \notin F$). Ainsi, on aurait $\mathrm{F}^{22} = \mathrm{X}^2 = \mathrm{Y} \cup \mathrm{a} = \mathrm{G}^{22}$. Ce qui contredit $G \in Gen_2(c)$.
- $a \in G$. On peut écrire $G = \hat{G} \cup a(a \notin \hat{G})$. Montrons que $\hat{G} \in min(\bigcup_{\hat{c} \in [c]_R\ et\ \hat{c} \neq c} Gen_1(\hat{c}))$

  En effet, on a: $G^{22} = Y\ \cup\ a,\ \ G^2 = (Y\ \cup\ a)^2$.

  $\Rightarrow (\hat{G} \cup a)^2 = (Y\ \cup\ a)^2$ (d'après l'hypothèse $G = \hat{G} \cup a$)

  $\Rightarrow \hat{G}^2 \cap a^2 = (Y\ \cup\ a)^2$ (d'après la propriété $(A \cup B)' = A' \cap B'$).

  De plus, $\hat{G}^2 = \hat{G}^1\ (a \notin \hat{G}) \Rightarrow \hat{G}^1 \cap a^2 = (Y\ \cup\ a)^2 = X$

  Donc, il existe un concept $\hat{c} = (\hat{G}^1, \hat{G}^{11}) \in [\gamma_a(c)]_{R_1}$. Montrons que $\hat{G} \in Gen_1(\gamma_a(c))$ et que $\hat{G}$ est minimal dans $[\gamma_a(c)]_{R_1}$.

Supposons le contraire:

- $\hat{G} \notin Gen_1(\gamma_a(c))$. Donc, $\exists\ F \subset \hat{G}$ t.q $F^{11} = \hat{G}^{11}$ et par suite $F^1 = \hat{G}^1$. De plus, $F^1 = F^2$ et $\hat{G}^1 = \hat{G}^2$. On aura par suite, $F^2 \cap a^2 = \hat{G}^2 \cap a^2 = X$. $\Rightarrow (F \cup a)^{22} = X^2 = (Y \cup a)$. ce qui contredit G est un générateur minimal de c.
- $\hat{G}$ n'est pas minimal dans $[\gamma_a(c)]_{R_1}$. Donc, $\exists\ F \subset \hat{G}$ t.q $F^2 \cap a^2 = \hat{G}^2 \cap a^2 = X$. D'après la condition précédente, on aurait une contradiction avec G est un générateur de c.

Il reste à montrer que $\hat{c} \neq \gamma_a(c)$. En effet, supposons le contraire. $\hat{c} = \gamma_a(c) \Rightarrow \hat{G}^{11} = Y$. Donc $\hat{G} \in Gen_1(\gamma_a(c))$ et par suite $\hat{G} \in Gen_2(c)$ (D'après la propriété 28)

Ce qui entraine $\hat{G}^2 = X^2 = (Y \cup a)$. Contradiction avec $(\hat{G} \cup a)$ est un générateur de c.

$(\supseteq)$.

- Si $G \in Gen_1(\gamma_a(c))$ alors $G \in Gen_2(c)$ (D'après la propriété 28).
- Si $G \in min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1} \ et\ \hat{c} \neq \gamma_a(c)} Gen_1(\hat{c})) \times \{a\}$ on a alors $G = \hat{G} \cup a$

  ($\hat{G} \in min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1} \ et\ \hat{c} \neq \gamma_a(c)} Gen_1(\hat{c}))$

  $\Rightarrow G^2 = (\hat{G} \cup a)^2 = \hat{G}^2 \cap a^2 = \hat{G}^1 \cap a^2 = X$ et par suite $G^{22} = X^2 = Y \cup a$.

  Montrons que $G \in Gen_2(c)$. Raisonnons par l'absurde. $\exists\ F \subset G$ t.q $F^{22} = Y \cup a$.

  Comme $F \subset \hat{G} \cup a$, deux cas se présentent:

  - $a \in F \Rightarrow F = \hat{F} \cup a \subset \hat{G} \cup a \Rightarrow \hat{F} \subset \hat{G}$

    $F^2 = X \Rightarrow (\hat{F} \cup a)^2 = X \Rightarrow \hat{F}^2 \cap a^2 = X$.

    $\Rightarrow \hat{F}^1 \cap a^2 = X$ (a $\notin \hat{F} \Rightarrow \hat{F}^1 = \hat{F}^2$)

    $\Rightarrow \hat{c_1} = (\hat{F}^1, \hat{F}^{11}) \in [\gamma_a(c)]_{R_1}$ t.q $\hat{F} \in Gen^1(\hat{c_1})$

    Ainsi, on a trouvé un concept dans $[\gamma_a(c)]_{R_1}$ ayant comme générateur $\hat{F} \subset \hat{G}$ ce qui contredit $\hat{G} \in \min(\text{Gen}_1(\hat{c}))\ \hat{c} \in [\gamma_a(c)]_{R_1}$.
  - $a \notin F$ alors $F \subset \hat{G}$.

    $F^2 = X \Rightarrow F^2 \cap a^2 = X \cap a^2 = X \Rightarrow F^1 \cap a^2 = X$. Contradiction (même raisonnement que le cas précédent).

■

Propriété 32 $\forall c \in \mathbf{N}_2(a)$. On a:

$$Gen_2(c) = min(\bigcup_{\hat{c} \in [\gamma_a(c)]_{R_1}} Gen_1(\hat{c})) \times \{a\}$$

Preuve. ($\subseteq$). Soit $G \in Gen_2(c) \Rightarrow G = \hat{G} \cup a$ (d'après la remarque $a \in G$)

$\Rightarrow G^2 = X = (\hat{G} \cup a)^2 = \hat{G}^2 \cap a^2 = \hat{G}^1 \cap a^2$ ($a \notin \hat{G} \Rightarrow \hat{G}^1 = \hat{G}^2$). $\Rightarrow \hat{G}^1 \cap a^2 = X$.

Donc, il existe un concept $\hat{c} = (\hat{G}^1, \hat{G}^{11}) \in [(\gamma_a(c))$.

Montrons que $\hat{G} \in Gen_1(\hat{c})$ et que $\hat{G}$ est minimal dans $[(\gamma_a(c))]_{R_1}$. Raissonnons par l'absurde. Soit $\hat{F} \subset \hat{G}$ t.q $\hat{F} \in [(\gamma_a(c))]_{R_1}$ et $\hat{F}^{11} = Y - a$. On a $\hat{F}^1 \cap a^2 = X \Rightarrow \hat{F}^2 \cap a^2 = X \Rightarrow (\hat{F} \cup a)^2 = X \Rightarrow (\hat{F} \cup a)^{22} = Y \Rightarrow \exists\ F = (\hat{F} \cup a) \subset G$ t.q $F^{22} = Y$ ce qui contredit G générateur de c.

($\supseteq$). Soit $G \in (2)$. $G = \hat{G} \cup a$ t.q $\hat{G} \in [(\gamma_a(c)]_{R_1}) \Rightarrow G^2 = (\hat{G} \cup a)^2 = \hat{G}^2 \cap a^2 = \hat{G}^1 \cap a^2 = X \Rightarrow G^{22} = X^2 = Y$

G est minimal sinon $\exists F \subset G$ t.q $F^{22} = Y$.

$F \subset G \Rightarrow F \subset \hat{G} \cup a$. D'après la remarque, $a \in F$. $\Rightarrow F = \hat{F} \cup a$. $F^{22} = Y \Rightarrow (\hat{F} \cup a)^{22} = Y$ et $(\hat{F} \cup a)^2 = X \Rightarrow \hat{F}^2 \cap a^2 = X \Rightarrow \hat{F}^1 \cap a^2 = X$ ce qui contredit $\hat{G} \in \min(\mathrm{Gen}^1(\hat{c}))$ ($\hat{c} \in [(\gamma_a(c))]_{R_1}$). ■

Bibliographie

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, September 1994.

[Ber89] C. Berge. Hypergraphs: Combinatorics of finite sets. North Holland, Amsterdam, 1989.

[Bir40] G. Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloquium Publications*. AMS, 1940.

[BM70] M. Barbut and B. Monjardet. *Ordre et Classification: Algèbre et combinatoire*. Hachette, 1970.

[BMUT97] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 255–264, 1997.

[DP92] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1992.

[EG02] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 549–564. Springer-Verlag, 2002.

[Fra04] C. Frambourg. *Etude algorithmique sur l'évolution des générateurs minimaux dans un espace de fermeture suite à un raffinement*. Thèse de maitrise, Université de Québec à Montréal, Décembre 2004.

[Gan84] B. Ganter. Two basic algorithms in concept analysis. preprint 831, Technische Hochschule, Darmstadt, 1984.

[GMA95] R. Godin, R. Missaoui, and H. Alaoui. Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations.* Springer-Verlag, 1999.

[Jin04] J. Jing. *Algorithms for on-line computation of iceberg lattices.* Master's thesis, Université de Montréal, October 2004.

[KO02] Sergei O. Kuznetsov and Sergei A. Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2-3):189–216, 2002.

[MR92] H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40(2):237–243, 1992.

[NR99] L. Nourine and O. Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71:199–204, 1999.

[NVRG05] K. Nehme, P. Valtchev, M.H. Rouane, and R. Godin. On computing the minimal generator family for concept lattices and icebergs. In *International Conference on Formal Concept Analysis (ICFCA'2005)*, Lens,France, 2005.

[Pas00] N. Pasquier. *Data mining : algorithmes d'extraction et de réduction des règles d'association dans les bases de données.* Thèse de doctorat, Université de Clermont-Ferrand II, Janvier 2000.

[PBTL99] N. Pasquier, Y. Bastide, T. Taouil, and L. Lakhal. Efficient Mining of Association Rules Using Closed Itemset Lattices. *Information Systems*, 24(1):25–46, 1999.

[PHM00] J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In *Proceedings, ACM SIGMOD Workshop DMKD'00*, pages 21–30, Dallas (TX), USA, 2000.

[PsF91] G. Piatetsky-shapiro and W.J. Frawley. Knowledge discovery in databases. In *AAAI/MIT Press*, 1991.

[RNVG04] M.H. Rouane, K. Nehme, P. Valtchev, and R. Godin. On-line maintenance of iceberg concept lattices. In *Contributions to the 12th ICCS*, Hunstville (AL), 2004. Shaker Verlag.

[SON95] A. Savasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st Int'l Conf. on Very Large Databases(VLDB)*, pages 432–444, Septembre 1995.

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

[Tao00] R. Taouil. *Algorithmique du treillis des fermés: application à l'analyse formelle de concepts et aux bases de données.* Thèse de doctorat, Université Blaise-Pascal, Janvier 2000.

[VGRH03] P. Valtchev, D. Grosser, C. Roume, and M. Rouane Hacene. GALICIA: an open platform for lattices. In B. Ganter and A. de Moor, editors, *Using Conceptual Structures: Contributions to 11th Intl. Conference on Conceptual Structures (ICCS'03)*, pages 241–254, Aachen (DE), 2003. Shaker Verlag.

[VHM03] P. Valtchev, M. Rouane Hacene, and R. Missaoui. A generic scheme for the design of efficient on-line algorithms for lattices. In B. Ganter A. de Moor, W. Lex, editor, *Proceedings of the 11th Intl. Conference on Conceptual Structures (ICCS'03)*, volume 2746 of *Lecture Notes in Computer Science*, pages 282–295, Berlin (DE), 2003. Springer-Verlag.

[VMGM02] P. Valtchev, R. Missaoui, R. Godin, and M. Meridji. Generating Frequent Itemsets Incrementally: Two Novel Approaches Based On Galois Lattice Theory. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):115–142, 2002.

[VML02] P. Valtchev, R. Missaoui, and P. Lebrun. A partition-based approach towards building Galois (concept) lattices. *Discrete Mathematics*, 256(3):801–829, 2002.

[VMRG03] P. Valtchev, R. Missaoui, M. H. Rouane, and R. Godin. Incremental maintenance of association rule bases. In *SIAM Workshop on Discrete Mathematics and Data Mining, San Fransisco, USA*, 2003.

[Wil82] R. Wille. Restructuring the lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered sets*, pages 445–470, Dordrecht-Boston, 1982. Reidel.

[Zak00] M.J. Zaki. Generating Non-Redundant Association Rules. In *Proceedings, KDD-00*, pages 34–43, Boston (MA), USA, 2000.

[ZH99] M.J. Zaki and C.-J. Hsiao. ChARM: An Efficient Algorithm for Closed Association Rule Mining. RPI Technical Report 99-10, Rensselaer Polytechnic Institute, 1999.