UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

APPROXIMATION DU TREILLIS DE CONCEPTS POUR LA FOUILLE DE DONNÉES

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

GANAËL JATTEAU

NOVEMBRE 2005

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

APPROXIMATION DU TREILLIS DE CONCEPTS POUR LA FOUILLE DE DONNÉES

présenté par
Ganaël Jatteau

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Rokia Missaoui ..... Directrice de recherche
Jurek Czyzowicz ..... President du jury
Nadia Baaziz ..... Membre du jury

Mémoire accepté le : 16 novembre 2005

J'adresse mes remerciements à ma famille pour son soutien et sa confiance,
à Rokia Missaoui pour sa gentillesse et sa forte contribution à ce mémoire,
ainsi qu'à toute l'équipe du laboratoire LaRIM dont j'ai beaucoup apprécié la compagnie.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations, sigles et acronymes

AFC Analyse Formelle de concepts

CIGA+ Closed Itemset Generation and Approximation

MF Motif Fermé

MFF Motif Fermé Fréquent

OLAP *On-Line Analytical Processing*

Résumé

La prospection de données ("*data mining*") est un processus de recherche de la structure cachée des données sous forme de liens, concepts, classes ou patrons ("*patterns*"). Elle fait l'objet d'intenses études théoriques et applications pratiques comme l'analyse du panier du consommateur, la gestion de la relation client, et la découverte de connaissances à partir d'une base de données multimédia. Ce projet de recherche traite plus particulièrement des treillis de concepts (appelé aussi treillis de Galois).

Puisque la prospection de données est utilisée comme support à la prise de décision et que les analystes sont rarement intéressés par la liste exhaustive (souvent très longue) des concepts et règles, l'élaboration d'une solution qui réduit la taille du résultat sera dans la plupart des cas un compromis satisfaisant permettant de se focaliser sur l'information essentielle. Pour faire face à ce problème, nous abordons deux approches différentes. La première consiste à réduire le nombre de concepts grâce à des techniques d'approximation. La deuxième approche revient à mettre en place des mécanismes pour manipuler aisément l'information générée à l'aide d'opérateurs de manipulation de treillis de concepts.

Abstract

Data mining is an interesting framework for conceptual clustering and association rule mining. It is used in many theorical and practical applications as market basket analysis or knowledge discovery from a multimedia database. In this document, we focus particulary on concept lattices, an important step for data mining and knowledge discovery. Since the output of a data mining task can be very large, even for a small dataset, analysts are rarely interested with a complete set of rules and concepts. Reducing the number of concepts is then a valuable approach which allow to focus on relevant information more easily. Basically, we consider two approaches : one consists in reducing the number of concepts by using some approximation, an other defines some operators to manipulate the data mining output.

Chapitre 1

Introduction

Ce projet de recherche se situe dans le cadre de la fouille de données (*data mining*) et traite plus particulièrement de l'Analyse Formelle de Concepts (*Formal Concept Analysis*). L'AFC est une branche de la théorie des treillis qui permet la génération de concepts, de treillis de concepts et à partir de là des règles d'associations. Un concept formel est un couple complet qui associe un ensemble d'objets (extension) à un ensemble d'attributs (intention) permettant ainsi de regrouper les objets qui ont des caractéristiques communes. L'AFC est un domaine de recherche actif qui évolue énormément chaque année à cause de son utilité grandissante. Même si l'AFC semble abstraite et très théorique, elle fait l'objet de plusieurs applications dans des domaines variés (réutilisation en génie logiciel, repérage de l'information, applications web). Le treillis de concepts est utile dès lors qu'on souhaite explorer de l'information volumineuse pour en extraire des patrons types. C'est le cas lorsqu'on cherche à découvrir des comportements très communs ou bien exceptionnels chez des groupes d'individus. L'AFC peut aussi bien servir à faire de la recherche d'images basée sur le contenu, découvrir de l'information sur une population ou bien classer des données dans un entrepôt. Toutes ces applications sont déjà réalisables grâce aux nombreux algorithmes existants pour générer les concepts, le treillis de concepts et les règles d'associations issus d'une base de données.

Même si plusieurs travaux ont proposé des algorithmes performants, un problème demeure quant à la quantité d'information générée. Le nombre de concepts issus d'une base de données peut être exponentiel par rapport à la taille des données initiales. Même pour des données peu volumineuses, le résultat peut devenir vite indigeste et difficile à manipuler. Ainsi, dans plusieurs applications de prospection de données, la produc-

tion d'un ensemble *exhaustif* de connaissances (règles d'association, concepts) peut être très coûteuse et comporter plusieurs éléments absolument pas pertinents pour un utilisateur donné. Aussi, il serait avantageux d'offrir des mécanismes de génération d'un sous-ensemble de ces connaissances qui pourraient si nécessaire inciter l'utilisateur soit à solliciter l'affichage d'autres connaissances ou à demander des détails sur les associations et les concepts issus d'un ensemble plus restreint de données. Ce constat nous conduit à considérer deux alternatives de production d'un ensemble concis de connaissances. Puisque toute l'information produite n'est pas pertinente, la première idée est d'offrir des mécanismes d'approximation afin de réduire le temps d'exécution des algorithmes et la taille du résultat tout en essayant de garder autant que possible l'information pertinente. La deuxième idée revient à proposer un ensemble d'opérateurs pour manipuler le vaste ensemble de connaissances qui a été généré et permettre de fournir à l'utilisateur des vues condensées sur un sous-ensemble du problème.

Dans une démarche similaire d'exécution efficace de la première étape du processus de production des règles d'association, nous proposons une approche d'approximation du treillis de concepts (algorithme LatMat) ainsi qu'une nouvelle méthode de génération complète ou partielle des concepts du treillis (algorithme CIGA+). Cette dernière solution est basée sur la construction et l'exploration d'un graphe de dépendances en vue de produire à un coût relativement faible un ensemble non nécessairement exhaustif de concepts. La sélection des concepts à garder se base non seulement sur le support des concepts (nombre d'objets possédant l'intention du concept), mais également sur les fréquences de cooccurrence d'attributs.

Ce document est organisé de la façon suivante. Tout d'abord, ce mémoire fournit quelques bases théoriques sur l'AFC puis traite l'état de l'art en décrivant brièvement quelques algorithmes de génération de concepts et de construction du treillis de concepts. Ensuite nous verrons l'algorithme LatMat qui génère un treillis de manière approximative. Le chapitre suivant étudie l'information perdue dans LatMat pour nous amener finalement à l'algorithme CIGA+. Cet algorithme se base sur trois paramètres de seuil et génère un ensemble complet ou approximatif des concepts du treillis. Le chapitre suivant aborde le problème d'une autre façon en proposant des opérateurs de manipulation du treillis inspirés de l'algèbre relationnel. Finalement, une conclusion est donnée dans le dernier chapitre.

Chapitre 2

Analyse formelle de concepts

L'analyse formelle de concepts [8] est une branche de la théorie des treillis basée sur la formalisation de la notion de concepts et de regroupement conceptuel. Elle permet, entre autres, la construction du treillis de concepts. Ce dernier s'est avéré être un cadre théorique intéressant pour la fouille de données puisqu'il permet la génération de concepts (regroupement conceptuel) et de règles d'association [25, 18, 38].

L'analyse formelle de concepts est un domaine de recherche très large. Dans ce document nous allons nous concentrer essentiellement sur les éléments qui permettent de définir et de construire un treillis de concepts ou bien de générer seulement les concepts.

Pour situer le problème, la construction du treillis de concepts et son exploitation représente la seconde étape du processus de la découverte de connaissances ("knowledge discovery") lequel comporte trois étapes :

1. Prétraitement des données
2. Fouille des données
3. Interprétation des résultats

Le prétraitement des données consiste principalement à sélectionner l'ensemble des données à traiter, traiter les données manquantes ou incohérentes et à effectuer des transformations de données. La construction du treillis quant à elle fait l'objet de ce document. L'interprétation des résultats représente un domaine à part et n'est pas décrite dans ce rapport. Elle fait appel aux techniques de visualisation permettant d'afficher les résultats d'une manière compréhensible par l'être humain.

2.1 Contexte formel

Avant de commencer l'étude sur les treillis de concepts, il est nécessaire de s'intéresser aux données d'entrée d'un algorithme de *"data mining"* représentées sous forme de *contexte formel*. Un contexte formel définit un ensemble d'objets O qui possèdent des propriétés (attributs). Il est représenté sous forme d'un tableau dont les objets sont en ligne et les attributs en colonne. Les valeurs des cellules sont de la forme $\langle attribut, valeur \rangle$; c.-à-d. que la cellule (i, j) du tableau contient la valeur de l'attribut j pour l'objet i. Dans ce document, nous utilisons un cas particulier du contexte formel où la valeur des attributs est booléenne. La table 2.1 représente un exemple de contexte formel ayant des valeurs booléennes. Cet exemple servira à illustrer plusieurs algorithmes.

Le fait d'utiliser des attributs (*items*) booléens peut paraître une contrainte pour la modélisation de problèmes généraux. Cependant, grâce aux opérations de discrétisation et binarisation, les données représentées sous forme $\langle attribut, valeur \rangle$ peuvent être converties en données booléennes moyennant une certaine perte d'information. C'est le cas des données stockées sous forme matricielle (bases relationnelles, images, etc...).

Objets \ Attributs	a	b	c	d	e	f
1	x	x	x	x	x	
2	x		x			
3		x	x			
4	x	x		x		
5			x	x		
6		x		x		
7				x		x

TAB. 2.1 – Exemple de contexte formel.

Définition 1 ***Contexte formel :*** *Le contexte formel K, permettant de représenter la relation binaire I entre les objets de l'ensemble O et les attributs de l'ensemble A, est défini par :*

$$K = (O, A, I)$$

Chaque couple de la relation binaire I, associant un attribut $a \in A$ et un objet $o \in O$ est représenté par : oIa.

Il y a de nombreuses façons d'illustrer l'utilisation d'un contexte formel. L'exemple le plus populaire est celui du `panier de la ménagère` qui consiste à enregistrer l'ensemble des articles présents dans le panier d'un consommateur. Dans cet exemple, chaque objet est un client (ou modèle de client) dont les attributs sont des articles achetés. C'est pourquoi on parle très souvent d'*item* pour désigner un attribut. Les premiers algorithmes de fouille de données ont fonctionné sur de tels exemples afin de découvrir des associations cachées entre les articles achetés ensemble. En extrayant des groupements d'articles achetés par une même personne, il est possible de déterminer plusieurs motifs (*patterns*) types qui résument des comportements d'achat généraux. Dans la pratique, beaucoup de problèmes se modélisent par un contexte formel. Toute information contenue dans une base ou entrepôt de données est discrétisable pour ensuite former une relation binaire possédant des objets et des attributs. L'étape de prétraitement des données consiste à adapter un problème pour le transformer sous forme de contexte qui a du sens et à partir duquel on peut extraire de l'information. Les spécialistes considèrent souvent cette étape comme étant la plus complexe du processus d'extraction de l'information. Il s'agit en fait de déterminer quel est le bon ensemble d'attributs booléens (*items*) qui peut représenter efficacement un problème.

Un contexte peut être vu sous la forme d'une matrice dont les coefficients sont les relations entre objets et attributs. On parle de dualité sémantique entre les objets et les attributs au niveau de la fouille puisqu'il est possible de transposer un contexte, tout comme on transpose une matrice, pour que les objets aient la position des attributs et vice versa. Il est donc possible d'avoir une approche objet ou bien attribut selon les besoins. Le résultat est toujours le même quelque soit l'approche.

L'intérêt d'utiliser une démarche plutôt qu'une autre va dépendre du nombre et de la taille des vecteurs que l'on souhaite manipuler. L'approche favorisée dépend de l'algorithme utilisé sachant qu'en général, le nombre d'objets est nettement supérieur au nombre d'attributs (les objets étant des enregistrements) et beaucoup d'algorithmes manipulent des vecteurs d'attributs pour traiter des données moins volumineuses. L'approche objet est donc l'approche utilisée par défaut et c'est pourquoi on préfère représenter les objets en ligne et les attributs en colonne dans un contexte formel.

2.2 Treillis de concepts

Un treillis de concepts est une structure de données qui établit des liens entre des *concepts formels*. Une étape préliminaire à la construction du treillis est l'obtention de tous les concepts qu'il contient. Dans cette section nous allons définir les éléments qui composent un treillis de concepts et voir quelques-unes de ses propriétés.

2.2.1 Correspondance entre objets et attributs

L'intérêt d'un treillis de concepts est d'organiser l'information concernant des groupements d'objets possédant des propriétés communes. On peut définir deux fonctions f et g qui permettent d'établir le lien (la correspondance) entre les objets et les attributs d'un contexte K :

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

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

Par exemple, dans la table 2.1, $g(\{a, c\}) = \{1, 2\}$ et $f(\{3, 5\}) = \{c\}$. Ce qui signifie que l'ensemble d'attributs $\{a, c\}$ possède comme ensemble d'objets communs $\{1, 2\}$. De la même manière, l'ensemble d'objets $\{3, 5\}$ possède en commun l'attribut $\{c\}$.

2.2.2 Fermeture des ensembles

Les deux fonctions f et g vont servir à calculer la fermeture de X et Y représentant respectivement un sous-ensemble d'objets et un sous-ensemble d'attributs. Pour cela on compose les fonctions f et g comme suit :

$$X'' = g(f(X))$$

$$Y'' = f(g(Y))$$

On dit qu'un ensemble est fermé s'il est égal à sa fermeture. Ainsi X est fermé si $X = X''$ et Y est fermé si $Y = Y''$.

Exemple avec la table 2.1 :

- si $X = \{3, 5\}$, alors $f(X) = \{c\}$ et donc $X'' = \{1, 2, 3, 5\}$
- si $X = \{1, 2, 4\}$, alors $f(X) = \{a\}$ et donc $X'' = \{1, 2, 4\}$
- si $Y = \{a, c\}$, alors $g(Y) = \{1, 2\}$ et donc $Y'' = \{a, c\}$
- si $Y = \{e\}$, alors $g(Y) = \{1\}$ et donc $Y'' = \{a, b, c, d, e\}$

Dans ces exemples, seuls les ensembles $\{1, 2, 4\}$ et $\{a, c\}$ sont fermés. Dans la pratique, le calcul d'une fermeture est très coûteux puisqu'il nécessite de calculer les deux correspondances $g(X)$ et $f(Y)$. Le calcul d'une seule correspondance nécessite à lui seul de parcourir tout le contexte (pouvant contenir un gros volume de données). Or, comme nous le verrons par la suite, un treillis de concepts ne comporte que des ensembles fermés. Tout l'intérêt que l'on porte à une méthode de construction du treillis de concepts réside dans son aptitude à minimiser le calcul des fermetures (voire même réussir à obtenir directement des ensembles fermés).

2.2.3 Concept

Définition 2 ***concept :*** *Un concept est un couple complet de la forme* (X, Y) *dans lequel l'extension (*extent*)* $X \subseteq O$ *et l'intention (*intent*)* $Y \subseteq A$. $X = g(Y)$ *et* $Y = f(X)$ *sont tous les deux des ensembles fermés.*

Si l'on reprend l'exemple précédent, $(\{1, 2, 4\}, \{a\})$[1] et $(\{12\}, \{ac\})$ sont tous les deux des concepts. Le deuxième concept signifie que les objets 1 et 2 ont en commun les propriétés a et c.

Certains algorithmes se limitent à l'obtention des concepts sans construire le treillis [7, 22, 39, 36] alors que d'autres incluent l'identification d'ordre entre les concepts et permettent donc la construction du treillis au complet [4, 8].

La construction du treillis en elle-même offre comme avantage de classifier de manière très ordonnée les concepts. Il est ensuite facile de sélectionner des ensembles d'objets ou d'attributs qui ont une influence plus forte que d'autres. Cette influence est souvent caractérisée par le support du concept[2].

[1] Dans ce qui suit, nous simplifions la notation en remplaçant par exemple $(\{1, 2, 4\}, \{a\})$ par (124, a).

[2] On appelle support d'un concept la cardinalité relative de l'extension de ce même concept

2.3 Treillis de concepts

Définition 3 *(Majorant, minorant) Soient $(E, \leq_E)$ un ensemble ordonné et S un sous-ensemble de E. Les éléments majorants (successeurs) et minorants (prédécesseurs) de S sont définis par :*

1. $Majorant(S) = \{x \in E \wedge \forall y \in S, y \leq_E x\}$
2. $Minorant(S) = \{x \in E \wedge \forall y \in S, x \leq_E y\}$

Définition 4 *(Relation de couverture) Soient $(E, \leq_E)$ un ensemble ordonné et x,y deux éléments de E. La relation de couverture entre deux éléments de E, notée $<_E$, correspond à la réduction transitive de $\leq_E$ définie par : $x <_E y$ si et seulement si tous les éléments $z \in E$ vérifiant $x \leq_E z \leq_E y$ satisfont $z = x$ ou $z = y$. L'élément y (resp. x) est alors appelé successeur immédiat de x (resp. prédécesseur immédiat de y).*

Définition 5 *(borne inférieure et borne supérieure) Soient $(E, \leq_E)$ un ensemble ordonné et S un sous-ensemble de E. Le plus petit majorant de l'ensemble S, s'il existe, est le plus petit élément de l'ensemble des majorants de S ($BorneInferieure(S)$ également appelé infimum de S). Dualement, le plus grand des minorants de l'ensemble S, s'il existe, correspond au plus grand élément de l'ensemble des minorants de S ($BorneSuperieure(S)$ également appelé supremum de S).*

1. $BorneInf(S) = Min \leq_E (Majorant(S))$

2. $BorneSup(S) = Max \leq_E (Minorant(S))$

On utilise l'opération *BorneInf* (*Meet*) lorsqu'on regroupe deux ou plusieurs concepts pour en former un seul par l'intersection de l'extension des concepts. L'opérateur *BorneSup* (*join*) est l'opération inverse qui consiste à relier des concepts en prenant l'intersection de leurs intentions.

Définition 6 *(Treillis de concepts) Un treillis $T = (C_{\leq})$ est un ordre partiel dans lequel chaque couple d'éléments $(X, Y) \in C_{\leq}$ admet un plus petit majorant, noté $BorneInf(x \wedge_T y)$, et un plus grand minorant, noté $BorneSup(x \wedge_T y)$. Un treillis $T = (C_{\leq})$ est dit complet si pour tout sous-ensemble $S \in C_{\leq}$, les éléments $BorneSup(S)$ et $BorneInf(S)$*

existent. Un treillis dans lequel seul $BorneInf$ ou $BorneSup$ existe est appelé semi-treillis (le treillis devient un arbre). On appelle diagramme de Hasse la représentation graphique du treillis.

Définition 7 *(Diagramme de Hasse) La représentation graphique d'un treillis $L = (E, \leq_L)$ s'exprime à l'aide d'un diagramme, appelé diagramme de Hasse, dans lequel les nœuds correspondent aux éléments de E et les arcs aux relations de couverture (successeurs et prédécesseurs immédiats) entre ces nœuds.*

Les couples $(X, Y) \in C_{\leq}$ représentent les nœuds du treillis et sont reliés entre eux par une relation d'ordre partiel. La figure 2.1 illustre le diagramme de Hasse issu du contexte de la table 2.1.

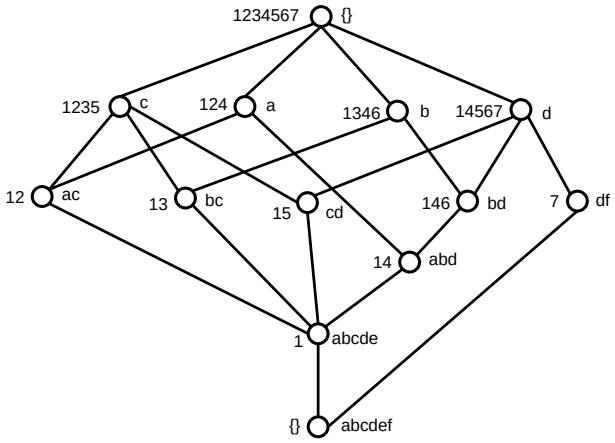

FIG. 2.1 – Treillis de concepts issu du contexte de la table 2.1.

2.4 Règles d'association

Dans la prochaine section, nous montrons que la notion de concept formel peut être utile pour la résolution efficace du problème de fouille de données. En effet, l'intention du concept coïncide avec la notion de motif fermé tel que décrit ci-après.

Soit $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ un ensemble de m items distincts. Une transaction T contient un ensemble d'items contenus dans $\mathcal{I}$, et possède un identifiant unique TID. Un sous-ensemble Y de $\mathcal{I}$ où $k = |Y|$ est un motif de taille k (*k-itemset*). Une base de transactions

AFC	**MFF**
contexte formel	base de transactions
objet	transaction
attribut	item
intention	motif fermé
extension	*tidset*

FIG. 2.2 – Correspondance des termes entre l'analyse formelle de concepts et le calcul des MFF

$\mathcal{D}$ est l'ensemble de transactions effectuées sur un sous-ensemble d'items de $\mathcal{I}$. $X \subset \mathcal{D}$ est un ensemble de transactions (appelé *tidset*) tandis que la fraction de transactions dans $\mathcal{D}$ contenant Y est appelée support de Y et notée $supp(Y)$. Ainsi, on dit qu'un motif est fréquent lorsque son support dépasse un support donné appelé *minsup*. À un *tidset* X donné correspond un motif dont la valeur est $f(X)$. Cela correspond à l'ensemble des items communs à toutes les transactions de X (voir sous-section 2.2.1). De la même manière, $g(Y)$ représente l'ensemble des transactions qui contiennent tous les items de l'ensemble Y.

Un motif Y est dit fermé lorsqu'il n'existe aucun motif Z de taille supérieure à Y mais avec le même support que Y. En d'autres termes, tout motif possède le même support que sa fermeture. De façon similaire, un *tidset* est fermé lorsqu'aucun autre *tidset* de taille supérieure possède le même support. Une autre façon d'écrire la condition de fermeture revient à dire que Y (resp. X) est fermé si et seulement si $Y = f(g(Y))$ (resp. $X = g(f(X))$).

Une règle d'association r est une implication de la forme $Y_1 \Rightarrow Y_2$, où Y_1 et Y_2 sont des sous-ensembles de $\mathcal{I}$, et $Y_1 \cap Y_2 = \emptyset$. Le support de la règle r est égal à $supp(Y_1 \cup Y_2)$ alors que la confiance représente le rapport $supp(Y_1 \cup Y_2)/supp(Y_1)$.

Des travaux relativement récents en fouille de données ont démontré que la génération de règles d'association s'effectue d'une manière plus efficace que l'algorithme Apriori [1] lorsqu'on identifie les MFF (motifs fermés fréquents) plutôt que les MF (motifs fermés).

Le calcul des règles d'association à partir de l'ensemble des MFF se fait de la façon suivante. Pour chaque motif fréquent Y, on génère les divers sous-ensembles non vides de Y. Ensuite, pour chaque sous-ensemble Y_1 de Y, une règle de la forme $Y_1 \Rightarrow Y - Y_1$ est retenue si le rapport $supp(Y)/supp(Y_1)$ est au moins égal à $minconf$ (seuil de confiance

minimal). Dans le cadre des MFF, certaines études se sont intéressées à la génération des ensembles non redondants de règles [18, 20, 34] où Y_1 est un générateur, c.-à-d., un sous-ensemble minimal de Y tel que sa fermeture est égale à Y [19, 27].

Chapitre 3

État de l'art

3.1 Génération des motifs fermés fréquents

AClose, Close [24], CHARM [39] et Closet+ [36] font partie des premiers algorithmes de calcul de MFF. Titanic [29] hérite de l'approche fondée sur la notion de générateur présente dans AClose, mais propose des simplifications intéressantes. Closet et son amélioration récente Closet+ [36] génèrent tous les deux les MFF comme les branches maximales d'un arbre appelé *FP-tree*, une structure basée sur un arbre préfixe (ou arbre *trie*) augmentée par une liste transversale de pointeurs. BAMBOO [37] est une version améliorée de Closet+ dans la mesure où il produit un résultat plus concis et propose des performances plus intéressantes. BAMBOO exploite la contrainte de support de longueur décroissante plus un certain nombre d'élagages et d'optimisations pour améliorer l'efficacité du calcul des MFF. La contrainte de support décroissant consiste à fournir un support sous forme d'une fonction décroissante qui varie selon la taille du motif de manière à réduire le nombre de motifs de petite taille. Les expérimentations conduites dans [37] montrent que BAMBOO est généralement plus performant que trois algorithmes connus de génération des MFF pour les divers types de bases de données. Toutefois, la fonction de support décroissant n'est pas établie par les auteurs d'une manière théorique mais davantage par des expérimentations et pour une base de données très spécifique.

Par la suite, nous définissons un algorithme, appelé CIGA+, qui est similaire à *BAMBOO* dans la mesure où il génère un sous-ensemble concis de MFF et propose des temps d'exécution intéressants. À la différence de BAMBOO qui base son élagage sur la fonction de support décroissant, CIGA+ utilise les cooccurrences d'items pour déterminer

rapidement les motifs de grande taille et s'inspire de propriétés des treillis de concepts en analyse formelle de concepts [9].

3.1.1 Algorithme CLOSET+

Plusieurs algorithmes utilisent les arbres *FP-Tree* [36] comme support à la génération des MFF. C'est le cas de l'algorithme CLOSET+ qui exploite un *FP-Tree* en profondeur d'abord. L'algorithme CIGA+ qui sera présenté par la suite, possède des points communs avec CLOSET+ puisqu'il exploite lui aussi un graphe en profondeur d'abord et utilise certaines propriétés de parcours de CLOSET+.

Définition 8 *Un FP-Tree [12] d'une base de transactions est un arbre préfixe des listes d'items fréquents dans les transactions.*

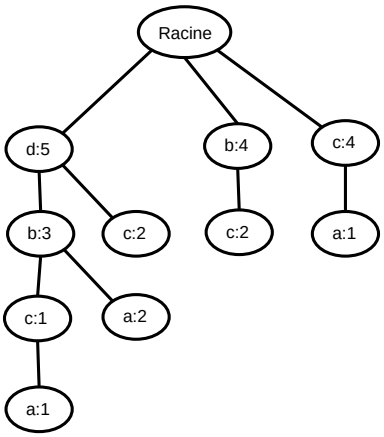

FIG. 3.1 – *FP-Tree* construit à partir de la table 2.1.

Un *FP-Tree* est construit en prenant les transactions une par une dans un contexte. Au préalable, les items sont ordonnés par ordre de support décroissant. Chaque transaction représente une branche linéaire d'items qui est préfixée dans l'arbre à partir de sa racine. Pour chaque nœud de l'arbre, on définit un support qui correspond au support du motif composé des nœuds qui le relient jusqu'à la racine.

Les MFF sont ensuite générés en parcourant cet arbre en profondeur. Le *FP-Tree* est projeté sur différents préfixes afin d'obtenir des motifs qui débutent par des items différents. Lorsque toutes les projections ont été générées, on est en mesure de produire

l'ensemble complet des MFF à partir de cette structure. La condition d'arrêt de la récursivité est basée sur la valeur du support. Lorsque celui-ci devient inférieur au support minimal, alors l'algorithme rebrousse chemin vers une autre branche du *FP-Tree*.

Les propriétés utilisées lors du parcours du graphe seront détaillées dans le chapitre sur l'algorithme CIGA+ qui les utilisent. Ces propriétés sont les suivantes :

1. Fusion d'item (*item merging*) : Si toutes les transactions T qui contiennent un motif Y_1 contiennent aussi un motif Y_2 (mais aucun sur-ensemble de Y_2), $Y_1 \cup Y_2$ est alors un motif fermé et il n'est pas nécessaire de chercher un motif qui contient Y_1 sans contenir Y_2.
2. Élagage sur le sous-motif : Si $Y_1 \subset Y_2$ et $supp(Y_1) = supp(Y_2)$ alors il n'est pas nécessaire de chercher la fermeture de Y_1 puisque ce n'est pas un motif fermé.

Dans la suite du document, nous utiliserons essentiellement l'algorithme BAMBOO [37] qui est une amélioration de CLOSET+. BAMBOO a comme particularité d'utiliser une contrainte de support décroissant en se basant sur le constat suivant : plus la taille d'un motif est grand et plus il y a de chance que son support soit faible. L'utilisation d'un support décroissant permet d'accélérer le calcul des MFF en durcissant la contrainte sur le support pour les motifs de petite taille tout en risquant d'écarter certains motifs du résultat. Malheureusement, les auteurs de BAMBOO ne fournissent pas de technique rigoureuse pour générer cette fonction qui est différente pour chaque type de base de données.

3.1.2 Algorithme CHARM

L'algorithme CHARM [39] permet lui aussi de générer les MFF issus d'un contexte formel. Il met en oeuvre une technique d'ordonnancement des concepts basée sur un arbre lexicographique (figure 3.2). D'autres algorithmes plus anciens ont déjà mis en pratique cette structure.

L'algorithme se divise en trois procédures (voir algorithme 1) :

- *CHARM*
  Procédure qui selectionne les concepts contenant un attribut et dont le support est supérieur à celui demandé par l'utilisateur. Ce travail revient à lire la base de donnée colonne par colonne.

Algorithme 1 : CHARM

Entrées : Contexte K, $supportMin$
Sorties : $Concepts$
$[P] \leftarrow \{(g(Y_i), Y_i) : Y_i \in I$ et $|Y_i| = 1$ et $|g(Y_i)| \geq supportMin\ \}$;
`CHARM-EXTEND`($[P]$,$Concepts = \emptyset$);
retourner $Concepts$

CHARM-EXTEND

Entrées : [P],$Concepts$
pour chaque $(X_i, f(X_i)) \in [P]$ **faire**
    $[P_i] \leftarrow \emptyset$ ($[P_i]$ `contiendra les futurs fils`);
    $X \leftarrow X_i$;
    pour chaque $(X_j, f(X_j)) \in [P]$ ***avec*** $i > j$ **faire**
        $X \leftarrow X \cup X_i$;
        $Y = f(X_i) \cap f(X_j)$;
        `CHARM-PROPERTY`($[P]$,$[P_i]$)
    si $[P_i] \neq \emptyset$ **alors**
        `CHARM-EXTEND`($[P_i]$, $Concepts$))
    `supprimer`($[P_i]$);
    $Concepts = Concepts \cup X$;

CHARM-PROPERTY

Entrées : $[P]$, $[P_i]$
si $support(X) \geq supportMin$ **alors**
    si $g(X_i) = g(X_j)$ **alors**
        `supprimer` X_j de $[P]$;
        remplacer les X_i par X;
    sinon si $g(X_i) \subset g(X_j)$ **alors**
        remplacer les X_i par X;
    sinon si $g(X_i) \supset g(X_j)$ **alors**
        supprimer les X_i de $[P]$;
        ajouter $X \times Y$ à $[P_i]$;
    sinon si $g(X_i) \neq g(X_j)$ **alors**
        ajouter $X \times Y$ à $[P_i]$;

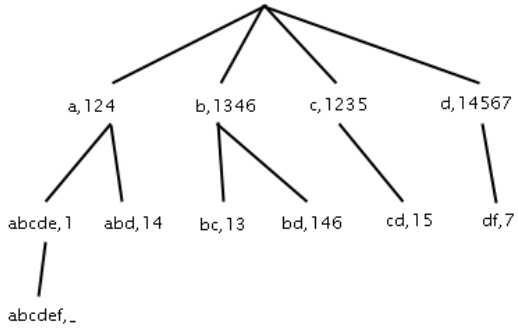

FIG. 3.2 – Indexation des concepts avec l'algorithme CHARM.

– *CHARM-EXTEND*
Charm-extend a pour rôle de faire l'union d'objets deux à deux afin de générer de nouveaux concepts candidats (variable *candidats* dans l'algorithme). La procédure est appelée recursivement tant que le candidat généré ne correspond pas à un candidat valide.

– *CHARM-PROPERTY*
Cette procédure teste la validité des candidats qui sont générés par *Charm-extend* en vérifiant plusieurs propriétés. Si le candidat est valide, alors on l'ajoute à l'ensemble des fils $[P_i]$.

3.2 Algorithmes de construction du treillis de Concepts

Cette section a pour objectif de présenter brièvement les principes de fonctionnement de quelques algorithmes de construction du treillis de concepts. La littérature fait état d'un nombre de plus en plus important d'algorithmes [4, 10, 16, 31].

3.2.1 Algorithme de Bordat

L'algorithme de Bordat [4] se charge de générer à la fois les concepts ainsi que la couverture (liens directs entre les concepts) d'une relation binaire. Ses performances sont bonnes sur des contextes peu denses de taille faible mais s'altèrent rapidement avec un grand volume de données. Il utilise les deux propriétés suivantes :

Algorithme 2 : BORDAT

Entrées : X, Y `//extension et intention du supremum`
Sorties : treillis de Concepts L

$c_i \leftarrow (X, Y)$;
$L \leftarrow L \cup c_i$;
$couverture \leftarrow$ `COUVERTURE-INFERIEURE`$((X, Y))$;
pour chaque *concept* $(D, E) \in couverture$ **faire**
  si $(D, E) \in L)$ **alors**
    enfant = `BORDAT`(D, E)
  sinon
    $enfant \leftarrow$ `retrouver`$((D, E), L)$
`créerLiens`$(c_i, Enfant)$;
retourner c_i

COUVERTURE-INFERIEURE

Entrées : (X,Y)
Sorties : $filsDirects$

$couverture \leftarrow \emptyset$;
$Z \leftarrow Y$;
tant que `trouverProchainObjet`$(X, Z, objet)$ **faire**
  $E \leftarrow \{objet\}$;
  $F \leftarrow E'$;
  $autreObjet \leftarrow objet$;
  tant que $autreObjet \neq$ `dernier`(X) **faire**
    $autreObjet \leftarrow$ `suivant`(X);
    si $\{autreObjet\}' \cap F \not\subseteq Z$ **alors**
      $E \leftarrow E \cup \{autreObjet\}$;
      $F \leftarrow F \cap \{autreObjet\}'$;
  si $F \cap Z = Y$ **alors**
    $couverture \leftarrow couverture \cup \{(E, F)\}$;
  $Z \leftarrow Z \cup F$;
si $\|filsDirects\| = 0$ **alors**
  $couverture \leftarrow \{(\emptyset, Y)\}$
retourner $filsDirects$;

1. Un concept possède toujours un sous-ensemble des objets de l'extension de son prédécesseur.
2. Un enfant c possède toujours, dans son intention, au moins un attribut de plus que l'intention de son parent c' et tous les nouveaux attributs sont uniques au sein de la couverture inférieure (liste des concepts fils) de c'.

Ces deux propriétés sont utiles pour retrouver les fils directs d'un concept. Elles sont utilisées dans la procédure COUVERTURE-INFERIEURE(X, Y) qui sert à déterminer les fils directs du concept (X, Y).

La procédure Bordat (algorithme 2) quant à elle est appelée à partir du supremum et construit le treillis de manière récursive. Pour chaque élément de la couverture inférieure (concepts fils), on vérifie si le concept a déjà été créé. Si c'est le cas, l'algorithme relie le nouveau concept à celui existant, sinon on construit ce concept (voir algorithme 2).

Variables utilisées dans l'algorithme de Bordat :

- X : sous-ensemble d'objets
- Y : sous-ensemble d'attributs
- L : treillis de concepts
- (D, E) : concept candidat

Dans cet algorithme, on est obligé de faire des recherches de concepts directement dans le treillis en construction (procédure $retrouver(X, Y)$). Une manière d'optimiser ce calcul est d'utiliser un arbre d'indexation TRIE dans lequel on classe toutes les extensions. Nous utiliserons la même structure par la suite pour classer des intentions.

3.2.2 Méthode de Nourine et Raynaud

Cet algorithme [23] offre des résultats intéressants même si d'autres algorithmes plus récents tendent à être plus performants. Il est composé de deux parties : (i) la génération des concepts organisés dans un arbre lexicographique puis (ii) identification de la relation d'ordre entre les concepts (couverture) afin de construire le treillis (algorithme 3).

La première partie de l'algorithme consiste à construire un arbre lexicographique contenant les concepts. Il s'agit d'une structure de recherche qui permet de retrouver rapidement un concept à partir de son extension.

La génération du treillis se base sur la propriété suivante :

$$|INT(c_1)| - |INT(c)| = |\{a | a \in A \, et \, \{a\}' \cap EXT(c_1) = EXT(c)\}|$$

où $INT(c)$ et $EXT(c)$ représentent respectivement l'intention et l'extension du concept c, et le concept c_1 est le prédécesseur (fils) immédiat du concept c. Pour un concept donné, l'algorithme calcule les $EXT(c_1) = a' \cap EXT(c)$ puis recherche le concept correspondant dans l'arbre lexicographique. Si la propriété précédente est respectée, alors on établit un lien entre c_1 et c. Pour s'assurer que c_1 est bien un prédécesseur immédiat, l'algorithme utilise un compteur qui s'incrémente à chaque fois qu'un attribut non présent dans c apparaît dans c_1. Si la condition $|INT(c_1)| = compteur(c_1) + |INT(c)|$ est vraie, alors c_1 est nécessairement un prédécesseur de c. La complexité de l'algorithme est en $\mathcal{O}(n.|O|.(|O| + |A|))$ où n est le nombre de concepts dans le treillis, O et A sont respectivement l'ensemble d'objets et l'ensemble d'attributs.

Algorithme 3 : Nourine et Raynaud. Construction du treillis.

Entrées : Concepts C
Sorties : Treillis de Concepts

```
1  C_a[1..m] = extensionsConceptsAttributs(C_a, C);
2  pour chaque concept ci ∈ C faire
3      compteur_ci ← 0
4  pour chaque concept c_i ∈ C faire
5      candidats ← ∅;
6      pour chaque attribut a ∈ INT(c_i) faire
7          E ← C_a[a] ∩ Extension(c_i) ;
8          candidat ← trouverConcept(C, E);
9          compteur_candidat = compteur_candidat + 1;
10         si compteur_candidat + |INT(c_i)| = |INT(candidat)| alors
11             rajouterLien(c_i, candidat)
12     mettre à zéro les compteurs;
```

3.2.3 Algorithme incrémental

Les algorithmes incrémentaux permettent de mettre à jour le treillis de concepts lors de l'ajout de nouveaux objets. Un certain nombre d'algorithmes incrémentaux existent

déjà [11, 31, 32]. Ces algorithmes partent d'un même principe qui spécifie qu'après mise à jour, un treillis comporte trois types de nœuds : (i) des nœuds inchangés (concepts existants dans le treillis initial), (ii) des nœuds modifiés (au niveau de l'extension des concepts) ou bien (iii) des nœuds rajoutés (dont l'intention n'était pas présente dans le treillis initial).

La complexité de ces algorithmes est d'ordre quadratique par rapport au nombre d'éléments dans le treillis de concepts [34].

3.3 Règles d'associations

La découverte des règles d'association se fait généralement en deux étapes : (i) la détermination de l'ensemble des motifs fermés fréquents (i.e., ceux dont le support dépasse un seuil déterminé), puis (ii) la génération des règles d'association à partir des MFF obtenus à la première étape. Des travaux de recherche relativement récents dans le domaine de la prospection de données (Bastide et al. 2003 ; Pasquier et al. 2000 ; Wang et al. 2003 ; Valtchev et al. 2002 ; Zaki 2002) ont démontré l'intérêt à produire l'ensemble des MFF lors du processus d'extraction des règles d'association. L'identification de ce nouvel ensemble d'"itemsets" assure une diminution considérable du temps d'extraction des "itemsets" fréquents et de génération des règles d'association. Le treillis de concepts est lui aussi un cadre qui s'avère intéressant pour la génération de règles grâce à l'utilisation de générateurs [19]. Dans ce cadre, la génération incrémentale de règles est possible tel que démontré dans [34].

Depuis l'apparition de l'algorithme Apriori, plusieurs algorithmes d'extraction des règles d'association ont vu le jour. Pour la plupart d'entre eux, le but est d'améliorer l'efficacité de la méthode initiale [13] tandis que le principal problème reste le vaste ensemble de MFF qui est difficilement manipulable à l'étape subséquente de production des règles. Pour réduire la taille de cet ensemble, plusieurs études ont déjà été conduites sur les motifs *fermés* [39, 24, 2, 26] et les motifs *maximaux* (MFM) [1] [3, 6]. Bien qu'il puisse y avoir un nombre exponentiellement plus grand de MFF que de MFM, les premiers ont l'avantage de n'engendrer aucune perte d'information.

[1] Les motifs fréquents maximaux sont les ensembles dont n'importe quel surensemble est non fréquent.

3.4 Approche proposée

3.4.1 Problématique

Les méthodes abordées jusqu'à présent font état de solutions exhaustives qui génèrent la totalité des concepts. Même pour des bases de taille moyenne, le nombre de ces concepts peut atteindre une taille considérable ce qui rend difficile la manipulation du résultat de l'algorithme. D'autres solutions ont été imaginées pour produire un ensemble plus concis des concepts [6] mais l'ensemble obtenu reste entièrement défini à l'avance et requiert un algorithme dédié pour chaque type d'ensemble généré. En d'autres termes, les algorithmes existants respectent tous les mêmes spécifications (à une entrée correspond une seule sortie) et la communauté scientifique s'applique à les optimiser autant que possible sans modifier leur fonction. C'est pourquoi il existe de nombreux algorithmes de génération du treillis et de calcul des MFF qui aboutissent au même résultat. Encore aujourd'hui beaucoup de chercheurs continuent à travailler sur ces algorithmes afin de réduire leur temps d'exécution et les rendre plus stables sur des données volumineuses ; conscients que la génération des concepts est une étape très coûteuse en calcul.

3.4.2 Méthodologie

L'approche que nous utilisons explore une autre manière de générer des concepts en utilisant de l'approximation. Deux solutions sont alors envisageables : (i) ayant comme objectif une génération globale des concepts, nous cherchons à nous en rapprocher le plus possible et (ii) nous laissons le choix à l'utilisateur d'un taux d'approximation en lui garantissant de garder dans le résultat les éléments les plus pertinents. Dans la première solution, l'objectif est de diminuer le temps d'exécution de l'algorithme quitte à écarter certains concepts sachant que l'utilisateur n'est pas intéressé à obtenir une solution exhaustive. La deuxième solution affiche clairement une volonté de réduire la taille du résultat tout en y gardant de l'information représentative des données afin d'optimiser les étapes postérieures à la génération des concepts (puisque l'ensemble obtenu est plus petit). Ces deux types de solutions feront l'objet de deux algorithmes appelés respectivement LatMat et CIGA+. Enfin, une troisième approche vient compléter la seconde en proposant des opérateurs pour manipuler les treillis de concepts. Contrairement à CIGA+ qui génère approximativement un ensemble concis de concepts, les opérateurs de manipulation de treillis ont l'avantage de réduire (ou d'augmenter) la taille des ré-

sultats de manière rigoureuse à la demande de l'utilisateur. Ces trois aspects couvrent une approche originale de la génération de concepts/treillis de concepts qui s'attache à fournir un résultat sur mesure. Nous nommerons cette démarche la fouille de données à la demande (*data mining on-demand*).

Chapitre 4

Algorithme LatMat

Ce chapitre décrit un algorithme visant à construire un treillis partiel. Cet algorithme s'appelle LatMat car il génère les concepts du treillis à partir d'opérations matricielles. On appelle treillis partiel, un treillis auquel il manque des concepts mais dont la relation d'ordre est respectée. L'intérêt d'avoir un treillis partiel n'est pas trivial. C'est même un inconvénient puisque dans le cas de l'algorithme LatMat, nous ne connaissons pas explicitement le nombre et surtout la qualité des concepts qui sont manquants dans le treillis. En revanche, nous cherchons à montrer que le taux de concepts manquants est faible par rapport à la taille du treillis. De plus, LatMat a comme avantage d'offrir une vitesse d'exécution supérieure aux algorithmes classiques qui construisent le treillis au complet. Cet algorithme s'adresse donc à des utilisateurs qui ne sont pas intéressés à obtenir une solution exacte et préfèrent obtenir un résultat rapidement. Ainsi on parle d'approximation.

Comme tout algorithme de génération du treillis de concepts, la méthode se décompose en deux étapes fondamentales : la génération des concepts et l'élaboration de la relation d'ordre. Pour optimiser l'algorithme, nous cherchons à lier ces deux étapes le plus possible même si la distinction reste assez nette.

4.1 Génération des concepts

Dans cette section, nous présentons la méthode utilisée pour générer les concepts. Elle se base sur le constat que la plupart des concepts sont directement issus d'une intersection entre deux objets.

4.1.1 Comparaisons deux à deux

Pour cela, l'algorithme se base sur des comparaisons d'objets[1] deux à deux dans le contexte formel. En faisant toutes les intersections possibles entre deux objets (incluant un objet avec lui-même), on génère rapidement des ensembles d'attributs fermés et donc un ensemble d'intentions. On récupère aussi les deux objets qui ont servi à former la comparaison sachant qu'ils appartiennent à l'extension qui est associée à l'intention calculée. On appelle "pseudo-concepts", ces semblants de concepts dont l'extension reste à compléter pour qu'elle soit fermée.

Prenons par exemple trois objets :

$$f(o_1) = \{a, b, d, e\}$$

$$f(o_2) = \{b, c, d\}$$

$$f(o_3) = \{c, d\}$$

Les intersections à prendre en compte sont : $f(o_1) \cap f(o_2)$, $f(o_1) \cap f(o_3)$ puis $f(o_2) \cap f(o_3)$. On obtient respectivement les intersections $\{bd\}$,$\{d\}$ et $\{cd\}$ qui forment de nouvelles intentions. Un si petit exemple n'est pas représentatif, mais on peut déjà écrire des semblants de concepts (pseudo-concepts) représentatifs des objets o_1, o_2 et o_3 et de leurs combinaisons deux à deux qui sont $\{(1, abde), (2, bcd), (3, cd), (12, bd), (13, d), (23, cd)\}$[2]. Il faut penser à rajouter les extrêmes (le supremum et l'infimum du treillis) manuellement s'ils n'ont pas été retrouvés. Tel est le cas lorsqu'un objet possède tous les attributs ou bien quand un attribut est possédé par tous les objets.

Tous les concepts ne sont pas générés en faisant uniquement ces comparaisons mais l'expérience nous montre que la quasi-totalité des concepts apparaissent quand même[3]. À partir de la même méthode, pour être sûr de générer tous les concepts, il faudrait ensuite faire les intersections 3 à 3 (complexité en $O\big((|O| \times |A|)^3\big)$), puis 4 à 4 (complexité en $O\big((|O| \times |A|)^4\big)$), etc... Plus tard, nous présenterons une autre méthode qui permet de retrouver les concepts manquants.

[1] Les termes "objet" et "transaction" sont interchangeables de même que "item" et "attribut".

[2] Les intentions sont fermées, mais les extensions doivent encore être complétées. Voici pourquoi on ne parle pas de concepts mais de pseudo-concepts.

[3] Sauf sur des contextes de densité élevée comme le montrent les expérimentations à la section 4.4.

Preuve de la fermeture des intentions générées

La démonstration est triviale. Les propriétés d'un objet constituent un fermé. Comme l'intersection de deux fermés constitue aussi un fermé, les intentions générées sont toutes fermées.

Il n'est pas nécessaire de vérifier la fermeture des pseudos-concepts générés puisqu'on sait déjà que les intentions sont fermées. Pour le moment, on cherche simplement à en accumuler un certain nombre grâce aux intersections deux à deux. Le calcul de l'extension se fera plus tard lors de la construction du treillis.

4.1.2 Baquets de stockage

Comme indiqué auparavant, les couples obtenus ne sont pas des concepts puisque la seule information que nous avons relative à l'extension sont les deux objets qui ont servi à composer l'intersection. Cette sous-section décrit une structure qui permet d'organiser les pseudo-concepts générés pour faciliter la construction du treillis. Le but est d'obtenir des pseudo-concepts directement ordonnés selon la taille de leur support pour pouvoir ensuite construire le treillis *à plat*, c.-à-d. étage par étage (voir figure 4.1) de la même manière qu'on bâtit une maison et sans avoir à consacrer une étape particulière pour compléter la fermeture des pseudo-concepts. Cette façon d'agir permet de traiter le treillis par niveaux et réduit la complexité du problème puisque les concepts de deux niveaux consécutifs sont généralement fortement liés entre eux.

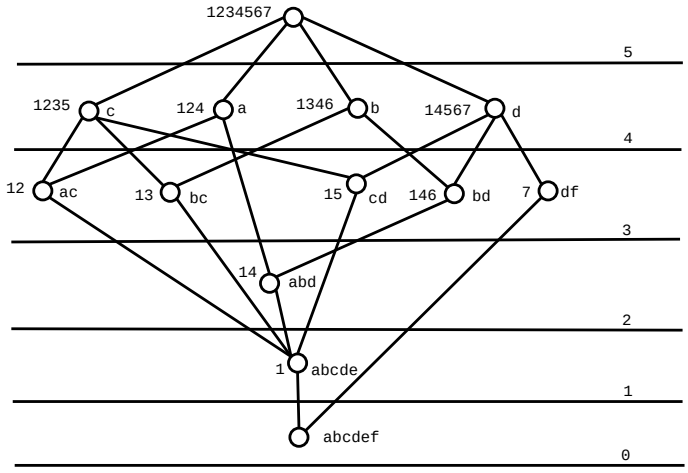

Fig. 4.1 – Séparation du treillis en niveaux.

Les pseudo-concepts sont placés dans des baquets au fur et à mesure de leur génération. On possède $|A+1|$ baquets (table 4.2) où A est l'ensemble des attributs dans la relation binaire. Chaque baquet est prévu pour stocker des pseudos-concepts d'un support donné et les pseudo-concepts sont donc placés dans le baquet qui correspond au bon support lors de leur génération. Pendant l'ajout d'un pseudo-concept, on vérifie si l'intention qu'il contient n'est pas déjà présente. Si c'est le cas, alors on fait l'union des extensions avec le pseudo-concept déjà présent dans les baquets. Dans le cas contraire, on ajoute simplement le pseudo-concept dans un baquet.

			67,d	
			57,d	
		7,df	56,d	
		6,bd	47,d	
		46,bd	45,d	
		3,bc	35,c	
		2,ac	34,b	
		16,bd	25,c	
		15,cd	24,a	37,∅
		13,bc	23,c	27,∅
1,abcde	14,abd	12,ac	17,d	26,∅

Fig. 4.2 – Baquets de stockage.

4.1.3 Arbre TRIE

Pendant les comparaisons deux à deux, on doit être capable de retrouver un pseudo-concept dans les baquets le plus rapidement possible. Cette fonction est utile lors de la création d'un nouveau pseudo-concept puisque l'on veut s'assurer qu'il n'existe pas déjà ; au risque de le dédoubler.

En générant les intentions, on va utiliser un arbre TRIE (arbre d'indexation) pour classer toutes les intentions obtenues pendant la génération des pseudo-concepts. Cette structure est déjà utilisée dans plusieurs algorithmes afin de classer les attributs ([4], [32], [39]). Chaque nœud de l'arbre TRIE modélise un attribut du contexte, sachant qu'un même attribut peut être représenté par plusieurs nœuds. La séquence qui va de la racine à un nœud quelconque de l'arbre passe par n nœuds et représente un *n-itemset*.

Pour être sûr qu'un même nœud n'apparaîsse pas deux fois dans une même branche, il faut se fixer un ordre pour classer les attributs. Ici on utilise l'ordre alphabétique, mais très souvent les items sont triés par supports décroissants (c.-à-d. du plus fréquent au moins fréquent). La figure 4.3 représente l'arbre TRIE associé aux intentions stockées dans les baquets de la figure 4.2. Pour savoir si une intention (motif fermé) a déjà été générée ou non, on marque les nœuds qui terminent une séquence déjà existante. Ces nœuds apparaissent en noir sur la figure.

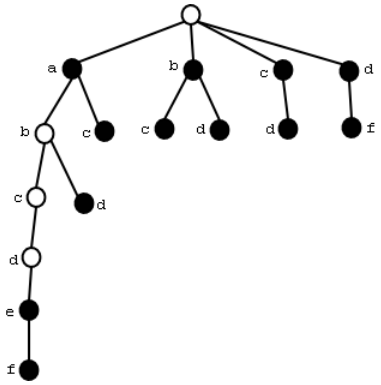

Fig. 4.3 – Arbre TRIE.

Dans cette structure, l'insertion d'un motif ne dépend pas du nombre de concepts mais du nombre d'attributs. Or on sait que le nombre total d'attributs est en général très inférieur au nombre de concepts ce qui assure une profondeur de l'arbre inférieure ou égale à $|A|$. Toutefois, le nombre de branches devient vite grand puisque l'arbre TRIE stocke jusqu'à $|O|^2$ intentions ($|O|$ est le nombre d'objets) et le nombre de nœuds dans l'arbre est supérieur au nombre d'intentions (c.-à-d. que les nœuds ne sont pas tous marqués).

Pendant le placement des pseudo-concepts, chaque nœud marqué de l'arbre TRIE pointe vers un pseudo-concept qui deviendra un concept. La figure 4.4 représente l'apparence finale de l'arbre TRIE lorsque les pseudo-concepts (initialement placés dans des baquets) sont complétés et munis d'un ordre partiel.

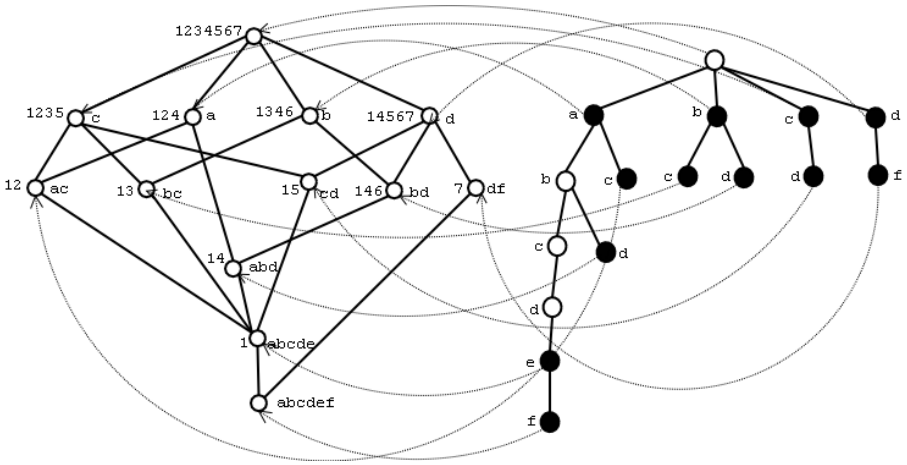

FIG. 4.4 – Liaison entre l'arbre TRIE et le treillis de concepts.

4.2 Construction de l'ordre partiel

Plusieurs travaux de recherche récents s'intéressent à la construction de l'ordre partiel. C'est le cas, par exemple, de l'algorithme de Nourine et Raynaud [23] dont l'étape de construction de l'ordre partiel est séparée de celle de génération des concepts. Dans notre cas, nous cherchons à exploiter le plus possible la structure de baquets afin de relier les pseudo-concepts entre eux. La méthode retenue dans LatMat pour construire l'ordre partiel se base sur un algorithme simple mais efficace et qui tire profit de la structure des baquets.

4.2.1 Parcours des baquets

L'algorithme de génération des liens cherche directement à relier les pseudo-concepts entre deux niveaux (baquets) consécutifs en testant l'inclusion des intentions entre elles. Comme dans beaucoup de cas, les concepts sont liés directement à leur niveau supérieur alors l'algorithme résout le problème de trouver des parents en fouillant dans un seul baquet. Mais il peut arriver qu'une intention ne soit incluse dans aucune des intentions du baquet suivant. L'algorithme passe alors au baquet de taille +1 jusqu'à finalement obtenir une intention capable de contenir l'intention courante. Cette condition finit toujours par être atteinte puisque le dernier baquet contient tous les attributs. Lorsqu'une intention est entièrement incluse dans celle d'un pseudo-concept du baquet de taille +1,

alors on ajoute un lien entre ces deux concepts. Le parcours des baquets s'effectue jusqu'à ce que tous les attributs du pseudo-concept soient présents dans ses successeurs.

4.2.2 Fermeture des pseudo-concepts

L'ajout d'un lien entre deux pseudo-concepts permet aussi de compléter l'extension du parent grâce à la notion d'héritage. Par définition, l'extension d'un concept est toujours incluse strictement dans l'extension de ses parents. Nous sommes donc capables de compléter l'extension d'un pseudo-concept quelconque en récupérant les objets qui sont contenus dans ses enfants. En appliquant cette règle pour l'ensemble des baquets, on finit par obtenir un ensemble de concepts fermés.

Cependant, dû à l'absence d'une partie des concepts, il est possible que certaines extensions ne soient toujours pas fermées une fois l'exécution de l'algorithme terminée. Cette possibilité, peu probable, intervient lorsque l'intersection de deux objets produit une intention qui ne possède pas de sur-ensemble dans la base des pseudo-concepts générés. Cette information perdue pour un nœud est rétablie au nœud supérieur grâce à l'information apportée par les nœuds adjacents du même niveau. Ce problème intervient seulement lorsqu'un concept attribut est manquant. Le concept attribut en question se retrouve alors au niveau supérieur et c'est dans ce cas précis que son extension risque de ne pas être fermée. Le problème se résout en effectuant un traitement supplémentaire sur tous les concepts attributs dont le nombre est égal à $|A|$. L'extension du pseudo-concept (X, Y) s'obtient en calculant $X = g(Y)$.

Prenons par exemple, le cas du pseudo-concept $(46, bd)$. Lorsqu'il est relié à son prédécesseur $(14, abd)$, on calcule la fermeture de l'union des extensions $\{4, 6\} \cup \{1, 4\}$ pour finalement obtenir le concept $(146, bd)$.

4.2.3 Algorithme

L'algorithme a été implanté sur la plate-forme Galicia en Java (cf *http ://galicia.sourceforge.net*). Il est composé de deux parties :

1. Génération des pseudo-concepts (algorithme 4)
2. Construction de l'ordre partiel (algorithme 5).

L'algorithme 4 se charge de générer les pseudo-concepts et de les stocker dans des baquets. L'intersection entre les objets est calculée à la ligne 3. Un arbre TRIE est

Algorithme 4 : Remplissage des baquets

Entrées : Contexte (O, A, R)
Sorties : Treillis de concepts partiel

```
1  pour i allant de 1 à |O| faire
2      pour j allant de i à |O| faire
3          intention ← f(o_i) ∩ f(o_j);
4          si intention existe déjà alors
5              ajouter ({o_i, o_j} à l'extention du pseudo-concept déjà existant;
6          sinon
7              ajouter le pseudo-concept (({o_i, o_j}, intention));
```

utilisé à la ligne 4 pour stocker les intentions générées. Lorsqu'une intention existe déjà, on complète le pseudo-concept correspondant, sinon on crée un nouveau pseudo-concept.

L'algorithme 5 génère l'ordre partiel. Les deux premières boucles parcourent les pseudo-concepts. La boucle *Pour* à la ligne 6 parcourt le baquet j pour relier les concepts du baquet i jusqu'à ce que la condition de la ligne 11 soit atteinte. La boucle *Tant que* de la ligne 10 est utile lorsqu'un concept ne possède pas tous ses parents dans le niveau qui est juste au-dessus de lui. Si c'est le cas, on incrémente la variable j pour tester le niveau suivant.

Algorithme 5 : Ordre Partiel

Entrées : Baquets contenant les pseudo-concepts
Sorties : Ordre partiel

```
1  pour i allant de 0 à |A| faire
2      pour chaque pseudo-concept pc du baquet i faire
3          intention ← INT(PC);
4          j ← i + 1;
5          tant que intention ≠ ∅ faire
6              pour chaque pseudo-concept pc+ du baquet j faire
7                  si INT(pc+) ⊆ intention alors
8                      lier(pc, pc+);
9                      EXT(pc+) ← EXT(pc+) ∪ EXT(pc);
10                     intention ← intention − INT(pc+);
11                     si intention = ∅ alors Arrêter;
12             j ← j + 1;
```

Définition des variables dans la procédure Remplissage des baquets :
- O : ensemble des objets
- A : ensemble des attributs
- o_i : objet i
- $f(o_i)$: ensemble des propriétés décrivant l'objet o_i
- ajoutBaquets(c) : rajouter l'élément c dans le baquet correspondant à la cardinalité d'intention de c
- viderBaquets() : consiste à parcourir les baquets l'un après l'autre dans un ordre décroissant de la cardinalité de l'intention des concepts. Au sein de chaque baquet, on récupère un élément à la fois jusqu'à ce que le baquet soit vide.

L'algorithme 5 débute son exécution à partir de l'infimum $(\emptyset, abcdef)$. On parcourt le baquet suivant pour trouver ses prédécesseurs. Comme l'intention du concept $(1, abcde)$ est incluse dans $\{abcdef\}$, alors $(1, abcde)$ devient un successeur de l'infimum. Il reste l'attribut f qui n'est toujours pas présent dans un des successeurs. On parcourt donc les baquets jusqu'à le rencontrer une première fois. Cet attribut apparaît seulement au troisième niveau dans le pseudo-concept $(7, df)$ qui devient à son tour un successeur de l'infimum. Les successeurs de chaque pseudo-concept sont calculés ainsi jusqu'à l'obtention de l'ordre partiel au complet.

4.2.4 Autre méthode de construction

Cette sous-section présente une autre solution qui permet de construire le treillis à l'aide d'un index pour repérer rapidement les concepts dans le treillis.

La méthode qui a été proposée a comme défaut de chercher aveuglément dans deux niveaux consécutifs du treillis pour relier les concepts. Lorsqu'un baquet est de grande taille, cette méthode atteint ses limites en parcourant tout le baquet à la recherche d'une intention qui soit incluse dans l'intention courante. L'idée ici est d'utiliser un index qui contient l'information pour chaque attribut sur sa plus haute position dans le treillis (figure 4.5). Ainsi les enfants du concept à placer sont très vite retrouvés puisque les index qui correspondent à ses attributs pointent directement vers eux.

La figure 4.5 illustre cette structure de données. Pour chaque attribut a_j, on mémorise pour chaque branche l'emplacement du plus haut concept possédant a_j. Un attribut donné peut donc posséder plusieurs pointeurs lorsqu'il est présent dans plusieurs

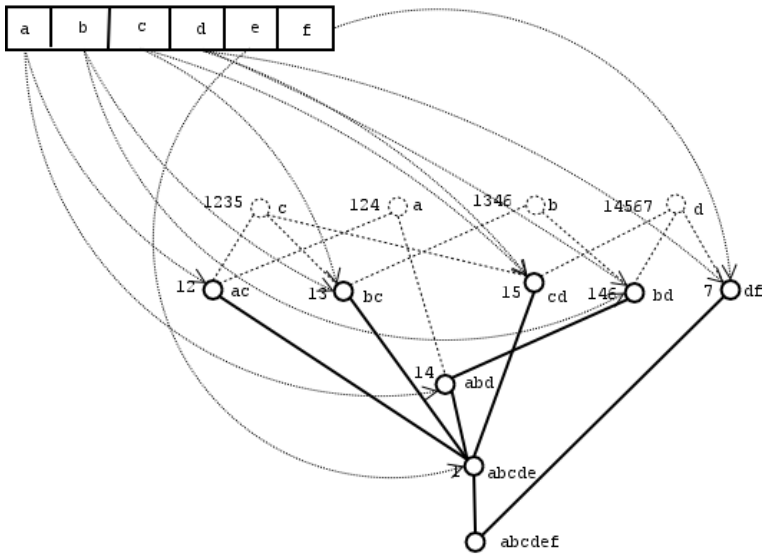

FIG. 4.5 – Élagage de la construction du treillis

branches. La principale difficulté de cette méthode est de mettre à jour l'index au fur et à mesure que des concepts sont rajoutés. Dans la pratique, sur des bases de tailles moyennes, un attribut donné apparaît dans un grand nombre de branches. L'algorithme effectue de nombreux parcours successifs de l'index pour le mettre à jour, ce qui rend finalement cette solution peu intéressante pour construire l'ordre partiel.

4.3 Analyse de complexité

La méthode proposée a une complexité fortement dépendante du nombre d'objets et d'attributs.

Comme mentionné auparavant, le remplissage des baquets nécessite le calcul de $|O|^2$ intersections. Pour chaque intersection, on effecue une recherche dans l'arbre TRIE en $\log(|A|)$ d'où finalement, une complexité de $(|O|.\log|A|)^2$ pour l'étape de remplissage au complet des baquets. Le coût de la construction de l'ordre partiel est quant à lui plus difficile à évaluer. Il dépend du nombre de concepts stocké dans les baquets sachant que cette répartition n'est pas uniforme. On dispose de $|A+1|$ baquets qui contiennent au plus $|O|^2$ pseudo-concepts. Un baquet contient alors en moyenne $\frac{|O|^2}{|A|+1}$ pseudo-concepts. Chaque élément d'un baquet est comparé avec les éléments du baquets suivant jusqu'à

obtenir tous ses parents et parfois, il doit aussi parcourir le baquet suivant. Considérons le cas où un élément est comparé avec tous les éléments du baquet suivant (qui semble une hypothèse raisonable). Chaque concept est alors comparé par rapport à $\frac{|O|^2}{|A|+1}$ éléments ; ce qui correspond à $|O|^2.\frac{|O|^2}{|A|+1}$ comparaisons au total. Finalement la complexité totale vaut $(|O|.\log|A|)^2 + |O|^2.\frac{|O|^2}{|A|+1}$

Pour se donner un ordre d'idée, la méthode de Nourine et Raynaud [23] et l'algorithme incrémental [34] ont des complexités de l'ordre de $\mathcal{O}(n^2)$ où n est le nombre d'éléments dans le treillis. Dans le pire des cas, on a $n = 2^{|O|}$.

4.4 Expérimentations

Cette section comporte deux volets. Nous étudions d'abord la viabilité de LatMat par rapport au taux d'information qui est perdue. Ensuite, nous faisons une étude comparative de LatMat avec d'autres algorithmes de construction du treillis.

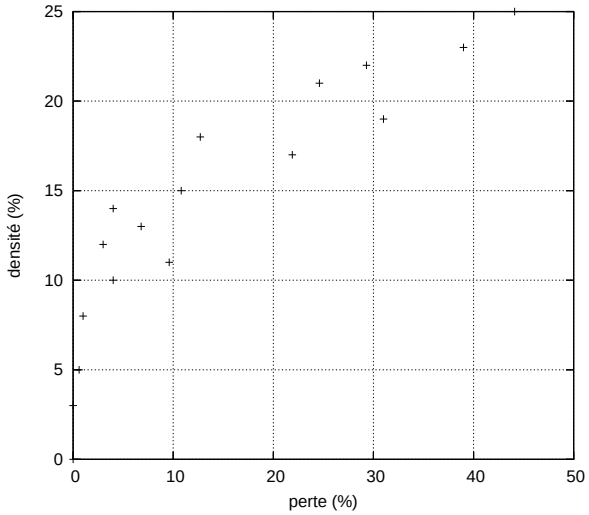

Fig. 4.6 – Perte obtenue en fonction de la densité du contexte.

Il est assez difficile d'évaluer le taux de concepts manquants dans LatMat en raison de la p-complétude du problème. Alors que la solution polynomiale utilisée par LatMat génère un résultat dont l'évolution est régulière, l'ensemble complet des concepts dépend

fortement des données. Des résultats empiriques montrent que le taux de concepts manquants est fortement lié à la densité[4] du contexte formel. La figure 4.6 montre des résultats d'expérimentations effectués sur des contextes générés aléatoirement ayant un nombre d'objets et d'attributs compris entre 50 et 100. Le nuage de points obtenu est eparpillé mais montre quand même une tendance. En règle générale, on s'aperçoit que le taux de concepts perdus est quasiment nul lorsque le contexte possède une densité inférieure à 10%. Cette hypothèse reste raisonable compte tenu des cas rencontrés dans la pratique.

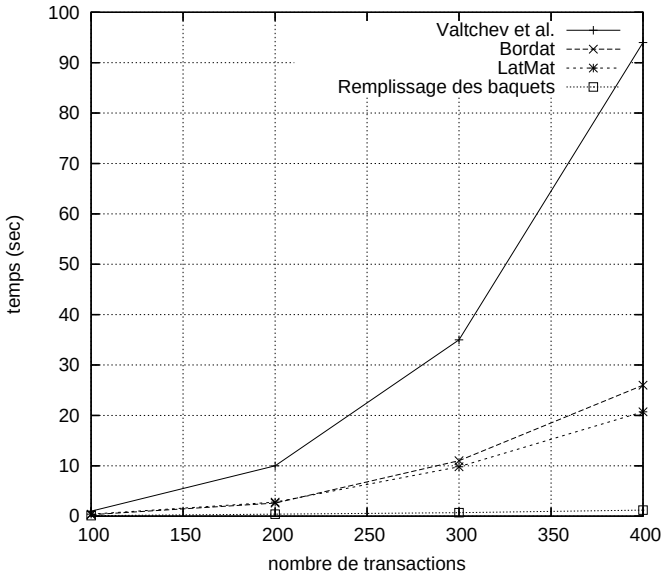

Fig. 4.7 – Comparaison de LatMat avec les algorithmes Bordat et Valtchev et al. sur un contexte à 100 attributs et de densité 10%.

La figure 4.7 compare les temps d'exécution de deux algorithmes avec LatMat. Les expérimentations ont été conduites sur la plate-forme Galicia[5] sous GNU Linux en utilisant un Pentium IV à 3 Ghz et avec 1024 Mo de mémoire. Les contextes utilisés ont été générés aléatoirement et possèdent une densité de 10% afin de minimiser le taux de concepts manquants dans LatMat. L'algorithme Bordat (voir la sous-section 3.2.1) est très performant lorsqu'il s'exécute sur des contextes de faible densité (c'est le cas

[4]La densité d'un contexte représente le rapport entre le nombre de relations $\langle attributs, valeurs \rangle$ vraies par le nombre de relations au total.

[5]http ://galicia.sourceforge.net

ici). L'algorithme incrémental Valtchev et al. [31] (voir la sous-section 3.2.3) est quant à lui moins performant mais offre un autre élément de comparaison. La figure 4.7 indique aussi le temps de remplissage des baquets avant de construire l'ordre partiel. Les courbes montrent des performances quasiment similaires pour Bordat et pour LatMat. Toutefois, on remarque que la génération des pseudo-concepts est presque instantanée quelque soit le nombre de transactions. L'étape critique de LatMat est donc la construction de l'ordre partiel qui n'est pas assez performant. Cela nous amène à conclure que nous pouvons utiliser LatMat pour construire un ensemble approximatif de concepts et donc de motifs fermés fréquents en ignorant le calcul de l'ordre partiel dans l'algorithme 5.

4.5 Conclusion

Ce chapitre a présenté une méthode simple de génération du treillis de concepts basé sur l'intersection des transactions deux à deux dans le contexte formel. L'idée qui semble naïve s'avère fournir des résultats intéressants surtout pour la génération des pseudo-concepts. La construction de l'ordre partiel, quant à elle, doit encore être améliorée puisqu'elle occupe une partie trop importante du temps d'exécution de l'algorithme. Même si la génération des pseudo-concepts est rapide, l'algorithme LatMat rejoint les performances d'autres algorithmes à cause de cette étape critique.

Chapitre 5

Concepts manquants

Comme l'algorithme LatMat applique une approximation aveugle, nous cherchons maintenant à identifier et à qualifier la perte d'information. Ce chapitre analyse quels sont les concepts manquants et propose une solution pour la construction du treillis au complet. Cette solution est présentée dans le but d'analyser la qualité du résultat de l'algorithme mais ne fait pas l'objet d'expérimentations. Le chapitre 6 reprend la même intuition dans le but de fournir une solution performante et fait l'objet d'expérimentations intensives.

5.1 Dénombrement des concepts

Le nombre de concepts issus d'un contexte peut croître exponentiellement avec le nombre d'attributs et d'objets alors que l'algorithme LatMat génère un nombre de concepts borné par $|O|^2$ où $|O|$ est le nombre d'objets. Cependant, il est rare que le nombre de concepts soit réellement exponentiel par rapport aux données. Sous l'hypothèse que le nombre d'attributs par objet est borné par une constante, le nombre de concepts a le même ordre de croissance que le nombre d'objets [11]. De même des expérimentations montrent que le nombre de concepts est de l'ordre du produit du nombre d'objets par la moyenne du nombre d'attributs par objet [11]. Ce qui rejoint l'hypothèse utilisée dans LatMat qui consiste à générer les concepts en temps polynomial. Mais à notre connaissance, il n'existe aucun algorithme ou heuristique d'estimation du nombre de concepts d'un contexte. Cette classe de complexité définit les problèmes dont le nombre de solutions ne peut être estimé sans énumérer l'ensemble de ses solutions [30].

Le seul cas où on obtient réellement un nombre de concepts exponentiel par rapport aux objets ($2^{|O|}$ concepts) intervient dans un seul type de contexte. Il s'agit d'une relation binaire carrée dont chaque objet possède toutes les propriétés sauf sur la diagonale (table 5.1). N'importe quel type d'intersection entre objets génère une intention différente.

Objets \ Attributs	att_1	att_2	...	att_{a-2}	att_{a-1}	att_a
obj_1		x	x	x	x	x
obj_2	x		x	x	x	x
⋮	x	x		x	x	x
obj_{o-2}	x	x	x		x	x
obj_{o-1}	x	x	x	x		x
obj_o	x	x	x	x	x	

TAB. 5.1 – Pire cas pour un contexte.

5.2 Identification

Les concepts manquants ont la particularité d'être issus d'une intersection entre au moins trois transactions. Cependant, la plupart des concepts issus d'une telle intersection sont aussi générés lors d'une intersection entre deux transactions. Il n'est donc pas aisé de découvrir les concepts manquants en gardant la même approche puisqu'ils n'ont apparament pas de propriétés caractéristiques.

Objets \ Attributs	a	b	c	d	e
1	x		x	x	
2	x			x	x
3	x		x		x

TAB. 5.2 – Un autre exemple de contexte.

`Exemple de disfonctionnement`

La table 5.2 fournit un exemple d'intersection impliquant trois objets. L'ensemble des intentions obtenues est $\{(acd), (ade), (ace), (ad), (ac), (ae)\}$. Si on applique l'algorithme

LatMat sur ce contexte, l'intention a n'est pas générée et donc seulement 86% des intentions totales sont présentes dans le résultat.

5.3 Obtention des concepts manquants

Cette section traite d'une méthode qui permet de compléter la liste des concepts. On donne donc la possibilité de générer un treillis complet moyennant un nombre de calculs plus important. Ce chapitre a pour objectif de fournir une intuition sur la découverte au complet des concepts et n'a pas pour but de fournir une solution performante. Aucune expérimentation n'est proposée puisque l'algorithme CIGA+ présenté dans le prochain chapitre s'inspire fortement de la méthode utilisée ici pour compléter la liste des concepts et fournir une solution intéressante.

5.3.1 Propriétés de l'arbre TRIE

L'arbre TRIE tel qu'il est utilisé dans LatMat permet de stocker toutes les intentions issues des comparaisons deux à deux. Même si toutes les intentions ne sont pas présentes dans cet arbre, on s'aperçoit que toutes y apparaissent en effectuant des combinaisons de branches. En effet, l'intersection des branches de l'arbre TRIE produit des intersections entre plusieurs objets. On est donc capable de retrouver tous les concepts de cette façon. Mais vu sous cet angle, le problème est loin d'être résolu puisqu'on est capable de faire les même intersections à partir de la relation binaire sans passer par l'arbre TRIE. De plus, faire toutes les intersections de branches de l'arbre TRIE est un problème de complexité exponentielle. L'explosion combinatoire engendrée par ce calcul rend l'approche non raisonnable.

On remarque d'autre part que les intersections issues de comparaisons d'objets deux à deux produisent nécessairement les intentions de plus grandes tailles selon la propriété :

$$(\{E_1\} \cap \{E_2\} \cap \{E_3\}) \subset (\{E_1\} \cap \{E_2\})$$

L'arbre TRIE peut donc servir de support pour la récupération des intersections manquantes et par de là même, les intentions manquantes. Il est possible alors d'utiliser une structure de données qui met en évidence cette propriété.

L'intuition derrière l'arbre TRIE consiste à regrouper toutes ses branches pour former un graphe orienté sans cycles. Le regroupement se fait en réunissant tous les nœuds qui portent le même attribut de manière à réunir leurs parents et leurs fils.

5.3.2 Utilisation d'un graphe de dépendances

Dans cette section, nous utilisons un graphe pour aider à la génération de nouveaux concepts. Un autre graphe de dépendance est aussi utilisé dans le chapitre suivant mais ses caractéristiques et son mode de construction sont totalement différents.

Afin d'illustrer la méthode d'identification des intentions manquantes, nous allons prendre un nouvel exemple (table 5.3) dans lequel des concepts sont manquants. L'ensemble des intentions deux à deux avec cet exemple est $\{abde, bcd, bce, bd, be, bc, ad, d\}$.

Objets \ Attributs	a	b	c	d	e
1	x	x		x	x
2		x	x	x	
3		x	x		x
4	x			x	

TAB. 5.3 – Nouvel exemple de contexte.

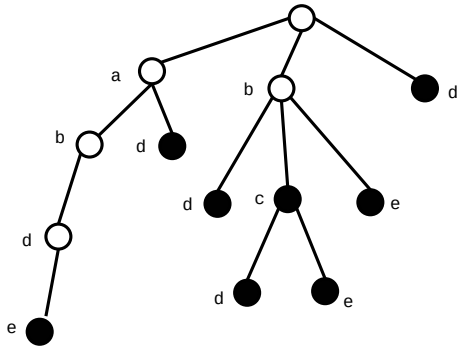

FIG. 5.1 – Arbre TRIE associé au graphe de la figure 5.2

Au lieu de construire l'arbre TRIE (figure 5.1), nous allons construire un graphe de dépendances des intentions (figure 5.2)[1]. Il s'agit d'un graphe orienté sans cycles dont chaque nœud représente un attribut en dehors du nœud initial qui symbolise l'attribut vide. La liste totale des intentions du treillis est un sous-ensemble des chemins existants à partir de sa racine. Le graphe de dépendances est donc une structure suffisament complète pour contenir l'ensemble des intentions. Ce graphe contient un seul point d'entrée qui est le nœud initial et ne contient qu'un seul nœud terminal[2] (attribut e de la figure 5.2).

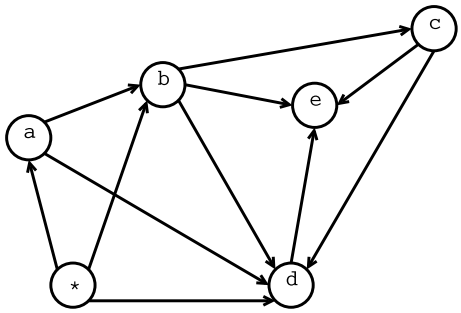

FIG. 5.2 – Graphe de dépendances

Les intentions manquantes dans l'arbre TRIE s'obtiennent par des intersections entre deux ou plusieurs branches. Le graphe de dépendances contourne ce problème en regroupant toutes les branches grâce à la fusion des attributs identiques. Chaque nœud du graphe de dépendances garde le même ensemble de parents et d'enfants et le nombre de noeuds correspond au nombre d'attributs du contexte (plus le noeud initial).

5.3.3 Découverte des concepts manquants

Le tableau 5.4 indique la liste de toutes les intentions qui sont découvertes dans le graphe de dépendances. Les colonnes indiquent leur niveau d'apparition. Pour chaque motif, trois choix sont possibles :

1. L'intention a déjà été découverte avec les intersections deux à deux.

[1]La construction de l'arbre TRIE et du graphe de dépendances ont la même complexité algorithmique.

[2]Un nœud terminal est un nœud d'où ne part aucune flèche.

2. Une nouvelle intention fermée a été découverte grâce au graphe de dépendances.
3. L'intention apparaît dans le graphe de dépendances mais n'est pas fermée.

Pseudo-intentions \ Caractéristiques	anciennes intentions	nouvelles intentions	intentions non fermées
a			x
ab			x
abde	x		
ad	x		
b		x	
bd	x		
bde			x
be			x
bc	x		
bcd	x		
bcde			x
bce	x		
be	x		
d	x		
de			x

TAB. 5.4 – Caractéristique des intentions obtenues.

Comme on peut le voir dans l'exemple de la table 5.4, le graphe de dépendances est capable de reconnaître 15 intentions (les chiffres servent uniquement à illustrer l'exemple). Sur ces 15 éléments, 9 sont fermées ce qui signifie que 60% des motifs contenus dans le graphe nous intéresse.

Pour chaque intention contenue dans le graphe, il est nécessaire de vérifier sa fermeture au cas par cas (la section suivante fournit des suggestions de réduction de la procédure). Dans notre exemple, les intentions $\{a\}$, $\{ab\}$ et $\{b\}$ viennent d'être découvertes. Il n'est pas nécessaire de calculer la fermeture pour les autres. Le calcul d'une fermeture en particulier se fait de façon naïve. On effectue deux calculs de correspondance avec les fonctions f et g définies plus haut.

Exemple de calcul de fermeture pour $\{ab\}$: On vérifie que $\{ab\} = f \circ g(\{ab\})$. On a $g(\{ab\}) = \{1\}$ et f($\{1\}$)=$\{$abde$\}$ donc $\{ab\} \neq f \circ g(\{ab\})$ ce qui implique que $\{ab\}$ n'est pas un ensemble fermé.

5.3.4 Élagage du calcul de la fermeture

La génération de candidats à partir du graphe de dépendances s'effectue en parcourant le graphe de toutes les façons possibles. Le calcul consiste donc à suivre des chemins (complets et non complets) dans lesquels pour chaque progression vers un nouvel état on génère un nouveau candidat.

Objets \ Attributs	a	b	c
1	x	x	
2		x	x
3	x	x	x

TAB. 5.5 – Exemple de contexte.

Lors du calcul des fermetures, il est possible de regrouper les calculs sur une branche complête du graphe en réutilisant les calculs intermédiaires pour chaque nœud. Prenons par exemple le contexte de la table 5.5. Ici on s'intéresse juste à calculer des fermetures (peu importe si toutes les intentions sont générées avec des intersections deux à deux). On souhaite calculer les fermetures de $\{a\}$, $\{ab\}$ et $\{abc\}$. On se déplace sur le chemin (abc).

1. On commence par calculer l'ensemble des objets qui possèdent la propriété $\{a\}$: $\{a\}' = \{13\}$.
2. Maintenant pour calculer, $g(\{ab\})$ connaissant $g(\{a\})$, on s'intéresse seulement aux objets 1 et 3. Finalement $\{ab\}' = \{13\}$.
3. Même chose pour $g(\{abc\})$: $\{abc\}' = \{3\}$
4. À présent, les trois correspondances d'attributs ont été calculées. On remonte le calcul en cherchant les correspondances des objets obtenus de la même manière que pour les attributs. On obtient finalement $\{3\}' = \{abc\}$.
5. puis $\{13\}' = \{ab\}$.

On conclut que les intentions $\{ab\}$ et $\{abc\}$ sont toutes les deux fermées. On sait aussi que $\{13, ab\}$ et $\{3, abc\}$ sont des concepts.

Chapitre 6

Algorithme CIGA+

6.1 Définition du problème

Dans ce qui suit, nous décrivons CIGA+ [15, 21], un algorithme qui utilise trois paramètres de seuil pour produire un ensemble complet ou approximatif des motifs fermés fréquents (MFF) en explorant un graphe de dépendances dans lequel les nœuds sont des items et les arêtes représentent des liens (fréquences de cooccurrence) entre deux items. Les seuils utilisés sont (i) le support minimal (*minsupp*), (ii) la confiance minimale entre deux items (*mincooc*), et (iii) la tolérance minimale (*mintol*). Le premier paramètre est déjà connu par la communauté du *data mining*, le second paramètre *mincooc* permet d'écarter des liens faibles entre deux items individuels en se basant sur leur fréquence de cooccurrence, et le dernier paramètre permet d'éliminer d'éventuels liens qui ne sont pas nécessaires entre deux items. Par conséquent, le graphe de dépendances est simplifié pour en faciliter son parcours et pour la production d'un ensemble plus réduit de MFF. Le paramétrage du seuil *mintol* dépend du degré d'approximation souhaité par l'utilisateur. Un seuil de 100% permet d'obtenir l'ensemble complet des MFF tandis qu'une valeur non nulle génère une approximation plus ou moins importante de cet ensemble.

Ce chapitre traite la génération des MFF[1] et le calcul des règles d'association faisant appel à une terminologie différente des treillis de concepts et qui est décrite dans la section 2.4.

[1]Rappel : un MFF est l'intention d'un concept dont le support est supérieur à un support minimal donné.

CIGA+ comporte deux étapes : la construction du graphe de dépendances (voir section 6.2) puis la génération des MFF (voir section 6.3). Les seuils de cooccurrence et de tolérance sont utilisés lors de la première étape tandis que le support est utilisé dans la seconde.

À titre d'exemple, nous allons considérer la petite base de transactions de la table 6.1. Elle comporte huit transactions et neuf items $\{a, b, c, d, e, f, g, h, i\}$ qui décrivent des paniers de consommateurs (achat de produits).

Tid	(a) Beurre	(b) Eau minérale	(c) Baguette	(d) Biscuits	(e) Foie gras	(f) Roquefort	(g) Saumon fumé	(h) Champagne	(i) Caviar
1	X	X					X		
2	X	X					X	X	
3	X	X	X				X	X	
4	X		X				X	X	X
5	X	X		X		X			
6	X	X	X	X		X			
7	X		X	X	X				
8	X		X	X		X			

TAB. 6.1 – Base de transactions.

Les items sont triés par ordre décroissant de leur support car les items les plus fréquents apparaissent plus souvent dans le résultat et donc sont traités en premier. Le tri des items est aussi une condition nécessaire pour appliquer l'élagage décrit dans la section 6.2.

6.2 Prétraitement

Cette section décrit l'étape de prétraitement qui consiste à construire un graphe de dépendances à partir d'une matrice de cooccurrences. Le graphe servira ensuite à la génération des MFF. Le graphe peut être partiel ou complet (i.e tous les MFF sont générés) selon les valeurs qui sont attribuées à *mincooc* et *mintol*. La signification de ces paramètres est expliquée dans cette section.

6.2.1 Matrice de cooccurrence

La construction du graphe de dépendances débute par la construction d'une matrice de cooccurrences (voir table 6.2) pour les items fréquents. Il s'agit d'une matrice triangulaire dont les lignes et les colonnes sont ordonnées par ordre décroissant du support des items. La valeur d'une cellule $cooc[i, j]$ (avec $j \geq i$) représente le nombre de fois que l'item a_j apparaît avec a_i et $cooc[i, i]$ représente le support (absolu) de l'item a_i. Par exemple, $cooc[a, d] = 4$.

	a	b	c	d	g	f	h	e	i
a	8	5	5	4	4	3	3	1	1
b		5	2	2	3	2	2	0	0
c			5	3	2	2	2	1	1
d				4	0	3	0	1	0
g					4	0	3	0	1
f						3	0	0	0
h							3	0	1
e								1	0
i									1

TAB. 6.2 – Matrice de cooccurrence.

Un ensemble de mesures sur des règles élémentaires (c.à.d. règles qui mettent en jeu deux items individuels) peut être directement extrait de la matrice de cooccurrence :

- Le support relatif d'une règle $a_i \Rightarrow a_j$ est $supp(a_i \Rightarrow a_j) = \frac{cooc[i,j]}{|\mathcal{D}|}$. Le paramètre *minsupp* est sa valeur minimale tolérée.
- La confiance d'une règle $a_i \Rightarrow a_j$ est $\frac{cooc[i,j]}{cooc[i,i]}$. Sa plus petite valeur tolérée est appelée *mincooc*.
- La tolérance d'un lien (a_j, a_i), où $supp(a_j) \leq supp(a_i)$ est équivalente à la confiance d'une règle $a_j \Rightarrow a_i$. Sa plus petite valeur acceptée est *mintol*.

Par exemple, $conf(f \Rightarrow d) = 100\%$, ce qui signifie que l'item f apparaît toujours en présence de l'item d.

6.2.2 Graphe de dépendances

On définit un graphe de dépendances $G =< \mathcal{N}, \mathcal{T} >$ comme étant un graphe orienté et acyclique dont les nœuds dans $\mathcal{N}$ correspondent aux items fréquents. Il représente

les dépendances qui ont lieu entre deux items selon leur fréquence de cooccurrence. Ainsi, une arête dans $\mathcal{T}$ allant du nœud a_i au nœud a_j, et notée (a_i, a_j), indique que $conf(a_i \Rightarrow a_j) \geq mincooc$ et $supp(a_i) \geq supp(a_j)$.

6.2.3 Élagage

Comme pour la plupart des algorithmes de type Apriori, un des élagages utilisés dans CIGA+ consiste à ignorer les items non fréquents (c.-à-.d., les items qui ont un support inférieur à *minsupp*). Les autres élagages permettent d'écarter des MFF que l'on juge non pertinents. Les paramètres sont exploités de la manière suivante : (i) utilisation de *minsupp* pour écarter les items non fréquents, (ii) utilisation de *mincooc* pour écarter les liens faibles entre deux items donnés dans le graphe de dépendances, et (iii) utilisation de *mintol* pour éliminer les arêtes inutiles entre deux items donnés.

Propriété 1 *Lorsque $g(a_j) \subseteq g(a_i)$ (c.-à-d. $conf(a_j \Rightarrow a_i) = 100\%$), alors l'arête (a_h, a_j) du graphe de dépendances est toujours inutile quelque soit a_h tel que $supp(a_h) \geq supp(a_i)$.*

La propriété 1 (illustrée par la figure 6.1) signifie que lorsqu'un item a_j apparaît systématiquement dans une transaction avec a_i, alors l'arête (a_h, a_j) dans le graphe de dépendances est redondante pour tout nœud a_h qui précède a_i. Ceci est toujours vrai dans la mesure où tout MFF contenant a_j va nécessairement inclure a_i, et par conséquent un chemin qui court-circuite a_i en allant de a_h à a_j conduit toujours vers un motif non fermé. Il est important de noter que a_i n'est pas nécessairement le parent direct de a_j, et que la présence de (a_h, a_j) dans le graphe de dépendances, par définition, implique $supp(a_h) \geq supp(a_j)$.

Dans notre exemple, l'item *Roquefort* apparaît toujours avec *biscuits* puisque *conf(Roquefort ⇒ biscuits) = 100%*. Ainsi, tout *MFF* qui contient *Roquefort* et tout autre item ayant un support supérieur ou égal au support de *biscuits* (e.g., *eau minérale*) va nécessairement contenir l'item *biscuits*. En d'autres termes, la fermeture de {*eau minérale*, *Roquefort*} contiendra toujours l'item *biscuits*. L'arête *(eau minérale, Roquefort)* est alors inutile.

La propriété 1 est adaptable pour approximer (simplifier) le graphe de dépendances. La propriété suivante en est une généralisation.

Propriété 2 *Si* $conf(a_j \Rightarrow a_i) \geq$ mintol, *alors l'arête* (a_h, a_j) *est rarement utile pour tout nœud* a_h *tel que* $supp(a_h) \geq supp(a_i)$.

La propriété 2 permet d'écarter l'arête (a_h, a_j) chaque fois que l'item a_j apparaît "presque toujours" avec a_i (en se basant sur la valeur de *mintol*) puisque tout motif qui contient a_j aura de fortes chances de contenir a_i.

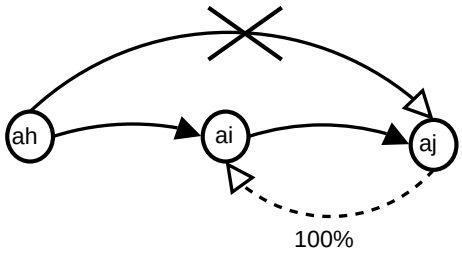

FIG. 6.1 – Illustration de l'élagage appliqué au graphe avec $mintol = 100\%$.

Un cas particulier de l'élagage basé sur *mintol* intervient lorsque a_h représente l'état initial. Dans un tel cas, cela implique que le nœud a_j ne sera pas connecté au nœud initial dès que la condition $conf(a_j \Rightarrow a_i) \geq mintol$ aura lieu.

Le graphe de dépendances de la figure 6.2 a été généré en utilisant $mintol = 100\%$. On peut remarquer qu'il n'y a pas systématiquement d'arêtes allant du nœud initial vers chaque nœud du graphe. Par exemple, on observe que *Roquefort* apparaît tout le temps en présence de l'item *biscuits* et par conséquent, une arête allant de {} vers *Roquefort* est non justifiée puisqu'il existe un chemin aboutissant à *Roquefort* en passant par *biscuits*, et que *Roquefort* ne peut pas constituer à lui seul un MFF. De même, les arêtes (*eau minérale, Roquefort*), (*beurre, Roquefort*) et (*baguette, Roquefort*) sont inappropriées.

6.2.4 Algorithme

Dans cette sous-section, nous présentons l'algorithme qui construit le graphe de dépendances.

Algorithme 6 : CONSTRUIRE_GRAPHE

Entrées : $cooc[n,n]$, $mincooc$, $mintol$

Sorties : ($G =< \mathcal{N}, \mathcal{T} >$)

```
1  Créer les nœuds N_1 jusqu'à N_n;
2  pour j allant de n jusqu'à 1 faire
3      placer_lien ← vrai;
4      pour i allant de j − 1 jusqu'à 1 faire
5          si cooc[i,j]/cooc[i,i] ≥ mincooc alors
6              T ← T ∪ (a_i, a_j);
7          fin
8          si cooc[i,j]/cooc[j,j] ≥ mintol alors
9              placer_lien ← faux;
10             sortir;
11         fin
12     fin
13     si placer_lien alors
14         T ← T ∪ ({}, a_i);
15     fin
16 fin
```

L'algorithme 6 construit le graphe de dépendances à partir de la matrice de cooccurrences. Il possède trois paramètres en entrée : la matrice de cooccurrence $cooc[n,n]$ où n est le nombre d'items fréquents, $mincooc$ et $mintol$. $\mathcal{N}$ représente l'ensemble de nœuds dans le graphe de dépendances. {} est le nœud initial et (a_i, a_j) représente l'arête allant du nœud a_i au nœud a_j.

La construction du graphe de dépendances s'effectue de la façon suivante. Comme l'indiquent les deux boucles *pour*, les items sont traités par ordre décroissant de leur support, c.-à-d. en considérant en premier les items les moins fréquents puis en intégrant progressivement ceux qui sont les plus fréquents. Deux items distincts a_i et a_j (identifiés respectivement par la ligne i et la colonne j de la matrice de cooccurrence) sont comparés pour déterminer si l'arête (a_i, a_j) doit être tracée. Sauf dans le cas où la propriété 1 (ou la propriété 2) est vérifiée, il existe une arête allant de a_i à a_j tant que la condition $supp(a_i) \geq supp(a_j)$ est vraie. La ligne 5 tient compte de la confiance (fréquence de

cooccurrence) tandis que les lignes 8 et 9 exploitent le seuil de tolérance (voir figure 6.1). En d'autres termes, la ligne 5 vérifie si a_i et a_j sont connectés dans la base de transactions tandis que la ligne 8 applique l'élagage mentionné dans la sous-section 6.2.3. À la ligne 14, les points d'entrée sont générés. Par défaut, aucun ("item") nœud du graphe de dépendances n'est connecté au nœud initial. Dès qu'un item n'apparaît "presque jamais" en présence d'un autre, alors on choisit de le relier au nœud initial.

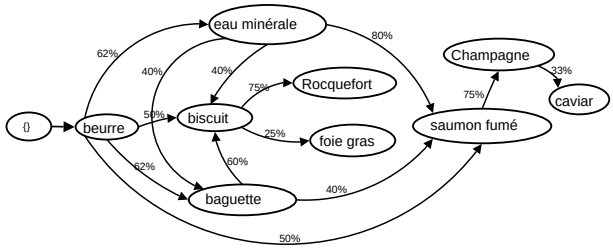

FIG. 6.2 – Graphe de dépendances complet avec *mincooc*=0 et un seuil de tolérance=100%.

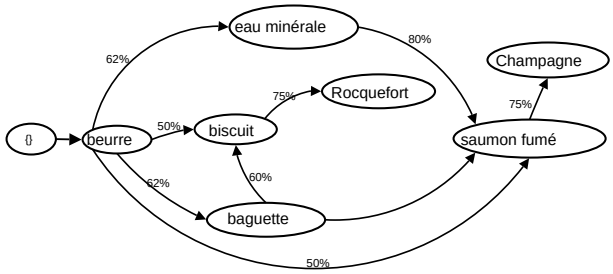

FIG. 6.3 – Graphe de dépendances avec $mincooc = 50\%$ et $mintol = 100\%$.

Comme mentionné ci-dessus, la confiance (exprimée comme la fréquence de cooccurrence entre deux items) est un moyen de réduire la taille du graphe de dépendances. La figure 6.3 représente le graphe partiel (par opposition au graphe complet de la figure 6.2) où le seuil de confiance *mincooc* vaut 50%. Par exemple, l'ensemble d'items {*beurre, eau minérale, baguette*} n'apparaîtra pas dans l'ensemble des MFF puisque le chemin correspondant dans le graphe n'existe pas du fait que *conf(eau minérale* $\Rightarrow$ *baguette)=40%*.

Preuve de complétude : Nous allons montrer que le graphe de dépendances G est une base complète pour générer tous les MFF lorsqu'aucune approximation n'est

utilisée (c.-à-d. *mincooc*=0% et *mintol*=100%). Tout d'abord, nous supposons qu'aucun élagage n'est appliqué en se basant sur la propriété 1. Dans ce cas, lorsque a_i et a_j apparaissent dans une même transaction T, alors, il y a toujours une arête allant de a_i vers a_j avec $supp(a_i) \geq supp(a_j)$. Si maintenant on tient compte de l'élagage décrit par la propriété 1, on cherche alors à montrer que l'arête (a_h, a_j) écartée ne contribue pas à la construction d'un MFF noté Z et représentant le chemin allant de l'état initial au nœud a_j. Ainsi, lorsque $g(a_j) \subseteq g(a_i)$, alors a_j n'aura qu'un seul prédécesseur a_i et le graphe G ne conduira pas à un MFF puisque $Z \supset \{a_h \cup a_j\}$ et $Z \cap a_i = \emptyset$. Ayant les propriétés $g(a_j) \subseteq g(a_i)$ et $g(Z) \subseteq g(a_j)$, alors $g(Z) \subseteq g(a_i)$ ce qui signifie que $supp(Z) = supp(Z \cup a_i)$ et donc Z n'est pas un MFF.

6.3 Génération des MFF

L'idée derrière CIGA+ est la suivante. Le support de $Y \cup a_j$ a plus de chances d'être élevé que le support de $Y \cup a_k$ si l'item le moins fréquent a_i dans Y est tel que $conf(a_i \Rightarrow a_j) > conf(a_i \Rightarrow a_k)$. Cela permet de se diriger rapidement vers les motifs de plus grande taille.

Par exemple, considérons le motif $Y = \{$*beurre, baguette*$\}$ (voir figure 6.2) où *baguette* est l'item dans Y qui possède le plus petit support. L'extension de Y est *g(Y)={3, 4, 6, 7, 8}*. Donc, le candidat le plus prometteur pour augmenter Y est *biscuits* plutôt que *saumon fumé* puisque *conf(baguette ⇒ biscuits)=3/5* tandis que *conf(baguette ⇒ saumon fumé)=2/5*. Ainsi, $g(Y \cup biscuits) = \{6, 7, 8\}$ et $g(Y \cup$ saumon fumé$) = \{3, 4\}$.

Dans cette section, nous décrivons tout d'abord l'algorithme CIGA+ en indiquant comment les *tidsets* fermés et les motifs sont calculés puis nous illustrons l'exécution de CIGA+ par un exemple.

6.3.1 Algorithme

L'algorithme CIGA+ (voir l'algorithme 7) possède quatre paramètres : la matrice de cooccurrences $cooc[n, n]$, *minsupp*, *mincooc* et *mintol*. La variable globale L est utilisée dans la procédure CIGA+_SUB pour stocker les *MFF*s calculés. CIGA+ procède en deux étapes. Tout d'abord, le prétraitement des données basé sur la construction du graphe de dépendances est effectué. Ensuite, un appel de la procédure récursive que

nous appelons CIGA+_SUB est effectué pour chaque nœud directement lié au nœud initial. Tant que le support du chemin courant (c.-à-d. l'ensemble d'items situés sur le chemin) est plus grand que *minsup* ou bien qu'un nœud terminal n'est pas atteint, l'algorithme 8 continue à parcourir le graphe de dépendances en explorant les successeurs du nœud courant.

Algorithme 7 : CIGA+

Entrées : $cooc[n,n]$,*minsup*, *mincooc*, *mintol*

Sorties : L : Ensemble des *MFF*

```
1  G ← CONSTRUIRE_GRAPHE (cooc[n,n],mincooc,mintol);
2  global L ← ∅;
3  pour chaque nœud a dans T({},a) faire
4  |  CIGA+_SUB(a, g(a),minsup);
5  fin
```

Algorithme 8 : CIGA+_SUB

Entrées : a, X, *minsupp*

```
1  X ← g(a) ∩ X;
2  candidat ← vrai;
3  si taille(X) > (minsup × |D|) et taille(X) > 1 alors
4  |  pour chaque succ de a faire
5  |  |  si CIGA+_SUB(succ,X,minsupp)=taille(X) alors
6  |  |  |  candidat = faux
7  |  |  fin
8  |  fin
9  |  si candidat alors
10 |  |  L ← L ∪ f(X);
11 |  fin
12 fin
13 retourner taille(X)
```

Dans l'algorithme 8, la variable a représente l'item associé au dernier nœud du chemin courant. X est le *tidset* associé au motif constitué du chemin courant dans le graphe. Le traitement dans l'algorithme 8 inclut : (i) le calcul de $X = g(Y)$ et du support de Y (c.-à-.d. la taille de X), (ii) une exploration récursive du graphe de dépendances, et (iii) le calcul des *MFF* (voir ligne 10). La première étape est la plus coûteuse. C'est pourquoi

il est important d'utiliser une structure de données adaptée pour stocker et retrouver les $g(a)$ ainsi qu'une manière efficace de calculer les intersections entre $g(a)$ et X. Nous avons utilisé un index pour les items individuels afin d'obtenir rapidement la valeur du *tidset* résultant.

Le calcul de $f(X)$ intervient lorsque le support du motif courant est inférieur à *minsupp* ou bien lorsqu'on atteint un nœud terminal (voir ligne 3 de l'algorithme 8). Chaque fois que l'on identifie une égalité entre le support de Y et $Y \cup a$ au cours du parcours, alors le calcul de la fermeture $g(Y)$ de Y n'est pas effectué puisque Y n'est pas fermé. Le calcul à la ligne 10 intervient lorsqu'aucun successeur du nœud courant a ne conserve le même support que celui du motif $Y = f(X)$ qui correspond au chemin se terminant par a. Ce type de calcul intervient plus ou moins souvent selon les seuils retenus pendant la construction du graphe de dépendances.

6.3.2 Calcul des *tidsets* et motifs fermés

La génération des MFF dans CIGA+ consiste à explorer le graphe de dépendances de façon récursive. Un chemin allant du nœud initial jusqu'au nœud courant représente la séquence d'items qui correspond au motif Y. Pour chaque chemin dans le graphe, le *tidset* correspondant [2] associé à la séquence d'items courante est calculé.

Propriété 3 *Soit X un tidset, Y un motif et $a_i \notin Y$. Alors, $X = g(Y \cup a_i)$ est équivalent à $g(Y) \cap g(a_i)$, et X est un tidset fermé.*

La propriété 3 permet de mettre à jour X en se basant sur la valeur de $g(Y)$ pour y inclure un nouvel item à partir d'une simple intersection. Cette propriété est utile pour générer rapidement le *tidset* associé au chemin courant dans le graphe de dépendances. De plus, nous savons que X est un *tidset* fermé puisqu'il est construit à partir d'une intersection entre deux *tidsets* fermés [9]. Par exemple, *g(beurre, cracker) = {5,6,7,8}*. Lorsque le nœud *Roquefort* est atteint (voir figure 6.3), le calcul de *g(beurre, cracker, Roquefort)* est effectué en faisant l'intersection des deux *tidsets* : $\{5, 6, 7, 8\}$ et $\{5, 6, 8\}$.

L'algorithme 8 parcourt le graphe en utilisant la propriété 3 pour générer les *tidsets* fermés X. Une fois que le *tidset* X relatif au chemin courant a été calculé et que sa taille est bien supérieure ou égale au support absolu, le motif correspondant $Y = g(X)$ peut

[2] On rappelle que $X = g(Y)$ et $Y = f(X)$.

être calculé. Cependant, un tel calcul se veut coûteux et parfois inutile. C'est la raison pour laquelle nous évitons de le faire systématiquement. Comme indiqué auparavant, un chemin dans le graphe de dépendances est exploré progressivement jusqu'à ce que le support de son motif associé devienne inférieur à *minsupp* ou bien qu'un nœud terminal soit atteint. Pendant ce parcours, à chaque fois que l'ajout d'un nœud au chemin courant ne réduit pas le support, on en déduit que le motif courant n'est pas fermé et on évite de calculer sa fermeture (ligne 5). Lorsque la condition de la ligne 3 n'a pas lieu, alors le motif devient un candidat intéressant et sa fermeture est calculée à la ligne 10.

6.3.3 Coût et structures de données

Le traitement complet de la génération des MFF comprend trois étapes principales : (i) la transformation des données, (ii) la construction du graphe de dépendances et (iii) l'extraction des MFF. La transformation des données et la construction du graphe font partie du prétraitement car leur coût est négligeable par rapport au calcul des MFF. La complexité de l'algorithme de construction du graphe de dépendances (algorithme 6) est $\mathcal{O}(m)$ où m représente le nombre d'items. La transformation des données, quant à elle, permet de réarranger la base pour obtenir de meilleures performances lors du calcul des MFF. L'opération la plus commune pendant ce calcul est l'intersection entre deux ensembles de transactions $g(I)$ et $g(a_i)$ où a_i est un item. Aussi, la base de données est réarrangée pour stocker tous les $g(a_i)$ sous forme binaire.

6.3.4 Exécution de l'algorithme

La figure 6.4 illustre l'exécution de l'algorithme CIGA+ à partir des données de la table 6.1 et du graphe de dépendances de la figure 6.2. Les paramètres *minsup*, *mincooc*, et *mintol* ont comme valeur 0%, 0% et 100% respectivement de sorte que l'algorithme génère l'ensemble complet des MFF.

Lorsque l'algorithme commence l'exploration du graphe de dépendances, chaque descendant direct du nœud initial est tout d'abord pris en compte (voir ligne 4 de l'algorithme 7). Dans la figure 6.4, on voit que seul le nœud a est directement accessible depuis la racine. $X = g(a)$ est alors égal à $\{1, 2, \ldots, 8\}$. Une fois que le nœud a est traité, ses successeurs b, c, et d sont alors pris en considération. Supposons que le chemin [a,c,g,h,i] est suivi. L'exploration du nœud h mène au calcul de $X = X_{a,c,g} \cap g(h) = \{3, 4\}$. Lorsque

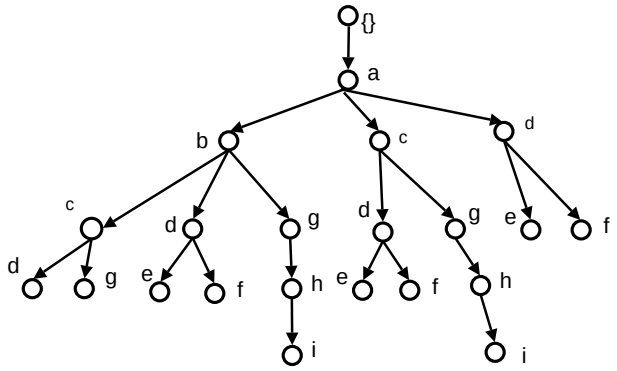

FIG. 6.4 – Simulation d'exécution de CIGA+.

le nœud i est atteint, X devient égal à $\{4\}$. Comme i est un nœud terminal, on calcule $f(X) = f(\{4\}) = \{a, c, g, h, i\}$ et ensuite on obtient un MFF. Sachant que le support reste constant en passant de la séquence $[a, c, g, h]$ à $[a, c, g]$, il n'est pas nécessaire d'effectuer un calcul de fermeture au niveau du nœud g. La table 6.3 représente l'ensemble complet des motifs fermés MF générés à partir de la table 6.1.

MF	support	MF	support
abcdefghi	0	abg	3/8
acghi	1/8	abc	3/8
abcgh	1/8	agh	3/8
abcdf	1/8	adf	3/8
acde	1/8	ag	4/8
abgh	2/8	ac	5/8
acdf	2/8	ab	5/8
acgh	2/8	ad	4/8
abdf	2/8	a	1

TAB. 6.3 – Liste des motifs fermés

6.4 Étude expérimentale

Il est difficile de faire une étude comparative de CIGA+ avec d'autres algorithmes puisqu'il n'existe aucune procédure qui utilise le même type d'approximation que la nôtre. Toutefois, BAMBOO [37] est un algorithme efficace de génération des MFF basé

sur *CLOSET+* qui permet aussi de faire de l'approximation mais sous d'autres formes et hypothèses. Comme les deux algorithmes partagent un objectif similaire, nous en faisons une étude comparative dans cette section. Cependant, nous écartons l'usage du support décroissant dans BAMBOO et optons pour un support fixe puisqu'aucune méthode rigoureuse n'est fournie par les auteurs pour générer la fonction du support décroissant pour une base de données spécifique. Sans la contrainte de support décroissant, BAMBOO est très proche de l'algorithme CLOSET+.

Les expérimentations ont été conduites sous GNU Linux en utilisant un Pentium IV à 3 Ghz et avec 1024 Mo de mémoire. La comparaison des deux algorithmes s'est faite selon leur temps d'exécution et le nombre de MFF générés en considérant trois types de bases de transactions connues dans la communauté de fouille de données : *Chess*, *Pumsb* et *Mushroom*.

Comme le montrent les figures ci-après, CIGA+ affiche des temps d'exécution plus faibles car il génère moins de MFF. Mais dans tous les cas, le taux d'accroissement des courbes que nous obtenons est fortement similaire pour les deux algorithmes, ce qui indique des performances identiques si on se ramène à la même échelle. Toutefois, CIGA+ est relativement plus stable et offre une performance relative plus avantageuse lorsque le volume de données devient grand ou lorsque le support minimal devient très faible. Par exemple, avec *minsup*=45%, BAMBOO génère une sortie dix fois plus volumineuse que celle produite par CIGA+, et nécessite 51 fois plus de temps à s'exécuter avec la base Pumsb en utilisant $mintol = 95\%$ et $mincooc = 10\%$.

La figure 6.5 montre les performances de CIGA+ par rapport à BAMBOO lorsque *mincooc*= 10% et *mintol*= 95%. La valeur non nulle du premier paramètre permet d'écarter certains MFF peu pertinents conduisant par la suite à un ensemble plus faible de règles d'associations. Avec *minsup*= 20%, CIGA+ genère une sortie qui est 254 fois plus petite que celle de BAMBOO et s'exécute 620 fois plus rapidement que BAMBOO. La courbe de la figure 6.6 confirme nos conclusions pour la base de données dense *Pumsb*.

La figure 6.8 compare les temps d'exécution de CIGA+ avec Bamboo lorsqu'aucune approximation n'est utilisée. CIGA+ est alors moins intéressant dans ce cas de figure. Celà s'explique par le fait que le graphe de dépendances devient exhaustif. Toutes les combinaisons de motifs sont générées (sauf dans le cas de la propriété 1) ce qui rend le

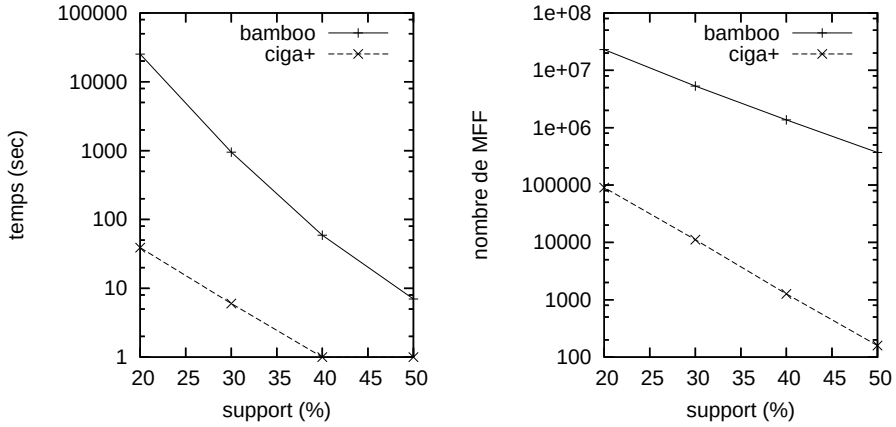

FIG. 6.5 – Influence du support sur les temps d'exécution et le nombre de MFF pour la base de données *Chess*.

graphe long à parcourir. Le test de fermeture des MFF devient facultatif et CIGA+ peut être modifié pour fonctionner sans approximation afin d'obtenir de meilleurs résultats.

Nous n'avons pas illustré l'influence du seuil de tolérance dans les graphiques. En effet, le comportement de CIGA+ face au paramètre *mintol* dépend fortement des données qui sont utilisées. Pour des données denses, l'élagage a tendance a être plus important au fur et à mesure que ce seuil diminue.

6.5 Ensembles rares

La recherche d'ensembles fréquents est utile pour mettre en évidence les comportements généraux d'un groupe d'individus (objets). Par opposition, la recherche des motifs fermés rares (MFR) permet de repérer des comportements exceptionnels en énumérant des listes d'items (propriétés) qui apparaissent peu souvent ensemble. À titre d'exemple, on pour trouver un MFR indiquant qu'il est rare d'avoir des transactions (paniers de consomateurs) dans lesquelles on retrouve du champagne avec du pain baguette.

Définition 9 *L'ensemble des MFR correspond à l'ensemble des motifs fermés ayant un support inférieur au support maximal* maxsupp.

L'algorithme CIGA+ est capable, moyennant une adaptation, de rechercher les MFR de manière exacte ou approximative selon les paramètres utilisés. Cela revient à modi-

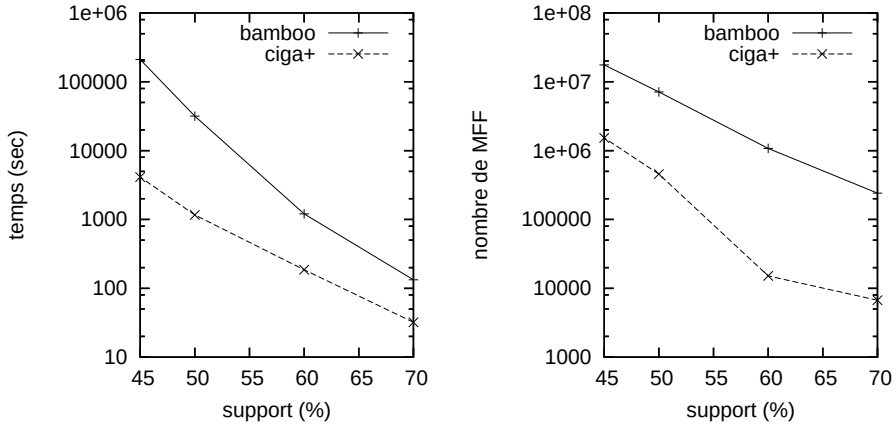

FIG. 6.6 – Influence du support sur les temps d'exécution et le nombre de MFF pour la base de données *Pumsb*.

fier le graphe de dépendances afin qu'il mette en évidence les items qui possèdent des confiances faibles deux à deux. En tout, deux modifications doivent être apportées :

1. Les arêtes du graphe de dépendances doivent représenter une confiance inférieure à une confiance maximale.
2. L'algorithme fonctionne avec un support minimal nul. On ne cherche la fermeture du motif que lorsque le support est inférieur au support maximal.

Cette solution génère tous les ensembles de transactions mais n'effectue un calcul de fermeture qu'en présence des ensembles rares.

Les expérimentations nous montrent que la confiance donne une forte influence sur le résultat. L'utilisation du seuil de tolérance quant à lui n'a d'effet que sur le taux d'approximation que l'on souhaite obtenir. Il permet de simplifier le nombre d'arêtes dans le graphe et donc d'améliorer les temps d'exécution de l'algorithme.

6.6 Conclusion et travaux futurs

Nous avons présenté un algorithme appelé CIGA+ qui permet de calculer un ensemble concis de MFF en se focalisant sur ceux qui sont les plus prometteurs. CIGA+ réduit l'ensemble des MFF grâce à l'utilisation de deux nouveaux paramètres : la confiance entre deux items élémentaires et le seuil de tolérance. Le premier paramètre a pour but d'écarter les liens faibles entre deux items individuels. Le seuil de tolérance *mintol* permet

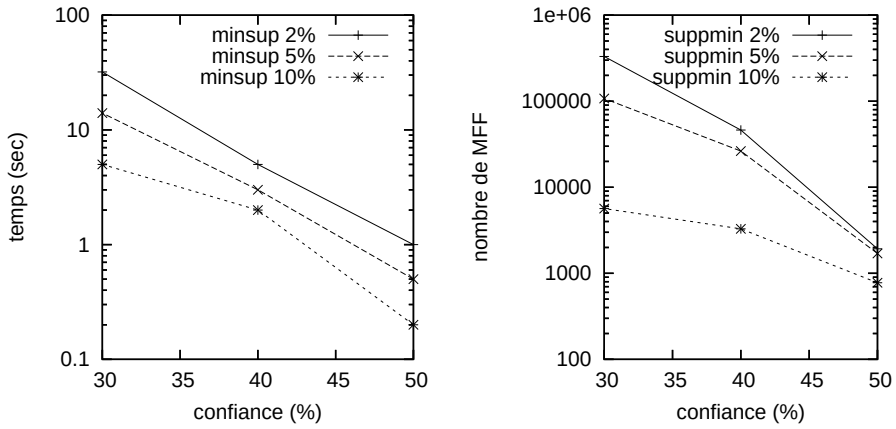

FIG. 6.7 – Influence du paramètre *mincooc* sur la base de données *Mushroom*.

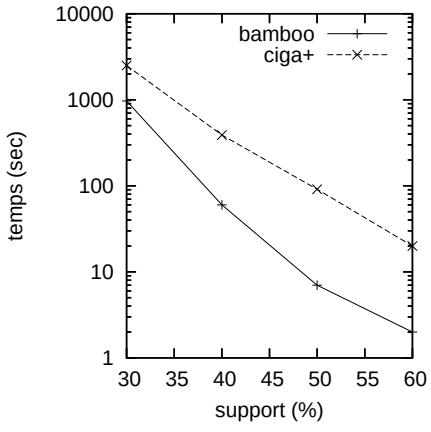

FIG. 6.8 – Comparaison de CIGA+ et Bamboo sans approximation sur la base *Chess*.

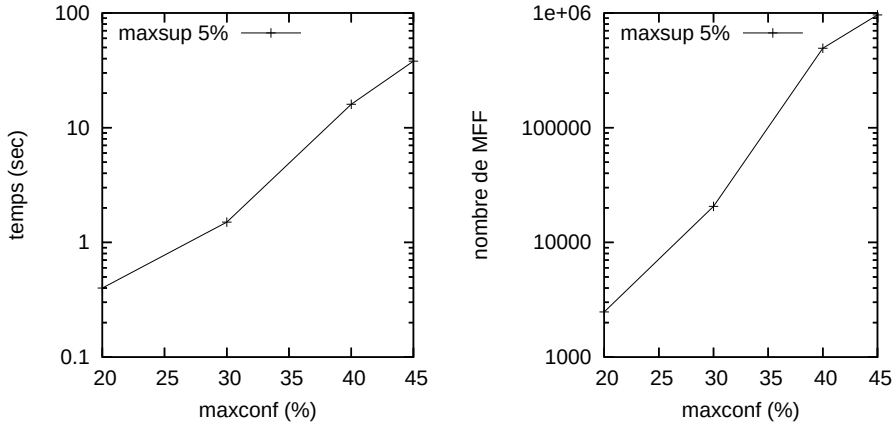

FIG. 6.9 – Recherche des ensembles rares et influence de la confiance sur la base Chess.

de simplifier le graphe de dépendances en identifiant les paires d'items qui apparaissent souvent ensemble. Les arêtes peu pertinentes du graphe de dépendances sont ainsi écartées, ce qui améliore le parcours du graphe et réduit l'ensemble de MFF qu'il contient. Le paramétrage de *mintol* dépend du degré d'approximation souhaité. Une valeur de 100% permet d'obtenir l'ensemble complet de MFF tandis qu'une valeur faible conduit vers une forte approximation de cet ensemble. En exploitant ces deux paramètres, l'utilisateur est en mesure d'avoir le contrôle sur le temps d'exécution de l'algorithme ainsi que sur la taille du résultat et le nombre de règles d'association qu'il souhaite obtenir.

Notre prochaine étape consiste à enrichir le cadre de CIGA+ en lui permettant de calculer les générateurs ainsi que les inclusions (ordre partiel) entre les MFF [9] calculés afin d'exploiter des travaux antérieurs sur la génération de règles d'associations et la construction du treillis de concepts. Le but étant de faire de CIGA+ une procédure complète pour la génération des règles d'associations. Le cadre proposé pourra aussi être adapté pour permettre une construction efficace du treillis de concepts et de l'iceberg[3] de concepts [29].

[3]Treillis partiel comportant seulement les concepts fréquents.

Chapitre 7

Opérateurs de manipulation du treillis de concepts

Jusqu'à maintenant nous avons vu des méthodes d'approximation qui permettent d'obtenir un ensemble plus concis de concepts. Dans ce chapitre, nous nous intéressons à l'aspect exploratoire d'un treillis directement construit de manière classique à partir du contexte ; c.-à-d. sans approximation. L'objectif est toujours de faciliter l'exploitation du résultat par un utilisateur qui n'est pas intéressé par la liste exhaustive des concepts et qui souhaite se concentrer sur des vues particulières. Les mécanismes que nous offrons permettent de se placer à différents niveaux de granularité dans le treillis, offrant ainsi plusieurs niveaux de détails spécialisés sur un/des sous-ensembles du problème. En se basant sur le savoir-faire des entrepôts de données, nous définissons un ensemble d'opérateurs qui s'inspirent des techniques OLAP (On-Line Analytical Processing) et de l'algèbre relationnel pour opérer des modifications sur un treillis de concepts.

7.1 Opérateurs sur les treillis

Nous décrivons deux opérations principales sur les treillis : la projection et l'assemblage. Plus de détails sur des résulats théoriques se trouvent dans [9, 32]. Nous ferons appel à un troisième opérateur que nous appelons *sélection*. Il consiste à extraire, à partir du treillis, une liste de concepts qui possèdent un ensemble d'objets et/ou d'attributs en commun. La sélection n'est pas une opération isomorphique puisqu'elle ne renvoit pas un treillis mais une liste de concepts (contrairement à la projection et l'assemblage).

Définition 10 *La* PROJECTION *de* $\mathcal{L} = B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *sur* $\mathcal{A}_1 \subset \mathcal{A}$ *est un treillis* $\mathcal{L}_1 = B(\mathcal{O}, \mathcal{A}_1, \mathcal{R}_1)$ *donné par* $\varphi : B(\mathcal{O}, \mathcal{A}, \mathcal{R}) \rightarrow B(\mathcal{O}, \mathcal{A}_1, \mathcal{R}_1)$.
La fonction φ *transforme le concept* (X, Y) *du treillis initial* $\mathcal{L}$ *en un concept du treillis résultant* $\mathcal{L}_1$ *en projetant son intention sur l'ensemble d'attributs* $\mathcal{A}_1$ *:*

$$\varphi((X, Y)) = ((Y \cap \mathcal{A}_1)', Y \cap \mathcal{A}_1)^1.$$

Définition 11 *L'* ASSEMBLAGE *de deux treillis* $\mathcal{L}_1 = B(\mathcal{O}, \mathcal{A}_1, \mathcal{R}_1)$ *et* $\mathcal{L}_2 = B(\mathcal{O}, \mathcal{A}_2, \mathcal{R}_2)$ *est une structure obtenue à partir du produit direct de* $\mathcal{L}_1$ *et* $\mathcal{L}_2$ *défini par* $\psi : B(\mathcal{O}, \mathcal{A}_1, \mathcal{R}_1) \times B(\mathcal{O}, \mathcal{A}_2, \mathcal{R}_2) \rightarrow B(\mathcal{O}, \mathcal{A}, \mathcal{R})$.
La fonction ψ *transforme un couple de concept* $\mathcal{L}_1$ *et* $\mathcal{L}_2$ *en un concept global en faisant l'intersection de leurs extensions respectives :*

$$\psi(\langle (X_1, Y_1)\ ,\ (X_2, Y_2) \rangle) = \langle X_1 \cap X_2\ ,\ (X_1 \cap X_2)' \rangle\ .$$

La fonction φ *est fondamentale dans les cadre des treillis imbriqués*[2] *puisque l'ensemble des noeuds qui représententent* $\mathcal{L}$ *dans* $\mathcal{L}_1 \times \mathcal{L}_2$ *sont exactement les images des concepts venant de* $\mathcal{L}$ *par la fonction* φ.

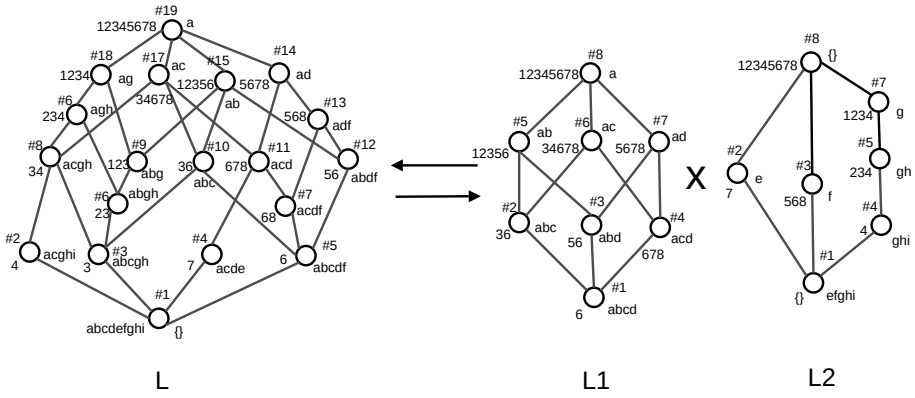

FIG. 7.1 – Projection du treillis de gauche sur les deux sous-ensembles d'attributs disjoints $abcd$ et $efghi$.

[1]On rappelle que l'opérateur ' appliqué à une extension (resp. intention) retourne l'intention (resp. extension) qui lui est associée.

[2]Un treillis imbriqué revient à décomposer les attributs en plusieurs sous-ensembles qui représentent des niveaux. Le niveau n du treillis imbriqué représente le treillis construit avec le sous-ensemble n d'attributs et dont les noeuds contiennent le treillis construit au niveau $n-1$.

7.2 Comparaison avec les opérateurs OLAP

Les opérateurs OLAP permettent d'effectuer des manipulations sur des cubes de données. L'objet de cette section n'est pas de fournir de la connaissance sur les entrepôts de données (*data warehouse*) mais simplement de proposer une adaptation des opérateurs OLAP au contexte de l'analyse formelle de concepts en vue de permettre à l'utilisateur de visualiser des treillis de concepts selon diverses perspectives en terme d'objets ou de propriétés. Nous montrons ici comment les différents opérateurs OLAP (*drill-down*, *roll-up*, *slice-and-dice*, et *drill through*) ainsi que d'autres opérateurs (*nest-unnest*, *select*, *split*) peuvent être transposés dans le cadre du *data mining* en tant qu'utilisations particulières de la projection et de l'assemblage :

- ***Roll-up*** : monte dans la hiérarchie d'attributs d'un ou plusieurs attributs (c.-à-d. modifie la granularité des données en montant d'une catégorie) ou ignore un sous-ensemble des attributs. Exemple : remplacer l'item "beurre" par son groupe "produit laitiers".
- ***Drill-down*** : descend dans la hiérarchie de dimensions d'un ou plusieurs attributs ou bien ajoute une nouvelle variable. Il en résulte une augmentation du nombre d'attributs.
- ***Slice*** : limite l'analyse à un seul attribut. Les colonnes du contexte sont tronquées. Par exemple, on ne s'intéresse qu'aux ventes de Champagne.
- ***Dice*** : focalise sur un sous-ensemble d'attributs et/ou d'objets.
- ***Select*** : effectue une sélection sur un ensemble d'objets. Seule une partie des individus (objets) est alors prise en compte.

D'autres opérateurs d'exploration peuvent être définis :

- ***Drill-through*** : retourne au contexte pour voir le sous-ensemble d'objets qui correspond à un concept donné.
- **Browse** : sélectionne un ensemble de nœuds dans le treillis qui correspondent au filtre ou à l'idéal d'un ensemble d'objets ou d'attributs.
- **Nest-unnest** : représente le treillis comme une structure de treillis imbriqué ou plate.
- ***Split*** : décompose le treillis $\mathcal{L}$ selon un attribut donné A_i en k treillis plus petits, où k represente le nomnre de modalités dans A_i. Cette opération est un cas particulier de l'imbrication où le treillis initial est équivalent au treillis de A_i et le treillis interne est la projection de $\mathcal{L}$ sur $\mathcal{L} - A_i$.

La table suivante résume les différentes opérations applicables sur un treillis vis à vis des opérations OLAP.

Méthode / Opérateurs OLAP	**Projection**	**Assemblage**	**Sélection**
Roll-up	X	X	
Drill-down	X	X	
Slice	X		
Dice	X		X

7.3 Algorithmes

Cette section analyse de plus près le fonctionnement de la projection et de l'assemblage et définit leur implémentation.

L'algorithme 9 implemente la projection et utilise deux arguments en entrée : l'ensemble d'attributs P_1 sur lequel on projette, et le nœud n qui représente au départ de l'algorithme l'infimum du treillis (c.-à-d. le nœud du bas). Cette procédure parcourt le treillis du bas vers le haut (jusqu'au *supremum*) de manière récursive. La partie ascendante de cette réccursivité a pour tâche de sélectionner et formatter les concepts à garder dans le résultat, tandis que la partie descendante réalise l'ordre partiel. La fonction *mettre*(n') retourne le nœud n du treillis initial qui correspond au nœud n' dans le treillis final à l'aide d'une correspondance entre les deux structures. La fonction *retrouver*(n) effectue l'opération inverse en retournant le nœud n associé au nœud n' du treillis projeté. La fonction *élagage_des_parents* vérifie si certains parents du nœud n sont eux-même inclus dans un autre parent du même nœud afin de les éliminer.

La table 7.1 fournit une exécution de l'algorithme et montre que l'*infimum* du résultat de la projection est rapidement atteint sachant que son filtre est très proche du résultat recherché. Dans de nombreux cas, la projection se comporte comme un algorithme glouton une fois que cet infimum a été atteint.

L'algorithme 10 réalise l'assemblage de deux treillis $\mathcal{L}_1$ et $\mathcal{L}_2$. Une manière simple d'implémenter cette opération consiste à comparer chaque nœuds de $\mathcal{L}_1$ avec chaque nœud de $\mathcal{L}_2$. Afin d'obtenir rapidement des concepts dans le résultat, chacun des treillis est parcouru de bas en haut à l'aide d'un parcours en profondeur.

Algorithme 9 : Projection

Entrées : Nœud n, Ensemble d'attributs P_1
Sorties : Nœud n'

```
1  si déjà_visité(n) alors
2      retourner retrouver(n')
3  successeurs ← ∅;
4  i1 ← INT(n) ∩ P1;
5  parents ← sort(parents(n), DECR);
6  elagage_des_parents();
7  pour chaque parents p faire
8      i2 ← INT(p) ∩ P1;
9      si i1 = i2 alors
10         retourner Projection(p, P1);
11     successeurs ← successeurs ∪ projection(p, P1);
12     déjà_visité(n) ← vrai ;
13 n' ← i2;
14 lien(n',successeurs);
15 mettre(n);
16 retourner n';
```

n	i1	successeurs	à garder
1	abcd	5	non
5	abcd	7,10,12	oui
10	abc	15,17	oui
15	ab	19	oui
19	a	NIL	oui
17	ac	19	oui
7	acd	11	non
11	acd	14,17	oui
14	ad	19	oui
12	abd	13,15	oui
13	ad	14	non

TAB. 7.1 – Exécution de l'algorithme de projection appliqué sur le treillis L (partie gauche) de la figure 7.1.

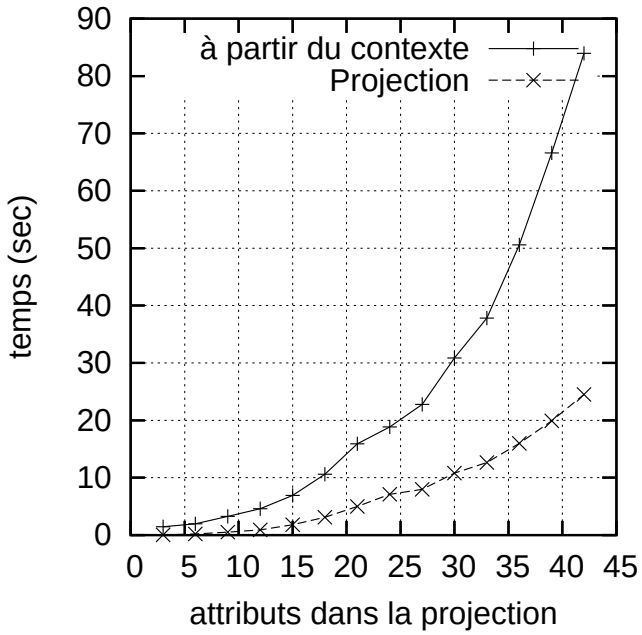

FIG. 7.2 – Comparaison des coûts d'exécution de la génération du treillis à partir du contexte avec l'algorithme de projection

Algorithme 10 : Assemblage

Entrées : $L_1 = (c_1, \leq_{L1})$, $L_2 = (c_2, \leq_{L2})$
Sorties : $L = (c, \leq_L)$

```
1 tri(c1) ; Tri(c2);
2 E ← ∅;
3 pour chaque ci dans c1 faire
4     pour chaque cj dans c2 faire
5         E ← EXT(ni) ∩ EXT(nj);
6         si E ⊄ E alors
7             c ← c ∪ (E, INT(ni) ∪ INT(nj));
8             lien(c,ci,cj);
9 retourner L
```

Plusieurs questions se posent quant à l'utilisation des opérateurs d'assemblage et de projection :

- Est-il plus intéressant d'exécuter une opération directement sur le treillis plutôt que de recalculer un treillis à partir du contexte (ou par apposition de contextes) ?
- Est ce que ces opérateurs ont d'autres intérêts ?

Pour répondre à la première question, nous avons conduit des expérimentations qui montrent que l'application de la projection à un treillis est plus efficace que la reconstitution du treillis à partir du contexte en y sélectionnant un sous-ensemble des attributs. La projection est d'autant plus efficace que le nombre d'attributs sur lesquels on projette augmente. La figure 7.2 illustre ceci pour des contextes de 500 objets et 50 attributs. D'autres expérimentations sur des bases plus grandes pourront être effectuées dans le futur. Pour l'algorithme d'assemblage [32], des tests ont montré que cette opération obtient elle aussi des résultats empiriques et théoriques intéressants.

D'autre part, la projection s'avère intéressante dans le cas des treillis imbriqués pour lesquels chaque structure sous-jacente est une projection d'un treillis complet sur un sous-ensemble d'attributs. L'affichage d'un treillis imbriqué nécessite donc de nombreux calculs de sous-treillis et l'algorithme de projection permet d'obtenir une navigation beaucoup plus fluide dans ce cas de figure.

Chapitre 8

Conclusion

Nous venons de voir trois méthodes très différentes qui ont toutes comme objectif d'aider l'utilisateur à manipuler un ensemble de concepts très grand tant au niveau du résultat proposé qu'au niveau des temps d'exécution. L'algorithme LatMat se focalise sur les temps d'exécution en utilisant une technique simpliste et déterministe de génération des concepts dont le temps de calcul est très faible par rapport à la construction du treillis. Le principal inconvénient de cet algorithme réside dans la perte d'information difficile à identifier. La qualité du résultat dépend très fortement du type des données puisque la quasi-totalité des concepts apparaît dans des contextes de faibles densités. Or, on sait que la densité d'un contexte est difficilement estimable. L'algorithme CIGA+ corrige ce défaut de LatMat en apportant une solution qui, à la base, n'est pas approximative mais qui le devient grâce à un paramétrage. L'utilisateur est donc en mesure de choisir lui-même la précision qu'il souhaite voir apparaître dans le résultat de l'algorithme. L'étude empirique de CIGA+ donne des résultats intéressants lorsque le taux d'approximation est moyen ou élevé. Grâce aux trois paramètres de CIGA+, l'utilisateur est capable de choisir à l'avance le degré d'approximation qui a alors un impact sur le volume des résultats et le temps de calcul. Un autre avantage de CIGA+ est de proposer rapidement les motifs de grandes tailles, c.-à-d. ceux qui sont les plus intéressants. Le résultat, même fortement approximé, contient alors suffisamment d'information pour générer des règles de grande taille. Enfin le troisième aspect abordé concerne une approche différente. Les opérateurs de manipulation de treillis, en particulier la projection, ne sont pas une forme d'approximation puisqu'ils nécessitent de calculer le treillis de manière classique pour ensuite y appliquer des transformations. La projection est utile pour proposer rapidement des vues pertinentes à l'utilisateur à partir du treillis complet

qui est très souvent illisible à cause de sa taille. L'assemblage, quant à lui, est utile pour compléter et mettre à jour un treillis sans devoir le recalculer entièrement.

La prochaine étape consiste à enrichir le cadre de CIGA+ en lui permettant de calculer les générateurs ainsi que l'ordre partiel entre les motifs fermés fréquents calculés afin d'exploiter au mieux les travaux de l'équipe sur les règles d'association. Le but étant de faire de CIGA+ une procédure complète pour la génération des règles d'associations. Concernant les opérateurs de manipulation du treillis, la solution proposée pour la projection est efficace et pourra être intégrée dans plusieurs applications, notamment la construction des treillis imbriqués. La version actuelle de l'algorithme d'assemblage est naïve et relativement coûteuse en temps. Cependant, des améliorations sont en cours d'étude.

Bibliographie

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

[2] Bastide, Y. & Taouil, R. & Pasquier, N. & Stumme, G. & Lakhal L., *Mining Frequent Patterns with Counting Inference* SIGKDD Explorations, ACM Computer, 2(2), pages 66-75, 2000.

[3] Bayard, R.J., *Efficiently Mining Long Patterns from Databases* , In Proc. of the ACM SIGMOD Conference on Management of Data, pages 85-93, 1998.

[4] Bordat, J.-P., *Calcul pratique du treillis de Galois d'une correspondance.* Mathématique et Sciences Humaines, 96, pages 31-47, 1986.

[5] Boros, E. & Gurvich, V. & Khachiyan, L. & Makino, K., *On the Complexity of Generating Maximal Frequent and Minimal Infrequent Sets*, Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science, pages133-141, March 14-16, 2002.

[6] Burdick, D. & Calimlim, M. & Gehrke, J., *MAFIA : a maximal frequent itemset algorithm for transactional databases.* In Proceedings of the 17th IEEE ICDE Conference (ICDE'01), Heidelberg, Germany, pages 443-452, 2001.

[7] Chein, M., *Algorithme de recherche de sous-matrices premières d'une matrice.* Bull. Math. de la soc. Sci. de la R.S. de Roumanie, 13, 1969.

[8] Ganter, B., *Two basic algorithms in concept analysis* (preprint). Technical Report 831, Technische Hochschule, Darmstadt, 1984.

[9] Ganter, B. & Wille, R., *Formal Concept Analysis : Mathematical Foundations.* Springer, Berlin-Heidelberg, 1999.

[10] Godin, R. & Missaoui, R., *An incremental concept formation approach for learning from databases.* Theoretical Computer Science, volume 133, pages 387-419, 1994.

[11] Godin, R. & Missaoui, R. & Alaoui, H., *Incremental concept formation algorithms based on galois lattices. Computational Intelligence*, 11(2), pages 246-267, 1995.

[12] Han, J. & Pei, J. & Yin, Y., *Mining Frequent Patterns without Candidate Generation*, Proc. 2000 ACMSIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, 2000.

[13] Hipp, J. & Guntzer, U. & Nakhaeizadeh, G., *Algorithms for association rule mining - a gen-eral survey and comparison*, SIGKDD Explorations, 2(1), pages 58-64, 2000.

[14] Jatteau, G. & Missaoui, R. & M., Sarifuddin, *Un automate pour la génération complète ou partielle des concepts du treillis de Galois*, Actes des 5èmes Journées d'Extraction et Gestion des Connaissances (EGC'2005), France, v.1, pages 147-158, Janvier 2005.

[15] Jatteau, G. & Missaoui, R., *Computing a Concise Set of Frequent Closed Itemsets for Association Rule Mining*, ICHSL.5/CAPS.5, Marrakech, pages 41-60, Novembre 2005.

[16] Kuznetsov, S. & Objedkov, S., *Algorithms for the construction of the set of all concept and their line diagram*, preprint MATH-AL-05-2000, Technische Univerität, Dresden, June 2000.

[17] Lakshmanan, L. & Raymond, Ng. & Han, J. & Pang, A., *Optimization of constrained frequent set queries with 2-variable constraints*, ACM SIGMOD Record, v.28 n.2, pages 157-168, June 1999.

[18] Lakhal, L. & Pasquier, N. & Bastide, Y. & Taouil, R., *Efficient mining of association rules using closed itemset lattices*, Information Systems, Volume 24 , pages 25-46, 1999.

[19] Le Floc'h, A. & Fisette, C. & Missaoui, R. & Valtchev, P. & Godin, R., *JEN : un algorithme efficace de construction de générateurs pour l'identification des règles d'association*, numéro spécial de la revue des Nouvelles Technologies de l'Information, Vol. 1, No. 1, pages 135-146, Editions Cépaduès, 2003.

[20] Luxenburger, M., *Implications partielles dans un contexte*, Mathématiques et Sciences Humaines, 29(113), pages 35-55, 1991.

[21] Missaoui, R. & Jatteau, G., *CIGA+ : un algorithme de calcul d'un ensemble concis de motifs fermés fréquents*, e-TI, volume 2, Novembre 2005.

[22] Norris, E.M., *An algorithm for computing the maximal rectangles in a binary relation*. Revue Roumaine de Mathématiques Pures et Appliqués, 23(2), pages 243-250, 1978.

[23] Nourine, L. & Raynaud, O., *A fast algorithm for building lattice*,Information Processing Letters, volume 71, pages 199-204, 1999.

[24] Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L., *Efficient Mining of Association Rules using Closed Itemset Lattices*. Information Systems, Elsevier Science, 24(1), pages 25-46, 1999.

[25] Pasquier, N., *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.

[26] Pei, J. & Han, J. & Mao, R., *Closet : An efficient algorithm for mining frequent closed itemsets*. In SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery, 2000.

[27] Pfaltz, J. & Taylor, C., *Scientific discovery through iterative transformations of concept lattices.* In Proceedings of the 1st International Workshop on Discrete Mathematics and Data Mining, Washington (DC), pages 65-74, 2002.

[28] Sarawagi S. & Agrawal R. & Megiddo N., *Discovery-driven exploration of OLAP data cubes.* Proc. of the Sixth Int'l Conference on Extending Database Technology (EDBT), Valencia, Spain, March 1998.

[29] Stumme, G. & Taouil, R. & Bastide, Y. & Pasquier, N. & Lakhal. L., *Computing Iceberg Concept Lattices with Titanic.* Data and Knowledge Engineering, 42(2), pages 189-222, 2002.

[30] Valiant, L. G. *The complexity of enumeration and reliability problems, SIAM Journal on Computing*, 8, pages 410-421, 1979.

[31] Valtchev, P. & Missaoui, R., *Building concept (Galois) lattices from parts : generalizing the incremental methods.* In H. Delugach and G.Stumme, editors, Proceedings, ICCS-01, volume 2120 of Lecture Notes in Computer Science, Stanford (CA), USA, Springer-Verlag, pages 290-303, 2001.

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

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

[34] Valtchev, P. & Missaoui, R. & Hacene, M. R. & Godin, R., *Incremental maintenance of association rule bases.* In Proceedings of the 2nd Workshop on Discrete Mathematics and Data Mining, San Francisco, 2003.

[35] Valtchev, P. & Missaoui, R. & Godin, R., *Formal Concept Analysis for Knowledge and Data Discovery : New Challenges, Proceedings of the Second International Conference on Formal Concept Analysis*, ICFCA, Sydney, Australia, pages 352-371, 2004.

[36] Wang, J. & Han, J. & Pei, J., *CLOSET+ : searching for the best strategies for mining frequent closed itemsets, ninth ACM SIGKDD international conference on Knowledge discovery and data mining.* 2003.

[37] Wang, J. & Karypis, G., *BAMBOO : Accelerating Closed Itemset Mining by Deeply Pushing the Length-Decreasing Support Constraint*, SDM, 2004.

[38] Zaki, M.J., *Generating non-redundant association rules*, KDD : pages 34-43, 2000.

[39] Zaki, M.J. & Hsiao, C.J., *CHARM : An Efficient Algorithm for Closed Itemset Mining*, In Proceeding of the 2nd SIAM International Conference on Data Mining (ICDM'02), Arlington, 2002.