Données semi-structurées. Découverte, maintenance et analyse de tendances

Pierre-Alain Laur, Maguelonne Teisseire, Pascal Poncelet

Données Semi Structurées : découverte, maintenance et analyse de tendances

P.A. Laur [1] – M. Teisseire [1] – P. Poncelet [2]

[1] LIRMM, 161 rue Ada, 34392 Montpellier cedex 5, France
{Laur, Teisseire}@lirmm.fr
[2] EMA/LGI2P, Ecole des Mines d'Alès
Site EERIE, Parc Scientifique Georges Besse, 30035 Nîmes cedex 1, France
Pascal.Poncelet@ema.fr

RÉSUMÉ. *La recherche de connaissances dans des données structurées a fait l'objet de nombreux travaux de recherche ces dernières années. Cependant, avec la popularité du Web, le nombre de documents semi structurés augmente très rapidement et il est judicieux de penser qu'une requête sur la structure des documents devient aussi importante qu'une requête sur les données elles mêmes. Dans cet article nous proposons une approche pour extraire de telles sous structures. De plus, les données évoluant sans cesse, nous étendons l'approche pour prendre en compte l'évolution de ces données sources dans le cadre d'un processus d'extraction. Enfin, nous montrons qu'il est possible d'analyser finement les tendances au cours des différentes évolutions des données sources.*

ABSTRACT. *Mining knowledge from structured data has been extensively addressed in the few past years. However, with the growing popularity of the Web, the number of semi structured documents available is rapidly increasing and it is judicious to assume that a query on document structure is almost as important as a query on data. In this paper, we propose an approach to extract such structures. Moreover, manipulated data is constantly being updated; we extend our approach to take into account source evolutions in a knowledge extraction process. Finally, we show that it is possible to analyze trends during the different data sources evolutions.*

MOTS-CLÉS : *données semi structurées, extraction de connaissances, évolutions des sources de données, tendances.*

KEYWORDS: *semi structured data, knowledge discovery, data sources evolutions, trends.*

1. Introduction

Avec la popularité du Web, le nombre de documents semi structurés disponibles augmente très rapidement. Cependant, malgré cette irrégularité, il est judicieux de penser qu'une requête sur la structure de ces documents est aussi importante qu'une requête sur les données [WaLi99]. Il peut, en effet, exister des similitudes parmi les objets semi structurés et il est même fréquent de constater que des objets qui décrivent le même type d'information possèdent des structures similaires. L'analyse de celles-ci peut alors fournir des renseignements importants pour : optimiser les évaluations de requêtes, obtenir des données générales sur le contenu, faciliter l'intégration de données issues de différentes sources, améliorer le stockage, faciliter la mise en place d'index ou de vues …De nombreux domaines d'applications existent et regroupent, par exemple, la bioinformatique, le Web sémantique ou le Web Usage Mining. Si jusqu'à présent la plupart des approches d'extraction de connaissances se sont concentrées sur des données structurées, de nouvelles approches ont été définies pour rechercher de telles sous structures et portent généralement le nom de Schema Mining [AsAb02, Wali97, WaLi99, LaPo03, LaTe03, Zaki02]. Cependant, même si des approches efficaces existent, elles ne considèrent malheureusement que des données de types statiques, i.e. ne prennent pas en compte le fait que les données puissent évoluer au cours du temps. Dans ce cas, les connaissances extraites ne sont alors plus représentatives du contenu des bases associées.

Dans cet article, nous nous intéressons à l'extraction de sous structures fréquentes, ou plus exactement de sous arbres fréquents, et attachons une attention particulière à leur évolution au cours du temps. En outre, nous proposons d'examiner quelles sont les tendances existantes dans le cadre de ces évolutions.

L'article est organisé de la manière suivante. Dans la section 2, après avoir effectué un rappel des définitions, nous précisons les problématiques abordées. Nous décrivons l'algorithme d'extraction dans la section 3. De manière à prendre en compte la maintenance des connaissances extraites, nous présentons un algorithme basé sur la bordure négative dans la section 4. La section 5 s'intéresse à l'analyse des tendances dans les connaissances extraites. Dans la section 6, nous décrivons le système AUSMS (Automatic Update Schema Mining System) ainsi que les outils de visualisation associés. La section 7 décrit un certain nombre d'expériences menées avec le système AUSMS. Dans la section 8, nous présentons les travaux existants dans les domaines abordés. Enfin nous concluons dans la section 9.

2. Définitions et Problématiques

Comme les modèles de bases de données semi structurées tels que XML [W3C03] et le modèle OEM [AbBu00], nous adoptons la classe d'arbres ordonnés et étiquetés suivante. Un *arbre* est un graphe connecté acyclique et une *forêt* est un graphe acyclique. Une forêt est donc une collection d'arbres où chaque arbre est un composant connecté de la forêt. Un *arbre enraciné* est un arbre dans lequel un nœud est différent des autres et est appelé la racine. Un *arbre étiqueté* est un arbre où chaque nœud est associé avec une étiquette. Un *arbre ordonné* est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés, i.e. si un nœud a k fils, nous pouvons les désigner comme premier fils, second fils et ainsi de suite jusqu'au $k^{\text{ième}}$ fils. Etant données les types de jeux de données qui sont les plus classiques dans un contexte de fouille de données, nous considérons deux types d'ordre. Les fils d'un nœud de l'arbre sont soit ordonnés par ordre lexicographique par rapport à leur étiquette (« les fils d'un nœud sont alors vus comme un ensemble de fils »), soit ordonnés en fonction de l'application (« les fils d'un nœud sont alors vus comme une liste de fils »). Nous considérons dans la suite que les arbres manipulés sont ordonnés, étiquetés et enracinés. En effet, ce type d'arbre s'applique à de nombreux domaines comme par exemple l'analyse des pages d'un site Web[1] ou l'analyse de fichiers logs de transactions. Dans la suite de cette section, nous définissons de manière plus formelle les concepts manipulés.

Ancètres et descendants Considérons un nœud x dans un arbre enraciné T de racine r. Chaque nœud sur le chemin unique de r à x est appelé un *ancètre* de x et est noté par $y \leq_l x$, où l est la longueur du chemin d'y à x. Si y est un ancètre de x alors x est un *descendant* de y. Chaque nœud est à la fois un ancètre et un descendant de lui-même. Si $y \leq_l x$, i.e. y est un ancètre immédiat, alors y est appelé le *parent* de x. Nous disons que deux nœuds x et y sont *frères* s'ils ont le même parent.

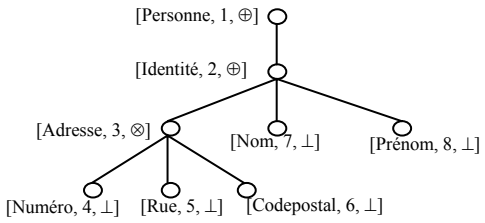

Figure 1. *Un exemple d'arbre manipulé*

[1]. En fait, dans le cas de données issues du Web, il existe de nombreux cycles dans les documents manipulés. Cependant, nous transformons ces graphes en graphes acycliques en utilisant l'approche présentée dans [WaLi99].

Nœuds et étiquettes Considérons T un arbre tel que $T=(N,B)$ où N représente l'ensemble des nœuds étiquetés et B l'ensemble des branches. La taille de T, notée $|T|$, correspond au nombre de nœuds dans T. Chaque nœud possède un numéro défini qui correspond à sa position lors d'un parcours en profondeur d'abord (parcours en pré-ordre) de l'arbre. Nous utilisons la notation n_i pour référencer le $i^{\text{ième}}$ nœud en fonction de la numérotation ($i=0...|T|-1$). En outre, il existe une fonction $I : N \rightarrow \{\otimes,\oplus,\perp\}$ qui affecte à chaque nœud $x \in N$ un indicateur sur l'ordre des fils. Si les fils sont ordonnés par l'application $I(x)= \otimes$; si les fils sont uniquement ordonnés par ordre lexicographique $I(x)=\oplus$ et si x ne contient pas de fils alors $I(x) = \perp$. Les étiquettes de chaque nœud sont issues de l'ensemble fini d'étiquettes $L=\{l, l_0, l_1, ..\}$, i.e. l'étiquette du nœud de nombre i est donnée par une fonction $l : N \rightarrow L$ qui fait correspondre à n_i l'étiquette associée $l(n_i) = y \in L$. Chaque nœud dans T est donc identifié par son numéro et son étiquette et possède un indicateur sur l'ordre de ses fils. Chaque branche, $b=(n_x,n_y) \in B$ est une paire ordonnée de nœuds où n_x est le parent de n_y.

Exemple : Considérons l'arbre représenté par la Figure 1. La racine de l'arbre possède l'étiquette Personne, il s'agit du premier nœud parcouru lors d'un parcours en profondeur d'abord (valeur 1), les fils du nœud (Identité) ne sont pas ordonnés donc l'indicateur est $\oplus$*. Considérons le nœud Adresse. Etant donné que ses fils sont ordonnés par l'application, i.e. ils ne sont donc plus par ordre lexicographique, nous avons comme indicateur d'ordre* $\otimes$*. Enfin, considérons le dernier nœud de l'arbre en bas à gauche. Son étiquette est Numéro, son numéro lors du parcours est 4 et comme il s'agit d'une feuille de l'arbre, son indicateur d'ordre est* $\perp$*. Le parent de ce nœud est Adresse et ses frères sont Rue et Code Postal.*

Sous arbres imbriqués Considérons $T = (N, B)$ un arbre étiqueté, ordonné et enraciné et considérons $S= (N_s, Bs)$ un autre arbre étiqueté, ordonné et enraciné. Nous disons que S est un *sous arbre imbriqué* de T, noté $S \leq T$, si et seulement si : a) $N_s \subseteq N$ et b) $B = (n_x, n_y) \in B_s$ si et seulement si $n_y \leq_l n_x$, i.e. n_x est un parent de n_y dans T. En d'autres termes, outre le fait que tous les sommets de S doivent être inclus dans les sommets de T, une branche apparaît dans S si et seulement si les deux sommets sont successifs et sur le même chemin de la racine à une feuille dans T. Si $S \leq T$, nous disons également que *T contient S*. Un (sous) arbre de taille k est aussi appelé un k-(sous)arbre.

Cette notion d'imbrication est généralisable à un ensemble de sous arbres contenus dans une base de données. Ainsi, considérons DB une base de données d'arbres, i.e. de forêts, et soit le sous arbre $S \leq T$ pour chaque $T \in DB$. Soit $\{t_1, t_2, ..., t_n\}$ les nœuds dans T avec $|T|=n$ et soit $\{s_1, s2, ..., s_m\}$ les nœuds dans S avec $|S|=m$. S possède une *étiquette de correspondance* $\{t_{i1}, t_{i2}, ..., t_{im}\}$ si et seulement si : a) $l(s_k) = l(t_{ik})$ pour $k = 1, ..., m$ et b) la branche $b(s_j,s_k)$ est dans S si et seulement si t_{ij} est un parent de t_{ik} dans T.

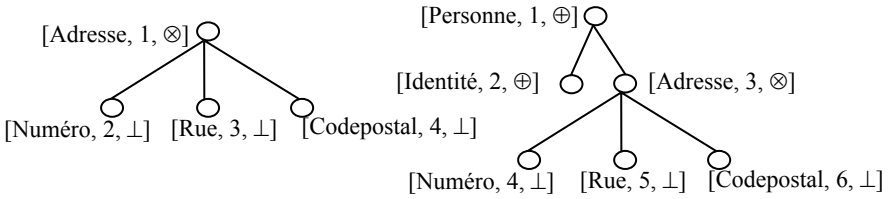

Figure 2. *Deux exemples de sous arbres non imbriqués*

Les conditions précédentes précisent que toutes les étiquettes des nœuds dans *S* ont une correspondance dans *T* et que la topologie des nœuds de correspondance dans *T* est la même que dans *S*. En outre, une étiquette de correspondance est unique pour chaque occurrence de *S* dans *T*.

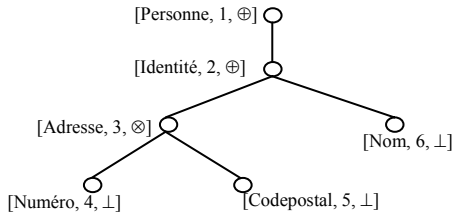

Figure 3. *Un exemple de sous arbre imbriqué*

Exemple : Considérons les deux arbres S_1 et S_2 de la Figure 2. Ces deux sous-arbres ne sont pas imbriqués dans l'arbre T de la Figure 1 car même si tous les nœuds de S_1 et S_2 sont présents dans T, i.e. $N_{s1} \subseteq N_T$ et $N_{s2} \subseteq N_T$, les branches des arbres S_1 et S_2 ne respectent pas la topologie des nœuds de correspondance dans T. En d'autres termes, pour l'arbre de gauche, nous pouvons constater que le nœud Adresse est de niveau 1 et ne possède pas de père Identité. Dans le cas de l'arbre de droite, nous pouvons constater que les nœuds ne sont pas à la même hauteur dans l'arbre, i.e. le lien de Parenté avec Identité n'est pas respecté. Considérons l'arbre S_3 représenté dans la Figure 3. Cet arbre est un sous arbre imbriqué de l'arbre T. En effet, tous les nœuds de S_3 sont présents dans T et la topologie des nœuds est respectée.

Extraction

Nous notons $\delta_T(S)$ le nombre d'occurrences du sous arbre *S* dans un arbre *T*. A partir de celui-ci, nous considérons que $d_T(S) = 1$ si $\delta_T(S) > 0$ et $d_T(S) = 0$ si $\delta_T(S) = 0$. Le *support* d'un sous arbre *S* dans la base de donnée *DB* est défini par $support(S) = \Sigma_{T \in DB} d_T(S)$, i.e. le nombre d'arbres dans *DB* qui contiennent au moins une occurrence de *S*. Un sous arbre *S* est dit *fréquent* si son support est supérieur ou égal à une valeur de support minimale (*minSupp*) spécifiée par l'utilisateur. Nous notons L^k, l'ensemble de tous les sous arbres fréquents de taille *k*.

Etant donnée une base de donnnées *DB* d'arbres, la recherche de sous arbres fréquents consiste à découvir l'ensemble L^{DB} de tous les sous arbres fréquents de cette base, i.e. les sous arbres dont le support est supérieur à *minSupp*.

Par la suite, de manière à faciliter l'expression des différents arbres, nous utilisons la notion de représentation d'arbre suivante. Considérons $T = (N, B)$ un arbre. Une *représentation d'arbre* est telle que : a) Si l'indicateur d'un nœud $x \in N$ vaut $\perp$ alors la représentation d'arbre vaut $x : \perp$; b) Soit un nœud $x \in N$ dont l'indicateur sur l'ordre des fils vaut $\oplus$ et soit $\{y_1, y_2, .., y_n\} \in N$ les fils du nœud x, alors la représentation d'arbre vaut $x : \{y_1, y_2, ..., y_n\}$; c) Soit un nœud $x \in N$ dont l'indicateur sur l'ordre des fils vaut $\otimes$ et soit $\{y_1, y_2, .., y_n\} \in N$ les fils du nœud x, alors la représentation d'arbre vaut $x : [y_1, y_2, ..., y_n]$.

Exemple : Considérons l'arbre T de la Figure 1. Sa représentation d'arbre associée est la suivante : Personne : {Identité : {Adresse : < Numéro : ⊥, Rue : ⊥, Code Postal : ⊥>, Nom : ⊥, Prénom : ⊥}.

Arbre_id	Arbre
T_1	Personne : {identite : {adresse : ⊥ , nom : ⊥}}
T_2	Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, compagnie : ⊥, directeur : <prenom : ⊥, nom : ⊥>, nom : ⊥}}
T_3	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>}, id : ⊥}
T_4	Personne : {identite : {adresse : ⊥, compagnie : ⊥, nom : ⊥}}
T_5	Personne : {identité : { adresse : ⊥, nom : ⊥ }}
T_6	Personne : {identite : { adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥>, directeur : <nom : ⊥, prenom : ⊥>, nom : ⊥}}

Figure 4. *Une base de données exemple*

Exemple : Considérons la base DB de la Figure 4. Supposons que la valeur du support minimal spécifiée par l'utilisateur soit de 50%, c'est-à-dire que pour être fréquent un sous arbre doit apparaître dans au moins trois arbres. Les seuls sous arbres fréquents dans DB sont les suivants : Personne : {identite : {adresse : ⊥ , nom : ⊥ }} et Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥ >}}. Le premier est contenu dans T_1 mais également dans T_4 et T_5. Le second est contenu dans T_2, T_3 et T_6. Par contre, Personne : {identite :{ adresse :<numero : ⊥, rue : ⊥, codepostal : ⊥ >, directeur :<nom : ⊥, prenom : ⊥ >, nom : ⊥}} est contenu dans T_2 et T_6 mais n'est pas fréquent puisque le nombre d'occurrences de ce sous arbre est inférieur à la contrainte de support minimal.

Maintenance

Considérons à présent l'évolution des sources de données. Soit *db* la base de données incrément où de nouvelles informations sont ajoutées ou supprimées. Soit $U=DB \cup db$, la base de données mise à jour contenant tous les arbres des bases *DB* et *db*. Soit L^{DB}, l'ensemble de tous les sous arbres fréquents dans *DB*. Le problème

de la maintenance des connaissances consiste à déterminer les sous arbres fréquents dans U, noté L^U, en tenant compte le plus possible des connaissances extraites précédemment de manière à éviter de relancer des algorithmes d'extraction sur la nouvelle base qui intègre les données mises à jour.

Exemple : De manière à illustrer un problème associé à la maintenance des connaissances lors de l'ajout de nouvelle données dans la base, considérons l'ajout de la base db composée d'une seule transaction T_7 = {Personne : {identite : {adresse : ⊥ , nom : ⊥ }}} *à la base DB de la Figure 4. Avec une même valeur de support minimal de 50%, un sous arbre, pour être fréquent, doit maintenant apparaître dans 4 transactions. Ainsi le sous arbre Personne : {identite : {adresse : ⊥ , nom : ⊥ }} reste fréquent car il apparaît dans* T_1, T_4, T_5 *et* T_7 *alors que Personne : {identite : {adresse : <numero : ⊥, rue : ⊥, codepostal : ⊥ >}} n'est plus fréquent car supporté uniquement par* T_2, T_3 *et* T_6.

Analyse de tendances

Le problème de l'analyse des tendances est complémentaire de celui de la maintenance des connaissances extraites. Dans le cas de l'analyse de tendances, nous devons maintenir, lors de chaque extraction de connaissances, la liste de tous les sous arbres fréquents pour voir comment ceux-ci évoluent au cours du temps. La problématique de l'analyse des tendances consiste donc à analyser au cours du temps les comportements des résultats (croissant, décroissant, cyclique...).

Exemple : Considérons la base DB de la Figure 4. Nous avons vu dans l'exemple précédent que le sous arbre Personne : {identite : {adresse : ⊥ , nom : ⊥ }} était fréquent pour un support de 50%. L'analyse de tendances permet de savoir comment cette structure évolue au cours du temps, i.e. par exemple si ce sous arbre a tendance à apparaître de plus en plus dans la base de données ou si, au contraire, à l'issue des mises à jour des données sources, ce sous arbre a plutôt tendance à disparaître.

3. Extraction

Pour rechercher les sous arbres fréquents, nous utilisons un algorithme, *FindSubStructure*, inspiré des algorithmes par niveau [AgSr95] et dont les principes généraux sont expliqués ci dessous :

Algorithm FindSubStructure

Input : un support minimal (minSupp), une forêt d'arbres
Output : l'ensemble L des structures fréquentes qui vérifient la contrainte de support minimal et un graphe BN constituant la bordure négative

---

1 : DB=*TreeToSequence* (G)
2 : *k = 1 ;*

3 : C_1 = {{i}/ i ∈ *ensemble d'éléments atomiques issus de la phase 1*};
4 : **While** ($C_k \neq \varnothing$) **do**
5 : **Foreach** $d \in DB$ **do** *VerifyCandidate* (d, C_k); **enddo**
6 : L_k = {$c \in C_k$/*support* (c) ≥ *minSupp*};
7 : *GenerateBN* (*BN*, C_k, L_k);
8 : k += 1;
9 : *CandidateGeneration* (C_k) ;
10 : **enddo**
10 : **Return** L^{DB} *où* L^{DB} *est l'union de j=0 à k des* L_j

De manière à résoudre efficacement la problématique de la recherche de sous arbres fréquents, il est indispensable de trouver une représentation de ceux-ci. Pour décrire un graphe G, un grand nombre de représentations peut être utilisé [GoMi90]. Etant donné les arbres manipulés, l'approche retenue est basée sur une représentation à l'aide de séquences définies de la manière suivante : Soit $N = \{n_1, n_2, \ldots n_n\}$ l'ensemble des nœuds d'un graphe $G = (N,B)$. Soit $I(n_i)$ et $prof(n_i)$ respectivement l'indicateur d'ordre et la position du nœud n_i dans le graphe G. Un élément est un ensemble de nœuds non vide noté $e = (\{I(n_1), n_1, prof(n_1)\} \{I(n_2), n_2, prof(n_2)\}, \ldots \{I(i_k), i_k, prof(i_k)\})$. Une séquence est une liste ordonnée, non vide, d'éléments notée $<e_1\ e_2\ \ldots\ e_n>$ où e_i est un élément.

REMARQUE : Par souci de lisibilité, nous utilisons par la suite indifféremment la notation condensée $(I(n_i)n_{iprof(ni)})$ au lieu de $\{I(n_i), n_i, prof(n_i)\}$. Ainsi, l'élément $(\{\oplus, Personne, 1\})$ s'écrira en notation condensée $(\oplus Personne_1)$.

Par manque de place, nous ne détaillons pas l'algorithme *TreeToSequence* mais nous en donnons les principes généraux. De manière à conserver les niveaux d'imbrications des arbres manipulés, la transformation d'un arbre en une séquence est opérée de la manière suivante : si deux éléments sont à un niveau terminal et si le premier est frère du second, nous les regroupons dans un même élément autrement ils sont inclus dans deux éléments séparés. La notion d'ordre de la valeur « liste-de » est prise en compte en créant de nouveaux éléments. La séquence composite qui résulte de l'union de ces éléments peut alors être perçue comme une navigation en « profondeur d'abord » dans l'arbre.

Exemple : Considérons l'arbre de la Figure 1. Sa transformation avec l'algorithme TreeToSequence est la suivante : $<(\{\oplus, Personne, 1\})\ (\{\oplus, Identité, 2\})\ (\{\otimes, Adresse, 3\})\ (\{\perp, Numéro, 4\})\ (\{\perp, Rue, 4\})\ (\{\perp, CodePostal, 4\})\ (\{\perp, Nom, 3\})\ (\{\perp, Prénom, 3\})>$. *Soit, en utilisant la notation condensée,* $<(\oplus Personne_1)\ (\oplus Identité_2)\ (\otimes Adresse_3)\ (\perp Numéro_4)\ (\perp Rue_4)\ (\perp CodePostal_4)\ (\perp Nom_3)\ (\perp Prénom_3)>$.

A l'issue de *TreeToSequence*, nous disposons d'une base de données d'arbres et pour chaque arbre extrait, un identifiant est associé qui servira de clé primaire. Ensuite l'algorithme effectue un parcours de *DB* pour déterminer quels sont les éléments qui interviennent assez régulièrement pour être retenus. A partir des éléments de taille 1 qui vérifient le support, nous générons des sous arbres de taille 2

qui sont appelés des candidats (C_2). Un nouveau parcours sur la base permet de retenir tous les éléments candidats de taille 2 dont le nombre d'occurrences est supérieur au support minimal. Ensuite, l'algorithme continue de la manière suivante : à chaque étape *k*, la base est parcourue pour compter le support des candidats de C_k (algorithme *VerifyCandidate*). Tous les candidats dont le nombre d'occurrences est supérieur au support minimal deviennent fréquents et sont ajoutés à : L^k. A partir de cet ensemble, de nouveaux candidats, de taille $k+1$, peuvent être construits (algorithme *CandidateGeneration*). Cet ensemble constituera l'ensemble des candidats de l'étape suivante. L'algorithme s'arrête quand la procédure de génération des candidats fournit un ensemble vide ou quand la procédure *VerifyCandidate* retourne un ensemble vide de fréquents.

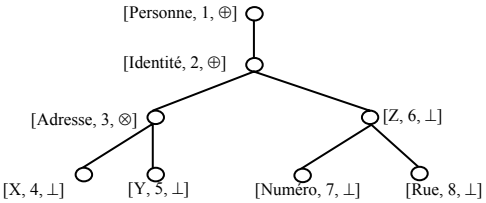

Figure 5. *Un arbre problématique*

Dans le but d'améliorer la procédure de génération des candidats ainsi que la gestion des éléments candidats, nous utilisons une représentation par bitmap inspirée de [ArGe02]. Un bitmap vertical est créé pour chaque élément de la base de données et chaque bitmap possède un bit correspondant à chaque arbre de la base. Si un élément i d'une séquence apparaît à la position j, alors le bit correspondant pour la position j du bitmap de l'élément i est positionné à 1 sinon le bit prend pour valeur 0. De manière à optimiser l'espace de stockage des bitmaps, des optimisations décrites dans [ArGe02] sont intégrées. Outre le fait que cette structure est particulièrement bien adaptée à notre problématique, elle offre également l'avantage de diminuer considérablement l'espace de stockage et est particulièrement bien adaptée à la recherche de structures longues. Cependant, la seule représentation par bitmaps ne permet pas de prendre en compte le niveau d'imbrication des sous arbres et il est important de conserver l'ordre d'apparition des éléments dans les séquences. L'exemple ci-dessous illustre ce problème.

Exemple : Considérons l'arbre représenté par la Figure 5. Sa représentation sous la forme de séquence est la suivante $S_1 = <(\oplus Personne_1)\ (\oplus Identité_2)\ (\otimes Adresse_3)\ (\perp X_4)\ (\perp Y_4)\ (\otimes Z_3)\ (\perp Numéro_4)\ (\perp Rue_4)>$. *Nous avons vu précédemment que la représentation pour l'arbre de la Figure 1 était :* $S_2 = <(\oplus Personne_1)\ (\oplus Identité_2)\ (\otimes Adresse_3)\ (\perp Numéro_4)\ (\perp Rue_4)\ (\perp CodePostal_4)\ (\perp Nom_3)\ (\perp Prénom_3)>$. *En associant directement un vecteur de bits par élément nous voyons que les éléments* $(\perp Numéro_4)$ *et* $(\perp Rue_4)$ *sont considérés de la même manière. Cependant, dans le cas des deux séquences, ces deux éléments ne sont pas comparables car leur père est*

différent. Dans le cas de la séquence S_1 le père est ($\otimes Z_3$) alors que pour la séquence S_2, le père est ($\otimes Adresse_3$).

Pour éviter le problème décrit précédemment, la construction de la liste de bitmaps est réalisée de la manière suivante. Lors du parcours d'une séquence de la base, si l'élément ajouté existe déjà dans la liste des bitmaps, nous vérifions si le père de cet élément est le même que celui de la liste. La liste étant créée au fur et à mesure, le père de l'élément de la liste est le prédécesseur dans la liste. Dans le cas où les éléments manipulés sont les mêmes, i.e. il s'agit de nœud de l'arbre possédant un même père, nous mettons à jour le bitmap de la liste. Autrement, nous créons un nouvel élément qui sera ajouté à la liste. Par souci d'efficacité, une table de hachage est associée à ces éléments.

ID	**Num**	$\oplus Personne_1$	$\oplus Identite_2$	$\otimes Adresse_3$	$\perp Numéro_4$	$\otimes Z_3$	$\perp Numéro_4$
1	1	1	0	0	0	0	0
1	2	0	1	0	0	0	0
1	3	0	0	1	0	0	0
-	-	0	0	0	1	0	0
2	1	1	0	0	0	0	0
2	2	0	1	0	0	0	0
-	-	0	0	1	0	1	0
-	-	0	0	0	0	0	1
3	1	1	0	0	0	0	0
3	2	0	1	0	0	0	0
-	-	0	0	1	0	0	0
-	-	0	0	0	1	0	0
…							

Figure 6. Représentation sous forme de bitmaps

Exemple : Considérons la Figure 6 qui représente une partie des différents bitmaps associés aux éléments de la base de données. La colonne ID correspond au numéro d'arbre dans la base et la colonne Num correspond à la position de l'élément dans la séquence. Si nous considérons la colonne $\oplus Personne_1$, nous voyons qu'apparaissant comme premier élément de la séquence, le premier bit est positionné à 1, les autres étant positionnés à 0. Considérons à présent la colonne correspondant au premier élément $\perp Numéro_4$. Comme il apparaît en quatrième position dans la séquence, le quatrième bit est positionné à 1. Nous pouvons constater qu'il existe une autre colonne $\perp Numéro_4$. Celle-ci représente le fait qu'il existe un autre sommet de même nom mais dont le père est différent. Dans ce cas particulier, il s'agit de $\otimes Z_3$ de l'exemple précédent.

Examinons à présent comment sont gérés les candidats. Les candidats de taille 1 étant générés comme expliqué précédemment, nous nous intéressons aux candidats de taille supérieure à 1.

Considérons dans un premier temps l'extension d'un élément à la racine de l'arbre. Soit *x* et *y* deux éléments fréquents de longueur 1 et *x* représente le nœud racine de l'arbre. Soit *B(x)* et *B(y)* les vecteurs correspondants aux éléments. Si le

nœud *y* est d'une profondeur supérieure au nœud *x*, i.e. *prof(y)=prof(x)+1* alors nous procédons à une extension de la séquence, notée *S-extension*, i.e. nous ajoutons un nouvel élément *y* comme fils du père *x*. Pour cela, un nouveau vecteur de bits, *B'(x)*, est généré à partir de *B(x)* tel que les bits situés avant la deuxième position soient positionnés à 0 et les autres à 1. La *S-extension* est alors réalisée via l'opérateur logique AND entre ce vecteur de bits transformé *B' (x)* et *B (y)*. Autrement, si *prof(x) = prof(y)* alors deux cas sont à considérer :

- Il s'agit de nœuds terminaux de l'arbre, i.e. leur identificateur d'ordre vaut $\perp$, il est alors nécessaire de réaliser une *S-extension* et une *I-extension*. La *I-extension*, l'extension d'un élément, est simplement réalisée à l'aide de l'opérateur logique AND entre *B(x)* et *B(y)*.
- Il ne s'agit pas de nœuds terminaux de l'arbre, i.e. leur identificateur d'ordre ne vaut pas $\perp$, et nous réalisons une *S-extension.*

$\oplus Personne_1$		$T(\oplus Personne_1)$		$\oplus Identite_2$		$\oplus Personne_1, \oplus Identite_2$
1		0		0		0
0		1		1		1
0		1		0		0
0		1		0		0
1	S-step	0	&	0	Résultat	0
0	→	1		1	→	1
0		1		0		0
0		1		0		0
1		0		0		0
0		1		1		1
0		1		0		0
0		1		0		0

Figure 7. *Un exemple de S-extension de $\oplus Personne_1$ par $\oplus Identite_2$*

Exemple : $\oplus Identite_2$ étant d'une profondeur supérieure à $\oplus Personne_1$, nous pouvons appliquer une S-extension représentée dans la Figure 7.

Les principes mis en avant précédemment sont, bien entendu, généralisables en considérant que *x* ne représente non plus un élément mais un sous arbre fréquent.

Algorithm CandidateGeneration

Input : Un ensemble *B* de vecteurs de bits représentant la base de données et triés par ordre, une profondeur *k* (*k >1*).

Output : *B* étendu d'un élément.

1 : **Foreach** $n \in B_1$ **do** // B_1 représente les éléments fréquents
2 : **Foreach** $n_{add} \in B_1$ **do**
3 : **If** $prof(n_{add})=prof(n)+1$ **then** $B+=(T(B(n))$ **AND** $B(n_{add})$; **endif**
4 : **else If** $prof(n_{add}) = prof(n)$ **then**
5 : **If** $id(n_{add}) = id(n) = \perp$ **then** $B+=(B(n)$ **AND** $B(n_{add})$; **endif**
6 : $B+=(T(B(n))$ **AND** $B(n_{add})$;

7 : **endif**
8 : **enddo**
9 : **enddo**

L'algorithme *VerifyCandidate* consiste juste à compter, dans les nouveaux vecteurs de bits créés lors de la phase de génération de candidats, le nombre de bits positionnés à 1 et à vérifier si ce nombre est supérieur au support minimal.

Lors de la recherche de candidats nous générons également la bordure négative [MaTo96] (algorithme *GenerateBN*) qui est composée de tous les sous arbres qui ne sont pas fréquents à l'issue de la phase de vérification sur la base de données. Cette bordure sera utilisée dans la phase suivante de prise en compte de l'évolution des données sources.

Nous stockons dans un arbre noté, T_{F+BN}, l'ensemble des arbres fréquents et les éléments de la bordure négative avec les valeurs de supports associées. Cet arbre admet deux types d'arcs : *S-arc* et *I-arc* correspondant respectivement aux S-extensions possibles à partir d'un nœud et aux I-extensions. Chaque nœud de l'arbre représente donc un sous arbre fréquent ou membre de la bordure négative. Soit x et y deux nœuds de l'arbre, il existe un *S-arc* entre x et y si le nombre d'éléments constituant le nœud x est égal au nombre d'éléments constituant la structure y moins un et que y soit une *S-extension* de x. Un *I-arc* existe si la condition sur le nombre d'éléments est respectée et que y soit une *I-extension* de x. Cette structure est utilisée pour assurer la maintenance des structures extraites de la base de données comme nous allons le voir dans la section suivante.

4. Maintenance

La bordure négative obtenue lors de l'étape précédente nous permet de tenir compte des mises à jour et de maintenir la connaissance extraite. En effet, pour éviter de ré-appliquer l'algorithme précédent lors de chaque mise à jour, nous stockons dans cette bordure l'information minimale pour déterminer rapidement les sous arbres fréquents. La prise en compte de l'évolution des données sources se divise en deux étapes : la détection et le stockage des évolutions des données puis les répercussions de ces évolutions sur les connaissances préalablement extraites. Dans un premier temps, nous montrons comment les évolutions sont prises en compte via l'algorithme *DiffSource* et nous présentons ensuite comment les utiliser pour maintenir la connaissance via l'algorithme *Update*.

Algorithm DiffSource

Input : S l'ensemble des sources de données, Δ_S l'historique des modifications.
Output : Δ_S' l'historique des modifications mis à jour.

1 : **Foreach** $s \in S$ **do**

2 : **If** $s_{new} \neq s_{old}$ **then**
3 : *UpdateDeltaRelation* (Δ_s, *opmaj, t);*
4 : **EndIf**
5 : **End do**

L'algorithme *DiffSource* fonctionne de la manière suivante. A partir d'un délai spécifié par l'utilisateur, les différentes sources de données de *S* sont comparées (s_{old} représente les données sources initiales, i.e. lors de la dernière analyse et s_{new} les données en cours d'analyse). Cette opération est effectuée par un agent qui agit soit de manière temporelle (durée écoulée depuis le dernier déclenchement), soit de manière directe (choix de l'utilisateur). Durant cette comparaison il n'est pas possible que cet agent soit relancé, i.e. pour faire une nouvelle comparaison ou appliquer les évolutions qu'il a détecté il faut attendre que celui-ci ait terminé son travail. L'agent est chargé de comparer les données sources et de propager les modifications. Ainsi, si la source de données a été modifiée, les mises à jour sont stockées dans l'ensemble Δs qui gère l'historique des modifications (algorithme *UpdateDeltaRelation*) pour chaque source. Par manque de place, nous ne détaillons pas cet algorithme mais en donnons les principes généraux. Inspirée des deltas relations des règles actives [ChAb98], l'idée générale est de ne refléter que les effets de bord des modifications de la structure, i.e. de minimiser les opérations de mises à jour. Par exemple, si un élément est ajouté et ensuite supprimé, nous n'avons pas à tenir compte des modifications et donc pas à les répercuter. Pour cela, nous transformons les séquences extraites des sources, les comparons avec celles stockées dans la base et ajoutons dans Δs les modifications.

Algorithm Update

Input : T_{BN+F} l'arbre contenant la bordure négative *BN* et les fréquents *F*, un support minimal *minSupp* spécifié par l'utilisateur, *S* l'ensemble des sources de données, Δ_S l'historique des modifications.
Output : T_{BN+F} ' l'arbre mise à jour.

1 : T_{BN+F} ' = T_{BN+F};
2 : *a_traiter = Ø;*
3 : **Foreach** *noeud e de T'* **do**
4 : *etat = etat(e) ;* // 0 si e est un élément de F, 1 si e est un élément de BN
5 : *Mettre à jour la valeur du support de e (supp(e)) à partir des données de* Δ_S*;*
6 : **If** *etat = 0* **then**
7 : **If** *supp(e) < minSupp* **then** *a_traiter = a_traiter + {e};* **endif**
8 : **Else If** *etat = 1* **then**
9 : **If** *supp(e) > minSupp* **then** *a_traiter = a_traiter + {e};* **endif**
10 : **enddo**
11 : **Foreach** *élément e* ∈ *a_traiter* **do**
12 : *Détruire tous les successeurs de e dans T';*

13 : *ConstruitExtension(e);*
14 : **enddo**

L'algorithme *Update* se décompose en trois étapes principales : récupération de l'information (lignes 1-2), mise à jour des supports et calcul de l'ensemble des éléments à traiter (lignes 3-10) et enfin, mise à jour via la fonction *ConstruitExtension* des éléments de l'ensemble précédent (11-14).

Dans la première étape, nous récupérons dans T_{BN+F}', la structure obtenue précédemment, i.e. T_{BN+F}. L'ensemble des éléments à considérer *a_traiter* est initialisé à Ø, cet ensemble représente les nœuds de T_{BN+F} qui changent d'état à l'issue de la prise en compte des évolutions contenues dans Δ_S et/ou du fait d'une nouvelle valeur de support *minSupp* différente de celle utilisée lors du calcul de T_{BN+F}. Cet ensemble contiendra les éléments de T_{BN+F} fréquents qui, après prise en compte des modifications (ou variation du support), deviendront non fréquents et donc membre de la bordure négative ainsi que les éléments de T_{BN+F} , membres de la bordure négative, qui deviendront fréquents

La seconde phase consiste à prendre en compte les modifications contenues dans l'ensemble Δ_S. Ces modifications ont pour effet de modifier le support des éléments de T_{BN+F} ′ (ligne 6). Un parcours en largeur sur T_{BN+F}' permet de mettre à jour ces supports. Ensuite, pour chaque élément de T_{BN+F}' nous vérifions si celui-ci a changé d'état par rapport à son état initial (stocké dans une variable (ligne 5)). Les deux états possibles d'un nœud de T_{BN+F}' sont : fréquent ou membre de la bordure négative. Si un nœud ne change pas d'état, les informations le concernant n'ont aucune conséquence sur la recherche de structures fréquentes. Dans le cas contraire, il y a des conséquences sur la structure T_{BN+F} ′ dans la mesure où des éléments qui ne l'étaient pas peuvent devenir fréquents, de nouveaux sous arbres peuvent apparaître et des sous arbres qui étaient fréquents peuvent ne plus l'être.

La dernière étape de l'algorithme considère les nœuds qui ont changé d'état. Le but de cette étape consiste à effectuer sur la structure les modifications nécessaires pour prendre en compte ces changements en effectuant le minimum de calculs possibles. Pour cela, nous utilisons les propriétés de l'arbre T_{BN+F} ′ et de l'algorithme *ConstruitExtension*. L'algorithme *ConstruitExtension* est un algorithme récursif qui admet en paramètre un nœud de T_{BN+F}'. Sa fonction est simple il applique l'algorithme *CandidateGeneration* sur le nœud passé en paramètre pour connaître les candidats possibles à partir du fréquent représenté dans la structure du noeud *e*. Pour chacun de ces candidats c_i, il construit un arc (*S-arc* ou *I-arc* selon le cas) entre *e* et c_i. Pour chaque c_i, nous appelons la procédure *VerifyCandidate* qui détermine si c_i est un fréquent ou un membre de la bordure négative. Si c_i est un membre de la bordure négative alors il n'y a plus aucune conséquence sinon il faut rappeler *ConstruitExtension* sur ce candidat. Après application de *CandidateGeneration* sur un nœud quelconque, toutes les conséquences relatives à l'application des modifications pour ce nœud sont terminées. La preuve formelle n'est pas détaillée ici faute de place mais elle découle directement de la façon dont les candidats sont

étendus par *CandidateGeneration*. Nous allons donc nous intéresser successivement à tous les nœuds changeant d'état, i.e. ceux contenus dans *a_traiter*. Pour chacun de ces nœuds nous devons prendre en compte les modifications : c'est-à-dire appliquer sur ce nœud l'algorithme *ConstruitExtension*. Pour appliquer cet algorithme tous les descendants du nœud considéré doivent être détruits (ligne 12). Durant cette étape, il peut arriver que des descendants soit eux-mêmes des éléments de l'ensemble *a_traiter*. Dans ce cas, ces éléments seront retirés de cet ensemble car ils n'ont plus de raison d'être. Lorsque l'appel à *ConstruitExtension* se termine pour un nœud donné, la structure T_{BN+F}' est à jour en ce qui concerne le changement d'état de ce nœud. Il reste alors à réitérer cet appel sur tous les noeuds ayant subi un changement d'état (boucle pour 11-14). La structure T_{BN+F}' finale contient tous les sous arbres fréquents, i.e. L^{DB} est mis à jour, et tous les éléments de la bordure négative après avoir intégré les modifications liées à Δ_S (et/ou à une nouvelle valeur de support *minSupp*).

Dans le pire des cas, nous pouvons affirmer que la complexité de l'algorithme *Update* reviendrait à relancer l'algorithme *ConstruitExtension* sur le nœud racine. Ce cas correspond, en effet, à un calcul depuis zéro des sous arbres fréquents et des éléments de la bordure négative. L'approche proposée possède donc de meilleures performances en moyenne qu'un calcul depuis zéro et que, dans le pire des cas, elle nécessite un nombre de calcul égal à l'extraction des sous arbres à partir de zéro.

5. Analyse de Tendances

Comme nous l'avons précisé dans la section 2, l'analyse des tendances consiste à rechercher quelles sont les évolutions des sous arbres fréquents au cours du temps. L'un des problèmes associé à cette analyse est le stockage des données : comment trouver une structure efficace pour maintenir les connaissances extraites au fur et à mesure ? Etant donné le nombre de résultats intermédiaires, il est indispensable de trouver une structure de représentation adaptée. Le second problème lié aux tendances est de rechercher rapidement les mêmes structures afin de suivre leurs évolutions au cours du temps et en fonction des desiderata de l'utilisateur (tendance croissante, décroissante, cyclique, …).

Algorithm TrendAnalysis

Input : $(L^{DB}_{t1} + L^{DB}_{t2} + \ldots + L^{DB}_{tn}) = L^{DB}_{t}$, l'ensemble des structures fréquentes stockées à des dates t_1 ..... t_n avec leur support à ces dates.

Output : H_f, l'historique de chaque fréquent f sous la forme de couple (date, support).

1 : **Foreach** *different frequent f in* L^{DB}_t **do**
2 : $H_f = \varnothing$;
3 : **Foreach** *frequent* $e \in L^{DB}_t$ **do**

4 : **If** $f = e$ **then** $H_f = H_f +$ *(e.date, e.support);* **endif**
5 : **enddo**
6 : **Return** H_f;
7 : **enddo**

Le principe utilisé est actuellement le suivant. L'algorithme *FindSubStructure* décrit dans la section 3 est exécuté pour différentes valeurs de support. L'utilisateur peut spécifier la valeur du pas utilisé. Par exemple, l'utilisateur peut choisir de faire varier les valeurs de support d'un pas de 5% et dans ce cas l'analyse commence par 0.05, 0.10, 0.15, … Les résultats obtenus, i.e. les sous arbres fréquents, sont alors stockés à l'aide de bitmaps comme expliqué précédemment. En effet, nous avons pu constater aux cours des différentes expérimentations que cette structure était très efficace non seulement dans le cas des traitements mais également dans le cas de l'espace mémoire occupé. L'algorithme *TrendAnalysis* montre comment les historiques sont calculés à partir des sous arbres fréquents stockés à des dates différentes. Pour chaque sous arbre dont on désire connaître l'historique, nous cherchons aux différentes date t_1, t_2, ..., t_n sa valeur de support. L'historique correspond alors à un ensemble de couples (date, support). A l'aide de cet historique, nous pouvons sélectionner ceux dont l'évolution du support correspond à un certain type de tendance (croissance, décroissance, …) au cours du temps. L'utilisation des bitmaps permet de comparer rapidement les différents arbres fréquents à l'aide de l'opérateur binaire AND (ligne 4). Les tendances décrivent simplement les évolutions qui respectent le choix de l'utilisateur avec les informations complémentaires sur les périodes de temps pendant lesquelles la tendance est vérifiée.

6. Le système AUSMS

Les différents algorithmes précédents ont été intégrés dans le système AUSMS (Automatic Update Schema Mining System), le but étant de proposer un environnement de découverte et d'extraction de connaissances pour des données semi structurées, depuis la récupération des informations jusqu'à la mise à jour des connaissances extraites. Ces principes généraux illustrés par la figure 8 sont assez similaires à ceux d'un processus classique d'extraction de connaissances.

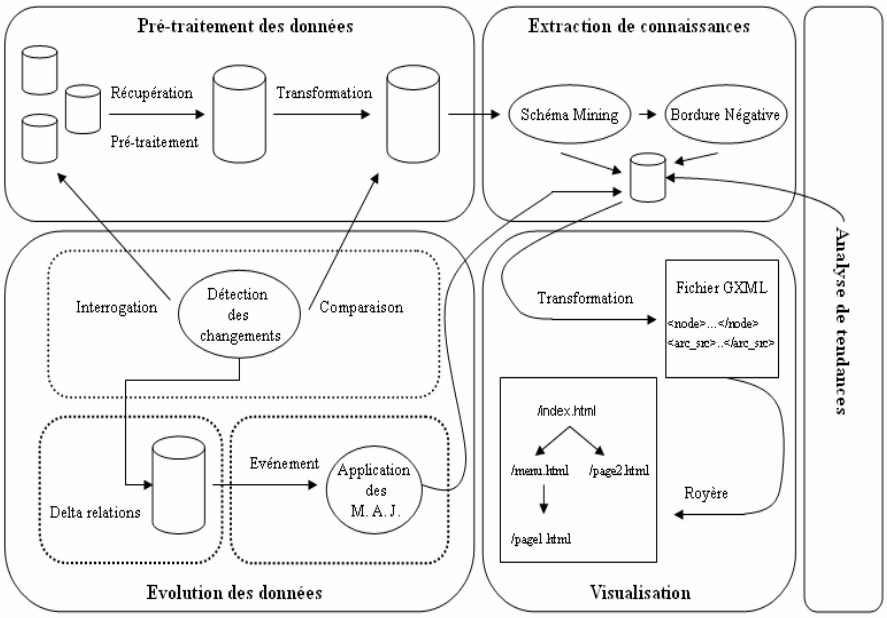

Figure 8. *Architecture générale*

La démarche se décompose en trois phases principales. Tout d'abord à partir de fichiers de données semi structurées brutes, un prétraitement est nécessaire pour éliminer les données inutiles. Par exemple, dans le cadre de données issues du Web, une étape de filtrage est réalisée de manière à éliminer les données qui ne sont pas utiles pour l'analyse : image, sons, vidéo, …. En outre, en fonction du point de vue de l'utilisateur, les sous structures « non intéressantes » sont également supprimées. Dans la seconde phase, un algorithme d'extraction de connaissances est utilisé pour extraire les sous arbres fréquents ce qui permet la maintenance des connaissances extraites et l'analyse de tendances. Les informations obtenues lors de la phase d'extraction sont maintenues dans une base de données. Enfin, l'exploitation par l'utilisateur des résultats obtenus est facilitée par un outil de visualisation des sous structures fréquentes, i.e. des sous arbres fréquents.

Alors que les modules précédents sont chargés de fournir et de maintenir des sous structures fréquentes, le module de visualisation permet d'afficher les structures extraites et offre un formalisme pour les décrire. Pour cela, nous utilisons, en premier lieu, GraphXML [HeMa00] qui est un langage de description de graphe en XML spécialement étudié pour des systèmes de visualisation et de dessin. GraphXML, en plus de fournir un langage de description, permet à l'utilisateur de rajouter de nombreuses informations aux graphes manipulés : date, couleur des arcs.

En second lieu, de manière à visualiser à la fois les structures extraites mais également leur apparition dans les données sources, nous utilisons « Graph Visualisation Framework » [MaHe00] qui propose un ensemble de classes java pour

visualiser et manipuler les graphes. Ce système, via une application nommée Royère, permet l'affichage des arbres décrits au format GraphXML.

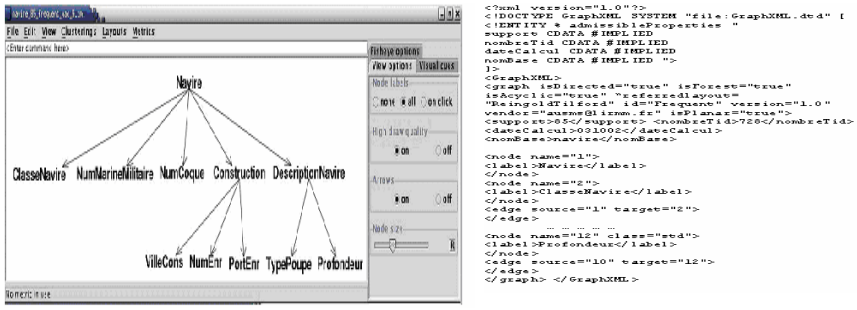

Figure 9. *Exemples de structures extraites*

Considérons la figure 9 qui représente des copies d'écran de structures visualisées. Nous trouvons à gauche un sous arbre fréquent issu de l'application de l'algorithme d'extraction sur une base de données de navires. A droite nous avons la description au format GraphXML de cette structure.

7. Expérimentations

De manière à valider notre approche, nous avons utilisé le système AUSMS sur de nombreux jeux de données. En ce qui concerne l'extraction, par manque de place, nous ne nous intéressons qu'à deux jeux de données différents et présentons principalement la méthodologie utilisée. Ensuite, nous présentons quelques simulations effectuées pour valider l'approche de maintenance à l'aide de la bordure négative. Enfin, nous illustrons des analyses de tendances.

7.1 Extraction

Lors de la première expérimentation, nous avons, de la même manière que [WaLi99], appliqué notre approche sur la base de données des films sur Internet (http://us.imdb.com) afin de rechercher des structures typiques de documents concernant le cinéma. Cette base de données regroupe les informations sur les films de 1892 à nos jours. Toutes les informations sont organisées sous la forme de pages HTML reliées entre elles par des liens hypertextes. Suite à l'examen de films, nous nous sommes intéressés à la partie de la base qui concerne les 250 meilleurs films afin d'en extraire les informations concernant les acteurs. Après avoir récupéré les données concernant les acteurs, nous avons extrait des pages HTML, les

informations significatives des pages concernant la structure des documents de type acteur. Cette étape a été réalisée en partie automatiquement à l'aide d'un parser mais également manuellement dans la mesure où le contenu des pages ne permettait pas d'extraire, sans ambiguïté, les structures sous jacentes. Au total, nous avons ainsi récupéré une base de 500 acteurs dont la profondeur maximale était de 5.

Sur la base de données ainsi obtenue, nous avons appliqué AUSMS. Le tableau ci-dessous illustre les résultats obtenus pour différentes valeurs de support. Par exemple, pour un support de 50%, nous avons trouvé le sous arbre fréquent suivant : *{actor :{name, dateofbirth, filmographyas :{title, notabletv}}}* indiquant que pour au moins 250 acteurs de la base, un acteur possède un nom, une date de naissance et une filmographie. Dans cette filmographie, il apparaît à la fois dans un film mais il a également effectué des apparitions à la télévision.

Support	Sous arbres fréquents
10%	* {actor:{name, birthname, dateofbirth, dateofdeath, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth, minibibliography, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth, sometimescreditas:{name}, filmographyas :{title, notabletv}}} * {actor:{name, birthname, dateofbirth ,trivia, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, filmographyas :{title, director, notabletv, producer}}}
20%	* {actor:{name, birthname, dateofbirth, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, dateofdeath, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, minibibliography, filmographyas :{title, notabletv}}} * {actor:{name, dateofbirth, sometimescreditas:{name},filmographyas :{title}}} * {actor:{name, dateofbirth, trivia, filmographyas :{title, notabletv}}} * {actor:{name, sometimescreditas:{name},filmographyas :{title, notabletv}}}
30%	* {actor:{name, birthname, dateofbirth, filmographyas :{title}}} * {actor:{name, dateofbirth, filmographyas:{title,notabletv}}} * {actor:{name, trivia, filmographyas:{title}}}
40%	* {actor:{name, dateofbirth, filmographyas:{title, notabletv}}}
50%	* {actor:{name ,dateofbirth, filmographyas:{title, notabletv}}}

D'autres expérimentations ont été réalisées pour mettre en relation la notion de structure et celle de parcours d'un utilisateur sur un site Web. En effet, nous pouvons raisonnablement considérer que le parcours d'un utilisateur sur un site Web est assimilable à un arbre dont la racine correspond à l'entrée dans le serveur. L'un des jeux de données utilisé est issu des fichiers log d'un site de e-commerce spécialisé dans la téléphonie. Le fichier avait une taille de 400 MO, contenait 12000 adresses IP différentes, concernait 900 pages visitées sur une période de deux jours

et en moyenne les arbres associés aux parcours avaient une profondeur de 5. Avec un support de 5%, nous avons trouvé le parcours fréquent suivant : <(/default.asp) (/Manage.asp) (/Paybox.asp) (/SecurePay.asp) (/SecurePayAtLeast.asp) (/RedirectSite.asp) (/PayboxDelivery.asp) (/Account.asp)> indiquant qu'au moins 600 personnes avec des IP différentes sont allées après la page d'accueil (default.asp) accéder à leur compte (Manage.asp) pour recharger leur crédit de temps (Paybox.asp, SecurePay.asp, SecurePayAtLeast.asp, RedirectSite.asp, PayboxDelivery.asp) et enfin verifier sur leur compte leur nouveau crédit temps (Account.asp).

7.*2 Maintenance*

De manière à expérimenter l'aspect incrémental de l'approche, nous décrivons dans cette section, des expériences simulées sur des jeux de données concernant les renseignements sur les navires qui ont été immatriculés au Canada ou qui ont navigué dans les eaux Canadiennes[2].

Le principe de la première expérimentation consiste, à partir d'une extraction précédente, à faire varier le support pour effectuer une mise à jour des connaissances. Dans ce cas précis, les sources ne sont pas modifiées, nous cherchons à tester l'aptitude de l'algorithme à utiliser les informations issues d'un calcul précédent (bordure négative) par comparaison à l'exécution de l'algorithme de recherche depuis zéro.

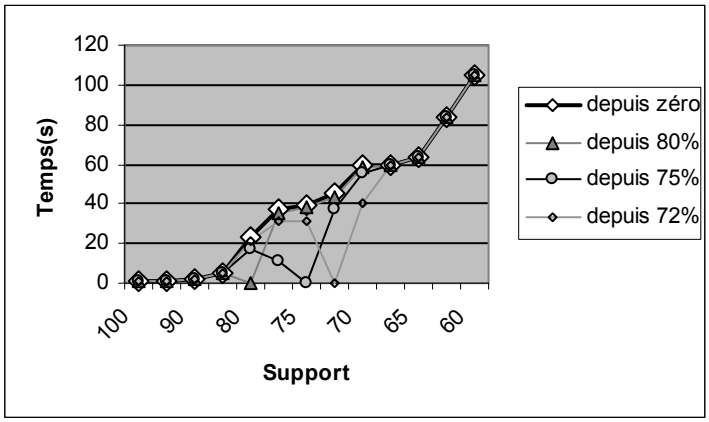

Figure 10. *Variation du support*

[2]. http://daryl.chin.gc.ca:8001/basisbwdocs/sid/title1f.html

Nous constatons dans la figure 10 que les courbes représentant les calculs de supports à partir d'un calcul existant (depuis 80%, depuis 75% et depuis 72%) sont toujours au dessous ou tangente à la courbe représentant un calcul depuis zéro. Ce qui signifie que, dans le pire des cas, l'algorithme de mise à jour est au moins aussi rapide que celui d'extraction depuis zéro. Nous constatons que pour les calculs effectués depuis 80%, les gains existent encore mais sont relativement faibles. En effet, pour une valeur de 80% l'information contenue dans la bordure négative n'est pas suffisante et nécessite de recalculer trop d'éléments, i.e. l'algorithme doit effectuer des calculs proches de ceux de l'algorithme de recherche depuis zéro ce qui explique une convergence des temps de calcul entre ces deux courbes. Il en est de même pour le calcul effectué depuis 72%. Cependant, si comme pour 80%, le nombre de point sur la courbe présentant des gains est similaire, le gain lui-même est plus important (i.e. la distance avec la courbe depuis zéro plus grande). Dans le cas de 75%, nous avons effectué une analyse plus fine (cf Figure 11).

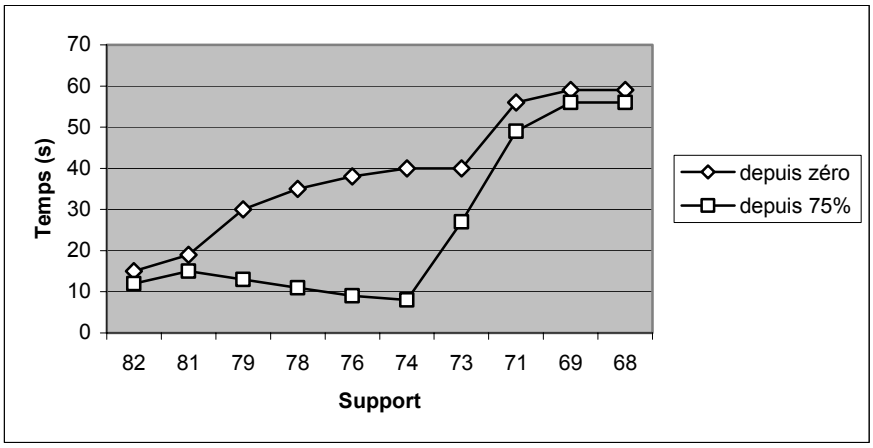

Figure 11 *Variation de support entre 68 et 82 %*

Dans cette figure, nous constatons que pour des supports proches d'un calcul initial, l'intérêt d'utiliser un tel algorithme est beaucoup plus conséquent car le gain en temps sur cette plage de support représente 65%. Nous poursuivons les expérimentations en ajoutant des transactions à notre base initiale.

L'ajout de transactions représente l'arrivée de nouvelles informations dans les sources de données utilisées. Afin de tester notre algorithme et étant donné que cette base de données est relativement statique nous avons coupé celle-ci en 5 bases de données contenant respectivement 50%, 70%, 80 %, 90% et 100% de la base initiale. Dans la figure 12, nous comparons les résultats pour différents supports entre un calcul depuis zéro et l'utilisation de l'algorithme de mise à jour.

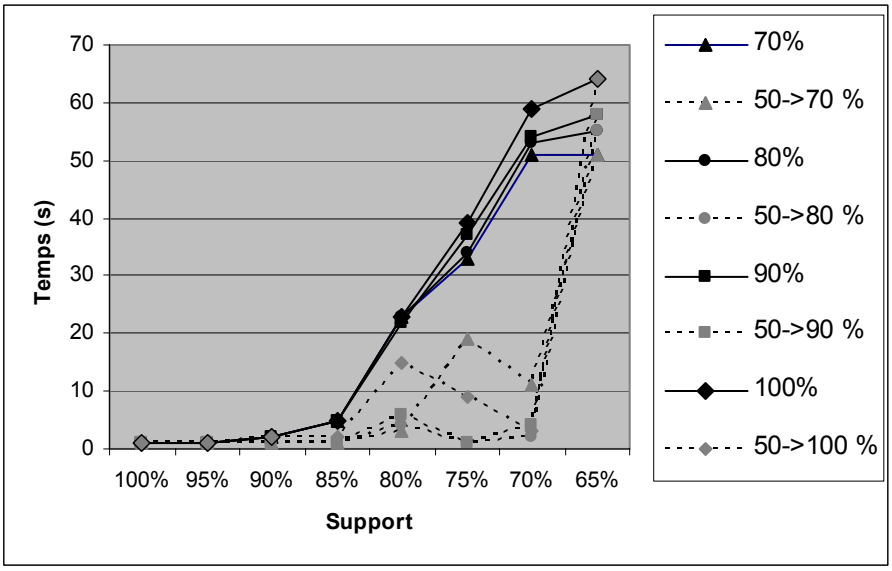

Figure 12. *Ajout de transactions*

Dans cette figure, la courbe 70%, 80%, 90%,100% représente le temps de calcul pour les supports situés en abscisse à partir de zéro. Les courbes 50->70%, 50->80%,50->90% et 50->100% représentent les temps de calcul en utilisant l'algorithme à partir d'une base de données contenant 50% des transactions initiales à laquelle sont ajoutées respectivement : 20%, puis 30%, puis 40% et enfin 50% des transactions restantes. Comme lors de l'expérimentation précédente, nous pouvons constater que l'algorithme est, dans le pire des cas, aussi rapide que celui depuis zéro (notamment pour les supports de 100%, 95% et 65%). Les courbes 50->70% et 50->80% ont des comportements similaires, signifiant que l'ajout, dans ce cas, de 20 ou 30 % de transactions de la base restant a des conséquences relativement similaire. Par contre dans le cas de 50->90% et 50->100%, nous constatons de moins bonnes performances à partir des supports allant de 80% à 65%. Globalement, l'utilisation de l'algorithme est rentable quelle que soit le support. Nous obtenons des gains allant de 38% (pour 50->80 %) à 53% (pour 50->70%). Nous poursuivons nos expérimentations dans le cas de la suppression.

Afin de tester la suppression, nous avons utilisé un protocole similaire à celui de l'ajout en découpant la base de données initiale (figure 13).

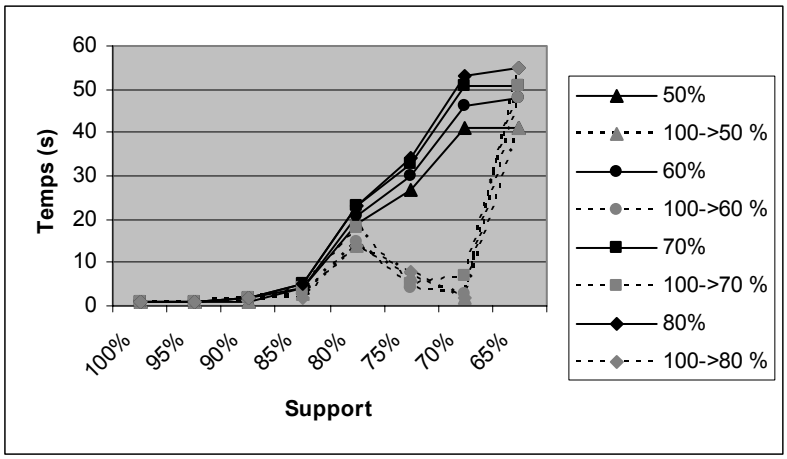

Figure 13. *Suppression de transactions*

Les courbes 100->80%, 100->70%, 100->60% et 100-60% représentent les temps de calcul à partir d'une base de données contenant 100% des transactions initiales à laquelle sont supprimées respectivement : 20%, puis 30%, puis 40% et enfin 50% des transactions. Pour des supports allant de 80 à 70%, il y a une forte augmentation des gains de l'algorithme. Par contre, pour des supports allant de 70% à 65%, les calculs nécessaires lors de la mise à jour sont importants et le gain devient minime tout comme pour des supports compris entre 100% et 85%. Globalement, l'utilisation de l'algorithme est rentable quelque soit le support comme dans le cas de l'ajout. Nous obtenons des gains allant de 49% (pour 100->80 %) à 53% (pour 100->70%). Faute de place, nous ne détaillons pas les modifications de transactions qui peuvent être vues comme une combinaison de l'ajout d'une nouvelle transaction et de la suppression d'une ancienne transaction.

7.3 Analyse de tendance

De manière à analyser les tendances sur un site Web, nous décrivons une expérience menée sur des jeux de données issus du serveur Web de l'association des anciens élèves de l'Ecole Nationale Supérieure de Chimie de Montpellier et qui concerne deux mois de connexion. Le fait d'obtenir de manière hebdomadaire les informations des serveurs (fichier log) a permis de réaliser différentes expériences. Nous considérons que le fichier nommé AAE (1) correspond, au fichier log hebdomadaire alors que le fichier AAE (2) correspond au cumul des différentes informations, i.e. au cumul de tous les fichiers AAE (1) sur la période. L'avantage

de comparer un fichier cumulé à un fichier régulier est que, dans le cas d'un fichier régulier, un comportement nouveau peut devenir fréquent alors qu'il risque d'être « noyé » par les autres comportements dans le cas du fichier cumulé.

De manière à illustrer la différence entre les deux types de fichiers, considérons la Figure 14. Cette figure illustre le comportement <(/societes/pharma.htm,/images/menu02.gif) ,(/images/menu03x.gif)(/images/menu 91.gif)> qui représente le fait que des usagers ont parcouru en même temps, i.e. dans une période de temps très courte, /societes/pharma.htm et /image/menu02.gif et qu'ensuite ils sont allés sur la page /images/menu03x.gif et qu'enfin ils sont allés sur l'url /images/menu91.gif.

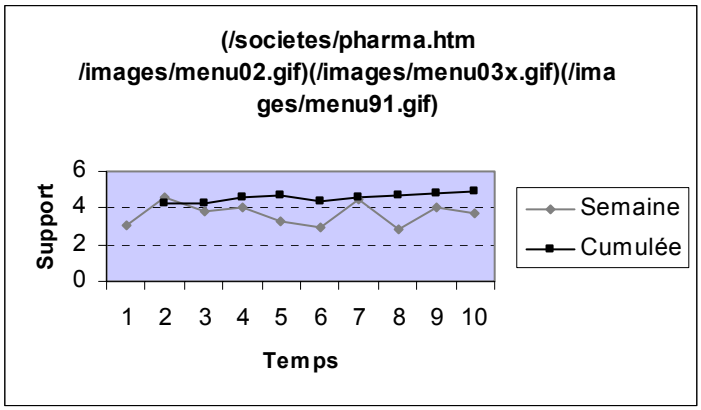

Figure 14. *Un exemple de tendances*

Dans le cas des données cumulées, nous pouvons constater que ce comportement est globalement croissant. En effet, l'ordonnée représentant le support, nous constatons qu'entre le début de l'analyse et la fin, de plus en plus de comportements d'usagers correspondent à ce motif. Par contre, si l'analyse est effectuée semaine après semaine, nous pouvons considérer que ce comportement n'est pas aussi croissant, qu'il le semblait, sur le cumul. En effet, nous constatons qu'entre la semaine 2 et la semaine 6 il était même plutôt décroissant et qu'un pic d'utilisation de ces urls a eu lieu lors de la semaine 7. La conclusion attendue se vérifie sur cette courbe. L'analyse globale ne permet pas de repérer tous les comportements, certains se retrouvent alors « noyés » dans des comportements plus globaux. Dans ce cas de figure, contrairement aux approches traditionnelles nous pouvons repérer ce type de comportement.

Considérons à présent les figures suivantes qui représentent des analyses de tendances sur une longue période de temps.

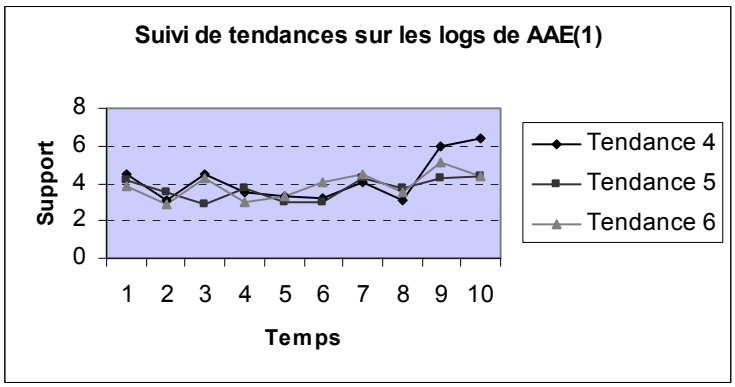

Figure 15. *Analyse de tendances par semaine sur les données AAE (1)*

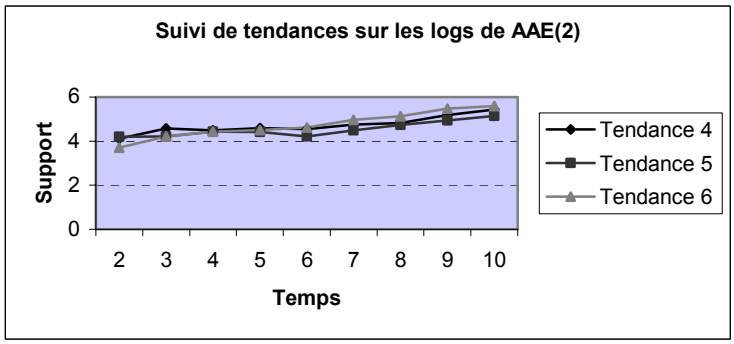

Figure 16. *Analyse de tendances sur les données cumulées AAE (2)*

Les figures 15 et 16 illustrent des tendances correspondant respectivement à :

tendance 4 : <(/,/css/00.css, /css01.css) (/images/meter150.gif)>

tendance 5 : <(/images/backgrd3.gif) (/images/menu02.gif) (/images/creuset.gif) (/images/menu91.gif)(/images/menu90.gif,/images/menu92.gif)>

tendance 6 : <(/css/00.css)(/images/backgrd3.gif) (/images/menu004a.gif)(/images/meter050.gif) (/images/menu003d.gif)>

De manière similaire à la figure 14 nous pouvons constater dans la figure 16 que les différentes tendances sont croissantes au cours du temps. Cependant, une analyse fine de la figure 15 montre que, malgré les apparences, le comportement au cours du temps des utilisateurs n'est pas aussi croissant qu'il y parait et qu'il existe même des semaines pour lesquelles celui-ci devient décroissant.

8. Travaux antérieurs

Notre approche de recherche de régularités structurelles est très proche de celle proposée dans [WaLi97, WaLi99] pour la recherche d'associations structurelles dans des données semi structurées. Les auteurs proposent une approche très efficace et des solutions basées sur une nouvelle représentation de l'espace de recherche. En outre, en proposant des optimisations basées sur des stratégies d'élagages, ils améliorent considérablement l'étape de génération des candidats. Dans [AsAb02], les auteurs proposent un algorithme appelé Find-Freq-Trees qui utilise également une approche basée sur une recherche par niveau comme dans l'algorithme Apriori [AgIm93] et étend la proposition en améliorant la technique d'énumération définie dans [Baya98] de manière à découvrir des sous structures dans de longues séquences. Enfin dans [Zaki02], l'auteur propose deux algorithmes TreeMinerH et TreeMinerV pour la recherche d'arbres fréquents dans une forêt. TreeMinerH reprend le principe du parcours en largeur de A-priori en améliorant la génération et le comptage des candidats à l'aide des classes d'équivalences, d'une structure d'arbre préfixé et de « scope list ». Quand à TreeMinerV, il propose de voir un arbre comme une structure verticale et associe à cette vision une méthode de parcours en profondeur très efficace pour la recherche de séquences longues. Dans ces deux algorithmes, la génération et le comptage des candidats sont effectués par des opérations ensemblistes sur les « scope list », la structure préfixée permet de réduire le nombre de transactions à parcourir dans la base de données. Cependant cette approche ne s'intéresse pas véritablement à la recherche des mêmes structures dans la mesure où le niveau d'imbrication n'est pas pris en compte. Outre la prise en compte des évolutions, la composante extraction possède deux différences essentielles. La première est issue de l'utilisation des vecteurs de bits pour représenter les éléments fréquents dans la base de données. Comme il a été montré dans [ArGe02], cette structure possède outre l'avantage d'être efficace (utilisation de l'opérateur logique AND pour générer les candidats) la particularité de pouvoir travailler sur de longues structures. La seconde différence réside dans le principe de constructions des candidats où nous minimisons le nombre de candidats générés étant donné que nous recherchons pour étendre uniquement les candidats de profondeur supérieure. Il est important de préciser que d'autres méthodes de recherche pour des structures d'arbre ou de graphes sont également proposées mais ne sont pas directement applicables à notre problématique. Ainsi, les auteurs de [WaSh96], proposent un algorithme de découverte de structures communes approximatives et l'appliquent à la découverte d'applications génomiques. Dehaspe et al. [DeTo98] présentent un algorithme efficace pour résoudre le problème de la découverte de sous structures fréquentes dans des graphes labellisés. Leur approche est basée sur l'utilisation d'ILP.

En ce qui concerne la maintenance des sous structures fréquentes extraites, il n'existe pas à notre connaissance de travaux dans ce domaine aussi nous nous intéressons par la suite aux travaux menés dans la maintenance des motifs séquentiels dont la problématique n'est pas trop éloigné de notre proposition en

considérant la ré-écriture de la base sous forme de séquences. Proche des motifs séquentiels et à l'origine de nombreuses approches, [ChHa96] propose un algorithme appelé FUP pour une fouille de données incrémentale dans le cas des règles d'associations. Cependant, la problématique de mise à jour incrémentale dans le cadre des motifs séquentiels est beaucoup plus complexe que celle des règles d'associations dans la mesure où l'espace de recherche, i.e. le nombre de combinaisons est beaucoup plus grand. Dans [PaZa99], les auteurs proposent un algorithme appelé ISM (Incremental Sequence Mining) basé sur l'approche SPADE [Zaki98], qui permet une mise à jour des séquences fréquentes quand de nouveaux clients et de nouvelles transactions sont ajoutés à la base de données. L'approche proposée construit un treillis de séquences qui contient tous les fréquents et les éléments de la bordure négative [MaTo96]. Quand de nouvelles informations arrivent, elles sont ajoutées à ce treillis. Le problème de cette approche est évidemment la taille croissante de la bordure négative qui dans notre cas est minimisée car basée sur des vecteurs de bits. Dans [MaPo00], l'algorithme ISE (Incremental Sequence Extraction) a été proposé pour la recherche de motifs fréquents, il génère des candidats dans toute la base de données en attachant les séquences de la base de données incrémentale à ceux de la base originale. Cette approche évite de garder les séquences contenues dans la bordure négative et le recalcul de ces séquences quand la base de données initiale a été mise à jour. Cependant, en ne conservant pas la bordure négative, il est nécessaire de parcourir plus souvent la base pour rechercher les candidats. Dans [ZhXu02] l'algorithme proposé utilise à la fois les notions de bordure négative de la base de données originelle et des notions de suffixes et préfixes contrairement à ISE. Pour contrôler la taille de cette bordure négative, ils introduisent un support minimum pour ces éléments réduisant ainsi la taille de celle-ci. De plus cet algorithme réalise une extension par préfixe et par suffixe (à l'aide de la bordure négative). Le problème de cet algorithme réside dans le choix de la valeur du support minimum pour la bordure négative. Comme nous pouvons le constater deux types d'approches existent. L'une basée sur la bordure négative et l'autre sur l'utilisation d'une approche spécifique de génération de candidats. Notre approche peut être considérée comme une extension de celle de [PaZa99] dans la mesure où d'une part elle est associée à une approche complète de maintenance et d'autre part elle prend en compte aussi bien les ajouts que les suppressions. Les expériences menées ont montré que le choix de la bordure négative pouvait considérablement améliorer le processus de maintenance.

Enfin, Il n'existe, à notre connaissance, que peu de travaux analysant les différentes tendances d'évolution des structures fréquentes au cours du temps. Cependant, de nombreux travaux existent pour analyser des tendances, notamment dans le cas de séries temporelles[3] (mouvements à long terme ou court terme, mouvements cycliques, mouvements aléatoires, …) ou dans des données textuelles. Proche de notre problématique, nous pouvons citer les travaux de [LeAg97] qui ont

[3]. Une présentation détaillée des tendances dans des séries temporelles est proposée dans [HaKa01].

largement inspiré ce travail. Les auteurs proposent un système pour identifier des tendances dans des documents textuels. Le principe utilisé est le suivant. Après un prétraitement sur les données, ils utilisent un algorithme de recherche de motifs séquentiels pour déterminer des phrases et conservent l'historique associé à chacun des motifs extraits. Ensuite, ils recherchent les phrases qui correspondent à une tendance à l'aide d'un langage de définition de formes. Les expérimentations menées par les auteurs concernent l'analyse de tendances dans des bases de données de brevets. Notre approche est une adaptation de ces travaux à la problématique de l'analyse des tendances dans le cas de base de données de sous arbres.

9. Conclusion

Dans cet article, nous avons proposé une approche pour rechercher des régularités dans des bases de données d'objets semi structurés. Ces objets sont représentés sous la forme d'arbres dans lesquels nous recherchons les sous arbres les plus fréquents, i.e. ceux qui apparaissent suffisamment fréquemment dans la base de données. Pour cela, nous représentons ces arbres sous la forme de séquences et nous utilisons un algorithme basé sur une structure de vecteurs de bits pour rechercher efficacement les sous arbres fréquents. L'approche que nous proposons possède en plus l'originalité d'intégrer différents composants : extraction, maintenance de la connaissance extraite pour tenir compte des mises à jour des données sources et enfin analyse de tendances. Les différentes expérimentations menées ont permis de montrer la validité de l'approche. Les connaissances extraites doivent pouvoir être utilisées, par exemple, pour faciliter la création d'index ou de vues sur les bases de données analysées. Nos travaux de recherche actuels se poursuivent dans cette direction.

10. Références

[AbBu00] S.Abiteboul, P. Buneman and D. Suciu, *"Data on the Web"*, Morgan Kaufmann, San Francisco, CA, USA, 2000.

[AgIm93] R. Agrawal, T. Imielinski and A. Swami, "*Mining Association Rules between Sets of Items in Large Databases"*, Proceedings of the International Conference on Management of Data (SIGMOD'93), pp. 207-216, Washington DC, USA, May 1993.

[AgSr95] R. Agrawal and R. Srikant, "*Mining Sequential Patterns*", Proceedings of the International Conference on Data Engineering (ICDE'95), pp. 3-14, Tapei, Taiwan, March 1995.

[ArGe02] J. Ares, J. Gehrke, T. Yiu and J. Flannick, "*Sequential Pattern Using Bitmap Representation*", Proceedings of Principles and Practice of Knowledge Discovery in Data (PKDD'02), Edmonton, Canada, July 2002.

[AsAb02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto and S. Arikawa, *"Efficient Substructure Discovery from Large Semi-Structured Data"*, Proceedings of the International Conference on Data Mining (ICDM'02), Washington DC, USA, April 2002.

[Baya98] R. J. Bayardo, *"Efficiently Mining Long Patterns from Databases"*, Proceedings of the International Conference on Management of Data (SIGMOD'98), pp. 85-93, Seattle, USA , June 1998.

[ChAb98] S. Chawathe, S. Abiteboul and J. Widom, *"Representing and Querying Changes History in Semistructured Data"*, Proceedings of the International Conference on Data Engineering (ICDE98), Orlando, USA, February 1998.

[ChHa96] D. W. Cheung, J. Han, V. Ng and C. Y. Wong, "*Maintenance of Discovered Association Rules in Large Databases: an Incremental Update Technique*", Proceedings of the International Conference on Data Engineering (ICDE'96), pp. 116-114, New Orleans, USA, February 1996.

[DeTo98] L. Dehaspe, H. Toivonen and R. D. King, *"Finding Frequent Substructures in Chemical-compounds"*, Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'98), pp. 30-36, New York, USA, August 1998.

[GoMi90] M. Gondran et Michel Minoux, *"Graphes et algorithmes"*, Editions Eyrolles, 1990.

[HaKa01] J. Han and M. Kamber, "*Data Mining – Concepts and Techniques*", Morgan Kaufmann Publishers, 2001.

[HeMa00] I. Herman and M.S. Marshall, "*GraphXML An XML based graph interchange format*", Centre for Mathematics and Computer Sciences (CWI), Technical Report INS-R0009, pp. 52-62, Amsterdam, 2000.

[LaPo03] P.A. Laur and P. Poncelet, "*AUSMS : un environnement pour l'extraction de sous-structures fréquentes dans une collection d'objets semi-structurées*", Actes des Journées d'Extraction et Gestion des Connaissances (EGC'03), Lyon, France, 2003.

[LaTe03] P.A. Laur, M. Teisseire and P. Poncelet, "*AUSMS: an Environment for Frequent Sub-Structures Extraction in a Semi-Structured Object Collection*", In proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA03), Prague, Czech Republic, pp. 38-45, September 2003.

[LeAg97] B. Lent, R. Agrawal and R. Srikant. "*Discovering Trends in Text Databases*". In Proceedings of the 3rd International Conference on Knowledge, Newport Beach, California, August 1997.

[MaHe00] M. Marshall, I. Herman and G. Melancon, *"An Object-oriented Design for Graph Visualization"*, Technical Report INS-R00001, Centre for Mathematics and Computer Sciences, Amsterdam, 2000.

[MaPo00] F. Masseglia, P. Poncelet and M. Teisseire, "*Incremental Mining of Sequential Patterns in Large Database*", Actes des 16ièmes Journées Bases de Données Avancées (BDA'00), Blois, France, Octobre 2000.

[MaTo96] H. Mannila and H. Toivonen. "*On an Algorithm for Finding all Interesting Sequences* ". In Proceedings of the 13th European Meeting on Cybernetics and Systems Research, Vienna, Austria, April 1996.

[PaZa99] S. Parthasarathy and M. J. Zaki, "*Incremental and Interactive Sequence Mining*", Proceedings of the Conference on Information and Knowledge Management (CIKM'99), pp. 251-258, Kansas City, USA, November 1999.

[WaLi97] K. Wang and H. Liu, "*Schema Discovery for Semi-structured Data"*, Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'97), pp. 271-274., Newport Beach, USA, August 1997.

[WaLi99] K. Wang and H. Liu, "*Discovering Structural Association of Semistructured Data*", In IEEE Transactions on Knowledge and Data Engineering, pp. 353-371, January 1999.

[WaSh96] J.TL. Wang, B.A. Shapiro, D. Shasha, K. Zhang and C.Y Chang, *"Automated Discovery Structures"*, In Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'96), pp. 70-75, 1996.

[W3C03] W3C, Extensible Markup Language (XML), http://www.w3.org/XML, June 2003.

[Zaki98] M. Zaki, "*Scalable Data Mining for rules*", PHD Dissertation, University of Rochester-NewYork, 1998.

[Zaki02] M. Zaki, *"Efficiently Mining Frequent Trees in a Forest "*, Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD'02), Edmonton, Canada, July 2002.

[ZhXu02] Q. Zheng, K. Xu, S. Ma and W. Lu, "*The Algorithms of Updating Sequential Patterns*", Proceedings of the International Conference on Data Mining (ICDM'02), Washington DC, USA, April 2002.