Recherche de sous-structures fréquentes pour l'intégration de schémas XML

Federico Del Razo López*, Anne Laurent*
Pascal Poncelet**, Maguelonne Teisseire*

* LIRMM - Université Montpellier II, 161 rue Ada 34392 Montpellier cedex 5
{delrazo,laurent,teisseire}@lirmm.fr
**EMA - LGI2P/Site EERIE, Parc Georges Besse 30035 Nîmes cedex 1
Pascal.Poncelet@ema.fr

Résumé. La recherche d'un schéma médiateur à partir d'un ensemble de schémas XML est une problématique actuelle où les résultats de recherche issus de la fouille de données arborescentes peuvent être adoptés. Dans ce contexte, plusieurs propositions ont été réalisées mais les méthodes de représentation des arborescences sont souvent trop coûteuses pour permettre un véritable passage à l'échelle. Dans cet article, nous proposons des algorithmes de recherche de sous-schémas fréquents basés sur une méthode originale de représentation de schémas XML. Nous décrivons brièvement la structure adoptée pour ensuite détailler les algorithmes de recherche de sous-arbres fréquents s'appuyant sur une telle structure. La représentation proposée et les algorithmes associés ont été évalués sur différentes bases synthétiques de schémas XML montrant ainsi l'intérêt de l'approche proposée.

1 Introduction

Étant donné l'explosion du volume de données disponibles sur Internet, il devient indispensable de proposer de nouvelles approches pour faciliter l'interrogation de ces grandes masses d'information afin de retrouver les informations souhaitées. L'une des conditions sine qua non pour permettre d'interroger des données hétérogènes est de disposer d'un (ou de plusieurs) "schéma général" que l'utilisateur pourra interroger et à partir duquel les données sources pourront être directement accédées. Malheureusement les utilisateurs ne disposent pas de moyen de connaître les modèles sous-jacents des données qu'ils souhaitent accéder et l'un des challenges dans ce contexte est donc de fournir des outils pour extraire, de manière automatique, ces schémas médiateurs. Un schéma médiateur est alors considéré comme une interface permettant à l'utilisateur l'interrogation des sources de données : l'utilisateur pose ses requêtes de manière transparente et n'a pas à tenir compte de l'hétérogénéité et de la répartition des données.

XML étant maintenant prépondérant sur Internet, la recherche de moyens d'intégration de tels schémas est un domaine de recherche actif. Si les recherches permettant l'accès aux données, quand un schéma d'interrogation est connu, sont maintenant bien avancées (Xylème, 2001), les recherches concernant la définition automatique d'un schéma médiateur restent incomplètes et non satisfaisantes (Tranier et al., 2004). Il est alors intéressant de considérer les

travaux réalisés dans le contexte de la fouille de données afin d'obtenir un schéma fréquent ou un ensemble de sous-schémas fréquents. Ces derniers offrent alors des éléments pertinents pour la construction du schéma médiateur. Dans le but de proposer une approche permettant de répondre à cette dernière problématique, nous nous focalisons sur la recherche de sous-structures fréquentes au sein d'une base de données de schémas XML. Une sous-structure fréquente est un sous-arbre se trouvant dans *"la plupart"* des schémas XML considérés. Cette proportion est examinée au sens d'un *support* qui correspond à un nombre minimal d'arbres de la base dans lesquels le sous-arbre doit se retrouver pour être considéré comme *fréquent*. Une telle recherche est complexe dans la mesure où il est nécessaire de traduire l'ensemble des schémas en une structure aisément manipulable. Cette transformation des données conduit parfois à doubler ou tripler la taille de la base initiale dès lors que l'on souhaite utiliser des propriétés spécifiques permettant d'améliorer le processus de fouille. Il n'existe pas de solution efficace à ce problème alliant une représentation compacte à des propriétés intéressantes. L'objet de cet article est la définition d'une approche de fouille de données de type XML répondant à cet objectif.

Cet article est structuré de la manière suivante : la section 2 introduit les définitions des différentes inclusions dans le contexte des structures hiérarchiques et propose un aperçu des principales approches existantes de fouille de données arborescente. Nous présentons également en détail la problématique étudiée. La section 3 présente notre proposition : une méthode de recherche de sous-schémas fréquents utilisant les propriétés d'une structure de données arborescentes compacte et originale. Les différentes expérimentations menées sur des bases de schémas XML sont décrites dans la section 4. Enfin, la section 5 conclut et présente les principales perspectives associées à nos travaux.

2 Définitions, problématique et travaux connexes

2.1 Définitions préliminaires

Un *arbre* est un graphe orienté, connexe sans cycle. Il est composé d'un ensemble de nœuds reliés par des arcs et il existe un nœud particulier nommé *racine*. Il s'agit d'un *arbre ordonné* s'il existe un ordre entre les fils d'un nœud et d'un *arbre non ordonné* sinon.

Définition 1 *Un arbre enraciné, étiqueté et ordonné $T = (N, B, \Sigma, \mathcal{L}, r, \prec)$ est tel que : N est un ensemble fini de nœuds ; B est un ensemble de branches ($B \subseteq N^2$). Chaque branche $b \in B$ est un couple ordonné (u, v) de nœuds où u est le* père *de v ; Σ est un ensemble fini d'étiquettes ; $\mathcal{L}$ est une fonction $\mathcal{L} : N \rightarrow \Sigma$, $\mathcal{L}(u) = l$, $\{u \in N, l \in \Sigma\}$; r est la racine de T, $r \in N$; et $\prec$ est une relation d'ordre entre les fils de chaque nœud interne. La taille de T, notée $|T|$, est le nombre de nœuds de T.*

Lorsque nous manipulerons plusieurs arbres, nous noterons, pour un arbre T, N_T, B_T, $\mathcal{L}_T$, et $\prec_T$ pour $N, B, \mathcal{L}, r$ et $\prec$. De plus, dans la suite de cet article, nous utilisons le mot *arbre* pour un un arbre enraciné, étiqueté et ordonné.

Définition 2 (inclusion) *Soient S et T deux arbres, nous disons que S est inclu dans un arbre T noté par $S \sqsubseteq T$, s'il existe une fonction injective ϕ: $N_S \rightarrow N_T$ des nœuds de S aux nœuds de T,qui vérifie les conditions suivantes pour tout nœud $u, v \in N_S$:*

1. ϕ *préserve les étiquettes,* $\mathcal{L}_S(u) = \mathcal{L}_T(\phi(u))$
2. ϕ *préserve la relation d'ordre entre frères, si* $u \prec_S v$ *alors* $\phi(u) \prec_T \phi(v)$
3. ϕ *préserve les relations :*
   (a) de parenté *tel que* $\forall u, v \in N_S$*, si* $(u, v) \in B_S$ *alors* $(\phi(u), \phi(v)) \in B_T$*, ou*
   *(b) d'*ancestralité *tel que* $\forall u, v \in N_S$*, si* $(u, v) \in B_S$ *alors* $(\phi(u), \phi(v)) \in B_T^+$.

Une inclusion est dite *induite* si les relations de parenté sont préservées. Par ailleurs, si les relations d'ancestralité sont respectées, il s'agit d'une inclusion *incrustée*.

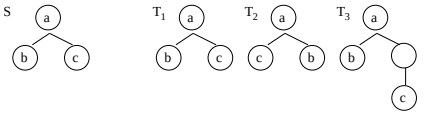


FIG. 1 – *Exemple d'arbres pour l'inclusion* induite *et* incrustée.

Par exemple, considérons les arbres S, T_1, T_2, et T_3 représentés dans la figure 1. Si les relations de parenté sont respectées, il s'agit d'une inclusion *induite*, donc S est inclus de manière induite dans l'arbre T_1 ($S \sqsubseteq T_1$). Si les relations d'ancestralité sont conservées, alors on trouve une inclusion *incrustée* avec $S \sqsubseteq T_1$ et $S \sqsubseteq T_3$. S n'est pas inclu dans T_2 car ϕ ne préserve pas l'ordre entre les frères.

Dans la suite de cet article, nous considérons une inclusion de type induite car nous souhaitons traiter l'ordre existant entre les nœuds dans la hiérarchie de façon directe et l'ordre entre les nœuds de même niveau de façon indirecte. Nous pouvons donc définir le support d'un sous-arbre selon cette inclusion de la manière suivante :

Définition 3 (support) *Soit D une forêt d'arbres. Soit S un arbre de D. Soit σ un support minimal spécifié par l'utilisateur.*
Le support de S est défini par $Support(S) = \frac{P(S)}{N}$ où
$P(S)$ est le nombre d'arbres dans D incluant S **(inclusion induite)** *;*
N est le nombre d'arbres dans D.
Si $Support(S) \geq \sigma$ alors l'arbre S est dit fréquent dans D.

2.2 Problématique

La problématique étudiée au sein de cet article est la recherche de sous-structures fréquentes, i.e. de sous arbres qui apparaissent suffisamment fréquemment dans des documents XML. Nous considérons, par la suite, qu'une étape initiale de pré-traitement est réalisée sur les documents XML de manière à ne retenir que leur structure sous forme d'arbre. Nous considérons également qu'à l'issue de cette phase, l'étiquetage des nœuds est homogène, i.e deux nœuds de même étiquette dans deux arbres différents partagent non seulement la même syntaxe mais également la même sémantique.

L'objectif consiste alors à rechercher, à partir de la forêt d'arbres obtenue D et en fonction d'un support minimal spécifié par l'utilisateur, les sous arbres qui apparaissent suffisamment fréquemment, i.e. dont leur nombre d'occurrences dans D est supérieur ou égal au support

minimal. Pour répondre à cette problématique, nous nous trouvons donc confrontés aux deux problèmes suivants :

1. *Quelle structure de représentation efficace utiliser ?* Idéalement, étant donné que nous considérons de grandes quantités d'arbres, nous souhaitons avoir une structure qui non seulement soit efficace en mémoire mais également adaptée aux traitements que nous souhaitons faire.
2. *Comment tester efficacement l'inclusion d'un arbre dans un sous arbre ?* Rechercher l'ensemble des sous arbres fréquents nécessite de parcourir tous les arbres et d'effectuer de très nombreuses comparaisons pour réussir à extraire des sous parties communes. Il est donc indispensable de pouvoir trouver rapidement à partir de quel nœud la comparaison peut être effectuée si nous souhaitons améliorer l'efficacité de la recherche.

2.3 Les travaux existants

Dans cette partie, nous nous intéressons non seulement aux approches de recherche mais nous examinons également les méthodes de représentation des arbres. Les travaux dans le domaine de la fouille de données arborescentes peuvent être distingués selon qu'ils traitent les arbres ordonnés ou non. Nous situant dans le contexte de schémas XML, il s'avère nécessaire de traiter l'ordre des éléments si celui-ci est spécifié. Nous nous focaliserons donc sur des propositions prenant en charge les arbres ordonnés.

A notre connaissance, il existe très peu de travaux proposant des méthodes d'extraction pour les arbres ordonnés (Zaki, 2002; Asai et al., 2002). Ainsi Zaki (2002) propose l'algorithme *TreeMiner* pour extraire des sous-arbres fréquents selon une inclusion incrustée. Une représentation originale des arbres facilite la gestion des candidats et offre des performances intéressantes. (Asai et al., 2002) traitent également de la problématique des arbres ordonnés selon la définition de l'inclusion induite. L'approche proposée, FREQT, adopte une structure de représentation du type « first-child/next-sibling » comme illustrée figure 2. Lors du processus de fouille, pour chaque structure fréquente, FREQT conserve la liste des nœuds les plus à droite dans les arbres de la base de données supportant cette structure. Nous illustrons ceci figure 3 où pour le fréquent a, les 6 positions dans la base de données sont stockées, et pour le fréquent $c - a$ les 3 positions les plus à droite sont stockées. Cette information représente les positions où cette structure est supportée dans la base.

Si nous examinons plus attentivement la représentation verticale adoptée dans *TreeMiner*, elle aboutit en fait à stocker trois fois la taille d'un arbre, i.e. $3|T|$. De la même manière la structure utilisée dans FREQT offre des performances attractives, mais cette représentation conduit également à tripler la taille de la base afin de stocker les informations nécessaires.

Même si elles n'abordent pas la même problématique, des approches de représentation efficaces des arbres en $2|T|$ ont été récemment proposées (Wang et al., 2004; Chi et al., 2004, 2003). Cependant, outre le fait qu'elles ne considèrent pas la notion d'ordre, elles n'utilisent pas des propriétés aussi intéressantes que les travaux précédents afin d'améliorer le processus d'extraction.

Notre objectif est donc de permettre une recherche de sous-arbres ordonnés mais, contrairement aux approches existantes dans ce contexte, d'utiliser une représentation peu coûteuse en mémoire, i.e. en $2|T|$. Cette structure doit en outre posséder des propriétés intéressantes

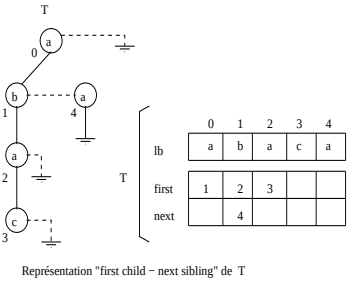


FIG. 2 – *FREQT : représentation d'un arbre.*

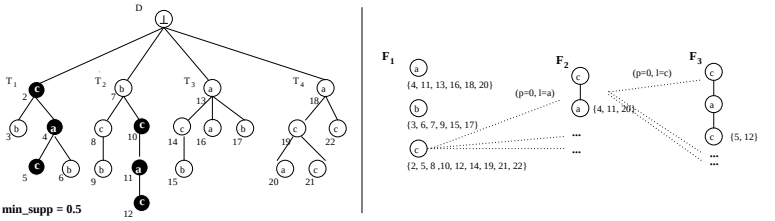


FIG. 3 – *FREQT : stockage de la liste des positions des fréquents.*

pour améliorer le processus d'extraction. C'est dans ce contexte que se situe notre proposition RSF décrite à la section 3.

3 Proposition

Dans cette section, nous proposons de nouveaux algorithmes permettant l'extraction efficace de sous-arbres fréquents ordonnés au sein d'une base de données arborescentes. Dans un premier temps, nous décrivons la structure adoptée et nous en soulignons ses intérêts. Dans un second temps, nous proposons un survol de notre approche d'extraction et nous montrons comment les propriétés de la structure sont utilisées pour améliorer le processus de fouille. Finalement, nous décrivons plus formellement les algorithmes proposés.

Pour illustrer nos propos, nous utiliserons la base d'arbres de la figure 4.

3.1 Représentation des arbres

Pour représenter les différents arbres manipulés au sein de notre approche, nous adoptons la représentation proposée dans (Del Razo et al., 2005). Un arbre est ainsi décrit à l'aide de deux vecteurs comme proposé dans Weiss (1998). Le premier, nommé st, conserve la position du père de chaque nœud. Les nœuds de l'arbre sont numérotés en profondeur d'abord. La racine de T correspond à l'index 0 et a pour valeur $s[0] = -1$ pour indiquer que la racine n'a

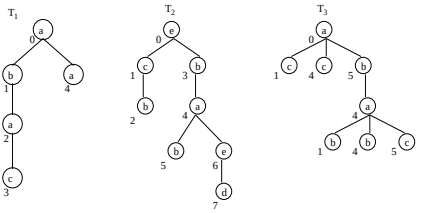


FIG. 4 – *La forêt d'arbres exemple.*

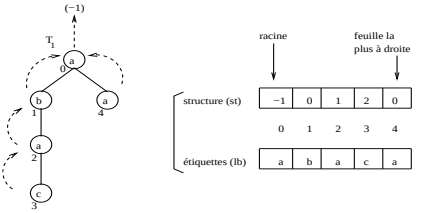


FIG. 5 – *RFS : représentation d'un arbre.*

pas de père. Les valeurs $st[i], i = 0, 2, ..., k-1$ correspondent aux positions du père des nœuds i, comme illustré figure 5.

Cette représentation permet de retrouver en temps constant le père d'un nœud. De plus, elle permet la localisation directe de la *feuille la plus à droite* par rapport à l'index k. En parcourant l'arbre, il est ainsi possible d'obtenir toutes les relations directes *père-fils* entre nœuds. Le deuxième vecteur, nommé lb, est utilisé pour enregistrer les étiquettes de l'arbre avec $lb[i], i = 0, 2, ..., k-1$ représentant l'étiquette de chaque nœud $n_i \in T$.

La structure adoptée permet une représentation des arbres peu coûteuse puisqu'elle se réduit à $2|T|$. De plus elle possède des propriétés intéressantes, évoquées au paragraphe suivant, pouvant être utilisées lors de la recherche de sous-structures fréquentes.

3.2 Aperçu général

Notre proposition est basée sur une approche classique de type «générer-élaguer», i.e. à chaque étape, nous générons différents candidats et nous testons si ceux-ci sont inclus dans la bases d'arbres. L'inclusion dans notre cas est bien entendu définie comme étant de type «induit».

La méthode de représentation des arbres que nous proposons permet de générer de manière efficace les sous-arbres candidats puis d'élaguer les sous-arbres non fréquents (après calcul du support).

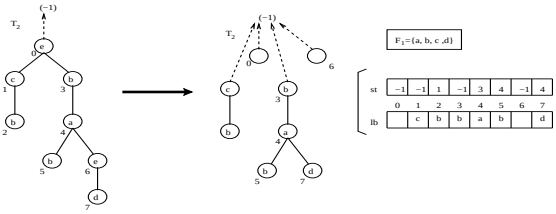


FIG. 6 – *Transformation d'un arbre de la base après génération de* F_1.

Considérons à présent comment les candidats sont générés. Les candidats de taille 1 sont tout d'abord obtenus en parcourant tous les nœuds des arbres de la base de données. Chaque nœud voit son support incrémenté lors de ce parcours et seuls sont conservés les nœuds dont le

support est supérieur au support minimal défini par l'utilisateur. La base de données est alors transformée pour ne conserver que les nœuds fréquents, comme illustré par la figure 6.

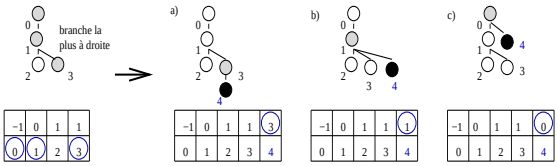


FIG. 7 – *Génération des candidats.*

Les candidats de taille 2 sont générés en combinant deux à deux tous les fréquents de taille 1. La génération des candidats de taille $k \geq 3$ s'effectue de la même manière que dans les approches classiques de type Apriori (Agrawal et Srikant, 1994), par combinaison des fréquents de taille $k-1$. Nous adoptons la stratégie de génération de candidats selon la branche la plus à droite comme proposée dans (Asai et al., 2002; Zaki, 2002) et illustrée figure 7. Nous pouvons ainsi constater l'intérêt de notre structure de représentation puisque, naturellement, il suffit d'ajouter un nouvel élément dans la représentation de l'arbre en spécifiant le père du nouveau nœud.

Le calcul du support de chaque candidat consiste à compter le nombre d'arbres de la base qui contiennent ce sous-arbre candidat. Ainsi pour chaque arbre de la base, nous recherchons les *points d'ancrage* sur lesquels la racine du sous-arbre à tester peut s'instancier. Ces points correspondent en fait aux nœuds dans l'arbre qui correspondent à la racine de l'arbre à tester. Pour chaque point d'ancrage trouvé, on cherche alors à instancier l'ensemble des nœuds de l'arbre candidat au sein de l'arbre courant testé, i.e. les fils du nœud à tester. Notons que dans le cas d'une inclusion *induite*, nous recherchons une instanciation *exacte* du candidat au sein des arbres de la base. Si tous les nœuds du candidat ont été trouvés, l'arbre supporte le candidat et le support de la structure candidate est alors incrémenté.

3.3 Les algorithmes

L'algorithme RFS (Algorithme 1) fonctionne de la manière suivante : un premier parcours sur la base est réalisé pour extraire les items dont le nombre d'occurrences est supérieur au support minimal. Ces items constituent des arbres résumés à une seule racine, l'item considéré. Nous obtenons ainsi l'ensemble F_1 des arbres fréquents de taille 1. Ces derniers sont combinés entre eux pour former des candidats de taille 2 et un parcours sur la base permet d'obtenir l'ensemble F_2 constitué des arbres de taille 2. L'algorithme se poursuit en générant des candidats de taille k+1 et en effectuant un parcours sur la base pour compter le nombre d'occurrences de chaque candidats. Lorsque plus aucun candidat ne peut être généré l'algorithme se termine.

L'algorithme GenCandidats(F_{k-1}) (Algorithme 2) décrit la génération des candidats qui utilise la branche la plus à droite des sous-arbres fréquents de taille $k-1$ afin de proposer des candidats de taille k. Pour chaque arbre fréquent de taille $k-1$, il génère un nouveau candidat en étendant l'arbre par la branche la plus à droite. Cette génération est obtenue par l'intermédiaire de la fonction Bpd. Ainsi, pour chaque nœud, nous lui ajoutons les seules extensions possibles, i.e. celles qui s'avèrent fréquentes dans F_2.

Entrée : $D = \{T_1, T_2, .., T_n\}$ base de données d'arbres ; σ le support minimal.
Sortie : F sous-arbres fréquents.

$F_1 \leftarrow$ arbres fréquents de taille 1;
$F_2 \leftarrow$ arbres fréquents de taille 2;
$F \leftarrow F_1 \cup F_2$;
pour $(k = 3; F_{k-1} \neq \emptyset; k++)$ **faire**
  $\mathcal{C}_k \leftarrow GenCandidats(F_{k-1})$;
  pour *chaque* $C \in \mathcal{C}_k$ **faire**
    si $Support(C) \geq \sigma$ **alors**
      $F_k \leftarrow F_k \cup C$;
  $F \leftarrow F \cup F_k$;
retourner F;

Algorithme 1: RSF(D, σ).

Entrée : F_{k-1} des $(k-1)$-sous-arbres fréquents.
Sortie : C_k des (k)-sous-arbres candidats.

$C_k \leftarrow 0$;
pour *chaque* $f \in F_{k-1}$ **faire**
  pour *chaque nœud* $n \in Bpd(f)$ **faire**
    pour *chaque* $< x, y > \in F_2$ **faire**
      si $\mathcal{L}(n) = \mathcal{L}(x)$ **alors**
        $tmp \leftarrow f + < n, y >$;
        $C_k = C_k + tmp$;
retourner C_k;

Algorithme 2: GenCandidats(F_{k-1}).

Entrée : C candidat.
Sortie : Support du candidat C.
$cnt \leftarrow 0$;
pour *chaque* $T \in D$ **faire**
  $trv \leftarrow faux$;
  pour $(i = 0; i < (|T| - |C|)$ && $\neg trv; ++i)$ **faire**
    si $(L(r_c) = L(T[i]))$ **alors**
      $trv \leftarrow Ancre(C, T, i)$;
    si *trv* **alors**
      $++cnt$;
$sup = \frac{cnt}{|D|}$;
retourner sup

Algorithme 3: Support(C).

Entrée : C candidat , T un arbre, i index de la racine de l'ancrage.
Sortie : *vrai* si T supporte C.

nbnœuds$\leftarrow 1$;
$N_{niv\_act} \leftarrow$ *NœudsNivSuiv*$(C, \emptyset)$;
while $(|N_{niv\_act}|! = \emptyset$ && $\neg Poursuit(N_{niv\_act}, T,$ *nbnœuds*$))$ **do**
  $N_{niv\_act} \leftarrow$ *NœudsNivSuiv*(C, N_{niv_act});
si *(nbnœuds* $= |C|$*)* **alors retourner** vrai;
sinon retourner faux;

Algorithme 4: Ancre(C, T, i).

Le calcul du support de chaque candidat consiste à compter le nombre d'arbres de la base qui contiennent ce sous-arbre candidat.

Pour chaque arbre de la base, une recherche est effectuée pour voir s'il existe des *points d'ancrage* sur lesquels la racine du sous-arbre à tester peut s'instancier (appel à l'algorithme Ancre). Si un sous-arbre existe son nombre d'occurrences est alors incrémenté et son support est retourné.

Considérons l'algorithme de gestion des points d'ancrage (Algorithme 4). Pour chaque point d'ancrage trouvé, i.e. pour chaque nœud du sous arbre candidat c qui possède le même

label dans l'arbre T, on cherche à instancier l'ensemble des nœuds de l'arbre candidat au sein de l'arbre couramment testé T. En d'autres termes, nous souhaitons projeter le sous-arbre candidat c dans l'arbre T. Ceci est réalisé par l'intermédiaire des algorithmes Ancre et Poursuit (cf. algorithmes 4 et 5).

L'algorithme Poursuit est utilisé pour chercher une instanciation *exacte* du candidat au sein des arbres de la base. Si tous les nœuds du candidat ont été trouvés, l'algorithme retourne alors la valeur $VRAI$ (l'arbre supporte le candidat). Il retourne la valeur $FAUX$ si tous les nœuds de l'arbre ont été parcourus sans trouver l'ensemble des nœuds du candidat.

Algorithme : Poursuit($N_{niv\_act}, T,$ *nbnœuds*))
Entrée : N_{niv_act} ensemble de nœuds à trouver ; T l'arbre ; *nbnœuds* le nombre de nœuds vérifiés.
Sortie : vrai si tous les nœuds de N_{niv_act} ont été trouvés.

pour *chaque* $n \in N_{niv\_act}$ **faire**
  si $(L(n) = L(\phi(n))$ && $Pere(n) = Pere(\phi(n)))$ **alors**
    $++$*nbnœuds*;
si *(nbnœuds*$= |N_{niv\_act}|$*)* **alors retourner** vrai;
sinon retourner faux;

Algorithme 5: Poursuite de la recherche.

4 Expérimentations

Nos expérimentations ont été réalisées avec un PC Pentium ayant 512 Mo RAM sous le système Linux 2.4. Les programmes ont été développés en C++ et compilés avec gcc 3.2.2.

Nous avons utilisé 6 bases de données construites en employant le programme de génération d'arbres XML proposé par (Termier et al., 2002). Ce programme propose différents paramètres pour spécifier le nombre d'arbres à générer, leur profondeur, le nombre d'étiquettes maximales. Les différentes valeurs utilisées pour les générations lors de nos expérimentations sont indiquées dans le tableau 1.

Paramètres	Valeurs
Nombre d'arbres à générer (x 1000)	10, 40, 70, 100, 130, 150
Profondeur maximal d'un arbre	5
Nombre maximal de branches par nœud	5
Nombre maximal d'étiquettes	50
Arbres fréquents semés dans la base générée	10
Probabilité qu'un nœud soit père	0.4

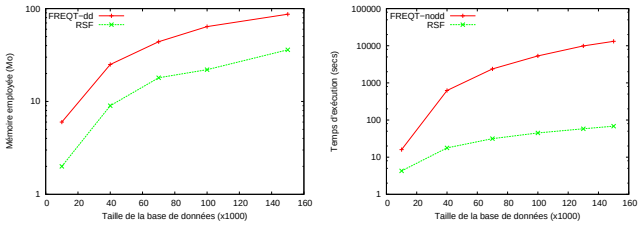
TAB. 1 – *Paramètres pour la construction de la base de données synthétiques.*

Nous avons souhaité évaluer notre proposition selon deux aspects : temps de réponse et occupation mémoire. En effet, nous argumentons notre proposition comme étant plus efficace pour un réel passage à l'échelle mais ceci n'est pas toujours synonyme d'efficacité en temps de réponse. En fait, les expérimentations réalisées prouvent que notre proposition répond aux deux critères.

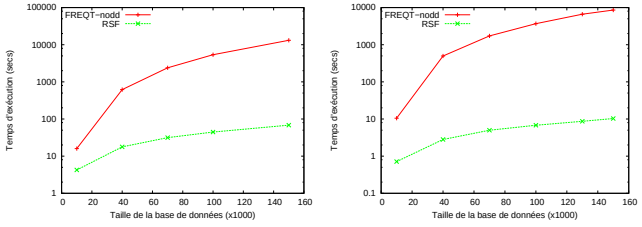
Pour évaluer les performances sur les temps d'exécution, nous nous sommes comparés à l'algorithme FREQT-nodd sans détection des duplicats de (Asai et al., 2002) permettant de rechercher des inclusions induites puis à une optimisation de celui-ci FREQT-dd limitant le parcours dans les arbres lors de la vérification des candidats.

La figure 8-(a) représente l'occupation mémoire utilisée pour la représentation de la base de schémas XML. Comme nous nous y attendions RSF occupe moins d'espace mémoire puisqu'il adopte une structure de représentation plus réduite que FREQT-nodd et FREQT-dd. Ces deux derniers adoptent la même structure.

Les figures 8-(b,c,d) indiquent les temps d'exécution obtenus par FREQT-nodd et RSF pour différents supports et différentes tailles de la base de schémas. Nous pouvons constater que RSF obtient dans tous les cas de meilleurs résultats.



(a) $min\_sup = 0.1$ (b) $min\_sup = 0.05$



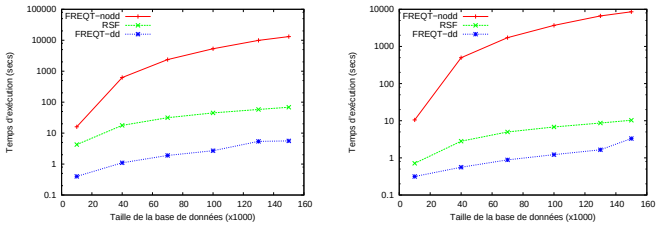
(c) $min\_sup = 0.1$ (d) $min\_sup = 0.5$

FIG. 8 – *Temps d'exécution par rapport à la taille de la base, FREQT-nodd et RSF.*

En analysant les résultats indiqués par la figure 9 comparant FREQT-nodd, FREQT-dd et RSF, il s'avère que les temps d'exécution sont nettement améliorés pour FREQT-dd. Le gain obtenu en terme d'espace mémoire de RSF est donc à comparer avec les temps d'exécution ob-

tenus à l'aide d'une optimisation basée sur le stockage d'informations supplémentaires comme indiqué au paragraphe 2.3.

Nous souhaitons à présent mettre en œuvre une optimisation de parcours de la structure proposée afin d'améliorer les performances en terme de temps d'exécution. Une telle optimisation est tout à fait réalisable et constitue l'une de nos perspectives principales. Nous devrions alors obtenir des performances supérieures à celles obtenues pour FREQT-dd tout en conservant une structure en $2|T|$.



(a) $min\_sup = 0.05$ (b) $min\_sup = 0.5$

FIG. 9 – *Mémoire et Temps d'exécution FREQT-nodd, RSF, et FREQT-dd.*

5 Conclusion et perspectives

Dans cet article, nous proposons une approche efficace d'extraction de sous-arbres fréquents. RSF est la première proposition de recherche de sous-arbres fréquents selon une inclusion induite à l'aide d'une représentation de la base de schémas en $2|T|$. Les premières expérimentations réalisées sur des données synthétiques soulignent l'intérêt de notre proposition par rapport aux approches de référence. Les perspectives immédiates concernant RSF suivent deux axes :

- Tout d'abord, il est possible d'améliorer l'algorithme en optimisant les parcours réalisés lors de la vérification des candidats comme proposé dans l'optimisation de FREQT (Asai et al., 2002). Toutefois, nous souhaitons mettre en place un procédé moins coûteux en terme d'espace mémoire.
- Ensuite, nous souhaitons utiliser la même structure de représentation des arbres pour réaliser une recherche de sous-arbres fréquents en se basant sur une inclusion incrustée.

Ces travaux ont pour objectif d'être utilisés dans le cadre de la médiation de données, les sous-arbres fréquents extraits servant de support à la construction automatique d'un schéma médiateur. Une telle solution peut également être adoptée dans le cadre de la fouille de données en ligne (data streams) pour le traitement à la volée de données XML. Cette perspective permettra de traiter les gros volumes de données transitant sur Internet de manière efficace et rapide.

Références

Agrawal, R. et R. Srikant (septembre 1994). Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th VLDB Conference (VLDB 02)*, Santiago, Chile, pp. 487–499.

Asai, T., K. Abe, S. Kawasoe, H. Arimura, et H. Sakamoto (2002). Efficient substructure discovery from large semi-structured data. In *Proceedings of the 2nd Annual SIAM Symposium on Data Mining, (SDM 02)*, Arlington, VA, USA, pp. 158–174.

Chi, Y., Y. Yang, et R. Muntz (2003). Indexing and mining free trees. In *Proceedings of the International Conference on Data Mining (ICDM 2003)*, Florida, USA, pp. 509–512.

Chi, Y., Y. Yang, et R. Muntz (2004). CMTreeMiner : Mining both closed and maximal frequent subtrees. In *Proceedings of the Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 04)*, Sydney, Australia, pp. 63–73.

Del Razo, F., A. Laurent, et M. Teisseire (2005). Représentation efficace des arborescences pour la recherche des sous-structures fréquentes. In *Actes de l'atelier Fouille de données complexes, Conférence Extraction et Gestion des Connaissances (EGC 2005)*, pp. 113–120.

Termier, A., M.-C. Rousset, et M. Sebag (2002). TreeFinder, a first step towards xml data mining. In *Proceedings of the IEEE Conference on Data Mining (ICDM 02)*, pp. 450–457.

Tranier, J., R. Baraer, Z. Bellahsene, et M. Teisseire (July, 7th - 9th 2004). Where's Charlie : family based heuristics for peer-to-peer schema integration. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS 04)*, Coimbra, Portugal, pp. 227–235.

Wang, C., Q. Yuan, H. Zhou, W. Wang, et B. Shi (May 2004). Chopper : An efficient algorithm for tree mining. *Journal of Computer Science and Technology 19*, 309–319.

Weiss, M. A. (1998). *Data Structures And Algorithm Analysis In C*.

Xylème, L. (2001). A dynamic warehouse for XML data of the Web. *IEEE Data Engineering Bulletin 24*(2), 40–47.

Zaki, M. (2002). Efficiently mining frequent trees in a forest. In *Proceedings of the SIGKDD'02 Conference*, Edmonton, Alberta, Canada, pp. 71–80.

Summary

The research of a mediator schema from XML schemas is a current problem where the results stemming from the mining of tree databases can be adopted. In this context, several propositions were proposed but the methods of representation of tree databases are often very memory-consuming when querying huge volumes of data. In this paper, we propose an algorithm of research of frequent sub-structures based on an original method of representation of XML schemas. We describe the structure adopted and the algorithms of research for frequent sub-trees leaning on such structure.